Descendants, Ancestors, Children and Parent: A Set-Based Approach to Efficiently Address XPath Primitives

Nicola Ferro and Gianmaria Silvello

Department of Information Engineering, University of Padua.
Via Gradenigo 6/B, 35131, Padova, Italy.

Abstract

XML is a pervasive technology for representing and accessing semi-structured data. XPath is the standard language for navigational queries on XML documents and there is a growing demand for its efficient processing.

In order to increase the efficiency in executing four navigational XML query primitives, namely descendants, ancestors, children and parent, we introduce a new paradigm where traditional approaches based on the efficient traversing of nodes and edges to reconstruct the requested subtrees are replaced by a brand new one based on basic set operations which allow us to directly return the desired subtree, avoiding to create it passing through nodes and edges.

Our solution stems from the *NEsted SeTs for Object hieRarchies (NESTOR)* formal model, which makes use of set-inclusion relations for representing and providing access to hierarchical data. We define in-memory efficient data structures to implement NESTOR, we develop algorithms to perform the descendants, ancestors, children and parent query primitives and we study their computational complexity.

We conduct an extensive experimental evaluation by using several datasets: digital archives (EAD collections), INEX 2009 Wikipedia collection, and two widely-used synthetic datasets (XMark and XGen). We show that NESTOR-based data structures and query primitives consistently outperform state-of-the-art solutions for XPath processing at execution time and they are competitive in terms of both memory occupation and pre-processing time.

Keywords:
In-memory XPath processing, NESTOR, set-based data models, data structures

1. Introduction

The *eXtensible Markup Language (XML)* [59, 60] is the standard technology for semi-structured data representation, processing, and exchange and it has been widely used and studied in several fields of computer science, such as databases, information retrieval, digital libraries, and the Web.

An XML document is a hierarchy which contains elements nested one inside another and it is naturally modeled as a tree, where elements are nodes and parent-child relations are edges among them. When it comes to process and access XML, fundamental operations rely on the retrieval of a subset of the XML nodes or the data contained in them by satisfying path constraints which are typically expressed in the *XML Path Language (XPath)* [56, 61].

Given the performance demands of XML processing and the complexity of XPath [42], efficient path queries which involve navigation and subtree reconstruction – i.e., the backbone of XPath –

are required in order to speed-up the access to XML data and to develop efficient end-services. Several solutions have been proposed over time to improve efficiency in processing XML path queries and all of them resort to focus on navigational aspects in some sense: use of secondary indexes to speed-up tree navigation and node selection [11, 41, 64]; node labeling techniques to avoid expensive joins and recursion arising from navigation in the tree representation produced by shredding XML into relation tables [2, 15, 45]; and, alternative methods to parse and navigate in-memory DOM trees [63].

In this paper we propose a paradigm shift for XPath querying by departing from the above mentioned navigational-like approaches and introducing a brand new one, relying on basic set operations. Instead of using edges between nodes or adjacency matrices for representing a tree, we represent a hierarchy as a family of nested sets where the inclusion relationship among sets allows us to express parent/child relations and each set contains the elements belonging to a specific sub-hierarchy. In this way, rather than navigating in a tree and reconstructing sub-hierarchies by traversing nodes and edges, we answer queries by serving the correct subset(s) which already contain all the requested elements (sub-hierarchy) just in one shot or may request minimal intersection/union operations to obtain the desired elements, thus avoiding the need to collect them one-by-one as it happens in the other approaches. This method provides a sizeable improvement in the time requested for answering a navigational query and it is competitive in terms of space occupation and pre-processing time.

More in detail, the proposed solution is based on the *NEsted SeTs for Object hieRarchies (NESTOR)* formal model [21] which is an alternative way, based on the notion of set-inclusion, for representing and dealing with hierarchical data, as XML is. Since XPath supports a number of powerful modalities and many applications do not need to use the full language but exploit only some fragments [5, 6, 27, 28], we focused this work only on those XPath fragments which, according to [5]: (i) support both *downward* and *upward* navigation; (ii) are *recursive*, thus allowing navigation also along the ancestor and descendant axes and not only parent and child axes; and (iii) are *non-qualified*, i.e. without predicates testing properties of another expression. Therefore, we focus on efficient in-memory execution of four kinds of navigational queries over hierarchical data – *descendants*, *ancestors*, *children*, and *parent* of a given element – which represent a basic means for accessing and retrieving data from XML.

The original contributions of the paper consist of the in-memory data structures and algorithms needed to instantiate the NESTOR formal model and perform the navigational queries listed above as well as a thorough experimentation against state-of-the-art solutions.

We present three alternative in-memory dictionary-based data structures instantiating the NESTOR model: *Direct Data Structure (DDS)*, *Inverse Data Structure (IDS)*, and *Hybrid Data Structure (HDS)*. For each query primitive we have two different modalities: the *set-wise modality* where we access the structure of the XML tree and the *element-wise modality* where we access the content of the XML tree. These data structures are defined with no assumption on document characteristics and underlying physical storage and they could be employed by any existing solutions for speeding-up XPath primitives execution. For each data structure and modality, we define algorithms for performing the *descendants*, *ancestors*, *children*, and *parent* operations and we study their computational complexity.

We compare the four target query primitives against state-of-the-art in-memory implementations of XPath – three java-based solutions (i.e., Xalan, Jaxen and JXPath) based on in-memory DOM navigation and a highly-efficient native XML database management system based on node labeling (i.e., BaseX)– in order to assess the benefits in terms of faster execution times. For experimentation, we used three different datasets adopted in the digital libraries, information retrieval and database fields respectively:

- **digital archives**: the main characteristic of archives lies in their *hierarchical structure* used to retain the *context* of the archival records. *Encoded Archival Description (EAD)* [48] is an XML-based representation of archives used to provide access to archival data. We have chosen digital archives since they are a challenging domain within digital libraries, which requires the use of all the four target query primitives and consist of deeply nested XML files.
- **collaborative knowledge**: Wikipedia is a mass collaboration effort for the creation and spreading of knowledge [19]. The *INitiative for the Evaluation of XML Retrieval (INEX)* [3, 22] prepared in 2009 the INEX Wikipedia Collection [50], an XML-ified and semantically-enriched version of the English Wikipedia. The 2009 INEX Wikipedia collection has been considerably exploited to investigate several aspects of XML retrieval, including mixing content-oriented and structure-oriented queries as well as focusing on efficiency [4, 23, 51]. This domain only focuses on the descendants query primitive.
- **synthetic data**: since we have a focus on efficiency, synthetic data allow us to investigate how the proposed data structures and query primitives behave when accessing XML files several orders of magnitude larger than those typically found in the two previous cases and XML trees with increasing complexity in terms of depth and the average and maximum number of children of a node, i.e. the node fan-out. We use two synthetic datasets widely adopted in the database field: XMark and XGen.

The experimental findings show that the NESTOR-based data structures and query primitives consistently outperform state-of-the-art XPath solutions at query time and are competitive also from the pre-processing time and main memory occupation viewpoints. From the results achieved the set-based approach proves to be highly efficient and can represent a valid alternative to tree navigation for fast XML querying.

NESTOR data structures and query primitives, as well as the code for conducting the experiments, have been implemented in Java and, to ease reproducibility of the results, they are available as open source at the following address: `http://nestor.dei.unipd.it/`.

The rest of the paper is organized as follows: Section 2 provides relevant background information; Section 3 describes the NESTOR data structures and the realization of the set-based query primitives; Section 4 discusses the experimental setup while Sections 5 and 6 present the experimental outcomes. Finally, Section 7 draws some conclusions and provides an outlook for future work.

2. Background

2.1. XPath Processing

XPath is a language for addressing parts of an XML document; it provides basic facilities for manipulation of several data types (e.g. strings, numbers and booleans) and adopts a path notation for navigating through the hierarchical structure of an XML document [56]. XPath forms an essential part of XPointer [58], XQuery [62] and *XSL Transformations (XSLT)* [57]. Consequently, as an XQuery or an XSLT stylesheet usually contains several XPath queries, "XPath queries are also probably the most common form of queries on XML" [28].

XPath models an XML document as a tree of nodes and its primary construct is the expression which is evaluated by an "XPath engine" to yield an object that can be a node-set, a boolean, a number or a string. One of the main kinds of expressions is the so-called "location path" which selects a set of nodes relative to a given node (i.e. *context node*); the output of evaluating such an expression is the node-set containing the nodes selected by the location path. Each part of an XPath expression (i.e. location step) can be composed of three parts: (i) an axis, which specifies the tree relationship between the nodes selected by the location step and the context node; (ii) a node test, which specifies the node type and expanded-name of the nodes selected by the location step; and (iii) zero or more predicates that can further refine the set of nodes selected by the location step.

2.1.1. Secondary Memory XPath Processing

There are four main approaches to XPath processing in secondary memory: (i) mapping to the relational model also known as XML shredding [18, 43, 49]; (ii) node labeling [2, 15, 45]; (iii) XML summary-based techniques [1, 52]; (iv) mapping from XPath/XQuery to relational queries.

The *mapping to the relational model* approach flattens out the hierarchical structure by shredding XML elements and attributes into relational tables and columns. This approach has some disadvantages due to the number of columns and tables which rapidly increases as the complexity of the XML files increases. Moreover, the fragmentation of the XML files may require complex and inefficient relational joins to reconstruct the needed information [44].

Node labeling techniques require the XML to be shred into a relational database, but they improve on the previous case by pre-computing encodings that exploit the arithmetic properties of the encodings themselves in order to avoid complex relational joins and recursion. However, this increase in performance comes at the cost of a not negligible time to compute the encodings and an additional amount of memory to store them. A relevant example of a commercial native XML database adopting this solution is eXist-db[1] which adopts a variation of the encoding schema by [39].

XML summary-based techniques, for each keyword and path expression of interest, pre-compute and score path information, called summaries, and store it. Examples of this approach are TReX [1] and TopX [52, 53].

Mapping from XPath/XQuery to relational queries methods rely on XML shredding as above and map XPath/XQuery requests into relational queries. Pathfinder[2] is a notable solution [31, 32, 35], which was integrated in MonetDB[3] under the name MonetDB/XQuery until 2011. MonetDB/XQuery implemented the XPath accelerator [8, 30, 33], which exploits pre/post order encoding techniques [36, 40] to provide efficient access to XML.

All these approaches are not directly comparable to NESTOR, since they do not work in main memory.

[1]`http://exist-db.org/`
[2]`http://db.inf.uni-tuebingen.de/projects/Pathfinder.html`
[3]`http://www.monetdb.org/XQuery/`

2.1.2. Main Memory XPath Processing

Xalan[4], Jaxen[5] and JXpath[6] are three state-of-the-art libraries which parse an XML file, create an in-memory *Document Object Model (DOM)* tree [55], and evaluate XPath expressions over it.

BaseX[7] is a state-of-the-art Java-based native XML database, which offers both in-memory and secondary-memory storage. Its main characteristic is the adoption of an ad-hoc indexing schema inherited from the pre/post order encoding proposed by [30] and implemented also in the MonetDB/XQuery database [8, 33]. The main difference between BaseX and MonetDB/XQuery resides in the optimization introduced by BaseX for the parent and ancestor axes, which grants a constant execution time access for them. Furthermore, BaseX uses compact memory structures and performs compression based on dynamic recognition of data types which, for instance, allows it to determine if a text node is a string or an integer to enable compact storage of the element. [29] compared BaseX against MonetDB/XQuery and eXist-db and found that in most cases BaseX was outperforming them in terms of both indexing and query evaluation time.

For all these reasons, Xalan, Jaxen, JXpath, and BaseX are very well-suited as a comparison for the NESTOR-based data structures we present in this work.

2.2. The NESTOR Model

The NESTOR model is defined by two set-based data models: The *Nested Set Model (NS-M)* and the *Inverse Set Data Model (INS-M)*. These models are defined in the context of set theory as a collection of subsets, their properties are formally proved as well as their equivalence to the tree in terms of expressive power [20, 21].

The most intuitive way to understand how these models work is to relate them to the tree. In Figure 1b we can see how a sample XML snippet can be represented through a tree and how this tree can be mapped into the NS-M and the INS-M. The XML elements are represented by nodes in the tree and by sets in the NS-M and the INS-M; the text data within each XML node are represented as lists of elements within each tree node and as elements belonging to sets in the NS-M and the INS-M.

In the NS-M each node of the tree is mapped into a set, where child nodes become *proper subsets* of the set created from the parent node. Every set is subset of at least of one set; the set corresponding to the tree root is the only set without any supersets and every set in the hierarchy is subset of the root set. The external nodes are sets with no subsets. The tree structure is maintained thanks to the nested organization and the relationships between the sets are expressed by the set inclusion order. Even the disjunction between two sets brings information; indeed, the disjunction of two sets means that these belong to two different branches of the same tree.

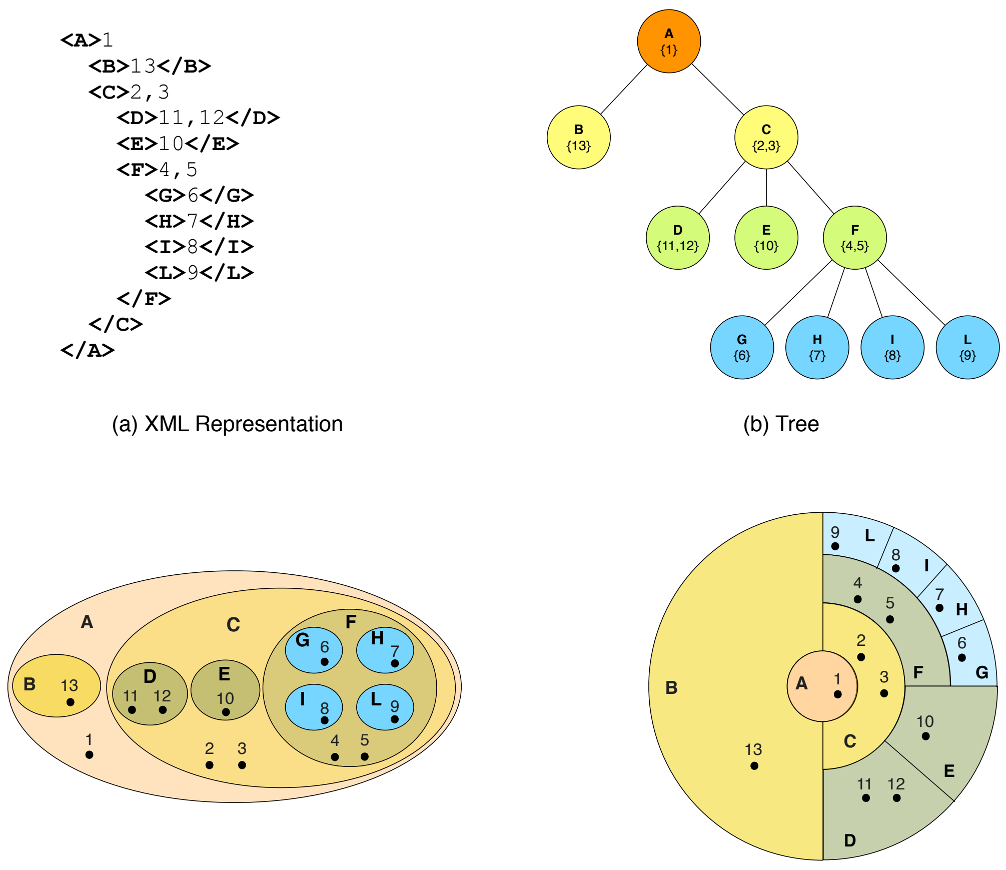
The second data model is the INS-M and we can see that each node of the tree is mapped into a set, where each parent node becomes a subset of the sets created from its children. The set created from the tree's root is the only set with no subsets and the root set is a proper subset of all the sets in the hierarchy. The leaves are the sets with no supersets and they are sets containing all the sets created from the nodes composing tree path from a leaf to the root. An important aspect of INS-M is that the intersection of every couple of sets obtained from two nodes is always a set representing a node in the tree. The intersection of all the sets in the INS-M is the set mapped from the root of the tree.

[4]`http://xml.apache.org/xalan-j/`

[5]`http://jaxen.codehaus.org/`

[6]`http://commons.apache.org/proper/commons-jxpath/`

[7]`http://www.basex.org/`



```
<A>1
  <B>13</B>
  <C>2,3
    <D>11,12</D>
    <E>10</E>
    <F>4,5
      <G>6</G>
      <H>7</H>
      <I>8</I>
      <L>9</L>
    </F>
  </C>
</A>
```

(a) XML Representation

(b) Tree

(c) Nested Set Model

(d) Inverse Nested Set Model

Figure 1: (a) A sample XML representation; (b) Tree representation of the XML; (c) NS-M representation; (d) INS-M representation.

There are some apparent similarities between the nested sets approach taken by NESTOR and some encodings used by relational databases. Indeed, the nested sets graphical representation we use for the *Nested Set Model (NS-M)* in Figure 1a has been used also by [10] as an intuitive means to explain how integer interval encodings work. However, while for integer interval encodings this is just a way of “teaching” them, in the case of NESTOR the inclusion relationship among sets is a constituent element of the model.

3. Data Structures and Primitives for Fast XML Access

In this section we propose three general-purpose in-memory data structures for putting NESTOR into action by providing general implementations of the set-based data models.

3.1. Data Structures

Let us consider a collection of subsets $\mathcal{C}$ which can be defined according to the NS-M as well as the INS-M – refer to Figure 1 for a graphical example. The data structures for producing $\mathcal{C}$ have to take into account the structural and the content components of such a collection of subsets. From the structural point-of-view, the information that has to be stored regards the

A	[1,2,3,4,5,6,7,8,9,10,11,12,13]	A	[B,C]	A	[]
B	[13]	B	[]	B	[A]
C	[2,3,4,5,6,7,8,9,10,11,12]	C	[D,E,F]	C	[A]
D	[11,12]	D	[]	D	[C,A]
E	[10]	E	[]	E	[C,A]
F	[4,5,6,7,8,9]	F	[G,H,I,L]	F	[C,A]
G	[6]	G	[]	G	[F,C,A]
H	[7]	H	[]	H	[F,C,A]
I	[8]	I	[]	I	[F,C,A]
L	[9]	L	[]	L	[F,C,A]
	MD		**DD**		**SD**

(a) The dictionary-based Direct Data Structure (DDS)

A	[1]	A	[]	A	[B,C,D,E,F,G,H,I,L]
B	[1,13]	B	[A]	B	[]
C	[1,2,3]	C	[A]	C	[D,E,F,G,H,I,L]
D	[1,2,3,11,12]	D	[C]	D	[]
E	[1,2,3,10]	E	[C]	E	[]
F	[1,2,3,4,5]	F	[C]	F	[G,H,I,L]
G	[1,2,3,4,5,6]	G	[F]	G	[]
H	[1,2,3,4,5,7]	H	[F]	H	[]
I	[1,2,3,4,5,8]	I	[F]	I	[]
L	[1,2,3,4,5,9]	L	[F]	L	[]
	MD		**DD**		**SD**

(b) The dictionary-based Inverse Data Structure (IDS)

A	[1]	A	[B,C]	A	[]
B	[13]	B	[]	B	[A]
C	[2,3]	C	[D,E,F]	C	[A]
D	[11,12]	D	[]	D	[C]
E	[10]	E	[]	E	[C]
F	[4,5]	F	[G,H,I,L]	F	[C]
G	[6]	G	[]	G	[F]
H	[7]	H	[]	H	[F]
I	[8]	I	[]	I	[F]
L	[9]	L	[]	L	[F]
	MD		**DD**		**SD**

(c) The dictionary-based Hybrid Data Structure (HDS)

Figure 2: NESTOR data structure instances of the tree shown in Figure 1.

inclusion dependencies between the sets; whereas, from the content point-of-view, we need to store the materialization of the sets (i.e. the elements belonging to each set).

In the following, without any loss of generality we assume that: (i) every element and set in $\mathcal{C}$ is identified by a unique handle, which is a label allowing us to identify and facilitate referencing them. Each set is associated with a capital letter and each element is associated with an integer number; (ii) the materialized sets are stored as sorted arrays of integers; (iii) for each set H in $\mathcal{C}$ we store two arrays of values, one containing the information about the supersets of H, and the other for the direct subsets[8] of H. These arrays are ordered by decreasing length; (iv) the arrays are grouped together in three different dictionaries.

With this approach every element in an array can be accessed in constant time (i.e. $O(1)$). The dictionaries are implemented by means of hash tables which guarantees $O(1)$ worst-case time

[8]A direct subset of a set $A \in \mathcal{C}$ is defined as $B \in \mathcal{C}$ such that $B \subset A$ and $\nexists C \in \mathcal{C} \mid B \subset C \subset A$.

Table 1: Space occupation of the data structures in a worst-case scenario where we have m sets structured as a chain and n elements.

	DDS	**IDS**	**HDS**
Space	O($m^2 + n^2$)	O($m^2 + n^2$)	O($m + n$)

for lookup [26].

For a collection of subsets $\mathcal{C}$ we consider the following three main dictionaries:

- **Materialized Dictionary** (`MD`), containing the materialization of the sets in $\mathcal{C}$.
- **Direct-Subsets Dictionary** (`DD`), containing the direct subsets of each set in $\mathcal{C}$.
- **Supersets Dictionary** (`SD`), containing the supersets of each set in $\mathcal{C}$.

These three dictionaries are employed in all the three proposed data structures – i.e. *Direct Data Structure (DDS)*, *Inverse Data Structure (IDS)* and *Hybrid Data Structure (HDS)*. DDS is a structure built around the constraints defined by the NS-M as depicted in Figure 2a. If we consider the tree shown in Figure 1a and modeled with the NS-M reported in Figure 1b, we can see that the materialized sets (`MD`) report the integer values corresponding to all the elements belonging to each set; we know that in the NS-M the set corresponding to the root of the tree contains all the elements of the tree. Indeed, we can see in Figure 2a that `A` contains all the elements in the collection. Furthermore, for each set, `DD` contains all its direct subsets – e.g. set `C` contains the sets `D`, `E`, `F` which are its direct subsets as shown in Figure 1b. `SD` contains the supersets of each set – e.g. `D` contains the sets `C`, `A`.

IDS is a structure built around the constraints of INS-M as reported in Figure 2b. In this case we can see that the materialized sets `MD` contain the elements belonging to the sets. As we can see in the DDS the set cardinality in `MD` decreases while going from the top – i.e. set `A` – to the bottom – i.e. sets `G`, `H`, `I`, `L` – whereas in IDS the cardinality increases. Furthermore, for each entry in `DD` there is just one set; indeed, in the INS-M each set has at most one direct subset with the sole exception of the set representing the root of the tree which has no subsets. `SD` reports for each set all its supersets – e.g. `F` contains `G`, `H`, `I`, `L` which are its supersets as shown in Figure 1c.

HDS can be seen as a mixture between DDS and IDS; indeed, its `DD` corresponds to the `DD` dictionary of DDS and its `SD` corresponds to the `DD` of IDS. Each set in `MD` is the result of the set difference between a set and its direct superset in the `MD` of IDS – e.g. in the `MD` of IDS the set `A` = [`1`] is the direct superset of `C` = [`1`, `2`, `3`], thus in the `MD` of HDS it is `C` = [`2`, `3`]. This is the major difference between HDS and the other two data structures and its aim is to reduce the number of set intersections in the element-wise primitives as explained below. HDS has the positive effect of reducing the space required by `MD` because for each set it stores only the elements that exclusively belongs to that set; by contrast, DDS stores all the elements belonging to a set and all its subsets and IDS stores all the elements belonging to a set and its supersets. HDS represents a trade-off between the in-memory space occupied by the structure and the time required for executing the primitives. From the set-wise point-of-view HDS takes the `DD` from the DDS and its `SD` is the `DD` taken from the IDS; in both cases these are the dictionaries occupying less memory space. Furthermore, for DDS and IDS, some information stored in `DD` and `SD` can be also derived from those in `MD`, whereas this cannot be done for the HDS which, in this respect, is less redundant than DDS and IDS.

Now we can determine the worst-case space occupation of these data structures reported in in Table 1 for the worst-case scenario. If we consider a collection of subsets composed of m sets and n elements, the worst-case scenario – for all the structures we presented – happens when the collection is structured as a chain where $m = n$.

Therefore, let us consider a DDS composed by n elements belonging to m sets with a chain structure say, $\{A_1, \dots, A_m\}$ such that $A_m \subset A_{m-1} \subset \dots \subset A_1$ and $|A_m| < |A_{m-1}| < \dots < |A_1|$. In this case, $|A_1| = n$, $|A_2| = n - 1, \dots, |A_m| = 1$, thus we need to store $n(n+1)/2$ elements – i.e. $O(n^2)$ size – for the materialized sets in the `MD` dictionary.

Furthermore, we need to store $2m$ arrays of sets, where m sets contain the information about the direct subsets and m contains the information about the supersets. The arrays of direct subsets occupy $O(m)$ size because every set has at most one direct set when we are considering a chain structure; the arrays of supersets occupy $O(m^2)$ size because A_m has zero supersets, A_{m-1} has one superset, so on and so forth until A_1 which has m supersets. Thus, we need to store $O(m(m+1)/2) = O(m^2)$ sets. This means that in the worst-case scenario we need to store $O(n^2)$ elements and $O(m + m^2) = O(m^2)$ sets.

The space occupation analysis for the IDS is equal to the DDS one, since in the worst-case scenario these two structures are equivalent. HDS inherits the `DS` dictionary from DDS which stores $O(m)$ sets in the worst-case and `DD` dictionary from IDS which stores $O(m)$ sets in the worst-case, thus it needs to store $O(m + m) = O(m)$ sets. Finally, each array in the `MD` dictionary contains only the elements that belong to a given sets, thus it needs to store $O(n)$ elements.

However, by means of the experimental analysis we will show how both real-world and synthetic datasets are far to be structured as chains and that the space required by the dictionary-based data structure is linear in the number of sets and elements.

3.2. NESTOR Primitives

When we deal with a collection of sets defined by the NESTOR model, we can distinguish between *set-wise* and *element-wise* primitives. As above, let us consider a general collection of sets $\mathcal{C}$ with a total number of $m \in \mathbb{N}$ sets, $n \in \mathbb{N}$ elements and where H is a set in the collection; then in the NESTOR model we define five set-wise primitives, and six element-wise primitives. A formal proof of the complexity of the different primitives is reported in Appendix A, along with their pseudo-code. In the following we add the suffix DDS, IDS or HDS to the name of the primitives when we refer to a specific data structure implementation of that primitive – e.g. the descendant primitive is indicated as Descendants-DDS when we refer to its DDS implementation, as Descendants-IDS when we refer to its IDS implementation and as Descendants-HDS when we refer to its HDS implementation.

Set-Wise Primitives

In Table 2 we present the references to the algorithms implementing the set-wise primitives in the three data structures, along with their computational complexities.

The worst-case scenario for all the set-wise primitives is represented by a collection of sets structured as a chain – i.e. an XML file where each node has one and only one child. Let us consider the DDS first. For the Descendants-DDS(H) primitive the worst-case input set H is the set corresponding to the root of the XML file because the operation has to return all the sets in the given collection. The algorithm starts by asking for the children of H and then it iterates over all the children of the children of H and so on and so forth until the last set in the collection is processed.

Table 2: Set-wise primitives and their computational complexities where m is the total number of sets in the collection of subsets.

Set-wise Primitive Algorithms						
	DDS		**IDS**		**HDS**	
	Reference	Cost	Reference	Cost	Reference	Cost
Descendants(H)	Alg. 6	$O(m)$	Alg. 8	$O(1)$	Alg. 6	$O(m)$
Ancestors(H)	Alg. 4	$O(1)$	Alg. 10	$O(m)$	Alg. 13	$O(m)$
Children(H)	Alg. 5	$O(1)$	Alg. 11	$O(m)$	Alg. 5	$O(1)$
Parent(H)	Alg. 7	$O(1)$	Alg. 9	$O(1)$	Alg. 12	$O(1)$

All other set-wise primitives are implemented to run in constant time for any collection of subsets and input set. Indeed, the Ancestors-DDS(H) has to access the supersets dictionary and to return the entry corresponding to set H. As anticipated in the Section 1, while with traditional navigational approaches this operation calls for recursive algorithms, the DDS carries out this operation avoiding any recursion and is a clear example of the paradigm shift realized by NESTOR.

Similarly, Children-DDS(H) has to return the entry for set H in the direct-subset dictionary. The Parent-DDS primitive can be implemented in $O(1)$ time by exploiting the fact that every set in a collection implemented by the DDS has at most one direct superset, and that the arrays in the superset dictionary are ordered by decreasing cardinality.

The Descendants-IDS(H) primitive is implemented to run in constant time because it returns the entry relative to set H in the supersets dictionary; similarly the Parent-IDS(H) has to return the entry of set H in the direct-subsets dictionary.

The Ancestors-IDS algorithm exploits the fact that each set in the IDS has at most one direct subset [21]. In the worst-case scenario, where the input set H is the lowest set of a chain and all the sets in the given collection are its supersets, we need to iterate over all the sets; despite the linear computational complexity, in the experimental section we will see that in practice this algorithm scales very well because all real-world datasets have a structure which is very different from a chain. The Children-IDS computational complexity depends on the number of supersets of the input set H which in the worst-case scenario corresponds to the cardinality of the input collection.

The HDS inherits the dictionary `DD` from the DDS and for this reason the algorithms implementing Descendants-HDS(H) and Children-HDS(H) are exactly the same as we described for DDS. By contrast, the algorithms implementing Parent-HDS(H) and Ancestors-HDS(H) share worst-case and computational time with the ones described for the IDS. Parent-HDS(H) only has to return the entry for set H in the `SD` dictionary (note that with IDS we had to return the entry in `DD`); Ancestors-HDS(H) works exactly as Ancestors-IDS(H), although to determine all the ancestors of a set it iterates over the Parent-HDS(H) primitive and not over Parent-IDS(H) (cfr. Algorithm 13).

Element-Wise Primitives

In Table 3 we present the references to the algorithms implementing the element-wise primitives in the DDS, IDS and HDS along with their computational complexities.

The Elements(H) primitive is implemented with the same algorithm for the three data structures. It runs in constant time because it returns the entry corresponding to set H in the materialized dictionary. The Descendant(H) primitive is implemented to run in $O(1)$ for the DDS

Table 3: Element-wise primitives and their computational complexities where m is the total number of sets and n is the total number of elements in the collection of subsets.

Element-wise Primitive Algorithms						
	DDS		**IDS**		**HDS**	
	Reference	Cost	Reference	Cost	Reference	Cost
Elements(H)	Alg. 3	$O(1)$	Alg. 3	$O(1)$	Alg. 3	$O(1)$
Descendants(H)	Alg. 14	$O(1)$	Alg. 18	$O(m+n)$	Alg. 23	$O(m+n)$
Ancestors(H)	Alg. 15	$O(m+n)$	Alg. 19	$O(1)$	Alg. 25	$O(m+n)$
Childrens(H)	Alg. 16	$O(n)$	Alg. 20	$O(m+n)$	Alg. 22	$O(n)$
Parent(H)	Alg. 17	$O(n)$	Alg. 21	$O(n)$	Alg. 24	$O(1)$

because it returns the entry of set H in the materialized dictionary. Indeed, the DDS is built in such a way that each set contains all the elements of its descendants; therefore, it is possible to answer this primitive without browsing the collection of sets (without browsing the hierarchy) or without passing through any set-wise primitive. As above, this is another example of the departure of NESTOR from traditional navigational approaches, since it avoids the recursion needed to collect element by element and returns all the requested elements in one-shot instead.

The same idea is exploited for realizing the Ancestor(H) element-wise primitive in the IDS. Indeed, in the IDS, each set contains all the elements of its ancestors; therefore, it is possible to answer this element-wise primitive with a single operation. We can see how these two primitives are independent with respect to their correspondent set-wise primitives. HDS cannot answer either of these two element-wise primitives with a single operation, because for the Descendant(H) primitive it has to iterate over all the subsets of H in the `MD` dictionary and return the elements for each set; for the Ancestor(H) primitive HDS has to do the same, but it iterates on all the ancestors of H rather than the descendants. The worst-case computational complexity for both these primitives is $O(n+m)$.

On the other hand, the Ancestor(H) primitive runs in $O(n+m)$ time for the DDS. It returns all the elements which belong to the sets which are ancestors of H. From the content point-of-view this is the most expensive primitive for the DDS because it has to make several unions between the supersets of H and then subtract from the result set the elements belonging to the subsets of H. The performances of this algorithm are improved by the possibility of exploiting the DDS characteristics to speed up the union and the difference operations.

A similar behavior is found in the Descendant(H) primitive for the IDS, where the algorithm has to join all the subsets of the input set H by removing common elements. From the definition of IDS we know that every couple of sets in IDS shares at least one element; therefore, unions in the IDS are quite expensive and in the worst-case scenario there are many of them. We find the same behavior for the Children-IDS element-wise primitive that indeed has the same computational complexity.

HDS behaves like IDS for the Descendant(H) primitive and like DDS for the Ancestor(H) primitive. This data structure is well-suited for answering the Parent(H) primitive, because it only has to return the entry of set H in the `MD` dictionary and thus it requires constant time also in the worst-case scenario; this is an improvement over both DDS and IDS and it is mainly due to the fact that HDS, in this case, does not require all the sets differences required for DDS and IDS. For the Children(H) primitive HDS requires a computational time linear in the number of elements that the children sets of H contain; we can see that it performs as well as DDS and better than IDS. As we can see HDS is a good trade-off solution between DDS and IDS that are nonetheless optimized for getting the descendants of a set in the former case and the ancestors in the latter.

All the remaining element-wise primitives have a computational complexity which depends only on the number of elements in the collection of subsets. This fact stresses the low correlation between structure and content in the NESTOR model. These algorithms exploit the set-wise primitives which run in constant time and then perform some set-theoretical unions and differences to return the desired set of elements; the complexity of these algorithms is dominated by these set-theoretical operations. It is worthwhile highlighting that the number of set-theoretical operations to be performed is very low when compared to the number of sets in the considered experimental collections of sets.

4. Experimental Setup

NESTOR query primitives on the three proposed in-memory data structures have been implemented in the Java programming language[9]. As anticipated in Section 2, we compare NESTOR against the Xalan 2.7.1, Jaxen 1.1.6, and JXpath 1.3 libraries and the BaseX 7.9 in-memory XML database.

Note that the nodes of an XML file are mapped into sets in NESTOR and the text elements and attributes into elements; for this reason set-wise primitives in NESTOR correspond to node-wise operations in XPath and element-wise primitives to text-wise operations.

We use the following evaluation measures for comparing the different solutions:

- *evaluation time*: XPath query processing is composed by compilation and evaluation. We consider only the time for evaluating the query and not the compilation time;
- *building time*: is the time required for parsing the XML files and creating the corresponding in-memory data structures;
- *space occupation*: is the amount of main memory needed for the created data structures.

Our primary measure is the evaluation time, since the focus of the paper is on efficient processing, but the building time and space occupation are needed to understand the trade-off between performances and resource consumption. We repeat each experiment 100 times and we report average values for the above evaluation measures.

The analysis was conducted by choosing the worst possible input set for each primitive, thus if a query performs well in this case it is guaranteed that it performs in the same way or better in the other cases. As an example the worst input set for the structural descendant query for the DDS is the one corresponding to the root of the XML tree, because in this case the primitive has to return all the sets (nodes) in the given dataset.

We used a six-processor 3.33GHz Intel(R) Xeon(R) machine with 96GB of RAM running Java 1.6. We adopted the Java `System.nanoTime()` method to measure the execution times; its resolution depends on the operating system which in our case is Linux Kernel 2.6.35. We measured the resolution of `System.nanoTime()` by means of the method proposed in [17] with 100 iterations, determining a time resolution of 1.5 μsec. This timer resolution affects the measure of the execution times in the range of $[10^{-4}, 10^{-3}]$ msec.

Appendix B reports all the details and statistics about how the test queries have been defined for the considered datasets.

[9]`http://nestor.dei.unipd.it`

4.1. Datasets

4.1.1. Digital Archives

Archival documents are strongly interlinked and their relationships have to be retained to preserve their informative content and provide understandable and useful information over time [25]. According to the *International Standard for Archival Description (General)* (ISAD(G)) [34], archival description proceeds from general to specific as a consequence of the provenance principle and has to show, for every unit of description (i.e. a record), its relationships and links with other units and to the general fonds[10], taking the form of a tree.

The digital encoding of ISAD(G) is the *Encoded Archival Description (EAD)* [48], which is an XML description of a whole archive, reflects the archival structure, holds relations between entities, retains context and provides the archival content.

XPath processing is central for accessing digital archival descriptions, indeed it is successfully used for searching and selecting portions of EAD files. For example, the *Retrieving EADs More Efficiently (README)* system [65] exploits XPath and XML retrieval techniques within EAD files and the Cheshire3[11] open source XML search engine [38], adopted by the UK Archives Hub, uses XPath expressions in the indexing phase for extracting data from EAD files.

We selected five EAD collections that provide us with real-world archival data:

- **Archive Hub** (AH 2005)[12]: a 2005 snapshot of the archival hub which is a gateway to many of the UK's richest historical archives.
- **International Institute of Social History** (IISG 2005)[13]: a 2005 snapshot of the EAD archive of the social history institute of the Royal Netherlands Academy of Arts and Sciences.
- **Nationaal Archief** (NA 2008)[14]: a 2008 snapshot of the EAD files of the National Archives of the Netherlands.
- **Library of Congress** (LoC 2014)[15]: a 2014 snapshot of the EAD finding aids of the Library of Congress.
- **University of Maryland** (UniMa 2014)[16]: a 2014 snapshot of the EAD finding aids of manuscript and archival collections at the University of Maryland Libraries.

These collections of archival data come from relevant institutions which are also typically ahead in the adoption and promotion of new standards and shared technologies. The archives have also been selected on the basis of their size and heterogeneity, in order to test NESTOR in a wide range of settings; we have also resorted to general hubs (AH2005) capable of aggregating archives from smaller institutions, in order to extend the spectrum of our investigations.

Table 4 reports the total number of EAD files, the max and median number of nodes, the depth, the size and the max fan out for each collection.

[10] A *fonds* is viewed primarily as the conceptual whole that reflects the organic process in which a records creator produces or accumulates series of records.

[11] `http://cheshire3.org/`

[12] `http://archiveshub.ac.uk/`

[13] `http://socialhistory.org/nl`

[14] `http://www.nationaalarchief.nl/`

[15] `http://findingaids.loc.gov/`

[16] `http://digital.lib.umd.edu/`

Table 4: Statistics of the EAD archival collections for files bigger than 10kB.

Collection	Files	Nodes		Depth		Size (KB)		Max Fan Out	
		max	median	max	median	max	median	max	median
AH 2005	233	14,648	158	21	6	760	15	1,332	23
IISG 2005	798	52,213	513	17	9	2,290	34	2,601	21
NA 2008	1681	160,061	880.5	18	9	9,750	58	10,271	34
LoC 2014	2083	188,862	685	18	10	15,510	58	5,000	32
UniMa 2014	662	69,766	711	10	8	2,960	40	6,861	43

The experiments conducted on collections that are so diverse in size and characteristics will show how the advantages in terms of allowed operations and performances of NESTOR over commonly adopted XPath-based solutions are substantial when we deal with large, deep and complex archives as well as with small and shallow ones.

4.2. Collaborative Knowledge

INEX adopts the traditional Cranfield paradigm [14] which make use of experimental collections composed of a corpus of documents (i.e. XML files in the INEX case), a set of topics and a set of relevance judgments stating whether or not a document is relevant to a given topic.

A topic is the expression of a user information need and consists of two parts: the first is focused only on content and best match retrieval, where *Content-Only (CO)* queries are used and effectiveness is the primary concern; the second is focused also on the structure and exact-match retrieval, where *Content-And-Structure (CAS)* queries are used and efficiency is also a concern. CAS queries are expressed in the *Narrowed Extended XPath I (NEXI)* language, which is very similar to XPath but restricted to the exclusive use of the descendant axis and extended for best-match retrieval [54].

The INEX 2009 Wikipedia Collection [23, 50] is particularly challenging from the efficiency point-of-view since "the collection has an irregular structure with many deeply nested paths, which turned out to be challenging for most systems" [4].

An example of an INEX CAS query is the following (i.e. topic 20 of INEX 2009):

```
//article[about(.,IBM)]//sec[about(., computer)]
```

From this topic we can derive the structure query which is: `//article//sec` asking for all the descendants of "sec" elements which are also "article" descendants related to IBM and computers.

We identify three query types that can be derived from INEX topics. All the structural parts of the 115 INEX 2009 topics can be answered by using different combinations of the NESTOR element-wise descendants primitive.

The first query type is called *descendant query*, an example taken from IXEX XPath queries is `//article`. This query asks for all the descendants of the XML element "article". The second query type is called *union descendants query*, an example taken from IXEX XPath queries is `//(sec|header)`; this query introduces the boolean operator "or" indicated with the pipe, asking for all the text elements which belong to either `sec` or `header` and to their descendants. The third query type is called *intersection descendants query*, an example taken from IXEX XPath queries is `//sec//p` which asks for all the elements belonging to the `p` elements and its descendants with the constraint that the `p` elements considered must be descendants of a `sec` element.

The INEX Wikipedia collection is composed of about 3 million files for a total space occupation on disk of 50.1 GB. In addition, there are more than 700 million text elements of which about 102

Table 5: Statistics of the INEX 2009 Wikipedia collection.

Files	Nodes		Depth		Size (KB)		Max Fan Out	
	max	median	max	median	max	median	max	median
2,666,190	69,560	136	85	18	4,384	8.15	5,401	14

Table 6: Statistics of the twelve selected XMark synthetic files.

	Size (MB)	# nodes	depth	max fan-out	average fan-out
XMark-01	0.581	8,518	12	127	3.69
XMark-02	1.182	17,132	12	255	3.70
XMark-03	2.385	33,140	12	510	3.60
XMark-04	4.840	67,902	12	1,020	3.65
XMark-05	9.595	134,831	12	2,040	3.65
XMark-06	18.855	265,975	12	4,080	3.66
XMark-07	38.145	533,750	12	8,160	3.66
XMark-08	76.016	1,066,768	12	16,320	3.66
XMark-08	152.350	2,140,644	12	32,640	3.66
XMark-10	305.191	4,276,108	12	65,280	3.67
XMark-11	610.043	8,554,409	12	130,560	3.67
XMark-12	1,221.750	17,107,471	12	261,120	3.66

million with at least 50 characters as summarized in Table 5. The biggest XML file counts 69,560 nodes, whereas the median is 136 meaning that there is considerable heterogeneity among the files in the collection. This can also be seen by looking at the size of the files: the maximum size is about 4MB, but the median is about 8KB; therefore, the Wikipedia collection is composed of many small XML files. Despite this fact, there are some very deep files – up to depth 85 with median 18 – with a very large maximum fan out – about 5 thousands nodes, but the median is only 8.15.

4.3. Synthetic Data

Tables 6 and 7 report the statistics about the two synthetic datasets we selected for the scalability evaluation of NESTOR. The first dataset – XMark – is composed of 12 XML files generated with a public library available at the following URL: `http://www.xml-benchmark.org/`; the second one (XGen) is composed of ten XML files generated by means of a Java library we share along with the implementation of NESTOR data structure and primitives.

The XMark dataset contains balanced XML documents with a growing size and number of elements modeling an auction website, a typical e-commerce application; the XGen dataset instead contains XML documents with roughly the same number of elements but very different structures – i.e. different depths, number of max and average fan outs. The XML documents in the XMark dataset contain long strings of text, whereas the documents in the XGen dataset contain short randomly generated strings. The 12 XMark files described in Table VI are balanced XML documents; indeed, they all have the same depth and the same average fan-out. They range from 0.5MB to 1.2GB in size and from 8 thousand to 17 million nodes, with a max fan-out ranging from 100 to 261 thousand. With this dataset we can analyze NESTOR data structures and primitives with progressively bigger XML files; in particular, we can see if the ancestor primitives are influenced

Table 7: Statistics of the ten selected XGen synthetic files.

	Size (MB)	**# nodes**	**depth**	**max fan-out**	**average fan-out**
XGen-01	945.106	10,000,043	27	100	50.40
XGen-02	995.277	10,000,025	29	200	100.50
XGen-03	830.503	10,000,184	26	500	250.54
XGen-04	896.776	10,000,509	24	1,000	500.60
XGen-05	719.455	10,000,155	22	2,000	1,002.32
XGen-06	866.558	10,003,624	24	5,000	2,490.94
XGen-07	717.845	10,000,776	18	9,994	4,880.81
XGen-08	599.369	10,011,516	14	19,983	10,195.03
XGen-09	639.106	10,004,482	13	49,866	24,581.04
XGen-10	642.200	10,017,259	13	99,010	49,346.10

more by the depth of the XML document or by its size. At the same time, we analyze the impact of the max and average fan-out on the children primitives and the impact of the number of nodes on the descendants ones. Vice versa the XGen dataset presents XML documents with the same number of nodes (10 million) but a heterogeneously structure; indeed, XML depth ranges from 27 to 13, max fan out from 100 to 100 thousand and the average fan-out from 50 to 50 thousand. Another characteristic of the XGen dataset is that its files have roughly the same number of nodes but a progressively smaller size ranging from about 1GB to 600MB; thanks to this peculiarity we can analyze the impact of file size and of node number on the query operations.

As we can see, both these datasets present much bigger and more complex XML documents than those we found in the two previous domains and they allow us to push NESTOR, XPath-based libraries and BaseX to their limits and to assess their future proof.

5. Building Time and Space Occupation

Figure 3 and Figure 4 show the building time and space occupation for the different datasets under examination.

As general trends for the building time, we see that DOM requires the smallest time and BaseX the largest time in almost all cases while the NESTOR data structures fall in-between. This is somehow expected since DOM simply parses the XML files and represents them in main memory but without any additional structure for improving access to them. By contrast, BaseX, requires the parsing time as DOM does, as well as the time needed to compute the node labels and apply dynamic compression. Similarly, NESTOR requires some additional time to build its data structures on top of the time needed for parsing.

As far as space occupation is concerned, DOM is typically demanding, while NESTOR data structures behave effectively. The picture is more varied in the case of BaseX, since its occupied space basically depends on the size of the node labels which, in turn, is influenced by the depth of the tree and the number of descendants. Furthermore, BaseX applies dynamic compression and so it gains more in the case of large textual elements, e.g. XMark, with respect to small textual elements, e.g. XGen.

In particular, when it comes to the NESTOR data structures, DDS requires more time and space than the others since its dictionaries contain more elements the closer we are to the root

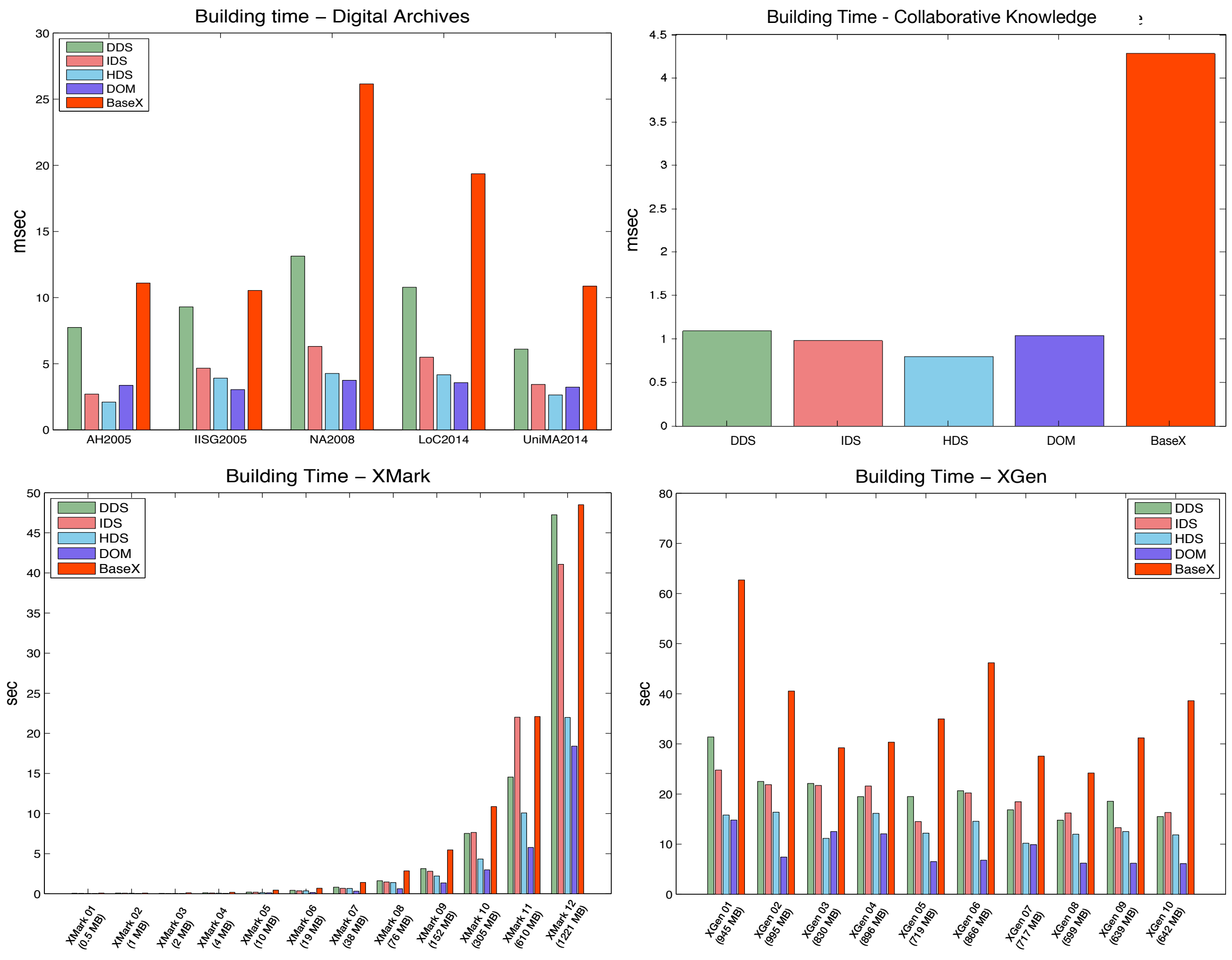


Figure 3: Average building time.

and they progressively reduce the number of elements in each dictionary going downwards. As a consequence, the DDS dictionaries contain on average more elements. Similar considerations can be made for IDS but it is able to better optimize the trade-off between depth and max-fan out of the tree than DDS. By contrast, as pointed out in Section 3, the HDS requires less time and space to be built because it basically stores "local" information about the direct super and sub sets, which corresponds to less elements on average.

6. Evaluation Time

This section discusses and compares NESTOR-based query primitives with respect to the other state-of-the-art solutions, in terms of time needed for actually evaluating a query, which is the main focus of our paper.

As a summary of what we discuss more in details below, as general trends we see that NESTOR data structures always outperform the other solutions, even if the specific data structure – DDS, IDS, HDS – may change from case to case. The second best approach is almost always BaseX followed by either Jaxen or JXPath, depending on the cases.

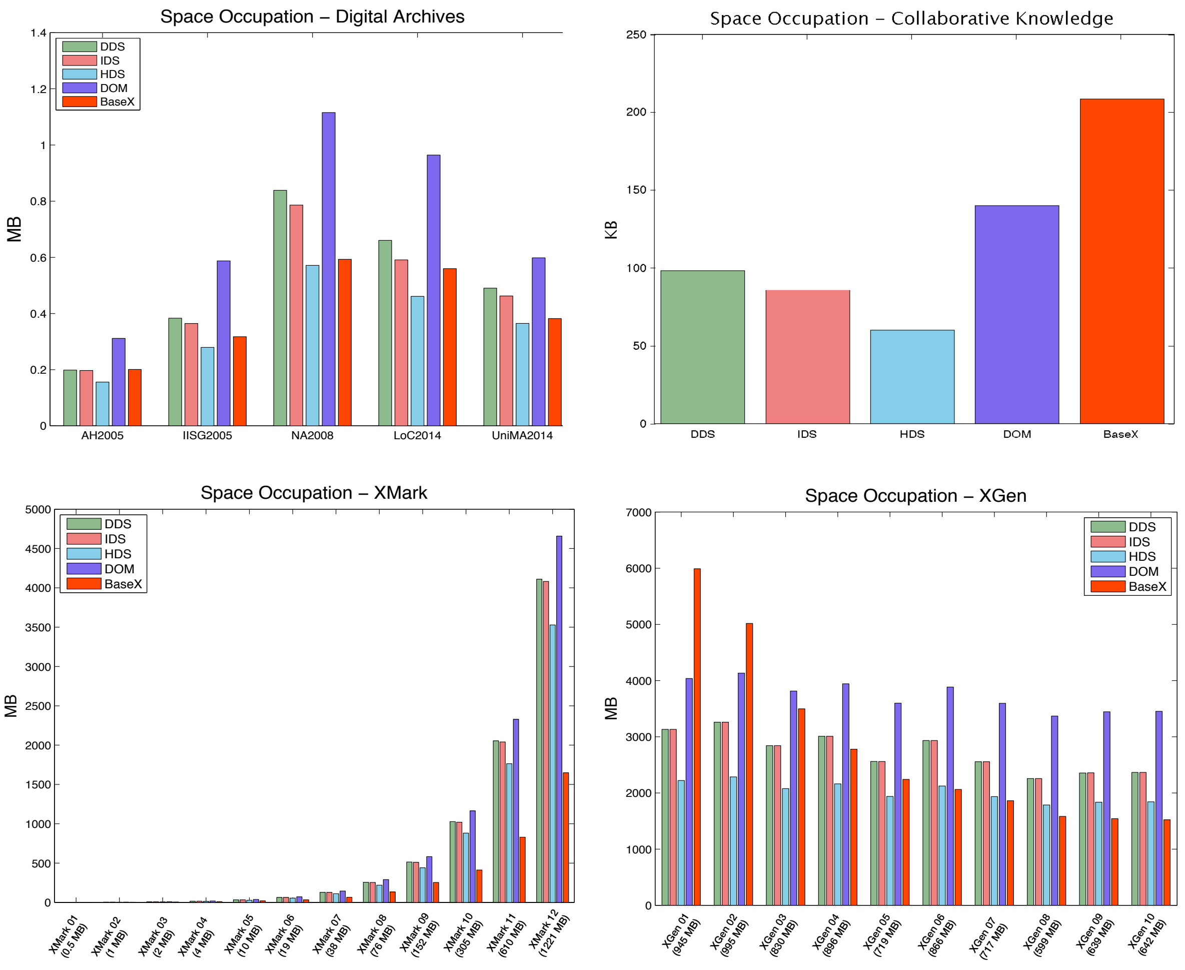


Figure 4: Average space occupation.

In the following, there is a subsection for each of the considered primitives – descendants, ancestors, children, and parent – where more detailed information is reported.

6.1. Descendants

Figure 5 shows the evaluation time for descendants primitives on the different datasets under examination.

As far as the *descendants set-wise primitive* is concerned (Figure 5 on the left), we can see that IDS greatly outperforms all the other approaches by running in constant time and gains, at least, between 3 (digital archives) and 8 (XGen) orders or magnitude. The great advantage of IDS is that it can answer to any set-wise descendant query just by performing a single lookup in the supersets dictionary (i.e., SD) and returning the array corresponding to the requested set. On the other hand, DDS and HDS behave similarly and increase with the number of nodes (i.e. sets in NESTOR) in the XML files as the other solutions do. Indeed, DDS and HDS have to perform several set union operations between the arrays of the direct subsets dictionary (DD); however, these operations can be performed more efficiently than a tree navigation and this explains why they are more efficient

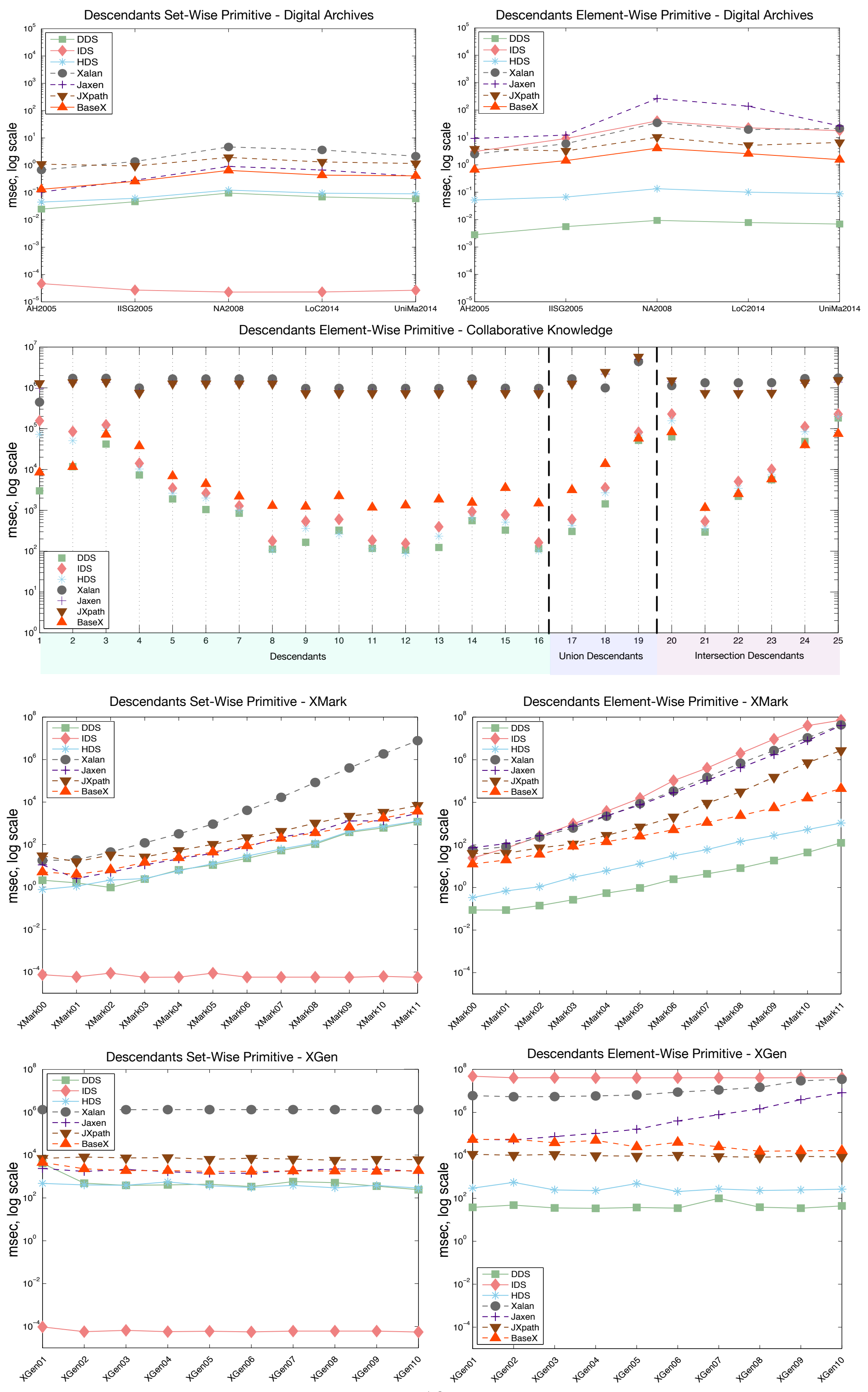


Figure 5: Evaluation time for descendants primitives on the different datasets.

than the other solutions. The worst performing library in this case is Xalan, followed by JXpath and Jaxen, which performs almost equally to BaseX.

In the case of the *descendants element-wise primitive* (Figure 5 on the right), DDS runs better than all the other approaches, gaining at least between 2 (XGen) and 3 (digital archives and XMark) orders of magnitude. This query is executed very efficiently by the DDS because it can be answered just by returning the correct array in the materialized dictionary (i.e. `MD`); indeed, in this case no hierarchy navigation nor collecting of elements node from node is required to answer this query.

The next top performing is HDS, which runs slightly slower than DDS due to its greater computational complexity, even though it is not as slows as one would have expected from the theoretical analysis in Table 3. Indeed, the real datasets are far from the theoretical worst-case scenario of Table 3 and the "local" approach of HDS pays off. On the other hand, IDS is one of the worst performing data structures, because it requires a large number of set unions between the arrays of elements in `MD`.

Xalan has very slow performances and it behaves quite similarly to IDS, closely followed by Jaxen. It is interesting to note how JXPath and BaseX, which are in-between best and worst performers, shows almost the same evaluation time for digital archives and XGen while BaseX gains more for the XMark dataset, where it better handles an increasing tree size but with constant depth.

When it comes to the collaborative knowledge domain, the descendants use case confirms the general trends discussed above with the sole exception that here JXpath behaves on a par with Xalan and Jaxen whereas BaseX performs much better. Moreover, in the intersection descendants use case, BaseX performs almost as well as the NESTOR data structures for most of the queries.

6.2. Ancestors

Figure 6 shows the evaluation time for ancestors primitives on the different datasets under examination.

As far as the *ancestors set-wise primitive* is concerned (Figure 6 on the left), we can see that DDS is the best solution, closely followed by IDS, HDS and BaseX. DDS and BaseX have a constant computational complexity for this operation, whereas IDS and HDS, in the worst-case scenario, increase linearly with the number of sets – i.e. the number of nodes. DDS answers this query just by returning the appropriate array of sets in the `SD` dictionary and thus it performs the operation as fast as BaseX which is specifically optimized for this operation. For the XMark datasets we can see that, even though the number of nodes increases from file to file, the performances of IDS and HDS are constant because the depth of the files is fixed and the number of ancestors is much lower than the total number of nodes. With XGen the number of nodes is fixed and the depth varies from file to file, but also in this case IDS and HDS run in constant time showing that in a real world environment they are very efficient despite their theoretical running time. For digital archives, Jaxen and JXpath perform better for small collections (i.e. AH 2005 and IISG 2005) than for the larger ones (i.e. NA 2008, LoC 2014 and UniMa 2014) and they perform very close for XGen and XMark being in between the best and the worst solutions. Xalan is the slowest library for all the considered datasets.

The ancestor element-wise primitive (Figure 6 on the right) shows that IDS, HDS and BaseX perform best among all the tested solutions. The ancestor element-wise primitive runs in constant time for IDS and BaseX as expected; indeed, to answer this query IDS has to return the appropriate array of elements in the `MD` dictionary without any navigation or collection of elements.

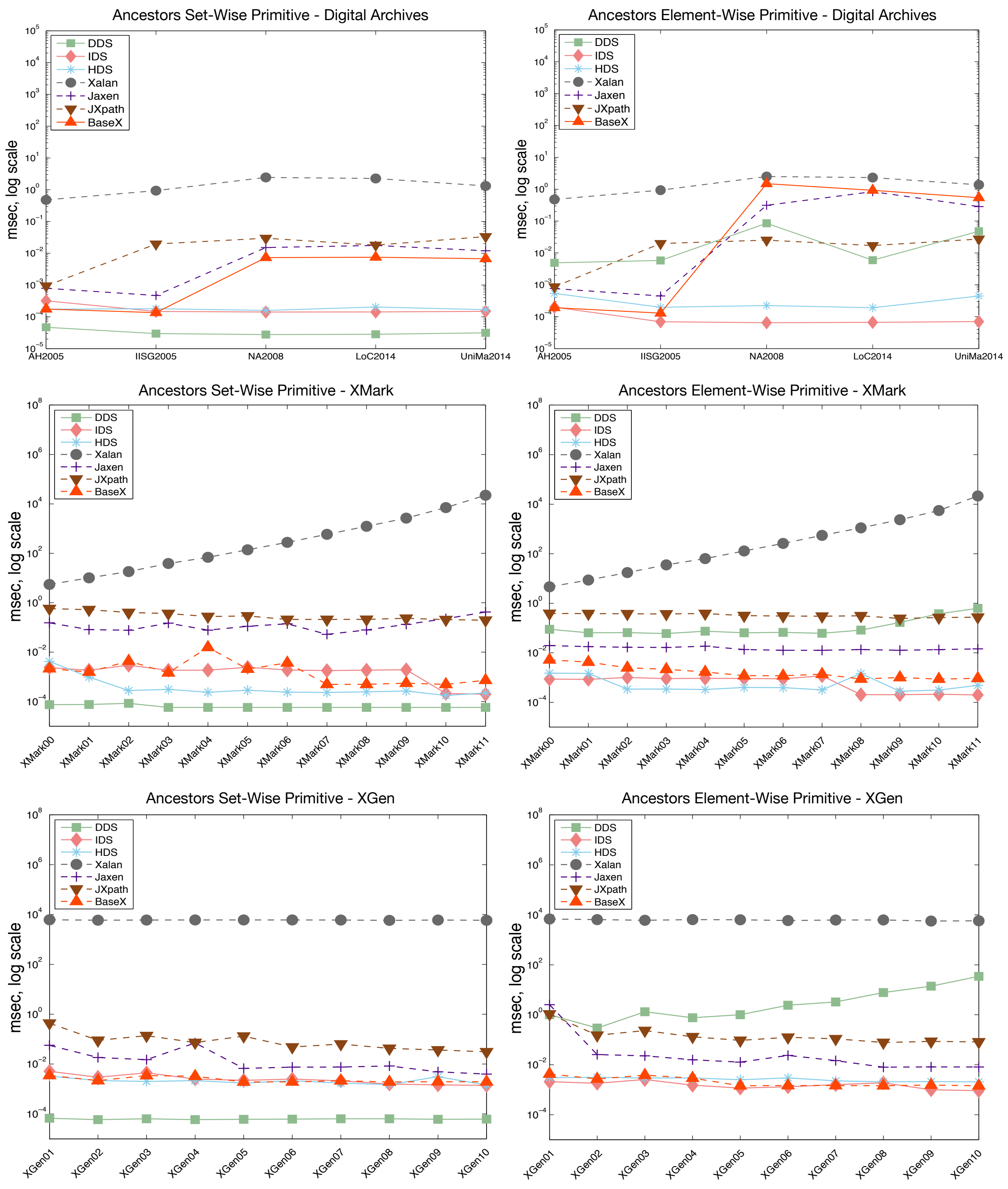


Figure 6: Evaluation time for ancestors primitives on the different datasets.

On the other hand, HDS has $O(n + m)$ worst-case running time but in the experiments it performs as well as the IDS because real datasets differ greatly from the theoretical worst-case situation. Furthermore, HDS returns the content of ancestors without requiring any set-based operation (unions, intersections and differences) and this speeds up the operation. DDS has the same theoretical computational complexity for the ancestor element-wise primitive as HDS, but it has to perform several set differences to return the correct output and these operations explain its overhead with respect to HDS. DDS performs as well as JXpath and better than Jaxen and Xalan for digital archives whereas they are more efficient than DDS when dealing with large XML files as in the case of XMark and XGen.

6.3. Children

Figure 7 shows the evaluation time for children primitives on the different datasets under examination.

As far as the *children set-wise primitive* is concerned (Figure 7 on the left), we can see that DDS and HDS are at least from 1 (digital archives) to 6 (XMark) orders of magnitude faster than all other solutions. DDS and HDS do not depend on the number of children of the given node and thus run in constant time. Note that DDS and HDS answer to this query in the same way given that the `DD` dictionary is built in the same way for both the data structures; basically, they just have to return the appropriate array of direct subsets. XMark and XGen datasets show that all other solutions increase linearly with the max fan-out of the nodes. BaseX and JXpath run faster than Jaxen and Xalan in most cases.

As far as the *children element-wise primitive* is concerned (Figure 7 on the right), we can see that all the solutions depend on the max fan-out of the nodes, but HDS runs at least one order of magnitude faster than BaseX and JXpath; also DDS is quite efficient for this operation. IDS is the least efficient NESTOR-based solution in this case. The main difference between DDS and IDS is due to the fact that the evaluation time on DDS depends on the number of direct subsets of the input set, whereas for IDS it depends on the number of all of its supersets; IDS has to perform many set differences to get the children elements, whereas DDS has to perform a smaller number of set unions. HDS is the best performing solution because it avoids any set-based operation between elements.

6.4. Parent

Figure 8 shows the evaluation time for parent primitives on the different datasets under examination.

As far as the *parent set-wise primitive* is concerned (Figure 8 on the left), we can see that all three NESTOR-based solutions are highly efficient running in constant time for all the datasets. HDS answers this query just by returning the correct array from the `SD` dictionary and IDS by returning the correct array from the `DD` dictionary; whereas DDS has to perform some simple intersections of sets with usually small cardinality which can be performed very efficiently. BaseX is optimized for the parent operation and it closely follows, however running one order of magnitude slower than NESTOR. Among the Java-based libraries, Xalan is the worst solution whereas Jaxen and JXpath behave quite well, being less than one order of magnitude slower than BaseX for digital archives and XGen.

As far as the *parent element-wise primitive* is concerned (Figure 8 on the right), we can see that HDS, as expected from Table 2, runs in constant time for all the datasets and it is the best solution for this operation exploiting the set-wise children query efficiency to answer the element-wise one.

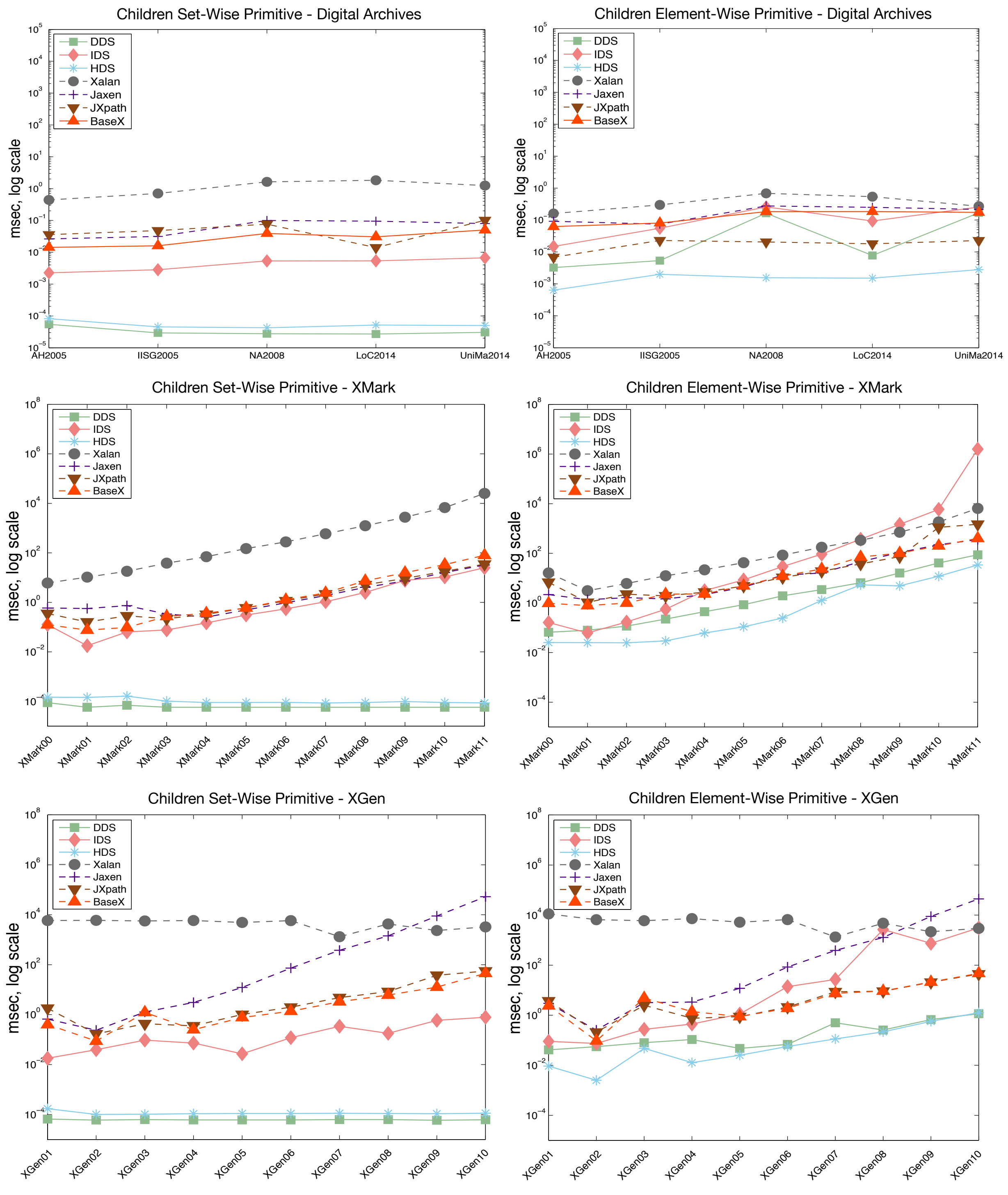


Figure 7: Evaluation time for children primitives on the different datasets.

IDS is as efficient as HDS with digital archives, whereas it runs slower with larger files such as XMark and XGen because it has to perform some set intersections to answer the query. BaseX

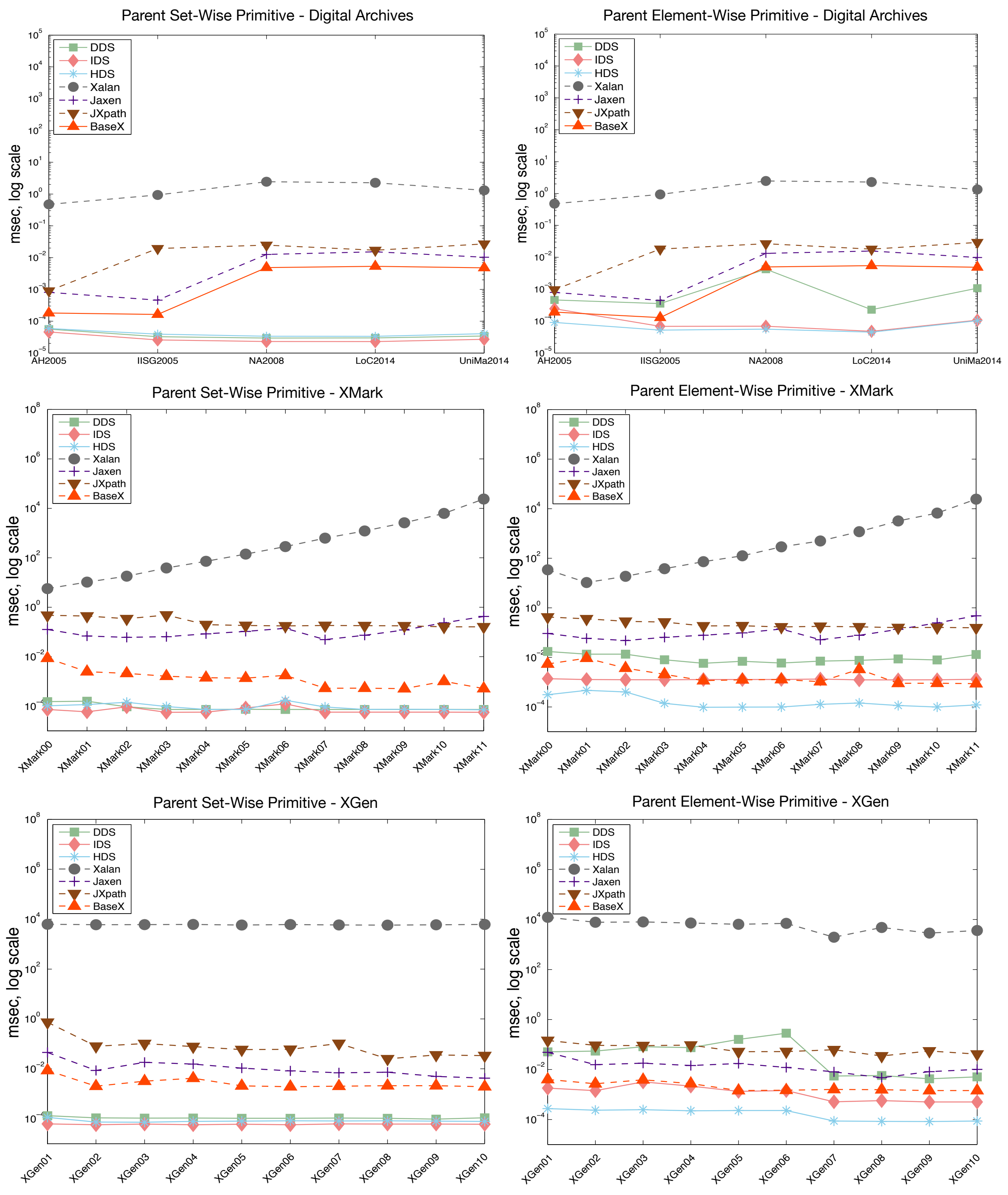


Figure 8: Evaluation time for parent primitives on the different datasets.

behaves on a par with IDS for XGen and XMark, whereas it is slower for digital archives. DDS is rather efficient even though it is the slower NESTOR data structure for this query; this is due to the possibly high number of set differences it has to perform to answer the query.

7. Conclusions and Future Work

In this paper we presented a brand new approach to address XML query primitives relying on basic set operations. This represents a paradigm shift with respect to the navigational-like approaches widely studied and employed in the past. The approach we proposed is based on the NESTOR formal model which represents hierarchical relationships between the nodes of a tree as inclusion dependencies between sets. We implemented the model by means of three alternative in-memory data structures – *Direct Data Structure (DDS)*, *Inverse Data Structure (IDS)*, and *Hybrid Data Structure (HDS)* – with the ultimate goal of enabling more efficient access to XML data. On top of these data structures we developed the *descendants, ancestors, children and parent* XML query primitives showing how they can exploit the characteristics of each data structure to limit and, in several cases, to completely avoid tree navigation and subtree reconstruction and studying the computational complexity.

In particular, we have experimentally shown that NESTOR data structures allow us to outperform state-of-the-art solutions and, on real datasets, they may even perform better than their theoretical worst-case complexity. For example, DDS descendants element-wise and IDS ancestors element-wise primitives are answered in constant time without requiring navigation, recursion or subtree reconstruction; whereas, in several other cases the operations rely on highly-efficient lookups plus set-based operations (intersection and union) between arrays of integers. In the majority of cases – see HDS element-wise and DDS set-wise primitives – the set-based operations can be executed very efficiently, whereas, in just very few cases – e.g. the IDS children element-wise primitive and the IDS descendants element-wise primitive – set-theoretical operations can be quite demanding thus leading to higher execution times.

The choice of the NESTOR data structure to be used depends on the application being considered. We have seen that in the digital archives context HDS is the best choice if all the four query primitives are equally frequent since it is highly effective (even though not always the best) for all the tested cases; on the other hand, if the ancestors element-wise primitive is highly used as it may happen in an application that has to recreate the archival context of a given unit and thus has to return all the elements from a leaf of the XML tree to the root, then IDS is the best choice by far.

The collaborative knowledge domain based on the Wikipedia INEX collection presents us with a specific context where all the XML queries are based on the descendants element-wise primitive. In this context, DDS is the best choice being optimized for returning all the descendant elements of a node in constant time without requiring any additional operations. Some of the INEX topics require to execute set-operations (i.e. unions and intersections) between the result sets of several descendants element-wise primitive operations and also in these cases we have seen that DDS is the most effective solution given that set-theoretical operations can be performed quite efficiently.

The evaluation conducted on synthetic datasets allows us to pinpoint that NESTOR primitives scale up well when the size, the depth, the max fan-out and the total number of nodes of an XML document grow. Indeed, for all the considered XML documents and for all the four query primitives at least one of the NESTOR data structures performs better that any other alternative solution.

Therefore, the experimental results confirm that for XPath primitives NESTOR-based data structures outperform consolidated and state-of-the-art solutions, such as BaseX or the best performing Java XPath libraries. From the pre-processing time point-of-view NESTOR data structure are highly competitive with the other solutions and in particular HDS proves to be the fastest one for the collaborative knowledge. From the space occupation perspective, HDS is the best solution for the digital archive and the collaborative knowledge domains while for big XML files with many nodes and fixed depth (i.e. XMark files) BaseX is the best solution optimizing the storage for XML documents with many nodes containing few data.

Many interesting extensions can be explored as future work, some of which are briefly discussed in the following.

Set-theoretical operations (unions and intersections) are particularly well-suited for being optimized by graphical processors [13, 46] and thus they could be executed very efficiently without requiring any significant change in the NESTOR data structures. NESTOR data structures have not been optimized for space occupation as, for example, BaseX is. Therefore, an interesting future work is to investigate the adoption of some compression techniques to improve on space occupation and to understand their impact on and trade-off with execution time. In this line of reasoning we plan to explore the use of minimal hash functions [9, 12] in place of hash tables, since they are used for efficient storage and fast retrieval. Indeed, these functions can be used within the data structures we presented here without requiring any changes in their specification.

The application of efficient algorithms developed in the context of formal concept mining [37] to the NESTOR operations represents another viable research direction that may lead to an improvement of average performances of the presented algorithms.

We plan to explore the efficiency of NESTOR data structures with respect to the full set of XPath operations, such as XPath predicates, in order to address the whole classification of XPath fragments of [5]. This work should then be complemented with the formal definition of creation, deletion and update operations on XML via NESTOR, the study of their properties and their experimental evaluation in order to have a fully-fledged management suite for XML based on the NESTOR approach that can efficiently deal also with dynamic and incremental XML.

We plan to investigate secondary memory representations of NESTOR in order to develop new query primitives and to compare with ready-to-use secondary memory-based solutions.

Finally, when it comes to further possible application domains of NESTOR, it would be interesting to explore how keyword-based access to (semi-)structured data [7] can benefit from the performance improvements and the efficiency gains provided by NESTOR in order to deal with the massive and heterogeneous amount of data these systems are faced with.

Acknowledgments

The authors wish to warmly thank Jaap Kamps who kindly shared some of the EAD collections and the INEX corpus and topics used for the experiments reported in this work. We also sincerely thank the anonymous reviewers whose suggestions helped in improving this paper.

References

[1] Ali, M. S., Consens, M., Gu, X., Kanza, Y., Rizzolo, F., 2006. Efficient , Effective and Flexible XML Retrieval Using Summaries. In: Fuhr, N., Lalmas, M., Trotman, A. (Eds.), Advances in XML Information Retrieval, Fifth International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2006. Lecture Notes in Computer Science (LNCS) 4518, Springer, Heidelberg, Germany, pp. 89–103.

[2] Alkhatib, R., Scholl, M. H., 2009. Compacting XML Structures Using a Dynamic Labeling Scheme. In: Sexton, A. (Ed.), Dataspace: The Final Frontier, 26th British National Conference on Databases, BNCOD 26, 2009. Vol. 5588 of Lecture Notes in Computer Science. Springer, Heidelberg, Germany, pp. 158–170.

[3] Baeza-Yates, R., Fuhr, N., Maarek, Y., 2006. Introduction to the Special Issue on XML Retrieval. ACM Transactions on Information Systems (TOIS) 24 (4), 405–406.

[4] Beckers, T., Bellot, P., Demartini, G., Denoyer, L., De Vries, C. M., Doucet, A., Fachry, K. N., Fuhr, N., Gallinari, P., Geva, S., Huang, W. C., Iofciu, T., Kamps, J., Kazai, G., Koolen, M., Kutty, S., Landoni, M., Lehtonen, M., Moriceau, V., Nayak, R., Nordlie, R., Pharo, N., SanJuan, E., Schenkel, R., Tannier, X., Theobald, M., Thom, J. A., Trotman, A., de Vries, A. P., 2010. Report on INEX 2009. SIGIR Forum 44 (1), 38–57.

[5] Benedikt, M., Fan, W., Kuper, G. M., 2005. Structural Properties of XPath Fragments. Theor. Comput. Sci. 336 (1), 3–31.

[6] Benedikt, M., Fan, W., Kuper, G. M., 2005. XPath Satisfiability in the Presence of DTDs. In: Gottlob, G., Afrati, F. (Eds.), Proc. 24th ACM SIGMOD–SIGACT–SIGART Symposium on Principles of Database Systems (PODS 2005). ACM Press, New York, USA, pp. 25–36.

[7] Bergamaschi, S., Ferro, N., Guerra, F., Silvello, G., 2015. Keyword-based Search over Databases: A Roadmap for a Reference Architecture Paired with an Evaluation Framework. LNCS Transactions on Computational Collective Intelligence (TCCI).

[8] Boncz, P., Grust, T., Van Keulen, M., Manegold, S., Rittinger, J., Teubner, J., 2006. Monetdb/xquery: A fast xquery processor powered by a relational engine. In: Dyreson, C. E., Li, F., Özsu, M. T. (Eds.), Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD 2006). ACM Press, New York, USA, pp. 479—490.

[9] Botelho, F. C., Pagh, R., Ziviani, N., 2007. Simple and Space-Efficient Minimal Perfect Hash Functions. Vol. 4619 of Lecture Notes in Computer Science. Springer, pp. 139–150.

[10] Celko, J., 2000. Joe Celko's SQL for Smarties: Advanced SQL Programming. Morgan Kaufmann, San Francisco, California, USA.

[11] Chandrasekar, S., Murthy, R., Thusoo, A., Tran, A.-T., Mukkamalla, S., Sedlar, E., Agarwal, N., 2009. Index for accessing XML data. US Patent 7,499,915.

[12] Chazelle, B., Kilian, J., Rubinfeld, R., Tal, a., 2004. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In: Ian Munro, J. (Ed.), Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004. SIAM, pp. 30–39.

[13] Che, S., Boyer, M., Meng, J., Tarjan, D., S., J. W., Skadron, K., Oct. 2008. A Performance Study of General-purpose Applications on Graphics Processors Using CUDA. J. Parallel Distrib. Comput. 68 (10), 1370–1380. URL `http://dx.doi.org/10.1016/j.jpdc.2008.05.014`

[14] Cleverdon, C. W., 1997. The Cranfield Tests on Index Languages Devices. In: Spärck Jones, K., Willett, P. (Eds.), Readings in Information Retrieval. Morgan Kaufmann Publisher, Inc., San Francisco, CA, USA, pp. 47–60.

[15] Cohen, E., Kaplan, H., Milo, T., 2010. Labeling Dynamic XML Trees. SIAM J. Comput. 39.

[16] Culpepper, J. S., Moffat, A., December 2010. Efficient Set Intersection for Inverted Indexing. ACM Trans. Inf. Syst. 29, 1:1–1:25.

[17] Davison, A., 2005. Killer Game Programming in Java. O'Reilly Media, Inc.

[18] DeHaan, D., Toman, D., Consens, M. P., Özsu, M. T., 2003. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In: Halevy, A. Y., Ives, Z. G., Doan, A. (Eds.), Proc. of the ACM SIGMOD International Conference on Management of Data, (SIGMOD 2003). ACM Press, New York, USA, pp. 623–634.

[19] Fallis, D., August 2008. Toward an Epistemology of Wikipedia. Journal of the American Society for Information Science and Technology (JASIST) 59 (10), 1662–1674.

[20] Ferro, N., Silvello, G., 2009. The NESTOR Framework: How to Handle Hierarchical Data Structures. In: Agosti, M., Borbinha, J., Kapidakis, S., Papatheodorou, C., Tsakonas, G. (Eds.), Proc. 13th European Conference on Research and Advanced Technology for Digital Libraries (ECDL 2009). Lecture Notes in Computer Science (LNCS) 5714, Springer, Heidelberg, Germany, pp. 215–226.

[21] Ferro, N., Silvello, G., 2013. NESTOR: A Formal Model for Digital Archives. Inf. Process. Manage. 49 (6), 1206–1240.

[22] Fuhr, N., Lalmas, M., December 2005. Introduction to the Special Issue on INEX. Information Retrieval 8 (4), 515–519.

[23] Geva, S., Kamps, J., Lehtonen, M., Schenkel, R., Thom, J. A., Trotman, A., 2009. Overview of the INEX 2009 Ad-Hoc Track. In: [24], pp. 4–25.

[24] Geva, S., Kamps, J., Trotman, A. (Eds.), 2010. Advances in XML Information Retrieval, Eight International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2009. Lecture Notes in Computer Science (LNCS) 6203, Springer, Heidelberg, Germany.

[25] Gilliland-Swetland, A. J., 2000. Enduring Paradigm, New Opportunities: The Value of the Archival Perspective in the Digital Environment. Council on Library and Information Resources, Washington, DC, USA.

[26] Goodrich, M. T., Tamassia, R., 2001. Data Structures and Algorithms in Java - 2nd Edition. John Wiley and Sons, Inc., Hoboken, NJ, USA.

[27] Gottlob, G., Koch, C., Pichler, R., 2003. XPath Processing in a Nutshell. SIGMOD Rec. 32 (1), 12–19.

[28] Gottlob, G., Koch, C., Pichler, R., Segoufin, L., 2005. The Complexity of XPath Query Evaluation and XML Typing. J. ACM 52 (2).

[29] Grün, C., 2010. Storing and querying large XML instances. Ph.D. thesis, University of Konstanz.

[30] Grust, T., 2002. Accelerating XPath Location Steps. In: Franklin, M. J., Moon, B., Ailamaki, A. (Eds.), Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD 2002). ACM Press, New York, USA, pp. 109–120.

[31] Grust, T., Rittinger, J., Teubner, J., 2007. Why Off-The-Shelf RDBMSs are Better at XPath than you Might Expect. In: Chan, C. Y., Ooi, B. C., Zhou, A. (Eds.), Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD 2007). ACM Press, New York, USA, pp. 949–958.

[32] Grust, T., Rittinger, J., Teubner, J., 2008. Pathfinder: XQuery Off the Relational Shelf. IEEE Data Eng. Bull. 31 (4), 7–14.

[33] Grust, T., van Keulen, M., Teubner, J., 2004. Accelerating XPath evaluation in any RDBMS. ACM Trans. Database Syst. 29, 91–131.

[34] International Council on Archives, March 1999. ISAD(G): General International Standard Archival Description, 2nd edition. Ottawa: International Council on Archives.

[35] Klinger, S., 2010. Pathfinder - Full Text or Extending a Purely Relational XQuery Compiler with a Scoring Infrastructure for XQuery Full Text. Ph.D. thesis, University of Konstanz.

[36] Knuth, D. E., 1997. The Art of Computer Programming, third edition. Vol. 1. Addison Wesley, Reading, MA, USA.

[37] Kuznetsov, S. O., Obiedkov, S., 2002. Comparing Performance of Algorithms for Generating Concept Lattices. Journal of Experimental and Theoretical Artificial Intelligence 14, 189–216.

[38] Larson, R. R., Sanderson, R., 2005. Grid-Based Digital Libraries: Cheshire3 and Distributed Retrieval. In: Marlino, M., Sumner, T., Shipman, F. (Eds.), Proc. 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL 2005). ACM Press, New York, USA, pp. 112–113.

[39] Lee, Y. K., Yoo, S.-L., Yoon, K., Berra, P. B., 1996. Index Structures for Structured Documents. In: Fox, E. A., Marchionini, G. (Eds.), Proc. 1st ACM International Conference on Digital Libraries (DL 1996). ACM Press, New York, USA, pp. 91–99.

[40] Li, Q., Moon, B., 2001. Indexing and Querying XML Data for Regular Path Expressions. In: Apers, P. M. G., Atzeni, P., Ceri, S., Paraboschi, S., Ramamohanarao, K., Snodgrass, R. T. (Eds.), Proc. 27th International Conference on Very Large Data Bases (VLDB 2001). Morgan Kaufmann, pp. 361–370.

[41] Liu, Z., Krishnaprasad, M., Chang, H., Arora, V., May 2008. Techniques of efficient XML query using combination of XML table index and path/value index. US Patent App. 11/601,146.
URL `https://www.google.com/patents/US20080120321`

[42] Mathis, C., Härder, T., Schmidt, K., Bächle, S., 2015. XML indexing and storage: fulfilling the wish list. Computer Science - Research and Development 30 (1), 51–68.

[43] Murthy, R., Liu, Z. H., Krishnaprasad, M., Chandrasekar, S., Tran, A.-T., Sedlar, E., Florescu, D., Kotsovolos, S., Agarwal, N., Arora, V., Krishnamurthy, V., 2005. Towards an Enterprise XML Architecture. In: [47], pp. 953–957.

[44] Nicola, M., Kumar-Chatterjee, P., 2010. DB2 PureXML Cookbook: Master the Power of IBM's Hybrid Data Server. Pearson plc publishing as IBM Press.

[45] O'Neil, P. E., O'Neil, E. J., Pal, S., Cseri, I., Schaller, G., Westbury, N., 2004. ORDPATHs: Insert-Friendly XML Node Labels. In: Weikum, G., König, A. C., Deßloch, S. (Eds.), Proceedings of the ACM SIGMOD International Conference on Management of Data, (SIGMOD 2004). ACM Press, New York, USA, pp. 903–908.

[46] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., Purcell, T. J., 2007. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26 (1), 80–113.

[47] Özcan, F. (Ed.), 2005. Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD 2005). ACM Press, New York, USA.

[48] Pitti, D. V., 1999. Encoded Archival Description. An Introduction and Overview. D-Lib Magazine 5 (11).

[49] Rys, M., Chamberlin, D. D., Florescu, D., 2005. XML and Relational Database Management Systems: The Inside Story. In: [47], pp. 945–947.

[50] Schenkel, R., Suchanek, F. M., Kasneci, G., 2007. YAWN: A semantically annotated Wikipedia XML corpus. In: Kemper, A., Schöning, H., Rose, T., Jarke, M., Seidl, T., Quix, C., Brochhaus, C. (Eds.), Datenbanksysteme in Business, Technologie und Web (BTW 2007), 12. Fachtagung des GI-Fachbereichs "Datenbanken und

Informationssysteme" (DBIS). Lecture Notes in Informatics (LNI) P-103, Gesellschaft für Informatik, Bonn, Germany, pp. 277–291.

[51] Schenkel, R., Theobald, M., 2009. Overview of the INEX 2009 Efficiency Track. In: [24], pp. 200–212.

[52] Theobald, M., Aji, A., Schenkel, R., 2009. TopX 2.0 at the INEX 2009 Ad-Hoc and Efficiency Tracks. In: [24], pp. 218–228.

[53] Theobald, M., Schenkel, R., Weikum, G., 2005. An Efficient and Versatile Query Engine for TopX Search. In: Böhm, K., Jensen, C. S., Haas, L. M., Kersten, M. L., Larson, P., Ooi, B. C. (Eds.), Proc. 31st International Conference on Very Large Data Bases (VLDB 2005). ACM Press, New York, USA, pp. 626–636.

[54] Trotman, A., Sigurbjörnsson, B., 2004. Narrowed Extended XPath I (NEXI). In: Fuhr, N., Lalmas, M., Malik, S., Szlávik, Z. (Eds.), Advances in XML Information Retrieval, Third International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2004. Lecture Notes in Computer Science (LNCS) 3493, Springer, Heidelberg, Germany, pp. 16–40.

[55] W3C, October 1998. Document Object Model (DOM) Level 1 Specification, Version 1.00 – W3C Recommendation 1 October 1998. `http://www.w3.org/TR/REC-DOM-Level-1/`.

[56] W3C, November 1999. XML Path Language (XPath) Version 1.0 – W3C Recommendation 16 November 1999. `http://www.w3.org/TR/xpath/`.

[57] W3C, November 1999. XSL Transformations (XSLT) Version 1.0 – W3C Recommendation 16 November 1999. `http://www.w3.org/TR/xslt/`.

[58] W3C, January 2002. XML Pointer Language (XPointer) – W3C Working Draft 16 August 2002. `http://www.w3.org/TR/xptr/`.

[59] W3C, September 2006. Extensible Markup Language (XML) 1.1 (Second Edition) – W3C Recommendation 16 August 2006, edited in place 29 September 2006. `http://www.w3.org/TR/xml11/`.

[60] W3C, November 2008. Extensible Markup Language (XML) 1.0 (Fifth Edition) – W3C Recommendation 26 November 2008. `http://www.w3.org/TR/xml/`.

[61] W3C, December 2010. XML Path Language (XPath) 2.0 (Second Edition) – W3C Recommendation 14 December 2010. `http://www.w3.org/TR/xpath20/`.

[62] W3C, December 2010. XQuery 1.0: An XML Query Language (Second Edition) – W3C Recommendation 14 December 2010. `http://www.w3.org/TR/xquery/`.

[63] Wang, F., Li, J., Homayounfar, H., 2007. A space efficient XML DOM parser. Data & Knowledge Engineering 60 (1), 185–207.

[64] Wang, W., Jiang, H., Wang, H., Lin, X., Lu, H., Li, J., 2005. Efficient Processing of XML Path Queries Using the Disk-based F&B Index. In: Proceedings of the 31st International Conference on Very Large Data Bases. VLDB '05. VLDB Endowment, pp. 145–156.

[65] Zhang, J., 2012. System Evaluation of Archival Description and Access. SIGIR Forum 45 (2), 109–110.

Appendix A: Pseudo-Code and Complexity Analysis of NESTOR Primitives

The dictionary-based data structure is composed of three dictionaries which are `MD` for the materialized sets, `DD` for the direct subsets of each set, and `SD` for the supersets of each set. Without loss of generality we can assume that the dictionaries are composed by $\langle k_i, L_i \rangle$ pairs, where k_i is the key represented by an integer value and L_i is a list of integers. This choice allows us to describe the pseudo-code of the algorithms without caring about the dimension of the arrays we manipulate. The theoretical complexity of the algorithms is the same but the pseudo-code is more readable by working with lists in place of arrays.

In order to work with a list `L` we need to point out some basic methods (please refer to [26] for details on the computational complexities of these methods). Let us consider a generic list, say `L`, and a natural number, say $i \in \mathbb{N}$, then `L.get(i)` returns the i^{th} element in `L`, `L.size()` returns the number of elements in `L` and it runs in $O(1)$ time, `L.add(x)` adds the element x to the end of `L` and it runs in $O(1)$ time, `L.remove(x)` removes the element x from `L` (if x is present in `L`, otherwise it does nothing) and it runs in $O(1)$ time. The method `L.remove` removes the last element in `L` and it runs in $O(1)$ time. `L.add(i,x)` adds the element x at the position i of list `L` and it runs in $O(1)$ time, `L.addAll(W)` adds all the elements of list `W` to list `L` (duplicates are not eliminated) and it runs in $O($`W.size()`$)$ time. `L.removeAll(W)` removes all the elements in list `W` from list `L` and it runs in $O($`W.size()`$)$ time; `L.isEmpty()` returns TRUE if the list is empty or FALSE otherwise, and it runs in $O(1)$ time. Lastly, we point out the method BinarySearch(`H`, `x`) which returns the index of the elements x in H if $x \in H$ or a negative integer otherwise; the complexity of this operation is $O(\log |H|)$ [26].

By considering these *dictionary-based* data structures we can also point out the computational complexities of some primitives we need to support working with sets. Let us consider a collection of sets $\mathcal{C}$ composed of $m \in \mathbb{N}$ sets and $n \in \mathbb{N}$ elements, two sets $H, K \in \mathcal{C}$ such that $|H| = n_1 \in \mathbb{N}$ and $|K| = n_2 \in \mathbb{N}$, and an element $x \in \mathcal{C}$. Without loss of generality, we assume that $|H| \leq |K|$ which means that $n_1 \leq n_2$.

The **Member**(H) operation over a set (H) with n_1 elements stored as a sorted array of integers can be implemented in $O(\log n_1)$ time by using binary search [26]. The **Size**(H) operation can be implemented in $O(1)$ time by returning the length of the array representing the materialized set given as input. In the following we present the algorithms **UnionDDS**(H, K) and **DifferenceDDS**(H, K) algorithm specifically implemented for the DDS; these algorithms exploit the DDS peculiarities to speed-up the union and difference set operations. For the IDS and HDS data structures, the **Intersect**(H, K) and the **Difference**(H, K) operations are implemented following the algorithm proposed in [16] which runs in $O(n_1 + n_1 \log(n_2/n_1))$ time.

ALGORITHM 1: UnionDDS

```
    Input: The sets $H, K \in \mathcal{C}$.
    Output: The list of elements representing $H \cup K$.
1 Without loss of generality we assume that $|H| > |K|$.
2 if BinarySearch(MD.get(H), MD.get(K).get(0)) $< 0$ then
3     return MD.get(H)
4 else
5     H.addAll(K)
6     return H
7 end
```

This algorithm can be applied only to the DDS and the sets defined accordingly to that structure. It checks if the first element of K is in H with binary search in $O(\log |H|)$; note that the statement `MD.get(K).get(0)` returns the first element in the array containing the elements of set K and `get(0)` retrieves the first element of the array, which is the desired one. Then if this element is in H, it returns H otherwise it returns the concatenation of H and K and this operation requires $O(|K|)$. In the following we also extensively use the difference operation between two sets in the DDS, and for this reason we define the DIFFERENCEDDS algorithm.

ALGORITHM 2: DifferenceDDS

```
Input: The sets $H, K \in \mathcal{C}$.
Output: The list of elements representing $H \setminus K$.
1  Without loss of generality we assume that $|H| > |K|$.
2  index ←BinarySearch(MD.get(H), MD.get(K).get(0))
3  R ← ∅
4  if index > −1 then
5      for i ← 0 to index do
6          R.add(H.get(i))
7      end
8      return R
9  else
10     return H
11 end
```

In the worst-case scenario this algorithm requires $O(|H|)$ time, whereas in the best scenario in which H and K are disjoint it runs in $O(\log |H|)$.

The ELEMENTS(H) primitive returns the elements belonging to a set, say H, and it is implemented in the same way both for the DDS and the IDS.

ALGORITHM 3: Elements

```
Input: The set $H \in \mathcal{C}$.
Output: The list of elements belonging to $H \in \mathcal{C}$.
1  return MD.get(H)
```

This algorithm simply returns the values in `MD` corresponding to the input set H. The computational complexity of this algorithm is $O(1)$.

DDS Set-Wise Primitives

The algorithms implementing ANCESTORS-DDS(H) and CHILDREN-DDS(H) are immediate because they only require access to the proper dictionary in the data structure and so we present them first.

ALGORITHM 4: Ancestors-DDS

```
Input: The set $H \in \mathcal{C}$.
Output: The list containing the ancestors of $H \in \mathcal{C}$.
1  return SD.get(H)
```

This algorithm simply returns the list of supersets of the input set H by accessing the appropriate list in the dictionary `SD`. The ancestors of a set H in a collection of sets $\mathcal{C}$ are the supersets of H. The computational complexity of this algorithm is $O(1)$.

ALGORITHM 5: Children-DDS

```
  Input: The set $H \in \mathcal{C}$.
  Output: The list containing the children of $H \in \mathcal{C}$.
1 return DD.get(H)
```

This algorithm simply returns the list of direct-subsets of the input set H by accessing a list in DD. The children of H in $\mathcal{C}$ are the direct-subsets of H. The complexity of this algorithm is $O(1)$.

Let us see the **Descendants-DDS** algorithm where we use a *queue* [26], say Q, which we can see as a list equipped with two additional methods: Q.element() which returns the last element in Q without removing it, and Q.remove() which returns the last element in Q and removes it from the queue. These methods are implemented to run in $O(1)$ time [26].

ALGORITHM 6: Descendants-DDS

```
  Input: The set $H \in \mathcal{C}$.
  Output: The list $S^+$ containing the subsets of $H \in \mathcal{C}$.
1 Q.add(H)
2 while !Q.isEmpty() do
3     $S^+$.addAll(Children-DDS(Q.element()))
4     Q.addAll(Children-DDS(Q.remove()))
5 end
6 $S^+$.add(H)
7 return $S^+$
```

We named a list as S^+ by using a superscript even though this notation is not usual in pseudo-code, but we think it can help the reader to understand the semantic of the list under consideration.

In Algorithm 6 we can see that the reference to set H is added to the queue Q (line 1), then, while the queue is not empty we add the direct subsets of the last element in the queue to the list S^+ (line 3) and to the queue Q (line 4); afterwards, the algorithm removes this element from Q. The algorithm processes all the direct subsets of H and all of their subsets iteratively.

The worst-case scenario is represented by a collection of sets $\mathcal{C}$ structured as a chain ($|\mathcal{C}| = m$), – i.e. each set in the collection has no more than one subset [21]. Furthermore, in the worst-case scenario we choose the top set as input set. This means that the *while cycle* at line 2 iterates on all the m sets in $\mathcal{C}$. Each operation inside the *while cycle* (line 3 and line 4) runs in constant time, so this algorithm runs in $O(m)$ time.

In the algorithm implementing parent operation for the DDS we exploit the fact that in the DDS every set has at most one direct superset.

ALGORITHM 7: Parent-DDS

```
  Input: The set $H \in \mathcal{C}$.
  Output: The set representing the parent of $H \in \mathcal{C}$.
1 $S^- \leftarrow$ Ancestors-DDS($H$)
2 if $S^-$.isEmpty() then
3     return $\varnothing$
4 else
5     without loss of generality we assume that $S^-$.size() = n where $n \in \mathbb{N}$ such that
      Size($S^-$.get(0)) $\leq$ Size($S^-$.get(1)) $\leq \ldots \leq$ Size($S^-$.get(n))
6     return $S^-$.get(0)
7 end
```

At line 1 the algorithm checks the list of all the supersets of H, if it is empty this means that

H is the top set of $\mathcal{C}$ and thus it has no supersets; otherwise, we assume that the sets in list $\mathtt{S}^-$ are increasingly ordered by their cardinality and thus, the first set in the list is the one with higher cardinality and it must be the direct superset of H. In the worst-case scenario the computational complexity of this algorithm is $O(1)$.

IDS Set-Wise Query Operations

Let us consider a collection of sets $\mathcal{C}$ where $H \in \mathcal{C}$ is a set.

ALGORITHM 8: Descendants-IDS

```
Input: The set $H \in \mathcal{C}$.
Output: The list containing the descendants of $H \in \mathcal{C}$.
return SD.get(H)
```

This algorithm returns the list of supersets of the input set H by accessing the appropriate list in `SD`. The descendants of H in $\mathcal{C}$ are the supersets of H. The computational complexity of this algorithm is $O(1)$.

The following algorithm simply returns the list of direct-subsets of H by accessing the appropriate list in `DD`; in the IDS the lists of direct-superst of each set are composed by at most one set representing its parent. The parent of H in $\mathcal{C}$ is the direct-subset of H. The computational complexity of this algorithm is $O(1)$.

ALGORITHM 9: Parent-IDS

```
Input: The set $H \in \mathcal{C}$.
Output: The set representing the parent of $H \in \mathcal{C}$.
return DD.get(H)
```

The ancestor operation for the IDS is realized by the **Ancestors-IDS** algorithm; here we exploit the fact that each set in a collection of sets has at most one direct subset.

ALGORITHM 10: Ancestors-IDS

```
Input: The set $H \in \mathcal{C}$.
Output: The list $\mathtt{S}^+$ containing the set representing the parent of $H \in \mathcal{C}$.
1  $\mathtt{S}^+ \leftarrow$ PARENT($H$)
2  if $\mathtt{S}^+$.isEmpty() then
3      return $\varnothing$
4  else
5      $\mathtt{D}^+ \leftarrow$ PARENT($H$)
6      while (!PARENT($\mathtt{D}^+$.get(0)).isEmpty()) do
7          $\mathtt{S}^+$.addAll(PARENT($\mathtt{D}^+$.get(0)))
8          $\mathtt{D}^+ \leftarrow$ PARENT($\mathtt{D}^+$.get(0))
9      end
10 end
11 $\mathtt{S}^+$.add(H)
12 return $\mathtt{S}^+$
```

The worst-case scenario is represented by $\mathcal{C}$ structured as a chain ($|\mathcal{C}| = m$) and by H such that the number of direct supersets of H is 0. This means that $|\mathtt{S}^+(H)| = m - 1$. The *while* cycle at line 6 is repeated until the list $\mathtt{D}^+$ is empty which means that the current set has no subsets. We know that at every iteration $\mathtt{D}^+$ contains at most one element and when H is the input set the cycle is repeated m times. The operations at lines 7 and 8 run in constant time, so in the worst-case scenario the algorithm runs in $O(m)$ time.

For the IDS we implement the **Children-IDS** algorithm. The operation at line 1 runs in $O(1)$ time. So the computational complexity is dominated by the *while cycle* at line 6 which is executed $O(|\mathtt{S}^-|)$ times; in the worst-case the input set H is the set with no subsets. In this scenario the while cycle is executed m times. The operations inside the cycle run in constant time and thus the overall computational complexity of the algorithm is $O(m)$.

ALGORITHM 11: Children-IDS

```
    Input: The set $H \in \mathcal{C}$.
    Output: The list of set $\mathtt{D}^-$ representing the children of $H \in \mathcal{C}$.
 1  $\mathtt{S}^- \leftarrow$ DESCENDANTS-IDS($H$)
 2  $\mathtt{i} \leftarrow \mathtt{0}$
 3  if $\mathtt{S}^-$.size() $< \mathtt{0}$ then
 4      return $\varnothing$
 5  else
 6      while $\mathtt{i} < \mathtt{S}^-$.size() do
 7          if PARENT-IDS(C.get(i)) == H then
 8              $\mathtt{D}^-$.add(C.get(i))
 9          end
10          $\mathtt{i} \leftarrow \mathtt{i} + \mathtt{1}$
11      end
12  end
13  return $\mathtt{D}^-$
```

HDS Set-Wise Primitives

The HDS inherits some algorithms from the DDS illustrated above; indeed, CHILDREN-HDS and DESCENDANTS-HDS work exactly as CHILDREN-DDS andDESCENDANTS-DDS sharing the same worst-cases and computational complexities.

ALGORITHM 12: Parent-HDS

```
    Input: The set $H \in \mathcal{C}$.
    Output: The parent of $H \in \mathcal{C}$.
 1  return SD.get(H)
```

This algorithm simply returns the list of supersets of the input set H by accessing the appropriate list in the dictionary SD which in the HDS contains only the parent of H. The computational complexity of this algorithm is $O(1)$.

ALGORITHM 13: Ancestors-HDS

```
    Input: The set $H \in \mathcal{C}$.
    Output: The list of sets $\mathtt{D}^-$ representing the ancestors of $H \in \mathcal{C}$.
 1  $\mathtt{S}^- \leftarrow$ PARENT-HDS($H$)
 2  if $\mathtt{S}^-$.isEmpty() then
 3      return $\varnothing$
 4  else
 5      while !PARENT-HDS($\mathtt{S}^-$.get($\mathtt{S}^-$.size() $-$ 1)).isEmpty() do
 6          $\mathtt{S}^-$.add(PARENT-HDS($\mathtt{S}^-$.get($\mathtt{S}^-$.size() $-$ 1)))
 7      end
 8  end
 9  return $\mathtt{S}^-$
```

The worst case scenario for this algorithm is represented by a collection of sets $\mathcal{C}$ structured as a chain, where $H \in \mathcal{C}$ is the last set (i.e. the only one without subsets). In this case the *while cycle* at line 5 takes $O(m)$, which is the computational complexity of this algorithm given that the operations at line 1 and at line 6 run in $O(1)$.

DDS Element-Wise Primitives

Just like the structural queries, the content queries are also declined into three versions, the first for the DDS, the second for the IDS and the third for the HDS. Firstly, we present the algorithms implementing the DDS element-wise primitives starting from the DescendantElements-DDS algorithm.

ALGORITHM 14: Descendant-DDS

```
    Input: The set $H \in \mathcal{C}$.
    Output: The list $\mathtt{E}^+$ containing the descendant elements of $H$.
1 return Elements(H)
```

The computational complexity of this algorithm is $O(1)$ in all possible scenarios because it has only to return the elements belonging to the input sets; indeed a set in a collection of subsets contains all the elements of its subsets by definition. The following algorithm shows the implementation of the AncestorElements-DDS primitive.

ALGORITHM 15: Ancestor-DDS

```
    Input: A collection of sets $\mathcal{C}$ and the set $H \in \mathcal{C}$.
    Output: The list of elements belonging to the ancestors of $H$.
1  $\mathtt{D}^+ \leftarrow \varnothing$
2  $\mathtt{S}^- \leftarrow$ Ancestors-DDS(H)
3  for $\mathtt{i} \leftarrow \mathtt{S}^-.\mathtt{size}() - 1$ to 1 do
4      $\mathtt{D}^+.\mathtt{addAll}\Big(\text{Children-DDS}\big(\mathtt{S}^-.\mathtt{get(i)}\big).\mathtt{remove}\big(\mathtt{S}^-.\mathtt{get(i+1)}\big)\Big)$
5  end
6  $\mathtt{D}^+.\mathtt{addAll}\Big(\text{Children-DDS}\big(\mathtt{S}^-.\mathtt{get(0)}\big)\Big)$
7  $\mathtt{E} \leftarrow \text{Elements}\big(\mathtt{D}^+.\mathtt{get(0)}\big)$
8  for $\mathtt{i} \leftarrow 1$ to $\mathtt{D}^+.\mathtt{size}() - 1$ do
9      $\mathtt{E} \leftarrow \text{UnionDDS}\Big(\mathtt{E}, \text{Elements}\big(\mathtt{D}^+.\mathtt{get(i)}\big)\Big)$
10 end
11 return $\text{DifferenceDDS}\big(\text{Elements}(\mathtt{0}), \mathtt{E}\big)$
```

The worst-case scenario for this algorithm is represented by a NS-C structured as a chain with $m \in \mathbb{N}$ sets and $n \in \mathbb{N}$ elements. The operation at line 2 is executed in $O(1)$ time, whereas the one at line 6 runs in $O(|\text{Children-DDS}\big(\mathtt{S}^-.\mathtt{get(i)}\big)|)$ time, where $i \in [0, m-1]$, but in our worst-case scenario for all $i \in [0, m-1]$ we have that $|\text{Children-DDS}\big(\mathtt{S}^-.\mathtt{get(i)}\big)| = 1$. We assume that the input set H is the bottom set of the chain which has m ancestors (supersets), so the *for cycle* at line 3 is executed m times; the operation inside this cycle requires $O(|\text{Children-DDS}\big(\mathtt{S}^-.\mathtt{get(i)}\big)|)$ time, but $|\text{Children-DDS}\big(\mathtt{S}^-.\mathtt{get(i)}\big)| = 1$ for all possible $i \in [0, m]$, so this operation runs in $O(1)$ time. This means that the list $\mathtt{D}^+$ after this cycle has size one and that the *for cycle* at line 8 is executed only once. The UnionDDS operation inside the cycle requires $O(|E|) = O(n)$ time. Lastly, in the worst case the DifferenceDDS operation at line 11 is executed in $O(n)$ time. It is worthwhile to note that the use of the UnionDDS and DifferenceDDS algorithms in place of the general ones produces no change on the computational complexity of the Ancestor-DDS algorithm, but it will have a tangible effect in the experimental results.

The following pseudo-code describes how the CHILDREN-DDS algorithm is implemented. In the worst-case scenario represented by a NS-C, say $\mathcal{C}$, structured as a chain, every set $H \in \mathcal{C}$ has at most one direct subset, that means at most one child. Therefore, both the *for cycles* at line 3 and line 6 are executed at most once. The complexity is dominated by the DIFFERENCE operation at line 7. The computational complexity of this operation depends on the cardinality of the sets $\mathtt{C_e}$ and ELEMENTS$\big(\mathtt{C.get(j)}\big)$. In the worst-case scenario it can happen that $|\mathtt{C_e}| = n$ and $|\text{ELEMENTS}\big(\mathtt{C.get(j)}\big)| = n - \varepsilon$, where $0 < \varepsilon \ll n$; in this case the computational complexity of the algorithm is $O(n)$.

ALGORITHM 16: Children-DDS

```
    Input: A collection of sets $\mathcal{C}$ and the set $H \in \mathcal{C}$.
    Output: The list of elements belonging to the children of $H$.
1  E ← ∅
2  $D^+$ ← CHILDREN-DDS(H)
3  for i ← 0 to $D^+$.size() − 1 do
4      $C_e$ ← ELEMENTS($D^+$.get(i))
5      C ← CHILDREN-DDS($D^+$.get(i))
6      for j ← 0 to C.size() − 1 do
7          $C_e$ ← DIFFERENCE($C_e$, ELEMENTS(C.get(j)))
8      end
9      E.addAll($C_e$)
10 end
11 return E
```

The following pseudo-code describes the PARENT-DDS algorithm.

ALGORITHM 17: Parent-DDS

```
    Input: A collection of sets $\mathcal{C}$ and the set $H \in \mathcal{C}$.
    Output: The list of elements belonging to the parent of $H$.
1  P ← PARENT-DDS($H$)
2  $P_e$ ← ELEMENTS($P$)
3  $D^+$ ← CHILDREN-DDS($P$)
4  $H_e$ ← ELEMENTS($H$)
5  for i ← 0 to $D^+$.size() − 1 do
6      $H_e$.addAll(ELEMENTS($D^+$.get(i)))
7  end
8  return DIFFERENCEDDS($P_e$, $H_e$)
```

As well as for the CHILDREN-DDS in the PARENT-DDS worst-case scenario, represented by a NS-C structured as a chain, every set has at most one direct subset and so the *for cycle* at line 5 is executed exactly once. The operation at line 6 requires $O\Big(|\text{ELEMENTS}\big(\mathtt{D^+.get(i)}\big)|\Big) = O(n)$ in the worst-case scenario. It is important to notice that we do not perform a UNION between sets but a concatenation of their elements. This can be done because all the direct subsets of a set are disjoint in the DDS. Lastly, the DIFFERENCEDDS operation at line 8 requires $O(n)$. Therefore, the computational complexity for the PARENT-DDS algorithm is $O(n)$.

IDS Element-Wise Primitives

In this subsection we describe the algorithms implementing the element-wise primitives for the IDS.

ALGORITHM 18: Descendant-IDS

```
Input: A collection of sets $\mathcal{C}$ and the set $H \in \mathcal{C}$.
Output: The list $E^-$ containing the elements belonging to the descendants of $H$.
1 $S^- \leftarrow$ DESCENDANTS-IDS(H)
2 E $\leftarrow \varnothing$
3 for i $\leftarrow$ 0 to $S^-$.size() $-$ 1 do
4     if (DESCENDANTS-IDS($S^-$.get(i)).isEmpty() then
5         E $\leftarrow$ UNION (E,ELEMENTS ($S^-$.get(i)))
6     end
7 end
8 return E
```

This algorithm does the set-theoretical union of the elements belonging to the supersets of the input set H which do not have any supersets. The worst-case scenario is represented by a collection of sets $\mathcal{C}$ with m sets and n elements, structured as a chain where the input set H is the bottom set. This means that the list of supersets of H contains all the m sets in $\mathcal{C}$, thus the *for cycle* at line 3 is executed m times; the only operation executed in the cycle is the *if* statement at line 4 which runs in constant time ($O(1)$). The *union* operation at line 5 is executed just once because in a chain there is only one set with no supersets; furthermore, when the union is performed the list E^- is empty, whereas the set (S^-.get(i)) (refer to line 5) contains all the n elements in $\mathcal{C}$. The computational complexity of the union at line 5 is $O(|E^-| + n) = O(n)$. Therefore, the computational complexity of the algorithm in the worst-case scenario is $O(m + n)$.

ALGORITHM 19: Ancestor-IDS

```
Input: The set $H \in \mathcal{C}$.
Output: The list $E^+$ containing the ancestor elements of $H$.
1 return Elements(H)
```

The computational complexity of this algorithm is $O(1)$ in all possible scenarios because it has only to return the elements belonging to the input sets; indeed a set in a collection of subsets contains all the elements of its subsets by definition. The following pseudo-code describes the implementation of the CHILDREN-IDS algorithm.

ALGORITHM 20: Children-IDS

```
Input: A collection of sets $\mathcal{C}$ and the set $H \in \mathcal{C}$.
Output: The list of elements belonging to the children of $H$.
1  $D^- \leftarrow$ CHILDREN-IDS(H)
2  if $D^-$.size() > 0 then
3      $C_e \leftarrow$ ELEMENTS($D^-$.get(0))
4      for i $\leftarrow$ 1 to $D^-$.size() $-$ 1 do
5          $C_e \leftarrow$ UNION($C_e$, ELEMENTS($D^-$.get(i)))
6      end
7      return DIFFERENCE ($C_e$, H)
8  else
9      return $\varnothing$
10 end
```

The worst-case scenario is represented by collection of sets structured as a chain in which every set has at most one superset. In this case the *for cycle* at line 4 is executed once and the UNION operation at line 5 runs in $O(n)$ time as well as the DIFFERENCE operation at line 7. The structural query Children-DDS at line 1 requires $O(m)$, thus the overall computational complexity of this algorithm in $O(m+n)$.

ALGORITHM 21: Parent-IDS

```
Input: A collection of sets $\mathcal{C}$ and the set $H \in \mathcal{C}$.
Output: The list of elements belonging to the parent of $H$.
P ← Parent-IDS($H$)
P_e ← Elements(P)
if Parent-IDS(P).size() > 0 then
    return Difference $\Big(\mathtt{P_e}, \text{Elements}\big(\text{Parent-IDS}(\mathtt{P})\big)\Big)$
else
    return P_e
end
```

The worst-case scenario is represented by a collection of sets structured as a chain and the computational complexity is dominated by the Difference operation at line 4 which runs in $O(n)$ time.

HDS Element-Wise Primitives

Let us see the Children-HDS algorithm.

ALGORITHM 22: Children-HDS

```
Input: A collection of sets $\mathcal{C}$ and the set $H \in \mathcal{C}$.
Output: The list of elements belonging to the children of $H$.
D+ ← Children-DDS(H)
if D+.size() > 0 then
    for i ← 0 to D+.size() − 1 do
        E.addAll(MD.get(D+.get(i)))
    end
    return E
else
    return ∅
end
```

The worst-case scenario for this algorithm is represented by a collection of sets with depth 2 where the top set has $m-1$ direct subsets such that these sets contain n elements in total. In this case, the operation at line 1 runs in constant time and the computational complexity is given by the *for cycle* at line 3 which requires $O(m)$ time. The operation at line 4 runs in $O(|MD.get(D^+.get(i))|)$ and given that $\sum_{i=0}^{D^+.size()-1}(|MD.get(D^+.get(i))|) = n$ the total running time is $O(n)$. Thus, Children-HDS runs in $O(n)$ if $n >> m$ and $O(m) otherwise$.

ALGORITHM 23: Descendant-HDS

Input: A collection of sets $\mathcal{C}$ and the set $H \in \mathcal{C}$.
Output: The list of elements belonging to the descendants of H.

```
1 S⁺ ← Descendants-DDS(H)
2 if S⁺.size() > 0 then
3     for i ← 0 to S⁺.size() − 1 do
4         E.addAll(MD.get(S⁺.get(i)))
5     end
6     return E
7 else
8     return ∅
9 end
```

The worst-case scenario for this algorithm is represented by a collection of sets structured as a chain. In this case operation at line 1 runs in $O(m)$ and the for cycle at line 3 is dominated by the operation at line 4 which runs in $O(n)$ (as discussed above); the running time of the algorithm in $O(m + n)$.

ALGORITHM 24: ParentElements-HDS

Input: A collection of sets $\mathcal{C}$ and the set $H \in \mathcal{C}$.
Output: The list of elements belonging to the parent of H.

```
1 return MD.get(PARENT-DDS(H))
```

The operation PARENT-DDS(H) runs in constant time for whichever collection of sets and thus also this algorithm runs in constant time. The worst-case scenario for this algorithm is represented by a collection of sets structured as a chain. In this case the operation at line 1 runs in $O(m)$ time and the for cycle at line 3 requires $O(n)$ time as for the Children-HDS algorithm, thus the global running time is $O(m + n)$.

ALGORITHM 25: Ancestor-HDS

Input: A collection of sets $\mathcal{C}$ and the set $H \in \mathcal{C}$.
Output: The list of elements belonging to the ancestors of H.

```
1 S⁻ ← Ancestors-HDS(H)
2 if S⁻.size() > 0 then
3     for i ← 0 to S⁻.size() − 1 do
4         E.addAll(MD.get(S⁻.get(i)))
5     end
6     return E
7 else
8     return MD.get(H)
9 end
```

Appendix B: Definition of the experimental queries

Digital Archives and Synthetic Datasets

For the digital archives and the synthetic datasets the operations used to test the different solutions are executed by selecting the worst-case situation for each operation. The worst-case coincides for the set-wise and the element-wise operations. For the NESTOR-based solutions the worst-case situation yields to the definition of the input set for the query primitives; for the XPath libraries and the native XML database it yields to the definition of a location path relative to a context node which coincides with the selected set for NESTOR-based solutions.

For the descendants operations the worst-case input is the root of an XML file because the selected solution has to return all the nodes, sets or elements in the tree. For the ancestors operations the worst-case input is the node at maximum depth in the XML file because the selected solution has to return all the nodes, sets or elements up to the root of the tree. For the children operations the worst-case input is the node with the max fan-out in the XML file; for the parent operations there is no clear worst-case so we selected a node in the XML with average fan-out and half way between the root and the leaf at maximum depth in the tree.

Collaborative Knowledge

For each topic we selected the CAS component and we excluded the content part thus obtaining an XPath query. Once the content part is removed we obtain the structure query which is `//group`. Therefore, from the 115 INEX 2009 topics it is possible to identify 25 different structure queries that we called "templates" because a single structure query addresses a whole class of topics which has the same structural constraints but a different content part.

We divided the templates into three classes. In Table .8 we report the following information: the identifier of the template, the XPath of the template, the NESTOR primitive used to answer it, the INEX topics which are addressed by the template and some statistics about the retrieval part.

For example, the first template – i.e. `//article` – is the most common one and it accounts for about 68% of INEX 2009 topics; we can see that all the files in the collection return elements for this query because `article` is the tag of the root of all the XML documents in the Wikipedia collection. Templates 2 and 3 also return many elements, indeed they ask for very common elements in the collection – i.e. `sec` and `p`. We can see that the other templates return a smaller number of elements because they are more specific and regard only a fraction of the XML files in the collection; for instance, template 6 regards only the Wikipedia articles about music which is a small subset (i.e. about 4%) of the whole collection.

In Table .9 we can see the query templates about union descendants queries. They ask for all the descendants of several distinct XML elements; for instance, template 19 asks for elements about person or chemist or alchemist or scientist or physicist, so the result is given by a union of descendants elements. As we can see, template 20 returns a large number of elements because it asks for two very common elements in the collection (`sec` or `p`), whereas in the other cases the returned set is smaller because the required elements are more specific. In this case the logical operator "or" in the XPath expression translates in a set-theoretical union of NESTOR element-wise descendants primitive.

Lastly, in Table .10 we can see some statistics regarding intersect structure queries. In this case the cardinality of returned sets of elements tends to be small because the required elements are very specific and regard a small portion of the collection. As we can see, in these cases the role of structure query is preponderant in filtering out information not relevant to the topic.

Table .8: Structure query templates for descendants queries.

ID	Template	#ret elem	# files (%)	avg
1	XPath: `//article` NESTOR: `descendants(article)` INEX topics: 1-4, 7, 8, 10, 12-15, 18, 19, 21-27, 31-38, 42, 46-50, 52-56, 59-63, 69-71, 73-84, 87, 89-104, 107, 109, 113	744,064,683	2,666,190 (100%)	279.07
2	XPath: `//sec` NESTOR: `descendants(sec)` INEX topics: 20, 44, 51, 85, 88	520,221,285	2,164,231 (81%)	240.37
3	XPath: `//p` NESTOR: `descendants(p)` INEX topics: 64	607,708,666	2,584,699 (97%)	235.11
4	XPath: `//group` NESTOR: `descendants(group)` INEX topics: 39, 41, 43, 66	33,975,852	1,146,929 (43%)	29.62
5	XPath: `//facility` NESTOR: `descendants(facility)` INEX topics: 111	9,174,614	189,706 (7%)	48.36
6	XPath: `//song` NESTOR: `descendants(song)` INEX topics: 106	9,810,986	95,318 (4%)	102.93
7	XPath: `//driver` NESTOR: `descendants(driver)` INEX topics: 112	2,303,728	20,545 (1%)	112.13
8	XPath: `//protest` NESTOR: `descendants(protest)` INEX topics: 58	163,995	6,293 (0%)	26.06
9	XPath: `//dog` NESTOR: `descendants(dog)` INEX topics: 30	2,502,607	63,677 (2%)	39.30
10	XPath: `//food` NESTOR: `descendants(food)` INEX topics: 11	1,008,336	25,271 (1%)	39.90
11	XPath: `//bycicle` NESTOR: `descendants(bycicle)` INEX topics: 16	154,819	16,784 (1%)	9.22
12	XPath: `//vehicles` NESTOR: `descendants(vehicles)` INEX topics: 28	1,123	801 (0%)	1.40
13	XPath: `//personality` NESTOR: `descendants(personality)` INEX topics: 29	1,471,164	22,050 (1%)	66.72
14	XPath: `//theory` NESTOR: `descendants(theory)` INEX topics: 110	991,775	24,305 (1%)	40.81
15	XPath: `//museum` NESTOR: `descendants(museum)` INEX topics: 115	2,280,387	82,581 (3%)	27.61
16	XPath: `//home` NESTOR: `descendants(home)` INEX topics: 17	38,461	2,357 (0%)	16.32
17	XPath: `//music_genre` NESTOR: `descendants(music_genre)` INEX topics: 12, 105	1,630,346	107,506 (4%)	15.17

Table .9: Structure query templates for union descendants queries.

ID	**Template**	**#ret elem**	**# files** (%)	**avg**
18	XPath: `//(classical_music | opera | orchestra | performer)` NESTOR: `descendants(classical_music)` ∪ `descendants(opera)` ∪ `descendants(orchestra)` ∪ `descendants(performer)` INEX topics: 6	15,554,795	331,259 (12%)	46.97
19	XPath: `//(person | chemist | alchemist | scientist | physicist)` NESTOR: `descendants(person)` ∪ `descendants(chemist)` ∪ `descendants(alchemist)` ∪ `descendants(scientist)` ∪ `descendants(physicist)` INEX topics: 5	159,310,613	1,401,408 (53%)	113.68
20	XPath: `//(p | sec)` NESTOR: `descendants(p)` ∪ `descendants(sec)` INEX topics: 9	625,726,431	2,585,836 (97%)	241.98

Table .10: Structure query templates for intersection descendants queries.

ID	**Template**	**#ret elem**	**# files** (%)	**avg**
21	XPath: `//painter//figure` NESTOR: `descendants(painter)` ∩ `descendants(figure)` INEX topics: 114	18,075	12,346 (0%)	1.46
22	XPath: `//aircraft//sec` NESTOR: `descendants(aircraft)` ∩ `descendants(sec)` INEX topics: 86	2,163,601	27,339 (1%)	79.14
23	XPath: `//movie//director` NESTOR: `descendants(movie)` ∩ `descendants(director)` INEX topics: 57	621,546	71,378 (3%)	8.71
24	XPath: `//(information | artifact)//p` NESTOR: `[descendants(information)` ∪ `descendants(artifact)]` ∩ `descendants(p)` INEX topics: 67	44,489,868	878,435 (33%)	50.65
25	XPath: `//(person | scientist)/link` NESTOR: `[descendants(person)` ∪ `descendants(scientist)]` ∩ `descendants(link)` INEX topics: 72	25,830,674	1,365,840 (51%)	18.91