# Supporting XPath Axes with Relational Databases Using a Proxy Index*

Olli Luoma

Department of Information Technology and Turku Centre for Computer Science
University of Turku, Finland
olli.luoma@it.utu.fi

**Abstract.** In recent years, a plethora of work has been done to develop methods for managing XML documents using relational databases. In order to support XPath or any other XML query language, the relational schema must allow fast retrieval of the parents, children, ancestors, or descendants of a given set of nodes. Most of the previous work has aimed at this goal using pre-/postorder encoding. Relying on this method, however, may lead to scalability problems, since the structural relationships have to be checked using nonequijoins, i.e., joins using $<$ or $>$ as their join condition. Thus, we discuss alternative methods, such as ancestor/descendant and ancestor/leaf indexes, and present a novel method, namely a so called proxy index. Our method allows us to replace nonequijoins with equijoins, i.e., joins using $=$ as their join condition. The results of our comprehensive performance experiments demonstrate the effectiveness of the proxy index.

## 1 Introduction

Because of its simplicity and flexibility, XML [1] has widely been adopted as the *lingua franca* of the Internet. Currently, XML is used not only as a platform-independent means to transfer data in computer networks, but also as a format to store large amounts of heterogeneous data in many modern application areas, such as bioinformatics [2]. It is thus widely agreed that without efficient means for managing XML documents, the potential of XML cannot be realized to its full extent, and the database community has been actively developing methods for storing and querying large amounts of XML data using XML query languages, most often XPath [3] and XQuery [4].

According to XPath [3], every well-formed XML document can be represented as an *XML tree*, a partially ordered tree with seven different node types. In addition to this tree representation, XPath also lists 12 *axes*[1], i.e., operators for tree traversal, such as `parent`, `child`, `ancestor`, `descendant`, `preceding`, and `following`. In order to provide adequate XPath support, all axes have to be implemented efficiently. The majority of the previous proposals, however,

* Supported by the Academy of Finland.

[1] In XPath 2.0, the `namespace` axis of XPath 1.0 was deprecated.

S. Bressan et al. (Eds.): XSym 2005, LNCS 3671, pp. 99–113, 2005.
© Springer-Verlag Berlin Heidelberg 2005

have considered only the `descendant` and `child` axes, and thus they lack the flexibility and generality intended by the original XPath recommendation.

In many previous proposals [5] [6] [7], the structural relationships between the nodes of an XML tree have been modeled using *pre-/postorder encoding*, i.e., by maintaining the pre- and postorder numbers of the nodes, or by using some similar method. This method forces us to check the nested relationships using expensive nonequijoins, which can easily lead to scalability problems [8] [9]. Thus, methods aiming at replacing nonequijoins with equijoins have been proposed; an *ancestor/descendant index* [10] maintains all ancestor/descendant pairs, an *ancestor/leaf index* [9] maintains the ancestor information only for the leaf nodes.

Maintaining the ancestor/descendant or ancestor/leaf information explicitly, however, consumes much more storage than pre-/postorder encoding. According to our previous experiments [9], a database based on ancestor/leaf approach consumes roughly twice and a database based on ancestor/descendant approach roughly four times the storage consumed by a database based on pre-/postorder encoding, which makes these methods somewhat impractical.

The main contributions of this paper can be listed as follows:

1. We present a novel method for modeling the structural relationships in XML trees, namely a *proxy index*. Our method supports all 12 XPath axes, provides good query performance, and encodes the structural information much more compactly than the ancestor/descendant and ancestor/leaf indexes.
2. We show that not only is it possible to support all axes of XPath using ancestor/descendant, ancestor/leaf, and proxy indexes, it is actually practical when large XML documents have to be queried efficiently. To support this claim, we present the results of our comprehensive performance experiments.

This paper proceeds as follows. In section 2, we review the related work and in section 3, we present the basics of the XPath query language; section 4 provides the reader with a more detailed discussion on the previous proposals. We present the proxy index in section 5 and the results of our performance evaluation in section 6. Section 7 concludes this article and discusses our future work.

## 2 Related Work

From a technical viewpoint, an XML management system (XMLMS) can be built in several ways. One option is to build a *native XML database*, i.e., to build an XML management system from scratch. This option allows one to design all features, such as query optimization and storage system, specifically for XML data. In the XML research literature, there are several examples of this approach, such as Lore [11] and NATIX [12].

Another approach followed, for instance, in [13] is to build an *XMLMS on top of an object-oriented database*. This approach allows us to take advantage of the rich data modeling capabilities of object-oriented databases. In this context, however, traversing large XML trees means traversing large object hierarchies, which can be a rigorous task, and thus the scalability of these systems

might be somewhat questionable [14]. Relational databases, on the contrary, provide a technically sound platform for data management, and thus building an *XMLMS on top of a relational database* is a viable option. This method allows the relational data and the XML data to coexist the same database, which is advantageous, since it is unlikely that XML databases can completely replace the traditional database technology in the near future. Unfortunately, there is some undeniable mismatch between the hierarchical XML and flat relational data models, and thus, after half a decade of XMLMS research, it still is not clear which one or ones of these approaches will prevail.

The previous proposals to manage XML data using relational databases can roughly be divided into two categories [5]. In the *structure-mapping approach* (*schema-aware approach*), the relational models are designed in accordance with the DTDs or the logical structure of the documents. The standard approach is to create one relation for each element type [14], but more sophisticated methods have also been proposed [15]. All of these methods, however, are at their best in applications where a large number of documents corresponding to a limited number of DTDs have to be stored. If this is not the case, we are at risk to end up with a very large number of relations, which complicates the XPath-to-SQL query translation [7] [16]. The other approach is the *model-mapping approach* (*schema-oblivious approach*) in which the database schemas represent the generic constructs of the XPath data model, such as element, attribute, and text nodes. Unlike in the structure-mapping approach, the database schema is fixed, and thus we can store any well-formed XML document without changing the schema. Having a fixed schema also simplifies the XPath-to-SQL query translation.

Thus far, the model-mapping approach has been followed in many proposals, such as XRel [5] (based on a variant of pre-/postorder encoding), XParent [10] (based on an ancestor/descendant index), and SUCXENT [17] (based on a variant of ancestor/leaf index). All of these methods, however, have only considered the `descendant` and `child` axes of XPath, and thus they provide a very limited XPath support. The XPath accelerator proposed by Grust [7] took a leap forward by providing support for all XPath axes and relative addressing. However, since XPath accelerator is based on pre-/postorder encoding, the nested relationships have to be checked using nonequijoins, and thus the scalability of the method may be somewhat in doubt [9].

## 3 XPath Axes

As mentioned earlier, the tree traversals in XPath are based on 12 axes which are presented in Table 1. These axes are used in *location steps* which, starting from a *context node*, result in a set of nodes in the relation defined by the axis with the context node. A location step of the form `axis::nodetest[predicate]` also includes a *node test* which can be used to restrict the name or the type of the selected nodes. An additional predicate can be used to filter the resulting node set further. For example, the location step $n$`/descendant::record[child::*]` selects all descendants of the context node $n$ which are named `record` and have one or more child nodes.

**Table 1.** The XPath axes and their semantics

| Axis | Semantics of $n$/Axis |
|---|---|
| `child` | Children of $n$. |
| `parent` | Parent of $n$. |
| `ancestor` | Transitive closure of `parent`. |
| `descendant` | Transitive closure of `child`. |
| `ancestor-or-self` | Like ancestor, plus $n$. |
| `descendant-or-self` | Like descendant, plus $n$. |
| `preceding` | Nodes preceding $n$, no ancestors. |
| `following` | Nodes following $n$, no descendants. |
| `preceding-sibling` | Nodes preceding $n$ and with the same parent as $n$. |
| `following-sibling` | Nodes following $n$ and with the same parent as $n$. |
| `attribute` | Attribute nodes of $n$. |
| `self` | Node $n$. |

XPath also provides a means for checking the string values of the nodes. The XPath query `/descendant-or-self::record="John Scofield Groove Elation Blue Note"`, for instance, selects all element nodes with label "record" for which the value of all text node descendants concatenated in document order matches "John Scofield Groove Elation Blue Note". In the scope of this paper, however, we will not concentrate on querying the string values; our focus is on the XPath axes.

## 4 Previous Proposals

In this section, we discuss the previous proposals for modeling nested relationships in XML documents using relational databases in detail. For brevity, we have omitted the document identifiers from the relations, but as in [9], these could easily be added to support storage and retrieval of multiple documents in a single database.

### 4.1 Pre-/Postorder Encoding

The pre-/postorder encoding [18] makes use of the following very simple property of pre- and postorder numbers. For any two nodes $n_1$ and $n_2$, $n_1$ is an ancestor of $n_2$, iff the preorder number of $n_1$ is smaller than the preorder number of $n_2$ and the postorder number of $n_1$ is greater than the postorder number of $n_2$. As in [7], we store all the nodes in a single relation `Node`:

```
Node(Pre, Post, Par, Type, Name, Value)
```

In this schema, the database attributes `Pre`, `Post`, and `Par` correspond to the preorder and postorder numbers of the node and the preorder number of the parent of the node, respectively. The database attribute `Type` corresponds to the type of the node and the database attribute `Name` to the name of the node.

`Value` corresponds to the string value of the node. Like many earlier proposals [5] [10] [17], we do not store the string values of element nodes.

As noticed by Grust [7], the pre- and postorder numbers, combined with the parent information, are sufficient to perform all XPath axes by using the query conditions presented in Table 2; node tests can be supported by using the additional query conditions presented in Table 3.

**Table 2.** Pre-/postorder encoding query conditions for supporting the axes

| Axis | Query conditions |
|---|---|
| `parent` | $n_{i+1}$`.Pre=`$n_i$`.Par` |
| `child` | $n_{i+1}$`.Par=`$n_i$`.Pre` |
| `ancestor` | $n_{i+1}$`.Pre<`$n_i$`.Pre AND `$n_{i+1}$`.Post>`$n_i$`.Post` |
| `descendant` | $n_{i+1}$`.Pre>`$n_i$`.Pre AND `$n_{i+1}$`.Post<`$n_i$`.Post` |
| `ancestor-or-self` | $n_{i+1}$`.Pre<=`$n_i$`.Pre AND `$n_{i+1}$`.Post>=`$n_i$`.Post` |
| `descendant-or-self` | $n_{i+1}$`.Pre>=`$n_i$`.Pre AND `$n_{i+1}$`.Post<=`$n_i$`.Post` |
| `preceding` | $n_{i+1}$`.Pre<`$n_i$`.Pre AND `$n_{i+1}$`.Post<`$n_i$`.Post` |
| `following` | $n_{i+1}$`.Pre>`$n_i$`.Pre AND `$n_{i+1}$`.Post>`$n_i$`.Post` |
| `preceding-sibling` | `preceding AND `$n_{i+1}$`.Par=`$n_i$`.Par` |
| `following-sibling` | `following AND `$n_{i+1}$`.Par=`$n_i$`.Par` |
| `attribute` | $n_{i+1}$`.Par=`$n_i$`.Pre AND `$n_{i+1}$`.Type="attr"` |
| `self` | $n_{i+1}$`.Pre=`$n_i$`.Pre` |

**Table 3.** Additional query conditions for supporting the node tests

| Axis | Query conditions |
|---|---|
| `node()` | |
| `text()` | $n_{i+1}$`.Type="text"` |
| `comment()` | $n_{i+1}$`.Type="comm"` |
| `processing-instruction()` | $n_{i+1}$`.Type="proc"` |
| `name` | $n_{i+1}$`.Type="elem" AND `$n_{i+1}$`.Name="name"` |
| `*` | $n_{i+1}$`.Type="elem"` |

For brevity, we will not discuss the XPath-to-SQL query translation in full detail. For our purposes, it is sufficient to say that the SQL queries can be generated simply by walking through all the location steps in an XPath query and by using the query conditions to perform each step. For example, the XPath query $n_1$`/descendant::record` can be translated into the following SQL query:

```
SELECT DISTINCT n2.*
FROM Node n1, Node n2
WHERE n2.Pre>n1.Pre AND n2.Post<n1.Post
AND n2.Type="elem" AND n2.Name="record"
ORDER BY n2.Pre;
```

The last row of the query is added to ensure that the nodes are returned in document order, as required by the XPath recommendation. Optional predicates

in queries can be evaluated by changing the tuple variable in the SELECT part of the query. The XPath query $n_1$`/descendant::*/following-sibling::record [child::title]`, for example translates into the following SQL query:

```
SELECT DISTINCT n3.*
FROM Node n1, Node n2, Node n3, Node n4
-- First step /descendant::*
AND n2.Pre>n1.Pre AND n2.Post<n1.Post
AND n2.Type="elem"
-- Second step /following-sibling::record
AND n3.Pre>n2.Pre AND n3.Post>n2.Post AND n3.Par=n2.Par
AND n3.Type="elem" AND n3.Name="record"
-- Third step /child::title
AND n4.Par=n3.Pre
AND n4.Type="elem" AND n4.Name="title"
ORDER BY n3.Pre;
```

In [7], Grust also described how the evaluation of `descendant` and `descendant-or-self` axes can remarkably be accelerated by tightening the query conditions for the pre- and postorder numbers of the descendant nodes. Using this idea, the XPath query $n_1$`/descendant::record` can be translated into the following "accelerated" SQL query in which *height* denotes the height of the tree:

```
SELECT DISTINCT n2.*
FROM Node n1, Node n2
WHERE n2.Pre>n1.Pre AND n2.Pre<=n1.Post + height
AND n2.Post<n1.Post AND n2.Post>=n1.Pre - height
AND n2.Type="elem" AND n2.Name="record"
ORDER BY n2.Pre;
```

However, if $n_1$ is the root of the tree the query is not accelerated at all, since every node in the tree has a preorder (postorder) number smaller (larger) than the postorder (preorder) number of $n_1$. In general, the closer to the root the context node resides, the less will the query be accelerated.

The XML research literature knows several methods similar to pre-/postorder encoding, such as the *nested sets model* [2] and the *order/size scheme* [6]. XRel [5] makes use of *region coordinates* which tell the indexes of the first and last character of the piece of text corresponding to an element, attribute or text node. Nevertheless, all of these methods impose the nested relationships to be checked using expensive nonequijoins.

### 4.2 Ancestor/Descendant Index

An extreme approach for modeling the nested relationships is to build an ancestor/descendant index, i.e., to explicitly maintain all ancestor/descendant pairs.

In order to evaluate `ancestor-or-self` and `descendant-or-self` axes efficiently, all nodes actually have to be in relation `AncDesc` with themselves, which raises the storage requirements even higher. Nevertheless, this approach can provide good query performance, and thus it has been followed in [10], for instance. To maintain the ancestor/descendant information, we use an `AncDesc` table:

```
Node(Pre, Par, Type, Name, Value)
AncDesc(Anc, Desc)
```

It is easy to find the query conditions for ancestor/descendant index; these are presented in Table 4. Evaluating the `ancestor` and `descendant` axes involves an equijoin on preorder numbers to ensure that the context node is not included in the result of the location step. When evaluating `following` and `preceding` axes, we use subqueries to filter the descendants and ancestors from the result.

**Table 4.** Ancestor/descendant index query conditions for performing the axes using `AncDesc` tuple variable $a_i$

| Axis | Query conditions |
|---|---|
| parent | $n_{i+1}$.Pre=$n_i$.Par |
| child | $n_{i+1}$.Par=$n_i$.Pre |
| ancestor | ancestor-or-self AND NOT $n_{i+1}$.Pre=$n_i$.Pre |
| descendant | descendant-or-self AND NOT $n_{i+1}$.Pre=$n_i$.Pre |
| ancestor-or-self | $n_{i+1}$.Pre=$a_i$.Anc AND $a_i$.Desc=$n_i$.Pre |
| descendant-or-self | $n_{i+1}$.Pre=$a_i$.Desc AND $a_i$.Anc=$n_i$.Pre |
| preceding | $n_{i+1}$.Pre $<$ $n_i$.Pre AND $n_{i+1}$ NOT IN ancestor |
| following | $n_{i+1}$.Pre $>$ $n_i$.Pre AND $n_{i+1}$ NOT IN descendant |
| preceding-sibling | preceding AND $n_{i+1}$.Par=$n_i$.Par |
| following-sibling | following AND $n_{i+1}$.Par=$n_i$.Par |
| attribute | $n_{i+1}$.Par=$n_i$.Pre AND $n_{i+1}$.Type="attr" |
| self | $n_{i+1}$.Pre=$n_i$.Pre |

Using the query conditions presented in Table 4 and the additional query conditions presented in Table 3, we can now translate the XPath query $n_1$/`descendant::record` into the following SQL query:

```
SELECT DISTINCT n2.*
FROM Node n1, Node n2, AncDesc a1
WHERE n2.Pre=a1.Desc AND a1.Anc=n1.Pre AND NOT n2.Pre=n1.Pre
AND n2.Type="elem" AND n2.Name="record"
ORDER BY n2.Pre;
```

It is worth noticing that in this query, there are no expensive nonequijoins, and thus we can expect this query to run faster than the corresponding query that is based on pre-/postorder encoding.

### 4.3 Ancestor/Leaf Index

Simply put, an ancestor/leaf index is an ancestor/descendant index built only the leaf nodes as descendants. To maintain the ancestor/leaf information, we employ an `AncLeaf` table:

```
Node(Pre, Par, Type, Name, Value)
AncLeaf(Anc, Leaf)
```

Obviously, the database attribute `Leaf` corresponds to the preorder number of the leaf node and `Anc` corresponds to the preorder number of the ancestor of the leaf node. Similarly to the ancestor/descendant approach, all leaf nodes are in relation `AncLeaf` with themselves.

To evaluate all the axes using ancestor/leaf index, we first define a "special axis" `special` which, for a `Node` tuple variable $n_i$, can be evaluated using `AncLeaf` variables $a_i$ and $b_i$ with query conditions $n_{i+1}$`.Pre=`$a_i$`.Anc AND` $a_i$`.Leaf=`$b_i$`.Leaf AND` $b_i$`.Anc=`$n_i$`.Pre`. In simple terms, the result of the `special` axis contains all ancestors and descendants of the context node and the context node itself. To filter the ancestors or descendants from the result of this query, we can use the preorder numbers of the nodes, as presented in Table 5.

**Table 5.** Ancestor/leaf index query conditions for performing the axes

| Axis | Query conditions |
|---|---|
| `parent` | $n_{i+1}$`.Pre=`$n_i$`.Par` |
| `child` | $n_{i+1}$`.Par=`$n_i$`.Pre` |
| `ancestor` | `special AND` $n_{i+1}$`.Pre<`$n_i$`.Pre` |
| `descendant` | `special AND` $n_{i+1}$`.Pre>`$n_i$`.Pre` |
| `ancestor-or-self` | `special AND` $n_{i+1}$`.Pre<=`$n_i$`.Pre` |
| `descendant-or-self` | `special AND` $n_{i+1}$`.Pre>=`$n_i$`.Pre` |
| `preceding` | $n_{i+1}$`.Pre<`$n_i$`.Pre AND` $n_{i+1}$ `NOT IN ancestor` |
| `following` | $n_{i+1}$`.Pre>`$n_i$`.Pre AND` $n_{i+1}$ `NOT IN descendant` |
| `preceding-sibling` | `preceding AND` $n_{i+1}$`.Par=`$n_i$`.Par` |
| `following-sibling` | `following AND` $n_{i+1}$`.Par=`$n_i$`.Par` |
| `attribute` | $n_{i+1}$`.Par=`$n_i$`.Pre AND` $n_{i+1}$`.Type="attr"` |
| `self` | $n_{i+1}$`.Pre=`$n_i$`.Pre` |

Using these query conditions and the additional query conditions presented in Table 3, we can evaluate our example query $n_1$`/descendant::record` using the following SQL query:

```
SELECT DISTINCT n2.*
FROM Node n1, Node n2, AncLeaf a1, AncLeaf b1
WHERE n2.Pre=a1.Anc AND a1.Leaf=b1.Leaf AND b1.Anc=n1.Pre
AND n2.Pre>n1.Pre
AND n2.Type="elem" AND n2.Name="record"
ORDER BY n2.Pre;
```

Although this query involves one nonequijoin on preorder numbers, it can be evaluated quite efficiently, since most of the work can be carried out using inexpensive equijoins. Provided that the database management system optimizes the query correctly, the nonequijoin is very likely performed after the equijoins, and thus the nonequijoin is performed on only a small number of tuples. To avoid unnecessary disk I/O during queries, the `AncLeaf` table should be sorted according to the database attribute `Leaf`.

## 5 Proxy Index

Both ancestor/descendant index and ancestor/leaf index suffer from the same problem: they maintain a lot of redundant information. In this section, we thus describe a *proxy index* as a method for representing the structural relationships in a more compact manner. Our idea is to select a set of inner nodes to act as *proxy nodes* and to maintain the ancestor/descendant information only for these proxies. The concept of proxy node can formally be defined as follows:

**Definition 1.** *Node $n$ is a proxy node of level $i$, if the length of the path from $n$ to some leaf node is $i$ or if $n$ is the root node and the length of the path from $n$ to some leaf node is smaller than $i$.*

To couple the proxy nodes together with their ancestors and descendants, we use the relation `ProxyReach`:

```
Node(Pre, Post, Par, Type, Name, Value)
ProxyReach(Proxy, Reach)
```

In this schema, the database attribute `Proxy` corresponds to the preorder number of the proxy node and `Reach` corresponds to the preorder number of a node that is reachable from the proxy, i.e., a node that is an ancestor or descendant of the proxy node or the proxy node itself.

Notice that each proxy node corresponds to an equivalence class induced by an equivalence relation $\equiv_{pi}$ on the leaf nodes defined as follows:

**Definition 2.** *For any two leaf nodes $n_1$ and $n_2$, $n_1 \equiv_{pi} n_2$, iff there exists a proxy node $n_3$ of level $i$ such that $n_3$ is the nearest proxy node ancestor for both $n_1$ and $n_2$.*

Using this notion, we can now define the ancestor/leaf index as a proxy index corresponding to the relation $\equiv_{p0}$. Obviously, the structural information can be encoded more compactly by increasing the value of $i$. Building an index corresponding to the relation $\equiv_{p1}$, however, will probably not get us far, since in the case of typical XML trees, most leaf nodes are text nodes without any siblings. Thus, with value 1 we will end up with almost as many tuples in the `ProxyReach` table as with value 0. For this reason, our experimental evaluation was carried out using value 2 which proved to provide a good balance between query speed and storage consumption.

Similarly to the ancestor/leaf index, we use a “special axis” `special` which, for a `Node` tuple variable $n_i$, can be evaluated using `ProxyReach` variables $a_i$ and $b_i$ with query conditions $n_{i+1}$.Pre=$a_i$.Reach AND $a_i$.Proxy=$b_i$.Proxy AND $b_i$.Reach=$n_i$.Pre. Here, the result of the `special` axis contains all descendants and ancestors of the proxy nodes that are reachable from $n_i$. Thus, we can use query conditions similar to those used in the pre-/postorder encoding approach to filter the unwanted nodes from the result of the axis, as presented in Table 6.

**Table 6.** Proxy index query conditions for performing the axes

| Axis | Query conditions |
|---|---|
| parent | $n_{i+1}$.Pre=$n_i$.Par |
| child | $n_{i+1}$.Par=$n_i$.Pre |
| ancestor | special AND $n_{i+1}$.Pre<$n_i$.Pre AND $n_{i+1}$.Post>$n_i$.Post |
| descendant | special AND $n_{i+1}$.Pre>$n_i$.Pre AND $n_{i+1}$.Post<$n_i$.Post |
| ancestor-or-self | special AND $n_{i+1}$.Pre<=$n_i$.Pre AND $n_{i+1}$.Post>=$n_i$.Post |
| descendant-or-self | special AND $n_{i+1}$.Pre>=$n_i$.Pre AND $n_{i+1}$.Post<=$n_i$.Post |
| preceding | $n_{i+1}$.Pre<$n_i$.Pre AND $n_{i+1}$.Post<$n_i$.Post |
| following | $n_{i+1}$.Pre>$n_i$.Pre AND $n_{i+1}$.Post>$n_i$.Post |
| preceding-sibling | preceding AND $n_{i+1}$.Par=$n_i$.Par |
| following-sibling | following AND $n_{i+1}$.Par=$n_i$.Par |
| attribute | $n_{i+1}$.Par=$n_i$.Pre AND $n_{i+1}$.Type="attr" |
| self | $n_{i+1}$.Pre=$n_i$.Pre |

Using these query conditions and the additional query conditions presented in Table 3, we can evaluate our example query $n_1$`/descendant::record` using the following SQL query:

```
SELECT DISTINCT n2.*
FROM Node n1, Node n2, ProxyReach a1, ProxyReach b1
WHERE n2.Pre=a1.Reach AND a1.Proxy=b1.Proxy AND b1.Reach=n1.Pre
AND n2.Pre>n1.Pre AND n2.Post<n1.Pre
AND n2.Type="elem" AND n2.Name="record"
ORDER BY n2.Pre;
```

Although this query involves two nonequijoins, it can still be evaluated rather efficiently, as explained in the section discussing ancestor/leaf index.

## 6 Experimental Results

This section presents the results of our experimental evaluation carried out using MySQL 5.0 Alpha running on Windows XP and a 2.00 GHz Pentium PC with 512 MB of RAM and standard IDE disks. As in [9], the database schemas corresponding to pre-/postorder encoding (PP), ancestor/descendant index (AD), ancestor/leaf index (AL), and proxy index (PR) were all extended with document identifiers; the proxy index was built on relation $\equiv_{p2}$.

### 6.1 Storage Requirements

To start with, we compared the storage consumption of the methods by storing three different sets of XML documents into databases designed according to PP, AD, AL, and PR[2]. For this purpose, we used the 1998 baseball statistics and a collection of four religious texts, both available at [19]. We also studied the storage requirements using a deeply nested and structurally complex XML document generated with XMLgen [20] using factor 1. The database sizes in both tuples and megabytes, as well as the sizes of original XML documents, are presented in Table 7.

**Table 7.** Database sizes

| | | PP | | AD | | AL | | PR | |
|---|---|---|---|---|---|---|---|---|---|
| | MB | Tuples | MB | Tuples | MB | Tuples | MB | Tuples | MB |
| Baseball | 0.6 | 52707 | 3.6 | 393095 | 15.1 | 233142 | 9.9 | 110608 | 5.2 |
| Religious | 6.8 | 94616 | 11.7 | 599005 | 29.0 | 366178 | 20.6 | 195006 | 15.0 |
| XMLgen | 113 | 3221932 | 305 | 23134116 | 998 | 13952004 | 686 | 8339906 | 486 |

As can be seen in Table 7, PR performs quite well in terms of storage consumption. Although PR consumes more storage than PP, the difference between PP and PR is not nearly as significant as between PP and AD or AL. In the next section, we will see that PR allows us to evaluate queries much more efficiently than PP, which justifies the bigger database size.

### 6.2 Query Performance

We evaluated the query performance of the different approaches using synthetic documents generated with XMLgen using factors 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, and 1.28. The sizes of these documents ranged approximately from 1.1 MB to 146 MB. From each document, we randomly selected 20 context nodes, and performed the `parent`, `child`, `ancestor`, `descendant`, `following-sibling`, and `preceding-sibling` axes starting from these context nodes.

According to our experiments, the evaluation time of all axes grows linearly with respect to the document size. Regardless of the document size, however, axes `parent`, `child`, `preceding-sibling`, `following-sibling`, and `descendant` can be evaluated in almost no time in all approaches, provided that the `descendant` queries in PP are accelerated.

The results concerning the `ancestor` axis, on the contrary, reveal one of the weaknesses of PP. Since this axis cannot be accelerated, the query evaluation time in PP grows rapidly compared to other approaches. In approaches AD, AL, and PR, not only the `descendant` axis, but also the `ancestor` axis can be

[2] Auxiliary indexes were built on `Node(Doc, Post)` `Node(Doc, Par)`, `Node(Type)`, `AncDesc(Doc, Desc)`, `AncLeaf(Doc, Leaf)`, `Proxy(Doc, Reach)`, and the first five characters of `Node(Name)`.

evaluated very efficiently. Evaluating this axis using the largest test document, for example, took almost 16 seconds in PP, whereas in AD, AL, and PR, almost no time at all was needed. The results are presented in Fig. 1; all times are averages for the 20 context nodes.

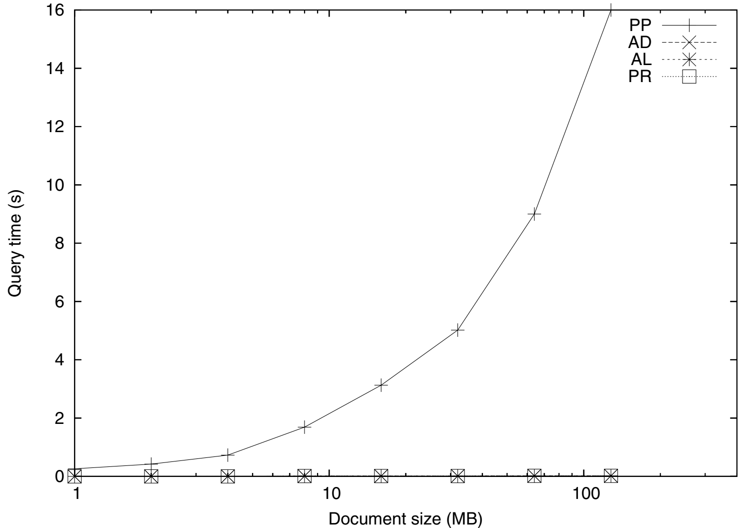


**Fig. 1.** Times for the `ancestor` axis (logarithmic scale for size)

As discussed section 4.1, the `descendant` and `descendant-or-self` axes are not accelerated in PP if the root is the context node. This is actually a rather serious drawback, since all queries based on absolute addressing traverse the tree starting from the root. To exemplify this problem, Fig. 2 presents the query times for the XPath query `/descendant-or-self::open_auction`. According to these results, the scalability of PP leaves a lot to be desired when the traversal starts from the root. Also in AL, the query times grow rather rapidly, but both PR and AD perform very well indeed even in the case of the largest test document. Although the SQL queries in PR look similar to the queries in AL, PR clearly outperforms AL, since it issues less disk accesses.

After these tests, we still wanted to see how the approaches perform when a large set of context nodes is used. To do this, we stored the result sets of our previous query `//open_auction` into separate relations and performed the location step `/descendant-or-self::description` starting from these node sets; the size of the context node set varied from 120 to 15360. In these tests, PP suffered from severe scalability problems with respect to the number of context nodes. For instance, even in the case of the smallest test document, PP took almost two minutes, whereas the other approaches needed less than a second, which clearly supports our presentiments on the lack of scalability in PP. Indeed, although the acceleration can make a substantial difference when the set of context nodes is small, PP does not scale well with respect to the number of

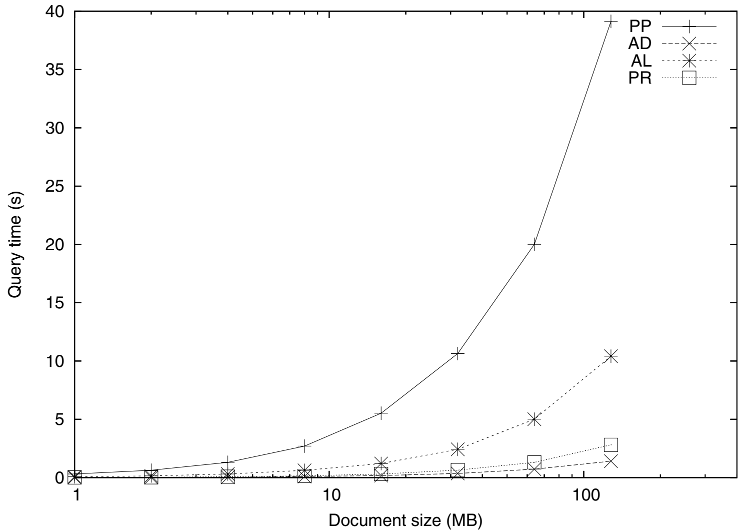


**Fig. 2.** Times for query `//open_auction` (logarithmic scale for size)

context nodes. PR performed better than AL and also in these tests, AD provided the best performance. These results are presented in Fig. 3.

All in all, our results were similar to those obtained in [10] and [17]. It must be pointed out that according to Grust [7], PP performs much better if the pre- and postorder numbers are indexed using R-trees. The purpose of this study, however, was to rely on standard relational database technology, and thus we did not consider this possibility.

## 7 Conclusion and Future Work

In this paper, we discussed methods for modeling the nested relationships in XML documents using relational databases. We also proposed a novel method, namely a proxy index, which maintains the ancestor/descendant information for a selected set of inner nodes. Our proposal makes it possible to check the nested relationships mainly using equijoins instead of nonequijoins, and thus our method can clearly outperform the widely used pre-/postorder encoding. Furthermore, our method encodes the structural relationships in a compact manner, and thus the storage consumption is low compared to other methods that model the structural relationships explicitly.

It would be interesting to study how the approaches discussed in this paper perform when relative addressing is used. Since PR performed quite well with large sets of context nodes, we believe that our approach performs well also when relative addressing of XPath is used. It would also be interesting to study how different clusterings, i.e., orderings of the tuples, affect the performances of the methods. Ordering the tuples according to their names instead of their preorder numbers, for example, might provide good results.

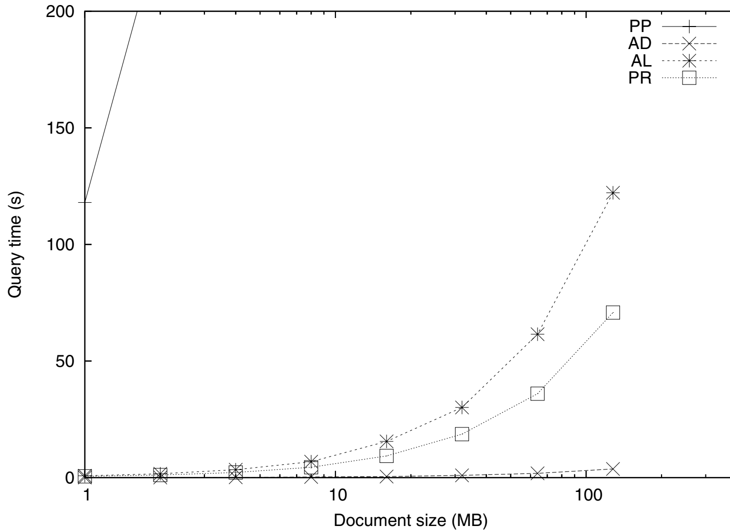


**Fig. 3.** Times for query `//open_auction//description` (logarithmic scale for size)

## References

1. W3C. Extensible Markup Language (XML) 1.0. `http://www.w3c.org/TR/REC-xml`.
2. A.B. Chaudri, A. Rashid, and R. Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems.* Addison-Wesley, 2003.
3. W3C. XML path language (XPath) 2.0. `http://www.w3c.org/TR/xpath20`.
4. W3C. XQuery 1.0: An XML query language. `http://www.w3c.org/TR/xquery`.
5. M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technologies*, 1(1): 110-141, 2001.
6. Q. Li and B. Moon. Querying XML data for regular path expressions. In *Proc. of the 27th Intl Conf. on Very large Databases*, pages 361-370, 2001.
7. T. Grust. Accelerating XPath location steps. In *Proc. of the 2002 ACM SIGMOD Conf. on Management of Data*, pages 109-120, 2002.
8. C. Zhang, J. Naughton, D. DeWitt, Qiong Luo, G. Lohman. On supporting containment queries in relational database management systems. In *Proc. of the 2001 ACM SIGMOD Conf. on Management of Data*, pages 425-436, 2001.
9. O. Luoma. Modeling nested relationships in XML documents using relational databases. In *Proc. of the 31st Annual Conf. on Current Trends in Theory and Practice of Informatics*, pages 259-268, 2005.
10. H. Jiang, H. Lu, W. Wang, and J. Xu Yu. Path materialization revisited: An efficient storage model for XML data. In *Proc. of the 13th Australasian Database Conf.*, pages 85-94, 2002.
11. J. McHugh, S. Abiteboul, R. Goldman, R. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3): 54-66, 1997.
12. C.C. Kanne and G. Moerkotte. Efficient storage of XML data. Poster abstract in *Proc. of the 16th Intl Conf. on Data Engineering*, page 198, 2000.

13. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. of the 1994 ACM SIGMOD Intl Conf. on Management of Data*, pages 313-324, 1994.
14. D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report, INRIA, 1999.
15. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D.J. DeWitt, and J.F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of the 25th Intl Conf. on Very Large Databases*, pages 302-314, 1999.
16. R. Krishnamurthy, R. Kaushik, and J.F. Naughton. XML-to-SQL query translation literature: The state of the art and open problems. In *Proc. of the 1st Intl XML Database Symposium*, pages 1-18, 2003.
17. S. Prakash, S.S. Bhowmick, and S. Madria. SUCXENT: An efficient path-based approach to store and query XML documents. In *Proc. of the 15th Intl Conf. on Database and Expert Systems Applications*, pages 285-295, 2004.
18. P.F. Dietz. Maintaining order in a linked list. In *Proc. of the 14th ACM Symposium on Theory of Computing*, pages 122-127, 1982.
19. `http://www.ibiblio.org/xml/examples`.
20. `http://monetdb.cwi.nl/xml/index.html`.