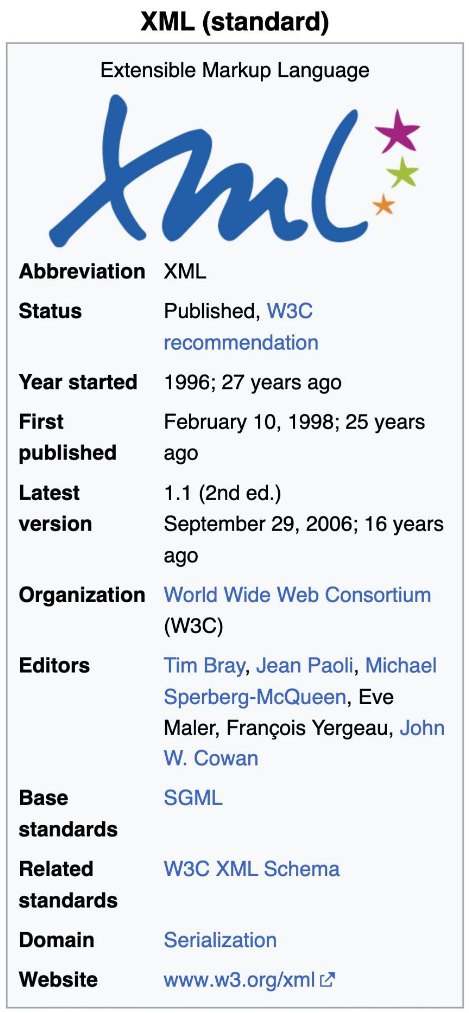
Relational Databases and XML

Sepehr Jalalian and Matt Oddo - UBC CPSC 504 - 2023.02.16

What is XML?

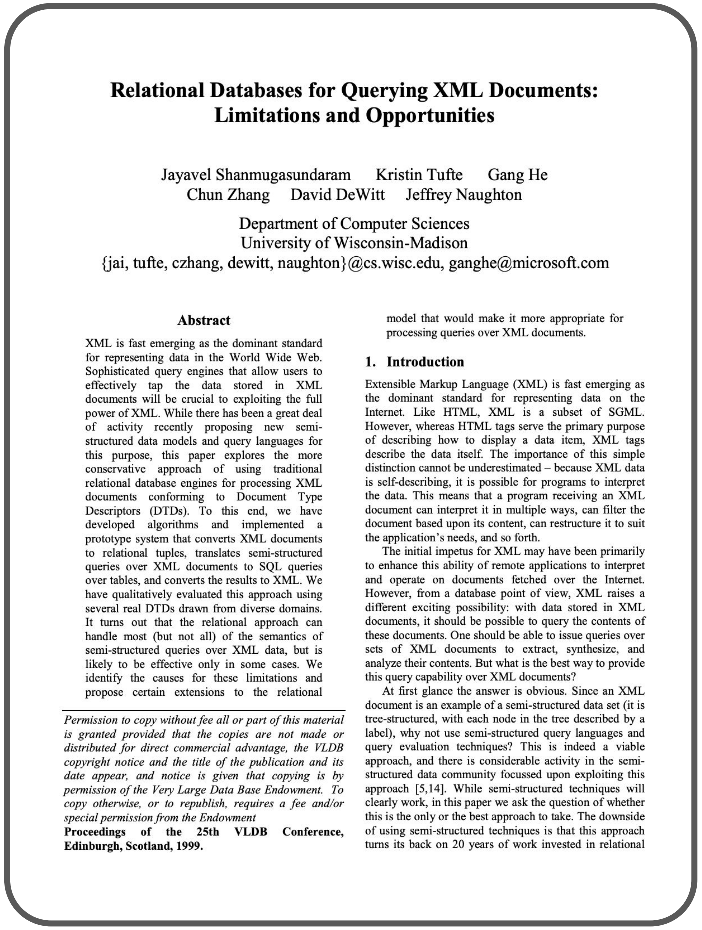
- Both HTML and XML are subsets of SGML, itself a successor of IBM's Generalized Markup Language (GML) from the 1960s.
- In HTML `</tags>` have the primary purpose of displaying data.
- In XML `</tags>` describe data itself, and also the data's organizational hierarchy.
- Introduced in 1996, XML quickly became the standard format to transmit information through the World Wide Web.
- The most up-to-date version of XML is from 2006.



XML (standard)

Extensible Markup Language

Abbreviation	XML
Status	Published, W3C recommendation
Year started	1996; 27 years ago
First published	February 10, 1998; 25 years ago
Latest version	1.1 (2nd ed.) September 29, 2006; 16 years ago
Organization	World Wide Web Consortium (W3C)
Editors	Tim Bray, Jean Paoli, Michael Sperberg-McQueen, Eve Maler, François Yergeau, John W. Cowan
Base standards	SGML
Related standards	W3C XML Schema
Domain	Serialization
Website	www.w3.org/xml



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.

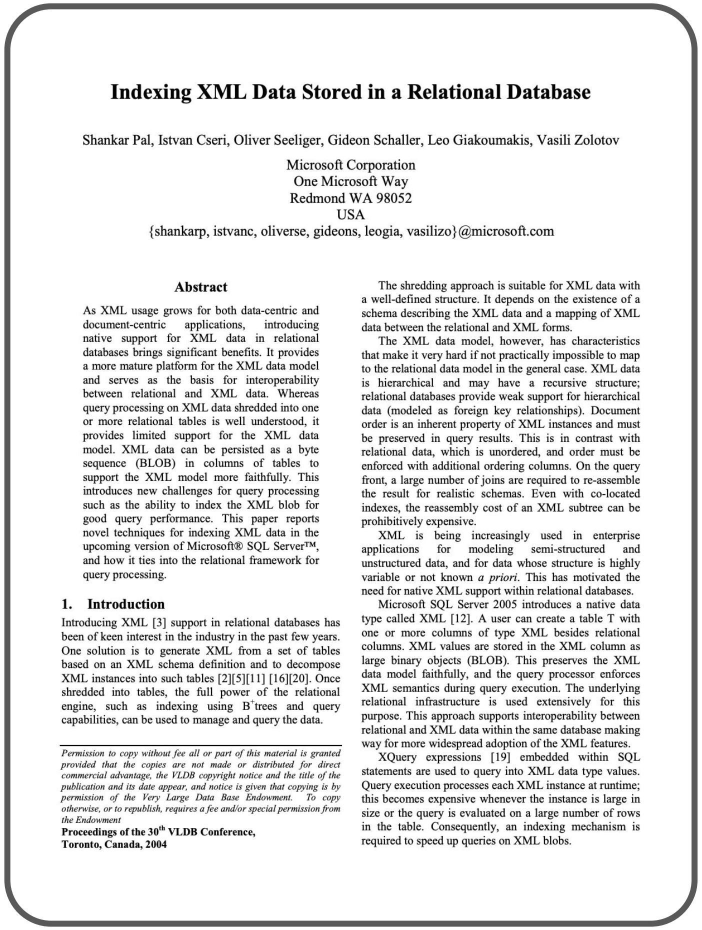
Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

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



Indexing XML Data Stored in a Relational Database

Shankar Pal, Istvan Cseri, Oliver Seeliger, Gideon Schaller, Leo Giakoumakis, Vasili Zolotov

Microsoft Corporation
One Microsoft Way
Redmond WA 98052
USA
{shankarp, istvanc, oliverse, gideons, leogia, vasilizo}@microsoft.com

Abstract

As XML usage grows for both data-centric and document-centric applications, introducing native support for XML data in relational databases brings significant benefits. It provides a more mature platform for the XML data model and serves as the basis for interoperability between relational and XML data. Whereas query processing on XML data shredded into one or more relational tables is well understood, it provides limited support for the XML data model. XML data can be persisted as a byte sequence (BLOB) in columns of tables to support the XML model more faithfully. This introduces new challenges for query processing such as the ability to index the XML blob for good query performance. This paper reports novel techniques for indexing XML data in the upcoming version of Microsoft® SQL Server™, and how it ties into the relational framework for query processing.

1. Introduction

Introducing XML [3] support in relational databases has been of keen interest in the industry in the past few years. One solution is to generate XML from a set of tables based on an XML schema definition and to decompose XML instances into such tables [2][5][11] [16][20]. Once shredded into tables, the full power of the relational engine, such as indexing using B^+trees and query capabilities, can be used to manage and query the data.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

The shredding approach is suitable for XML data with a well-defined structure. It depends on the existence of a schema describing the XML data and a mapping of XML data between the relational and XML forms.

The XML data model, however, has characteristics that make it very hard if not practically impossible to map to the relational data model in the general case. XML data is hierarchical and may have a recursive structure; relational databases provide weak support for hierarchical data (modeled as foreign key relationships). Document order is an inherent property of XML instances and must be preserved in query results. This is in contrast with relational data, which is unordered, and order must be enforced with additional ordering columns. On the query front, a large number of joins are required to re-assemble the result for realistic schemas. Even with co-located indexes, the reassembly cost of an XML subtree can be prohibitively expensive.

XML is being increasingly used in enterprise applications for modeling semi-structured and unstructured data, and for data whose structure is highly variable or not known *a priori*. This has motivated the need for native XML support within relational databases.

Microsoft SQL Server 2005 introduces a native data type called XML [12]. A user can create a table T with one or more columns of type XML besides relational columns. XML values are stored in the XML column as large binary objects (BLOB). This preserves the XML data model faithfully, and the query processor enforces XML semantics during query execution. The underlying relational infrastructure is used extensively for this purpose. This approach supports interoperability between relational and XML data within the same database making way for more widespread adoption of the XML features.

XQuery expressions [19] embedded within SQL statements are used to query into XML data type values. Query execution processes each XML instance at runtime; this becomes expensive whenever the instance is large in size or the query is evaluated on a large number of rows in the table. Consequently, an indexing mechanism is required to speed up queries on XML blobs.

Relational Databases for Querying XML Documents

WHAT WAS HAPPENING THEN?

- A semi-structured query language was traditionally used to query over semi-structured data sets in XML format, but very cumbersome and slow.
- Examples listed are XML-QL, Lorel, UnQL, and XQL (from Microsoft).

THE NEW IDEA

- Convert XML to relational table, query it with mature relation model tools, and convert the output back to XML.

Relational Databases for Querying XML Documents

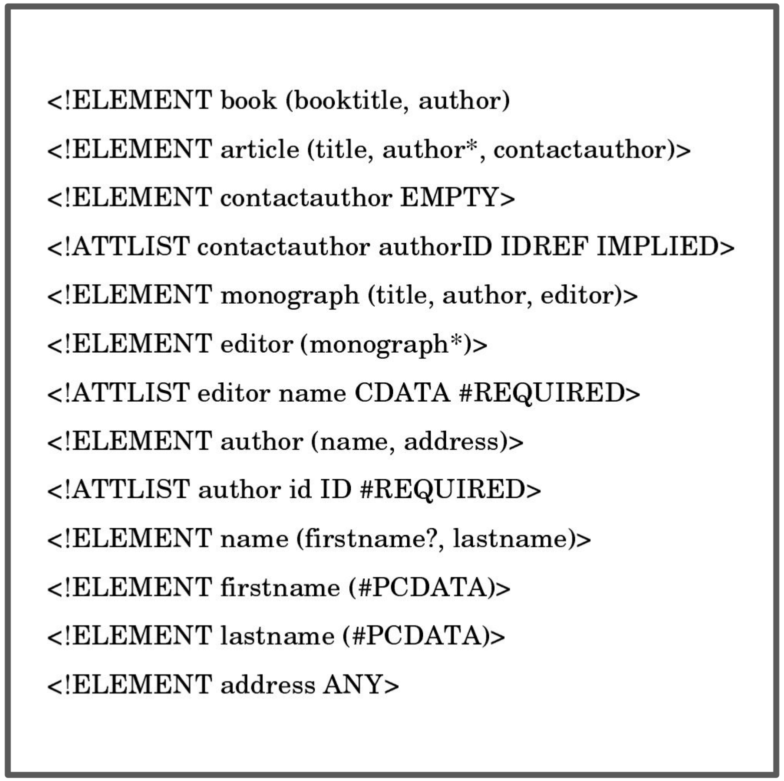
EXECUTION IN **FOUR STEPS**

1. Process a Document Type Description (DTD) to generate a relational schema.

2. Parse XML documents conforming to DTDs and load them into tuples of relational tables in a standard commercial DBMS.

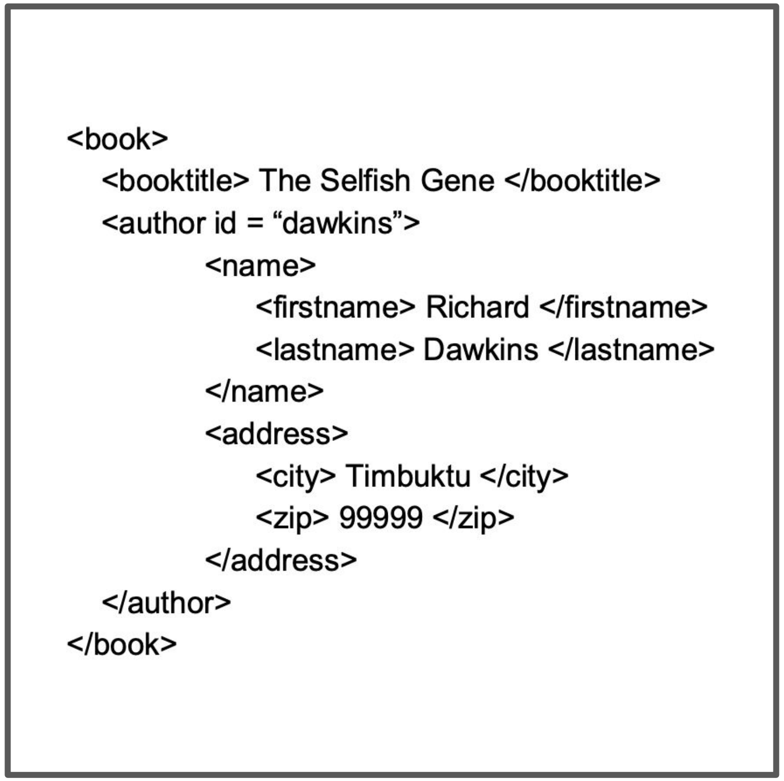
3. Translate semi-structured queries over XML documents into SQL queries over the corresponding relational data.

4. Convert the results back to XML.



```
<!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>
```

A Document Type Descriptor (DTD) specification



```
<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>
```

An XML document that conforms to this DTD

Relational Databases for Querying XML Documents

- Note that the DTD comes from the **document community**, so lacks the expressive power that database people wanted - simultaneously making it computationally intractable.
    - The XML Schema Description (XSD) standard was created by W3C to fix this and make document definitions more powerful.
    - While it granted tons of expressive power, XSD was severely criticised because it is *impossible* to understand.

Relational Databases for Querying XML Documents

DISCUSSION (Groups of 3-4)

The authors demonstrate using the traditional relational database engines for processing XML documents. On the other hand, XML databases and query languages were under development at that time.

- Would you rather create an XML database and query processing system from scratch, or use a relational backend. Why? If it depends, what does it depend on?

- In a more broad sense, what are the pros and cons of leveraging mature technology to solve a different problem versus providing a dedicated solution to the new problem from scratch? (Jianhao)

Relational Databases for Querying XML Documents

[STEP 1] Process a DTD to generate a relational schema:

- It is tempting to map XML DTD to relations.
  - However, there is no correspondence with ER diagram model, so schema conversion (inlining algorithm) is required.
- But first, DTDs are actually problematic because they can be very complex, so the authors describe **three initial simplification transformations**.
  - Without DTD simplification you have direct mapping elements to relations, which can lead to excessive fragmentation of the document (too many joins!).

$(e_1, e_2)^* \rightarrow e_1^*, e_2^*$
$(e_1, e_2)? \rightarrow e_1?, e_2?$
$(e_1|e_2) \rightarrow e_1?, e_2?$

FLATTENING

Convert a nested definition into a flat representation (e.g. binary operators "," and "|" do not appear inside any operator).

$e_1^{**} \rightarrow e_1^*$
$e_1^*? \rightarrow e_1^*$
$e_1?^* \rightarrow e_1^*$
$e_1?? \rightarrow e_1?$

SIMPLIFICATION

Reduce many unary operators to a single unary operator.

$\ldots, a^*, \ldots, a^*, \ldots \rightarrow a^*, \ldots$
$\ldots, a^*, \ldots, a?, \ldots \rightarrow a^*, \ldots$
$\ldots, a?, \ldots, a^*, \ldots \rightarrow a^*, \ldots$
$\ldots, a?, \ldots, a?, \ldots \rightarrow a^*, \ldots$
$\ldots, a, \ldots, a, \ldots \rightarrow a^*, \ldots$

GROUPING

Groups subelements having the same name (e.g. two a* subelements are grouped into one a* subelement)

Also authors further simplify DTDs by making all **"+"** operators into **"*"** operators.

With simplified DTDs in place, the authors propose **three** inlining algorithms to convert XML/DTD into Relational Model

BASIC
INLINING
TECHNIQUE

SHARED
INLINING
TECHNIQUE

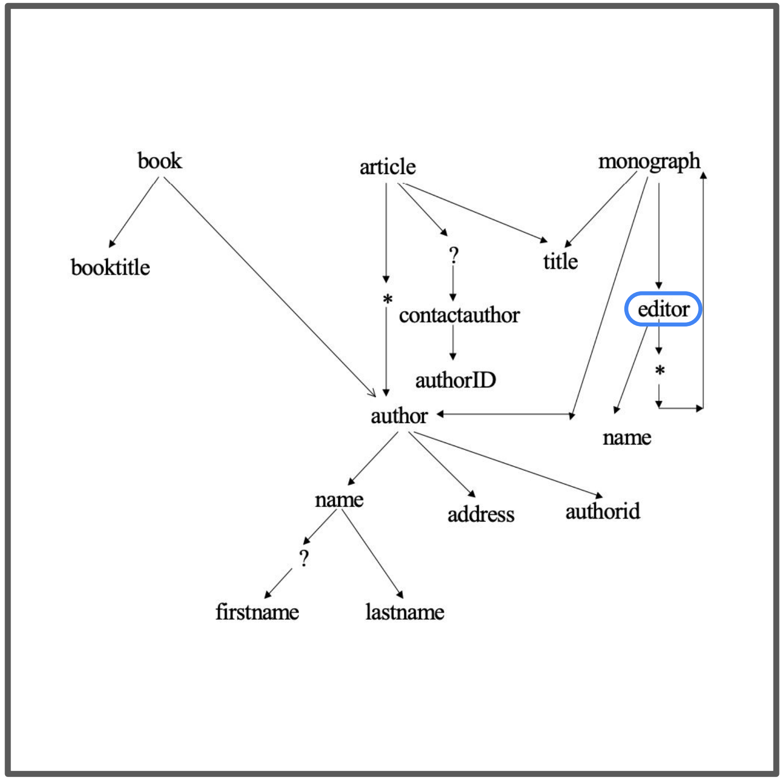
HYBRID
INLINING
TECHNIQUE

Relational Databases for Querying XML Documents

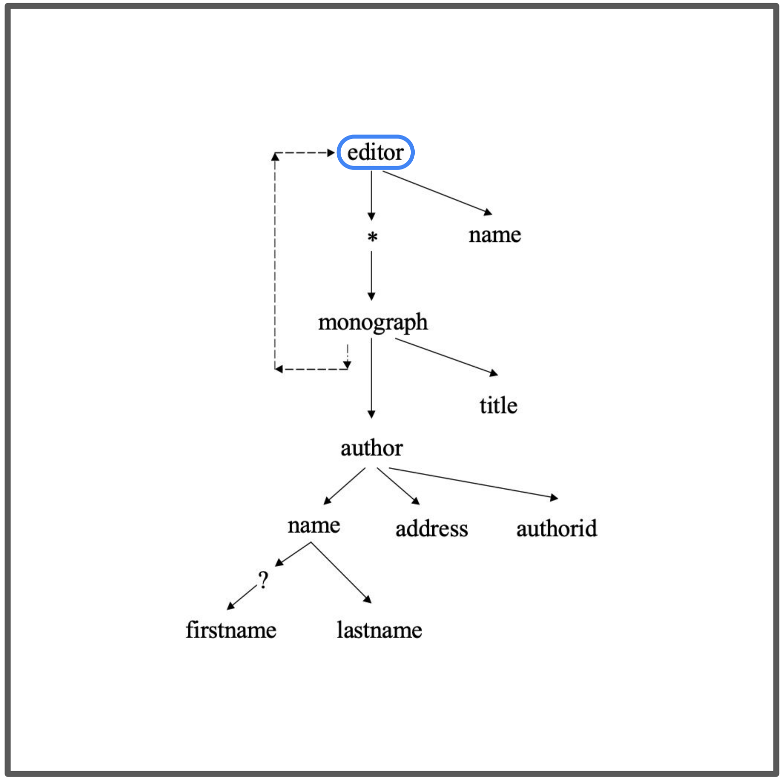
[STEP 2] Parse XML documents conforming to DTDs and load them into tuples of relational tables in a standard commercial DBMS.

- **BASIC** INLINING TECHNIQUE - Relations for every element inline with the element's descendants, flat structure.

    - Good for certain queries:
        - “List all authors of a book”
    - But bad for:
        - “List all authors having first name Jack” (union of 5 queries!)
        - Large number of relations
        - Complicated to handle DTD recursion
        - Separated schema for each root element
        - High resource consumption for schema translation



A graph of the DTD specification shown before, note cycle.



Tree graph from the editor node, note recursion.

Relational Databases for Querying XML Documents

[STEP 2] Parse XML documents conforming to DTDs and load them into tuples of relational tables in a standard commercial DBMS.

- **SHARED** INLINING TECHNIQUE
    - Inspired by BASIC, first ensures an element node is represented in exactly one relation.
    - Identifies element nodes which are represented in multiple relations in BASIC and shares between them by creating separating relations.
    - Results in a surprisingly small number of relations!

BASIC INLINING TECHNIQUE

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)

SHARED INLINING TECHNIQUE

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)

Relational Databases for Querying XML Documents

[STEP 2] Parse XML documents conforming to DTDs and load them into tuples of relational tables in a standard commercial DBMS.

- **SHARED** INLINING TECHNIQUE - Identify nodes in multiple relations and share them by creating separate relations (nodes with in-degree greater than one).

    - Good for certain queries:
        - Reduced relations through shared element.
        - Reduced joins “List all authors having first name Jack”.
    - But bad for:
        - Less relations, more joins
        - Inefficient when comparing to BASIC.
        - Increased number of joins starting at a particular node.

Relational Databases for Querying XML Documents

[STEP 2] Parse XML documents conforming to DTDs and load them into tuples of relational tables in a standard commercial DBMS.

- **HYBRID** INLINING TECHNIQUE - Inlines elements with in-degree greater than one that are not recursive or reached through a "*" node.
    - Combines the join reduction properties of BASIC with the sharing features of SHARED.
    - Reduces number of joins but increases number of SQL queries, and the number of joins scales linearly with the path length (qualitative eval).

Relational Databases for Querying XML Documents

DISCUSSION (Groups of 3 or class-wide)

Their evaluation metric (given in section 3.6.1) is:

"the average number of SQL joins required to process path expressions of a certain length N"

- This metric can be measured as the product of the average number of SQL queries generated for path expressions of length N and the average number of joins in each SQL query for path expressions of length N.

- Do you think this is a good idea? What about the two partial metrics above?
  Why or why not?

Relational Databases for Querying XML Documents

[STEP 3] Translate semi-structured queries over XML documents into SQL queries over the corresponding relational data.

- Simple path expressions - add the relations corresponding to the start of the root path expression and translate path expressions to joins.
- Simple recursive path expressions - determine the initialization of the recursion and the actual recursive path expression.
- Arbitrary path expressions - translate them into possibly many simple (recursive) path expressions.

Relational Databases for Querying XML Documents

[STEP 4] Finally, convert query results back into XML (using XML-QL)

- Simple Structuring - each tuple gets XML tags.
  Very simple indeed, and follows what is expected.

- However for tag variables and grouping, results get increasingly unexpected. The idea breaks down further considering complex element construction and heterogenous results.

CONCLUSION - Converting XML to relational for query processing has some limitations. Kudos for trying though!

Simple Structuring

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

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

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

Tag Variable

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

```
(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>
```

Grouping

```
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 </>
            </>
          </>
```

```
(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>
```

Indexing XML Data Stored in a Relational Database

It’s 2004 and XMLs are all over the place! I want to have them stored somewhere where I can efficiently fetch them. But how?

- Microsoft SQL Server offers a special column that stores your XML document as a byte sequence (BLOB). Then for good query performance, you must index this XML BLOB.

- Conversion into relational model, or *shredding* into INFOSET (XML as relational data), allows indexing using B+trees and query capabilities:

    - Unlocks ability to manage and query the data
    - Works best if your XML has a well-defined structure

- However, shredding arbitrary XML into a relational table is very difficult. You may need an overwhelming amount of joins, which is very expensive!

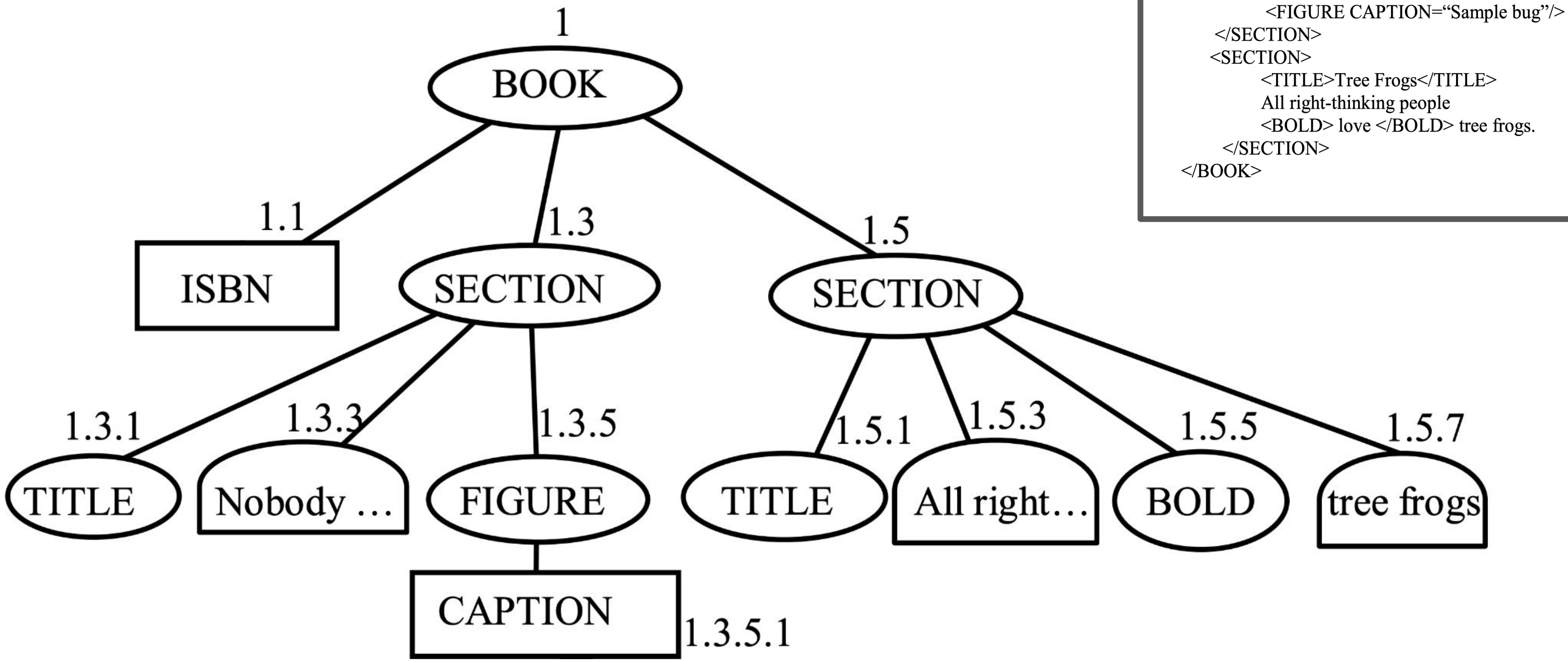
Indexing XML Data Stored in a Relational Database

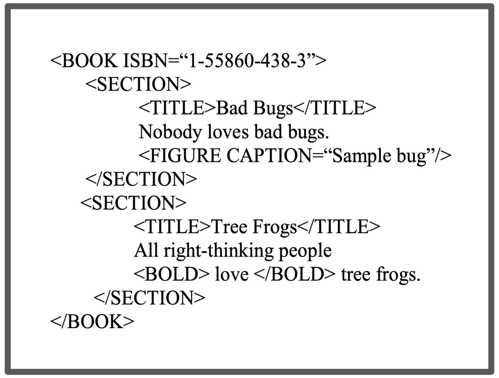
The Primary XML index:

- ORDPATH labels all nodes in XML tree:
    - Nodes behaves like a primary key in the INFOSET
    - Mechanism for labeling nodes in an XML tree
    - Preserves XML structural fidelity
    - Allows insertion of nodes anywhere without re-labeling
    - Independent of XML schemas typing XML instances
    - Encodes the parent-child relationship
- An advantage is ORDPATH allows full power of the relational engine, however XML is hierarchical and can have recursive structure.

Indexing XML Data Stored in a Relational Database

- Node labels in ORDPATH





```
<BOOK ISBN="1-55860-438-3">
    <SECTION>
        <TITLE>Bad Bugs</TITLE>
        Nobody loves bad bugs.
        <FIGURE CAPTION="Sample bug"/>
    </SECTION>
    <SECTION>
        <TITLE>Tree Frogs</TITLE>
        All right-thinking people
        <BOLD> love </BOLD> tree frogs.
    </SECTION>
</BOOK>
```

Indexing XML Data Stored in a Relational Database

DISCUSSION (In pairs)

- We have seen two approaches in processing XML data:
  - Decomposing XML into relational tables
  - Storing XML as BLOBs
- Can you think of an application in which either of these methods is more suitable over the other?

Indexing XML Data Stored in a Relational Database

The Secondary XML indexes:

- Provided on top of ORDPATH (additional columns) to improve query performance for each special type of queries, helps with bottom-up evaluation:
    - PATH
    - PATH_VALUE
    - PROPERTY
    - VALUE
    - CONTENT
- Qualifying nodes in the secondary XML indexes conduct a back-join with ORDPATH to enable continuation of query execution with those nodes.
- Significant performance gains.

Indexing XML Data Stored in a Relational Database

Authors ran an experimental evaluation XMark, an XML query benchmark that models an auction scenario.

- Four query results at 30 scale factor:

Q1	Performs extremely well (595.3) with PATH_VALUE index
Q5	No changes (execution time gain always under 2).
Q15	PATH_VALUE index gain (18.3)
Q16	PATH_VALUE index gain (48.2)

- Performance gains overall due to parsing XML BLOB multiple times (all indexes)
- PATH and PATH_VALUE contribute the most gain of the secondary indexes.

- Overall no figures to show experimental results, only tables. Suspicious.

Indexing XML Data Stored in a Relational Database

DISCUSSION (groups of 4)

- Does the emphasis on indexing surprise you?
- What are disadvantages of having several indexes?
- Does this seem like it would be more or less of an issue than for OO? What about join processing?