Holistic Source-centric Schema Mappings for XML-on-RDBMS

A Thesis
Submitted for the Degree of
Master of Science (Engineering)
in the Faculty of Engineering

By
Priti Patil

Supercomputer Education and Research Centre
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012, INDIA

September 2005

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 the XML Schema to their relational counterparts, especially with regard to table configurations. However, for a viable real-world implementation, a *holistic* mapping that incorporates all fundamental aspects of relational schemas, including table configurations, integrity constraints, indices, triggers and views, is required. In this thesis, we address this lacuna and present the Elixir system for producing holistic relational schemas that are tuned to the XML 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. For example, Elixir significantly extends prior table configuration techniques, based on XML schema transformations, to seamlessly preserve XML integrity constraints. On a variety of real and synthetic XML schemas operating under a representative set of XQuery queries, we find beneficial side effects of incorporating these constraints in terms of more efficient table configurations and a substantial reduction in the configuration search space. With regard to index selection, too, Elixir makes path-index choices at the XML source and then maps them to relational equivalents – our experiments show that this is more desirable than the prevalent practice of using the relational engine's index advisor to identify a good set of indices. Elixir can also map XML triggers and XML views to obtain relational triggers and relational views respectively.

In a nutshell, the Elixir system attempts to make progress towards achieving "industrial-strength" mappings for XML-on-RDBMS.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past five years, XML (eXtensible Markup Language), by virtue of its powerful and flexible data formatting capabilities, has become a dominant standard for information exchange between applications, especially on the internet. As an increasing amount of XML data are being processed, efficient and reliable storage of XML data becomes an important issue. For persistently storing information from XML sources, there are primarily two technological choices available: A specialized native XML store (e.g. Tamino [47], Natix [29], Timber [27]), or a standard relational engine (e.g. IBM DB2 [25], Oracle [38], Microsoft SQL Server [19]). From a pragmatic viewpoint, the latter approach brings with it the benefits of highly functional, efficient and mature technology. However, there is a fundamental mismatch in the way the information is modeled in XML and in relational database, this is because XML has a flexible and extensible tree structure, whereas relational databases has a strict homogeneous flat table structure.

1.1 XML-to-relational mapping

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 relational mappings. Mapping primitives can be broadly classified as follows:

- generic methods, which do not use any schema of stored XML documents.
- schema-driven methods, which are based on pre-defined schema of stored XML docu-

ments.

- user-defined methods, which are based on user-defined mapping.

1.1.1 Generic Methods

One of the first proposals for mapping XML documents was Edge mapping approach [21], where entire XML tree is stored in a single table schema (`SourceId`, `Tag`, `Ordinal`, `TargetId`, `Data`). The table contains identifiers of nodes connected by the edge (`SourceId` and `TargetId`), name of the edge (`Tag`), a flag that indicates whether the edge is internal or points to a leaf (`Data`), and an ordinal number of the edge within sibling edges (`Ordinal`). If target node is leaf, then its text is included in `Data` otherwise `Data` is assigned `NULL` value. Similarly, Binary [21] approach groups the tuples in Edge table based on tag name (horizontal partitioning). This leads to better clustering and improves query performance. Moreover, the storage space is reduced, as tag name is not stored for each tuple in partition table. XRel [55] shreds the XML documents into nodes, where each node is encoded with a unique range (region encoding). The encoded nodes are stored in relational database management system along with path information from root to the node. XParent [28] uses two tables: `Path-table` and `Data-table`. `Path-table` stores all distinct paths identified by unique ID; `Data-table` stores all the node-pairs: `SourceId` and `TargetId`. Monet [44] partitions the `Edge` table based on distinct label paths appeared in the XML document, i.e. for each distinct label-path, Monet creates separate table. This type of structural clustering reduces the scans over large amounts of data irrelevant to the query.

1.1.2 Schema-driven Mapping Methods

Schema-driven mapping methods are based on existing schema of stored XML documents, which is mapped to a relational schema. A schema is a definition of the syntax of an XML-based language (i.e., it defines a class of XML documents). A schema language is a formal language for expressing schemas. There have been many schema language proposals, such as DTD (Document Type Definition) [7], XML Schema [49], XML-Data [35], DCD (Document

Content Description) [8], Schematron [43], etc. DTD and XML schema are the most widely used standards. Most of the schema-driven mapping methods are based on either a DTD or a XML schema of stored XML documents. The schema-driven mapping methods can be further classified in two classes : fixed and cost-based methods.

Fixed methods

Fixed methods do not use any other information than the source schema itself; their mapping algorithm is straightforward. For example, shared inlining [45] is fixed method, in which elements having multiple occurrences are mapped into tables, whereas elements with a single occurrence are mapped as a column of the table corresponding to its parent element. Note that while most techniques consider primitives that map XML constructs to pure relational systems, some [32, 41] leverage object-relational features of relational systems. Some of the techniques such as X2R [13], are the extension of hybrid inline method [45] that preserves the content, structure, and semantic information as expressed in key and foreign key constraints. RRXS [12] pioneers in translation from XFDs (XML Functional Dependencies) to relational dependencies and it creates third normal form decomposition.

Cost-based methods

Cost-based methods use the additional information (usually query statistics, element statistics, etc.) and focus on creating an optimal schema for a certain application. LegoDB [5, 40] takes a cost-based approach, to derive a mapping that best suits a given application (characterized by a schema, query workload and document samples). It uses the information in the XML schema to derive several possible mapping alternatives, and selects the one that leads to the lowest cost for executing a given query workload over sample documents. Recently Microsoft researchers have proposed a search algorithm [10] that explores the combined space of logical and physical design, in conjunction with the relational advisor.

1.1.3 User-defined Mapping Methods

User-defined mapping methods are often used in commercial systems. This approach requires that the user first defines a target schema and then expresses required mapping using a system-dependent mechanism.

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 [38] provides an XMLTYPE to map XML data into an object table or view. IBM's DB2 Extender [25] provides two primary storage and access methods for XML documents: XML column and XML collection. MS SQL Server [19] uses OpenXML rowset providers to support XML. It maps XML data into an edge table, a parent-child hierarchical graph representation of XML data. POET [39] is an object-oriented database system. It maps each XML element into a separate object.

1.2 Storing XML in RDBMS

A typical system for storing XML using RDBMS is shown in Figure 1.1. *Relational schema generator* generates relational mapping for XML data. Relational mapping can be done by any of the methods, described in previous section. For relational mapping, the main objective is to find a relational configuration, which requires less storage size, yet handles XML query efficiently and correctly. XML documents are shredded by *XML shredder* and are stored in relational database. *Query mapping processor* maps input XQueries to SQL queries in order to retrieve data from relational database. The query results in the form of relational tuples are tagged and published back as XML data by *XML converter*.

A common feature of much of the previous work is that it has focused on *isolated* components of the relational schema, typically the table configurations (refer to Figure 1.2). A complete relational schema, however, consists of much more than just table configurations – it also includes integrity constraints, indices, triggers, and views. Therefore, viable XML-to-relational systems that intend to support real-world applications need to provide a *holistic* mapping that incorporates all fundamental aspects of relational schemas. In this thesis, we attempt to address

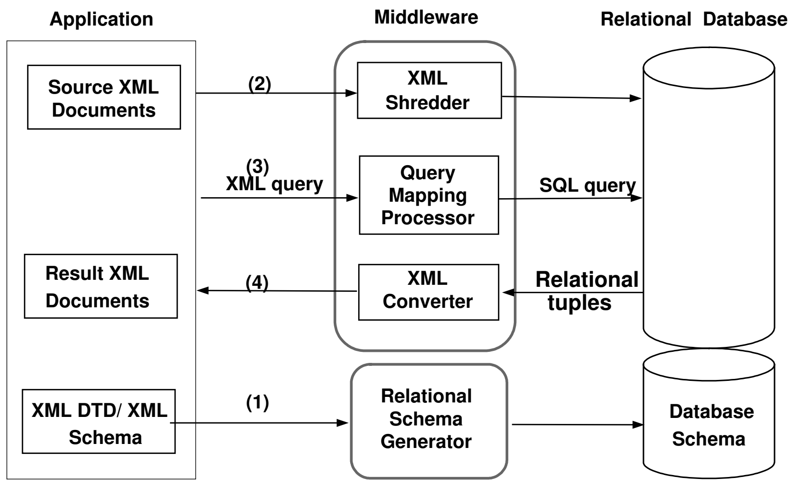


Figure 1.1: A typical system for storing XML using RDBMS

this issue by presenting a system called **ELIXIR** (**E**stablishing ho**LI**stic schemas for **X**ML **I**n **R**dbms) that produces holistic relational schemas tuned to the application workload (refer to Figure 1.3). Elixir incorporates schema-driven cost-based XML-to-relational mapping technique.

1.3 The Elixir system

The Elixir system is built around the LegoDB cost-based table-configuration framework [5, 22, 40], and has been successfully evaluated on a variety of real-world and synthetic XML schemas operating under a representative set of XQuery queries, using the DB2 database engine as the backend.

In producing XML-to-relational mappings, there are two possibilities: A *source-centric* approach, wherein the optimization of the mapping is carried out in the XML space, and then translated to the equivalent in the relational space; or a *target-centric* approach, where a mapping is made from the XML space to the relational space, and then optimized in the relational space to fine-tune the mapping. A key design feature of Elixir is that it performs *all* its mapping-related optimizations in the XML source space, rather than in the relational target space. The

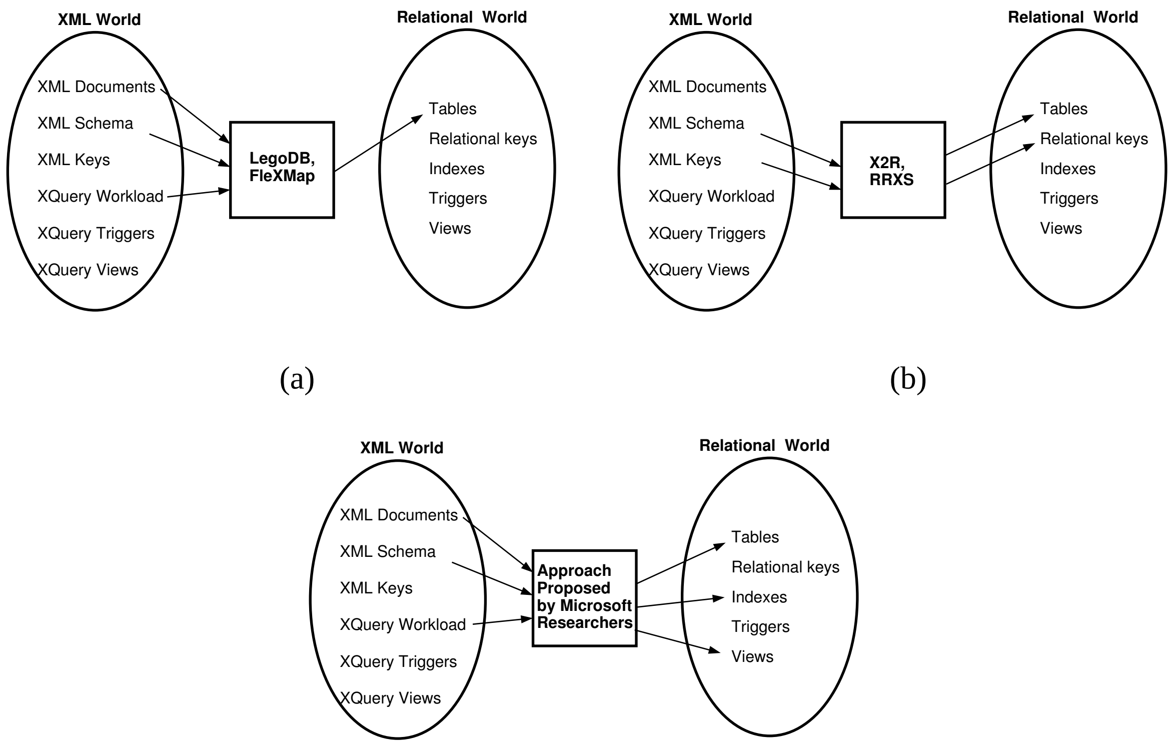


Figure 1.2: Existing XML-to-relational Mapping approaches

evaluation of the quality of these optimizations is done at the target, and the feedback is used to guide the optimization process in the XML space, in an iterative manner, resulting in a *dynamically-derived* mapping tuned to the application.

This approach is based on our observation that an organic understanding of the XML source can result in more informed choices from the performance perspective. As a case in point, Elixir significantly extends prior table-configuration cost based techniques, based on XML schema transformations, to seamlessly preserve the *unique*, *key* and *keyref* integrity constraints. In relational databases, XML key constraints can always be checked using triggered procedures involving joins or unions. However, such stored procedures are much more expensive to evaluate than key and foreign key constraints in relational databases [14]. To fully leverage database technology for constraint checking, we therefore wish to map XML key and keyref constraints to relational key and foreign key constraints. Cost-based strategies use schema transformations to explore the search space of different relational configurations. Our study shows that

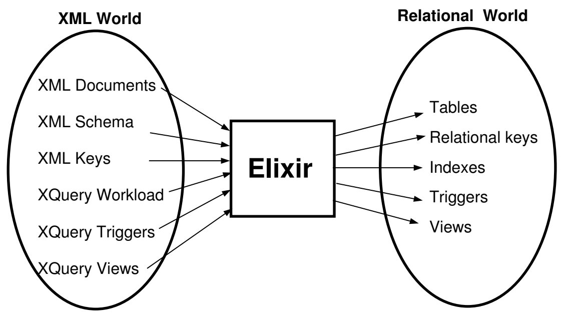


Figure 1.3: Our proposed XML-to-relational Mapping approach

propagating XML keys to relations in the form of primary keys and foreign keys results in the invalidation of schema transformations. We have developed the rules that are based on XML keys to determine the validity of a transformation before applying that transformation. We have also introduced more powerful variant of type split and type merge, which is necessary for mapping XML keys to relational keys. Beneficial side-effects of incorporating these constraints are improved table configurations and a substantial reduction in the optimization search space.

With regard to index selection too, we quantitatively show that the source-centric approach is preferable – that is, it is better to choose the best set of path-indices at the XML source and then map these choices to relational equivalents, as compared to using the relational engine's index advisor to identify a good choice (the latter approach has been taken in a recent paper by Microsoft researchers [10]). 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. This is consistent with the observation in [33] that schema decomposition and query translation are interdependent and should therefore be handled in an integrated manner.

In addition to production of table configurations, integrity constraints, indices, Elixir can also map XML triggers and XML views to SQL triggers and relational views, respectively. We demonstrate that only a subset of XML triggers appear to be directly mappable to SQL triggers and Elixir incorporates an algorithm for detecting such mappable triggers and generating the associated mapping. For the remainder, that is, the non-mappable triggers, Elixir uses stored procedures that can be called by the middleware at run-time. While the costs of mappable trig-

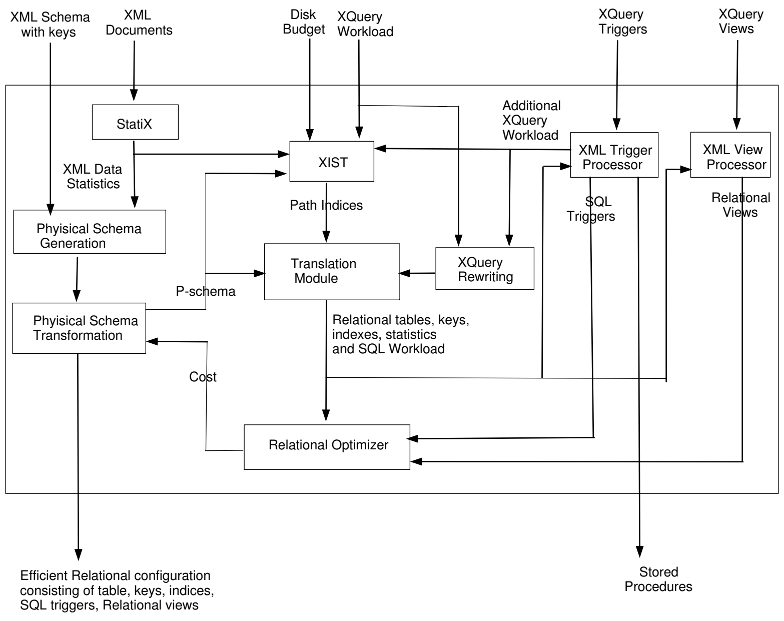


Figure 1.4: Architecture of the Elixir system

gers are natively modeled by the relational optimizer, an additional query workload equivalent to the non-mappable triggers is included in the XML query workload. The advantage of considering XML triggers (i.e. creating SQL triggers for *mappable XML triggers* and additional query workload for *non-mappable XML triggers*) is that the resultant relational configuration is efficient not only for the given workload but also for the queries involved in triggered actions. Our experimental results show that considering XML triggers during the tuning of relational configuration result in better final relational configuration as compared to that when XML triggers are ignored. With regard to views as well, translating materialized views specified in XML to relational backends, result in better relational configuration.

In a nutshell, the Elixir system attempts to make progress towards achieving "industrial-strength" mappings for XML-on-RDBMS.

1.4 Architecture of Elixir system

In designing Elixir, we have consciously attempted, wherever possible, to incorporate the ideas previously presented in the literature – in particular, we use the LegoDB system [5], with its associated FleXMap [40] and StatiX [22] components, and the XIST path-index selection tool proposed in [42].

The overall architecture of the Elixir system is depicted in Figure 1.4. Given an XML schema and statistics extracted from XML documents (using StatiX [22]), Elixir first generates an initial physical schema. Propagation of XML keys to relational keys is possible only for the *valid* physical schema that is obtained after applying *valid schema transformations* (details of *valid schema transformations* are given in Chapter 4). Valid schema transformations are then repeatedly applied to initial physical schema and the process of schema/query translation and cost estimation is repeated for each transformed physical schema until a good configuration is found.

XML Trigger Processor maps *mappable XML triggers* to SQL triggers and *non-mappable XML triggers* to stored procedures, which can be called by middleware at runtime. To account for the cost of the *non-mappable triggers,* Elixir adds query workload equivalent to *non-mappable triggers* to input query workload. XML view processor maps XML views and materialized XML views specified by the user to relational views and materialized query tables, respectively.

XIST (XML Index Selection Tool) [42] selects the set of indices given a combination of a query workload, XML schema, and data statistics. It evaluates the benefit of an index by comparing the total execution costs for all queries in the workload before and after index is available. In addition, it compares this benefit with the cost of updating the index and recommends a set of indices that is most effective for a constraint on the amount of disk space. The advantage of using XIST is that it can make best use of information extracted from the XML schema and the statistics information. For efficient computation of path indexes at the relational backend, we convert the path index to set of relational indices. We also need to rewrite the XQueries to take benefit from the available path indices. This query rewriting is based on the concept of *path equivalence classes* of XML schema. These relational indices are given to optimizer, in addition to relational tables, statistics, and SQL workload (equivalent to rewritten XQueries), for computing the cost of the queries.

To make our objectives concrete, a sample fragment of inputs from XML banking application are shown in Figure 1.5 and a relational mapping derived from Elixir for these inputs is shown in Figure 1.6.

1.5 Contributions

In summary, the contributions of the thesis work include

- Techniques for translating XML Schema integrity constraints to relational constraints, for integrating these XML Schema integrity constraints into the optimization process, and quantitative demonstration of their benefit in pruning the mapping search space.
- Techniques for propagating XML index selections to the relational target, quantitative demonstration of their improvement over the choices made by the relational index advisor, and utilizing the index choices to guide the XQuery-to-SQL translation process.
- Techniques for handling XML Triggers and XML views, empirical results of improvement of final relational configuration obtained (due to consideration of XML Triggers and XML views) during tuning process of relational configuration.
- Incorporation of these techniques in the Elixir system, which produces holistic schema mappings from XML sources to relational backends.

1.6 Organization

The remainder of this thesis is organized as follows: Related work is reviewed in Chapter 2. In Chapter 3, an overview of the Elixir system is presented. The constraint mapping technique and its integration with cost-based optimization is discussed in Chapter 4. Index mapping is addressed in Chapter 5. Chapter 6 explains the mapping procedure for XML triggers and XML views. Our conclusions are summarized in Chapter 7.

```
– – XML Schema
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema>
    <xsd:element name=”bank”>
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name=”country” type=”CountryType” minOccurs=”0” maxOccurs=”unbounded”/>
            </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/savings-acc-number OR
    $customer/acc-number = $account/checking-acc-number) 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/savings-acc-number OR
       $customer/acc-number = $account/checking-acc-number) 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-number>$customer/acc-number</customer-acc-number>
        <balance>$customer/balance</balance>
    </customer-balance>
DATA INITIALLY IMMEDIATE REFRESH IMMEDIATE

. . .
```

Figure 1.5: Sample input from XML Banking Application

– – 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, Savings-or-Checking-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 Account-key UNIQUE (Savings-or-Checking-account-number, parent-Country);
ALTER TABLE Customer ADD CONSTRAINT Account-fkey FOREIGN KEY (account-number, parent-Country) REFERENCES Account(Savings-or-Checking-account-number, parent-Country);
. . .
– – Recommended Indices
CREATE INDEX name-index ON Customer(name);
CREATE INDEX acc-number-index ON Account(Savings-or-Checking-account-number, 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 customer-name STRING,
                 IN city-name STRING, IN city-state STRING,...)
BEGIN
    Send-mail(customer-name, city-name, city-state, ...)
END
```

. . .
– – Materialized Query Tables

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

. . .
– – Relational views

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

. . .

Figure 1.6: Example Elixir Mapping

Chapter 2

Survey of Related Research

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 integrates cleanly with a cost-based dynamic mapping process. In this chapter, we shall briefly overview previous works that are related to this thesis.

2.1 XML-to-relational mapping

Several XML-to-relational mapping techniques have been proposed, which define set of rules to map XML schema primitives to their relational counterparts. This section includes the brief summary of these techniques.

2.1.1 Inlining Techniques

In [45], authors have proposed various DTD-driven inlining techniques for XML-to-relational mapping: Basic, Shared and Hybrid. The decision of whether to create a table for an element or to inline it with its parent is central to these approaches and is made on the basis of whether or not an element is shared by other elements in the DTD. These solutions vary in the amount of redundancy they may generate (an element could be inlined in several of its referencing elements). In all of them, key-foreign key relationship is used to capture document structure.

2.1.2 Constraints-Preserving Inlining Techniques

Inlining methods discussed in earlier section, only take into account structural constraints of the existing XML documents. With regard to XML keys, CPI [36], X2R [13] techniques have been proposed for mapping to relational equivalents. These systems are applicable for static mapping techniques like those proposed in [45]. A framework called XFD is presented in [12], to express functional dependencies including keys, then map them to relational dependencies and finally create a third normal form decomposition.

In this thesis, 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 the earlier work and Elixir is that the former produces 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. This section provides brief overview of constraints-preserving inlining techniques.

CPI

CPI is a constraints-preserving algorithm based on the hybrid inlining algorithms proposed in [45]. Given a DTD, CPI can derive semantic knowledge from it and it is possible for CPI to preserve the knowledge by representing it as semantic constraints in relational database. This is different from techniques that merely consider the structural constraints of DTD. CPI prevents the possibility that transformation algorithm may cause inconsistency between the DTD and the generated relational schema. The algorithm derives semantic knowledge only from DTD files. This leads to another potential problem. DTD is the simplest XML structural definition language. It has limited expressive power to represent semantic constraints compared with XML schema, in which semantic information captured is far greater than in a DTD.

X2R

X2R maps an XML document together with its constraints into a relational schema so as to check XML key and keyref constraints using key and foreign key constraints. There are some differences between the X2R algorithm, hybrid-inlining, and CPI. First, X2R starts from a set of key and reference relations, which capture semantic information. Second, relations that are separated in hybrid inlining may be coalesced by paths that end with a disjunction. Moreover, the key and the referential relations may inline ancestors other than the parent, i.e. the context node may be a non-parent ancestor. In the process of X2R, the notion of constraint relations, which explicitly capture XML key and keyref constraints, is proposed. It also presents a mapping from an XML document to a relational instance, which extends constraint relations to capture the complete content and structure of the document. Unlike CPI, mapping in X2R is guided by the XML key and keyref constraints rather than by the DTD. Another direct benefit of this storage mapping is the ability to efficiently check XML constraints using relational key and foreign key constraints.

RRXS Redundancy Reducing XML storage in relations

As in relational databases, functional dependencies for XML (XFDs) are used to describe the property that the values of some attributes of a tuple uniquely determine the values of other attributes of the tuple. The difference lies in that attributes and tuples are basic units in relational databases, whereas in XML data, they must be defined using path expressions. For example, consider a constraints such as if two books have the same ISBN, they must have the same title. In the form of XFD, it can be written as follows:

$$//\mathsf{book}/\mathsf{ISBN}/\mathrm{value}() \rightarrow //\mathsf{book}/\mathsf{title}/\mathrm{value}()$$

RRXS [12] provides a framework (XFDs) to express structural and semantic constraints. It uses a reduced set of the input XFDs to guide the design of the target relational schema, by translating XFDs to relational functional dependencies and creating a third normal form (3NF) decomposition.

2.1.3 Cost-based Flexible Mapping Techniques

Most of the previous techniques focus on lossless translation of XML documents into relational database. Various techniques are proposed to make sure that the content, structure, and semantics are preserved in the produced relational configuration. However, all previous techniques give fixed mapping regardless of the types of XML applications used. This is not desirable since different applications imply different query patterns and thus impose different demands on the underlying relational database.

Considering all these, it might be a good idea to make use of application characteristics to guide XML-to-relational mapping process. LegoDB [5, 40] proposed a novel cost-based approach to generate relational storage mappings for XML data by taking into account the application characteristics such as XML schema, query workload, and document samples. LegoDB system exploits a richer set of mapping primitives. In addition to parent-child relationships, LegoDB also takes into account additional schema constructs such as choice and repetition, and it allows multiple mapping functions for a given construct. For example, besides the option to create a table for a set-valued element, LegoDB also considers inlining one or more occurrences of the repeated element within its parent (through the repetition split transformation). LegoDB uses the information in the XML schema to derive several possible mapping alternatives and selects the one that leads to the lowest cost for executing a given query workload over sample documents.

2.2 Index selection

Index selection is one of the important aspects of physical database design. In this section, we will review various index selection techniques proposed in native XML databases and relational databases.

2.2.1 In Native XML databases

For native XML databases, a variety of path indices such as Dataguides [24], T-indices [37], APEX [16], etc. have been proposed.

In [37], Milo and Suciu describe T-indexes, a generalized path index structure for semi-structured documents. A particular T-index is associated with a set of paths that match a path template. Their approach uses bisimulation relations to efficiently group together nodes that are indistinguishable with respect to the given template into path equivalence classes. If two nodes are bisimilar, they have the same node label and their parents share the same label. In the 1-index [37], data nodes that are bisimilar from the root node are stored in the same node of the index graph. The size of the 1-index can be very large compared to the data size, thus A(k)-index [31] has been proposed to make a trade off between the index performance and the index size.

Chung et al. have proposed APEX [16], an adaptive path index for XML documents. The main contributions of APEX are the use of data-mining techniques to identify frequently used subpaths, and the implementation of index structures that enable incrementally updates to match the workload variations. APEX exploits the query workload to find indices that are most likely to be useful.

In [30], Kaushik et al. have proposed F&B indexes that use the structural features of the input XML documents. F&B indexes are forward-and-backward indices for answering branching path queries. Authors have also proposed some heuristics in choosing indices, such as prioritizing short path indices over long path indices [30].

Recently proposed XIST [42] is a tool that can be used by an XML DBMS as an index selection tool. XIST exploits XML structural information, data statistics, and query workload to select the most beneficial indices. XIST employs a technique that organizes paths that are evaluated to the same result into equivalence classes and uses this concept to reduce the number of paths considered as candidates for indexing. XIST selects a set of candidate paths and evaluates the benefit of an index on each candidate path based on performance gains for non-update queries and penalty for update queries. XIST also recognizes that an index on a path can influence the benefit of an index on another path and accounts for such index interactions.

While in principle, any of these could have been used for source-centric index choices in Elixir, we have chosen to use the XIST [42] tool because of its workload and resource-based index choices, an essential feature in practice.

2.2.2 In Relational databases

Many commercial relational database systems employ index selection features in their query optimizers. For example, IBM's DB2 Universal Database (UDB) uses DB2 Advisor [52], which recommends candidate indices based on the analysis of workload of SQL queries and models the index selection problem as a variation of the knapsack problem. The Microsoft SQL Server [11] uses simpler single-column indices in an iterative manner to recommend multi-column indices. That is, the indices on fewer columns are considered before indices on more number of columns.

Recently, Microsoft researchers [10] have proposed a search algorithm that explores the combined space of logical and physical design, in conjunction with the relational advisor for given XML schema, sample documents and query workload. The index advisor of the Microsoft SQL server 2000 is used to get the recommendation for indexes, materialized views, and partitions to improve the performance of queries in the workload. 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.

2.3 XML Triggers

In order to make XML repositories fully equipped with data management capabilities, suitable query and update languages are being developed. However, once the user is allowed to perform updates, it is perceivably necessary to guarantee the correctness of his/her updates, especially if document validity or semantic constraints are violated [6]. This problem can be addressed by exploiting the well-grounded concept of active rules.

XQuery [4] is a language from the W3C designed to query and format XML data. In [6], authors have proposed *Active XQuery*, which is an active extension to W3C proposed standard XQuery [4] language for defining XQuery triggers.

In [46], authors have addressed the issue of triggers over XML view of relational data by translating triggers over XML views to SQL triggers and update over relational data will trigger the action. However, in Elixir, updates are done on the XML data and updates are done in transparent manner to the relational data.

2.4 XML Views

Since the early days of data models, the concepts of views were used to give different perspectives and abstraction for underlying base data, for different users and uses.

Serge Abiteboul [1] have proposed a declarative notion of XML views. Abiteboul pointed out that, a view for XML, unlike classical views, should do more than just providing different presentations of underlying data [1]. In addition, he argues that an XML view specification should rely on a data model (like ODMG model) and a query language. Later, Sophie Cluet et al. [18] formally provided an XML view definition as

”....A view defined by a set of pairs $< p, p >$, called mappings, where p is a path in the abstract DTD and p a path in some concrete DTD..” [18].

In [15], authors have proposed a systematic approach to design valid XML views. In our system, we assume only valid XML views are provided as input. In [2], authors have proposed a framework for exploiting materialized XPath views to expedite processing of XML queries. They have developed XPath matching algorithm to determine when such materialized XPath views can be used to answer a user query containing XPath expressions.

Chapter 3

Elixir System and Performance Methodology

Input XML source environment consists of element-schema, constraints (*unique,key,keyref*), document statistics, and query workload, XML triggers, and XML views as also an index space budget. The Elixir system aims to establish an efficient and holistic relational schema, consisting of table configurations, relational keys, a set of relational indices that adhere to the space budget, SQL triggers, and views for given input XML source environment. In this chapter, we describe the detailed algorithm of the Elixir system.

3.1 Input

Consider input XML source environment given to Elixir system (refer to Figure 3.1). The XML element-schema, constraints, query workload, XML Triggers, XML views are typically supplied by the user, the XML document statistics can be generated by tools like StatiX [22] from the document repository. This section discusses the various inputs taken by Elixir.

3.1.1 XML Schema

XML Schema [49] describes contents, structure, and semantics of XML documents. XML schemas provide a consistent way to validate XML. XML Schema reproduces the full capabilities of DTD [7], so existing DTD document schemas can be translated to XML Schema without problems. However, it goes beyond these capabilities, allowing additional types of constraints to

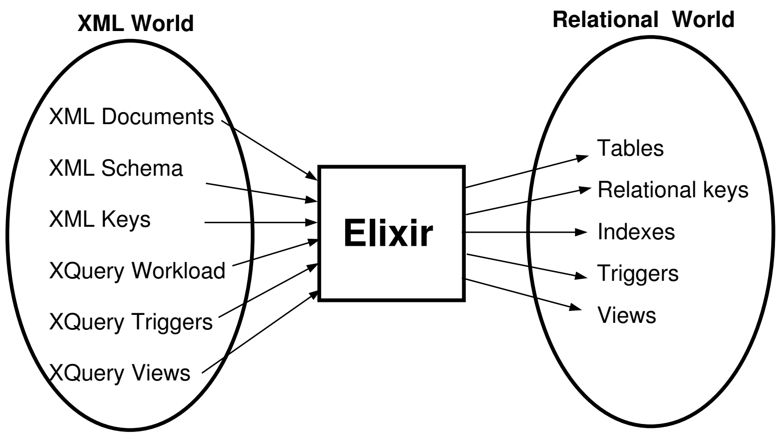


Figure 3.1: Elixir system

be specified such as more built-in data types, support for user-defined data types, more flexible occurrence indicators, import/export mechanism, integrity constraints mechanism, refinement mechanism, and extensibility mechanism.

3.1.2 XML Constraints

Before giving XML schema key specifications, consider the sample document shown in Figure 3.2. The document contains information about bank customers by country and bank branches by country and cities. Suppose, we wish to assert that all accounts should have unique account number. For example, since there is savings-account with account number 101, we cannot add savings account or checking account with the same account number. We might wish to assert that customer's account number should be one of the account numbers, which is defined as savings-account-number or checking-account-number.

In XML Schema, three types of identity constraints can be defined: *unique*, *key*, and *keyref*. Examples of these XML constraints are shown in the following XML schema fragment:

```
<element name="country" type="Country">
    <key name="account-number-key">
        <selector xpath=".//account"/>
        <field xpath="savings-acc-number | checking-acc-number">
    </key>
    <keyref name="customer-account" refer="account-number-key">
        <selector xpath="./customer"/>
        <field xpath="acc-number">
    </key>
</element>
```

3.1.3 XQuery Workload

Different queries have different importance according to the frequency of execution. Thus, Elixir system uses the XQuery workload, which consists of set of XQueries along with their frequency of execution. Frequency of execution is used for weighting the cost of the query. All previous approaches such as LegoDB [5], FleXMap [40], [10] have considered only *read only* workload of the queries. Elixir considers read only queries as well as update queries. As update extension is not part of XQuery standard, we have used update extension proposed in [48]. An update is a sequence of primitive operations of the following types:

Insert(content): inserts new content (which can be simple type, element, attribute, or reference) into target. An attempt to insert an attribute with the same name as an existing attribute fails. An attempt to insert a reference with the same name as an existing IDREFS adds an extra entry into the IDREFS.

Delete(child): if the child is a member of the target object, it is removed. Valid types for child include simple type, attribute, IDREF within an IDREFS list, and element. If the child is a reference within an IDREFS, only the single entry is removed – the remainder of the IDREFS is preserved.

Rename(child, name): if the child is a non-simple type member of the target object, it is given a new name. Note that we cannot rename an individual IDREF within an IDREFS; such a rename operation will rename the entire IDREFS.

```
<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>O112</id>
        <address>...</address>
      </head-office>
      <branch-office>
        <id>O321</id>
        <address>...</address>
      </branch-office> ...
      <atm>
        <id>A1231</id>
        <address>...</address>
      </atm> ...
      <account>
        <savings-acc-number>101</savings-acc-number>
        <balance>1232423</balance>
      </account>
      <account>
        <checking-acc-number>102</checking-acc-number>
        <balance>645634</balance>
      </account>...
    </city> ...
  </country> ...
</bank>
```

Figure 3.2: Sample XML Document (bank.xml)

Replace(child, content): atomic replace operation, equivalent to (Insert(content), Delete(child)).

3.1.4 XML Triggers

As XML triggers are not a part of XQuery standard, here we use *Active XQuery* [6], an active extension to the W3C-proposed standard XQuery [4] language, adapting the SQL3 notions.

An XQuery trigger consists of four components: the triggering operation, the triggering granularity, the trigger condition, and the trigger action. A trigger is invoked when one of its triggering operations occur. It is being considered when its condition is under evaluation. It is executed when its action is performed.

The syntax of an XQuery trigger [6] is the following:

```
CREATE TRIGGER Trigger-Name
[WITH PRIORITY Signed-Integer-Number]
(BEFORE|AFTER)
(INSERT|DELETE|REPLACE|RENAME)+
OF XPathExpression (,XPathExpression)*
[FOR EACH (NODE | STATEMENT)]
[XQuery-Let-Clause]
[WHEN XQuery-Where-Clause]
DO (XQuery-UpdateOp|ExternalOp)
```

- The `CREATE TRIGGER` clause is used to define a new XQuery trigger, with the specified name.

- Rules can be prioritized in an absolute ordering, expressed with an optional `WITH PRIORITY` clause, which takes as argument any signed integer number. If this clause is omitted, the default priority is zero.

- The `BEFORE/AFTER` clause expresses the triggering time relative to the operation.

- Each trigger is associated with a set of update operations (insert, delete, rename, replace), adopted from the update extension of XQuery [48].

- The operation is relative to elements that match an XPath expression (specified after the OF keyword), i.e. a step-by-step path descending the hierarchy of documents (according to [17] and its update-related extensions). One or more predicates (XPath filters) are allowed in the steps to eliminate nodes that fail to satisfy given conditions. Once evaluated on document instances, the XPath expressions result into sequences of nodes, possibly belonging to different documents.

- The optional clause `FOR EACH NODE/STATEMENT` expresses the trigger granularity. A statement-level trigger executes once for each set of nodes extracted by evaluating the XPath expressions mentioned above, while a node-level trigger executes once for each of those nodes. Based on the trigger granularity, it is possible to mention the transition variables in the trigger :
  - If the trigger is node-level, variables `OLD_NODE` and `NEW_NODE` denote the affected XML element in its before and after state.
  - If the trigger is statement-level, variables `OLD_NODES` and `NEW_NODES` denote the sequence of affected XML elements in their before and after state.
- An optional *XQuery-Let-Clause* is used to define XQuery variables whose scope covers both the condition and the action of the trigger. This clause extends the `REFERENCING` clause of SQL3, because it can be used to redefine transition variables.
- The `WHEN` clause represents the trigger condition, and can be an arbitrarily complex XQuery where clause. If `WHEN` clause is omitted, default value is `TRUE`.
- The action is expressed by means of the `DO` clause, and it can contain accomplished through the invocation of an arbitrarily complex update operation. In addition, a generic *ExternalOp* syntax indicates the possibility of extending the XQuery trigger language with support to external operations, permitting, e.g., to send mail or to invoke SOAP procedures.

For a complete syntax of XQuery refer to [4] and for the syntax of the update language, refer to [48].

3.1.5 XML Views

The notion of views is essential in databases. It allows various users to see data from different viewpoints. Although XQuery [4] currently does not provide standard for defining XML views, we can easily extend it to include the definition of views [15] as follows:

”`CREATE VIEW` *view_name* `AS`” followed by FLWR expression

Above definition can be extended to define materialized XML views 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 a 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 TABLE` *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 wants 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 issues `REFRESH TABLE` statement.

3.1.6 XML Documents

Statistical information (about the values and structure) from the given XML document is necessary to derive accurate relational statistics, which are needed by the relational optimizer to accurately estimate the cost of the query workload. We have used recently proposed StatiX [22], which is a XML Schema-aware statistics framework to gather the statistics of input XML documents.

3.1.7 Disk Budget

Disk budget is the limit for the size of indexes in the output relational configuration. Elixir also allows user to specify no disk limit by providing disk limit as -1.

3.2 The Elixir schema mapping algorithm

A high-level pseudocode of the mapping algorithm of Elixir system is given in Algorithm 1. Since the space of potential relational mappings is exponentially large, a greedy heuristic is used to find an efficient mapping [5]. The first step in the algorithm (line 2) is to obtain the equivalence classes, which represent the structural equivalent groups of the XML schema – details in Chapter 5. Next, a relational schema is generated on the basis of original XML environment (line 3). XML views are mapped to relational views by *MapXMLViews* (line 4). In next step, additional query workload to account for *Non-mappable XML trigger* is obtained (line 5) and then, the runtime cost of the XML workload, after translation to SQL, on this schema, is determined by accessing the relational engine's query optimizer (lines 6, 7) – in our current system, the IBM DB2 engine is utilized for this purpose. Subsequently, the original XML schema is transformed in a variety of ways (lines 9, 10), the relational runtime cost for each of these new schemas is evaluated, and the transformed schema with the lowest cost is identified (line 25). 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 procedure for obtaining a relational schema – function *ConvertToHolisticRelSchema* – is described in Algorithm 2. Here, the function *GenerateRelationsAndKeys* generates the table configuration and relational keys for the given XML element-schema and keys (line 1). Additional workload corresponding to *non-mappable XML triggers* is obtained using *GetAdditional-Workload* (line 2). Subsequently, the appropriate indices in the XML world are determined – in our current system, this is done using the XIST tool [42], which takes an XML element-schema, query workload (which consists of input XML query workload and additional query workload), and disk space constraint as input, and recommends the most beneficial *path indices*. XIST uses path equivalence classes (EQs) to reduce the number of paths considered as candidates for indexing. For each path index recommended by XIST, the appropriate relational indices need to be created such that the corresponding path can be evaluated efficiently (line 6). The conversion of path indices to relational indices involves adding of extra columns to the relations, and therefore this function returns a modified relational table configuration. Elixir applies disk limit to the path-indices by taking into account the size of their relational equivalents (details are given in Chapter 5). The statistical summary of XML data in the form of structural and value histograms is converted to relational statistics such as, for tables, the number of data pages and number of rows, and for columns, the number and distribution of distinct values, and the number of nulls (line 9). Finally, XML triggers are processed and *mappable XML triggers* are converted to SQL triggers and *non-mappable XML triggers* are converted to stored procedures (details are given in Chapter 6) .

The translation of XML queries to SQL – function *TranslateToSQL* – uses the path indices recommended by XIST and the path equivalence classes to come up with a good mapping.

With regard to the XML schema transforms, we consider those presented in [5], namely, *Inline/Outline*, *Type-split/merge*, *Union distribution/factorization*, and *Repetition split/merge*. A major difference, however, is that only a *subset* of the applicable transforms may be valid at each step, because the other transforms lead to violations of the XML key constraints. Therefore, in each iteration, a list of valid transforms is generated from the set of applicable transforms (lines 8,9). Each valid transform is applied in turn to the schema and the transform that results in the minimum cost relational configuration is chosen to produce the XML schema that will be used as input in the next iteration of the algorithm (lines 12-24). This process is repeated until the current relational configuration reaches a fixed point and cannot be improved.

3.3 Performance Methodology

Elixir has been successfully evaluated on a variety of real-world and synthetic XML schemas operating under a representative set of XQuery queries, using the DB2 database engine as the backend. Our experimental setup consists of a standard Pentium-IV machine running Linux, with DB2 UDB v8.1 as the backend database engine. Four representative real-world XML schemas: *Genex* [23], *EPML* [20], *ICRFS* [26], *TourML* [50], 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.[1] The salient summary statistics of these documents are given in Table 3.1.

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

	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 3.1: Details of the schemas used in the experiments

As Elixir aims to establish an efficient and holistic relational schema, we use *cost of final relational configuration* as the performance metric in our experiments. It is the cost given by optimizer (in timerons) for executing the target workload on the relational configuration obtained at the end of tuning process.

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

Algorithm 1 Elixir Schema Mapping Algorithm

```
Input: $xS$: XML schema, $xK$: XML keys, $xW$: XML query workload, $xTr$: XML Triggers, $xV$:
    XML Views, $xStats$: XML data statistics, $dlimit$ : disk space constraint
Output: $rT$: relational table-configuration, $rK$: relational keys, $rStats$: relational statistics, $rI$: rela-
    tional indices, $rTr$: relational triggers, $rSp$: relational stored procedures, $rV$: relational views
 1: $PrevCost$ = $\infty$;
 2: $EQ$ = FindEQs ($xS$);
 3: ($rT$, $rK$, $rStats$, $rI$, $xPI$, $rTr$, $rSp$) = ConvertToHolisticRelSchema ($xS$, $xK$, $xW$, $EQ$, $xStats$,
    $dlimit$);
 4: $rV$ = MapXMLViews ($xV$, $rT$);
 5: $xAw$ = GetAdditionalWorkload ($xTr$, $xW$);
 6: $SQL\_W$ = TranslateToSQL ($rT$, $xPI$, $EQ$,($xW$+$xAw$));
 7: $Cost$ = GetCost ($rT$, $rK$, $rStats$, $SQL\_W$, $rI$, $rTr$, $rV$);
 8: while $Cost < PrevCost$ do
 9:   $PrevCost$ = $Cost$;
10:   $xforms$ = ApplicableTransfroms($xS$);
11:   $vxforms$ = FilterInvalidTransforms ($xforms$, $xK$);
12:   for all $T_v$ in $vxforms$ do
13:     $xS'$ = ApplyTransform($T_v$, $xS$);
14:     $EQ'$ = FindEQs ($xS'$);
15:     ($rT'$, $rK'$, $rStats'$, $rI'$, $xPI$, $rTr$, $rSp$) = ConvertToHolisticRelSchema ($xS'$, $xK$, $xW$, $EQ'$,
        $xStats$, $dlimit$);
16:     $rV$ = MapXMLViews ($xV$, $rT$);
17:     $xAw$ = GetAdditionalWorkload ($xTr$, $xW$);
18:     $SQL\_W$ = TranslateToSQL ($rT'$, $xPI$, $EQ'$,($xW$+$xAw$));
19:     $Cost'$ = GetCost ($rT'$, $rK'$, $rStats'$, $SQL\_W$, $rI'$, $rTr$, $rV$);
20:     if $Cost' < Cost$ then
21:       $Cost$ = $Cost'$;
22:       $xform$ = $T_v$;
23:     end if
24:   end for
25:   $xS$ = ApplyTransform ($xform$, $xS$);
26: end while
27: ($rT$, $rK$, $rStats$, $rI$, $xPI$, $rTr$, $rSp$) = ConvertToHolisticRelSchema( $xS$, $xK$, $xW$, $EQ$, $xStats$,
    $dlimit$);
28: $rV$ = MapXMLViews ($xV$, $rT$);
29: return ($rT$, $rK$, $rStats$, $rI$, $rTr$, $rSp$, $rV$)
```

Algorithm 2 Deriving holistic relational schema

Function: ConvertToHolisticRelSchema
Input: xS: XML schema, xK: XML keys, xW: XML query workload, xTr: XML Triggers EQ: Equivalence classes of xS, $xStats$: XML data statistics, $dlimit$: disk space constraint
Output: rT: relational table configuration, rK: relational keys, $rStats$: relational statistics rI: relational indices, xPI: XML path indices, rTr: relational triggers, rSp: relational stored procedures

```
 1: ($rT$, $rK$) = GenerateRelationsAndKeys ($xS$, $xK$);
 2: $xAw$ = GetAdditionalWorkload ($xTr$, $xW$);
 3: $xPI$ = XIST($xS$, ($xW$ + $xAw$), $EQ$, $dlimit$);
 4: $I$ = $\{\}$;
 5: for all $PI$ in $xPI$ do
 6:     ($rT''$, $rI'$) = ConvertIndex ($rT$, $PI$, $EQ$);
 7:     $T$ = $T'$; $I$ = $I \cup I'$;
 8: end for
 9: $rStats$ = ConvertStats ($rT$, $xStats$);
10: ($rTr$, $rSp$) = ProcessXMLTriggers($xTr$, $rT$);
11: return ($rT$, $rK$, $rStats$, $rI$, $xPI$, $rTr$, $rSp$)
```

Chapter 4

XML Constraints to Relational Constraints

XML Schema allows one to mix DTD features with semantic information, such as integrity constraints in the form of keys and foreign keys. Integrity constraints are useful for semantic specification, query optimization, and data integration. In this chapter, we discuss the technique for translating XML integrity constraints to relational constraints. Initially, we describe the XML keys and related concepts in detail. In Section 4.2, the technique for generating constraints-preserving relations (by propagating XML keys to relational keys) is presented. Integration of these XML Schema integrity constraints into the optimization process is addressed in Section 4.3. Finally, we describe experimental evaluation of the technique discussed in this chapter.

4.1 XML Keys

XML Schema provides three integrity constraints: *unique*, *key* and *keyref*. 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 [9], the *unique* and *key* constraints can be written as

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

while the *keyref* constraint can be 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 as *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:

$$\begin{aligned} c &::= \; . \mid / \mid .q \mid /q \mid .//q \mid (c|c) \\ q &::= \; l \mid (q/q) \mid - \end{aligned}$$

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.

Using above notation, example keys for the sample *bank.xml* document shown in Figure 3.2, are given below:

- *account-number-key* : (//country,(.//account, {savings-acc-number | checking-acc-number}))

  Within a country, each account is uniquely identified by savings account number or checking account number.

- *customer-account* : (//country,(./customer,{acc-number})) KEYREF *account-number-key*

  Within a country, each customer refers to a savings account number or checking account number by acc-number.

Consider the bank example discussed earlier. As for the semantics of the document, one might wish to assert that country is identified by name and within a country, city is identified by its name and state. Within a country, offices i.e. head offices and branch offices are uniquely identified by their ID. Similarly, within a country, ATMs are also identified by their ID. For example, since there is already the branch office having ID ”O321” in India, we could not add another head office or branch office with the same ID in India. However, we could add another office with same ID in country other than India. Using syntax described earlier, these additional constraints can be written as follows:

- *country-key* : (//bank, (./country, {name}))
- *city-key* : (//country, (./city, {./name,./state}))
- *office-key* : (//country, (./city/head-office | ./city/branch-office , {id}))
- *atm-key* : (//country, (./city/atm, {id}))
- *customer-key* : (//country, (./customer, {cust-id}))

4.2 Generating Constraint-Preserving Relations

The *GenerateRelationsAndKeys* procedure takes an XML schema with constraints as input and produces a constraint-preserving equivalent relational schema. In relational databases, XML key constraints can always be checked using triggered procedures, involving joins or unions. However, such stored procedures are much more expensive to evaluate than key and foreign key constraints in relational databases [14]. For example, consider the XML key *account-number-key* : (//country,(.//account, {savings-acc-number | checking-acc-number})) and relational configuration as follows :

```
TABLE City (City-id-key INT, name STRING, state STRING,
Head-office-address STRING, parent-country INT)
TABLE SAccount(SAccount-id-key INT, savings-acc-number INT, balance
INT, parent-City INT)
TABLE CAccount(CAccount-id-key INT, checking-acc-number INT, balance
INT, parent-City INT)
```

Then to check the *account-number-key*, we need to define a trigger, which gets triggered when row is added in `SAccount` or `CAccount` and trigger performs the following steps:

1. Join `SAccount` with `City` and select `SAccount-id-key`, `Country-id-key` as `Account-id-key`, `Country-id-key`

2. Join `CAccount` with `City` and select `CAccount-id-key`, `Country-id-key` as `Account-id-key`, `Country-id-key`

3. Union the results obtained in step 1 and 2

4. Check if there is any duplicate value for pair of `Account-id-key`, `Country-id-key`. If yes, then it implies that addition of row is violating XML key, thus triggering action (i.e. addition of row) should be roll backed. Otherwise, there is no violation of *account-number-key*.

Execution of such stored procedure is very inefficient as compared to the constraints checking using primary keys and foreign keys [14]. Thus, to fully leverage database technology for constraint checking, we therefore wish to map XML key and keyref constraints to relational key and foreign key constraints. Recently proposed X2R [13] technique address the problem of mapping an XML document together with its constraints, into a relational schema so as to check XML key and keyref constraints using key and foreign key constraints.

Our technique is superficially similar to the X2R storage mapping algorithm [13], but a crucial difference is that they tailor the schema to fit the key constraints, thereby risking efficiency, whereas Elixir takes the opposite approach of integrating the key constraints with an efficient schema. Specifically, X2R uses XML keys to define constraint relations, relational keys and then uses inlining into constraint relations for the nodes that are not mapped in constraint mapping. In contrast, the *GenerateRelationsAndKeys* procedure 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

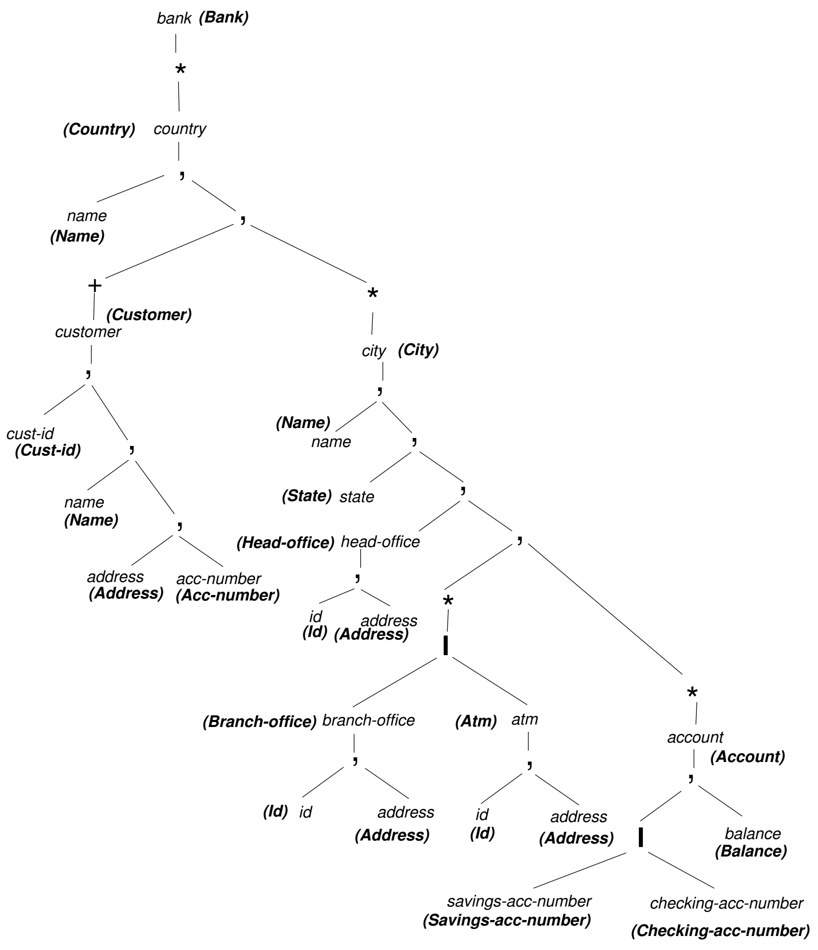


Figure 4.1: Schema tree for bank schema

cost-based iterative process to find the best constraint-preserving schema (this iterative process is discussed in the following section).

4.2.1 Schema Tree

The input XML schema is first converted into a *schema tree* using the representation proposed in FleXMap [40], in which the XML schema is expressed 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 discussed earlier is shown in Figure 4.1. 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 the optimization process (discussed in Section 4.3).

4.2.2 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 $N_1, N_2 \ldots, N_n$ 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:

account-number-key : (//country,(.//account, {savings-acc-number | checking-acc-number}))

As per this key, both **Savings-acc-number** and **Checking-acc-number** need to be mapped to the same column of the relation `Account`. Thus, the nodes corresponding to **Savings-acc-number** and **Checking-acc-number** from the schema tree should be associated.

4.2.3 From Schema Tree to Table Configuration

In the next step, the XML-to-relational mapping procedure proposed in LegoDB [5] is used in Elixir to create the table configuration, with an enhancement to handle 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.

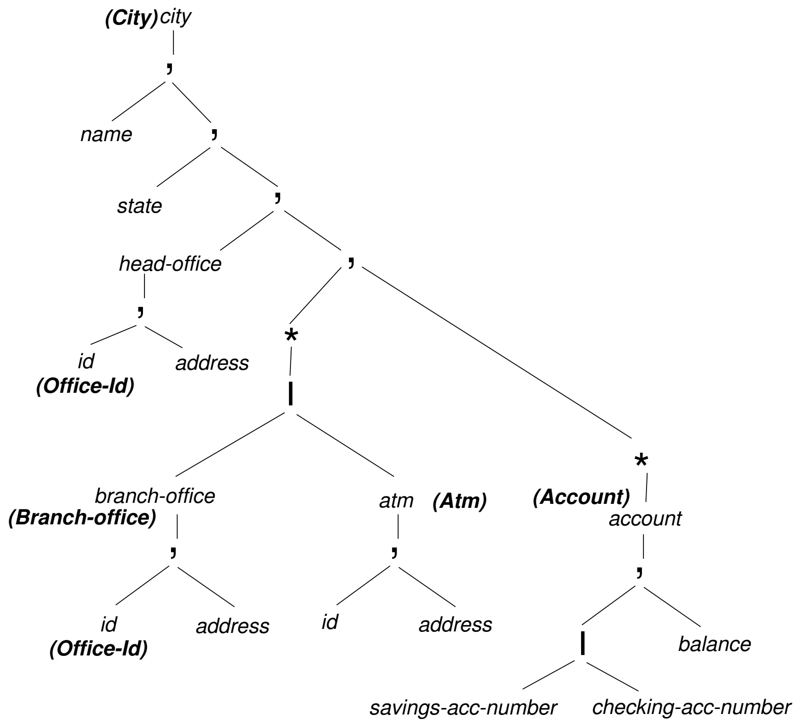


Figure 4.2: Partial Bank schema tree

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.

Note here that, in case of shared types, it is possible that two or more path expressions map to same relational schema. For shared types, we create the column corresponding to all parents, which stores ids for respective parents. Thus, it is possible to reverse the mapping without losing semantics.

For example, executing the above process on the schema tree shown in Figure 4.2 leads to the relational configuration, which is as follows:

```
TABLE City(                             TABLE Branch-office(
  City-id-key INT,                        Branch-office-id-key INT,
  name STRING,                            address STRING,
  state STRING,                           parent-City INT)
  Head-office-address STRING,
  parent-country INT)

TABLE Office-Id(                        TABLE Atm(
  Office-id-key INT,                      Atm-id-key INT,
  id STRING,                              id INT,
  parent-City INT,                        address STRING,
  parent-Branch-office INT)               parent-City INT)

TABLE Account(
  Account-id-key INT,
  Savings-or-checking-acc-number INT,
  acc-number-flag INT,
  balance INT,
  parent-City INT)
```

Note here that the relational configuration consists of five tables corresponding to the type names **City**, **Branch-office**, **Office-Id**, **Atm**, and **Account**. All the simple types are mapped to columns. The associated tree of **Savings-acc-number** and **Checking-acc-number** is mapped to column `Savings-or-checking-acc-number`, and an additional column, `acc-number-flag`, is created for identifying the account number type.

4.2.4 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 *valid schema trees*, 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,
  Savings-or-checking-acc-number 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 **Savings-acc-number** and **Checking-acc-number** is mapped to column `Savings-or-checking-acc-number`, 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 *account-number-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 {`Savings-or-checking-acc-number`, `parent-Country`}. The resulting final relational configuration is as follows:

```
TABLE Account (
   Account-id-key INT,
   Savings-or-checking-acc-number 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.

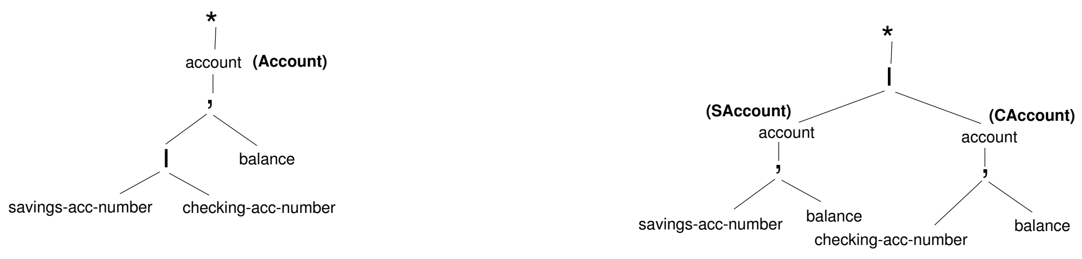
For example, consider a keyref *customer-account* : (//country,(./customer,$\{$acc-number$\}$)) KEYREF *account-number-key* and the relation for type **Customer** is as follows:

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

For the XML keyref *customer-account*, 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 *account-number-key* i.e. $\{$`Savings-or-checking-acc-number`, `parent-Country`$\}$ of `Account`.

4.3 Integration with Cost-based Search

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 [5, 40], 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*.



(a) Before union distribution (b) After union distribution

Figure 4.3: Union distribution of savings-acc-number | checking-acc-number

4.3.1 Filtering of Schema Transformations

As mentioned earlier, Elixir restricts the search space to only *valid schema trees* by filtering out the invalid schema transformations. In this section, we will explain the motivation for filtering of transformations followed by the procedure for filtering invalid transforms.

Motivation

Consider union distribution of account = savings-acc-number | checking-acc-number is distributed (refer to Figure 4.3(a)) , then the resulting schema tree is shown in Figure 4.3(b) and corresponding relational configuration will have account-numbers stored in two relations as follows:

```
TABLE SAccount(                   TABLE CAccount(
   SAccount-id-key INT,              CAccount-id-key INT,
   savings-acc-number INT,           checking-acc-number INT,
   balance INT,                      balance INT,
   parent-City 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 *account-number-key* constraint, savings-acc-number and checking-account-number should be mapped to 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,
   Savings-or-checking-acc-number 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 configuration, we need to explore space of only valid relational configuration. The simple solution for this is that carry out the transformation on schema tree and then check if relational keys equivalent to 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. This solution results in to lot of unnecessary work, which can be avoided, if we can detect the invalidity schema transformations before carrying out the schema transformation. In remaining chapter, 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 4.1, 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.

Union Distribution and Factorization

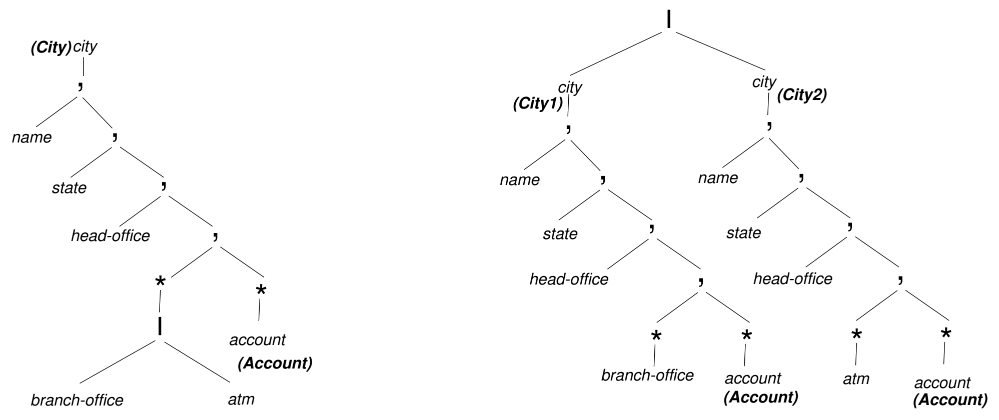
Union distribution can be used to separate out components of a union: $(a, (b|c)) \equiv (a, b)|(a, c)$. Conversely, the union factorization transform would factorize a union. Assume that union $t_1|t_2$ is being distributed, where t_1 and t_2 are subtrees of the schema tree.

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

Case 1: Consider the example discussed in previous section. Now we will try to analyze the cause for invalidation. The subtrees corresponding to savings-acc-number (t_1) and checking-account-number (t_2). Note that both the subtrees are on same field path of the *account-number-key* constraint. Thus, if the union distribution of these tree i.e. $t_1|t2$ is distributed, then in the resulting configuration, t_1 and $t2$ 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.**

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 4.4). 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.



(a) Before union distribution (b) After union distribution

Figure 4.4: Schema tree after union distribution of branch-office | atm

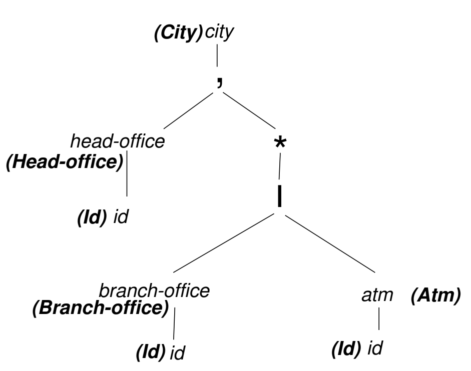
Type-Split and Type-Merge

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 4.5(a), the type **Id** is shared by the types **Head-office**, **Branch-office** and **Atm**.

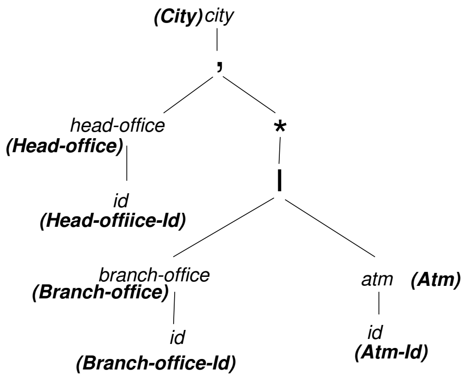
While, in principle, Type-Split and Type-Merge can be done with various subsets of the type occurrences in the schema, earlier work [40] focused on the extremes of *type-split-all* and *type-merge-all*. For example, the type **Id** is fully split in Figure 4.5(b) into **Head-office-Id**, **Branch-office-Id**, 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 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 4.5(c).

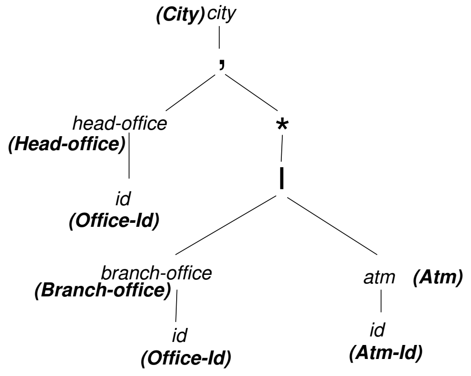
Let T be the type to be split and $Parent_1, \ldots, Parent_n$ be the parents of T with distinct annotations. The following procedure is used for selective type-split/merge:



(a) type-merge-all



(b) type-split-all



(c)selective type split and merge

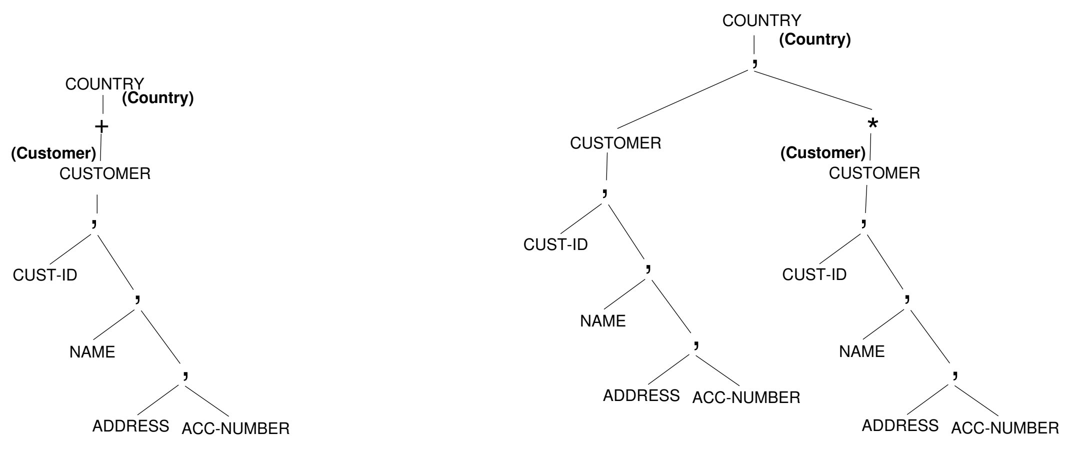
Figure 4.5: Type-Split and Type-Merge

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 4.5(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 4.5(c). This schema is consistent with the *office-key*



(a) Before Repetition Split (b) After Repetition Split

Figure 4.6: Example of Repetition Split

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 4.5(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 4.5(c).

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 (refer to Figure 4.6 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.

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 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 *account-number-key* constraint, **Savings-acc-number** and **Checking-acc-number** should be mapped to the same column of the relation `Account`. Thus, the nodes corresponding to **Savings-acc-number** and **Checking-acc-number** from the schema tree should be associated. The field tree corresponding to the field path of *account-number-key* is the tree obtained after associating the trees corresponding to **Savings-account-number** and **Checking-account-number**. 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.

4.3.2 Evaluating Configuration Efficiency

The XML schema tree obtained by applying the transformations is translated to relational configurations using the procedure explained in Section 4.2. The quality of the new relational configuration is assessed by computing cost estimates of executing the given query workload. This requires accurate statistics but since it is not practical to scan the base data to produce the statistics for each derived relational configuration, it is crucial that these statistics be accurately propagated as transformations are applied [40]. Merge operations preserve the accuracy of statistics, whereas split operations do not. Hence, in order to preserve the accuracy of the statistics, before the search procedure starts, all possible valid split operations are applied to the user-given XML schema, resulting in the so-called “fully-decomposed” schema [22]. Statistics are then collected for this fully decomposed schema, and subsequently, during the search process, only valid merge operations are considered. We use StatiX [22] to collect statistics of the filtered decomposed schema i.e. the schema obtained by applying only valid distributive transformations, and the cost of executing the query workload is obtained from the backend relational optimizer.

4.4 Experimental Evaluation

In this section, we present our experimental evaluation of the Elixir system. Specifically, we investigate the performance effects of source-centric inclusion of keys in the mapping process.

4.4.1 Experimental settings

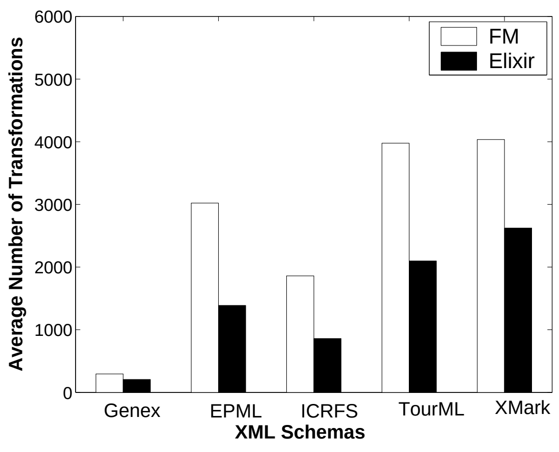
We carry out the experiments on the 4 real-world schema – *Genex* [23], *EPML* [20], *ICRFS* [26], *TourML* [50] and a synthetic XMark benchmark schema. Using the ToXgene tool [51], three types of documents were generated for each XML schema by varying the distribution of elements as *all-uniform*, *uniform-exponential*, 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 SQL equivalent of the query workload range from 5 to 15.

4.4.2 Effect of Keys

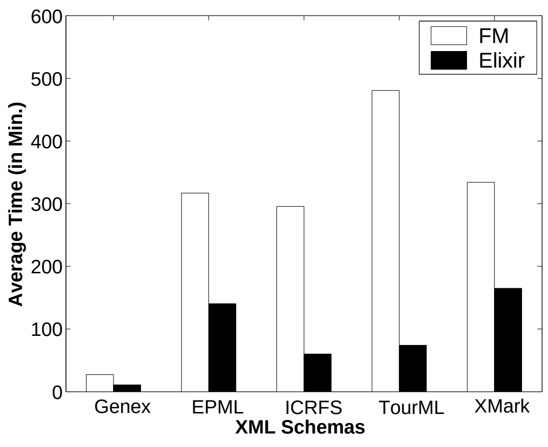
To serve as a baseline for assessing the effect of key inclusion, we compare the performance of Elixir (in the absence of other features like indices, triggers, and views) with that of FleXMap (FM) [40], which is a framework for expressing XML schema transformations and for searching the equivalent relational 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 [40].

Runtime Efficiency

We first 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 4.7(a) for the five XML schemas. We see there that the reduction ranges from 30% to 60%, arising out of the restrictive distributive transformation and selective merging transformations discussed in Section 4.3. For example, for the ICRFS schema, the average numbers



(a) Comparison of Search space



(b) Comparison of Time efficiency

Figure 4.7: Impact of Keys

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 4.7(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 here 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.

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 4.1. In this table, the $*$ indicates situations where the final FleXMap configuration did not satisfy the key constraints. Note here

	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 4.1: Cost of final relational configuration

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.

Chapter 5

Index Selection in Elixir

In this chapter, we discuss 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 marked contrast to the technique recently advocated by Microsoft researchers [10], where they use the index advisor of the SQL Server relational engine to propose a good set of indices.

Solving logical design and physical design independently leads to suboptimal performance [10] (i.e. if we first select logical mapping without considering physical design and then optimize the physical design of selected mapping leads to suboptimal performance) . On similar lines, Elixir explores the combined space of logical design (relational mapping by means of schema transformations) and physical design (index selection).

For finding good XML indices, we leverage the recently proposed XIST tool [42], which makes *path index* recommendations for given input consisting of an XML schema, query workload, data statistics, and disk budget. The benefit of an index is assessed by comparing the total execution costs for all queries in the workload with and without the index, and this benefit is compared against the cost of updating the index. Finally, the most effective set of indices that fit within the given disk budget is recommended.

A variety of path indices have been proposed for native XML databases, including DataGuide [24], T-index [37], APEX [16], etc. In object-oriented databases too, path indices

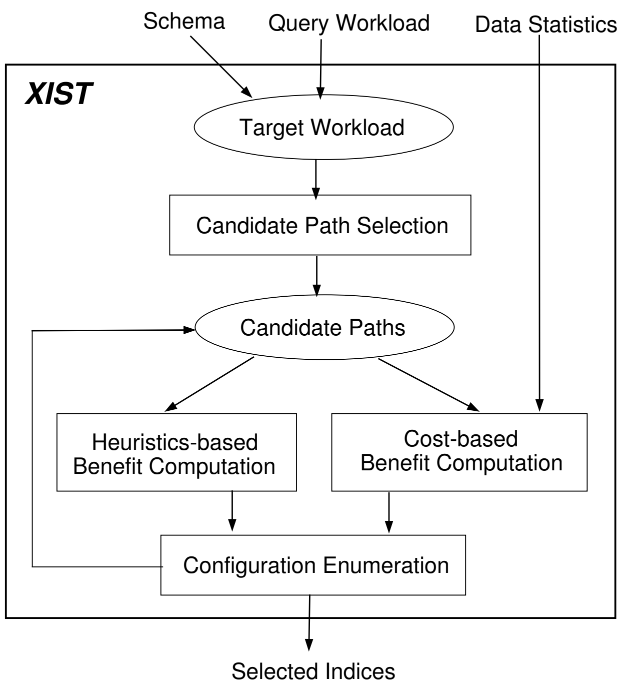


Figure 5.1: The XIST architecture

are used for efficient processing of queries [3]. However, the path index concept is not directly available in relational databases and therefore a mapping has to be established between the path indices and the defacto standard B-tree indices used in the relational world.

In this chapter, we describe the XIST tool in detail and the issues involved in mapping the XIST choices to the relational world, which are as follows: Firstly, a strategy to convert path indices to equivalent relational indices has to be designed (Section 5.2). Secondly, the disk space usage of the *relational indices* should be within the user-specified budget – therefore, an equivalence mapping between the disk budgets in the two spaces has to be identified (Section 5.3). Finally, the XQuery-to-SQL translation process should take advantage of the presence of the relational indices (Section 5.4).

5.1 XIST tool

Figure 5.1 shows an overview of XIST [42], which consists of four modules that adapt to a given set of input configuration. The first module is the *candidate path selection* module,

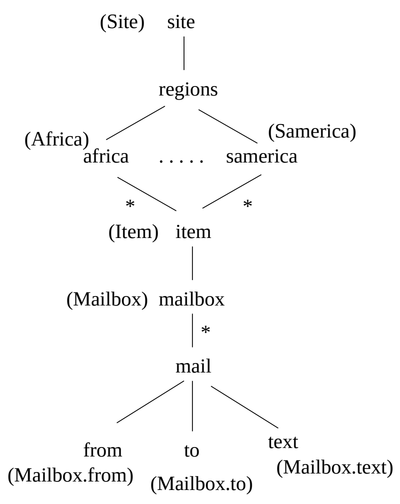


Figure 5.2: Example relational configuration

which eliminates a large number of potentially irrelevant path indices. It uses the following two techniques: (i) If the query workload is available, this module eliminates paths that are not in the query workload, and (ii) If the schema is available, the tool identifies and prunes equivalent paths that can be evaluated using a common index.

To compute the benefits of indices on candidate paths, XIST use either the *cost-based benefit computation* module or the *heuristic-based benefit computation* module, depending on the availability of data statistics. When data statistics are available, the cost based benefit computation module is employed. When data statistics are not available, the heuristic-based benefit computation module is operated instead.

The last module is *configuration enumeration*, which in each iteration chooses an index from the candidate index set that yields the maximum benefit. The configuration enumeration module continues selecting indices until a space constraint, such as a limit on the available disk space, is met.

5.2 Path Index to Relational Index conversion

We now discuss a strategy to convert path indices to equivalent relational indices. Consider an XML-to-relational mapping, as shown in Figure 5.2 for a fragment of XMark benchmark schema [53]. 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, and mailbox, respectively. For this environment, assume that the following path index, PI, has been recommended: /site/regions/africa/item/mailbox/mail/from. To evaluate this path index, the four relations $\{$ `Site`, `Africa`, `Item`, and `Mailbox`$\}$ have to be joined.

5.2.1 Naive approach for converting Path Index to Relational Index

An obvious translation process for converting Path Index to Relational Index, is to simply build the indices on the key and foreign-key pair for each parent-child involved in the path, namely are `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 that 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 created are `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 that need to be created for a path index is a function of the path length, and can therefore become very expensive to maintain.

5.2.2 Approach based on concept of *equivalence classes*

We propose an alternative and less expensive approach, which use the concept of *equivalence classes* [42] to reduce the number of relational indices. To explain this approach we first explain the concept of equivalence classes of XML schema, followed by detailed algorithm.

Algorithm 3 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 $e_1/e_2/\dots/e_n$ = $xPI$;
 2: $split\_paths$ = SplitPath($xPI$, $EQ$); // Split path index according to equivalence groups
 3: $rI$ = $\{\}$; $rT'$ = $rT$;
 4: let $\{c_1,c_2,\dots,c_m\}$ be the EQ groups for split_paths $\{p_1,p_2,\dots,p_m\}$
 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 ($rT'$, $rI$)
```

Equivalence classes

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 [42].

For example, the paths site/regions/africa, regions/africa, and africa belong to a common equivalence class. However, the paths africa/item and item are not in the same equivalence class because the path africa/item matches with the items that 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.

Detailed Algorithm

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 3).

Algorithm 4 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 ($split\_paths$)
```

In this algorithm, the first step is to split the path index such that each sub path corresponds to different equivalence classes (line 2). For details of SplitPath refer to Algorithm 4. 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 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).

To make the above algorithm concrete, 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 naive approach).

5.3 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 [22]. 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

$$S_I = \frac{N}{P_k * average\_page\_occupancy\_factor} * S_{page} \tag{5.1}$$

Note that, here we assume that $average\_page\_occupancy\_factor$ is 0.69, which is the typical fill factor for B+ tree index [54].

5.4 Query Rewriting for Path Indices

Recent work [34] has discussed use of integrity constraints information to guide XQuery-to-SQL query translation. In this section, we focus on the use of available path indices to guide XQuery-to-SQL query translation. XQuery-to-SQL translation, which is aware of the available path indices, can derive a more efficient rewriting of the query.

The procedure for achieving this conversion is described in Algorithm 5. Here, the first step is to identify all paths in the schema that satisfy the query (line 2). For each path, split the path such that each sub path is either an element or is a path corresponding to the available path index, and then compute the path equivalence classes for each sub path. This can be achieved by first splitting the path using the SplitPath function of Algorithm 4 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 to the EQ classes (line 16), and the final query is the union of all these queries (line 23).

Consider the example query

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

Algorithm 5 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 ($W\_SQL$)
```

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 as follows:

```
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, then 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’
```

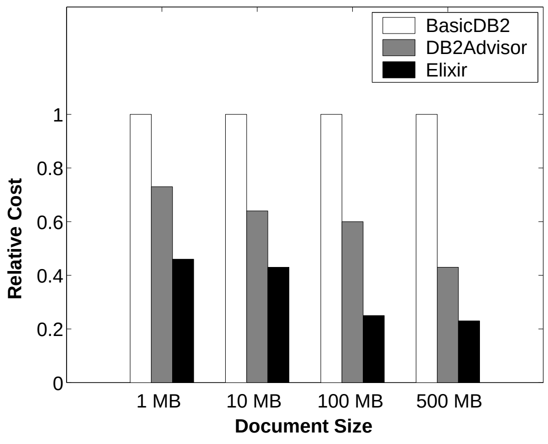
5.5 Experimental evaluation

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 which is on similar lines to Microsoft researchers work in [10].

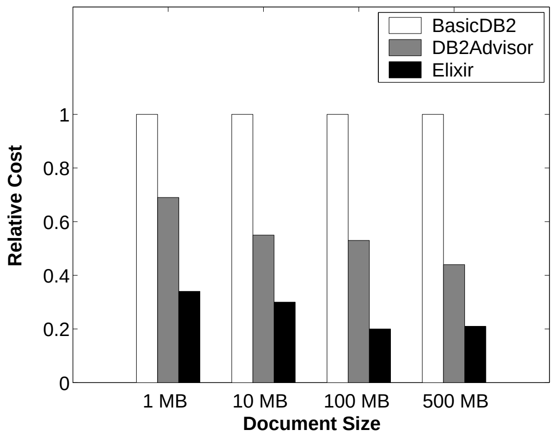
We report here the results of experiments, on two real world schemas: EPML [20], ICRFS [26] and a synthetic XMark benchmark schema [53] with various sizes of XML documents ranging from 1 MB to 500 MB. The query workload involves 20 queries (with identical frequencies).

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 5.3, 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 for other schemas were also similar.

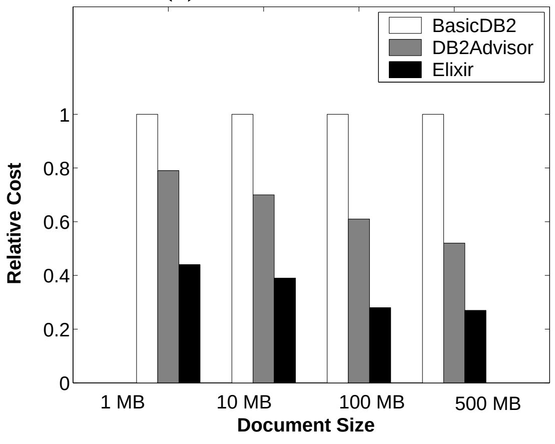
Analysis of the set of indices chosen by Elixir and *DB2Advisor* is summarized as follows: The SQL workload equivalent to the given XQuery workload involves several joins. DB2 attempts to improve the query performance by creating multicolumn indexes or single column indexes with include clause. Elixir,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



(a) For EPML schema



(b) For ICRFS schema



(c) For XMark schema

Figure 5.3: Impact of Index selection with space constraint

Chapter 6

Mapping of XML Triggers and XML Views

Having described core components of holistic schema, we move on to specialized components like XML triggers and XML views. In this chapter, we describe how XML triggers and XML views are mapped in Elixir.

6.1 XML Triggers

In order to make XML repositories fully equipped with data management capabilities, suitable query and update languages are being developed. However, once the user is allowed to perform updates, it is perceivably necessary to guarantee the correctness of his/her updates, especially if document validity or semantic constraints are violated [6]. This problem can be addressed by exploiting the well-grounded concept of active rules. In [6], authors have proposed *Active XQuery*, which is an active extension to W3C proposed standard XQuery [4] language for defining XQuery triggers. As XQuery triggers (XML triggers) are not part of standard , we have used the extension of XQuery for defining triggers, which is proposed in [6]. In this section, we discuss how XML triggers are mapped by Elixir.

6.1.1 Mappable XML Triggers and Non-mappable XML Triggers

Our main goal of handling XML triggers in Elixir is to map the XML triggers to SQL triggers. Compared to relational updates XQuery updates can be seen as bulk statements since they may involve arbitrarily large fragments of documents that are inserted or dropped by means of single statement. For example consider a fragment of Bank schema (refer to Figure 4.2) and corresponding relational mapping. When bank sets up its operation in new city, the application performs insert query on the XPath /bank/country/city, which is as follows:

```
FOR $country IN /bank/country
WHERE $country/name/text() = "INDIA"
UPDATE $country/city {
INSERT

     <name>Nasik</name>
     <state>Maharashtra</state>
     <head-office>...</head-office>
     <branch-office>...</branch-office> ...
     <atm>...</atm> ...
     <account>
       <savings-acc-number>201</savings-acc-number>
       <balance>1232423</balance>
     </account> ...
}
```

Above insert XQuery results into several SQL insert queries that are as follows:

```
INSERT INTO City (1000,’Nasik’,’Maharashtra’, ...)
INSERT INTO Branch-office (...)
...
INSERT INTO Office-Id (...)
...
INSERT INTO Atm (...)
...
INSERT INTO Account (...)
...
```

Consider an XML trigger, which sends e-mail to all customers who are from the same country as that of inserted city, for giving information about newly opened offices and ATMs.

```
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-head-office-address = NEW_NODE/head-office-address
    LET $city-branch-offices = NEW_NODE/branch-office
    ...
    FOR $customer IN NEW_NODE/../country/customer
      send-email ($customer, $city-name, $city-state, $city-head-off
            $city-head-office-address, $city-branch-offices,...)
)
```

The above trigger needs to be executed after all the insert statements to relations `City`, `Branch-office`, `Office-Id`, `Atm`, `Account` are executed. However, in relational SQL triggers, we cannot specify triggering operation as a set of operations on different tables. Clearly, such XML triggers are not mappable to relational triggers. We refer to such triggers as *non-mappable XML triggers*. If we can define the SQL trigger, which has same semantics as that of XML trigger, then such XML triggers are called as *mappable XML triggers*. Elixir maps *mappable XML triggers* to SQL triggers and *non-mappable XML triggers* to stored procedure, which can be called by middleware at runtime. Cost of SQL triggers that are invoked by the query are taken into account by relational optimizer. To model cost of the *non-mappable triggers*, Elixir adds query workload equivalent to *non-mappable triggers* to input query workload.

6.1.2 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` 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.

6.1.3 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
```

6.1.4 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.

6.2 XML Views

The notion of views is essential in databases. It allows various users to see data from different viewpoints. Although XQuery [4] currently does not provide standard for defining XML views, we can easily extend it to include the definition of views [15] 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 TABLE` *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 TABLE` statement.

Elixir maps XML views to relational views by translating XQuery to equivalent SQL query and translates XQueries on the XML views to the SQL queries on relational views. Additionally, if user specifies the materialized XML view, in order to improve the performance of XQueries, then it is mapped to materialized relational views to improve the performance of equivalent SQL queries.

For example, to make the balance inquiry faster user can specify the materialized XML view as follows:

```
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>$customer/cust-id</id>
            <acc-number>$customer/acc-number</customer-acc-number>
            <balance>$customer/balance</balance>
        </customer-balance>
DATA INITIALLY IMMEDIATE
REFRESH IMMEDIATE
```

Its equivalent relational materialized view can be specified as follows:

```
CREATE TABLE customer_balance AS
(SELECT Customer.id, Customer.acc-number, Account.balance
 FROM   Customer, Account
 WHERE  Customer.acc-number = Account.Savings-or-checking-acc-number
DATA INITIALLY IMMEDIATE
REFRESH IMMEDIATE
```

(Above syntax is for defining materialized query tables in DB2.)

6.3 Experimental Evaluation

In this section, we present our experimental evaluation of benefit obtained, due to inclusion of XML Triggers and XML views, in the tuning process. We carry out the experiment on two representative real-world XML schemas: *EPML* [20], *ICRFS* [26] and a synthetic XMark benchmark schema [53].

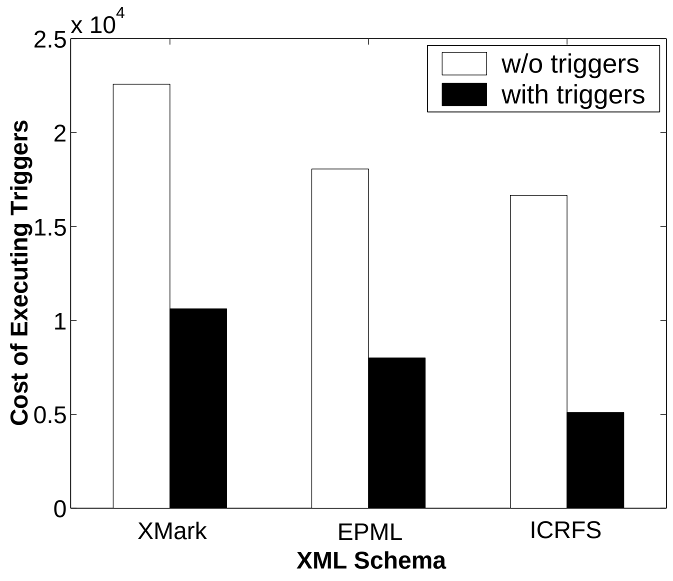


Figure 6.1: Effect of XML triggers on tuning process

6.3.1 Effect of XML Triggers

To assess the effect of XML triggers in tuning process, for each schema, we created input workload consisting of 10 read-only queries, 10 update queries, and 5 XML triggers. We carry out two sets of experiments. First, we do not consider XML trigger while tuning and calculate the cost of execution of triggers on the final relational configuration, referred as *w/o triggers*. Other set of experiments considers triggers while tuning, referred as *with triggers*. We compare the cost of execution of triggers on the final relational configuration. Our experimental results shows that cost for execution of triggers on the final relational configuration in *with triggers* is less than that in *w/o triggers* because in *with triggers* final relational configuration get tuned to triggered actions (refer to Figure 6.1).

6.3.2 Effect of XML Views

As mentioned earlier, Elixir maps materialized XML view specified by user to materialized relational views, to improve the performance of equivalent SQL queries. For each schema, we created 5 materialized views and 20 queries consisting of mix of view queries and non-view queries. Note that, here we have considered only materialized views, because virtual views does not affect cost of the query, and thus do not affect the tuning process. Our experimental results

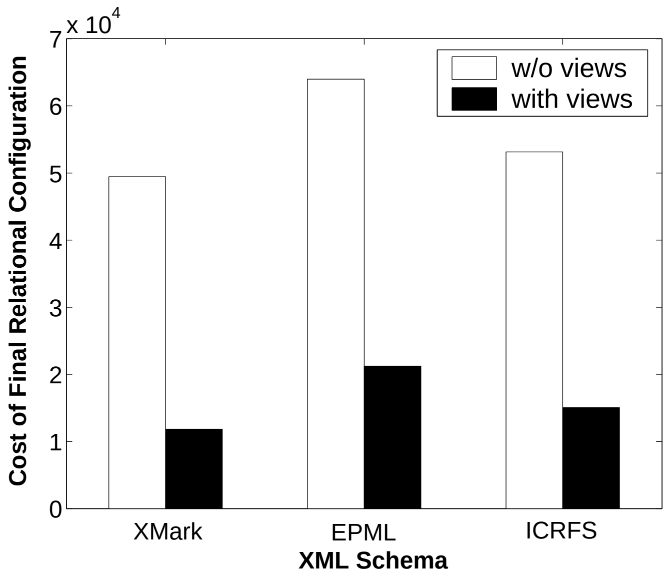


Figure 6.2: Effect of XML Views tuning process

shows that considering views in tuning process results in better final relational configuration (refer to Figure 6.2).

6.4 Overall performance of Elixir system

The previous discussion highlighted the various techniques used in Elixir and their impact on the final relational configuration. Summary of the techniques used in Elixir and their benefits is given in table 6.1. Please note that a holistic comparison of relational schema quality can not be done because the earlier mapping (FleXMap) can generate invalid schemas. Now, we present the results for the running time of Elixir i.e. user response time. User response time is the important metric for feasibility of the system. We report here the results of two representative real-world XML schemas: *EPML* [20], *ICRFS* [26] and a synthetic XMark benchmark schema [53]. Target workload consists of 20 *read only* queries and 10 update queries. For each schema, we created 5 XML triggers and 5 materialized views for frequent queries. Figure 6.3 shows the total time required and details about time required in different steps of tuning process – Mapping, Index selection, Optimizer. Mapping time is the time required to carry out the schema transformations. The number of schema transformations done in tuning process depends on the

Technique	Benefit
Key constraint propagation	Search space reduction : 30%-60% compared to FleXMap
	Runtime reduction : 50%-85% compared to FleXMap
	Better relational configurations
Source-centric index selection	Cost of final relational configuration is reduced by 35- 60% as compared to that in *DB2Advisor* (target-centric index selection)
Mapping XML triggers	Cost for execution of triggers on the final relational configuration, obtained by considering triggers, is reduced by 50-70% as compared to that when triggers are not considered during tuning.
Mapping of XML views	Considering views in tuning process results in 65-75% reduction of cost of final relational configuration as compared to the cost when views are ignored during tuning process.

Table 6.1: Summary of techniques used in Elixir

complexity of schema - such as number of repetitions, unions, similar types, nesting depth, etc. In our experiments, total numbers of transformations tried out were as follows: 637 (XMark), 505 (ICRFS), 1673 (EPML). We can see that time required for schema transformation (mapping) is less than 10% of the total time (refer to Figure 6.3). Each schema transformation corresponds to different relational configuration. For each relational configuration, indices are selected using XIST and are converted to relational indices. The time required by XIST, for index selection, ranges from 25% to 35%. Each relational configuration is evaluated using relational optimizer. The time required for evaluation ranges from 60% to 70%, which involves creation of tables in database, loading of statistics, getting cost from optimizer, and deletion of tables. As we have implemented the system outside the relational engine, we cannot obtain the cost without creating tables, loading statistics thus the time required for evaluation is more as compared to other operations. If Elixir system is implemented inside relational engine, then the time required for evaluation can be reduced.

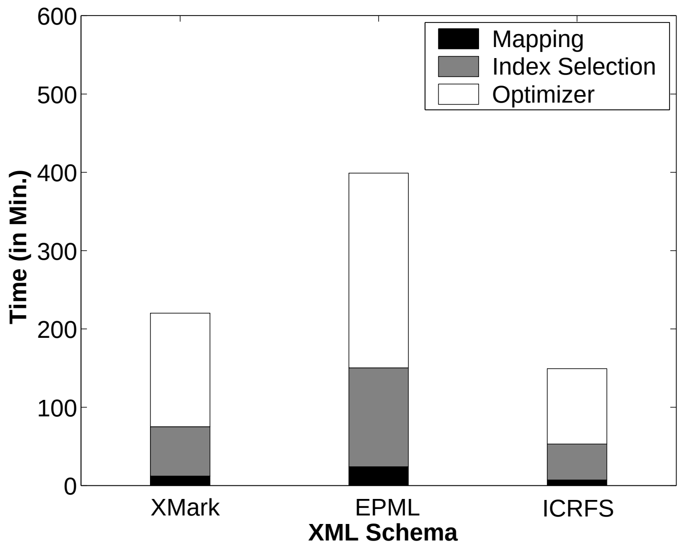


Figure 6.3: Elixir Performance

Chapter 7

Conclusions

In this thesis, we studied the problem of producing holistic schema mappings from XML repositories to relational backends. Our goal was to ensure both cost-based and source-centric optimization of the mapping process. To this end, we proposed the Elixir system, which delivers relational schemas that include table configurations, keys, indices, triggers, and views. The system incorporates techniques for propagating keys through cost-based dynamic mappings, as compared to the heuristic-based static mappings of the prior literature. Further, through a detailed experimental study on real-world and synthetic schemas, we showed that incorporation of keys substantially reduces the search space and runtime required for cost-based optimization, as compared to FleXMap.

With regard to indices, we presented techniques for efficiently mapping source-centric index choices (made by the XIST tool) to the relational target. Our experimental results comparing this approach to using the relational engine's index advisor indicate that better quality configurations can be achieved with a source-centric approach.

Apart from the above core components, a holistic mapping includes other features such as views, triggers, etc. Elixir incorporates the techniques for achieving their mappings to relational world. Empirical results shows the improvement of final relational configuration obtained due to consideration of XML Triggers and XML views in the tuning process of relational configuration.

In a nutshell, the Elixir system attempts to make progress towards achieving "industrial-strength” mappings for XML-on-RDBMS.

7.1 Future Work

The work that we have presented in this thesis can be extended in the following ways:

- Elixir uses various schema transformations for exploring the search space of relational configurations. These transformations include *Inline/Outline*, *Type-split/merge*, *Union distribution/factorization*, and *Repetition split/merge*. These transformations only exploit the structural relationship between various elements.

  Some queries involve join of two different XPaths on the nodes that are semantically related (we refer these queries as *semantic join queries*). Merging of such semantically related nodes i.e. mapping those nodes along with related information in single relation will improve the performance of *semantic join queries*. Information about such semantically related nodes can be obtained from the integrity constraints defined in XML schema. In future, we plan to study the transformations that use the semantic information from the XML schema.

- As mentioned in Section 6.4, the run time performance of Elixir can be improved, if implemented inside relational engine. In future, we are planning to implement Elixir system inside *postgreSQL*, an open source platform.

- Currently, Elixir maps the materialized views provided by user. In future, we plan to incorporate the source-centric technique for recommendation of materialized views.

References

[1] S. Abiteboul. On Views and XML. In *Proc. of 18th ACM Symp. on Principles of Database Systems (PODS)*, May 1999.

[2] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A Framework for Using Materialized XPath Views in XML Query Processing. In *Proc. of 30th Intl. Conf. on Very Large Data Bases (VLDB)*, August 2004.

[3] E. Bertino and C. Guglielmina. Path-Index: An Approach to the Efficient Execution of Object-Oriented Queries. *Data and Knowledge Engineering*, 10, 1993.

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

[5] 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.

[6] A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active XQuery. In *Proc. of 18th IEEE Intl. Conf. on Data Engineering (ICDE)*, February 2002.

[7] J. Bosak and et al. W3C XML Specification DTD. *http://www.w3.org/XML/1998/06/xmlspec-report*.

[8] T. Bray and et al. DCD (Document Content Description). *http://www.w3.org/TR/NOTE-dcd*.

[9] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. *Computer Networks*, 39(5), 2002.

[10] 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.

[11] S. Chaudhuri and V. Narasayya. An Efficient, Cost–Driven Index Selection Tool for Microsoft SQL Server. In *Proc. of 23rd Intl. Conf. on Very Large Data Bases (VLDB)*, September 1997.

[12] Y. Chen, S. Davidson, C. Hara, and Y. Zheng. RRXS: Redundancy reducing XML storage in relations. In *Proc. of 29th Intl. Conf. on Very Large Data Bases (VLDB)*, September 2003.

[13] 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.

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

[15] Y. Chen, T. Ling, and M. Lee. Designing Valid XML Views. In *Proc. of 21st Intl. Conf. on Conceptual Modeling (ER)*, October 2002.

[16] C. Chung, J. Min, and K. Shim. APEX: An Adaptive Path Index for XML Data. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 2002.

[17] J. Clark and et al. XML Path Language (XPath) Specification. W3C Recommendation, 16 November 1999. *http://www.w3.org/TR/xquery/*.

[18] S. Cluet, P. Veltri, and D. Vodislav. Views in a Large Scale XML Repository. In *Proc. of 27th Intl. Conf. on Very Large Data Bases (VLDB)*, September 2001.

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

[20] EPC Markup Language. *http://wi.wu-wien.ac.at/ mendling/EPML/*.

[21] D. Florescu and D. Kossman. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), 1999.

[22] 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.

[23] Gene Expression Markup Language. *http://www.ncgr.org/genex*.

[24] R. Goldman and J. Widom. Dataguides: Enabling Query Formulation and optimization in semistructured databases. In *Proc. of 23rd Intl. Conf. on Very Large Data Bases (VLDB)*, August 1997.

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

[26] ICRFS XML schema. *http://www.insureware.com/abouti/ mlines.shtml*.

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

[28] H. Jiang, H. Lu, W. Wang, and J. Yu. XParent: An Efficient RDBMS-Based XML Database System. In *Proc. of 18th IEEE Intl. Conf. on Data Engineering (ICDE)*, March 2002.

[29] C. Kanne and G. Moerkotte. Efficient Storage of XML data. In *Proc. of 16th IEEE Intl. Conf. on Data Engineering (ICDE)*, February 2000.

[30] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. Covering Indexes for Branching Path Expressions. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 2002.

[31] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Efficient Indexing of Paths in Graph Structured Data. In *Proc. of 18th IEEE Intl. Conf. on Data Engineering (ICDE)*, February 2002.

[32] M. Klettke and H. Meyer. XML and object-relational database systems - enhancing structural mappings based on statistics. In *Proc. of 3rd Intl. Workshop on Web and Databases (WebDB)*, May 2000.

[33] 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.

[34] 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.

[35] A. Layman and et al. XML-Data. *http://www.w3.org/TR/1998/NOTE-XML-data*.

[36] D. Lee and W. Chu. Constraints–preserving Transformation from XML Document Type Definition to Relational Schema. In *Proc. of 19th Intl. Conf. on Conceptual Modeling (ER)*, October 2000.

[37] T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proc. of 7th Intl. Conf. on Database Theory (ICDT)*, January 1999.

[38] Oracle XML DB: An oracle technical white paper. *http://technet.oracle.com/tech/xml/content.html*.

[39] POET. *http://www.x-solutions.poet.com/*.

[40] 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.

[41] K. Runapongsa and J. Patel. Storing and querying XML data in object-relational DBMSs. In *Proc. of 7th Intl. Conf. on Extending Database Technology (EDBT)*, March 2002.

[42] 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.

[43] Schematron: An XML Structure Validation Language using Patterns in Trees. *http://xml.ascc.net/resource/schematron/schematron.html*.

[44] A. Schmidt, M. Kersten, M. Wendhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *Proc. of 3rd Intl. Workshop on Web and Databases (WebDB)*, May 2000.

[45] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of 25th Intl. Conf. on Very Large Data Bases (VLDB)*, September 1999.

[46] F. Shao, A. Novak, and J. Shanmugasundaram. Triggers over XML Views of Relational Data. In *Proc. of 21st IEEE Intl. Conf. on Data Engineering (ICDE)*, April 2005.

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

[48] I. Tatarinov, Z. Ives, A. Halevy, and S. Weld. Updating XML. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 2001.

[49] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML schema Part 1: Structures, May 2001. *http://www.w3.org/TR/xmlschema-1/*.

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

[51] ToXgene - the ToX XML Data Generator. *http://www.cs.toronto.edu/tox/toxgene/*.

[52] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend its Own Indexes. In *Proc. of 16th IEEE Intl. Conf. on Data Engineering (ICDE)*, February 2000.

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

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

[55] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions On Internet Technology (TOIT)*, 1(1), 2001.