Holistic Schema-Mappings for XML-on-RDBMS

Priti Patil Jayant R. Haritsa

Technical Report
TR-2005-02

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India

http://dsl.serc.iisc.ernet.in

Holistic Schema Mappings for XML-on-RDBMS

Priti Patil Jayant R. Haritsa

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

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 documents, XML schema with constraints, 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 input, 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 [31], Natix [20], Timber [19]), or a standard relational engine (e.g. IBM DB2 [17], Oracle [25], MS-SQL Server [11]). 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 [13] or XML Schema [35]) 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 documents, XML schema with constraints, XQuery 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) [17] medium.

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 [21] 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 [17], and can be easily ported to any standard RDBMS. The prototype is implemented in Ocamlc (Objective Caml) [24], a strongly-typed functional programming language, and has been successfully evaluated on a variety of real-world and synthetic XML schemas [35] for representative XQuery [2] queries. To make our objectives concrete, a sample fragment of inputs from 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 deliver industrial-strength mappings for XML-to-RDBMS.

Organization

The remainder of this paper is organized as follows: In Section 2, an overview of the Elixir system is presented. The constraint mapping technique and its integration with cost-based optimization is discussed in Section 3; index mapping is addressed in Section 4; and Section 5 explains the mapping procedure for XML triggers and XML views. The experimental framework and the results are highlighted in Section 6. Related work is reviewed in Section 7, and our conclusions are summarized in Section 8.

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 [15], 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[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* [28] 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 previously presented in the literature. Specifically, for schema transformations, we leverage the LegoDB framework [3], with its associated FleXMap [27] search tool and StatiX [15] statistics tool; the Index Processor component is based on the XIST path-index selection tool [28]; and, the DB2 relational engine [17] 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 XML to Relational Constraints

In this section, we first provide an overview of the various integrity constraints supported by XML Schema, and then present the procedure for mapping XML constraints to the equivalent constraints on a given relational schema.

3.1 XML Constraints

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.

To define a *unique* or *key* constraint for XML, the following factors have to be specified: 1) the context in which the key must hold; 2) the set of nodes on which the key is defined; and 3) the values, which distinguish each element of the set. To define a *keyref* constraint, the key to which it refers needs to be additionally specified.

Using the syntax of [5], the *unique* and *key* constraints are written as

[1]Disk usage is measured with respect to 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 1: Example Elixir Mapping

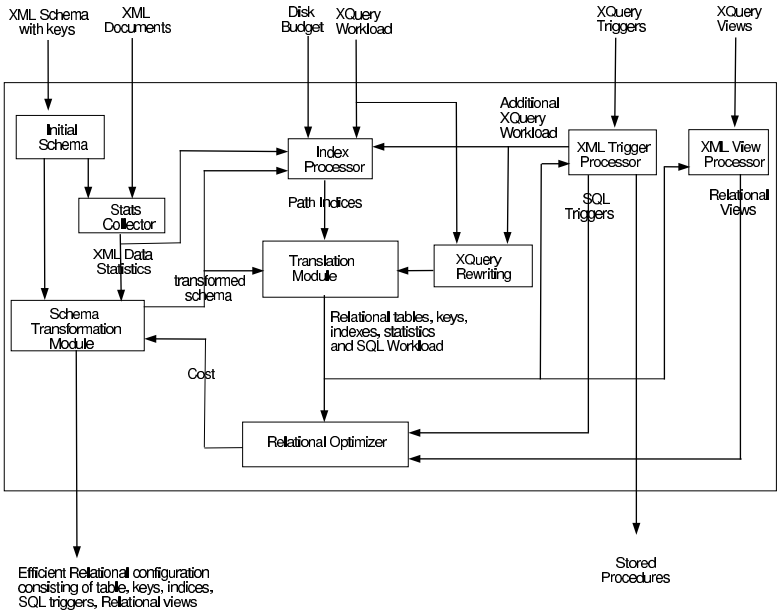


Figure 2: Architecture of the Elixir system

$$K : (Q, (Q', \{P_1, \ldots, P_p\}))$$

while the *keyref* constraint is written as

$$R : (Q, (Q', \{P_1, \ldots, P_p\})) \text{ KEYREF } K$$

where Q, Q', and P_1,...,P_p are all path expressions. The element within which the key is defined is called the *context element* E, and Q, the path leading to this context element, is called the *context path*. On similar lines, the set of nodes on which the key is to be defined, relative to the context element, is called the *target node set*, and Q', the path leading to this set of nodes, is called the *target path*. P_1,...,P_p are the *field paths*, which are relative to the target path and identify the set of nodes whose values are used to distinguish nodes of the target node set. Finally, K is the name of the *unique* or *key* constraint, and R is the name of the *keyref* constraint.

The path expression language used to define keys in XML schema is a restriction of XPath, and includes navigation along the child axis, disjunction at top level, and wildcards in paths. This path language can be expressed as follows:

$$c ::= . \mid / \mid .q \mid /q \mid .//q \mid (c|c)$$
$$q ::= l \mid (q/q) \mid -$$

where "/" denotes the root or is used to concatenate two path expressions, "." denotes the current context, l is an element tag or attribute name, "-" matches a single label, and ".//" matches zero or more labels out of the root.

3.2 Generating Constraint-Preserving Relations

The *Translation Module* takes an XML schema with constraints as input and produces a constraint-preserving equivalent relational schema.

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 (Note here, country is a *context*), each account is uniquely identified by a savings or checking account number.

- *cust-acc*: (//country,(./customer,{acc-number})) KEYREF *acc-num-key*
  Within a country, each customer refers to a savings or checking account number by acc-number.

Note here, for *acc-num-key* – country is a context element, .//account identifies the set of nodes on which the key is defined, sav-acc-num | check-acc-num identifies the values that distinguish each element of the set. Keyref *cust-acc* refers to *acc-num-key*.

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 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, X2R uses XML keys to define constraint relations and relational keys, and then uses inlining into constraint relations for the nodes that are not mapped in

```
<bank>
    <country>
        <name>India</name>
        <customer>
            <cust-id>1</cust-id>
            <name>abc</name>
            <address>...</address>
            <acc-number>101</acc-number> ...
        </customer> ...
        <customer>
            <cust-id>2</cust-id>
            <name>xyz</name>
            <address>...</address>
            <acc-number>102</acc-number>
        </customer> ...
        <city>
            <name>Bangalore</name>
            <state>Karnataka</state>
            <head-office>
                <id>0112</id>
                <address>...</address>
            </head-office>
            <branch-office>
                <id>0321</id>
                <address>...</address>
            </branch-office> ...
            <atm>
                <id>A1231</id>
                <address>...</address>
             </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>
```

Figure 3: Sample XML Document (bank.xml)

constraint mapping. In contrast, the *Translation Module* of Elixir, first produces a schema using the LegoDB fixed mapping process, and then integrates the keys with this schema. Yet another difference is that in X2R the schema production is a one-time process, whereas Elixir employs a cost-based iterative process to find the best constraint-preserving schema.

In the remainder of this section, we describe the various steps involved in the production of the constraint-preserving relational schema.

Schema Tree

The XML schema is first converted into the *schema tree* representation proposed in FleXMap [27], in terms of the following type constructors: *sequence*(","), *repetition*("*"), *option*("?"), *union*("|"), $< tagname >$ (corresponding to a tag), and $< simpletype >$ corresponding to base types (e.g. integer). To make this concrete, a schema tree for the banking example is shown in Figure 4. The schema tree nodes are *annotated* with the *names* of the types and these annotations are shown in boldface and parenthesized next to the tags (the base types are omitted for readability). Each annotated node corresponds to a separate table in the relational schema, and although we start off with every node being annotated, nodes may lose their annotation during

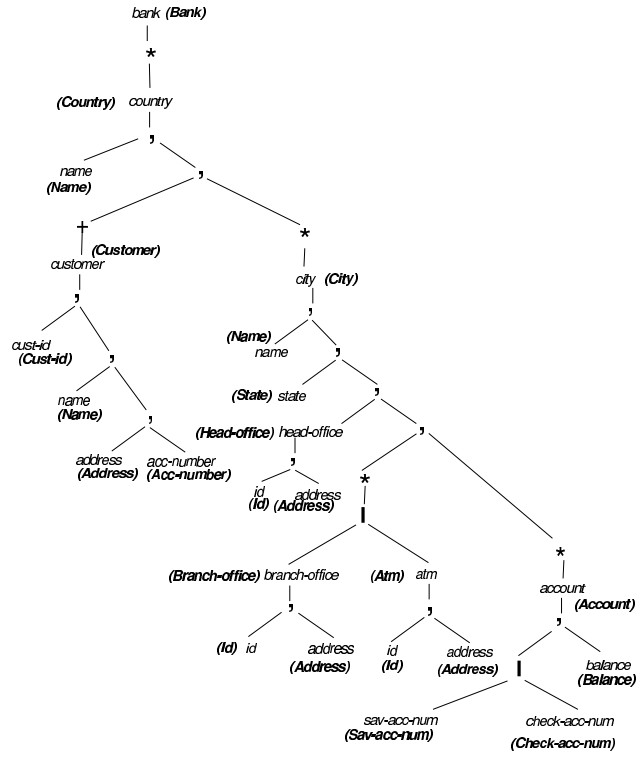


Figure 4: Schema tree for bank schema

the optimization process, as discussed later.

Step 1: Association of Subtrees

In the first step, subtrees corresponding to different paths of a single field path are *associated*. Let $t_{P_1}, \ldots, t_{P_p}$ be the subtrees of the schema tree corresponding to field paths $P_1, \ldots, P_p$. If P_i is of the form $(p_1|p_2|\ldots|p_n)$ where $p_1, \ldots, p_n$ are the different paths of a single field path P_i and N_1,N_2...,N_n are the corresponding nodes in the schema tree, these nodes need to be associated so as to map them all to a common attribute of a common relation. For example, consider the key: *acc-num-key* : (//country,(.//account, {sav-acc-num | check-acc-num})). As per this key, both **Sav-acc-num** and **Check-acc-num** need to be mapped to the same column of the relation `Account`. Thus, the nodes corresponding to **Sav-acc-num** and **Check-acc-num** from the schema tree should be associated.

Step 2: From Schema Tree to Table Configuration

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, as described below:

1. Create table T_N corresponding to each annotated node N, with a *key* column, and a *parent-id* column that points to a key column of the table corresponding to the *closest named ancestor* of the current node, if it exists.

2. If the annotated node is a simple type, then T_N additionally contains a column corresponding to that type to store its values.

3. For each associated group of descendants, create an additional column to which all descendants in the group are mapped, and create a column to identify the descendant in the group.

Step 3: Incorporation of Relational Keys

After mapping the XML schema to tables, the final step is to incorporate the relational keys that are equivalent to the original XML keys. Since Elixir restricts its attention to *constraint-valid schema trees*, as described later in this section, it is assured that the subtrees $t_{P_1}, \ldots, t_{P_p}$ will always have the parent with the same type name, which means that they will all get mapped to columns of a single relation. Let $C_{P_1}, \ldots, C_{P_p}$ be these corresponding columns of the relation T_N, and let E be the *context element*. The relational key is now defined as follows:

- If E is an immediate parent of N, then there must be a column, named *parent-E*, storing the key for E. Otherwise, add an additional column *parent-E* to T_N for storing the Id of ancestor element E. The *parent-E* column is required to distinguish between different contexts created by context element E.

- Create $\{C_{P_1}, \ldots, C_{P_p}$, *parent-E*$\}$ as a key/unique for relation T_N.

For example, consider the following relational configuration obtained after the second step for type **Account**:

```
TABLE Account ( Account-id-key INT,
Sav-or-check-acc-num INT, acc-number-flag
INT, balance INT, parent-City INT)
```

Note here that for type **Account**, a relation named `Account` is created. The associated tree of **Sav-acc-num** and **Check-acc-num** is mapped to column `Sav-or-check-acc-num`, and an additional column, `acc-number-flag`, is created for identifying the account number type. All the remaining simple type children are mapped to columns of relation `Account`.

For the XML key *acc-num-key*, the context element is country, which is not an immediate parent of **Account**. Therefore, a column has to be added to `Account` relation, which refers to `country-id-key` and create key as {`Sav-or-check-acc-num`, `parent-Country`}. The resulting final relational configuration is as follows:

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

A similar process to the above can be used for integrating *keyref* constraints – the only difference is the following: Let K_r be the relational key corresponding to XML key/unique K, obtained using above rule, and let R be the keyref that refers to K. Use same rule for R with a change that instead of defining key/unique, define the foreign key that refers to K_r.

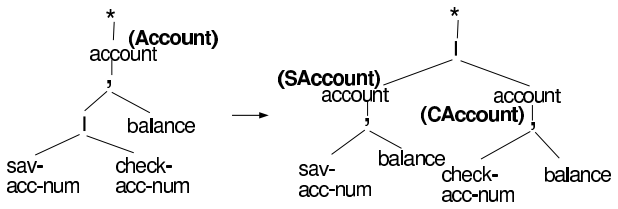


Figure 5: Invalid union distribution

For example, consider a keyref *cust-acc* : (/country,(./customer,{acc-number})) KEYREF *acc-num-key* and the relation for type **Customer** is as follows:

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

For the XML keyref *cust-acc*, the context element is country, which is an immediate parent of **Customer**. There is no need to add column to `Customer` relation that refers to `country-id-key`, as it is already present. Create foreign key as {`Acc-number`, `parent-Country`}, which refers to the relational key equivalent to *acc-num-key* i.e. {`Sav-or-check-acc-num`, `parent-Country`} of `Account`.

3.3 Filtering of Schema Transformations

Cost-based strategies explore the optimization space, by applying various transformations to the XML schema, and evaluating the costs of the corresponding relational configurations. A rich set of transformations have been proposed in [3, 27], that exploit the regular expressions and typing present in XML Schema. These transformations include *Inline/Outline*, *Type-split/merge*, *Union distribution/factorization*, and *Repetition split/merge*. Elixir restricts the search space to only *valid schema trees* by filtering out the invalid schema transformations, an example of which is shown next.

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(SAccount-id-key INT,
sav-acc-num INT, balance INT, parent-City
INT)
TABLE CAccount(CAccount-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. According to *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. By avoiding this distribution, the following relational configuration is obtained:

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

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 lot of unnecessary work, which can be avoided if we can detect the invalidity schema transformations *before* carrying out the schema transformation.

In remaining section, we discuss each schema transformations and rules for filtering these invalid schema transformations.

Before we describe the schema transformations and filtering process in detail, the following notions are required: Given an XML key, a *Key Path* is the concatenation of Q, Q', P_i where P_i is one component of the *field path*. Thus, a key will have n key paths, where n is the number of field paths in that key. For Example, *city-key* has two key paths: //country/city/name and //country/city/state.

A subtree t of the schema tree is said to be *reachable* by path P if its root node is traversed while traversing P along schema tree t. For example, consider t = (branch-office | atm) in Figure 4, with P being //country/city/branch-office/id. Traversing t according to P will have to include the root node ("|") in order to reach the branch-office element. Thus, the schema subtree t is reachable by path P.

3.3.1 Union Distribution and Factorization

Union distribution can be used to separate out components of a union: $(a,(b|c)) \equiv (a,b)|(a,c)$. Conversely, the union factorization transform would factorize a union. Assume that union $t_1|t_2$ is being distributed, where t_1 and t_2 are subtrees of the schema tree.

This union distribution will be invalidated by the XML key constraints in the following two cases:

Case 1: Consider the example discussed previously. Now we will try to analyze the cause for invalidation. The subtrees corresponding to sav-acc-num (t_1) and check-acc-num (t_2). Note that both the subtrees are on same field path of the *acc-num-key* constraint. Thus, if the union distribution of these 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.**

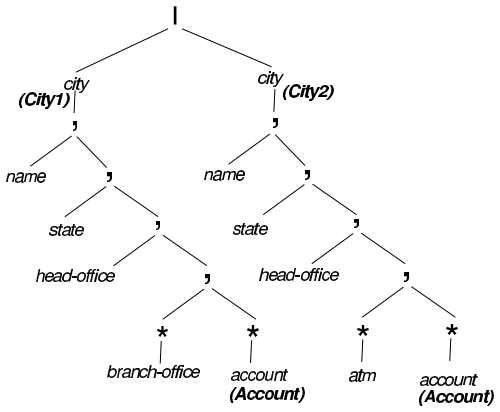


Figure 6: Invalid union distribution

Case 2: Consider union distribution of branch-office (t_1) and atm (t_2), which results in a relational configuration (incorrectly) storing name and state in two separate relations (refer to Figure 6). The analysis of this case shows that the union distribution of $t_1|t_2$, where the siblings of t_1 and t_2 (i.e. name and state) are key fields, result in distribution of the key fields into multiple relations. This is also true even if sibling is not a key field but its descendant is a key field. Thus, the general rule is that **if subtrees t_1 and t_2 are not on the field path, but their common parent is on the key path, then union distribution of $t_1|t_2$ is invalid.**

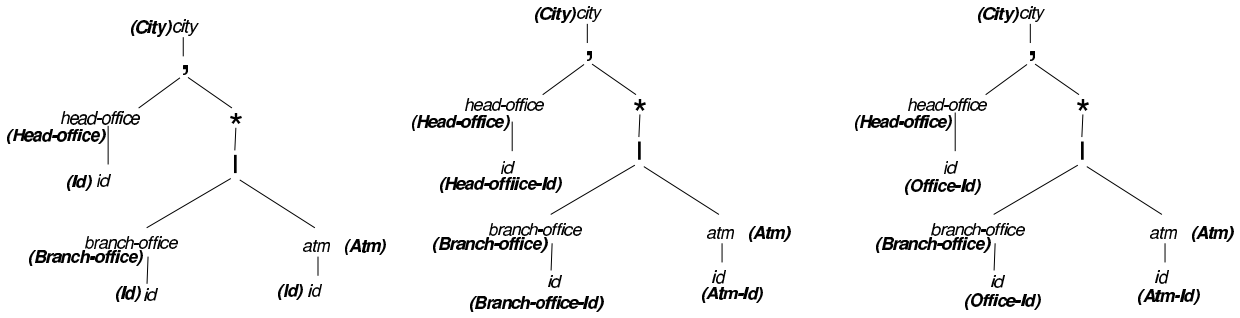
Turning our attention to Union Factorizations, we find that they are *always valid* even in the presence of constraints. The reason is that XML keys never imply that particular information should *not* be stored in a single relation, i.e. applying union factorization on a pair of elements stored in different relations will result in storing these elements into individual columns of a single relation. Thus, Union Factorization is valid in all cases.

3.3.2 Type-Split and Type-Merge

Type-Split and Type-Merge are based on the renaming of nodes. A type is said to be shared when it has distinct annotated parents. For example, in Figure 7(a), the type **Id** is shared by the types **Head-office**, **Branch-office** and **Atm**.

While, in principle, Type-Split and Type-Merge can be done with various subsets of the type occurrences in the schema, earlier work [27] focused on the extremes of *type-split-all* and *type-merge-all*. For example, the type **Id** is fully split in Figure 7(b) into **Head-office-Id**, **Branch-office-Id**, and **Atm-Id**. Similarly, while merging, full merging of **Head-office-Id**, **Branch-office-Id**, and **Atm-Id** into type **Id** is attempted.

Consider the XML constraints *office-key* : (//country, (./city/head-office | ./city/branch-office, {id})) and *atm-key* : (//country, (./city/atm, {id})). In order to define the relational keys for *office-key*, the **Head-office-Id** and **Branch-office-Id** should be mapped to the same column, i.e. they should be type-merged, and for *atm-key*, **Atm-Id** should be mapped



(a) type-merge-all (b) type-split-all (c) selective type split and merge

Figure 7: Type-Split and Type-Merge

to the other relation. According to *office-key* and *atm-key*, both the transformations i.e *type-split-all* and *type-merge-all*, are invalid. Thus, we need to do selective type-merge and selective type-split, as shown in Figure 7(c).

Let T be the type to be split and $Parent_1, \ldots, Parent_n$ be the parents of T with distinct annotations. The following procedure is used for selective type-split/merge:

Step 1: Do the *type-split-all* of T into $T_1, \ldots, T_n$ corresponding to the parents $Parent_1, \ldots, Parent_n$.

Step 2: Group the parents into different classes according to *key paths*.

Step 3: Merge types T_i whose parents are in the same class.

Consider the partial schema shown in Figure 7(a). In Step 1, **Id** is type-split as **Head-office-Id**, **Branch-office-Id** and **Atm-Id**. In Step 2, classes of the parents are formed according to key paths, which are {**Head-office-Id**, **Branch-office-Id**} and {**Atm-Id**}. Then, parents in the same class are merged – thus, **Head-office-Id** and **Branch-office-Id** are type-merged into **Office-Id**, as shown in Figure 7(c). This schema is consistent with the *office-key* and *atm-key* constraints. The selective merge is necessary because it assures that type splits, which violate any key, will be filtered.

A similar procedure can be used for type-merge, which is as follows: Group the parents into different *key classes* according to *key paths* and place the remaining parents in a separate class, called *Non-key class*. The classes formed i.e. *key classes* and *Non key class*, represent the valid type-merges.

For example, assume that Figure 7(b) is the input schema tree in which type Id is already type-split. Thus, while exploring the relational configuration search space, the type merge of **Head-office-Id**, **Branch-office-Id** and **Atm-Id** has to be considered. Grouping the parents according to key paths results in two *key classes*: {**Head-office-Id**, **Branch-office-Id**} and {**Atm-Id**}, while Non key class is {}. Thus, it is valid to type-merge **Head-office-Id** and **Branch-office-Id**, as in Figure 7(c).

3.3.3 Repetition Split and Merge

Repetition Split and Merge exploit the relationship between sequencing and repetition in regular expressions by turning one into the other. They are based on the law over regular expressions $(a+ == a, a*)$.

Consider the repetition split of type *T*. Let $E_1, E_2, \ldots, E_n$ be the children of *T*, which could be of type element or attribute. If at least one of the children of T is on the field path, then the repetition split is invalid because then the child becomes mapped to two elements. For example, consider repetition split of customer under constraint *customer-key*: (//country, (./customer, {cust-id})). Repetition split of type **Customer**, followed by inlining will result in storing cust-id in two relations, conflicting our goal of defining a relational key corresponding to *customer-key*. Thus, this repetition split is invalid.

Note that, like union factorization and for the same reason, repetition merge is always valid.

3.3.4 Type Inline and Outline

A type can be outline or inlined by, respectively, annotating a node or removing annotations. For each key, the process of determining inline or outline for the element can be done in two steps:

Step 1 :
  Outline-all-field-paths = false
  For each field path of field
    Let field-tree = tree obtained by associating different paths in the field $P_1, \ldots, P_p$
    Inline all the elements of field tree
    If field-tree is shared
    (i.e. their parents are mapped to different relations) then
      Outline-all-field-paths = true
Step 2:
  Let final-tree = associate all trees of fields
  If Outline-all-field-paths then
    outline root of final tree.

Let *field-tree* be the tree obtained by associating different paths of the field path, and let *final-tree* be the tree obtained after associating all field trees corresponding to field

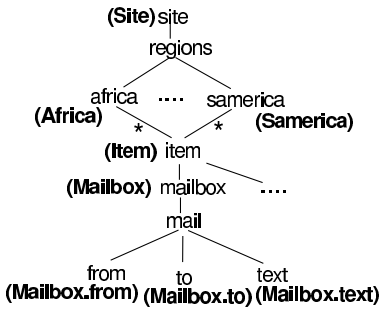


Figure 8: Example relational configuration

paths $P_1, \ldots, P_p$. The first step performs the inlining of all field-trees and checks if any of these trees are shared. The second step associates all the field-trees and then, if there are one or more shared field-trees, outlines the root of the final-tree so that all the field-trees are mapped to the same relation.

According to *acc-num-key* constraint, **Sav-acc-num** and **Check-acc-num** should be mapped to the same column of the relation `Account`. Thus, the nodes corresponding to **Sav-acc-num** and **Check-acc-num** from the schema tree should be associated. The field tree corresponding to the field path of *acc-num-key* is the tree obtained after associating the trees corresponding to **Sav-acc-num** and **Check-acc-num**. Since this field tree is not shared, the associated tree is inlined in the type **Account.** For *office-key* constraint, the final tree will contain id. Since it is shared between head-office and branch-office, the tree should be outlined.

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 [28], 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 these issues.

4.1 Path Index to Relational Index conversion

Consider an XML-to-relational mapping, as shown in Figure 8 for a fragment of the XMark benchmark schema [34]. 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.

Direct approach

An obvious translation process is to simply build the indices on the key and foreign-key pair for each parent-child involved in the path, namely, `Site.Site-id-key`, `Africa.parent-Site`, `Africa.Africa-id-key`, `Item.parent-Africa`, `Item.Item-id-key` and `Mailbox.parent-Item`. If we assume that for each relation, the column which stores IDs is defined as the primary key, and that relational engines by default create an index on the primary key column, then the additional indices that have to be explicitly created are only `Africa.parent-Site`, `Item.parent-Africa`, and `Mailbox.parent-Item`. Further, a value index has to be created on `Mailbox.from` to reflect the last element (from) in the path index, which is of simple type. Overall, the single path index has resulted in four (additional) relational indices. As indices are available on all the join attributes, the resulting join query can be evaluated efficiently. However, the drawback of this approach is that the number of relational indices, which need to be created for a path index is a function of *the path length*, and can therefore become very expensive to create and maintain.

Approach based on concept of *equivalence classes*

An alternative and less expensive approach is to use the concept of *equivalence classes* [28] 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. The detailed algorithm to establish equivalence classes is given in [28].

Algorithm 1 Converting Path Index to Relational Indices

Function: ConvertIndex
Input: rT: tables, xPI: Path index, EQ: path equivalence classes
Output: rT': tables, rI: relational indices equivalent to xPI

1: let $e_1/e_2/\ldots/e_n = xPI$;
2: $split\_paths$ = SplitPath(xPI, EQ); // Split path index according to equivalence groups
3: $rI = \{\}$; $rT' = rT$;
4: let $\{c_1,c_2,\ldots,c_m\}$ be the EQ groups for split_paths $\{p_1,p_2,\ldots,p_m\}$
5: **for** i = 2 to m **do**
6: Te_i = table from rT' to which last element of p_i is mapped;
7: Te_{i-1} = table from rT' to which last element of p_{i-1} is mapped;
8: Add column (C) to Te_i which stores information about the parent from Te_{i-1};
9: Define foreign key on column (C) of Te_i, which refers to key of Te_{i-1};
10: $rI = rI \cup \{(Te_i,C)\}$
11: **end for**
12: **if** last element of p_m is of simple type **then**
13: (Te_m,Ce_m) = relation and column to which last element of p_m is mapped;
14: $rI = rI \cup \{(Te_m,Ce_m)\}$
15: **end if**
16: return (rT', rI)

Algorithm 2 Algorithm to split path according to path equivalence classes

Function: SplitPath
Input: P: Path, EQ: path equivalence classes
Output: $split\_paths$: split paths according to EQs

1: j=n; $split\_paths = \{\}$;
2: **while** j $>$ 1 **do**
3: let $e_i/\ldots/e_j$ be the longest equivalent path of e_j;
4: $split\_paths = split\_paths \cup \{e_i/\ldots/e_j\}$;
5: j = i-1;
6: **end while**
7: return ($split\_paths$)

For example, the paths site/regions/africa, regions/africa, and africa belong to a common equivalence class. But, the paths africa/item and item are not in the same equivalence class because the path africa/item matches with the items, which are children of the africa element whereas the item path matches to *all* item nodes, some of which may not be children of the africa element.

Based on the above approach, we have developed a procedure that uses the path equivalence classes (EQs) to convert the index to relational indices corresponding to each EQ on the path (refer to Algorithm 1).

In this algorithm, the first step is to split the path index such that each subpath corresponds to different equivalence classes (line 2). For details of SplitPath refer to Algorithm 2. For each equivalence class, the information about the closest parent that is mapped to a relation and is from the previous equivalence class is stored (lines 4 through 11). Then, indices on the columns added in the previous step are created (line 10). If the last element of the path index is of simple type, then an index is created on the column to which it is mapped (lines 12 through 15).

For example, consider the path index, *PI* = /site/regions/africa/item/mailbox/mail/from. In the first step, PI is split into /site/regions/africa and /item/mailbox/from. The next step adds the column `parent-Africa` to the relation `Mailbox`, and an index is created on this column, namely, `Mailbox.parent-africa`. Also since the last element, from, is of simple type, an index is created on the `Mailbox.from` column. Note that overall, the path index PI has resulted in only *two* relational indices (as compared to the four of the direct approach).

4.2 Disk Budget Maintenance

The user-specified disk budget of XML indices has to be considered with regard to the space occupied by the equivalent *relational indices*. To estimate the size of the relational indices, we use the following heuristic formula, which computes the size of the index using the cardinality of the table (N) and size (S_{ic}) of the data type of that column as inputs. The column statistics can be obtained from the XML data statistics of the corresponding type [15]. Assuming the index is implemented as B+ tree, the size of index can be calculated as follows:
Let S_{ic} = total size of the indexed columns
Let S_{page} = size of page typically 4KB
Let S_{ptr} = size of the pointer typically 4 bytes
Let P_k = Total number of keys in a page = $\frac{S_{page}}{S_{ic}+S_{ptr}}$

Then Size of the Index (S_I) is given as

$$S_I = \frac{N}{P_k * average\_page\_occupancy\_factor} * S_{page} \quad (1)$$

Note that, here we assume that $average\_page\_occupancy\_factor$ is 0.69, which is the typical fill factor for B+ tree index [36].

4.3 Query Rewriting for Path Indices

The use of integrity constraints to guide XQuery-to-SQL query translation has been recently discussed in [22]. 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.

The procedure for achieving this conversion is described in Algorithm 3. The first step in the procedure for achieving this conversion is to identify all paths in the schema that satisfy the query (line 2). For each path, split the path such that each subpath is either an element or is a path corresponding to the available path index, and then compute the path equivalence classes for each subpath. This can be achieved by first splitting the path using a SplitPath function of Algorithm 2 and then finding the equivalence classes for each split path (lines 6 through 15). The relational query components are generated by joining the relations corresponding

Algorithm 3 Algorithm for Translating XQuery-to-SQL

Function: TranslateToSQL
Input: rT: Tables with statistics, xPI: Path indices, EQ : path equivalence classes, xW : XML query workload
Output: W_SQL : SQL queries equivalent to xW

1: **for all** q in xW **do**
2: let $paths$ = all paths in the schema that satisfy the query
3: SQL = {};
4: **for all** P in $paths$ **do**
5: let $ep_1/ep_2/\ldots/ep_n$ = P; //ep_i is either an element or a available path index (xPI)
6: $path\_EQs$ = {}
7: **for** i = 1 to n **do**
8: **if** ep_i is element **then**
9: $path\_EQs$ = $path\_EQs \cup \{ep_i\}$;
10: **else**
11: $path\_EQs$ = $path\_EQs \cup$ SplitPath(ep_i, EQ);
12: **end if**
13: **end for**
14: $join\_relations$ = {};
15: let $\{c_1,c_2,\ldots,c_m\}$ be the EQ classes for $path\_EQs$ $\{p_1,p_2,\ldots,p_m\}$
16: P_SQL = join of available tables form rT corresponding to the EQ classes
17: SQL = $SQL \cup P\_SQL$;
18: **end for**
19: W_SQL = $W\_SQL \cup$ {query which is union of all queries that are in SQL}
20: **end for**
21: return (W_SQL)

to the EQ classes (line 16), and the final query is the union of all these queries (line 23).

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'
```

5 Mapping XML Triggers and Views

We now move on to the advanced components of XML triggers [4] and XML views [1].

5.1 Triggers

Triggers are primarily used to execute a specific logic upon updates to the database. As XQuery triggers (XML triggers) are not part of standard, we have used the extension of XQuery for defining triggers, which is proposed in [4]. To leverage the power of relational databases, our aim is to map the XML trigger to relational triggers, an example of which is shown in Figure 9.

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 of a single statement. For example, when a bank sets up operation 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, which sends 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.

5.1.1 Detecting Mappability of XML trigger

Mappability of XML trigger is determined as follows:

- If the triggering time is `BEFORE` then the XML trigger is always mappable. In case of `BEFORE` trigger, it is always possible to define equivalent SQL trigger on the relation R, which corresponds to node identified by path specified in *triggering operation*. Trigger action is executed before doing the *triggering operation* on the relation R, which is the exact semantics of XML trigger.
- If the triggering operation is `DELETE` then the XML trigger is always mappable. Though deletion operation on XPath (P) may cause the deletion of rows from different relations (corresponding to node itself and its descendants), it can be written as one `DELETE` SQL statement on single relation R. R is the relation corresponding to node identified by P. Deletion of descendants of node is automatically done because of defining the foreign key with `CASCADE DELETE`

```
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

Figure 9: Mapping XML trigger to SQL trigger

option. Thus, if we define a delete trigger on relation R, then integrity constraints are enforced before execution of trigger, which simulates the exact semantics of delete XML trigger.

- If the triggering time is `AFTER` and triggering operation is `INSERT` or `REPLACE` then
  1. Convert XPath expressions specified in triggering event to simple XPath expressions containing only child axis and no wild characters.
  2. Group these simple XPath expressions according to the relation in which the nodes identified by XPath are mapped.
  3. For each group
     - Check if all descendants of the elements or attributes, identified by XPath expression from group, are mapped to the same relation. This condition ensures that SQL equivalent of *triggering operation* contains single insert or update statement.
     - Check if none of the elements other than descendants of the elements or attributes, identified by XPath expressions from group, is mapped to the relation corresponding to the group. This condition ensures that there are no false triggers i.e. there are no triggers on the path, which are incorrect according to semantics of corresponding XML trigger.
     - If both of the above conditions (*mappability conditions*) are true then XML trigger is mappable otherwise, it is non-mappable.
- If the triggering time is `AFTER` and triggering operation is `RENAME` then XML trigger is not mappable. `RENAME` operation always involve two sub-operations i.e. insertion followed by deletion. Thus, XQuery involving `RENAME` operation on the given XPath expression always result into more than one SQL statement causing the XML trigger to be non-mappable.

For example consider a trigger as follows:
`CREATE TRIGGER Office-or-atm-trigger`
`AFTER INSERT OF` //id
...

Simple XPath expressions corresponding to //id are /bank/country/head-office/id | /bank/country/branch-office/id | /bank/country/atm/id

Next step is to group the Simple XPath expressions according to relations. These groups are as follows:

Group-1: /bank/country/head-office/id | /bank/country/branch-office/id

Group-2: /bank/country/atm/id

The relations corresponding to Group-1 and Group-2 are `Office-Id` and `Atm-Id`, respectively. As there are no descendants for id, *mappability conditions* specified earlier are true, thus this XML trigger is mappable for both the groups.

XML trigger can result in more than one SQL trigger. For example, above XML trigger is converted into two SQL triggers, one on relation `Office-Id` and other on relation `Atm-Id`. Note that, XML trigger could be mappable for some XPath expressions and non-mappable for remaining XPath expressions.

5.1.2 Mappable XML trigger to SQL trigger

Different components of SQL triggers can be derived from XML trigger as follows:

name of SQL trigger: As explained earlier one XML trigger might be mapped to more than one SQL trigger. If XML trigger is mapped to single SQL trigger then we can use the XML trigger name to define SQL trigger. Otherwise, we generate unique name for each trigger concatenating the name of XML trigger and relation (on which trigger is defined).

triggering time: Triggering time for SQL trigger is same as that of XML trigger i.e. `BEFORE` or `AFTER`.

triggering operation type: Triggering operation type can be decided as per triggering operation specified in XML trigger.

- `INSERT`: If the node identified by XPath expression of XML trigger is of simple type, then equivalent SQL of the insert XQuery on the specified path is an update query. In this case triggering operation type in SQL trigger is `UPDATE` and the column that corresponds to the simple type. If the node identified by XPath expression of XML trigger is nested element then operation type in SQL trigger is `INSERT`.
- `DELETE` : If the node identified by XPath expression of XML trigger is of simple type, then equivalent SQL of the delete XQuery on the specified path is an update query (which sets its value to `NULL`). In this case, triggering operation type in SQL trigger is `UPDATE` and the column that corresponds to the simple type. If the node identified by XPath expression of XML trigger

is nested element, then operation type in SQL trigger is DELETE.

- RENAME : For mappable triggers, RENAME operation results in update SQL statement. Thus, triggering operation type of SQL trigger should be UPDATE.

triggering granularity: NODE, STATEMENT level granularity can be defined in XML trigger; these are mapped to ROW, STATEMENT level granularity of SQL trigger, respectively.

trigger-condition: XML trigger condition can be converted to SQL trigger condition using XML-to-SQL translator. In addition, trigger condition also have the queries corresponding to checking of path filters specified in path of *triggering operation*.

trigger-action: XML trigger action can be converted to SQL trigger action using XML-to-SQL translator.

Consider XML trigger *Increment-Counter* for bank example. Here we assume that there is an additional element acc-counter that keeps track of number of accounts in a branch office and customer has additional element branch that keep track of the branch from which he has got account.

```
CREATE TRIGGER Increment-Counter
AFTER INSERT OF //customer
LET $branch-id = NEW_NODE/branch
FOR EACH NODE
DO (
    FOR $branch-node = //branch-office
    LET $counter = $branch-node/acc-counter
    WHERE $branch-node/id=$branch-id
    UPDATE $branch-node
        {REPLACE $counter WITH $counter + 1} )
```

Using above procedure SQL trigger equivalent to above XML trigger is as follows:

```
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
```

5.1.3 Processing non-mappable XML Triggers

Non-mappable XML triggers cannot be mapped to equivalent relational trigger. The solution to this is to map these triggers to stored procedure, which can be called by middleware at runtime. Elixir model the cost of *Non-mappable XML triggers* by including an additional, equivalent query workload in the input XML query workload. We will first describe the mapping of *non-mappable XML trigger* to stored procedure and then their incorporation in tuning process.

Non-mappable XML triggers to stored procedure

XML trigger action is converted to an equivalent stored procedure by converting XQuery statements to SQL statements; the variables referred in trigger action are considered as parameters. For example, the XML trigger *NewCityTrigger* defined previously, the stored procedure corresponding to this trigger is as follows:

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

Values of NEW_NODE or NEW_NODES, which are needed for computation of procedure parameters, can be evaluated during XML-to-SQL translation. Values of OLD_NODE or OLD_NODES can be evaluated by inserting appropriate select statement before their deletion. For example translation of the insert query on city is translated as follows:

```
DECLARE city-name String;
DECLARE city-state String;
INSERT INTO City
   (1000,'Nasik','Maharashtra',...);
SET city-name = 'Nasik';
SET city-state = 'Maharashtra';
INSERT INTO Branch-office (...)
...
INSERT INTO Office-Id (...)
...
INSERT INTO Atm (...)
...
CALL NewCityTrigger(customer-name, city-name,
                    city-state,...)
```

Incorporation of Non-mappable triggers in tuning process

As mentioned earlier, Elixir models the cost of non-mappable triggers by including additional query workload in the input XML query workload. The query workload equivalent to *non-mappable XML triggers* consists of two query categories:

1. Select queries that are used to evaluate the variables, which are passed as a parameter to corresponding stored procedure. In addition, the select queries (inside WHEN clause) that are used to evaluate the condition of trigger.
2. Update queries, which are executed as a trigger action. These are the queries that become part of stored procedure.

Target workload also consists of the frequencies of the queries. Calculation of the frequency for the queries from additional workload can be done as follows:

- For each trigger, determine the *trigger-count* (i.e. number of times the trigger is likely to be triggered). This can be easily done by summing the frequencies of the queries, which are likely to perform the triggering operation specified in the XML trigger. *Trigger-count* can be used as the frequency for the select queries.
- If trigger is conditional then *actual-trigger-count* (i.e. the number of time the trigger action is performed) is less than *trigger-count*. Rough estimates for the *actual-trigger-count* can be obtained by giving the conditional query (specified in the WHEN clause) to relational optimizer and then getting the cardinality of the result. This estimated cardinality to actual cardinality is the *fraction* that indicates the probability with which the trigger will be executed. Thus, *actual-trigger-count* can be obtained as (*fraction* * *trigger-count*). *Actual-trigger-count* can be used as the frequency for update queries.

5.2 XML Views

The notion of views is essential in databases. It allows various users to see data from different viewpoints. Although XQuery [2] currently does not provide standard for defining XML views, we can easily extend it to include the definition of views [10] as follows:

"CREATE VIEW *view_name* AS" followed by FLWR expression

Above definition can be extended to define materialized XML views can be defined as follows:

CREATE MATERIALIZED VIEW *view_name* AS
FLWR expression
DATA INITIALLY (IMMEDIATE | DEFERRED)
REFRESH (IMMEDIATE | DEFERRED)

DATA INITIALLY IMMEDIATE clause allows user to populate data in table immediately. The clause DATA INITIALLY DEFERRED means that data is not inserted as part of the CREATE TABLE statement. Instead, user has to do a REFRESH TABLE statement to populate table. Syntax for REFRESH TABLE is as follows:

REFRESH XML VIEW *view_name*

Since the materialized view is built on underlying data that is periodically changed, user must specify how and when he wants to refresh the data in the view. User can specify that he want an IMMEDIATE refresh or DEFERRED refresh. The clause REFRESH DEFERRED means that the data in the table only reflects the results of the query as a snapshot at the time user issue REFRESH XML VIEW statement.

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.

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-number
           = $account/sav-acc-number or
       $customer/acc-number
           = $account/check-acc-number
 return
  <customer-balance>
      <id>$customer/cust-id</id>
      <acc-number>
            $customer/acc-number
      </acc-number>
      <balance>$customer/balance</balance>
   </customer-balance>
DATA INITIALLY IMMEDIATE REFRESH IMMEDIATE
```

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

```
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
```

6 Experimental Evaluation

In this section, we present our experimental evaluation of the Elixir system. Our experiments were run on a vanilla Pentium-IV PC running Linux, with DB2 UDB v8.1 [17] as the backend database engine. Four representative real-world XML schemas: *Genex* [16], *EPML* [14], *ICRFS* [18], *TourML* [32], 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 [34].[2] The salient summary statistics of these schemas are given in Table 1.

	Genex	EPML	ICRFS	TourML	XMark
# unions	0	9	1	0	0
# repetitions	9	115	11	57	21
height	4	13	6	10	9
(#E + #A)	75	159	63	145	93
#keys	15	15	16	24	14

E: Element, A: Attribute

Table 1: Details of XML schemas

6.1 Effect of Keys

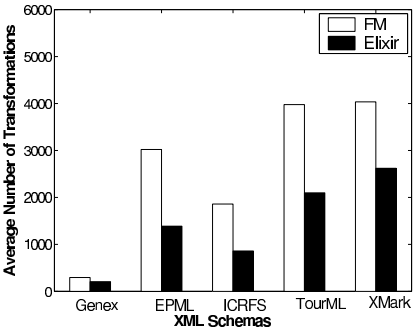
To serve as a baseline for assessing the effect of key inclusion, we compare the performance of Elixir (in the absence of indices, triggers, and views) with that of FleXMap (FM) [27], which is a framework for expressing XML schema transformations and for searching the equivalent relational configuration space. Specifically, we compare against the DeepGreedy (DG) search algorithm, which was found to be the best overall among the various search alternatives considered in [27]. Using the ToXgene tool [33], 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. The number of joins in the SQL equivalent of the query workload range from 5 to 15.

6.1.1 Runtime Efficiency

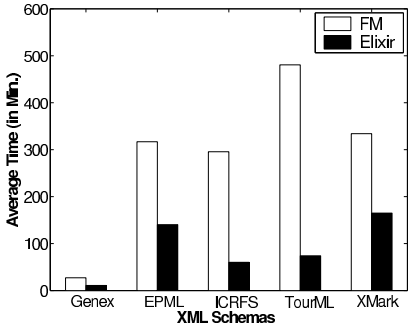
We compare the runtime efficiency of Elixir and FleXMap with regard to the following metrics: (a) The percentage reduction in search space, and (b) The time speedup due to this reduction. The average number of transformations evaluated by Elixir and FM are shown in Figure 10(a) for the five XML schemas. We see there that the reduction ranges from 30% to 60%, arising from the filtering out of invalid transformations, discussed in Section 3.2. 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 10(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

[2]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

Figure 10: Impact of Keys

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 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.

6.1.2 Table Configuration Quality

We also compared the quality of the final relational configuration in terms of the *cost* of executing the user query workload, and the results are shown in Table 2. In this table, the $*$ indicates situations where the final FleXMap configuration did not satisfy the key constraints. Note here that the final relational configuration (valid or invalid) is dependent on the data statistics. The reason for this is different set of transformations are selected in tuning process for different data statistics and resulting in different relational configuration.

In the remainder, we see that Elixir typically obtains configurations that are significantly lower cost as compared to FleXMap. The primary reason for the improvement is that Elixir explores an additional part of the overall search space of relational configurations due to performing selective type-merge/split guided by the XML key constraints.

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 [14] 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 11, 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 [26].

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 Effect of XML Triggers and Views

To assess the effect of XML triggers in the tuning process, we created for each schema an input workload consisting of 10 read-only queries, 10 update queries, and 5 XML triggers. We carried out two sets of experiments: In the first, the XML triggers are added *after* the tuning process, referred to as *w/o triggers*, whereas in the second set, triggers are included in the tuning process, referred as *with triggers*. Our experimental results on the cost of executing triggers in these two environments, a sample of which is shown in Figure 12, clearly demonstrat that including triggers directly in the optimization process can result in significant improvements of the final relational configuration.

To evaluate the impace of Views, we created for each schema 5 materialized views and 20 queries consisting of a mix of view queries and non-view queries. Note that here we have considered only materialized views, because virtual views do not affect the cost of the query, and thus do not affect the tuning process. Our experimental results, a sample of which is shown in Figure 13, again shows that considering views directly in the tuning process results in significantly better final relational configurations.

6.4 Overall Performance of Elixir system

While the previous experiments evaluated the performance in isolation for various components, the overall performance when all components are integrated is shown in Figure 14. 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-time process, and therefore time may not be a major issue; and,

	all-uniform		*uniform-exponential*		*all-exponential*	
	FM	*Elixir*	*FM*	*Elixir*	*FM*	*Elixir*
Genex	*	6193	6459	3367	*	6191
EPML	2335	1707	*	1151	*	425
ICRFS	4565	4249	*	4414	*	9630
TourML	*	11817	*	3356	*	5146
XMark	2626	1956	*	1453	3265	1645

Table 2: Cost of final relational configuration

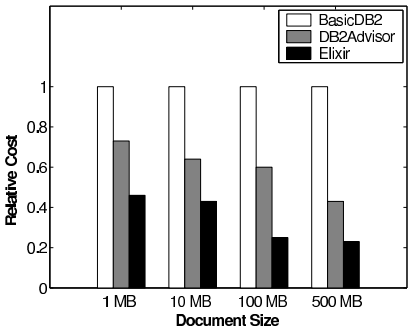


Figure 11: Index selection

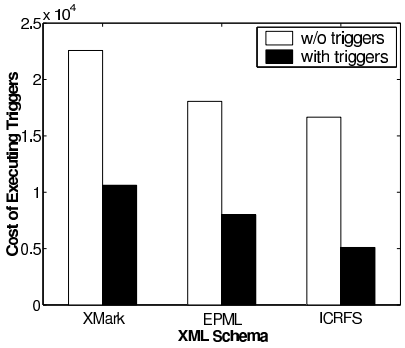


Figure 12: Effect of XML triggers

further, (b) the breakup of the runtime indicates significant potential for improvement – the heavy overhead due to the Optimizer is largely attributable to our using the database engine *from the outside*, so if the Elixir system is implemented within the relational engine, then the time required for evaluation may be significantly reduced.

7 Related Work

A rich body of literature has arisen in the last few years with regard to efficiently storing XML documents in RDBMS. Most of this prior literature focuses on isolated components of the mapping from the XML space to the relational space, and assumes static mappings between the spaces. In contrast, our goal is to design a holistic mapping that covers all the major components, and in addition, integrates cleanly with a cost-based dynamic mapping process.

With regard to XML keys, techniques have been proposed in X2R [7] and CPI [23] for mapping to relational equivalents. These systems are applicable for static mapping techniques like those proposed in [29]. The more general problem of expressing functional dependencies in XML is considered in [12, 9]. In this paper, we have focused only on integrity constraints in the form of *key* and *keyrefs* but our approach can be easily extended to handle XFDs. Further, functional dependencies are not currently part of the XML Schema standard. A major difference between all of this earlier work and Elixir is that they produce a relational mapping, which is optimized for updates (enforcing constraints efficiently), whereas Elixir produces a relational mapping optimized for the actual query workload of the application that involves both read only and update queries.

Recently, industrial researchers [6] have proposed a search algorithm that explores the combined space of logical and physical design, in conjunction with the relational advisor. On the other hand Elixir takes a consistently source-centric approach, where all optimization is done in the XML world, rather than at the relational target. Moreover, the techniques they suggest to prune the search space can also be incorporated in Elixir to improve the time efficiency. Finally, they do not take into account XML keys in mapping to the relational world.

In [30], authors have addressed the issue of triggers over XML view of relational data by translating triggers over XML views to SQL triggers, with updates to the relational data triggering the action. However, in Elixir, updates are done on the XML data, and not directly on the relational data. A systematic approach to design valid XML views is discussed in [10]. In our system, we assume only valid XML views are provided as input.

Nearly all leading relational vendors are also introducing XML capabilities. Many commercial tools (DB2, Oracle) provide basic support for querying XML documents using a relational engine. For instance, Oracle [25] provides an XMLTYPE to map XML data into an object table or view. User defined schema annotations can be used to control how collections in an XML document are stored in the database. IBM's DB2 Extender [17] provides two primary storage and access methods for XML documents: XML column and XML collection. In XML collection, Document Access Definition (DAD) specifies how to map XML documents into tables. The DAD can identify elements and attributes to be indexed (in side tables) for fast access. MS SQL Server [11] uses OpenXML rowset providers to support XML. It maps XML data into an edge table, a parent-child hierarchical graph representation of XML data. A major difference between all these tools and Elixir is that these XML tools relies on user defined relational mapping, whereas Elixir uses cost based approach to find the relational mapping optimized for the actual query workload of the application (read only queries and update queries).

8 Conclusions and Future Work

In this paper, we studied the problem of producing *information-preserving holistic schema* mappings from XML repositories to

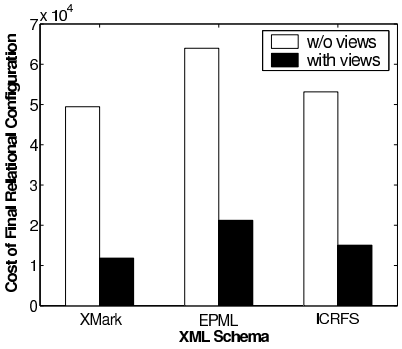


Figure 13: Effect of XML Views

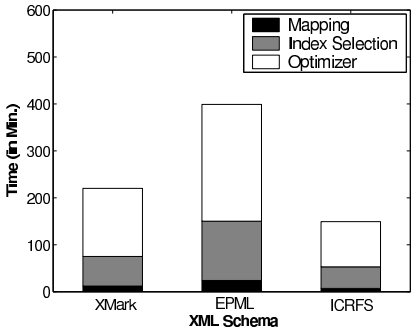


Figure 14: Elixir Performance

relational backends. We proposed the Elixir system, which captures all 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 the *PostgreSQL* public-domain relational engine.

References

[1] S. Abiteboul. On Views and XML. In *Proc. of PODS*, 1999.

[2] S. Boag et al. XQuery 1.0: An XML Query Language, May 2001. *http://www.w3.org/TR/xquery/*.

[3] P. Bohannon et al. From XML schema to relations: A cost based approach to XML storage. In *Proc. of ICDE*, 2002.

[4] A. Bonifati et al. Active XQuery. In *Proc. of ICDE*, 2002.

[5] P. Buneman et al. Keys for XML. *Computer Networks*, 39(5), 2002.

[6] S. Chaudhuri et al. Storing XML (with XSD) in SQL Databases: Interplay of Logical and Physical Designs. In *Proc. of ICDE*, 2004.

[7] Y. Chen, S. Davidson, and Y. Zheng. Constraints preserving schema mapping from XML to relations. In *Proc. of WebDB*, 2002.

[8] Y. Chen, S. Davidson, and Y. Zheng. Validating constraints in XML. Technical Report MS-CIS-02-03, Dept. of Computer and Information Science, U. of Pennsylvania, 2002.

[9] Y. Chen et al. RRXS: Redundancy reducing XML storage in relations. In *Proc. of VLDB*, 2003.

[10] Y. Chen, T. Ling, and M. Lee. Designing Valid XML Views. In *Proc. of Conceptual Modeling (ER)*, 2002.

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

[12] S. Davidson et al. Propagating XML constraints to relations. In *Proc. of ICDE*, 2003.

[13] DTD. *http://www.w3.org/XML/1998/06/xmlspec-report*.

[14] EPML. *http://wi.wu-wien.ac.at/~mendling/EPML/*.

[15] J. Freire et al. Statix: Making XML count. In *Proc. of SIGMOD*, 2002.

[16] GENEX. *http://www.ncgr.org/genex*.

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

[18] ICRFS. *http://www.insureware.com/abouti/ mlines.shtml*.

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

[20] C. Kanne and G. Moerkotte. Efficient Storage of XML data. In *Proc. of ICDE*, 2000.

[21] 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.

[22] R. Krishnamurthy, R. Kaushik, and J. Naughton. Efficient XML-to-SQL Query Translation: Where to Add the Intelligence? In *Proc. of VLDB*, 2004.

[23] D. Lee and W. Chu. Constraints–preserving Transformation from XML Document Type Definition to Relational Schema. In *Proc. of Conceptual Modeling (ER)*, 2000.

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

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

[26] P. Patil. Holistic Schema Mappings for XML-on-RDBMS. *http://dsl.serc.iisc.ernet.in/publications/thesis/priti.pdf*.

[27] M. Ramanath et al. Searching for efficient XML-to-relational mappings. In *Proc. of XSym*, 2003.

[28] K. Runapongsa et al. XIST: An XML Index Selection Tool. In *Proc. of XSym*, 2004.

[29] J. Shanmugasundaram et al. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of VLDB*, 1999.

[30] F. Shao et al. Triggers over XML Views of Relational Data. In *Proc. of ICDE*, 2005.

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

[32] Tourism Markup Language. *http://www.opentourism.org*.

[33] ToXgene. *http://www.cs.toronto.edu/tox/toxgene/*.

[34] XMark. *http://monetdb.cwi.nl/xml/*.

[35] XML schema. *http://www.w3.org/TR/xmlschema-1/*.

[36] A. Yao. On random 2-3 trees. *Acta Informatica*, 9(2), 1978.