Relational Databases for Querying XML Documents: Limitations and Opportunities

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

Outline

• Motivation and Problem Definition
• Querying XML using a RDBMS
  – Physical schema design
  – Query mapping
  – Result construction
• Extensions to Relational Model
• Conclusion and Future Work
XML in One Slide

• Hierarchical document format for information exchange in WWW
• Self describing data (tags)
• Nested element structure having a root
• Element data can have
  – Attributes
  – Sub-elements

XML Example

```xml
<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>
```
What is the big deal about XML?

• Fast emerging as dominant standard for data representation on WWW

• Exciting database opportunity:
  – Unlike HTML, tags are not only for presentation
  – Can capture semantics
  – Can query the web if we can query XML!!!
Usage Scenario

Application/User Query over XML Documents

Query Engine

XML Document

XML Result (processed or displayed in browser)

XML Document

XML Document

XML Query Languages: XML-QL

WHERE <book>
  <booktitle> The Selfish Gene </booktitle>
  <author>
    <lastname> $l </lastname>
  </author>
</>
</> IN http://www.amazon.com/listings.xml,
http://www.barnesandnoble.com/books.xml

CONSTRUCT <lastname> $l </lastname>
Usage Scenario

Application/User Query over XML Documents → XML Result (processed or displayed in browser)

Query Engine


Key Requirement

• Presence of schema for XML documents
  – For issuing queries
  – For applications to interpret data
• Current proposals
  – Document Type Descriptors (DTDs)
  – With typing
    • Document Content Descriptors (DCDs)
    • XML Schemas
DTDs: Schema for XML Docs

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

• Given:
  – DTDs
  – Collection of XML documents conforming to DTDs
• Query:
  – Based on DTD schemas
  – Over collection of XML documents, performing selections, joins, etc.
  – Producing an XML result

What do we use to Query XML?

• XML a prime example of semi-structured data
• Invest in development of semi-structured query engines? [e.g., Lore]
What do we use to Query XML?

- XML a prime example of semi-structured data
- Invest in development of semi-structured query engines? [e.g., Lore]
  - Ignores three decades of research/development of relational model
  - Requires new tools for use with relational data

- Can we not leverage relational technology?
Outline

• Motivation and Problem Definition
• Querying XML using a RDBMS
  – Physical schema design
  – Query mapping
  – Result construction
• Extensions to Relational Model
• Conclusion and Future Work

Our Approach

Commercial RDBMS (DB2)

Automatic Translation Layer

Relational Schema

DTD

XML Documents

XML-QL Query

XML Result

Tuples

SQL Query

Relational Result

Translation Information
Scope of the Work

• Schema/Data mapping:
  – Automate storage of XML in RDBMS (Oracle’s iFS, DB2’s XML Extender)
• Query mapping:
  – Provide XML views of relational sources
• Result construction:
  – Export existing data as XML

Outline

• Motivation and Problem Definition
• Querying XML using a RDBMS
  – Physical schema design
  – Query mapping
  – Result construction
• Extensions to Relational Model
• Conclusion and Future Work
DTDs to Relations: Issues

- Complex DTD specifications
- Two level nature of relational schema (tuples and attributes) vs. arbitrary nesting of XML DTD
- Recursion

DTM Graph
**DTD to Relational Schema**

- Naïve Approach:
  - Each Element $\rightarrow$ Relation
  - Each Attribute of Element $\rightarrow$ Column of Relation
  - Connect elements using foreign keys
- Problem?
  - Fragmentation!

**Fragmentation: Example**

```
<!ELEMENT author (name, address)>
<!ATTLIST author id ID #REQUIRED>
<!ELEMENT name (firstname?, lastname)>  
<!ELEMENT firstname (#PCDATA)> 
<!ELEMENT lastname (#PCDATA)>  
<!ELEMENT address ANY>  
```

```
author (authorID: integer, id: string) 
name (nameID: integer, authorID: integer) 
firstname (firstnameID: integer, nameID: integer, value: string) 
lastname (lastnameID: integer, nameID: integer, value: string) 
address (addressID: integer, authorID: integer, value: string) 
```

- Results in 5 relations
- Just retrieving first and last names of an author requires three joins!
Shared Inlining Technique

• Intuition:
  – Inline as many sub-elements as possible
  – Do not inline only if it is a shared, recursive or set sub-element.

• Technique:
  – Necessary and Sufficient Condition for shared/recursive element:
    \[ \text{In-degree} \geq 2 \text{ in DTD graph} \]

Issues with Sharing Elements

• Parent of elements not fixed at schema level
• Need to store type and ids of parents (or if there are no parents)
  – parentCODE field (type of parent)
  – parentID field (id of parent)
  – Not foreign key relationship
Shared: Relational Schema


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)


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)

Shared Inlining Techniques: Pros

+ Reduces number of joins for queries like "get the first and last names of an author"
+ Efficient for queries such as "list all authors with name Jack"
Shared Inlining Technique: Cons

- Sharing whenever possible implies extra joins for path expressions
  - “Article with a given title name”

Hybrid Inlining Technique

- Inlines some elements that are shared in Shared
  - Elements with in-degree >= 2 that are not set sub-elements or recursive
- Handles set and recursive sub-elements as in Shared
### Hybrid: Relational Schema

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>article</td>
<td>(articleID: integer, article.contactauthor.isroot: boolean, article.contactauthor.authorid: string, article.title.isroot: boolean, article.title: string)</td>
</tr>
</tbody>
</table>

### Hybrid Inlining Technique: Pros

+ Reduces joins through shared elements (that are not set or recursive elements)
+ Shares some strengths of *Shared*:
  - Reduces joins for queries like “get first and last names of a book author”
Hybrid Inlining Technique: Cons

- Requires more SQL sub-queries to retrieve all authors with first name Jack
  - Tradeoff between reducing number of queries and reducing number of joins
  - *Shared* and *Hybrid* target query- and join-reduction respectively

Qualitative Evaluation

- 37 real DTDs from Oasis XML web page (http://www.oasis-open.org/cover/xml.html)
- Query set: All path expressions (that are valid in a given DTD) of a given length
- Metric: Number of joins in query
- Study benefit of inlining
- Also study tradeoff:
  - Number of joins per SQL query
  - Number of SQL queries
Path Expression with Length 3

Classification of DTDs

- Group 1: $J_{\text{hybrid}} < J_{\text{shared}}$, $Q_{\text{hybrid}} > Q_{\text{shared}}$, $TJ_{\text{hybrid}} < TJ_{\text{shared}}$
  - 35.14%

- Group 2: $J_{\text{hybrid}} < J_{\text{shared}}$, $Q_{\text{hybrid}} >> Q_{\text{shared}}$, $TJ_{\text{hybrid}} \sim TJ_{\text{shared}}$
  - 5.40%

- Group 3: $J_{\text{hybrid}} < J_{\text{shared}}$, $Q_{\text{hybrid}} >> Q_{\text{shared}}$, $TJ_{\text{hybrid}} > TJ_{\text{shared}}$
  - 16.22%

- Group 4: $J_{\text{hybrid}} \sim J_{\text{shared}}$, $Q_{\text{hybrid}} \sim Q_{\text{shared}}$, $TJ_{\text{hybrid}} \sim TJ_{\text{shared}}$
  - 43.24%
Evaluation

- *Shared* and *Hybrid* have pros and cons
- In many cases, *Shared* and *Hybrid* are nearly identical
  - Number of joins per SQL query ~ path length
  - Mainly due to large number of set nodes
  - Problem as join processing is expensive!

Outline

- Motivation and Problem Definition
- Querying XML using a RDBMS
  - Physical schema design
  - Query mapping
  - Result construction
- Extensions to Relational Model
- Conclusion and Future Work
Semi-structured Queries to SQL Queries

- Semi-structured queries have a lot more flexibility than SQL
- Allow path expressions with various operators and wild cards
- Discussion based on *Shared* approach

Queries with Simple Path Expressions: Example

```sql
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>
```
**Queries with Simple Path Expressions: Translation**

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”

**Queries with Recursive Path Expressions: Example**

WHERE <*.monograph>  
   <editor.(monograph.editor)*>  
       <name> $n </name>  
   </>  
   <title> Subclass Cirripedia </title>  
</> IN * CONFORMING TO pubs.dtd  
CONSTRUCT <result> $n </result>
Queries with Recursive Path Expressions: Translation

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

Queries with Arbitrary Path Expressions

WHERE <(article|monograph).$.name> $n </>
CONSTRUCT <name> $n </>

• Split complex path expression to (possibly many) simple recursive path expressions
• Has effect of splitting a single XML-QL/Lorel query to (possibly many) SQL queries
• Can handle nested recursive queries (though DB2 does not support it)
Outline

• Motivation and Problem Definition
• Querying XML using a RDBMS
  – Physical schema design
  – Query mapping
  – Result construction
• Extensions to Relational Model
• Conclusion and Future Work

Structuring Scenarios

• Simple structuring
• Tag variables
• Grouping
• Complex element construction
• Heterogeneity
• Nested queries
Simple Structuring: Query

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

CONSTRUCT <author>
    <firstname> $f </firstname>
    <lastname> $l </lastname>
</author>

Simple Structuring: Translation

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

<author>
    <firstname> Darwin </firstname>
    <lastname> </lastname>
</author>

(Richard, Dawkins)
(NULL, Darwin)
Grouping: Query

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

Grouping: Translation

<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>

(Darwin, book, Origin of Species)
(Darwin, book, Descent of Man)
(Darwin, monograph, Subclass Cirripedia)
(Dawkins, book, The Selfish Gene)
Complex Results: Query

• Assume article has multiple authors and titles

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

Complex Results: Translation

• Returning single table:
  – Results in Multi-Valued Dependencies

• Returning many tables:
  – Database functionality (outer join, optimization) duplicated outside

• Similar problem with recursive types

• Problem with Relational Model!
Outline

- Motivation and Problem Definition
- Querying XML using a RDBMS
  - Physical schema design
  - Query mapping
  - Result construction
- Extensions to Relational Model
- Conclusion and Future Work

Extensions to Relational Model

- Support for sets
- Untyped/variable typed references
- Information retrieval style indices
- Flexible comparison operators
- Multiple query optimization/execution
- Complex recursion
Support for Sets

• Set-valued attributes
  – Reduce fragmentation
• Nesting operators
  – Construct complex XML results

Untyped/Variable Typed References

• IDREFs not typed in XML
• Joins through IDREFs in relational model:
  – One join for every possible reference type
  – Proliferation of joins
• Better idea is to have database support
Information Retrieval Style
Indices

• Query “ANY” fields in a DTD
• Limited query support for XML fragments
  – Reduces fragmentation
• Example: Oracle’s ConText Search Engine, DB2’s Text Extender

Outline

• Motivation and Problem Definition
• Querying XML using a RDBMS
  – Physical schema design
  – Query mapping
  – Result construction
• Extensions to Relational Model
• Conclusion and Future Work
Conclusion

• Relational model can be used to query XML documents! But …
• Several limitations make model awkward or inefficient
• Extensions remove some of limitations
• Open Question:
  – Is it better to develop a specialized XML database or extend the relational model?

Future Research Directions

• Study impact of extensions
• Performance comparison with native XML approaches
• Exploit queryable XML sources:
  – May have high performance RDBMS “under wraps”
  – Push processing to queryable source
More Details?

- “Relational Databases for Querying XML Documents: Limitations and Opportunities”, VLDB’99

XML Query Languages: XQL

- Last name of the author of the book “The Selfish Gene”

book[booktitle = ‘The Selfish Gene’]/author/lastname
XML Query Languages: Lorel

SELECT X.author.lastname
FROM book X
WHERE X.booktitle = "The Selfish Gene"

Complexity of DTDs

• DTD element specifications can be of arbitrary complexity
• `<!ELEMENT a ((b|c|e)?,(e?|(f?,(b,b)*))*)>` is valid!
• Fortunately, can simplify DTD for translation purposes:
  – Key observations: not necessary to regenerate DTD from relational schema
  – XML query languages do not query over complex structures of elements
Simplification of DTDs

Flattening Transformations

\[(e_1, e_2)^* \rightarrow e_1^*, e_2^*
\]
\[(e_1, e_2)? \rightarrow e_1?, e_2?\]
\[(e_1|e_2) \rightarrow e_1?, e_2?\]

Simplification Transformations

\[e_1^{**} \rightarrow e_1^*
\]
\[e_1^*? \rightarrow e_1^*
\]
\[e_1^?* \rightarrow e_1^*
\]
\[e_1^?? \rightarrow e_1?
\]

Grouping Transformations

..., a*, ..., a*, ... \(\rightarrow\) a*, ...
..., a*, ..., a?, ..., a*, ... \(\rightarrow\) a*, ...
..., a?, ..., a*, ..., a?, ... \(\rightarrow\) a*, ...
..., a*, ..., a?, ..., a*, ... \(\rightarrow\) a*, ...
..., a, ..., a, ..., a*, ... \(\rightarrow\) a*, ...

Basic Inlining Technique

• Intuition:
  – Inline as many sub-elements as possible
  – Do not inline only if it is a set sub-element. Connect relations using foreign keys

• Complications:
  – A document can be rooted at any element
    • Create separate relational schema for each root
  – Recursion
    • Detect cycles in schema
Basic Relational Schema

- **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)
- **editor** (editorID: integer, editor.parentID: integer, editor.name: 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)

Basic Inlining Technique: Pros

- Reduces number of joins for queries like “get the first and last names of a book author”
- Efficient for queries such as “list all authors of books”
Basic Inlining Technique: Cons

• Queries like “list all authors with name Jack”
  – Union of 5 queries!
• Large number of relations:
  – Separate relational schema for each element as root (minor)
  – Unrolling recursive strongly connected components (major)

Experimental Results

• Is Basic practical?
  – Many DTD have recursion
  – Large strongly connected components
  – Schema translation program ran out of virtual memory!!!
• Concentrate on Shared vs. Hybrid
Varying Path Expression Length

Group 1 DTD

Group 3 DTD

Tag Variables: Query

WHERE <Sp>

  <author>
    <firstname> $f </firstname>
    <lastname> $l </lastname>
  </author>

</>

</> IN * CONFORMS TO pubs.dtd

CONSTRUCT <Sp>

  <author>
    <firstname> $f </firstname>
    <lastname> $l </lastname>

  </author>
</>


Tag Variables: Translation

Heterogeneous Results

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

- Rewritten as separate queries
- Results merged together
Nested Queries

• Very short answer:
  – Can be rewritten as SQL queries
  – Use outer joins to make connection

Flexible Comparison Operators

• Comparison between values of different types
• Problem present even with typed schemas
  – Different schemas may represent “comparable” information as different types
Multiple Query Optimization/Execution

- Complex path expressions translated to (possibly) many SQL queries
- SQL queries “share” scans, selections, joins, etc.
- More efficient to optimize and execute as a group

More Powerful Recursion

- Path expressions can be of arbitrary complexity
- Some systems support fixed-point operators (e.g., DB2)
- “Nested fixed-point” operators not supported
- Required for completeness