A Structure-Based Filtering Method for XML Management Systems

Olli Luoma

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

Abstract. To answer queries, many XML management systems perform structural joins, i.e., they determine all occurences of parent/child or ancestor/descendant relationships between node sets. These joins are often one of the most time-consuming phases in query evaluation, so it is desirable to reduce the size of the node sets before performing the joins. This problem has earlier been approached by using signatures built on the content of the nodes, but in this paper, we propose a novel method in which the nodes are filtered based on the structural properties of their subtrees. To achieve this, we use a schema graph which summarizes the structures of XML documents more accurately than conventional summarization methods.

1 Introduction

Because of its simplicity and flexibility, Extensible Markup Language (XML) [1] has proved very useful in many application areas. Today, XML is used not only as a standard for data exchange, but also as a core for development and deployment platforms, such as Microsoft .NET. Furthermore, there are many application areas, such as bioinformatics, where XML serves as a format to store heterogeneous information [2]. Storing, querying, and updating XML documents presents an interesting research area and there has indeed been a significant amount of research on XML data management.

Every well-formed XML document can be represented as an XML tree, a partially ordered labeled tree in which each element, attribute, and text node[1] corresponds to an element, attribute, or piece of text in the document, and the ancestor/descendant relationships between the nodes correspond to the nesting relationships between elements, attributes, and pieces of text [3]. XML trees can be modeled using several different methods, such as parent/child indexes, ancestor/descendant indexes, pre-/postorder encoding [4], absolute or relative region coordinates [5], and virtual nodes [6]. All of these methods encode the information of parent/child or ancestor/descendant relationships which is needed to perform structural joins.

[1] According to the original XPath recommendation there are seven different node types, four of which have been omitted here to simplify the discussion.

F. Galindo et al. (Eds.): DEXA 2004, LNCS 3180, pp. 401–410, 2004.

In many systems, query processing is speeded up by using a *schema tree*, a structural summary which partitions the nodes of an XML tree into equivalence classes according to their label paths. In [7], however, we proposed a schema graph as an alternative way to summarize the structures of XML documents. Since a schema graph creates a more precise, structure-based partitioning of the nodes, it can be used for two purposes. Firstly, queries that select nodes based only on their structural properties, such as "find all employees who have children", can be evaluated very quickly [7]. Secondly, queries that select nodes based on both structure and content, can be speeded up by using the schema graph as a filtering structure. For example, when evaluating query "find all employees who have a child named Alina", we can filter out those employees who do not have any children in the first place.

The remainder of this paper is organized as follows. Section 3 explores the possibilities of a schema graph as a filtering structure. Section 4 describes Xeek, our prototype system, and in Section 5, the results of experimental evaluation are presented. Section 6 concludes this article and discusses our future work.

2 Related Work

A lot of work has been carried out to develop methods to manage XML documents. The proposed methods can be divided into three categories. In the *flat streams approach*, the documents are considered as byte streams. Large streams are distributed on disk pages using the file system or a BLOB manager in a database management system. Since accessing the structures of the documents requires parsing, this method is hardly suitable for XML management systems. In the *metamodeling approach*, the documents are first represented as trees which are then stored into a database. This provides fast access to the XML trees and, consequently, this method has been used in many XML management systems [8] [9]. The *mixed approach* aims to combine the previous two approaches. Some systems store the data in two redundant repositories, one flat and one metamodeled, which allows fast retrieval but creates significant storage overhead [10]. The other option is the hybrid approach in which the coarser structures of the documents are modeled as trees and finer structures as flat streams [11].

Many content-based filtering methods derived from information retrieval have been applied to structured documents [6] [12]. However, a structure-based filtering method for XML management systems has previously been proposed only by Park and Kim [13]. Their idea is to attach a signature built over the labels of the descendants to each node of the XML tree. This signature can then be used to filter out the nodes which cannot satisfy the conditions set by a query. However, since their method requires accessing the XML tree before filtering, it is fundamentally different from the method proposed in this paper. Furthermore, our schema graph partitions the nodes of an XML tree very accurately, so nodes can be filtered based on complex structural conditions. This can often reduce the size of joined node sets more than a filtering method that relies on simple label signatures.

3 Using a Schema Graph as a Filtering Structure

As mentioned above, a *schema graph* creates an accurate, structure-based partitioning of the nodes of an XML tree, which makes it possible to filter the nodes based on their structural position. To define a schema graph formally we first need to define two concepts for the nodes of an XML tree, namely a *label* (Definition 1), and a *label path* (Definition 2).

Definition 1. *A label $l(n)$ for node n is the name of the corresponding element if n is an element node and the name of the corresponding attribute preceded by an @-sign if n is an attribute node.*

Notice that Definition 1 does not define any label for text nodes. However, text nodes, as well as element and attribute nodes, do have a label path which is defined in Definition 2.

Definition 2. *Let $s(n)$ denote the parent of node n. The label path $p(n)$ of an element or attribute node n is $/l(n)$ if n is a root node, and $p(s(n))/l(n)$ otherwise. The label path of a text node n is $p(s(n))$.*

Many XML management systems, such as Lore [8], BUS [9], and XRel [14], summarize the documents using a *schema tree*, an index structure which partitions the nodes according to their label paths[2]. A schema tree allows fast retrieval of the nodes based on their label paths, but to achieve structure-based filtering on the summary level we need a summarization method which creates more accurate partitioning. Thus, our filtering method takes advantage of a schema graph, an acyclic, directed graph, which allows fast retrieval of the nodes based not only on their label paths, but also on the structures of their subtrees. The partitioning created by a schema graph is described formally in Definition 3; the difference between schema tree and schema graph is illustrated in Fig. 1.

Definition 3. *Let N denote the set of nodes in an XML tree and let $C(n)$ denote the set of attribute and element nodes that are children of node n. A vertex in a schema graph corresponds to an equivalence class $[n]_g$ induced by an equivalence relation $\equiv_g$ on N such that for any $n_1, n_2 \in N$, $n_1 \equiv_g n_2$, iff*

$$C(n_1) = \emptyset \wedge C(n_2) = \emptyset \wedge p(n_1) = p(n_2), \text{ or}$$

$$C(n_1) \neq \emptyset \wedge C(n_2) \neq \emptyset \wedge (\forall c_1 \in C(n_1) : \exists c_2 \in C(n_2) : c_1 \equiv_g c_2) \wedge (\forall c_2 \in C(n_2) : \exists c_1 \in C(n_1) : c_2 \equiv_g c_1).$$

In simple terms, two nodes in an XML tree are equivalent if their label paths are identical and their subtrees are structurally similar. Notice that according to Definition 3 there cannot exist two nodes $n_1, n_2 \in N$ such that $p(n_1) \neq p(n_2)$ and $n_1 \equiv_g n_2$, so we can define a label path also for a vertex in a schema graph in Definition 4.

[2] In Lore, for example, this index is called a *DataGuide*, but in the current paper, any structure that partitions the nodes according to their label paths is called a schema tree.

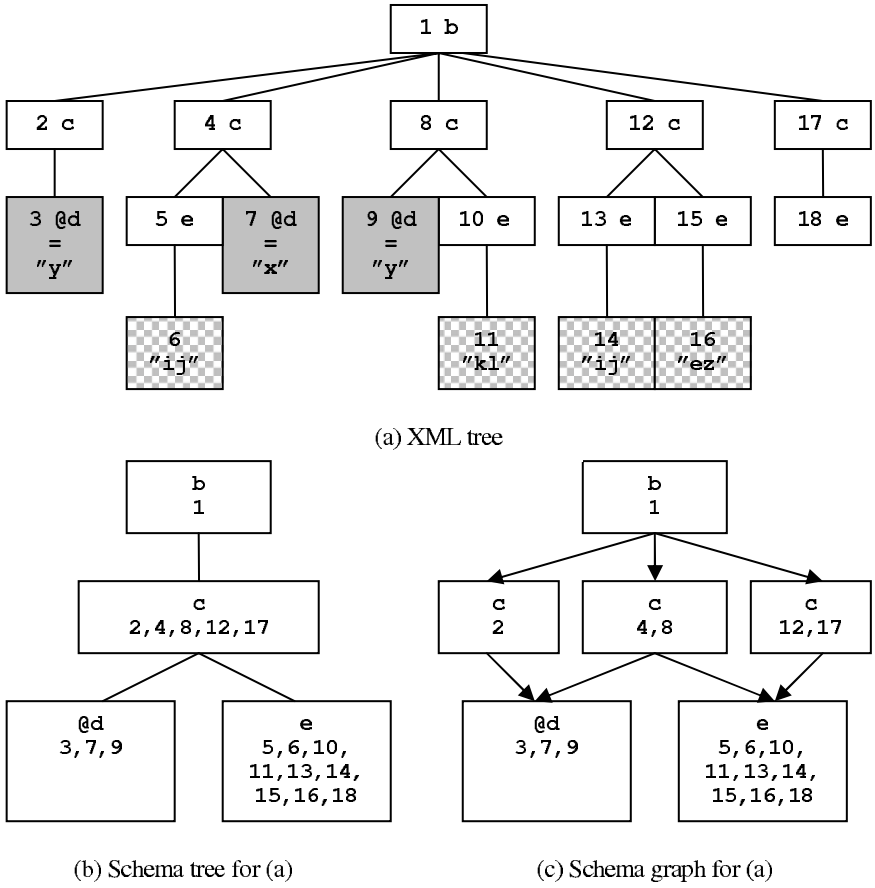

Fig. 1. An XML tree, a schema tree, and a schema graph

Definition 4. *A label path $p(v)$ for a vertex v in a schema graph is the label path of any node belonging to the equivalence class corresponding to v.*

Consider, for example, evaluating XPath query `/b/c[//@d="y"]` using the XML tree and the schema tree presented in Fig. 1. We first use the schema tree to find the set $N_1 = \{2, 4, 8, 12, 17\}$ of element nodes with the label path `/b/c` and then the set $N_2 = \{3, 7, 9\}$ of attribute nodes with label path matching `/b/c//@d`. Both of these operations can be performed very quickly. After this, we scan N_2 to find the set $N_3 = \{3, 9\}$ of attribute nodes with label path matching `/b/c//@d` and value "y". Sets N_1 and N_3 are then structurally joined to find all nodes in N_1 that have a descendant in N_3, so the result of this query is $\{2, 8\}$.

Let us now consider evaluating the same query using the schema graph presented in Fig. 1 to filter the nodes. First, we use the schema graph to find the set $N_1 = \{2, 4, 8\}$ of element nodes that satisfy the condition `/b/c[//@d]`, i.e.,

the set of nodes that belong to the equivalence class corresponding to a vertex v in the schema graph such that $p(v) =$ `/b/c`, and from which a vertex with label path matching `/b/c//@d` can be reached. Notice that this set is now smaller than the set N_1 obtained using the schema tree. We then proceed similarly as in our first example to find the set $N_3 = \{3, 9\}$ of attribute nodes with label path matching `/b/c//@d` and value "y". After this, we structurally join sets N_1 and N_3 to get the final result $\{2, 8\}$.

The algorithms needed for evaluating queries using a schema graph as a filtering structure are presented in Fig. 2. As in XRel [14], for example, queries have to be first represented as query trees in which exactly one node is active and each node has two features, a label path and a value; for element nodes, the value is defined as an empty string. Notice that algorithm `nodeVertexMatch` is intrinsically complex as such[3], but this problem can be avoided rather easily. In our prototype system Xeek, for example, we achieved good results by using a relational database system to implement the schema graph redundantly as a set of pairs (v_1, v_2), where vertex v_2 is reachable from vertex v_1.

Notice also that our filtering method can take full advantage of the structural conditions involved in complex queries, which sets it apart from the method proposed by Park and Kim [13]. For example, to answer XPath query `//s[s[d/n="Aino"][s/n="Aapo"]]`, we join four node sets, set N_1 of nodes that satisfy the condition `//s[s[d/n][s/n]]`, set N_2 of nodes that satisfy the condition `//s/s[d/n][s/n]`, set N_3 of nodes with label path `//s/s/d/n` and value "Aino", and set N_4 of nodes with label path `//s/s/s/n` and value "Aapo". Assuming that every label present in our query is part of the signature, the filtering method proposed by Park and Kim would have resulted in N_1 satisfying a much looser condition `//s[//s][//d][//n]` and N_2 satisfying `//s/s[//d][//n][//s]`. Furthermore, in our method, the filtering actually takes place on the summary level, i.e., before accessing the XML tree.

4 Xeek - A Prototype System

To test our idea of structure-based filtering, we implemented Xeek, a prototype system based on a relational database. The basic Xeek schema consists of five relations:

```
Element(DocId, Start, End, VertexId)
Attribute(DocId, Start, End, VertexId, Value)
Text(DocId, Start, End, VertexId, Value)
Vertex(VertexId, PathExp)
ReachVertex(VertexId, ReachVertexId)
NodeSet(NodeSetId, DocId, Start, End)
```

[3] Since implementing the schema graph as a graph is hardly practical, we actually did define the schema graph as a partitioning criterion, not as a graph structure, in Definition 3.

```
evaluate(n)
// in: Node n of a query tree
// out: Node set corresponding to the query tree defined by n
   N = getNodes(n)
   for each m in children(n) such that not activePath(m) do
     if contentCondition(m) then
       N = joinFirst(N, evaluate(m))
   for each l in children(n) such that activePath(l) do
     N = joinSecond(N, evaluate(l))
   return N

joinFirst(N1, N2)
// in: Node sets N1, N2
// out: Nodes of N1 that have a descendant in N2

joinSecond(N1, N2)
// in: Node sets N1, N2
// out: Nodes of N2 that have an ancestor in N1

contentCondition(n)
// in: Node n of a query tree
// out: TRUE iff n or some of its descendants has a value

activePath(n)
// in: Node n of a query tree
// out: TRUE iff n or some of its descendants is the active node

getNodes(n)
// in: Node n of a query tree
// out: All nodes in an XML tree matching the query tree defined by n
   if value(n) == empty then
    return all nodes of an XML tree corresponding to any vertex v in a
    schema graph such that nodeVertexMatch(n, v)
   else
    return all nodes m of an XML tree corresponding to any vertex w in a
    schema graph such that nodeVertexMatch(n, w) and value(m) = value(n)

nodeVertexMatch(n, v)
// in: Node n of a query tree, vertex v of schema graph
// out: TRUE iff v matches the query tree defined by n
   if children(n) == empty then
    return (labelPath(n) matches labelPath(v))
   else
    return (for each node m in children(n) there exists vertex w such
    that w is reachable from vertex v and nodeVertexMatch(m, w))
```

Fig. 2. Algorithms for evaluating queries using a schema graph

Each tuple in relations `Element`, `Attribute`, and `Text` corresponds to an element, attribute, or text node, respectively. The database attributes `DocId` and `VertexId` represent the document identifier and the identifier of the schema graph vertex, respectively, and the database attributes `Start` and `End` represent the *region coordinates* of a node, i.e., a pair of numbers that point to the start and end positions of the corresponding part of the XML document. The database attribute `Value` represents the value of an attribute or text node. The underlined database attributes serve as the primary keys in each of the relations.

Relations `Vertex` and `ReachVertex` are used to model the schema graph. Each tuple in relation `Vertex` corresponds to a vertex in a schema graph. The database attributes `VertexId` and `PathExp` correspond to the identifier and the label path of the vertex, respectively. For technical reasons [14], the labels in a path are separated using "#/" instead of "/". Each tuple in relation `ReachVertex` corresponds to a pair of vertices in a schema graph in which the vertex identified by `ReachVertexId` is reachable from the vertex identified by `VertexId`. `NodeSet` is a temporary relation used to store the resulting node sets corresponding to each node in a query tree during query evaluation.

In Xeek, the query evaluation consists of three phases. First, Xeek validates a query and generates a query tree using similar methods to those proposed in [14]. In the second phase, Xeek *materializes* the nodes needed for structural joins, i.e., stores the node sets corresponding to each node of the query tree into relation `NodeSet`. For example, to materialize the node sets corresponding to query `//c[@d='y']`, Xeek executes the following SQL queries:

```
INSERT INTO NodeSet SELECT 1, e1.DocId, e1.Start, e1.End
FROM Vertex v1, Vertex v2, ReachVertex r2, Element e1
WHERE v1.PathExp LIKE '#%/c' AND v2.PathExp LIKE '#%/c#/@d'
AND r2.VertexId = v1.VertexId AND r2.ReachVertexId = v2.VertexId
AND e1.VertexId = v1.VertexId;

INSERT INTO NodeSet SELECT 2, a1.DocId, a1.Start, a1.End
FROM Vertex v1, Attribute a1
WHERE v1.PathExp LIKE '#%/c#/@d' AND a1.VertexId = v1.VertexId
AND a1.Value = 'y';
```

The first of these two SQL queries materializes the set of nodes satisfying condition `//c[@d]`. Notice that without filtering we would have materialized the set of nodes that only satisfy the condition `//c`.

In the third phase, Xeek performs the structural joins on the materialized node sets. The result of our example query can be obtained from the `NodeSet` table using the following SQL query:

```
SELECT n1.DocId, n1.Start, n1.End FROM NodeSet n1, NodeSet n2
WHERE n1.NodeSetId = 1 AND n2.NodeSetId = 2
AND n1.DocId = n2.DocId AND n2.Start BETWEEN n1.Start AND n1.End;
```

5 Experimental Results

We evaluated the effectiveness of our method using three different sets of XML documents: the 7.65 MB collection of Shakespeare's plays marked up in XML [15], a synthetic 111 MB XMark document generated using XMLgen [16], and a 127 MB XML document generated from the DBLP database [17]. We stored these documents into a MySQL database using Xeek; the sizes of the relations in each case are presented in Table 1.

Notice that, although Xeek models the schema graph redundantly, the sizes of relations `Vertex` and `ReachVertex` are rather modest for the Shakespeare collection and the DBLP document. The structure of the deeply nested XMark document, in contrast, is much more irregular, so the combined size of the relations `Vertex` and `ReachVertex` is rather large, about 10 % of the database. It is worth noticing, however, that the best-case and worst-case sizes of both schema tree and schema graph are identical [7]. The best-case behaviour of both structures is yielded by an XML tree where all element nodes that share the same label path also share the same subtree structures. The worst-case behaviour is yielded by an XML tree where every node has a different label.

Table 2 shows our small but versatile query set for the different collections and Table 3 the materialization and the join times for each query, both with filtering and without filtering. Notice that the materialization times with filtering are seldom much longer than without filtering, although the filtering requires more accesses to the schema graph in the materialization phase. In many cases, the materialization times with filtering are even shorter, because filtering reduces the amount of tuples that have to be written in the `NodeSet` table.

As our results demonstrate, the small penalty paid in the materialization phase usually pays off in the join phase. In the best case, structural filtering can improve the query evaluation time by an order of magnitude. When evaluating query Q6, for example, filtering reduces the sizes of node sets to be structurally joined to 713, 2210, and 14314 from 21750, 2210, and 14314, respectively, which reduces the time needed for structural joins considerably.

One interesting detail in the results is the relatively good join performance while evaluating queries using the Shakespeare collection. Since the Shakespeare collection consists of many XML documents, the structural joins can partially be performed using equijoins on document identifiers. In contrast, the joins on the XMark and DBLP documents have to be completely performed using much slower nonequijoins on the region coordinates.

6 Conclusion and Future Work

In this paper, we proposed a new structure-based filtering method for XML management systems which is utilizes an accurate, structure-based partitioning created by a schema graph. Our filtering method can take advantage of even the most complex structural conditions set by queries, which sets it apart from previous proposals. We implemented our method using a relational database and presented the positive results of our performance studies.

Table 1. Database sizes

	Shakespeare		XMark		DBLP	
Relation	Tuples	Size(MB)	Tuples	Size(MB)	Tuples	Size(MB)
Element	179618	11.0	1666315	112	3332130	164
Attribute	0	0	381878	30.9	404276	32.9
Text	147383	16.4	988027	145	3003323	234
Vertex	232	0.02	52747	3.75	552	0.04
ReachVertex	3231	0.08	1041318	29.3	3779	0.13
Total	330464	27.5	4130285	321	6744060	431

Table 2. Query set

#	Data	Query
Q1	Shak.	`/PLAY/ACT[//SPEAKER='EDMUND']`
Q2	Shak.	`//SPEECH[SPEAKER='HAMLET']/STAGEDIR`
Q3	Shak.	`//SPEECH[SPEAKER='KING LEAR'][STAGEDIR='Aside']`
Q4	XMark	`//person[//interest/@category='category620']`
Q5	XMark	`//item[@featured='yes']`
Q6	XMark	`//item[@featured='yes']//mail//keyword`
Q7	DBLP	`//article[author='Jukka Teuhola']`
Q8	DBLP	`//inproceedings[crossref='conf/safecomp/1998']`
Q9	DBLP	`//article[@rating='SUPERB']/author`

Table 3. Query performance

	With filtering		Without filtering	
#	Mat.(sec)	Join(sec)	Mat.(sec)	Join(sec)
Q1	0.60	0.00	0.77	0.00
Q2	1.36	0.03	0.57	0.52
Q3	1.33	0.02	0.57	0.35
Q4	0.49	0.45	0.64	1.09
Q5	1.53	6.13	0.98	61.02
Q6	1.64	14.36	1.10	>600
Q7	2.19	1.48	2.18	1.48
Q8	2.09	4.05	4.28	7.83
Q9	3.67	7.06	9.88	11.27

We plan to extend Xeek to a full-fledged XML storage system. To achieve this, we need to develop ways to efficiently construct the result documents from the result sets. During the performance studies we encountered some cases where the join order of the tables determined by the MySQL query optimizer was far from optimal. Thus, we will seek ways to avoid the pitfalls of bad query optimization by studying how different optimization and join methods used in RDBMSs affect the performance when the database is used to manage XML trees.

References

1. World Wide Web Consortium. Extensible Markup Language (XML) 1.0. `http://www.w3c.org/TR/REC-xml`, 2000.
2. A.B. Chaudri, A. Rashid, and R. Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems.* Addison-Wesley, 2003.
3. World Wide Web Consortium. XML Path Language (XPath) Version 1.0. `http://www.w3c.org/TR/xpath`, 2000.
4. P.F. Dietz. Maintaining order in a linked list. In *Proc. of the 14th ACM Symposium on Theory of Computing*, pages 122-127, 1982.
5. D.D. Kha, M. Yoshikawa, and S. Uemura. An XML indexing structure with relative region coordinate. In *Proc. of the 17th IEEE Intl Conf. on Data Engineering*, pages 212-220, 2001.
6. Y.K. Lee, S. Yoo, K. Yoon, and B. Berra. Index structures for structured documents. In *Proc. of the 1st Intl Conf. on Digital Libraries*, pages 91-99, 1996.
7. O. Luoma. Indexing XML data with a schema graph. In *Proc. of the IASTED Intl Conf. on Databases and Applications*, pages 274-279, 2004.
8. 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.
9. D. Shin, H. Jang, and H. Jin. BUS: An effective indexing and retrieval scheme in structured documents. In *Proc. of the 3rd ACM Intl Conf. on Digital Libraries*, pages 235-243, 1998.
10. T.W. Tak and J. Annevelink. Integrating a structured-text retrieval system with an object-oriented database system. In *Proc. of the 20th Intl Conf. on Very Large Databases*, pages 740-749, 1994.
11. 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.
12. Y. Chen and K. Aberer. Combining pat-trees and signature files for query evaluation in document databases. In *Proc. of the 10th Intl Conf. on Database and Expert Systems Applications*, pages 473-484, 1999.
13. S. Park and H.J. Kim. A new query processing techique for XML based on signature. In *Proc. of the 7th Intl Conf. on Database Systems for Advanced Applications*, pages 22-31, 2001.
14. 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.
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/`