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

Presentation: Chris (Xiangyu Shi)
Discussion: Kaiyun Guo
Slides adapted from Pei Lee, Modified by Rachel Pottinger

What is XML?

Definition of XML

- XML stands for Extensible Markup Language.
- It is a markup language and file format for data.
- It is a subset of Standard Generalized Markup Language (SGML), similar to HTML.

HTML vs. XML

- HTML has a primary purpose of displaying data.
- XML describes data itself.

Purpose of XML

- Serialization; Designed to store, transmit, and represent data on the Internet.

Motivation

XML quickly became the standard format to transmit information through the WWW.

Database point of view: Challenge lies in effectively querying the data stored in XML documents.

Traditional Approach

- Use of semi-structured query languages: XML-QL, Lorel, UnQL, XQL.

Innovative Methodology

- Proposes leveraging existing relational database technology.
- Convert XML documents to relational structures, enabling the use of SQL for queries and reformatting the query outcomes back into XML.
- The key is Document Type Descriptors(DTD).

Discussion (Group of 4)

While there are many semi-structured data methods, the paper prefers to adapt XML to relational database systems.

- 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?

XML

- Self-describing, consists of nested element structures, starting with a root element.
- Element data can be in the form of attributes or sub-elements.
- E.g.

```
<student>
<name>John</name>
<phone>604xxxxxxxx</phone>
<phone>778xxxxxxxx</phone>
</student>
```

DTD

- Schema for XML: it describes the structure of XML documents by specifying the names of its sub-elements and attributes.
- E.g.
  - [*] = zero or more
  - [+] = one or more
  - [?] = zero or one

```
<!ELEMENT student(name,
phone+, fax*)>
```

Four steps - General idea of approach

- 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.
- Third, we translate semi-structured queries over XML documents into SQL queries over the corresponding relational data.
- Finally, we convert the results back to XML

STEP 1 - Process a DTD

- DTDs can be very complex which is a problem
- Three initial simplification transformations

Flattening Transformations	Simplification Transformations	Grouping Transformations
$(e_1, e_2)^* \rightarrow e_1^*, e_2^*$ $(e_1, e_2)? \rightarrow e_1?, e_2?$ $(e_1 \mid e_2) \rightarrow e_1?, e_2?$	$e_1^{**} \rightarrow e_1^*$ $e_1^*? \rightarrow e_1^*$ $e_1?^* \rightarrow e_1^*$ $e_1?? \rightarrow e_1?$	$\ldots, a^*, \ldots, a^*, \ldots \rightarrow a^*, \ldots$ $\ldots, a^*, \ldots, a?, \ldots \rightarrow a^*, \ldots$ $\ldots, a?, \ldots, a^*, \ldots \rightarrow a^*, \ldots$ $\ldots, a?, \ldots, a?, \ldots \rightarrow a^*, \ldots$ $\ldots, a, \ldots, a, \ldots \rightarrow a^*, \ldots$
Convert a nested definition into a flat representation (i.e. "," and "|" do not appear inside any operator)	Reduce many unary operators to a single unary operator.	Groups sub-elements having the same name (i.e. two a* sub-elements are grouped into one a*)

STEP 2 - DTD to a relational schema

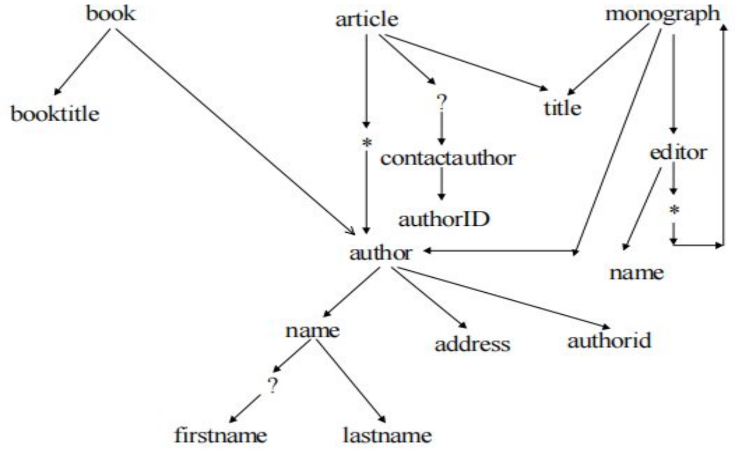
- Creating relational schemas based on a structured data model like the Entity-Relationship(ER) model - quite straightforward.
- XML DTDs don’t have a correspondence to the ER model.
- Directly mapping elements to relations can lead to excessive fragmentations of the documents.
- Three Techniques
    - The Basic Inlining Technique
    - The Shared Inlining Technique
    - The Hybrid Inlining Technique

STEP 2 - Basic Inlining Technique

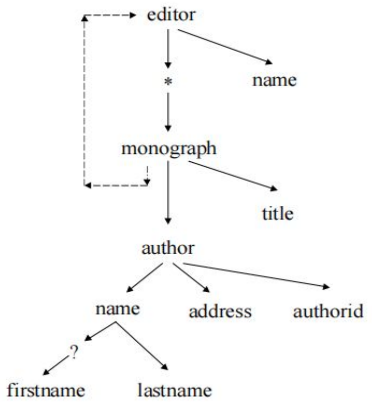
- Create relations for every element.
- To solve the fragmentation problem by inlining as many descendants of an element as possible into a single relation.
    - Set-valued attributes and Recursion

```
<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>
```

XML document



DTD graph



Element graph for *editor*

STEP 2 - 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)

Example relational schema of a DTD

- E.g. Book relation to tuple:

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

- Pros :
  - Good for certain type of queries Such as “List all authors of books“.
- Cons:
  - Large number of relations.
  - Inefficient for queries such as “list all authors having first name Jack” .
  - Complicated to handle DTD recursion.
  - Separated schema for each root element.
  - High resource consumption for schema translation.

STEP 2 - Shared Inlining Technique

- Avoid the drawbacks of Basic tech.
- Ensure an element node is represented in exactly one relation.
- Identify the element nodes that are represented in multiple relations in Basic and share them by creating separating relations (element nodes with in-degree greater than one).

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)

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)

STEP 2 - Shared Inlining Technique

- Pros :
  - Reduced relations through shared elements
  - Good for certain type of queries (e.g. list all authors having first name Jack)

- Cons:
  - Inefficient when comparing to Basic Inlining
    (increased no. of joins starting at a particular node)

- Hybrid!

STEP 2 - Hybrid Inlining Technique

- Combine Basic and Shared (Join reduction + Sharing).
- Based on Shared inlining.
- Additionally inline elements with in-degree greater than one that are not recursive or reached through a “*” node. (E.g. author is inlined with book and monograph; monograph and editor are represented exactly once.)

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)

- It reduces number of joins but increases number of SQL queries.

Evaluation and Conclusion

- Qualitative Evaluation of Basic, Shared and Hybrid Tech
  - Using 37 DTDs from Roin Cover’s SGML/XML Web page
  - Metric: the average num of SQL joins required to process path expressions of a certain length.
- Evaluation Results
  - Basic tech ran out of virtual memory, too many relations!
  - Hybrid generally reduces the number of join per query(offset by an increase in the number of SQL queries required)
  - Hybrid vs. Shared: Hybrid tech more efficient in certain scenarios but heavily depends on the specific structure of the DTDs
- Summary
  - Potential advantages: leveraging established tech and high-performance system; Seamless XML and relational data queries.
  - Handle most queries on XML, barring certain types of complex recursion

Discussion (Group of 2)

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"

- Do you think this is a good idea? Why or why not?
- What could be a better choice?

Indexing XML Data Stored in a Relational Database

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

Microsoft Corporation

Background

- Trend: Increasing use of XML in enterprise applications
  - (i.e. Modeling data: semi-structured/unstructured/highly-variable structure/not known a *priori*)
- Shredding approach: generate XML from a set of tables based on an XML schema definition and to decompose XML instances into such tables.
  - DB uses the full power of the relational engine.
  - Suitable for a well-defined structured of XML data.
  - Difficulties:
    - XML data is hierarchical and may have recursive structures.
    - Relational data is unordered vs. XML has the document order.
    - Need large number of joins in the query processing. (Very very expensive)

Motivation

- XML Data Type
  - Introduced by Microsoft SQL Server 2005.
  - Stored XML values as large binary objects(BLOB).

```
Create  table DOCS (ID int primary key, XDOC xml)
```

- XQuery
  - Embedded within SQL statements.
  - Processes each XML instance at runtime.
  - Indexing XML instances to speed up queries.

Discussion (Group of 3)

We have seen two approaches in processing XML data:

- Decomposing XML into relational tables
- Storing XML as BLOBs with different indexing

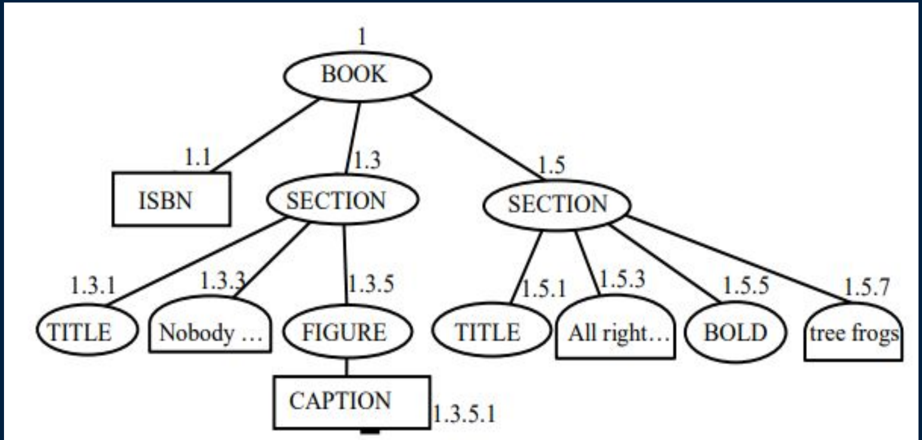
Can you come up with some use-cases where one would work better than the other?

Can these method be extend to other unstructured data formats (like JSON)?

Node Labeling

- ORDPATH
  - Mechanism for labeling nodes in an XML tree.
  - Preserves structural fidelity.
  - Allows insertion of nodes anywhere without re-labeling.
  - Independent of XML schemas typing XML instances.
  - Encodes the parent-child relationship by extending parent's ORDPATH with a labeling component for child.

ORDPATH Node Label



Discussion (Group of 2)

Only positive odd integers are assigned during an initial load; even-numbered and negative integer component values are reserved for later insertions into an existing tree.

- The authors leave all negative and even integers out from their numbering on the ORDPATH. Does this seem like enough? Too much?

Primary XML Index

- The B+ tree that that materializes the Infoset content of each XML instance in the XML column.
- Useful for query optimization but introduces redundancy.
- Index benefits from using the SQL type system.
- Optimizations (i.e.single-row storage for simple elements and prefix compression)

ORDPATH	TAG	NODE_ TYPE	VALUE	PATH_ ID
1	1 (BOOK)	1 (Element)	Null	#1
1.1	2 (ISBN)	2 (Attribute)	'1-55860-438-3'	#2#1
1.3	3 (SECTION)	1 (Element)	Null	#3#1
1.3.1	4 (TITLE)	1 (Element)	'Bad Bugs'	#4#3#1
1.3.3	10 (TEXT)	4 (Value)	'Nobody loves Bad bugs.'	#10#3#1
1.3.5	5 (FIGURE)	1 (Element)	Null	#5#3#1
1.3.5.1	6 (CAPTION)	2 (Attribute)	'Sample bug'	#6#3#1
1.5	3 (SECTION)	1 (Element)	Null	#3#1
1.5.1	4 (TITLE)	1 (Element)	'Tree frogs'	#4#3#1
1.5.3	10 (TEXT)	4 (Value)	'All right-thinking people'	#10#3#1
1.5.5	7 (BOLD)	1 (Element)	'love '	#7#3#1
1.5.7	10 (TEXT)	4 (Value)	'tree frogs'	#10#3#1

Query Compilation and Execution

- XQuery expressions are translated into relational operations on an Infoset table.
  - Identifying rows in the Infoset table that correspond to the elements specified in the XQuery expression.
  - Reassembling these rows into an XML result.

- Execute query by shredding XML blobs at runtime vs. to operate on XML indexes
  - Queries that retrieve the whole XML instance, its cheaper to retrieve the XML blobs.
  - Re-assembly cost outweighs the cost of parsing the XML blobs, then chose XML blobs.

Secondary XML indexes

- Created on the primary XML index to speed up different type of query:
  - PATH and PATH_VALUE
  - PROPERTY
  - VALUE
  - CONTENT
- Help with button-up evaluation
  - After the qualifying XML nodes have been found in the secondary XML indexes, a back join with the primary XML index enables continuation of query execution with those nodes.
  - This yields significant performance gains. (Can reduce the time and resources needed to execute complex queries.)

Secondary XML indexes

- PATH and PATH_VALUE
  - Helps evaluation of path expression.
  - Built on the columns PATH_ID, ID(primary key of the base table) and ORDPATH
  - The cost is relatively independent of the path length.
- PROPERTY
  - Property lookup for objects
  - (ID, PATH_ID, VALUE and ORDPATH)
- VALUE
  - Value-based queries.
  - (VALUE, PATH_ID, ID and ORDPATH)
- Content
  - Full text index
  - Word break

XML indexes’ Query performance

- XMark is an XML query benchmark that models an auction scenario.
  - 20 queries for testing different functionalities(i.e.exact match, ordered access, regular path expressions)
- Comparisons:
  - Primary XML index better in ordered access query but for reference chasing query is slower than the execution on XML blob.
  - PATH_VALUE index much faster for exact match query and gain large performance in regular path expression query.
  - PROPERTY index gains pronounced compared to the other XML index types in construction of complex result query.
  - VALUE index performs very well in exact match query.

Discussion (Group of 4)

We have read theory papers, method papers, and 10-year award papers.

This is the first industrial session paper we've read so far.

How is the focus of industrial session paper differ from others?

- Authorship
- Target Audience
- Content and Structure
- …

Conclusions

- Indexing XML instances stored in a relational database in an undecomposed form.
- B+ tree-based primary XML index
- Secondary indexes
- Performance measurements using the XMark benchmark