LegoDB: Customizing Relational Storage for XML Documents

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

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

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

1 Introduction

XML is becoming the predominant data exchange format in a variety of application domains (supply-chain, scientific data processing, telecommunication infrastructure, etc.). Not only is an increasing amount of XML data now being processed, but XML is also increasingly being used in business-critical applications. Efficient and reliable storage is an important requirement for these applications. By relying on relational engines for this purpose, 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.

Because of 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. Most database vendors already offer solutions to address the need for reliable XML storage. However, current products (*e.g.,* [10]) require developers to go through an often lengthy and complex process of manually defining a mapping from XML into relations.

Strategies that automate the process of generating XML-to-relational mappings have been proposed in the literature (see, *e.g.,* [2, 4, 7, 8, 9]). Due to the flexibility of the XML infrastructure, different XML applications exhibit widely different characteristics (*e.g.,* permissive vs. strict schemas, different access patterns). 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. As we show in [1], a fixed mapping or a mapping that does not take the application characteristics into account is unlikely to work well for more than a few of the wide variety of XML applications.

The purpose of this demonstration is to present the LegoDB system, which is aimed at providing XML developers with an efficient storage solution tuned for a given application.

2 Motivation

We motivate the need for finding appropriate storage mappings with an XML application scenario inspired from the Internet Movie Database (IMDB) [6]. 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`). Sample data reflective of real-world information that conforms with this schema in shown in Figure 2.

Three possible relational storage mappings for the IMDB schema are shown in Figure 3. Configuration (a) results from inlining as many elements as possible in a given table, roughly corresponding to the strategies presented in [8]. 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 (`Show_Part1`) and 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 point is that one cannot decide which of these configurations will perform

*Contact Author

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

```
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

```
<imdb>
  <show type="Movie">
    <title>Fugitive, The</title>
    <year>1993</year>
    <aka>Auf der Flucht</aka>
    <aka>Fuggitivo, Il</aka>
    <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>
    <video_sales>72,450,220</video_sales>
  </show>

  <show type="TV series">
    <title>X Files, The</title>
    <year>1994</year>
    <aka>Akte X - Die unheimlichen
         Fälle des FBI</aka>
    <aka>Aux frontieres du Reel</aka>
    <seasons>
      <number>10</number>
      <years>1993 1994 1995 1996 1997
             1998 1999 2000 2001</years>
    </seasons>
    <description>
      A paranoic FBI agent teams up with
      a frustrated female scientist to chase
      DNA modified aliens financed by the NSA.
    <episode>
      <name>Ghost in the Machine</name>
      <guest_director>
         Jerrold Freedman
      </guest_director>
    </episode>
    <episode>
      <name>Fallen Angel</name>
      <guest_director>
          Larry Shaw
      </guest_director>
    </episode>
  </show>
  ....
</imdb>
```

Figure 2: Sample IMDB Document

well without taking the application (*i.e.,* a query workload and data statistics) into account.

For example, the first storage mapping shown in Figure 3, which is what would have been generated by previous heuristic approaches, 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 3(a) and (b). Finally, the (potentially large) `description` element need not be inlined unless it is frequently queried.

3 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 the best mapping for a given application. Experiments in [1, 5] show that the LegoDB mapping engine is very effective in practice and can lead to reductions of over 50% in the running times of queries as compared to previous mapping techniques. The LegoDB system 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, and need not be an expert in the underlying 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 [5], we extend XML Schema with statistics in order to support accurate cost estimation for XQuery workloads.

4 LegoDB Architecture

The architecture of LegoDB, shown in Figure 4, is composed of two main components: *storage design* and *run-*

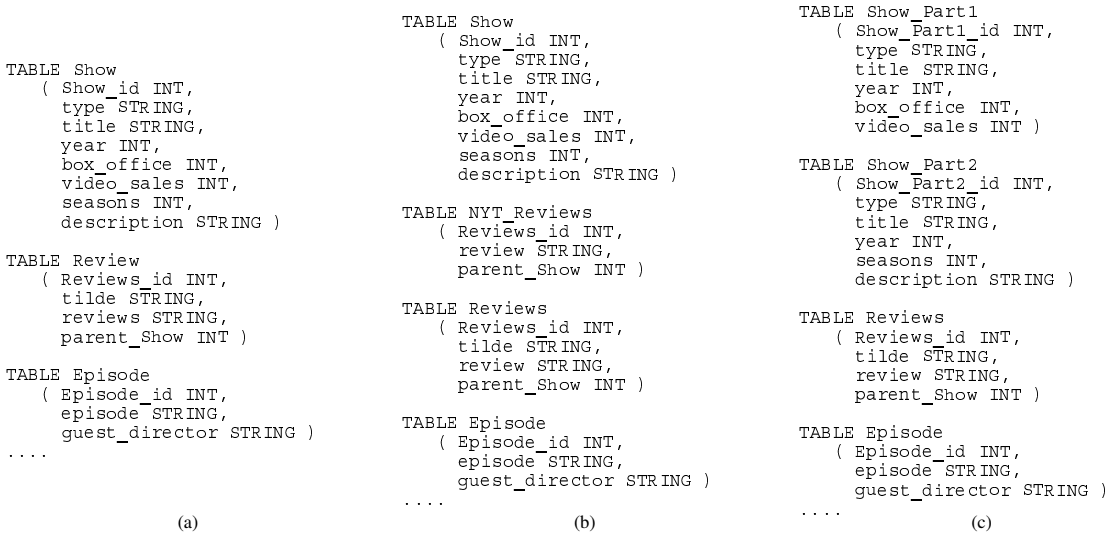

Figure 3: Three storage mappings for shows

time support. These components are described in the remainder of this section.

4.1 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 for storage design component (see Figure 4) 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. Details about statistics extraction in LegoDB can be found in [5].

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

Physical Schema Transformation. The system then starts the search for an efficient relational configuration. It does that by repeatedly transforming p-schemas, *i.e.,* generating new p-schemas that are structurally different, but that validate the same documents. Note that because p-schema types are mapped into relations, by performing schema transformations, LegoDB generates a series of distinct relational configurations.

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 selected tables. This information is then input to the 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.

For a more detailed description of the various modules, definitions of the physical XML Schemas and the XML Schema transformations, the reader is referred to [1].

4.2 Runtime Support

The runtime support component of LegoDB (see Figure 4) operates as follows: After a configuration is selected, the corresponding tables are created in the RDBMS. The *DB Loader* module shreds the input XML document 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 tools for mapping XQuery to SQL mapping tool can be used in LegoDB (for instance [3]).

5 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 a variety of schemas and datasets (IMDB, DBLP, etc.) how LegoDB derives efficient configurations and mappings that are adequate for a given application scenario. We will also

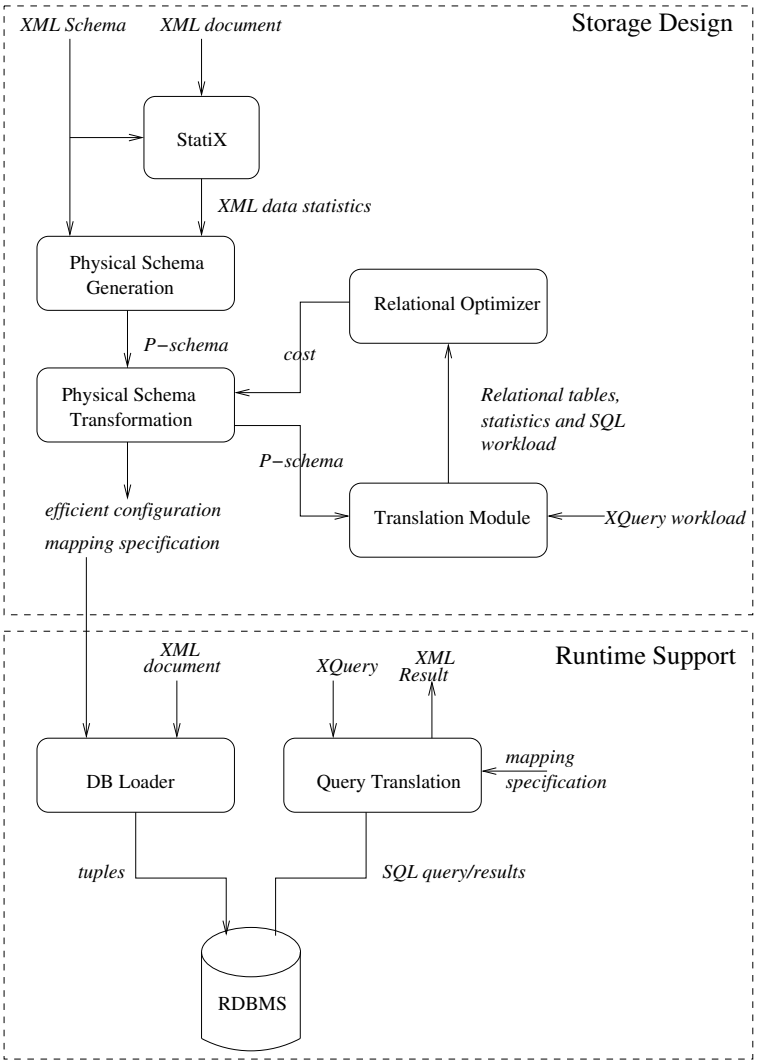

Figure 4: LegoDB Architecture

show how runtime support components are used to load the data and run queries. 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] A. Deutsch, M. Fernandez, and D. Suciu. Storing semi-structured data with STORED. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 431–442, 1999.

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

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

[5] 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.

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

[7] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *Proc. of Intl. Workshop on the Web and Databases (WebDB)*, pages 47–52, 2000.

[8] 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 Intl. Conf. on Very Large Data Bases (VLDB)*, pages 302–314, 1999.

[9] 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.

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