Bridging the XML–Relational Divide with LegoDB: A Demonstration*

Philip Bohannon‡ Juliana Freire† Jayant R. Haritsa+ Maya Ramanath+ Prasan Roy‡ Jérôme Siméon‡

Abstract

We present LegoDB, a cost-based XML storage mapping engine that automatically explores a space of possible XML-to-relational mappings and selects an efficient mapping for a given application.

1. Introduction

XML is becoming the predominant data exchange format in a variety of application domains (supply-chain, scientific data processing, telecommunication infrastructure, etc.). By relying on relational engines for storage purposes, XML developers can benefit from a complete set of data management services (including concurrency control, crash recovery, and scalability) and from the highly optimized relational query processors. However, due to the mismatch between the XML and the relational models and the many different ways to map an XML document into relations, it is very hard to tune a relational engine and ensure that XML queries will be evaluated efficiently. In fact, many current products (*e.g.,* [5]) require developers to go through an often lengthy and complex process of manually defining a mapping from XML into relations. In this demonstration, we present our **LegoDB** system, which is aimed at automatically providing XML developers with an efficient storage solution that is tuned for the given application.

We motivate the need for finding appropriate storage mappings with an XML application scenario inspired from the Internet Movie Database (IMDB) [4]. This database, whose XML Schema is shown in Figure 1, 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`).

```
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*
type Episode =
   episode[ name[ String ],
   guest_director[ String ]
                  ...
```

Figure 1. XML Schema for IMDB documents

Three possible relational storage mappings for the IMDB schema are shown in Figure 2. Configuration (a) results from inlining as many elements as possible in a given table, roughly corresponding to the strategies presented in [6]. 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` table into Movie shows (`Show1`) and TV shows (`Show2`). Even though each of these configurations can be the best for a given application, there are cases where they perform poorly. 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 example, the first storage mapping shown in Figure 2 inlines several fields that are not present in all the data, making the `Show` relation wider than necessary. Similarly, when the entire `Show` relation is exported as a single document, the records corresponding to movies need not be joined with the `Episode` table, but this join is required by mappings 2(a) and (b). Finally, the (potentially large) `description` element need not be inlined unless it is frequently queried.

*An earlier version of this demo was previously presented at the 28th Intl. Conf. on Very Large Data Bases (VLDB), August 2002.

†Computer Science & Engineering, Oregon Graduate Institute, Beaverton, OR 97006, USA. juliana@cse.ogi.edu (Contact Author)

+Database Systems Lab, SERC, Indian Institute of Science, Bangalore 560012, INDIA. {haritsa,maya}@dsl.serc.iisc.ernet.in

‡Lucent Bell Labs, 600 Mountain Avenue, Murray Hill, NJ 07974, USA. {bohannon,prasan,simeon}@research.bell-labs.com

Proceedings of the 19th International Conference on Data Engineering (ICDE'03)
1063-6382/03 $ 17.00 © 2003 IEEE

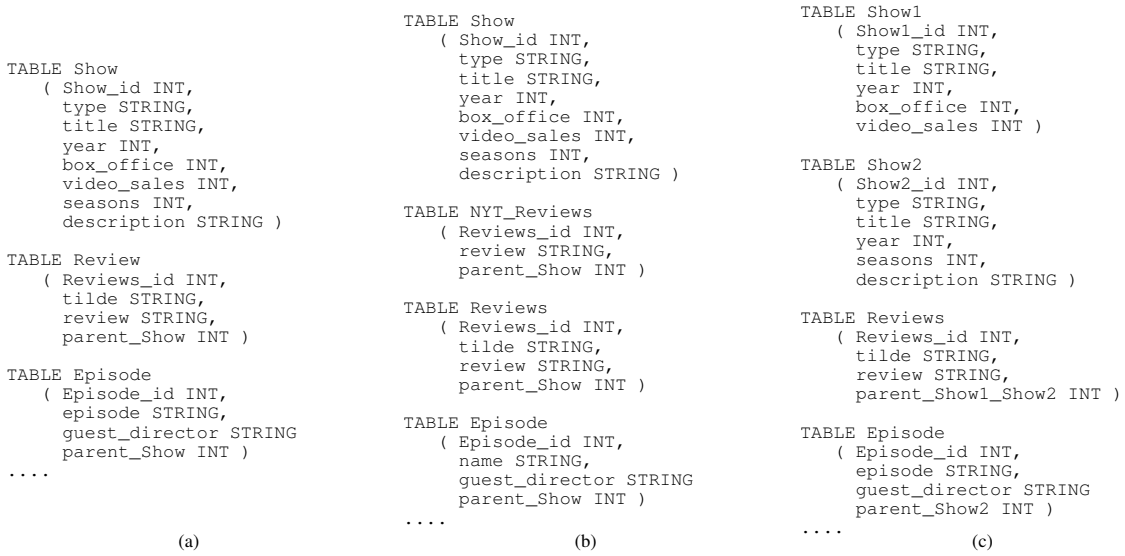

Figure 2. Three storage mappings for shows

2. XML Storage with LegoDB

LegoDB is a cost-based XML storage mapping engine that automatically explores a space of possible XML-to-relational mappings and selects an efficient mapping for a given application. It is based on the following principles:

Logical/Physical independence. An XML application developer should be able to design her application at a logical level, *i.e.,* using XML-driven design tools, without requiring expertise in relational technology.

Automatic mapping. The generation of XML-to-relational mappings must be automatic — developers should not be required to manually specify mappings.

Application-driven mapping. The storage design should take into account the requirements of the target application. LegoDB takes application characteristics into account and uses a cost-based approach in order to find the *best* storage for a given application.

Leverage existing technologies. LegoDB leverages current XML and relational technologies whenever possible. The target application characteristics are modeled using XML Schema, an XQuery workload, and a set of sample XML documents. The best among the derived configurations is selected using cost estimates obtained by a standard relational optimizer.

Extend existing technologies. LegoDB develops new specific extensions to existing technologies whenever necessary. Notably, in [1], we propose novel XML Schema rewriting techniques to generate a space of possible relational mappings, and in [3], we extend XML Schema with statistics in order to support accurate cost estimation for XQuery workloads.

3. LegoDB Architecture

The architecture of LegoDB, shown in Figure 3, is composed of two main components: *storage design* and *runtime support*, described below.

3.1. Storage design

LegoDB takes, as inputs, parameters that describe the target application (an XML Schema, an XQuery workload, and a set of sample documents) and outputs an efficient relational configuration (a set of relational tables) as well as a mapping specification. The modules of the storage design component (see Figure 3) are the following:

StatiX. The first task in the system is to extract statistical information (about the values and structure) from the given XML document, and this is done by the *StatiX* module. This information is necessary to *derive* accurate relational statistics that are needed by the relational optimizer to accurately estimate the cost of the query workload.

Physical Schema Generation. The statistics together with the XML Schema are sent to the *Physical Schema Generation* module, which outputs a *physical schema*, or p-schema. An important feature of p-schemas is that

Proceedings of the 19th International Conference on Data Engineering (ICDE'03)
1063-6382/03 $ 17.00 © 2003 IEEE

there exists a *fixed* mapping between p-schema types and relational tables.

Physical Schema Transformation. The system then searches for efficient relational configurations by repeatedly transforming p-schemas, *i.e.,* generating new p-schemas by adding/removing types and changing regular expressions into equivalent expressions. But, each transformed p-schema will validate *exactly* the same set of documents as the original schema. Note that because p-schema types are mapped into relations, LegoDB generates a series of distinct relational configurations by performing schema transformations.

Translation Module. For each transformed p-schema, the *Translation Module* generates a set of relational tables, translates the XQuery workload into the SQL equivalent, and derives the appropriate statistics for the generated tables. This information is then input to the relational optimizer for cost estimation.

The design phase produces an XML-to-relational mapping that has the lowest cost among the alternatives explored by LegoDB. It is important to note that: the relational optimizer is used by LegoDB as a black box to obtain cost estimations; and the quality of the selected mapping depends on the accuracy of the estimates computed by the optimizer. A more detailed description of the various system modules is available in [1, 3].

3.2. Runtime Support

The runtime support component of LegoDB (see Figure 3) operates as follows: After a configuration is selected, the corresponding tables are created in the RDBMS. The *DB Loader* module shreds the input XML documents and loads it into these tables. Once the relational database is created and loaded, the *Query Translation* module is used to perform query translation on behalf of the target XML application. Note that other XQuery to SQL mapping tools can also be used in LegoDB (for instance, [2]).

4. Demonstration

The proposed demonstration will show the complete process – storage design and runtime support – for storing and querying XML in a relational database. We will show for different schemas and datasets (IMDB, DBLP, etc.) as well as different applications, how LegoDB derives efficient configurations and mappings. The steps will include the collection of statistics from a given set of XML documents and searching a space of relational configurations. Further, we will demonstrate (for the first time) the impact of the choice

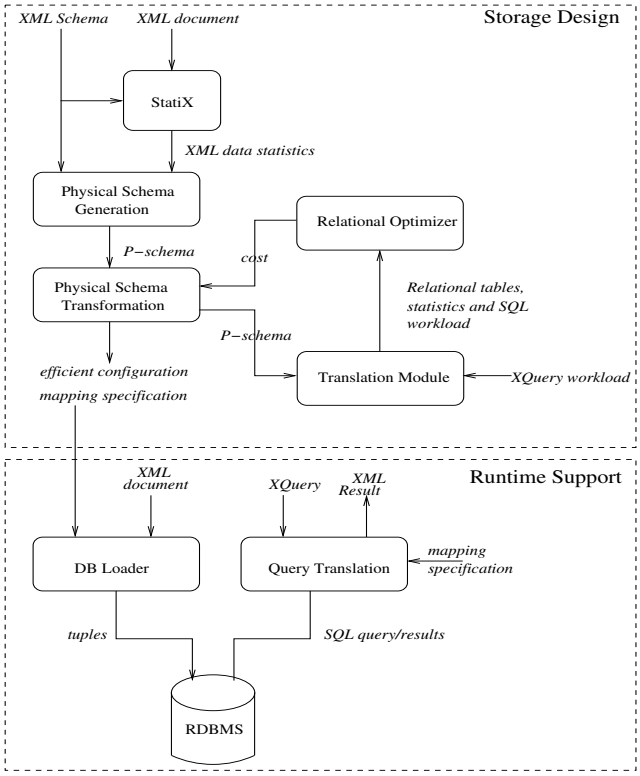

Figure 3. LegoDB Architecture

of search algorithm on the resulting relational configurations. We will also show how runtime support components are used to load the XML data into the selected relational schema and run XQuery queries on the relational database by translating the XQuery query to SQL. Finally, we will illustrate the performance improvement obtained by LegoDB by comparing query evaluation times of configurations selected by LegoDB against configurations derived by mapping strategies proposed in the literature.

References

[1] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, 2002.

[2] M. Fernandez, W. Tan, and D. Suciu. Silkroute: trading between relations and XML. *WWW9/Computer Networks*, 33(1-6):723–745, 2000.

[3] J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Statix: Making XML count. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 2002.

[4] Internet Movie Database. http://imdb.com.

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

[6] J. Shanmugasundaram et al. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of Intl. Conf. on Very Large Data Bases (VLDB)*, 1999.

Proceedings of the 19th International Conference on Data Engineering (ICDE'03)
1063-6382/03 $ 17.00 © 2003 IEEE