Modeling Nested Relationships in XML Documents Using Relational Databases

Olli Luoma

Department of Information Technology, University of Turku, Finland
olli.luoma@it.utu.fi

Abstract. Structural joins, i.e., operations that determine all occurrences of parent/child or ancestor/descendant relationships between node sets, are at the heart of XML management systems. To perform these joins, the systems exploit the information about the nested relationships between elements, attributes, and pieces of text in XML documents. Since performing the structural joins is often the most time-consuming phase in query evaluation, the method chosen to model the nested relationships has a considerable impact on the overall effectiveness of any XML management system. In this paper, we discuss four different methods for modeling the nested relationships using relational databases. We also propose a novel modeling method and present the results of our comprehensive performance experiments.

1 Introduction

Since its advent, XML [1], a self-describing markup language recommended by the World Wide Web Consortium, has rapidly been adopted as the standard for data representation and interchange in computer networks. In recent years, XML has increasingly been employed to perform more and more duties; in modern Web service environments, for example, XML can be used as a means to model software components which automatically construct themselves around the information expressed in XML [2]. The importance of XML has also passed over to the area of databases, where XML serves as a format to store heterogeneous information which cannot easily be organized into tables, columns, and rows [3]. Storing, querying and updating XML documents presents an interesting research problem and a plethora of work has been done on the subject.

From a technical viewpoint, an XML management system can be built in several ways. The first option is to build a specialized XML data manager. By building the data manager from scratch one is able to tailor indexing, storage management, and query optimization specifically to suit XML data. Obviously, this is a tempting option and many *native XML databases*, such as Lore [4] and NATIX [5], have been developed. However, native XML databases often suffer from scalability and stability problems and it will still take years before they reach the maturity that can be expected from a practicable XML management system.

Another option is to build an *XML management system on top of an object-oriented database* [6] [7]. At first glance, object-oriented databases with their rich data modeling capabilities may seem to be a perfect solution to the problem of XML data management, but this is not quite the case. Navigating large object hierarchies is a rigorous task, and hence the scalability of XML management systems built on top of an object-oriented database often leaves a lot to be desired. Relational databases, on the contrary, provide maturity, stability, portability, and scalability, so the third alternative, an *XML management system on top of a relational database*, is a very viable option. This alternative also allows the XML data and the relational data coexist in the same database, which is advantageous, since it is unlikely that XML databases can completely replace the existing relational database technology in any application area [3] [8].

Since relational databases were originally designed to support non-hierarchical data, a method for mapping XML data into relational schemas is needed. The existing mapping methods can roughly be divided into two categories [9]. In the *structure-mapping approach*, the database schemas are designed to represent the logical structure or the DTDs of the documents. The basic method is to create one relation for each element type [8], but more sophisticated methods in which the database schema is designed based on detailed analysis of the document DTDs have also been proposed [10]. Nonetheless, retaining the optimality of such a schema can be a rigorous task, since inserting new documents with different DTDs may result in redesigning the schema and rebuilding the relations.

The other method is the *model-mapping approach* in which the database schemas represent the generic constructs of the XML data model, i.e., element, attribute, and text nodes. The schemas designed according to the model-mapping approach are fixed, so documents can be stored without any information of their DTDs. Furthermore, having a fixed schema simplifies the transformation of XPath queries into SQL queries [11]. For the aforementioned reasons, we believe that the model-mapping approach will yield better results than its less generic counterpart.

When the model-mapping approach is pursued, an incoming document is first represented as an *XML tree*, a partially ordered, labeled tree in which each element, attribute, and text node corresponds to an element, attribute, or piece of text in the document, respectively; the ancestor/descendant relationships between the nodes correspond to the nested relationships between elements, attributes, and pieces of text [12]. This tree is then stored into the database and queried using the query facilities provided by the database management system.

In this paper, we focus on modeling the ancestor/descendant relationships. We discuss four different methods: parent/child index, pre-/postorder encoding, ancestor/descendant index, and our own proposal, the *ancestor/leaf index*, which maintains the ancestor information for the leaf nodes only. Obviously, this information can be used to perform structural joins between leaf nodes and inner nodes; structural joins between inner nodes are performed by checking whether common leaf node descendants for these inner nodes can be found.

The remainder of this paper is organized as follows. In section 2, we briefly review the related work, and in section 3, we present the different methods for modeling the nested relationships. The results of our experiments are presented in section 4; section 5 concludes this article and discusses our future work.

2 Related Work

From the vast amount XML-to-SQL query translation literature [11], the papers concentrating on mapping XML data into relations pursuing the model-mapping approach are relevant to our work. Previously, many such methods have been proposed, two of which are most essential here, namely XRel [9] and XParent [13]. In XRel, the nested relationships are modeled using region coordinates; a similar method was also presented in [3]. According to the authors, XRel provides better overall performance than Edge [8], a structure-mapping method proposed by Florescu and Kossmann. However, the performance evaluation was carried out using only one set of XML documents, so the scalability of this approach is still somewhat in doubt.

XParent, on the contrary, models the structural relationships between nodes using a parent/child index. The authors compared the effectiveness of their method against Edge and XRel, but again, the performance evaluation was carried out using one set of documents and a very limited set of queries. The authors also discussed using an ancestor/descendant index, but did not present any results from this alternative.

3 Modeling Nested Relationships

In this section, we present the relational schemas used in our tests. Our relational schemas are designed according to the XPath data model, and thus the element, attribute, and text nodes[1] are stored in three relations `Element`, `Attribute`, and `Text`, respectively. The nodes are identified using their preorder numbers which also preserve the information of the order among the nodes.

Since path expressions regularly appear in XPath queries, we preserve the information about the label paths of the nodes using a `Path` table which allows fast retrieval of the nodes according to their label paths. Similar decomposition of the nodes is used also in both XRel and XParent as well as in many native XML databases [4].

3.1 Parent/Child Index

The most obvious method for modeling the structural relationships between nodes is to build a *parent/child index* which, for a given set of nodes, allows fast

[1] According to the original XPath recommendation, there are seven different node types, but for simplicity, we have omitted root, comment, namespace, and processing instruction nodes.

retrieval of their parent or child nodes. Since each node has at most one parent, the identifier of the parent node can be inlined into the same relation with the node itself. Hence, this approach results in the following relational schema:

```
Path(PathId, PathExp)
Element(DocId, PreId, ParId, PathId)
Attribute(DocId, PreId, ParId, PathId, Value)
Text(DocId, PreId, ParId, PathId, Value)
```

In the above schema, the database attributes `PathId` and `PathExp` represent the path identifier and path expression, respectively. For technical reasons [9], the labels in a path are separated using "`#/`" instead of "`/`". In relations `Element`, `Attribute`, and `Text`, the database attributes `DocId`, `PreId`, `ParId`, and `PathId` represent the document identifier, node identifier, the identifier of the parent node, and path identifier, respectively. The database attribute `Value` represents the value of an attribute or text node. In each relation, the underlined set of attributes serves as the primary key.

However, if we are to retrieve all ancestors of a given node, we need recursive joins. This can be done using the linear recursion queries of the SQL3 standard, but if the database management system does not provide such feature, we can determine the number of needed joins by constructing the SQL queries in accordance with the document DTDs. Obviously, this can lead to very complicated queries and query generation, since each document may have a different DTD. Furthermore, XML documents often have to be managed without any DTDs at all, and thus this method lacks the genericity provided by the other three alternatives discussed in this paper.

3.2 Pre-/Postorder Encoding

The information about ancestor/descendant relationships can also be implicitly encoded by using *region coordinates* [9] or *pre-/postorder encoding* [14]. The structural joins can be performed by taking advantage of the following simple property of preorder and postorder numbes: 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. Since both preorder and postorder numbers are needed to perform the structural joins, we also include the postorder number `PostId` in the primary key although attributes `DocId` and `PreId` would be enough to identify a node.

```
Path(PathId, PathExp)
Element(DocId, PreId, PostId, PathId)
Attribute(DocId, PreId, PostId, PathId, Value)
Text(DocId, PreId, PostId, PathId, Value)
```

For brevity, we have omitted the actual algorithms for translating XPath queries into SQL queries; for a detailed description we refer the reader to [9]. The idea is to translate the XPath queries first into query trees which are then translated into SQL queries. For example, the query tree corresponding to the XPath query `//a/b[//c='ecm']/d` translates into the following SQL query:

```
SELECT DISTINCT e3.DocId, e3.PreId
FROM Element e1, Element e3, Text t2, Path p1, Path p2, Path p3
-- Match path expressions:
WHERE p1.PathExp LIKE '#%/a#/b'
AND p2.PathExp LIKE '#%/a#/b#%/c'
AND p3.PathExp LIKE '#%/a#/b#/d'
-- Retrieve nodes:
AND e1.PathId = p1.PathId
AND t2.PathId = p2.PathId
AND e3.PathId = p3.PathId
-- Match values:
AND t2.Value = 'ecm'
-- Perform structural joins:
AND t2.DocId = e1.DocId
AND t2.PreId > e1.PreId
AND t2.PostId < e1.PostId
AND e3.DocId = e1.DocId
AND e3.PreId > e1.PreId
AND e3.PostId < e1.PostId
```

Notice that the structural joins are performed using nonequijoins on preorder and postorder numbers, which can lead to scalability problems when querying large XML documents. However, splitting the data into many small documents can be expected to help, since part of the structural joins can then be performed using equijoins on document identifiers.

3.3 Ancestor/Descendant Index

By building an *ancestor/descendant index*, i.e., by calculating the transitive closure over the parent/child relation, we can perform the structural joins completely without the expensive nonequijoins. To maintain the ancestor/descendant information, we employ a new relation `AncDesc`.

```
Path(PathId, PathExp)
Element(DocId, PreId, PathId)
Attribute(DocId, PreId, PathId, Value)
Text(DocId, PreId, PathId, Value)
AncDesc(DocId, AncId, DescId)
```

Obviously, this rather extreme approach can result in a very large `AncDesc` table. However, since no nonequijoins are needed, this approach should usually perform better than the pre-/postorder encoding, and thus in applications where the query performance is of paramount importance, it might just pull its weight. If this approach is pursued, the XPath query `//a/b[//c='ecm']/d` translates into the following SQL query:

```
SELECT DISTINCT e3.DocId, e3.PreId
FROM Element e1, Element e3, Text t2, Path p1, Path p2, Path p3,
AncDesc d2, AncDesc d3
-- Match path expressions as in previous query.
-- Retrieve nodes as in previous query.
-- Match values as in previous query.
-- Perform structural joins:
AND t2.DocId = e1.DocId
AND d2.DocId = t2.DocId
AND d2.AncId = e1.PreId
AND d2.DescId = t2.PreId
AND e3.DocId = e1.DocId
AND d3.DocId = e3.DocId
AND d3.AncId = e1.PreId
AND d3.DescId = e3.PreId
```

3.4 Ancestor/Leaf Index

We can easily represent the ancestor/descendant information in a more compact manner by using an *ancestor/leaf index* which, essentially, is an ancestor/descendant built on the leaf nodes only. More formally, an ancestor/leaf index for an XML tree is a set of pairs (n_1, n_2), where n_2 is a leaf node and n_1 is an element node located on the path from n_2 to the root of the tree. According to this definition, nodes n_1 and n_2 do not have to be distinct, so the leaf nodes of type element serve a dual purpose. We maintain the ancestor/descendant information for the leaf nodes in relation `AncLeaf`.

```
Path(PathId, PathExp)
Element(DocId, PreId, PathId)
Attribute(DocId, PreId, PathId, Value)
Text(DocId, PreId, PathId, Value)
AncLeaf(DocId, AncId, LeafId)
```

The leaf nodes are joined with element nodes as they are joined using the ancestor/descendant index. The structural joins between element nodes can be performed by checking whether the nodes have common leaf node descendants. We must also be able to determine which set of the element nodes contains the ancestors. In many cases, this information can be deduced based on the label paths of the nodes, but there are some situations where this is not the case[2]. However, to simplify the query translation, we always use the lengths of the path expressions to determine which one of two sets of element nodes contains the ancestor or descendant nodes. Hence, the XPath query `//a/b[//c='ecm']/d` translates into the following SQL query:

[2] One example of such an instance would be evaluating query //a[a] using document <a><a><a/></a></a>. Without checking the heights, the ancestor/leaf index would also, incorrectly, return the leaf element <a/>.

```
SELECT DISTINCT e3.DocId, e3.PreId
FROM Element e1, Element e3, Text t2, Path p1, Path p2, Path p3,
AncLeaf f2, AncLeaf f3, AncLeaf f4
-- Match path expressions as in previous query.
-- Retrieve nodes as in previous query.
-- Match values as in previous query.
-- Perform structural joins:
-- Structural join between leaf nodes and element nodes:
AND t2.DocId = e1.DocId
AND f2.DocId = t2.DocId
AND f2.AncId = e1.PreId
AND f2.LeafId = t2.PreId
-- Structural join between element nodes:
AND LENGTH(p1.PathExp) < LENGTH(p3.PathExp)
AND e3.DocId = e1.DocId
AND f3.DocId = e3.DocId
AND f3.AncId = e1.PreId
AND f4.DocId = f3.DocId
AND f4.AncId = e3.PreId
AND f4.DescId = f3.DescId
```

Notice that the nonequijoin is now performed using the `Path` table which usually contains only a small number of rows, so this join is not as expensive as the nonequijoins performed using the pre-/postorder encoding.

4 Experimental Results

Because of the lack of genericity in the parent/child approach, we conducted the performance evaluation only for the pre-/postorder encoding (PP), the ancestor/descendant index (AD), and the ancestor/leaf index (AL). We used three different sets of XML documents: the 7.65 MB collection of Shakespeare's plays [15], a synthetic 111 MB XMark document generated using XMLgen [16], and a 127 MB XML document generated from the DBLP database [17]. The Shakespeare collection consisted of 37 documents and the other collections consisted of only one document.

Table 1. Database sizes for PP

	Shakespeare		XMark		DBLP	
Relation	Tuples	Size(MB)	Tuples	Size(MB)	Tuples	Size(MB)
Path	57	0	548	0	145	0
Element	179618	8	1666315	73	3332130	135
Attribute	0	0	381878	27	404276	28
Text	147383	12	1188922	139	3005857	201
Total	327058	20	3237663	239	6742408	364

Table 2. Database sizes for AD

	Shakespeare		XMark		DBLP	
Relation	Tuples	Size(MB)	Tuples	Size(MB)	Tuples	Size(MB)
Path	57	0	548	0	145	0
Element	179618	6	1666315	58	3332130	107
Attribute	0	0	381878	22	404276	23
Text	147385	11	1188922	129	3005857	175
AncDesc	1406939	44	16807315	520	16243275	513
Total	1733999	61	20044978	729	22985683	818

Table 3. Database sizes for AL

	Shakespeare		XMark		DBLP	
Relation	Tuples	Size(MB)	Tuples	Size(MB)	Tuples	Size(MB)
Path	57	0	548	0	145	0
Element	179618	6	1666315	58	3332130	107
Attribute	0	0	381878	22	404276	23
Text	147383	11	1188922	129	3005857	175
AncLeaf	729554	22	9278809	289	9904635	319
Total	1056612	39	12516472	498	16647043	624

Table 4. Query evaluation times (in seconds)

#	Query	PP	AD	AL	Tuples
1	`//ACT/TITLE`	0.00	0.00	0.00	185
2	`//ACT[//SPEAKER='EDMUND']`	0.48	0.03	0.02	5
3	`//ACT[//STAGEDIR='Aside']`	5.94	0.14	0.08	89
4	`//ACT[//SPEAKER='EDMUND']/TITLE`	4.16	4.76	21.48	5
5	`//people//profile`	0.17	0.16	0.16	12832
6	`//item[//location='Finland']`	2.91	1.54	1.39	16
7	`//item[@featured='yes']`	>300	1.13	0.45	2210
8	`//item[@featured='yes']//location`	>300	11.06	16.03	2210
9	`//article/author`	3.65	3.30	3.30	221465
10	`//article[@rating='SUPERB']`	8.39	2.00	2.00	11
11	`//article[author='Jukka Teuhola']`	15.34	0.22	0.09	27
12	`//article[author='Donald E. Knuth']/year`	61.91	2.05	1.94	55

We stored these collections into MySQL databases pursuing approaches PP, AD, and AL, and built indexes on `Element(PathId)`, `Attribute(PathId)`, `Text(PathId)`, `AncDesc(DocId, DescId)`, and `AncLeaf(DocId, LeafId)`. We also built indexes on first three characters of the attribute `Value` in relations `Attribute` and `Text`. The database sizes for PP, AD, and AL are presented in Tables 1-3. According to these experiments, AD results in three times and AL in two times larger database than PP; the size of `AncLeaf` table is roughly half of the size of the `AncDesc` table.

We evaluated the query performance of the three approaches by using four queries for each collection; the queries and the query evaluation times are presented in Table 4. Queries 1-4 were evaluated using the Shakespeare collection, queries 5-8 using the XMark document, and queries 9-12 using the DBLP document. Queries 1, 5, and 9 are simple path expression queries which do not involve structural joins, and thus all three approaches perform well.

Queries 2, 3, 6, 7, 10, and 11 involve a structural join between leaf node and inner node, so these queries provide information especially about the performance of PP against AD and AL. On large documents (queries 6, 7, 10, and 11), AD and AL quite clearly outperform PP, since in PP, the structural joins completely have to be performed using expensive nonequijoins on pre- and postorder numbers. However, when the collection is splitted into many documents (queries 2 and 3 on the Shakespeare collection), PP performs relatively well. Queries 7 and 8 involve structural joins between massive node sets of thousands of nodes, so PP performs very poorly. Thus, it can be argued that the scalability of PP leaves a lot to be desired.

Queries 4, 8, and 12 involve structural joins also between inner nodes, so these queries can be used to compare AD against AL. Overall, AL seems to perform almost as well as AD, but as an interesting detail, we found that both AD and PP outperform AL on query 4. Thus, although AL does not suffer from the severe scalability problems of PP demonstrated by queries 7 and 8, joining node sets with large number of leaf node descendants pursuing AL is still rather expensive. This finding also suggests that splitting large documents into smaller entities before inserting them into the database would lead to considerable performance gains in PP, since part of the structural joins can then be carried out using equijoins on document identifiers.

5 Concluding Remarks

In this paper, we discussed four different methods for modeling the nested relationships between elements, attributes, and pieces of text in XML documents. We also proposed a new approach, namely the ancestor/leaf index. We presented the relational schemas designed according to these models and presented our experimental results which clearly demonstrated the trade-off between storage consumption and query performance. When building an XML management system on a relational database, one should consider both this trade-off and the requirements of the application area. For example, if the management system will be used to store only web pages, using ancestor/descendant index or ancestor/leaf index would be a waste of space, since web pages written in XML are usually only kilobytes in size.

One interesting detail in our experimental results was the relatively good performance of pre-/postorder encoding when many small documents were queried. Considering that this approach only consumes half of the disk space consumed by the ancestor/leaf approach, it might be worthwhile to develop methods for splitting large XML trees before inserting them into the database. In this case, we obviously need ways to perform structural joins between node sets that reside in separate trees, which presents an interesting research problem.

References

1. World Wide Web Consortium. Extensible Markup Language (XML) 1.0. `http://www.w3c.org/TR/REC-xml`, 2000.
2. D. Florescu, A. Grünhagen, and D. Kossmann. An XML Programming Language for Web Service Specification and Composition. *IEEE Data Engineering Bulletin*, 24(2): 48-56, 2001.
3. A. B. Chaudri, A. Rashid, and R. Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems.* Addison-Wesley, 2003.
4. 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.
5. 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.
6. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. of 1994 ACM SIGMOD Intl Conf. on Management of Data*, pages 313-324, 1994.
7. R. v. Zvol, P. Apers, and A. Wilschut. Modelling and querying semistructured data with MOA. In *Proc. of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats.*, 1999.
8. D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report 3684, INRIA, 1999.
9. 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.
10. 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.
11. R. Krishnamurthy, R. Kaushik, 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.
12. World Wide Web Consortium. XML Path Language (XPath) Version 1.0. `http://www.w3c.org/TR/xpath`, 2000.
13. 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.
14. P.F. Dietz. Maintaining order in a linked list. In *Proc. of the 14th ACM Symp. on Theory of Computing*, pages 122-127, 1982.
15. J. Bosak. The complete plays of Shakespeare marked up in XML. `http://www.ibiblio.org/xml/examples/shakespeare`
16. R. Busse, M. Carey, D. Florescu, M. Kersten, I. Manolescu, A. Schmidt, and F. Waas. XMark - an XML benchmark project. `http://monetdb.cwi.nl/xml/index.html`
17. M. Ley. Digital bibliography library project. `http://dblp.uni-trier.de/`