From XML to Relations: A Cost-Based Approach to XML Storage

Philip Bohannon
Juliana Freire*
Prasan Roy
Jerome Simeon
Bell Labs - Lucent Technologies
Managing XML Data

- XML has become the lingua-franca of the Web
- Applications are manipulating an increasing amount of XML data
  - Oasis.org - more than 400 different XML schemas
- XML is extensible and flexible: it can be used in applications with widely different requirements
- There is no one-size-fits-all solution for all applications
- How to store, query and publish XML data?
  - Need adaptable and flexible solutions
- LegoDB is a component-based XML data management system
  - The database-anywhere paradigm: portable and adaptable to any data and any environment
XML in One Slide

- W3C standard
- Hierarchical document format for information exchange
- Self-describing data: tags capture semantics
- XML document is a labeled tree: nested element structure having a root
- Element data can have
  - Attributes
  - Sub-elements
Sample XML Dataset: Internet Movie Database

```
<imdb>
  <show>
    <title>Fugitive, The</title>
    <year>1993</year>
    <review>
      <suntimes>
        <reviewer>Roger Ebert</reviewer>
        <rating>Two thumbs up!</rating>
        <comment>This is a fun action movie, Harrison Ford at his best.</comment>
      </suntimes>
    </review>
    <review>
      <nyt>The standard Hollywood summer movie strikes back.</nyt>
    </review>
    <box_office>183,752,965</box_office>
  </show>
  <show>
    <title>X Files, The</title>
    <year>1994</year>
    <seasons>4</seasons>
  </show>
</imdb>
```
XML, DTDs and XML Schema

**DTD**

```xml
<!ELEMENT imdb (show*, actor*)>
<!ELEMENT show(title, year?, reviews*, (box_office| seasons))>
<!ELEMENT title (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT review(#PCDATA)>
<!ELEMENT box_office (#PCDATA)>
<!ELEMENT seasons (#PCDATA)>
```

**XML Schema**

```xml
type IMDB = imdb [Show*, Actor*]

type Show = show[title[String], year[Integer]{0,1}, reviews[~[String]]*, (Movie| TV)]

type Movie = box_office[Integer] | TV:

type TV: seasons[Integer]
```

**Types**

- Basic types
- IMDB
- Show
- Title
- Year
- Review
- Box Office
- Seasons

**Cardinality**

- `*` (zero or more)
- `?` (zero or one)
- `+` (one or more)

**Wildcards**

- `~[Type]` (sequence of any type)

**Diagram**

```
  *  
 /  
*-- imdb
    |
    *  
     |
     *-- show
       |
       *-- title
       |
       ?   
       |
       *-- year
           |
           *-- reviews
               |
               *-- box_office
                   |
                   *-- seasons
                       |
                       ...  
```

**Types**

- `type IMDB = imdb [Show*, Actor*]`
- `type Show = show[title[String], year[Integer]{0,1}, reviews[~[String]]*, (Movie| TV)]`
- `type Movie = box_office[Integer]`
Querying XML

- Presence of schema for XML documents
  - For applications to interpret data
  - For issuing queries

Find the title, year and box office proceeds for all 2001 movies

For $v$ in document("imdbdata")/imdb/show
Where $v/year=2001$
Return $v/title, v/year, v/box_office
Storing XML

- How to store XML data?
  - Text files, native, object-oriented, relational, ...

- The right choice depends on application!

- For some applications it makes sense to use a relational database:
  - Leverage many years of development of relational technology, e.g., concurrency control/transactions, scalability, robustness, etc.
  - Integrate with existing data stored in an RDBMS

But storing and querying XML data in an RDBMS is a non-trivial task
XML and Relational Databases

- There is a mismatch between the relational model and that of XML
  - Relational: Normalized, flat and fragmented
  - XML: Un-normalized, nested and monolithic
- How to store XML data into relational tables?
  - Need to map the nested and irregular XML data into flat and regular tables
- How to evaluate XML queries over relational tables?
  - Need to map XQuery into SQL
Problem: Storing XML in RDMSSs

Storage Design:
- XML Schema
- XML Docs

Query Translation:
- XQuery Query
- XML results

Translation Layer:
- SQL Query
- Relational Result
- Tuples
- Relational Schema

Commercial RDBMS
Mapping an XML Schema into Tables

There are many alternative mappings!

(I) Inline as many elements as possible

(II) Partition reviews table—one for NYT, one for rest

(III) Split Show table into TV and Movies
A particular mapping is unlikely to be the best for all applications
The LegoDB Storage Mapping Engine

- **An optimization approach:**
  - automatically explores a space of possible mappings
  - selects the mapping which has the lowest cost for a given application

- **Important features:**
  - *Application-driven:* takes into account schema, data statistics and query workload
  - *Logical/physical independence:* interface is XML-based (XML Schema, XQuery, XML data statistics)
  - *Leverage existing technology:* XML standards; XML-specific operations for generating space of mappings; relational optimizer for evaluating configurations
How does LegoDB work?

XML Schema → Physical Schema Generation → Physical Schema Transformation → Good configuration

XML data statistics → Traditional Relational Query Optimizer

Physical schema → Relational schema, stats and workload

Query/Schema/Stats Translation

XQuery workload
P-Schema: Physical XML Schemas

- There is no straightforward mapping from XML Schema to relations
  
  ```
  type Show = show[... , reviews[^[String]]*, ...]
  ```

- XML Schema has no information about the data to be stored
  - `Cost` is important for storage
  - e.g., cardinality of collections, number of distinct values for attributes, etc.

- P-Schemas
  - are as expressive as XML schema
  - contain useful statistics about data to be stored
  - there is a fixed mapping from a p-schema type into a relation
Creating a P-Schema

- Ensure that each type name contains a structure that can be mapped into a relation

- Stratify XML Schema - modify the grammar for types in the XML Query Algebra
  - physical types, optional types

- Annotate schema with appropriate statistics
P-Schema: Formal Definition

scalar type  \( s ::= \) Integer \( | \) String \( | \) Boolean

physical scalar  \( ps ::= \) \( ps<\#size, \#min, \#max, \#distincts > \)
named type  \( nt ::= \) \( X\)

optional type  \( ot ::= \) \( nt\)

physical type  \( pt ::= \) \( nt\)

schema item  \( si ::= \) type \( X = pt\)

schema  \( ::= \) schema \( Sn = si, si, \ldots \) end
Mapping a P-Schema into Relations

- **Mapping follows the type stratification:**
  - Physical types are mapped into columns
  - Optional types are mapped into columns that may contain NULL values
  - Named types are mapped into tables and are used to keep track of parent-child relationship and generation of fkeys

- **Mapping Algorithm**

  Create a relation RT for each type name T

  For each relation RT
  - create a key
  - create a foreign key To_PT_Key to all relations RPT st PT is a parent type of T
  - create a column for each element associated with T
Mapping an XML-Schema into Relations: Example

**Original schema**

```
type Show = show [ 
    title [ String ],
    year[ Integer ],
    reviews[ String ]*,
    ... ]
```

**Stratified Schema**

```
type Show = show [ 
    title [ String ],
    year[ Integer ],
    Reviews*, ... ]
type Reviews = 
    reviews[ String ]
```

**P-Schema: Schema+statistics**

```
type Show = show [ 
    title [ String<#40,#1000> ],
    year[ Integer <#4,#1800,#2100,#300> ],
    Reviews*<#10>, ... ]
type Reviews = 
    reviews[ String<#800,#1000> ]
```

**Relational schema + stats**

```
TABLE Show( Show_id INT, title STRING, year INT )
TABLE Review ( Review_id, review String, parent_Show INT )
```
The Search Space: Transforming P-Schemas

- **Key idea:** A given document can be validated by different XML Schemas:
  - Different but equivalent regular expressions can be used to define an element
  - The presence or absence of a type name does not change the semantics of an XML Schema

- Applying transformations that manipulate the types (but preserve the element structure of schema) leads to a space of distinct relational configurations

- Define XML Schema transformations that
  - Exploit the structure of the schema, and
  - lead to *useful* relational configurations
Inlining/Outlining

- nest a type definition into its parent vs. associate a type name with an element

```
type TV = seasons[Integer], Description
type Description = description[String]
```

```
type TV = seasons[Integer],
description[String]
```

XML

```
TABLE TV (TV_id INT, seasons STRING, description STRING, parent_Show INT)
```

Relational

```
TABLE TV (TV_id INT, seasons STRING, description STRING, parent_Show INT)
```

Join TV and Description vs. Wider TV table
Union Distribution/Factorization

**XML**

```
type Show = show [
    title [String],
    year [Integer],
    (Movie | TV)]

type Movie = box_office[Integer]

type TV = seasons[Integer]
```

**Relational**

```
TABLE Show(Show_ID INT, type STRING, title STRING, year INT)

TABLE Movie(Movie_ID INT, box_office INT, parent_Show INT)

TABLE TV(TV_id INT, seasons INT, description STRING, parent_Show INT)
```

```
type Show = (Show1 | Show2)

type Show1 = show [title [String],
    year [Integer], box_office[Integer]]

type Show2 = show [title [String],
    year [Integer], seasons[Integer]]
```

Query attributes of Movies and TV shows together vs separately
Repetition Split

XML

Relational

Join vs. selection
Other transformations

- **Wildcard: materialize element names**

  type Reviews = reviews[~[String]]
  type Reviews = reviews[
    (NYTReviews|OtherReviews) ]
  type NYTReviews = nyt[String]
  type OtherReviews = (~!nyt)[String]

- **From union to options: inline elements of union**

  type Show = show[@type[String],
    title[String],
    year[Integer],
    (box_office[Integer]
    | seasons[Integer]) ]
  type Show = show[@type[String],
    title[String],
    year[Integer],
    box_office[Integer]?,
    seasons[Integer]?]
The translation module

- For each transformed p-schema need to generate a relational configuration
- P-schema $\Rightarrow$ Relations: fixed p-schema mapping
- XQuery $\Rightarrow$ SQL: path expressions to joins/selections
- XML stats $\Rightarrow$ Relational stats
  - Stats must be derived
  - Precise stats are crucial
  - proposed a new statistics model for XML (StatiX - SIGMOD’02)
Searching for a good configuration

- **Cost is key**: use a relational optimizer as a black box
  - Support different cost-models
  - Quality of selected configuration depends on the accuracy of the optimizer!
- **Set of possible configurations** that result from applying the rewritings is very large - possibly infinite!
- **How to search for the optimal solution?**
  - Our first approach: use a greedy search
  - Need to investigate alternative search strategies
Experimental settings

- LegoDB prototype implements: Physical schema creation, transformations and schema/query/stats translation
- Greedy search limited to inlining/outlining (other transformations tested separately)
- Used a Volcano-based relational optimizer (Roy et al, SIGMOD 2000)
- Data: Internet Movie Database (IMDB)
- Queries:
  - Lookup: interactive SPJ, e.g., Find alternate titles for a given show
  - Publish: document-oriented, e.g., List all shows and their reviews
Greedy Search

• Start all-inlined (greedy-si) vs start all outlined (greedy-so)
• Both strategies converge to similar costs and final configurations
• Final configurations have considerably lower costs
• Greedy-so converges faster than greedy-si for lookup queries
• Greedy-si converges faster for publish queries
Other experiments

- Sensitivity to variations in workload: how the resulting configuration performs if the workload changes
  - Create a spectrum of workloads that combine lookup and publish queries
  - Find the best configuration for each workload
  - For each configuration, evaluate its cost across the entire workload spectrum

- Effectiveness of XML Schema Transformations
  - Union distribution
  - Repetition Split
  - Wildcard
Sensitivity to variations in workload

All-inlined is 2-5 times worse than the configuration derived by LegoDB!
Effectiveness of Union Distribution

![Bar chart showing cost improvement for union queries. The chart compares the performance of union distribution and all inline methods. Each bar represents a query (Q4 to Q19), with the dark area indicating the cost improvement for union distribution.](chart.png)
Effectiveness of Repetition Split

(a) Lookup

(b) Publish
Effectiveness of Wildcard Xform

Find the NYT reviews for all shows produced after 1999

(Shows * Reviews) vs. (Shows * NYTReviews)

<table>
<thead>
<tr>
<th>Total reviews</th>
<th>10,000</th>
<th>100,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>NYT percent.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50%</td>
<td>5.42</td>
<td>6.3</td>
</tr>
<tr>
<td>25%</td>
<td>5.42</td>
<td>5.1</td>
</tr>
<tr>
<td>12.5%</td>
<td>5.42</td>
<td>4.4</td>
</tr>
</tbody>
</table>
Experiments: Summary

- **Greedy search is effective**
  - selects efficient mapping alternatives for a variety of workloads in a reasonable time
  - lead to reductions of over 50% in the running times of queries compared to previous mapping techniques

- **Inline everything is not always a good strategy!**
  - High cost for accessing *wide relations*

- **Selected configurations are robust to variations on workloads and are always superior than an all-inlined strategy**

- **XML transformations lead to configurations that have lower costs than if only inlining/outlining are considered**
Related Work

- **STORED** (Deutsch et al): looks at the data
  - maps schema-less data: find highly supported patterns
  - we may break-up these patterns into multiple relations if the result if more efficient for a given application

- **Basic, Shared, Hybrid** (Shanmugasundaram et al): DTD-based mapping
  - Fixed mapping techniques: inline as much as possible
  - Simplify DTDs - information is lost
  - We exploit the complex structures in a DTD/XML Schema

- **Many other proposals for fixed strategies**

- **LegoDB**: flexible mapping
  - considers data, schema, and query workload

- **User-defined mappings** (DB2, Oracle)
  - requires developers to understand XML and relational technology, hard to determine a good mapping--lots of choices
Conclusions

- Novel cost-based approach for generating relational storage mappings for XML data
  - takes application characteristics into account (schema + data + queries)
  - generates configurations that had not been explored before
- We implemented a system and our performance study indicates that storage mappings of significantly improved quality can be found
- Transforming XML Schema vs. relations:
  - XML Schema is more expressive: some transformations require integrity constraints that are beyond what can be expressed in relational databases
  - Extensibility: LegoDB framework can be applied to other storage models