A New Inlining Algorithm for Mapping XML DTDs to Relational Schemas

Shiyong Lu, Yezhou Sun, Mustafa Atay, Farshad Fotouhi

Department Of Computer Science
Wayne State University
Detroit, MI 48202
{shiyong, sunny, matay, fotouhi}@cs.wayne.edu

Abstract. XML is rapidly emerging on the World Wide Web as a standard for representing and exchanging data. It is critical to have efficient mechanisms to store and query XML documents to exploit the full power of this new technology. While one approach is to develop native XML repositories that support XML data models and query languages directly, the other approach is to take advantage of the mature technologies that are provided by current relational or object-relational DBMSs. There is active research along both approaches and it is still not clear which one is better than the other. We continue our effort on the second approach. In particular, we have developed an efficient algorithm which takes an XML DTD as input and produces a relational schema as output for storing and querying XML documents conforming to the input DTD. Our algorithm features several significant improvements over the shared-inlining algorithm including overcoming its incompleteness, eliminating redundancies caused by shared elements, performing optimizations and enhancing efficiency.

1 Introduction

With the trend of increasing amount of XML documents on the World Wide Web, it is critical to have efficient mechanisms to store and query XML documents to exploit the full power of this new technology. As a result, various XML query languages have been proposed such as XML-QL [7], XQL [14], Lorel [12] and XML-GL [5], and more recently XQuery [6], and XML has become one of the most active research fields attracting different researchers from various communities.

Currently, two approaches are being investigated for storing and querying XML data. One approach is to develop native XML repositories that support XML data models and query languages directly. This includes Software AG's Tamino [2] and eXcelon's XIS [1], among others. The other approach is to take advantage of the mature technologies that are provided by current relational or object-relational DBMSs. The major challenges of this approach include: (1) XML data model needs to be mapped into the target model such as the relational model; (2) queries posed in XML query languages need to be translated into ones in the target query languages such as SQL or OQL; and (3) the query results from the target database engines need to be published back to XML format. Recently, Kurt and Atay have performed an experimental study to compare the efficiency of these two approaches [13]. However, since both approaches are still under active research and development, it is too early to conclude which one is better than the other.

Related work. Several mechanisms have been proposed to store XML data in relational or object-relational databases [8] [10] [16] [11] and publish relational or object-relational data as XML data [15] [4] [9]. Some of them use XML DTDs [16] and others consider situations in which DTDs are not available [8] [10] [11]. Two recent evaluations [19] [11] of different XML storage strategies indicate that the shared-inlining algorithm [16] overperforms other strategies in data representation and performance across different datasets and different queries when DTDs are available.

In this paper, we propose a new inlining algorithm that maps XML DTDs to relational schemas. Our algorithm is inspired by the shared-inlining algorithm [16] but features several improvements over it. We will discuss these improvements in Section 3.3.

Organization. The rest of the paper is organized as follows. Section 2 gives a brief overview of XML Document Type Definitions (DTDs), Section 3 describes our new inlining algorithm that maps an input DTD to a relational schema in terms of three steps: (1) simplifying input DTDs (Section 3.1); (2) creating and inlining DTD graphs (Section 3.2); (3) generating relational schemas (Section 3.3). The section ends with a discussion of the improvements we have made over the shared-inlining algorithm, which is considered as the best strategy when DTDs are available [19] [11]. A full evaluation and comparison is underway and will be presented in the near future. Section 3.4 illustrates the three steps of our algorithm using a real input DTD, and demonstrates how XML documents conforming to the DTD can be stored. Finally, Section 4 concludes the paper and provides some directions for future work.

2 XML DTDs

XML Document Type Definitions (DTDs) [3] describe the structure of XML documents and are considered as the schemas for XML documents. In this paper, we model both XML elements and XML attributes as XML elements since XML attributes can be considered as XML elements without further nesting structure. A DTD D is modeled as a set of *XML element definitions* $\{d_1, d_2, \cdots, d_k\}$. Each XML element definition d_i $(i = 1, \cdots, k)$ is in the form of $n_i = e_i$, where n_i is the name of an XML element, and e_i is a *DTD expression*. Each DTD expression is composed from XML element names (called *primitive DTD expressions*) and other DTD subexpressions using the following operators:

- **Tuple operator**. $(e_1, e_2, \cdots, e_n)$ denotes a tuple of DTD subexpressions. In particular, we consider (e) is a singleton tuple. The tuple operator is denoted by ",".

– **Star operator**. e^* represents zero or more occurrences of subexpression e.
– **Plus operator**. e^+ represents one or more occurrences of subexpression e.
– **Optional operator**. $e?$ represents an optional occurrence (0 or 1) of subexpression e.
– **Or operator**. $(e_1 \mid e_2 \mid \cdots \mid e_n)$ represents one occurrence of one of the subexpressions e_1, e_2, $\cdots$, e_n.

We ignore the encoding mechanisms that are used in data types PCDATA and CDATA and model both of them as data type *string*. The DOCTYPE declaration states which XML element will be used as the schema for XML documents. This XML element is called the *root* element. However, we assume that arbitrary XML elements defined in the DTD might be selected, inserted, deleted and updated individually. We define a DTD expression formally as follows.

Definition 1. *A DTD expression e is defined recursively in the following BNF notation where n range over XML element names and e_1, $\cdots$, e_n range over DTD expressions.*

$$\begin{aligned} e ::= {} & \mathit{string} \mid n \mid e^+ \mid e^* \mid e? \\ & \mid (e_1, \cdots, e_n) \mid (e_1 \mid \cdots \mid e_n) \end{aligned}$$

where the symbol "::=" should be read as "is defined as" and "|" as "or".

3 Mapping XML DTDs to relational schemas

In this section, we propose a new inlining algorithm that maps an input DTD to a relational schema. The algorithm contains the following three steps:

1. *Simplifying DTDs.* Since a DTD expression might be very complex due to its hierarchical nesting capability, this step greatly simplifies the mapping procedure.
2. *Creating and inlining DTD graphs.* We create the corresponding DTD graph based on the simplified DTD, and then inline as many descendant elements as possible to an XML element. In contrast to the shared-inlining algorithm, our inlining rules eliminate the redundancy caused by shared elements in the generated relational schema and can deal with arbitrary input DTDs including those that contain arbitrary cycles.
3. *Generating relational schemas.* After a DTD graph is inlined, we generate a relational schema based on it.

We describe these three steps in Sections 3.1, 3.2 and 3.2 respectively and conclude the section by a discussion of the improvements we have made over the shared-inlining algorithm. Finally, Section 3.4 illustrates these steps using a real XML DTD and demonstrates how XML documents conforming to this DTD can be stored based on the generated schema.

3.1 Simplifying DTDs

Most complexity of a DTD comes from the complexity of DTD expressions such as <!ELEMENT a ((b+, c*, d?)?, (e?, f, (g*, h?)+)?)>. However, as far as an XML query language is concerned, what matters is the siblings and parent-child relationships between elements. We apply the transformation rules listed in Figure 1 in the given order:

1. Apply rule 1 recursively and the resulting DTD will not contain +.
2. Apply rule 2 recursively and the resulting DTD will not contain + and ?.
3. Apply rule 3 recursively and the resulting DTD will not contain +, ? and |.
4. Apply rules 4(a) and 4(b) recursively and the resulting DTD will take the form $(e_1, e_2, \cdots, e_n)$. Each $e_i = e$ or e^* $(i = 1, \cdots, n)$ where e is an element name. Therefore, a DTD is in some flattened form after this step.
5. Apply rules 5(a), 5(b), 5(c) and 5(d) recursively and the resulting DTD will take the form $(e_1, e_2, \cdots, e_n)$ such that each e_i contains distinct element name.

1. $e^+ \rightarrow e^*$.
2. $e? \rightarrow e$.
3. $(e_1 \mid \cdots \mid e_n) \rightarrow (e_1, \cdots, e_n)$.
4. (a) $(e_1, \cdots, e_n)^* \rightarrow (e_1^*, \cdots, e_n^*)$.
   (b) $e^{**} \rightarrow e*$.
5. (a) $\cdots, e, \cdots, e, \cdots \rightarrow \cdots, e^*, \cdots, \cdots$.
   (b) $\cdots, e, \cdots, e^*, \cdots \rightarrow \cdots, e^*, \cdots, \cdots$.
   (c) $\cdots, e^*, \cdots, e, \cdots \rightarrow \cdots, e^*, \cdots, \cdots$.
   (d) $\cdots, e^*, \cdots, e^*, \cdots \rightarrow \cdots, e^*, \cdots, \cdots$.

Fig. 1. DTD simplification rules

From an XML query language's point of view, two pieces of information are essential: (1) The parent-child relationships between XML elements; and (2) the relative order relationships between siblings. The above transformation maintains the former but not the later. Fortunately, we can introduce an ordinal attribute for each generated relation to encode the order of XML elements when an XML element (and its containing subelements) is inserted into the database, so that any XML query conforming to the input DTD can be evaluated over the generated relational schema.

Example 1. Use the above simplification procedure, one can transform <!ELEMENT a ((b+, c*, d?)?, (e?, f, (g*, h?)+)?)> to a simplified version
<!ELEMENT a (b*, c*, d, e, f, g*, h*)>.

The following theorem indicates that our simplification procedure is complete and in addition, the resulting DTD expression is a tuple of element names or their stars.

Theorem 1. *Our DTD simplification procedure is complete in the sense that it accepts every input DTD and each resulting DTD expression is in the form of* $(e_1, e_2, \cdots, e_n)$ *where* $e_i = e$ *or* e^* $(i = 1, \cdots, n)$*,* e *is an element name and each* e_i *contains a distinct XML element name.*

Proof. We omit the proof since it is obvious.

Discussion. Compared to the transformation rules defined in the shared-inlining algorithm [16], we made several improvements over it:

- *Completeness.* Our rules consider all possible combinations of operators and XML elements whereas the shared-inlining algorithm only lists some important combinations. For example, there is no rule that corresponds to $(e_1 \mid \cdots \mid e_n)?$ in the shared-inlining algorithm.
- *Efficiency.* We enforce the application of the rules in the order given. Earlier rules totally transform away some operators from the input DTD, and in each step, the number of rules to be matched is greatly reduced. This improves the efficiency of the simplification procedure significantly.
- *Further simplification.* We observe that the role of "?" corresponds to the notion of nullable column in the relational table. We transform away "?" and this greatly simplifies the resulting DTD graph (to be described in the next subsection) since it does not contain "?" any more.

3.2 Creating and inlining DTD graphs

In this step, we create the corresponding DTD graph based on the simplified DTD, and then inline as many descendant elements to an element as possible. The rationale is that these inlined elements will eventually produce a relation. Therefore, we only inline a child c to a parent p when p can contain at most one occurrence of c in order to avoid introducing redundancy into the generated relation. Theorem 1 indicates that after the simplification procedure, any input DTD is now in a canonical form, i.e., each DTD expression is a tuple of distinct element names or their stars. As a result, in the corresponding DTD graph, each node represents an XML element, and each edge represents an operator of ',' or '*'. Our inlining procedure considers the following three cases.

1. **Case 1:** *Element* a *is connected to* b *by a ,-edge and* b *has no other incoming edges.* In other words, b is a non-shared node. In this case, a can contain at most one occurrence of b, and we will combine node b into a while maintaining the parent-child relationships between b and its children.
2. **Case 2:** *Element* a *is connected to* b *by a ,-edge but* b *has other incoming edges.* In other words, b is a shared node. We do not combine b into a in this case since b has multiple parents.
3. **Case 3:** *Element* a *is connected to* b *by a *-edge.* In this case, each a can contain multiple occurrences of b element, and we do not combine b into a.

Only case 1 allows us to inline an element to its parent. We define the notion of inlinable node as follows.

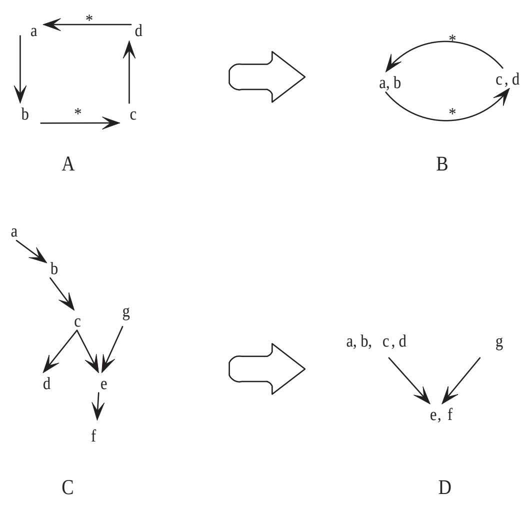

Fig. 2. Inlining DTD graphs

Definition 2. *Given a DTD graph, a node is inlinable if and only if it has exactly one incoming edge and that edge is a ,-edge.*

Definition 3. *Given a DTD graph and a node e in the graph, node e and all other inlinable nodes that are reachable from e by ,-edge constitute a tree (since we assume a DTD graph is consistent, thus there is no ,-edge cycle in the graph). This tree is called the inlinable tree for node e (it is rooted at e).*

Example 2. In Figure 2.A, nodes b and d are inlinable but nodes a and c are not inlinable. The inlinable tree for a contains nodes a and b, whereas the inlinable tree for c contains nodes c and d. In Figure 2.C, nodes b, c, d and f are inlinable, but nodes a, e and g are not inlinable. The inlinable tree for a contains nodes a, b, c and d, and the inlinable tree for node e contains nodes e and f.

The notion of inlinable tree formalizes the intuition of "inlining as many descendant elements as possible to an element". We illustrate our inlining algorithm in pseudocode in Figure 3. Essentially, it uses a depth-first-search strategy to identify the inlinable tree for each node and then inline that tree to its root. A field *inlinedSet* of set type is introduced for each node e to represent the set of XML element nodes that has been inlined to this node e (initially e.inlinedSet = {e}). For example, in Figure 2.C, after the inlining procedure, a.inlinedSet = {a, b, c, d}. The algorithm is efficient as indicated in the following theorem.

Theorem 2 (Complexity). *Our inlining algorithm can be performed in $O(n)$ where n is the number of elements in the input DTD.*

Proof. This is obvious since each node of the DTD graph is visited at most once.

```
Algorithm Inline(DTDGraph G)
Begin
    For each node e in G do
        If not visited(e) then
            InlineNode(e)
        End If
    End For
End

Algorithm InlineNode(Node e)
Begin
    Mark e as "visited"
    For each child c of e do
        If not visited(c) then
            InlineNode(c)
        End If
    End For
    For each child c of e do
        If inlinable(c) then
            e.inlinedSet $\cup =$ c.inlinedSet;
            assign all children of c as the children of e
            and then delete c from G
        End If
    End For
End
```

Fig. 3. The inlining procedure

Example 3. Using our inlining procedure given in Figure 3, the DTD graph shown in Figure 2.A will be inlined into one shown in Figure 2.B, and the DTD graph shown in Figure 2.C will be inlined into one shown in Figure 2.D.

We observe that after our inlining algorithm is applied, a DTD graph has the following property: nodes are connected by ,-edge or *-edge and ,-edge must point to a shared node. This observation is the basis of the final step of the algorithm: generating relational schemas.

3.3 Generating relational schemas

After a simplified DTD graph is inlined, the last step is to generate a relational schema based on this inlined DTD graph. The generated schema supports the select, insert, delete and update [18] of an arbitrary XML element declared in the input DTD. The following four steps will be performed on the inlined DTD graph to generate a set of relations.

1. For each node e, a relation e is generated with the following relational attributes.

   (a) ID is the primary key, and for each XML attribute A of e, a corresponding relational attribute A is generated with the same name.
   (b) If $\mid$ *e.inlinedSet* $\mid \geq 2$, we introduce attribute *nodetype* to indicate the type of the XML element stored in a tuple.
   (c) The names of all the terminal XML elements in *e.inlinedSet*. Since a non-terminal XML element is stored with values for ID and *nodetype* and the storage of the XML subelements it contains, no additional attribute is needed for it (this will become more clear later).
   (d) If there is a ,-edge from e to node c, then introduce $c.ID$ as a foreign key of e referencing relation c.
2. If there are at least two relations $t_1(ID)$ and $t_2(ID)$ generated by step 1, then we combine all the relations of the form $t(ID)$ into one single relation *table1(ID, nodetype)* where *nodetype* indicates which XML element is stored in a tuple.
3. If there are at least two relations $t_1(ID, t_1)$ and $t_2(ID, t_2)$ generated by step 1, then we combine all the relations of the form $t(ID, t)$ into one single relation *table2(ID, nodetype, pcdata)* where *nodetype* indicates which XML element is stored in a tuple.
4. If there is at least one $*$ edge in the inlined DTD graph, then we introduce relation edge($\underline{\text{parentID}}$, $\underline{\text{childID}}$, parentType, childType) to store all the parent-child relationships corresponding to *-edges. The domains of *parentType* and *childType* are the set of XML element names defined in the input DTD.

Essentially, step 1 converts each node e in the inlined DTD graph into a separate relation e. If there are some other XML element nodes that have been inlined to it (i.e., $\mid$ *e.inlinedSet* $\mid \geq 2$), relation e will be used to store all these XML elements, and attribute *nodetype* will be introduced to indicate which XML element is the root for each tuple. Since step 1 might produce a set of relations in the forms of $t(ID)$ and $t(ID, t)$, Step 2 and 3 optimize them by performing a horizontal combining of them into *table1(ID, nodetype)* and *table2(ID, nodetype, pcdata)*. These optimizations reduce the number of target relations and will facilitate the mapping from XML operations to relational SQL operations. Finally, one single relation *edge($\underline{\text{parentID, childID}}$, parentType, childType)* stores all the many-to-many relationships between arbitrary two XML elements.

Although our inlining algorithm is inspired by the shared-inlining algorithm, we made several significant improvements over it:

– *Completeness.* Our algorithm is complete in the sense that it can deal with any input DTDs including arbitrary cyclic DTDs. The shared-inlining algorithm defines a rule to deal with two mutually recursive elements and it is not clear how a DTD with a cycle involving more than two elements is handled (see Figure 4.A for such an example). In addition, the shared-inlining algorithm checks the existence of recursion explicitly, we do not need to do this checking and cycles are dealt with naturally.

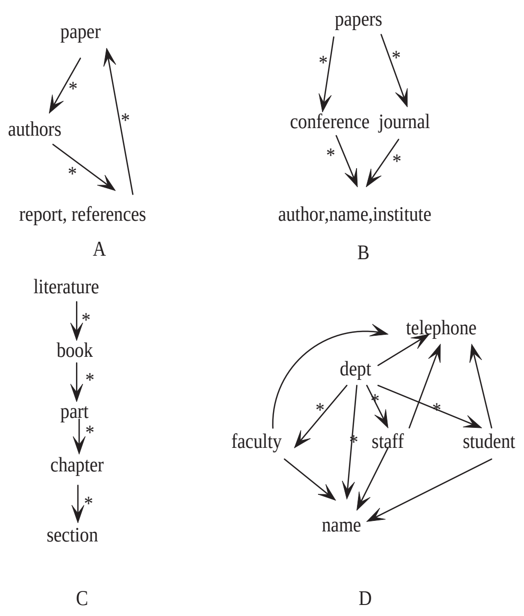

Fig. 4. Four inlined DTD graphs

- *Redundancy elimination for shared nodes.* A node is shared if its in-degree is more than one. Our algorithm deals with shared nodes differently from the shared-inlining algorithm. For example, for the shared node *author* in Figure 4.B, the shared-inlining algorithm will generate a separate relation *author(authorID, author.parentID, author.parentCODE, author.name.isroot, author.name, author.institute.isroot, author.institute).* This schema implies a great deal of redundancy if an author writes hundreds of conference or journal papers, In contrast, we create a relation *author(ID, nodetype, name, institute)* for *author*, and translate its parent $*$-edges (and all other $*$-edges) into another separate relation *edge(parentID, childID, parentType, childType).* Our strategy eliminates the above redundancy and bears the same spirit as the rule of mapping many-to-many relationships into separate relations in translating Entity-Relationship (ER) diagrams into relational schemas.
- *Optimizations.* Two situations are very common in XML documents: (1) there are XML elements which do not have any attributes and their single purpose is to provide a tag name (e.g., Figure 4.C) for supporting nested structure; and (2) there are terminal nodes that are shared by several XML elements (such as *name* and *telephone* in Figure 4.D). If we created a separate relation for each such kind of element, then we would produce a set of relations of the form of $t(ID)$ (case 1) or $t(ID, t)$ (case 2). Hence, instead, we create two relations *table1(ID, nodetype)* and *table2(ID, nodetype, pcdata)* which conceptually combine all these relations. These optimizations greatly

reduce the number of relations in the generated schema and facilitates the translation of XML queries into relational queries.

– *Efficiency*: The shared-inlining algorithm introduces an attribute *parentID* for each node under the $*$ operator while the $*$ operator itself is never translated into a separate relation. This facilitates the traversal of XML documents upwards (from children to parents) but not downwards (from parents to children). For example, in Figure 4.D, the shared-inlining algorithm will generate relations *dept*, *faculty*, *staff*, etc. Given a faculty, it is very easy to locate which department he is from based on an index on *facultyID* and *faculty.parentID* of relation *faculty*. However, it would be difficult to navigate downwards for path expressions such as *dept//name* (get all the names reachable from element *dept*), since one needs to consider the fact that *dept* actually has three kinds of children (faculty, staff, and student), and all these three ways of reaching a *name* have to be combined. In contrast, We will translate all $*$-edges into one single relation *edge(parentID, childID, parentType, childType)*, and create two indices on *parentID* and *childID* respectively. In this way, both upward navigation and downward navigation are supported efficiently.

```
<!DOCTYPE publication [
    <!ELEMENT publication (journal*, conference*)>
    <!ELEMENT journal (name, editors, paper+)>
    <!ELEMENT conference (name, paper+)>
    <!ELEMENT paper (ptitle, authors,
                        (volume, number)?)>
    <!ATTLIST paper year CDATA>
    <!ELEMENT editors (person+)>
    <!ELEMENT authors (person+)>
    <!ELEMENT person (pname, institute, techreport*)>
    <!ELEMENT techreport (title, references)>
    <!ELEMENT references (paper+)>
    <!ELEMENT institute (#PCDATA)>
    <!ELEMENT pname (#PCDATA)>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT ptitle (#PCDATA)>
    <!ELEMENT title (#PCDATA)>
    <!ELEMENT volume (#PCDATA)>
    <!ELEMENT number (#PCDATA)>
]>
```

Fig. 5. A publication DTD

3.4 A complete example

In this section, we illustrate different steps of our algorithm with a real DTD example, and demonstrate how XML documents conforming to this DTD can be stored based on the generated schema.

An XML DTD for publications is shown in Figure 5. After the simplification step (using the rules defined in Figure 1), the input DTD is simplified into one with the following new XML element definitions. The definitions for other XML elements remain the same.

– <!ELEMENT journal (name, editors, paper*)>.
– <!ELEMENT conference (name, paper*)>.
– <!ELEMENT paper (ptitle,authors,volume,number)>.
– <!ELEMENT editors (person*)>.
– <!ELEMENT authors (person*)>.
– <!ELEMENT references (paper*)>.

Due to space limit, we omit the DTD graph for the simplified DTD and the inlined DTD graph and leave them as an exercise for the reader. Finally, the following eight relations will be generated.

– publication(<u>ID</u>) stores XML element *publication*.
– conference(<u>ID</u>, name.ID) stores XML element *conference*.
– journal(<u>ID</u>, nodetype, name.ID) stores XML elements *journal* and *editors*.
– name(<u>ID</u>, PCDATA) stores XML element *name*.
– paper(<u>ID</u>, nodetype, ptitle, volume, number, year) stores XML elements *ptitle*, *authors*, *volume*, *number* and *year*.
– person(<u>ID</u>, nodetype, pname, institute) stores XML elements *person*, *pname* and *institute*.
– techreport(<u>ID</u>, nodetype, title) stores XML elements *techreport*, *title* and *references*.
– edge(<u>parentID, childID,</u> parentType, childType) stores all the parent-child relationships between two XML elements.

4 Conclusions and future work

We have developed a new inlining algorithm that maps a given input DTD to a relational schema. Our algorithm is inspired by the shared-inlining algorithm but features several improvement over it including overcoming its incompleteness, eliminating redundancies caused by shared elements, performing optimizations and enhancing efficiency. Future work includes a full evaluation of the performance of our approach versus other approaches and adapting our algorithm to one that maps XML Schemas [17] (an extension to DTDs) to relational schemas. Based on this schema mapping scheme, the mappings from XML data to relational data, and from XML queries to relational queries need to be investigated.

References

1. *eXtensible Information Server (XIS).* eXcelon Corporation. `http://www.exln.com`.
2. *Tamino XML Server.* Software AG. `http://www.softwareag.com/tamino`.
3. T. Bray, J. Paoli, C. Sperberg-McQueen, and E. Maler. *Extensible Markup Language (XML) 1.0*, October 2000. `http://www.w3.org/TR/REC-xml`.
4. M. J. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Publishing object-relational data as XML. In *WebDB (Informal Proceedings)*, pages 105–110, 2000.
5. S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: a graphical language for querying and restructuring WWW data. In *International World Wide Web Conference (WWW)*, Toronto, Canada, May 1999.
6. D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanascu. *XQuery: A Query Language for XML*, February 2001. `http://www.w3.org/TR/xquery`.
7. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. *XML-QL: A Query Language for XML*, August 1998. `http://www.w3.org/TR/NOTE-xml-ql/`.
8. A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 431–442, Philadephia, Pennsylvania, June 1999.
9. M. Fernndez, W. Tan, and D. Suciu. SilkRoute: Trading between relations and XML. In *Proc. of the Ninth International World Wide Web Conference*, 2000.
10. D. Florescu and D. Kossman. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), 1999.
11. D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing xml data in a relational database. In *Proc. of the VLDB*, 1999.
12. R. Goldman, J. McHugh, and J. Widom. *From Semistructured Data to XML: Migrating the Lore Data Model and Query Languages*, 1999.
13. A. Kurt and M. Atay. An experimental study on query processing efficiency of native-XML and XML-enabled relational database systems. In *Proc. of the 2nd International Workshop on Databases in Networked Information Systems (DNIS'2003), Lecture Notes in Computer Science, Volume 2544*, pages 268–284, Aizu-Wakamatsu, Japan, December 2002.
14. J. Robie, J. Lapp, and D. Schach. *XML Query Language (XQL)*, 1998. `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.
15. J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *VLDB Journal*, 10(2–3):133–154, 2001.
16. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. *The VLDB Journal*, pages 302–314, 1999.
17. C. Sperberg-MCQueen and H. Thompson. *W3C XML Schema*, April 2000. `http://www.w3.org/XML/Schema`.
18. I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In *Proc. of ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA, 2001.
19. F. Tian, D. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative XML storage strategies. *ACM Sigmod Record*, 31(1), March 2002.