Relational Databases for Querying XML Documents: Limitations and Opportunities

Jayavel Shanmugasundaram Kristin Tufte Gang He
Chun Zhang David DeWitt Jeffrey Naughton

Department of Computer Sciences
University of Wisconsin-Madison
{jai, tufte, czhang, dewitt, naughton}@cs.wisc.edu, ganghe@microsoft.com

Abstract

XML is fast emerging as the dominant standard for representing data in the World Wide Web. Sophisticated query engines that allow users to effectively tap the data stored in XML documents will be crucial to exploiting the full power of XML. While there has been a great deal of activity recently proposing new semi-structured data models and query languages for this purpose, this paper explores the more conservative approach of using traditional relational database engines for processing XML documents conforming to Document Type Descriptors (DTDs). To this end, we have developed algorithms and implemented a prototype system that converts XML documents to relational tuples, translates semi-structured queries over XML documents to SQL queries over tables, and converts the results to XML. We have qualitatively evaluated this approach using several real DTDs drawn from diverse domains. It turns out that the relational approach can handle most (but not all) of the semantics of semi-structured queries over XML data, but is likely to be effective only in some cases. We identify the causes for these limitations and propose certain extensions to the relational model that would make it more appropriate for processing queries over XML documents.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

1. Introduction

Extensible Markup Language (XML) is fast emerging as the dominant standard for representing data on the Internet. Like HTML, XML is a subset of SGML. However, whereas HTML tags serve the primary purpose of describing how to display a data item, XML tags describe the data itself. The importance of this simple distinction cannot be underestimated – because XML data is self-describing, it is possible for programs to interpret the data. This means that a program receiving an XML document can interpret it in multiple ways, can filter the document based upon its content, can restructure it to suit the application's needs, and so forth.

The initial impetus for XML may have been primarily to enhance this ability of remote applications to interpret and operate on documents fetched over the Internet. However, from a database point of view, XML raises a different exciting possibility: with data stored in XML documents, it should be possible to query the contents of these documents. One should be able to issue queries over sets of XML documents to extract, synthesize, and analyze their contents. But what is the best way to provide this query capability over XML documents?

At first glance the answer is obvious. Since an XML document is an example of a semi-structured data set (it is tree-structured, with each node in the tree described by a label), why not use semi-structured query languages and query evaluation techniques? This is indeed a viable approach, and there is considerable activity in the semi-structured data community focussed upon exploiting this approach [5,14]. While semi-structured techniques will clearly work, in this paper we ask the question of whether this is the only or the best approach to take. The downside of using semi-structured techniques is that this approach turns its back on 20 years of work invested in relational database technology. Is it really the case that we cannot use relational technology, and must start afresh with new techniques? Or can we leverage relational technology to provide query capability over XML documents?

In this paper we demonstrate that it is indeed possible to use standard commercial relational database systems to evaluate powerful queries over XML documents. The key that makes this possible is the existence of Document Type Descriptors (DTDs) [2] (or an equivalent, such as DCDs [4] or XML Schemas [16]). A DTD is in effect a schema for a set of XML documents. Without DTDs or their equivalent, XML will never reach its full potential, because a tagged document is not very useful without some agreement among inter-operating applications as to what the tags mean. Put another way, the reason the Internet community is so excited about XML is that there is the vision of a future in which the vast majority of files on the web are XML files conforming to DTDs. An application encountering such a file can interpret the file by consulting the DTDs to which the document conforms.

Our approach to querying XML documents is the following. First, we process a DTD to generate a relational schema. Second, we parse XML documents conforming to DTDs and load them into tuples of relational tables in a standard commercial DBMS (in our case, IBM DB2). Third, we translate semi-structured queries (specified in a language similar to XML-QL [9] or Lorel [1]) over XML documents into SQL queries over the corresponding relational data. Finally, we convert the results back to XML.

The good news is that this works. A main contribution of this paper is the description of an approach that enables one to take the XML queries, data sets, and schemas so foreign to the relational world and process them in relational systems without any manual intervention. This means that we are presented with a large opportunity: all of the power of relational database systems can be brought to bear upon the XML query problem.

However, the fact that something is possible does not necessarily imply that it is a good idea. Our experience with implementing this system and using it with over 30 different XML DTDs has revealed that there are a number of limitations in current relational database systems that in some instances make using relational technology for XML queries either awkward or inefficient. Relational technology proves awkward for queries that require complex XML constructs in their results, and may be inefficient when fragmentation due to the handling of set-valued attributes and sharing causes too many joins in the evaluation of simple queries. Another contribution of this paper is the identification of those limitations, and a discussion of how they might be removed. It is an open question at this point whether the best approach is to start with relational technology and try to remove those limitations, or to start with a semi-structured system and try to add the power and sophistication currently found in relational query processing systems.

1.1 Related Work

There has been a lot of work developing special purpose query engines for semi-structured data [5,14]. Many of the abstracts submitted to the XML query languages workshop use this approach [18]. Our goal in this paper, however, is to investigate the use of relational database systems to process queries on semi-structured documents. In this sense, our work is similar to the work on STORED [10]. However, our approach differs in important ways. The STORED approach uses a combination of relational and semi-structured techniques to process any semi-structured documents. We begin with the assumption that the document conforms to a schema and store the document entirely within the relational system. Further, we handle recursive queries, address the issue of constructing the result in XML and evaluate the relational approach using real DTDs.

Oracle 8i provides some basic support for querying XML documents using a relational engine [17]. However, the translation from document schemas to relational schemas is manual and not automatic as in our approach. In addition, Oracle 8i does not provide support for semi-structured queries over XML documents and provides only primitive support for converting results to XML.

There has also been work on processing SGML data using an OODBMS [6]. The conclusion was that this is feasible with some extensions to OO query languages. Our work considers a more restricted set of documents (XML, rather than SGML) and considers mapping to the relational model, rather than a general OO model.

Our method of eliminating wild cards and alternations in path expression queries to enable processing by a relational engine bears some similarities to the work on compile time optimization of path expressions in semi-structured query engines [12,15]. Our different focus, however, results in modified techniques.

1.2 Roadmap

The rest of the paper is organized as follows. Section 2 gives an overview of XML documents, schemas and query languages. The algorithms for translating DTDs and XML documents to a relational format and an evaluation of the algorithms using real DTDs are given in Section 3. Section 4 describes the translation of queries over XML documents to SQL queries. Section 5 deals with the conversion of the results to XML. Section 6 concludes by proposing extensions to the relational model that will make it more suitable for processing XML documents.

2. Overview of XML, XML Schemas and XML Query Languages

In this section, we give a very brief overview of XML, XML schemas and XML query languages. Further details can be obtained from the references.

2.1 Extensible Markup Language

Extensible Markup Language (XML) is a hierarchical data format for information exchange in the World Wide Web. An XML document consists of nested element structures, starting with a root element. Element data can be in the form of attributes or sub-elements. Figure 1 shows an XML document that contains information about a book. In this example, there is a book element that has two sub-elements, booktitle and author. The author element has an id attribute with value “dawkins” and is further nested to provide name and address information. Further information on XML can be found in [3,8].

```
<book>
   <booktitle> The Selfish Gene </booktitle>
   <author id = “dawkins”>
            <name>
                <firstname> Richard </firstname>
                <lastname> Dawkins </lastname>
            </name>
            <address>
                <city> Timbuktu </city>
                <zip> 99999 </zip>
            </address>
   </author>
</book>
```

Figure 1

```
<!ELEMENT book (booktitle, author)
<!ELEMENT article (title, author*, contactauthor)>
<!ELEMENT contactauthor EMPTY>
<!ATTLIST contactauthor authorID IDREF IMPLIED>
<!ELEMENT monograph (title, author, editor)>
<!ELEMENT editor (monograph*)>
<!ATTLIST editor name CDATA #REQUIRED>
<!ELEMENT author (name, address)>
<!ATTLIST author id ID #REQUIRED>
<!ELEMENT name (firstname?, lastname)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT address ANY>
```

Figure 2

2.2 DTDs and other XML Schemas

Document Type Descriptors (DTDs) [2] describe the structure of XML documents and are like a schema for XML documents. A DTD specifies the structure of an XML element by specifying the names of its sub-elements and attributes. Sub-element structure is specified using the operators * (set with zero or more elements), + (set with one or more elements), ? (optional), and | (or). All values are assumed to be string values, unless the type is ANY in which case the value can be an arbitrary XML fragment. There is a special attribute, id, which can occur once for each element. The id attribute uniquely identifies an element within a document and can be referenced through an IDREF field in another element. IDREFs are untyped. Finally, there is no concept of a root of a DTD – an XML document conforming to a DTD can be rooted at any element specified in the DTD. Figure 2 shows a DTD specification, while Figure 1 gives an XML document that conforms to this DTD.

Document Content Descriptors (DCDs) [4] and XML Schemas [16] are extensions to DTDs. For our purposes, the main difference between these and DTDs is that they allow typing of values and set size specification. If DCDs and XML Schemas become standard, the additional information would aid in our translation process; for example, we could create tables with integer attributes where appropriate instead of using just strings. The types in the current DCD proposal are compatible with types supported by current relational systems. More complex types will require object-relational extensions.

2.3 XML Query Languages

```
SELECT X.author.lastname
FROM book X
WHERE X.booktitle = “The Selfish Gene”
```

Figure 3

```
WHERE <book>
                <booktitle> The Selfish Gene </booktitle>
                <author>
                      <lastname> $l </lastname>
                </>
            </> IN a.xml, b.xml
CONSTRUCT <lastname> $l </lastname>
```

Figure 4

There are many semi-structured query languages that can be used to query XML documents, including XML-QL [9], Lorel [1], UnQL [5] and XQL (from Microsoft). All these query languages have a notion of path expressions for navigating the nested structure of XML. XML-QL uses a nested XML-like structure to specify the part of a document to be selected and the structure of the result XML document.

Figure 4 shows an XML-QL query to determine the last name of an author of a book having title “The Selfish Gene”, specified over a set of XML documents conforming to the DTD in Figure 2. The last names thus selected will be nested within a lastname tag, as specified in the construct clause of the query. Lorel is more like SQL and its representation of the same query is shown in Figure 3. In this paper, we use a combination of XML-QL and Lorel (modified appropriately for our purposes).

3. Storing XML Documents in a Relational Database System

In this section, we describe how to generate relational schemas from XML DTDs. The main issues that must be addressed include (a) dealing with the complexity of DTD element specifications (b) resolving the conflict between the two-level nature of relational schemas (table and attribute) vs. the arbitrary nesting of XML DTD schemas and (c) dealing with set-valued attributes and recursion.

3.1 Simplifying DTDs

In general, DTDs can be complex and generating relational schemas that capture this complexity would be unwieldy at best. Fortunately, one can simplify the details of a DTD and still generate a relational schema that can store and query documents conforming to that DTD. Note that it is not necessary to be able to regenerate a DTD from the generated relational schema. Rather, what is required is that (a) any document conforming to the DTD can be stored in the relational schema, and (b) any XML semi-structured query over a document conforming to the DTD can be evaluated over the relational database instance.

Most of the complexity of DTDs stems from the complex specification of the type of an element. For instance, we could specify an element a as <!ELEMENT a ((b|c|e)?,(e?|(f?,(b,b)*))*)>, where b, c, e and f are other elements. However, at the query language level, all that matters is the position of an element in the XML document, relative to its siblings and the parent-child relationship between elements in the XML document. We now propose a set of transformations that can be used to "simplify" any arbitrary DTD without undermining the effectiveness of queries over documents conforming to that DTD. These transformations are a superset of similar transformations presented in [10].

$$
\begin{aligned}
&(e_1, e_2)^* \rightarrow e_1^*, e_2^* \\
&(e_1, e_2)? \rightarrow e_1?, e_2? \\
&(e_1|e_2) \rightarrow e_1?, e_2?
\end{aligned}
$$

Figure 5

$$
\begin{aligned}
&e_1^{**} \rightarrow e_1^* \\
&e_1^*? \rightarrow e_1^* \\
&e_1?^* \rightarrow e_1^* \\
&e_1?? \rightarrow e_1?
\end{aligned}
$$

Figure 6

$$
\begin{aligned}
&..., a^*, ..., a^*, ... \rightarrow a^*, ... \\
&..., a^*, ..., a?, ... \rightarrow a^*, ... \\
&..., a?, ..., a^*, ... \rightarrow a^*, ... \\
&..., a?, ..., a?, ... \rightarrow a^*, ... \\
&..., a, ..., a, ... \rightarrow a^*, ...
\end{aligned}
$$

Figure 7

The transformations are of three types: (a) flattening transformations which convert a nested definition into a flat representation (i.e., one in which the binary operators "," and "|" do not appear inside any operator – see Figure 5) (b) simplification transformations, which reduce many unary operators to a single unary operator (Figure 6) and (c) grouping transformations that group sub-elements having the same name (for example, two a* sub-elements are grouped into one a* sub-element - see Figure 7). In addition, all "+" operators are transformed to "*" operators. Our example specification would be transformed to: <!ELEMENT a (b*, c?, e*, f*)>.

The transformations preserve the semantics of (a) one or many and (b) null or not null. The astute reader may notice that we have lost some information about relative orders of the elements. This is true; fortunately, this information can be captured when a specific XML document is loaded into this relational schema (e.g., by position fields in the tuples representing some of the elements.) We now explore techniques for converting a simplified DTD to a relational schema.

3.2 Motivation for Special Schema Conversion Techniques

Traditionally, relational schemas have been derived from a data model such as the Entity-Relationship model. This translation is straightforward because there is a clear separation between entities and their attributes. Each entity and its attributes are mapped to a relation.

When converting an XML DTD to relations, it is tempting to map each element in the DTD to a relation and map the attributes of the element to attributes of the relation. However, there is no correspondence between elements and attributes of DTDs and entities and attributes of the ER-Model. What would be considered "attributes" in an ER-Model are often most naturally represented as elements in a DTD. Figure 2 shows a DTD that illustrates this point. In an ER-Model, *author* would be an "entity" and *firstname*, *lastname* and *address* would be attributes of that entity. In designing a DTD, there is no incentive to make *author* an element and *firstname*, *lastname* and *address* attributes. In fact, in XML, if *firstname* and *lastname* were attributes, they could not be nested under name because XML attributes cannot have a nested structure. Directly mapping elements to relations is thus likely to lead to excessive fragmentation of the document.

3.3 The Basic Inlining Technique

The Basic Inlining Technique, hereafter referred to as *Basic*, solves the fragmentation problem by inlining as many descendants of an element as possible into a single relation. However, *Basic* creates relations for every element because an XML document can be rooted at any element in a DTD. For example, the *author* element in Figure 2 would be mapped to a relation with attributes *firstname*, *lastname* and *address.* In addition, relations would be created for *firstname, lastname* and *address*.

We must address two complications: set-valued attributes and recursion. In the example DTD in Figure 2, when creating a relation for article, we cannot inline the set of authors because the traditional relational model does not support set-valued attributes. Rather, we follow the standard technique for storing sets in an RDBMS and create a relation for author and link authors to articles using a foreign key. Just using inlining (if we want the process to terminate) necessarily limits the level of nesting in the recursion. Therefore, we express the recursive relationship using the notion of relational keys and use relational recursive processing to retrieve the relationship. In order to do this in a general fashion, we introduce the notion of a DTD graph.

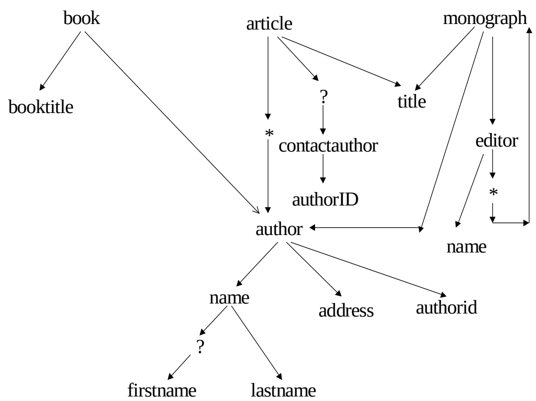

Figure 8

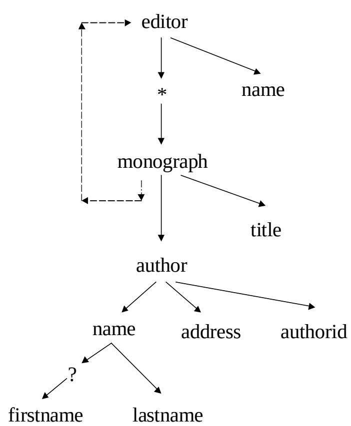

Figure 9

A DTD graph represents the structure of a DTD. Its nodes are elements, attributes and operators in the DTD. Each element appears exactly once in the graph, while attributes and operators appear as many times as they appear in the DTD. The DTD graph corresponding to the DTD in Figure 2 is given in Figure 8. Cycles in the DTD graph indicate the presence of recursion.

The schema created for a DTD is the union of the sets of relations created for each element. In order to determine the set of relations to be created for a particular element, we create a graph structure called the *element graph*. The element graph is constructed as follows.

Do a depth first traversal of the DTD graph, starting at the element node for which we are constructing relations. Each node is marked as “visited” the first time it is reached and is unmarked it once all its children have been traversed.

If an unmarked node in the DTD graph is reached during depth first traversal, a new node bearing the same name is created in the element graph. In addition, a *regular* edge is created from the most recently created node in the element graph with the same name as the DFS parent of the current DTD node to the newly created node.

If an attempt is made to traverse an already marked DTD node, then a *backpointer* edge is added from the most recently created node in the element graph to the most recently created node in the element graph with the same name as the marked DTD node.

The element graph for the *editor* element in the DTD graph in Figure 8 is shown in Figure 9. Intuitively, the element graph expands the relevant part of the DTD graph into a tree structure.

Given an element graph, relations are created as follows. A relation is created for the root element of the graph. All the element’s descendents are inlined into that relation with the following two exceptions: (a) children directly below a “*” node are made into separate relations – this corresponds to creating a new relation for a set-valued child; and (b) each node having a backpointer edge pointing to it is made into a separate relation – this corresponds to creating a new relation to handle recursion. Figure 10 shows the relational schema that would be generated for the DTD in Figure 2. There are several features to note in the schema. Attributes in the relations are named by the path from the root element of the relation. Each relation has an ID field that serves as the key of that relation. All relations corresponding to element nodes having a parent also have a parentID field that serves as a foreign key. For instance, the *article.author* relation has a foreign key *article.author.parentID* that joins authors with articles. The XML document in Figure 1 would be converted to the following tuple in the book relation:

(1, The Selfish Gene, Richard, Dawkins,
<city>Timbuktu</city><zip>99999</zip>, dawkins)

The ANY field, address, is stored as an uninterpreted string; thus the nested structure is not visible to the database system without further support for XML (see Section 6). Note that if the author Richard Dawkins has authored many books, then the author information will be replicated for each book because it is replicated in the corresponding XML documents.

While *Basic* is good for certain types of queries, such as “list all authors of books”, it is likely to be grossly inefficient for other queries. For example, queries such as “list all authors having first name Jack” will have to be executed as the union of 5 separate queries. Another disadvantage of *Basic* is the large number of relations it creates. Our next technique attempts to resolve these problems.

book (bookID: integer, book.booktitle : string, book.author.name.firstname: string, book.author.name.lastname: string, book.author.address: string, author.authorid: string)

booktitle (booktitleID: integer, booktitle: string)

article (articleID: integer, article.contactauthor.authorid: string, article.title: string)

article.author (article.authorID: integer, article.author.parentID: integer, article.author.name.firstname: string, article.author.name.lastname: string, article.author.address: string, article.author.authorid: string)

contactauthor (contactauthorID: integer, contactauthor.authorid: string)

title (titleID: integer, title: string)

monograph (monographID: integer, monograph.parentID: integer, monograph.title: string, monograph.editor.name: string, monograph.author.name.firstname: string, monograph.author.name.lastname: string, monograph.author.address: string, monograph.author.authorid: string)

editor (editorID: integer, editor.parentID: integer, editor.name: string)

editor.monograph (editor.monographID: integer, editor.monograph.parentID: integer, editor.monograph.title: string, editor.monograph.author.name.firstname: string, editor.monograph.author.name.lastname: string, editor.monograph.author.address: string, editor.monograph.author.authorid: string)

author (authorID: integer, author.name.firstname: string, author.name.lastname: string, author.address: string, author.authorid: string)

name (nameID: integer, name.firstname: string, name.lastname: string)

firstname (firstnameID: integer, firstname: string)

lastname (lastnameID: integer, lastname: string)

address (addressID: integer, address: string)

Figure 10

book (bookID: integer, book.booktitle.isroot: boolean, book.booktitle : string)

article (articleID: integer, article.contactauthor.isroot: boolean, article.contactauthor.authorid: string)

monograph (monographID: integer,monograph.parentID: integer, monograph.parentCODE: integer, monograph.editor.isroot: boolean, monograph.editor.name: string)

title (titleID: integer, title.parentID: integer, title.parentCODE: integer, title: string)

author (authorID: integer, author.parentID: integer, author.parentCODE: integer, author.name.isroot: boolean, author.name.firstname.isroot: :boolean, author.name.firstname: string, author.name.lastname.isroot: boolean, author.name.lastname: string, author.address.isroot: boolean, author.address: string, author.authorid: string)

Figure 11

3.4 The Shared Inlining Technique

The Shared Inlining Technique, hereafter referred to as *Shared*, attempts to avoid the drawbacks of *Basic* by ensuring that an element node is represented in exactly one relation. The principal idea behind *Shared* is to identify the element nodes that are represented in multiple relations in *Basic* (such as the *firstname*, *lastname* and *address* elements in the example) and to share them by creating separate relations for these elements.

We must first decide what relations to create. In *Shared*, relations are created for all elements in the DTD graph whose nodes have an in-degree greater than one. These are precisely the nodes that are represented as multiple relations in *Basic*. Nodes with an in-degree of one are inlined. Element nodes having an in-degree of zero are also made separate relations, because they are not reachable from any other node. As in *Basic*, elements below a "*" node are made into separate relations. Finally, of the mutually recursive elements all having in-degree one (such as *monograph* and *editor* in Figure 8), one of them is made a separate relation. We can find such mutually recursive elements by looking for strongly connected components in the DTD graph.

Once we decide which element nodes are to be made into separate relations, it is relatively easy to construct the relational schema. Each element node X that is a separate relation inlines all the nodes Y that are reachable from it such that the path from X to Y does not contain a node (other than X) that is to be made a separate relation. Figure 11 shows the schema derived from the DTD graph of Figure 8. One striking feature is the small number of relations compared to the *Basic* schema (Figure 10).

Inlining an element X into a relation corresponding to another element Y creates problems when an XML document is rooted at the element X. To facilitate queries on such elements we make use of isRoot fields.

The element sharing in *Shared* has query processing implications. For example, a selection query over all authors accesses only one relation in *Shared* compared to five relations in *Basic*. Despite the fact that *Shared* addresses some of the shortcomings and shares some of

book (bookID: integer, book.booktitle.isroot: boolean, book.booktitle : string, author.name.firstname: string, author.name.lastname: string, author.address: string, author.authorid: string)

article (articleID: integer, article.contactauthor.isroot: boolean, article.contactauthor.authorid: string, article.title.isroot: boolean, article.title: string)

monograph (monographID: integer, monograph.parentID: integer, monograph.parentCODE: integer, monograph.title: string, monograph.editor.isroot: boolean, monograph.editor.name: string, author.name.firstname: string, author.name.lastname: string, author.address: string, author.authorid: string)

author (authorID: integer, author.parentID: integer, author.parentCODE: integer, author.name.isroot: boolean, author.name.firstname.isroot: boolean, author.name.firstname: string, author.name.lastname.isroot: boolean, author.name.lastname: string, author.address.isroot: boolean, author.address: string, author.authorid: string)

Figure 12

the strengths of *Basic*, *Basic* performs better in one important respect – reducing the number of joins starting at a particular element node. Thus we explore a hybrid approach that combines the join reduction properties of *Basic* with the sharing features of *Shared*

3.5 The Hybrid Inlining Technique

The Hybrid Inlining Technique, or *Hybrid*, is the same as *Shared* except that it inlines some elements that are not inlined in *Shared*. In particular, *Hybrid* additionally inlines elements with in-degree greater than one that are not recursive or reached through a "*" node. Set sub-elements and recursive elements are treated as in *Shared*. Figure 12 shows the relational schema generated using this hybrid approach. Note how this schema combines features of both *Basic* and *Shared* – *author* is inlined with *book* and *monograph* even though it is shared, while *monograph* and *editor* are represented exactly once.

So far, we have implicitly assumed that the data model is unordered, i.e., the position of an element does not matter. Order could, however, be easily incorporated into our framework by storing a position field for each element.

3.6 A Qualitative Evaluation of the Basic, Shared and Hybrid Techniques

In this section we qualitatively evaluate our relation-conversion algorithms using 37 DTDs available from Robin Cover's SGML/XML Web page [8]. We did not pose any criterion for selecting DTDs except for availability for easy download and validity. Some DTDs were excluded because they did not pass our XML parser, the IBM alphaWorks xml4j.

3.6.1 Evaluation Metric

Our major concern in evaluating the algorithms is the efficiency of query processing. Our metric is *the average number of SQL joins required to process path expressions of a certain length N.* We use this metric because path expressions are at the heart of query languages proposed for semi-structured data. We are particularly concerned about path expressions because we use a relational database which uses joins to process path expressions.

This subsection logically contains "forward references" to Section 4, in which we describe how SQL queries are generated from semi-structured XML queries. However, the only point from Section 4 that is necessary to understand the results here is that a single semi-structured query could give rise to a union of several SQL queries, and that each of these queries may contain some number of joins. The use of *Basic* vs. *Shared* vs. *Hybrid* determines how many queries are generated, and how many joins are found in each query. Although *Basic* and *Hybrid* reduce the number of joins *per SQL query*, their higher degree of inlining could cause more SQL queries to be generated. For each algorithm, each DTD, and a variable number of path lengths, we make the following measurements:

- The average number of SQL queries generated for path expressions of length N.
- The average number of joins in each SQL query for path expressions of length N.
- The total average number of joins in order to process path expressions of length N (the product of the two previous measurements.)

In Sections 3.6.2 and 3.6.3, we assume that path expressions start from an arbitrary element in the DTD. We relax this assumption in Section 3.6.4.

3.6.2 Evaluation Results for Expression Paths of Length 3

In this section we show the results for path expressions of length 3, which is the longest path length applicable to all 37 DTDs. We shall examine the results for other path lengths in the next section. In the interest of space, we show the results only for a subset of the DTDs and summarize the others.

First we consider whether the *Basic* approach is practical. For 11 of our 37 DTDs, *Basic* did not run to completion because it ran out of virtual memory. The reason for this is that *Basic* generates huge numbers of relations if DTDs have large strongly connected components. We can see this effect clearly on some of the DTDs that *Basic* did run to completion. One 19 node

DTD has a SCC size of 4, and the number of relations created is 204 times as many as created by *Hybrid*, totalling 3462 relations. Due to this severe limitation of *Basic*, we concentrate on the comparisons between *Shared* and *Hybrid*.

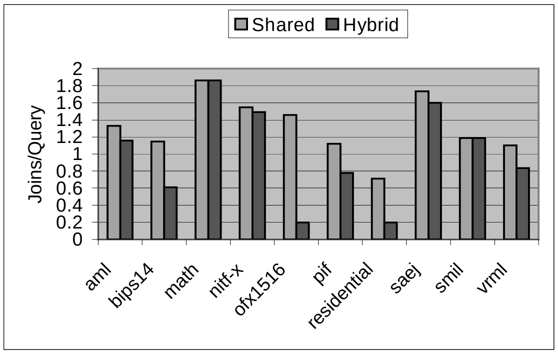

Figure 13

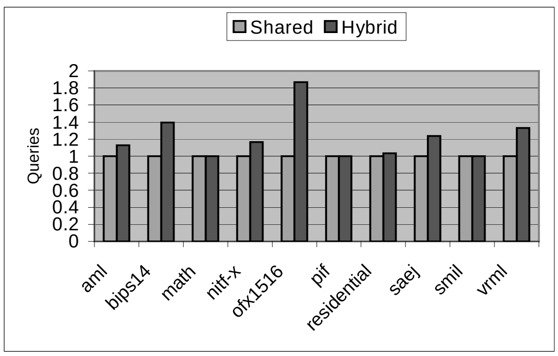

Figure 14

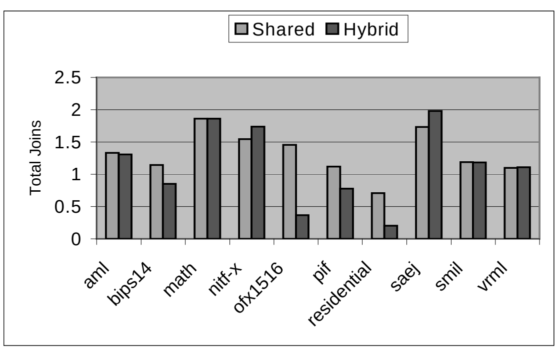

Figure 15

Figures 13, 14 and 15 show results for 10 of the DTDs. As shown in Figure 13, *Hybrid* eliminates a large number of joins for some DTDs, whereas for others, *Hybrid* and *Shared* produce about the same number of joins. Figure 14 shows that for some DTDs, querying over 3-length path expressions using *Hybrid* requires more SQL queries than using *Shared*, while for other DTDs, the number of SQL queries is the same. Note that for any path expression, *Shared* always produces at least the number of joins per SQL query as *Hybrid*, and *Hybrid* always produces at least the number of SQL queries as *Shared*. Figure 15 shows the total number of joins.

Using the average total number of joins required to process path expressions of length 3, we can roughly categorize the 37 DTDs into four groups:

Group 1. DTDs for which *Hybrid* reduces a large percentage of joins per SQL query but incurs a smaller increase in the number of SQL queries. The net result is *Hybrid* requires fewer joins than *Shared*. Example: DTD “ofx1516”.

Group 2. DTDs for which *Hybrid* reduces a large percentage of joins per SQL query and incurs a comparable increase in the number of SQL queries. The total number of joins is about the same. Example: DTD “vrml”.

Group 3. DTDs for which *Hybrid* reduces some joins per SQL query, but not enough to offset the increase in the number of SQL queries; therefore *Hybrid* generates more joins for a path expression than *Shared*. Example: DTD “saej”.

Group 4. DTDs for which both *Shared* and *Hybrid* produce about the same number of joins per SQL query, and about the same number of SQL queries, resulting in approximately the same total number of joins. Example: DTD “math”.

Hybrid inlines more than *Shared* in Groups 1, 2 and 3. This reduces the number of joins per SQL query but increases the number of SQL queries. The net increase or decrease in the total number of joins depends on the structure of the DTD. In Group 4, most of the shared nodes are either set nodes or involved in recursion. Since *Shared* and *Hybrid* treat set nodes and recursive nodes identically, there is no significant difference in their performance in Group 4.

	Group 1	Group 2	Group 3	Group 4
Num DTDs	13	2	6	16

The number of DTDs in each group from all 37 DTDs is summarized in the table above. We can infer that in a large number of DTDs (Group 4), most of the shared nodes are either set nodes or recursive nodes.

3.6.3 Results for Path Expressions of Other Lengths

In the previous section, we showed the results for path expressions of length 3. In order to see how the results carry over to other path lengths, let us examine how the number of joins scales with the path length. We found that for almost all the DTDs, the number of joins scales linearly with the path length, the only difference is the scaling factor, which is determined by the structure of the DTD. Furthermore, the gap between the performance of *Shared* and *Hybrid* typically widens when the path lengthens. Figure 16 and Figure 17 show the scaling for two DTDs in group 1 and group 3 respectively.

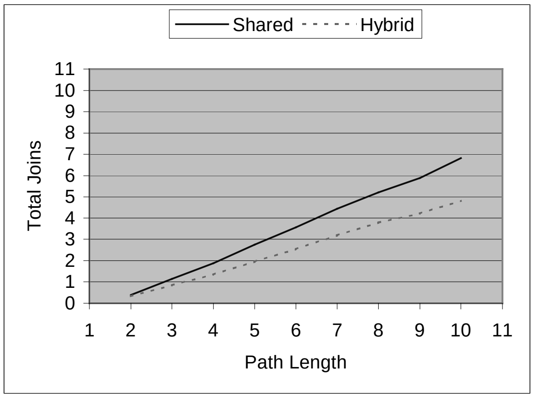

Figure 16

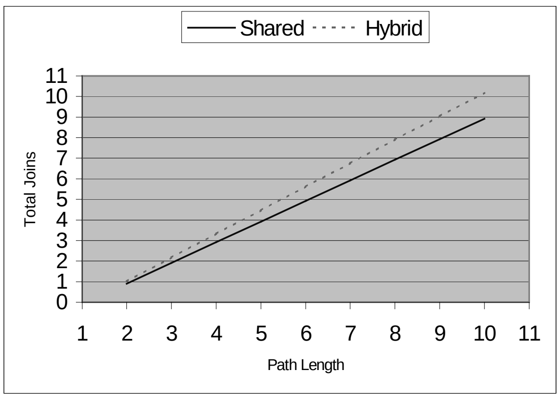

Figure 17

3.6.4 Evaluation Using Path Expressions Starting From the Document Root

So far, we have examined the performance of our algorithms assuming path expressions start from an arbitrary node in the DTD graph. What is different if the path expressions start from the root of a document? The real difference is in the total number of joins. A path expression starting from the root of a document is always converted to one SQL query - therefore the total number of joins is equivalent to the number of joins per SQL query. Since the *Hybrid* algorithm always produces fewer joins per SQL query, it is always better than *Shared* for path expressions that start from the document root.

For DTDs in groups 3 and 4 (the majority of DTDs), both *Shared* and *Hybrid* are practically the same. The main issue is the excessive fragmentation of the DTDs that leads to the number of joins being almost equal to the length of the path expression (Figure 17). This is likely to be very inefficient in the relational model, especially for long path lengths. The main cause of this fragmentation is the presence of set sub-elements. Section 6 includes a proposed extension to alleviate this problem.

4. Converting Semi-Structured Queries to SQL

Semi-structured query languages have a lot more flexibility than SQL. In particular, they allow path expressions with various operators and wild cards. The challenge is to rewrite these queries in SQL exploiting DTD information. In this section, we consider only queries with string values as results. Queries with more complex result formats are dealt with in Section 5. For ease of exposition, we present the translation algorithm only in the context of the *Shared* approach. The generalization to the other approaches is straightforward.

4.1 Converting Queries with Simple Path Expressions to SQL

Consider the following XML-QL query, and an equivalent Lorel-like query, over the DTD in Figure 2 that asks for the first and last name of the author of a book with title “The Selfish Gene”. Note that we have slightly extended the XML-QL syntax to query over all documents conforming to a DTD.

```
WHERE <book>
              <booktitle> The Selfish Gene </booktitle>
              <author>
                   <name>
                        <firstname> $f </firstname>
                        <lastname> $l </lastname>
                   </name>
              </author>
         </book> IN * CONFORMING TO pubs.dtd
CONSTRUCT <result> $f $l </result>
```

```
Select Y.name.firstname,
       Y.name.lastname
From   book X, X.author Y
Where X.booktitle = “Databases”
```

As can be seen from the Lorel-like representation, this query essentially consists of five path expressions, namely, *book*, *X.author*, *Y.name.firstname*, *Y.name.lastname* and *X.booktitle*. Of these path expressions, *book* is the root path expression and the others are dependent path expressions. This query is translated into SQL as follows: (a) first, the relation(s) corresponding to start of the root path expression(s) are identified and added to the from clause of the SQL query, then (b) if necessary, the path expressions are translated to joins among relations (when elements are inlined, joins are not necessary). The SQL query generated in this fashion for the example query above is shown in Figure 18. Note that a join condition has been added to the where clause to link the book and author and a selection (A.parentCODE = 0, where 0 indicates that the parent of the author is a book) is performed on author to make sure that only authors reached through book are considered.

```
Select A."author.name.firstname",
        A."author.name.lastname"
From   author A, book B
Where B.bookID = A.parentID
         AND A.parentCODE = 0
         AND B."book.booktitle" = "The Selfish Gene"
```

Figure 18

4.2 Converting Simple Recursive Path Expressions to SQL

Consider the following XML-QL query that requires the names of all editors reachable directly or indirectly from the monograph with title "Subclass Cirripedia". The corresponding XML-QL query (and an equivalent Lorel-like query) is shown below:

```
WHERE <*.monograph>
                <editor.(monograph.editor)*>
                      <name> $n </name>
                </>
                <title> Subclass Cirripedia </title>
            </> IN * CONFORMING TO pubs.dtd
CONSTRUCT <result> $n </result>
```

```
Select Y.name
From   *.monograph X, X.editor.(monograph.editor)* Y
Where X.title = "Subclass Cirripedia"
```

There are two interesting features about this query. The first is the tag "*.monograph" which states that we are interested in monographs reachable from any path. The second is the tag "editor.(monograph.editor)*" that specifies all editors reachable directly or indirectly from a monograph. The trick in converting this to a least fix-point query such as that supported by IBM DB2 is to determine (a) the initialization of the recursion and (b) the actual recursive path expression. In the example above, the initialization of the recursion is the path expression *.monograph.editor with the selection condition monograph.title = "Subclass Cirripedia" and the recursive path expression is monograph.editor. Each can be converted to a SQL fragment just like a simple path expression. The final query is the union of the two SQL fragments within a least fix-point operator. The query generated in this fashion is shown in Figure 19, in IBM DB2 syntax. Note that the "with clause" is the equivalent of the least fix-point operator in DB2.

```
With Q1 (monographID, name) AS
(Select X.monographID, X."editor.name"
 From monograph X
 Where X.title = "Subclass Cirripedia"
UNION ALL
 Select Z.monographID, Z."editor.name"
 From Q1 Y, monograph Z
 Where Y.monographID = Z.parentID AND
           Z.parentCODE = 0
)
Select A.name
From Q1 A
```

Figure 19

4.3 Converting Arbitrary Path Expressions to Simple Recursive Path Expressions

In general, path expressions can be of arbitrary complexity. For example, we could have a query that asks for all the name elements reachable directly or indirectly through *monograph*. This would be represented in a Lorel-like language as (an equivalent query can be expressed in XML-QL):

```
Select X
From monograph.(#)*.name X
```

We have a general technique that takes path expressions appearing in such queries (in this example "monograph.(#)*.name") and translates them into possibly many simple (recursive) path expressions. SQL queries are then generated for each simple recursive path expression. This notion of splitting a path expression to many simple path expressions is crucial to processing queries having arbitrary path expressions in SQL. The details of the technique are tedious and we omit them here in the interest of space.

Our technique is general enough to handle path expressions with nested recursion (e.g., "(a.(b)*.c)*"). However, relational database systems such as IBM DB2 cannot currently handle these queries because they do not have support for nested recursive queries.

5. Converting Relational Results to XML

In the previous section, we assumed that the results of a query were string values. We relax this assumption in this section and explore how the tabular results returned by SQL queries can be converted to complex structured XML documents. This is perhaps the main drawback in using current relational technology to provide XML querying – constructing arbitrary XML result sets is difficult. In this section we give some examples, using XML-QL as the illustrative query languages because it provides XML structuring constructs.

5.1 Simple Structuring

Consider the query in Figure 20 that asks for the first name and last name of all the authors of books, nested appropriately. Constructing such results from a relational system is natural and efficient, since it only requires attaching the appropriate tags for each tuple (Figure 21).

5.2 Tag Variables

A tag variable is one that ranges over the value of an XML tag. Some queries requiring tag variables in their results are naturally translated to the relational model. Consider the query in Figure 22 that ask for names of authors of all publications, nested under a tag specifying the type of publication. This can be handled by generating a relational query that contains the tag value as an element of the result tuple. Then at result generation

```
WHERE <book>
        <author>
          <firstname> $f </firstname>
          <lastname> $l </lastname>
        </>
      </> IN * CONFORMS TO pubs.dtd
CONSTRUCT  <author>
             <firstname> $f </firstname>
             <lastname> $l </lastname>
           </author>
```

Figure 20

```
(Richard, Dawkins)
(NULL, Darwin)
```

```
<author>
  <firstname> Richard </firstname>
  <lastname> Dawkins </lastname>
</author>
<author>
  <lastname> Darwin </lastname>
</author>
```

Figure 21

```
WHERE <$p>
        <author>
          <firstname> $f </firstname>
          <lastname> $l </lastname>
        </>
      </> IN * CONFORMS TO pubs.dtd
CONSTRUCT <$p>
            <author>
              <firstname> $f </firstname>
              <lastname> $l </lastname>
            </author>
          </>
```

Figure 22

```
(book, Richard, Dawkins)
(book, NULL, Darwin)
(monograph, NULL, Darwin)
```

```
<book>
  <author>
    <firstname> Richard </firstname>
    <lastname> Dawkins </lastname>
  </author>
</book>
<book>
  <author>
    <lastname> Darwin </lastname>
  </author>
</book>
<monograph>
  <author>
    <lastname> Darwin </lastname>
  </author>
</monograph>
```

Figure 23

time, the tag attribute in the result tuple can be converted to the appropriate XML tag (Figure 23).

5.3 Grouping

Consider the query in Figure 24 that requires all the publications of an author (assuming an author is uniquely identified by his/her last name) to be grouped together, and within this structure, requires the titles of publications to be grouped by the type of the publication. The relational result from the translation of this query will be a set of tuples having fields corresponding to last name of author, title of publication and type of publication. However, we cannot use the relational group-by operator to group by last name and type of publication because the SQL group-by semantics implies that we should apply an aggregate function to title, which does not make sense. Thus, the options are either (a) have the relational engine order the result tuples first by last name and then by type and scan the result in order to construct the XML document or (b) get an unordered set of tuples and do a grouping operation, by last name and then by type, outside the relational engine. The first approach is illustrated in Figure 25.

Figure 25 illustrates several points. The first is that treating tag variables as attributes in the result relation provides a way of uniformly treating the contents of the result XML document. In this case, we are able to group by the tag variable just like any other attribute. The second observation is that some relational database functionality (hash-based group-by) is either not fully exploited or is duplicated outside.

5.4 Complex Element Construction

Unfortunately, returning tag values as tuple attributes cannot handle all result construction problems. In particular, queries that are required to return complex XML elements are problematic. Consider a query that asks for all article elements in the XML data set, and furthermore assume that an article may have multiple authors and multiple titles. In object-relational terminology, article has two set-valued attributes, authors and titles, corresponding to two set sub-elements in XML terminology.

```
WHERE <book>
        <article> $a </article>
      </> IN * CONFORMS TO pubs.dtd
CONSTRUCT  <article> $a </>
```

To create the appropriate result, we must retrieve all authors and all titles for each article. This is difficult to do in the relational model because flattening multiple set-valued attributes into tuple format gives rise to a multi-valued dependency [11] and is likely to be very inefficient when the sets are large, for example, if papers have many authors and many titles. There appears to be no efficient way to tackle this problem in the traditional relational model. One solution would be to return separate relations, each flattening one set-valued attribute and “join” these relations outside the database while constructing the XML document. However, this requires duplication of database functionality both in terms of execution and optimization. This solution would be particularly bad for an element with many set-valued attributes. A related problem occurs when reconstructing recursive elements. We return to these issues in Section 6.

```
WHERE <$p>
            <(title|booktitle)> $t </>
            <author>
               <lastname> $l </lastname>
            </>
        </> IN * CONFORMS TO pubs.dtd
CONSTRUCT <author ID=authorID($l)>
                <name> $l </name>
                <$p ID=pID($p)>
                    <title> $t </>
                </>
            </>
```

Figure 24

```
(Darwin, book,  Origin of Species)
(Darwin, book, Descent of Man)
(Darwin, monograph, Subclass
 Cirripedia)
(Dawkins, book, The Selfish Gene)
```

```
<author>
   <name> Darwin </name>
   <book>
       <title> Origin of Species </title>
       <title> The Descent of Man </title>
  </book>
   <monograph>
       <title> Subclass Cirripedia </title>
   </monograph>
</author>
<author>
    <name> Dawkins </name>
    <book>
        <title> The Selfish Gene </title>
    </book>
</author>
```

Figure 25

5.5 Heterogeneous Results

Consider the following XML-QL query that creates a result document having both titles and authors as elements (this is the heterogeneous result). This is easily handled in our approach for translating queries because this query would be split into two queries, one for selecting titles and another for selecting authors. The results of the two queries can be handled in different ways, one constructing title elements and another constructing author elements. The results can then be merged together.

```
WHERE <article>
                <$p> $y </>
            </article> IN * CONFORMING TO pubs.dtd
CONSTRUCT <$p> $y </>
```

5.6 Nested Queries

XML-QL is structured in terms of query blocks and one query block can be nested under another. These nested queries can be rewritten in terms of SQL queries, using outer joins (and possibly skolem function ids) to construct the association between a query and a sub-query. The details are complex and we omit it in the interest of space.

6. Conclusions

With the growing importance of XML documents as a means to represent data in the World Wide Web, there has been a lot of effort on devising new technologies to process queries over XML documents. Our focus in this paper, however, has been to study the virtues and limitations of the traditional relational model for processing queries over XML documents conforming to a schema. The potential advantages of this approach are many – reusing a mature technology, using an existing high performance system, and seamlessly querying over data represented as XML documents or relations. We have shown that it is possible to handle most queries on XML documents using a relational database, barring certain types of complex recursion.

Our qualitative evaluation based on real DTDs from diverse domains raises some performance concerns – specifically, in many cases relatively simple XML queries require either many SQL queries, or require a few SQL queries with many joins in them. It is an open question whether semi-structured query processing techniques can do this kind of work more efficiently. The fact that semi-structured models represent a sequence of joins as a path expression, or handle what is logically a union of queries by using wildcards and "or" operators, does not automatically imply more efficient evaluation strategies.

Our experience has shown that relational systems could more effectively handle XML query workloads with the following extensions:

Support for Sets: Set-valued attributes would be useful in two important ways. First, storing set sub-elements as set-valued attributes [19,21] would reduce fragmentation. This is likely to be a big win because most of the fragmentation we observed in real DTDs was due to sets. Second, set-valued attributes, along with support for nesting [13], would allow a relational system to perform more of the processing required for generating complex XML results.

Untyped/Variable-Typed References: IDREFs are not typed in XML. Therefore, queries that navigate through IDREFs cannot be handled in current relational systems without a proliferation of joins – one for each possible reference type.

Information Retrieval Style Indices: More powerful indices, such as Oracle8i's ConText search engine for XML [17], that can index over the structure of string attributes would be useful in querying over ANY fields in a DTD. Further, under restricted query requirements, whole fragments of a document can be stored as an indexed text field, thus reducing fragmentation.

Flexible Comparisons Operators: A DTD schema treats every value as a string. This often creates the need to compare a string attribute with, say, an integer value, after typecasting the string to an integer. The traditional relational model cannot support such comparisons. The problem persists even in the presence of DCDs or XML

Schemas because different DTDs may represent "comparable" values as different types. A related issue is that of flexible indices. Techniques for building such indices have been proposed in the context of semi-structured databases [14].

Multiple-Query Optimization/Execution: As outlined in Section 4, complex path expressions are handled in a relational database by converting them into many simple path expressions, each corresponding to a separate SQL query. Since these SQL queries are derived from a single regular path expression, they are likely to share many relational scans, selections and joins. Rather than treating them all as separate queries, it may be more efficient to optimize and execute them as a group [20].

More Powerful Recursion: As mentioned in Section 4, in order to fully support all recursive path expressions, support for fixed point expressions defined in terms of other fixed point expressions (i.e., nested fixed point expressions) is required.

These extensions are not by themselves new and have been proposed in other contexts. However, they gain new importance in light of our evaluation of the requirements for processing XML documents. Another important issue to be considered in the context of the World Wide Web is distributed query processing – taking advantage of queryable XML sources. Further research on these techniques in the context of processing XML documents will, we believe, facilitate the use of sophisticated relational data management techniques in handling the novel requirements of emerging XML-based applications.

7. Acknowledgements

Funding for this work was provided by DARPA through Rome Research Laboratory Contract No. F30602-97-2-0247 and NSF through NSF Award CDA-9623632.

8. References

1. S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, "The Lorel Query Language for Semistructured Data", International Journal on Digital Libraries, 1(1), pp. 68-88, April 1997.
2. J. Bosak, T. Bray, D. Connolly, E. Maler, G. Nicol, C. M. Sperberg-McQueen, L. Wood, J. Clark, "W3C XML Specification DTD", http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm.
3. T. Bray, J. Paoli, C. M. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0", http://www.w3.org/TR/REC-xml.
4. T. Bray, C. Frankston, A. Malhotra, "Document Content Description for XML", http://www.w3.org/TR/NOTE-dcd.
5. P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, "A Query Language and Optimization Techniques for Unstructured Data", Proceedings of the ACM SIGMOD Conference, Montreal, Canada, June 1996.
6. V. Christophides, S. Abiteboul, S. Cluet, M. Scholl, "From Structured Documents to Novel Query Facilities", Proceedings of the ACM SIGMOD Conference, Minneapolis, Minnesota, May 1994.
7. G. Copeland, S. Khoshafian, "A Decomposition Storage Model", Proceedings of the ACM SIGMOD Conference, Austin, Texas, May 1985.
8. R. Cover, "The SGML/XML Web Page", http://www.oasis-open.org/cover/xml.html.
9. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, "XML-QL: A Query Language for XML", http://www.w3.org/TR/NOTE-xml-ql.
10. Deutsch, M. Fernandez, D. Suciu, "Storing Semi-structured Data with STORED", Proceedings of the ACM SIGMOD Conference, Philadelphia, Pennslyvania, May 1999.
11. R. Fagin, "Multi-valued Dependencies and a New Normal Form for Relational Databases", ACM Transactions on Database Systems, 2(3), pp. 262-278, 1977.
12. M. Fernandez, D. Suciu, "Optimizing Regular Path Expressions Using Graph Schemas", Proceedings of the Fourteenth ICDE Conference, Orlando, Florida, February 1998.
13. Jaeschke, H. J. Schek, "Remarks on the Algebra of Non First Normal Form Relations", Proceedings of the ACM Symposium on Principles of Database Systems, Los Angeles, California, March 1982.
14. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom, "Lore: A Database Management System for Semistructured Data", SIGMOD Record, 26(3), pp. 54-66, September 1997.
15. J. McHugh, J. Widom, "Compile-Time Path Expansion in Lore", Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, Jerusalem, Israel, January 1999.
16. Microsoft Corporation, XML Schema, http://www.microsoft.com/xml/schema/reference/star.asp.
17. Oracle Corporation, "XML Support in Oracle 8 and beyond", Technical white paper, http://www.oracle.com/xml/documents.
18. The Query Languages Workshop (QL'98), http://www.w3.org/TandS/QL/QL98/, December 1998.
19. K. Ramasamy, J. F. Naughton, D. Maier, "Storage Representations for Set-Valued Attributes", Working Paper, Department of Computer Sciences, University of Wisconsin-Madison.
20. T. Sellis, "Multiple-Query Optimization", ACM Transactions on Database Systems, 12(1), pp. 23-52, June 1990.
21. Zaniolo, "The Database Language GEM", Proceedings of the ACM SIGMOD Conference, San Jose, California, May 1983.