

Xeek: An Efficient Method for Supporting XPath Evaluation with Relational Databases^{*}

Olli Luoma

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

Abstract. We introduce Xeek, a method for managing XML documents using relational databases. Most previous proposals in which the XPath axes are evaluated using only the pre- and postorder numbers of the nodes have quite convincingly been shown to suffer from scalability problems. Thus, Xeek stores some redundant structural information which can be used speed up XPath evaluation even when a large set of context nodes is used. Our idea is to select a set of inner nodes to act as proxy nodes and store the ancestor information only for the proxies. To demonstrate the effectiveness of our method, we present the results of our experiments in which Xeek outperformed the previous methods based on relational databases as well as an established commercial XML database product.

1 Introduction

Relational database management systems offer a powerful and reliable means for storing and querying structured data, and thus they are at the core of data management in all organizations. Recently, however, some modern application areas, such as bioinformatics [1] and Web services [2], have created a need for storing and querying information that cannot easily be organized into tables, columns, and rows. XML [3], a document description metalanguage recommended by the World Wide Web Consortium, provides a means for representing heterogeneous data in a simple and platform-independent manner, and thus database management systems are constantly called to the challenge of managing XML data.

As an answer to this challenge, a plethora of methods for managing XML data using relational databases have been proposed [4]. In these methods, an incoming XML document is usually first represented as an *XML tree* in which element and attribute nodes correspond to the metadata and text nodes to the textual content in the original document. Ancestor/descendant relationships between the nodes correspond to the nesting relationships in the document¹ [5]. This tree is then stored into the database and queried using some XML query

^{*} Supported by the Academy of Finland.

¹ The XPath recommendation actually lists seven different node types, but for simplicity, we limit the discussion to element, attribute, and text nodes.

language, such as XPath [5] or XQuery [6], which can be mapped to SQL in an external software layer. Similarly, the resulting relations are mapped to XML outside the RDBMS. This is indeed a tempting option, since it allows us to leverage the well tried indexing and query optimization capabilities of relational databases.

At the heart of the XPath query language, there are 12 *axes*, i.e., operators for XML tree traversals, such as `parent`, `child`, `ancestor`, `descendant`, `preceding`, and `following`. In most of the previous proposals, these axes have been supported by using pre- and postorder numbers of the nodes or some similar method [7] [8] [9] [10] [11]. In the context of relational databases, however, checking structural relationships using pre- and postorder numbers requires very expensive *nonequijoins*, i.e., joins based on inequality comparisons. Thus, one can argue that systems relying on a relational database and pre-/postorder encoding lack the scalability that can be expected from a practicable XML management system [12] [13] [14].

Because of the lack of scalability of the pre-/postorder encoding, many alternative methods aiming at replacing the *nonequijoins* with *equijoins*, i.e., joins based on equality comparisons, have been proposed. The extreme approach, of course, is to store all ancestor/descendant pairs into a separate table [14] [15], but more sophisticated methods consuming less storage have also been proposed. In [13], for example, we described a method which stores the ancestor information only for the leaf nodes and in [14], we generalized this idea by introducing a *proxy index* in which the ancestor/descendant information is stored only for a subset of inner nodes, namely so called *proxy nodes*. Our original proposal, however, consumed quite a lot of storage, and thus as one of the contributions of this paper, we propose an improvement which allows us store the structural information in a more compact manner. Furthermore, our improved design provides better query performance than the original proposal. In summary, Xeek contributes to the XML-RDBMS research by providing the following features:

1. Good query performance for all XPath axes.
2. Modest storage consumption compared to query performance.
3. Ability to balance between storage consumption and query performance by tuning just one parameter.

The rest of this paper is organized as follows. In section 2, we discuss the related work and in section 3, we present the basics of the XPath query language; Xeek is described in section 4. Section 5 discusses the results of our experimental evaluation and section 6 concludes this article.

2 Related Work

The existing approaches for managing XML data using relational databases can roughly be divided into two categories [8]. One option is to extract the logical structures or DTDs of the documents and define relational schemas in accordance with the DTDs; this approach is usually known as the *structure-mapping*

approach or *schema-aware approach*. Early examples of this approach include the approach proposed by Florescu and Kossmann [16] in which one relation for each element type is created. Shanmugasundaram *et al.* [17] proposed a method for designing the relational schemas based on a detailed analysis of the document DTDs. However, if we are to store a large number of documents with different DTDs we might end up flooding the database with table definitions [10] [4]. The other option is to design a relational schema which represents the generic constructs of the XML data model, such as element, attribute, and text nodes. This *model-mapping approach* or *schema-oblivious approach* allows us to store any well-formed XML document without changing the schema, which simplifies the XPath-to-SQL query translation.

In addition to our own previous work [13] [14] [18], the model-mapping approach has been followed in several other proposals, such as XRel [8], XParent [15], XPath accelerator [10], and SUCXENT [19]. Unlike our own proposals, XRel, XParent, and SUCXENT support only the **descendant** and **child** axes of XPath, and thus one can argue that they provide a very limited XPath support. XPath accelerator, on the other hand, is based on pre-/postorder encoding, which makes it prone to scalability problems. Nevertheless, the relational schema of XPath accelerator is very similar to Xeek, and thus we regard Grust's XPath accelerator as a very relevant piece of work. For this reason, we also compare the performance of Xeek with XPath accelerator.

A completely different approach is to build a *native XML management system*, i.e., to build a XML management system from scratch. In the XML research literature, there are several examples of this approach, such as the early proposals LORE [20] and NATIX [21], and more recent ones, such as TIMBER [22]. As a part of this research, multiple algorithms for joining XML tree node sets have been proposed, such as the multi-predicate merge join [12], staircase join [23], and just recently a partition-based P-join [24]. Interestingly, the idea behind some of these methods is somewhat similar to the idea behind our proposal in the sense that they also aim at avoiding nonequijoins. In P-join, for example, the nodes are first partitioned into nine disjoint subsets. This information is then used to prune all nodes located in partitions that cannot contain nodes satisfying the query conditions without considering their pre- and postorder numbers at all.

3 XPath Basics

As mentioned earlier, XPath [5] is based on a tree representation of a *well-formed* XML document, i.e., a document that conforms to the syntactic rules of XML. A simple example of an XML tree corresponding to the XML document `<c d="y"/><c d="y"><e>k1</e></c><c><e>ez</e></c>` is presented in Fig. 1. The nodes are numbered in both pre- and postorder; unlike in [8], for example, the attribute nodes are numbered similarly to all other nodes although according to the XML recommendation, there is actually no order among attribute nodes.

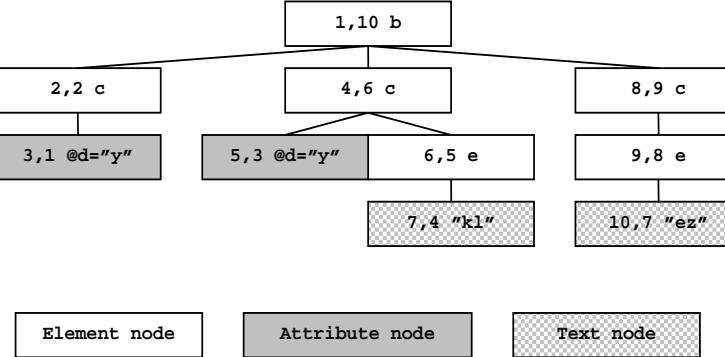


Fig. 1. An XML tree

The tree traversals in XPath are based on 12 axes which are presented in Table 1. In simple terms, an XPath query can be thought of as a series of *location steps* of the form `/axis::nodetest[predicate]` which start from a context node - initially the root of the tree - and select a set of related nodes specified by the axis. A *node test* 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. The location step `n/child::c[child::*]`, for example, selects all children of the context node `n` which are named "c" and have one or more child nodes.

Table 1. XPath axes and their semantics

Axis	Semantics of $n/$ Axis
<code>parent</code>	Parent of n .
<code>child</code>	Children of n , no attribute nodes.
<code>ancestor</code>	Transitive closure of <code>parent</code> .
<code>descendant</code>	Transitive closure of <code>child</code> , no attribute nodes.
<code>ancestor-or-self</code>	Like ancestor, plus n .
<code>descendant-or-self</code>	Like descendant, plus n , no attribute nodes.
<code>preceding</code>	Nodes preceding n , no ancestors or attribute nodes.
<code>following</code>	Nodes following n , no descendants or attribute nodes.
<code>preceding-sibling</code>	Preceding sibling nodes of n , no attribute nodes.
<code>following-sibling</code>	Following sibling nodes of n , no attribute nodes.
<code>attribute</code>	Attribute nodes of n .
<code>self</code>	Node n .

Using XPath, it is also possible to set conditions for 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". Notice also that the result of an XPath query is the concatenation of the parts of the document corresponding to the result nodes. In our example case, query `/descendant-or-self::c/descendant-or-self::*`, for example, would result in no less than `<c d="y"/><c d="y"><e>k1</e></c><c><e>ez</e></c><e>k1</e><e>ez</e>`.

4 Xeek

The overall architecture of Xeek is very similar to the methods discussed in [8] and [10]. Roughly speaking, Xeek consists of three components: the *database builder*, the *query translator*, and the *XML builder*. The database builder simply parses an incoming document using a SAX parser and issues a set of SQL INSERT commands to insert the nodes into the database. The query translator takes an XPath query as an input, parses the query, and issues an SQL query to retrieve a set of nodes that satisfy the query. Finally, the purpose of the XML builder is to build the result documents. This is done by iterating the results of the SQL queries generated by the query translator and issuing another set of SQL queries to retrieve the information needed to build the actual results. Thus, one can think that the database builder and the XML builder work in opposite manners.

4.1 The Relational Schema

The relational schema of Xeek is based on the idea of a proxy index [14] which stores the ancestor/descendant information only for a selected subset of inner nodes. The proxy nodes can actually be selected according to any criterion as long as they cover all of the leaf nodes, i.e., there does not exist a leaf node that is not a descendant of any of the proxies. In Xeek, we use a simple definition formulated as follows:

Definition 1. *Node n is a proxy node of level i , if n has exactly $i-1$ ancestors or if n is a leaf node having less than $i-1$ ancestors.*

According to this definition, there are three proxy nodes of level 2 in the XML tree presented in Fig. 1, namely the nodes identified by preorder numbers 2, 4, and 8. In our original proposal, we defined the *proxy index* as a set of pairs (n_1, n_2) where n_1 is a proxy node and n_2 is a node that is *reachable* from the proxy, i.e., is an ancestor or descendant of the proxy node or the proxy node itself. In Xeek, however, we only store a set of pairs (n_1, n_2) where n_1 is a proxy node and n_2 is a node that is an ancestor of the proxy or the proxy node itself, which results in a smaller index. For example, building a proxy index for a tree of size n using value 1 for i produced in our original proposal an index with n pairs, whereas Xeek stores just one pair. If value ∞ is used for i , i.e., the leaf

nodes are used as proxies, both Xeek and our original proposal produce similar indexes.

It is also interesting to notice that since the root node is always the only proxy node of level 1 and the leaf nodes are always the only proxy nodes of level ∞ , both XPath accelerator and the ancestor/leaf approach [13] in which the ancestor information is stored only for the leaf nodes can be regarded as special cases of the proxy index. Thus, a proxy index provides us with a simple method for balancing between these two extremes by tuning the value of i . Increasing the value of i obviously results in a larger index but also yields better query performance. To implement the idea of the proxy index in Xeek, we use the following relational schema:

```
Node(Pre, Post, Proxy, ProxySelf, Par, Type, Name, Value)
AncProxy(Anc, Proxy)
```

In relation **Node**, the database attributes **Pre** and **Post** correspond to the preorder number and the postorder number of the node and the database attribute **Proxy** corresponds to the preorder number of the leftmost² proxy node reachable from the node. For proxy nodes and their ancestors, the database attribute **ProxySelf** stores the same value as **Pre** and for descendants of proxy nodes, this attribute stores the same value as **Proxy**. Attribute **Par** corresponds to the preorder number of the parent of the node and **Type** to the type of the node. The Database attributes **Name** and **Value** correspond to the name and the string value of the node, respectively; string values are stored only for text nodes. Finally, relation **AncProxy** is used to store the actual proxy index. In both relations, the underlined attributes constitute the primary key.

The semantics of the database attributes **Proxy** and **ProxySelf** may seem a bit peculiar, but the use of these attributes is actually rather simple. If we want to find the ancestors of a given node n , we first use the **Node** and **AncProxy** tables to find all ancestors of the leftmost proxy reachable from n . This is done by equijoining the **Node** table with the **AncProxy** table on attribute **Proxy**. The result of this join is then equijoined with the **Node** table matching the values of **Anc** and **ProxySelf**. Regardless of the level of the proxy nodes, the result of these equijoins is guaranteed to contain the ancestors of n , and hence we can use more expensive nonequijoins on database attributes **Pre** and **Post** to filter out the nodes which are not ancestors of n . Thus, the selection of the proxy node level affects not only the balance between storage consumption and query performance, but also between equijoins and nonequijoins.

4.2 XPath-to-SQL Query Translation

By using an algorithm very similar to the algorithms discussed in [8] and [10], the query translator first transforms an XPath query into a *query tree* in which each node corresponds to a location step and branches correspond

² Term *leftmost* means the node having the smallest preorder number.

to predicates. The query translator also locates the *active node* of the query tree, i.e., the node which corresponds to the node set that is finally returned as an answer to the query. An example of a query tree corresponding to `/descendant::a/following::b="abc" [preceding::c]/child::*="ijk"` is presented in Fig. 2; the active node is underlined.

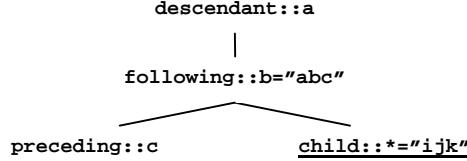


Fig. 2. A query tree

The query translator then transforms the query tree into an SQL query. This is done using the query conditions for the axes presented in Table 2 and the conditions for node tests presented in Table 3; the string value conditions of the query are matched against the database attribute `Value`. In Table 2, the tuple variable n_i corresponds to the set of context nodes and tuple variable n_{i+1} to the resulting set of nodes. The special axis `p-anc` corresponds to the query conditions $n_{i+1}.\text{ProxySelf}=p_{i+1}.\text{Anc}$ AND $p_{i+1}.\text{Proxy}=n_i.\text{Proxy}$ and the axis `p-desc` corresponds to inverse query conditions $n_{i+1}.\text{Proxy}=p_{i+1}.\text{Proxy}$ AND $p_{i+1}.\text{Anc}=n_i.\text{ProxySelf}$ ³. In both cases, tuple variable p_{i+1} is an `AncProxy` variable.

In simple terms, our query translation algorithm works as follows. Each node in the query tree is first assigned a tuple variable n_i where i is the preorder number of the node in the query tree. Furthermore, tuple variable n_0 is assigned to the root of the XML tree, i.e., the initial context node. The SQL query is then generated by joining each node in the query tree to its parent with the query conditions presented in Table 2; tuple variable n_1 is joined with tuple variable n_0 . If the node of the query tree involves node tests, the conditions in Table 3 are used. The database attribute `Value` is used to check string value tests. Finally, the tuple variable corresponding to the active node is used in the `SELECT` and `ORDER BY` parts of the query. Using this algorithm, the query tree presented in Fig. 2 is translated into the following SQL query:

```

SELECT DISTINCT n4.* 
FROM Node n0, Node n1, Node n2, Node n3, Node n4, AncProxy p1
-- First step: /descendant::a
WHERE n0.Pre=1 AND p1.Anc=n0.ProxySelf AND n1.Proxy=p1.Proxy
  
```

³ It can be shown that for any context node n , $n/\text{ancestor-or-self}::* \subseteq n/\text{p-anc}::*$ and $n/\text{descendant-or-self}::* \subseteq n/\text{p-desc}::*$

```

AND n1.Pre>n0.Pre AND n1.Post<n0.Post AND n1.Type='elem'
AND n1.Name='a'
-- Second step: /following::b="abc"
AND n2.Pre>n1.Pre AND n2.Post>n1.Post AND n2.Type='elem'
AND n2.Name='b' AND n2.Value='abc'
-- Third step: /preceding::b
AND n3.Pre<n2.Pre AND n3.Post<n2.Post AND n2.Type='elem'
AND n3.Name='c'
-- Fourth step: /child::*="ijk"
AND n4.Par=n2.Pre AND n2.Type='elem' AND n2.Value='ijk'
ORDER BY n4.Pre;

```

Notice that although evaluating the `ancestor`, `ancestor-or-self`, `descendant`, and `descendant-or-self` involves nonequijoins, these should not be too expensive to evaluate. Provided that the query optimizer optimizes the queries correctly, the conditions based on equijoins are checked first. These conditions restrict the number of nodes that have to be checked using inequality comparisons, which accelerates the queries considerably. Also notice that our original proposal [14] did not include the attributes `Proxy` and `ProxySelf` in relation `Node`, and thus we had to perform a possibly large selfjoin on the table storing the proxy index to find all nodes that are reachable from any of the proxy nodes reachable from the context node, which in some cases proved to be expensive. Furthermore, our original proposal consumed considerably more storage.

Table 2. Query conditions for axes.

Axis	Query conditions
<code>parent</code>	$n_{i+1}.\text{Pre}=n_i.\text{Par}$
<code>child</code>	$n_{i+1}.\text{Par}=n_i.\text{Pre}$
<code>ancestor</code>	$\text{p-anc} \text{ AND } n_{i+1}.\text{Pre}< n_i.\text{Pre} \text{ AND } n_{i+1}.\text{Post}> n_i.\text{Post}$
<code>descendant</code>	$\text{p-desc} \text{ AND } n_{i+1}.\text{Pre}> n_i.\text{Pre} \text{ AND } n_{i+1}.\text{Post}< n_i.\text{Post}$
<code>ancestor-or-self</code>	$\text{p-anc} \text{ AND } n_{i+1}.\text{Pre}\leq n_i.\text{Pre} \text{ AND } n_{i+1}.\text{Post}\geq n_i.\text{Post}$
<code>descendant-or-self</code>	$\text{p-desc} \text{ AND } n_{i+1}.\text{Pre}\geq n_i.\text{Pre} \text{ AND } n_{i+1}.\text{Post}\leq n_i.\text{Post}$
<code>preceding</code>	$n_{i+1}.\text{Pre}< n_i.\text{Pre} \text{ AND } n_{i+1}.\text{Post}< n_i.\text{Post}$
<code>following</code>	$n_{i+1}.\text{Pre}> n_i.\text{Pre} \text{ AND } n_{i+1}.\text{Post}> n_i.\text{Post}$
<code>preceding-sibling</code>	<code>preceding</code> AND $n_{i+1}.\text{Par}=n_i.\text{Par}$
<code>following-sibling</code>	<code>following</code> AND $n_{i+1}.\text{Par}=n_i.\text{Par}$
<code>attribute</code>	$n_{i+1}.\text{Par}=n_i.\text{Pre} \text{ AND } n_{i+1}.\text{Type}=\text{"attr"}$
<code>self</code>	$n_{i+1}.\text{Pre}=n_i.\text{Pre}$

Table 3. Query conditions for node tests

Node test	Query conditions
<code>node()</code>	
<code>text()</code>	$b.Type = "text"$
<code>comment()</code>	$b.Type = "comm"$
<code>processing-instruction()</code>	$b.Type = "proc"$
<code>name</code>	$b.Type = "elem" \text{ AND } b.Name = "name"$
<code>*</code>	$b.Type = "elem"$

4.3 Generating Result Documents

After the result of the SQL query generated by the algorithm described in the previous section has been retrieved, Xeek still needs to build the result documents. To do this, the XML builder iterates the result set and issues a **descendant-or-self** query for each of the nodes in the result set. This is done by issuing the following query in which the variables *pre*, *post*, and *proxySelf* denote the preorder number of the node, the postorder number of the node, and the preorder number of the leftmost proxy reachable from the node:

```
SELECT n.*  
FROM Node n, AncProxy a  
WHERE a.Anc=proxySelf AND n.Proxy=a.Proxy  
AND n.Pre>=pre AND n.Post<=post  
ORDER BY n.Pre;
```

The result sets of these queries are then iterated to stream out the part of the document that corresponds to the node. To build the result documents, Xeek obviously has to issue thousands of **descendant-or-self** queries, and thus these queries should be as efficient as possible.

5 Experimental Results

In our experimental evaluation, we used Windows XP and Microsoft SQL Server 2000 running on 2.00 GHz Pentium PC equipped with 2 GB of RAM and standard IDE disks. We implemented all approaches using Java and extended the relational schema of Xeek (abbreviated XK) with document identifier Doc. The performance of our method was compared against other methods based on relational databases, namely Grust's XPath accelerator (XA) [10], our previous proposal (PR) [14], and the ancestor/descendant approach (AD) [14] in which a separate table is used to store all ancestor descendant pairs. The relational schemas of XA, PR, and AD were also extended with document identifiers; auxiliary indexes were built on `Node(Doc, Post)`, `Node(Doc, Par)`, `Node(Doc, Proxy)`, `Node(Doc, ProxySelf)`, `Node(Type)`, `Node(Name)`, and `AncProxy(Doc, Proxy)`.

We also compared Xeek to X-Hive [25], a rather established commercial XML database product. These tests supported our presentiments of the weak scalability of native XML databases since they revealed some severe scalability problems in X-Hive. Xeek, on the contrary, performed well also in these tests.

5.1 Proxy Node Selection

We studied the optimal number of nodes to be promoted as proxies by building databases with different numbers of proxy nodes, i.e., by selecting proxy nodes from level 1, 2, 3 etc., from a synthetic XML document generated with XMLgen [26] using factor 0.1. We then used these databases to perform **ancestor** and **descendant** steps starting from a relatively large set of context nodes. The size of our test document was approximately 11 MB and the number of nodes in the document was 324217. The results are presented in Fig. 3.

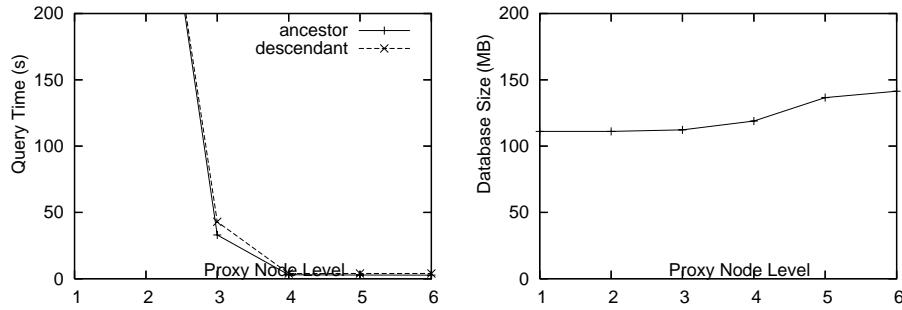


Fig. 3. Query times and database sizes with different proxy node levels

Obviously, the results only apply to this particular document, but we can use these results to estimate the number of nodes that should generally be promoted as proxies. In our tests, good query performance was achieved by selecting proxies from level 4, which means that roughly 40000 nodes were promoted; selecting proxies from level 5 or 6 did not result in better query performance. Thus, there seems to be very little to be gained by promoting more than 10 percent of nodes. This observation should be rather independent of the document, so it should generally be safe both in terms of storage consumption and query performance to select roughly 10 percent of the nodes to act as proxies.

5.2 Storage Requirements

We compared the storage consumption of XK, XA, PR, and AD by storing three different sets of XML documents into databases designed according to these approaches. In these tests, the periodic table of elements and the 1998 baseball statistics, both available at [27], were used. We also studied the storage requirements by storing an 111 MB XML document generated with XMLgen.

The relative database sizes in both tuples and megabytes are presented in Fig. 4; all values are scaled so that XK has value 1. XK performs quite well in terms of storage consumption even when compared to XA. The difference between XK and PR, on the contrary, as well as the difference between XK and AD is significant in the case of deeply nested XMLgen and Baseball documents.

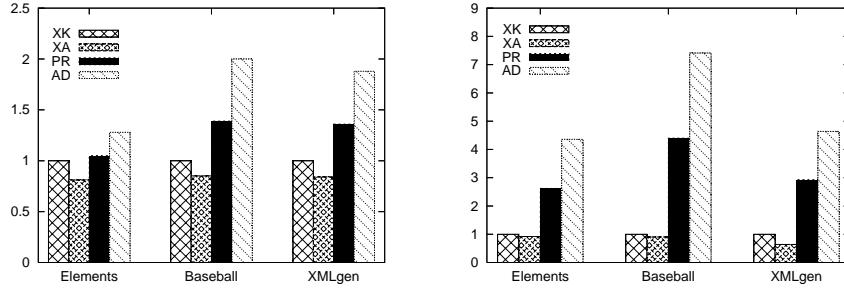


Fig. 4. Relative database sizes both in megabytes (left) and tuples (right)

5.3 Query Performance

We evaluated the query performance of our approach using synthetic documents generated with XMLgen using factors 0.001, 0.01, 0.1, and 1; the sizes of these documents ranged from 0.11 MB to 111 MB. Using these documents, we performed four major axes, i.e., `ancestor`, `descendant`, `preceding`, and `following`, using all nodes with label "item" as context nodes. Since the other axes involve more restrictive query conditions they are generally easier to evaluate than the major axes. For this reason, we omitted the results concerning the other axes. To test the `ancestor` axis in XK, for example, we used the following SQL query:

```

SELECT DISTINCT n2.*
FROM Node n1, Node n2, AncProxy a2
WHERE n1.name='item'
AND a2.Doc=n1.Doc AND a2.Proxy=n1.Proxy
AND n2.Doc=a2.Doc AND n2.ProxySelf=a2.Anc
AND n2.Pre<n1.Pre AND n2.Post>n1.Post
ORDER BY n2.Doc, n2.Pre;

```

It is worth pointing out that the number of context nodes was rather large - the size of the context node set ranged from 44 to 43500 - and furthermore, since the context nodes are scattered rather evenly across the documents, all of the axes also resulted in a large node set. These features separate our experimental setup from many previous experiments in which only a small set of context nodes was used. It should be obvious that all approaches can perform well with a small amount of context nodes, but this is not a very realistic scenario. It is easy to write XPath queries which are not very restrictive and involve massive node

sets, and hence scalability with respect to the number of context nodes is an important factor.

The results concerning the **ancestor** and **descendant** axes presented in Fig. 5 clearly reveal the lack of scalability in XA. Even though the information about the height of the tree is used to tighten the query conditions [10], performing these axes is extremely expensive. In the case of 11 MB document generated using factor 0.1, for example, XK, PR, and AD needed only seconds to perform these operations, whereas XA needed minutes. Thus, one can argue that the elegance of XA comes at the cost of scalability. In the case of the **ancestor** axis, XK, PR, and AD were equally efficient, but in the case of **descendant** axis, XK outperforms both PR and AD. Even though AD evaluates the descendant axis completely using equijoins, it is quite clearly outperformed by XK, since it issues much more disk reads than XK⁴. The good performance of **descendant** axis is especially important since they are used to build the result documents as discussed in section 4.3. The **preceding** and **following** axes are performed with similar query conditions in all approaches, and hence the results of all approaches for these axes are combined in Fig. 6. Only the times needed to retrieve the nodes which satisfy the query are measured; the time for building the result documents is not included.

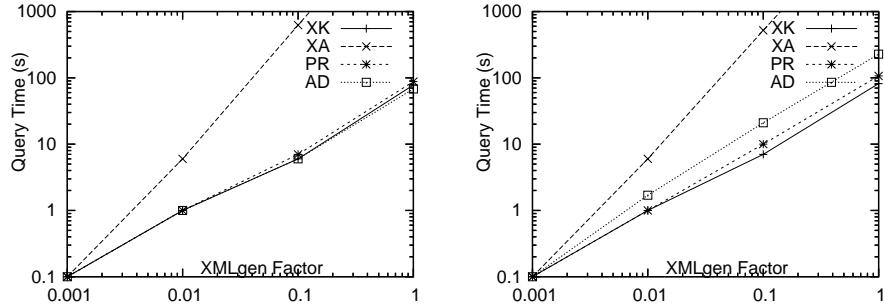


Fig. 5. Performance for **ancestor** (left) and **descendant** (right) axes

We also compared the different approaches using MySQL 5.0, but for brevity, these results have been omitted. Overall, MySQL provided results similar to those obtained using SQL Server, but we found the **preceding** and **following** axes to be much slower in MySQL than in SQL Server. This is very probably due to the sophisticated hash join algorithms used in SQL Server and for this reason, XK should be implemented on MySQL only if the performance of these axes is not essential. However, since SQL server stores tens of bytes of metadata per row the SQL Server databases were considerably larger than the MySQL

⁴ One should notice that SQL Server avoided the pitfall of favouring hash joins over more efficient nested loop joins in AD. Thus, although they use some nonequijoins, both PR and XK can indeed outperform AD.

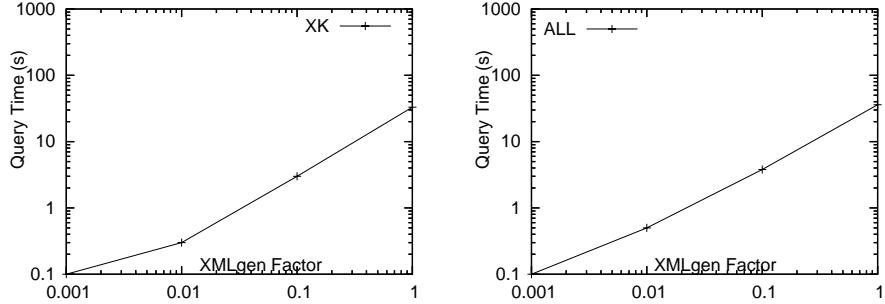


Fig. 6. Performance for preceding (left) and following (right) axes

databases. All in all, our results were in line with those obtained in [15], [19], and [14].

Table 4 presents the query set used for comparing Xeek against X-Hive, a commercial native XML database; in Table 4, abbreviation “//” stands for descendant-or-self axis. Queries 1 and 2 are simple path expression queries whereas the other queries involve predicates with different selectivity. Every element with label “address” contains an element with label “street”, and thus evaluating queries 5 and 6 involves massive node sets. The predicates used in queries 3 and 4, on the contrary, are rather restrictive, and thus queries 3 and 4 should be much easier to evaluate than queries 5 and 6. The preceding axis has been used since it very often reveals scalability problems in XML management systems. This is especially true if this axis is performed starting from a large set of context nodes.

Table 4. Query set for Xeek vs. X-Hive tests

#	Query
1	//address
2	//address/preceding::item
3	//address[//*=“Connecticut”]
4	//address[//*=“Connecticut”]/preceding::item
5	//address[//street]
6	//address[//street]/preceding::item

Table 5 presents the query times and the times needed to build the result documents as well as the number of nodes in the results set for the queries presented in Table 4. An 111 MB synthetic XML document generated with XMLgen was used in these tests. In the case of queries 1, 3, and 5, both Xeek and X-Hive performed well: the result documents were obtained in a matter of just seconds. However, it is interesting to notice that Xeek needs the time to build the result documents whereas in X-Hive, most of the time is used to locate the nodes. Thus, there seems room for improvement in the way Xeek builds the

result documents. As another interesting detail, one may notice that Xeek needs quite a lot of time to build the results for queries 2, 4, and 6. This is due to the fact that the nodes with label "item" have generally more descendants than the nodes with label "address".

Queries 2, 4, and 6, i.e., the queries involving the **preceding** axis, reveal severe scalability problems in X-Hive. All of these queries took over 5 minutes to evaluate, and thus they were timed out. In the case of queries 2 and 6, this is rather understandable since they involve performing the **preceding** axis starting from a very large set of context nodes. In the case of query 4, however, the bad query performance is very surprising since the size of the context node set is only 101, i.e., the size of the result of query 3. Even in the case of an 11 MB test document, X-Hive actually needed over 5 minutes to evaluate queries 2 and 6, whereas query 4 was evaluated in just a matter of seconds. In other words, increasing the size of the data by a factor of 10 increased the query time by a factor of 100. Thus, one can argue that the scalability of X-Hive leaves a lot to be desired. Xeek, on the contrary, scaled up well and the query times increased linearly with respect to the size of the data.

Table 5. Query times for Xeek vs. X-Hive tests

Xeek			X-Hive				
#	Query (s)	Build (s)	Total (s)	Query (s)	Build (s)	Total (s)	Nodes
1	1.3	7.1	8.4	8.2	1.4	9.6	12715
2	1.0	35.2	36.2	>300	N/A	N/A	21750
3	0.6	0.2	0.8	8.5	0.5	9.0	101
4	2.4	35.3	37.7	>300	N/A	N/A	21750
5	1.2	7.1	8.3	8.3	1.4	9.7	12715
6	1.0	35.0	36.0	>300	N/A	N/A	21750

6 Conclusion

This paper discussed Xeek, a method for supporting XPath axes with relational databases. The main intuition behind our method is to store a small amount of redundant information about the structural relationships between the nodes of XML trees, which allows Xeek to evaluate **ancestor**, **ancestor-or-self**, **descendant**, and **descendant-or-self** axes very efficiently. This is achieved by selecting a small amount of inner nodes to act as proxy nodes and storing the ancestor information for these proxies using a separate table. Our method also provides relatively good query performance for other XPath axes, and thus we feel that our proposal provides a good compromise between storage consumption and query performance. This claim is supported by the results of our experimental evaluation in which Xeek performed well compared to both methods based on relational databases and a native XML management system.

References

1. N. Bales, J. Brinkley, E. S. Lee, S. Mathur, C. Re, and D. Suciu. A framework for XML-based integration of data, visualization and analysis in a biomedical domain. *XSym*, pages 207-221, 2005.
2. D. Florescu and D. Kossmann. An XML programming language for Web service specification and composition. *IEEE Data Engineering Bulletin*, 24(2): 48-56, 2001.
3. W3C. Extensible Markup Language (XML) 1.0. <http://www.w3c.org/TR/REC-xml>.
4. R. Krishnamurthy, R. Kaushik, and J.F. Naughton. XML-to-SQL query translation literature: The state of the art and open problems. *XSym*, pages 1-18, 2003.
5. W3C. XML path language (XPath) 2.0. <http://www.w3c.org/TR/xpath20>.
6. W3C. XQuery 1.0: An XML query language. <http://www.w3c.org/TR/xquery>.
7. P.F. Dietz. Maintaining order in a linked list. In *Proc. of the 14th ACM Symposium on Theory of Computing*, pages 122-127, 1982.
8. 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.
9. Q. Li and B. Moon. Querying XML data for regular path expressions. *VLDB*, pages 361-370, 2001.
10. T. Grust. Accelerating XPath location steps. *ACM SIGMOD*, pages 109-120, 2002.
11. L. Yuen and C.K. Poon. Relational index support for XPath axes. *XSym*, pages 84-98, 2005.
12. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. On supporting containment queries in relational database management systems. *ACM SIGMOD*, pages 425-436, 2001.
13. O. Luoma. Modeling nested relationships in XML documents using relational databases. *SOFSEM*, pages 259-268, 2005.
14. O. Luoma. Supporting XPath axes with relational databases using a proxy index. *XSym*, pages 99-113, 2005.
15. H. Jiang, H. Lu, W. Wang, and J. Xu Yu. Path materialization revisited: An efficient storage model for XML data. *ADC*, pages 85-94, 2002.
16. D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report, INRIA, 1999.
17. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D.J. DeWitt, and J.F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. *VLDB*, pages 302-314, 1999.
18. O. Luoma. A structure-based filtering method for XML management systems. *DEXA*, pages 401-410, 2004.
19. S. Prakash, S.S. Bhowmick, and S. Madria. SUCXENT: An efficient path-based approach to store and query XML documents. *DEXA*, pages 285-295, 2004.
20. 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.
21. M. Brantner, S. Helmer, C.C. Kanne, and G. Moerkotte. Full-fledged algebraic XPath processing in Natix. *ICDE*, pages 705-716, 2005.
22. H.V. Jagadish, S. Al-Khalifa, A. Chapman, L.V.S. Lakshmanan, A. Nierman, S. Paparizos, J.M. Patel, D. Shrivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TMBER: A native XML database. *VLDB Journal*, 11(4): 274-291, 2002.
23. S. Mayer, M. van Keulen, T. Grust, and J. Teubner. An injection with tree awareness: Adding staircase join to PostgreSQL. *VLDB*, pages 1305-1308, 2004.

24. N. Tang, J. Xu Yu, K. F. Wong, K. Lü, and J. Li. Accelerating XML structural join by partitioning. *DEXA*, pages 280-289, 2005.
25. <http://www.x-hive.com>
26. <http://monetdb.cwi.nl/xml/index.html>.
27. <http://www.ibiblio.org/xml/examples>.