

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

```

<!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)>
    
```

Types

Basic types

XML Schema

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

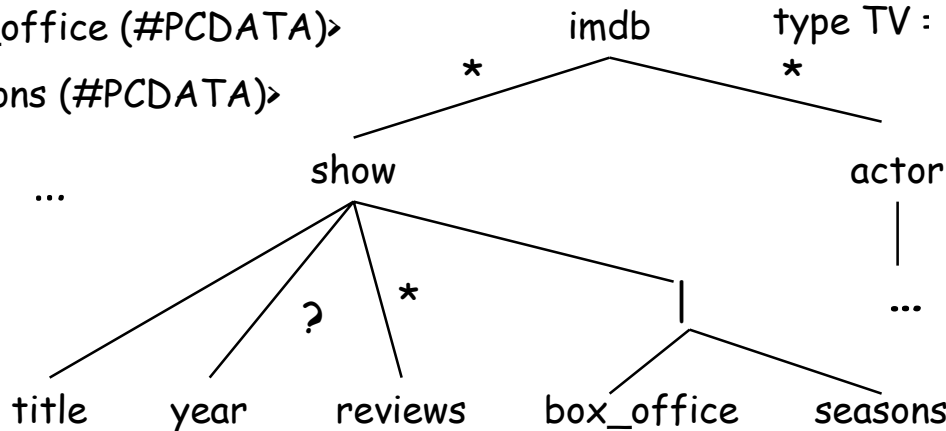
```

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

```
type Movie = box_office[Integer]
```

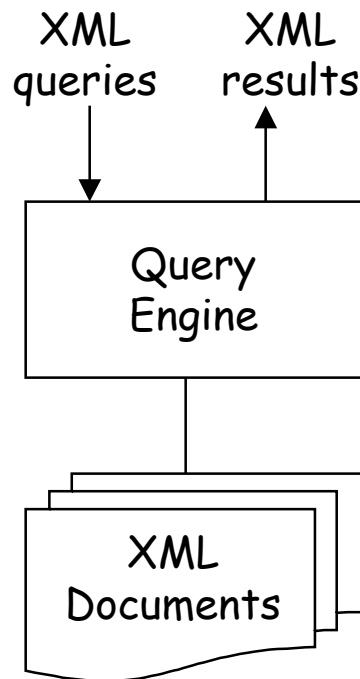
```
type TV : Wildcards r]
```

Cardinality



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

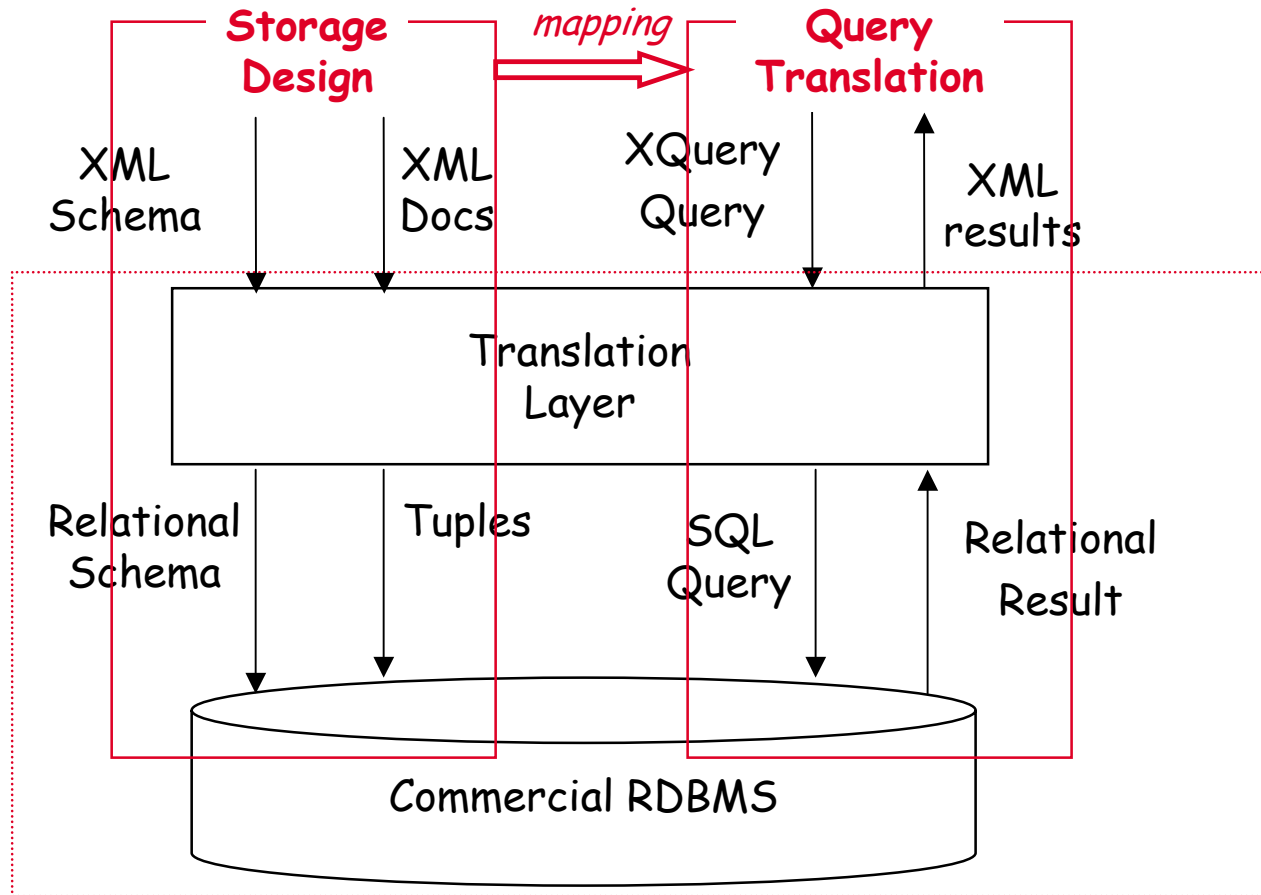
XML and Relational Databases

- ◆ 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 RDBMSs



Lucent Technologies
Bell Labs Innovation



Mapping an XML Schema into Tables

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

...

TABLE Show
(show_id INT,
title STRING,
year INT,
box_office INT,
seasons INT)

TABLE Review
(review_id INT,
tilde STRING,
review STRING,
parent_Show INT)

TABLE Show
(show_id INT,
title STRING,
year INT,
box_office INT,
seasons INT)

TABLE NYTReview
(review_id INT,
review STRING,
parent_Show INT)

TABLE Review
(review_id INT,
tilde STRING,
review STRING,
parent_Show INT)

TABLE Show1
(show1_id INT,
title STRING,
year INT,
box_office INT)

TABLE Show2
(show2_id INT,
title STRING,
year INT,
seasons INT)

TABLE Review
(review_id INT,
tilde STRING,
review STRING,
parent_Show INT)

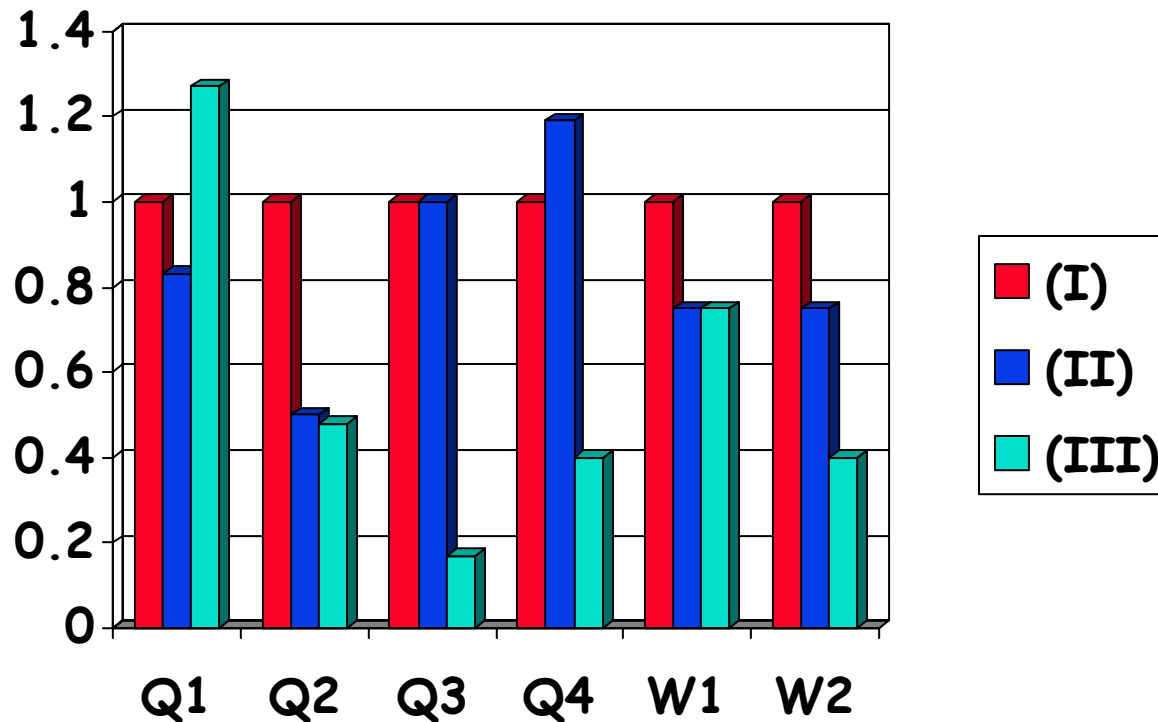
There are many alternative mappings!

(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

Mappings and Performance Implications



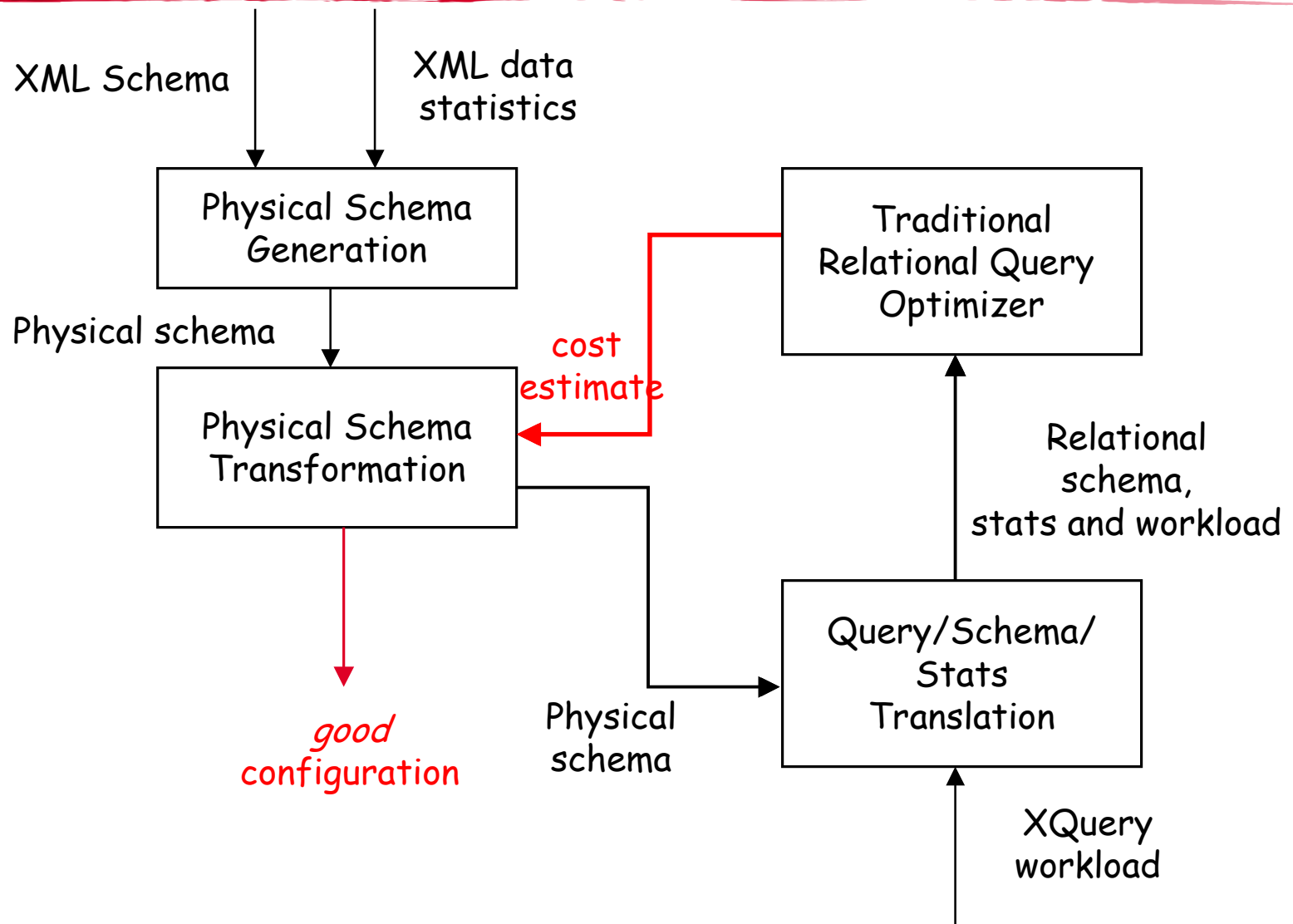
A particular mapping is unlikely to be the best for all applications

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?



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	<i>s</i>	::= Integer String Boolean	
→ physical scalar	<i>ps</i>	::= <i>ps</i> < #size, #min, #max, #distincts >	
named type	<i>nt</i>	::= <i>X</i>	type name
		<i>nt</i> <i>nt</i>	choice
		∅	empty choice
		<i>nt</i> { <i>n, m</i> } < #count >	repetition
→ optional type	<i>ot</i>	::= <i>nt</i>	named type
		<i>s</i>	optional scalar
		<i>l</i> [<i>ot</i>]	optional element
		<i>ot</i> , <i>ot</i>	optional sequence
		()	empty sequence
→ physical type	<i>pt</i>	::= <i>nt</i>	named type
		<i>ot</i> {0, 1}	optional type
		<i>s</i>	scalar
		<i>l</i> [<i>pt</i>]	element
		<i>pt</i> , <i>pt</i>	sequence
		()	empty sequence
schema item	<i>si</i>	::= type <i>X</i> = <i>pt</i>	type declaration
schema		::= schema <i>S</i> _{<i>n</i>} = <i>si</i> , <i>si</i> , ... end	schema

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



Lucent Technologies
Bell Labs Innovation

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

Original schema

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

type Reviews =
  reviews[ String ]
```

Stratified Schema

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

type Reviews =
  reviews[ String<#800,#1000> ]
```

P-Schema: Schema+statistics

```
TABLE Show( Show_id INT,
title STRING, year INT )

TABLE Review ( Review_id,
review String,
parent_Show INT )
```

Relational schema + *stats*

The Search Space: Transforming P-Schemas

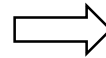
- ◆ **Key idea:** A given document can be validated by different XML Schemas:
 - Different but equivalent regular expressions can be used to define an element
 - The presence or absence of a type name does not change the semantics of an XML Schema
- ◆ Applying transformations that manipulate the types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

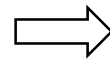


XML

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

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

```
TABLE Description (Description_id INT,
description STRING, parent_TV INT)
```



Relational

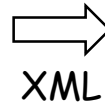
```
TABLE TV (TV_id INT,
seasons STRING, description
STRING, parent_Show INT)
```

Join TV and Description vs. Wider TV table

Union Distribution/Factorization



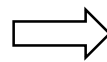
```
type Show = show [
  title [ String ],
  year [ Integer ],
  (Movie|TV) ]
type Movie = box_office [ Integer ]
type TV = seasons [ Integer ]
```



```
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

```
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)
```



Relational

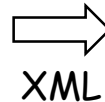
```
TABLE Show1 (Show_ID INT,
  type STRING, title STRING,
  year INT, box_office INT)
TABLE Show2 (Show_ID INT,
  type STRING, title STRING,
  year INT, seasons INT)
```

Repetition Split



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

```
type Reviews = reviews[ String ]
```

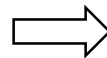


```
type Show = show [ @type[ String ],  
                  title [ String ],  
                  year [ Integer ],  
                  reviews[ String ]? ,  
                  Reviews* ,  
                  ... ]
```

```
type Reviews = reviews[ String ]
```

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

```
TABLE Reviews(Reviews_ID  
             INT, reviews STRING,  
             parent_Show INT)
```



Relational

```
TABLE Show(Show_ID INT,  
           type STRING, title STRING,  
           year INT, reviews[ String ])
```

```
TABLE Reviews(Reviews_ID  
             INT, reviews STRING,  
             parent_Show INT)
```

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

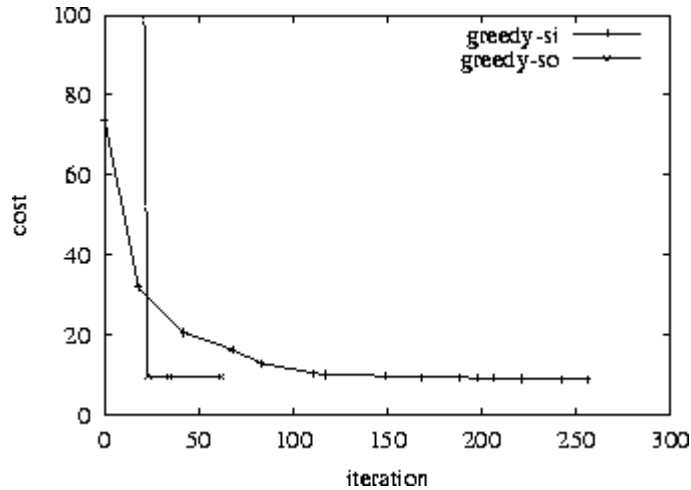
Experimental settings

- ◆ 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

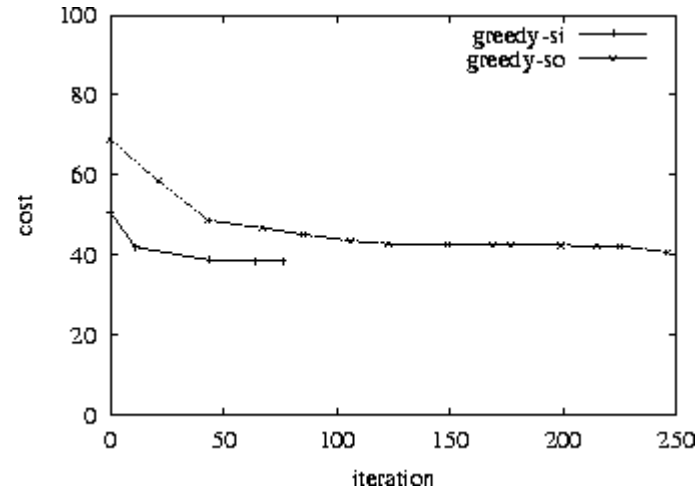
Greedy Search



Lookup



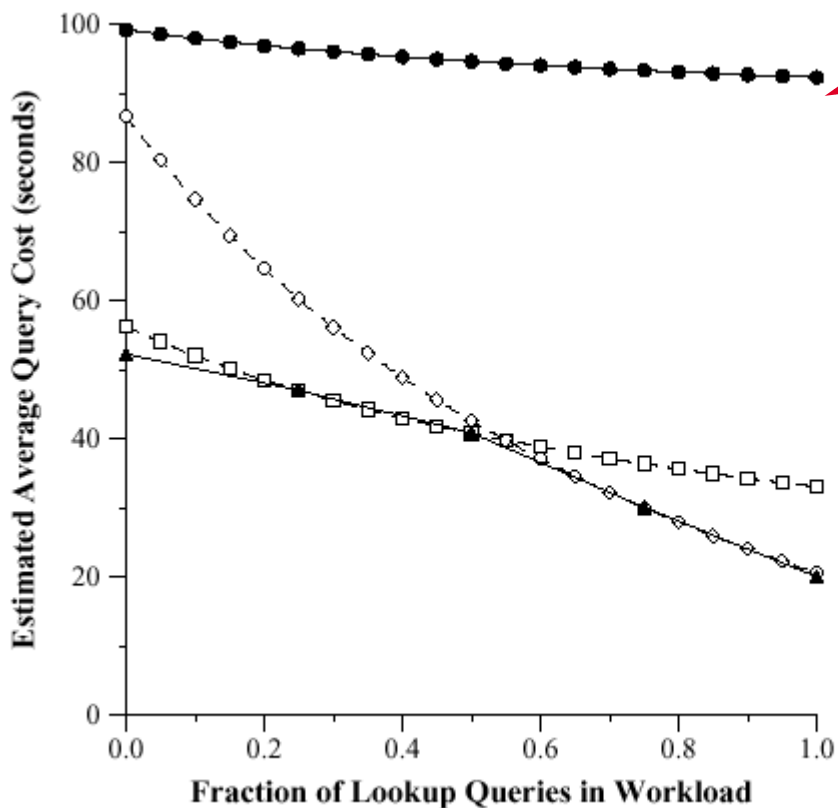
Publish



- 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

- ◆ 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

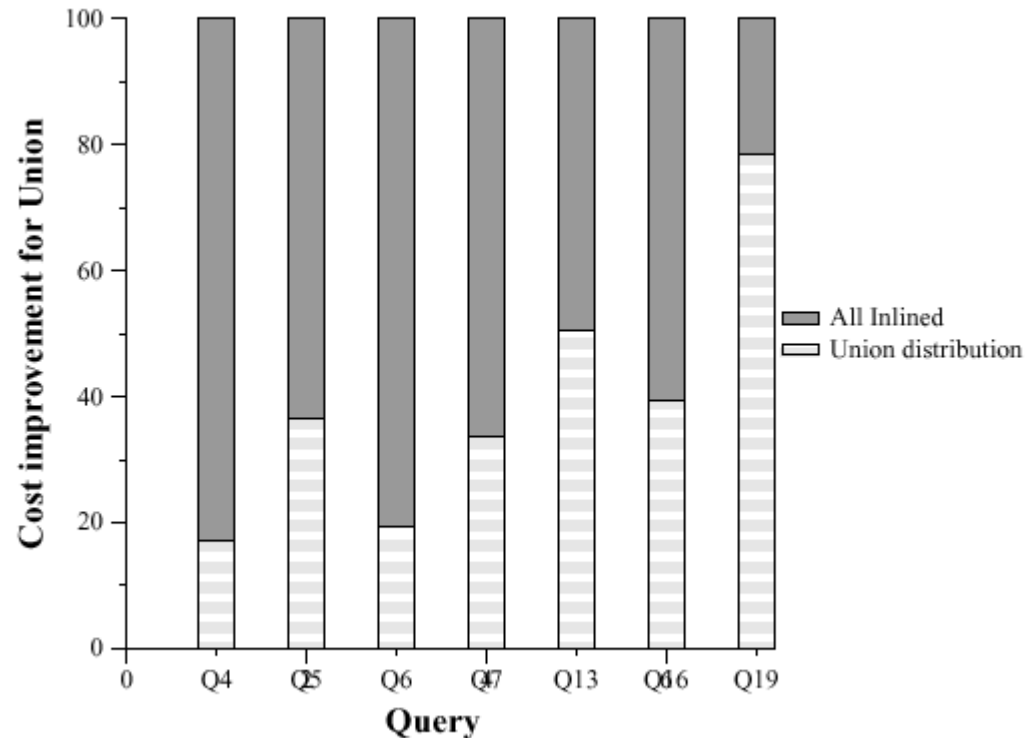
Sensitivity to variations in workload



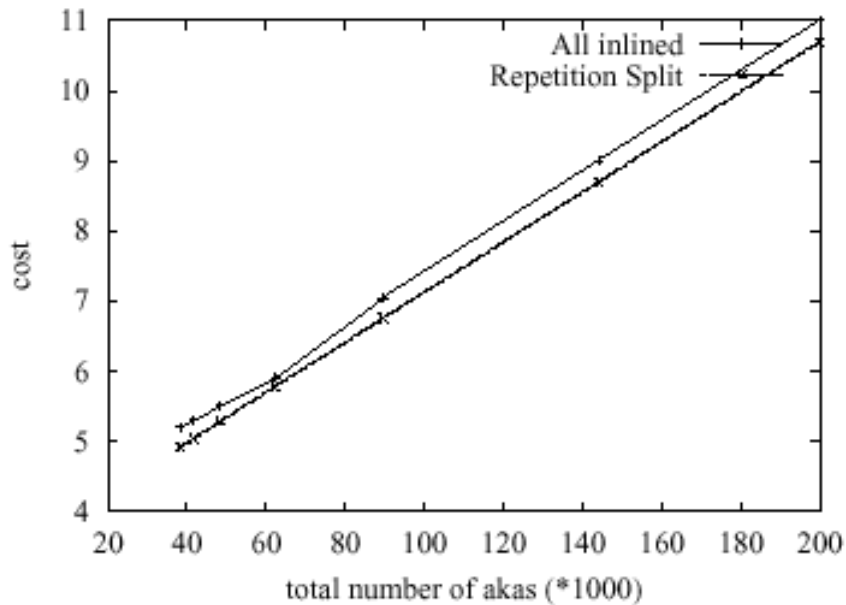
All-inlined is 2-5 times worse than the configuration derived by LegoDB!

- C[All-INLINED]
- C[0.25]
- ◇ C[0.75]
- ▲ OPT

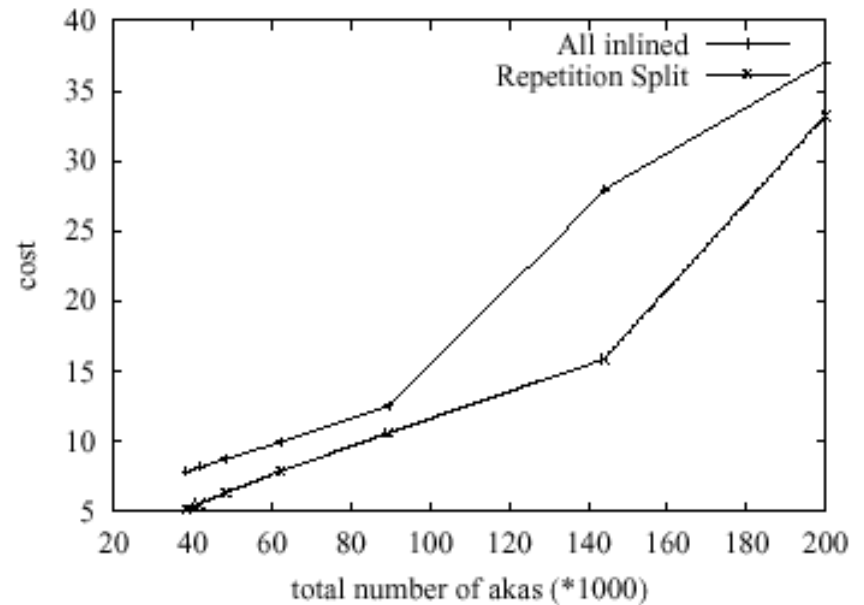
Effectiveness of Union Distribution



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)

Total reviews	10,000		100,000	
NYT percent.	Q1 (AI)	Q1 (W)	Q2(AI)	Q2(W)
50%	5.42	6.3	48	26.3
25%	5.42	5.1	48	15
12.5%	5.42	4.4	48	9.4

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 is 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