Optimizing XML Path Queries over Relational Databases

by

Maria Cecilia Pilar

A thesis submitted in conformity with the requirement
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto
2002

ABSTRACT

Optimizing XML Path Queries over Relational Databases

by

Maria Cecilia Pilar

Master of Science

Graduate Department of Computer Science

University of Toronto

2002

The amount of data available on the Internet grows rapidly, and data becomes semistructured. The *Extensible Markup Language* (XML) has become a very important standard for data representation and exchange over the Internet. Therefore, a mechanism for managing XML documents is needed. A strong candidate is relational databases, for which management issues are solved. However, relational technology fails to deliver good performance for regular path queries, which are the distinctive features of XML query languages.

This thesis was developed within the ToX project. In this thesis, we present the ToX Relational mapping scheme which corresponds to the ToX relational backend. Along with the mapping itself, we propose a set of optimizations to efficiently evaluate regular path queries. We explore the use of an encoding for determining containment relationships, and the use of materialized views as a query optimization tool. We accompany the exposition with an experimental evaluation.

For Christian

ACKNOWLEDGEMENTS

Pursuing graduate studies has been a wonderful and enriching experience. I would like to express my gratitude to all the people that contributed to the successful completion of this work.

I would like to thank my supervisor, Professor Alberto Mendelzon, for his continued guidance and support during the preparation of this thesis. He honored me with his insights and valuable comments. I am grateful to Professor Ken Sevcik for being the second reader, and for his constructive suggestions. I also acknowledge the generous financial support I received from the Department of Computer Science, the Toronto Open Fellowship and my supervisor.

My thanks to all my friends in Toronto for being like my family here in Canada. To Flavio for revealing us the beauties of Toronto. Mariela and Patricia for their in-conditional support and friendship. Carlos, Danny, Diego, Lily, Mariana, Gustavo, Fernando, thank you all!

Words cannot express my deepest gratitude and appreciation to my family. My mother Emma and my father Mario for their emotional support and love. My sisters, Mariana and Silvia, for always being there, although they are far away. My grandfather Osvaldo, Nelida and Diva for showing me how to enjoy a healthy and wonderful life.

Finally, I want to express all my love to Christian, my husband and best friend. His permanent encouragement and support gives me the confidence I need. He made me believe I was capable of succeed on this dream. Without Christian this thesis would not have been written. All my love to him.

TABLE OF CONTENTS

DEDICATION . . . . **iv**

ACKNOWLEDGEMENTS . . . . **v**

LIST OF TABLES . . . . **xi**

LIST OF FIGURES . . . . **xiv**

CHAPTERS

I. Introduction 1

1.1 Introduction 1

1.2 The Toronto XML Engine - ToX 8

1.2.1 Overview 8

1.2.2 Architecture 9

1.3 Motivation 11

1.4 Thesis Outline 14

II. Related Work 16

2.1 Mapping XML Data to Databases 16

2.2 Optimizing Regular Path Queries 23

2.3 Answering Queries using Materialized Views 25

2.4 XML Data Repositories 27

III. ToX Relational Mapping Scheme 29

3.1 Introduction 29

3.2 ToX Relational Mapping Scheme 31

3.2.1 Overview 31

3.2.2 Comparing the ToX Relational Mapping Scheme with Other Approaches 34

3.3 Mapping an XML Document into the ToX Relational Mapping Scheme 35

3.4 Querying the ToX Relational Mapping Scheme 39

3.5 Implementation 43

3.5.1 Architecture 46

3.5.2 Mapping 47

3.5.3 Relational Document Navigator 48

3.6 Extending the ToX Relational Mapping Scheme Using Ancestor-Descendant Information 50

3.6.1 Determining Ancestor-Descendant Relationship 51

3.6.2 Adding the Encoding to the ToX Relational Mapping Scheme 53

3.6.3 Exploiting the Encoding in the Relational Document Navigator 54

IV. Optimizing Regular Path Queries Using Materialized Views 60

4.1 Materialized Views 61

4.1.1 Overview 61

4.1.2 Determining What Views to Materialize 61

4.1.3 Deriving Materialized View Definitions 63

4.1.4 Populating Materialized Views 67

4.2 Managing Materialized Views in the toxRelational Component . . . 68

4.2.1 Limitations 71

4.3 Exploiting Materialized Views within the Relational Document Navigator 73

4.3.1 Deriving a Rewriting 74

4.3.2 Implementing the Rewriting 76

V. Experiments 83

5.1 Experimental Setup 83

5.1.1 Experimental Platform 83

5.1.2 DBMS 84

5.1.3 Data Set Description 85

5.1.4 Experimental Procedure 86

5.2 Bulkloading Performance Evaluation 87
5.2.1 Loading Times 88
5.2.2 Database Size 90
5.3 Individual Queries 90
5.3.1 EBOC Data Set 91
5.3.2 DBLP Data Set 100
5.4 Query Workload 107

VI. Conclusions and Future Work 110

6.1 Conclusions 110
6.2 Future Work 111

APPENDICES

A. ToX Document Navigator 115

B. From the Document Navigator to SQL Queries 120

C. ToXgene Template 128

BIBLIOGRAPHY **132**

LIST OF TABLES

Table

3.1 *Edge* Relation Structure 32

3.2 *Leaf* Relation Structure 33

3.3 *Attribute* Relation Structure 34

3.4 Some Navigational Methods of the Document Navigator 41

3.5 *Document Catalog* Relation Structure 48

4.1 An Example of a Workload for the DBLP Database 63

5.1 Data Sets Characteristics 86

5.2 ToX Relational Mapping Information about the DBLP Data Set 90

5.3 Performance Results of the Materialization Procedure for the EBOC Data Set (ms) 94

5.4 EBOC Queries Resolution Strategy 95

5.5 Performance Results of the Queries over the EBOC Data Set (ms) - First Execution . . . 95

5.6 Performance Results of the Queries over the EBOC Data Set (ms) - Average Subsequent Executions . . . 96

5.7 Performance Results of the Materialization Procedure for the DBLP Data Set (ms) . . . 102

5.8 DBLP Queries Resolution Strategy . . . 103

5.9 Performance Results of the Queries over the DBLP Data Set (ms)- First Execution . . . 103

5.10 Performance Results of the Queries over the DBLP Data Set (ms) - Average Subsequent Execution . . . 103

A.1 ToX Document Navigator XDocument Interface . . . 116

A.2 ToX Document Navigator XNode Interface . . . 117

A.3 ToX Document Navigator XElement Interface . . . 118

A.4 ToX Document Navigator XAttribute Interface . . . 118

B.1 SQL Queries Related to the Creation of an XDocument Object . . . 121

B.2 SQL Queries Related to Child Element Information . . . 123

B.3 SQL Queries Related to Parent Information 124

B.4 SQL Queries Related to Sibling Element Information 124

B.5 SQL Query Related to Descendants Element Information 126

B.6 SQL Queries Related to Attributes Information 127

LIST OF FIGURES

<u>**Figure**</u>

1.1 XML Document for the DBLP Database 3

1.2 XML Graph Representation of the DBLP Database 4

1.3 ToX Architecture 10

1.4 Query using XQuery Core Specification 12

1.5 Query using Extended XQuery Core 13

3.1 ToX Relational Mapping Algorithm 36

3.2 Extended XML Graph for the DBLP database 37

3.3 Instances of *Edge*, *Leaf*, and *Attribute* Relations plus an index over the *Edge* Relation 38

3.4 Document Navigator *getDescendants(Regular Expression)* Algorithm . . . 44

3.5 Instance of the DBLP Database 45

3.6 Query Graph Associated with Q 45

3.7 The toxRelational Component Architecture 46

3.8 From the Document Navigator to SQL 49

3.9 Encoding Example 53

3.10 New ToX Relational Mapping Algorithm 55

3.11 Mapping the XML document in Figure 1.1 into the Extended ToX Relational Mapping Scheme 56

3.12 Encoding-based Processing Algorithm of Descendant Queries 57

4.1 Materialized View Content Example 64

4.2 Heuristic Rules - Example Graphical Representation 67

4.3 Entity Relationship Diagram of the ToX Relational Catalog 70

4.4 Algorithm for Deriving a Maximal Rewriting 75

4.5 Algorithm to Check Whether a Rewriting is Exact or Maximal 76

4.6 Algorithm for Exploiting Materialized Views 78

4.7 Automaton Representation for Query Q_1 81

4.8 Automaton Representation for Query Q_4 81

4.9 Example Extracted From the DBLP Database 81

5.1 Influence of the Database Caching Procedure in the Query Processing Time 87

5.2 Comparison between the Different Loading Methods 89

5.3 EBOC Data Set DTD 92

5.4 Summary Performance Results for the EBOC Data Set - First Execution . 99

5.5 Summary Performance Results for the EBOC Data Set - Average Subsequent Repetitions 100

5.6 DBLP Data Set DTD 101

5.7 Summary Performance Results for the DBLP Data Set - First Execution . 106

5.8 Summary Performance Results for the DBLP Data Set - Average Subsequent Executions 106

5.9 Summary Performance Varying DBLP Data Set Size 107

5.10 Summary Performance Varying DBLP Data Set Size 108

5.11 Summary Performance Varying XMark-like Data Set Fanout 109

CHAPTER I

Introduction

In this chapter we motivate this work, giving an introduction to the context in which it was developed, and some information about the system in which this work is embedded. Finally we present an outline of the rest of the thesis.

1.1 Introduction

As the amount of data available on the Internet grows rapidly, more and more of the data becomes semistructured and hierarchical. The data has no absolute schema fixed in advance, and its structure is irregular or incomplete. Semistructured data arises when the source does not impose a rigid structure, and when the data is obtained by combining several heterogeneous data sources during the process of data integration. Semistructured data is often called schemaless or self-describing, because there is no separate description of its type or structure and its data.

The *Extensible Markup Language* (XML), as a format for semistructured data, has become a very important standard for the representation and exchange of data over the Internet. Using XML as a data representation standard, it is possible to represent not only the data itself, but also its semantics. XML was designed specifically to describe content, rather than presentation. It is a textual representation of data. In Figure 1.1 a segment of the Digital Bibliography and Library Project (DBLP) database [Ley01] is encoded in XML. One basic component of an XML document is the *element*, which is a piece of text bounded by matching tags such as <*conferences*> in the example. Inside an element, we may have "raw" text, other elements, or a mixture of the two. Tags must be balanced, and they are not predefined. *Attributes* are another important component of XML documents. They are defined as (*name, value*) pairs, and represent unique "properties" of an element. In our example, several *key* attributes exist. Therefore, combining elements, attributes and exploiting the hierarchical nature of its data, XML is able to capture some of the semantics of the data.

Graph Representation of XML

An XML Graph is an edge-labelled representation of an XML document. It is derived from the *de facto* data model for semistructured data: the *Object Exchange Model* (OEM) [PGMW95]. The model was extended with the notion of order to deal with the ordered collections managed by XML. The XML graph is a graph with an initial node: the root of the XML document. The graph shows containment relationships between elements, attributes, and text values. The names of these entities are the labels of the edges in the XML graph. When the XML document does not contain ID and IDREFS, then the XML Graph is, in fact, a tree. Several modifications and extensions to the data model have been proposed [Abi97, Suc, Bun97]. In Figure 1.2, we show the XML graph associated with the prior example (Figure 1.1).

```
<conferences>
    <conference key="LICS">
        <issues>
            <issue key="2001">
                <inproceedings key="AlonMNSV01">
                    <author>Noga Alon</author>
                    <author>Tova Milo</author>
                    <author>et al.</author>
                    <title> Typechecking XML Views of
                            Relational Databases.</title>
                    <pages>421-430</pages>
                    <year>2001</year>
                    <booktitle>LICS</booktitle>
                </inproceedings>
            </issue>
        </issues>
    </conference>
    <conference key="SIGMOD">
        <issues>
            <issue key="2001">
            ...
            </issue>
        </issues>
    </conference>
    ...
</conferences>
```

Figure 1.1: XML Document for the DBLP Database

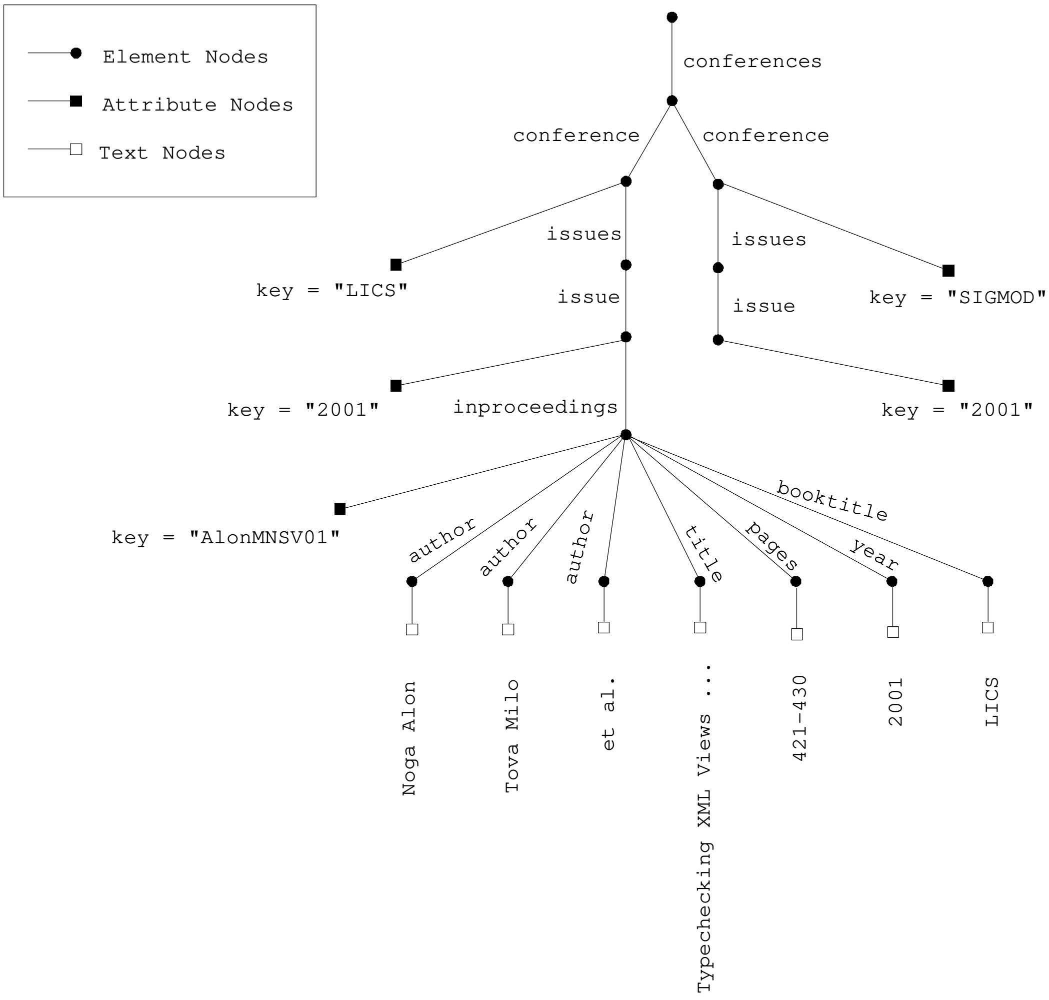

Figure 1.2: XML Graph Representation of the DBLP Database

Typing XML Documents

XML is an attractive data representation standard because it offers a simple, intuitive, and uniform text-based syntax. Also, it is extensible, which means that new structure can be added by creating and nesting new tags. Because of these characteristics, the XML representation provides unlimited potential in representing any kind of data. This unlimited potential is one of XML's most important strengths, but it turns out that it is too difficult for applications to manage it. Therefore, some structural constraints are needed to bound the unlimited XML data representation. In order to specify and enforce XML structure, *Document Type Definitions* (DTDs) have been used [DTD]. A DTD is a context-free grammar that specifies the potential structure of an XML document. It can be used, to a certain extent, as a schema, but DTDs have some limitations in managing type information (e.g., there does not exist the notion of atomic types, unique type per element name, etc.). Due to these limitations, DTDs did not cope with the expectations of being a schema language, and therefore, new schema formalisms have been proposed [DCD, XML]. Among them, *XML Schema* [XML] has become a recommendation of the World Wide Web Consortium as an XML schema language.

Querying XML Documents

The distinguishing feature of query languages for semistructured data, and consequently for XML, is their ability to navigate semistructured nested data. To achieve path navigation, semistructured query languages exploit the notion of regular path expressions. A regular path expression is a sequence of edge labels $l_1, l_2, \cdots, l_n$ combined with operators. The *concatenation* or *conjunction*, *alternative* or *disjunction* and the *descendants-or-self* operator are common in regular path expressions. An alternative step, using the *alternative* operator "|" establishes alternative labels for a step. For instance, a regular path expression for the DBLP database with an alternative step could be Q_1 = */conferences/conference/issues/issue/(proceedings | inproceedings)*. In Q_1, the regular path expression retrieves both *proceedings* and *inproceedings* elements satisfying the first part of the regular expression. The *descendants-or-self* operator is a combination of the "any" and the Kleene closure operator. Using the *descendants-or-self* operator "//", a regular path expression can specify a label that can be found at any level of nesting of the XML hierarchy. An example of this operator can be Q_2 = */conferences//inproceedings*. Query Q_2 searches for all *inproceedings* elements, at any level of the hierarchy, that are descendants of a *conferences* element. The result of applying a regular path expression to an XML document is a set of nodes satisfying it. But it is desirable that the result of any semistructured query returns, also, a piece of semistructured data. Therefore, any semistructured query language has to manage, apart from regular path expressions, element constructors, joins between XML documents, schema information extraction, etc. Among all the proposed query languages the more popular ones are XQuery [xqua], Quilt [CRF00], XML-QL [DFF+99], UnQL [BDHS96], XQL [XQL], and Lorel [AQM+97].

Storing XML Documents

There are several approaches to store XML data. The first and most obvious approach is to store XML documents as plain files in the file system. Although this approach is straightforward, it is the most inefficient in terms of query performance. To make this approach feasible, some index structures have been proposed [FLS98, RM01]. Another strategy for storing XML documents is to use special-purpose databases. Example research prototypes are Lore [MAG+97], Strudel [FFK+98], and ToX [BBM+01]. The use of special-purpose databases is attractive because they can capture all the distinguishing features of semistructured data. However, some time will pass until special-purpose databases can reach a mature point in development, so that they can hold efficiently large amounts of data. Other storing approaches use object-oriented, relational, or object-relational databases. Among them, Relational Database Management Systems (RDBMS) are mature enough to manage and to evaluate queries over structured data efficiently. Consistency, concurrency and recovery issues are already solved in the relational world. So far, all these features are the most convincing arguments to extend the use of relational databases to manage XML data. Although it is known that the requirements for processing XML are vastly different from those for processing structured data, a lot of effort has been put into exploiting the relational potential.

This thesis was developed within the framework of the Toronto XML Engine (ToX) [BBM+01]. Its main objective is to store XML documents into relational databases, proposing optimization strategies to improve regular path query performance. The main contributions are:

- We have defined the ToX Relational mapping scheme, which is an XML to relational mapping.
- We have extended the ToX Relational mapping scheme with an encoding to determine ancestor-descendant relationships in constant time.
- We have proposed an optimization using materialized views to speed up regular path queries. We have defined workload information from which the structures of the materialized views are derived, and we have studied a strategy to exploit the materialization during the query answering procedure.
- We have implemented all the code necessary to test the ToX Relational mapping scheme, and the proposed optimizations.
- We have run experiments showing different aspects of the techniques developed

in this thesis.

1.2 The Toronto XML Engine - ToX

1.2.1 Overview

ToX is a repository for XML data and metadata [BBM+01]. It supports heterogeneous data storage and indexing. ToX provides a mechanism to develop new XML technologies and services, and also can be viewed as a data integration environment.

This thesis focuses on exploiting a relational database as a backend for ToX. The module is known as the toxRelational component. It stores the contents of the XML documents into a relational database. One of the concerns of the toxRelational component is the performance of the query answering procedure. To optimize this process, the toxRelational component exploits traditional RDBMS indexing mechanisms plus a number of other optimizations.

As we mentioned, ToX supports heterogeneous data storage mechanisms. This characteristic is totally invisible to end users. The multiplicity of backends allows ToX to improve performance in XML document management. When a new XML document in registered in ToX, one of its components, the *Catalog Manager*, determines the best storage mechanism and indexing strategy based on the *Document Structuredness* and a workload. The *Document Structuredness* is a fuzzy measure that characterizes a document within the spectrum of textual (e.g., a book) and data (e.g., a catalog of books) documents. These categories of documents not only differ in their characteristics, but also in the way they are queried. In the current implementation, the formalization and definition of the Document Structuredness metric is still under active research.

The ToX *Query Processor* supports XQuery Core [XQub] as the query language to access the data stored in the repository. The XQuery Core specification was implemented and extended within the framework of ToX [Rog02]. Some characteristics of the ToX query processor are given in the next section as an introduction to the motivation for this thesis.

In order to transparently deal with multiple storage strategies, a unique access mechanism was specified and developed within the toxRelational framework. This unique access mechanism conforms to an application programming interface (API) that hides the implementation details of the backends. The API defines the way an XML document is accessed and queried. Basically, it supports the navigation of an XML document stored in the repository for answering a query, regardless the physical details of the underlying backend. For this reason, the API is known as the Navigational API.

1.2.2 Architecture

ToX architecture can be seen in Figure 1.3. In the architecture representation, we can see not only all modules, but also their interactions.

The only component visible to the user is the *ToX Manager* module. Through it, the user can interact with the repository. When a new XML document is registered, the *ToX Catalog Manager*, in conjunction with the *Storage Optimizer*, selects the best storage strategy. Ideally, ToX will use the Document Structuredness and a workload as basis for this selection. In the current implementation, users can force the use of a particular backend. If the XML Schema or DTD of the XML document is available at registration time, the instance documents are checked. The schema information is stored as metadata together with some statistical information about the documents. ToX manages the notion of *collections*. A collection is a set of XML

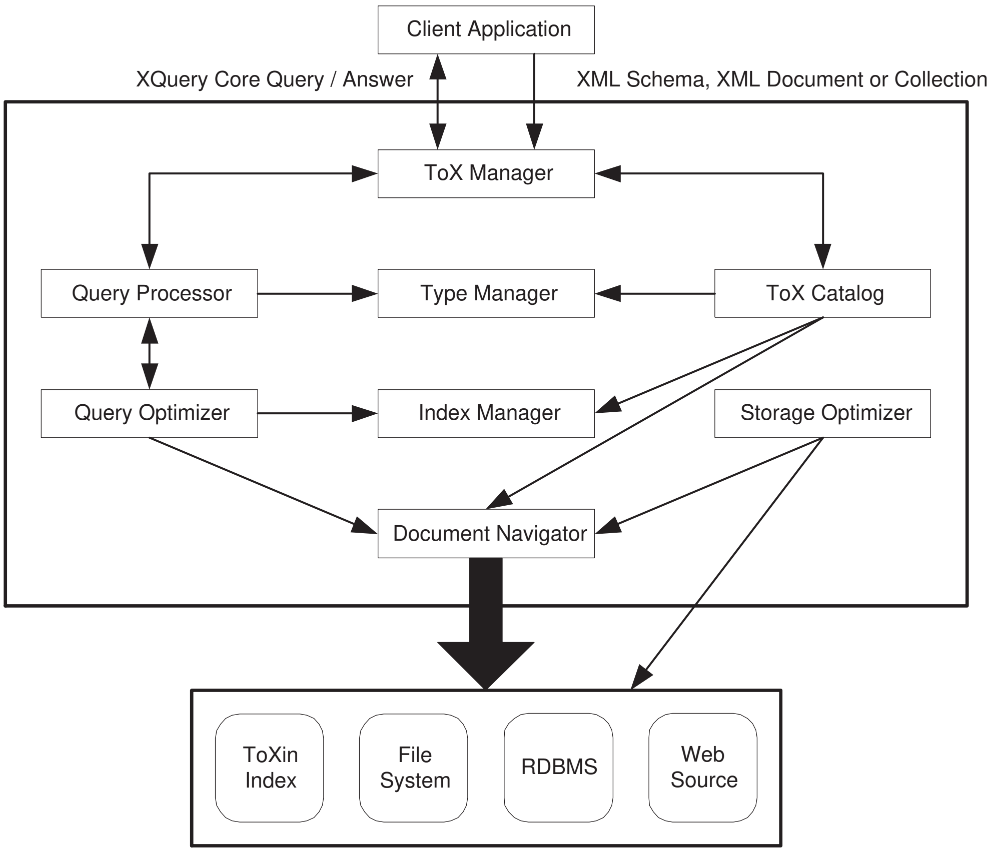

Figure 1.3: ToX Architecture

documents, possibly satisfying the same DTD or XML Schema. In the case that metadata is associated with a collection, then only documents conforming to the metadata can be stored into it. When a query is issued to the repository, various modules are involved in its answer. The *Query Processor* and the *Query Optimizer*, with information collected from the *Type Manager*, parse and optimize the query, submitting calls to the Navigational API. Internally, these calls are pushed to the corresponding backend from which the answer is retrieved. Indexing mechanisms (in the *Index Manager*) are exploited in order to accelerate the response time.

1.3 Motivation

Our intention is to use relational databases to store XML documents. To achieve our objective, we have to provide an efficient management mechanism for XML documents. Common management issues are already solved in the relational world, so we have to focus our attention on those issues where the relational technology fails to deliver good performance. To distinguish those issues it is necessary to explore the storing and querying mechanisms. This knowledge will help in the detection of inefficiencies to attack.

In order to motivate this thesis, we need to present some details about the ToX Query Processor and the Navigational API. XQuery Core is used as the query language of ToX. The original XQuery Core specification was implemented and extended within the framework of ToX. Recalling that regular path queries are the most distinctive feature of semistructured query languages, the XQuery Core implemented within ToX was extended to support them. The original specification allows XQuery Core to manage only simple path expressions composed of single steps. To support navigation of paths in the original XQuery Core, several *for* statements have to be combined with the *typeswitch* operator [XQub, section 6.2.1.3: Simple Navigation].

```
for $a in (/conferences) return

    for $b in ($a/conference) return
        <2001 Conferences>
        {
                for $c in (descendants-or-self($b)/inproceedings) return
                        if ($c/year = "2001") then
                                $c/title
                        else()
        }
        </2001 Conferences>
```

Figure 1.4: Query using XQuery Core Specification

To clarify this point we present an example. Suppose that we want to extract from the DBLP database presented in Figure 1.1 the following information: "List the title of proceedings of conferences during 2001". The answer to the query can be found by filtering by year the data satisfying the */conferences/conference//inproceedings* regular path. In Figure 1.4 we show the resultant XQuery Core query conforming to the original specification.

In the example, we can appreciate how the regular path expression */conferences/conference//inproceedings* was decomposed into several nested *for* statements. The first two statements descend through the first two levels of the hierarchy, while the third statement in combination with the *descendants-or-self* operator allows descending through all levels of the hierarchy. The filtering by year is performed by an *if then else* statement.

One goal of the ToX Query Processor is to simplify the query expressions as much as possible. Therefore, the XQuery Core implemented within ToX adds an extension in this respect. This extension adds a new valid expression to the

```
for $a in (/conferences/conference//inproceedings) return
<2001 Conferences>
{
   if ($a/year = 2001) then
      $a/title
   else()
}
</2001 Conferences>
```

Figure 1.5: Query using Extended XQuery Core

ones defined by the XQuery Core specification, and can be stated as follows: An abbreviated regular path expression without predicates is a valid expression within XQuery Core.

In Figure 1.5 we present the query of the previous example, but using the Extended XQuery Core representation. As can be seen, the query representation has been simplified.

With this extension we have reduced the complexity of the query notation, but we have added the necessity of managing regular path expressions at the Query Processor level. Two possibilities exist in dealing with regular path expressions: manage them at the Query Processor level without knowledge of the structure of the XML document, and the details of its storage mechanism; or push the regular path expression directly to the backends. The second alternative is the most advantageous because each backend can find an efficient way to manage this type of expression. Exploiting this advantage, the ToX Query Processor delegates to each backend the task of dealing with regular path expressions. One particular method called *get-Descendants(Regular Expression)*, within the Navigational API, has the objective of pushing a regular expression down to each backend.

The motivation of this thesis is to manage efficiently regular path expressions without predicates in the relational backend. There is no traditional RDBMS indexing technique that generates an index to access efficiently such expressions. Although recursive SQL queries are not part of the SQL standard, some commercial relational databases support them. This type of queries can be used to solve regular path expressions, but currently the performance achieved by them is very poor. Moreover, when the regular expression contains the *descendants-or-self* operator it is not always obvious how to construct a recursive SQL query to solve it. Although we are restricting this exposition to regular path queries without predicates we believe that the same argument can be applied to regular path queries with predicates. As part of our future work, we plan to extend the toxRelational framework in this respect. During this exposition we use the terms regular path queries and regular path expressions as synonyms.

1.4 Thesis Outline

In the next chapter, we present an overview of the main research areas related to our work. In Chapter III we present our XML to Relational mapping scheme, along with its strengths and limitations. In the same chapter, we introduce the first optimization strategy that takes us towards our ultimate objective: solving regular path expressions efficiently. We add to our XML to Relational mapping scheme a known *Encoding* which allows the efficient identification of ancestor-descendant relationships. In Chapter IV, we present our second optimization strategy: exploiting *Materialized Views* as a query optimization tool. We define, populate, and exploit an optimal configuration of materialized views. In Chapter V, we present several experiments conducted to demonstrate the feasibility of our approach, and to measure the improvements achieved by the optimizations proposed in this thesis. Finally, in Chapter VI, we present our conclusions and suggest possible directions for future work.

CHAPTER II

Related Work

Our work is related to three main areas of research: mapping XML data to databases, optimizing regular path queries, and answering queries using materialized views. In this chapter, we provide an overview of each of these areas. As ToX is an XML repository, we also include a brief description of other comparable systems.

2.1 Mapping XML Data to Databases

Lately, a lot of effort has been applied to studying storage alternatives for XML data [Bar00, TDCZ00]. Techniques ranging from storing XML data as files (in the file system) to employing relational or object-relational databases have been developed. Also, several attempts to develop native storage mechanisms for XML data have been made. However, a long time will pass until this native technology reaches a mature point of development, and therefore, alternative strategies have to be exploited. Among these alternatives, relational databases are one of the strongest candidates. In this thesis, we will focus our attention on strategies to map XML data into relational and object-relational databases.

Grammar or Schema Independent Mappings

The first approaches in the development of this research area were based on the use of static relational schemas to store XML data. These methods are called *grammar or schema independent* because the structure of the XML document is not taken into account to generate the mapping. Several approaches [FK99b, SYU99, JLWY02a] have been developed in this context.

Florescu and Kossmann [FK99b, FK99a] described various alternatives to store XML data in relational databases. The objective behind their study was to examine how the simpler and more obvious approaches to map XML to relational databases behave. In this work, an XML document is represented as an ordered edge-labelled graph. For clarity, the authors divided the mapping problem into two subproblems: a) mapping elements and subelements; and b) mapping values. Among the strategies developed to store elements and subelements, three are worth mentioning: the *Edge Approach*, the *Binary Approach*, and the *Universal Approach*. In the *Edge Approach*, each edge in the edge-labelled graph is stored, as a tuple, in a single relational table. The *Binary Approach* groups all edges with the same label into the same relational table. Finally, the *Universal Approach* stores all edges in the same table, which corresponds to the result of a full outer join of the binary tables obtained by the *Binary Approach*. The difference between the *Edge Approach* and the *Universal Approach* is that in the former each tuple corresponds to a single edge in the edge-labelled graph, while in the latter each tuple corresponds to several edges corresponding to a path. In the *Universal Approach*, each different attribute name in the XML document is represented as a different field in the Universal relation. The authors also proposed two strategies to store XML values in a relational table: the *Separate Value Approach*, and the *Inlining Approach*. The first strategy distinguishes XML values by data types, storing each type in a different relational table. The second strategy stores XML values in the same relational table, together with their corresponding elements and subelements, using a different column for each data type. Some known drawbacks of all the proposed mappings are the number of join operations required for querying an XML document, the data fragmentation that results and the number of null values. Also, in some cases, it is required to know the schema in order to evaluate regular path queries. Despite all these drawbacks, the proposed techniques can be successfully employed by applications.

Another schema independent approach, known as XRel, was presented by Shimura, Yoshikawa, Uemura, and Amagas [SYU99, YASU01]. In this work, an XML document is decomposed into simple paths, and stored in an object-relational database. The mapping can be classified as a node-oriented approach, because it maintains nodes rather than edges. XRel stores, for each node in the XML graph, a single path and a pair of numbers associated with its starting and ending positions. The pair is called a *region*, and maintains the containment relationship (ancestor-descendant relationship). As the authors are using an object-relational database, a new data type, called *REGION*, was introduced to hold the *region* pair, in conjunction with two predicates to test inclusion relationships between different *regions*. The mapping is formed by four relational tables: *Element*, *Attribute*, *Text*, and *Path*. The first three tables store nodes of type element, attribute, and text respectively. The *Path* table stores information about simple paths. A simple path is a path from the root to a node in the XML graph. A simple path is identified by the string that contains the concatenation of labels along it. Due to this storage mechanism, answering regular path queries can become very inefficient, because each simple path has to be tested to determine if it satisfies the regular path query. Then, all simple paths that satisfy the regular path query have to be retrieved.

A more recent development in the area of schema independent mappings is XParent, developed by Jiang, Lu, Wang, and Yu [JLWY02a, JLWY02b]. XParent is an edge-oriented approach because it maintains edges individually. The mapping consists of four relational tables: the *LabelPath*, the *DataPath*, the *Element*, and the *Data*. The *LabelPath* table keeps all distinct label paths as tuples. The *DataPath* keeps the core structure of the XML data, based on the parent-child relationship. The *Element* and *Data* tables store information about element nodes and data nodes respectively. The XParent mapping uses a very similar schema to that of XRel. The main difference between them is that the *region* notion from XRel was replaced by a unique identifier in XParent. The XParent authors claimed that regular path expressions can be efficiently managed by manipulating the *LabelPath* table.

Grammar or Schema Dependent Mappings

Next, we will describe some of the developments in the area of *grammar or structure dependent* mappings. In these mappings, the relational schema is generated based on the characteristics of the XML document being mapped. DTDs, XML Schemas or simply the structure of a particular XML document instance can be used to extract the distinguishing features. Therefore, a relational schema can store XML documents with similar structural characteristics.

Monet is the schema dependent mapping presented by Schmidt, Kersten, Windhouwer, and Waas [SKWW00]. This mapping explores the structure of a particular XML document instance, and derives from it the corresponding relational schema. The main characteristic of Monet is the idea of storing, in the same relational table, all associations of the same type. An association is a pair of nodes of an XML document that are connected by some relationship (e.g., parent-child or element-attribute association). Storing associations of the same type in the same relational table causes tables to contain semantically closely related information. Therefore, Monet generates as many relational tables as associations exist in the XML document. The main disadvantage of this mapping is the high degree of data fragmentation, which implies that many join operations between tables are required during the query evaluation. However, the authors claimed that small amounts of data are involved in those joins, and hence the performance is more than acceptable. Nevertheless, processing regular path queries involving the descendants-or-self operator can be quite complicated using Monet because a DTD, XML Schema or the document instance structure have to be studied in order to determine which are the associations relevant to a particular query.

STORED [DFS99a, DFS99b] is an attempt to extract the most commonly used "structures" in an XML document, and to store these structures in a relational database. The uncommon structures are stored in a semistructured overflow database. Therefore, STORED can be categorized as a schema dependent mapping. STORED does not require a DTD, or an XML Schema as input. For determining which XML elements should be included in relational tables, and which should be kept in the overflow structures, STORED uses a novel approach: a modification of the apriori association rule algorithm developed by Wang and Li [WL98]. Parameters for STORED are the XML documents, a query workload, and some space constraints to apply to the relational schema (e.g., maximum disk space to be used by the relational tables, or maximum number of relational tables in the resultant schema). By applying the data mining algorithm, STORED can determine which path prefixes and substructures occur frequently in both the input XML data and the queries. Frequent substructures are prioritized, and the optimal relational mapping is selected taking into account the input constraints. The combination of a relational storage and a semistructured storage can produce performance benefits, but also adds complexity to the access mechanisms. The known disadvantages of STORED are the overhead produced by the mining algorithm, the difficulty of preserving the original XML document order, and its inability to express recursion.

XStorM [WLOT01] simply changes the management of the overflow database introduced by STORED. In the case of XStorM, uncommon structures are placed in overflow tables, within the same relational database where the common ones are placed. The authors of this technique try to avoid the overhead produced by the necessity of accessing an overflow database stored in the file system. They claim that a join operation between tables, in the same relational database, is much cheaper than combining access methods for relational databases and files.

Shanmugasundaram, Tufte, Zhang, He, DeWitt, and Naughtonet developed a technique [STZ+99] for analyzing the information in a DTD and deriving a suitable relational schema for XML documents conforming to it. The DTD analysis searches for shared and repetitive elements in the DTD, as well as for recursion in containment relationships. This approach does not preserve ordering relationships, and only supports limited forms of recursion. The authors proposed three different approaches to generate the relational schema. The main difference among them is the amount of redundant information allowed, which leads to different numbers of join operations between tables during the query evaluation. One thing to notice is that this technique does not require the scanning of particular XML document instances to derive a suitable relational schema. As no statistics about particular instances of XML documents are studied, it is possible that some portions of the relational schema will never be used. This situation will arise when the DTD contains some structures that are never used by any XML document that conforms to it. Although, the relational schema will be general enough to hold any XML document conforming to the original DTD, it also adds unnecessary complexity to the access mechanism.

One step further into schema dependent mapping was done by Klettke and Meyer [KM00]. The authors claimed that DTDs are only one of the considerations in defining a mapping from XML to databases. They pointed out that other aspects, such as frequency of element and attribute occurrences in XML document collections or most-often queried elements and attributes, must be taken into account to achieve the most efficient mapping scheme. Then, they proposed, not only to analyze the DTD of a particular XML document collection, but also to calculate weights determining the degree of relevance for each node in the DTD graph. This approach calculates weights derived from three different sources: the XML data, the queries, and the DTD structure. In summary, the best relational schema is generated based on the structures extracted from the DTD, and the weight information calculated for each node.

To conclude with schema dependant mappings, we present LegoDB [BFRS02]. LegoDB is a tool that implements a cost-based framework that automatically finds an efficient XML to relational mapping for a given target application. LegoDB parameters are: an XML Schema describing the XML data to be processed, a query workload for the target application, and data statistics. The authors of LegoDB, introduced the notion of physical XML Schema or *p-schema*, which is an XML Schema document specification extended with the data statistics about the underlying XML data. The authors also defined a fixed mapping from a *p-schema* to a relational schema. In order to generate an optimal relational schema, several storage configurations had to be analyzed. To obtain the space occupied by a storage configuration, several algebraic transformations are applied to a *p-schema*. Once the space of configurations is known, each query in the workload is translated to SQL (for each configuration), and also the data statistics are converted into relational statistics. With all this information in hand, LegoDB is able to exploit the relational optimizer to determine which configuration has the lowest cost. This procedure is applied iteratively until significant improvements are not longer obtained (i.e., the difference in cost between successive iterations is below a certain threshold).

2.2 Optimizing Regular Path Queries

A lot of effort has been applied to the area of optimizing regular path queries in the context of semistructured data. The objective of all those studies was to accelerate the evaluation of regular path queries exploiting the graph that represents the semistructured data. Also, prior studies of object databases can be applied to this context.

The advent of query languages featuring regular path expressions (presented as generalized path expressions or GPE's) was the main motivation of the work of Christophides, Cluet, and Moerkotte [CCM96]. They presented an algebraic approach to deal with GPE's in the context of object-oriented systems. Based on the known drawbacks of traditional techniques for evaluating GPE's, they proposed to extend the object algebra. This extension assigns to the query optimizer the task of dealing with GPE's. The object algebra was extended with two new operators: one dealing with paths at the schema level, and one dealing with paths at the instance level. The first operator, known as the *S-inst*, is applied to a set of tuples with the following parameters: a sequence of attributes and path variables, and a type restriction. The tuples contained in the set are extended with all the possible instantiations satisfying the original restrictions. The second operator, called *D-inst*, is applied to the set of tuples generated by the prior operator, restricting the search in the instance database. This work also includes an exposition about how the optimizer can exploit these new operators when dealing with GPEs.

The evaluation of regular expressions at runtime can be expensive. In the context of the LORE system [MAG+97], some work was done by McHugh and Widom [MW99] to overcome this inefficiency. The authors explore efficiency improvements by performing compile-time expansion of regular path expressions based on a structural summary called DataGuides [FLS98]. Compile-time expansion can eliminate significant amounts of unnecessary database exploration at run-time. Two strategies are applied to eliminate regular path expressions at compile-time: path expansion and alternation elimination. The idea is to precompute the query over the DataGuide, extracting all labelled paths that conform to the regular path expression. This precomputation eliminates unnecessary instance exploration. This work is similar in spirit to that done by Fernandez and Suciu using graph schemas [FS98]. A graph schema describes partial knowledge of the graph structure. It is used to restrict the search to certain fragments of the graph.

In the context of specialized XML search engines, a new index called eXtended Support Relations (XASR) was developed by Fiebig and Moerkotte [FM00] to optimize regular path queries. XASR is a scalable index that captures the structure of XML documents. It is based on Access Support Relations [KM92], which is an index that accelerates the evaluation of path queries in the context of object databases. XASR generalizes its predecessor in several ways. XASR does not materialize all possible paths in an XML document, because without a DTD, it is impossible to predict the set of potential paths. Even if the DTD were available, the existence of recursion can lead to an infinite set of possible paths. Another difference from Access Support Relations is that XASR supports generalized path expressions. XASR supports queries over the structure of an XML document. For each node of the XML graph, it stores two numbers: preorder and postorder. The ancestor-descendant relationship between any two nodes can be determined by comparing those numbers. In this way the index is used to prefilter all nodes that satisfy the generalized path expression within the query, without accessing the XML document instance.

Several recent works [AKM01, AR02, CKM02, CKM02] focus on defining a new labelling scheme, not only to define unique identifiers, but also to improve efficiency of structural queries and version management of XML document content. The main idea is to encode in the label of each node information about the XML hierarchy. With this labelling, the hierarchical relationship between two nodes can be determined by performing a comparison of their labels. There exist two approaches to labeling schemes: static and persistent settings. The former can be applied when the whole structure to be labelled is known in advance. One drawback of the static labelling is that, after any update, the whole labelling must be recomputed. The latter is not sensitive to updates, because the labels are assigned dynamically as the nodes are inserted. The criteria for assessing the quality of different labelling schemes is usually the lengths of the assigned labels.

2.3 Answering Queries using Materialized Views

Historically, materialized views have played an important role in the context of query optimization. The idea is to exploit precomputed data stored in materialized views to answer partially or completely a given query. To achieve this goal several issues need to be studied. A materialized view configuration has to be selected, and also a rewriting algorithm has to be implemented. The goal of the rewriting algorithm is to reformulate the original query in terms of the existing materialized views.

In the context of XML, Abiteboul [Abi99] pointed out that the problem of views becomes critical for integrating heterogeneous data sources, and for providing some structured interface on top of chaotic semistructured data. Abiteboul presented the first analysis in the area of defining views in XML. He proposed several definitions for views, and stated the similarities and differences between views in the XML context, and views in other known contexts.

A lot of work has been done in the area of answering queries using views. Halevy in his survey [Hal00] presents a taxonomy of the field. The main distinction among the works in the area is whether they focus their attention on solving a query optimization and physical data independence problem, or whether their goal is data integration. Halevy clearly differentiates between the two approaches, and gives a brief description of the more relevant works.

In the context of semistructured data, Lausen, Wei, and Marron [LWM] studied which issues need to be considered in query rewriting. The lightweight directory access protocol (LDAP) was used as the underlying data model for this work. They introduce a logical language for expressing LDAP-like queries (L^3DAP) and study query rewriting with emphasis on undefined attributes and constraints inherent from the LDAP semistructured data model. L^3DAP queries are expressed as a set of conjunctive queries. The authors apply a known algorithm, called the bucket algorithm [Lev99], for performing the rewriting.

Papakonstantinou and Vassalos [PV99] have done further work in the area of rewriting queries using views. They developed an algorithm for finding a query rewriting in MSL, the query language used in the context of the TSIMMIS system [GMPQ+97]. The algorithm is based on appropriately generalized containment mappings, chase, and unification, which are techniques developed for structured, relational data. It is important to note that the MSL queries involved in this study do not support regular path expressions.

Finally, an approach to rewriting regular expressions and regular path queries using views was presented by Calvanese, De Giacomo, Lenzerini, and Vardi [CGLV99]. Their work employs an automata-based algorithm to find a maximal rewriting, and also to check if the maximal rewriting is, in fact, an exact rewriting. In this thesis, we implement the automata-based approach proposed in Calvanese et al.'s work. A detailed description can be found in Chapter IV.

2.4 XML Data Repositories

The *Lightweight Object Repository* (LORE) [MAG+97] is a Database Management System (DBMS) designed specifically for managing semistructured data. The data managed by LORE is not confined to a schema, and it may be irregular or incomplete. In general, LORE attempts to take advantage of the structure where it exists, but also handles irregular data as gracefully as possible. LORE's data model is the Object Exchange Model (OEM) [PGMW95], which is simple and self-describing. LORE query language, known as Lorel, is an extension of OQL [CAD+94] that introduces extensive type coercion and powerful path expressions. Another novel aspect of LORE is the use of DataGuides [FLS98] in place of a standard schema. A DataGuide is a "structural summary" of the current database. LORE is fully functional and publically available [MAG+00].

Kweelt [Sah00] is a modular and extensible framework to query XML, where users can replace and extend components as they wish. Kweelt is an implementation of the Quilt Query Language [CRF00]. The main goal of the Kweelt implementation was to isolate completely the query engine from the storage module. The current implementation supports two different backends: a DOM backend, and a relational database backend. In the relational backend, the XRel mapping scheme [SYU99] was implemented. A systematic translation is made from Quilt to SQL, splitting single steps in the original path expressions. However, the authors of this implementation point out that this is not the best way to solve queries, and that a much more efficient way is needed. The main design issue of this implementation was to push as much work as possible to the backends, using as few queries as possible.

Strudel [FFK+98] applies concepts from database management systems to the process of building Web sites. Strudel illustrates the value of declarative specification of the content and the structure of Web sites, and the use of declarative query languages to automatically define them. The underlying data model of Strudel uses labelled directed graphs. This data model is appealing because Web sites are graphs with irregular structures and non-traditional schemas. The Strudel data repository can store data that lacks schema information. The Strudel data repository indexes both the structure of XML documents and the data itself. Strudel provides a novel language, StruQL, for querying and restructuring semistructured data.

Xyleme [ACGR01] is a dynamic Warehouse for XML data obtained from the Web or from private repositories. Xyleme uses the Natix native storage manager as its data repository. Natix uses a hybrid approach to store XML data, where some data is stored as trees and the rest is kept as byte streams. Within the Xyleme project, some pattern tree matching mechanisms were developed to accelerate the query answering procedure. As a data warehouse, Xyleme addresses all issues associated with data acquisition and maintenance. The notion of page importance is combined with a user defined specification for how XML pages are refreshed in the repository. Other relevant subsystems are the change control management and the subscription management.

CHAPTER III

ToX Relational Mapping Scheme

3.1 Introduction

In this chapter, we will describe the toxRelational component, which is the relational backend of ToX. Given that we are going to use a relational database to manage and query XML data, the first goal is to define a *mapping scheme*. A mapping scheme determines which tables and indices have to be constructed in the relational database, and how an original XML document is to be mapped into them. As we explained in Chapter II, there exist several meaningful mapping schemes, ranging from mapping schemes based on the schemas of the XML document (DTD, XML Schema or instance structure) to others totally independent of it.

In deciding which kind of mapping scheme we are going to use, we have to take into consideration several issues. We will describe these issues from two different points of view:

i) Mapping procedure. The mapping procedure for a grammar independent mapping scheme is straightforward because it has to decompose the XML document into its components, storing them into a fixed relational schema. Analyzing different relational schemas of grammar independent mappings, we can conclude that usually they are composed of a small number of relations. On the other hand, the mapping procedure for grammar dependent mappings is more complex because it includes algorithms for analyzing the structure of the XML document in order to generate an good relational schema automatically. From past approaches, we can observe that the generated relational schema is composed by a huge number of relations. Basically these approaches search for structured portions of XML documents to generate the relational schema. Due to XML's irregular nature, this normalization leads to high degree of data fragmentation.

ii) Querying procedure. Both independent and dependent mapping schemes, with their different access methods, face identical problems. Among them, we can determine the number of SQL queries generated by the translation from an XML query, the number of join operations within each SQL query, and how much recursion occurs in the case of regular path XML queries.

For this thesis, we have chosen a mapping scheme that is completely schema independent. Our goal was to select an interchangeable and general mapping scheme, so that all the reasoning we did can be applied directly to an alternative mapping scheme. Our mapping is an extension of the *YAM* mapping scheme [AB02] developed within the ToX project. The goal of the extension is to accelerate the query processing procedure, and, to achieve that objective, we have added extra information to the original mapping scheme.

The ultimate goal of the toxRelational component is to support several mapping schemes in order to allow its evolution, and also to provide multiple alternatives for storing XML documents. This goal basically emerged from the knowledge that no single mapping scheme is good for all document structures and access strategies. For this reason, in the current architecture of the toxRelational component, a new mapping scheme can be easily plugged-in, by implementing the mapping itself and the required access methods. Implementing the mapping itself means coding the routine that traverses the XML document while inserting tuples in the corresponding relational tables (Section 3.3). Implementing the access methods means generating all SQL statements needed to access the new mapping scheme. We will describe how to implement the access methods in Section 3.4.

3.2 ToX Relational Mapping Scheme

3.2.1 Overview

The ToX Relational mapping scheme consists of three relations: *Edge*, *Leaf*, and *Attribute*. This mapping can be categorized as node-oriented since it stores the nodes of an XML document taking into account their types. Next, we describe the relations that compose the ToX Relational mapping scheme together with a set of suggested indices. In Section 3.3 we present an example of how an XML document is mapped into the ToX Relational mapping scheme.

Edge Relation

The *Edge* relation stores one tuple per edge between elements in the ordered edge-labelled graph. In Table 3.1 we describe the fields of the *Edge* relation. During the mapping process, a unique *Object Identifier* (OID) is assigned to each element in the XML graph. Also, the order of the elements is captured and stored (in the *Ordinal* field) using a very naive strategy: storing the position of an element among its siblings. The preorder field captures the depth-first search traversal order of the elements in the XML document. Although the preorder uniqueness feature makes it possible to use this number as a unique identifier for each element, we decided to use an *Object Identifier* independently assigned for clarity.

Field Name	Definition
Docid	Unique document identifier
Source	Object identifier corresponding to the source element of the edge
Target	Object identifier corresponding to the target element of the edge
Name	Tag name of the element
Ordinal	Order among its sibling elements
Preorder	Preorder number in the depth-first traversal

Table 3.1: *Edge* Relation Structure

In order to select a useful set of indices, it is necessary to know how the relations are accessed for querying. The Document Navigator is used to access the XML documents stored in the repository. Therefore, we analyzed all SQL statements issued from the relational implementation of the Document Navigator, to derive the set of indices for each relation in the mapping scheme. (Refer to Appendix B for a list of SQL statements issued from the Relational Document Navigator).

Based on this analysis, we propose to construct four indices for the *Edge* relation: one combining the *Docid* and *Source* fields, other combining *Docid* and *Target* fields, the third one combining *Docid*, *Source* and *Target* fields, and finally an index on the *Name* field. With this index configuration we ensure that all SQL statements issued from the Relational Document Navigator can be answered using a precomputed relational index. (Refer to Section 3.4 for information about the access methods.)

Leaf **Relation**

The *Leaf* relation stores one tuple per edge between an element and a TEXT or CDATA value in the XML document (i.e., leaves in the XML graph). In Table 3.2 we present the fields of the *Leaf* relation. Note that, in the current implementation, *Character* and *Integer* are the only detected data types for leaf values.

Field Name	**Definition**
Docid	Document identifier
Source	Object identifier corresponding to the source element of the edge
Target	Object identifier corresponding to the target TEXT or CDATA of the edge
Type-Value	Data type of the value
Char-Value	Character value
Int-Value	Integer value
Ordinal	Order among its sibling elements
Preorder	Preorder number in the depth-first traversal

Table 3.2: *Leaf* Relation Structure

In term of indices, based on our analysis of the querying procedure, we propose to construct three indices: one combining the *Docid* and *Source* fields, other combining *Docid* and *Target* fields, and finally an index combining *Docid*, *Source* and *Target* fields. This schema is analogous to the one we proposed to use for the *Edge* relation.

***Attribute* Relation**

This relation stores one tuple per edge between an element and an attribute. In Table 3.3 we describe its fields. Note that, among all possible attribute types, ID and IDREFS types are the only detected by the current implementation.

From our analysis of the query procedure, we concluded that, for the *Attribute* relation, two indices are needed: one combining the *Docid* and *Source* fields, and other one combining the *Docid*, *Source*, and *Attr-Name* fields.

Field Name	Definition
Docid	Document identifier
Source	Object identifier corresponding to the element parent of the attribute
Attr-Name	Name of the attribute
Char-Value	Character value of the attribute
Attr-Type	Attribute type

Table 3.3: *Attribute* Relation Structure

3.2.2 Comparing the ToX Relational Mapping Scheme with Other Approaches

The ToX Relational mapping scheme can be classified as a grammar or schema independent mapping because the structure of the XML document is not taken into account to generate the mapping. Comparing this mapping with the ones described in Chapter II, we find that the structure of the relational schema is very similar to that of the *Edge* approach proposed by Florescu et al. [FK99b, FK99a], and to that of the XRel mapping proposed by Shimura et al. [SYU99, YASU01].

The structure of the ToX Relational mapping scheme differs from the one of the *Edge* approach in storing nodes of different types in different relations, instead of all nodes in the same relation. Hence, the ToX *Edge* relation will have fewer tuples than the *Edge* relation of the Florescu et al. approach, because, while the ToX *Edge* relation stores only nodes of type element, the Florescu et al. *Edge* relation stores all nodes in the document regardless of their types. Taking into account that the data stored in the *Edge* relation (for both mappings) is the one used for traversing the XML hierarchy (i.e., to answer regular path queries), the performance of this traversal will be somewhat better in the case of the ToX *Edge* relation because of its reduced size.

The main difference between the ToX Relational mapping scheme and the XRel mapping is the way the two mappings maintain the ancestor-descendant relationships. The ToX Relational mapping scheme stores in the *Edge* relation the parent-child relationship, while the XRel mapping stores the path from the root to each node in the XML document, and a region number to maintain containment relationships.

3.3 Mapping an XML Document into the ToX Relational Mapping Scheme

The mapping operation is done by a depth-first traversal of the XML graph. For each node in the XML graph, we determine its type and act accordingly. Each node in the XML graph is traversed only once. In Figure 3.1 we present our mapping algorithm.

Let see with an example how the ToX Relational mapping scheme works. In Figure 3.2 we show a subset of the XML graph for the DBLP database presented in Chapter I annotated with the values assigned by the mapping algorithm. In this extended XML graph, the values of OID are shown for all nodes, and the Name and Preorder values are shown for all edges. Next, in Figure 3.3, we show the instances of the *Edge*, *Leaf* and *Attribute* relations that result from the mapping procedure along with a subset of one of the generated indices. Note that the *Docid* field was omitted in the example because it deals with a single XML document.

In the *Edge* relation, there is a tuple for the *conferences* element. The algorithm assigns a unique object identifier to the source node and the target node of this edge. The Name field holds the tag name of the element, which is *conferences*, and the Ordinal field contains the value "1", because the element has no other siblings. Finally, the Preorder field is the preorder number assigned during the depth-first traversal. Similarly, two tuples in this relation have the string *conference* as the value of their Name fields. We can observe that both tuples share the same Source

```
INPUT: Root Node of the XML graph
OUTPUT: Decomposition of an XML document into the ToX
                    Relational Mapping Scheme

PROCEDURE:
   Node = Root of the XML-graph
   constructMapping(Node, null, 1)


   procedure constructMapping(Node, ParentNode, Preorder)

      Type = getNodeType(Node)
      begin case
            case Type = ELEMENT

                  Increment global Preorder
                  processAttributes(Node)

                  for each child in children(Node)
                        constructMapping(child, Node, Preorder)
                  end for each

                  Insert new tuple into the Edge relation from ParentNode to Node

            case Type = CDATA or TEXT

                  Increment global Preorder
                  Insert new tuple into the Leaf relation from ParentNode to Node

      end case
   end procedure

   procedure processAttributes(Node)

      for each attribute in attributes(Node)
            AttrType = getAttributeType(attribute)
            Insert new tuple (Node, Attribute, AttrType) into the Attribute relation
      end for each

   end procedure
```

Figure 3.1: ToX Relational Mapping Algorithm

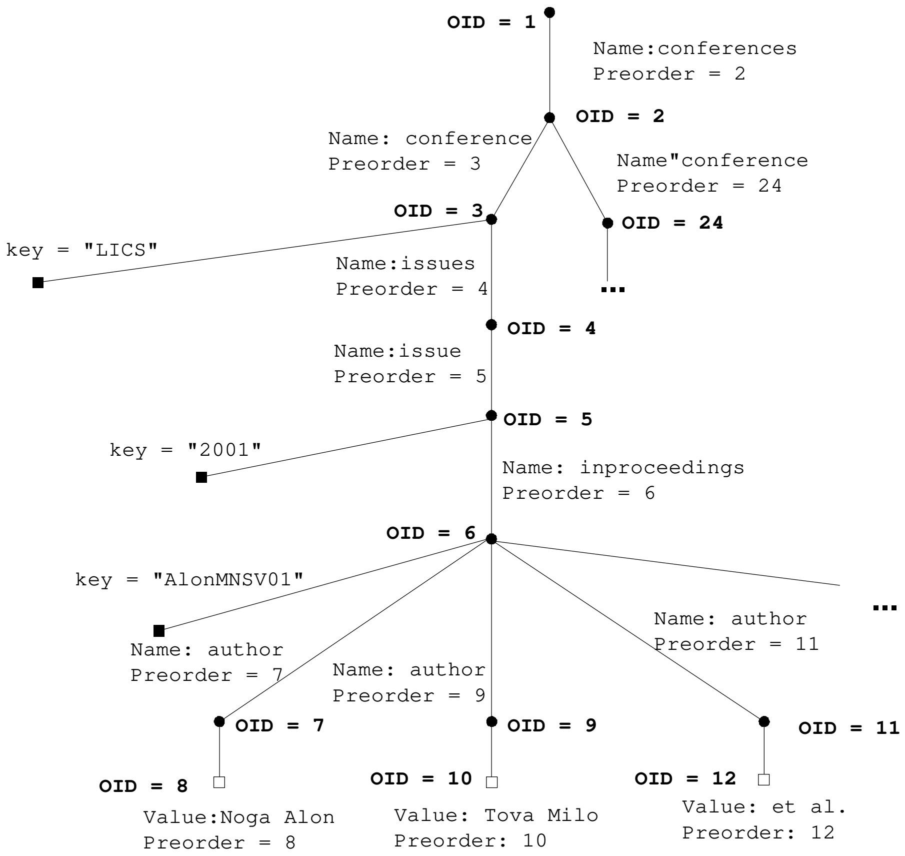

Figure 3.2: Extended XML Graph for the DBLP database

Edge Relation

Source	Target	Name	Ordinal	Preorder
1	2	conferences	1	2
2	3	conference	1	3
2	24	conference	2	24
3	4	issues	1	4
4	5	issue	1	5
5	6	inproceedings	1	6
6	7	author	1	7
6	9	author	2	9
6	11	author	3	11
...	...	...	...	...

Leaf Relation

Source	Target	Type-Value	Char-Value	Int-Value	Ordinal	Preorder
7	8	CHAR	Noga Alon		1	8
9	10	CHAR	Tova Milo		1	10
11	12	CHAR	et al.		1	12
...	...	...	...	...	...	...

Attribute Relation

Source	Attr-Name	Char-Value	Attr-Type
3	key	LICS	ID
5	key	2001	ID
6	key	AlonMNSV01	ID
...	...	...	...

Index over the Edge Relation (Docid + Source + Target)

Source	Target
1	2
2	3
2	24
3	4
4	5
5	6
6	7
6	9
6	11
...	...

Figure 3.3: Instances of *Edge*, *Leaf*, and *Attribute* Relations plus an index over the *Edge* Relation

object identifier. This happens because there exist two edges labelled *conference*, children of the same node. The values of their Ordinal fields are “1” and “2” respectively, because they are siblings and we have to capture the original order between them.

A similar procedure was followed to populate the *Leaf* relation. In this example, all values are of type *character*. Finally, the *Attribute* relation holds several attributes with type *ID* named *key*. The attribute types are extracted from the schema information, if it is available.

In the example we show one of the indices over the *Edge* table. The index combines the *Docid*, *Source*, and *Target* fields. Note that the *Docid* field was omitted because all tuples corresponds to the same XML document. This index will be used by some of the SQL statements issued by the Relational Document Navigator. (Refer to Appendix B for details on the access methods.)

3.4 Querying the ToX Relational Mapping Scheme

As we already mentioned in Chapter I, all backends in ToX can be queried using the Navigational API. This API is known within ToX as the *Document Navigator*. The Document Navigator allows the query processor to navigate the stored XML documents, regardless of the complexity or details of each backend. The Document Navigator was designed to facilitate the query processing procedure while hiding implementation details. Its functionality is similar in spirit to the one described in the Document Object Model Specification (DOM) [DOM98]. The Document Navigator specification was not defined only for ToX internal use. Any external application can exploit the Document Navigator potential for traversing and constructing XML documents.

The Document Navigator interface represents the logical structure of the XML graph. The XML graph is built on demand, while the query processor (or the external application) is answering a given XML query. The Document Navigator recognizes different entities within an XML document. Some examples of these entities are: *XDocument*, *XElement*, *XAttribute*, and *XNode*. These entities are comparable to those of the DOM tree (Document, Element, Attribute, and Node). In the Document Navigator, several methods are defined over each type of entity. Queries are performed over nodes. The ToX query processor normally starts querying an XML document from its root node. From this initial node, the query processor can invoke different access methods, possibly over different types of entities, to finally construct the answer to the query.

We can classify the access methods of the Document Navigator in two categories: the navigational methods, and the constructional methods. The first category allows the query processor to navigate the XML graph, traversing it in some predefined ways (e.g., return the children, return the parent, etc.). The second category includes all access methods that permit the creation and reordering of entities in an XML graph (the original XML graph, or a newly created answer).

In Table 3.4 we present a brief description of the most important navigational methods within the Document Navigator. A complete specification of the Document Navigator can be found in Appendix A.

In the relational implementation of the Document Navigator, each access method is translated into SQL statements that retrieve the necessary information to answer the input query. As the motivation of this thesis is to speed up the response time for regular path queries, we will describe in more detail the *getDescendants(Regular Expression)* method of the Document Navigator. This method deals with regular path queries, taking as input a regular expression. After process-

Entity	**Method**	**Description**
Document	getRoot()	Returns the root node of a given XML Document
Node	getAncestors()	Returns a list that contains all ancestors of this node (not necessarily in order)
Node	getAncestors(RegExp)	Returns a list that contains all ancestors of this node reached by a path that satisfies the given regular expression
Node	getChildren()	Returns a list that contains all children of this node (not necessarily in order)
Node	getChildren(name)	Returns a list that contains all children of this node with the given name (not necessarily in order)
Node	getDescendants()	Returns a list that contains all descendants of this node (not necessarily in order)
Node	getDescendants(RegExp)	Returns a list that contains all descendants of this node reached by a path that satisfies the given regular expression
Node	getType()	Returns the type of this node
Node	getValue()	Returns the value of this node
Node	hasChildren()	Returns true/false whether this element has children or not
Node	hasParent()	Returns true/false whether this element has a parent or not

Table 3.4: Some Navigational Methods of the Document Navigator

ing the regular expression, the *getDescendants(Regular Expression)* method returns a list of nodes satisfying it (i.e., the method returns the root nodes of the subtrees that satisfy the given regular expression). For example, the execution of *getDescendants(/conferences/conference/issues/issue)* over the XML document in Figure 3.2 will return a list with a single node representing the *inproceedings* element.

In order to present the algorithm that performs the hierarchy navigation, we need to define a *query graph*. Query graphs are transition graphs constructed from the non-deterministic finite automaton (NFA) corresponding to a regular expression.

Definition 1. ***Query Graph.*** *Let $P = (S, \Sigma, \delta, s_0, F)$ be a NFA, where S is the set of states; Σ a set of input symbols (i.e., alphabet); δ is the transition function that maps state-symbol pairs to the set of states; s_0 is the distinguished start (or initial) node; and F is the set of accepting (or final) states. The* ***query graph*** *associated with P is a directed graph $Q = (S, s_0, E_Q, \psi_Q, \Sigma, \lambda_Q, F)$ where S is the set of states; $s_0 \in S$ is a distinguished node called the initial node; E_Q is the set of edges; ψ_Q is the function that maps E_Q to $S \times S$; Σ is the alphabet of Q; λ_Q is the labelling function mapping E_Q to Σ; and F is the set of final states. If state $t \in \delta(s, a)$ for $s, t \in S$ and $a \in \Sigma$, then there is an edge $e \in E_Q$ with $\psi_Q(e) = (s, t)$ and $\lambda_Q(e) = a$.*

By using *Thomson's Construction* [ASU86] we compute first the NFA corresponding to a regular expression, and then we construct the query graph associated with it.

The *getDescendants(Regular Expression)* algorithm is shown in Figure 3.4. The algorithm traverses, at the same time, the XML graph and the query graph associated with the regular expression. Given an initial node, the algorithm traverses its children. To obtain the list of children ($children(Node)$ in Figure 3.4), an SQL statement is issued. This SQL statement exploits all available relational indices. For each child the algorithm verifies whether or not a transition exists in the query graph from the set of current states (StateSet) where the label name equals to the child name. If a transition is found, then the algorithm checks whether the new state reached is a final state in the query graph. If the state is final it means that the regular expression was satisfied, and therefore the child node is one of the wanted descendants, and it is added to the list. If the state is not final, then we add it to the set of current states and continue recursively with the procedure. At the end, all nodes satisfying the regular expression will be stored in the list of descendants. Note that neither the hierarchy nor the structure among nodes is maintained in the result. At worst, each edge in the XML graph will be traversed once, and therefore the complexity of the *getDescendants(Regular Expression)* is $O(n)$ where n is the number of edges in the XML graph.

Now, we present an example to clarify the *getDescendants(Regular Expression)* algorithm using the DBLP database. The initial node is the root of the XML document (a *conferences* element). In Figure 3.5 we present an instance of the DBLP database. Let Q be the regular path query *conferences/conference/issues/issue/ inproceedings/author*. The query graph associated with Q is shown in Figure 3.6. The result of applying the $getDescendants(RegularExpression)$ algorithm to the XML graph will be the three *author* element nodes in it, because when the algorithm reaches those nodes, it also reaches a final state in the query graph. Therefore, those nodes satisfy the regular expression associated with the query graph. When the algorithm reaches any other element node in the XML graph, it is not the case that it also reaches a final state in the query graph, and hence only the elements labelled *author* contribute to the result of the algorithm.

3.5 Implementation

```
INPUT: Query graph Q = (S, s_0, E_Q, ψ_Q, Σ, λ_Q, F)
            Initial Document Navigator Node
OUTPUT: List of nodes satisfying the regular expression associated with Q

PROCEDURE:
   Node = Initial Document Navigator Node
   getDescendants(Node, null, null)


   procedure getDescendants(Node, Descendants, StateSet)

      for each child in children(Node)
            for each edge e_Q s.t. ψ_Q(e_Q) = (s, t) with s ∈ StateSet, and
                        λ_Q(e_Q) = getName(child)

                  if (t ∈ F) then
                        Add child to Descendants
                  else
                        Add t to StateSet
                  end if
                  getDescendants(child, Descendants, StateSet)
            end for each
      end for each
   end procedure
```

Figure 3.4: Document Navigator *getDescendants(Regular Expression)* Algorithm

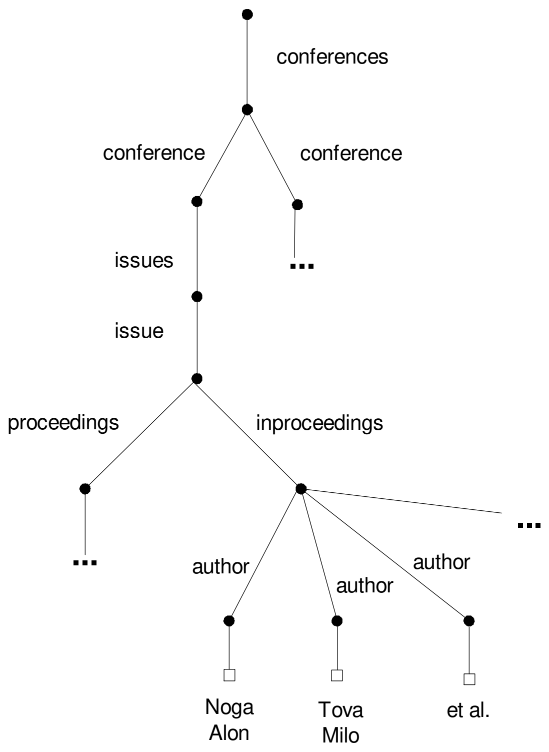

Figure 3.5: Instance of the DBLP Database

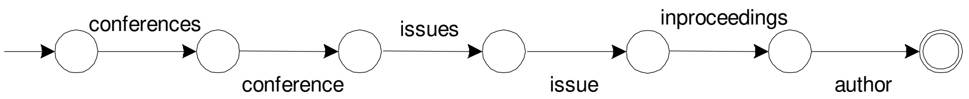

Figure 3.6: Query Graph Associated with Q

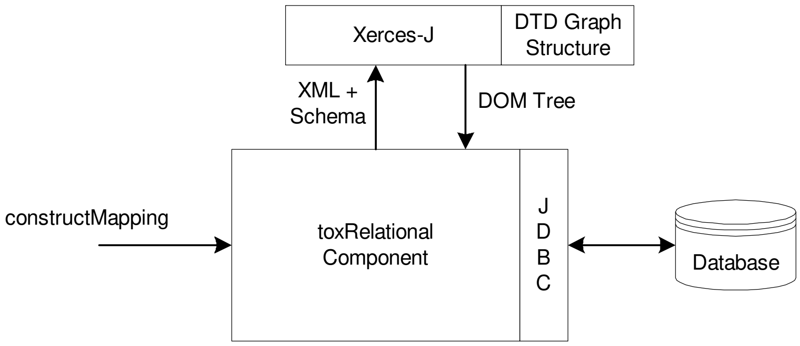

Figure 3.7: The toxRelational Component Architecture

3.5.1 Architecture

We implemented the toxRelational component in Java, and we used Java Database Connectivity (JDBC 2.0) technology to access the relational database. JDBC provides cross-platform DBMS connectivity to a wide range of SQL databases [JDB]. We also used the Xerces Java Parser (Xerces-J Version 1.4.1). The Xerces-J parser supports the XML 1.0 recommendations, and contains advanced parser functionality, such as support for the W3C's XML Schema recommendation version 1.0, DOM Level 2 version 1.0, and the Simple API for XML (SAX) Version 2, in addition to supporting the industry-standard DOM Level 1 and SAX version 1 APIs. The standard DOM is an application programming interface (API) for XML documents. It defines the logical structure of XML documents, and also provides an API that defines the way each node in the DOM tree is accessed and manipulated. DOM is a node-labelled tree-like structure. We decided to use the DOM tree generated by the Xerces-J parser. We traverse the DOM tree, converting it from its initial representation to an edge-labelled representation in order to generate the input for the constructMapping procedure (Figure 3.1). In Figure 3.7 we can see the architecture just described.

The ToX Relational mapping scheme requires some structural information about each attribute, such as its type. Unfortunately, the Xerces-J parser does not provide external access to the grammar information of the XML document (i.e., its DTD or XML Schema). Thus, we extended the Xerces-J parser to extract such information. We added a DTD Graph structure to the result of the parsing operation. Hence, the Xerces-J parser generates not only the DOM tree, but also a DTD Graph (if the information is available). At any time during the DOM traversal, we can know its typing information by extracting data from the DTD Graph. Therefore, to extract the necessary structural information about attributes, we used the DTD Graph structure.

3.5.2 Mapping

The ToX Relational mapping scheme stores elements, attributes and text values. Other types of nodes, such as processing instructions and comments, are not captured by the current implementation.

As we said the toxRelational component can support several mapping schemes, and stores multiple XML documents using one of the supported mapping schemes. Information about XML documents and mapping schemes must be stored in order to manage the repository. In the current implementation, this information is stored in a single relation called *Document Catalog*. The structure of the *Document Catalog* relation is described in Table 3.5. The XMLName, XMLPath, DTDName and DTDPath fields only make sense in the case that the toxRelational component is used directly by an external application. (Within ToX all references to XML documents are done by the unique document identifier.)

We performed some experiments concerning the loading times using the current implementation and also we measured the size of the resulting database. The

Field Name	Definition
Docid	Document identifier
XMLName	Name of the XML file
XMLPath	Information about the file system path where the document is stored
DTDName	Name of the DTD associated with this document
DTDPath	Information about the file system path where the DTD is stored
Mapping-Type	Identification of the mapping scheme
RootId	Object identifier of the root element for this document

Table 3.5: *Document Catalog* Relation Structure

complete details of these experiments and their results are presented in Chapter V Section 5.2.

3.5.3 Relational Document Navigator

Like other backends in ToX, the toxRelational component accepts query requests via the Document Navigator. We implemented the complete set of access methods according to the Document Navigator specification (see Appendix A). Each method in the Document Navigator is translated into a set of SQL statements over the relational backend. Thereafter, the relational results (tuples in the relational backend) are formatted and converted into the required output format. As an example, in Figure 3.8, we present a translation from the *getChildren()* access method of the Document Navigator to an SQL statement. The Document Navigator *getChildren()* simply retrieves the children of the current node (in this example, *NodeA*). In the ToX Relational mapping scheme, the parent-child relationship can be derived from the *Source* and *Target* fields stored in the relations. In this case, we need to retrieve the children of a particular node, which can be either elements (stored in the *Edge* relation) or values (stored in the *Leaf* relation). Therefore, the SQL statement generated by the Document Navigator contains the union of the results of two

$NodeA.getChildren() \Longrightarrow$	**select** target, ordinal, name, 0 as type-value, "" as char-value, -1 as int-value, preorder **from** Edge **where** docid = NodeA.DOCID and source = NodeA.OID **union all** **select** target, ordinal, "" as name, type-value, char-value, int-value, preorder **from** Leaf **where** docid = NodeA.DOCID and source = NodeA.OID

Figure 3.8: From the Document Navigator to SQL

individual SQL statements (one that retrieves tuples from the *Edge* relation, and another one that retrieves tuples from the *Leaf* relation). The conditions that the tuples must satisfy to be a child of *NodeA* are that they belong to the same XML document (i.e., docid = *NodeA*.DOCID) and that their source object identifier is equal to the object identifier of *NodeA* (i.e., source = *NodeA*.OID). In Appendix B we present the main methods of the Document Navigator and their translations into SQL statements.

In the previous section, we explained the *getDescendants(Regular Expression)* algorithm. The algorithm traverses simultaneously the XML graph and the query graph. The XML graph is traversed using the $getChildren()$ method of the Document Navigator just explained. As we can derive from the algorithm, an SQL query will be posted to the relational database for each node searching for children nodes. Therefore, with each SQL statement, we will descend one level in the XML hierarchy. We can say that this is a level by level or step by step approach for traversing the XML graph. This approach is easy to implement, and it also has the characteristic that all SQL queries are predefined. The knowledge of the SQL queries allows us to create a suitable index configuration for them. We also avoid the self-join operations and recursive SQL queries. These two operations, depending on the relation characteristics can be very expensive, and also recursive SQL queries are not supported in all commercial RDBMS, which reduces the chances of portability. Although the step by step approach has all these advantages, previous studies [Sah00] have suggested that it might not be the best way of accessing a relational database that holds XML data. In this study, the authors suggested that the implementation should dynamically generate the set of SQL statements that exactly answers the input regular expression. This means generating one or more SQL statements to answer the input regular expression. Because this approach of generating a dynamic and optimal SQL query to access the data seems very complex, we decided to implement the straightforward approach and leave for future improvements the implementation of a general algorithm.

As we explained in Chapter I, the ToX query processor pushes all regular path queries to the backend to improve performance. Therefore, the toxRelational component has to provide an efficient way to manage regular path queries in the relational backend. In the next section, we present the first step in optimizing regular path queries using an encoding that maintains containment relationships.

3.6 Extending the ToX Relational Mapping Scheme Using Ancestor-Descendant Information

An XML graph models the XML document as a tree. The ToX Relational mapping scheme represents the XML graph in several relations. Using this mapping, the parent-child relationship between two nodes can be determined by issuing a single SQL statement. Although this knowledge is enough for answering regular queries, the query answering procedure can be inefficient in the cases where the path lengths to be traversed are either very long or not known. This is exactly the case that arises when the regular path query contains the *descendants-or-self* operator. To clarify this issue, we present a simple example. Suppose that query Q_1 extracts all *inproceedings* elements from *conferences* in the DBLP data set presented in Chapter I (Figure 1.1). Query Q_1 can be represented by the regular path query */conferences//inproceedings*. This regular path query contains the descendants-or-self operator, which extracts elements at any level in the XML hierarchy. The *getDescendants(Regular Expression)* method from the Document Navigator will be executed to answer this query. If we apply the *getDescendants(Regular Expression)* algorithm, the entire hierarchy below the */conferences* elements will be traversed, which means that almost the whole XML document will be scanned. Although this is a valid solution, it is worth seeking an alternative that is more efficient.

As long as we find an efficient way to determine the ancestor-descendant relationship between two nodes, we can avoid the mentioned inefficiency in some cases. Suppose that we find a procedure to check the ancestor-descendant relationship between two nodes. With this algorithm in hand, query Q_1 can be solved in a very different way. We can retrieve independently the *conferences* elements and the *inproceedings* elements in the database, and then we can apply the algorithm retrieving only the pairs of elements *conferences-inproceedings* (C_i, I_j) where the *inproceedings* I_j element is a descendant of the *conferences* C_i element. This new resolution algorithm avoids the massive traversal of the XML hierarchy. Depending on the XML document and the query characteristics, it might be the case that this new method will perform better.

3.6.1 Determining Ancestor-Descendant Relationship

We are going to present an encoding due to Li and Moon [LM01] that uses the tree traversal order to determine the ancestor-descendant relationship between any two nodes in a tree. This encoding is based on the Dietz numbering scheme developed in the context of linked lists. Using the mentioned encoding, the ancestor-descendant relationships can be determined in constant time, only by examining a pair of numbers. The encoding is known as the *Extended Preorder and Range of Descendants*. Interestingly, this encoding was not defined particularly for XML data, and in fact it can be applied to any tree-like structure representation.

The *Extended Preorder and Range of Descendants* encoding associates with each node in a tree a pair of numbers $(order, size)$ satisfying the following rules:

Rule 1: For a tree node y and its parent x, $order(x) < order(y)$ and $order(y) + size(y) \leq order(x) + size(x)$. In other words, the interval $[order(y), order(y) + size(y)]$ is contained in the interval $[order(x), order(x) + size(x)]$

Rule 2: For two sibling nodes x and y, if x is the predecessor of y in preorder traversal, then $order(x) + size(x) < order(y)$

Therefore, for a given node x, $size(x) \geq \sum_y size(y)$ for all $y's$ that are direct children of x. Then, the ancestor-descendant relationship for a pair of nodes can be determined by examining their *order* and *size* values. Using the following Lemma, proved by Li and Moon, the ancestor-descendant relationship can be determined:

Lemma 1. *For two given nodes x and y of a tree, x is an ancestor of y if and only if $order(x) < order(y) \leq order(x) + size(x)$*

In Figure 3.9 we present an example. Each node of the tree was numbered following the definition of the Extended Preorder and Range of Descendants encoding. The following relationships in the tree can be derived applying Lemma 1:

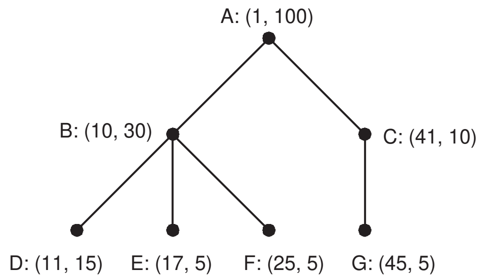

Figure 3.9: Encoding Example

- Node F is descendant of Node B because

$$\begin{cases} 10 < 25 \;\Rightarrow\; order(B) < order(F) \; and \\ 25 \leq 10 + 30 = 40 \;\Rightarrow order(F) \leq order(B) + size(B) \end{cases}$$

- Node G is descendant of Node A because

$$\begin{cases} 1 < 45 \;\Rightarrow order(B) < order(F) \; and \\ 45 \leq 100 + 1 = 101 \;\Rightarrow order(F) \leq order(B) + size(B) \end{cases}$$

- but Node G is not descendant of Node B because

$$\begin{cases} 10 < 45 \Rightarrow \; order(B) < order(F) \;\; but \\ 45 \nleq 40 \;\Rightarrow order(F) \nleq order(B) + size(B) \end{cases}$$

3.6.2 Adding the Encoding to the ToX Relational Mapping Scheme

The encoding stores the *order* and *size* values for each node. The *order* is exactly the preorder number generated during the depth-first traversal of the XML document. As this number is already stored in the ToX Relational mapping scheme, we only need to add the *size* value to it. In order to add this new element to the mapping scheme a very small change to the mapping structure is required. A new field named *size* is added to the *Edge* and *Leaf* relations.

This extension to the ToX Relational mapping scheme leads to a modification of the mapping algorithm presented in Figure 3.1. In Figure 3.10 we show the modified version of the mapping algorithm with the computation of the size value for each node. In Figure 3.11 we show an example of how the XML document presented in Figure 1.1 gets mapped into the ToX Relational mapping scheme.

3.6.3 Exploiting the Encoding in the Relational Document Navigator

As we already pointed out, the encoding will be useful in the cases when the regular path query contains the descendants-or-self operator. As we explained in the previous chapter, the *getDescendants(Regular Expression)* is the method from the Document Navigator that deals with this kind of queries, pushing regular expressions into the backends. Therefore, to exploit the encoding in the Relational Document Navigator, we have to extend the *getDescendants(Regular Expression)* algorithm presented in Figure 3.4. This extension does not change the overall approach taken by the previous version of the *getDescendants(Regular Expression)* algorithm. We developed a new procedure to be executed instead of the original *getDescendants(Regular Expression)*. The new procedure is called *getDescendantsExploitingEncoding(Regular Expression)* and it is presented in Figure 3.12.

To exploit the encoding, we internally simplify the complexity of the regular expressions managed by the Document Navigator. If we avoid combining a descendants-or-self operator with any other expression, then we can easily apply the optimization any time this operator appears. Therefore, a regular path query is

```
INPUT: Root Node of the XML graph
OUTPUT: Decomposition of an XML document into the ToX
            Relational Mapping Scheme
PROCEDURE:
  Node = Root of the XML-graph
  constructMapping(Node, null, 1)

  procedure constructMapping(Node, ParentNode, Preorder, totalSize)
    Type = getNodeType(Node)
    begin case
        case Type = ELEMENT
            Increment global Preorder
            processAttributes(Node)
            for each child in children(Node)
                totalSize = constructMapping(child, Node, Preorder, totalSize)
            end for each
            if totalSize > Preorder then
                currentSize = totalSize + 1;
            else
                currentSize = Preorder + 1;
            end if
            Insert new tuple into the Edge relation from ParentNode to Node
            with size = currentSize

        case Type = CDATA or TEXT
            Increment global Preorder
            currentSize = SIZE
            Insert new tuple into the Leaf relation from ParentNode to Node
            with size = currentSize
            totalSize = totalSize + currentSize
    end case
    return totalSize
  end procedure

  procedure processAttributes(Node)
    for each attribute in attributes(Node)
        AttrType = getAttributeType(attribute)
        Insert new tuple (Node, Attribute, AttrType) into the Attribute relation
    end for each
  end procedure
```

Figure 3.10: New ToX Relational Mapping Algorithm

Edge Relation

Source	Target	Name	Ordinal	Preorder	Size
1	2	conferences	1	1	195
2	3	conference	1	2	151
3	4	issues	1	3	150
4	5	issue	1	4	149
5	6	inproceedings	1	5	148
6	7	author	1	6	9
...	...	...	...	...	...

Leaf Relation

Source	Target	Type-Value	Char-Value	Int-Value	Ordinal	Preorder	Size
7	8	CHAR	Noga Alon		1	7	1
...	...	...	...	...	...	...	...

Figure 3.11: Mapping the XML document in Figure 1.1 into the Extended ToX Relational Mapping Scheme

analyzed and divided by the Relational Document Navigator into several subexpressions following some constraints. The Relational Document Navigator is going to divide the input regular path query into several subexpressions taking into account the following constraints:

1. no subexpressions contain a descendants-or-self operator as part of it, or
2. if the descendants-or-self operator is in a subexpression, then it is the only operator in it.

Once the set of subexpressions is determined, we have to evaluate them in the appropriate order to retrieve the result. So, starting from the initial set of nodes (i.e., any nodes in the XML tree), each subexpression is solved. If the subexpression contains the descendants-or-self operator, the nodes satisfying the subexpression will be retrieved regardless their position in the XML document. No massive traversal of the XML hierarchy will be involved. Once the set of nodes is known, we apply

```
INPUT: Regular Expression Q, Initial Document Navigator Node
OUTPUT: List of nodes satisfying the regular expression associated with Q
PROCEDURE:
    Node = Initial Document Navigator Node
    getDescendantsExploitingEncoding(RegExp)

    procedure getDescendantsExploitingEncoding(RegExp)
        PartialRegExp, Expression, RemainingRegExp =
                ExtractRegularExpressionWithoutDescendants-or-self(RegExp)
        while (PartialRegExp not empty)
            StateSet = constructQueryGraph(PartialRegExp)
            nodeList1 = getDescendants(Node, Descendants, StateSet)
            nodeList2 = solveDescendants-or-self(Expression)
            nodeList1 = filterRealDescendants(nodeList1, nodeList2)
            PartialRegExp, Expression, RemainingRegExp =
                    ExtractRegularExpressionWithoutDescendants-or-self(RegExp)
        end while
    end procedure

    procedure solve-Descendants-or-Self(Element)
        Select all Element elements from the XML Document
    end procedure

    procedure filterRealDescendants(nodeList1, nodeList2)
        for each node2 in nodeList2
            for each node1 in nodeList1
                if (Preorder(node1) < Preorder(node2) ≤
                            Preorder(node1) + Size(node1) ) then
                    Add node2 to Descendants list
                end if
            end for each
        end for each
        return Descendants
    end procedure
```

Figure 3.12: Encoding-based Processing Algorithm of Descendant Queries

the conditions of the encoding to determine which of the selected nodes are in fact descendants of the nodes selected by the previous subexpression.

The described procedure is shown in Figure 3.12. The algorithm performs the partitioning of the subexpressions and the matching between lists of nodes using the encoding condition. The *ExtractRegularExpressionWithoutDescendants-or-self* procedure is not described in detail, but basically it partitions the original regular expression into simpler subexpressions that satisfy the established constraints. The *constructQueryGraph* procedure constructs the automaton associated with the input *PartialRegExp*.

Finally, we present a concrete example which will help make things clear. Suppose we want to extract all the *authors* of papers included in *inproceedings* of *conferences* from the DBLP database. The corresponding regular path query will look like */conferences//inproceedings/author*. Applying the stated constraints to our example, the Relational Document Navigator will decompose the original expression into the following three subexpressions: e_1 = */conferences*; e_2 = *descendants-or-self(inproceedings)*, and e_3 = */author*. Note that in this case, the slash (/) at the beginning of a subexpression does not necessarily mean that the subexpression has to be evaluated from the root of the XML document. It simply means that we have to evaluate the subexpression from its initial set of nodes. Starting from the initial set of nodes (i.e., any nodes in the XML tree), we obtain all nodes that satisfy the */conferences* subexpression. As a result, we obtain a list of nodes L_1, satisfying the subexpression e_1. Then, L_1 is used as the initial set of nodes to solve the subexpression e_2. As subexpression e_2 contains a descendants-or-self operator, we have to apply the variation of the algorithm presented in Figure 3.12 . So, in the example, we search for *inproceedings* elements in the whole database, whether or not they are descendants of any node in the initial nodes list L_1. Once we have all *inproceedings* elements in the L_2 list, we match the two lists, applying the condition established by the encoding. As a result, we obtain a new node list L_3 of *inproceeding* elements that are descendants of *conferences* elements in L_1. Finally, we use L_3 as the initial set of nodes for subexpression e_3.

In Chapter V we will present several experiments using the ToX Relational mapping scheme. We will show the improvements achieved through applying the optimization proposed in the previous section. We will see that usually it is advantageous to be able to determine the ancestor-descendant relationships in constant time. Also we include in Chapter V an example of a query where the performance is worsened due to the application of this optimization. We also include a description of a possible approach to avoid detect and avoid these cases.

CHAPTER IV

Optimizing Regular Path Queries Using Materialized Views

In the previous chapter we showed how we optimize regular path queries containing the descendants-or-self operator within the toxRelational component. The next step is to optimize any regular path query, regardless of the operators it might include.

Materialized views have received significant attention in a wide variety of data management problems. Among them, query optimization has gained importance [Hal00]. In this chapter we explore whether materialized views can be used to optimize the performance of regular path queries. To answer this question some issues need to be addressed, such as how to select the materialized views, how to populate and maintain them, and how to exploit them during the query answering procedure. Next, we will address each of these issues, and we will present how we manage materialized views within the toxRelational component.

Pursuing the objective of this work, optimizing regular path queries, we decided to investigate the feasibility of using materialized views as a query optimization tool. We focus on regular path queries, while other types of queries supported by the Document Navigator (i.e., queries dealing with elements or attributes) are left for a future analysis since the index configurations presented in Chapter III can help in obtaining reasonably good performance for solving them.

4.1 Materialized Views

4.1.1 Overview

A view is defined as a query over a database. Therefore, a view definition is simply a query. A view is said to be materialized if a physical copy of the answer of its query definition is kept in the database.

As we previously stated, in the context of XML, a database is defined as a directed labelled graph. A view in this context can be seen as a set of paths satisfying a given query definition. As in this thesis we are dealing with regular path expressions, a view in this framework can be defined by a given regular expression. Therefore, we are going to deal with materialized views defined by regular expressions.

Once the materialized views are defined and populated, we have to be able to exploit their precomputed results during the query answering procedure. The problem of answering queries using views has been widely studied and can be stated as follows:

Definition 2. *Given a query Q and a set of views $\{v_1, v_2, \ldots, v_n\}$, find a representation of Q by means of the views, and then answer the query on the basis of this representation.*

4.1.2 Determining What Views to Materialize

The first problem that arises when dealing with materialized views is determining the set of views to materialize. The main goal is to materialize a set of views that can generate significant improvements in performance. One possible way to select a materialized view configuration is to study a typical workload. Obviously this configuration will be suitable for a specific application, the one that generates the workload, and not in general. The workload will reflect the types of queries that are normally executed by the application. We can define a typical workload as a set $W = \{Q_1 : f_1, Q_2 : f_2, \ldots, Q_n : f_n\}$ where Q_i is a regular path query (i.e., a regular expression) and f_i is its frequency of appearance. In the case of the toxRelational component a typical workload can be obtained tracing the use of the Document Navigator. Obviously, not all queries in the typical workload can be materialized because some space and time contraints will hold. Therefore, from the typical workload, we have to select those queries that can generate significant improvements in performance. Therefore, we have to define an objective function to maximize, taking into account all the constraints that apply. The materialized view configuration will be optimal in terms of the defined objective function. Since this problem is outside the scope of this thesis, we present an approach to tackle it as part of our future work (see Chapter VI, Section 6.2).

The optimal materialized view configuration may contain some regular expressions that are not exactly the ones in the original workload. The materialized view configuration can be composed by prefixes of the workload regular expressions. As an example, Table 4.1 shows a synthetic workload that imitates a typical one. Remember that a typical workload can only be obtained studying a real application and extracting the most frequent queries. As a result of the discovery procedure it is likely that the prefix */conferences/conference/issues/issue* will emerge as a candidate to be materialized, because it is a common prefix for all the queries in the workload.

For the rest of this thesis, we assume that we have already selected the set of regular expressions to materialize. In the current implementation, the user must

Regular Path Query	Frequency (percentage)
/conferences/conference/issues/issue/inproceedings/author	30
/conferences/conference/issues/issue/inproceedings/title	30
/conferences/conference/issues/issue/proceedings/volume	40

Table 4.1: An Example of a Workload for the DBLP Database

decide which are the regular expressions to materialize, and which XML documents stored in the repository will populate each of them (i.e., for which XML documents the materialized view will be populated with the paths that satisfy a given regular expression).

4.1.3 Deriving Materialized View Definitions

The key idea of using materialized views is to accelerate the query answering procedure. As we said, a regular path query traverses the hierarchy of the XML document, and the toxRelational component has to perform costly join operations to answer them. Therefore, one way to optimize the query answering procedure is to give direct access, in constant time, to different levels of the hierarchy. That is, to materialize the results of some join operations. To implement this simple idea, we will use materialized views.

Each materialized view will store the data satisfying a regular expression. Therefore, each tuple of the materialized view will correspond to a path (that satisfies a regular expression) instead of to an edge (as in the case of the *Edge* table of the ToX Relational mapping scheme). In Figure 4.1 we present a sample of the content of the materialized view storing the regular expression */conferences/conference/issues/issue/inproceedings/author* for the XML document shown in Figure 3.2. A particular *author* element can be selected directly (i.e., accessing only one tuple) instead of descending through the hierarchy of the XML document.

conferences	conference	issues	issue	inproceedings	author
2	3	4	5	6	7
2	3	4	5	6	9
2	3	4	5	6	11

Figure 4.1: Materialized View Content Example

Even though we show only the OID values for each materialized element, in the current implementation additional data is stored.

The definition we propose to use for the materialized views is similar to the one presented by Kemper and Moerkotte in their "Access Support Relations" work (ASR) [KM92] developed in the context of object databases. In this context one of the more performance-critical operations is object accesses along reference chains leading from one object instance to another. The basic idea of ASR is to maintain separate structures to redundantly store those objects that are frequently traversed in database queries.

Access Support Relations deals with simple path expressions, that is, path expressions containing only the concatenation operator. Our approach is more general in that respect because it supports complex regular expressions. This generalization makes the process of defining the materialized views more complex. Therefore, we have taken the basic ideas of Access Support Relations, and we have extended them to support complex regular expressions.

In order to support complex regular expressions, we developed a set of heuristic rules to derive the materialized view definitions. These rules were inferred using performance tests results. In order to understand the heuristics, we have to define the concept of single step and complex step. Given a regular path expression $l_1.l_2 \ldots l_n$, a single step is any l_i that appears in it, and a complex step is one or more single steps combined by operators. The idea of the procedure is to decompose the regular expression E, into single steps e, following the precedence of operators. Based on all the decomposed steps, the procedure recursively generates the definition of a view to hold the original expression E. The recursion will be managed by the precedence of operators, obeying the following heuristics rules:

1. Each single step e will generate an attribute in the materialized view.

2. Two complex or single steps combined by the concatenation operator will generate a new view combining the attributes of the original materialized views. Example: consider the regular expression $E = a/b$. The definition of the materialized view for storing E will be $V_1 = [Attr_a, Attr_b]$. The first attribute will store a elements, while the second attribute will store b elements, each of whose parent is an a element.

3. Two complex or single steps combined by the alternative or disjunction operator will generate a view with the same definition as the one with maximum number of constitutents between the materialized views generated by the input complex or single steps. Example: consider the regular expression $E = a|(b/c)$. By applying the previous heuristic rules to the expression E, two partial materialized view definitions are obtained. Element a will be stored in $V_1 = [Attr_a]$, while the subexpression b/c will be stored in the $V_2 = [Attr_b, Attr_c]$. Applying the current rule to combine both subexpressions will generate a new view $V_3 = [Attr_1, Attr_2]$, where $Attr_1$ will hold a or b elements, and $Attr_2$ will hold $null$ or c elements (depending on the value of $Attr_1$).

4. Complex or single steps combined by the descendants-or-self operator will generate a view with the same definition as the one generated by the original complex or single step to which the operator was applied. Note that no additional attributes will be generated to hold intermediate steps involved in the

descendants-or-self operator.

We implemented the set of heuristic rules defined above using a regular expression parser. The parser takes care of the precedence of operators. The materialized views are created recursively using the information exposed by the parser. As a result, a set of materialized views will be generated to store paths satisfying the different regular expressions to be materialized.

To clarify the above heuristic rules, we present a complete example. Consider the regular expression $E = a/b//c/(d \mid (e/f))$. In Figure 4.2 we show a graphical representation of the heuristic rules applied to E. The representation must be read by levels from top to bottom. Each figure in a particular level shows the intermediate definition of the materialized view being created. Each box in the figure corresponds to an attribute in the materialized view. Each box shows the symbols that the corresponding view attribute can hold. At the lowest level of the graphical representation we can observe the derived definition for the materialized view. This last definition is the one that will be created by the toxRelational component to store the regular expression E. In the example, the derived materialized view definition has five attributes that are sufficient to hold the regular expression E. Note that a particular attribute can be shared by more than one expression (as in the case of the d and e). In the example, when a d element populates the fourth attribute of the materialized view, the fifth attribute of the materialized view will be set to null.

Applying the presented heuristic rules, it is possible to derive the same materialized view definition for different regular expressions. To solve this problem, additional information is stored for each regular expression being materialized. (Refer to Section 4.2 for details on this information).

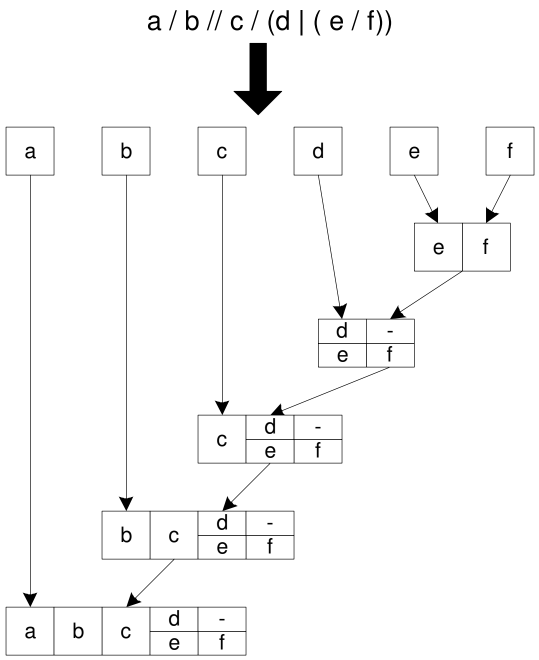

Figure 4.2: Heuristic Rules - Example Graphical Representation

4.1.4 Populating Materialized Views

Once we have defined the materialized views, we have to populate them. To perform this operation we simply have to retrieve all paths satisfying the query definition of the materialized view from the ToX Relational mapping scheme. The population itself is a straightforward task, but it can be costly. The procedure followed by the toxRelational component to populate the view is the same as the one used to solve a query. In order to exploit the materialized views in answering a query, we have to make sure that they are kept up-to-date with respect to the original XML data. The problem of keeping materialized views up to date is a field that has been studied extensively [ZGM99, AMR+98, LD00].

4.2 Managing Materialized Views in the toxRelational Component

In the current implementation the set of regular expressions to materialize must be defined by the user. Next, they are analyzed by the regular expression parser with the embedded heuristic rules. As output a set of materialized view definitions will be derived. Then, the materialized views are created and populated using the data of the associated XML documents stored in the repository.

As we explained, each materialized view has its own dynamically generated definition, and it is populated for particular XML documents. The procedure that derives the materialized view definition is executed only once per regular expression in the optimal materialized view configuration. The term dynamical is used to emphasize the procedure that derives the materialized view definition based on the heuristic rules. When a query is entered, the toxRelational component must determine the set of materialized views that can be relevant for solving it. Therefore, the Tox Relational catalog presented in Chapter III has been extended to store information about: materialized view definitions, optimized XML documents (i.e., XML documents for which some paths were materialized), and statistical information about the materialization procedure and the views (i.e., number of tuples in the materialized view, time consumed by the population procedure, etc.). With all this information in hand, the toxRelational component is able to dynamically generate the SQL queries necessary to access the materialized views if it is required. Finally, statistics about materialized views will allow the toxRelational component to decide whether the use of materialized views will lead to a reduction of the query response time. This decision making process should be based on a cost model, which is part of our future work. The complete set of statistics relevant for the decision making process is yet not known.

In Figure 4.3 we show the entity relationship diagram of the extended ToX

Relational catalog. We show the entity relationship diagram for clarity, and to express the relationships between the different catalog tables. This diagram follows the IDEF1X notation [Bru95] where squared-corner tables represent independent relational entities, and rounded-corner tables represent dependent entities that inherit from other relations part of their key as foreign keys (shown as FK in the diagram). All the relationships shown in the diagram are defined to be one to many (one tuple in one relation to many tuples in the other relation).

As can be seen in the entity relationship diagram, several tables compose the Tox Relational catalog. The *Workload Catalog* table stores information about different workloads input to the component. The support for multiple workloads is based on the fact that different XML documents, possible belonging to very different domains, can be stored in the ToX repository at the same time. Those XML documents will have different workloads for which the user may want to optimize their query answering procedure. For example bibliographic XML documents and auction XML documents can be stored in the same database, but they support very different workloads. Therefore, the catalog is prepared to manage several workloads. Each workload will lead to the generation of a set of materialized views and these views will be populated by different XML documents. The *Plans Catalog* table stores information about each regular expression selected to be materialized. Notice that each regular expression is accompanied by a prefix regular expression which indicates the starting point for the materialization. If the prefix regular expression is empty, then the materialization for a particular XML document will start from its root element. On the other hand, if the prefix regular expression is not empty the materialization for a particular XML document will start from those objects satisfying the prefix regular expression. Each regular expression stored in the *Plans Catalog* can be associated with one or more materialized views. This information plus some statistical information about each of those views can be found in the *Execution Plans* catalog table, while their definition can be found in the *View Definition* catalog table.

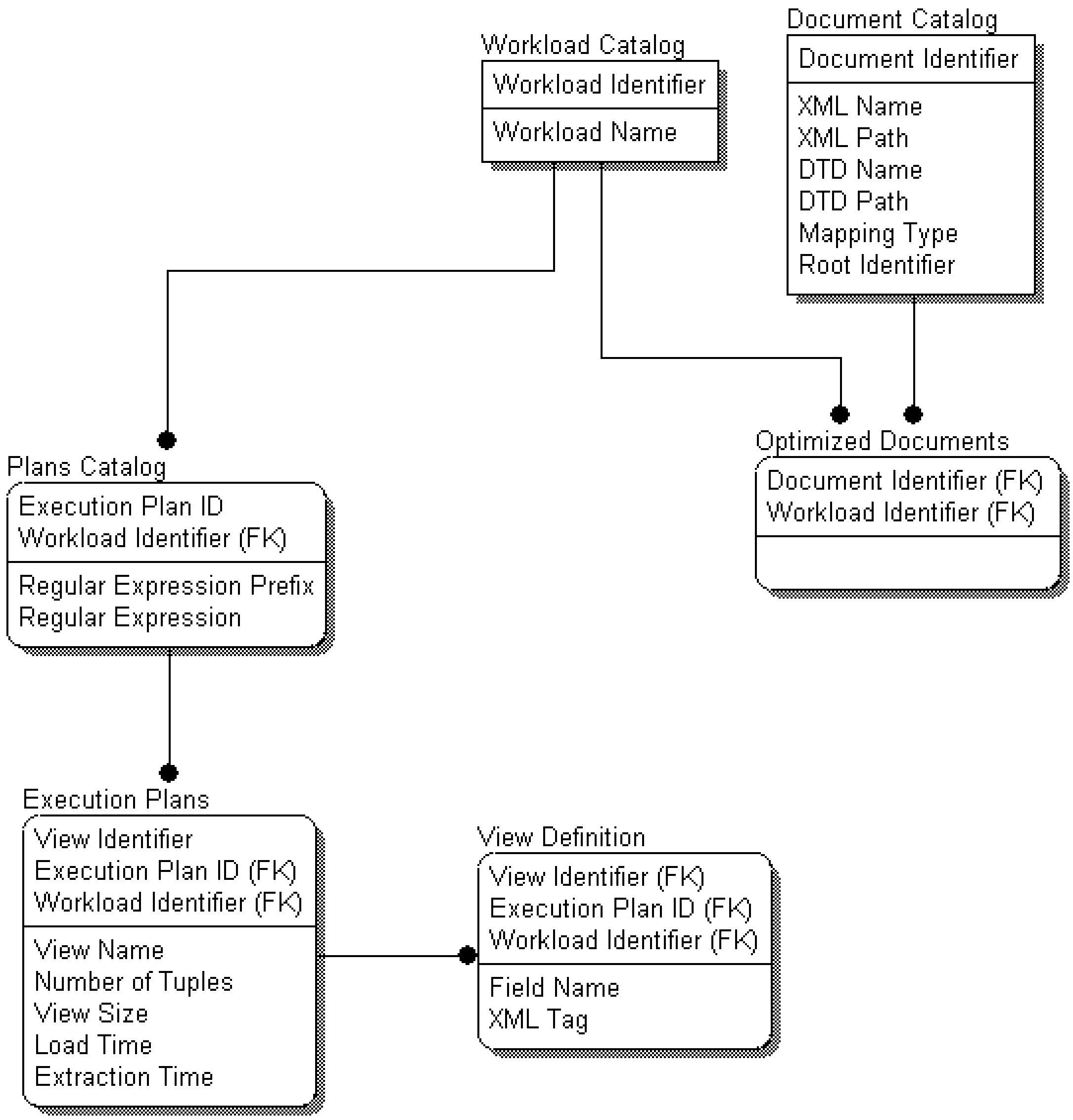

Figure 4.3: Entity Relationship Diagram of the ToX Relational Catalog

The statistical information currently stored in the *Execution Plans* catalog table are some examples of possible useful statistics: number of tuples, view size (in blocks), load and extraction time. The *View Definition* catalog table contains the association between each attribute in a materialized view and one or more element names in the XML document. Finally, the *Optimized Documents* table indicates which XML document was optimized for which workload (i.e., specifies that some paths of the named XML documents have populated some materialized view). Therefore, for those XML documents in the *Optimized Documents* catalog table, there exists a chance that the query answering procedure can use precomputed results stored in some materialized view.

As we pointed out earlier, materialized views can be exploited only in the *getDescendants(Regular Expression)* method of the Document Navigator. Now we will show how the toxRelational component determines which views can be relevant for solving a particular regular expression. The toxRelational component checks if the XML document being queried was optimized for some workloads accessing the information in the *Optimized Documents* catalog table. Each of those workloads may contain several regular expressions. In this way, all the information is available so that the toxRelational component can exploit it. In Section 4.3 we will show how this information is exploited.

4.2.1 Limitations

As stated, a prefix regular expression accompanies each regular expression in the workload indicating the starting point for the materialization. Although any type of prefix is supported by the toxRelational component, the cases where a materialized view was created based on a non-empty prefix needs further study, because it is not always clear if we can safely use that view to answer a query. As an example suppose we have materialized *title* elements at any level in the XML hierarchy with */conferences* prefix from the DBLP database. The materialized view to store the *//title* elements will not preserve the information about the ancestors of the materialized elements due to heuristic rule 3 presented earlier in this chapter. So, in this case, only *title* elements will be stored in the materialized view. Now, suppose we want to use the materialized view to solve the query */conferences//inproceedings/title*. Although all titles in the result of this query are stored in the view, we cannot use the view during the answering procedure because we do not have enough information to determine which of the *title* elements stored in the view are descendants of */conferences//inproceedings* elements. Another problematic example of a non-empty prefix is */conferences//(inproceedings | proceedings)/title*. In this case, we cannot determine from those *title* elements stored in the view which ones correspond to *inproceedings* and which ones correspond to *proceedings*. The absence of the containment relationship in the view generates the necessity of accessing the original repository if we want to check the relationship between two elements. These additional accesses to the original mapping tables to check the containment relationships may degrade the performance in such a way that the improvements achieved from using the materialized view optimization are lost. One possible solution for this problem is to store with each materialized element (i.e., each materialized tuple) a reference to the instance path where it belongs in the original XML document. Another approach is to exploit structural information to determine containment relationships between elements. This approach can be applied only if the names of the elements in the XML document are unique (i.e., there do not exist different elements with the same name at different levels of the XML hierarchy). Since we have not implemented any of these approaches, the current implementation only uses the materialized views in the cases where the prefix is empty or where the prefix is non-empty but it is exactly contained in the regular path query. In those cases where the information is not sufficient to safely answer the query, the materialized view optimization is not used (as in the example above).

4.3 Exploiting Materialized Views within the Relational Document Navigator

The next step is to exploit the materialized views within the Relational Document Navigator. We need to detect if a particular query can be rewritten using a set of materialized views. Recalling that each view is defined by a regular expression, the method for rewriting queries using views requires, in fact, a technique for rewriting a regular expression using a set of regular expressions. The following problem statement defines the problem of query rewriting using views:

Given a regular expression E_0 and a finite set $\varepsilon = \{E_1, E_2, \ldots, E_k\}$ of regular expressions over the alphabet Σ, reformulate, if possible, E_0 by a suitable combination of $E_1, E_2, \ldots, E_k$.

Calvanese et al. [CGLV99] defined an algorithm for rewriting regular expressions using a set of regular expressions. We have implemented this algorithm as the basis for exploiting the materialized views within the toxRelational component. Next we will describe the proposed algorithm.

Let ε be the set of view definitions, which are regular path expressions defined over the alphabet Σ. We assume that, associated with the finite set ε, we always have an alphabet Σ_ε containing exactly one symbol per regular expression in ε, and we denote the regular expression associated to the symbol $e \in \Sigma_\varepsilon$ with $re(e)$. For our purpose, the alphabet Σ will be formed by the element names present in the XML document, while the alphabet Σ_ε will contain one symbol V_i per regular expression i defining a view.

Our goal is to determine the best rewriting for a given query. In the work of Calvanese et al. [CGLV99], the best rewriting is called maximal. Maximal rewritings are the ones that capture in the best possible way the language defined by the original regular expression E_0. A maximal rewriting can be an exact rewriting if it is the one that defines the same regular language as E_0, can be only maximal if the rewriting defines the maximal language contained in the one defined by E_0. Formally,

Definition 3. *A rewriting R of E_0 wrt ε is Σ-maximal if for each rewriting R' of E_0 wrt ε we have that $exp_\Sigma(L(R')) \subseteq exp_\Sigma(L(R))$. A rewriting R of E_0 wrt ε is Σ_ε-maximal if for each rewriting R' of E_0 wrt ε we have that $L(R') \subseteq L(R)$.*

Theorem 1. *Let R be a rewriting of E_0 wrt ε. If R is Σ_ε-maximal then it is also Σ-maximal.*

This result is important because, as we will see in the next section, it is the basis for proving the correctness of the rewriting algorithm.

4.3.1 Deriving a Rewriting

Given E_0 and ε, we are interested in deriving a maximal rewriting of E_0 wrt ε. The procedure described by Calvanese et al. [CGLV99] to derive a maximal rewriting can be seen in Figure 4.4.

The following theorem defines the nature of the rewriting achieved by the rewriting algorithm [CGLV99]:

Theorem 2. *The automaton R_{ε,E_0} is a Σ_ε-maximal rewriting of E_0 wrt ε.*

Combining the results from Theorem 1 and Theorem 2, Calvanese et al. conclude that the algorithm finds a maximal rewriting wrt to Σ.

INPUT: Regular Expression E_0, set ε of regular expressions
OUTPUT: Automaton R_{ε,E_0} of the rewriting

PROCEDURE:

1. Construct a deterministic automaton $A_d = (\Sigma, S, s_0, \rho, F)$ such that $L(A_d) = L(E_0)$.
2. Define the automaton $A' = (\Sigma_\varepsilon, S, s_0, \rho\prime, S - F)$, where $s_j \in \rho'(s_i, e)$ if and only if $\exists w \in L(re(e))$ such that $s_j \in \rho^*(s_i, w)$.
3. $R_{\varepsilon,E_0} = \overline{A'}$, (i.e., the complement of A').

Figure 4.4: Algorithm for Deriving a Maximal Rewriting

Once the maximal rewriting is determined, we have to check if the maximal rewriting captures exactly the same language defined by E_0, and therefore is exact. The following definition states the condition that a rewriting has to satisfy in order to be an exact rewriting.

Definition 4. *A rewriting R of E_0 wrt ε is exact if $exp_\Sigma(L(R)) = L(E_0)$*

We show in Figure 4.5 the algorithm proposed by Calvanese et al. [CGLV99] to determine whether R_{ε,E_0} is an exact rewriting of E_0.

Formally, the algorithm just described can be stated as follows:

Theorem 3. *The automaton R_{ε,R_0} is an exact rewriting of E_0 wrt ε if and only if $L(A_d \bigcap \overline{B}) = \emptyset$ (B is constructed following the procedure in algorithm in Figure 4.5.)*

A detailed exposition of the complete rewriting procedure can be found in the work of Calvanese et al. [CGLV99]. Also, the authors have analyzed the complexity of the described procedures. They proved that generating a maximal rewriting, and verifying whether the rewriting is exact can be done in 2EXPTIME. Both the algo-

INPUT: R_{ε,E_0}
OUTPUT: Is the rewriting EXACT or MAXIMAL?
PROCEDURE:

1. Construct an automaton B over Σ that accepts $exp_\Sigma(L(R_{\varepsilon,E_0}))$ as follows. First construct the automaton A_i such that $L(A_i) = L(re(E_i))$ where $i = 1, \ldots, k$. Then, obtain B by replacing each edge labeled by e_i in R_{ε,E_0} by a fresh copy of A_i, identifying the start state of the fresh copy with the source of the edge, and the accepting state with the target of the edge. Observe that, since R_{ε,E_0} is a rewriting of E_0, $L(B) \subseteq L(A_d)$.
2. Check whether $L(A_d) \subseteq L(B)$, that is, check whether $L(A_d \bigcap \overline{B}) = \emptyset$.

Figure 4.5: Algorithm to Check Whether a Rewriting is Exact or Maximal

rithm for obtaining a deterministic automaton from a nondeterministic one, and the algorithm for complementing an automaton are exponential, so the overall complexity is due primarily to the manipulation operations over the automata.

4.3.2 Implementing the Rewriting

The information concerning the materialized views is kept in the ToX Relational catalog. To exploit the views during the query answering procedure, we have implemented the algorithms described in the previous section.

Given a regular path query Q, before applying the rewriting algorithm we have to check whether Q is one of the existent materialized views. We do not check regular expression equivalence, but we search for a perfect match. A perfect match will exist in the case that the given regular path query is identical to one of the materialized regular expressions. Therefore, if the regular expression is equivalent to one of the materialized views but is not identical, we do nothing. As part of our future work we will add a containment algorithm to determine equivalence of regular expressions.

Given that the materialized views were created selecting regular expressions from a workload, it is probable that some query Q has been materialized. If a perfect match is found, the result of Q can be retrieved using only information stored in the associated materialized view. If a perfect match is not found, the rewriting algorithm in Figure 4.4 has to be executed. The input for the rewriting algorithm will be the regular path query Q as E_0 and the set ε will be composed of all materialized views associated with the document being queried. This information is retrieved from the ToX Relational catalog as explained in Section 4.2. The rewriting algorithm will generate the automaton E_{ε,E_0} of a maximal rewriting for Q with respect to the set of views. The maximal rewriting generated by the algorithm might be either empty or exact. If the rewriting is empty then no materialized view can be used in the query answering procedure, and therefore the original regular path query Q has to be posted to the original repository. If the rewriting is exact, then the data stored in a materialized view or in a combination of materialized views can be used to answer the query Q. In this case, no additional accesses to the mapping are needed, and the answer will be valid whenever the materialized views are up-to-date. Finally, if the maximal rewriting is neither empty not exact, then only some prefix of the query Q can be retrieved from the materialized views. Therefore, we retrieve information from the materialized views, and we use this information as the set of initial nodes for retrieving the rest of the data from the original mapping tables. To retrieve the remaining data from the original mapping tables, we have to detect which are the parts of the query that remain unsolved. The implementation uses the automaton of query Q, and the automaton of the maximal rewriting to detect the unsolved portions, traversing them simultaneously. In Figure 4.6 we show the procedure just described. This algorithm corresponds to the new version of the *getDescendants(Regular Expression)* presented in 3.12.

The materialized views created by the toxRelational component satisfy the *Exact View Assumption*, which means that we guarantee that the extensions of the

INPUT: Regular Path Query Q
OUTPUT: Answer of query Q

PROCEDURE:
    $Node$ = Initial Node of the XML-graph (any node)
    getDescendantsExploitingMV(Q)

procedure getDescendantsExploitingMV(Q)

    $ListOfViews$ = getViewsForThisDocument()
    if ExactMatchingExists(Q, $ListOfViews$) **then**
        An exact materialization exists, then answer Q with data retrieved from the corresponding materialized view
        return Answer
    end if

    $Rewriting$ = applyRewritingAlgorithm(Q, $ListOfViews$)

    if $Rewriting$ is empty **then**
        No rewriting exists.
        Apply algorithm showed in Figure 3.12
        return Answer
    end if

    if $Rewriting$ is exact **then**
        An exact rewriting exists. The result can be retrieved from a unique view or a combination of views.
        return Answer
    end if

    A maximal rewriting exists.
    Retrieve prefixes from view or views
    Retrieve the rest of the answer from the original data:
        - traverse simultaneously:
            1. the automaton of query Q, and
            2. the one of the maximan rewriting
        - Detect unsolved portions of the query.
end procedure

Figure 4.6: Algorithm for Exploiting Materialized Views

materialized views are exactly all tuples in the database satisfying the view query definitions at view materialization time. If an update to any document being queried occurs, then we cannot be sure that the views will remain up to date.

Next, we present an example to clarify the rewriting procedure, showing in some cases the automaton representation. Using the same data set presented in Chapter I we show a set of queries, a set of materialized views, and some possible rewritings.

Queries

- $Q_1 = conferences/conference/issues/issue/inproceedings/author$
- $Q_2 = conferences/conference/issues/issue/inproceedings/title$
- $Q_3 = conferences/conference/issues/issue/(inproceedings \mid proceedings)$
- $Q_4 = conferences/conference/issues/issue/inproceedings/(title \mid author)$

Views

- $V_1 = conferences/conference/issues/issue$
- $V_2 = conferences/conference/issues/issue/inproceedings/title$
- $V_3 = conferences/conference/issues/issue/inproceedings$
- $V_4 = conferences/conference/issues/issue/proceedings$

Possible Rewritings

- V_1 is a maximal rewriting of Q_1, Q_2 and Q_3, because V_1 is a common prefix for all the mentioned queries. So, to answer these queries, first we have to retrieve data from V_1. The direct access to V_1 saves the query answering procedure from having to traverse four levels of the XML document hierarchy. The set of nodes obtained from V_1 will be used as initial nodes for retrieving the complete answer of the query from the original repository. In the case of Q_1 we have to start from the *issue* elements obtained from V_1, and from those elements query the original repository for all the *inproceedings/author* child elements. In Figure 4.7 we show the automaton representation of query Q_1 indicating with a dashed box those parts of the query that remained unsolved after the rewriting.

- V_2 is an exact rewriting for Q_2, and therefore no extra accesses are needed. View V_2 can also generate a maximal rewriting for Q_4. In Figure 4.8 we show the automaton representation of query Q_4 indicating the part of the query that needs to be retrieved from the original repository with a dashed box. Note that to solve the remaining part of the query we cannot begin from the *inproceedings* elements extracted from V_2, because it may be the case that a particular *inproceedings* element does not contain a *title* element, and therefore is not included in view V_2, but it contains an *author* element and therefore it has to be in the answer of Q_4. As an example of this situation consider the subset of the DBLP database shown in Figure 4.9. The *inproceeding* element shown in the example will not be included in the materialized view V_2 because it does not have a *title* element as a child. However, this element has to be in the answer of Q_4 because it contains *author* elements as children.

- V_3 alone can be a maximal rewriting for all queries, and together with V_4 (i.e., $V_3 \mid V_4$) can be an exact rewriting for Q_3. In the case of the exact rewriting, data retrieved from views V_3 and V_4 form the answer to query Q_3.

- V_4 alone is an empty rewriting for any of the given queries.

One weak point of the rewriting algorithm is the fact that, if more than

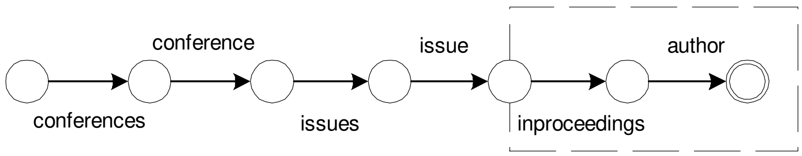

Figure 4.7: Automaton Representation for Query Q_1

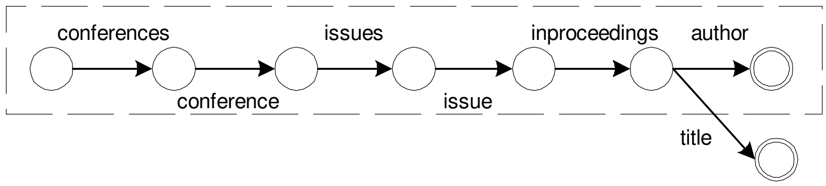

Figure 4.8: Automaton Representation for Query Q_4

```
<conferences>
    <conference key="LICS">
        <issues>
            <issue key="2001">
                <inproceedings key="AlonMNSV01">
                    <author>Noga Alon</author>
                    <author>Tova Milo</author>
                    <author>et al.</author>
                    <pages>421-430</pages>
                    <year>2001</year>
                    <booktitle>LICS</booktitle>
                </inproceedings>
            </issue>
        </issues>
    </conference>
    ...
</conferences>
```

Figure 4.9: Example Extracted From the DBLP Database

one possible rewriting exists, then it is not able to choose always the best one. In fact, sometimes it combines several rewritings into a single one. For example, suppose that V_i = *conferences* and V_j = *conferences/conference*. Let Q_k = *conferences/conference/issues*. It is possible that the rewriting algorithm generates a maximal rewriting combining V_i and V_j while in fact, V_j alone is a rewriting. Therefore, two interesting issues to pursue as part of our future work are: (i) defining a way to determine from a set of valid rewritings which one is the best, and (ii) finding a way to determine whether a generated maximal rewriting is redundant, as in the example above.

In this chapter, we have explained the framework we defined and developed for using materialized views as an optimization tool. We presented some of the issues that are important when dealing with materialized views. We presented the problem of selecting an optimal materialized view configuration based on workload information and we leave the complete analysis as part of our future work. Then, we described some heuristic rules inferred from a performance study and we explained how the toxRelational component applies them to automatically derive the materialized views definitions (based on the regular expressions conforming the optimal materialized view configuration). Finally, we presented an algorithm by Calvanese et al. for rewriting regular expressions, describing how we implemented it and used it within the toxRelational component. Also several examples were presented to clarify the entire rewriting procedure. Some performance results obtained by the use of the materialized view optimization can be found in Chapter V.

CHAPTER V

Experiments

In this chapter, we present a performance study of the ToX Relational mapping scheme and the optimizations proposed in this thesis. In order to study the behavior of the various optimizations, we carried out a series of performance experiments. We are interested in the evaluation of regular path queries, and we use the query processing time as performance metric. Also, we are interested in the query processing time when some characteristics of the data sets are modified, such as the total size, or the degree of fanout of the nodes in the data set.

5.1 Experimental Setup

5.1.1 Experimental Platform

We implemented the toxRelational component in Java, and we used Java Database Connectivity (JDBC 2.0) technology to access the relational database. We used the Xerces Java Parser (Xerces-J Version 1.4.1) to parse and process the XML documents. All the experiments presented herein were conducted on a Pentium III 533 MHz PC with 256 MB RAM with 20 GB hard disk running Windows 2000.

5.1.2 DBMS

The commercial DBMS used was IBM DB2 V7.1 for Windows. We performed an extensive study of the database configuration parameters to improve the overall performance. Next we identify the database configuration parameters that we modified, together with their definitions and their values before and after modification.

1. DBHEAP: specifies the size, in pages (4KB), of the database heap that is used to hold control information on all open cursors accessing the database. The default value was 600 pages and we increased it to 2000 pages.
2. STMTHEAP: specifies the heap size, in pages (4KB), that can be used for compiling SQL statements. The default value was 2048 pages and we increased it to 4096 pages.
3. APPLHEAPSZ: specifies the size, in pages (4KB), of the application heap that is available for each individual agent. The default value was 128 pages and we increased it to 4096 pages.
4. LOGFILSIZ: specifies the amount of disk storage, in pages (4KB), allocated to log files for data recovery. The default value was 250 pages and we increased it to 4000 pages.

The database has areas of storage into which database pages are temporarily read and modified. These areas are known as buffer pools. The purpose of the buffer pools is to improve the database system performance, because data can be accessed much faster from memory than from disk. The buffer pools are used as intermediate storage for already accessed pages. Different policies are used for replacing the pages stored in the buffer pools, trying to retain the pages expected to be re-read soon. We set the total size of the buffer pools in our system to 32 MB.

The ToX Relational mapping scheme, the ToX Relational catalog, and the index configuration schemes described in Chapters III and IV were created and populated in the relational database.

5.1.3 Data Set Description

We conducted our experiments using two real data sets and one synthetic data set. The real data sets contain information from very different domains and with very different structure. The synthetic data set follows the structure defined by the XMark Benchmark [BCF+01] and was generated using ToXgene [BMK02]. ToXgene is a template-based generator for complex, semantically-correlated collections of XML documents. The data generation process in ToXgene is based on a conceptual description of the data to be generated (the templates). Next, we summarize the data sets characteristics:

- Evidence-Based On Call (EBOC) [Liv]. This data set corresponds to a series of recommendations about health issues. This data set is highly structured and nested.
- Digital Bibliography and Library Project database (DBLP) [Ley01]. This data set corresponds to a subset of the original DBLP database. It includes conference papers, and it is a classical example of a bibliographic database containing nested data.
- Synthetic XMark-like data set generated with ToXgene. This data set contains information about auction sales, based on the XMark benchmark defined by Busse et al. [BCF+01].

	EBOC	DBLP	XMark-like
XML Document File Size (MB)	3	62	0.3 to 1.5
Levels of Nesting	15	6	6
Number of Tuples in the mapping scheme	144024	2994301	1660 to 24760

Table 5.1: Data Sets Characteristics

Some quantitative characteristics of the three data sets are shown in Table 5.1. As we can see, the mix of data sets covers a wide range of sizes and levels of nesting.

The basic template used to generate the XMark-like data set was taken from the ToXgene distribution. Maintaining the structure imposed by the XMark benchmark, we created several data sets varying the fanout of nodes in the output XML document. The distribution of fanouts followed an uniform distribution. Also we customized some probability distributions from the original template. In Appendix C, we present a subset of the ToXgene template used for generating the synthetic data sets.

5.1.4 Experimental Procedure

We selected the queries for the experiments in order to test various aspects of regular path queries without predicates. We performed 10 repetitions of each query. As we already said, the relational database uses buffer pools as caching mechanism, storing in main memory pages that are expected to be re-read soon. The initial period of time that the database spends loading pages into the buffer pools is known as the warm up period. During this period the response time of the database declines with respect to the normal response time. Therefore, we decided to show the first execution time separately (calling it the worst case scenario). We also present the average execution time of the other nine repetitions (calling it the best case scenario.

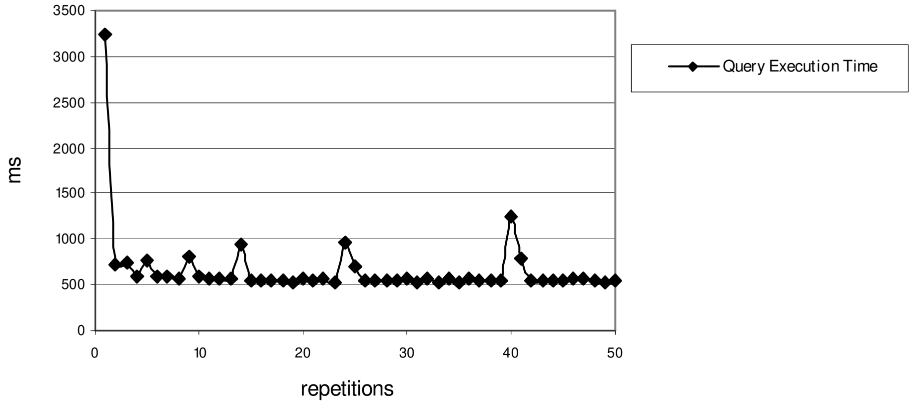

Figure 5.1: Influence of the Database Caching Procedure in the Query Processing Time

Therefore, the actual response time of the toxRelational component will be around those values, depending on the status of the database engine.

To demonstrate the differences in performance during the warm up period and the normal operation periods, we chose a query and we performed 50 repetitions. In Figure 5.1 we show the query processing time for each repetition. As expected the first execution consumed more time than the subsequent repetitions. Also, note that the query processing time stabilized after some repetitions, and that once in a while one repetition consumed more than the minimum processing time. A possible explanation for this behavior is that the buffer pools got full, and hence the page cleaner procedure has to be executed.

5.2 Bulkloading Performance Evaluation

Our motivation and main concern in the design and development of the toxRelational component was to improve the performance of regular path queries over relational databases. Here we also discuss some common measures used to compare

XML-to-relational mappings: the total time consumed by the loading procedure, and the size of the resulting database.

5.2.1 Loading Times

The loading time, also known as the bulkloading time, is totally dependent on the implementation. As we previously said, we implemented the toxRelational component using Java and JDBC. We are aware of the existence of several ways of performing updates in a relational database using JDBC methods: single SQL statements, prepared SQL statements, batch updates, embedded SQL statements, and external load utility. Any of these methods could be used in the toxRelational component. Hence we decided to implement all of them to determine which is the most efficient. In the particular case of using Insert SQL statements, the main difference between the methods is the precise moment the statement is compiled and executed. This detail leads to considerable differences in performance. For single SQL statements, each SQL is compiled and executed independently. When using prepared SQL statements, the structure of the insert SQL statement is compiled only once, and then, the precompiled version is executed once per insert. The batch update permits execution of several insert SQL statements at once, but each insert is compiled and executed as in the case of single SQL statements. Using embedded SQL statements is simply a way to embed SQL directly into Java code. It allows the programmer to manipulate low-level data structures that are more efficient in terms of performance. Finally, the external load utility allows the loading of all tuples at once, from precomputed files.

We performed our loading experiments using the DBLP data set. We took a subset of the original data set (60 MB) with size 3 MB because the procedures require a huge amount of main memory. In fact, almost all memory consumed by the

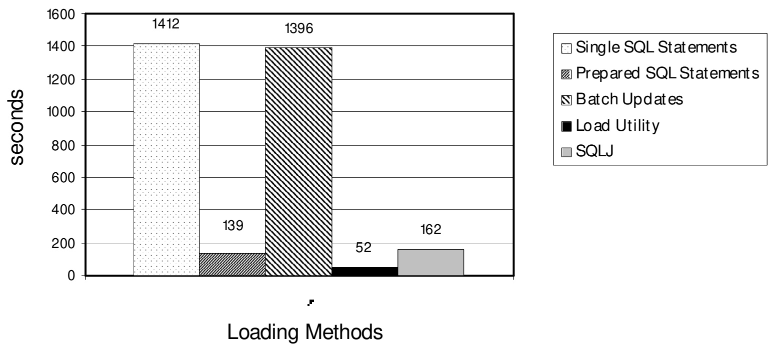

Figure 5.2: Comparison between the Different Loading Methods

loading procedure is used by DOM. We plan to re-implement the loading procedure using SAX, which is much more efficient in terms of memory usage, since the whole XML tree is not constructed in memory. Note that the original 60 MB file used for the query performance experiments was processed in a different machine with 2 GB of main memory. Then, precomputed flat files were generated and loaded into the database using the external load utility.

The results obtained from loading the XML document representing the DBLP database using each of the different methods are shown in Figure 5.2. From the results, we can conclude that, among all JDBC methods, the external load utility, the prepared SQL statements and the SQL embedded in Java are the ones that achieve the best overall performance. However, while the implementation of the first method is intricate (i.e., we have to handle transactions carefully), the implementations using prepared SQL statements or embedded SQL are straightforward. In spite of this simplicity, embedding SQL statements into Java code requires the use of extra libraries, and due to that, the code is not easily portable. For all these reasons, we recommend the use of prepared SQL statements when Java and JDBC are being used for the implementation.

	Edge Table	Leaf Table	Attribute Table
Number of tuples	113282	102866	21266
Average Row Length (bytes)	87	87	1991
Size in blocks (4K)	2335	2499	355
Number of indices	4	3	1
Size in blocks of indices (4K)	5600	3854	232

Table 5.2: ToX Relational Mapping Information about the DBLP Data Set

5.2.2 Database Size

In Table 5.2 we present some characteristics of the relational instance resulting from the mapping of the DBLP database into the ToX Relational mapping scheme. We have to add the length of the VARCHAR fields to the average row length that appears in the table. The size of the generated relational instance (without indices) is approximately equal to the size of the original XML document.

5.3 Individual Queries

We chose the queries trying to cover many aspects of regular path queries. We included queries using the descendant-or-self operator, branching queries including the alternative or disjunction operator, the conjunction operator, and different combinations of the mentioned operators. We used the EBOC and the DBLP data sets for these experiments. The objective of these experiments is to show how the optimizations proposed in this thesis performed. With the DBLP data set, we also performed some experiments varying the size of the data set to verify the behavior of the query answering procedure.

5.3.1 EBOC Data Set

In order to understand the queries and their results, it is important to have a general idea about the structure of the underlying data set. In Figure 5.3 we show the portion of the EBOC DTD relevant to our experiments. The DTD reveals the nesting and depth of the EBOC data set. In the DTD graph, we include the element and subelement names as labels accompanied by an operator. The operator represents the regular expression present in the original DTD. The operator $*$ means that 0 or more repetitions of the element can appear, the operator $+$ reflects that 1 or more repetitions of the same element can occur, and the operator ? means that the element is optional.

Next, we will present the set of queries and materialized views used in the experiments, and an analysis of the results.

Queries

- Q_1 = *guide04/specialties/specialty/specialty-type/us-english*
- Q_2 = *guide04/content/prognosis/sub-section/content-section/instruction/statement/p*
- Q_3 = *guide04/specialties/specialty/sub-specialty/(us-english | uk-english)*
- Q_4 = *guide04/specialties//sub-specialty/us-english*
- Q_5 = *guide04/content/prognosis/sub-section//instruction/statement*
- Q_6 = *guide04/title//topic/(us-english | uk-english)*

Queries Q_1 and Q_2 are simple queries of different depths. The EBOC data set contains a small number of instances of paths "*guide04/specialties/...*" compared with the number of instances of paths "*guide04/content/...*". Query Q_3 introduces

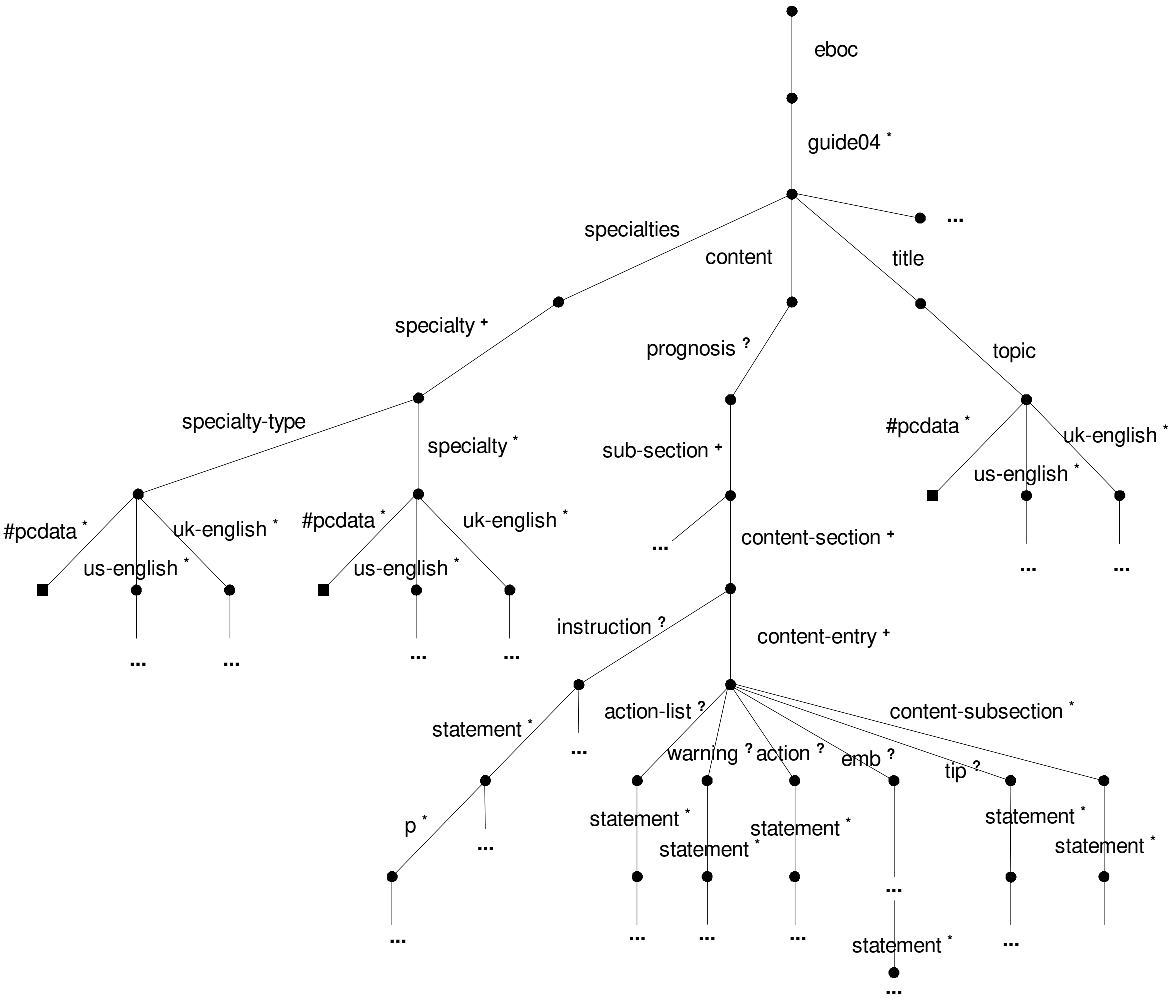

Figure 5.3: EBOC Data Set DTD

the alternative or disjunction operator over the same */guide04/specialty/...* sparsely populated subtree. Queries Q_4 and Q_5 introduce the descendants-or-self operator, while Q_6 combines the descendants-or-self operator with the alternative operator. The last three queries contain the descendants-or-self operator at different levels in the expression to show the influence of the position of this operator on the response time. The mix of queries is general enough to cover a variety of interesting cases.

Materialized Views

We have deliberately selected the regular path queries to be materialized in order to show different types of rewritings when applying the materialized view optimization procedure. However, in a real application, a workload analysis has to be performed to determine the best materialized view configuration.

- V_1 = *guide04/specialties/specialty/specialty-type*
- V_2 = *guide04/content/prognosis/sub-section*
- V_3 = *guide04/specialties/specialty/us-english*
- V_4 = *guide04/specialties/specialty/uk-english*
- V_5 = *guide04/specialties//sub-specialty*
- V_6 = *guide04/content/prognosis*
- V_7 = *guide04/title//topic/us-english*
- V_8 = *guide04/title//topic/uk-english*

	Materialization Time	Number of Tuples
V_1	3004	39
V_2	3215	52
V_3	3175	8
V_4	3034	8
V_5	4586	44
V_6	2103	35
V_7	2564	5
V_8	681	5

Table 5.3: Performance Results of the Materialization Procedure for the EBOC Data Set (ms)

Materialization Procedure Results

The total time to perform the materialization depends on the size of the data set from which a view is being extracted, and the selectivity of the view query definition. Because this procedure is executed in batch mode, the performance is not critical and can be improved. The materialization procedure solves each regular path query and stores the result in the corresponding materialized view. Also during the materialization, some statistics about the materialized views are collected. In Table 5.3 we show the total time consumed by the materialization procedure for each view, and also the number of tuples forming the materialized view.

Query Performance Results

In Table 5.4 we show the type of rewriting used for answering each query and the number of objects retrieved. In Tables 5.5 and 5.6, a summary of the performance results for the EBOC data set is shown. The column **N-O** is for "No Optimization", the **AD-O** column is for "Ancestor-Descendant Optimization", the **MV-O** is for "Materialized View Optimization", and **Combined** is for "Combined

	Rewriting Type	Objects Retrieved
Q_1	MAXIMAL (V_1)	23
Q_2	MAXIMAL (V_2)	29
Q_3	EXACT (V_3 | V_4)	16
Q_4	MAXIMAL (V_5)	8
Q_5	MAXIMAL (V_6)	54
Q_6	EXACT (V_7 | V_8)	24

Table 5.4: EBOC Queries Resolution Strategy

	N-O	AD-O	MV-O	Combined
Q_1	2223	2223	982	982
Q_2	4237	4326	3605	3605
Q_3	2283	2293	341	341
Q_4	4116	1362	445	1032
Q_5	-	9274	1502	1280
Q_6	1723	1753	340	340

Table 5.5: Performance Results of the Queries over the EBOC Data Set (ms) - First Execution

Optimization". As stated by each table, the first table corresponds to execution times with empty or cold buffers, while the second one indicates average execution times with warm buffers.

We will organize the analysis of the results based on the optimization involved.

Ancestor-Descendant Optimization Evaluation

Queries Q_1, Q_2, and Q_3 do not contain the descendants-or-self operator, and therefore there is no difference in performance between the **N-O** and **AD-O** procedure. The great differences in the performance of Q_1 and Q_3 with respect to Q_2 is because the later traverses a subtree in the hierarchy of the XML document which

	N-O	AD-O	MV-O	Combined
Q_1	690	698	310	310
Q_2	3976	3975	931	931
Q_3	706	708	107	107
Q_4	3790	1032	280	280
Q_5	-	7598	1000	460
Q_6	510	761	85	85

Table 5.6: Performance Results of the Queries over the EBOC Data Set (ms) - Average Subsequent Executions

is highly populated (a lot of instances exist) with respect to the sparsely populated subtree traversed by Q_1 and Q_3.

On the contrary, queries Q_4 and Q_5 achieved an outstanding improvement in performance when applying the **AD-O** due to the presence of the descendants-or-self operator. Moreover, Q_5 cannot be solved using the **N-O** procedure because too much time and too much memory are needed. The degree of improvement achieved by the **AD-O** depends on the position of the descendants-or-self operator in the query. The nearer the descendants-or-self operator is to the beginning of the regular path query, the more extensive the traversal of the document will be in the non-optimized procedure, and therefore, the greater the degree of improvement achieved by the **AD-O**.

In the query Q_6 results, notice that the performance achieved by the **AD-O** is worse than the one achieved by the **N-O** procedure. This particular behavior can be explained analyzing the structure of the EBOC data set (see Figure 5.3). Looking at the structure, notice that the descendants-or-self operator in query Q_6 is useless because it is asking for *topic* element descendants of *guide04/title* elements. If we look at the structure of the DTD, we detect that a *topic* element is the only valid direct descendant of */guide04/title*, and moreover, that no other *topic* element is valid in other parts of the XML document. Therefore, the **AD-O** procedure is performing unnecessary work because in this particular case the ancestor-descendant relationship does not need to be checked. This behavior can be avoided if the query answering procedure checks the structure to detect these cases. Also this structural analysis can check the validity of the query to avoid computational work in the cases where queries seek infeasible structures. In the current implementation, the structure of the data set is not taken into account to determine the optimization strategy to apply. We plan to include this structural analysis in a future implementation.

Materialized View Optimization Evaluation

Next we analyze the results obtained by the **MV-O** procedure. We will organize the results analysis based on the type of rewriting achieved by the rewriting algorithm.

From the results of query Q_1, we can observe that the **MV-O** outperformed the other two strategies. For this query, the rewriting algorithm generated a maximal rewriting using view V_1. View V_1 contains four steps of the regular path query Q_1. So, after retrieving data from V_1, the query answering procedure must retrieve data from the original mapping scheme tables. The total query time when using the **MV-O** is composed of the time consumed by the rewriting algorithm, and the time consumed in retrieving the data (from the views or from the original mapping tables). In all cases, the rewriting algorithm consumed an insignificant amount of time compared with the time needed to retrieve data from views and tables, even though the complexity of the algorithm is 2EXPTIME (Section 4.3).

A similar behavior was detected for query Q_2, where the results achieved by the **MV-O** procedure were also significant. For query Q_2, the rewriting algorithm generated a maximal rewriting using view V_2. View V_2 contains only four steps of the eight necessary to answer this query. Therefore, the remaining steps have to be retrieved from the mapping scheme tables. Due to the nature of the rewriting (more steps have to be retrieved from the original mapping tables), the improvements achieved for this query are not as outstanding as the results obtained by the rewriting of query Q_1.

In answering query Q_4, a maximal rewriting was generated by the rewriting algorithm. As in the previous cases, the improvement in performance obtained by the **MV-O** procedure was substantial. The maximal rewriting exploited the materialized view V_5. View V_5 contains three steps of the four needed for answering query Q_4. The remaining step has to be retrieved from the mapping scheme tables.

In answering query Q_3, a new type of rewriting, the exact rewriting, was exploited. This is the reason why the **MV-O** procedure achieved a remarkable improvement over the **AD-O** procedure. The rewriting algorithm generated an exact rewriting for query Q_3, combining the materialized views V_3 and V_4 using a conjunction operator. Therefore, the entire answer for this query can be retrieved from a combination of views, and no extra accesses to the mapping scheme tables are needed.

Query Q_6 was solved using an exact rewriting generated by the rewriting algorithm. This rewriting is formed by the conjunction of materialized views V_7 and V_8. The improvement in performance is also significant in this case.

Combined Optimization Evaluation

In answering query Q_5 we combined two optimization strategies. First, a maximal rewriting was generated, and then to solve the remainder of the query, the ancestor-descendant optimization was used. The rewriting algorithm generated

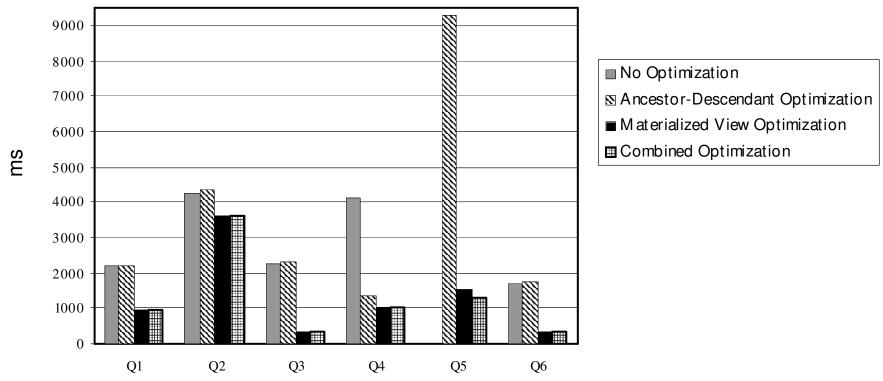

Figure 5.4: Summary Performance Results for the EBOC Data Set - First Execution

a maximal rewriting exploiting the materialized view V_6, which contains three of the steps needed for answering the query. As usual, the mapping scheme tables were queried to retrieve the remaining steps, which include a descendants-or-self operator. Therefore, we can apply the **AD-O** to solve this remaining expression. So, in solving Query Q_5, we show how several optimizations can be combined to improve the overall query performance.

Figures 5.4 and 5.5 show graphically the information presented in Tables 5.5 and 5.6. They summarize the performance results of the set of queries over the EBOC data set, using the four different optimization strategies: "No Optimization", "Ancestor-Descendant Optimization", "Materialized View Optimization", and "Combined Optimization". In all cases, the results show how the Combined Optimization and the Materialized View Optimization outperformed the other strategies. Depending on the query characteristics, the Combined and Materialized View Optimization can achieve improvements in performance in the range of 15% to 80% with respect to the other two optimization strategies.

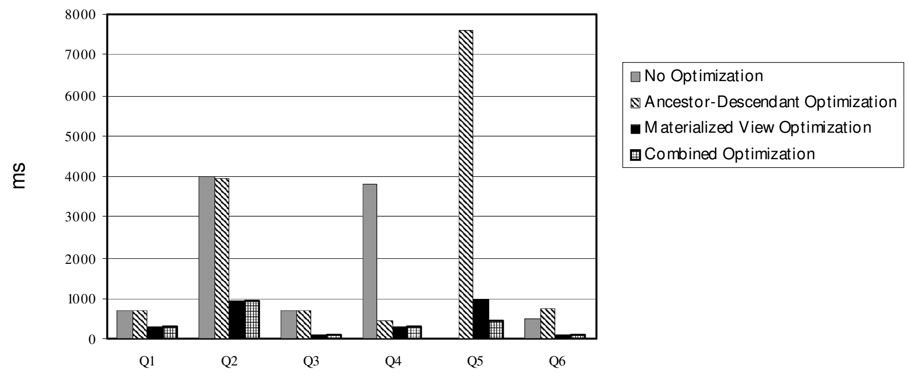

Figure 5.5: Summary Performance Results for the EBOC Data Set - Average Subsequent Repetitions

5.3.2 DBLP Data Set

For the DBLP data set, we present in Figure 5.6 the relevant portion of its DTD. As in the case of the EBOC data set, knowing something about the structure clarifies the procedure needed to solve the regular path queries. In this case we have annotated the DTD graph using the same notation as in the case of the EBOC data set. We can see from the DBLP DTD graph that the structure of this data set is simpler and flatter than that of the EBOC data set.

As in the previous case, the queries and materialized views are followed by an analysis of the results.

Queries

- $Q_7 = conferences/conference/issues$
- $Q_8 = conferences/conference/issues/issue$
- $Q_9 = conferences//conference/issues/issue//booktitle$

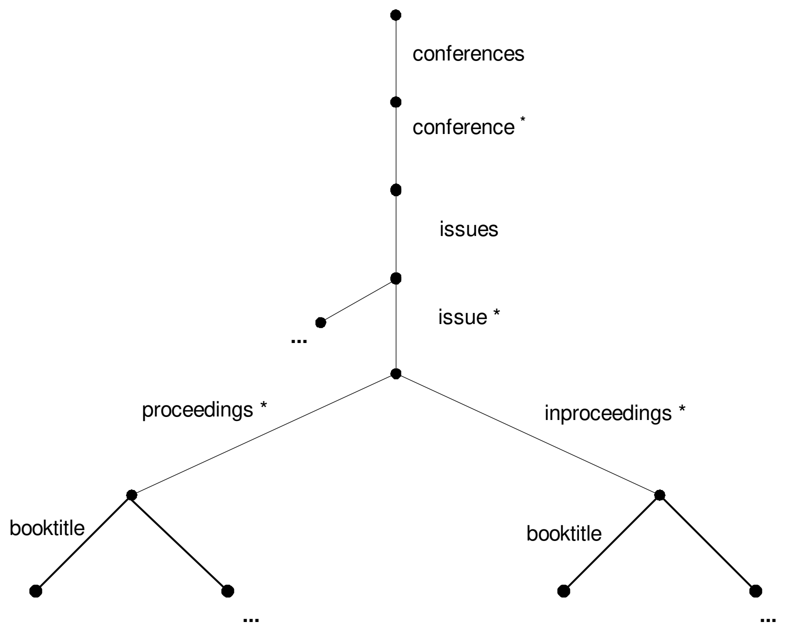

Figure 5.6: DBLP Data Set DTD

- Q_{10} = *conferences/conference/issues/(issue | volume)*
- Q_{11} = *conferences/conference/issues//proceedings*

Queries Q_7 and Q_8 are simple regular path queries. Queries Q_9 and Q_{11} include the descendants-or-self operator, while query Q_{10} involves the alternative or disjunction operator. The mix of queries is general enough to cover a variety of interesting cases.

Materialized Views

The materialized views were selected to show different types of rewritings.

- V_9 = *conferences/conference/issues*

	Materialization Time	Number of Tuples
V_9	25557	859
V_{10}	69009	3642
V_{11}	50000	0
V_{12}	25600	859

Table 5.7: Performance Results of the Materialization Procedure for the DBLP Data Set (ms)

- $V_{10} = conferences/conference/issues/issue$
- $V_{11} = conferences/conference/issues/volume$
- $V_{12} = conferences/conference$

Materialization Procedure Results

The total time consumed by the materialization procedure depends on the data set size, and the view query definition selectivity. In the case of the DBLP data set, the total time consumed by the materialization procedure for each query is shown in Table 5.7. The DBLP data set is more than 20 times bigger than the EBOC data set, and that is why the materialization procedure took a larger amount of time. In the case of the view V_{11} the number of materialized tuples was 0 and still the materialization time was enormous. As we already said, the materialization procedure was implemented without any optimization.

We present in Table 5.8 the type of rewriting used by the Materialized View Optimization to solve each query, and the number of objects retrieved. In addition, in Tables 5.9 and 5.10, we summarize the performance results for the DBLP data set. The same notation as in the previous case is used. As with the EBOC results we have organized the analysis based on the optimization involved. Note that in this

	Rewriting Type	Objects Retrieved
Q_7	PERFECT	859
Q_8	MAXIMAL (V_9)	3642
Q_9	NO REWRITING	147570
Q_{10}	EXACT ($V_{10} \mid V_{11}$)	3642
Q_{11}	MAXIMAL (V_{12})	1963

Table 5.8: DBLP Queries Resolution Strategy

	N-O	AD-O	MV-O
Q_7	17305	11657	611
Q_8	27139	23174	13680
Q_9	-	97721	87776
Q_{10}	26849	23204	2464
Q_{11}	-	21487	16084

Table 5.9: Performance Results of the Queries over the DBLP Data Set (ms)- First Execution

case, a **Combined Optimization** column is not shown, because there is no query that can be optimized combining two different optimizations.

	N-O	AD-O	MV-O
Q_7	10993	10993	353
Q_8	22284	22277	12422
Q_9	-	53779	53790
Q_{10}	22282	22287	2053
Q_{11}	-	821	742

Table 5.10: Performance Results of the Queries over the DBLP Data Set (ms) - Average Subsequent Execution

Ancestor-Descendant Optimization Evaluation

Queries Q_7, Q_8, and Q_{10} do not contain the descendants-or-self operator, and therefore there is no noticeable difference in the performance between the **N-O** and the **AD-O** procedures. As can be seen in the table, queries Q_9 and Q_{11} cannot be solved using the **N-O** procedure, and using the **AD-O** some good results were obtained. The **N-O** procedure for queries Q_9 and Q_{11} consumed too much time and memory to be considered a feasible solution. Overall, the **AD-O** procedure yields good results.

Materialized View Optimization Evaluation

Next we analyze the results obtained by applying the **MV-O** procedure. In all cases, a performance improvement was achieved, and as in the EBOC queries, the degree of improvement differs depending on the type of rewriting that was generated by the rewriting algorithm.

The ideal case is shown for query Q_7, where a perfect materialization was found for the whole query, and therefore no extra accesses to the original mapping tables were needed. View V_9 was defined by the same regular expression as query Q_7.

The **MV-O** procedure achieves an important improvement in performance when the rewriting algorithm generates an exact rewriting. Query Q_{10} is an example of this situation, where two views are combined by the rewriting algorithm to produce the query answer. The rewriting involves materialized views V_{10} and V_{11}.

Query Q_9 shows the case when no rewriting is possible, and therefore, the original query answering procedure is used. Although no valid rewriting was found, the rewriting algorithm was executed. No additional overhead due to this execution is noticeable in the results. The rewriting algorithm takes an insignificant amount of time compared to that consumed by the procedure of retrieving the data itself, despite its complexity.

In answering query Q_8, the rewriting algorithm came up with a maximal rewriting using view V_9. In this case, extra accesses to the mapping tables were needed to retrieve the remaining steps of the expression. Yet the gain in performance obtained by the **MV-O** procedure was significant.

Finally, in answering query Q_{11}, the rewriting algorithm generated a maximal rewriting that achieves an important improvement in performance. A maximal rewriting was generated exploiting the materialized view V_{12}. In this case the improvement was less significant than in other cases, because the materialized view only contains the first two steps of all those needed by the query. Extra accesses to the mapping tables were performed in order to retrieve the remaining steps of the query.

Figures 5.7 and 5.8 represent graphically the results presented in Tables 5.9 and 5.10. They summarize the performance results of the set of queries on the DBLP data set, using the three different execution strategies: "No Optimization", "Ancestor-Descendant Optimization", and "Materialized View Optimization". In all the cases, the results show how the materialized view optimization outperformed the other two query answering procedures. In all cases, the improvement achieved by the Materialized View Optimization lies between 80% and 10% over the other two optimization strategies.

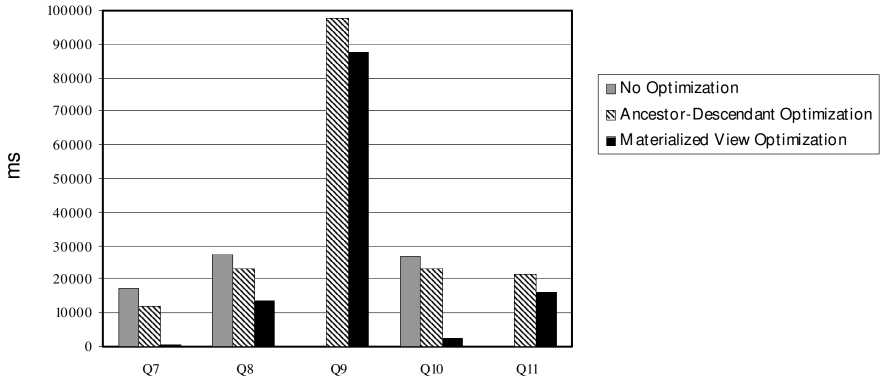

Figure 5.7: Summary Performance Results for the DBLP Data Set - First Execution

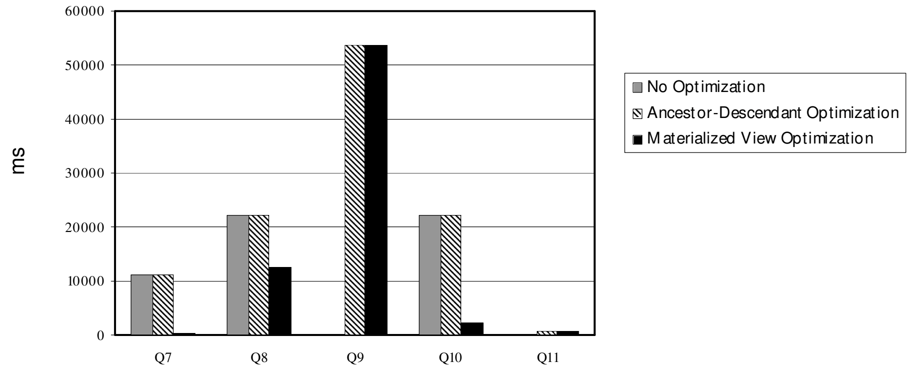

Figure 5.8: Summary Performance Results for the DBLP Data Set - Average Subsequent Executions

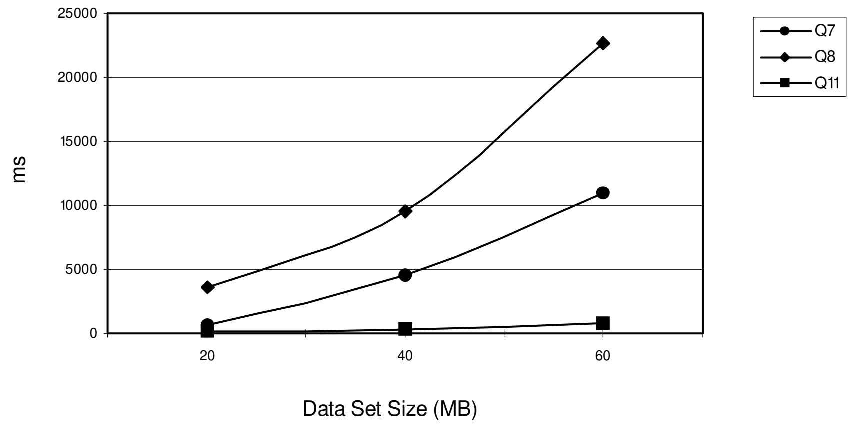

Figure 5.9: Summary Performance Varying DBLP Data Set Size

Varying Data Set Size

We generated subsets of the DBLP data set of various sizes to study the impact of the data set size on the query response time. From the original 60 MB data set, we took two subsets: one with size 20 MB and other with size 40 MB. We selected three of the previous queries, and we executed them over the different data sets, applying the ancestor-descendant optimization. The results show that the query response time remains acceptable as the data set size increases. Note that the relationships between the query answering behaviors are maintained for all three data sets. In Figure 5.9 we show the results for the selected queries for each data set.

5.4 Query Workload

The remaining experiments cover the cases where we have a fixed set of queries (a workload), and data sets with varying characteristics. We used one real data set, and one synthetic data set.

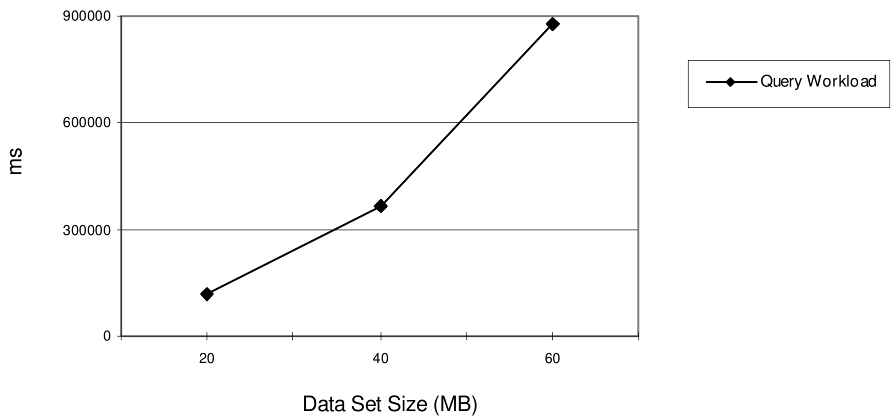

Figure 5.10: Summary Performance Varying DBLP Data Set Size

We used the three subsets of the DBLP data set for this experiment. We defined a query workload composed of several queries. As the workload was synthetic, we also defined some synthetic frequencies for each query in the workload. We executed randomly the queries in the workload following their frequencies. Finally, we calculated the average response time over 100 executions, including executions with cold and warm buffers. In Figure 5.10 we show the results for the workload over each data set. In this case we can observe the same behavior as with single queries.

For the last experiment, a synthetic data set was used. Several XMark-like data sets were generated using ToXgene, varying the fanout allowed for each element. We constructed five data sets with different minimum and maximum fanouts: 0 to 10, 10 to 20, 20 to 30, 30 to 40 and 40 to 50. A typical workload for the XMark data set was defined by the XMark Benchmark project [BCF+01]. We took the proposed workload and we extracted the regular expressions used in each query. Then, we selected the ten most frequently used regular expressions and we assigned an synthetic frequency to each one. The frequency was calculated taking into account number of occurrences within the proposed workload. Then, we executed the queries in random order following the assigned frequencies. After 100 repetitions, we calculated

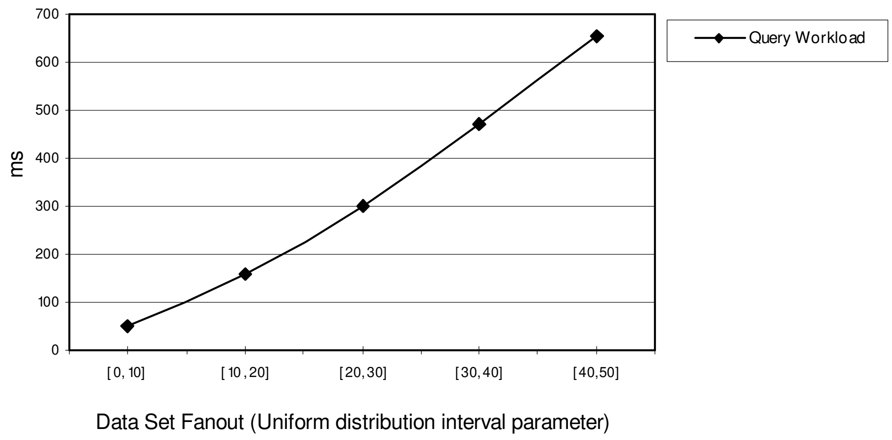

Figure 5.11: Summary Performance Varying XMark-like Data Set Fanout

the average response time. In Figure 5.11 we present the results obtained. Also in this case we confirmed that the results were quite acceptable when varying the data set fanout.

CHAPTER VI

Conclusions and Future Work

6.1 Conclusions

In this thesis we have presented the ToX Relational mapping scheme, and several optimizations to improve the performance of regular path queries in the context of XML data stored in relational databases.

The ToX Relational mapping scheme stores XML documents registered within the ToX repository. The ToX Relational mapping scheme is an extension of the YAM mapping [AB02] developed within the context of ToX. We extended the original mapping scheme with order information, structural information about attributes, and an encoding that maintains containment relationships (ancestor-descendant relationships). This encoding is exploited during the query answering procedure as an optimization mechanism when the regular path query contains the descendants-or-self operator.

In this work, we also studied the feasibility of using materialized views as a query optimization tool. Based on a selected materialized view configuration, we defined the structure of the views taking into account their view query definitions. We extended the structure proposed by Kemper and Moerkotte in their Access Support Relations work [KM92] with the possibility of managing more complex regular expressions. We defined several heuristic rules in order to automatically derive the structure of the views. In order to use the materialized views during the query answering procedure, we adapted and implemented an algorithm for rewriting a regular expression using other regular expressions, as proposed by Calvanese et al. [CGLV99]. The rewriting algorithm was embedded into the Relational Document Navigator and it is fully functional. We have also described some extensions that can be used to exploit the available structural information to enhance the performance of the query answering procedure.

Finally, to test all the proposed optimizations, we performed several experiments to demonstrate performance improvements. We verified that all the solutions proposed in this thesis were feasible to implement and we confirmed that the materialized view optimization is a promising approach in optimizing regular path queries in the context of XML data stored in a relational database.

6.2 Future Work

In its current state, the toxRelational component allows the storage and querying of XML data. There are, however, several enhancements that could improve the performance and applicability of the component.

In order to support reconstructions of entire XML documents, the ToX Relational mapping scheme has to store all other entities specified by the XML data representation specification, such as processing instructions, or comments. The current mapping scheme stores information about attribute types, but it does not exploit this information during the query processing. This extension will support following IDs and IDREFs during the processing of a regular path query. To implement this extension, we have to deal with possible infinite cycles that may appear in the presence of IDs and IDREFs when the regular path queries involve them.

With respect to the materialized views optimization, several issues need to be addressed. The framework presented in this thesis is the first step in optimizing queries using views. We need a procedure to choose the set of regular expressions to be materialized. A convenient approach to select a materialized view configuration is to study a typical workload. From this information, the more frequently used database paths can be detected. With this information in hand, we will be able to determine the more appropriate paths to materialize using the data mining algorithm presented by Wang and Liu [WL98, WL00]. This algorithm takes as input a set of graph structures, and it discovers substructures that appear frequently (with a minimum support) in the original graphs. For our purposes, we can model the set of queries of the workload as graphs, and we can apply the algorithm to discover the most common substructures. During the discovery procedure, the information about frequencies of appearance in the workload can be used to assign weights to each graph. Then, the path expressions most frequently queried are given high weights, influencing the results of the discovery procedure. As output, a set of regular path queries will be selected. Ideally, all these queries can be materialized, but in practice other constraints may apply, and therefore another filtering step must be performed. Typical constraints are: disk space, materialization time, etc. Given a set of possible materializations and a set of constraints, we face an optimization problem. To formalize and solve this optimization problem, we have to define a cost model combining the set of possible materializations and the constraints. As a result, an optimal materialized view configuration will be generated.

Regarding the rewriting algorithm, several issues need to be studied. The current implementation does not take into account structural information about the XML documents to determine the best optimization strategy to apply. Also further study is needed on how to exploit materialized views in the cases where maximal or exact rewritings are not achieved, or multiple maximal rewritings are found. Another interesting optimization would be to store precomputed query plans to avoid the execution of the rewriting algorithm for frequent queries.

We have to consider the development of an optimal access mechanism for the ToX Relational mapping scheme. This mechanism has to be embedded into the Relational Document Navigator instead of the step-by-step approach used by the current implementation. This mechanism will generate SQL statements dynamically to access the data stored in the mapping scheme to answer regular path queries with as few SQL statements as possible. Although we cannot be sure that this alternative is better, we think it is worthwhile to test it.

Other interesting issues to study are how to support update of XML documents that are stored in the ToX Relational mapping scheme, and how to maintain the materialized views up to date with respect to changes in those documents.

Finally, we have to study the cases where the regular path queries contain predicates. We have to verify that all the reasoning made through out this thesis remains applicable and point out some particular issues that can arise from the presence of predicates.

APPENDICES

APPENDIX A

ToX Document Navigator

Specification

The Document Navigator is an interface defined within the framework of ToX to hide the implementation details of the different backends. The Document Navigator allows an application to navigate XML documents stored in the ToX repository regardless of the implementation. Each new backend added to ToX must implement this specification.

The Document Navigator specification is similar in spirit to the Document Object Model specification [DOM98]. It includes 4 basic entities: XDocument, XElement, XNode and XAttribute. These entities correspond to the Document, Element, Node and Attribute entities of DOM. To interact with each entity several methods have been defined. Combining methods over different entities, an XML document stored in the repository can be fully traversed.

The two main entities of this interface are the XDocument and the XNode. An XDocument represents an entire XML document or an XML document fragment. An XNode represents any node of the XML document.

Method	**Description**
getRoot()	Returns the root node of a given XML Document
setRoot(*root*)	Sets the XElement *root* to be the root of this XDocument
createElement(*name*)	Creates an XElement in the backend with the given name
createNode(*name*)	Creates an XNode in the backend with the given name
createAttribute(*name*)	Creates an XAttribute in the backend with the given name

Table A.1: ToX Document Navigator XDocument Interface

The XDocument interface represents an entire document or a document fragment. Since XNode objects cannot exist outside the context of an XDocument, the XDocument interface contains the construction methods needed to create these objects. The methods defined by this interface are shown in Table A.1.

The XNode interface is the primary interface for navigating an XML document. It defines the behavior of a single node in the XML document regardless of its type. There exists three types of XNodes representing elements, attributes and text values. Some characteristics are shared by all types of XNodes, and others are exclusive of each type. To support the exclusive characteristics, the XNode interface was extended by the XElement and XAttribute interfaces. The methods supported by the XNode interface are listed in Table A.2.

The XElement interface defines the behavior of the element nodes. XElements may have XAttributes associated with them. The methods of this interface are described in Table A.3.

Finally, the XAttribute interface defines the behavior of attribute nodes. XAttributes always belong to XElements, but are not considered their children. Therefore, XAttributes do not have parents (as defined for other types of XNodes), but owner elements. Table A.4 summarizes the methods defined by the XAttribute interface.

Method	Description
getChildren()	Returns a list that contains all children of this node (not necessarily in order)
getChildren(*name*)	Returns a list that contains all children of this XNode with the given name (not necessarily in order)
getDescendants()	Returns a list that contains all descendants of this node (not necessarily in order)
getDescendants(*Regular Expression*)	Returns a list that contains all descendants of this XNode reached by a path satisfying the given *Regular Expression*
getAncestors()	Returns a list that contains all ancestors of this XNode (not necessarily in order)
getAncestors(*Regular Expression*)	Returns a list that contains all ancestors of this XNode reached by a path satisfying the given *Regular Expression*
getFollowingSiblings()	Returns a list that contains all following sibling elements of this XNode
getFollowingSiblings(*name*)	Returns a list that contains all following sibling elements of this XNode with the given name
getName()	Returns the name of this XNode
getNextSibling()	Returns the next sibling element of this XNode
getNextSibling(*name*)	Returns the next sibling element of this XNode with the given name
getParent()	Returns the parent of this XNode
getPrecedingSiblings()	Returns a list that contains all preceding sibling elements of this XNode
getPrecedingSiblings(*name*)	Returns a list that contains all preceding sibling elements of this XNode with the given name
getPreviousSibling()	Returns the previous sibling element of this XNode
getPreviousSibling(*name*)	Returns the previous sibling element of this XNode with the given name
getType()	Returns the type of this XNode (ATTRIBUTE, ELEMENT, or UNTYPED)
getValue()	Returns the value of this XNode
hasChildren()	Returns true/false whether this XNode has children or not
hasNextSibling()	Returns true/false whether this XNode has next sibling element or not
hasParent()	Returns true/false whether this XNode has parent or not
hasPreviousSibling()	Returns true/false whether this XNode has previous sibling element or not

Table A.2: ToX Document Navigator XNode Interface

Method	**Description**
getAttributes()	Returns a list of all the attributes of this XElement
getAttribute(*name*)	Returns the attribute called *name*
hasAttributes()	Returns true/false whether this XElement has attributes or not
hasAttributes(*name*)	Returns true/false whether this element has an attribute called *name*
removeAttribute(*XAttribute*)	Removes the XAttribute from this XElement
setAttribute(*XAttribute*)	Adds the XAttribute to the list of attributes of this XElement

Table A.3: ToX Document Navigator XElement Interface

Method	Description
getOwnerElement()	Returns the XElement node associated with this XAttribute

Table A.4: ToX Document Navigator XAttribute Interface

Document Navigator Versus DOM

From a functionality point of view, the Document Navigator interface and the DOM interface differ in several aspects. The Document Navigator interface only supports the mentioned four entities, while the DOM interface is more general and supports all entities defined by the XML specification. The Document Navigator has a restricted set of constructional methods. One advantage of the Document Navigator compared to DOM is the possibility of node navigation based on a regular expression.

From an implementation point of view, the interfaces follow very different approaches. While the Document Navigator reconstructs the XML document during the traversal, the DOM interface reconstructs the XML document prior to any navigation. Therefore, the DOM interface is very inefficient in terms of memory usage.

APPENDIX B

From the Document Navigator to SQL Queries

The relational implementation of the Document Navigator translates each method of the interface into SQL queries. The Document Navigator interface contains navigational and constructional methods that were described in Appendix A. The methods concerning creation, modification, and deletion of elements in the backend were not implemented in the toxRelational component. In this appendix, we show some of the navigational methods and their translation to SQL queries. Some methods were omitted due to their similarities with other methods shown here.

The SQL queries shown here follow the syntax of SQL supported by the IBM's UDB DB2 database.

XDocument Interface

In the XDocument interface, only one method accesses the relational database. The creation of a new XDocument object based on an XML document stored in the repository corresponds to the first interaction between any application and a particular backend. The parameters needed to create an XDocument object are the *Unique Document Identifier* (DOCID) and the *Root Object Identifier* (ROOTID).

XDocument creation	⇒	**select** target, ordinal, name, preorder, size **from** Edge **where** docid = DOCID **and** source = ROOTID

Table B.1: SQL Queries Related to the Creation of an XDocument Object

Both values can be obtained from the toxRelational catalog tables. In Table B.1 we show the SQL query that collects the data required to create an XDocument object. The creation of an XDocument is accompanied by the creation of an XNode object representing its root element. Based on the parameters DOCID and ROOTID the relevant information to create this XNode object is retrieved. This information is retrieved from the *Edge* table.

XNode Interface

The methods in the XNode interface can be divided into several categories depending on the type of information they deal with. The categorization divides the methods into child-related methods, parent-related methods, sibling-related methods, and ancestor-descendant-related methods. Next we will show some of the navigational methods for each of the four categories, recalling that some methods are omitted due to their similarities with others presented here. In all the SQL queries shown for this interface, the *Unique Document Identifier* (DOCID) and *Unique Object Identifier* (OID) are the only parameters. These can be obtained from the XNode object that is the source of the query. Note that to simplify the SQL queries, we omitted the fields *type-value* and *int-value* from the exposition. In the current implementation the type of the returned value is checked.

The methods related to information about children are shown in Table B.2.

The *hasChildren()* and *getChildren()* methods extract information from both the *Edge* and the *Leaf* tables, because an element node can have as children both other elements and text values (i.e., mixed content). On the contrary, the *hasChildren(name)* and *getChildren(name)* methods only access the *Edge* table because a text value cannot have a name associated with it. Therefore, in these cases, only element children from the *Edge* table are returned.

We have to point out that some of these queries use the **WITH** SQL clause, which simply constructs a temporary table containing partial results. This temporary table is valid only for the duration of the query processing.

Table B.3 shows all methods of the XNode interface related to parent information. In the methods *hasParent()* and *getParent()*, only the *Edge* table is queried because elements are the only entities that can be parents of other entities. Neither text values (stored in the *Leaf* table) nor attributes (stored in the *Attribute* table) can be parents of other entities.

Representative methods of those related to information about siblings are shown in Table B.4. All the methods related to sibling elements have to retrieve information from both the *Edge* and the *Leaf* tables, due to the existence of elements with mixed content. All these queries need the PREVIOUS or NEXT ordinal value, which can be derived from the XNode object source of the query.

To finish with the XNode interface, we present in Table B.5 some representative methods related to ancestor-descendant information. The *getDescendants()* method retrieves all descendants nodes of a particular XNode object, following the order of the depth first order traversal. In this case, we use a recursive SQL query to reconstruct the paths in the original XML document. The first SQL query inside the **WITH** clause constitutes the seed of the recursion. For the first recursive SQL query, the seed of the recursion is formed by all the child nodes of the XNode object

hasChildren()	⇒	**with** all-children (total-children) **as** (
		    select count(*)
		    from Edge
		    where docid = $NodeA$.DOCID
		    and source = $NodeA$.OID
		    union all
		    select count(*)
		    from Leaf
		    where docid = $NodeA$.DOCID
		    and source = $NodeA$.OID)
		select sum(total-children)
		from all-children

hasChildren(name)	⇒	**select** count(*) **from** Edge **where** docid = $NodeA$.DOCID **and** source = $NodeA$.OID **and** name = 'NAME'
getChildren()	⇒	**select** target, ordinal, name, ' ' as char-value, preorder, size **from** Edge **where** docid = $NodeA$.DOCID **and** source = $NodeA$.OID **union all** **select** target, ordinal, ' ' as name, char-value, preorder, size **from** Leaf **where** docid = $NodeA$.DOCID **and** source = $NodeA$.OID
getChildren(name)	⇒	**select** target, ordinal, name, ' ' as char-value, preorder, size **from** Edge **where** docid = $NodeA$.DOCID **and** source = $NodeA$.OID **and** name = 'NAME'

Table B.2: SQL Queries Related to Child Element Information

hasParent()	$\Rightarrow$	**select** count(*) **from** Edge **where** docid = $NodeA$.DOCID **and** target = $NodeA$.OID
getParent()	$\Rightarrow$	**select** source, target, ordinal, name, preorder, size **from** Edge **where** docid = $NodeA$.DOCID **and** target = $NodeA$.OID

Table B.3: SQL Queries Related to Parent Information

hasPreviousSibling()	$\Rightarrow$	**with** temporal (total-siblings) **as** (
		    select count(*)
		    from Edge
		    where docid = $NodeA$.DOCID
		    and source = $NodeA$.OID
		    and ordinal = PREVIOUS-ORDINAL
		    union all
		    select count(*)
		    from Leaf
		    and source = $NodeA$.DOCID
		    and ordinal = PREVIOUS-ORDINAL)
		select sum(total-siblings) **from** temporal

getPreviousSibling()	$\Rightarrow$	**select** target, ordinal, name, ‘ ’ as char-value, preorder, size **from** Edge **where** docid = $NodeA$.DOCID **and** source = $NodeA$.OID **and** ordinal = PREVIOUS-ORDINAL **union all** **select** target, ordinal, ‘ ’ as name, char-value, preorder, size **from** Leaf **where** docid = $NodeA$.DOCID **and** source = $NodeA$.OID **and** ordinal = PREVIOUS-ORDINAL

Table B.4: SQL Queries Related to Sibling Element Information source of the query. The next SQL query within the first recursive SQL extracts the remaining nodes using as starting nodes the ones collected by the seed of the recursion. This first recursive SQL query reconstructs all element paths in the subtree rooted at the XNode that is the source of the query. The second recursive query uses as seed all those nodes already collected by the first recursive query. From those nodes, the SQL query extracts all text values from the *Leaf* table that are children of any of the nodes that are seeds of the recursion.

XElement Interface

All the methods of the XElement interface deal with attributes. We present in Table B.6 some of these methods. All the queries of this interface extract information from the *Attribute* table. As in the previous cases, the values of *Unique Document Identifier* (DOCID) and *Unique Object Identifier* (OID) are extracted from the XElement object that is the source of the query.

XAttribute Interface

None of the methods of the XAttribute interface access the relational database.

getDescendants()	$\Rightarrow$	**with** descen-edge (source, target, ordinal, name, char-value,
		preorder, size) **as** (
		    select source, target, ordinal, name, ‘ ’ as char-value,
		    preorder, size
		    from Edge
		    where docid = $NodeA$.DOCID
		    and target = $NodeA$.OID
		    union all
		    select child.source, child.target, child.ordinal, child.name,
		    ‘ ’ as char-value, child.preorder, child.size
		    from descen-edge parent, basicedge child
		    where parent.target = child.source
		    and child.target <> $NodeA$.OID
		    and child.docid = $NodeA$.DOCID)
		, descen-leaf (source, target, ordinal, name, char-value,
		preorder, size) **as** (
		    select source, target, ordinal, name, char-value,
		    preorder, size
		    from descen-edge
		    union all
		    select child.source, child.target, child.ordinal, ‘ ’ as name,
		    child.char-value , child.preorder, child.size
		    from descen-leaf parent, basicleaf child
		    where child.docid = $NodeA$.DOCID
		    and parent.target = child.source)
		select source, target, ordinal, name, char-value, preorder, size
		from descen-leaf
		order by preorder

Table B.5: SQL Query Related to Descendants Element Information

hasAttributes()	⇒	**select** count(*) **from** Attribute **where** docid = $NodeA$.DOCID **and** source = $NodeA$.OID
getAttributes()	⇒	**select** source, target, attr-name, char-value, attr-type **from** Attribute **where** docid = $NodeA$.DOCID **and** source = $NodeA$.OID
getAttributes(name)	⇒	**select** source, target, attr-name, char-value, attr-type **from** Attribute **where** docid = $NodeA$.DOCID **and** source = $NodeA$.OID **and** attr-name = "NAME"

Table B.6: SQL Queries Related to Attributes Information

APPENDIX C

ToXgene Template

In this appendix we present a subset of the ToXgene template used to generate the synthetic data sets for this work.

ToXgene is a template-based generator for large collections of synthetic XML documents. ToXgene templates used the ToXgene Specification Language (TSL) for defining the input templates. TSL templates are essentially XML Schema specifications annotated with specifications of probability distributions and CDATA content descriptors, used for generating both attributes and elements. Like XML Schema, TSL templates are XML documents. TSL-specific annotations are prefixed by the string **tox**.

A TSL template is an XML document whose root is a **tox-template** element. The main declarations in a TSL template are:

- **tox-distribution**: defines a random number generator according to the given distribution.
- **simple types**: defines a gene for literals.
- **complex types**: defines a gene for document fragments.

- **tox-list**: defines a main memory repository for values that can be shared during the creation of the actual XML document.
- **tox-document**: defines the generation of one or more XML documents of a given type.

We include a subset of the TSL template we used for generating the synthetic data sets including examples of **tox-distribution**, **simple type**, and **tox-document**. For each element and attribute in the XML document, we included a declaration inside the **tox-document** section of the template.

```
<?xml version="1.0"?>
<!DOCTYPE tox-template SYSTEM
              "http://www.cs.toronto.edu/tox/toxgene/ToXgene1_1.dtd"
[
  <!ENTITY num_categories "99">
  <!ENTITY num_edges "380">
  <!ENTITY num_items "2174">
  <!ENTITY num_open_auctions "1199">
  <!ENTITY num_people "2549">
]>
<tox-template>
  <tox-distribution name="f_name" type="exponential" minInclusive="0"
                                   maxInclusive="9999" mean="3333">
  </tox-distribution>
  <tox-distribution name="l_name" type="exponential" minInclusive="0"
                                     maxInclusive="9999" mean="3333">
  </tox-distribution>
  ...
  <!-- this type is used to pick random categories for the catgraph -->
  <simpleType name="pick_category">
        <restriction base="nonNegativeInteger">
          <maxInclusive value="&num_categories;"/>
        </restriction>
  </simpleType>
  ....
```

```
  <tox-document name="output/auction_F0_10">
    <element name="site">
      <complexType>
        <element name="regions">
            <complexType>
                <!-- 550 items in africa -->
                <element name="africa">
                  <complexType>
                      <element name="item" minOccurs="0"
                                                maxOccurs="10">
                          <complexType mixed="true">
                            <tox-scan path="[item_list/item]" name="i"
                                                  where="LT([id],55)">
                              <attribute name="id">
                                  <tox-expr value="'item'#[id]"/>
                              </attribute>
                              <tox-alternatives>
                                <tox-option odds="30">
                                  <attribute name="featured">
                                      <simpleType>
                                        <restriction base="string">
                                          <tox-value>
                                              yes
                                          </tox-value>
                                        </restriction>
                                      </simpleType>
                                    </attribute>
                                </tox-option>
                                ...
      </complexType>
    </element>
    ...
  </tox-document>
</tox-template>
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[AB02] M. Arenas and D. Barbosa. YAM: Yet Another (XML/Relational) Mapping (a.k.a.) the ball, 2002. Unpublished. University of Toronto.

[Abi97] S. Abiteboul. Querying semi-structured data. In *Proceedings of International Conference on Database Theory*, pages 1–18, 1997.

[Abi99] S. Abiteboul. On views and XML, 1999. SIGMOD Record, 28(4):30-38.

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web. From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers, 2000.

[ACGR01] S. Abiteboul, S. Cluet, G.Ferram, and M. Rousset. A dynamic warehouse for XML data of the Web. *Bulletin of the Technical Committee on Data Engineering*, 24(2):40–56, 2001.

[AKM01] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *Symposium on Discrete Algorithms*, pages 547–556, 2001.

[AMR+98] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental maintenance for materialized views over semistructured data. In *Proceedings of the International Conference on Very Large Data Bases*, pages 38–49, 1998.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[AR02] S. Alstrup and T. Rauhe. Improved labeling scheme for ancestor queries. In *Symposium on Discrete Algorithms*, 2002.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison Wesley, 1986.

[Bar00] D. Barbosa. On storage alternatives for semistructured data. Qualifying Oral Document, Department of Computer Science, University of Toronto, 2000.

[BBM+01] D. Barbosa, A. Barta, A. Mendelzon, G. Mihaila, F. Rizzolo, and P. Rodriguez-Gianolli. ToX - the Toronto XML Engine. In *Workshop on Information Integration on the Web*, pages 66–73, 2001.

[BCF+01] R. Busse, M. Carey, D. Florescu, I. Manolescu, A. Schmidt, and F. Waas. XMark, the XML benchmark project, 2001. Available at http://monetdb.cwi.nl/xml/index.html.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. Technical report, Department of Computer and Information Science, University of Pennsylvania. Available at http://www.cis.upenn.edu/ db/abstracts/sigmod96.html, 1996.

[BFRS02] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML Schema to relations: A cost-based approach to XML storage. In *Proceedings of the 18th International Conference of Data Engineering*, pages 64–75, 2002.

[BMK02] D. Barbosa, A. Mendelzon, and J. Keenleyside. ToXgene: An extensible template-based data generator for XML. In *Fifth International Workshop on the Web and Databases. Available at http://www.cs.utoronto.ca/tox/toxgene*, 2002.

[Bru95] T. Bruce. *Designing Quality Databases with IDEF1X Information Models.* Dorset House Publishing, 1995.

[Bun97] P. Buneman. Semistructured data. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems*, pages 117–121, 1997.

[CAD+94] R. Cattell, T. Atwood, J. Dubl, G. Ferran, M. Loomis, and D. Wade. *The Object Database Standard: ODMG.* Morgan Kaufmann Publishers, 1994.

[CCM96] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 413–422, 1996.

[CGLV99] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Vardi. Rewriting of regular expressions and regular path queries. In *Proceedings of the 18th ACM SIGACT Symposium on Principles of Database Systems*, pages 194–204, 1999.

[CKM02] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In *Proceedings of Principles of Database Systems*, pages 271–281, 2002.

[CRF00] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB (Informal Proceedings)*, pages 53–62, 2000.

[DCD] Data Content Definition. World Wide Web Consortium, 2001. Available at http://www.w3.org/TR/NOTE-dcd.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. *Eighth International World Wide Web Conference*, 31(11-16):1155–1169, 1999.

[DFS99a] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data in relations. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats.*, 1999.

[DFS99b] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 431–442, 1999.

[DOM98] Document Object Model (DOM) Level 1 Specifications, 1998. World Wide Web Consortium. Recommendation. Available at http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001.

[DTD] Data Type Definition Specification. World Wide Web Consortium, 1998. Available at http://www.w3.org/XML/1998/06/xmlspec-report.htm.

[FFCM99] D. Flanagan, J. Farley, W. Crawford, and K. Magnusson. *Java Enterprise in a Nutshell.* O'Reilly, 1999.

[FFK+98] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a web-site management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 414–425, 1998.

[FK99a] D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical report, INRIA. Available at http://www.inria.fr/rrrt/rr-3680.html, 1999.

[FK99b] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin, 22(3):27–34*, 1999.

[FLS98] D. Florescu, A. Levy, and D. Suciu. Query optimization algorithm for semistructured data. Technical report, Stanford University. Available at http://www-db.stanford.edu/pub/papers/qo.ps, 1998.

[FM00] T. Fiebig and G. Moerkotte. Evaluating queries on structure with eXtended Access Support Relations. In *WebDB (Informal Proceedings)*, pages 41–46, 2000.

[FS98] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *International Conference Data Engineering*, pages 14–23, 1998.

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajamaran, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems.*, 8(2):117–132, 1997.

[GT01] Gosta Grahne and Alex Thomo. Algebraic rewritings for optimizing regular path queries. In *International Conference on Database Theory*, pages 301–315, 2001.

[Hal00] A. Halevy. Answering queries using views: a survey. Technical report, Computer Science Department, University of Washington. Available at http://www.cs.washington.edu/homes/alon/site/files/view-survey.ps, 2000.

[ILW00] Z. Ives, A. Levy, and D. Weld. Efficient evaluation of regular path expressions on streaming XML data. Technical report, Department of Computer Science. University of Washington. Available at ftp://ftp.cs.washington.edu/tr/2000/05/UW-CSE-00-05-02.PS.Z, 2000.

[Ive99] Z. Ives. Efficient evaluation of path expressions in XML. Technical report, Ph.D. General Examination. Department of Computer Science. University of Washington. Available at http://citeseer.nj.nec.com/393968.html, 1999.

[JDB] JDBC Data Access API. Sun Microsystems Inc. Core 2 Specification, 2000. Available at http://java.sun.com/products/jdbc/download.html.

[JLWY02a] H. Jiang, H. Lu, W. Wang, and J. Yu. Path materialization revisited: An efficient storage model for XML data. In *Proceedings of the 13th. Australasian Database Conference*, 2002.

[JLWY02b] H. Jiang, H. Lu, W. Wang, and J. Yu. XParent: An efficient RDBMS-Based XML database system, 2002. Available at http://ihome.ust.hk/ jianghf/files/icde2002-jiang.pdf.

[KM92] A. Kemper and G. Moerkotte. Access Support Relations: An indexing method for object bases. *Information Systems*, 17(2):117–145, 1992.

[KM00] M. Klettke and H. Meyer. XML and Object-Relational database systems - enhancing structural mappings based on statistics. In *WebDB (Informal Proceedings)*, pages 63–68, 2000.

[LD00] H. Liefke and S. Davidson. View maintenance for hierarchical semistructured data. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 114–125, 2000.

[Lev99] A. Levy. Logic-based techniques in data integration. In *Workshop on Logic-Based Artificial Intelligence*, 1999.

[Ley01] M. Ley. Digital Bibliography and Library Project, 2001. Available at ftp://ftp.informatik.uni-trier.de/pub/users/Ley/bib/records.tar.gz.

[Liv] C. Livingstone. Evidence-Based On Call. EPOCare Project, University of Toronto, 2001.

[LM01] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *The Very Large Data Bases Journal*, pages 361–370, 2001.

[LWM] G. Lausen, F. Wei, and P. Marron. Aspects of rewriting queries on semistructured data. Technical report, University of Freiburg, Institute for Computer Science. Germany. Available at http://citeseer.nj.nec.com/451370.html.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[MAG+00] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. LORE: Lightweight Object Repository, 2000. Available at http://www-db.Stanford.edu/lore/release/.

[MW99] J. McHugh and J. Widom. Compile-time path expansion in Lore. In *Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats. Available at http://dbpubs.stanford.edu:8090/pub/1998-14*, 1999.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange across Heterogeneous Information Sources. In *11th Conference on Data Engineering*, pages 251–260, 1995.

[PV99] Y. Papakonstantinou and V. Vassalos. Query rewriting using semistructured views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data. Available at http://www-db.stanford.edu/pub/papers*, 1999.

[RM01] F. Rizzolo and A. Mendelzon. Storing XML data with ToXin. In *Fourth International Workshop on the Web and Databases (in conjunction with ACM SIGMOD 2001). Available at http://www.cs.toronto.edu/~flavio*, 2001.

[Rog02] Y. Rogers. A query processor for XML documents. Master's thesis, Department of Computer Science, University of Toronto, 2002.

[Sah00] A. Sahuguet. Kweelt, the making-of: Mistakes made and lessons learned. Technical report, Department of Computer and Information Science, University of Pennsylvania. Available at db.cis.upenn.edu/DL/kweelt-TR.ps.gz, 2000.

[SKWW00] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *WebDB (Informal Proceedings)*, pages 47–52, 2000.

[STZ+99] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 302–314, 1999.

[Suc] D. Suciu. Semistructured data and XML. International Conference on Foundations of Data Organization, Invited Talk, 1998.

[SYU99] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and retrieval of XML documents using Object-Relational databases. In *Database and Expert Systems Applications*, pages 206–217, 1999.

[TDCZ00] F. Tian, D. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative XML storage strategies. Technical report, Department of Computer Science, University of Wisconsin. Available at www.cs.wisc.edu/ czhang/doc/publications/feng6page.pdf, 2000.

[WL98] K. Wang and H. Liu. Discovering typical structures of documents: A road map approach. In *1st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 146–154, 1998.

[WL00] K. Wang and H. Liu. Discovering structural association of semistructured data. *Knowledge and Data Engineering*, 12(2):353–371, 2000.

[WLOT01] W. Wang, M. Lee, B. Ooi, and K. Tan. XStorM: A scalable storage mapping scheme for XML data. In *World Wide Web Journal*, pages 101–120, 2001.

[XML] XML Schema specification. World Wide Web Consortium, 2001. Available at http://www.w3.org/TR/xmlschema-0.

[XQL] XML query language - XQL. World Wide Web Consortium, 1998. Available at http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[xqua] XQuery 1.0: An XML Query Language. World Wide Web Consortium, 2001. Working Draft. Available at http://www.w3c.org/TR/xquery.

[XQub] XQuery 1.0 Formal Semantics. World Wide Web Consortium. Working Draft, 2001. Available at http://www.w3c.org/TR/query-semantics.

[YASU01] M. Yoshikawa, O. Amagasa, T. Shimura, and S. Uemura. XRel: A path-based approach to storage and retrieval of XML documents using relational databases. In *ACM Transactions On Internet Technology*, volume 1(1):110-141, 2001.

[ZGM99] Y. Zhuge and H. Garcia-Molina. Self-maintainability of graph-structured views. Technical report, Stanford University. Available at http://www-db.stanford.edu/ zhuge/papers/self.ps, 1999.