A Holistic XML-to-RDBMS Mapper (Demonstration)

Priti Patil   Jayant R. Haritsa

Database Systems Lab, SERC/CSA
Indian Institute of Science, Bangalore, INDIA
{priti,haritsa}@dsl.serc.iisc.ernet.in

1 Introduction

Over the past few years, XML's powerful and flexible data formatting capabilities have made it a dominant standard for information exchange between applications, especially on the Internet. As an increasing amount of XML data is being processed, efficient and reliable storage of XML data has become an important issue. For persistently storing information from XML sources, there are primarily two technological choices available: A specialized native XML store (e.g. Tamino [13], Timber [6]), or a standard relational engine (e.g. IBM DB2 [5], Oracle [9], MS-SQL Server [3]). From a pragmatic viewpoint, the latter approach brings with it the benefits of highly-functional, efficient, and mature technology. Therefore, a rich body of literature has emerged in the last five years on the mechanics of hosting XML documents on relational backends. Specifically, there have been several proposals for generating efficient mappings between XML schema (e.g. DTDs or XML Schema) and relational schema.

A common limitation of much of the prior work is that it has focused on *isolated* components of the relational schema, typically the table configurations. A complete relational schema, however, consists of much more than just table configurations – it also includes integrity constraints, indices, triggers, and views. Therefore, viable XML-to-relational systems that intend to support real-world applications need to provide a *holistic* mapping that incorporates all fundamental aspects of relational schemas.

In this demonstration, we will present a walk-through of a prototype system called **ELIXIR** (Establishing hoLIstic schemas for XML In Rdbms) that produces holistic relational schemas tuned to the application workload. The prototype is built using Ocamlc (Objective Caml) [8], a strongly-typed functional programming language, around the well-known LegoDB XML-to-RDBMS framework [1, 4, 11], and uses the IBM DB2 database engine as the relational backend. It has been successfully evaluated on a variety of real-world and synthetic XML schemas operating under a representative set of XQuery queries.

Elixir is based on LegoDB's principled *cost-based* approach to mapping design, thereby *automatically* delivering efficient mappings that are *tuned* to the XML application. This is in marked contrast to the mapping tools currently provided by commercial database systems, wherein the user is expected to play a significant role in the design and the tuning is largely manual.

A novel design feature of Elixir is that it performs *all* its mapping-related optimizations in the XML source space, rather than in the relational target space. For example, Elixir significantly extends prior table configuration techniques, based on XML schema transformations, to seamlessly preserve XML integrity constraints. With regard to index selection, too, Elixir makes path-index choices at the XML source and then maps them to relational equivalents – our experiments show that this is more desirable than the prevalent practice of using the relational engine's index advisor to identify a good set of indices. Finally, Elixir maps XML triggers and XML views to obtain relational triggers and relational views, respectively. An integrated approach to the design of these techniques ensures that the interactions between the XML inputs and their effects on the relational outputs are automatically taken into account during the optimization process.

In a nutshell, the Elixir system attempts to provide high-quality "industrial-strength" mappings for XML-on-RDBMS, and this demonstration will showcase its features. The complete technical details of Elixir are available in [10].

2 System Architecure

The overall architecture of the Elixir system is depicted in Figure 1. Given an XML schema, a set of documents valid under this schema, and the user query workload, the system first creates an equivalent canonical "fully-normalized" initial XML schema [4], corresponding to an extremely fine-grained relational mapping, and in the rest of the procedure attempts to

International Conference on Management of Data
COMAD 2005b, Hyderabad, India, December 20–22, 2005
©Computer Society of India, 2005

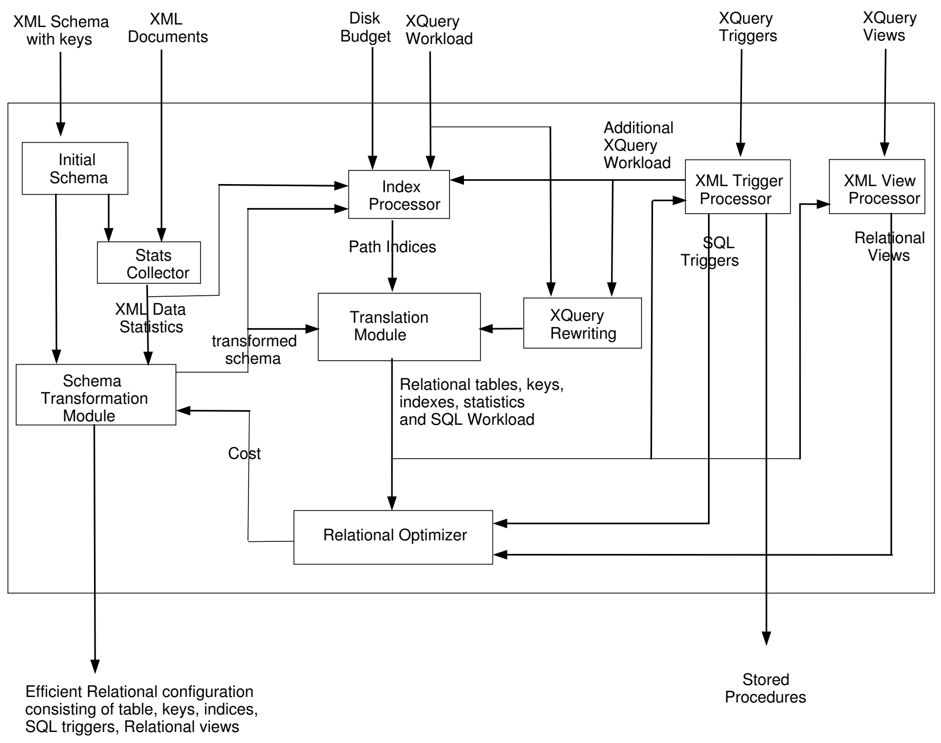

Figure 1: Architecture of the Elixir system

design more efficient schemas by merging relations of this initial schema.

Summary statistical information of the documents for the canonical schema is collected using the StatsCollector module. The estimated runtime cost of the XML workload, after translation to SQL, on this schema is determined by accessing the relational engine's query optimizer. Subsequently, the original XML schema is transformed in a variety of ways using various schema transformations, the relational runtime costs for each of these new schemas is evaluated, and the transformed schema with the lowest cost is identified. This whole process is repeated with the new XML schema, and the iteration continues until the cost cannot be improved with any of the transformed schemas. The choice of transformations is conditional on their adhering to the constraints specified in the XML schema, and this is ensured by the Translation Module.

In each iteration, the Index Processor component, selects the set of XML path-indices that fit within the disk space budget[1], and deliver the greatest reduction in the query runtime cost. These path indices are then converted to an equivalent set of relational indices. The XQuery queries are also rewritten to benefit from the path indices, with the query rewriting based on the concept of *path equivalence classes* [12] of XML Schema.

The XML Trigger Processor is responsible for handling all XML triggers – it maps each trigger to either an equivalent SQL trigger, or if it is not mappable (as discussed in [10]), represents it with a stored procedure that can be called by the middleware at runtime. To account for the cost of the non-mappable triggers, queries equivalent to these triggers are added to the input query workload.

Finally, the XML View Processor maps XML views and materialized XML views specified by the user to relational views and materialized query tables, respectively.

To implement the prototype of the above architecture, we have consciously attempted, wherever possible, to incorporate the ideas previously presented in the literature. Specifically, for schema transformations, we leverage the LegoDB framework [1], with its associated FleXMap [11] search tool and StatiX [4] statistics tool; the Index Processor component is based on the XIST path-index selection tool [12]; and, the DB2 relational engine [5] is used as the backend.

As an example, a sample fragment of a relational mapping derived from Elixir for an XML banking application is shown in Figure 2, including table definitions, key constraints, index selections, views, and triggers.

[1]Disk usage is measured with respect to the space taken by the equivalent *relational* indices.

– – XML Schema

```
<xsd:schema>
  <xsd:element name="bank">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="country"
                 type="CountryType"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  ...
</xsd:schema>
```

– – XML Documents

```
<?xml version="1.0"?>
<bank>
    <country>
        <name>India</name>
        ...
    </country> ...
</bank>
...
```

– – XML Query workload

```
FOR $customer IN //customer
FOR $account IN //account
WHERE ($customer/acc-number =
                $account/sav-acc-num
OR $customer/acc-number =
                $account/check-acc-num)
AND $customer/cust-id = '1000'
return <balance>$account/balance</balance>
# Frequency 20000
FOR $country IN /bank/country
WHERE $country/name/text() = "INDIA"
UPDATE $country/city
 { INSERT <name>Nasik</name>  ...}
# Frequency 100
```

– – XQuery Triggers

```
CREATE TRIGGER NewCityTrigger
AFTER INSERT OF /bank/country/city
FOR EACH NODE DO (...)

...
```

– – XML Views

```
CREATE VIEW important_customer AS
FOR $customer IN //customer
FOR $account IN //account
WHERE ($customer/acc-number =
            $account/sav-acc-num
OR $customer/acc-number =
            $account/check-acc-num)
AND $account/balance > 100000
return <balance>$account/balance</balance>

...
```

– – Materialized XML views

```
CREATE MATERIALIZED VIEW customer_balance AS
FOR $customer IN //customer
FOR $account IN //account
WHERE $customer/acc-number =
            $account/savings-acc-number
OR $customer/acc-number =
            $account/checking-acc-number
return
  <customer-balance>
    <id>$customer/cust-id</id>
    <acc-num>$customer/acc-number</acc-num>
    <balance>$customer/balance</balance>
  </customer-balance>
DATA INITIALLY IMMEDIATE REFRESH IMMEDIATE

...
```

(a) Input

– – Tables

```
CREATE TABLE Customer (Customer-id-key INTEGER
PRIMARY KEY, id INTEGER NOT NULL, name VARCHAR(25),
address VARCHAR(25), acc-number INTGER NOT NULL,
parent-Country INTEGER, parent-City INTEGER);
CREATE TABLE Account (Account-id-key INTEGER
PRIMARY KEY, sav-or-check-account-number INTEGER,
parent-Country INTEGER, Balance DECIMAL(10,2));
...
```

– – Relational keys equivalent to XML keys

```
ALTER TABLE Customer ADD CONSTRAINT Customer-key
UNIQUE (id, parent-Bank);
ALTER TABLE Account ADD CONSTRAINT Acc-key UNIQUE
(sav-or-check-acc-num, parent-Country);
ALTER TABLE Customer ADD CONSTRAINT Acc-fkey
FOREIGN KEY (acc-number, parent-Country) REFERENCES
Account(sav-or-check-acc-num, parent-Country);
...
```

– – Recommended Indices

```
CREATE INDEX name-index ON Customer(name);
CREATE INDEX acc-num-index ON
Account(sav-or-check-acc-num, parent-Country);
...
```

– – SQL Triggers

```
CREATE TRIGGER Increment-Counter
AFTER INSERT ON Customer
REFERENCING NEW AS new_row
FOR EACH ROW
BEGIN ATOMIC
  UPDATE Branch-office
  SET Acc-counter = Acc-counter + 1
  WHERE Branch-office.Id = new_row.Branch
END

...
```

– – Stored Procedure

```
CREATE PROCEDURE NewCityTrigger
  (IN cust-name STRING, IN city-name STRING,
  IN city-state STRING,...)
BEGIN
  Send-mail(cust-name, city-name, city-state, ...)
END

...
```

– – Relational views

```
CREATE VIEW important_customer AS
    (SELECT C.id, C.acc-number, A.balance
     FROM   Customer C, Account A
     WHERE  C.acc-number = A.sav-or-check-acc-number
     AND    A.balance > 10000)

...
```

– – Materialized Query Tables

```
CREATE TABLE customer_balance AS
 (SELECT C.id, C.acc-number, A.balance
  FROM   Customer C, Account A
  WHERE  C.acc-number = A.sav-or-check-acc-number)
DATA INITIALLY IMMEDIATE
REFRESH IMMEDIATE


...
```

(b) Output

Figure 2: Example Elixir Mapping

3 Integrated Schema-Centric Approach

In producing XML-to-relational mappings, there are two possibilities: A *source-centric* approach, wherein the optimization of the mapping is carried out in the XML space, and then translated to the equivalent in the relational space; or a *target-centric* approach, where a mapping is made from the XML space to the relational space, and then optimized in the relational space to fine-tune the mapping. A key design feature of Elixir is that it performs *all* its mapping-related optimizations in the XML source space, rather than in the relational target space. The evaluation of the quality of these optimizations is done at the target, and the feedback is used to guide the optimization process in the XML space, in an iterative manner, resulting in a *dynamically-derived* mapping tuned to the application. This approach is based on our observation that an organic understanding of the XML source can result in more informed choices from the performance perspective.

A case in point of the above approach is that Elixir identifies a good set of indices in the XML space and then maps them to equivalent indices in the relational space. This is marked contrast to the industrial practice recommended in [2], where the relational engine's index advisor is used to arrive at the index choices. For finding good XML indices, Elixir leverages the recently proposed XIST tool [12], which makes *path-index* recommendations, given an input consisting of an XML schema, query workload, data statistics, and disk budget. However, XIST does not make use of semantic information such as keys from the XML schema. As keys and the choice of path indices are closely related, we have extended XIST to use the information about keys by giving priority to the paths corresponding to keys during the index selection process. An additional benefit of source-based index choices is that the knowledge can be used to guide the XQuery-to-SQL translation during query processing (details in [10]). Finally, our experimental results in [10] quantitatively demonstrate that the source-centric approach is preferable to a target-centric approach.

In principle, the collection of techniques incorporated in Elixir can each be applied independently for mapping specific input features from the XML world to their relational counterparts, and thereby produce holistic schemas. However, this can result in inefficient performance due to not taking their inherent relationships into account – for example, generating an optimized relational mapping and then defining triggers on this mapping can be significantly worse than intrinsically considering the triggers during the optimization process. Therefore, Elixir consciously takes an *integrated* approach to producing efficient holistic schemas – for example, the choice of path indices is dependent on the XML keys, while the choice of schema transformations is influenced by the presence of XML triggers and views. This integration ensures that all the interactions between the XML inputs and their impact on the relational outputs are automatically taken into account during the optimization process.

4 Demonstration

To the best of our knowledge, the Elixir system is the first to attempt production of holistic XML-to-RDBMS mappings. In this demonstration, we will provide a walkthrough of its features, and explain how its cost-based and source-centric approach to mapping can result in efficient relational schemas that are tuned to the application workload.

Acknowledgements

This work was supported in part by a Swarnajayanti Fellowship from the Dept. of Science & Technology, Govt. of India.

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 IEEE ICDE*, 2002.

[2] S. Chaudhuri, Z. Chen, K. Shim and Y. Wu. Storing XML (with XSD) in SQL Databases: Interplay of Logical and Physical Designs. In *Proc. of IEEE ICDE*, 2004.

[3] A. Conrad. A survey of MS-SQL Server 2000 XML features. *http://msdn.microsoft.com/library/en-us/dnexxml/html/xml07162001.asp?frame=true*.

[4] J. Freire, J. Haritsa, M. Ramanath, P. Roy and J. Siméon. Statix: Making XML count. In *Proc. of ACM SIGMOD*, 2002.

[5] IBM DB2 XML Extender. *http://www-3.ibm.com/software/data/db2/extenders/xmlext/library.html*.

[6] H. Jagadish et al. TIMBER: A Native XML Database. *The VLDB Journal*, 11(4), 2002.

[7] R. Krishnamurthy, V. Chakaravarthy and J. Naughton. On the Difficulty of Finding Optimal Relational Decompositions for XML Workloads: a Complexity Theoretic Perspective. In *Proc. of ICDT*, 2003.

[8] Objective Caml. *http://caml.inria.fr/ocaml/*.

[9] Oracle XML DB. *http://technet.oracle.com/tech/xml/content.html*.

[10] P. Patil and J. Haritsa. Holistic Schema Mappings for XML-on-RDBMS. Tech. Rep. TR-2005-02, DSL, Indian Institute of Science. *http://dsl.serc.iisc.ernet.in/publications/report/TR/TR-2005-02.pdf*.

[11] M. Ramanath, J. Freire, J. Haritsa and P. Roy. Searching for efficient XML-to-relational mappings. In *Proc. of XSym*, 2003.

[12] K. Runapongsa, J. Patel, R. Bordawekar and S. Padmanabhan. XIST: An XML Index Selection Tool. In *Proc. of XSym*, 2004.

[13] Tamino. *http://www1.softwareag.com/Corporate/products/tamino/prod_info/default.asp*.