ACCELERATING XPATH AXES THROUGH STRUCTURAL PARTITIONING

Olli Luoma

Department of Information Technology, 20014 University of Turku, Finland

Keywords: XML, XPath, XML databases.

Abstract: The query evaluation algorithms of practically all XML management systems are based on structural joins, i.e., operations which determine all occurrences of parent/child, ancestor/descendant, preceding/following etc. relationships between node sets. In this paper, we present a simple method for accelerating structural joins which is very easy to implement on different platforms. Our idea is to split the nodes into disjoint partitions and use this information to avoid unnecessary structural joins. Despite its simplicity, our proposal can considerably accelerate XPath evaluation on different XML management systems. To exemplify this, we describe two implementation options of our method - one built from the scratch and one based on a relational database - and present the results of our experiments.

1 INTRODUCTION

In many modern applications areas, such as bioinformatics and Web services, there is a need to efficiently store and query large heterogeneous data sets marked up using XML (W3C, 2006a), i.e., represented as a partially ordered, labeled XML tree. This imposes a great challenge on data management systems which have traditionally been designed to cope with structured rather than semistructured data. In recent years, a lot of work has thus been done to develop new methods for storing and querying XML data using XML query languages, such as XPath (W3C, 2006b) and XQuery (W3C, 2006c).

At the heart of the XPath query language, there are 12 axes, i.e., operators for tree traversal. In this paper, we present a method for accelerating the evaluation of the major axes, i.e., the ancestor, descendant, preceding, and following axes, through partitioning. We make use of the observation that starting from any node, the main axes partition the document into four disjoint partitions. Our idea is to partition the nodes more accurately, store the partition information, and use this information to filter the nodes that cannot be contained in the result before actually performing the axis.

Since our method is based on simple geometric properties of the preorder and postorder numbers of the nodes, it is somewhat similar to the XML indexing approaches based on spatial structures, such as R-trees (Grust, 2002) or UB-trees (Krátký et al., 2004). The method described in this paper, however, is very easy to implement even using B-trees, and thus it can easily be used to accelerate XPath evaluation in XML management systems built on relational databases, such as XRel (Yoshikawa et al., 2001) and XPath accelerator (Grust, 2002). Furthermore, our approach can also be tailored to be used with other node identification schemes, such as the order/size scheme.

The rest of our paper is organized as follows. In section 2, we take a short look at the related work and in section 3, we present our own partitioning method. In section 4, we discuss the implementation of our method and in section 5, the results of our experimental evaluation. Section 6 concludes this article and discusses our future work.

2 RELATED WORK

In the XML research literature, there are numerous examples of different XML management systems.
ACCELERATING XPATH AXES THROUGH STRUCTURAL PARTITIONING

These systems can roughly be categorized into the following three categories:

- The flat streams approach handles the documents as byte streams (Peng & Chawathe, 2003) (Barton et al., 2003). Obviously, accessing the structure of the documents requires parsing and consumes a lot of time, but this option might still be viable in a setting where the documents to be stored and queried are small.

- In the metamodeling approach, the XML documents are first represented as trees which are then stored into a database. With suitable indexes this approach provides fast access to subtrees, and thus the metamodeling approach is by far the most popular in the field of XML management research. Unlike in the flat streams approach, however, rebuilding large parts of original documents from a large number of individual nodes can be rather expensive. The partitioning method discussed in this paper also falls into this category.

- The mixed approach aims at combining the previous two approaches. In some systems, the data is stored in two redundant repositories, one flat and one metamodeled, which allows the stored documents to be queried and the result documents to be built efficiently, but obviously creates some storage overhead. This problem could be tackled by compression to which XML data is often very amenable. There are also examples of a hybrid approach in which coarser structures of the XML documents are modeled as trees and finer structures as flat streams (Fiebig et al., 2003).

A lot of work has been carried out to accelerate structural joins (Al-Khalifa et al., 2002), i.e., operations which determine all occurrences of parent/child, ancestor/descendant, preceding/following etc. relationships between node sets. Some approaches are based on indexes built on XML trees, whereas others aim at designing more efficient join algorithms. The former category includes the so called proxy index (Luoma, 2005b) (Luoma, 2006) which effectively partitions the nodes into overlapping partitions so that the ancestors of any given node are contained within the same partition. Thus, when the ancestors of a given node are retrieved it is sufficient to check the partitions to which the node is assigned. Conversely, descendants can also be found efficiently since there is no need to check the partitions to which the node is not assigned. However, this approach is suitable for accelerating only ancestor/descendant operations. The method discussed in this paper, on the contrary, can also accelerate preceding/following operations.

The other group of methods include, for example, the staircase join (Grust & van Keulen, 2003). The idea of the staircase join is to prune the set of context nodes, i.e., the initial nodes from which the axis is performed. For instance, for any two context nodes \( n \) and \( m \) such that \( m \) is a descendant of \( n \), all descendants of \( m \) are also descendants of \( n \), and thus \( m \) can be pruned before evaluating the descendant axis. Another method based on preprocessing the nodes was proposed in (Tang et al., 2005). In this approach the nodes were partitioned somewhat similarly to the method discussed in this paper. However, both of these methods require a considerable amount of programming effort since they work by preprocessing the data rather than by building indexes. In the context of relational databases, for example, this would mean either reprogramming the DBMS internals or programming a collection of external classes to implement the join algorithms. Our method, on the contrary, requires very little programming effort even when implemented using a relational database, which we regard as the main advantage of our approach.

3 XPATH BASICS

As mentioned earlier, XPath (W3C, 2006b) 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

```
<b><c d="y"/></c>
<c d="y"><e>kl</e></c><c><e>ez</e></c></b>
```

is presented in Figure 1. The nodes are identified using their preorder and postorder numbers which encode a lot of structural information.

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 \( /\text{axis}::\text{nodetest}\{\text{predicate}\} \) 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 \( n/\text{child}\: :\: c[\text{child}::]\) for example, selects all children of the context node \( n \) which are named ”c” and have one or more child nodes. As a shorthand for axes child, descendant-or-self, and attribute, one can use \( ., // \), and \( @ \), respectively.

1In preorder traversal, a node is visited before its subtrees are recursively traversed from left to right and in postorder traversal, a node is visited after its subtrees have been recursively traversed from left to right.
Using XPath, it is also possible to set conditions for the string values of the nodes. The XPath query \texttt{//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 \texttt{//!c/*}, for example, would result in no less than \texttt{<c d="y"/>\texttt{<}\texttt{c} \texttt{d="y"/>\texttt{<e>kl</e>\texttt{<c}"ez"></c>\texttt{<e>kl</e>\texttt{<c}>ez</c>\texttt{>}}.}

\section{PARTITIONING METHOD}

As mentioned earlier, we rely on \textit{pre/postorder encoding} (Dietz, 1982), i.e., we assign both preorder and postorder numbers for the nodes. This encoding provides us with enough information to evaluate the four major axes (Grust, 2002) as follows\footnote{This is actually a bit inaccurate since the attribute nodes, for example, are not in the result of the \texttt{descendant} axis. However, if all nodes are treated equally, the proposition holds.}:

\begin{table}
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Axis} & \textbf{Semantics of n/Axis} \\
\hline
parent & Parent of \textit{n}. \\
child & Children of \textit{n}, no attribute nodes. \\
ancestor & Transitive closure of \textit{parent}. \\
descendant & Transitive closure of \textit{child}, no attribute nodes. \\
ancestor-or-self & Like \textit{ancestor}, plus \textit{n}. \\
descendant-or-self & Like \textit{descendant}, plus \textit{n}, no attribute nodes. \\
prescing & Nodes preceding \textit{n}, no ancestors or attribute nodes. \\
presceeding & Nodes following \textit{n}, no descendants or attribute nodes. \\
following & Preceding siblings of \textit{n}, no attribute nodes. \\
following-sibling & Following siblings of \textit{n}, no attribute nodes. \\
\texttt{attribute} & Attribute nodes of \textit{n}. \\
\texttt{self} & Node \textit{n}. \\
\hline
\end{tabular}
\caption{XPath axes and their semantics.}
\end{table}

\textbf{Proposition 1}. Let \texttt{pre}(\textit{n}) and \texttt{post}(\textit{n}) denote the preorder and postorder numbers of node \textit{n}, respectively. For any two nodes \textit{n} and \textit{m}, \textit{n} \in \texttt{m/ancestor}:: if\texttt{f} \texttt{pre}(\textit{n}) < \texttt{pre}(\textit{m}) and \texttt{post}(\textit{n}) > \texttt{post}(\textit{m}), \textit{n} \in \texttt{m/descendant}:: if\texttt{f} \texttt{pre}(\textit{n}) > \texttt{pre}(\textit{m}) and \texttt{post}(\textit{n}) < \texttt{post}(\textit{m}), \textit{n} \in \texttt{m/prescing}:: if\texttt{f} \texttt{pre}(\textit{n}) < \texttt{pre}(\textit{m}) and \texttt{post}(\textit{n}) < \texttt{post}(\textit{m}), and \textit{n} \in \texttt{m/following}:: if\texttt{f} \texttt{pre}(\textit{n}) > \texttt{pre}(\textit{m}) and \texttt{post}(\textit{n}) > \texttt{post}(\textit{m}).

This observation is exemplified on the left side of Figure 2 which shows the partitioning created by the major axes using node (6,5) as the context node. The ancestors of node (6,5) can be found in the upper-left partition, the descendants in the lower-right partition.
**ACCELERATING XPATH AXES THROUGH STRUCTURAL PARTITIONING**

The predecessors in the lower-left partition, and the followers in the upper-right partition. Based on the previous observation, the following proposition is obvious:

**Proposition 2.** For any two nodes \( n \) and \( m \) and for any \( p > 0 \), \( \lfloor \text{pre}(n)/p \rfloor \leq \lfloor \text{pre}(m)/p \rfloor \) and \( \lfloor \text{post}(n)/p \rfloor \geq \lfloor \text{post}(m)/p \rfloor \) if \( n \in m/\text{ancestor}::* \), \( \lfloor \text{pre}(n)/p \rfloor \geq \lfloor \text{pre}(m)/p \rfloor \) and \( \lfloor \text{post}(n)/p \rfloor \leq \lfloor \text{post}(m)/p \rfloor \) if \( n \in m/\text{descendant}::* \), \( \lfloor \text{pre}(n)/p \rfloor \leq \lfloor \text{pre}(m)/p \rfloor \) and \( \lfloor \text{post}(n)/p \rfloor \leq \lfloor \text{post}(m)/p \rfloor \) if \( n \in m/\text{following}::* \), and \( \lfloor \text{pre}(n)/p \rfloor \geq \lfloor \text{pre}(m)/p \rfloor \) and \( \lfloor \text{post}(n)/p \rfloor \geq \lfloor \text{post}(m)/p \rfloor \) if \( n \in m/\text{ancestor-or-self} \).

Proposition 2 provides us with simple means for partitioning the nodes into disjoint subsets. The partitioning is exemplified in Figure 2 (right) which shows the nodes of our example tree partitioned using value 4 for \( p \). It should now be obvious that when searching, for instance, the ancestors of node (6,5), it is sufficient to check the nodes in the shaded partitions since other partitions cannot contain any ancestors. In what follows, values \( \lfloor \text{pre}(n)/p \rfloor \) and \( \lfloor \text{pre}(n)/p \rfloor \) where \( n \) is a node residing in partition \( P \) are simply referred to as the preorder and postorder number of \( P \) and denoted by \( \text{pre}(P) \) and \( \text{post}(P) \), respectively.

Many minor axes can also benefit from partitioning. The ancestor-or-self and descendant-or-self axes, for example, behave similarly to ancestor and descendant axes, and thus they can be accelerated using the same partition information. The results of preceding-sibling and following-sibling, on the other hand, are subsets of preceding and following and the results of parent and child subsets of ancestor and descendant so they can also be accelerated using the partitioning. However, other minor axes than ancestor-or-self and descendant-or-self are generally very easy to evaluate, and thus using the partition information to evaluate them is often just unnecessary overhead especially if we have an index which can be used to locate the nodes with a given parent node efficiently (Luoma, 2005a). One should also notice that in practice, the nodes are not distributed evenly and there are several empty partitions. Thus, the number of non-empty partitions is usually much smaller than \( p^2 \).

5 IMPLEMENTATION OPTIONS

5.1 Relational Implementation

To test our idea, we designed a relational database according to the principles described in the previous section. As the basis of our implementation, we chose the XPath accelerator (Grust, 2002). In order to store the partition information, we employed relation \( \text{Part} \), and thus we ended up with the following relational schema:

\[
\text{Node}(\text{Pre}, \text{Post}, \text{Par}, \text{Part}, \text{Type}, \text{Name}, \text{Value})
\]

\[
\text{Part}(\text{Part}, \text{Pre}, \text{Post})
\]

As in the original proposal, the database attributes \( \text{Pre}, \text{Post}, \text{Par}, \text{Type}, \text{Name}, \) and \( \text{Value} \) of the \( \text{Node} \) relation correspond to the pre- and postorder numbers, the preorder number of the parent, the type, the name, and the string value of the node, respectively. The database attributes \( \text{Type}, \text{Name}, \) and \( \text{Value} \) are needed to support node tests and string value tests; the axes can be evaluated using database attributes \( \text{Pre}, \text{Post}, \text{Par}, \text{Type}, \text{Name}, \) and \( \text{Value} \).
Post, and Par. The partition information is contained in the Part table in which the database attributes Pre and Post correspond to the preorder and postorder number of the partition, respectively. The database attribute Par in the Node relation is a foreign key referencing to the primary key of the Part relation.

For the sake of brevity, we do not describe the XPath-to-SQL query translation in detail; the details can be found in (Yoshikawa et al., 2001) and (Grust, 2002). For our purposes, it is sufficient to say that in order to perform structural joins, these systems issue SQL queries which involve nonequi-joins, i.e., joins using < or > as the join condition, which can easily lead to scalability problems (Luo-

5.2 Native Implementation

We also implemented a simple XPath processor which parses an XML document and splits the nodes corresponding to the document into partitions which are maintained in the main memory. For each node, our system maintains its preorder number, postorder number, reference to its parent, type, name, and string value. In other words, the representation of the nodes is similar to the relational implementation discussed earlier. The partitions are lexically sorted according to their pre- and postorder numbers and the nodes within the partitions are sorted in preorder. Thus, given node \( n \), set of partitions \( S \), and axis \( a \), the following join algorithm simply iterates the partitions and checks only the partitions which can contain nodes in \( n/axis::* \). For the sake of brevity, we do not treat all axes in algorithm join; operator + is used as a shorthand for operation which adds an item to a set and \( par(n) \) denotes the parent of node \( n \).

... return result

The structural joins within a partition, then, are carried out using algorithm joinPart:

Again, our algorithm is heavily simplified. However, it is easy to extend the algorithm to support node tests and string value tests by checking the type, name, and string value attributes of the node. Notice also that we could dramatically lower the number
of structural joins by considering the fact that the all nodes in partition \( P \) such that \( \text{pre}(P) < \lfloor \text{pre}(n)/p \rfloor \) and \( \text{post}(P) < \lfloor \text{post}(n)/p \rfloor \) are guaranteed to be in \( n/\text{precedes}::* \), and thus they could be added to the result without performing the structural joins. We also implemented this option but found out that did not lead to considerable gains in evaluation times. However, the benefits obviously depend on the implementation and in some cases, this approach can further accelerate the joins.

6 EXPERIMENTAL RESULTS

6.1 Relational Implementation

We implemented the relational option using Windows XP and Microsoft SQL Server 2000 running on 2.00 GHz Pentium PC Equipped with 512 MB of RAM and standard IDE disks. The algorithms needed to build the databases, as well as the algorithms for query translation, were implemented using Java. We built indexes on \( \text{Node} (\text{Post}), \text{Node} (\text{Name}), \text{Part} (\text{Pre}), \text{and Part} (\text{Post}) \) and a clustered index on \( \text{Node} (\text{Part}) \). As our test data sets, we used the 1998 baseball statistics (52707 nodes) and the collection of Shakespeare’s plays (327129 nodes), both available at http://www.ibiblio.org/examples. The sizes of these documents were 640 kB and 7.47 MB, respectively; values 1, 2, 4, ..., 256 were used as a partitioning factor \( p \).

Figure 3 presents the query performance for the major axes using a relatively large set of context nodes; in these figures, the word "Partitions" refers to the value of \( p \) or the number of partitions per (pre-order or postorder) dimension. In the case of baseball.xml, nodes with name "PLAYER" (1226 nodes) and in plays.xml, nodes with name "SCENE" (750 nodes) served as the context nodes. One should notice that our method can lead to considerable performance gains in ancestor and descendant axes even with a very small amount of partitions. In the case baseball.xml and 16 partitions per dimension, for example, there were only 55 partitions in total. Thus, there were only 55 rows in the Part table, which is certainly acceptable considering that there are no less than 52707 nodes in the Node table. Even in the case of 256 partitions per dimension, there were only 772 rows in the Part table for "1998statistics.xml" and 859 rows for "plays.xml".

In the case of preceding and following axes, however, no real acceleration could be observed. These axes could be actually evaluated very efficiently even without the partitionin, which is due to the very sophisticated join algorithms implemented in SQL Server. Nevertheless, considering that the ancestor and descendant axes were very time-consuming to evaluate without the partition information, we are still convinced that the small amount of partition information can indeed pull its weight.

6.2 Native Implementation

The evaluation of our native implementation was carried out using the same equipment and data sets which were used in the relational case. Furthermore, the same number of partitions per dimension were used but in these tests, we randomly selected the context nodes. Figures 4 and 5 present the results obtained using the native implementation; all results are averages for 100 iterations. Overall, the ancestor and descendant axes behaved similarly to the relational case, i.e., they were considerably accelerated. However, the preceding and following axes were also accelerated by roughly a factor of two. This is actually intuitively clear since an average node has much more predecessors and followers than it has ancestors and descendants, and thus the preceding and following axes usually result in much larger node sets. In the case of ancestor axis, for example, the partitioning can help us to filter out a massive amount of nodes, i.e., the most of the descendants, predecessors, and followers, whereas in the case of preceding axis, a much smaller amount of nodes with respect to the size of the result can be filtered.

One should also notice that ordering the nodes according to their preorder numbers favors the preceding axis over the following axis. After the preorder numbers have been checked in the case of preceding axis, we still have to filter out the ancestors using the postorder numbers of the nodes. In the case of following axis, on the contrary, the descendants have to be filtered, which is a harder task since a node in an XML tree usually has more descendants than it has ancestors. Conversely, sorting the nodes according to their postorder numbers favors the following axis.

7 CONCLUDING REMARKS

In this paper, we discussed a partitioning method which can considerably accelerate the evaluation of XPath axes in different XML management systems. Our method is based on simple properties of the pre-order and postorder numbers of the nodes in an XML tree, which makes it very easy to implement. This was exemplified by presenting two different imple-
Figure 3: Relational results 1998statistics.xml (left) and plays.xml (right).

Figure 4: Number of joins for the major axes using random context nodes for 1998statistics.xml (left) and plays.xml (right); native implementation.

Figure 5: Times for the major axes using random context nodes for 1998statistics.xml (left) and plays.xml (right); native implementation.
ments of our idea which both indicated that our approach can lead to considerable performance gains.

REFERENCES


