Index-Trees for Descendant Tree Queries on XML documents
(long version)

Jérémy Barbay

University of Waterloo, School of Computer Science,
200 University Ave West, Waterloo, Ontario, Canada, N2L 3G1
Phone 1-519-885-1211x7824, Fax: 1-519-885-1208, `jbarbay@uwaterloo.ca`

Abstract. We propose an index structure and an algorithm to answer *Descendant Tree queries*, the subset of XPath queries limited to the descendant axis, in computational models where elements can be accessed in non-sequential order. The indexing scheme uses classical labeling techniques, but structurally represents the ancestor-descendant relationships between nodes of same type, in order to allow faster searches. The algorithm solves XPath location path queries along the `descendant` axis in a holistic way, looking at the whole query rather than just concentrating on individual components, as opposed to "set-at-a-time" strategies. Using an adaptive analysis, we prove tight bounds on the worst case complexity in the comparison model, and a reasonable upper bound on the worst case complexity in the hierarchical memory model, which accounts for the number of I/O transfers at any fixed level.
Keywords: XPath location paths; comparison model; holistic; cache-oblivious; adaptive.

1 Introduction

1.1 Context

XML [30] is a rapidly emerging format for exchanging data on the web. It standardizes document tree structures, so that general tools can be developed and used for the many distinct applications adopting this standard. Among those tools, search engines are prominent, and are strongly based on path navigation. XPath [9] is a language for addressing nodes of an XML document by their positions in its tree structure. Each node is specified by the path from the root of the document to it, the path being incrementally described by a sequence of *location steps*. The simplest queries describe a chain of node tests in specified relations, and more complex queries describe a tree structure of node tests. For instance, the expression `//hidden//figure` corresponds to the set of `figure` nodes, descendant of `hidden` nodes, where `//` stands for the `descendant` axis. The query `//hidden[//section]//figure` has a branch, and corresponds to the `figure` nodes of the previous query sharing the `hidden` sub-tree with a `section` node. The XPath 2.0 [9] specifications define two syntaxes (abbreviated, used above, and un-abbreviated), 13 axes, and many expressions for predicates, but only a small subset of them is considered in most applications and studies. To evaluate such queries, a naive tree traversal strategy could cause a scan of the whole XML data tree even where there are few results.

As relational database implementations are now quite mature and well optimized, a natural approach to query efficiently XML documents has been to input XML documents in relational databases, and to use the existing relational technologies to query those [19, 27]. This results in *one-set-at-a-time* strategies, which decompose a query such as `//hidden[//section]//figure` in simpler queries, such as `//hidden//figure` and `//hidden//section`, and then combine the answers. This is termed as *structural join* [2], and as it is the most costly operation performed to solve a search query, various optimizations have been proposed. One solution consists in labeling and indexing the document, and to solve the queries using only the index. Some indexing schemes which take advantage of numbering schemes and encode each element as an interval [1, 3, 19] have been introduced, and algorithms which use in-memory stacks to insure that each input list from the index is scanned only once [2, 21]. Another solution consists in compressing the XML document. Buneman *et al.* [11] separate the textual content from the tree structure of the document, to compress the tree structure, and to use specific algorithm to search directly the compressed structure. Min *et al.* [24]

propose to compress the whole document in conjunction with a labeling scheme also encoding each element as an interval.

Holistic algorithms [10, 13] consider the whole query rather than just concentrating on the individual localization steps. They improve over one-set-at-a-time strategies, in which intermediate result sizes can get large, even when the input and output sizes are more manageable. Bruno *et al.* [10] proposes instead to study algorithms which avoid unnecessarily large intermediate results by scanning all lists in parallel and computing one final element of the answer set at a time. This approach supposes a native implementation to treat XML queries, an approach getting more and more interest [17, 20], as opposed to the storage and querying of XML documents in relational databases.

The Stream Model, where elements of the index can be accessed only sequentially, is a basic assumption used in all theoretical studies, whether they are based on a relational database implementation or a native one. It is a natural model for XML indexes and query results in relational databases, and is probably a cultural inheritance from the funding papers on the optimizations of structural joins. However in practice the assumption that all elements must be accessed sequentially does not hold for all applications, and stream indexes are implemented in B-trees, to reduce the complexity to sub-linear [19, 10], improving on the linear worst case complexity lower bound of the theoretical analysis, based on the stream model without B-trees. XR-trees [21] are an interesting exception, where a variant of B-trees is used, and theoretically analyzed, to index the nodes per tag. But the algorithms using this structure are still following the “set-at-a-time” strategy, with its associated limitations: although it performs well on structural joins, it might perform more joins than necessary, and be out-performed by a holistic algorithm.

The comparison model, a standard model in algorithmic analysis, allows random access to any element in constant time. It is mostly ignored in the context of XML queries, where people prefer to consider the more restricted Stream model. Among the various efficient techniques available, Bentley and Yao [8] propose several algorithms for unbounded search, which can be used to search efficiently for several items in a sorted array [4–6, 14, 15]. One of those algorithms permits the search for δ ordered elements in a sorted array of size n using $2\delta\log_2(1+n/\delta)$ comparisons. It is sometime called *one sided binary search* [29, 28], *exponential search* [12], or *doubling search* [4–6, 14, 15]: we will use this last name. A first analysis would conclude that the complexities of those algorithms are linear in n in the worst case, i.e. when $\delta = n$. A finer analysis distinguishes the two variables δ and n and concludes that the algorithm’s complexity is $O(\delta\log(1+n/\delta))$. Such finer analysis has proved useful in the study of adaptive algorithms for sorting problems [16, 23, 25] and for the computation of the intersection of sorted arrays [4, 5, 14, 15], but have not been used yet to analyze the complexity of queries on XML documents.

Those algorithms use binary search and as such do not perform very well when the data does not hold in memory, e.g. in the **hierarchical memory** [18] model, which accounts for the memory transfers between the levels of the memory (such as between cached and main memory, or between main memory and swapped memory on hard-drive). Bender *et al.* [7] propose a *cache-oblivious* implementation called *finger search*, which permits a similar performance in the hierarchical memory model: δ ordered elements can be found using $O(\delta(\log_B(1+n/\delta)))$ transfers of memory blocs of size B, where the size of the memory bloc is imposed by the cache system.

1.2 Contributions

We consider models where elements of the index can be accessed in any order, such as the *comparison* and the *hierarchical memory* models, which allow to use faster algorithmic techniques.

As traditional inverted list indexes do not permit such techniques, we propose using *index-trees* which structurally represent in the index the ancestor-descendant relationships between nodes of same type. The impending improvement in performance is similar in a minor way to the one obtained by the use of B-trees or variants for algorithms in the stream model, in the sense that some node references are not accessed in the index, which permits a sub-linear complexity. It is much more important though, as the semantic of the tree structure permits to skip an optimal number of elements of the index. This permits a proven sub-linear worst case complexity, which is optimal in a fine analysis grouping the instances upon their difficulty for a non-deterministic algorithm.

We propose a holistic algorithm using index-trees to solve XPath location paths along the `descendant` axis. As Bruno *et al.*’s holistic algorithm [10] for Twig Pattern Matching, the algorithm presented here treats the query as a whole and builds the answer to the query incrementally, one match at a time, in the document order. We define an adaptive analysis of XPath location steps along the `descendant` axis using index-trees. For this analysis, we distinguish several variables playing an important role in the difficulty of instances: the size k of the query; the number

n of nodes of the document concerned by the query (which is different from the size of the document, see Section 4); the maximal number h of nodes of same type on a branch of the document; the minimal number δ of comparisons needed to check the result of the query, for instance by a non-deterministic algorithm; and, in the particular model of hierarchical memory, the size B of the blocks exchanged at the memory level considered. This permits an analysis of the worst case complexity of descendant tree queries using index-trees in the comparison and hierarchical memory models, respectively $\Theta\left(\delta hk \log(1+n/\delta hk)\right)$ and $O\left(\delta hk\left(\log_B(1+n/\delta hk)+\log^*(1+n/\delta hk)\right)\right)$. The tight bound on the complexity in the comparison model is of theoretical interest, but the upper bound on the number of cache transfers in a hierarchical memory is of practical importance, as the number of cache transfers is a closer estimate of the running time of algorithms on huge amount of data.

1.3 Outline

The rest of the paper is organized as follows. We define shortly in Section 2 our labeling scheme, query language and index structure. In Section 3, we present our algorithm to solve descendant tree queries. We also prove its correctness, and we compare informally its performances with those of previous algorithms. In Section 4, we define a measure of the difficulty of an instance, analyze the complexity of the algorithm, and prove its optimality in the comparison model, as compared to deterministic algorithms. We finally conclude with a summary of results and some perspectives in Section 5.

2 Definitions

2.1 Labelling scheme for XML documents

Most of the examples in this paper are based on a simple XML document describing an article, from which Figure 1 gives the first lines, each line being prefixed by its number. In this sample of the document, `article` is the *type* of the `<article>` tag; the `<article>` tag marks the beginning of the document; the `<title>` and `</title>` tags correspond to the title of an article or of a section; and the `<section>` tag marks the beginning of a section, which ending will be marked by a `</section>` tag. The whole document contain tags of other types, among which the `<figure>` and `</figure>` tags, which correspond to subtrees defining the schemes of the article; and the `<hidden>` and `</hidden>` tags, which correspond to subtrees temporarily excluded from the publication process, for instance because they are still under-work. Each pair of tags, such as `<title>` and `</title>`, corresponds to a *node* of the tree described by the document, and its type is the type of those tags. In an index, the nodes of the document are referenced by labels. A good labeling scheme helps to solve navigation steps efficiently. We borrow the labeling scheme defined by Bruno *et al.* [10], where each node is referenced by an interval of integers. This labeling scheme reduces the decision whether two nodes are in a descendant relationship to a finite number of comparisons on their respective labels.

Definition 1 (label). *The* label of an internal node a *of the document is a pair* $s{:}e$ *where* s *and* e *are the respective ranks in the document order of the corresponding starting and ending* a*-tags. The* label of a leaf b *(e.g. a* `text` *or an attribute node) of the document is the rank* s *(noted* s *or* $s{:}s$*) in the document order of* b*.*

When there is exactly one tag or text per line of the document, the line numbers then correspond to the rank of the corresponding element, making it easier to compute the label of a node. For instance, the first `title` node has label $2{:}4$, because the corresponding `<title>` and `</title>` tags are on lines 2 and 4. Figure 2 presents a part of the tree representation of the same document, each node being annotated with its label. The interval formed by the labels of a node is a superset of all the labels of its descendant nodes, hence the labels permit the reduction of the the resolution of location steps to searches in sorted arrays. Given a context node of label $(s{:}e)$, the `descendant` axis corresponds to a search for nodes of label $(s'{:}e')$ such that $s{<}s'{<}e$, and location steps along the `ancestor` axis can be solved in exactly the same way using symmetric inequalities. Note that because of the tree-structure of the document, given two labels $(s{:}e)$ and $(s'{:}e')$, $s{<}s'{<}e$ implies $s{<}e'{<}e$ and $s'{\leq}e'$ by definition: it is obviously not necessary to check more than $s{<}s'{<}e$.

```
1:<article>
2: <title>
3:Index-Trees for (...)
4: </title>
5: <section>
6: <title>
7:Introduction
8: </title>
9: <section>
10: <title>
11:Context
12: </title>
13: <para>
(...)
```

Fig. 1. The numbered first lines of the document.

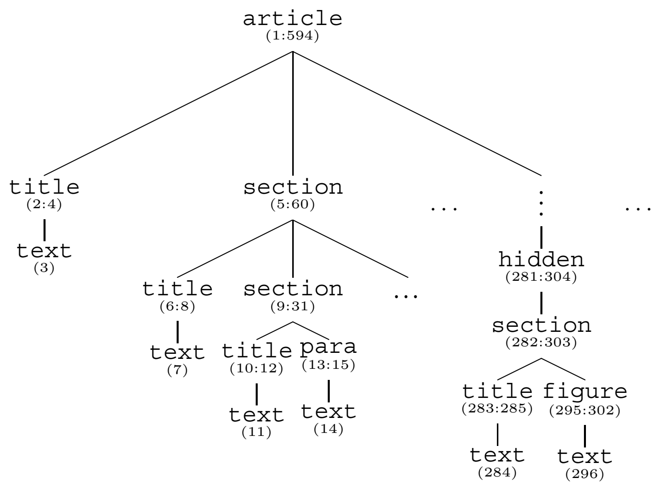

Fig. 2. A subset of the XML tree.

2.2 Query Language: a subset of XPath

In this paper, we focus on a subset of XPath queries consisting of `descendant` axis navigation steps (`//`), branches (`[...]`) and node-tests: Figure 3 gives such a query. The answer to this query is the list of `figure` labels, such that each corresponding node is descendant of a `hidden` node, which itself is ancestor of a `section` node. The purpose could be for instance to find the figures which have been removed from the publication process together with a whole section.

Q = `//hidden[//section]//figure`

Fig. 3. A simple XPath Query Q

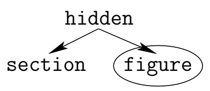

Fig. 4. The query tree of Q

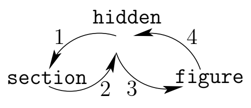

Fig. 5. The iterated tour of Q

Such queries can be represented as trees with a distinguished node, and are called *descendant tree queries*: Figure 4 gives the descendant tree corresponding to the query given Figure 3. We will denote by $\texttt{dist}(Q)$ the node-test distinguished by the query; by $\texttt{tour}(Q)$ the iterated tour of the nodes of Q defined by its preorder traversal, and by $|\texttt{tour}(Q)|$ the period of this tour; by ad and da respectively represent the `ancestor-descendant` and `descendant-ancestor` axes. For instance, the distinguished node of the descendant tree query given in Figure 4 is $\texttt{dist}(Q)$=`figure`; and the tour of this tree is $\texttt{tour}(Q)$=(`hidden, section, hidden, figure, hidden, ...`), of period $|\texttt{tour}(Q)| = 4$, as shown on Figure 5.

2.3 Our data-structure: Index-trees

We propose an index based on a tree structure which structurally represents the ancestor-descendant relationships of nodes, and which permits the use of algorithmic techniques taking advantage of the comparison and hierarchical memory models.

Definition 2 (index-tree). *The* index-tree *corresponding to a node α of a descendant tree query is a tree $I[\alpha]$ containing the labels of all the nodes matching α, such that any ancestor-descendant relationship in D corresponds to a parent-child or ancestor-descendant relationship in $I[\alpha]$, and such that the children of each node of an index-tree are consecutive elements in a sorted set.*

The simplest way to encode an index-tree is to create a dynamic tree which, for each node, lists pointers to the children in a sorted array. Such an encoding has only a constant factor space overhead in comparison with inverted list indexes commonly used for XML documents, and performs well in the comparison model. The lists of pointers to children can also be implemented using an exponential tree [7]. Such an encoding's space is within a constant factor of the array encoding, and performs well in both the comparison model and the hierarchical memory model. But as the exponential tree are much more complicated to represent than arrays, all the index-trees of this article are represented as arrays: it is sufficient to understand the underlying principle.

For a node-type α which nodes have no descendants of the same type, the index-tree is simply an inverted list of the labels corresponding to nodes of type α. In the document of Figure 1, the `figure` and `hidden` nodes are in this case: each index-tree, given Figure 6, consists just of a list of labels in document order.

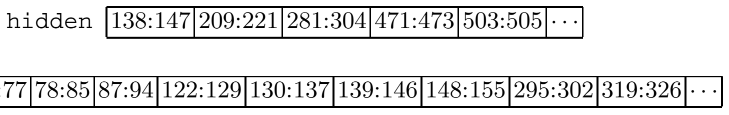

Fig. 6. A subset of the index-trees for `hidden` and `figure` nodes.

The tree structure of index-trees is apparent only for *recursive* node-types, which nodes can have descendants of same type. In the document of Figure 1, the `section` nodes are in this case: their index-tree, given Figure 7, is a tree structure of arrays of labels, where each array pointed by a label a corresponds to a list of `section` nodes accessible from a by a path containing no `section` node.

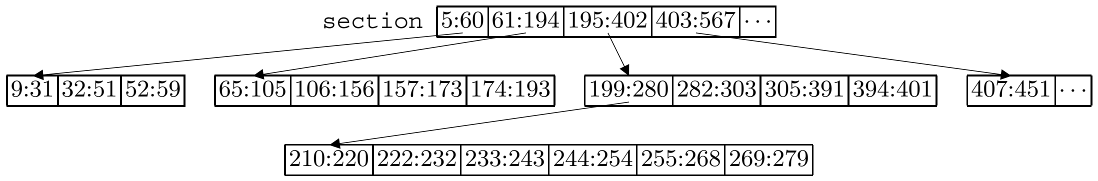

Fig. 7. A subset of the index-tree for `section` nodes.

We will denote by $\texttt{parent}(a)$ the function returning the label of the parent of a in its index-tree, if it has one; $\texttt{firstChild}(a)$ the function returning the label of the first child of a in its index-tree; $\texttt{rightSibling}(a)$ the function returning the first right sibling of a in its index-tree if there is one (there is at least ∞ if $a \neq \infty$), and returning ∞ if $a = \infty$; $\texttt{lastRightSibling}(a)$ the function returning the last right sibling of a, the ∞ label excluded, in its index-tree; and $\texttt{hasParent}(a)$, and $\texttt{hasChild}(a)$ the boolean functions true if and only if respectively a has a parent or some children in its index-tree.

A minor improvement brought by the use of index-trees is similar to the one obtained by the use of B-trees or variants, in the sense that not all labels are accessed in the index, which permits a sub-linear complexity. But the usefulness of index-tree goes far beyond this: as there can be only one ancestor (resp. descendant) b of a given node a at each level of a subtree, b can be searched with a number of comparisons logarithmic in the size of the array: this the reason to introduce Index-Trees.

2.4 Doubling search

Doubling search looks for an element x in a sorted array A of unknown size, starting at position $init$. It returns a value p such that $A[p-1]<x\leq A[p]$, called the *insertion point* of x in A.

Searching for the insertion point p of x in A from position $init$ is can be done in two phases: In the first phase the algorithm performs $i = \lceil \log_2(p-init) \rceil$ comparisons to find the interval $[init+2^i-1, init+2^{i+1}-1)$, and containing

p. To do so, it compares x to the elements occupying positions $(init, init+1, init+3, init+7, \ldots, init+2^{i+1}-1)$. In the second phase, a simple binary search on this interval, of size 2^i, finds p using i comparisons. The total number of comparisons performed is then $2\lceil\log_2(p{-}init)\rceil$, but more sophisticated algorithms improve the complexity by a constant factor [8, 26].

For instance, the first `hidden` node of the document, has label $138{:}147$ (see the `hidden` index-tree of Figure 6). A doubling search for descendants of this `hidden` node in the `figure` index-tree, results in the comparisons shown Figure 8. The first three arrows correspond to the first phase of the algorithm, the other corresponds to the binary search, and the label finally found is $139{:}146$.

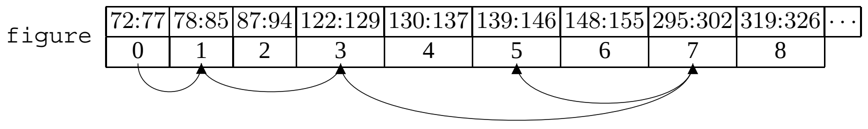

Fig. 8. Comparisons performed during the doubling search in the index-tree for `figure` nodes.

This technique applies as well to recursive node-types, when indexed by an index-tree. For instance, the `figure` node of label $122{:}129$ is a descendant of both `section` nodes of label $61{:}194$ and $106{:}156$ (see the `section` index-tree of Figure 7). A doubling search in the array at the first level finds the label of the first `section` ancestor; and a second search in the array pointed by this node finds the label of the second `section` ancestor. The application of the doubling search algorithm to the label search in a sorted array of labels necessitates a few more comparisons which don't change the order of its complexity. It is implemented in function `SearchInArray`(a, b), which is given by Algorithm 5.

An index based on inverted lists, even if optimized by B-tree or variants, do not permit the use of binary nor doubling search algorithms to find the ancestors or descendants of recursive nodes. The reason is that, in a list of recursive node labels, a comparison does not permit the division in two of the search space, as it is the case in the sorted arrays of each node of an index-tree. For instance, the labels of the `section` nodes, ancestors of the `figure` node in the previous example, are not consecutive in the preordered list of `section` labels (given in Figure 9). From a test on the `section`-node label $65{:}105$, no algorithm can decide to reduce its research to the left or right side of the array, as both sides can (and here do) contain the label of an ancestor of the `figure` node.

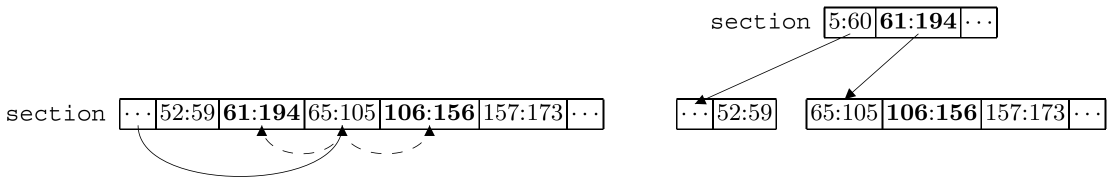

Fig. 9. A subset of the list of `section` node labels, in document order, and the corresponding fragment of the `section` index-tree. The labels in bold font correspond to the ancestors of the `figure` node of label $122{:}129$, and are not consecutive.

3 Algorithm

We propose a holistic algorithm composed of four functions, described in Section 3.2. Each function is presented with a short description, its algorithm, and an execution example. The correctness of each function, and hence of the whole algorithm, is proved in Section 3.3, and the space, comparison and memory transfer complexities are briefly discussed in Section 3.4. The complexity of the algorithm is formally analyzed in Section 4. In general we denote by Q the descendant tree query; by $\alpha, \beta \in Q$ some of its location step; by $A = I[\alpha]$ the index-tree corresponding to the node-test of α (resp. $B = I[\beta]$ for β); and by $a \in A$ a label in A (resp. $b \in B$). Table 1 gives the notations and denominations of some helpful relations between labels. Furthermore, to simplify the algorithm, we signal the end of

$a \subset b$	$b.s < a.s$ and $a.e < b.e$	a is a *descendant* of b. b is an *ancestor* of a.
$a < b$	$a.s < b.s$	a is a *preorder predecessor* of b. b is a *preorder successor* of a.
$a\hat{}b$	$a.e < b.s$	a is a *predecessor* of b. b is a *successor* of a.
$a \xrightarrow{r} b$	$(r = ad$ and $b \subset a)$ or $(r = da$ and $a \subset b)$	a is *in relation r with* b, or b is *in relation r from* a.

Table 1. Notations given a relation $r \in \{ad, da\}$ on labels a, b.

the arrays by a "fake" label ∞, such that $\forall a$, a is a predecessor of ∞, and such that $\forall r$, ∞ is in relation r with ∞.

3.1 General outline of the algorithm

Given an index I and a query Q:

```
note $\alpha$ the first location step of the tour of $Q$;
**while** no index-tree is empty **do**
    match $a$, the first element of $I[\alpha]$, with $\alpha$;
    **while** some location step of $Q$ are unmatched **do**
        note $\alpha$ the last matched test-node;
        note $\beta$ the next test-node in the tour of $Q$;
        move up and down the index tree $I[\beta]$ using doubling searches, to find the first label $b \in I[\beta]$ such that $a', b$
        matches $\alpha, \beta$, where $a'$ is either $a$ or a preorder successor of $a$;
        eliminate all predecessors of $b$ in $I[\beta]$;
        **if** the relations $a, b$ and $\alpha, \beta$ are the same **then** match $b$ with $\beta$;
        **else** unmatch all nodes of $Q$, and match $b$ with $\beta$; **end if**
    **end while**
    outputs the label matching the distinguished node of $Q$;
**end while**
```

The algorithm outputs the labels of all the nodes matching Q. Each search in $I[\beta]$ eliminates some labels which cannot be part of a match of Q, and eventually one search will result in a partial match. Choosing the location steps by following a tour of the descendant tree query insures that the algorithm will perform at most k searches, for each search performed by a non-deterministic algorithm. When all labels of an index-tree $I[\alpha]$ have been eliminated, no more match of Q can be found: the algorithm can then terminate.

3.2 Functions

The function `Enumerate` (Alg. 1) builds the list of nodes corresponding to the location steps of a descendant tree query Q. To do so, it iteratively calls the functions `Next` and `Skip`; which respectively finds the label of the next node matching Q, and skips it once it is reported; till all elements of one of the index-trees have been considered. For instance, given the index-trees of Figures 6 and 7, and the descendant tree query given Figure 4, the first call to the function `Next` updates the pointers of P and returns the label $a = 295{:}302$, which corresponds to the first `figure` node, descendant of a `hidden` node, which itself is ancestor of a `section` node. Entering the loop, the label a is added to the list R and disabled by updating the corresponding value in P to its successor in the `figure` index-tree, $319{:}326$. The loop is then iterated till `Next` returns ∞, when finally the function returns R.

Algorithm 1 `Enumerate`(I, Q)

Given an index I, a query tree Q; the function returns the list of all labels corresponding to Q.

```
for all $\alpha \in Q$ do
    $P[\alpha] \leftarrow$ first element of $I[\alpha]$ ;
end for
$R \leftarrow$ empty list; $a \leftarrow \texttt{Next}(Q, P)$;
while $a \neq \infty$ do
    $R \leftarrow R \cup \{a\}$;
    $P[\texttt{dist}(Q)] \leftarrow \texttt{Skip}(P[\texttt{dist}(Q)])$;
    $a \leftarrow \texttt{Next}(Q, P)$;
end while
return $R$;
```

The function `Skip` (Alg. 2) returns the next label in preorder traversal of an index-tree. Its implementation is straightforward. For instance, given the `figure` label $295{:}302$, which has no children, but has a sibling, the function `Skip`

Algorithm 2 `Skip`(a)

Given a label $a \neq \infty$, the function returns the next label in the same index-tree in the preorder traversal, or ∞ if none exists.

```
if $\texttt{hasChild}(a)$ then return $\texttt{firstChild}(a)$ ;endif
while $\texttt{rightSibling}(a) = \infty$ and $\texttt{hasParent}(a)$ do
    $a \leftarrow \texttt{parent}(a)$;
end while
return $\texttt{rightSibling}(a)$;
```

returns directly its successor $319{:}326$. If given the `section` label $269{:}279$, which has no children and no sibling, the function `Skip` returns directly the first right sibling of its parent, $319{:}326$.

The function `Next` (Alg. 3) searches for the next match of the query Q among preorder successors of the labels given in P. During its search, it updates the values of P, so that the next search ignores the labels already disabled. Note that there is always a match corresponding to Q, if only the set of ∞ labels which terminate the array of the roots of the index-trees. Once the match is found, the function returns the distinguished node of this match. To perform the search of the next match of Q, each node of Q is successively bound to a label of the corresponding index-tree. The nodes are considered in the order given by the iterated tour of Q. Each node is bound at least once every period of a tour, which is at most $2k$: this permits a better complexity on easy instances (see Section 4). For instance, given the

Algorithm 3 `Next`(Q, P)

Given a query tree Q, and an array P of pointers to labels of the index-trees; the function returns the first label matching the distinguished node of Q, while ignoring all labels preceding the positions indicated by P in the index-trees.

```
$\alpha \leftarrow$ any element of $\texttt{tour}(Q)$; $s \leftarrow 1$;
while $s < |\texttt{tour}(Q)|$ do
    $\beta \leftarrow$ the node following $\alpha$ in $\texttt{tour}(Q)$;
    $r \leftarrow$ the relation corresponding to $(\alpha, \beta)$ in $Q$;
    $P[\beta] \leftarrow \texttt{SearchInTree}(P[\alpha], r, P[\beta])$;
    if $(P[\alpha] \xrightarrow{r} P[\beta])$ then $s \leftarrow s + 1$; else $s \leftarrow 1$; endif
    $\alpha \leftarrow \beta$;
end while
return $P[\texttt{dist}(Q)]$;
```

descendant tree query Q of Figure 4 and the array P pointing to the first labels of the index-trees of Figures 6 and 7, the function `Next` successively binds the nodes of Q in the order defined by the tour of Q, as shown in Figure 10. The root of Q (any other node of Q could have been chosen) is first bound to the first `hidden` label available, $138{:}147$ (Fig. 10a). As no label in the `section` index-tree is a descendant of this label, the `hidden` node of Q is unbound, and the `section` node of Q is bound to the label $157{:}173$, which corresponds to the first `section` node possibly a descendant of a preorder successor of the `hidden` node (Fig. 10b). Similarly, no label in the `hidden` index-tree is an ancestor of the node of label $157{:}173$. The `section` node is unbound, and the `hidden` node is bound to the label $209{:}221$, the first node possibly an ancestor of a successor of the `section` node (Fig. 10c).

Following the tour of Q, it is now the turn of the `figure` node to be bound. No `figure` node is a descendant of the node of label 209:221, hence the `hidden` node of Q is unbound, and the `figure` node of Q is bound to the label $295{:}302$ (Fig. 10d). This node is finally part of a match of the query, so the function successfully binds the `hidden` node to the label $282{:}303$ (Fig. 10e) and the `section` node to the label $281{:}304$ (Fig. 10f). Having found a match to the tree descendant query, the function then returns the label $295{:}302$, which corresponds to the distinguished node of Q, the `figure` node.

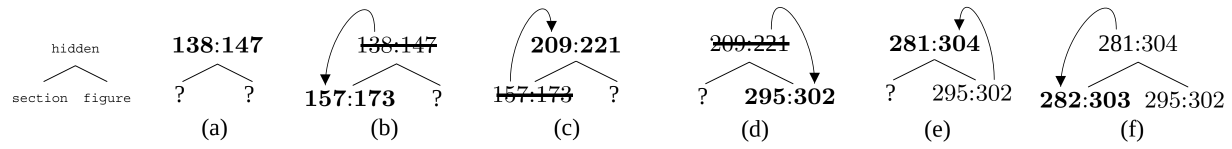

Fig. 10. The successive bindings of the nodes of Q during an execution of the function `Next`. The newly bound labels are in bold, and the newly disabled labels are crossed. The arrows indicate the tour of Q performed by the function.

The function `SearchInTree` (Alg. 4) searches, among b_0 and its preorder successors in its index-tree, for a label b in relation r from a. It returns this label if there is one, and otherwise returns the first label which could be in relation r from a successor of a. If even such a label does not exist in the index-tree, it returns the label ∞, the last element of the array at the root of the index-tree. To do so, the function first moves up from b_0 in the index-tree, as long as all labels of the array containing b are predecessors of a. Once found an array containing at least one label which is either an ancestor, a descendant, or a successor of a, the function returns this label if $r = da$. Otherwise it goes down the index-tree, searching in each array for an ancestor or descendant of a, till it finds a descendant and returns it, or finds that none can exist. If there is no descendant of a in the index-tree of b_0, the function searches the label of the first node possibly a descendant of a preorder successor of a, if necessary going up the index-tree to find a finite label. For instance, consider the `hidden` label a=138:147, the `section` label b=5:60, and the relation r=ad. In the array containing b, the label 61:194 corresponds to an ancestor of a (see Fig. 11). Any `section` node descendant of a has to be a descendant of this node, hence the function updates b down one level in the `section` index-tree. In this new array the label $b = 106{:}56$ corresponds also to an ancestor of a, which has no `section` descendants: hence there are no `section` nodes among the descendants of a.

Any preorder successor a' of a is either a descendant of b or a successor of it: b cannot be a descendant of a'. Hence the label of the first section node possibly descendant of a preorder successor of a is the sibling of b, of label $157{:}173$, which is returned by the function. Note that if the function is called with $b = 23{:}51$, instead of the first label of the `section` index-tree, the function behaves the same, after checking that no label of the array containing b can be in relation with a (see Fig. 7).

The function `SearchInArray` (Alg. 5) searches, in the part of the array on the right side of b_0, for a label which can be an ancestor or descendant of the label a. It returns this label if there is one, and otherwise returns the first label which could possibly be an ancestor or descendant of a preorder successor of a. To do so, it simply performs a doubling search for the insertion rank of $a.s$ among the $b.s$ components of the labels of the array. Then the relative

Algorithm 4 `SearchInTree`(a, r, b_0)

Given the labels a and b_0, and a relation $r \in \{ad, da\}$; the function returns, if there is one, the first label b, placed after b_0 in the preorder traversal of the index-tree of b_0, such that $a \xrightarrow{r} b$. Otherwise it returns the first label which could be in relation with a successor of a.

```
$b \leftarrow b_0$;
while hasParent($b$) and lastRightSibling($b$)$\hat{}a$ do
    $b \leftarrow$ parent($b$);
end while
$b \leftarrow$ SearchInArray($a, b$);
if $r = da$ then return $b$; endif
while hasChild($b$) and $a \subset b$ do
    $b \leftarrow$ SearchInArray($a$, firstChild($b$));
end while
if $a \subset b$ then $b \leftarrow$ rightSibling($b$); endif
while $b = \infty$ and hasParent($b$) do
    $b \leftarrow$ rightSibling(parent($b$));
end while
return $b$;
```

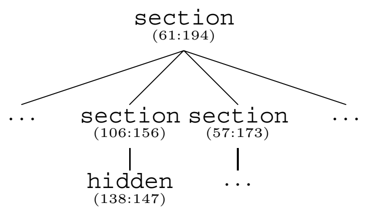

Fig. 11. No `section` node is a descendant of the `hidden` node of label a=138:147. The first `section` node which could be a descendant of a preorder successor of the node of label a has label 157:173.

position of a can be deduced by simply comparing $a.s$ with the $b.e$ component of the elements in the array around the insertion rank. For instance, given the `hidden` label 138:147 , and the `figure` label 139:146, the function performs

Algorithm 5 `SearchInArray`(a, b_0)

Given the labels a and b_0 the function returns the first label b succeeding to b_0 in the same array, such that either $a \subset b$, $b \subset a$, or $a\hat{}b$.

```
Using doubling search, find $b_1$ and $b_2$,
consecutive labels from the same array as $b_0$ such that
$b_0.s \leq b_1.s < a.s \leq b_2.s$;
if $a.s < b_1.e$ then return $b_1$; else return $b_2$;
```

the comparisons described in Figure 8, to find the consecutive labels b_1=130:137 and b_2=139:146. As $138 > 137$, the function finally returns $b_2 =$ 139:146.

3.3 Correctness proof

We prove the correctness of our algorithm in four steps, the last one being the proof of the correctness of Function `Enumerate`. Lemma 1 states the correctness of Function `SearchInArray`, which is simply adapting the doubling search algorithm to search on arrays of labels. Lemma 2 states the correctness of Function `SearchInTree`, and is used in the proof of Lemma 3 which states the correctness of Function `Next`. Lemma 4 states the correctness of Function `Skip`, and is used with Lemma 3 to prove Theorem 1, which states the correctness of Function `Enumerate` and hence of the general algorithm.

Lemma 1 (Correctness of `SearchInArray`). *Given the labels a and b_0, Algorithm 5 returns the first label b, succeeding to b_0 in its index-tree, such that either $a \subset b$, $b \subset a$, or $a\hat{}b$.*

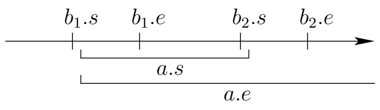

Fig. 12. The possible positions of a in comparison to b_1 and b_2 in Algorithm 5.

Proof. By definition of the doubling search, $b_0.s \leq b_1.s < a.s \leq b_2.s$ (see fig. 12). By definition of the labelling scheme, each pair of intervals is related either by an inclusion or by an empty intersection. By definition of the index-tree, there can be only one label b in the array such that $a \subset b$, and all labels b such that $b \subset a$ must be consecutive in the array.

If $a.s < b_1.e$, then $a.e \leq b_1.e$ and b_1 is the unique label of the array such that $a \subset b_1$: that is the correct value to be returned. On the other hand if $a.s > b_1.e$, then $b_1\hat{}a\hat{}b_2$ of $b_2 \subset a$: either way, b_2 is then the label corresponding to the definition of the correct output of the algorithm. □

Lemma 2 (Correctness of `SearchInTree`**).** *Given the labels a and b_0, and a relation $r\in\{ad, da\}$, Algorithm 4 returns, if there is one, the first label b, preorder successor of b_0 in its index-tree, such that $a \xrightarrow{r} b$. Otherwise it returns the first label which could be in relation with a preorder successor of a.*

Proof. Throughout Algorithm 4, the values taken by variable b delimit which labels are a potential output of the algorithm or not. For conciseness, we call *disabled* the preorder predecessor of b, *available* the others, and b_f the correct return value of Algorithm 4.

The path from the root to b in the index-tree delimits, in each array crossed, the boundary between disabled and available labels. The labels in the arrays not crossed by this path have recursively the same status than their parent. The two successive loops which constitutes the algorithm, disable more labels by updating b with preorder successors, till $b = b_f$ for the relation r.

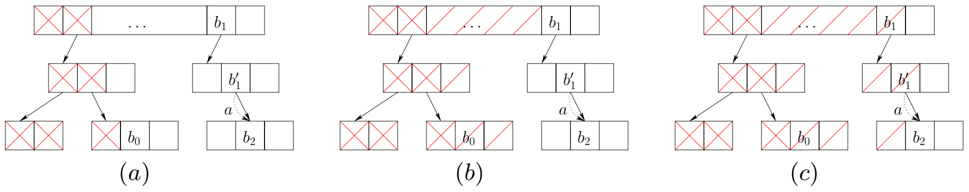

Fig. 13. Schematic representation of the index-tree for b-nodes during the execution of the function `SearchInTree`, at the time of the call (a), after the first loop (b) and after the second loop (c). The crossed locations correspond to disabled labels, and the noted labels are such that $b_0\hat{}b_1 \supset b_1' \supset a \supset b_2$. The algorithm returns b_1 if $r{=}da$, and b_2 if $r{=}ad$.

Whether $r = da$ or $r = ad$, either a and b_f are on the same root-to-leaf path in the document, or $a\hat{}b_f$. The first loop disables the sub-trees of the index-tree which can't possibly intersect a root-to-leaf path containing a, by updating b up from b_0 when `lastRightSibling`$(b)\hat{}a$. It finds the first array on the path from b_0 up to the root, containing at least one label which is either an ancestor, a descendant, or a successor of a: in Figure 13, this is the array containing b_1. The following doubling search (through the call to `SearchInArray`, proved correct in Lem. 1), finds such a label.

At this point, b is the searched label for $r{=}da$ (descendant-ancestor relation): if b is an ancestor of a, then by construction it is the first one available; otherwise, it is the first available label which could be an ancestor to a successor of a, as all ancestors of b have already been disabled (by being anterior to b_0) and all predecessors of b in the array are such that $b.s < a.s$.

The second loop updates b in the index-tree down the path intersecting the root-to-leaf path containing a, as long as b is a descendant of a: at the end of this loop either $b \subset a$ or $a\hat{}b$: in Figure 13 it corresponds to the path containing

b_1, b'_1 and b_2. At this point, the ancestors of b are either anterior to b_0 or ancestors of a, and the left siblings of b are not on a root-to-leaf path containing a. If $b \subset a$, then by construction b is the first descendant of a available, it won't be modified anymore by the function and it will be returned. If $a \subset b$ but b has no descendant of the same type, then b cannot be an ancestor of any preorder successor of a, hence the algorithm returns its preorder successor in the index-tree.

If $b = \infty$, a whole array has been disabled. The third loop updates b up the index-tree to find the next available label different from ∞, or ∞ if there is none in the whole index-tree. Hence the correctness of the algorithm, as $a\hat{}\infty$. □

Lemma 3 (Correctness of `Next`). *Given a query tree Q, and an array P of pointers to labels of the index-trees, Algorithm 3 returns the first label matching Q, while ignoring all labels being preorder predecessors of the labels of P.*

Proof. The algorithm counts in variable s the number of consecutive matches in the tour of Q. As the call to the function `SearchInTree` does not change anything if the labels pointed by P are already in relation, any value of s larger than the period $|\texttt{tour}(Q)|$ of the iterated tour of Q indicates that a match is complete: any label returned by the algorithm corresponds to a match. As Lemma 2 proved that the function `SearchInTree` returns the *first* label b, preorder successor of b_0, such that either $a \xrightarrow{r} b$ or $a\hat{}b$, no label part of a match is disabled: the label returned by the algorithms corresponds to the first match available. □

Lemma 4 (Correctness of `Skip`). *Given a label $a \neq \infty$, Algorithm 2 returns the preorder successor of a in its index-tree, or ∞ if none exists.*

Proof. For any node of a tree, the next node in the preorder traversal is its child if it has one, the first right sibling if it has one, or the next node to its parent if it has one. As the array at the root of each index-tree is terminated by a label ∞ which has no children, the algorithm will return the label ∞ at the end of the traversal. □

Theorem 1 (Correctness of `Enumerate`). *Given an index I, a query tree Q, Algorithm 1 returns the list of all labels corresponding to Q.*

Proof. Algorithm 1 is a simple application of the functions `Next` (Alg. 3) and `Skip` (Alg. 2): Iteratively, the function computes the first match constituted of available labels by a call to function `Next`, adds it to the variable R, disables the label matched by the distinguished node of the descendant tree query by a call to function `Skip`, till all the labels are disabled in at least one of the index-trees. The correctness of the functions used is proved by Lemmas 3 and 4: hence Algorithm 1 returns the list of all labels corresponding to Q. □

3.4 Comparisons with Other Algorithms

Space complexity The index itself is of size linear in the size of the document, which is comparable to the space required by other data structures. As the algorithm outputs sequentially the nodes matching the query, it needs only to maintain a partial matching and the positions in each index delimiting the disabled labels. If all positions in the index hold in a single memory unit, its needs are linear in the size of the query, and are independent of the input and output sizes. If the document is so large that the positions in the index take more than one unit of memory, it will need $O(k \log(N))$ bytes, where k is the size of the query and N is the size of the document.

Hence its memory requirements, apart from the index itself, are small. Those requirements are comparable to the requirements of other holistic algorithms based on native implementations. They are quite small in comparison to the worst case space requirements of algorithms in the stream model, where intermediary results can be of size linear in the size of the document, even if the final result is empty.

Time Performance In a model where the cost of memory accesses is uniform, doubling search is performing better than sequential access, making our algorithm much more efficient than any other algorithm based on the stream model.

In a model where memory is divided in different levels forming a hierarchy, with different access costs for each level of the hierarchy, performance cannot be measured by the number of comparisons performed. In this context, the performance is usually studied only on the interface between two levels, the slowest one being called the *main memory*, and the fastest one being called the *cache*.

The number of *memory transfers* is then a more adequate measure of the performance of algorithms. By implementing each sorted set of the index-trees as *exponential trees* defined by Bender *et al* [7] and using their cache-oblivious algorithm for doubling search, the number of memory transfer is kept logarithmic in the number of nodes concerned. The result from Bender *et al.* is not known to be optimal (yet), but its optimality would imply directly the optimality of our algorithm, by a proof similar to the proof of Theorem 3.

Hence our algorithm, using efficiently the tree-structure of the index-tree, can largely outperform the usual algorithms in the stream model: Section 4 proves how much it does.

4 Adaptive Analysis

4.1 Certificates and non-deterministic algorithms

All correct algorithms have in common that they must check a *certificate* of their result, which can be much larger than the solution of the problem. In particular, queries without a match can have a certificate of arbitrary size, very small or as large as the size of the instance. For instance, the document presented in Figure 14 has no node matching

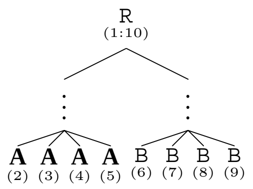

Fig. 14. An easy document for the query $A//B$.

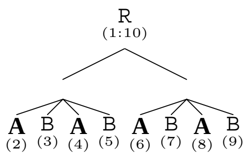

Fig. 15. A difficult document for the query $A//B$.

the query $Q = B//A$. A single comparison, between the last label 5 of the A index-tree, and the first label 6 of the B index-tree, certifies it: all the A labels are equal or preorder predecessors of 5 and all the B labels are equal or preorder successors of 6, hence $5 < 6$ implies that all the A labels are preorder predecessors of all the B nodes. On the other hand, the document presented in Figure 15 also has no node matching $Q = A//B$, but this is more difficult to certify: 7 comparisons are needed to check this ($2 < 3 < 4 < 5 < 6 < 7 < 8 < 9$).

All correct deterministic, probabilistic or non-deterministic algorithms must compute a certificate of some sort, but non-deterministic algorithms can just *guess* a certificate and check it. Then the complexity of the best non-deterministic algorithm corresponds to the size of the certificate, and is called the *non-deterministic complexity* of the instance. The function `NonDeterministic` (Alg. 6) is a non-deterministic algorithm computing the list of all labels corresponding to Q with the minimum number of comparisons needed.

Of course this algorithm do not use any binary nor doubling search, because the non-determinism permits a direct guess of the insertion ranks. Beside this point, the algorithm performs all the steps that a randomized or deterministic algorithm would perform: it traverses the index-trees in parallel, looking for either some match or some proof that the label currently considered do not correspond to a part of a match.

Definition 3 (non-deterministic complexity). *The minimal number of comparisons that a non-deterministic algorithm performs on an instance is called the* Non-Deterministic Complexity *of the instance. It is a lower bound of the complexity of any algorithm on this instance.*

This lower bound on the complexity in the comparison model is usually weak, but is still a good measure of the relative difficulty of the instances. When an instance is more difficult for a non-deterministic algorithm, then it is also more difficult for restricted algorithms, such as probabilistic or randomized algorithms.

Algorithm 6 `NonDeterministic`(I, Q)

Given an index I, a descendant tree query Q; the function returns the list of all labels corresponding to Q.

```
$R \leftarrow \emptyset$;
**for all** $\alpha \in Q$ **do**
  $P[\alpha] \leftarrow$ first element of $I[\alpha]$ ;
**end for**
**while** $\forall \alpha \in Q,\ I[\alpha] \neq \infty$ **do**
  guess $\beta \in Q$; $\alpha \leftarrow$Parent$(\beta)$ ;
  $a \leftarrow P[\alpha]$;
  $P[\beta] \leftarrow \min\{b \in I[\beta],\ b \geq P[\beta]$ and $(b \subset P[\alpha]$ or $b\hat{}P[\alpha])\}$;
  **if** $\forall \beta \in Q,\ P[\beta] \subset P[$Parent$(\beta)]$ **then**
    $R \leftarrow R \cup \{P[\text{dist}(Q)]\}$;
    $P[\text{dist}(Q)] \leftarrow \min\{b \in I[\text{dist}(Q)],\ b > P[\text{dist}(Q)]\}$;
  **end if**
  **return** $R$;
**end while**
```

It is important to note that the non-deterministic complexity in the comparison model does not correspond to a lower bound on the number of cache transfer in the hierarchical memory model: all the comparisons could occur in the same memory bloc. Nevertheless, it is still a good measure of the difficulty of the instances for deterministic or randomized algorithms in the hierarchical memory model, because it corresponds to the minimal size of a certificate, that any correct algorithm must check, whatever the model.

4.2 Difficulty of an instance

Measures of difficulty distinguishes between the instances of same size but distinct difficulties. The obvious measures of difficulty for descendant tree queries on XML documents, provided their index-trees, are the sizes of the constituents of the instance: the size k of the tree-query Q, and the number n of nodes of the document concerned by Q:

Definition 4 (concerned nodes). *A node of the document is* concerned *by a query Q if it matches any node-type test of Q. For each node-type test α of Q, note n_α the number of nodes matching α. Then the number of nodes of the document concerned by Q is $n = \sum_\alpha n_\alpha$.*

Most often, n is smaller than the size of the document, especially when all node-type tests of the query are distinct. In the worst case n is smaller than k times the size of the document. Some other properties can make some documents more difficult than others, among which is the height of the document, and in particular the maximal height of the index-trees.

Definition 5 (recursivity). *The* recursivity *of an instance is the maximal number h of nodes of same type on a root-to-leaf path in it.*

The recursivity of an instance depends only of the document structure. One aspect of the relation between the tree-query and the document which creates variation in the difficulty of the instance is the number of nodes concerned, but there are deeper ones, such as the performance of a non-deterministic algorithm such as Algorithm 6: the performance of the best non-deterministic algorithm is a natural lower bound of the performance of any probabilistic or deterministic algorithm.

Definition 6 (alternation). *For a given instance composed of a descendant tree query, an XML document and its index-trees, we denote by δ the number of negative tests and whole matches performed by Algorithm 6 on this instance. As it is the number of times when Algorithm 6 starts over the matching process from one single node (and "alternate" its template match), we call this measure of difficulty the* alternation *of the instance.*

For instance the alternation of the instance formed by the query $Q = A//B$ and the document of Figure 14 is $\delta = 1$; and the alternation of the instance formed by the query $Q = A//B$ and the document of Figure 15 is $\delta = 7$;

Note that, for instances without match, the alternation is exactly the number of comparisons performed by the Algorithm 6, i.e. the non-deterministic complexity of the instance. For instances with some matches, the alternation is the number of matches plus the number of comparisons disabling labels. This is different from the non-deterministic complexity, but adequate for a study of the complexity of deterministic algorithms: a document, built by the adversary of a deterministic algorithm A, will maximize the number of comparisons performed by A for each failed match; to the extent where A performs as many comparisons on a successful match than on a failed match.

4.3 Complexity of the algorithm in the Comparison Model

We analyze the worst-case complexity of our algorithm by an adaptive analysis in function of δ, h, k, and n. The worst-case complexity in the comparison model is proved optimal in our adaptive model by Theorem 3. Note that an instance of recursivity h has no index-tree of height larger than h.

Theorem 2 (Complexity in the Comparison Model). *Our algorithm performs $O\left(\delta hk\log(1+n/\delta hk)\right)$ comparisons on an instance of alternation δ, composed of a descendant tree query of k nodes, concerning n nodes, of a document of recursivity h.*

Proof. Let δ be the alternation of the instance, and $(a_i)_{i\leq\delta}$ the corresponding ordered sequence of labels chosen initially or just after a failed or successful match, as guessed by a non-deterministic algorithm. By definition, δ is the minimal length of such a sequence. For commodity, note a_0 a pedigree preceding any other in the document.

Each doubling search performed by the algorithm is said to be "in phase i" if the label a searched is either placed between a_{i-1} and a_i, or equal to a_i, for all $i \in \{0, \ldots, \delta - 1\}$. Such a phase is called *positive* if a_i is a match, and *negative* otherwise. There are exactly δ such phases, and as the nodes are considered following the iterated tour of Q, in each phase the algorithm performs at most $2k$ calls to the function `SearchInTree`. During each of these calls, in the worst case b is going up to the root and down to a leaf. As the index-tree is of height at most h, at most h calls to the function `SearchInArray` are performed during a call to the function `SearchInTree`.

For each phase $i \in \{0, \ldots, \delta\}$, for each call $j \in \{1, \ldots, k\}$ to the function `SearchInTree`, and for each of the levels $l \in \{1, \ldots, h\}$ of the corresponding index-tree, let be $n_{(i,j,l)}$ the number of labels skipped by the function `SearchInArray`. Each call to the function `SearchInArray` performs then $O(\log(1 + n_{(i,j,l)}))$ comparisons, summing to $O(\sum_{i=0}^{\delta-1}\sum_{j=1}^{k}\sum_{l=1}^{h}\log(1+n_{(i,j,l)}))$. This is smaller than

$$O\left(\delta hk\log\left(1+\sum_{i=0}^{\delta-1}\sum_{j=1}^{k}\sum_{l=1}^{h}n_{(i,j,l)}/\delta hk\right)\right),$$

because of the concavity of the function $\log(1 + x)$. As each search starts exactly where the previous one ended, $\sum_{i=0}^{\delta-1}\sum_{j=1}^{k}\sum_{l=1}^{h}n_{(i,j,l)} \leq n$, and the algorithm's complexity is $O(\delta hk\log(1 + n/\delta hk))$. □

4.4 Lower Bound

In this section we show that the complexity of the algorithm presented in Section 3 is asymptotically optimal.

Theorem 3 (Lower Bound in the Comparison Model). *For any $\delta \geq 1$, $h \geq 1$, $k \geq 2$, $n \geq \delta hk$ any deterministic algorithm answering descendant tree queries, on instances formed by a descendant tree query of $O(k)$ nodes, concerning $O(n)$ nodes, of a document of recursivity $O(h)$; has complexity $\Omega\left(\delta hk\log(n/\delta hk)\right)$ in the comparison model.*

Proof. Here is a summary of the proof: we show first how the lower bound in the case where $(\delta = 1, h = 1, k = 2, n \geq 2)$ is exactly the lower bound for the binary search in a sorted array. Then we successively generalize the lower bound to the cases where the values of h,k and δ are relaxed.

We first consider the descendant tree query $A//B$ of size 2; on documents of $n + 2$ nodes, such that the root is of type R, has $n - 1$ children of type B and 1 child of type A, which itself has a child of type B (see Fig. 16). This forms instances of alternation $\delta = 1$, as guessing the B-node child of the A-node suffices to certify the match. The index-tree for B is an array of n elements among which exactly one is the solution. As each comparison permits only to divide the search space by two, $\log_2 n$ comparisons are necessary to any deterministic algorithm solving the instance, hence the lower bound for the instances such that $(\delta = 1, h = 1, k = 2, n \geq 2)$.

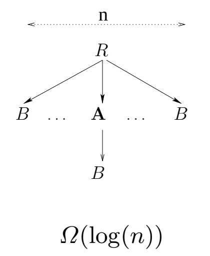

Fig. 16. $(\delta{=}1, h{=}1, k{=}2, n{\geq}2)$

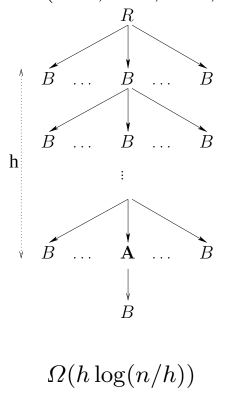

Fig. 17. $(\delta{=}1, h{\geq}1, k{=}2, n{\geq}2h)$

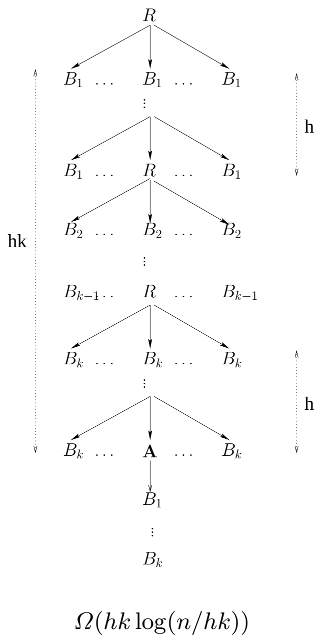

Fig. 18. $(\delta{=}1, h{\geq}1, k{\geq}2, n{\geq}hk)$

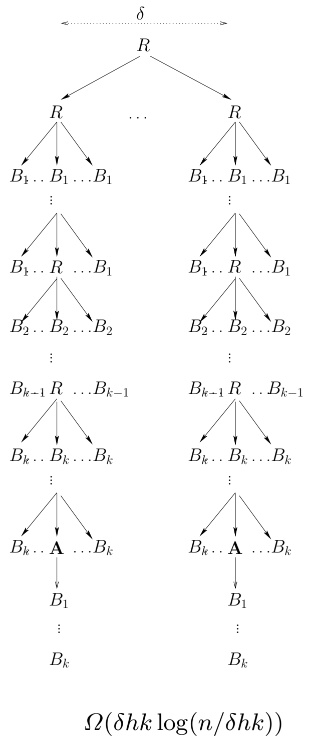

Fig. 19. $(\delta{\geq}1, h{\geq}1, k{\geq}2, n{\geq}\delta hk)$

Next we relax the recursivity h of the document: this is the maximal height of an index-tree. We consider the query $Q = A//B$ of length 2; on documents of size $n + 2$, containing 1 node of type R, 1 node of type A, and n nodes of type B, such that the root and the B nodes form a tree with a unique trunk of B nodes, and such that the A node is parent of a B-node (see Fig. 17). The alternation of this kind of instance is always $\delta = 1$, as guessing the positions of the B-node child of an A-node is sufficient to certify the unique match. An adversary strategy can force any deterministic algorithm to perform $\Omega(h \log n/h)$ comparisons on such instances, by hiding at each level the positions of the B-nodes which are parents, and hiding in the whole tree the A-node. Hence the lower bound, for the instances such that $(\delta = 1, h \geq 1, k = 2, n \geq 2h)$.

Then we relax k, the size of the descendant tree query. We consider the descendant tree query

$$Q = A//B_1//B_2//\ldots//B_k,$$

of length $k + 1$; on documents of size $n + k + 2$, containing k nodes of type R, 1 node of type A, and n/k nodes of type B_i for all $i \in \{1, \ldots, k\}$; such that it can be constructed recursively from $i = 1$ to $i = k$ from the construction

$\mathcal{T}$ for $k = 1$, by replacing the A-node at each step by a copy of $\mathcal{T}$ where each node of type B is replaced by a node of type B_i (see Fig. 18). Each root-to-leaf path contains no more than h nodes of each type; the alternation of each instance is $\delta = 1$ because guessing non-deterministically the unique descendant of A in each index-tree is sufficient to certify the only match of this instance. As before, an adversary can make any deterministic algorithm perform $\Omega(hk \log(n/hk))$ comparisons to check the entire match, hence the lower bound, for the instances such that $(\delta = 1, h \geq 1, k \geq 2, n \geq hk)$.

Finally we relax the difficulty δ of the instances. We consider instances of difficulty δ formed by the same query $Q = A//B_1//B_2//\ldots//B_k$, of length $k+1$; on documents formed by δ documents from the previous case (see Fig. 19). As previously, each root-to-leaf path contains no more than h nodes of each type. The alternation of the instance is exactly δ, as there are δ matches, and guessing the positions of the δ descendants of A-nodes in each index-tree is sufficient to certify the result. Each sub-instance is independent, and an adversary can force any deterministic algorithm to perform $\Omega(\delta hk \log(n/\delta hk))$ comparisons, hence the general lower bound, for the instances such that $(\delta \geq 1, h \geq 1, k = 2, n \geq \delta(hk))$. □

4.5 Complexity of the algorithm in the Hierarchical Memory Model

The complexity in the comparison model is of theoretical interest, but the number of I/O transfers in a hierarchical memory is a better estimate of the real performance of the algorithm. We prove a similar upper bound on the number of cache transfers performed by our algorithm. It is conjectured to be optimal, at the condition that the algorithm to perform doubling searches in exponential binary trees itself is optimal.

Theorem 4 (Complexity in the Hierarchical Memory Model). *Our algorithm performs* $O\left(\delta hk\left(\log_B(1+n/\delta hk) + \log^*(1+n/\delta hk)\right)\right)$ *memory transfers on an instance of alternation* δ*, composed of a descendant tree query of* k *nodes, concerning* n *nodes, of a document of recursivity* h*, between any pair of successive levels of the memory hierarchy such that the slowest one is loaded in the fastest level by blocks of size* B.

Proof. Using the same argument as the proof of Theorem 2, in conjunction with the memory transfer complexity of doubling search, from Bender *et al.* [7]. Extra care must be taken with the function $\log^*(x)$, originally defined on integers as the reverse of $n \rightarrow 2^{\hat{}\hat{}}n$ (such that $\log^*(2^{\hat{}\hat{}}n) = n$). To be used on numbers such as $n/\delta hk$, it must be extended to real numbers.

If extended as a step function $\log^*(x) : \mathbb{R} \rightarrow \mathbb{N}$, the function is not concave. But it can be extended to a continuous and derivable function on real numbers instead, $\log^*(x) : \mathbb{R} \rightarrow \mathbb{R}$ which is concave, hence the final result. Note that the difference between the values of both extensions is always smaller than 1: for the order of the complexity, it does not matter which extension is considered. □

5 Conclusion

In this paper we showed how the structure of an index-tree is adapted to the comparison model and to the hierarchical memory models, in order to answer XPath location steps along the `descendant` axis. We gave an holistic algorithm using index-trees, which has smaller memory requirements than algorithms based on a "set-at-a-time" approach. We introduced a measure of the difficulty of the instances, we proved tight bounds on the asymptotic complexity of the problem in the comparison model, and a reasonable upper bound on the number of I/O transfers.

This is important for several reasons. On one hand, it breaks the tradition of analysis in the stream model, inherited from relational databases, and prove the inefficiency of this model. This is one more argument in favor of native and mixed implementations of XML query systems, which can take advantage of both relational and semi-structural systems. On the other hand, the worst-case logarithmic complexity is a first milestone on the way to real scalability of the treatment of XML documents, and our techniques improve on previous efforts in this direction. The *index-tree* is an improvement on the previous use of B-trees, used in practical implementations to encode the list of labels in traditional indexes. B-trees are a reasonable choice for the indexes of non-recursive node types, but they are not efficient for recursive node types. The doubling search algorithm is an improvement on the fast-forwarding [22] technique which has been seen to perform well in practice, as compared to binary search and sequential access algorithms. Scalability is currently the most important issues of the search in XML documents: with the development of XHTML and of multiple exchange formats, the volume of data to index is increasing always faster, and algorithms of complexity linear or worse are just not acceptable.

In the future we plan to generalize this technique to other axes, and in particular to the `child` axis. Even with labels augmented with some information about the level of the corresponding node (as done in similar studies [19, 10]) the same holistic approach cannot be applied to solve location steps along the child axis with index-trees, because labels cannot be disabled solely based on their preorder position during a child axis location step. Consider for instance the document of Figure 20: the `figure` child of the first `section` node is a *successor* of the `figure` child of the second `section` node. In such a configuration, disabling the label of the first `figure` label for the first `section` node must not disable it for the second `section` node (see Fig. 21). An index structured by level as in Figure 22 or

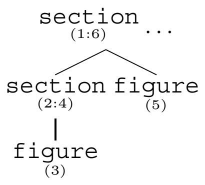

Fig. 20. A document on which holistic algorithm with index-trees cannot solve the query `//section/figure`.

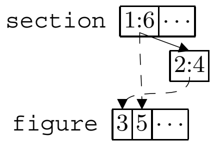

Fig. 21. The corresponding index-trees: a dashed arrow links each `section` node to its `figure` child.

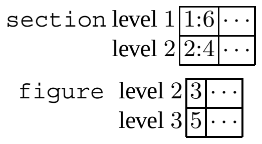

Fig. 22. Levelled indexes would permit the use of doubling search for the `child` and `parent` axes.

by *Path Sequences* [22] shouldn't have this problem, and should permit the use of holistic techniques to solve location steps along `parent-child`, `following-sibling` and all forward axes.

We are planning to work on a compact description of sets of matches, so that we can extend our results to *Twig Pattern Matching* [10]. Imagine a descendant tree without any distinguished node: it is just a *motif*. While a descendant tree has at most n matches in a tree of n nodes, a motif of size k can have up to n^k matches in the same document. But somehow, those matches have many nodes and substructures in common, and the set of all matches can be represented in a concise way, maybe using Ordered Binary Decision Diagrams (OBDDS). This would be precious in many ways, for instance to keep intermediary results short when looking for motifs, or to transmit the result of the query in a compact way and generate the potentially exponential list of matches only at the last moment.

We also consider extending our indexing technique to *rooted oriented graphs*, so that we can solve query without ignoring the links defined in XML documents. XML, in addition to model trees in a normalized way, also permits to define additional *links* between the nodes of a document, via `href` attributes. The documented is then no longer represented as a tree, but as a directed graph, most often acyclic but not always. The naive and brutal approach to treat those links in the structural queries, aside from ignoring them, is to duplicate all linked subtrees. It seems that indexing all the links of the document, and performing a doubling search in this additional index each time a node is failed to be found in the usual index would permit to "follows" those links directly in the index without going back to the document. It would be the efficient way to include links in a search engine, but it is not clear how it can be reflected by our current measure of difficulty: more work is necessary on this topic.

Acknowledgements: This paper was started at the University of British Columbia as a postdoctoral fellow, and finished at the University of Waterloo as an assistant professor. We would like to thank Laks V S Lakshmanan and Ganesh Ramesh for introducing us to the XML world, Mirela Andronescu and many people from UBC for interesting comments on the way to present the problem, and Ashraf Aboulnaga, Ihab Ilyas, Alexandro López-Ortiz and Ian Munro for advice on the presentation of the results.

References

1. S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 547–556. Society for Industrial and Applied Mathematics, 2001.
2. S. Al-Khalifa, H. V. Jagadish, N. Koudas, and J. M. Patel. Structural joins: A primitive for efficient xml query pattern matching.
3. S. Alstrup and T. Rauhe. Improved labeling scheme for ancestor queries. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 947–953. Society for Industrial and Applied Mathematics, 2002.
4. J. Barbay. Optimality of randomized algorithms for the intersection problem. In A. Albrecht, editor, *Lecture Notes in Computer Science*, volume 2827 / 2003, pages 26–38. LNCS, Springer-Verlag Heidelberg, November 2003. 3-540-20103-3.
5. J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proceedings of the thirteenth ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 390–399. ACM-SIAM, ACM, January 2002.
6. J. Barbay and C. Kenyon. Deterministic algorithm for the t-threshold set problem. In H. O. Toshihide Ibaraki, Noki Katoh, editor, *Lecture Notes in Computer Science*, pages 575–584. Springer-Verlag, 2003. Proceedings of the 14th Annual International Symposium on Algorithms And Computation (ISAAC).
7. M. A. Bender, R. Cole, and R. Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 195–207. Springer-Verlag, 2002.
8. J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information processing letters*, 5(3):82–87, 1976.
9. A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Siméon. Xml path language (xpath) 2.0. Technical report, W3C Working Draft, November 2003. http://www.w3.org/TR/xpath20/.
10. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321. ACM Press, 2002.
11. P. Buneman, M. Grohe, and C. Koch. Path queries on compressed xml. In *VLDB*, 2003.
12. D. Z. Chen. Cse 413 - analysis of algorithms - fall. Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556 USA, 2003.
13. B. Choi, M. Mahoui, and D. Wood. On the optimality of holistic algorithms for twig queries. In V. Maríк, W. Retschitzegger, and O. Stepánková, editors, *DEXA*, volume 2736 of *Lecture Notes in Computer Science*, pages 28–37. Springer, 2003.
14. E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the 11^{th} ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752, 2000.
15. E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments, Lecture Notes in Computer Science*, pages 5–6, Washington DC, January 2001.
16. V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992.
17. M. Fernandez, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing xquery 1.0: The galax experience. In *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases*, pages 1077–1080. Morgan Kaufmann, 2003.
18. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285. IEEE Computer Society, 1999.
19. T. Grust. Accelerating xpath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120. ACM Press, 2002.
20. H. Jagadish, S. Al-Khalifa, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, and Y. Wu. Timber: A native xml database. *VLDB*, 11(4):274–291, 2002.
21. H. Jiang, H. Lu, W. Wang, and B. Ooi. Xr-tree: Indexing xml data for efficient structural join, 2003.
22. I. Manolescu, A. Arion, A. Bonifati, and A. Pugliese. Path sequence-based xml query processing. submitted for publication.
23. K. Mehlhorn. Sorting presorted files. In Springer, editor, *Proceedings of the 4th GI-Conference on Theoretical Computer Science*, volume 67 of *Lecture Notes in Computer Science*, pages 199–212, 1979.
24. J.-K. Min, M.-J. Park, and C.-W. Chung. Xpress: a queriable compression for xml data. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 122–133. ACM Press, 2003.
25. O. Petersson and A. Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59:153–179, 1995.
26. J.-C. Raoult and J. Vuillemin. Optimal unbounded search strategies. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 512–530. Springer-Verlag, 1980.
27. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *The VLDB Journal*, pages 302–314, 1999.
28. S. S. Skiena. *The Algorithm Design Manual*. TELOS, State University of New York, Stony Brook, NY, 1997.
29. S. S. Skienna. The algorithm design manual: Hypertext edition. CDROM, 1997.
30. F. Yergeau, T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (xml) 1.0 (third edition). Technical report, W3C Recommendation, February 2004. http://www.w3.org/TR/2004/REC-xml-20040204/.