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

Philip Bohannon Juliana Freire Prasan Roy Jérôme Siméon
Bell Laboratories
{bohannon,juliana,prasan,simeon}@research.bell-labs.com

Abstract

As Web applications manipulate an increasing amount of XML, there is a growing interest in storing XML data in relational databases. Due to the mismatch between the complexity of XML's tree structure and the simplicity of flat relational tables, there are many ways to store the same document in an RDBMS, and a number of heuristic techniques have been proposed. These techniques typically define fixed mappings and do not take application characteristics into account. However, a fixed mapping is unlikely to work well for all possible applications. In contrast, LegoDB is a cost-based *XML storage mapping engine that explores a space of possible XML-to-relational mappings and selects the best mapping for a given application. LegoDB leverages current XML and relational technologies: 1) it models the target application with an XML Schema, XML data statistics, and an XQuery workload; 2) the space of configurations is generated through XML-Schema rewritings; and 3) the best among the derived configurations is selected using cost estimates obtained through a standard relational optimizer. In this paper, we describe the LegoDB storage engine and provide experimental results that demonstrate the effectiveness of this approach.*

1 Introduction

XML has become an important medium for representing, exchanging and accessing data over the Internet. As applications are processing an increasing amount of XML, there is a growing interest in storing XML data in relational databases so that these applications can use a complete set of data management services (including concurrency control, crash recovery, and scalability) and benefit from the highly optimized relational query processors. Due to the mismatch between the complexity of XML's tree structure and the simplicity of flat relational tables, there are many ways to store the same document in an RDBMS, and a number of fixed heuristic XML-to-relational mapping strategies have been proposed [7, 10, 13, 17, 18]. However, a fixed mapping is unlikely to work well for all of the possible access patterns different applications may present. For example, a Web site may perform a large volume of simple lookup queries, whereas a catalog printing application may require large and complex queries with deeply nested results. On the other hand, recent versions of commercial RDBMSs (see *e.g.,* [22]) allow the developer to specify their own XML to relational mapping. Although more flexible, this approach requires development effort, and the mastering of two complex technologies (XML and RDBMS). Moreover, it might be hard, even for an expert, to determine a good mapping for a complex application. In this paper, we introduce a novel cost-based approach to XML storage design. We describe the design and implementation of LegoDB, an XML storage mapping system based on this approach, that automatically finds an efficient relational configuration for a target XML application.

The three main design principles behind LegoDB are: cost-based search, logical/physical independence, and reuse of existing technology. Since the effectiveness of a one-size-fits-all mapping is improbable given the wide variety of XML applications (with data ranging from flat to nested, schemas ranging from structured to semistructured, and access patterns ranging from traditional SPJ queries to full-text or recursive queries), our first principle is to take the application into account. More precisely, given some parameters describing the target XML application, the LegoDB engine explores various relational configurations in order to find the most efficient for the target application. Our second principle is to support logical/physical independence. Developers of XML applications should deal with XML structures and queries, and should not be concerned with the physical storage. The LegoDB interface is purely XML-based, and isolates the developer from the underlying storage engine—in our case, relational. Our third principle is to leverage existing XML and relational technologies whenever possible. LegoDB relies on: 1) XML Schema and XQuery to model the target application, 2) XML Schema rewritings to generate a search space of storage mappings,

and 3) a traditional relational optimizer to obtain cost estimates for these mappings. The paper makes the following contributions:

- We introduce the notion of *physical XML Schemas* (p-schemas) which extend XML Schemas in two significant ways: they contain data statistics and they can be easily mapped into relational tables. We define a fixed mapping from *p-schemas* to relational configurations, and the corresponding mapping from XML documents to databases (Section 3).
- We propose a set of *p-schema* rewritings: when successively applied to a *p-schema*, these rewritings lead to a space of alternative storage configurations. Because the proposed rewritings exploit XML Schema structures, the resulting search space contains new storage configurations that have not be explored by previous approaches (Section 4).
- We use a relational optimizer to obtain cost estimates for each storage configuration. For a given *p-schema*, we map XML data statistics into relational statistics, and an XQuery workload into SQL to provide as inputs to the optimizer.
- Due to the nature of XML Schema, *p-schema* transformations may lead to a large (possibly infinite) search space. We present a simple greedy evaluation strategy that explores an interesting subset of this space (Section 4.2).
- We give experimental results which show that LegoDB is able to find efficient storage designs for a variety of workloads in a reasonable time. Our results indicate that our cost-based exploration selects storage designs which would not be arrived at by previously-proposed heuristics, and that in most cases, these designs have significantly lower costs (Section 5).

Our goal is to cover the main components of the mapping engine, but to keep with space limitations, discussions on statistics and query translation are omitted.

2 LegoDB Approach and Architecture

We motivate our approach with an XML application scenario inspired from the Internet Movie Database (IMDB) [12]. Figure 1(a) shows a Document Type Definition (DTD) [3] for a subset of the IMDB information. The IMDB DTD contains a collection of shows, movie directors and actors. Each show can be either a movie or a TV show. Movies and TV shows share some elements (*e.g.,* `title` and `year` of production), but there are also elements that are specific to each show type (*e.g.,* only movies have a `box_office`, and only TV shows have `seasons`). Figure 1(b) shows an XML Schema description of the IMDB data written in the type syntax of the XML Query Algebra [8].[1] A full description of DTD, XML Schema and XML Query Algebra types for the IMDB scenario, as well as a sample document, can be found in the full version of the paper [2].

DTDs vs. XML Schema Like the DTD, an XML Schema describes elements (*e.g.,* `show`) and attributes (*e.g.,* `@type`), and uses regular expressions to describe allowed subelements (*e.g.,* `imdb` contains `Show*`, `Director*`, `Actor*`). XML Schema also has specific features that are useful for storage. First, one can specify precise data types (*e.g.,* `String`, `Integer`) instead of just text. Also, regular expressions are extended with more precise cardinality annotations for collections (*e.g.,* {1,10} indicates that there can be between 1 to 10 `aka` elements for `show`). Finally, XML Schema supports *wildcards*: for instance, `~[AnyType]` indicates that the `review` element can contain an element with arbitrary name and content. As a result, XML Schema can specify parts of a schema for which no precise structural information is available.

XML Schema, p-schema, and storage mapping Another important difference between XML Schema and DTDs is that the former distinguishes between *elements* (*e.g.,* a `show` element) and their *types* (*e.g.,* the `Show` type). The type name never appears in the document, but can be used to classify nodes in the XML tree. LegoDB uses this classification as the basis for storage mappings. Figures 2(b) and (c) illustrate a simple XML Schema to relational mapping. Each XML Schema type groups a set of elements and attributes together. The LegoDB mapping engine creates a table for each such type (*e.g.,* `Show`) and maps the contents of the elements (*e.g.,* `type`, `title`) into columns of that table. Finally, the mapping generates a key column that contains the `id` of the corresponding element (*e.g.,* `Show_id` column), and a foreign key that keeps track of the parent-child relationship (*e.g.,* `parent_Show` column). Clearly, it is not always possible to map types into relations. For instance, the schema in Figure 2(a), although equivalent to that of Figure 2(b), indicates that the type `Show` can contain many `review` elements. These elements cannot be directly mapped into one column of a table. In Section 3, we introduce *physical schemas* as the subset of XML Schemas that can be directly mapped into relations.

Schema transformations There are many different XML Schemas that describe the exact same set of documents: different regular expressions (*e.g.,* `(a(b|c*))` `((a,b)|(a,c*))`) can be used to describe the same element content; and the children of an element can be *directly included* (*e.g.,* `title` in `Show`) or can be *referred to* through a type name (*e.g.,* see the type `Year`). As our

[1] This notation captures the core semantics of XML Schema, abstracting away some of the complex features of XML Schema which are not relevant for our purposes (*e.g.,* the distinction between groups and complexTypes, local vs. global declarations, etc).

```
<!DOCTYPE imdb [
  <!ELEMENT imdb (show*, director*, actor*)>

  <!ELEMENT show
     (title, year, aka+, reviews*,
      ((box_office, video_sales)
       |(seasons, description, episode*)))>
  <!ATTLIST show
            type CDATA #REQUIRED>

  <!ELEMENT title (#PCDATA)>
  <!ELEMENT year (#PCDATA)>
  <!ELEMENT aka (#PCDATA)>
  <!ELEMENT review (#PCDATA)>

  <!ELEMENT box_office (#PCDATA))>
  <!ELEMENT video_sales (#PCDATA))>

  <!ELEMENT seasons (#PCDATA))>
  <!ELEMENT description (#PCDATA))>
  <!ELEMENT episode (name,guest_director)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT guest_director (#PCDATA)>
]>
```

(a)

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

type Show =
  show [ @type[ String ],
  title[ String ],
  Year, Aka{1,10}, Review*,
  (Movie | TV) ]

type Year = year[ Integer ]
type Aka = aka[ String ]
type Review = review[ ~ [ String ] ]

type Movie =
  box_office[ Integer ],
  video_sales[ Integer ]

type TV =
  seasons[ Integer ],
  description[ String ],
  episode[ name[ String ],
    guest_director[ String ]]*

...
```

(b)

Figure 1. DTD and XML Schema for the IMDB documents

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

(a) Initial XML Schema

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

type Reviews =
  reviews[ String ]
```

(b) *P-schema*

```
TABLE Show
      ( Show_id INT,
        type STRING,
        title STRING,
        year INT )
TABLE Review
      ( Review_id,
        review String,
        parent_Show INT )
```

(c) Relational confi guration

Figure 2. XML Schema, p-schema, and relational configuration

```
TABLE Show
   ( Show_id INT,
     type STRING,
     title STRING,
     year INT,
     box_office INT,
     video_sales INT,
     seasons INT,
     description STRING )

TABLE Review
   ( Reviews_id INT,
     tilde STRING,
     reviews STRING,
     parent_Show INT )

....
```

(a)

```
TABLE Show
   ( Show_id INT,
     type STRING,
     title STRING,
     year INT,
     box_office INT,
     video_sales INT,
     seasons INT,
     description STRING )

TABLE NYT_Reviews
   ( Reviews_id INT,
     review STRING,
     parent_Show INT )

TABLE Reviews
   ( Reviews_id INT,
     tilde STRING,
     review STRING,
     parent_Show INT )

.....
```

(b)

```
TABLE Show_Part1
   ( Show_Part1_id INT,
     type STRING,
     title STRING,
     year INT,
     box_office INT,
     video_sales INT )

TABLE Show_Part2
   ( Show_Part2_id INT,
     type STRING,
     title STRING,
     year INT,
     seasons INT,
     description STRING )

TABLE Reviews
   ( Reviews_id INT,
     tilde STRING,
     review STRING,
     parent_Show INT )

....
```

(c)

Figure 3. Three storage mappings for shows

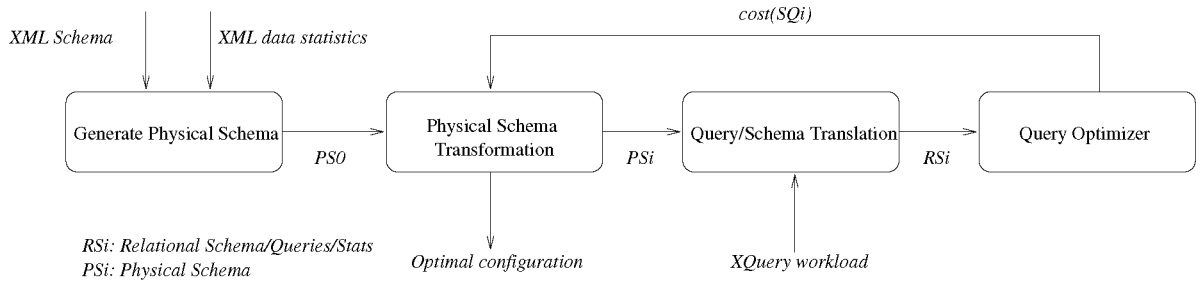


Figure 4. Architecture of the Mapping Engine

mapping generates one relation for each type, the presence or absence of type names affects the resulting relational schema. In Section 4, we define a space of storage configurations by introducing rewritings that preserve the semantics of the schema, but yield different storage configurations.

Cost-based evaluation of XML storage Figure 3 shows three possible relational storage mappings that are generated, from the schema in Figure 1, by our transformations. Configuration (a) results from inlining as many elements as possible in a given table, roughly corresponding to the strategies presented in [18]. Configuration (b) is obtained from configuration (a) by partitioning the `Reviews` table into two tables: one that contains New York Times reviews, and another for reviews from other sources. Finally, configuration (c) is obtained from configuration (a) by splitting the `Show` (`Show_Part1`) or TV shows (`Show_Part2`). Even though each of these configurations can be the best for a given application, there are cases where they perform poorly. The key remark that justifies the LegoDB approach is that one cannot decide which of these configurations will perform well without taking the application (*i.e.,* a query workload and data statistics) into account.

For instance, consider the following XQuery [5] queries:

Q1:
```
FOR  $v in imdb/show
WHERE $v/year = 1999
RETURN ($v/title, $v/year,
        $v/nyt_reviews)
```

Q2:
```
FOR $v in imdb/show
RETURN $v
```

Q3:
```
FOR $v in imdb/show
WHERE $v/title = c3
RETURN $v/description
```

Q4:
```
FOR $v in imdb/show
RETURN
<result>
 { $v/title,
   $v/year,
   (FOR $e IN $v/episode
    WHERE
      $e/guest_director = c4
   RETURN $e) }
</result>
```

Query Q1 returns the title, year, and the New York Times reviews for all 1999 shows and query Q2 publishes all the information available for all shows in the database. Queries 1 and 2 are typical of a publishing scenario (*i.e.,* to send a movie catalog to an interested partner). Query Q3 retrieves the description of a show based on the title, and query Q4 retrieves episodes of shows directed by a particular guest director c4. Queries 3 and 4 contain selection criteria and are typical of interactive lookup queries, such as the ones issued against the IMDB Web site itself. We then define two workloads, Publish and Lookup, where Publish $= \{Q1 : 0.4, Q2 : 0.4, Q3 : 0.1, Q4 : 0.1\}$ and Lookup $= \{Q1 : 0.1, Q2 : 0.1, Q3 : 0.4, Q4 : 0.4\}$, where each workload contains a set of queries and an associated weight that reflects the importance of each query for the application. The following table shows the cost, as estimated by the LegoDB optimizer, for each query and workload, when run against the storage configurations shown in Figure 3. (These costs are normalized by the costs of Storage Map 1.)

	Storage Map 1 (Fig 3(a))	Storage Map 2 (Fig 3(b))	Storage Map 3 (Fig 3(c))
Q1	1.00	0.83	1.27
Q2	1.00	0.50	0.48
Q3	1.00	1.00	0.17
Q4	1.00	1.19	0.40
Publish	1.00	0.75	0.75
Lookup	1.00	1.01	0.40

It is important to remark that only the first one of the three storage mappings shown in Figure 3 can be generated by previous heuristic approaches (of which we are aware). However, this mapping has significant disadvantages for the workloads we considered. First, due to its treatment of union, it inlines several fields which are not present in all the data, making the `Show` relation wider than necessary. Second, when the entire `Show` relation is exported as a single document, the records corresponding to movies need not be joined with the `Episode` tables, but this join is required by mappings 3(a) and (b). Finally, the (potentially large) `Description` element need not be inlined unless it is frequently queried.

LegoDB architecture The architecture of the LegoDB mapping engine is depicted in Figure 4. Given an XML Schema and statistics extracted from an example XML dataset, we first generate an initial physical schema (PS0). This physical schema and the XQuery workload are then input into the Query/Schema Translation module, which in turn generates the corresponding relational catalog (schema and statistics) and SQL queries that are input into a relational optimizer for cost estimation. Schema transformation operations are then repeatedly applied to PS0, and the

process of Schema/Query translation and cost estimation is repeated for each transformed PS until a *good* configuration is found.

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

Figure 5. Stratified XML Schema Types

3 From XML Schema to Relations

Physical XML Schemas We now introduce the notion of physical XML Schema (*p-schema*). *P-schemas* have the following properties: (i) they are based on XML Schema, (ii) they contain statistics about the XML data to be stored, and (iii) there exists a fixed, simple mapping from *p-schemas* into relational configurations. As we have seen in the previous section, not all XML Schemas can be easily mapped into relations. However, by inserting appropriate type names for certain elements, one can satisfy condition (iii) above, while preserving the semantics of the original schema. The `Show` type of Figure 2(a) cannot be stored directly into a relational schema because there might be multiple `reviews` elements in the data. However, the equivalent schema in Figure 2(b), where a separate type name exists for that element, can be easily mapped into the relational schema shown in Figure 2(c). The *p-schema* also stores data statistics. These statistics are extracted from the data and inserted in the original physical schema PS0 during its creation. A sample *p-schema* with statistics for the type `Show` is given below:

```
type Show =
  show [ @type[ String<#8,#2> ],
         year[ Integer<#4,#1800,#2100,#300> ],
         title[ String<#50,#34798> ],
         Review*<#10> ]

type Review =
   review[ String<#800> ]
```

The notation `Scalar<#size,#min,#max,#distincts>` indicates for each XML Schema datatype the corresponding size (*e.g.,* 4 bytes for an integer), minimum and maximum values, and the number of distinct values. The notation `String<#size,#distincts>` indicates the length of a string as well as the number of distinct values. The notation `*<#count>` indicates the *relative* number of `reviews` elements within each element of type `Show` (*e.g.,* in this example, there are 10 reviews per show).

Stratified physical types The main difficulty in defining *p-schemas* is to make sure the type structures allow an easy mapping into relations. For that purpose, we introduce the notion of stratified physical types, adapted from the original syntax for types of [8]. The grammar for stratified physical types is shown on Figure 5. This grammar contains three different productions for types: named, optional, and physical types). Each production refers to the previous one, ensuring that type names are always used within collections or unions in the schema. Physical types contain only singleton elements, nested singleton elements, and optional types. Optional types are used to represent optional nested elements. Finally, named types only contain named types and ensure that complex regular expressions (such as union and repetition) do not contain nested elements.

Mapping p-schemas into relations Assuming the above stratified types, mapping a *p-schema* into relations is now straightforward:

- Create one relation R_T for each type name T.
- For each relation R_T, create a key that will store the node id of the corresponding element.
- For each relation R_T, create a foreign key `To_`PT`_Key` to all relations R_{PT} such that PT is a parent type of T
- A column is created in R_T for each sub-element of T that is a physical type.
- If the data type is contained within an optional type then the corresponding column can contain a null value.

The mapping procedure follows the type stratification: elements in the physical type layer are mapped to columns, elements within the optional types layer are mapped to columns that allow null values, and named types are used only to keep track of the child-parent relationship and for the generation of foreign keys. For a given *p-schema* ps, the relational schema defined by the above mapping is referred to as $rel(ps)$. A detailed definition of the mapping of *p-schemas* into relations is given in [2].

It is noteworthy to mention that this mapping deals with recursive types, and maps XML Schema wildcards (*i.e.,*~ elements) appropriately. Take for example the definition of the **AnyElement** in the XML Query Algebra:

```
type AnyElement = ~[ (AnyElement|AnyScalar)* ]
type AnyScalar  = String | Integer | ...
```

This type is valid for all possible elements with any content. In other words, this is a type for untyped XML documents. Note also that this definition uses both recursive types (`AnyElement` is used in the content of any element) and a wildcard (~). Again, applying the above rules, one would construct the following relational schema:

```
TABLE String       ...  TABLE AnyElement =
 ( __data STRING,  ...   ( Element_id INT,
   parent INT )    ...     tilde STRING,
                           parent_Element INT )
```

This also shows that using XML Schema and the proposed mapping, LegoDB can deal with structured and semistructured documents in an homogeneous way. Indeed the `AnyElement` table is similar to the *overflow* relation that was used to deal with semistructured document in the STORED system [7].

Mapping XQuery queries Although query mapping is an important part of the optimization process, rewriting XML queries into their equivalent SQL counterparts is not the focus of this paper and we omit any further discussion on this issue. We refer the interested reader to recently proposed mapping algorithms from XML query languages to SQL [4, 9].

4 Schema Transformations and Search

In this section, we describe possible transformations for *p-schemas*. By repeatedly applying these transformations, LegoDB generates a space of alternative *p-schemas* and corresponding relational configurations.

4.1 XML transformations

Before we define the *p-schema* transformations, it is worth pointing out that there are important benefits to performing these transformations at the XML Schema level as opposed to transforming relational schemas. Much of the semantics available in the XML schema is not present in a given relational schema and performing the equivalent rewriting at the relational level would imply complex integrity constraints that are not within the scope of relational keys and foreign keys. As an example, consider the rewriting on Figure 3(c): such partitioning of the `Show` table would be very hard to come up with just considering the original schema of Figure 3(a). On the other hand, we will see that this is a natural rewriting to perform at the XML level. In addition, working at the XML Schema level makes the framework more easily extensible to other non-relational stores such as native XML stores and flat files, where a search space based on relational schemas would be an obstacle.

There is a large number of possible rewritings applicable to XML Schemas. Instead of trying to give an exhaustive set of rewritings, we focus on a limited set of such rewritings that correspond to interesting storage alternatives, and that our experiments show to be beneficial in practice.

Inlining/Outlining As we pointed out in Section 2, one can either associate a type name to a given nested element (outlining) or nest its definition directly within its parent element (inlining). Rewriting an XML schema in that way impacts the relational schema by inlining or outlining the corresponding element within its parent table. Inlining is illustrated in below:

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

type Description =
  description[ String ]
```

Inlining Transformation

Two conditions must be satisfied for this transformation to be permissible. First, the type name must occur in a position where it is not within the production of a named type (*i.e.,* it must comply with the type stratification). Second, since this rewriting implies that one table is removed from the relational schema, the corresponding type cannot be shared.

Note that inlining is the basis of the strategies proposed in [18]. Inlining has some similarities with vertical partitioning. It reduces the need for joins when accessing the content of an element, but at the same time it increases the size of the corresponding table and the cost of retrieving individual tuples. In the inlining example above, the benefits of inlining or outlining `description` element within the `TV` type depend both on the frequence of accesses to this element in the workload as well as its length. Our search algorithm decides whether to outline or inline that element based on the cost of each derived configuration.

Union Factorization/Distribution Union types are often used to add some degree of flexibility to the schema. As queries can have different access patterns on unions, *e.g.,* access either parts together or independently, it is essential that appropriate storage structures can be derived. In our framework, we use simple distribution laws on regular expressions to explore alternative storage for union. The first law (`(a,(b|c)) == (a,b|a,c)`) allows distribution of a union within a regular expression and is illustrated in Figures 6(a) and (b). Note that the common part of the schema (`title`, etc.) is now duplicated, while each part of the union is distributed. The second law (`a[t1|t2] == a[t1]|a[t2]`) allows to distribute a union across an element and is illustrated in Figure 6(c). Here the distribution is done across element boundaries. Note that at the relational level, this results in the schema on Figure 3(c).

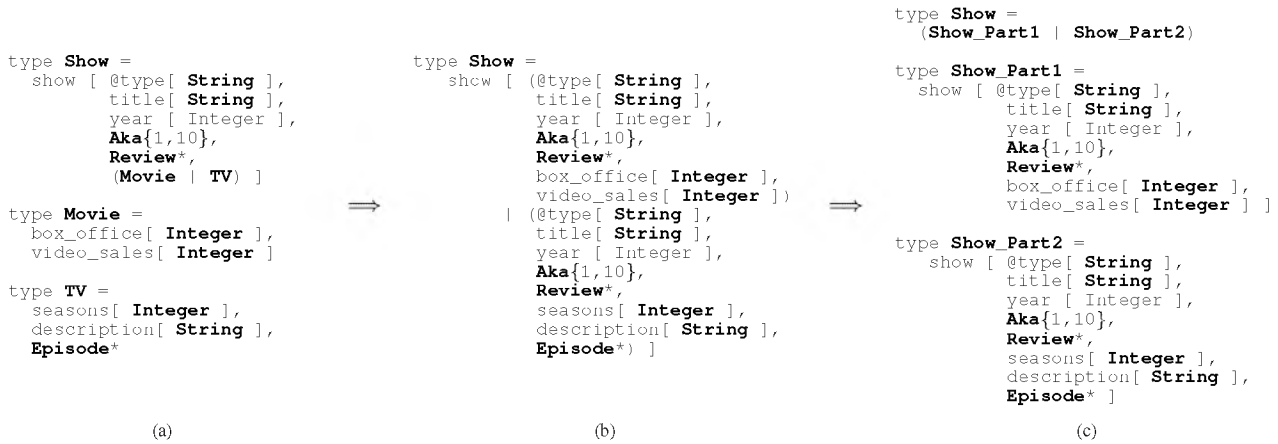
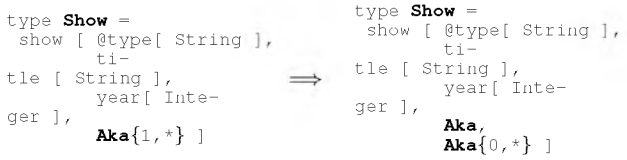


Figure 6. Union Distribution (XML Schema)

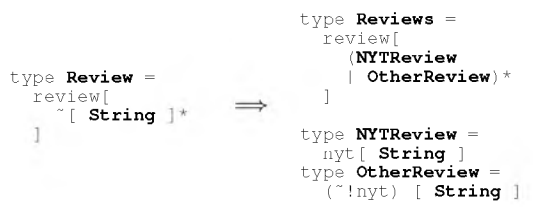
This transformation highlights the advantages of working in the space of XML Schemas. The corresponding horizontal partitioning of the relational schema of Figure 3 would not be easily found by a relational physical-design tool, since the information about the set of attributes involved in the union would have been lost.

Repetition Merge/Split Another useful rewriting exploits the relationship between sequencing and repetition in regular expressions, by turning one into the other. The corresponding law over regular expressions (`a+ == a,a*`) is illustrated below for the `aka` element in the `Show` type of Figure 1(b). Note that this transformation (followed by inlining the unrolled occurrence of `aka` into `Show`) is an alternative considered in [7].



Repetition Split Transformation

Wildcard rewritings Wildcards are used to indicate a set of element names that can or cannot be used for a given element. We use '~' to indicate that any element name can be used, and '~!a' to indicate that any name but `a` can be used. In some cases, queries access specific elements within a wildcard. In that context, it might be interesting to materialize an element name as part of a wildcard as illustrated below:



Wildcard Rewriting Transformation

This transformation can be thought of as distributing of the (implicit) union in the wildcard over the element constructor (*i.e.,* `~` ≡ `(nyt_reviews|(~!nyt_reviews))`). Here again this results in some form of non-trivial horizontal partitioning over relations. As we show in Section 5, this rewriting is useful if some queries access NYTimes reviews independently of reviews from other sources.

From union to options All of the previously proposed rewritings preserve exactly the semantics of the original XML schema. This last rewriting that was proposed in [18] does not have this property, but allows to inline elements of a union using null values. This relies on the fact that a union is always contained in a sequence of optional types (i.e., `(t1|t2)` ⊂ `(t1?,t2?)`). This often results in tables with a large number of null values, but allows the system to inline part of a union, which might improve performance for certain queries.

4.2 Search Algorithm

The set of configurations that result from applying the previous schema transformations is very large (possibly infinite, *e.g.,* when applying repetition merge). For that reason, we use a *greedy heuristic* to find an efficient configuration. The exploration of the space of storage mappings is described in Algorithm 4.1. The algorithm begins by deriving an initial configuration $pSchema$ from the given XML Schema $xSchema$ (line 3). Next, the cost of this configuration, with respect to the given query workload $xWkld$ and the data statistics $xStats$ is computed using the function $GetPSchemaCost$ which will be described in a moment (line 3). The greedy search (lines 5-16) iteratively updates $pSchema$ to the cheapest configuration that can be derived from $pSchema$ using a single transformation. Specifically, in each iteration, a list of candidate configurations $pSchemaList$ is created by applying all applicable trans-

formations to the current configuration $pSchema$ (line 7). Each of these candidate configurations is evaluated using $GetPSchemaCost$ and the configuration with the smallest cost is selected (lines 8-14). This process is repeated until the current configuration can no longer be improved.

Algorithm 4.1 Greedy Heuristic for Finding an Efficient Configuration

```
   Procedure GreedySearch
   Input:   xSchema : XML schema,
                xWkld : XML query workload,
                xStats : XML data statistics
   Output:  pSchema : an efficient physical schema
1  begin
       minCost = ∞;
       pSchema = GetInitialPhysicalSchema(xSchema)
       cost = GetPSchemaCost(pSchema, xWkld, xStats)
5      while (cost < minCost) do
           minCost = cost
           pSchemaList = ApplyTransformations(pSchema)
           for each pSchema' ∈ pSchemaList do
               cost' = GetPSchemaCost(pSchema', xWkld, xStats)
10             if cost' < cost then
                   cost = cost'
                   pSchema = pSchema'
               endif
           endfor
15     endwhile
       return pSchema
   end.
```

We now outline how $GetPSchemaCost$ computes the cost of a configuration given a $pSchema$, the XML Query workload $xWkld$, and the XML data statistics $xStats$. First, $pSchema$ is used to derive the corresponding relational schema. This mapping is also used to translate $xStats$ into the corresponding statistics for the relational data, as well as to translate individual queries in $xWkld$ into the corresponding relational queries in SQL. The resulting relational schema and the statistics are taken as input by a relational optimizer to compute the expected cost of computing a query in the SQL workload derived as above; this cost is returned as the cost of the given $pSchema$. Note that the algorithm does not put any restriction on the kind of optimizer used (transformational or rule-based, linear or bushy, etc. [11]); though for the exercise to make sense it is expected that it should be similar to the optimizer used in the target relational system.

5 Experimental study

LegoDB prototype We have implemented the LegoDB components shown in Figure 4. Our initial prototype is limited to exploring inlining/outlining rules in the greedy search—the other XML transformations are explored separately. To evaluate the cost of alternative configurations in our mapping engine, we used a variation of the Volcano relational query optimizer [11], as described in [14]. This relies on a cost model that takes into account number of seeks, amount of data read, amount of data written, and CPU time for in-memory processing. Our cost model is fairly sophisticated and its accuracy has been verified by comparing its estimates with numbers obtained by running queries on Microsoft SQL-Server 6.5 (see [14]).

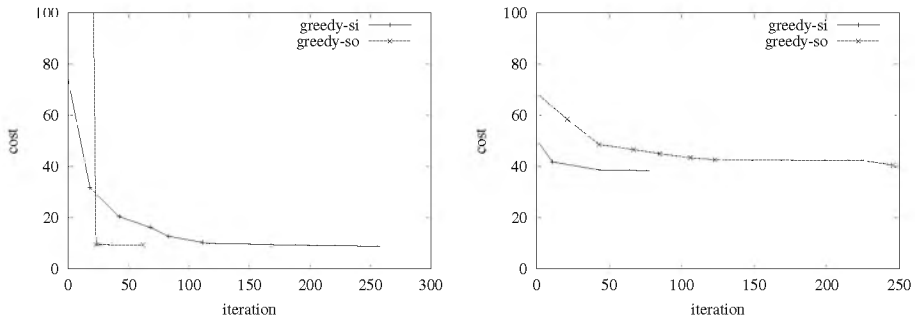
Experimental Settings We use an XML Schema based on the data from the Internet Movie Database (IMDB)[12] which contains information about movies, actors and directors. We compose workloads by drawing on two classes of queries: *lookup* queries and *publishing* queries. Lookup is representative of interactive SPJ queries, such as *Find the alternate titles for a given show*. Publishing queries are more document-oriented and return all available information about a particular element (or set of elements), for example *List all shows and their reviews*. Detailed statistics that include information about all elements (cardinalities, sizes, etc), as well as the XML schema and XQuery workloads can be found in the full version of the paper [2].

Efficiency of Greedy Search In this experiment, we demonstrate the efficiency of the greedy search heuristic described in Section 4.2. We experimented with two variations of the greedy search: *greedy-so* and *greedy-si*. In the *greedy-so* search, all elements in the initial physical schema are outlined (except base types) and during the search, inlining transformations are applied. For *greedy-si*, all elements are initially inlined (except elements with multiple occurrences) and during the search, outlining transformations are applied.

For the purpose of this experiment, we considered two workloads: *Lookup*, which contains five lookup queries, and *Publish*, which consists of three queries that publish information about shows, directors and actors. Figure 7 shows the cost of the configurations obtained by *greedy-so* and *greedy-si* on successive iterations for each of these workloads. Each iteration took approximately 3 seconds.

An interesting observation is that *greedy-so* converges to the final configuration a lot faster than *greedy-si* for lookup queries, while the opposite happens for publish queries, *i.e., greedy-si* converges faster. The traversals made by lookup queries are localized. Therefore, the final configuration has only a few inlined elements. Naturally, *greedy-so* can reach this configuration earlier than *greedy-si*. On the other hand, since the publish queries typically traverse larger number of elements, the final configuration has several inlined elements. In this case, therefore, *greedy-si* can reach this configuration earlier than *greedy-so*. Also, the curves often have a point after which the improvement between iterations decreases considerably. This suggests that, as an optimization, we could stop the search as soon as the improvement falls below a certain threshold.

As the graphs show, *greedy-so* has higher initial costs for both workloads since it leads to a large number of tables which must be joined to compute the queries. However, note that both strategies converge to similar costs (the final



(a) Lookup

(b) Publish

Figure 7. Cost at each greedy iteration

configurations are also similar). This trend was observed for all variations of schemas, statistics and workloads we experimented with. For simplicity of presentation, *greedy-si* is the search strategy used in the experiments below.

Sensitivity of configurations to varied workloads An important feature of the LegoDB framework is that the storage is designed taking an application and its query workload into account. One interesting question is how the resulting configuration performs if the workload changes. For example, the search interface of IMDB offers users a fixed set of queries. However, the frequency of these queries may vary over time. For example, in the week before the Academy Awards, the frequency of queries about movies may increase considerably. Because in many instances it may not be feasible to re-generate a new configuration and re-load the data, it is important that a chosen storage configuration leads to acceptable performance even when the frequency of queries varies.

In order to assess the sensitivity of our resulting configurations to changes in workloads, we created a spectrum of workloads that combined the lookup queries and publish queries in the ratio $k : (1-k)$, where $k \in [0,1]$ is the fraction of lookup queries in the particular workload. Using the same statistics and XML schema, we ran LegoDB for three workloads corresponding to $k = 0.25, 0.50$ and 0.75, resulting in the three configurations C[0.25], C[0.50] and C[0.75] attuned to the respective workloads. Next, we gathered these three resulting configurations and evaluated their costs across the entire workload spectrum; the cost of a configuration is defined as the average cost of processing a query on that configuration. We did a similar evaluation with the all-inlined configuration, C[ALL-INLINED]. For the sake of comparison, we also plotted a curve OPT giving, for each workload in the spectrum, the cost of the configuration obtained by LegoDB for that specific workload. (Note that, in contrast to the other curves, OPT does not correspond to a fixed schema.) The results are shown in

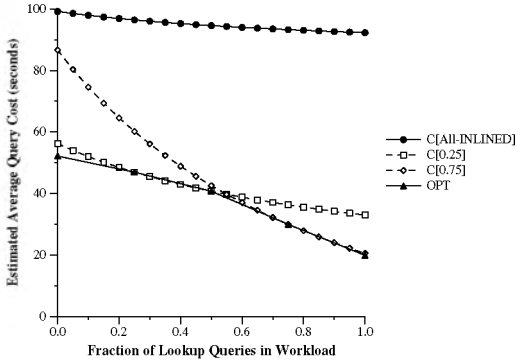


Figure 8. Sensitivity to workload variations

Figure 8.

Before discussing the results, it is important to understand how inlining affects the cost of a configuration with respect to a query workload. For queries that traverse the schema contiguously and access all related attributes, inlining helps by precomputing the numerous joins that may be required during the traversal. On the other hand, inlining could be a bad idea for other kinds of queries, for example: (a) the query does limited, localized traversals and/or does not access all the attributes involved, and so does not benefit from the inlining but nevertheless pays the overhead of scanning wider relations; (b) the query has highly selective selection predicates — this could render a selection scan on the inlined wider relation more expensive than evaluation of the query by joining the filtered non-inlined leaner relations, especially in the presence of appropriate indexes; (c) the query involves join of attributes not structurally adjacent in the XML Schema (*e.g.,* `actor` and `director`) — since inlining causes respective relations to widen due to the inclusion of several additional attributes not required in the join, the join is significantly more expensive than in the case of other configurations. These two opposing factors lead to the possibility of different inlining decisions for

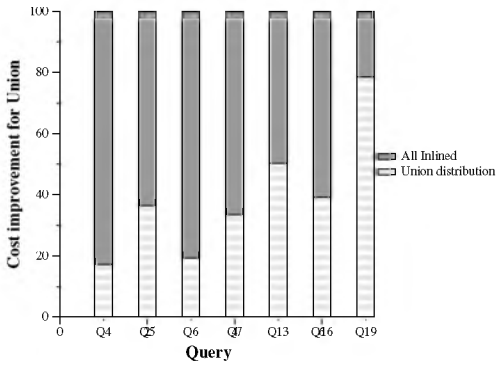


Figure 9. Union distribution vs. all-inlined

different workloads, each optimal in a certain region in the spectrum.

Overlap between the curves for C[0.25] and C[0.75] with the curve for OPT in the graph suggests that we can partition our spectrum into two regions: the region defined by $k \in [0, 0.55)$ and the region defined by $k \in [0.55, 1]$ such that C[0.25] is the optimal configuration for *all* workloads in the former region and C[0.75] is the optimal configuration for *all* workloads in the latter (or near enough). Moreover, the curves for C[0.25] and C[0.75] cross at a small angle. This further implies that even if the two workloads lie in different regions but are not too distant, the optimal configurations for the two are close enough in cost. This shows that the configurations found by LegoDB are very robust with respect to the variations in the workloads.

At the extremes of the spectrum, however, we found a significant difference in performance of the C[0.25] and C[0.75]. Since these two configurations are based on slightly different inlining decisions, we see that both publish and lookup queries are sensitive to these decisions, and that inlining is indeed an important transformation. However, C[ALL-INLINED] that includes all the inlining decisions in the above configurations (and some more) performed two to five times worse than optimal. This demonstrates that beyond a point, the overheads due to inlining significantly outweigh any benefits.

In summary, the above analysis clearly demonstrates that the cost-based approach of LegoDB leads to configurations that are not only 50% to 80% less costly than the rule-of-the-thumb approach of ALL-INLINED, but also are very robust with respect to the variations in the workloads.

Union Distribution In order to measure the effectiveness of union distribution we compared the costs of various queries for the configurations illustrated in Figure 3(a) (all elements inlined) and Figure 3(c) (where union is distributed over `show`).

As shown on Figure 9, the configuration obtained through union distribution has lower costs for all queries.

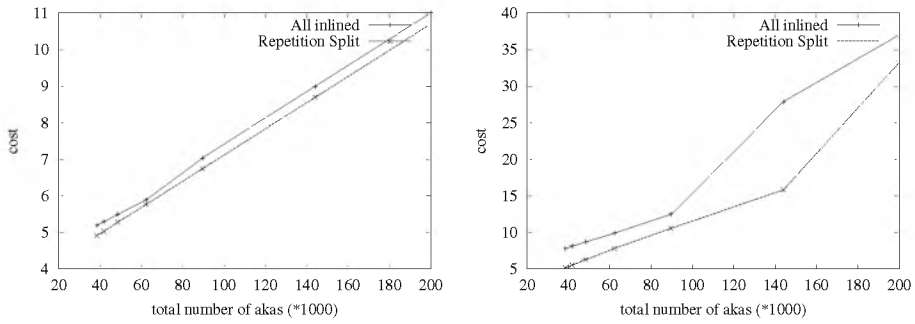
Total reviews	10,000		100,000	
Query	Q1		Q2	
NYT perc.	inlined	wild	inlined	wild
50%	5.42	6.3	48	26.3
25%	5.42	5.1	48	15
12.5%	5.42	4.4	48	9.4

Table 1. Wildcard-transformed vs. all-inlined

As we explained in Section 4, the union distribution is equivalent to horizontally partitioning over the `Show` table into a table that contains information about movies, and a table that contains information about TV shows. Because the new tables are smaller, queries that refer to elements in only one of those tables will be cheaper. These results are rather intuitive. A less intuitive finding is that even queries that access elements from both movies and TV shows can become cheaper under the union rewriting. For instance, the following selection query $\Pi_{title,description}(\sigma_{title=c}shows)$ that returns the title and description of a given show, must be rewritten as the union of two queries: $\Pi_{title,box\_office}(\sigma_{title=c}movies)$ and $\Pi_{title,description}(\sigma_{title=c}tv\_shows)$ over the transformed schema. Not only does each subquery operate on tables with fewer tuples, but these tables are also narrower which reduce the cost of selection.

Repetition Split Another transformation we considered in Section 4 is splitting repetitions. The effectiveness of such a transformation is highly dependent on the characteristics of the data and on the query workload. Consider for example two queries: a lookup query that finds all of the alternate titles (akas) for a given show title; and a publishing query which retrieves all information for all shows. The costs for these two queries under the *All Inlined* and the *Repetition-Split* transformed configurations for a varied number of total akas are given in Figure 10. For this example, the main effect of the Repetition Split transformation is that it reduces the size of the `Aka` table. As a result, the cost reduction is bigger for the publishing query—since the lookup query involves a selection on `title` and this selection can be pushed, the size of the `Aka` table will impact the `show-aka` join to a lesser extent than in the publishing query where no selection is performed. Also note that as the size of the `Aka` table increases (and becomes much larger than the `Show` table), the cost difference between the two configurations decreases.

Wildcards The wildcard's rewriting proposed in Section 4 effectively partitions the set of elements tagged by the wildcard into two different sorts, corresponding to the wildcard labels that are present in the data. Consider for example the query *Find the NYTimes reviews for shows produced in 1999*. The equivalent queries under the configurations in Figure 3(a) and (b) are respectively:



(a) Lookup (b) Publish

Figure 10. Repetition-split vs. all-inlined

$\Pi_{title,data}(\sigma_{year=1999}(shows) \bowtie \sigma_{source=\text{"NYT"}}(reviews))$ and $\Pi_{title,data}(\sigma_{year=1999}(shows) \bowtie nyt\_reviews)$. Table 1 shows the cost of these two queries for varying percentage of New York Times reviews, when the total number of reviews is 10,000 and 100,000. As expected, whereas the cost for Query 1 remains constant, the cost for Query 2 decreases with the size of the `nyt_reviews` table.

6 Related Work

Recently, many approaches have been suggested for mapping XML documents to relations for storage [7, 10, 13, 17, 18, 19]. In [7], Deutsch, Fernandez and Suciu propose the STORED system for mapping between (schemaless) semi-structured data and the relational data model. They focus on a data mining technique which simultaneously solves schema discovery and storage mapping by identifying "highly supported" tree patterns for storage in relations. Even though they considered a cost optimization approach to the problem, they found it to be impractical, as in the absence of a schema, optimization is shown to be exponential in the *size of the data*. In contrast, we explore a space of storage structures but rely on the *schema* and *statistics* rather than directly mining the data. We use heuristics (*e.g.,* the greedy approach) to avoid an exponential search, but still explore a variety of useful mappings. In fact, the LegoDB strategy may lead to substantially different configurations than what is produced by the data-mining approach used by STORED. For example, we may break an extremely common pattern of data into multiple relations if the result is more efficient for the query workload.

In [18], Shanmugasundaram *et al* propose three strategies to map DTDs into relational schemas. The basic idea behind these mappings is to create tables that correspond to XML elements defined in a DTD, inlining as many subelements as possible so as to reduce fragmentation—multivalued elements and elements involved in recursive associations must be kept in separate tables. The three proposed mappings differ from one another in the degree of redundancy: they vary from being highly redundant (where an element can be stored in multiple tables), to containing no redundancy. While we do not consider mappings which duplicate data, we share with [18] the use of the schema to derive a heuristically "good" initial storage mapping (*e.g.,* for the *greedy-si* search strategy), and the use of a modified schema for the storage mapping language. Regardless of the particular strategy, the mapping process of [18] begins by simplifying an input DTD into a DTD that can be easily mapped into relations. Instead of *simplifying* away hard-to-map XML Schema constructs, LegoDB takes advantage of them (through the use of our schema transformations) to generate a space of mappings. And as we have shown in Section 5, mappings that result from the XML-specific transformations may lead to significantly better configurations for a given application than mappings based on an inline-as-much-as-possible approach.

Schmidt *et al* [17] propose a highly fragmented relational storage model. Their experiments show that this approach performs well on the main-memory-oriented Monet database, a result in stark contrast to the conclusions presented in [18] where fragmentation and a large number of joins is identified as a key problem. These disparate performance results only emphasize the need for automated tools, like LegoDB, to determine the appropriate storage mapping for a given application *and DBMS platform.* Finally, while the search space in our work does not include horizontal fragmentation of tables based on incoming paths, our rewriting rules can be extended to consider this style of transformation.

Florescu and Kossman [10] and Tian *et al* [20] compare the performance of several approaches to XML storage. Shimura *et al* [19] propose an inverted-list-style storage structure in which nodes are mapped to regions in the document, and paths are present as strings in a "Path" ta-

ble. In all three of these cases, one or more *fixed* mappings are used, where we explore a *space* of storage mappings. Mappings from DTDs into nested schema structures of OO or OR/DBMS have been proposed [6, 13]. While Klettke and Meyer consider statistics and queries in the proposed heuristic mapping, no attempt is made to compare estimated costs for multiple mappings.

Several commercial DBMSs already offer some support for storing, querying, and exporting XML documents [22, 15]; however, the user must still design an appropriate storage mapping.

While LegoDB is (to our knowledge) the first XML storage mapping tool to take advantage of cost-based optimization, similar approaches have been applied to problems in relational storage design, such as index selection (*e.g.,* [16]) and view materialization (*e.g.,* [1, 21]) in physical optimization for relational DBMSs. Note that physical design tools are complementary to LegoDB, and can be applied to further optimize the relational schemas produced by our mapping, either during the search process or simply on the final schema produced.

7 Conclusions

We have introduced LegoDB, a cost-based framework for XML storage. LegoDB explores a space of alternate storage configurations and evaluates the quality of each configuration by estimating its performance on an application workload. We make original use of XML Schema as a support the description and exploration of new possible storage configurations. The LegoDB system isolates the application developer from the underlying storage engine by taking XML Schemas, an XQuery workload and XML statistics as input. Our initial performance study indicates that XML storage performances can be significantly improved with such a cost-based approach. We consider this work as a first step towards a general purpose storage configuration engine for XML. As future work, we plan to adapt our approach to other storage platforms, extend the subset of XQuery we support, and possibly develop more efficient search strategies.

References

[1] S. Agrawal, S. Chaudhuri, and V.R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proc. of VLDB*, pages 496–505, 2000.

[2] P. Bohannon, J. Freire, P. Roy, and J. Sim´eon. From XML schema to relations: A cost-based approach to XML storage. Technical report, Bell Laboratories, 2001. Full version.

[3] J. Bosac, T. Bray, D. Connolly, E. Maler, G. Nicol, C.M. Sperberg-McQueen, L. Wood, and J. Clark. Guide to the W3C XML specifi cation ('XMLspec") DTD, June 1998. http://www.w3.org/XML/1998/06/xmlspec-report.

[4] M.J. Carey, J. Kiernan, J. Shanmugasundaram, E.J. Shekita, and S.N. Subramanian. XPERANTO: Middleware for publishing object-relational data as XML documents. In *Proc. of VLDB*, pages 646–648, 2000.

[5] D. Chambelin, J. Clark, D. Florescu, J. Robie, J. Sim´eon, and M. Stefanescu. XQuery 1.0: An XML query language. W3C Working Draft, June 2001.

[6] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. of SIGMOD*, pages 313–324, 1994.

[7] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. of SIGMOD*, pages 431–442, 1999.

[8] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Sim´eon, and P. Wadler. The XML query algebra, February 2001. http://www.w3.org/TR/2001/query-algebra.

[9] M.F. Fernandez, W.C. Tan, and D. Suciu. SilkRoute: Trading between relations and XML. *WWW9/Computer Networks*, 33(1-6):723–745, 2000.

[10] D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing XML in a relational database. Technical Report 3680, INRIA, 1999.

[11] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and effi cient search. In *Proc. of ICDE*, 1993.

[12] Internet Movie Database. http://www.imdb.com.

[13] M. Klettke and H. Meyer. XML and object-relational database systems - enhancing structural mappings based on statistics. In *Proc. of WebDB*, pages 63–68, 2000.

[14] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Effi cient and extensible algorithms for multi query optimization. In *Proc. of SIGMOD*, pages 249–260, 2000.

[15] M. Rys. State-of-the-art XML support in RDBMS: Microsoft SQL Server's XML features. *Bulletin of the Technical Committee on Data Engineering*, 24(2):3–11, June 2001.

[16] V.R. Narasayya S. Chaudhuri. AutoAdmin 'what-if' index analysis utility. In *Proc. of SIGMOD*, pages 367–378, 1998.

[17] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Effi cient relational storage and retrieval of XML documents. In *Proc. of WebDB*, pages 47–52, 2000.

[18] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of VLDB*, pages 302–314, 1999.

[19] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and retrieval of XML documents using object-relational databases. In *Proc. of DEXA*, pages 206–217, 1999.

[20] F. Tian, D. DeWitt, J. Chen, and C. Zhung. The design and performance evaluation of various XML storage strategies. http://www.cs.wisc.edu/niagra/Publications.html, 2001.

[21] O. Tsatalos, M. Solomon, and Y. Ioannidis. The GMAP: A versatile tool for physical data independence. In *Proc. of VLDB*, pages 367–378, 1994.

[22] Oracle's XML SQL utility. http://technet.oracle.com/tech/xml/oracle_xsu.