Holistic Schema Mappings for XML-on-RDBMS

Priti Patil* and Jayant R. Haritsa

Database Systems Lab, SERC,
Indian Institute of Science, Bangalore 560012, INDIA

Abstract. When hosting XML information on relational backends, a mapping has to be established between the schemas of the information source and the target storage repositories. A rich body of recent literature exists for mapping *isolated* components of XML Schema to their relational counterparts, especially with regard to table configurations. In this paper, we present the Elixir system for designing "industrial-strength" mappings for real-world applications. Specifically, it produces an *information-preserving holistic* mapping that transforms the complete XML world-view (XML schema with constraints, XML documents XQuery queries including triggers and views) into a full-scale relational mapping (table definitions, integrity constraints, indices, triggers and views) that is tuned to the application workload. 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. Further, unlike the XML mapping tools of commercial database systems, which rely heavily on user inputs, Elixir takes a principled cost-based approach to automatically find an efficient relational mapping. A prototype of Elixir is operational and we quantitatively demonstrate its functionality and efficacy on a variety of real-life XML schemas.

1 Introduction

For persistently storing information from XML sources, there are primarily two technological choices available: A specialized native XML store (e.g. Tamino [25], Natix [11], Timber [10]), or a standard relational engine (e.g. IBM DB2 [20], Oracle [24], MS-SQL Server [22]). 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 [17] or XML Schema [29]) and relational schema. A common feature of much of this work is that it has focused on *isolated* components of the relational schema, typically the table configurations. However, viable XML-to-relational systems that intend to support real-world applications will need to provide an *information-preserving holistic* mapping that transforms the complete XML world-view (XML schema with constraints, XML documents, XQuery

* Currently with IBM India Software Lab.

M.L. Lee, K.L. Tan, and V. Wuwongse (Eds.): DASFAA 2006, LNCS 3882, pp. 741–756, 2006.
© Springer-Verlag Berlin Heidelberg 2006

queries including triggers and views) into a full-scale relational schema (table definitions, integrity constraints, indices, triggers and views). In this paper, we address this issue by presenting a system called ELIXIR (Establishing hoLIstic schemas for XML In Rdbms) which produces such "industrial-strength" XML-to-RDBMS mappings.

By taking a principled cost-based approach to mapping design, Elixir *automatically* delivers efficient mappings that are *tuned* to the XML application. This is in marked contrast to the XML 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. For example, in DB2's XML Extender, the user needs to have intimate knowledge of the application to specify mapping of each XML node to either a table or a column using the Document Access Definition (DAD) medium [20].

A novel 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 database engine, and the feedback is used to guide the optimization process in the XML space, in an iterative manner, resulting in a *dynamically-derived* mapping that is *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 a performance perspective – as a case in point, making index choices at the XML source and then mapping them to relational equivalents proves to be substantially better than directly using the relational engine's index advisor, which is the current industrial practice [6]. 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, consistent with the observation in [12] that schema decomposition and query translation are interdependent and should therefore be handled in an integrated manner.

A related feature of Elixir is its *integrated* approach to producing efficient holistic schemas – for example, the choice of indices is affected by the XML constraints. This integration ensures that all the interactions between the XML inputs and the effects of these inputs on the relational outputs are automatically taken into account during the optimization process.

Currently, a prototype of Elixir is operational on the DB2 relational engine [20], and can be easily ported to any standard RDBMS. The prototype is implemented in Ocamlc (Objective Caml) [23], a strongly-typed functional programming language, and has been successfully evaluated on a variety of real-world and synthetic XML schemas [29] for representative XQuery [2] queries. To make our objectives concrete, a sample fragment of inputs from an XML banking application and a relational mapping derived from Elixir for these inputs is shown in Figure 1.

To the best of our knowledge, Elixir is the first system to aim towards delivering industrial-strength mappings for XML-to-RDBMS. In the remainder of this paper, we describe its highlights – the complete technical details are available in [14].

```
– – XML Schema
<xs:element name="country" type="CountryType"
          minOccurs="0" maxOccurs="unbounded">
 <xs:key name="acc-num-key">
   <xs:selector xpath=".//account"/>
   <xs:field xpath="./sav-acc-num |
                  ./check-acc-num"/>
 </xs:key>
 <xs:keyref name="cust-acc" refer="acc-num-key">
   <xs:selector xpath=".//customer"/>
   <xs:field xpath="./acc-num"/>
 <xs:keyref>
</xs:element>
...

– – XML Documents

<bank>
   <country>
      <name>India</name>
      <customer>
         <cust-id>1</cust-id>
         <acc-num>101</acc-num> ...
      </customer> ...
      <city>
         ...
         <account>
            <sav-acc-num>101</sav-acc-num>
            <balance>1232423</balance>
         </account>
      </city> ...
  </country> ...
</bank>

– – XML Query workload

FOR $cust IN //customer
FOR $acc IN //account
WHERE ($cust/acc-num = $acc/sav-acc-num
OR $cust/acc-num = $acc/check-acc-num)
AND $cust/cust-id = '1000'
return <balance>$acc/balance</balance>
# Frequency 20000

– – XQuery Triggers

CREATE TRIGGER Increment-Counter
AFTER INSERT OF //Customer
...
CREATE TRIGGER NewCityTrigger
AFTER INSERT OF /bank/country/city
...

– – XML Views

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

(a) Input

```
– – Tables
CREATE TABLE Customer (Cust-id-key
INTEGER PRIMARY KEY, id INTEGER NOT NULL,
name VARCHAR(25),...);
CREATE TABLE Account (Acc-id-key
INTEGER PRIMARY KEY, ...);
...
– – Relational keys equivalent to XML keys

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-num, 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(...)
BEGIN
  Send-mail(cust-name, city-name, ...)
END

...
– – Relational views

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

...
```

(b) Output

Fig. 1. Example Elixir Mapping

2 Architecture of Elixir System

The overall architecture of the Elixir system is depicted in Figure 2. 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 [9], corresponding to an extremely fine-grained relational mapping, and in the rest of the procedure attempts to 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 (measured with respect to the equivalent *relational* indices), and deliver the greatest reduction in the query runtime cost. These path indices are then converted to an equivalent set of

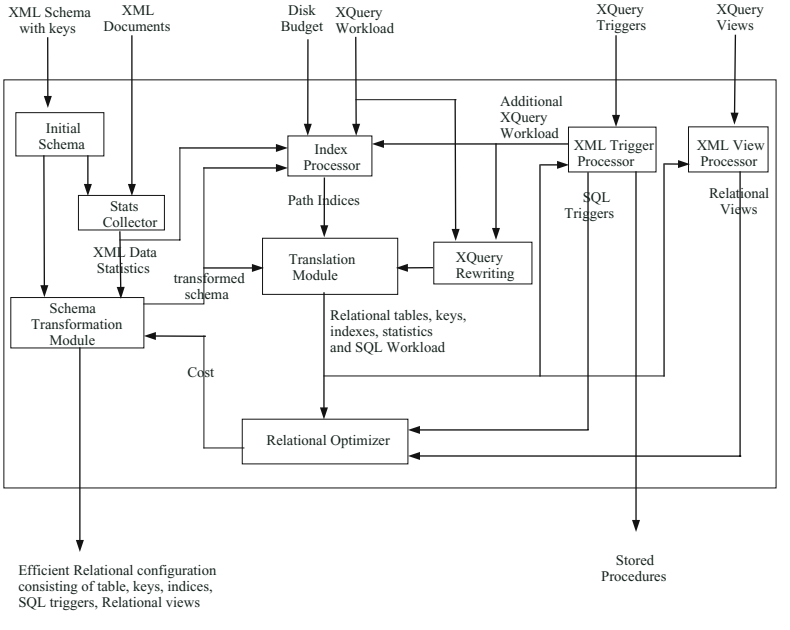


Fig. 2. Architecture of the Elixir system

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* [16] 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 Section 5), 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 above architecture, we have consciously attempted, wherever possible, to incorporate the ideas and systems previously presented in the literature. Specifically, for schema transformations, we leverage the LegoDB framework [3], with its associated FleXMap search tool [15] and StatiX [9] statistics tool; the Index Processor component is based on the XIST path-index selection technique [16]; and, the DB2 relational engine [20] is used as the backend.

In the following sections, we discuss in detail the generation of the various components of the holistic relational schema, including Table Configurations, Key Constraints, Indices, Triggers and Views.

3 Generating Constraint-Preserving Relations

XML Schema supports a rich set of integrity and cardinality constraints. The *Translation Module* takes an XML schema with such constraints as input and produces a constraint-preserving equivalent relational schema. For example, XML Schema supports three integrity constraints: *unique*, *key* and *keyref*, with similar semantics to their relational counterparts – *unique* ensures no duplication among non-null values; *key* ensures all values are unique and non-null; and *keyref* ensures reference to XML nodes. Due to hierarchical data model of XML, *context* is also specified for integrity constraints to define the different sets of nodes to be distinguished.

Using the syntax of [5], example constraints for the sample *bank.xml* document shown in Figure 3 are given below:

- *acc-num-key*: (//country,(.//account, {sav-acc-num | check-acc-num}))
  Within a country (here country is a *context*), each account is uniquely identified by a savings or checking account number.
- *cust-acc*: (//country,(./customer,{acc-num})) KEYREF *acc-num-key*
  Within a country, each customer refers to a savings or checking account number by acc-num.

An obvious way of supporting XML constraints in an RDBMS is to use triggered procedures, but this is highly inefficient [8], and should therefore only be used for those constraints (such as cardinality constraints) that do not have a relational equivalent. Specifically, the XML *key* and *keyref* constraints should

```
<bank>
    <country>
        <name>India</name>
        <customer>
            <cust-id>1</cust-id>
            <acc-num>101</acc-num> ...
         </customer> ...
         <city>
            <name>Bangalore</name>
            <state>Karnataka</state>
            <head-office> ... </head-office>
            <branch-office> ... </branch-office> ...
            <atm> ... </atm> ...
            <account>
                <sav-acc-num>101</sav-acc-num>
                <balance>1232423</balance>
            </account>
            <account>
                <check-acc-num>102</check-acc-num>
                <balance>645634</balance>
            </account>...
        </city> ...
    </country> ...
</bank>
```

Fig. 3. Sample XML Document (bank.xml)

```
TABLE Account(
    Acc-id-key INT,
    sav-acc-num INT,
    check-acc-num INT,
    balance INT,
    parent-City INT)
```

(a) Using LegoDB mapping

```
TABLE Account(
    Acc-id-key INT,
    sav-or-check-acc-num INT,
    parent-Country INT,
    acc-num-flag INT,
    balance INT,
    parent-City INT)
```

(b) Inclusion of relational key

Fig. 4. Generating relational keys for XML key – *acc-num-key*

be mapped to relational key and foreign-key constructs. We have developed a three-step algorithm for implementing this mapping – this technique is superficially similar to the X2R storage mapping algorithm [7], but a crucial difference is that they tailor the schema to fit the key constraints, thereby risking efficiency, whereas we take the opposite approach of integrating the key constraints with an efficient schema.

Specifically, Elixir starts by converting the XML schema into the *schema tree* representation proposed in FleXMap [15]. Then, in the first step, subtrees corresponding to different paths that need to be mapped to a single column are "associated", with the need for association determined from the XML keys. For example, for *acc-num-key*, the subtrees corresponding to sav-acc-num and check-acc-num have to be associated. In the next step, the XML-to-relational mapping procedure proposed in [3] is extended to create table configurations in the presence of the associated trees. After mapping the XML schema to tables, the final step is to incorporate the relational keys that are equivalent to the original XML keys.

An example output for the initial generic mapping of Figure 4(a)) is shown in Figure 4(b). Here, the elements sav-acc-num and check-acc-num are mapped to a single column `sav-or-check-acc-num`, and an additional column, `acc-num-flag`, is created for identifying the account number type. Further, since the context element for *acc-num-key* is country, which is not an immediate parent of **Account**, a `parent-Country` column, which refers to `country-id-key`, is added to distinguish between different contexts.

Similarly we can define an equivalent relational foreign key for the *cust-acc* XML keyref. Specifically, create the following relation:

```
TABLE Customer (Cust-id-key INT, Cust-id INT, Name STRING, Address
STRING, Acc-num STRING, parent-Country INT)
```

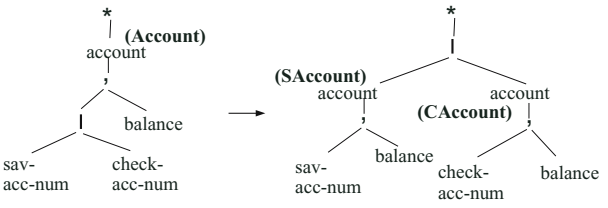


Fig. 5. Invalid union distribution due to *acc-num-key* constraint

where the foreign key is {`Acc-num`, `parent-Country`}, referring to the key attribute pair of the `Account` relation.

Cost-based strategies, such as those proposed in [3], explore the optimization space by applying various transformations to the XML schema (which exploit the standard rules of regular expressions in XML Schema for *unions* and *repetitions*), and evaluating the costs of the corresponding relational configurations. Elixir restricts the mapping search space to only *constraint-valid schema trees* by filtering out the invalid schema transformations. For example, consider the union account = sav-acc-num | check-acc-num shown before and after distribution in Figure 5. The corresponding relational configuration will have account numbers stored in two relations as follows:

```
TABLE SAccount(SAcc-id-key INT, sav-acc-num INT, balance INT, parent-City
INT)
TABLE CAccount(CAcc-id-key INT, check-acc-num INT, balance INT,
parent-City INT)
```

Here our goal is to map the XML key and keyref in the form of primary key and foreign key, respectively. However, according to the *acc-num-key* constraint, sav-acc-num and check-acc-num should be mapped to a single column, in order to define the relational key, thereby rendering the union distribution invalid.

This example shows that not all relational configurations obtained by schema transformations are valid. Thus, while exploring the search space of relational configurations, we should explore only the space of valid configurations. The simple solution for this is to carry out the transformation on the schema tree and then check if relational keys equivalent to the given XML constraints can be defined on the resulting relational configuration. If it is not possible then that relational configuration can be ignored, otherwise it should be evaluated for the given query workload. However, this solution results in considerable unnecessary work, which can be avoided if we can detect the invalidity schema transformations *before* carrying out the schema transformation.

For example, assume that union $t_1|t_2$ is being distributed, where t_1 and t_2 are subtrees of the schema tree. Now we will try to analyze the cause for invalidation. Note that both the subtrees, corresponding to sav-acc-num (t_1) and check-acc-num (t_2), are on the same field path of the *acc-num-key* constraint. Thus, if the union distribution of this tree i.e. $t_1|t_2$ is distributed, then in the resulting configuration, t_1 and t_2 will be mapped to different relations. In general, **if subtrees t_1 and t_2 are both on the same field path, then union distribution of $t_1|t_2$**

is invalid. The complete set of rules to detect when schema transformations are invalid w.r.t. XML schema constraints is given in [14]. A useful side-effect of incorporating the constraints during the schema design process is that the mapping process completes faster due to the reduction of the optimization search space.

4 Index Selection in Elixir

We move on in this section to a different component of the holistic mapping, namely deciding on the best choice of relational *indices*, given a disk space budget. As mentioned earlier, Elixir takes the approach of finding a good set of indices in the XML space and then mapping them to equivalent indices in the relational space. This is in marked contrast to current industrial practice [6], where the index advisor of the relational engine is used to propose a good set of indices after the schema mapping has been carried out.

For finding good XML indices, we leverage the recently proposed XIST tool [16], which makes *path-index* recommendations given an input consisting of an XML schema, query workload, data statistics, and disk budget. We have extended XIST to make use of semantic information such as keys, which are closely linked to index selection, by giving priority to the paths corresponding to keys during the index selection process. This is in keeping with Elixir's general philosophy of exploring the *combined* search space of logical design (i.e. schema transformations) and physical design (i.e. indices) since solving them independently leads to suboptimal performance [6].

After making the choice of XML path-indices, a strategy to convert path indices to the equivalent relational indices has to be designed. Secondly, the disk usage of the *relational indices* should be within the user-specified budget – therefore, an equivalence mapping between the disk occupancies in the XML and relational spaces has to be formulated. Finally, the XQuery-to-SQL translation process should take advantage of the presence of the relational indices. In the remainder of this section, we describe our approach to handle the first and third issues – the second issue is discussed in [14].

4.1 Path Index to Relational Index Conversion

Consider an XML-to-relational mapping, as shown in Figure 6 for a fragment of the XMark benchmark schema [28]. Here, a non-leaf node is annotated with a relation name, while a leaf node is annotated with the name of a relational column. Relations `Site`, `Africa`, ..., `Samerica`, `Item`, and `Mailbox` are created for elements site, africa, ..., samerica, item, mailbox, respectively. For this environment, assume that the following path index, PI, has been recommended: /site/regions/africa/item/mailbox/mail/from. To evaluate PI, the four relations {`Site`, `Africa`, `Item`, and `Mailbox`} have to be joined.

An obvious translation process is to simply build the indices on the key and foreign-key pair for each parent-child involved in PI. However, the drawback of

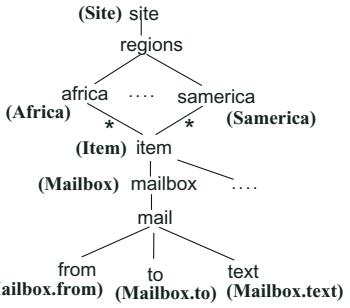


Fig. 6. Example relational configuration

1. `Mailbox.parent-africa`
2. `Mailbox.from`

(a) Equivalence Class-based Approach

1. `Africa.parent-Site`
2. `Item.parent-Africa`
3. `Mailbox.parent-Item`
4. `Mailbox.from`

(b) Direct Approach

Fig. 7. Relational indices for path index /site/regions/africa/item/mailbox/mail/from

this direct approach is that the number of relational indices created for a path-index is a function of the *path-length*, and can therefore become very expensive to create and maintain. An alternative and less expensive approach is to use the concept of *equivalence classes* [16] to reduce the number of relational indices. Two paths P_1 and P_2 are in the same equivalence class if the evaluation of both paths against XML data results in selection of the same nodes. These equivalence classes can be determined directly from the XML schema and are valid for all XML documents conforming to the XML schema.

We have developed a procedure (details in [14]) that uses these path equivalence classes (EQs) to convert the path-index only to the relational indices corresponding to each EQ on the path. For example, if we assume that for each relation, the column which stores IDs is the primary key, and that an index exists on the primary key by default, then the equivalent relational indices for PI are as shown in Figure 7(a) (for comparative purposes, the indices recommended by the Direct approach are shown in Figure 7(b)).

4.2 Query Rewriting for Path Indices

The use of integrity constraints to guide XQuery-to-SQL query translation has been recently discussed in [13]. Here, we focus on the use of available path indices to guide XQuery-to-SQL query translation, and thereby derive a more efficient rewriting of the query. For example, consider the query:

```
for    $mail = /site/regions/africa/item/mailbox/mail
where $mail/from/text() = "priti@dsl.serc.iisc.ernet.in"
return count($mail)
```

The relevant path P here is /site/regions/africa/item/ mailbox/mail/from. If there is no path index on P, then the SQL translation of the above query will be:

```
select  count(*)
from   Site S, Africa A,  Item I, Mailbox M
```

```
where S.site-key = A.parent-site
      and A.africa-key = I.parent-africa
      and I.item-key = M.parent-item
      and M.from = 'priti@dsl.serc.iisc.ernet.in'
```

On the other hand, if a path index on P is available, the translation module uses this information to translate the query as follows:

```
select count(*)
from   Africa A, Mailbox M
where A.africa-key = M.parent-africa
      and M.from = 'priti@dsl.serc.iisc.ernet.in'
```

While the above was an illustrative example, the complete algorithm for incorporating indices in the XQuery-to-SQL translation process is given in [14].

5 Mapping XML Triggers and Views

We now move on to the advanced components of XML triggers [4] and XML views [1]. Triggers are primarily used to execute a specific logic upon updates to the database. To leverage the power of relational databases, our aim is to map the XML triggers to relational triggers, an example of which is shown in Figure 8.

```
CREATE TRIGGER Increment-Counter
AFTER INSERT OF //CUSTOMER
FOR EACH NODE
 LET $branch_id = NEW_NODE/branch
 LET $branch_node =
          //branch-office[id=$branch_id]
 LET $counter = $branch_node/acc-counter
DO (
FOR $branch_node
UPDATE $branch_node
REPLACE $counter WITH $counter + 1 )
```

(a) XML trigger

```
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
```

(b) Equivalent SQL trigger

Fig. 8. Mapping XML triggers to SQL triggers

A problem specific to the XML domain, however, is that compared to relational updates, XQuery updates may be seen as *bulk* statements since they may involve arbitrarily large fragments of documents that are inserted or dropped through a single statement. For example, when a bank sets up operations in a new city, the corresponding XQuery update could result in several SQL insert statements on the tables corresponding to the update path.

In this situation, consider the following XML trigger, sending e-mail to advertise the new office to all customers from the same country as the inserted city:

```
CREATE TRIGGER NewCityTrigger AFTER INSERT OF /bank/country/city
FOR EACH NODE DO (
   LET $city-name = NEW_NODE/name
   LET $city-state = NEW_NODE/state
   LET $city-head-office-id = NEW_NODE/head-office-id
   LET $city-branch-offices = NEW_NODE/branch-office
   ...
   FOR $customer IN NEW_NODE/../country/customer
     send-email ($customer, $city-name, $city-state,
                 $city-head-office-id, $city-branch-offices, ...)  )
```

The above trigger needs to be executed after all the insert statements to the `City`, `Branch-office`, `Office-Id`, `Atm`, `Account` relations have been executed. However, in the current SQL standard, triggers cannot be specified relative to a set of operations on different tables. We refer to such triggers as *non-mappable XML triggers* and model them instead as stored procedures that can be called by the middleware at runtime.

While the costs of mappable triggers are natively modeled by the relational optimizer, an additional query workload equivalent to the non-mappable triggers is included in the XML query workload. Our experiments have shown that in practice XML triggers play an important role in determining the choice of the final relational configuration.

Turning our attention to XML views, Elixir maps these views to relational views by first converting the XML view definition to the equivalent SQL view definition, and then translating XQuery queries on the XML views to SQL queries on relational views. Additionally, if the user specifies a *materialized* XML view, then this view is mapped to materialized relational views. The complete mapping algorithm is given in [14], and an illustrative example is shown below.

Consider a user specifying the following materialized XML view to make the balance inquiry query execute faster:

```
CREATE MATERIALIZED VIEW customer_balance AS
    FOR $customer IN //customer
    FOR $account IN //account
    WHERE $customer/acc-num = $account/sav-acc-num or
          $customer/acc-num = $account/check-acc-num
    return
        <customer-balance>
            <id>$customer/cust-id</id>
            <acc-num>$customer/acc-num</acc-num>
            <balance>$customer/balance</balance>
        </customer-balance>
DATA INITIALLY IMMEDIATE REFRESH IMMEDIATE
```

Elixir maps this XML materialized view to the following equivalent relational materialized view:

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

6 Experimental Evaluation

In this section, we present our experimental evaluation of the Elixir system. Our experiments were run on a Pentium-IV PC running Linux, with DB2 UDB v8.1 [20] as the backend database engine. Four representative real-world XML schemas: *Genex* [19], *EPML* [18], *ICRFS* [21] and *TourML* [26], which deal with gene expressions, business processes, enterprise analysis and tourism, respectively, are used in our study. In addition, we also evaluate the performance for the synthetic XMark benchmark schema [28].[1]

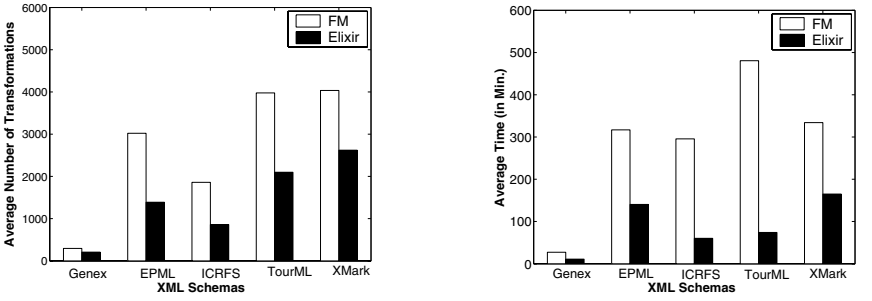
6.1 Effect of Keys

To serve as a baseline for assessing the effect of key inclusion, we compare the performance of Elixir (in the absence of indices, triggers, and views) with that of FleXMap (FM) [15], a framework for expressing XML schema transformations and for searching the equivalent relational configuration space. Using the ToXgene tool [27], three types of documents were generated for each XML schema by varying the distribution of elements as *all-uniform*, *uniform-exponential*, and *all-exponential*, resulting in documents with uniform data, moderately skewed data, and highly skewed data, respectively. The query workload involves 10 representative queries for each XML schema.

We compare the runtime efficiency of Elixir and FleXMap with regard to the following metrics: (a) The percentage reduction in search space, and (b) The response time speedup due to this reduction. The average number of transformations evaluated by Elixir and FM are shown in Figure 9(a) for the five XML schemas. We see there that the reduction ranges from 30% to 60%, arising from the filtering out of invalid transformations, discussed in Section 3. For example, on the ICRFS schema, the average number of transformations performed by FleXMap are about 1860, whereas Elixir only requires about 860.

The time speedup due to the search space reduction is shown in Figure 9(b), which captures the average time required to obtain the final relational configuration for the same set of schemas. Here, we observe that the runtime reductions range from 50% to 85%. It is interesting to note that the speedup is *super-linear* in the percentage space reduction. For example, the 50% search space reduction for ICRFS may be expected to result in a speedup of 2, but the speedup actually obtained is greater than 4. The reason for this is as follows: A given XQuery workload satisfies more paths in the fully decomposed schema of FleXMap resulting in *more subqueries* in the equivalent SQL workload, as compared to the

[1] Since XMark is available only as a DTD, we created the equivalent XML Schema and incorporated keys by mapping the IDs and IDREFs.



(a) Search space

(b) Time efficiency

Fig. 9. Impact of Keys

number derived from the restrictive decomposed schema of Elixir. Thus, the time required for evaluating the cost of an individual transformation using the relational optimizer is more for FleXMap than for Elixir. In a nutshell, Elixir has "fewer and cheaper" transformations.

6.2 Effect of Index Selection

We now move on to evaluating the impact of index selection. Specifically, we compare Elixir, with its path-index-based selection, against two alternatives: *BasicDB2*, where the system has only its default primary key indices, and *DB2Advisor*, where DB2's Index Advisor tool is used to suggest a good set of indices, similar to [6].

We report here the results of experiments on the EPML schema [18] with various sizes of XML documents ranging from 1 MB to 500 MB. The query workload involves 20 representative queries. The index disk budget was set to be 10 percent of the space occupied by the XML document repository, a common rule-of-thumb in practice. The results for this set of experiments are shown in Figure 10, where we see that the cost of the final relational configuration is significantly lower for Elixir as compared to *BasicDB2* as well as *DB2Advisor*. The results obtained on other schemas were similar and are available in [14].

Analysis of the set of indices chosen by Elixir and *DB2Advisor* indicates the following: The SQL workload equivalent to the given XQuery workload involves several joins. DB2 attempts to improve the query performance by creating multicolumn indexes or single column indexes (with `include` clause). Elixir, on the other hand, uses the path indices suggested by XIST and converts path indices to equivalent single column relational indices. Further, the sets of indexes chosen by DB2Advisor and Elixir are quite different in that the overlap is only between 20% to 50%.

6.3 Overall Performance of Elixir System

While the previous experiments evaluated the performance in isolation for various components (the trigger and view performance is available in [14]), the

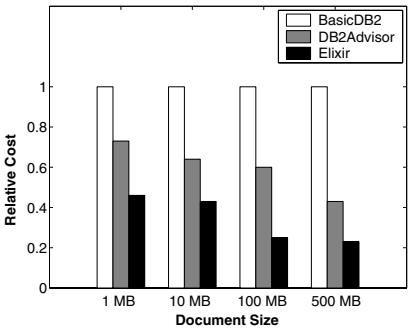


Fig. 10. Index selection

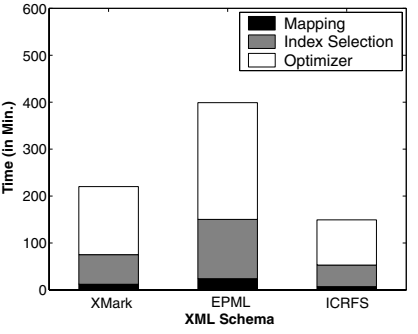


Fig. 11. Elixir Performance

overall performance when all components are integrated is shown in Figure 11. This figure shows both the total time for producing the final relational schema as well as the breakup of this time in different steps of the tuning process – Mapping, Index selection, and Optimizer. With regard to overall time, we find that it is in the range of a few hours for each schema. While this may seem excessive at first glance, note that (a) the schema generation process is typically a one-shot process, and therefore time may not be a major issue; and, further, (b) the breakup of the runtime indicates significant potential for improvement – the heavy overhead (60% to 70%) is largely attributable to our using the optimizer *from the outside*, which involves fresh creation of tables, loading of statistics, costing the mapping and table deletion, in *each* iteration of the mapping process. We expect that this overhead would be substantially reduced if the Elixir system were implemented inside the relational engine since the optimizer could be instrumented to directly provide the cost for the new mappings. Finally, note that due to the absence of comparable systems for producing holistic schemas, we only provide absolute performance results here.

7 Conclusions and Future Work

In this paper, we studied the problem of producing *information-preserving holistic schema* mappings from XML repositories to relational backends. Specifically, we proposed the Elixir system, which captures most significant aspects of the XML world and delivers relational schemas that include table configurations, keys, indices, triggers, and views, featuring an integrated, cost-based and source-centric optimization of the mapping process. A detailed experimental study on real-world and synthetic schemas demonstrated the effectiveness of our techniques with regard to both the final quality of the relational configuration as well as the mapping time. In a nutshell, the Elixir system achieves "industrial-strength" mappings for XML-on-RDBMS providing lossless translation (structure and semantics including constraints and triggers) and performance tuning (indices and materialized views). Our future plans include implementation of the

Elixir system inside public-domain relational engines and extending the schema mapping to include security components.

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

References

1. S. Abiteboul. On Views and XML. In *Proc. of 18th ACM Symp. on Principles of Database Systems (PODS)*, May 1999.
2. S. Boag et al. XQuery 1.0: An XML Query Language, May 2001. `http://www.w3.org/TR/xquery/`.
3. P. Bohannon, J. Freire, P. Roy and J. Siméon. From XML schema to relations: A cost based approach to XML storage. In *Proc. of 18th IEEE Intl. Conf. on Data Engineering (ICDE)*, March 2002.
4. A. Bonifati, D. Braga, A. Campi and S. Ceri. Active XQuery. In *Proc. of 18th IEEE Intl. Conf. on Data Engineering (ICDE)*, February 2002.
5. P. Buneman, S. Davidson, W. Fan, C. Hara and W. Tan. Keys for XML. *Computer Networks*, 39(5), 2002.
6. 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 20th IEEE Intl. Conf. on Data Engineering (ICDE)*, March 2004.
7. Y. Chen, S. Davidson and Y. Zheng. Constraints preserving schema mapping from XML to relations. In *Proc. of 5th Intl. Workshop on Web and Databases (WebDB)*, June 2002.
8. Y. Chen, S. Davidson and Y. Zheng. Validating constraints in XML. Tech. Report MS-CIS-02-03, Dept. of Computer and Information Science, Univ. of Pennsylvania, 2002.
9. 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*, June 2002.
10. H. Jagadish et al. TIMBER: A Native XML Database. *VLDB Journal*, 11(4), 2002.
11. C. Kanne and G. Moerkotte. Efficient Storage of XML data. In *Proc. of 16th IEEE Intl. Conf. on Data Engineering (ICDE)*, February 2000.
12. 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 9th Intl. Conf. on Database Theory (ICDT)*, January 2003.
13. R. Krishnamurthy, R. Kaushik and J. Naughton. Efficient XML-to-SQL Query Translation: Where to Add the Intelligence? In *Proc. of 30th Intl. Conf. on Very Large Data Bases (VLDB)*, August 2004.
14. P. Patil and J. Haritsa. Holistic Schema Mappings for XML-on-RDBMS. Tech. Report `http://dsl.serc.iisc.ernet.in/publications/report/TR/TR-2005-02.pdf`.
15. M. Ramanath, J. Freire, J. Haritsa and P. Roy. Searching for efficient XML-to-relational mappings. In *Proc. of 1st Intl. XML Database Symp. (XSym)*, September 2003.
16. K. Runapongsa, J. Patel, R. Bordawekar and S. Padmanabhan. XIST: An XML Index Selection Tool. In *Proc. of 2nd Intl. XML Database Symp. (XSym)*, August 2004.

17. DTD. http://www.w3.org/XML/1998/06/xmlspec-report.
18. EPML (EPC Markup Language). http://wi.wu-wien.ac.at/~mendling/EPML/.
19. GENEX (Gene Expression Markup Language). http://www.ncgr.org/genex.
20. IBM DB2 XML Extender. http://www-3.ibm.com/software/data/db2/extenders/xmlext/library.html.
21. ICRFS (ICRFS XML schema). http://www.insureware.com/abouti/mlines.shtml.
22. A survey of MS-SQL Server 2000 XML features. http://msdn.microsoft.com/library/en-us/dnexxml/html/xml07162001.asp?frame=true.
23. Objective Caml. http://caml.inria.fr/ocaml.
24. Oracle XML DB: An oracle technical white paper. http://technet.oracle.com/tech/xml/content.html.
25. Tamino. http://www1.softwareag.com/Corporate/products/tamino/prod_info/default.asp.
26. Tourism Markup Language. http://www.opentourism.org.
27. ToXgene (ToX XML Data Generator). http://www.cs.toronto.edu/tox/toxgene/.
28. XMark. http://monetdb.cwi.nl/xml/.
29. XML schema. http://www.w3.org/TR/xmlschema-1/.