http://webdam.inria.fr/

Web Data Management

XML query evaluation

Serge Abiteboul
INRIA Saclay & ENS Cachan

Ioana Manolescu
INRIA Saclay & Paris-Sud University

Philippe Rigaux
CNAM Paris & INRIA Saclay

Marie-Christine Rousset
Grenoble University

Pierre Senellart
Télécom ParisTech

http://webdam.inria.fr/Jorge/

Contents

In previous chapters, we presented algorithms for evaluating XPath queries on XML documents in PTIME with respect to the combined size of the XML data and of the query. In this context, the entire document is assumed to fit within the main memory. However, very large XML documents may not fit in the memory available at runtime to the query processor. Since access to disk-resident data is orders of magnitude slower than access to the main memory, this dramatically changes the problem. When this is the case, performance-wise, the goal is not so much in reducing the algorithmic complexity of query evaluation, but in designing methods reducing the number of disk accesses that are needed to evaluate a given query. The topic of this chapter is the efficient processing of queries of disk-resident XML documents.

We will use extensively depth-first tree traversals in the chapter. We briefly recall two classical definitions:

preorder To traverse a non-empty binary tree in preorder, perform the following operations recursively at each node, starting with the root node: 1. Visit the root; 2. Traverse the left subtree; 3. Traverse the right subtree.

postorder To traverse a non-empty binary tree in postorder, perform the following operations recursively at each node, starting with the root node: 1. Traverse the left subtree; 2. Traverse the right subtree; 3. Visit the root.

Figure 1 illustrates the issues raised by the evaluation of path queries on disk-resident XML documents. The document represents information about some auctions. It contains a list of items for sale, as well as a collection of the currently open auctions. See Figure 1. A page size has been chosen, typically reasonably small so that one can access some small unit of information without having to load too much data. A very simple method has been used to store the document nodes on disk. A preorder 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, etc. Each thick-lined box in the figure represents a page. Observe, that a node may be stored in a different page than its parent. When this is the case, the "reference" of the child node in the parent page may consist

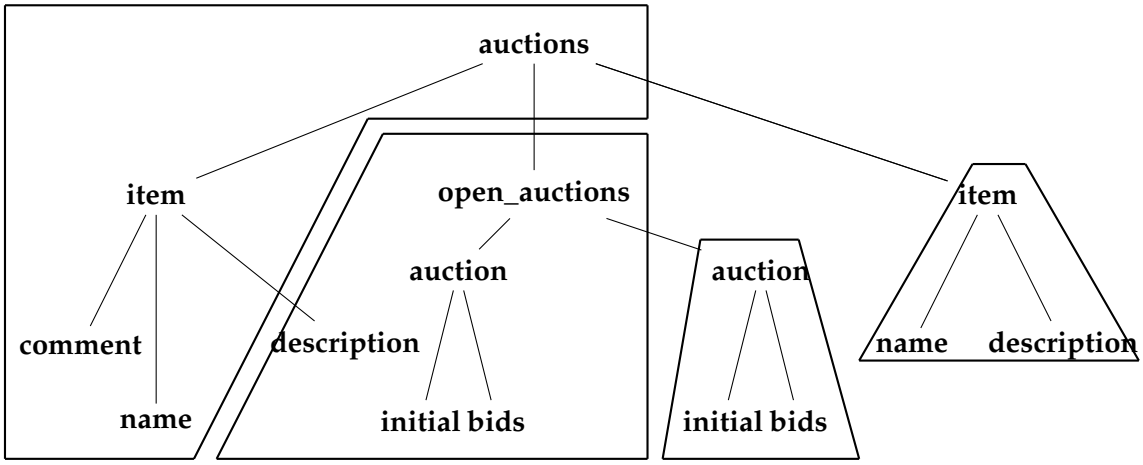

Figure 1: Simple page-based storage of an XML tree.

of (i) the ID of the page that stores the child node and (ii) the offset of the child node in that separate page.

We now consider the processing of simple queries on the document in Figure 1:

- `/auctions/item` requires the traversal of two disk pages;
- `/auctions/item/description` and `/auctions/open_auctions/auction/initial` both require traversing three disk pages;
- `//initial` requires traversing all the pages of the document.

These examples highlight the risk of costly disk accesses even when the query result is very small. As a consequence, there is a risk of very poor performance on documents stored in such a naïve persistent fashion. Indeed, the example was meant to illustrate that navigation on disk-resident structures may be costly and should be avoided. This is indeed a well-known principle since the days of object-oriented databases.

How can one avoid navigation? Two broad classes of techniques have been proposed.

Smart fragmentation. Since XML nodes are typically accessed by a given property (most often, their names and/or their incoming paths), fragmentation aims at decomposing an XML tree into separate collections of nodes, grouped by the value of interesting, shared property. The goal is to group nodes that are often accessed simultaneously, so that the number of pages that need to be accessed is globally reduced. We present the main approaches for fragmentation in Section 1.

Rich node identifiers. Even on a well-fragmented store, some queries may require combining data from more than one the stored collections. This can be seen as "stitching" back together separated fragments, and amounts to performing some joins on the identifiers of stored nodes. (The node identifiers for Figure 1 consist of (page, offset) pairs.) To make stitching efficient, sophisticate node identifier schemes have been proposed. We present some in Section 2 and discuss interesting properties they provide. We present XML query evaluation techniques exploiting node identification schemes in Section 3.

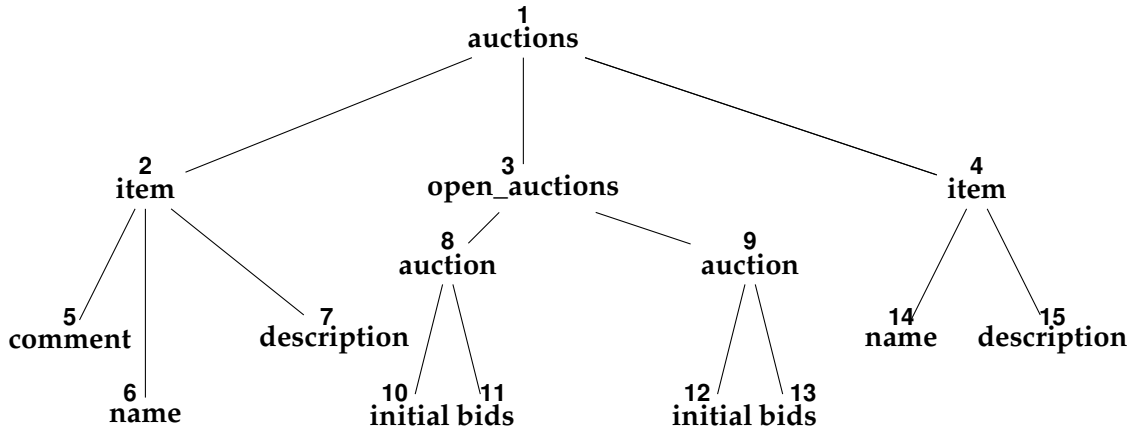

Figure 2: Sample XML document with simple node identifiers.

pid	cid	clabel
-	1	auctions
1	2	item
2	5	comment
2	6	name
2	7	description
1	3	open_auctions
3	8	auction
...	...	...

Figure 3: Partial instance of the *Edge* relation for the document in Figure 2.

1 Fragmenting XML documents on disk

A set of simple alternatives have been explored for fragmenting XML documents, and they can all be conceptualized with the help of some set of relations. The evaluation of XML queries then turns into a sequence of two steps: translating the XML query into a relational (XML) query, and evaluating the relational query on the tables representing the XML document content.

To illustrate fragmentation approaches, we will rely on the sample auction document in Figure 2, where next to each node, we show an integer that allows identifying the node.

The simplest approach considers a document to be a collection of edges, stored in a single **Edge(pid, cid, clabel)** relation. Here, **cid** is the ID of some node (child node), **pid** stands for the ID of its parent, and **label** is its label (the label of the child node). For instance, part of the document in Figure 2 is encoded by the relation shown in Figure 3.

Let us now explain the processing of XPath queries relying on such a store. Here and throughout the section, we consider the evaluation of an XPath query up to the point where the identifiers of the result nodes are known. It is then usually straightforward to retrieve the full XML elements based on their identifiers.

- The query `//initial` can now be answered by evaluating the expression:

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

The *Edge* storage is quite beneficial for queries of this form, i.e., `//a` for some node label `a`, since, with the help of an appropriate index on *Edge.clabel*, one can evaluate such queries quite efficiently.

- The query `/auctions/item` translates to the expression:

$$\pi_{\text{cid}}((\sigma_{clabel=auctions}(Edge)) \bowtie_{cid=pid} (\sigma_{clabel=item}(Edge)))$$

Observe that the two nodes in the query translate to two occurrences of the *Edge* relation. Similarly, queries such as

`/auctions/item/description` and
`/auctions/open_auctions/auction/initial`

require joining several instances of the *Edge* table. Such queries become problematic when the size of the query increases.

- Now consider a query such as `//auction//bid`. In the absence of schema information, the distance between the XML nodes matching `auction` and their descendants matching `bid` is unknown. Therefore, such queries lead to a union of different-lengths simpler queries, where the `//` has been replaced with chains of `/` steps, introducing nodes with unconstrained labels (using `*`):

`//auction/bid`	$\pi_{cid}(A \bowtie_{cid=pid} B)$
`//auction/*/bid`	$\pi_{cid}(A \bowtie_{cid=pid} Edge \bowtie_{cid=pid} B)$
`//auction/*/*/bid`	$\cdots$

where $A = \sigma_{clabel=auctions}(Edge)$ and $B = \sigma_{clabel=bid}(Edge)$.

In principle, there is no bound on the number of intermediate nodes, so we have to evaluate an infinite union of XPath queries. In practice, some bound may be provided by the schema or the maximum depth of the document may be recorded at the time the document is loaded. This limits the number of queries to be evaluated. However, it is clear that the processing of such queries, featuring `//` at non-leading positions, is very costly in such an setting.

Tag-partitioning A straightforward variation on the above approach is the *tag-partitioned Edge* relation. The *Edge* relation is partitioned into as many tag-based relations as there are different tags in the original document. Each such relation stores the identifiers of the nodes, and the identifiers of their respective parents. (There is no need to repeat the label that is the same for all nodes in such a collection.) Figure 4 illustrates tag-partitioning for the document in Figure 2.

Clearly, the tag-partitioned *Edge* store reduces the disk I/O needed to retrieve the identifiers of elements having a given tag, since it suffices to scan the corresponding tag-partitioned relation. However, the partitioning of queries with `//` steps in non-leading position remains as difficult as it is for the *Edge* approach.

auctionsEdge

pid	cid
-	1

itemEdge

pid	cid
1	2
1	4

open_auctionsEdge

pid	cid
1	3

auctionEdge

pid	cid
3	8
3	9

Figure 4: Portions of the tag-partitioned *Edge* relations for the document in Figure 2.

/auctions

pid	cid
-	1

/auctions/item

pid	cid
1	2
1	4

/auctions/item/name

pid	cid
2	6
4	14

Paths

path
/auctions
/auctions/item
/auctions/item/comment
/auctions/item/name
...

Figure 5: Relations resulting from the path-partitioned storage of the document in Figure 2.

Path-partitioning The *Path-partitioning* fragmentation approach aims at solving the problem raised by `//` steps at arbitrary positions in a query. The idea is roughly to encode the Dataguide (see page **??**) of the XML data set in a set of relations. There is one relation for each distinct parent-child path in the document, e.g., `/auctions/item/name`. There is also an extra table, namely `path` containing all the unique paths. Figure 5 illustrates the resulting relations for the document in Figure 2.

Based on a path-partitioned store, a linear query such as `//item//bid` can be evaluated in two steps:

- Scan the `path` relation and identify all the parent-child paths matching the given linear XPath query;
- For each of the paths thus obtained, scan the corresponding path-partitioned table.

On a path-partitioned store, 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, this relation is typically small, much smaller than the data set itself. 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 quite large, and in pathological cases, even larger than the data itself.

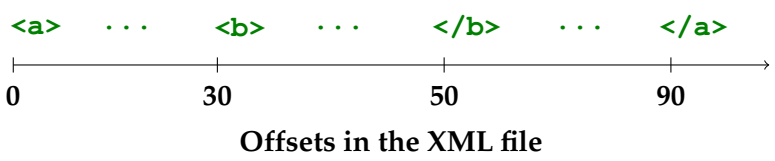

Figure 6: Region-based identifiers.

2 XML node identifiers

We have seen that, within a persistent XML store, each node must be assigned a unique identifier (or ID, in short). Such an identifier plays the role of a primary key to the node, that allows distinguishing it from other nodes even if they have the same label and pairwise identical children. In a fragmented store, moreover, these identifiers are essential to be able to reconnect nodes according to their relationships and reconstruct the original XML document.

Going beyond these simple ID roles, we show next that it is very useful to encapsulate in identifiers some knowledge of the element's position within the original document. Indeed, with particular node identifiers, it even becomes possible, just by considering two XML node IDs, to decide how the respective nodes are related in the document. We will see how this can be exploited to efficiently process queries.

2.1 Region-based identifiers

Typically, the identifiers we consider exploit the tree structure of XML. First, we present an identification scheme that is based on the tags signaling the beginning and the end of XML elements. We assume that the tags are properly nested (just like parentheses in arithmetical expressions), i.e., that the document is well-formed.

Figure 6 shows an example. This document is the serialization of an XML tree with a root labeled a, that includes some text, a child labeled b that only includes text, followed by some text. On the horizontal axis, we show the offset, within the XML file, at which begin tags, such as `<a>`, respectively, end tags, such as `</a>`, are encountered. In the figure, the `a` element is the root of the document, thus its begin tag starts at offset 0. The `b` element starts at offset 30 in the file, and finishes at offset 50. Observe that due to the well-formedness of an XML document, an element's begin tag must follow the begin tags of its ancestors. Similarly, an element's end tag must precede the end tags of its ancestors.

The so-called region-based identifier scheme simply assigns to each XML node n, the pair composed of the offset of its begin tag, and the offset of its end tag. We denote this pair by $(n.begin, n.end)$. In the example in Figure 6:

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

Comparing the region-based identifiers of two nodes n_1 and n_2 allows 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
- $n_2.end < n_1.end$.

Thus, by considering only the identifiers $n_1 = (0,90)$ and $n_2 = (30,50)$, and without having to access the respective nodes or any other data structures, one can decide that the element identified by n_1 is an ancestor of the element identified by n_2. Contrast this simple decision procedure with the costly navigation required to answer a similar question on the page-fragmented storage shown in Figure 1!

Region-based node identifiers are the simplest and most intuitive structural XML IDs. The simple variant presented above, using offsets in the serialized XML file, can be implemented easily, since one only needs to gather offsets inside the file corresponding to the serialized document. However, for the purpose of efficiently evaluating tree pattern queries, one does not need to count all characters in the file, but only the information describing how elements are structurally related in the XML tree. Popular variants of region-based identifiers based on this observation include the following:

(Begin tag, end tag). Instead of counting characters (offsets), count only opening and closing tags (as one unit each) and assign the resulting counter values to each element. Following this scheme, the <a> element in Figure 6 is identified by the pair $(1,4)$, while the <b> element is identified by the pair $(2,3)$.

The pair (Begin tag, end tag) clearly allows inferring whether an element is an ancestor of another by simple comparisons of the ID components.

(Pre, post). The *(pre, post)* identifiers are computed as follows:

- Perform a preorder traversal of the tree. Count nodes during the traversal and assign to each node its corresponding counter value. This would assign the so-called *pre number* of 1 to the *a*-node, and the pre number of 2 to the *b*-node.
- Perform a post-order traversal of the tree. Count nodes during the traversal and assign to each node its corresponding counter value, called *post number*. For instance, the post number of <a> in Figure 6 is 2, and the post number of <b> is 1.

The (pre, post) IDs still allow inferring whether an element n_1 is an *ancestor* of another one n_2, i.e., if $n_1.pre \leq n_2.pre$ and $n_2.post \leq n_1.post$.

It is also useful to be able to decide whether an element is *a parent* of another. The following variant allows to do so:

(Pre, post, depth). This scheme adds to the (pre, post) pair an integer representing the depth, or distance from the document root, at which the corresponding individual node is found. An element identified by n_1 is the parent of an element identified by n_2, if and only if the following conditions hold:

- n_1 is an ancestor of n_2 and
- $n_1.depth = n_2.depth - 1$.

For illustration, Figure 7 shows the XML document of the running example, with nodes adorned with (pre, post, depth) identifiers.

Region-based identifiers are quite compact, as their size only grows logarithmically with the number of nodes in a document.

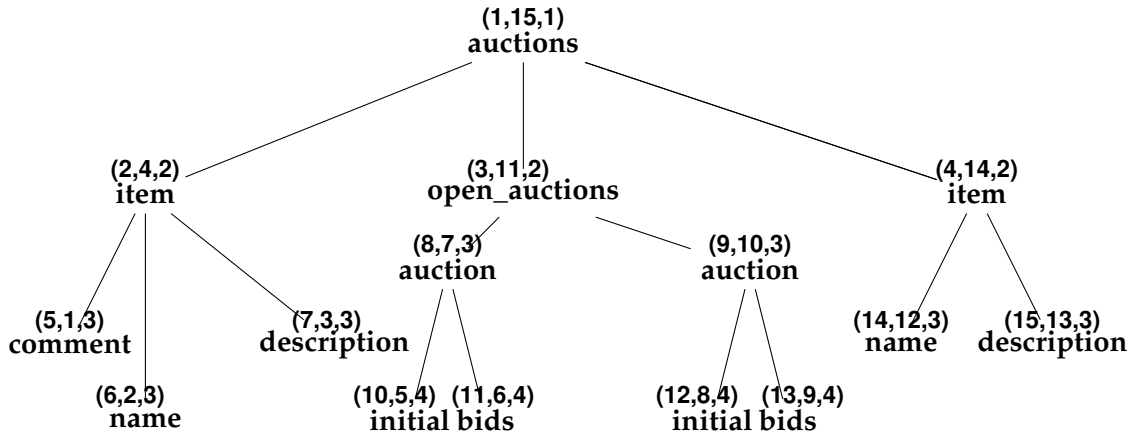

Figure 7: Sample XML document with (pre, post, depth) node identifiers.

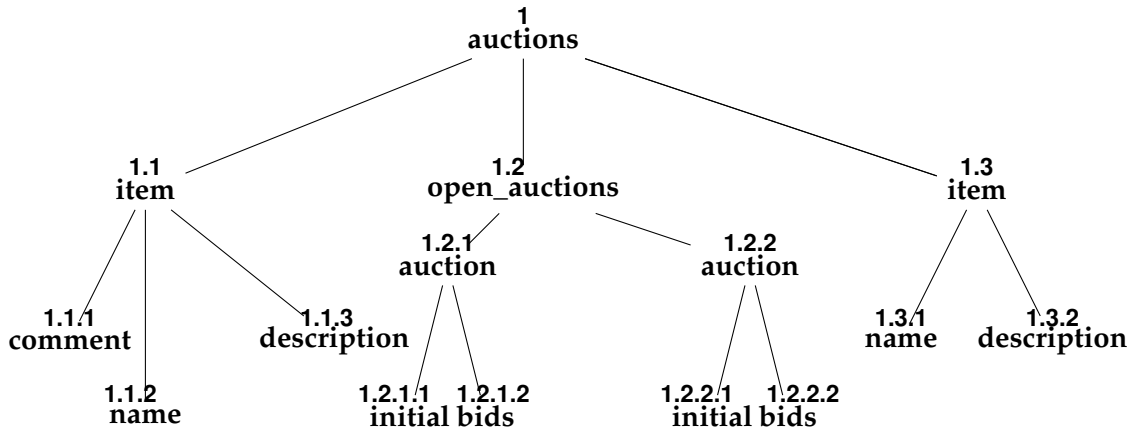

Figure 8: Sample XML document with Dewey node identifiers.

2.2 Dewey-based identifiers

Another family of identifiers borrows from the well-known Dewey classification scheme, widely used, for example, in libraries long before computers and databases took over the inventories. The principle of Dewey IDs is quite simple: the ID of a node is obtained by adding a suffix to the ID of the node's parent. The suffix should allow distinguishing each node from all its siblings that are also constructed starting from the same parent ID. For instance, the root may be numbered 1, its first child 1.1, its second child 1.2 and so on. For example, Figure 8 shows our sample XML document with the nodes adorned with Dewey node IDs.

Dewey IDs encapsulate structural information in a much more explicit way than region IDs. We illustrate some aspects next. Let n_1 and n_2 be two identifiers, of the form $n_1 = x_1.x_2.\ldots.x_m$ and $n_2 = y_1.y_2.\ldots.y_n$. Then:

- 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 also allow establishing other relationships such as preceding-sibling and before (respectively, following-sibling, and after). The node identified by n_1 is a preceding sibling of the node identified by n_2 if and only if (i) $x_1.x_2.\ldots.x_{m-1} = y_1.y_2.\ldots.y_{n-1}$; and (ii) $x_m < y_n$.
- Given two Dewey IDs n_1 and n_2, one can find the ID of the *lowest common ancestor (LCA)* of the corresponding nodes. The ID of the LCA is the longest common prefix of n_1 and n_2. For instance, in Figure 8, the LCA of the nodes identified by 1.2.1.1 and 1.2.2.2 is 1.2. Determining the LCA of two nodes is useful, for instance, when searching XML documents based on a set of keywords. In this context, the user does not specify the size or type of the desired answer, and the system may return the smallest XML subtrees containing matches for all the user-specified keywords. It turns out that such smallest XML subtrees are exactly those rooted at the LCA of the nodes containing the keywords.

As just discussed, Dewey IDs provide more information than region-based IDs. The main drawback of Dewey IDs is their potentially much larger size. Also, the fact that IDs have lengths that may vary a lot within the same document, complicates processing.

2.3 Structural identifiers and updates

Observe that identifier of a node for all the forms mentioned above may change when the XML document that contains this node is updated. For instance, consider a node with Dewey ID 1.2.2.3. Suppose we insert a new first child to node 1.2. Then the ID of node 1.2.2.3 becomes 1.2.3.3.

In general, offset-based identifiers need to change even if a simple character is added to or removed from a text node in the XML document, since this changes the offsets of all nodes occurring in the document *after* the modified text node. Identifiers based on the (start, end) or (pre, post) model, as well as Dewey IDs, are not impacted by updates to the document's text nodes. One may also choose to leave them unchanged when removing nodes even full subtrees from the document. If we do that (leave the identification unchanged when subtrees are removed), we introduce gaps in the use of identifiers in all three methods, but these gaps do not affect in any way the computation of structural joins.

The management of insertions, on the other hand, is much more intricate. When inserting an XML node n_{new}, the identifiers of all nodes occurring after n_{new} in the preorder traversal of the tree, need to change. Depending on the ID model, such changes may also affect the IDs of n_{new}'s ancestors. The process of re-assigning identifiers to XML nodes after a document update is known as *re-labeling*.

In application scenarios where XML updates are expected to be frequent, re-labeling may raise important performance issues.

3 XML query evaluation techniques

We present in the next section, techniques for the efficient evaluation of XML queries, and in particular for tree pattern queries.

3.1 Structural join

The first techniques concern structural joins and can be seen as foundational to all the others. Structural joins are physical operators capable of combining tuples from two inputs, much in the way regular joins in the relation case do, but based on a structural condition (thus the name). Formally, let p_1 and p_2 be some partial evaluation plans in an XML database, such that attribute X in the output of p_1, denoted $p_1.X$, and attribute Y from the output of p_2, denoted $p_2.Y$, both contain structural IDs. Observe that the setting is very general, that is, we make no assumption on how p_1 and p_2 are implemented, which physical operators they contain etc. Let $\prec$ denote the binary relationship "isParentOf" and $\prec\!\!\prec$ denote the binary relationship "isAncestorOf". Formally then, the structural join of p_1 and p_2 on the condition that $p_1.X$ be an ancestor of $p_2.Y$ is defined as:

$$p_1 \bowtie_{X \prec\!\!\prec Y} p_2 = \{(t_1, t_2) \mid t_1 \in p_1, t_2 \in p_2, t_1.X \prec\!\!\prec t_2.Y\}$$

and the structural join on the parent relation $\prec$ is similarly defined by:

$$p_1 \bowtie_{X \prec Y} p_2 = \{(t_1, t_2) \mid t_1 \in p_1, t_2 \in p_2, t_1.X \prec t_2.Y\}$$

We have seen that expressive node IDs allow deciding just by comparing two IDs whether the respective nodes are related or not. Now, what is needed is an efficient way of establishing how the nodes from *sets* of tuples are related, i.e., how to efficiently evaluate a join of the form $p_1 \bowtie_{X \prec\!\!\prec Y} p_2$. Efficiency for a join operator means reducing its CPU costs, and avoiding to incur memory and I/O costs. For our discussion, let $|p_1|$ denote the number of tuples output by p_1, and $|p_2|$ the number of tuples output by p_2.

One can consider different kinds of joins:

Nested loop join The simplest physical structural join algorithms could proceed in *nested loops*. One iterates over the output of p_1 and for each tuple, one iterates over the output of p_2. However, this leads to CPU costs in $O(|p_1| \times |p_2|)$, since each p_1 tuple is compared with each p_2 tuple.

Hash join As in traditional relational database settings, one could consider *hash joins* that are often called upon for good performance, given that their CPU costs are in $O(|p_1| + |p_2|)$. However, hash-based techniques cannot apply here, because the comparisons that need to be carried are of the form "is id_1 an ancestor of id_2?" which do not lead themselves to a hash-based approach.

Stack-based join To efficiently perform this task, *Stack-based* structural joins operators have been proposed originally for *(start, end)* ID scheme. They can be used for other labeling schemes as well. We discuss these joins next.

To be able to use stack-based joins, the structural IDs must allow efficiently answering the following questions:

1. Is id_1 the identifier of the parent of the node identified by id_2? The same question can be asked for ancestor.

2. Does id_1 start after id_2 in preorder traversal of the tree? Or in other words, does the opening tag of the node identified by id_1 occur in the document after the opening tag of the node identified by id_2?

<root$_{(1,5)}$> <list$_{(2,4)}$> <list$_{(3,3)}$> <para$_{(4,2)}$> <para$_{(5,1)}$/> </para> </list> </list> </root>

Ordered by ancestor ID (list ID)

(2,4)	(4,2)
(2,4)	(5,1)
(3,3)	(4,2)
(3,3)	(5,1)

Ordered by descendantID (para ID)

(2,4)	(4,2)
(3,3)	(4,2)
(2,4)	(5,1)
(3,3)	(5,1)

Figure 9: Sample XML snippet and orderings of the (list ID, para ID) tuples.

3. Does id_1 end after id_2? Or in other words, does the closing tag of the node identified by id_1 occur in the document after the closing tag of the node identified by id_2?

Assuming each of these questions can be answered fast, say in constant time, based on the values of id_1 and id_2, stack-based structural joins can be evaluated in $\Omega(|p_1| + |p_2|)$, which is very efficient. Intuitively, we start with the two lists of identifiers. Condition (1) allows deciding whether a pair (i, j) with i from one list and j from the other is a solution. Because of Conditions (2) and (3), we can just scan the two lists (keeping some running stacks) and do not have to consider *all* such (i, j) pairs.

More precisely, two algorithms have been proposed, namely, StackTreeDescendant (or STD in short) and StackTreeAncestor (or STA in short). The algorithms are quite similar. They both require the inputp_1 tuples sorted in the increasing order[1] of $p_1.X$, and the input p_2 tuples in the increasing order of $p_2.Y$. The difference between STD and STA is the order in which they produce their output: STD produces output tuples sorted by the ancestor ID, whereas STA produces them sorted by the descendant ID.

To see why the two orders do not always coincide, consider the XML snippet shown in Figure 9, with the *(start, end)* IDs appearing as subscripts in the opening tags. In the figure, we show the (ancestor, descendant) ID pairs from this snippet, where the ancestor is a list, and the descendant is a paragraph (labeled "para").

Observe that the second and third tuples differ in the two tuple orders. This order issue is significant, since both STD and STA require inputs to be ordered by the IDs on which the join is to be made. Thus, when combining STD and STA joins in larger query evaluation plans, if the order of results in one join's output is not the one required by its parent operator, Sort operators may need to be inserted, adding to the CPU costs, and delaying the moment when the first output tuple is built. We now introduce the STD algorithm by means of an example, leaving STA as further reading.

Figure 10 shows a sample XML document with *(start, end)* IDs shown as subscripts of the nodes. In this example, the structural join to be evaluated must compute all pairs of (ancestor, descendant) nodes such that the ancestors are labeled b and the descendants are labeled g.

Figure 10 shows a sequence of snapshots during STD execution, with arrows denoting transitions from one snapshot to the next one.

In each snapshot, the first table shows the inputs, i.e., the ordered lists of IDs of the b nodes, respectively, of the g nodes. The algorithm's only internal data structure is a stack, in

[1] This is the lexicographical order, i.e., $(i, j) < (i', j')$ if $i < i'$ or if $i = i'$ and $j < j'$.

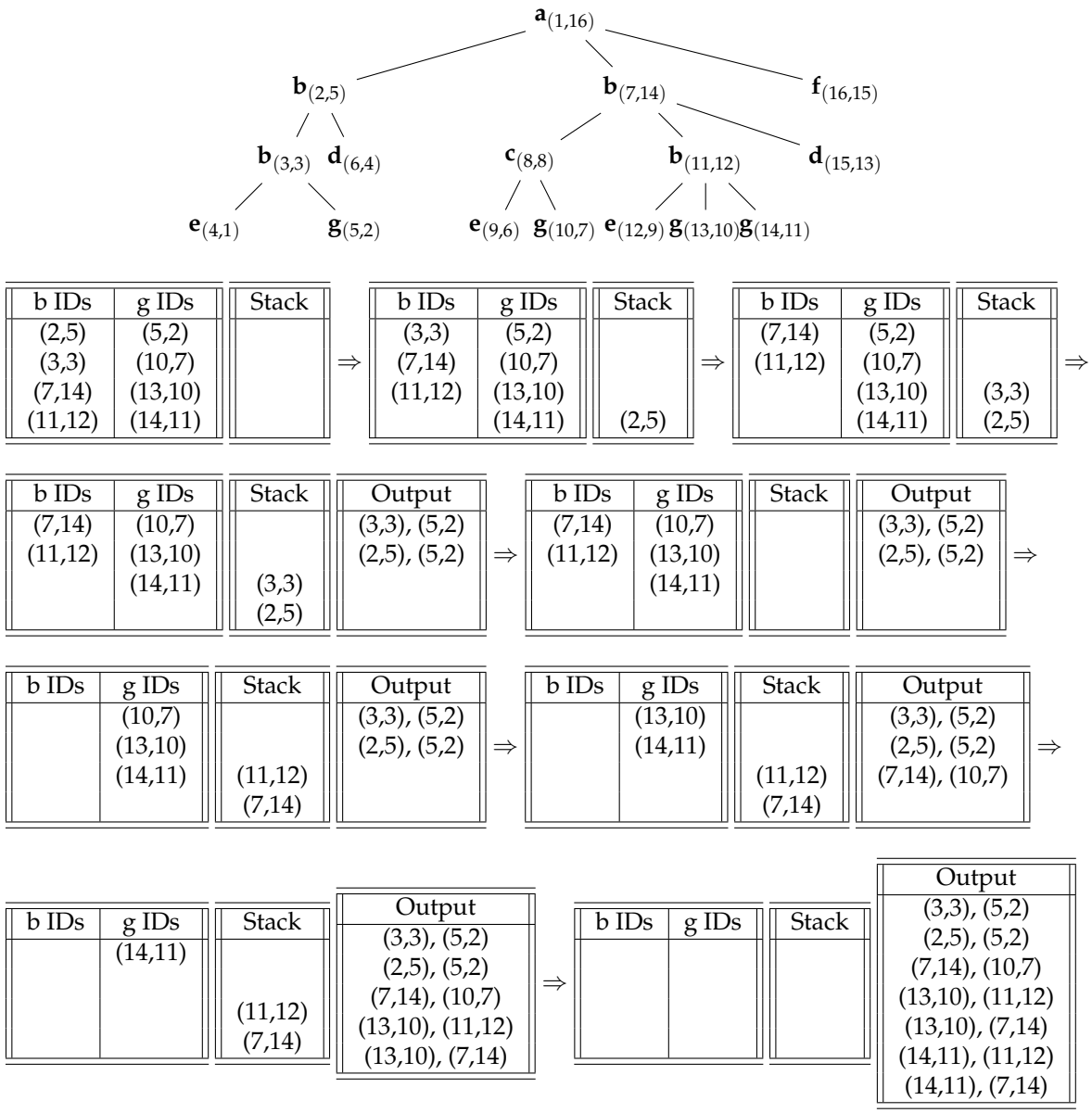

Figure 10: Sample XML tree, and successive snapshots of the inputs and stack during the execution of a StackTreeDescendant (STD) structural join.

which *ancestor* node IDs are all successively pushed, and from which they are popped later on.

STD execution starts by pushing the ancestor (that is, *b* node) ID on the stack, namely (2,5). Then, STD continues to examine the IDs in the *b* ancestor input. As long as the current ancestor ID is a descendant of the top of the stack, the current ancestor ID is pushed on the stack, without considering the descendant IDs at all. This is illustrated by the second *b* ID, (3,3) pushed on the stack, since it is a descendant of (2,5). The third ID in the *b* input, (7,14), is not a descendant of current stack top, namely (2,5). Therefore, STD stops pushing *b* IDs on the stack and considers the current descendant ID, to see if it has matches *on the stack*. It turns

out that the first g node, namely (5,2), is a descendant of both b nodes on the stack, leading to the first two output tuples. Observe that the stack content does not change when output is produced. This is because there may be further descendant IDs to match these ancestor IDs on the stack.

Importantly, *a descendant ID which has been compared with ancestor IDs on the stack and has produced output tuples, can be discarded after the comparisons*. In other words, we are certain that this descendant has encountered *all its ancestors from the ancestor stream*. This is because of the way in which ancestor IDs are pushed on the stack (push as long as they are ancestors of each other). Indeed, all the ancestors of a given g node, for instance, are on a single vertical path in the document, and therefore, they are guaranteed to be on the stack at once. In our example, once the (5,2) node ID has lead to producing output, it is discarded.

As the STD execution continues, the g ID (10,7) encounters no matches on the stack. Moreover, (10,7) occurs in the original 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 in document order. Thus, if the current g ID is after the stack nodes, all future g IDs will also occur "too late" to be descendants of the nodes in the current stack. Therefore, at this point, the stack is emptied. This explains why *once an ancestor ID has transited through the stack and has been popped away, no descendant ID yet to be examined could produce a join result with this ancestor ID*.

The previous discussion provides the two reasons for the efficiency of STD:

- a single pass over the descendants suffices (each is compared with the ancestor IDs on the stack only once);
- a single pass over the ancestors suffices (each transits through the stack only once).

Thus, the STD algorithm can apply in a streaming fashion, reading each of its inputs only once and thus with CPU costs in $\Omega(|p_1| + |p_2|)$.

Continuing to follow our illustrative execution, 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) which produces a result with (7,14) and is then discarded. The next descendant ID is (13,10), which leads to two new tuples added in the output, and similarly the descendant ID (14,11) leads to two more output tuples.

Observe in Figure 10 that, true to its name, the STD algorithm produces output tuples sorted by the descendant ID.

3.2 Optimizing structural join queries

Algorithm STD allows combining two inputs based on a structural relationship between an ID attribute of one plan and an ID attribute of the other. Using STD and the similar Stack-TreeAncestor (STA), one can compute matches for larger query tree patterns, by combining sets of identifiers of nodes having the labels appearing in the query tree pattern.

This is illustrated in Figure 11 which shows a sample tree pattern and a corresponding structural join plan for evaluating it based on collections of identifiers for a, b, c and d nodes. The plan in Figure 11 (b) is a first attempt at converting the logical plan into a physical executable plan. In Figure 11 (b), the first logical structural join is implemented using the STD algorithm, whose output will be sorted by b.ID. The second structural join can be implemented by STD (leading to an output ordered by c.ID) or STA (leading to an output ordered by b.ID),

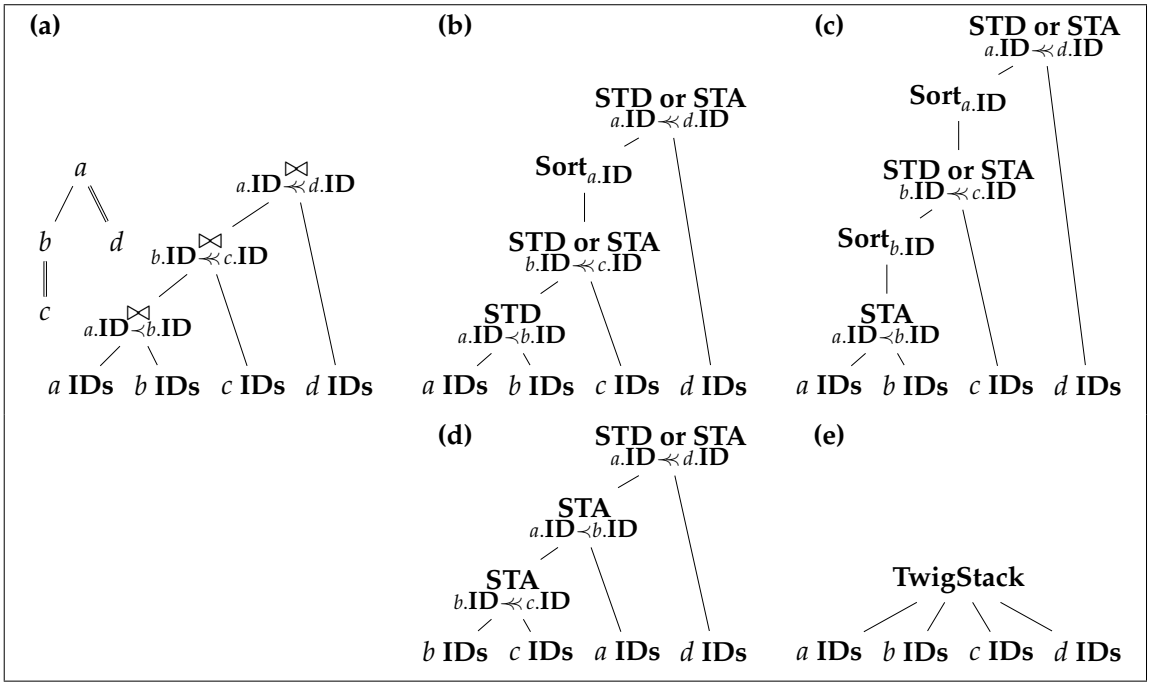

Figure 11: Tree pattern query and a corresponding logical structural join plan (a); possible physical plans (b)-(e).

but in both cases, the output of this second structural join is not guaranteed to be sorted by a.ID. Therefore, a Sort operator is needed to adapt the output of the second join to the input of the third join, which requires its left-hand input sorted by a.ID.

The plan in Figure 11 (c) uses STA instead of STD for the first structural join (between a.ID and b.ID). This choice, however, is worse than the one in Figure 11 (b), since a Sort operator is needed after the first join to re-order its results on b.ID, and a second Sort operator is needed to order the output of the second physical join, on a.ID. The Sort operators are blocking, i.e., they require completely reading their input before producing their first output tuple. Therefore, they increase the time to the first output tuple produced by the physical plan, and the total running time.

In contrast to the plans in Figure 11 (b) and (c), the plan in Figure 11 (d) allows evaluating the same query and does not require any Sort operator. Observe however that the plan does not apply joins in the same order as the other ones in Figure 11. In general, it turns out that for any tree pattern query, there exist some plans using the STD and STA physical operators, and which do not require any Sort (also called *fully pipelined plans*). However, one cannot ensure *a fully pipelined plan for a given join order*. For instance, in Figure 11, the reader can easily check that no fully pipelined plan exists for the join order $a \prec b$, $b \prec\prec c$, $a \prec\prec d$.

This complicates the problem of finding an efficient evaluation plan based on structural joins, because two optimization objectives are now in conflict:

- avoiding Sort operators (to reduce the time to the first output, and the total running time) and
- choosing the join order that minimizes the sizes of the intermediary join results.

Algorithms for finding efficient, fully-pipelined plans are discussed in [WPJ03].

3.3 Holistic twig joins

The previous Section showed how, in a query evaluation engine including binary structural join operators, one can reduce the total running time by avoiding Sort operators and reduce the total running time and/or the size of the intermediary join results, by chosing the order in which to perform the joins.

A different approach toward the goals of reducing the running time *and* the size of the intermediary results consists in devising a new (logical and physical operator), more precisely, an n-ary structural join operator, also called a *holistic* twig join. Such an operator builds the result of a tree pattern query in a single pass over all the inputs in parallel. This eliminates the need for storing intermediary results and may also significantly reduce the total running time.

Formally, let q be a tree pattern query having n nodes, such that for $1 \leq i \leq n$, the i-th node of the pattern is labeled a_i. Without loss of generality, assume that q's root is labeled a_1. For each node labeled a_i, $2 \leq i \leq n$, we denote by a_i^p is the parent of a_i in q.

We first define the *logical* holistic twig join operator. Assume available a set of logical sub-plans lp_1, lp_2, ..., lp_n such that for each i, $1 \leq i \leq n$, the plan lp_i outputs structural IDs of elements labeled a_i. The logical holistic structural join of lp_1, lp_2, ..., lp_n based on q, denoted $\bowtie_q (lp_1, lp_2, \ldots, lp_n)$, is defined as:

$$\sigma_{(a_2^p \prec\!\!\!\prec a_2) \wedge (a_3^p \prec\!\!\!\prec a_3) \wedge \ldots \wedge (a_n^p \prec\!\!\!\prec a_n)}(lp_1 \times lp_2 \times \ldots \times lp_n)$$

In the above expression, we assumed that all edges in q correspond to ancestor-descendant relationship. To account for the case where the edge between a_i and a_i^p is parent-child, one needs to replace the atom $a_i^p \prec\!\!\!\prec a_i$ in the selection, by $a_i^p \prec a_i$.

We now turn to discussing efficient *physical* holistic twig join operators. For ease of explanation, we first present an algorithm called PathStack, for evaluating *linear* queries only, and then generalize to the TwigStack algorithm for the general case.

Algorithm PathStack The algorithm uses as auxiliary data structures one stack S_i for each query node n_i labeled a_i. During execution, PathStack pushes IDs of nodes labeled a_i in the corresponding stack S_i. At the end of the execution, each stack S_i holds exactly those IDs of nodes labeled a_i, which participate to one or more result tuples.

PathStack works by continuously looking for the input operator (let's call it iop_{min}, corresponding to the query node n_{min} labeled a_{min}) whose first ID has the smallest *pre* number among all the input streams. This amounts to finding the first element (in document order) among those not yet processed, across all inputs. Let's call this element e_{min}.

Once e_{min} has been found, PathStack inspects all the stacks, looking for nodes of which it can be guaranteed that they will not contribute to further query results. In particular, this is the case for any nodes *preceding* e_{min}, i.e., ending before the start of e_{min}. It is easy to see that such nodes cannot be ancestors neither of e_{min}, nor of any of the remaining nodes in any of the inputs, since such nodes have a starting position even bigger than e_{min}'s. PathStack pops all such entries, from all stacks.

PathStack then pushes the current n_{min} entry on $S_{a_{min}}$, if and only if suitable ancestors of this element have already been identified and pushed on the stack of a_{min}^p, the parent node of a_{min} in the query. If this is the case and a new entry is pushed on $S_{a_{min}}$, importantly, a

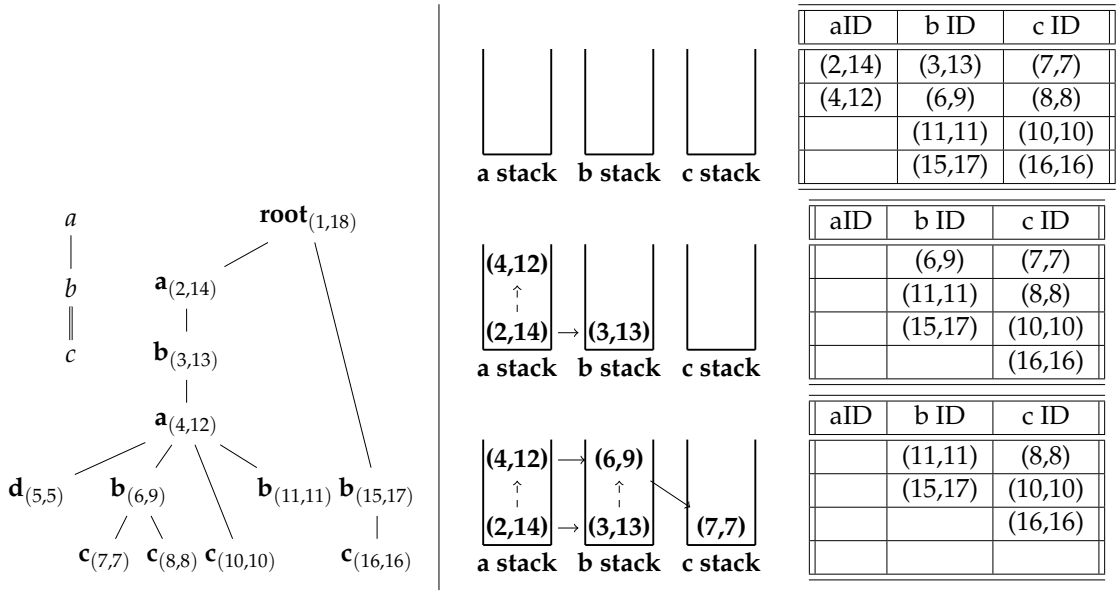

Figure 12: Sample tree pattern, XML document, and stacks for the PathStack algorithm.

pointer is stored *from the top entry in* $S_{a^p_{min}}$, *to the new (top) entry in* $S_{a_{min}}$. Such pointers record the connections between the stack entries matching different query nodes, and will be used to build result tuples out (see below). Finally, PathStack advances the input operator iop_{min} and resumes its main loop, identifying again the input operator holding the first element (in document order) not yet processed etc.

When an entry is pushed on the stack corresponding to the query *leaf*, we are certain that the stacks contain matches for all its ancestors in the query, matches that are ancestors of the leaf stack entry. At this point, two steps are applied:

1. Result tuples are built out of the entries on the stacks, and in particular of the new entry on the stack of the query leaf;

2. This new entry is popped from the stack.

Figure 12 illustrates the algorithm through successive snapshots of the input streams and stacks, for the sample document shown at left in the Figure. The first execution snapshot is taken at the start: all stacks are empty and all streams are set to their first element. The search for the iop_{min} operator is done by comparing the top elements in all the streams. In this case, the smallest element ID is (2,14) in the stack of a, therefore n_{min} is set to this node, and n_{min} is pushed on the correspondint stack S_a. The stream of a advances to the next position and we seek again for the new iop_{min}, which turns out to be the stream of b IDs. The new value of n_{min} is the b node identified by (4, 13); this node is therefore pushed on the b stack and the pointer from (2,14) to (3,13) records the connection between the two stack entries (which corresponds to a structural connection in the document). The b stream is advanced again, and then iop_{min} is found to be the a stream, leading to pushing (4, 12) on the a stack.

At this point, we must check the entries in the stacks corresponding to descendants of a, and possibly prune "outdated" entries (nodes preceding the newly pushed a in the document). In our case, there is only one entry in the b stack. Comparing its ID, (3, 13) with the ID (4, 12)

of the newly pushed *a* element, we decide that the (3, 13) entry should be preserved in the *b* stack for the time being.

After advancing the *a* stream, PathStack's data structures take the form shown in the middle snapshot at right in Figure 12.

The process is repeated and pushes the (6, 9) identifier on the *b* stack. No entry is eliminated from the *c* stack since it is still empty. The algorithm then pushes the identifiers (7,7) on the *c* stack and connects it to the current to p of the *b* stack, namely (6, 9). This is the situation depicted at the bottom right of Figure 12.

Observe that (7, 7) is a match of the *c* node which is a leaf in the query. Thus, at this point, the following result tuples are built out of the stacks, based on the connections between stack entries:

*a***ID**	*b***ID**	*c***ID**
(2,14)	(3,13)	(7,7)
(2,14)	(6,9)	(7,7)
(4,12)	(6,9)	(7,7)

The (7,7) entries is then popped from its stack, and (8,8) takes its place, leading similarly to the result tuples:

*a***ID**	*b***ID**	*c***ID**
(2,14)	(3,13)	(8,8)
(2,14)	(6,9)	(8,8)
(4,12)	(6,9)	(8,8)

Now, the smallest element not yet processed is identified by (10, 10) in the stream of *c* elements. This element is *not* pushed in the *c* stack, because it is not a descendant of the current top of the *b* stack (namely, (6, 9)).

Continuing execution, the *b* element identified by (11, 11) is examined. It leads to expunging from the stacks all entries whose end number is smaller than 11, and in particular, the *b* entry (6,9). (11,11) is then pushed on the *b* stack. When (15,1) is read from the *b* stream, all existing *a* and *b* entries are popped, however (15, 1) is not pushed due to the lack of a suitable ancestor in the *a* stack. Finally, (16,16) is not pushed, by a similar reasoning.

The algorithm has two features of interest.

- No intermediary result tuples: results are built directly as n-tuples, where n is the size of the query. This avoids the multiplication, say, of every *a* element by each of its *b* descendants, before learning whether or not any of these *b*s had a *c* descendant:
- Space-efficient encoding of results: Thanks to the pointer structures and to the way the algorithm operates, the total size of the entries stored on the stacks at any point in times is bound by $|d| \times |q|$, however, such stack entries allow encoding up to $|d|^{|q|}$ query results, a strong saving in terms of space (and thus, performance).

Algorithm TwigStack This algorithm generalizes PathStack with support for multiple branches. The ideas are very similar, as the main features (no intermediary results and space-efficient encoding of twig query results) are the same. Figure 13 shows a twig (tree) pattern query derived from the linear query of Figure 12 by adding a lateral *c* branch. The document is the same in both Figures.

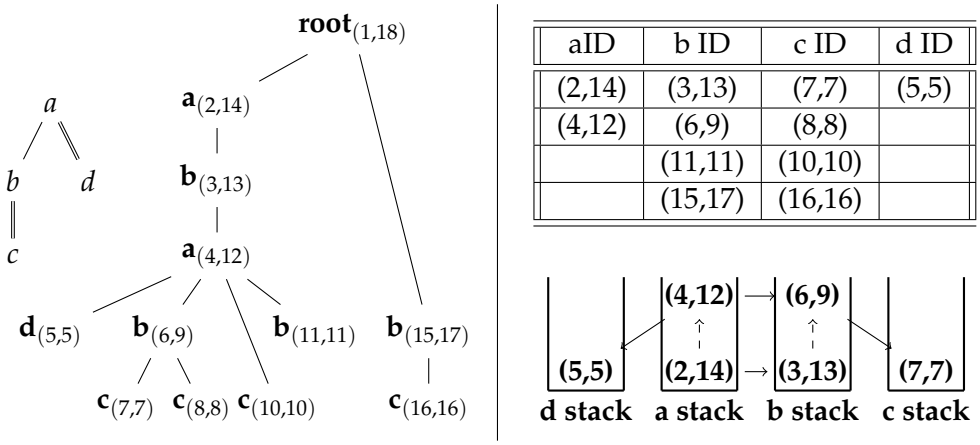

Figure 13: Sample tree pattern, XML document, and stacks for the TwigStack algorithm.

At right in Figure 13 we show the input streams at the beginning of the execution (top right) as well as a snapshot of the stacks at the moment during execution when (7,7) has been pushed on the *c* stack. Now that there is a match for each query leaf node, the following tuples are output:

*a***ID**	*b***ID**	*c***ID**	*d***ID**
(2,14)	(3,13)	(7,7)	(5,5)
(2,14)	(6,9)	(7,7)	(5,5)
(4,12)	(6,9)	(7,7)	(5,5)

One can show [BKS02] that if the query pattern edges are only of type ancestor/descendant, TwigStack is I/O and CPU optimal among all sequential algorithms that read their inputs in their entirety.

4 Further reading

The XML storage, indexing, and query processing area is still very active and has witnessed a wide variety of techniques and algorithms.

Interval-based node IDs were first proposed in the context of efficient relational storage for XML documents in XRel [YASU01] and XParent [JLWY02]. As pointed out in the original structural join paper [AKJP+02], they are however inspired by known works in information retrieval systems, which assign positional identifiers to occurrences of words in text, and closely related to the Dietz numbering scheme [Die82]. Dewey-style XML identifiers were used for instance in [SSB+00].

Compact labeling of trees is now a mature topic, see [AAK+06, KMS07].

Simple region and Dewey-based ID schemes suffer when the document is updated, since this may require extensive node re-labeling. ORDPATH IDs [OOP+04] remedy to this problem. An ORDPATH ID is a dot-separated sequence of labels, in the style of Dewey, but using integer labels and playing on the odd/even character of the numbers to allow inserting nodes at any place without re-labeling any existing node. ORDPATHs were implemented within the Microsoft SQL Server's XML extension. Other interesting node identifier schemes, trading

between compactness, efficient query evaluation and resilience to updates, are described in [WLH04], [LLCC05] and among the most recent works, in [XLWB09].

Stack-based structural joins operators have been proposed in [AKJP+02]. Heuristics for selecting the best join order in this context are presented in [WPJ03]. The TwigStack algorithm is presented in [BKS02]. Numerous improvements have been subsequently brought to TwigStack, mainly by adding B-Tree style indexes on the inputs. This enables skipping some of the input IDs, if it can be inferred that they will not contribute to the query result.

Given the robustness of relational database management systems, many works have considered using relational DBMSs to store and query XML documents. The first works in this area are [FK99, STZ+99]. An interesting line of works starting with [Gru02, GvKT03, GST04] considered the efficient handling of XML data in relational databases relying on (start, end) identifiers. The special properties satisfied by such IDs (and which are due to the nested nature of XML documents) is exploited in these works to significantly speed up the processing of XPath queries and more generally, of (XQuery) tree pattern queries. These works also provide an extensive approach for faithfully translating XML queries expressed in XPath and XQuery, into relational algebra (endowed with minimal extensions). In a more recent development, this algebraic approach has been compiled to be executed by the MONETDB column-based relational store [BGvK+06].

Manipulating XML documents within relational databases and *jointly with relational data* has been considered quite early on in the industrial world. The ISO standard SQL/XML [ISO03] is an extension to the SQL language, allowing a new elementary data type `xml`. A relation can have columns of type `xml`, which can be queried using XPath/XQuery invoked by a built-in function, within a regular SQL query. SQL/XML also provides facilities for declaratively specifying a mapping to export relational data in an XML form.

5 Exercises

The following exercises will allow you to derive a set of results previously established in the literature, which have been used to efficiently implement tree pattern query evaluation.

Exercise 5.1 *(inspired from [GvKT04])*

Recall the notions of pre, post and level numbers assigned to nodes in an XML document:

- *the pre number of node n, denoted n.pre, is the number assigned to n when n is first reached by a pre-order traversal of the document;*
- *the post number of node n, denoted n.post, is the number assigned to n when n is first reached by a post-order traversal of the document;*
- *the level number of node n is* 0 *if n is the root element of the document, otherwise, it is the number of edges traversed on the path from the root element to n.*

Moreover, we denote by $size(n)$ the number of descendants of a node n. We denote the height of a document d by $h(d)$.

1. *Prove that for any XML node n:*

$$n.pre - n.post + size(n) = n.level$$

2. *Let n be an XML node and n_{rl} be the* rightmost leaf *descendant of n. In other word, n_{rl} is attained by navigating from n to its last child r_1 (if it exists), from r_1 to its last child r_2 (if its exist) and so on, until we reach a childless node, i.e. the leaf n_{rl}. Prove that:*

$$n_{rl}.pre \leq h(d) + n.post$$

3. *Using point 2. above, show that for any descendant m of a node n:*

$$n.pre \leq m.pre \leq h(d) + n.post$$

4. *Let n_{ll} be the* leftmost leaf *descendant of n. Node n_{ll} is attained from n by moving to the first child l_1 of n (if such a child exists), then from l_1 to its first child l_2 (if such a child exists) and so on, until we reach a leaf (which is n_{ll}). Show that:*

$$n_{ll}.post \geq n.pre - h(d)$$

5. *Using the answer to point 4. above, show that for any node m descendant of a node n:*

$$n.pre - h(d) \leq m.post \leq n.post$$

6. *Assume an XML storage system based on the following relations:*

`Doc(ID, URL, h)`	*For each document, its internal ID, its URL, and its height*
`Node(docID, pre, post, level, label)`	*For each XML node, the ID of its document, its pre, post and level numbers, and its label*

Let `dID` *be the identifier of a document d, and* `npre`, `npost` *and* `nlevel` *be the pre, post and level numbers of an XML node $n \in d$. Write the most restrictive (selective) SQL query retrieving the pre numbers and labels of all n descendants in d.*

Exercise 5.2 *(inspired from [WPJ03]) We consider a tree pattern query evaluation engine based on structural identifiers and binary structural joins (implemented by the STA and STD operators). For any node label l, we denote by l**ID** a relation storing the structural identifiers of all nodes labeled l. Recall that a physical plan without any Sort operator is termed a* fully pipelined plan*. For instance, $STA_{a \prec\!\!< b}(a\mathbf{ID}, STA_{b \prec c}(b\mathbf{ID}, c\mathbf{ID}))$ is a fully pipelined plan for the query $//a[//b/c]$.*

1. *Let q_{flat} be a "flat" XPath query of the form $/r[.//a_1][.//a_2]\ldots//a_k$. Propose a fully pipelined plan for evaluating q_{flat}.*

2. *For the same query q_{flat}, consider the following the join order ω_1: first verify the predicate $r \prec\!\!< a_1$, then the predicate $r \prec\!\!< a_2$, then $r \prec\!\!< a_3$ and so on, until the last predicate $r \prec\!\!< a_k$. Is there a fully pipelined plan for evaluating q_{flat} respecting the join order ω_1?*

3. *Let q_{deep} be a "deep" XPath query of the form $/r//a_1//a_2\ldots//a_k$. Propose a fully pipelined plan for evaluating q_{deep}.*

4. *For the query q_{deep} introduced above, let ω_2 be a join order which starts by verifying the $r \prec\!\!\prec a_1$ predicate, then the predicate $a_1 \prec\!\!\prec a_2$, then the predicate $a_2 \prec\!\!\prec a_3$ and so on, until the last predicate $a_{k-1} \prec\!\!\prec a_k$. Is there a fully pipelined plan for evaluating q_{deep} respecting the join order ω_2? If yes, provide one. If not, explain why.*

5. *For the same query q_{deep}, now consider the join order ω_3 which starts by verifying the predicate $a_{k-1} \prec\!\!\prec a_k$, then the predicate $a_{k-2} \prec\!\!\prec a_{k-1}$ and so on, until the predicate $r \prec\!\!\prec a_1$. Is there a fully pipelined plan for evaluating q_{deep} respecting the join order ω_3? If yes, provide one. If not, explain why.*

6. *Show that for any XPath query q, there is at least a fully pipelined plan for evaluating q.*

7. *Propose an algorithm that, given a general XPath query q and a join order ω, returns a fully pipelined plan respecting the join order ω if such a plan exists, and returns failure otherwise.*

Exercise 5.3 *(Inspired from [BKS02]) Consider a tree pattern query q and a set of stacks such as those used by the PathStack and TwigStack algorithms. Propose an algorithm which, based on stack entries containing matches for all nodes in q, computes the tuples corresponding to the full answers to q:*

- *in the case where q is a linear path query (algorithm PathStack);*
- *in the general case where q is a twig pattern query (algorithm TwigStack).*

[AAK+06] Serge Abiteboul, Stephen Alstrup, Haim Kaplan, Tova Milo, and Theis Rauhe. Compact labeling scheme for ancestor queries. *SIAM J. Comput.*, 35(6):1295–1309, 2006.

[AKJP+02] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, 2002.

[BGvK+06] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proc. ACM Intl. Conf. on the Management of Data (SIGMOD)*, pages 479–490, 2006.

[BKS02] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proc. ACM Intl. Conf. on the Management of Data (SIGMOD)*, 2002.

[Die82] P. Dietz. Maintaining order in a linked list. In *Proc. ACM SIGACT Symp. on the Theory of Computing (STOC)*, 1982.

[FK99] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[Gru02] Torsten Grust. Accelerating XPath location steps. In *Proc. ACM Intl. Conf. on the Management of Data (SIGMOD)*, pages 109–120, 2002.

[GST04] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL hosts. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 252–263, 2004.

[GvKT03] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 524–525, 2003.

[GvKT04] Torsten Grust, Maurice van Keulen, and Jens Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. on Database Systems*, 29:91–131, 2004.

[ISO03] ISO/IEC 9075-14:2003, Information technology – Database languages – SQL – Part 14: XML-Related Specifications (SQL/XML), 2003.

[JLWY02] Haifeng Jiang, Hongjun Lu, Wei Wang, and Jeffrey Xu Yu. XParent: An efficient RDBMS-based XML database system. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 335–336, 2002.

[KMS07] Haim Kaplan, Tova Milo, and Ronen Shabo. Compact labeling scheme for XML ancestor queries. *Theory Comput. Syst.*, 40(1):55–99, 2007.

[LLCC05] Jiaheng Lu, Tok Wang Ling, Chee Yong Chan, and Ting Chen. From region encoding to extended Dewey: On efficient processing of XML twig pattern matching. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, 2005.

[OOP+04] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-friendly XML node labels. In *Proc. ACM Intl. Conf. on the Management of Data (SIGMOD)*, pages 903–908, 2004.

[SSB+00] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently publishing relational data as XML documents. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 65–76, 2000.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, 1999.

[WLH04] Xiaodong Wu, Mong Li Lee, and Wynne Hsu. A prime number labeling scheme for dynamic ordered XML trees. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, 2004.

[WPJ03] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 443–454, 2003.

[XLWB09] Liang Xu, Tok Wang Ling, Huayu Wu, and Zhifeng Bao. DDE: from Dewey to a fully dynamic XML labeling scheme. In *Proc. ACM Intl. Conf. on the Management of Data (SIGMOD)*, 2009.

[YASU01] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Trans. on Internet Technology*, 1(1):110–141, 2001.