Fast Access to XML Data: A Set-Based Approach*

Nicola Ferro and Gianmaria Silvello

Department of Information Engineering, University of Padua, Italy
silvello@dei.unipd.it

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.

1 Introduction

The *eXtensible Markup Language (XML)* 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)* [6].

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

* Extended Abstract [5]

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 [4] 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, we focused this work only on those XPath fragments which, according to [2]: (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.

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.

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.

2 Background

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 [3,4,1].

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.

3 Data Structures and Primitives for Fast XML Access

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 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).

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

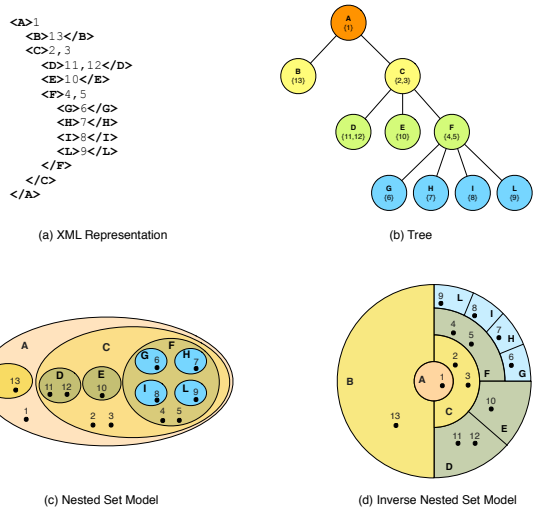

Fig. 1. (a) A sample XML representation; (b) Tree representation of the XML; (c) NS-M representation; (d) INS-M representation.

- **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.

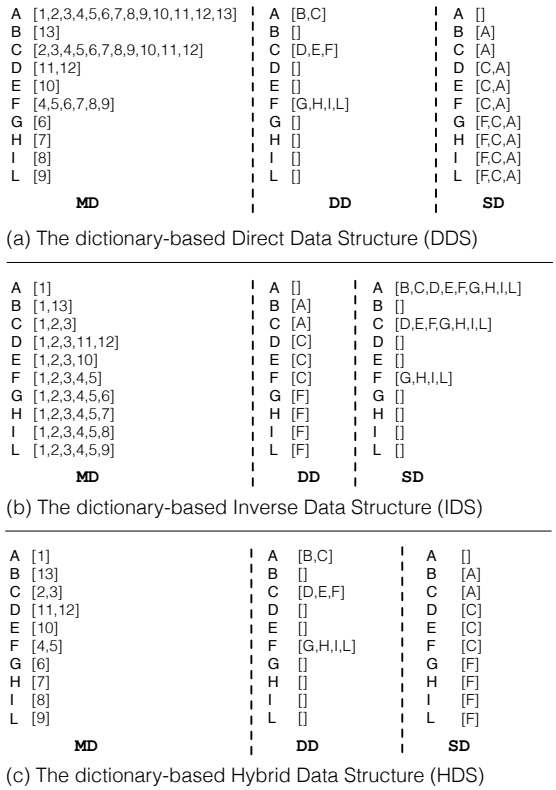

Fig. 2. NESTOR data structure instances of the tree shown in Figure 1.

`SD` reports for each set all its supersets – e.g. `F` contains $\texttt{G}, \texttt{H}, \texttt{I}, \texttt{L}$ 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 $\texttt{A} = [1]$ is the direct superset of $\texttt{C} = [1, 2, 3]$, thus in the `MD` of HDS it is $\texttt{C} = [2, 3]$.

3.1 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. 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

Table 1. 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**
	Cost	Cost	Cost
Descendants(H)	$O(m)$	$O(1)$	$O(m)$
Ancestors(H)	$O(1)$	$O(m)$	$O(m)$
Children(H)	$O(1)$	$O(m)$	$O(1)$
Parent(H)	$O(1)$	$O(1)$	$O(1)$

Table 2. 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**
	Cost	Cost	Cost
Elements(H)	$O(1)$	$O(1)$	$O(1)$
Descendants(H)	$O(1)$	$O(m+n)$	$O(m+n)$
Ancestors(H)	$O(m+n)$	$O(1)$	$O(m+n)$
Childrens(H)	$O(n)$	$O(m+n)$	$O(n)$
Parent(H)	$O(n)$	$O(n)$	$O(1)$

implementation, as DESCENDANTS-IDS when we refer to its IDS implementation and as DESCENDANTS-HDS when we refer to its HDS implementation.

In Table 1 we present the computational complexity of the algorithms implementing the set-wise primitives in the three data structures.

In Table 2 we present the references to the algorithms implementing the element-wise primitives in the DDS, IDS and HDS along with their computational complexities.

4 Experiments

NESTOR query primitives on the three proposed in-memory data structures have been implemented in the Java programming language[1]. 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. 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.

Table 3 reports the statistics about the two synthetic datasets we selected for the experiments.

In Figure 3 we report the evaluation times for the XPath operations we are analysing.

As a summary, 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.

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

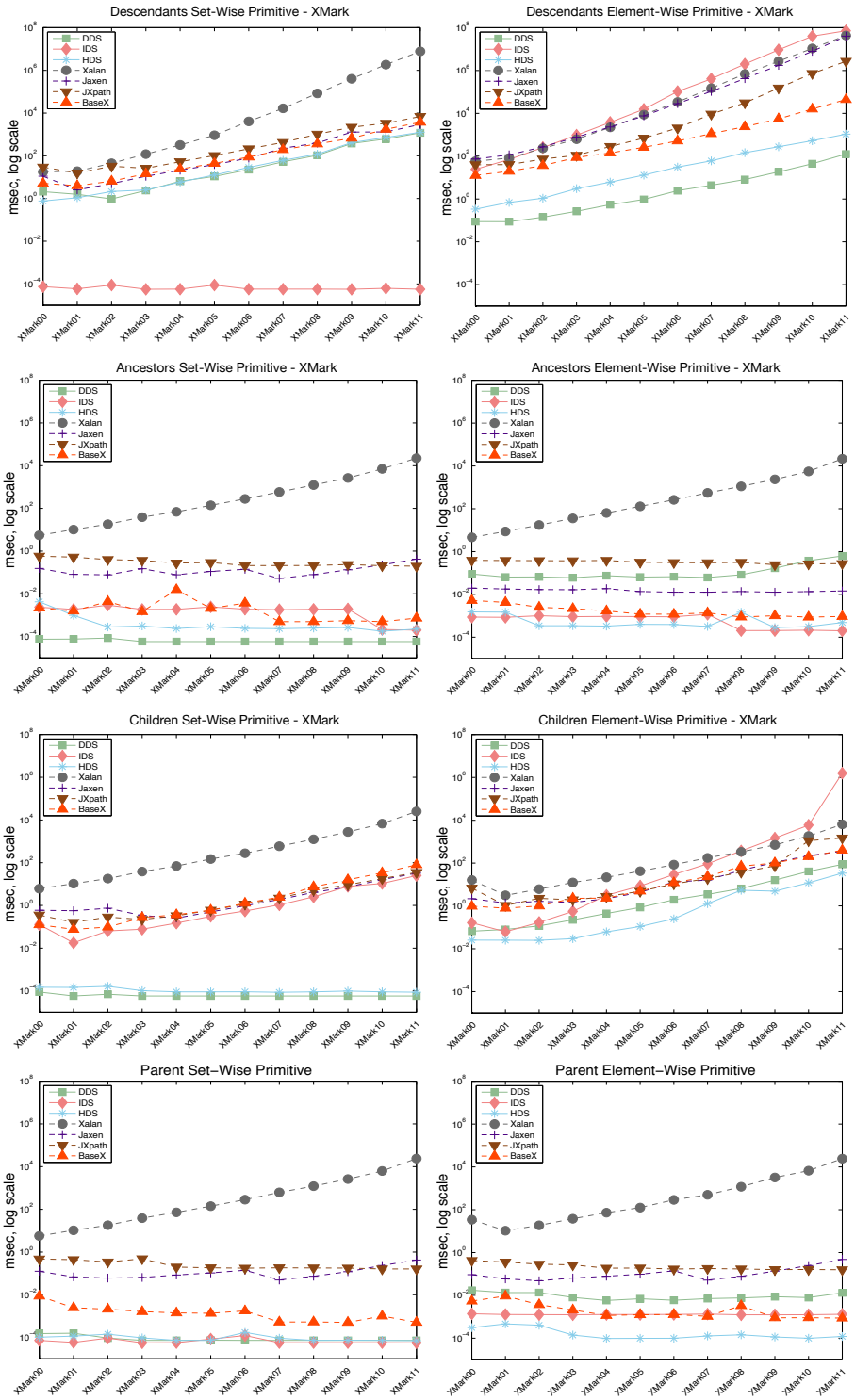

Fig. 3. Evaluation time for the XPath operations under evaluation.

Table 3. 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

5 Final Remarks

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. In particular, we have experimentally shown that NESTOR data structures allow us to outperform state-of-the-art solutions. The evaluation conducted on the XMark dataset 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.

As future work 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 [2]. Furthermore, we will study how NESTOR can be used with different levels of memory and extended to disk-oriented XML querying.

References

1. M. Agosti, N. Ferro, and G. Silvello. The NESTOR Framework: Manage, Access and Exchange Hierarchical Data Structures. In*Proc. 18th Italian Symposium on Advanced Database Systems (SEBD 2010)*, pages 242–253. Esculapio, 2010.
2. M. Benedikt, W. Fan, and G. M. Kuper. Structural Properties of XPath Fragments. *Theor. Comput. Sci.*, 336(1):3–31, 2005.
3. N. Ferro and G. Silvello. The NESTOR Framework: How to Handle Hierarchical Data Structures. In *Proc. 13th European Conference on Research and Advanced Technology for Digital Libraries (ECDL 2009)*, pages 215–226. LNCS 5714, Springer, Heidelberg, Germany, 2009.
4. N. Ferro and G. Silvello. NESTOR: A Formal Model for Digital Archives. *Inf. Process. Manage.*, 49(6):1206–1240, 2013.
5. N. Ferro and G. Silvello. Descendants, Ancestors, Children and Parent: A Set-Based Approach to Efficiently Address XPath Primitives. *Inf. Process. Manage.*, 52(3):399–429, 2016.
6. W3C. XML Path Language (XPath) 2.0 (Second Edition) – W3C Recommendation 14 December 2010. `http://www.w3.org/TR/xpath20/`, December 2010.