An Extended Preorder Index for Optimising XPath Expressions

Martin F O'Connor[1], Zohra Bellahsène[2], and Mark Roantree[1]

[1] Interoperable Systems Group, Dublin City University, Ireland. Email: {moconnor,mark.roantree}@computing.dcu.ie
[2] LIRMM, UMR 5506 CNRS Université Montpellier II, France. Email: bella@lirmm.fr

Abstract. Many of the problems with native XML databases relate to query performance and subsequently, it can be difficult to convince traditional database users of the benefits of using semi- or unstructured databases. Presently, there still lacks an index structure providing efficient support for structural queries and the traditional data-centric and content queries. This paper presents an extended index structure based on the preorder traversal rank and the level (or depth) rank of each node in a document tree. The extended index fully supports the navigation of all XPath axes while efficiently supporting data-centric queries. The ability to start path traversals from arbitrary nodes in a document tree also enables the extended index to support the evaluation of path traversals embedded in XQuery expressions. Furthermore, an encoding technique is presented where properties of the level ranking may be exploited to provide efficient and optimised level-based XPath evaluations.

1 Introduction

XML has been adopted as the new standard for data exchange on the World Wide Web and increasingly so in industry as the standard data interchange format. The key ingredient to its successful adoption is the expressive and extensible nature of XML. The basic structure underlying XML is the tree, which represents semi-structured data. Semi-structured data consists of an irregular and non-uniform organisation; it may have data with missing attributes and some attributes may be of different types within different data items. All of these variations are acceptable in XML documents. Thus, it may be seen that XML provides for an unlimited number for tree dialects, some of which have been formally described by Document Type Descriptors (DTDs) or XML Schemas, while others are employed in an ad-hoc schema-less manner. The database community is well advanced in adapting its technology to host large XML collections and to query these collections efficiently. It will be essential, though, that these new technologies support the XML query language specifications such as XPath [13] and XQuery [12]. These specifications are key enablers in maintaining the interoperability among XML repositories.

1.1 Motivation

The operations and path traversals required in the querying of tree structured data present difficult challenges. There has been much activity on the specification and provision of extensions to existing indexing mechanisms and processing models to enable the efficient exploitation of the structural properties of XML. The goal of this activity is to support, not only rapid navigational or structural queries but efficient content-based queries as well [1] [15]. There have also been several proposals [5] [7] for new index structures to deal with these problems. However, virtually all of the proposals focus on support for step evaluation along the child and descendant-or-self axes, to the detriment of the remaining XPath axes. Moreover, these proposals often rely on query processing algorithms which call for implementation techniques that lie outside their natural domain. An example is the relational domain where such proposals incur associated drawbacks such as additional software layers and transactional and performance issues. Indeed, as trees in their abstract form may be queried using path expressions, the XPath language was defined to model and query an XML document as a tree of nodes. The XQuery specification moreover facilitates embedded path traversals that may commence from any arbitrary node. Presently, there still lacks an index structure facilitating embedded XPath traversals from arbitrary nodes while providing at the same time, efficient and optimised XPath traversal evaluations incorporating both structural and navigational queries and the traditional content and data-centric queries. Our PreLevel Index structure fills this gap.

1.2 Contribution

In this paper, we present a new tree encoding mechanism based solely on the preorder traversal rank and the level (or depth) rank of each node in the document tree. We define new conjunctive range predicates based on our tree encoding to support the evaluation of location steps along the principle XPath axes and provide proofs to validate them. We then present an Extended Index structure (hereafter, referred to as the PreLevel Structure) based on our tree encoding that fully supports all XPath axes. Both the preorder traversal rank and level rank values may be determined during the initial parsing of the XML document and thus, the PreLevel Structure has minimal computational overhead associated with its construction. The ability to start traversals from arbitrary context nodes in an XML tree also enables the PreLevel Structure to support the evaluation of path traversals embedded in XQuery expressions. Furthermore, using our PreLevel Structure, the properties of the level rank of a node may be exploited to provide efficient and optimised level-based XPath evaluations.

The paper is organised as follows: Section 2 reviews the *partition* property of the XPath language and presents our PreLevel encoding and the newly derived conjunctive range predicates that facilitate XPath axis navigation, together with formal proofs of their derivation. Section 3 presents the tabular representation of the PreLevel Structure, explaining its construction and illustrating an evaluation of a step location along the descendant axis. Section 4 highlights various features

of the PreLevel Structure and outlines some of the optimised XPath queries possible. Section 5 reviews related work and we conclude in Section 6.

2 Presenting the PreLevel Structure Encoding

In this section, the XPath partition property is reviewed and the PreLevel encoding mechanism is introduced. For each of the *primary* XPath axes, the new conjunctive range predicates for performing a location step along the axis are presented and the corresponding proofs provided. The conjunctive range predicates have been derived from the intrinsic properties of the *preorder* traversal ranks and *level* ranks alone.

2.1 XPath Partition Property

The basic data type underlying XML is the tree. Thus, the XPath language was defined to model and query an XML document as a tree of nodes. The XPath 2.0 working draft [13] also specifies the following partitioning property: the **ancestor**, **descendant**, **preceding**, **following** and **self** axes partition an XML document (ignoring attribute and namespace nodes), partitions are disjoint and together they contain all nodes in the XML document. Thus, as a given context node resides in the **self** axis, all other nodes in the XML document fall into one of four partitions, as identified by the axes specified above (hereafter referred to as the *primary* axes).

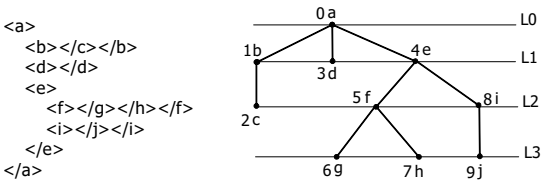
2.2 The PreLevel Encoding

The PreLevel structural index is an extension to the XPath Accelerator presented in [2]. The PreLevel encoding is based solely on the *preorder traversal* rank encoding and a *level* rank encoding. The *size* information is not recorded as in the encoding mechanism in [3]. The *level* (or depth) function takes one parameter, a node, and returns the *level* rank value of the node. Figure 1(a) depicts a sample XML document and Figure 1(b) depicts the corresponding XML tree with a *preorder* and *level* rank encoding.

Thus, *level(v)* $= m$ if the path from the root of the tree to the node v has length m; for example, *level(a)* $= 0$ and *level(f)* $= 2$. The *XPath Partition* property introduced in Section 2.1 is preserved by the combined *preorder traversal* and *level* rank encoding. The remaining XPath axes (**parent**, **child**, **descendant-or-self**, **ancestor-or-self**, **following-sibling** and **preceding-sibling**) determine either supersets or subsets of one of the *primary* axes and may be evaluated from them.

2.3 Navigating the Descendant Axis

The **descendant** axis selects all children of the given context node, and their children recursively, with the resulting nodes in document order [13]. The new



(a) Sample XML document

(b) PreLevel encoded XML tree.

Fig. 1. Sample XML document and associated PreLevel encoded tree.

conjunctive range predicate defining a location step along the **descendant** axis, based on the PreLevel encoding, is as follows:

Lemma 1.

$$
\begin{aligned}
v \in c/\textbf{descendant} \Leftrightarrow\ & pre(v) > pre(c) && (i)\\
\wedge\ & level(v) > level(c) && (ii)\\
\wedge\ & \forall x : pre(x) \in (pre(c)\,,\, pre(v)) && (iii)\\
& \Rightarrow\ level(x) \neq level(c)
\end{aligned}
$$

Lemma 1 states that an arbitrary node v is a **descendant** of a given context node c if and only if:

(i) the *preorder* rank of v is greater than the *preorder* rank of c, and
(ii) the *level* rank of v is greater than the *level* rank of c, and
(iii) for all nodes (let us label them x) having a *preorder* rank greater than *pre(c)* and less than *pre(v)*, that none of those nodes have a *level* rank the same as *level(c)*.

Proof: Condition *(i)* ensures that the *preorder* rank of node v is greater than the *preorder* rank of the context node c. In essence, the first condition exploits the properties of *preorder traversal* to ensure that the arbitrary node v appears, in document order after the given context node c. Condition *(ii)* ensures the *level* rank of node v is greater than the *level* rank of node c. Conditions *(i)* and *(ii)* are intuitive if node v is to be a **descendant** of node c. The third condition ensures that node v does not have another **ancestor** at the same *level* as the given context node c. If there is another **ancestor** at the same *level* as the context node c, then the context node could not be the **ancestor** of node v. This can be stated with certainty due to the properties of *preorder traversal* - namely that a node is visited immediately before its children, and the children are visited from left to right. So, if there is another node at the same *level* as node c, then that node must have a higher *preorder* rank than node c but also a *preorder* rank less than node v (the range requirement of condition *(iii)* ensures this). Thus, although the identity of the **ancestor** at *level(c)* has not been definitely established, it has been definitively determined that the **ancestor** of node v cannot be node c - by finding any other node at the same *level* and within the

range specified. Only if there is no node at the same *level* as the context node c and within the range specified, can it be stated with certainty that the context node c is an **ancestor** of node v, and conversely that node v is a **descendant** of the context node c.

An illustration of Lemma 1 now follows. While referring to the conjunctive range predicate in Lemma 1 and to the illustration in Figure 2; let v = node h; let c = node e. To determine if node h is a **descendant** of the context node e, one must examine the conditions:

(i) Is *pre(h)* > *pre(e)*...(7 > 4)...condition holds true.
(ii) Is *level(h)* > *level(e)*...(3 > 1)...condition holds true
(iii) For all nodes whose *preorder* rank is greater than *pre(e)* and less than *pre(h)*, these nodes are located within the shaded area in Figure 2, do any of them have a *level* rank the same as *level(e)*, in this case 1? No, they do not and therefore, the condition holds true.

All three conditions are true, thus node h is a **descendant** of the context node c.

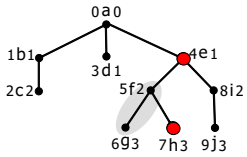


Fig. 2. Example of navigating the descendant axis of a PreLevel encoded XML tree.

Now, let us take an example whereby the conjunctive range predicate will return false. By following the above example, but assigning node d to be the context node c, conditions *(i)* and *(ii)* hold true, but condition *(iii)* fails because node e has the same *level* rank as node d.

2.4 Navigating the Ancestor Axis

The **ancestor** axis selects all nodes in the document that are ancestors of a given context node [13]. Thus, the new conjunctive range predicate defining a location step along the **ancestor** axis, based on the PreLevel encoding, is:

Lemma 2.

$$\begin{aligned} v \in c/\textbf{ancestor} \Leftrightarrow\ & pre(v) < pre(c) && (i)\\ \wedge\ & level(v) < level(c) && (ii)\\ \wedge\ & \forall x : pre(x) \in (pre(v)\,,\, pre(c)) && (iii)\\ & \Rightarrow level(x) \neq level(v) \end{aligned}$$

Lemma 2 states that an arbitrary node v is an **ancestor** of a given context node c if and only if:

(i) the *preorder* rank of v is less than the *preorder* rank of c, and
(ii) the *level* rank of v is less than the *level* rank of c, and
(iii) for all nodes (let us label them x) having a *preorder* rank greater than *pre(v)* and less than *pre(c)*, that none of those nodes have a *level* rank the same as *level(v)*.

Proof: Condition *(i)* exploits the properties of *preorder traversal* to ensure the arbitrary node v appears in document order before the given context node c. Condition *(ii)* exploits the *level* rank properties to ensure node v appears higher in the document tree than node c . Condition *(iii)* ensures that the given context node c does not have another **ancestor** at the same *level* as node v. If there is any other node at the same *level* as node v, then node v could not be the **ancestor** of the context node c. This can be stated with certainty due to the properties of *preorder traversal* - namely that a node is visited immediately before its children, and the children are visited from left to right. So, if there is another node at the same *level* as node v, then that node must have a higher *preorder* rank than node v but also a *preorder* rank less than the context node c (the range requirement of condition *(iii)* ensures this). Only if there is no node at the same *level* as node v and within the range specified, can it be stated with certainty that node v is an **ancestor** of the context node c.

2.5 Navigating the Preceding Axis

The **preceding** axis selects all nodes in document order that appear before the given context node, excluding all **ancestors** of the context node [13]. The new conjunctive range predicate, based on the PreLevel encoding, defines a location step along the **preceding** axis as follows:

Lemma 3.

$$\begin{aligned} v \in c/\textbf{preceding} \Leftrightarrow\ & pre(v) < pre(c) && (i)\\ \wedge\ & \exists x : pre(x) \in (pre(v)\,,\, pre(c)] && (ii)\\ & \Rightarrow level(x) \in (0\,,\, level(v)] \end{aligned}$$

Lemma 3 states that an arbitrary node v is member of the **preceding** axis of a given context node c if and only if:

(i) The *preorder* rank of v is less than the *preorder* rank of c, and
(ii) There exists a node (let us label it x) whose *preorder* rank is greater than *pre(v)* and less than or equal to *pre(c)*, and that the *level* rank of x is less than or equal to *level(v)*.

Proof: Condition *(i)* exploits the properties of *preorder traversal* to ensure the arbitrary node v appears, in document order, before the given context node c. Condition *(ii)* ensures that node v is not an **ancestor** of the context node c. Due to the properties of *preorder traversal,* the existence of any other node which has a *preorder* rank greater than *pre(v)* and less than or equal to *pre(c)*, and which has a *level* rank less than or equal to node v, rules out any possibility that node v is the **ancestor** of node c. Thus, conditions *(i)* and *(ii)* together ensure that an arbitrary node v is a member of the **preceding** axis of given context node c.

2.6 Navigating the Following Axis

The **following** axis selects all nodes that appear after the given context node in document order, excluding the **descendants** of the context node [13]. The new conjunctive range predicate defining a location step along the **following** axis based on the PreLevel encoding is:

Lemma 4.

$$\begin{aligned} v \in c/\textbf{\textit{following}} \Leftrightarrow\ & pre(v) > pre(c) && (i)\\ \wedge\ & \exists x : pre(x) \in (pre(c)\,,\, pre(v)] && (ii)\\ & \Rightarrow level(x) \in (0\,,\, level(c)] \end{aligned}$$

Lemma 4 states that an arbitrary node v is member of the **following** axis of a given context node c if and only if:

(i) The *preorde*r rank of v is greater than the *preorder* rank of c, and
(ii) There exists a node (let us label it x) whose *preorder* rank is greater than *pre(c)* and less than or equal to *pre(v)*, and that the *level* rank of x is less than or equal to *level(c)*.

Proof: Condition *(i)* exploits the properties of *preorder traversal* to ensure the arbitrary node v appears in document order after the given context node c. Condition *(ii)* ensures that node v is not a **descendant** of the context node c. The second condition is validated by verifying that there is another node, with a *preorder* rank greater than *pre(c)* and less than or equal to *pre(v)*, and which has a *level* rank less than or equal to the *level* rank of the context node c. If any such node exists, then due to the properties of *preorder traversal* - namely that a node is visited immediately before its children and the children are visited from left to right - the context node c cannot be the **ancestor** of node v, and conversely node v cannot be the **descendant** of the context node c. Thus, conditions *(i)* and *(ii)* together ensure that an arbitrary node v is a member of the **following** axis of given context node c.

3 Extended Index Structure.

In this section we present a tabular representation for the PreLevel encoding that facilitates optimised algorithms for the efficient evaluation of XPath expressions. We adapt the tabular encoding of the *XPath Accelerator* originally proposed in [2] and extend it to incorporate our Extended Preorder Index and Level Index.

3.1 Tabular Encoding

The PreLevel encoding facilitates a tabular representation of XML documents, namely the *Extended Preorder Index*. The primary column of the Extended Preorder Index consists of the *preorder* ranks sorted in ascending order. The second column contains the *level* ranks that correspond to the associated *preorder* ranks of the primary column. Extra columns may be added to the Extended Preorder

Index to hold further node properties as defined by the XPath/XQuery data model, such as name, node type (node, element, attribute, comment) and more. In particular, to support the **parent** axis in our tabular encoding, we add a column containing the parent's *preorder* rank of each node to the Extended Preorder Index. However, in order to efficiently evaluate an XPath location step along all of the XPath axes, a second index is required. This second index is introduced (hereafter referred to as the *Level Index*) and consists of two columns only, the *level* rank column and the *preorder* rank column. The first column in the Level Index is the *level* rank column, sorted in ascending order, the second column being the *preorder* rank column, again sorted in ascending order. The Extended Preorder Index and Level Index combined may also be referred to as the *PreLevel Structure*. Several observations should be made at this point.

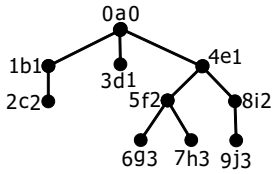
- Both the *preorder* ranks and the *level* ranks may be determined during the initial parsing of the XML document tree, and thus have minimal computational overheads associated with them.
- Each node in the XML tree has a single *preorder* rank and a single *level* rank associated with it. Thus, the Extended Preorder Index contains a one-to-one mapping. However, as many nodes may reside at the same *level*, the Level Index contains a one-to-many mapping - it is an inverted index.
- Both the Extended Preorder Index and the Level Index can be constructed in parallel during the initial parsing of the XML document tree. The act of parsing of an XML document (reading from top to bottom and left to right) corresponds to a *preorder traversal*. Thus, the Extended Preorder Index is constructed in a sorted list, sorted on the *preorder* rank in ascending order. It may not be obvious that the Level Index is also constructed in a sorted list. When the *preorder traversal* begins, the *level* information is recorded also (*level* 0 for the root node). As the *preorder traversal* progresses, all new *levels* and the associated *preorder* ranks are recorded. As the *preorder traversal* encounters nodes on a *level* already recorded, the *preorder* ranks are simply appended to the list of existing *preorder* ranks at that *level*. Thus, depending on the structure used at implementation time, for example a linked list, when the *preorder traversal* has been completed, we are left with a column of unique *level* ranks, sorted in ascending order with each *level* rank pointing to a linked list of *preorder* ranks and each linked list also sorted in ascending order.
- Lastly, in order to facilitate a lookup of the Level Index in constant time, a *position* column is included in the Extended Preorder Index. During the construction of the Level Index, before any *preorder* ranks have been inserted, each *level* is assigned a counter initialised to zero. As a *preorder* rank is added (or appended) to the Level Index, the counter at that *level* is incremented by one and its value is written in the *position* column of the Extended Preorder Index, in the row of the related *preorder* rank. Thus, the *position* value, when obtained using a lookup of the Extended Preorder Index, facilitates a direct jump to a given *preorder* rank within the Level Index in constant time. The *position* column is the key to enabling the evaluation of location steps on

the *primary* XPath axes in constant time and to the optimised evaluations of *level-based* queries (to be introduced in §4.2).

The main issue is to compute the conjunctive range predicates for each of the XPath *primary* axes in *constant time*. This is demonstrated in Section 3.2.

3.2 Example of an Evaluation along the Descendant Axis

The sample PreLevel encoded tree and the corresponding PreLevel Structure, are illustrated in Figure 3. A high level algorithm detailing the steps to evaluate a location step along the **descendant** axis in constant time is provided in Algorithm 1.



(a) A PreLevel encoded XML tree

Pre	Level	Pos	Parent	Kind	Name	...
0	0	1		Elem	A	
1	1	1	0	Elem	B	
2	2	1	1	Elem	C	
3	1	2	0	Elem	D	
4	1	3	0	Elem	E	
5	2	2	4	Elem	F	
6	3	1	5	Elem	G	
7	3	2	5	Elem	H	
8	2	3	4	Elem	I	
9	3	3	8	Elem	J	

(b) Extended Preorder Index

Level	Pre
0	0
1	1, 3, 4
2	2, 5, 8
3	6, 7, 9

(c) Level Index

Fig. 3. Sample XML tree and the corresponding PreLevel Structure.

Let us now illustrate Algorithm 1. Let v = node h; let c = node e; nodes are represented by their *preorder* rank. It can be verified that *pre(h)* is greater than *pre(e)* (i.e. 7>4), and that *level(h)* is greater than *level(e)* (i.e. 3>1). The Level Index is used to identify the next *preorder* rank greater than *pre(e)* at *level(e)* (i.e. *null*). This information is obtained in constant time as the *position* column of the Extended Preorder Index facilitates a direct jump to *pre(e)* within the *level(e)* index. Note, the next *preorder* rank greater than *pre(e)* at *level(e)*, should it exist, must appear immediately after *pre(e)* because the index is sorted in ascending order. If the next *preorder* rank after *pre(e)* at *level(e)* is greater than *pre(h)*, the node being tested, then node h must be a **descendant** of node e. This can be stated with certainty as the properties of *preorder traversal* require a node's children to be visited immediately after its parent. Also, as in this case, if there are no *preorder* ranks greater than *pre(e)* at *level(e)*, indicated with *null*, node h must be a **descendant** of node e. The fact that there may be no *preorder* ranks greater than *pre(e)* at *level(e)* simply means that node e is the root node of the rightmost subtree rooted at *level(e)*.

Algorithm 1 To determine if an arbitrary node v is a **descendant** of a given context node c.

Name: IsNodeDescendant
Given: An arbitrary node v, a context node c.
Returns: Boolean (TRUE or FALSE)
begin
  //Using the Extended Preorder Index
  if (*pre(v)* <= *pre(c)*) **or** (*level(v)* <= *level(c)*) **then**
    return FALSE;
  endif
  //Using the Level Index
  next_pre := next preorder rank after *pre(c)* at *level(c);*
  if (next_pre > *pre(v)*) **or** (next_pre == **null**) **then**
    return TRUE;
  else
    return FALSE;
  endif
end

This subsection has illustrated an evaluation of a location step along the **descendant** axis in constant time, however an evaluation along the **ancestor** axis in constant time may be illustrated in a similar fashion by adapting the algorithm appropriately. An evaluation along the **following** and **preceding** axes may also be evaluated in constant time however lack of space prevents this demonstration here but may be referenced in [8].

4 Optimised XPath Queries

The PreLevel Structure enables an efficient encoding mechanism that supports highly optimised structural and navigational queries as well as content and data-centric queries.

4.1 Evaluating the Size of a Subtree

Using our PreLevel Structure, the size of a subtree tree rooted at an arbitrary node v can be determined very efficiently. The evaluation of the subtree size is independent of the actual size of the subtree (and indeed the size of the entire document tree) but rather dependent on the number of levels between the given node v and the root node of the entire document tree. In [6], a comprehensive study of over 190,000 XML trees was performed revealing that 99% of all the documents had less than 8 levels. The vast majority of the remaining 1% of documents had less than 30 levels, with only a tiny fraction having more than 30 levels. Thus, it may be seen that the number of levels (or depth) in an XML tree is sufficiently small so as to be deemed to have a minimal computational impact on our evaluation. The size of the subtree evaluated with our algorithm is accurate and no extra information beyond the *preorder* and *level* ranks are

Algorithm 2 To determine the size of subtree rooted at an arbitrary node v

```
Name:    SizeOfSubtree
Given:   An arbitrary node v,
         The maximum preorder rank in document tree max_pre.
Returns: subtree_size
begin
  //Using the Extended Preorder Index, determine if node v is a leaf node
  if (level(pre(v) + 1) <= level (v)) then subtree_size := 1;
    return subtree_size;
  endif
  //Using the Level Index
  next_pre := next preorder rank after pre(v) at level(v);
  //limit will contain the maximum upper preorder rank of the preorder interval (non-inclusive)
  //specifying the subtree nodes.
  limit := next_pre;
  init_level := level(v) - 1;
  //par(v) returns the preorder rank of the parent node of v
  par_pre := par(v);
  //For each level between level(v) and root node, find first node with preorder rank > pre(v)
  for (count = init_level; count > 0; count --)
    next_pre := next preorder rank after par_pre at level(par_pre);
    if (limit != null) then
      if (next_pre != null) and (next_pre < limit) then limit := next_pre;
      endif
    endif
    par_pre := par(par_pre);
  endfor
  if (limit != null) then subtree_size := limit - pre(v);
  else subtree_size := (max_pre - pre(v)) + 1;
  endif
  return subtree_size;
end
```

necessary to determine the size of the subtree. A more detailed explanation of this algorithm may be found in [8]. An algorithm demonstrating the steps required to evaluate the size of a subtree rooted at an arbitrary node v using our PreLevel Structure is provided in Algorithm 2.

The *SizeOfSubtree* function facilitates the efficient evaluation of *all members* of the **descendant** and **following** axes of a given node v. By exploiting the *parent* column in the Extended Preorder Index we can also very efficiently evaluate *all members* of the **ancestor** and **preceding** axes for any given arbitrary node v. The remaining XPath axes (**parent**, **child**, **descendant-or-self**, **ancestor-or-self**, **following-sibling**, and **preceding-sibling**) determine either supersets or subsets of one of the *primary* axes and may be evaluated from them.

4.2 Optimised Level-based Queries

The PreLevel Structure makes a notable contribution to the efficient processing of XPath expressions by facilitating optimised evaluations of *level-based* queries.

A *level-based* query is such that the results of the query reside at a particular level in the XML tree.

Taking the **descendant** axis as an example, all nodes that are a **descendant** of an arbitrary node v will reside in a *preorder-defined* interval, delimited by lower and upper *preorder* ranks. Thus, using our Level Index, it is easy to identify a sequence of nodes residing at a particular *level* that belong to a *preorder-defined* interval. For example, given a query to select all grandchildren of an arbitrary node v; the result of such a query will be represented using the Level Index as an interval or array with lower and upper *preorder* bounds residing at a specific *level*. The *position* column of the Extended Preorder Index facilitates a direct jump to the lower and upper *preorder* bounds within the Level Index.

The Level Index is sorted in ascending order and can be searched very efficiently using a binary search algorithm with a time complexity of $O(lg\ n)$. The lower bound of the *preorder* interval containing node v's **descendants** at a given level l, is obtained by performing a binary search at level l for the first *preorder* rank greater than *pre(v)*. In a similar fashion, the upper bound of the *preorder* interval containing node v's **descendants** at a given level l, is obtained by performing a binary search at level l for the last *preorder* rank preceding a *container preorder* rank of node v's **descendants**. A *container preorder* rank is a *preorder* rank greater than the largest *preorder* rank in node v's **descendants**. Due to the properties of *preorder traversal*, a valid *container preorder* rank for node v's **descendants** is the next *preorder* rank greater than *pre(v)* at *level(v)*. The *container preorder* rank can be obtained in constant time using a lookup of the Level Index and provides an upper bound for node v's **descendants** at an arbitrary level l.

Thus, given the *preorder* rank of a context node, the upper and lower bounds of the interval containing the context node's **descendants** at an arbitrary level l can be obtained using the Level index, requiring only two lookup operations of time complexity $O(lg\ n)$ each, at level l. The processing of nodes at intermediary levels is unnecessary for all levels between the context node and the level to be queried.

The optimal time complexity for reading n values from an array of size n is linear, i.e. *O(n)*. Thus, given that the results of a *level-based* query is an array subset of the Level Index, which is always sorted in document order; and given that the *position* column of the Extended Preorder Index facilitates a direct jump to the lower and upper *preorder* bounds within the Level Index; when both lower and upper bounds of the interval have been obtained, the actual results of the *level-based* query may be retrieved in *optimal* time. Indeed, once the interval is know, the solution is optimal for retrieving all **descendants** of a given node v that reside at an arbitrary level l. A sample algorithm to evaluate all **descendants** of a given node v residing at an arbitrary level l is provided in Algorithm 3.

In a similar fashion, the solution for identifying all members of the **following-sibling** and **preceding-sibling** axes are also optimal. It should be noted that queries along the **descendant**, **descendant-or-self** and **child** axes of an ar-

Algorithm 3 To determine all the **descendants** of an arbitrary node v at a given level m

Name: AllDescendantsAtLevelM
Given: An arbitrary node v,
The maximum preorder rank in document tree *max_pre*,
A level m, where m is the path length from root node to node v.
Returns: A sequence of document nodes labelled *descendants* or the *empty_sequence*

```
begin
  //Using Extended Preorder Index, determine if v is a leaf node
  if (level(pre(v) + 1) <= level (v)) then
    return empty_sequence;
  endif
  //Using the Level Index
  next_pre := next preorder rank after pre(v) at level(v);
  //Convert relative level rank to absolute level rank of document tree.
  queryLevel := level(v) + m;
  if (next_pre != null) then
    start_pre := next preorder value > pre(v) at queryLevel;
    descendants = all nodes in interval [start_pre , next_pre) at queryLevel;
  else
    descendants = all nodes in interval (pre(v) , max_pre] at queryLevel;
  endif
  return descendants;
end
```

bitrary node constitute the core of XPath subexpressions embedded in XQuery statements and provide the most challenging and highly computational tasks for XPath/XQuery processors.

5 Related Work

In [11], the experience of building Jungle, a secondary storage manager for Galax, an open source implementation of the family of XQuery 1.0 specifications is presented. They chose to implement the Jungle XML indexes using the *XPath Accelerator*. However, one significant limitation they encountered was the evaluation of the **child** axis, which they found to be as expensive as evaluating the **descendant** axis. They deemed this limitation to be unacceptable and designed their own alternative indexes to support the **child** axis. Although the XPath Accelerator *pre/post* encoding scheme has since been updated in [3] to use *pre/level/size,* which Jungle has yet to incorporate, our PreLevel Structure as demonstrated in Section 4.2 supports highly efficient evaluations of not just children, but grandchildren and indeed all nodes at any particular level of an arbitrary node. The ability to efficiently evaluate level-based queries by considering only the nodes at the level concerned and eliminating the need for large scale processing at the intermediary levels, is the principle contribution of the PreLevel Structure. The Jungle implementation experience also highlighted the significant overhead imposed at document loading time by a *postorder traversal*,

a necessary component in the construction of the indexing system proposed in [14].

There has been much research into the development and specification of new indexing structures to efficiently exploit the properties of XML. There have been several initiatives to extend the relational data model to facilitate the XML data model and once again the XPath Accelerator has been at the forefront [4] [1] [15]. In [10], the key issue of whether the ordered XML data model can be efficiently represented by the relational tabular data model is examined and the authors propose three new encoding methods to support their belief that it can. In [5], a new system called XISS is proposed for indexing and storing XML data, specifying three new structures to support content queries, and a new numbering scheme, based on the notion of extended preorder to facilitate the evaluation of **ancestor-descendant** relationships between elements and attributes in constant time. In [9], a hierarchical labelling scheme called ORDPATH, implemented in the upcoming version of Microsoft SQL Server, is proposed. Each node on an XML tree is labelled with an ordinal value, a compressed binary representation of which, provides efficient document ordering evaluation as well as structural evaluation. In addition, the ORDPATH scheme supports insertion of new nodes in arbitrary positions in the XML tree, without requiring the re-labelling of any nodes.

6 Conclusion

There is an urgent need for an indexing structure capable of supporting very efficient structural, navigational and content-based queries over both document-centric and data-centric XML. Our PreLevel Structure makes a significant contribution toward this goal. In this paper we have presented a new tree encoding mechanism based solely on the *preorder traversal* rank and the *level* rank of a node. We constructed new conjunction range predicates based on the PreLevel encoding to support the evaluation of location steps along the *primary* XPath axes and provided proofs of their derivation. We then presented a tabular encoding for our PreLevel Structure - the Extended Preorder Index and Level Index - to enable the navigation of all XPath axes and demonstrated how these indexes have a minimal computational overhead associated with their construction. The tabular representation of the PreLevel Structure allows for flexible implementation strategies. Finally, accompanied by several algorithms, we detailed how our tabular encoding facilitates efficient XPath queries and expression evaluations. In particular, the properties of the Level index may be exploited to provide highly optimised level-based query evaluations as well as the optimal retrieval of their results.

As part of our future work, we are investigating the possibility of supporting efficient XML updates. In tandem with our research, we have short listed several open-source native XML databases and are examining them with a view to providing an implementation of our work to date.

References

1. Daniela Florescu and Donald Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.
2. Torsten Grust. Accelerating XPath Location Steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on the Management of Data*, volume 31, pages 109–120. SIGMOD Record, ACM Press, 2002.
3. Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, pages 252–263. Morgan Kaufmann, 2004.
4. Torsten Grust and Jens Teubner. Relational Algebra: Mother Tongue–XQuery: Fluent. In *1st Twente Data Management Workshop on XML Databases and Information Retrieval*. Enschede, The Netherlands, 2004.
5. Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th International Conference on Very Large Databases (VLDB)*, pages 361–370. Morgan Kaufmann, 2001.
6. Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. The XML Web: A First Study. In *Proceedings of the 12th International World Wide Web Conference (WWW2003)*, pages 500–510. ACM Press, 2003.
7. Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Proceedings of the 7th International Conference on Database Theory*, pages 277–295. LNCS 1540, Springer, 1999.
8. Martin O'Connor, Zohra Bellashène, and Mark Roantree. Level-based Indexing for Optimising XPath Expressions. Technical report, Interoperable Systems Group, Dublin City University, 2005. Available from: www.computing.dcu.ie/˜isg/technicalReport.html.
9. Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proceedings of the 2004 ACM SIGMOD International Conference on the Management of Data*, volume 33, pages 903–908. SIGMOD Record, ACM Press, 2004.
10. Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and Querying Ordered XML using a Relational Database System. In *Proceedings of the 2002 ACM SIGMOD International Conference on the Management of Data*, volume 31, pages 204–215. SIGMOD Record, ACM Press, 2002.
11. Avinash Vyas, Mary F. Fernández, and Jérôme Siméon. The Simplest XML Storage Manager Ever. In *Proceedings of the 1st International Workshop on XQuery Implementation, Experience and Perspectives <XIME-P/> in cooperation with ACM SIGMOD*, pages 37–42, 2004.
12. World Wide Web Consortium. *XQuery 1.0: An XML Query Language*, W3C Working Draft edition, April 2005.
13. World Wide Web Consortium. *XML Path Language (XPath) 2.0*, W3C Working Draft edition, February 2005.
14. Pavel Zezula, Giuseppe Amato, Franca Debole, and Fausto Rabitti. Tree Signatures for XML Querying and Navigation. In *Proceedings of the 1st International XML Database Symposium 2003*, pages 149–163. Springer, September 2003.
15. Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on the Management of Data*, volume 30, pages 425–436. SIGMOD Record, ACM Press, 2001.