Chapter 8

Evaluating XPath Queries

Introduction

- When XML documents are small and can fit in memory, evaluating XPath expressions can be done efficiently
- But what if we have very large documents stored on disk?
- How should they be stored (fragmented)?
- How can we query them efficiently (by reducing the number of disk accesses needed)?

Fragmentation

- A large document will not fit on a single disk page (block)
- It will need to be *fragmented* over possibly a large number of pages
- Updates to the document may result in further fragmentation

Pre-order traversal

Recall pre-order traversal of a tree:

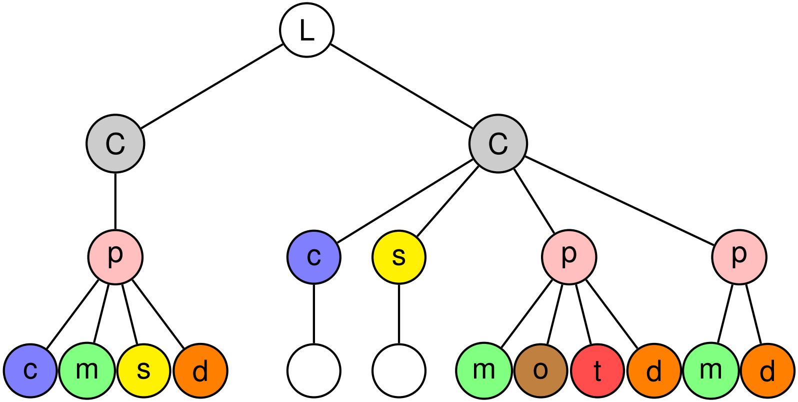
- To traverse a non-empty tree in pre-order, perform the following operations recursively at each node, starting with the root node:
    1. Visit the node
    2. Traverse the root nodes of subtrees of the node from left to right

Fragmentation based on pre-order traversal

A very simple method to store the document nodes on disk is as follows:

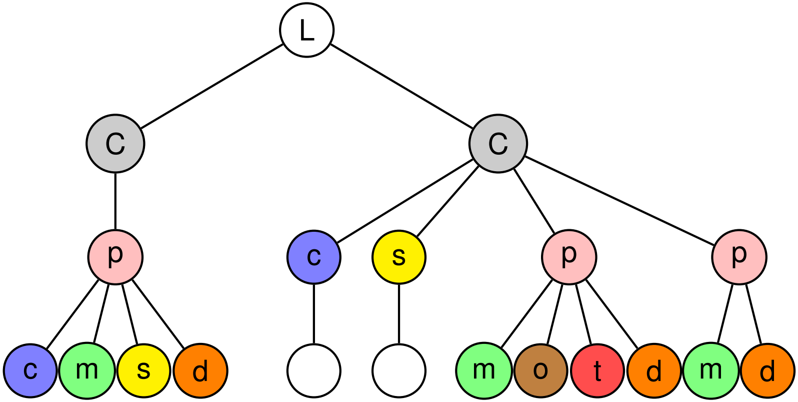
- A pre-order traversal of the document, starting from the root, groups as many nodes as possible within the current page
- When the page is full, a new page is used to store the nodes that are encountered next
- and so on, until the entire tree has been traversed

CD library example — first two CDs



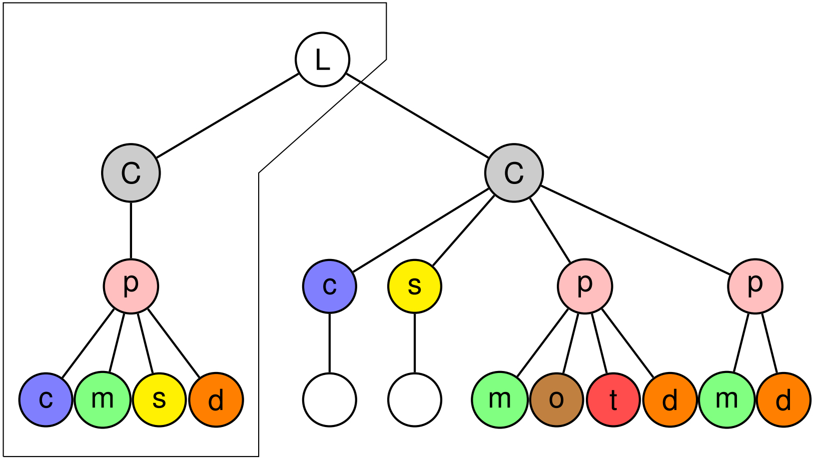
CD library example — first two CDs

Stored as 3 fragments



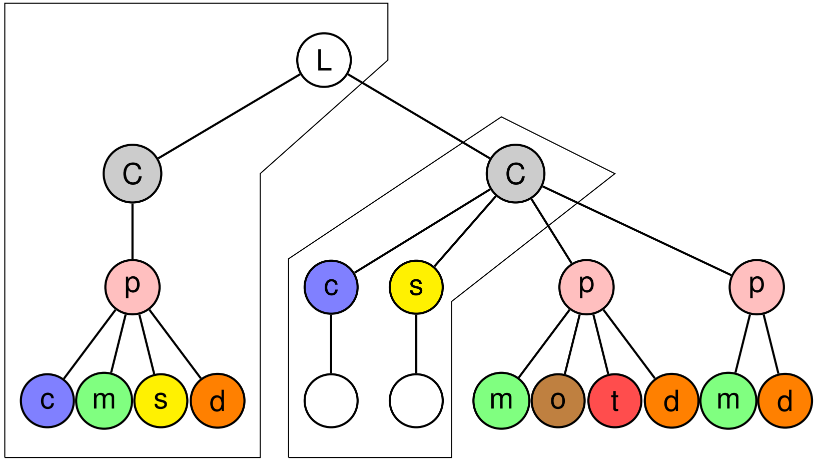
CD library example — first two CDs

Stored as 3 fragments



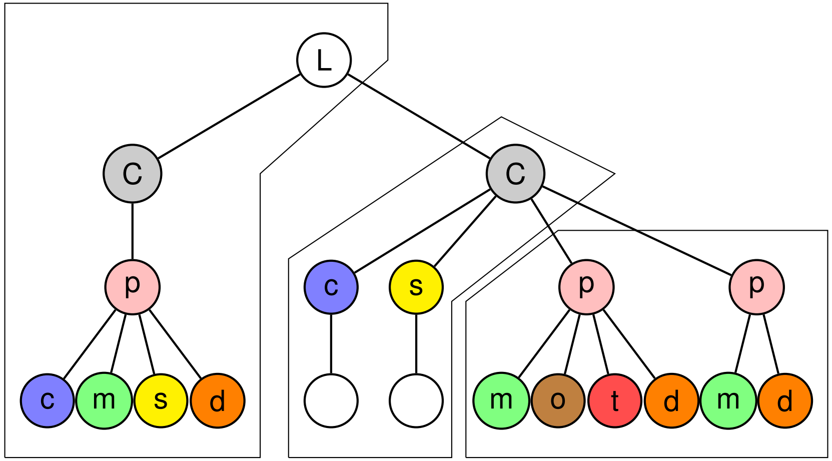
CD library example — first two CDs

Stored as 3 fragments



CD library example — first two CDs

Stored as 3 fragments



Simple XPath queries

- Selecting both CDs nodes requires accessing 2 fragments
- Evaluating `/CD-library/CD/performance` requires accessing all 3 fragments
- This is very small example, but one can see that such fragmentation could lead to very bad performance

Simple XPath queries

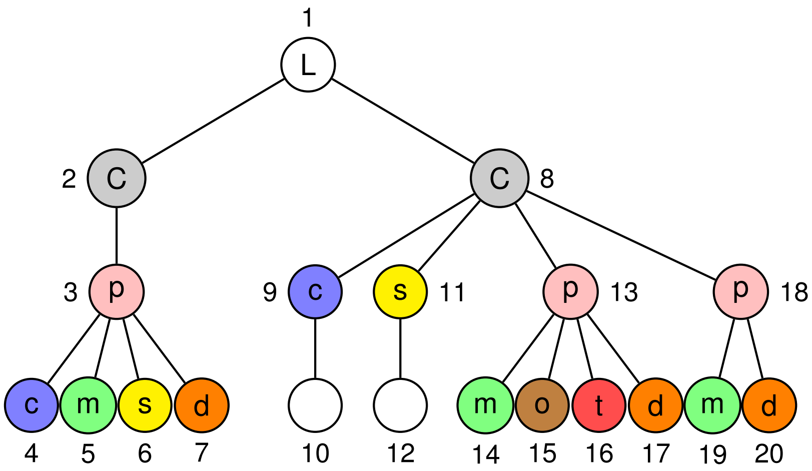
- Selecting both CDs nodes requires accessing 2 fragments
- Evaluating `/CD-library/CD/performance` requires accessing all 3 fragments
- This is very small example, but one can see that such fragmentation could lead to very bad performance
- Two improvements:
  - *Smart fragmentation*: Group those nodes that are often accessed simultaneously together
  - *Rich node identifiers*: Sophisticated node identifiers reducing the cost of join operations needed to “stitch” back fragments

Representation on disk

- One of the simplest ways to represent an XML document on disk is to
  - Assign an identifier to each node
  - Store the XML document as one relation (which may be fragmented) representing a set of edges

Simple node identfiers

Here node identifiers are simply integers, assigned in some order



The *Edge* relation

pid	cid	clabel
-	1	CD-library
1	2	CD
2	3	performance
3	4	composer
3	5	composition
3	6	soloist
3	7	date
1	8	CD
...	...	...

- “pid” is the id of the parent node
- “cid” is the id of the child node
- “clabel” is the element name of the child node
- (attributes and text nodes can be handled similarly)

Processing XPath queries

- `//composer`: can be evaluated by a simple lookup

$$\pi_{cid}(\sigma_{clabel='composer'}(Edge))$$

Processing XPath queries

- `//composer`: can be evaluated by a simple lookup

$$\pi_{cid}(\sigma_{clabel='composer'}(Edge))$$

- `/CD-library/CD`: requires one join

$$\pi_{cid}((\sigma_{clabel='CD-library'}(Edge)) \bowtie_{cid=pid} (\sigma_{clabel='CD'}(Edge)))$$

Processing XPath queries (2)

- `/CD-library//composer`: many joins potentially needed

$$Let \ \ A := (\sigma_{clabel='CD-library'}(Edge))$$

$$Let \ \ B := (\sigma_{clabel='composer'}(Edge))$$

`/CD-library/composer`	$\pi_{cid}(A \bowtie_{cid=pid} B)$
`/CD-library/*/composer`	$\pi_{cid}(A \bowtie_{cid=pid} Edge \bowtie_{cid=pid} B)$
`/CD-library/*/*/composer`	...
...	...

- This assumes the query processor does not have any schema information available which might constrain where `composer` elements are located

Element-partitioned Edge relations

- A simple improvement is to use *element-partitioned* Edge relations
- Here, the Edge relation is partitioned into many relations, one for each element name

CD-library

pid	cid
-	1

CD

pid	cid
1	2
1	8

performance

pid	cid
2	3
8	13
8	18

composer

pid	cid
3	4
8	9

Element-partitioned Edge relations (2)

- This saves some space (element names are not repeated)
- It also reduces the disk I/O needed to retrieve the identifiers of elements having a given name
- However, it does not solve the problem of evaluating queries with // steps in non-leading positions

Path-partitioned approach to fragmentation

- *Path-partitioning* tries to solve the problem of `//` steps at arbitrary positions in a query
- This approach uses one relation for each distinct path in the document, e.g., `/CD-library/CD/performance`
- There is also another relation, called Paths, which contains all the unique paths

Path-partitioned storage

`/CD-library`:

pid	cid
-	1

`/CD-library/CD`:

pid	cid
1	2
1	8

`/CD-library/CD/composer`:

pid	cid
8	9

`/CD-library/CD/performance/composer`:

pid	cid
3	4

Paths:

path
`/CD-library`
`/CD-library/CD`
`/CD-library/CD/performance`
`/CD-library/CD/performance/composer`
...

Path-partitioned storage (2)

- Based on a path-partitioned store, a query such as `//CD//composer` can be evaluated in two steps:
  - Scan the Paths relation to identify all the paths matching the given XPath query
  - For each such path, scan the corresponding path-partitioned relation
- So for `//CD//composer`, the paths would be
  - `/CD-library/CD/composer`
  - `/CD-library/CD/performance/composer`
- So only these two relations need to be scanned

Path-partitioned storage (3)

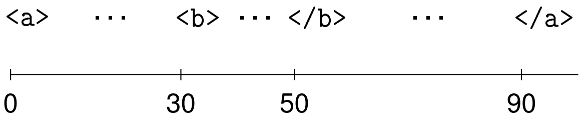
- The evaluation of XPath queries with many branches will still require joins across the relations
- However, the evaluation of `//` steps is simplified, thanks to the first processing step, performed on the path relation
- For very structured data, the path relation is typically small
- Thus, the cost of the first processing step is likely negligible, while the performance benefits of avoiding numerous joins are quite important
- However, for some data, the path relation can be large, and in some cases, even larger than the data itself

Node identifiers

- Node identifiers are needed to indicate how nodes are related to one another in an XML tree
- This is particularly important when the data is fragmented and we need to reconnect children with their parents
- However, it is often even more useful to be able to identify other kinds of relationships between nodes, just by looking at their identifiers
- This means we need to use identifiers that are richer than simple consecutive integers
- We will see later how this information can be used in query processing

Region-based identifers

- The region-based identifier scheme assigns to each XML node *n* a pair of integers
- The pair consists of the offset of the node's start tag, and the offset of its end tag
- We denote this pair by $(n.start, n.end)$
- Consider the following offsets of tags:



- the region-based identifier of the `<a>` element is the pair $(0, 90)$
- the region-based identifier of the `<b>` element is the pair $(30, 50)$

Using region-based identifiers

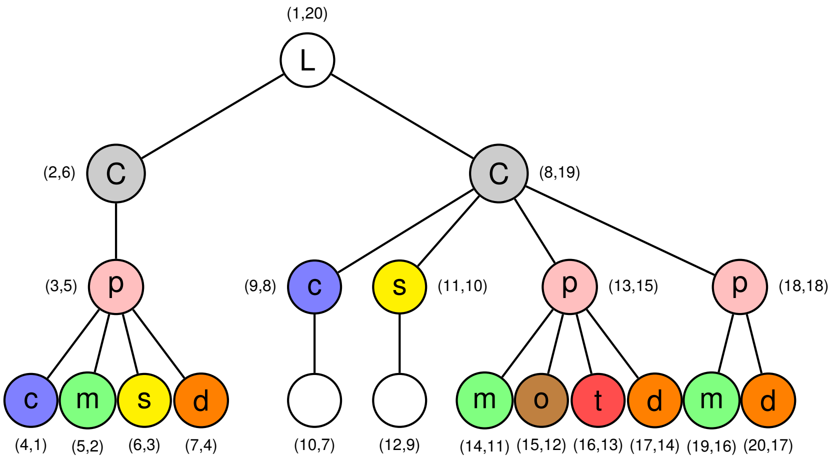
- Comparing the region-based identifiers of two nodes n_1 and n_2 allows for deciding whether n_1 is an ancestor of n_2
- Observe that this is the case if and only if:
  - $n_1.start < n_2.start$, and
  - $n_1.end > n_2.end$
- There is no need to use byte offsets:
  - (Start tag, end tag). Count only opening and closing tags (as one unit each) and assign the resulting counter values to each element
  - (Pre, post). Pre-order and post-order index (see next slides)
- Region-based identifiers are quite compact, as their size only grows logarithmically with the number of nodes in a document

Post-order traversal

Recall post-order traversal of a tree:

- To traverse a non-empty tree in post-order, perform the following operations recursively at each node, starting with the root node:
    1. Traverse the root nodes of subtrees of the node from left to right
    2. Visit the node

Example of (pre, post) node identifiers



Using (pre, post) identifiers to find ancestors

- The same method as for other region-based identifiers allows us to decide, for two nodes n_1 and n_2, whether n_1 is an *ancestor* of n_2
- As before, this is the case if and only if:
  - $n_1.pre < n_2.pre$, and
  - $n_1.post > n_2.post$

  where $n_i.pre$ and $n_i.post$ are the pre-order and post-order numbers assigned to node n_i, respectively

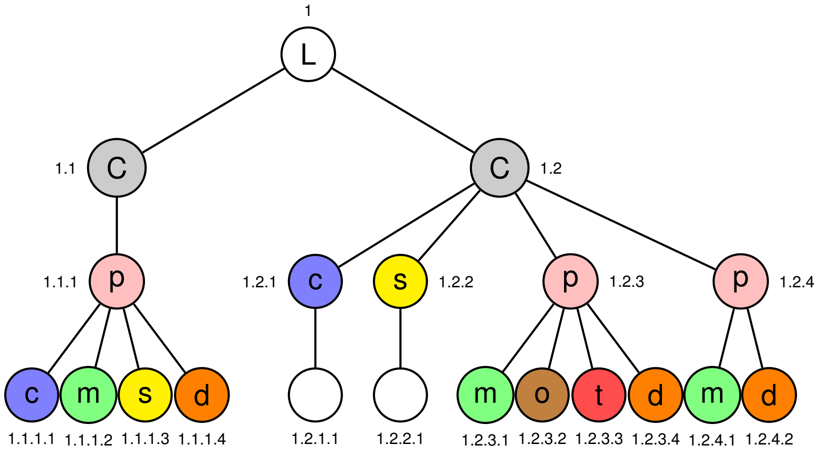
Using (pre, post) identifiers to find parents

- One can add another number to a node identifier which indicates the *depth* of the node in the tree
- The root is assigned a depth of 1; the depth increases as we go down the tree
- Using ($pre, post, depth$), we can decide whether node n_1 is a *parent* of node n_2
- Node n_1 is a parent of node n_2 if and only if
  - n_1 is an ancestor of n_2 and
  - $n_1.depth = n_2.depth - 1$

Dewey-based identifiers

- These identifiers use the principal of the Dewey classification system used in libraries for decades
- To get the identifier of a child node, one adds a suffix to the identifier of its parent (including a separator)
- e.g., if the parent's identifier is 1.2.3 and the child is the second child of this parent, then its identifier is 1.2.3.2

Example of Dewey-based identifiers



Using Dewey-based identifiers

- Let n_1 and n_2 be two identifiers, of the form: $n_1 = x_1.x_2.\ldots.x_m$ and $n_2 = y_1.y_2.\ldots.y_n$
- The node identified by n_1 is an ancestor of the node identified by n_2 if and only if n_1 is a *prefix* of n_2
- When this is the case, the node identified by n_1 is the *parent* of the node identified by n_2 if and only if $n = m + 1$
- Dewey IDs allow finding other relationships such as preceding-sibling and preceding (respectively, following-sibling, and following)
- The node identified by n_1 is a preceding sibling of the node identified by n_2 if and only if
  1. $x_1.x_2.\ldots.x_{m-1} = y_1.y_2.\ldots.y_{n-1}$ and
  2. $x_m < y_n$
- The main drawback of Dewey identifiers is their length: the length is variable and can get large

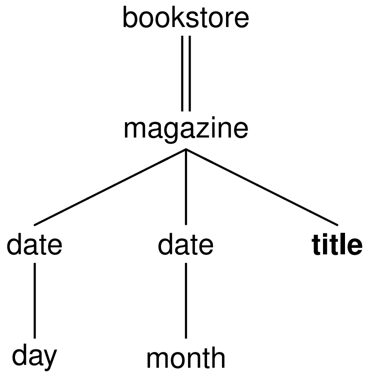
Structural identifiers and updates

- Consider a node with Dewey ID 1.2.2.3
    - Suppose we insert a new first child for node 1.2
    - Then the ID of node 1.2.2.3 becomes 1.2.3.3
- In general:
    - Offset-based identifiers need to be updated as soon as a character is inserted or removed in the document
    - (start, end), (pre, post), and Dewey IDs need to be updated when the elements of the documents change
    - It is possible to avoid re-labelling on deletions, but gaps will appear in the labelling scheme
    - Re-labelling operations are quite expensive

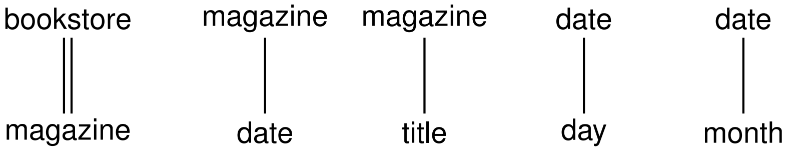
Tree pattern query evaluation

- Assume we have element-partitioned relations using (pre, post) identifiers
- Assume we want to evaluate a tree pattern query
- One way is to decompose the query into its “basic” patterns:
  - Each basic pattern is just a pair of nodes
  - connected by a child edge or a descendant edge
- We particularly want an efficient way of evaluating basic patterns that use the descendant operator

Tree Pattern Example

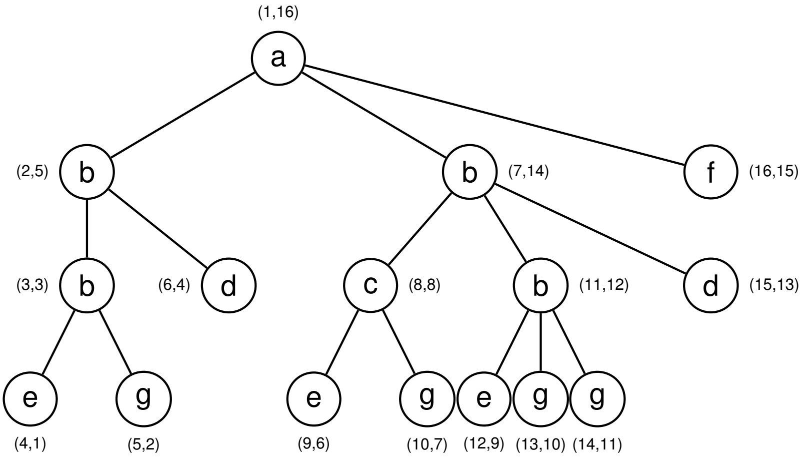


Decomposed Tree Pattern Example



Example tree with (pre, post) identifiers

(Taken from the book "Web Data Management")



Element-partitioned relations for example

a

pre	post
1	16

b

pre	post
2	5
3	3
7	14
11	12

c

pre	post
8	8

d

pre	post
6	4
15	13

e

pre	post
4	1
9	6
12	9

f

pre	post
16	15

g

pre	post
5	2
10	7
13	10
14	11

Evaluation of descendant patterns

- Assume we want to evaluate the basic pattern corresponding to `b//g`
- This pattern may need to be joined to the results calculated for other basic patterns
- So, in general, we need to find all pairs (x, y) of nodes where
  - x is an element with name `b`
  - y is an element with name `g`
  - y is a descendant of x

Evaluation of descendant patterns (2)

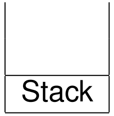
- We could take every node ID from the b relation and compare it to every node ID from the g relation
- Each time we can test whether the g-node is a descendant of the b-node using the (pre, post) identifiers
- But this method will take time proportional to $n \times m$, if there are n b-nodes and m g-nodes
- In particular, one of the relations is scanned many times
- This is similar to a nested-loops implementation of a relational join, which is known to be inefficient
- Can we do better?

Stack-based join algorithm

- We will look at an elegant method for evaluation of descendant patterns that uses an auxiliary *stack*
- This is called the *stack-based join* (SBJ) algorithm
- SBJ reads each ID from each relation only *once*
- SBJ assumes that the IDs in each relation are *sorted*, essentially by their pre-order values (as in the earlier slide)
- We will illustrate the method by example

Stack-based join algorithm — example

b IDs	g IDs
(2,5)	(5,2)
(3,3)	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)



Stack-based join algorithm — example

b IDs	g IDs
	(5,2)
(3,3)	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)

Stack
(2,5)

- SBJ starts by pushing the first ancestor (that is, b node) ID, namely (2,5), on the stack

Stack-based join algorithm — example

	(5,2)
(3,3)	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)
b IDs	g IDs

(2,5)
Stack

- SBJ starts by pushing the first ancestor (that is, b node) ID, namely (2,5), on the stack
- Then, STD continues to examine the IDs in the b ancestor input
- While the current ancestor ID is a descendant of the top of the stack, the current ancestor ID is pushed on the stack

Stack-based join algorithm — example

	(5,2)
	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)
b IDs	g IDs

(3,3)
(2,5)
Stack

- SBJ starts by pushing the first ancestor (that is, b node) ID, namely (2,5), on the stack
- Then, STD continues to examine the IDs in the b ancestor input
- While the current ancestor ID is a descendant of the top of the stack, the current ancestor ID is pushed on the stack
- So the second b ID, (3,3), is pushed on the stack, since it is a descendant of (2,5)

Stack-based join algorithm — example (2)

	(5,2)
	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)
b IDs	g IDs

(3,3)
(2,5)
Stack

Output

- The third ID in the b input, (7,14), is not a descendant of current stack top, namely (3,3)
- Therefore, SBJ stops pushing b IDs on the stack and considers the first descendant ID, to see if it has matches on the stack

Stack-based join algorithm — example (2)

b IDs	g IDs
	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)

Stack
(3,3)
(2,5)

Output
(3,3), (5,2)
(2,5), (5,2)

- The third ID in the b input, (7,14), is not a descendant of current stack top, namely (3,3)
- Therefore, SBJ stops pushing b IDs on the stack and considers the first descendant ID, to see if it has matches on the stack
- The first g node, namely (5,2), is a descendant of both b nodes on the stack, leading to the first two output tuples

Stack-based join algorithm — example (2)

b IDs	g IDs
	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)

Stack
(3,3)
(2,5)

Output
(3,3), (5,2)
(2,5), (5,2)

- The third ID in the b input, (7,14), is not a descendant of current stack top, namely (3,3)
- Therefore, SBJ stops pushing b IDs on the stack and considers the first descendant ID, to see if it has matches on the stack
- The first g node, namely (5,2), is a descendant of both b nodes on the stack, leading to the first two output tuples
- Note that the stack does not change when output is produced
- This is because there may be further descendant IDs to match the ancestor IDs on the stack

Stack-based join algorithm — example (3)

	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)
b IDs	g IDs

(3,3)
(2,5)
Stack

Output
(3,3), (5,2)
(2,5), (5,2)

- A descendant ID which has been compared with ancestor IDs on the stack and has produced output tuples, can be discarded
- Now the g ID (10,7) encounters no matches on the stack
- Moreover, (10,7) occurs in the document after the nodes on the stack
- Therefore, no descendant node ID yet to be examined can have ancestors on this stack
- This is because the input g IDs are sorted

Stack-based join algorithm — example (3)

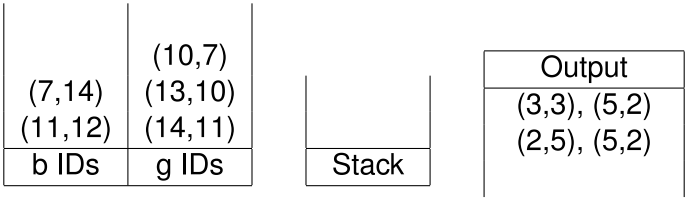
b IDs	g IDs
	(10,7)
(7,14)	(13,10)
(11,12)	(14,11)

Stack

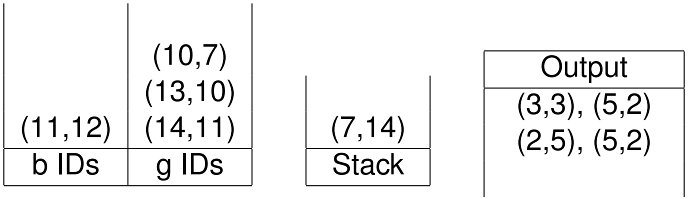
Output
(3,3), (5,2)
(2,5), (5,2)

- A descendant ID which has been compared with ancestor IDs on the stack and has produced output tuples, can be discarded
- Now the g ID (10,7) encounters no matches on the stack
- Moreover, (10,7) occurs in the document after the nodes on the stack
- Therefore, no descendant node ID yet to be examined can have ancestors on this stack
- This is because the input g IDs are sorted
- Therefore, at this point, the stack is emptied

Stack-based join algorithm — example (4)

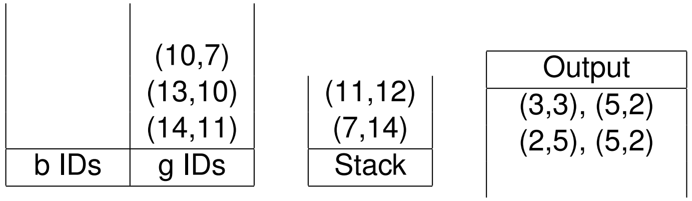


Stack-based join algorithm — example (4)



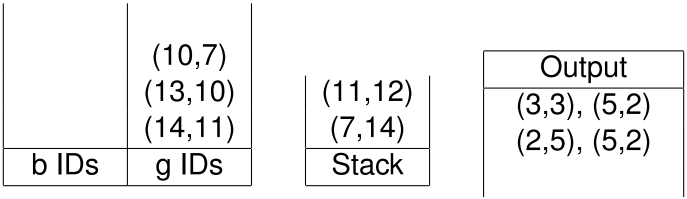
- Next the ancestor ID (7,14) is pushed on the stack

Stack-based join algorithm — example (4)



- Next the ancestor ID (7,14) is pushed on the stack
- followed by its descendant, in the ancestor input, (11,12)

Stack-based join algorithm — example (4)



- Next the ancestor ID (7,14) is pushed on the stack
- followed by its descendant, in the ancestor input, (11,12)
- The next descendant ID is (10,7)

Stack-based join algorithm — example (4)

b IDs	g IDs
	(13,10)
	(14,11)

Stack
(11,12)
(7,14)

Output
(3,3), (5,2)
(2,5), (5,2)
(7,14), (10,7)

- Next the ancestor ID (7,14) is pushed on the stack
- followed by its descendant, in the ancestor input, (11,12)
- The next descendant ID is (10,7)
- This which produces a result with (7,14) and is then discarded

Stack-based join algorithm — example (5)

b IDs	g IDs
	(13,10)
	(14,11)

Stack
(11,12)
(7,14)

Output
(3,3), (5,2)
(2,5), (5,2)
(7,14), (10,7)

Stack-based join algorithm — example (5)

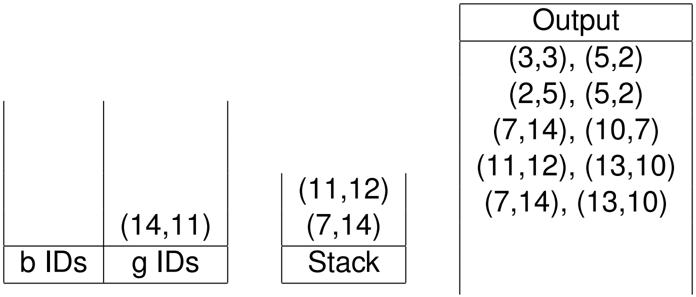
b IDs	g IDs
	(13,10)
	(14,11)

Stack
(11,12)
(7,14)

Output
(3,3), (5,2)
(2,5), (5,2)
(7,14), (10,7)

- The next descendant ID is (13,10)

Stack-based join algorithm — example (5)



- The next descendant ID is (13,10)
- This leads to two new tuples added to the output

Stack-based join algorithm — example (5)

b IDs	g IDs
	(14,11)

Stack
(11,12)
(7,14)

Output
(3,3), (5,2)
(2,5), (5,2)
(7,14), (10,7)
(11,12), (13,10)
(7,14), (13,10)

- The next descendant ID is (13,10)
- This leads to two new tuples added to the output
- The next descendant ID is (14,11)

Stack-based join algorithm — example (5)

b IDs	g IDs

Stack
(11,12)
(7,14)

Output
(3,3), (5,2)
(2,5), (5,2)
(7,14), (10,7)
(11,12), (13,10)
(7,14), (13,10)
(11,12), (14,11)
(7,14), (14,11)

- The next descendant ID is (13,10)
- This leads to two new tuples added to the output
- The next descendant ID is (14,11)
- This also leads to two more output tuples

Other approaches

- The stack-based join algorithm is as efficient as possible for single descendant basic patterns
- But an overall algorithm for tree pattern evaluation still has to join the answers from basic patterns back together
- The size of intermediate results can be unnecessarily large
- Another approach is to evaluate the entire pattern in one operation
- One algorithm for this is called *holistic twig join*

Summary

- We considered some issues for dealing with querying large XML documents
- These included methods for fragmenting documents
- and efficient evaluation methods, particularly for ancestor-descendant basic patterns
- For more information, see Chapter 4 on “XML Query Evaluation” in the book “Web Data Management”
- The original stack-based join algorithm is from S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas, and D. Srivastava. “Structural joins: A primitive for efficient XML query pattern matching.” In Proc. Int. Conf. on Data Engineering (ICDE), 2002.
- Holistic twig join is described in N. Bruno, N. Koudas, and D. Srivastava. “Holistic twig joins: optimal XML pattern matching.” In Proc. ACM Int. Conf. on the Management of Data (SIGMOD), 2002.