A Comparison of Structure-Generic Relational Storage Schemes of XML Data

Weining Zhang and Douglas Pollok
Department of Computer Science
University of Texas at San Antonio

Abstract

XML data is often modeled as node-labeled trees. In a structure-generic relational storage scheme, the structure of the XML data is shredded into pieces of a generic type, such as nodes or edges, and data representing these structural pieces is stored in relational tables. In this paper, we consider four structure-generic relational storage schemes. We describe the translation of XML queries into SQL queries for each scheme. We compare the performance of these storage schemes through experiments. Our study shows that each scheme performs well in some cases, and no single scheme is a clear winner in all cases.

Keywords: XML database, storage scheme, query translation, performance evaluation, relational database

1 Introduction

The eXtensible Markup Language (XML) has become the standard of electronic data exchange over the Internet. Applications in industry, business, science, education and other fields have already started to represent, exchange, and process XML data. The efficient storage and retrieval of XML data becomes an important issue for databases that support XML applications. One approach is to build XML databases by leveraging relational database technology [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. Due to the mismatch between the tree-like structure of XML data and the flat table structure of relational data, such a system must map XML data to relational tables, translate XML queries into relational queries against these tables, and construct new XML documents from the answers to the relational queries. A number of methods have been proposed recently to map XML data to relational tables, translate XML queries into relational queries against these tables, and construct new XML documents from the answers to the relational queries. A number of methods have been proposed recently to map XML data to relational tables [2, 4, 5, 6, 10, 11]. These methods can be broadly classified into structure-specific methods and structure-generic methods.

In a structure-specific method [2, 11], the schema of relational tables is determined based on the structure of a given XML document. For example, in [11], the DTD of XML documents is analyzed to determine subgraphs that should be inlined into a table. Thus, XML documents of a bank may be stored in tables for customers, accounts, etc. These methods require complex analysis of XML schema or data, and the table schema is sensitive to structural changes of the XML data. Due to the lack of direct representation of structural information in the relational data, the translation from XML queries to SQL queries, and from relational data to XML data can be quite complex.

In a structure-generic method [5, 6, 12, 13, 14], the XML data is shredded into pieces of a generic type, such as nodes or edges. Data representing these structural pieces is stored in relational tables. The table schema is therefore determined by the type of the structural pieces rather than the specific structure of the XML data. Thus, no analysis of XML schema or XML data is needed and the table schema is stable with respect to structural changes of the XML data. Since the structural information is explicitly represented in the data, the translation from XML queries to SQL queries and from relational data to XML data is straightforward. In the rest of this paper, we use the storage scheme to mean the structure-generic relational storage scheme.

The evaluation of XML queries requires information about the structure of the XML data. The effectiveness and efficiency of a storage scheme depends critically on what structural information is explicitly stored and how to derive structural information that is not explicitly stored. Several storage schemes [5, 6, 12] store tree nodes and their positions in tables, and derive other structural information such as parent-child or ancestor-descendant relationships through (structural) joins. However, structural joins are inefficient in existing RDBMSs, especially for computing the parent-child relationships. A number of new algorithms have been proposed to improve the performance of the structural join [13, 14]. However, these algorithms are yet to be included in existing systems. Another storage scheme stores edges in tables [3, 15]. Since using edges to compute ancestor-descendant relationships in a relational system requires a recursive query involving many joins, it was considered impractical [3, 16]. However, our study [17] shows that while an edge based scheme may be slow for computing ancestor-descendant relationships, it can be more efficient than a node position based scheme for computing parent-child relationships. One way to improve the edge based scheme is to store additional structural information about ancestor-descendant relationships. This gives rise to a storage scheme that stores pairs of nodes representing all ancestor-descendant relationships. Yet another method is to combine the node position and edge based schemes into a hybrid scheme, in which edges can be used to compute
parent-child relationships and node positions can be used to compute ancestor-descendant relationships.

In this paper, we compare the four storage schemes mentioned above. We describe the table schema and the translation from XML queries into SQL queries according to a decomposition-based approach. We compare the storage space and the query performance of the storage schemes on a commercial RDBMS using an XML data set of William Shakespeare’s Plays and a set of XML test queries. Our experiments show that each scheme performs well in some cases, and no single scheme is a clear winner in all cases. To the best of our knowledge, a direct comparison of these schemes on a commercial RDBMS has not been reported previously.

The rest of the paper is organized as follows. In Section 2, we briefly describe the XML data model and basic types of XML subqueries. In section 3, we describe the four relational storage schemes and the translation from basic XML queries to SQL queries. In Section 4, we describe our experiments and report our findings. Finally, we conclude the paper in Section 5.

2 Preliminaries

In this section, we briefly describe the XML data model and present a decomposition of XML queries into basic types of subqueries.

2.1 An XML Data Model

An XML document consists of a collection of elements. Each element has a tag name and may contain zero or more sub-elements. The logical structure of an XML document can be viewed as a node-labeled tree. A fragment of the tree representing the Shakespeare play Hamlet is shown in Figure 1. Each node in the tree represents an element or a value string, and is labeled by the tag of the element or the text string of the value, respectively. Internal nodes represent elements, and leaf nodes represent values. We refer to a node labeled by a tag A as an A node, and that labeled by a value string a value node. The parent-child and ancestor-descendant relationships among nodes, as well as paths in the tree, are defined conventionally.

2.2 Basic Types of XML Queries

In this paper, XML queries are expressed in XPath. We assume that the reader is familiar with the syntax and semantics of XPath. As an example, the (branching) path query //scene[title="Scene I"]/speech/speaker//* finds the names of speakers who speak in any scene titled “Scene I”. To evaluate this query, we first find the scene nodes that are descendants of the document root and have a child title node with the value string “Scene I”. For each of these scene nodes, we then find a set of child speech nodes, and for each of these speech nodes, we find a set of child speaker nodes. Finally, for each of these speaker nodes, we return the node itself together with all of its descendants. The answer is the union of descendants of all speaker nodes that are evaluated. With respect to the data tree in Figure 1, the value node BERNADO is included within the answer. We can view a path query as a combination of subqueries of the following basic types.

1. **Node-Value (NV) Query**. This type of query is denoted by A=v, where A is a tag name, and v is a value. It finds A nodes that have a child v value node.

2. **Parent-Child (PC) Query**. This type of query is denoted by p/c or p[c], where p is the tag of the parent node, and c is the tag of the child node. With p/c, it finds c nodes that have a parent p node. With p[c], it finds p nodes that have a child c node.

3. **Ancestor-Descendant (AD) Query**. This type of query is denoted by either a//d or a[//d], where a is the tag of the ancestor nodes, and d is the tag of the descendant nodes. Similarly, a/d finds the d nodes and a[/d] finds the a nodes.

4. **Subtree (ST) Query**. This type of query is denoted by n/*, where n is a tag, and * is a wildcard representing any tag. This query returns the n nodes together with all descendants of the n nodes.

Given a relational storage scheme, each of these basic types of subqueries can be translated into a simple SQL query. A general method of translating a path query into an SQL query is to decompose it into basic subqueries, translate each subquery into an SQL query, and then combine the translated SQL queries into a single one. For example, the previous path query may be decomposed into the following subqueries: //scene, scene[title], title="Scene I", scene/speech, speech/speaker, and speaker/*.
3 Structure-Generic Storage Schemes

In this section, we describe the four structure-generic relational storage schemes mentioned in Section 1. For each storage scheme, we describe the table schema of the storage structure and the translation of basic subqueries into SQL queries. Due to limited space, we could not give a complete treatment in this paper. Interested readers are referred to [17] for details.

3.1 Node-Position Table (NPT)

In this scheme, an XML document is shredded into a collection of nodes. Each node is associated with its start and end positions in the XML document. In this paper, we use the word count as the position. Each word in an XML document is counted for one position with the first word of the document at position 1. For each element, the position of the first word of the starting tag and the position of the last word in the ending tag serve as the start and end positions of the corresponding node. For each value, the start (end) position is the position of its first (last) word. The start position of each node is unique in a given document.

The storage scheme consists of two tables.

Nodes(docid, start, end, tag, level, vid)
Values(vid, value)

where the docid is a unique identity of an XML document, the start and end are positions of a node as previously described, the level of a node in the XML tree is 0 for the root, 1 for children of the root, etc., and the vid is an identity of a value which is unique within the entire database. The underlined attributes form the primary keys of the tables. If the vid of a node is NULL in table Nodes, the node is an internal node representing an element and has no value.

The structural information of the XML data can be derived using the interval defined by the start and end positions. Specifically, the interval of a node is always contained in the intervals of its ancestor nodes. The node level can be used to distinguish a parent from an ancestor. The NPT scheme differs from the indexes described in [12] in that the position is defined by word count rather than character count, and that values are stored as a whole rather than as individual words.

The translation of a basic query to an SQL query is easy. For example, an NV query \(A = v\) can be translated into the following SQL query.

\[
\text{Select } N.* \\
\text{From Nodes } N, \text{ Values } V \\
\text{Where } N.tag=A \text{ and } V.value=v \text{ and } N.vid=V.vid
\]

and a PC query \(A/B\) can be translated into the following SQL query.

\[
\text{Select } B.*
\]

3.2 Edge Table (ET)

In this scheme, XML documents are shredded into edges. The scheme consists the following two tables.

Edges(docid, parent_id, child_id, parent_tag, child_tag, vid)
Values(vid, value)

where the parent_id and child_id are unique within a given document. The vid associates a value with the child node. The underlined attributes form the primary keys of the tables.

A NV subquery can be translated into an SQL query similar to the NV query in the NPT scheme. The PC subquery \(A/B\) can be translated into the following SQL query.

\[
\text{Select } * \\
\text{From Edges } E \\
\text{Where } E.parent_tag=A \text{ and } E.child_tag=B
\]

Since the Edges table explicitly represents the parent-child relationship, this query is a single table query. An AD subquery needs to be translated into a recursive SQL query traversing the edges from ancestor to descendant. An ST subquery is similar to the AD subquery, with the exception that the condition on descendant tags is omitted.

3.3 Ancestor-Descendant Table (ADT)

In this scheme, the XML data is shredded into pairs of ancestor and descendant nodes. The scheme contains the following tables.

AD(docid, anc_id, desc_id, anc_tag, desc_tag, distance, vid)
Values(vid, value)

The latter SQL query uses the positions and levels of nodes to test if the A node is the parent of the B node. Some simple adjustments are needed if the A or B nodes are the output of other subqueries, or if the subquery is in the form \(A[B]\). For example, if the A nodes are the output of another subquery, the condition \(A.tag=A\) can be removed. For query \(A[B]\), we simply change \(B.*\) to \(A.*\) in the select clause. An AD query is translated to an SQL query similar to the previous one, with the condition on level omitted. An ST query is also translated into a similar SQL query in which the condition on level and \(B.tag\) are omitted and conditions on start and end also include the equality. The adjustments needed for the AD and ST queries is similar to those for the PC query.
where the distance is the number of edges between an ancestor and a descendant in the path connecting the two nodes. Again, the underlined attributes form the primary keys.

The NV subquery and PC subquery are translated into SQL queries similar to those for the ET scheme, with the AD table replacing the Edges table, anc_id replacing parent_id, etc.

A PC subquery A/B can be translated into the following SQL query.

Select * From AD
Where anc_tag=A and desc_tag=B
and distance=1

An AD subquery A/B can be translated into a similar SQL query, with the condition on distance omitted. An ST subquery A/* can also be translated into a similar query, with the conditions on desc_tag and on distance both omitted.

### 3.4 Hybrid Tables (HT)

In this scheme, the NPT and ET schemes are combined. The scheme has the following tables:

- **Edges**(docid, parent_id, child_id, parent_tag, child_tag, vid)
- **Nodes**(docid, node_id, start, end, tag, level, vid)
- **Values**(vid, value)

The Nodes table is the same as the one in the NPT scheme, except that a new attribute node_id is added. In addition to node positions, each node is assigned an identity that is unique within the document. The Edges table is the same as the one in the ET scheme. The node identities in Nodes and Edges tables are the same for each node in the tree. A PC subquery is translated to the same SQL query in the ET scheme and all other three types of subqueries are translated to the corresponding SQL queries as in the NPT scheme.

### 4 Experiments and Results

In this section, we describe our experiments with the four structure-generic storage schemes and report our findings.

#### 4.1 The Experimental Environment

The XML data set used in the experiments is a set of 37 Shakespeare plays in XML documents [18]. The following table shows the storage size of the data.

<table>
<thead>
<tr>
<th>Tables</th>
<th># Tuples</th>
<th>Total Size</th>
<th>Index Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>179,618</td>
<td>4,310,832</td>
<td>4,168KB</td>
</tr>
<tr>
<td>Edges</td>
<td>179,581</td>
<td>5,028,268</td>
<td>13,968KB</td>
</tr>
<tr>
<td>AD</td>
<td>677,407</td>
<td>27,096,280</td>
<td>34,178KB</td>
</tr>
<tr>
<td>Values</td>
<td>148,060</td>
<td>7,736KB</td>
<td>6,968KB</td>
</tr>
</tbody>
</table>

The test queries used in our experiments are given in Table 1. Each query specifies a path pattern that starts at the document root and ends at an internal or a leaf node. The number of nodes matching the end node of the path is listed in the table as the number of answers for each query. According to the semantics of XPath, the answer to a query should contain not only the nodes matching the path pattern, but also all of their descendants. The total number of these answer nodes is also listed in Table 1. Among the 12 queries, Q1-Q3 are simple path queries that end at internal nodes and retrieve large sections of the XML data tree; Q4-Q6 are branching path queries that end at internal nodes and retrieve specific subtrees with decreasing sizes; Q7-Q9 are simple path queries that end at leaf nodes and emphasize on the parent-child relationships; Q10-Q12 are path queries that end at leaf nodes and emphasize on the ancestor-descendant relationships. In our experiment, each SQL query also sorts the answer based on the node order in the original XML documents to allow us to reconstruct an XML document from the answer.

The experiments were performed on a PC with a Pentium IV 1.7 GHz processor, 512MB of main memory, MS Windows 2000 SP2, and a 100GB HD 7200RPM 8.9ms average seek time. We used a commercial RDBMS with default system settings. There were no other running applications during the experiments. Each query was executed six times, with the first time being discarded. The time reported is the average of the last 5 executions.

#### 4.2 Experiment Results

The results of our experiments are shown in Figure 2.

For queries Q1-Q3, the answer to Q3 is a subset of that to Q2, which in turn is a subset of that to Q1. Although these 3 queries vary greatly in their number of answers, the total number of answer nodes are almost identical, thus the number of tuples returned are almost identical. Naturally, the time to complete these queries is also
Table 1. Test Queries

<table>
<thead>
<tr>
<th>Query</th>
<th>XPath Expression</th>
<th># Answers</th>
<th>Total # Answer Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q01</td>
<td>/play</td>
<td>37</td>
<td>179,618</td>
</tr>
<tr>
<td>Q02</td>
<td>/play/act</td>
<td>185</td>
<td>177,491</td>
</tr>
<tr>
<td>Q03</td>
<td>/play/act/scene</td>
<td>748</td>
<td>176,450</td>
</tr>
<tr>
<td>Q04</td>
<td>/play[title=&quot;Hamlet&quot;]</td>
<td>1</td>
<td>6636</td>
</tr>
<tr>
<td>Q05</td>
<td>/play[title=&quot;Hamlet&quot;]/act[title=&quot;Act I&quot;]</td>
<td>1</td>
<td>1475</td>
</tr>
<tr>
<td>Q06</td>
<td>/play[title=&quot;Hamlet&quot;]/act[title=&quot;Act I”]/scene[title=&quot;Scene I&quot;]</td>
<td>1</td>
<td>320</td>
</tr>
<tr>
<td>Q07</td>
<td>//play/title</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>Q08</td>
<td>//speech/speaker</td>
<td>31061</td>
<td>31061</td>
</tr>
<tr>
<td>Q09</td>
<td>//speaker/line</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Q10</td>
<td>//play/title</td>
<td>1031</td>
<td>1031</td>
</tr>
<tr>
<td>Q11</td>
<td>//speech/speaker</td>
<td>31061</td>
<td>31061</td>
</tr>
<tr>
<td>Q12</td>
<td>//speaker/line</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

very similar. As shown in Figure 2, the ET scheme has the worst performance, with the other three methods performing about on par with each other. The performance of ET increases slightly from Q1 to Q3 as the number of recursive joins is reduced. The increasing number of PC subqueries from Q1 to Q3 causes the performances of other schemes to decrease slightly. The results of Q1-Q3 clearly indicate that when a large portion of the XML tree is retrieved, there is little difference among the NPT, ADT and HT.

For queries Q4-Q6, the ET once again is by far the least efficient. The other schemes perform similarly with each other. For Q4, the ADT scheme outperforms the others because in the evaluation plan the RDBMS chooses a Nested Loop join for the final join instead of a hash join as it does for NPT and HT schemes. This simple change in the evaluation plan seems to have a big impact on the query performance. From Table 1 and Figure 2, it can be seen that the performances of Q1-Q6 are not affected as much by the number of answers as by the total number of answer nodes.

In queries Q7-Q9, we can clearly see that the use of the Edges table to check for parent-child relationships (which are just edges) proves it’s worth for queries Q8 and Q9. Especially for Q8, the performance of the ET is over 30 times greater than the NPT. For these three queries, the HT has exactly the same performance as the ET. The ADT scheme takes a little longer due to the much larger number of tuples in the AD table.

For query Q9, the NPT performs well because of the condition on node level. That is, the Speaker node must be one level higher than the Line node, which is used by the query optimizer to choose a query plan that quickly eliminates any possible matches (notice that Speaker nodes are siblings of Line nodes).

In queries Q10-Q12, the ADT has the best performance as one would expect, since ancestor-descendant relationships are explicitly stored. The ET performs rather well in these queries because some of the ancestor nodes in these queries are in fact the leaf nodes in the data and so the recursive joins will stop at these nodes. The NPT and HT (which use the Nodes table to compute the ancestor-descendant relationships) perform very badly for Q12 because there is no hint in the query that could allow the RDBMS to generate an evaluation plan to determine quickly that Speaker nodes do not have any descendant. Unlike query Q9, even if we provide a level condition for Q12 that the level of Speaker nodes must be higher than that of the Line nodes, this inequality condition can not be used by the RDBMS in the hash join of the query evaluation plan.

Queries Q9 and Q12 may seem odd for this XML data set. However, they illustrate a potential problem of NPT. In this XML tree, a large number of nodes are labeled with Speaker or Line, but few (actually none) of the Line nodes are descendants of Speaker nodes. For the ET or ADT, this fact can be found very quickly from the explicitly stored ancestor-descendant relationships. With the NPT, however, this fact must be computed through a join of large input relations resulting in poor performance. Although a Stack Tree join algorithm [13] may improve the performance of NPT, it is not currently supported by the RDBMS and even if it is, our analysis in [17] shows that the ET is still more efficient in these cases.

To summarize, our experimental results show that each storage scheme performs well in some cases, but none of the four storage schemes is a clear winner in all cases. While ADT seems to have a good performance in all cases, its storage size is much larger than that of the other schemes, thus may cause problems if the XML data trees have many nodes and many levels, or if the structure of the tree changes frequently. The ET suffers from the high cost of recursive joins for AD and ST queries. The NPT is good for AD queries when input node sets are small or the output is large, and is a bad choice for the opposite situation. The HT behaves exactly the same as the ET in dealing with PC queries and behaves as the NPT in dealing with AD queries.
5 Conclusions

In this paper, we compared four structure-generic relational storage schemes of XML data. We described table schema for each storage scheme and the translation of XML queries into SQL queries with a query decomposition method. Our experimental results indicate that each scheme performs well in some cases, but none of the four storage schemes is a clear winner in all cases. One possible direction for future research is to compare these structure-generic schemes with structure-specific storage schemes. Since RDBMSs do not understand the tree structure of XML data, the translated SQL queries need to be hand-tuned to obtain an optimal performance. One issue for further study is to give RDBMSs hints that would allow them to choose the optimal query evaluation plans.

References


