Matching pairwise divergent paths in XML streams

Christian Rinderknecht

Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary
`rinderkn@caesar.elte.hu`

10th December 2013

Abstract

It is more and more common to query XML databases using the path language XPath or the more expressive language XQuery. A lot of attention has been devoted to solve efficiently the matching of streams of XML elements against XPath expressions, in particular when the relationship between the query nodes is either parent/child or ancestor/descendant. In the latter case, the semantics of XPath allows two descendants to be themselves in an ancestor/descendant relationship. In many cases, this leads to too general queries. We propose a novel primitive operation as a stricter interpretation of the ancestor/descendant relation that restricts the matches to the ending nodes of paths diverging from a common node in the data. We propose and compare several algorithms for answering this new type of query.

Keywords: divergent paths, XML database, tree-pattern matching, XPath

1 Introduction

The XPath expression `a//b[//c]` is interpreted as "`a//b` and `a//c`", if we omit the unique target semantics, according to which XPath expressions evaluate to a sequence of nodes with same tag, here `b`. Many research papers [5, 11, 13] are devoted to matching efficiently these expressions, considered as tree patterns or queries, against XML trees as a whole or streams of XML elements, either by solving the binary joins and then stitching them back to answer the original query [1, 4, 7, 12] or by considering them as a whole [3, 8, 9]. The standard interpretation of the XPath expressions implies that, in the previous example, `b` may be an ancestor or a descendant of `c`, but it is striking that most authors assume in their examples that the nodes matched by `b` and `c` are *not* in an ancestor or descendant relationship. We surmise that this tacit assumption for picking examples is often closer to what the query specifier or

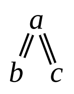
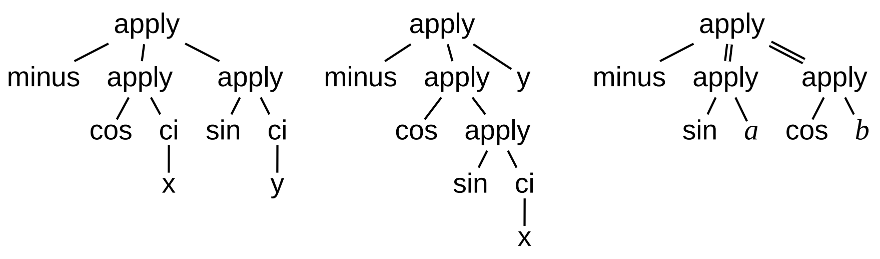


Figure 1: a//b[//c]



a. $\cos x - \sin y$ **b.** $\cos(\sin x) - y$ **c.** Pattern with variables a and b.

Figure 2: Two MathML trees and a query.

reader has in mind, rather than the usual interpretation. In other words, the reader of an XPath query probably tends to conceive the query as a tree, as in FIGURE 1, and to assume that the matching subtree is isomorphic, instead of reading the query as a set of binary joins that, once stitched together, may yield a degenerate case, that is, a path. In particular, the expression `a//b[//b]` is equivalent to `a//b` and there is no way in XPath to actually denote two different `b` nodes on two divergent paths, i.e., no path including the other, as the graphic representation seems to suggest.

But can we rely solely on this psychological assumption to convey another interpretation of the XPath expressions? We believe that, beyond the useful purpose of avoiding misunderstandings by making more assumptions explicit, it is indeed useful to suppose sometimes that the matched subtree is isomorphic to the query tree, because this allows some additional constraints to be implicitly included in the query.

As an illustration, consider in FIGURE 2 the MathML trees corresponding to the expressions $\cos x - \sin y$ and $\cos(\sin x) - y$. These trees have very different interpretations, despite having a structure close enough to be matched by the same pattern. By interpreting the query pattern in a more restrictive way than in XPath, we can get less matches. For example, if we want to match the difference between a possibly nested sine application and a possibly nested cosine application, *but not their composition*, we can say that the two nodes apply in the query in FIGURE 2c cannot match nodes which are in a descendant or ancestor relationship, and then the query only matches the tree in FIGURE 2a. Furthermore, since we know that the subtraction is not commutative when interpreting these formulas, we can impose that the two matched nodes apply must follow the *same order* as in the pattern, i.e., the sibling order, yielding here no match at all.

The paper is structured as follows. We start with the next section 2 by stating mode precisely the problem we want to solve; then we propose a naive solution in section 3; follows a stack-based algorithm, in section 4, and an index-based procedure, in section 5; this paper being concluded in section 6 with a short comparison of the proposed algorithms and possible future work.

2 Context and problem

Streaming of XML elements A *tag* is a pair made of a *kind*, noted κ, and a status "opening" or "closing". An *element*, noted ε, is a triple made of an opening tag, some contents and a closing tag of the same kind as the opening tag. The contents is made of a series of plain text and other (embedded) elements. Elements can be empty, in which case the contents is empty. By extension, an element is said to be of kind κ if its tags are of kind κ. Let us define the function K on elements which gives the kind of its argument. A set of kinds is noted $\mathcal{K}$. Kinds are totally ordered. For example, let us consider the following original XML file (left column) where there is only one tag per numbered line.

```
 1 <c> text$_1$                        <c range="1--22"> text$_1$ </c>
 2   <a> text$_2$                      <a range="2--13"> text$_2$ </a>
 3     <b> text$_3$                    <b range="3--6"> text$_3$ </b>
 4       <a> text$_4$                  <a range="4--5"> text$_4$ </a>
 5       </a>
 6     </b>
 7     <b> text$_5$                    <b range="7--8"> text$_5$ </b>
 8     </b>
 9     <c> text$_6$                    <c range="9--12"> text$_6$ </c>
10       <a> text$_7$                  <a range="10--11"> text$_7$ </a>
11       </a>
12     </c>
13   </a>
14   <a> text$_8$                      <a range="14--15"> text$_8$ </a>
15   </a>
16   <b> text$_9$                      <b range="16--21"> text$_9$ </b>
17     <c> text$_{10}$                 <c range="17--20"> text$_{10}$ </c>
18       <b> text$_{11}$               <b range="18--19"> text$_{11}$ </b>
19       </b>
20     </c>
21   </b>
22 </c>
```

Here, the kinds of elements are a, b and c, i.e., $\mathcal{K} = \{a, b, c\}$. Elements are streamed by increasing order of their opening tag line number in the XML file, as shown in the right column, where we recorded the line numbers of the opening tags and the matching closing tags into an attribute `range`. This scheme works if we assume one tag per line, otherwise tags must be numbered by order of appearance in the file. Finally, let us assume that there is a function T which takes an element and returns its textual contents (a series of pieces of text corresponding to the text nodes). This allows us to abstract away the text in the elements.

Let $L(\varepsilon)$ and $U(\varepsilon)$ be respectively the lower bound of the range of element ε and the upper bound of the range of ε. By definition, element ε_1 is said to be *lower than* element ε_2, noted $\varepsilon_1 < \varepsilon_2$, if and only if $L(\varepsilon_1) < L(\varepsilon_2)$. Elements are streamed by increasing lower bounds of their range.

Al-Khalifa et al. [1] assume that the database provides multiple streams of sorted elements of the same kind of tag. In our example, we have the following three streams: [a_1, a_2, a_3, a_4], [b_1, b_2, b_3, b_4] and [c_1, c_2, c_3]. We can interleave these streams into one stream by picking repeatedly the minimum element of the streams, like merge sort: [c_1, a_1, b_1, a_2, b_4, c_2, a_3, a_4, b_2, c_3, b_3]. (This unique stream does not need to be statically constructed by the database.) In this paper, we prefer to assume a unique stream of sorted elements, although, for the sake of clarity, we keep the subscripting of tags as in the multiple-streams framework. The following table on the left side shows the stream as the interleaving of multiple streams, whilst the right table displays the same stream with a single-stream view, i.e., considering a generic series of elements [ε_1, ε_2, ..., ε_{11}], with a supplementary column K for the kind of element.

	T	L	U
c_1	$text_1$	1	22
a_1	$text_2$	2	13
b_1	$text_3$	3	6
a_2	$text_4$	4	5
b_4	$text_5$	7	8
c_2	$text_6$	9	12
a_3	$text_7$	10	11
a_4	$text_8$	14	15
b_2	$text_9$	16	21
c_3	$text_{10}$	17	20
b_3	$text_{11}$	18	19

	K	T	L	U
ε_1	c	$text_1$	1	22
ε_2	a	$text_2$	2	13
ε_3	b	$text_3$	3	6
ε_4	a	$text_4$	4	5
ε_5	b	$text_5$	7	8
ε_6	c	$text_6$	9	12
ε_7	a	$text_7$	10	11
ε_8	a	$text_8$	14	15
ε_9	b	$text_9$	16	21
ε_{10}	c	$text_{10}$	17	20
ε_{11}	b	$text_{11}$	18	19

Containment and disjointedness Since the stream of elements is computed from a valid XML file, any two elements in the stream are either in a *containment* or else a *disjointedness* (non-overlapping) relationship. By definition, an element ε_1 is contained in ε_2 (or "is a descendant of ε_2"), noted $\varepsilon_1 \sqsubset \varepsilon_2$, if $L(\varepsilon_2) < L(\varepsilon_1)$ and $U(\varepsilon_1) < U(\varepsilon_2)$. Containment is neither reflexive (it is strict inclusion) nor symmetric, but it is transitive. Elements ε_1 and ε_2 are disjoint, noted $\varepsilon_1 \sharp \varepsilon_2$, if and only if $U(\varepsilon_1) < L(\varepsilon_2)$ or $U(\varepsilon_2) < L(\varepsilon_1)$, i.e., $\varepsilon_1 \not\sqsubset \varepsilon_2$ and $\varepsilon_2 \not\sqsubset \varepsilon_1$. Disjointedness is neither reflexive nor transitive, but it is symmetric.

Problem statement Let us note $\mathcal{E}$ the infinite set of all possible XML elements. A *pattern*, or *query*, graphically represented in FIGURE 3, is a tuple of kinds κ_i, noted $\langle \kappa_1 \mid \kappa_2, \ldots, \kappa_n \rangle$. A *stream* of XML elements is a series [ε_1, ε_2, ...], such that $i < j \Rightarrow \varepsilon_i < \varepsilon_j$. A *complete match* of a query q, noted μ_q, is a finite mapping from the indexes of the kinds in the query $q = \langle \kappa_1 \mid \kappa_2, \ldots, \kappa_n \rangle$ to $\mathcal{E}$, such that

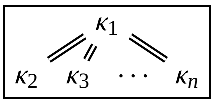


Figure 3: Query $\langle \kappa_1 \mid \kappa_2, \ldots, \kappa_n \rangle$

$$1 \leqslant i \leqslant n \Rightarrow K \circ \mu_q(i) = \kappa_i, \tag{1}$$

$$2 \leqslant i \leqslant n \Rightarrow \mu_q(i) \sqsubset \mu_q(1), \tag{2}$$

$$2 \leqslant i, j \leqslant n \Rightarrow \mu_q(i) \not\sqsubset \mu_q(j). \tag{3}$$

For example, let $q = \langle a \mid b, b\rangle$ and the input stream [d_1, a_1, c_1, b_1, c_2, a_2, b_2], then there is a match μ_q which satisfies $\mu_q(1) = a_1$, $\mu_q(2) = b_1$, $\mu_q(3) = b_2$. Alternatively, we can write instead $\mu_q = [1 \mapsto a_1, 2 \mapsto b_1, 3 \mapsto b_2]$ (the order of the bindings in the map is not meaningful). This is an *ordered match* because it enjoys the additional condition: $i < j \Rightarrow \mu_q(i) < \mu_q(j)$. Otherwise, a match is said to be *unordered*. For example, $[1 \mapsto a_1, 2 \mapsto b_2, 3 \mapsto b_1]$ is an unordered match. Conditions (1) and (2) are the standard interpretation of XPath queries. We add here condition (3), which constrains the elements matching the pattern descendants to be pairwise disjoint.

This problem is perhaps more intuitive as a property on XML trees.

The XML trees We make no assumption here on whether the outgoing stream is generated or not from an XML file actually stored in the database, but thinking in terms of that tree is intuitive. For example, FIGURE 4 displays a tree from which the stream [d_1, a_1, c_1, b_1, c_2, a_2, b_2, d_2, a_3, b_3, c_3] is produced by a preorder traversal. Note that two elements are in a containment relationship if and only if their corresponding nodes both belong to the same rooted path, i.e., a path including the root. Now, let us assume the tree in FIGURE 4, and a query $\langle a \mid b, c\rangle$, then the unique ordered match is $[1 \mapsto a_1, 2 \mapsto b_1, 3 \mapsto c_2]$. We can think of a match as a map from the nodes of the query tree to some nodes of the XML tree (condition (1)) such that the descendants in the query tree are mapped to descendants of the mapping of the query root (condition (2)) and such that these descendants are located in paths diverging from the mapping of the query root (condition (3)). Coming back to our example, if we are interested in the unordered matches as well, we must add $[1 \mapsto a_1, 2 \mapsto b_2, 3 \mapsto c_1]$. By contrast, if we interpret the query as in XPath, i.e., allowing containment between descendants, we would have to add $[1 \mapsto a_1, 2 \mapsto b_1, 3 \mapsto c_1]$, $[1 \mapsto a_1, 2 \mapsto b_2, 3 \mapsto c_2]$ and $[1 \mapsto a_3, 2 \mapsto b_3, 3 \mapsto c_3]$.

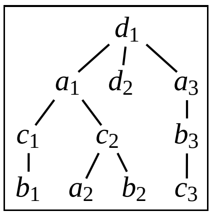


Figure 4: XML tree with tags a, b, c and d

Rightmost branches The *rightmost branch* of a tree is the longest rooted path made of the successive rightmost children. In the example at FIGURE 4, the rightmost branch is [d_1, a_3, b_3, c_3]. Adding a new node, i.e., the next available element in the input stream, changes only the rightmost branch because the elements are ordered by increasing positions of their opening tags (preorder traversal). Consider previous stages of our

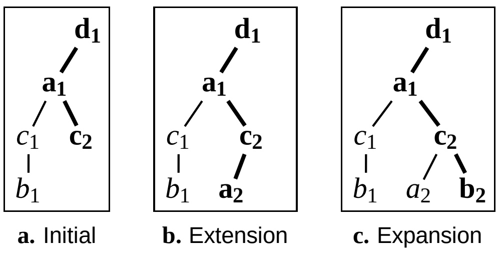


Figure 5: Rightmost branches (in bold)

XML tree in FIGURE 5: before adding a_2 (FIGURE 5a), after adding a_2 (FIGURE 5b) and after adding b_2 (FIGURE 5c). This sequence illustrates the fact that inserting a node either extends the rightmost branch (as adding a_2) or creates a new branch rooted in the previous rightmost branch (as adding b_2). We call the first case a *rightmost extension* and the second is known as *rightmost expansion* of a tree in the mining literature (see Asai et al. [2]).

3 Naive approach

Given a query $\langle \kappa_1 \mid \kappa_2, \ldots, \kappa_n \rangle$, a naive approach to solve our problem consists in storing the elements read so far in lists according to their kind (as in the multiple streams framework) and, when a new element is read, we check for all the combinations of elements that make up a match. This procedure is inefficient because it reconsiders previous invalid combinations after an element is read. We can easily improve it by storing all the partial matches found so far and then trying to complete *only them* with one more element each time, until a complete match is found. For example, let us reconsider the stream [d_1, a_1, c_1, b_1, c_2, a_2, b_2, d_2, a_3, b_3, c_3] from the XML tree in FIGURE 4, and let us seek both ordered and unordered matches to the query $\langle a \mid b, c \rangle$. In FIGURE 6, the first line corresponds to the sorted list of a-elements read so far, the second to the b-elements etc. Each column shows the state of the lists after each element is read from the input stream. At each step, i.e., for each column, the new valid combinations are respectively $\varnothing$, $\{a_1\}$, $\{a_1, a_1c_1\}$, $\{a_1, a_1c_1, a_1b_1\}$, $\{a_1, a_1c_1, a_1b_1, \mathbf{a_1b_1c_2}, a_1c_2\}$, $\{a_1, a_1c_1, a_1b_1, \mathbf{a_1b_1c_2}, a_1c_2, a_2\}$, $\{a_1, a_1c_1, a_1b_1, \mathbf{a_1b_1c_2}, a_2, a_1c_2, \mathbf{a_1b_2c_1}, a_1b_2\}$ — solutions in bold. Note that, since the kinds in this query are pairwise distinct, we can simplify the notation for complete and partial matches by just enumerating elements, like a_1c_1 and a_1b_2 etc. The drawbacks of this procedure are that it uses too much memory and it is too slow. Indeed, it needs to keep in memory the complete lists of elements and it tries to form new combinations with elements which are not disjoint, like $a_1b_1c_1$ in our example ($b_1 \sqsubset c_1$).

$\varnothing$	a_1	a_1	a_1	a_1	a_1, a_2	a_1, a_2
$\varnothing$	$\varnothing$	$\varnothing$	b_1	b_1	b_1	b_1, b_2
$\varnothing$	$\varnothing$	c_1	c_1	c_1, c_2	c_1, c_2	c_1, c_2
d_1	d_1	d_1	d_1	d_1	d_1	d_1

Figure 6: Input lists

4 A stack-based algorithm

We seek to design an algorithm inputting a query and a stream, reading the elements from the latter one by one and outputting as soon as possible all the matches of the stream so far against the query pattern. If no match is found after reading an element, the algorithm can be run again on the remaining stream with a data structure that keeps the intermediary results. It is an on-line algorithm, alternatively, it can be conceived as creating a stream of matches from a stream of elements and a query.

The rightmost branch as a stack of elements Since only the rightmost branch changes throughout insertions, we can keep it in memory instead of the whole XML tree. The way it changes also allows us to implement it as a stack, as Al-Khalifa et al. observed [1]: the bottom of the stack is the root element and the top is the end of the rightmost branch, which is always the last inputted element. An extension is thus implemented by simply pushing the new element on the stack; an expansion is achieved by popping elements until an extension becomes possible (two phases).

Attributes of stack elements Each element in the stack is associated with sets, called *attributes*, containing *partial matches*, that is, incomplete matches of the query. The implementation of extensions and expansions on a stack yields two kinds of attributes: the *inherited attributes* result from an extension (or the second phase of an expansion), and the *synthesised attributes* result from the first phase of an expansion. (We reuse the terminology of attributed grammars.) As an illustration, let us assume a query $\langle a \mid b, c\rangle$ and the current state of the XML tree shown in FIGURE 5a. A complete match $[1 \mapsto a_1, 2 \mapsto b_1, 3 \mapsto c_2]$ appears as $\mathbf{a_1b_1c_2}$ for the sake of brevity. Element d_1 has no attribute since its kind is not in the query. Element a_1 has no inherited attributes and it has some synthesised attributes which are partial matches made of itself and its contained elements which are not in the stack, that is, c_1 and b_1. Element c_2 has some inherited attributes made by combining itself with the synthesised attributes of its ancestors in the stack, that is, a_1 and d_1, and it has no synthesised attributes itself since it has no contained elements in the stack (yet).

Stack specification Formally, let $\textsc{Empty}$ denote any empty stack; $\textsc{Push}(x, S)$ the stack whose top element is x and remaining stack is S; $\textsc{Pop}(S)$ is the pair whose first component is the top element of stack S and the second is the remaining stack, assuming $S \neq \textsc{Empty}$ (this definition amounts to say that $\textsc{Pop}$ is the exact inverse function of $\textsc{Push}$, that is to say $S' = \textsc{Push}(x, S) \Leftrightarrow \textsc{Pop}(S') = (x, S)$). In order to model the rightmost branch of an XML tree, we need a stack whose elements are triples $(\varepsilon, \iota, \sigma)$, where ε is an element, ι denote inherited attributes of ε, and σ are the synthesised attributes of ε. For example, $\textsc{Push}((d_1, \varnothing, \varnothing), \textsc{Empty})$ denotes the stack after reading d_1 from the stream (namely, the first line of the table in FIGURE 7).

Insertion Let us assume a query $q = \langle \kappa_1 \mid \kappa_2, \ldots, \kappa_n\rangle$. Let $\textsc{Insert}(\varepsilon, S)$ be a pair whose first component is a (possibly empty) series of matches and the second component is the stack resulting from the insertion of element ε into the stack S.

The (complete) matches are obtained by reading element ε from the input stream and combining it with previous partial matches, while inserting it into the stack S. They are streamed out by increasing roots and, if matching with order, by increasing descendants, else the order is undefined (for matches with same root). For the sake of brevity, we do not give here the loop that calls INSERT, and instead focus directly on INSERT itself. (We use the algorithmic language defined by Cormen et al. in their textbook [6]. For additional clarity, we write the code in static single-assignment style.) For example, consider FIGURE 7. If S is the stack before adding c_2, S' the stack after adding c_2, and C the complete matches using c_2, then $(C, S') = \text{INSERT}(c_2, S)$, and $C = \{[1 \mapsto a_1, 2 \mapsto b_1, 3 \mapsto c_2]\}$. Moreover, FIGURE 7 shows S', except that, for the sake of clarity, we will continue to show the matches in the stack in bold *even if they are actually streamed out*. For example, in FIGURE 11a and FIGURE 11b, the matches $\mathbf{a_1b_1c_2}$ and $\mathbf{a_1b_2c_1}$ are shown inside the stack as inherited attributes.

	Attributes	
Element	Inherited	Synthesised
d_1	$\varnothing$	$\varnothing$
a_1	$\varnothing$	a_1c_1, a_1b_1
c_2	$\mathbf{a_1b_1c_2}, a_1c_2$	$\varnothing$

Figure 7: Stack after adding c_2

```
INSERT($\varepsilon, S$)
1  if $S$ = EMPTY                                            ▷ $\varepsilon$ is the first element.
2      then return $\varnothing$, PUSH(($\varepsilon, \varnothing, \varnothing$), $S$)                        ▷ No match yet.
3  ($\varepsilon_1, \iota_1, \sigma_1$), $S_1 \leftarrow$ POP($S$)               ▷ Let us destructure the top of stack $S$.
4  if $\varepsilon \sqsubset \varepsilon_1$                          ▷ Does the top $\varepsilon_1$ of the stack contain $\varepsilon$?
5      then $C, \mathcal{P} \leftarrow$ COMPLETE($\varepsilon, S$)            ▷ New partial and complete matches.
6              return $C$, PUSH(($\varepsilon, \mathcal{P}, \varnothing$), $S$)       ▷ Matches and rightmost extension.
7  ($\varepsilon_2, \iota_2, \sigma_2$), $S_2 \leftarrow$ POP($S_1$)            ▷ Destructure one more layer of stack $S$.
8  return INSERT($\varepsilon$, PUSH(($\varepsilon_2, \iota_2, \sigma_2 \cup$ FILTER($\varepsilon_1, \iota_1 \cup \sigma_1$), $S_2$))      ▷ Expansion.
```

(In the pseudo-code, we write pairs without parentheses when there is no ambiguity.) There are three cases when defining $\text{INSERT}(\varepsilon, S)$: (*i*) the stack S is empty because ε is the first element in the stream, at lines 1–2; (*ii*) the top element of S contains ε (which implies the extension of the rightmost branch), at lines 4–6; (*iii*) ε and the top of S are disjoint (which implies a rightmost expansion), at lines 7–8. If S is empty, the resulting stack only contains ε with neither inherited nor synthesised attributes, as shown in FIGURE 8. Otherwise, we pop up element ε_1 with its inherited and synthesised attributes, ι_1 and σ_1, and we get the remaining stack S_1,

	Attributes	
Element	Inherited	Synthesised
ε	$\varnothing$	$\varnothing$

Figure 8: ε is the first element

	Attributes	
Element	Inherited	Synthesised
	S	
ε	COMPLETE(ε, S)	$\emptyset$

Figure 9: Rightmost extension

at line 3. If the top of the stack, ε_1, contains ε (line 4), then it means that we have to extend the rightmost branch, which is achieved by pushing ε on S (line 6). The inherited attributes $\mathcal{P}$ of ε are computed by completing the partial matches in S and complete matches can also be found in the process, at line 5. The element ε has no synthesised attributes since it contains no elements yet. See the table in FIGURE 9.

Otherwise, the top of the stack does not contain ε, which means that we have to perform a rightmost expansion. This implies that the stack S_1 must be non-empty; in other words, S must contain at least two elements. Indeed, a rightmost expansion means that we grow another branch rooted on the rightmost branch, but not at its end (else it would be an extension), so the branch contains at least two nodes. Let us call ε_2 the top element of S_1, and ι_2 and σ_2 its inherited and synthesised attributes; S_2 is the remaining stack, at line 7. Please consider the table at FIGURE 10a.

The rightmost expansion is realised by means of an extension at the node where the new rightmost branch is rooted, i.e., when the biggest element containing the new element ε is on the top of the stack. Then the first step of the rightmost expansion consists in inserting recursively ε in a stack equal to S *without its top element* ε_1, until the condition for an extension is satisfied. The attributes of ε_1 are not lost: they are added (by a set union) to the synthesised attributes of the new top element ε_2 (see line 8). In terms of the XML tree, this amounts to move up the attributes of the rightmost leaf and cut this leaf, and so on until an extension is possible. Check FIGURE 10b.

As an example, consider the transition from FIGURE 5c to FIGURE 5b by inserting b_2. The stack before the insertion is shown in FIGURE 11a. FIGURE 11b shows the result of the rightmost expansion caused by the insertion of b_2. Note that we

	Attributes	
Element	Inherited	Synthesised
	S_2	
ε_2	ι_2	σ_2
ε_1	ι_1	σ_1

a. Stack before expansion

	Attributes	
Element	Inherited	Synthesised
	S_2	
ε_2	ι_2	$\sigma_2 \cup \iota_1 \cup \sigma_1$

b. Recursive expansion

Figure 10: Rightmost expansion

	Attributes	
Element	Inherited	Synthesised
d_1	$\varnothing$	$\varnothing$
a_1	$\varnothing$	a_1c_1, a_1b_1
c_2	$\mathbf{a_1b_1c_2}, a_1c_2$	$\varnothing$
a_2	$\varnothing$	$\varnothing$

a. Stack before adding b_2

	Attributes	
Element	Inherited	Synthesised
d_1	$\varnothing$	$\varnothing$
a_1	$\varnothing$	a_1c_1, a_1b_1
c_2	$\mathbf{a_1b_1c_2}, a_1c_2$	$\varnothing$
b_2	$\mathbf{a_1b_2c_1}, a_1b_2$	$\varnothing$

b. Stack after adding b_2.

Figure 11: The stack before and after inserting element b_2

assume in this figure that we search also for unordered matches of $\langle a \mid b, c\rangle$ thus $[1 \mapsto a_1, 2 \mapsto b_2, 3 \mapsto c_1]$, i.e., $\mathbf{a_1b_2c_1}$, is valid in spite of $c_1 < b_2$.

Completion In order to understand how the inherited attributes are computed (in case of a rightmost extension), we must now define precisely function COMPLETE.

```
COMPLETE($\varepsilon, S$)
1  if $S$ = EMPTY                                    ▷ If the stack is empty
2      then return $\varnothing, \varnothing$                       ▷ then there is no completion.
3  $(\varepsilon_1, \iota_1, \sigma_1), S_1 \leftarrow$ POP($S$)          ▷ Otherwise, destructure the top of the stack
                    ▷ and combine $\varepsilon$ with the synthesised attributes $\sigma_1$ of the top:
4  if $T(\varepsilon_1) = \kappa_1$                            ▷ if the top $\varepsilon_1$ matches the query root,
5      then $C_1, \mathcal{P}_1 \leftarrow$ COMBINE($\varepsilon, \sigma_1 \cup \{[1 \mapsto \varepsilon_1]\}$)     ▷ it may combine with $\varepsilon$
6      else $C_1, \mathcal{P}_1 \leftarrow$ COMBINE($\varepsilon, \sigma_1$)                    ▷ or not.
7  $C_2, \mathcal{P}_2 \leftarrow$ COMPLETE($\varepsilon, S_1$)         ▷ Complete the remaining partial matches.
8  return $C_1 \cup C_2, \mathcal{P}_1 \cup \mathcal{P}_2$                ▷ New complete and partial matches.
```

If stack S is empty, then let us return no new matches (lines 1–2). Otherwise, let us pop element ε_1, whose synthesised attributes are σ_1, and let S_1 be the remaining stack, at line 3. The next step is to try to combine ε with the partial matches in σ_1 and get new matches C_1 and new partial matches $\mathcal{P}_1$ (line 6) but we must not forget a possible new partial match involving only ε and ε_1. Thus we first check whether the top of the stack, ε, matches the query root, at line 4. If so, we also must try to combine ε_1 and ε, at line 5. For example, consider the partial match a_1c_2 in FIGURE 7. Next, we complete the remaining stack S_1 with ε and obtain new matches C_2 and new partial matches $\mathcal{P}_2$ (line 7). We merge these new matches and we merge separately the partial matches (line 8).

Combination Let us define now COMBINE, which is the function that tries to complete a set of partial matches with a given element. Since we deal here with partial matches, we must be more precise here. Let us note $\mathcal{D}$ the function that returns the kind indexes of a given match (the *domain* of the match). For example, if

$q = \langle a \mid b, c\rangle$ and $\mu_q = [1 \mapsto a_1, 2 \mapsto b_1, 3 \mapsto c_2]$, then $\mathcal{D}(\mu_q) = \{1,2,3\}$. Let us note $\overline{\mathcal{D}}(\mu_q)$ the complementary set, e.g., if $\mu_q = [1 \mapsto a_1, 3 \mapsto c_2]$, then $\mathcal{D}(\mu_q) = \{1,3\}$ and $\overline{\mathcal{D}}(\mu_q) = \{2\}$. Then the match μ_q is complete if and only if $\overline{\mathcal{D}}(\mu_q) = \varnothing$. Let us note $\mu_q \oplus i \mapsto \varepsilon$ the extension of a match μ_q with the binding $i \mapsto \varepsilon$. The operator $\oplus$ can be formally defined, for all $j \in \mathcal{D}(\mu_q) \cup \{i\}$, as

$$(\mu_q \oplus i \mapsto \varepsilon)(j) \triangleq \begin{cases} \varepsilon & \text{if } i = j \\ \mu_q(j) & \text{otherwise} \end{cases}$$

Moreover, let us assume we have a function Choose that takes a set of matches and returns one of them paired with the complementary matches. (This is a way to defer to the implementation the actual iteration order.) The following definition of Combine returns both ordered and unordered matches. For ordered matches only, see further.

```
Combine($\varepsilon, \sigma$)
1  if $\sigma = \varnothing$                                  ▷ If there are no partial matches
2     then return $\varnothing, \varnothing$                  ▷ then return no new matches.
3  $\mu_q, \sigma' \leftarrow$ Choose($\sigma$)     ▷ Pick a partial match $\mu_q$; the remaining ones are $\sigma'$.
4  $C, \mathcal{P} \leftarrow$ Combine($\varepsilon, \sigma'$)   ▷ Combine $\varepsilon$ with the remaining partial matches.
5  if $\exists i \in \overline{\mathcal{D}}(\mu_q).T(\varepsilon) = \kappa_i$   ▷ If $\varepsilon$ completes $\mu_q$ at index $i$, i.e., $\varepsilon$ matches $\kappa_i$,
6     then if $|\overline{\mathcal{D}}(\mu_q)| = 1$           ▷ then if $i$ was the sole unused index
7             then return $C \cup \{\mu_q \oplus i \mapsto \varepsilon\}, \mathcal{P}$   ▷ we make a complete match,
8           return $C, \mathcal{P} \cup \{\mu_q \oplus i \mapsto \varepsilon\}$           ▷ else it is a partial match.
9  return $C, \mathcal{P}$          ▷ If $\varepsilon$ does not extend $\mu_q$, ignore the partial match $\mu_q$.
```

If the set of partial matches σ is empty (line 1), then let us return an empty set of new matches (line 2). Otherwise, let us arbitrarily choose any partial match μ_q in σ and name the remaining partial matches σ' (line 3). Next, let us recursively combine ε with the remaining partial matches (line 4). If μ_q can be extended by ε, that is to say, at least one of its index i is unused and ε matches κ_i (line 5), then a new partial or complete match can be made. If index i was the last unused in μ_q (line 6), then $\mu_q \oplus i \mapsto \varepsilon$ is a complete match (line 7), otherwise it is partial (line 8). If ε can not extend μ_q (either because it matches no node of the query or because the sole matching nodes in the query are already matched by nodes of the XML tree read so far), we just ignore it (line 9).

5 A table-based algorithm

5.1 Basic definitions

Tables. The *tables* we shall consider further have two entries, the *vertical entries* and the *horizontal entries*. For the sake of brevity, we shall speak of *entries* instead of vertical entries. For example, in the table at figure 12, x, y and z are the (vertical)

	f	g	h
x	1	2	3
y	4	5	6
z	7	8	9

Figure 12: A table

entries and the a, b and c are the horizontal entries. The *rows* are the horizontal sections of the table containing exactly one horizontal entry. For example $(x, 1, 2, 3)$ is a row (more precisely, the first row). Dually, the *columns* are sections of the table which contain exactly one vertical entry. For example $(g, 2, 5, 8)$ is a column (more precisely, the second column). A *table index*, or *index* for short, is a positive, non-zero, integer which characterises an (horizontal) entry. Entries are ordered by ascending indexes, starting with 1. This way we can speak of "first entry", "second entry" etc. A *cell* is the intersection of a row and a column. The first cell of a given row is the intersection of this row with the first column. Dually, the first cell of a given column is the intersection of this column with the first row. We shall denote the contents of a cell using a functional notation, e.g. $f(x)$ is the cell at the intersection of the column f and the raw whose entry is x, in other words: $f(x) = 1$; another example is $h(y) = 6$ etc.

Elements and unification. Let us extend the concept of element to cope with *undefined element*, noted Ω, which have no kind. Now, a *ground element* is an element which is not undefined (and thus is kinded). The *unification* of two elements ε_1 and ε_2, noted $\varepsilon_1 \otimes \varepsilon_2$, is an element defined by cases as follows:

$$\Omega \otimes \Omega = \Omega$$
$$\Omega \otimes \varepsilon = \varepsilon$$
$$\varepsilon \otimes \Omega = \varepsilon$$

and if $\varepsilon_1 \neq \Omega$ and $\varepsilon_2 \neq \Omega$, then $\varepsilon_1 \otimes \varepsilon_2$ is undefined.

***n*-tuples and unification.** Let us extend the concept of n-tuple of elements to cope with n-tuples of ground or undefined elements. We also constrain the ith component of the tuple to be of kind κ_i — this ensures the unicity of a solution. If the context is clear, an *n-tuple* will refer to an n-tuple of elements. A *ground tuple* is a tuple made of ground elements. A *solution tuple* is a tuple of elements which satisfy the condition stated in section 2, i.e., any two elements are disjoint (non-overlapping). In our example, (Ω, b_1, c_2) is a triple, (a_1, b_1, c_2) is a ground triple and (a_4, b_1, c_2) is a solution triple. Moreover, we can extend the unification on elements to n-tuples as follows. Let $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ and $(\varepsilon'_1, \varepsilon'_2, \varepsilon'_3)$ be two triples of elements. Then, by definition

$$(\varepsilon_1, \varepsilon_2, \varepsilon_3) \otimes (\varepsilon'_1, \varepsilon'_2, \varepsilon'_3) = (\varepsilon_1 \otimes \varepsilon'_1, \varepsilon_2 \otimes \varepsilon'_2, \varepsilon_3 \otimes \varepsilon'_3)$$

	$\overline{\tau}$				$\overline{\omega}$			
	$\overline{\tau}\|_1$	$\overline{\tau}\|_2$	$\dots$	$\overline{\tau}\|_n$	$\overline{\omega}\|_1$	$\overline{\omega}\|_2$	$\dots$	$\overline{\omega}\|_n$
1	___	___	$\dots$	___	___	___	$\dots$	___
2	___	___	$\dots$	___	___	___	$\dots$	___
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 1: General shape of the τ-tables

	$\overline{\tau}$			$\overline{\omega}$		
	$\overline{\tau}\|_a$	$\overline{\tau}\|_b$	$\overline{\tau}\|_c$	$\overline{\omega}\|_a$	$\overline{\omega}\|_b$	$\overline{\omega}\|_c$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
7	Ω	Ω	c_2	$\{7,8,10\}$	$\{7,9\}$	$\emptyset$
8	Ω	b_1	c_2			
9	a_2	Ω	c_2			
10	Ω	b_4	c_2			
11	a_2	b_4	c_2			
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 2: Example of τ-table

5.2 The τ-table

Let us define a global table, called the *τ-table*. The entries are indexes in increasing order (first is 1). The first column, named $\overline{\tau}$, contains n-tuples of elements. The second column, named $\overline{\omega}$ contains n-tuples whose components are sets of entries (indexes) in the same table and called *ω-sets*. These tuples are called *ω-tuples* when n is implied. For the sake of clarity, the first column, $\overline{\tau}$, can be divided into n sub-columns, one for each kind of element, noted $\overline{\tau}|_j$ for the jth sub-column of kind κ_j, and the second column can be presented following the same schema, with sub-columns named $\overline{\omega}|_j$. The shape of the τ-table is shown in table 1. The cell at the intersection of the raw whose entry is i and the sub-column is $\overline{\omega}|_j$, noted $\overline{\omega}|_j(i)$, is interpreted as a set of entries in the τ-table such that the jth component of the n-tuple is Ω. Formally[1]:

$$\forall i \in [\![1,p]\!].\forall j \in [\![1,n]\!].\forall k \in \overline{\omega}|_j(i).\overline{\tau}|_j(k) = \Omega$$

For example, consider the table 2, were the kinds are a, b and c. For more simplicity, assuming that the kinds are ordered $a < b < c$, we can note $\overline{\omega}|_a$ instead of $\overline{\omega}|_1$, $\overline{\omega}|_b$ instead of $\overline{\omega}|_2$ and $\overline{\omega}|_c$ instead of $\overline{\omega}|_3$. Note that

[1]The interval on integers from x to y included is noted $[\![x,y]\!]$ and "For all x then (do) $P(x)$" is noted $\forall x.P(x)$.

- not all rows are separated by horizontal lines, which gives the feeling that some successive rows have a common meaning (e.g. entries from 7 to 11 appear as gathered because of the line before row 7 and after row 11);
- many cells which should contain an ω-set are empty because the corresponding ω-set is undefined (see sub-columns $\overline{\omega}|_a$, $\overline{\omega}|_b$ and $\overline{\omega}|_c$).

The ω-set $\overline{\omega}|_a(7) = \{7, 8, 10\}$ refers to the entries 7, 8 and 10. These entries have Ω in the sub-column $\tau|_a$: the full triples are respectively (Ω, Ω, c_2), (Ω, b_1, c_2) and (Ω, b_4, c_2). Similarly, the ω-set $\overline{\omega}|_b(7) = \{7, 9\}$ refers to *some* entries whose triples have a b component equal to Ω: (Ω, Ω, c_2) and (a_2, Ω, c_2).

5.3 The attribute table

In order to tackle our problem, we need a second kind of data structure. Let us map every node in the XML tree to a record, called *node attributes*, made of a table index and an n-tuple of ω-sets. This table indexes and the elements of these ω-sets are entries to the τ-table we defined in section 5.2. For the sake of clarity, we shall gather these node attributes into a table, called the *attribute table*, whose (horizontal) entries are the nodes (modelling the elements), the first vertical entry is the table index and the second vertical entry is the n-tuple of ω-sets, or ω*-tuple* when n is implied. For the sake of clarity, the column of ω-tuples is divided in n sub-columns, as in the τ-table, and is named $\overline{\omega}$, just as in the τ-table: the possible ambiguity is removed by looking at the argument: $\overline{\omega}(\varepsilon)$ refers to the attribute table (ε is the notation for an element) whereas $\overline{\omega}(i)$ refers to the τ-table (i is the notation for an index).

Keep in mind that it is important to access each attribute in constant time from each node, that is why it is better to implement this table as attributes of the nodes in the XML tree. The table excerpt 3 shows the attributes of nodes a_2, b_4 and c_2. The first column, named I, contains one index of the τ-table. For example, $I(a_2) = 4$ and the ω-set $\overline{\omega}|_a(c_2) = \{7, 8, 10\}$ contains entries (here, indexes) to the τ-table 2.

		$\overline{\omega}$		
	I	$\overline{\omega}\|_a$	$\overline{\omega}\|_b$	$\overline{\omega}\|_c$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_2	4	$\varnothing$	$\{4\}$	$\{4\}$
b_4	5	$\{5\}$	$\varnothing$	$\{5, 6\}$
c_2	7	$\{7, 8, 10\}$	$\{7, 9, 12\}$	$\{12, 13, 14\}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 3: Example of node attributes

5.4 Algorithm

The algorithm for solving the problem is as follows. For each element ε of kind κ inputted from the XML stream do the following in turn.

1. Create and initialise a row in the τ-table:

   (a) if the last row has index m (if there is no row, then take $m = 0$), then create an empty row in the τ-table at index $m + 1$;

   (b) initialise the corresponding tuple with ε as component of kind κ and the remaining components of the triple with Ω. Formally, let $j \in [\![1, n]\!]$ such that $\overline{\tau}|_j = \kappa$, then[2]

   $$\overline{\tau}|_j(m + 1) \leftarrow \varepsilon \text{ and } \forall k \neq j. \overline{\tau}|_k(m + 1) \leftarrow \Omega$$

   (c) set the ω-set corresponding to kind κ to $\varnothing$ and the others to $\{m + 1\}$.

   $$\overline{\omega}|_j(m + 1) \leftarrow \varnothing \text{ and } \forall k \neq j. \overline{\omega}|_k(m + 1) \leftarrow \{m + 1\}$$

2. Create and initialise a row in the attribute table:

   (a) create an empty row in the attribute table whose entry is element ε;

   (b) initialise the first cell $I(\varepsilon)$ with the index resulting of the addition of the same element in the τ-table: $I(\varepsilon) \leftarrow m + 1$;

   (c) initialise the ω-sets of ε with $\varnothing$: $\forall j \in [\![1, n]\!]. \overline{\omega}|_j(\varepsilon) \leftarrow \varnothing$.

3. Insert the element in the XML tree (this modifies only the rightmost branch because of the stream order, see FIGURE 5).

4. If the insertion results in the creation of a new branch (instead of growing the previous branch), perform a *rightmost expansion* (see FIGURE 5 and further formal definition);

5. Perform the expansion of the newly added tuple (or *τ-expansion*) in the τ-table (see further definition): this results in a set of new tuples. If one of these tuples is a solution, then stop else input another element from the stream.

Let us now define the operations we just mentioned, following the order in which they are applied.

[2]The assignment of y to x is noted $x \leftarrow y$.

5.5 Insertion in the τ-table

Let us illustrate first the step (1) of the algorithm on an example. Consider the τ-table just before the insertion of node b_4:

	$\overline{\tau}\|_a$	$\overline{\tau}\|_b$	$\overline{\tau}\|_c$	$\overline{\omega}\|_a$	$\overline{\omega}\|_b$	$\overline{\omega}\|_c$
1	Ω	Ω	c_1	$\{1\}$	$\{1\}$	$\emptyset$
2	a_1	Ω	Ω	$\emptyset$	$\{2\}$	$\{2\}$
3	Ω	b_1	Ω	$\{3\}$	$\emptyset$	$\{3\}$
4	a_2	Ω	Ω	$\emptyset$	$\{4\}$	$\{4\}$

The following rows illustrate the steps (1a), (1b) and (1c):

5						

5	Ω	b_4	Ω			

5	Ω	b_4	Ω	$\{5\}$	$\emptyset$	$\{5\}$

5.6 Insertion in the attribute table

Let us first illustrate on an example the step (2) of the algorithm. Here is the attribute table before addition of node b_4:

	I	$\overline{\omega}\|_a$	$\overline{\omega}\|_b$	$\overline{\omega}\|_c$
c_1	1	$\emptyset$	$\emptyset$	$\emptyset$
a_1	2	$\emptyset$	$\emptyset$	$\emptyset$
b_1	3	$\emptyset$	$\emptyset$	$\emptyset$
a_2	4	$\emptyset$	$\emptyset$	$\emptyset$

Steps (2a), (2b) and (2c) are straightforward:

b_4				

b_4	5			

b_4	5	$\emptyset$	$\emptyset$	$\emptyset$

The cell $I(b_4)$ is set to 5 because 5 is the entry **in the τ-table** corresponding to the insertion of b_4 (see corresponding τ-table at section 5.5).

5.7 Insertion in the XML tree

This operation corresponds to the step (3) of the algorithm. Let us specify it with the help of a rewrite system [10]. Let us note EMPTY the empty XML tree and NODE(ε, f) the non-empty tree whose root is the element ε and sub-trees are the forest l. A forest is a list of trees, *where the first tree is the rightmost tree for the ordering of roots*. The empty list is noted [ ] and the non-empty list whose head is x and tail l is noted $[x \mid l]$, as in Prolog. Because the children of a node constitute a forest, i.e., a list of subtrees, we need a function ADD such that

- $\text{ADD}(\varepsilon, t)$ is the XML tree resulting from the insertion of element ε into the XML tree t;
- $\text{ADD}(\varepsilon, f)$ is the XML forest resulting from the insertion of element ε into the XML forest f.

The corresponding *ordered* rewrite system is

$$
\begin{aligned}
\text{ADD}(\varepsilon, \text{NODE}(r, f)) &\xrightarrow{1} \text{NODE}(r, \text{ADD}(\varepsilon, f))\\
\text{ADD}(\varepsilon, [\text{NODE}(r_1, f_1) \mid f_2]) &\xrightarrow{2} [\text{NODE}(r_1, \text{ADD}(\varepsilon, f_1)) \mid f_2] \qquad \text{if } \varepsilon \sqsubset r_1\\
\text{ADD}(\varepsilon, x) &\xrightarrow{3} [\text{NODE}(\varepsilon, [\,]) \mid x]
\end{aligned}
$$

Rule ($\xrightarrow{1}$) adds an element ε to the current, non-empty, XML tree $\text{NODE}(r, f)$. Rule ($\xrightarrow{2}$) handles the recursive descent along the rightmost branch, as long as the element to add is contained in the rightmost tree. In rule ($\xrightarrow{3}$), variable x can match EMPTY, when the first element is added, or an empty forest, when performing an extension, or a non-empty forest when making an expansion. This system is correct if and only if the stream was generated from a single XML tree and if the elements are streamed out in increasing order. As shown in FIGURE 5, the addition of a node either extends linearly the current rightmost branch (*extension*) or creates a new branch (*expansion*). Let us consider the latter case. For example, figure 13 shows the XML trees before and after the addition of node b_4. The dotted edges mark the branch of the previous rightmost branch that is no more part of the new rightmost branch (in thick solid edges). The node in the rightmost branch from which a new branch is attached is called the *knot*. Here, the knot is a_1.

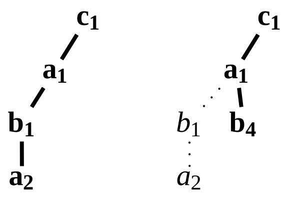


Figure 13: Rightmost branch before and after inserting b_4

5.8 Rightmost expansion

Let us define now the operation of *rightmost expansion*, or simply *expansion*, since we only deal here with the rightmost branch. The expansion of the rightmost branch occurs when a diverging branch is created by the addition of a node. This operation consists in the following ordered steps.

For each node ε from the end of the previous rightmost branch to the knot *excluded* (i.e., graphically, following the dotted edges, bottom-up, in the XML tree),

1. add to the ω-sets of the node the corresponding ω-sets in the τ-table found at the entry given by the first cell (in the attribute table):

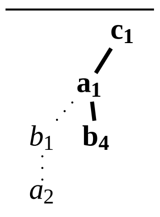
$$\forall j \in [\![1, n]\!].\overline{\omega}|_j(\varepsilon) \leftarrow \overline{\omega}|_j(\varepsilon) \cup \overline{\omega}|_j(I(\varepsilon))$$

2. add all the newly updated ω-sets of the node to their corresponding ω-sets in the parent node; if ε' is the parent of ε then

$$\forall j \in [\![1, n]\!].\overline{\omega}|_j(\varepsilon') \leftarrow \overline{\omega}|_j(\varepsilon') \cup \overline{\omega}|_j(\varepsilon)$$

For example, here is the XML tree, the attribute table and the τ-table just after the insertion of node b_4:

XML tree



Attributes

	I	$\overline{\omega}\vert_a$	$\overline{\omega}\vert_b$	$\overline{\omega}\vert_c$
c_1	1	$\varnothing$	$\varnothing$	$\varnothing$
a_1	2	$\varnothing$	$\varnothing$	$\varnothing$
b_1	3	$\varnothing$	$\varnothing$	$\varnothing$
a_2	4	$\varnothing$	$\varnothing$	$\varnothing$
b_4	5	$\varnothing$	$\varnothing$	$\varnothing$

τ-table

	$\overline{\tau}\vert_a$	$\overline{\tau}\vert_b$	$\overline{\tau}\vert_c$	$\overline{\omega}\vert_a$	$\overline{\omega}\vert_b$	$\overline{\omega}\vert_c$
1	Ω	Ω	c_1	$\{1\}$	$\{1\}$	$\varnothing$
2	a_1	Ω	Ω	$\varnothing$	$\{2\}$	$\{2\}$
3	Ω	b_1	Ω	$\{3\}$	$\varnothing$	$\{3\}$
4	a_2	Ω	Ω	$\varnothing$	$\{4\}$	$\{4\}$
5	Ω	b_4	Ω	$\{5\}$	$\varnothing$	$\{5\}$

We find that a new branch is created, whose knot is a_1. Therefore, we must proceed to a rightmost expansion. The branch which is not part anymore of the rightmost branch is the path (a_1, b_1, a_2). We consider in turn a_2 and b_1 (i.e., bottom-up), but not a_1 because it is the knot (and, as such, still belongs to the rightmost branch). First, a_2 is not the knot and its kind is a, so we apply step (1):

$$\begin{cases} \overline{\omega}|_a(a_2) \leftarrow \overline{\omega}|_a(a_2) \cup \overline{\omega}|_a(I(a_2)) \\ \overline{\omega}|_b(a_2) \leftarrow \overline{\omega}|_b(a_2) \cup \overline{\omega}|_b(I(a_2)) \\ \overline{\omega}|_c(a_2) \leftarrow \overline{\omega}|_c(a_2) \cup \overline{\omega}|_c(I(a_2)) \end{cases} \Longleftrightarrow \begin{cases} \overline{\omega}|_a(a_2) \leftarrow \varnothing \\ \overline{\omega}|_b(a_2) \leftarrow \{4\} \\ \overline{\omega}|_c(a_2) \leftarrow \{4\} \end{cases}$$

Next, we apply step (2), i.e., we add (by set union) each ω-set of a_2 to its corresponding ω-set of the parent node (in the old rightmost branch), i.e., b_1:

$$\begin{cases} \overline{\omega}|_a(b_1) \leftarrow \overline{\omega}|_a(b_1) \cup \overline{\omega}|_a(a_2) \\ \overline{\omega}|_b(b_1) \leftarrow \overline{\omega}|_b(b_1) \cup \overline{\omega}|_b(a_2) \\ \overline{\omega}|_c(b_1) \leftarrow \overline{\omega}|_c(b_1) \cup \overline{\omega}|_c(a_2) \end{cases} \Longleftrightarrow \begin{cases} \overline{\omega}|_a(b_1) \leftarrow \varnothing \\ \overline{\omega}|_b(b_1) \leftarrow \{4\} \\ \overline{\omega}|_b(b_1) \leftarrow \{4\} \end{cases}$$

Thus the attribute table is now (changes in bold):

	I	$\overline{\omega}\rvert_a$	$\overline{\omega}\rvert_b$	$\overline{\omega}\rvert_c$
c_1	1	$\emptyset$	$\emptyset$	$\emptyset$
a_1	2	$\emptyset$	$\emptyset$	$\emptyset$
b_1	3	$\emptyset$	$\{\mathbf{4}\}$	$\{\mathbf{4}\}$
a_2	4	$\emptyset$	$\{\mathbf{4}\}$	$\{\mathbf{4}\}$
b_4	5	$\emptyset$	$\emptyset$	$\emptyset$

Now, we consider node b_1. It is not the knot, so we proceed with expansion step (1): we add the ω-sets of entry 3 **in the τ-table** to the ω-sets of b_1:

$$\begin{cases} \overline{\omega}|_a(b_1) \leftarrow \overline{\omega}|_a(b_1) \cup \overline{\omega}|_a(I(b_1)) \\ \overline{\omega}|_b(b_1) \leftarrow \overline{\omega}|_b(b_1) \cup \overline{\omega}|_b(I(b_1)) \\ \overline{\omega}|_c(b_1) \leftarrow \overline{\omega}|_c(b_1) \cup \overline{\omega}|_c(I(b_1)) \end{cases} \Longleftrightarrow \begin{cases} \overline{\omega}|_a(b_1) \leftarrow \emptyset \cup \{3\} \\ \overline{\omega}|_b(b_1) \leftarrow \{4\} \cup \emptyset \\ \overline{\omega}|_c(b_1) \leftarrow \{4\} \cup \{3\} \end{cases}$$

The attribute table is now (changes in bold)

	I	$\overline{\omega}\rvert_a$	$\overline{\omega}\rvert_b$	$\overline{\omega}\rvert_c$
c_1	1	$\emptyset$	$\emptyset$	$\emptyset$
a_1	2	$\emptyset$	$\emptyset$	$\emptyset$
b_1	3	$\{\mathbf{3}\}$	$\{4\}$	$\{4, \mathbf{3}\}$
a_2	4	$\emptyset$	$\{4\}$	$\{4\}$
b_4	5	$\emptyset$	$\emptyset$	$\emptyset$

Next, we apply expansion step (2), i.e., we add the new ω-sets of b_1 to the corresponding sets of node a_1:

	I	$\overline{\omega}\rvert_a$	$\overline{\omega}\rvert_b$	$\overline{\omega}\rvert_c$
c_1	1	$\emptyset$	$\emptyset$	$\emptyset$
a_1	2	$\{\mathbf{3}\}$	$\{\mathbf{4}\}$	$\{\mathbf{4}, \mathbf{3}\}$
b_1	3	$\{3\}$	$\{4\}$	$\{4, 3\}$
a_2	4	$\emptyset$	$\{4\}$	$\{4\}$
b_4	5	$\emptyset$	$\emptyset$	$\emptyset$

5.9 τ-expansion

Relative ω-unification. Before we give the definition of the τ-expansion used in the step (5) of the algorithm, we need to define another kind of unification, called *relative ω-unification*, on ω-sets (as found in the τ-table). Let ω_1 and ω_2 be two ω-sets and i an index. Their unification relatively to i, noted $\omega_1 \otimes_i \omega_2$, is another ω-set such that

$$\begin{cases} \emptyset \otimes_i \omega = \emptyset \\ \omega \otimes_i \emptyset = \emptyset \\ \omega_1 \otimes_i \omega_2 = \{i\} \quad \text{if } \omega_1 \neq \emptyset \text{ and } \omega_2 \neq \emptyset \end{cases}$$

We extend the ω-unification to tuples of ω-sets as follows. Let $\overline{\omega}_1$ and $\overline{\omega}_2$ be two n-tuples or ω-sets over the same kinds, then their unification relatively to an index i is another n-tuple of ω-sets such that

$$\overline{\omega}_1 \otimes_i \overline{\omega}_2 = (\overline{\omega}_1|_1 \otimes_i \overline{\omega}_2|_1, \overline{\omega}_1|_2 \otimes_i \overline{\omega}_2|_2, \ldots, \overline{\omega}_1|_n \otimes_i \overline{\omega}_2|_n)$$

Union of n-tuples of ω-sets. We also need to extend the set union (on ω-sets) to n-tuples of ω-sets. Let $\overline{\omega}_1$ and $\overline{\omega}_2$ be two n-tuples of ω-sets, then the union of $\overline{\omega}_1$ and $\overline{\omega}_2$, noted $\overline{\omega}_1 \sqcup \overline{\omega}_2$, is an n-tuple of ω-sets such that each component is the union the corresponding components in $\overline{\omega}_1$ and $\overline{\omega}_2$. Formally: $\overline{\omega}_1 \sqcup \overline{\omega}_2 = (\overline{\omega}_1|_1 \cup \overline{\omega}_2|_1, \overline{\omega}_1|_2 \cup \overline{\omega}_2|_2, \ldots, \overline{\omega}_1|_n \cup \overline{\omega}_2|_n)$.

τ-expansion. The the last entry in the τ-table is $I(\varepsilon)$ since we assume here that the last inserted element is ε. Recall that the totally ordered set of kinds is $\mathcal{K} = \{\kappa_1, \kappa_2, \ldots, \kappa_n\}$, and that $K(\varepsilon)$ is the kind of the element ε. The τ-expansion consists in the following steps.

1. $k \leftarrow I(\varepsilon)$
2. For each parent ε' of the newly added node ε (at the end of the rightmost branch) until the root *included*, do
   (a) Unify the tuple of the newly added node, $I(\varepsilon)$, with the tuples of its parent $I(\varepsilon')$ as given by the ω-set of the same kind as ε in the τ-table; then unify the ω-sets of $I(\varepsilon)$ with the ω-sets of the tuples in question, relatively to new indexes k (one k for each element of the ω-set of kind $K(\varepsilon)$ of entry $I(\varepsilon')$). Formally, for all $i \in \overline{\omega}|_{K(\varepsilon)}(I(\varepsilon'))$,

   $$\begin{aligned} k &\leftarrow k + 1 \\ \overline{\tau}(k) &\leftarrow \overline{\tau}(I(\varepsilon)) \otimes \overline{\tau}(i) \\ \overline{\omega}(k) &\leftarrow \overline{\omega}(I(\varepsilon)) \otimes_k \overline{\omega}(i) \end{aligned}$$

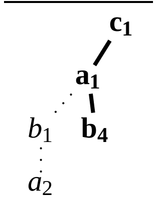
   If no component of $\overline{\tau}(k)$ is Ω, then $\overline{\tau}(k)$ is a solution, thus stop.

   (b) Merge all the ω-tuples of entries $I(\varepsilon) + 1$ to k in the τ-table and merge them to the ω-tuple of entry $I(\varepsilon)$:

   $$\overline{\omega}(I(\varepsilon)) \leftarrow \bigsqcup_{p=I(\varepsilon)}^{k} \overline{\omega}(p)$$

Let us recall the attribute table after the rightmost expansion due to the insertion of node b_4 and the τ-table at the same moment:

XML tree



Node attributes

	I	$\overline{\omega}\vert_a$	$\overline{\omega}\vert_b$	$\overline{\omega}\vert_c$
c_1	1	$\varnothing$	$\varnothing$	$\varnothing$
a_1	2	$\{3\}$	$\{4\}$	$\{4,3\}$
b_1	3	$\{3\}$	$\{4\}$	$\{4,3\}$
a_2	4	$\varnothing$	$\{4\}$	$\{4\}$
b_4	5	$\varnothing$	$\varnothing$	$\varnothing$

τ-table

	$\overline{\tau}\vert_a$	$\overline{\tau}\vert_b$	$\overline{\tau}\vert_c$	$\overline{\omega}\vert_a$	$\overline{\omega}\vert_b$	$\overline{\omega}\vert_c$
1	Ω	Ω	c_1	$\{1\}$	$\{1\}$	$\varnothing$
2	a_1	Ω	Ω	$\varnothing$	$\{2\}$	$\{2\}$
3	Ω	b_1	Ω	$\{3\}$	$\varnothing$	$\{3\}$
4	a_2	Ω	Ω	$\varnothing$	$\{4\}$	$\{4\}$
5	Ω	b_4	Ω	$\{5\}$	$\varnothing$	$\{5\}$

In our example, the τ-expansion starts by considering the parent node of $\varepsilon = b_4$ in the XML tree, which is a_1. We hence have $K(\varepsilon) = K(b_4) = b$, $I(a_1) = 2$, $I(\varepsilon) = I(b_4) = 5$ and $\overline{\omega}|_{K(\varepsilon)}(I(\varepsilon')) = \overline{\omega}|_b(2) = \{2\}$. Step (2a) consists in doing, for $i = 2$

$$\left\{\begin{array}{l} \quad k \leftarrow 6 \\ \overline{\tau}(6) \leftarrow \overline{\tau}(5) \otimes \overline{\tau}(2) \\ \overline{\omega}(6) \leftarrow \overline{\omega}(5) \otimes_6 \overline{\omega}(2) \end{array}\right. \Longleftrightarrow \left\{\begin{array}{l} \quad k \leftarrow 6 \\ \overline{\tau}(6) \leftarrow (\Omega, b_4, \Omega) \otimes (a_1, \Omega, \Omega) \\ \overline{\omega}(6) \leftarrow (\{5\}, \varnothing, \{5\}) \otimes_6 (\varnothing, \{2\}, \{2\}) \end{array}\right.$$

$$\Longleftrightarrow \left\{\begin{array}{l} \quad k \leftarrow 6 \\ \overline{\tau}(6) \leftarrow (a_1, b_4, \Omega) \\ \overline{\omega}(6) \leftarrow (\varnothing, \varnothing, \{6\}) \end{array}\right.$$

We hence have $\overline{\tau}|_c(6) = \Omega$, so $\overline{\tau}(6)$ is not a solution. The τ-table now is

	$\tau\vert_a$	$\tau\vert_b$	$\tau\vert_c$	$\omega\vert_a$	$\omega\vert_b$	$\omega\vert_c$
1	Ω	Ω	c_1	$\{1\}$	$\{1\}$	$\varnothing$
2	a_1	Ω	Ω	$\varnothing$	$\{2\}$	$\{2\}$
3	Ω	b_1	Ω	$\{3\}$	$\varnothing$	$\{3\}$
4	a_2	Ω	Ω	$\varnothing$	$\{4\}$	$\{4\}$
5	Ω	b_4	Ω	$\{5\}$	$\varnothing$	$\{5\}$
6	$\mathbf{a_1}$	$\mathbf{b_4}$	$\boldsymbol{\Omega}$	$\boldsymbol{\varnothing}$	$\boldsymbol{\varnothing}$	$\{\mathbf{6}\}$

Note that we do not draw a line between the rows 5 and 6. The reason is that the new row 6 has been produced by the τ-expansion of row 5. Now, let us apply step (2b) and merge upward the ω-sets of the new entries (here entry 6) with the ω-sets of $I(\varepsilon)$:

$$\begin{aligned} \overline{\omega}(5) \leftarrow \bigsqcup_{p=5}^{6} \overline{\omega}(p) &\Longleftrightarrow \overline{\omega}(5) \leftarrow \overline{\omega}(5) \sqcup \overline{\omega}(6) \\ &\Longleftrightarrow \overline{\omega}(5) \leftarrow (\{5\}, \varnothing, \{5\}) \sqcup (\varnothing, \varnothing, \{6\}) \\ &\Longleftrightarrow \overline{\omega}(5) \leftarrow (\{5\}, \varnothing, \{5, 6\}) \end{aligned}$$

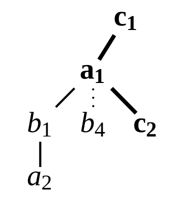
Finally, the τ-table after insertion of b_4 is (changes in bold)

	$\tau\|_a$	$\tau\|_b$	$\tau\|_c$	$\omega\|_a$	$\omega\|_b$	$\omega\|_c$
1	Ω	Ω	c_1	$\{1\}$	$\{1\}$	$\varnothing$
2	a_1	Ω	Ω	$\varnothing$	$\{2\}$	$\{2\}$
3	Ω	b_1	Ω	$\{3\}$	$\varnothing$	$\{3\}$
4	a_2	Ω	Ω	$\varnothing$	$\{4\}$	$\{4\}$
5	Ω	b_4	Ω	$\{5\}$	$\varnothing$	$\{5, \mathbf{6}\}$
6	a_1	b_4	Ω	$\varnothing$	$\varnothing$	$\{6\}$

5.10 Completing the example

Let us complete our continued example as suspended in section 5.9 and input elements in the remaining stream [c_2, a_3, a_4, b_2, c_3, b_3] until a solution triple appears. Let us insert first c_2 in the XML tree. The situation is

XML tree



Node attributes

	I	$\overline{\omega}\|_a$	$\overline{\omega}\|_b$	$\overline{\omega}\|_c$
c_1	1	$\varnothing$	$\varnothing$	$\varnothing$
a_1	2	$\{3\}$	$\{4\}$	$\{4, 3\}$
b_1	3	$\{3\}$	$\{4\}$	$\{4, 3\}$
a_2	4	$\varnothing$	$\{4\}$	$\{4\}$
b_4	5	$\varnothing$	$\varnothing$	$\varnothing$

τ-table

	$\overline{\tau}\|_a$	$\overline{\tau}\|_b$	$\overline{\tau}\|_c$	$\overline{\omega}\|_a$	$\overline{\omega}\|_b$	$\overline{\omega}\|_c$
1	Ω	Ω	c_1	$\{1\}$	$\{1\}$	$\varnothing$
2	a_1	Ω	Ω	$\varnothing$	$\{2\}$	$\{2\}$
3	Ω	b_1	Ω	$\{3\}$	$\varnothing$	$\{3\}$
4	a_2	Ω	Ω	$\varnothing$	$\{4\}$	$\{4\}$
5	Ω	b_4	Ω	$\{5\}$	$\varnothing$	$\{5, 6\}$
6	a_1	b_4	Ω	$\varnothing$	$\varnothing$	$\{6\}$

6 Conclusion

As we mentioned in section 4, Al-Khalifa et al. [1] used a similar stack data structure to compute efficiently structural joins (*stack-tree join algorithm*). The idea of having two kinds of attributes for each element in the stack is also found in the same work, where they are called *inherit list* and *self list*, as well as the way and the moment these data are propagated along the stack. The difference lies in the kind of information which is stored in the attributes: in Al-Khalifa et al.'s work, it is joined pairs of elements, while in ours it is partial matches involving disjointed descendants.

The positive aspect of using a stack-based algorithm, with respect to a table-based algorithm, is that the memory requirement is lower, since only the current rightmost branch is needed. Also, at all times, the attributes contain only the partial matches which can be completed in the future. The disadvantage, in comparison with the table-based algorithm is the completion process: there is no clever way to combine only the partial matches which can be combined, and not the others. This can be achieved by means of the τ-table, at the expense of some more memory. A sparse table encoding (by means of adjacency lists) would lower the memory requirement for the attribute table, so it is theoretically hard to decide which algorithm is better. In the same vein, notice also that, in the stack-based algorithm, the partial matches could share some information, in order to reduce the size of the attributes. For example, partial matches with the same root, i.e., all the partial matches of shape $\langle \kappa \mid \ldots \rangle$, with the same κ, could be grouped in a data structure indexed by κ. But, still, some general and efficient representation of the partial matches has to be devised. Also as a future work, implementations ought to be done and a benchmark to be run in order to compare to two algorithms.

References

[1] Shurug Al-Khalifa, H. Jagadish, Nick Koudas, Jignesh Patel, Divesh Srivastava, and Yuqing Wu. Structural joins: a primitive for efficient XML query pattern matching. In *Proceedings of the International Conference on Database Engineering (ICDE)*. IEEE, February 2002.

[2] Tatsuya Asai, Hiroki Arimura, Takeaki Uno, Shinichi Nakano, and Ken Satoh. Efficient tree mining using reverse search. In *Proceedings of the International Symposium on Information Science and Electrical Engineering (ISEE)*, pages 401–404. Kyushu University (Japan), November 2003.

[3] Nicolas Bruno, Nicolas Koudas, and Divesh Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the Special Interest Group on Management of Data (SIGMOD/PODS)*. University of Wisconsin, Madison (USA), Association for Computing Machinery (ACM), June 2002.

[4] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proceedings of the 28th VLDB conference*. Hong Kong,China, August 2002.

[5] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *Proceedings of the 27th VLDB conference*. Rome, Italy, September 2001.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, MacGraw-Hill, second edition, September 2001.

[7] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient structural joins. In *Proceedings of the 2003 IEEE conference on Data Engineering*. Bangalore, India, March 2003.

[8] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *Proceedings of the 29th VLDB conference*. Berlin, Germany, September 2003.

[9] E. Jiao, T. W. Ling, and C. Y. Chan. PathStack : A holistic path join algorithm for path query with not-predicates on XML data. In *Proceedings of the 2005 DASFAA conference*. Beijing, China, April 2005.

[10] Jan Van Leeuwen, editor. *Formal Models and Semantics*, volume B, chapter Rewrite Systems, pages 243–320. Elsevier, 1990.

[11] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th VLDB conference*. Rome, Italy, September 2001.

[12] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *Proceedings of the 2003 IEEE conference on Data Engineering*. Bangalore, India, March 2003.

[13] C. Zhang, J. F. Naughton, Q. Luo, D. J. DeWitt, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the 2001 ACM-SIGMOD conference*. Santa Barbara, USA, May 2001.