XML-to-SQL Query Translation

By

Rajasekar Krishnamurthy

A dissertation submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

2004

Abstract

Developing techniques for managing and querying the growing body of XML data is becomingly increasingly important. A popular approach to evaluating XML queries is to translate them to relational queries and then to use a relational database system to evaluate the result.

The XML and relational data models are significantly different, and as a result, the corresponding query languages (XQuery and SQL respectively) also differ significantly. This mismatch raises some interesting questions: (i) From a functionality perspective, is it possible to handle all XML data sets using this approach or are there any fundamental limitations in SQL that create problems? (ii) From a performance perspective, are there any implications on the quality of the SQL queries produced due to this mismatch between the two data models? In this thesis, we address the above two questions in two different scenarios: XML storage and XML publishing. In the former, the goal is to use relational databases to store and query existing XML data, while in the latter, existing relational data is exported as XML.

We demonstrate that it is possible to translate path expression queries (an important class of XML queries) into a single SQL query, even in the presence of recursion in the XML schema and the XML query. We then show that the SQL queries output by previously published algorithms often blindly reflect the hierarchical nature of the XML schema, even when it is clearly unnecessary. We present algorithms that avoid this problem by using additional semantic information intelligently. Since the form and nature of semantic information available differs for the XML storage and XML publishing scenarios, we need different mechanisms for achieving this goal in the two scenarios.

Experiments with a commercial relational database system show that the SQL queries output by our algorithms can be far more efficient than the queries output by previous translation algorithms.

Acknowledgements

I thank my advisor Jeff Naughton for all his help and mentoring during my graduate study. His invaluable advice and constant guidance has helped me mature as a researcher in the last few years. He gave me complete freedom to work on topics that interested me. He also taught me several important things about doing research such as focusing on the core aspects of any problem and making the correct simplifying assumptions during this process. He is also primarily responsible for improving my presentation skills: both written and oral.

I would like to thank David DeWitt and Raghu Ramakrishnan for their support and encouragement, especially during my job search. I would also like to thank Jin-yi Cai for the technical discussions we had. I would like to thank David DeWitt, Raghu Ramakrishnan, Jin-yi Cai and Dharmaraj Veeramani for being on my committee.

Special thanks are due for Raghav Kaushik and Venkat Chakaravarthy for working with me. I really enjoyed all the fruitful discussions we had and without their collaboration this dissertation would not be in its present form.

I would like to thank Jayavel Shanmugasundaram and Eugene Shekita for introducing me to the problem of querying XML data using RDBMS during my internship at IBM-Almaden. Jayavel was also supportive and very helpful during my job search.

During my graduate study, I had the pleasure of working with a lot of people — Ashraf Aboulnaga, Jennifer Beckham, Josef Burger, Venkatesan Chakaravarthy, Jianjun Chen, David DeWitt, Leonidas Galanis, Alan Halverson, Jaewoo Kang, Raghav Kaushik, Jerry Kiernan, Ameet Kini, Qiong Luo, Jeffrey Naughton, Naveen Prakash, Raghu Ramakrishnan, Ravishankar Ramamurthy, Ajith Nagaraja Rao, Jayavel Shanmugasundaram,

Eugene Shekita, Feng Tian, Stratis Viglas, Yuan Wang and Chun Zhang. I really enjoyed working with them and learned a lot from all these interactions. I would also like to thank Neoklis Polyzotis for his feedback on my work and Brian Forney for the long conversations we had.

I had a great bunch of friends during my stay in Madison: Charles, Koushik, Muthian, Prabu, Raghav, Ram, Ravi, Sricharan, Venkat and Venkatanand. I would like to thank all of them for their friendship and for making my stay in Madison pleasurable.

I would like to thank Balaji for all his support and encouragement and for being a great friend. I would also like to thank Karthik, Murali and Sriram for their friendship.

Above all, I would like to thank my family: my parents, Balu, Janani, Srini, Malini and Srikar, for their love and support.

Contents

Abstract **i**

Acknowledgements **iii**

1 Introduction **1**

1.1 Schema-Based XML Storage 3

1.2 XML Publishing 6

2 Background and Current State of the Art **10**

2.1 XML Publishing 12

2.1.1 XML View Definition 13

2.1.2 Materializing the XML View 14

2.1.3 Evaluating XML Queries 14

2.1.4 Open Problems 17

2.1.5 Recursive XML View Schema and Linear Recursion in SQL . . . 19

2.2 Schema-Oblivious XML Storage 21

2.2.1 Relational Schema Design 22

2.2.2 Query Translation 24

2.2.3 Summary and Open Problems 25

2.3 Schema-Based XML Storage 28

2.3.1 Relational Schema Selection 28

2.3.2 Query Translation 29

2.3.3 Discussion and Open Problems 31

2.4 Summary 34

3 Schema-based XML storage: Recursive Schemas and Queries 36

3.1 Formal Model 37

3.1.1 XML Schema Graph 37

3.1.2 XML to Relational Mappings 38

3.1.3 Path Expression Queries 43

3.2 Query Translation Over Recursive XML schemas 43

3.2.1 PathId stage 45

3.2.2 SQLGen stage 53

3.2.3 A Note on Query Translation Time 64

3.2.4 Extensions to the Algorithm 65

3.3 Related Work 66

3.4 Summary 67

4 Mapping-aware Query Translation 68

4.1 Motivation for mapping_aware techniques 69

4.2 Exploiting the "lossless from XML" constraint for Tree XML Schemas . . 73

4.2.1 Basic Idea behind the Algorithm 73

4.2.2 Terminology 78

4.2.3 The Pruning Stage 80

4.2.4 The SQLGen Stage 82

4.2.5 Some Alternative Solutions 87

4.3 Exploiting the "lossless from XML" constraint for complex XML Schemas 90

4.3.1 Combinability for Complex Schema 90

4.3.2 The Pruning Stage 95

4.3.3 Schema-Oblivious Storage 99
4.4 Experimental Study 103
4.5 Summary 105

5 Generating Efficient SQL Queries in the Publishing Scenario 106

5.1 Motivation 110
5.2 Problem Definition 113
5.3 The SQL Optimization approach 118
5.3.1 Previous work on Relational Query Minimization 118
5.3.2 Impact on the SQL Optimization approach 120
5.4 Intelligent Query Translation 121
5.4.1 Outline of our approach 122
5.4.2 Bijective mappings 124
5.4.3 Prefix Elimination Optimality 125
5.4.4 The query translation algorithm 126
5.5 The constraint_aware approach 129
5.5.1 Terminology 129
5.5.2 Precomputation Phase 130
5.5.3 Run-Time Query Translation Algorithm 137
5.5.4 Analysis 140
5.6 Extensions to More General Cases 145
5.6.1 Path Expression Queries Involving Non-Leaf Nodes 145
5.6.2 Beyond Path Expressions 146
5.6.3 Beyond Bijective Mappings 147
5.7 Summary 148

6 Conclusions and Future Work **150**

Bibliography **152**

A Queries used in the experiments in Chapter 4.4 **162**

List of Tables

1 Summary of various published techniques 11

2 Execution time of translation algorithm 64

3 Part of Relational Schema for ADEX dataset 102

4 Part of Relational Schema for XMark dataset 102

5 Relative performance improvement obtained by the mapping_aware algorithm 103

List of Figures

1 High-Level Taxonomy of interaction between XML and RDBMS 2

2 Using RDBMS to store and query XML data 4

3 Publishing Relational data as XML 7

4 Focus of published solutions 11

5 Sample XML-to-Relational mapping schema 33

6 Sample XML-to-Relational mapping schema 37

7 SQL Query associated with a path p in the XML schema 41

8 Query Translation Algorithm handling recursion in XML schema and query 44

9 Example to illustrate PathId 44

10 Example to illustrate duplicate counting in PathId 45

11 PathId Algorithm 48

12 SQLGen Algorithm 54

13 Sample recursive schema to explain the SQLGen algorithm 55

14 SQLGen Algorithm for a DAG Component 56

15 SQLGen Algorithm for a Recursive Component 60

16 SQL query output by the XML_to_SQL algorithm for $Q = /E0//E10$ . . 62

17 XMark benchmark schema and a sample relational decomposition 70

18 Query Translation Algorithm using the "lossless from XML" constraint . 73

19 Result of PathId stage for Q_1 and Q_2 74

20 Pruning stage for tree XML schema 80

21 Example mapping S_1 to explain the pruning algorithm 85

22 Modified pruning stage 88

23 Example mapping S_2 to explain issues in defining combinability for complex schema 91
24 Pruning stage for recursive XML schema 95
25 Examples to illustrate the mapping_aware algorithm 96
26 Examples to illustrate the mapping_aware algorithm 97
27 XMark schema mapped to the Edge relation 100
28 Part of ADEX XML schema 102
29 Stages in using an RDBMS to evaluate an XML query 107
30 Sample relational schema and corresponding XML view 110
31 View Definition expressed as an XQuery query over the default view . . 115
32 Example view to illustrate (non)bijective mappings 123
33 Algorithm to check if a relational column is bijectively mapped 134
34 Algorithm to compute *lda* information 135
35 constraint_aware query translation algorithm for path expression queries . 138
36 Prefix-Elimination phase 139

Chapter 1

Introduction

XML is emerging as the universal standard format for data exchange, and as a result, XML data management is becoming increasingly important. Relational database systems (RDBMSs) have dominated the commercial data management space for several decades. So, using RDBMSs to manage XML data is an attractive option. Unfortunately, the XML data model differs substantially from the relational data model, so using RDBMSs to support XML data poses a number of interesting and challenging problems. This thesis focuses on query translation, which lies at the core of managing XML data using relational database systems. Because SQL is the query language used by all relational database systems, we refer to this query translation problem as "XML-to-SQL query translation".

The main scenarios in which XML-to-SQL query translation is required are shown in Figure 1. In XML storage, the goal is to use an RDBMS to store and query existing XML data. Based on whether or not the XML schema is used to decide on the corresponding relational schema, the techniques can be classified as schema-based or schema-oblivious XML storage. By contrast, in XML Publishing, the goal is to treat existing relational data sets as if they were XML. In other words, an XML view of the relational data set is defined and XML queries are posed over this view.

Note that the above scenarios have the common property that the data is stored (physically) in a relational database system and users (or applications) write queries over

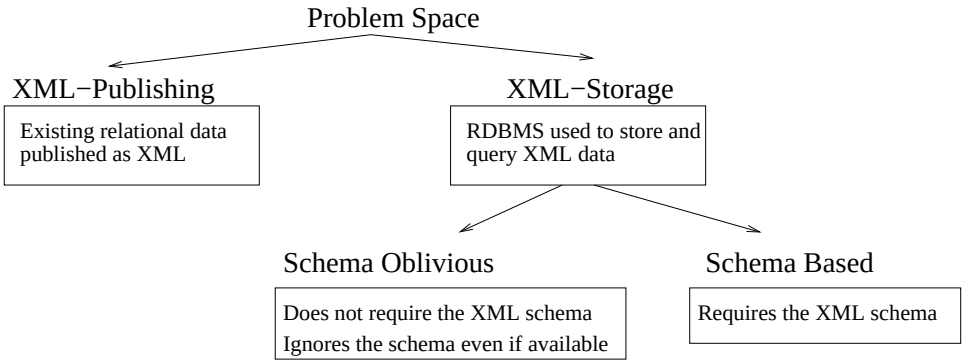

Figure 1: High-Level Taxonomy of interaction between XML and RDBMS

an XML view of this data. We believe that the main advantages of this approach are as follows:

- In the XML storage scenario, where the input data is a collection of XML documents, using an RDBMS to store and query the data set allows us to leverage the advancements made in relational technology in the last few decades. The vast amount of knowledge accumulated about the relational model and the presence of fairly mature commercial RDBMSs can be tapped to handle XML workloads.

- A large amount of commercial data is currently stored in RDBMSs. XML originated as a data exchange format. In this role, organizations need to be able to exchange data, which is currently primarily relational. This makes the XML publishing scenario important.

- In a few years, organizations can be expected to receive and handle large amounts of data in both relational and XML format. This implies that data management capabilities across both types of data is essential. Using a relational database to handle both types of data is an attractive option for being able to seamlessly query across the entire data set.

In this thesis, we present solutions for the XML-to-SQL query translation problem

in the schema-based XML storage and XML publishing scenarios. We focus on a class of XML queries called path expression queries, which are the building block of XQuery. In the next two sections, we describe the schema-based XML storage and XML publishing scenarios in more detail and also outline the contributions of this thesis in the corresponding scenario.

1.1 Schema-Based XML Storage

A popular approach to store and query XML data is to use existing relational database systems. Two main advantages of using RDBMS to handle XML workloads are: (i) We can leverage decades of research and development in relational database technology. If we can efficiently cross the data model boundary, then we get all the advantages of mature commercial relational database systems. (ii) In the future, applications are likely to produce and consume both relational and XML data. So, database systems will need to support both data formats. Using a single storage engine and query processor for both data formats is likely to be more efficient than having two loosely coupled database systems. In this direction, using existing relational database systems for this purpose is a promising approach.

The various steps involved in using a relational database for supporting XML workloads are shown in Figure 2. An XML-enablement layer on top of the relational query processor handles the conversion across data models. Note that this layer can either be a part of the relational database or it can be middleware. There are two main components in the XML-enablement layer: the *shredder* and the *query translator*. The functions of the two components are described below.

- Given an XML schema, the shredder decides on a good relational schema and

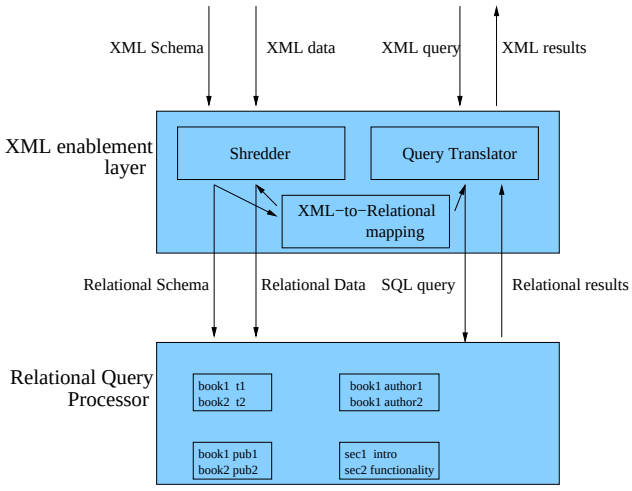

Figure 2: Using RDBMS to store and query XML data

creates the appropriate tables. The association between the XML schema and the relational schema is maintained in (what we refer to as) the *XML-to-Relational mapping*. In this step, the shredder may also use other information such as XML data statistics and query workload information.

- The next step is to load XML data into the database. Given XML data conforming to the XML schema, the shredder obtains the corresponding relational data (based on the mapping) and loads them into the RDBMS.

- Given an XML query, the query translator obtains an equivalent SQL query based on the mapping. This query is executed by the relational query processor. The resulting relational results are converted into XML by the query translator and returned to the user.

There has been a lot of research in the last few years on developing efficient algorithms for both the shredding and the query translation components. A description of the various published solutions and the support present in current commercial relational database systems is given in Chapter 2. While a lot of forward progress has been made in this

direction, we see that some basic questions are still open. Some of the contributions of this thesis are as follows.

- **Recursive XML Schemas and Recursive XML queries:** In a recent study of real-world XML schemas [Cho02], out of the 60 schemas analyzed, more than half (35) of them were recursive, which suggests that recursive XML schemas are common in practice. Furthermore, recursion is ubiquitous in XML queries, as it appears in any path expression query that uses the descendant axis (//). All these suggest that it is important to handle recursive XML schemas and recursive XML queries for schema-based XML storage. But, there is no published XML-to-SQL query translation algorithm that handles recursive XML schemas.

  In Chapter 3, we present a generic algorithm XML_to_SQL that translates path expression queries to SQL in the presence of recursion in the XML schemas and queries. We show how the support for linear recursion in SQL99 is sufficient for this purpose. An interesting aspect of this algorithm is the use of the SQL99 *with* construct to handle recursive queries even over non-recursive schemas.

- **Quality of SQL queries produced by query translator:** Translating XML queries to SQL involves translating queries over hierarchical schemas into queries over flat relational schemas. This turns out to be problematic — a closer look at the queries generated by the published translation algorithms shows that the hierarchical nature of the exported XML schema is often blindly reflected in the generated SQL query, even when this is clearly not necessary. As a result, in many cases even simple path expression queries result in unnecessarily complex SQL queries. This problem is aggravated when the input XML query includes a traversal of the descendant axis (//), because it does not have a simple equivalent

in SQL.

A natural question to ask next is whether the phenomenon of large, complex SQL queries arising from simple XML queries is avoidable, or if it is intrinsic due to the mismatch in data models. We show by example in Chapter 4 that complex SQL is not necessary in many cases — while the SQL generated by the published translation algorithms is complex, usually there is a much simpler equivalent SQL query. This observation motivated us to search for techniques that make use of readily available semantic information to improve the quality of the generated SQL. In particular, the fact that all the relational data resulted from the shredding of XML documents that conformed to the given XML schema allows us to use the XML-to-Relational mapping information in an intelligent fashion. We extend the XML_to_SQL algorithm from Chapter 3 to use the semantic information present in the XML-to-Relational mapping and generate efficient SQL queries. The details of this *mapping_aware* algorithm are given in Chapter 4.

1.2 XML Publishing

While XML is growing into the universal data exchange format, a large fraction of existing data is stored in relational databases. This motivates the need for publishing existing relational data as XML (XML Publishing). Here, an XML view of the relational data set is defined and XML queries are posed over this view. The various steps involved in this process are shown in Figure 3. Notice how the shredder in the XML storage scenario is replaced by a *view constructor* in the publishing scenario as the data is already present in the RDBMS. A number of view definition mechanisms and query translation algorithms have been proposed for this setting in the published literature and these are described in

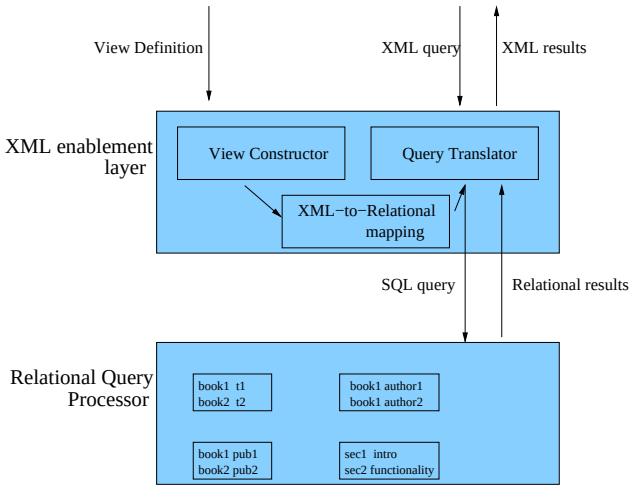

Figure 3: Publishing Relational data as XML

Chapter 2.1.

At a high level, XML-to-SQL query translation in the XML Publishing scenario is very similar to the corresponding problem in the schema-based XML storage scenario. In fact, as discussed in [SSK+01], it is possible to view the latter as a subset of the XML publishing scenario. To see this, notice that once XML data is shredded into relations, we can view the resulting data as if it were pre-existing relational data. Now by defining a reconstruction view that mirrors the XML-to-relational mapping used to shred the data, the query translation algorithms in the XML publishing domain are directly applicable for the schema-based XML storage domain. Indeed, this is the approach adopted in commercial relational database systems such as Oracle XML DB [OXD], Microsoft SQL Server 2000 SQLXML [SXM] and IBM DB2 XML Extender [DB2] and also in research literature, such as [BFRS02, SSK+01].

The XML_to_SQL algorithm presented in Chapter 3 for handling recursive XML schema and queries can be adapted for the XML Publishing scenario as well. The details of the algorithm will change based on the view definition language, but the main ideas about how to handle recursive schemas, directed acyclic graph (DAG) schemas and

recursive queries remain the same.

On the other hand, the optimizations we propose in the *mapping_aware* algorithm (Chapter 4) to improve the quality of the final SQL queries use semantic information present in the XML-to-Relational mapping. Also, in the XML storage scenario, the data in the RDBMS originates from an XML document and there is some semantic information associated with this. For example, every decomposition proposed in the literature stores each XML element "exactly once" in the relational tables. The *mapping_aware* algorithm in Chapter 4 uses all this information in reasoning about equivalent SQL queries. By contrast, in the XML publishing domain, existing relational data may be mapped to XML in uncontrolled ways – some parts of the relational data may be exported multiple times in the XML view, while other parts may not be exported at all. This basic difference between the XML storage and the XML publishing scenarios makes mapping_aware query translation a far more difficult task in the XML publishing domain. One way to perform mapping_aware translation in this case is to use the constraints on the underlying relational data along with the mapping information. We follow this approach in this thesis.

As we show in Chapter 5, translating path expression queries into SQL over tree XML view definitions is closely related to the problem of relational query minimization under bag semantics. The techniques for query minimization in the published literature rely on algorithms for query containment or query equivalence. Unfortunately, these problems become intractable even in simple scenarios.

In Chapter 5, we present our approach in which we identify a subset of tree XML-to-relational mappings called *bijective* mappings. Bijective mappings have the desirable property that they can be optimized using containment and equivalence algorithms under set semantics instead of multiset semantics. By using the fact that the XML-to-Relational

mapping determines the class of SQL queries that are likely to be output by the XML-to-SQL query translation algorithm, we precompute some summary information using the relational integrity constraints. We use this information during runtime query translation to generate efficient SQL queries. Our *constraint_aware* algorithm works correctly even over non-bijective mappings; it identifies the bijective portions of the mapping and performs more efficient query translation in those parts.

Roadmap

The rest of the thesis is organized as follows. In Chapter 2, we present a survey on XML-to-SQL query translation algorithms. In Chapters 3 and 4, we look at the query translation problem for the schema-based XML storage scenario. In Chapter 3, we present the XML_to_SQL algorithm for translating path expression queries into SQL over recursive XML-to-Relational mappings. We extend this algorithm in Chapter 4 to make it *mapping_aware*, improving the quality of the final SQL queries. In Chapter 5, we consider the query translation problem for the XML publishing domain. We present the *constraint_aware* algorithm that uses relational integrity constraints to generate efficient SQL queries. Finally, we present our conclusions and discuss future work in Chapter 6.

Chapter 2

Background and Current State of the Art

Beginning in 1999, the database research literature has seen an explosion of publications with the goal of using an RDBMS to store and/or query XML data. The problems addressed and solved in this area are diverse. Some publications deal with using an RDBMS to store XML data; others deal with exporting existing relational data in an XML view. The papers use a wide variety of XML query languages, including subsets of XQuery, XML-QL, XPath, and even "one-off" new proposals; they use a wide variety of languages or ad-hoc constructs to map between the relational and XML schema; and they differ widely in what they "push to SQL" and what they evaluate in middleware.

This diversity renders it difficult to know how the various results presented fit together, and even makes it hard to know what if any open problems remain. As a first step to rectifying this situation, we present a classification of the problem space and discuss how almost 40 papers fit into this classification. As a result of this study, we find that some basic questions are still open. We also describe how some of these questions are answered by the work presented in this thesis.

In this chapter, we use the classification in Figure 1 to characterize almost 40 published solutions to the XML-to-SQL query translation problem. The various published techniques are summarized in Table 1, where for each technique we identify the scenario

Table 1: Summary of various published techniques

Technique	Scenario	Subproblems solved	Class of XML Schema considered	Class of XML Queries handled
XPeranto	XP/GAV	VD,QT	tree	XQuery
SilkRoute	XP/GAV	VD,QT	tree	XML-QL
Rolex	XP/GAV	QT	tree	XSLT
[JMS02]	XP/GAV	QT	tree	XSLT
[BCF+02]	XP/GAV	VD	recursive	-
Oracle XML DB	XP/GAV, XS/SB	VD,SS,QT	recursive	SQL/XML restricted XPath[1]
SQL Server 2000 SQLXML	XP/GAV, XS/SB	VD,SS,QT	bounded depth recursive	restricted XPath[2]
DB2 XML Extender	XP/GAV, XS/SB	VD,QT	non-recursive	SQL extensions through UDFs
Agora	XP/LAV	QT	non-recursive	XQuery
MARS	XP/GAV + XP/LAV	QT	non-recursive	XQuery
STORED	XS/SO	SS,QT	all	STORED
Edge	XS/SO	SS,QT	all	path expressions
Monet	XS/SO	SS	all	-
XRel	XS/SO	SS,QT	all	path expressions
[TVB+02]	XS/SO	SS,QT	all	order-based queries
Dynamic intervals [DTCO03]	XS/SO	QT	all	XQuery
[ML03, STZ+99]	XS/SB	SS	recursive	-
[BFRS02, HSJJ02] [KM00, LC00, RP02]	XS/SB	SS	tree	-

XP/GAV: XML Publishing, Global-as-view XP/LAV: XML Publishing, Local-as-view
XS/SO: XML Storage, schema-oblivious XS/SB: XML Storage, schema-based
QT: Query Translation VD: View Definition SS: Storage scheme
restricted XPath[1]: child and attribute axes
restricted XPath[2]: child, attribute, self and parent axes

	Tree Schema		Recursive Schema	
Simple Queries (path expressions)	XP	a lot	XP	none
	XS/SO	a lot	XS/SO	a lot
	XS/SB	a lot	XS/SB	none
Complex Queries	XP	a lot	XP	none
	XS/SO	some	XS/SO	some
	XS/SB	a lot	XS/SB	none

Figure 4: Focus of published solutions

solved and the part of the problem handled within that scenario. We will look at each of these in more detail in the rest of this section. In addition to the characteristics from our broad classification, the table also reports, for each solution, the class of schema considered, the class of XML queries handled, whether it uses the "global as view" or "local as view" approach (if the XML publishing problem is addressed), and what subproblems are solved. A summary of the focus of published solutions is given in Figure 4.

The rest of this chapter is organized as follows. We survey known algorithms in the published literature for XML-publishing, schema-oblivious XML storage and schema-based XML storage in Sections 2.1, 2.2, and 2.3 respectively. For each scenario, we first survey the solutions that have been proposed in published literature, and discuss problems that remain open. When we look at XML support in commercial RDBMS as part of this survey, we will restrict our discussion to those features that are relevant to XML-to-SQL query translation.

2.1 XML Publishing

The following tasks arise in the context of allowing applications to query existing relational data as if it were XML:

- Defining an XML view of relational data.
- Materializing the XML view.
- Evaluating an XML query by composing it with the view.

In XML query languages like XPath and XQuery, part of the query evaluation may involve reconstructing the subtrees rooted at certain elements, which are identified by other parts of the query. Notice how materializing an XML view is a special case of this

situation, where the entire tree (XML document) is reconstructed. In general, solutions to materialize an XML view are used as a subroutine during query evaluation.

2.1.1 XML View Definition

In XPeranto [SKS+01, SSB+00], SilkRoute [FMS01, FTS00] and Rolex [BGK+02], the view definition languages permit definition of tree XML views over the relational data. In [BCF+02], XML views corresponding to recursive XML schema (recursive XML view schema) are allowed.

In Oracle XML DB [OXD] and Microsoft SQL Server 2000 SQLXML [SXM], an annotated XSD XML schema is used to define the XML view. Recursive XML views are supported in XML DB. In SQLXML, along with non-recursive views, there is support for a limited number of depths of recursion using the max-depth annotation. In IBM DB2 XML Extender [DB2], a Document Access Definition (DAD) file is used to define a non-recursive XML view. IBM XML for Tables [XTa] provides an XML view of relational tables and is based on the Xperanto [SSB+00] project.

In the above approaches, the XML view is defined as a view over the relational schema. In a data integration context, Agora [MFK01] uses the local-as-view approach (LAV), where the local source's schema are described as views over the global schema. Toward this purpose, they describe a generic, virtual relational schema closely modeling the generic structure of an XML document. The local relational schema is then defined as views over this generic, virtual schema. Contrast this with the other approaches where the XML view (global schema) is defined as a view over the relational schema (local schema). This is referred to as the global-as-view approach (GAV).

In Mars [DT03a], the authors consider the scenario where both GAV-style and LAV-style views are present. The focus of [DT03a, MFK01] is on non-recursive XML view schema.

2.1.2 Materializing the XML View

In XPeranto [SSB+00], the XML view is materialized by pushing down a single "outer union" query into the relational engine, whereas in SilkRoute [FMS01], the middleware system issues several SQL queries to materialize the view. In [BCF+02], techniques for materializing a recursive XML view schema are discussed. They argue that since SQL supports only linear recursion, the support for recursion in SQL is insufficient for this purpose. Instead, the recursive materialization is performed in middleware by repeatedly unrolling a fixed number of levels at a time. We discuss this in more detail in Section 2.1.5, where we show that the limited support for recursion in SQL is not an obstacle in handling recursive XML views. Later in Chapter 3.2.4, we present an algorithm to materialize recursive XML views using a single SQL query.

2.1.3 Evaluating XML Queries

In XPeranto [SKS+01], a general framework for processing arbitrarily complex XQuery queries over XML views is presented. They describe their XQGM query representation, an extension of a SQL internal query representation called the Query Graph Model (QGM). The XQuery query is converted to an XQGM representation and composed with the view definition. Rewrite optimizations are performed to eliminate the construction of intermediate XML fragments and to push down predicates. The modified XQGM is translated into a single SQL query to be evaluated inside the relational

engine.

In SilkRoute [FMS01], a sound and complete query composition algorithm is presented for evaluating a given XML-QL query over the XML view. An XML-QL query consists of patterns, filters and constructors. Their composition technique evaluates the patterns on the view definition at compile-time to obtain a modified XML view, and the filters and constructors are evaluated at run-time using the modified XML view.

In [JMS02], the authors present an algorithm for translating XSLT programs into efficient SQL queries. The main focus of the paper is bridging the gap between XSLT's functional, recursive paradigm, and SQL's declarative paradigm. They also identify a new class of optimizations that need to be done either by the translator or by the relational engine, in order to optimize the kind of SQL queries that result from such a translation. In Rolex [LBN03], a view composition algorithm for composing an XSLT stylesheet with an XML view definition to produce a new XML view definition is presented. They differ from [JMS02] mainly in the following ways: (1) they produce an XML view query rather than an SQL query, (2) they address additional features of XSLT like priority and recursive templates.

As part of the Rainbow system, in [ZPR02], the authors discuss processing and optimization of XQuery queries. They describe the XML Algebra Tree (XAT) algebra for modeling XQuery expressions, propose rewriting rules to optimize XQuery queries by canceling operators and describe a cutting algorithm that removes redundant operators and relational columns from the XAT. However, the final XML to SQL query generation is not discussed.

We note here that in Rolex [BGK+02], the world view is changed so that a relational system provides a virtual DOM interface to the application. The input in this case is not a single XML query but a series of navigation operations on the DOM tree that needs to

be evaluated on the underlying relational data.

The Agora [MFK01] project uses an LAV approach and provides an algorithm for translating XQuery FLWR expressions into SQL. Their algorithm has two main steps — translating the XML query into a SQL query on the generic, virtual relational schema, and rewriting this SQL query into a SQL query over the real relational schema. In the first step, they cross the language gap from XQuery to SQL, and in the second step they use prior work on answering queries using views.

In MARS [DT03a, DT03b], a technique for translating XQuery queries into SQL is given, when both GAV-style and LAV-style views are present. The basic idea is to compile the queries, views and constraints from XML into the relational framework, producing relational queries and constraints. Then, a Chase and BackChase (C&B) algorithm is used to find all minimal reformulations of the relational queries under the relational integrity constraints. Using a cost-estimator, the optimal query among the minimal reformulations is obtained, which can then be executed. The MARS system also exploits integrity constraints on both the relational and XML data. The system achieves the combined effect of rewriting-with-views, composition-with-views, and query minimization under integrity constraints.

Oracle XML DB [OXD] provides an implementation of the majority of the operators that will be incorporated into the forthcoming SQL/XML standard [INC]. SQL/XML is an extension to SQL, using functions and operators, to include processing of XML data in relational stores. The SQL/XML operators [EM02] make it possible to query and access XML content as part of normal SQL operations and also provide methods for generating XML from the result of an SQL Select statement. The SQL/XML operators allow XPath expressions to be used to access a subset of the nodes in the XML view. In XML DB, the approach is to translate the XPath expression into an equivalent SQL

query through a query re-write step that uses the XML view definition. In the current release (Oracle9i Release 2), simple path expressions with no wild cards or descendant axes (//) get rewritten. Predicates are supported and get rewritten into SQL predicates. The XPath axes supported are the child and attribute axis.

Microsoft SQL Server 2000 SQLXML [SXM] supports the evaluation of XPath queries over the annotated XML Schema. The XPath query together with the annotated schema is translated into a *FOR XML* explicit query that only returns the XML data that is required by the query. Here, *FOR XML* is a new SQL select statement extension provided by SQL Server. In the current release (SQLXML 3.0), the attribute, child, parent and self axes are supported, along with predicates and XPath variables.

In IBM DB2 XML Extender [DB2], powerful user-defined functions (UDFs) are provided to store and retrieve XML documents in XML columns, as well as to extract XML element or attribute values. Since it does not provide support for any XML query languages, we will not discuss XML Extender any further in this discussion.

2.1.4 Open Problems

We see that a number of open problems remain. We describe them below and also point out how some of these issues are addressed in this thesis.

1. With the exception of [BCF+02, OXD], the above work considers only non-recursive XML views of relational data. While Oracle XML DB [OXD] supports path expression queries with the child and attribute axes over recursive views, it does not support the descendant (//) axis. Translating XML queries (with the // axis) over recursive view schema remains open. In [BCF+02], the problem of materializing recursive XML view schema is considered. However, as we have mentioned, that

work does not use SQL support for recursion, simulating recursion in middleware instead. The reason for this given by the authors is that the limited form of recursion supported by SQL cannot handle the forms of recursion that arise in with recursive XML schema. We return to this question at the end of this section. The following were open questions in the context of SQL support for recursion:

- What is the class of queries/view schema for which the current support for recursion in SQL are adequate?
- If there are cases for which SQL support for recursion is inadequate, how do we best leverage this support? (Instead of completely simulating recursion in middleware.)

In Chapter 3, we present an algorithm to translate path expression queries into SQL, when the XML schema and the XML query may be recursive. We also show how recursive XML views can be materialized using a single SQL query. This demonstrates that SQL support for recursion is sufficient for the class of GAV-style views considered in the published literature.

2. Any query translation algorithm can be evaluated by two metrics: its functionality, in terms of the class of XML queries handled; and its performance, in terms of the efficiency of the resulting SQL query. Most of the translation algorithms have not been evaluated thoroughly by either metric, which gives rise to a number of open research problems.

   - Functionality: Among the GAV-style approaches, except XPeranto, all the above discussed work deals with languages other than XQuery. Even in the case of XPeranto, the class of XQuery handled is unclear from [SKS+01]. It

would be interesting to precisely characterize the class of XQuery queries that can be translated by the methods currently in the literature.

- Performance: There has been almost no emphasis on the quality of the SQL queries produced by the query translation algorithms.

In Chapter 5, we look at the quality of the SQL queries output by previously published query translation algorithms. We see that there is a lot of scope for improving the quality of the SQL queries and present an algorithm that translates path expression queries into efficient SQL queries.

3. GAV vs. LAV: While for the GAV-style approaches, XML-to-SQL query translation corresponds to view composition, for the LAV-style approaches it corresponds to answering queries with views. It is not clear for what class of XML views the equivalent query rewriting problem has published solutions. As pointed out in [MFK01], state-of-the-art query rewriting algorithms for SQL semantics do not efficiently handle arbitrary levels of nesting, grouping, etc. Similarly, [DT03a] works under set-semantics and so cannot handle certain classes of XML view schema and aggregation in XML queries. Comparing across the three different approaches — GAV, LAV and GAV+LAV, in terms of both functionality and performance is an open issue.

2.1.5 Recursive XML View Schema and Linear Recursion in SQL

In this section we return to the problem of recursive XML view schema and whether or not they can be handled by the support for recursion currently provided by SQL.

Consider the problem of materializing a recursive XML view schema. In [BCF+02], it is mentioned that even though SQL supports linear recursion, this is not sufficient for materializing a recursive XML view. The reason for this is not elaborated in the paper. The definition of an XML view has two main components to it: the view definition language and the XML schema of the resulting view. Hence, it must be the case that either the XML schema of the view or the view definition language is more complex than what SQL linear recursion can support. Clearly, if the view definition language is complex enough (say the parent-child relationship is defined using non-linear recursion), linear recursion in SQL will not suffice. However, most view definition languages proposed define parent-child relationships through much simpler conditions (such as conjunctive queries). The question arises whether SQL linear recursion is sufficient for these view definition languages, for arbitrary XML schema.

In [Cho02], the notion of linear and non-linear recursive DTDs is introduced. The natural question here is whether the notions of linear recursion in SQL and DTDs correspond. It turns out that the definition of non-linear recursive schema in [Cho02] has nothing to do with the traditional Datalog notion of linear and non-linear recursion [AHV95]. For example, consider a classical part-subpart database. Suppose that the DTD rule for a part element is: part → pname, part*.

According to [Cho02], this is a non-linear recursive rule as a part element can derive multiple part sub-elements. Hence, the entire DTD is non-linear recursive. Indeed, it can be shown that this DTD is not equivalent to any linear-recursive DTD. Now, suppose the underlying relational schema has two relations, Part and Subpart with the columns: (partid,pname) and (partid,subpartid) respectively. Now, the following SQL query extracts all data necessary to materialize the XML view:

```
WITH RECURSIVE AllParts(partid,pname,rtolpath) as (
          select partid,pname,''
```

```
            from Part(partid,pname)
         union all
            select P.partid,P.pname,rtolpath+A.partid
            from AllParts A, Subpart S, Part P
            where S.partid = A.partid and S.subpartid = P.partid)
select * from AllParts
```

In the above query, the root-to-leaf path is maintained for each part element through the `rtolpath` column in order to extract the tree structure. Note however that the core SQL query executes the following linear-recursive Datalog program.

AllParts(partid,pname) ← Part(partid,pname)

AllParts(subpartid,subpname) ←

AllParts(partid,pname) Subpart(partid,subpartid) Part(subpartid,subpname)

So, we see that a non-linear recursive rule in the DTD gets translated into a linear recursive Datalog (SQL) rule. This implies that the notion of linear recursion in DTDs and SQL (Datalog) do not have a direct correspondence. Hence, the class of XML view schema/view definition languages for which SQL linear recursion is adequate to materialize the resulting XML views needs to be examined.

2.2 Schema-Oblivious XML Storage

Recall that in this scenario, the goal is to find a relational schema that works for storing XML documents independent of the presence or absence of a schema. The main problems addressed in this sub-space are:

- Relational schema design: which generic relational schema for XML should be used?
- Query translation algorithms: given a decision for the relational schema, how do

we translate from XML queries to SQL queries.

2.2.1 Relational Schema Design

In STORED [DFS99], given a semi-structured database instance, a STORED mapping is generated automatically using data mining techniques — STORED is a declarative query language proposed for this purpose. This mapping has two parts: a relational schema and an overflow graph for the data not conforming to the relational schema. We classify STORED as a schema-oblivious technique since the data since data inserted in the future is not required to conform to the derived schema. Thus, if an XML document with completely different structure is added to the database, the system sticks to the existing relational schema without any modification whatsoever.

In [FK99], several mapping schemes are proposed. According to the Edge approach, the input XML document is viewed as a graph and each edge of the graph is represented as a tuple in a single table. In a variant known as the Attribute approach, the edge table is horizontally partitioned on the tag name yielding a separate table for each element/attribute. Two other alternatives, the Universal table approach and the Normalized Universal approach are proposed but shown to be inferior to the other two. Hence, we do not discuss these any further.

The binary association approach [SKWW00] is a path-based approach that stores all elements that correspond to a given root-to-leaf path together in a single relation. Parent-child relationships are maintained through parent and child ids.

The XRel approach [YASU01] is another path-based approach. The main difference here is that for each element, the path id corresponding to the root-to-leaf path as well as an interval representing the region covered by the element are stored. The latter is similar

to interval-based schemes for representing inverted lists proposed in [LM01, ZND+01].

In [TVB+02], the focus is on supporting order based queries over XML data. The schema assumed is a modified Edge relation where the path id is stored as in [YASU01], and an extra field for order is also stored. Three schemes for supporting order are discussed.

In [DTCO03], all XML data is stored in a single table containing a tuple for each element, attribute and text node. For an element, the element name and an interval representing the region covered by the element is stored. Analogous information is stored for attributes and text nodes.

There has been extensive work on using inverted lists to evaluate path expression queries by performing containment joins [CVZ+02, JLWO03, LM01, BKS02, AKJP+02, WJLY03, ZND+01]. In [ZND+01], the performance of containment algorithms in an RDBMS and a native XML system are compared. All other strategies are for native XML systems. In order to adapt these inside a relational engine, we would need to add new containment algorithms and novel data structures. The issue of how we extend the relational engine to identify the *use* of these strategies is open. In particular, the question of how the optimizer maps SQL operations into these strategies needs to be addressed.

In [Gru02], a new database index structure called the XPath accelerator is proposed that supports all XPath axes. The preorder and postorder ranks of an element are used to map nodes onto a two-dimensional plane. The evaluation of the XPath axis steps then reduces to processing region queries in this pre/post plane. In [GvKT03], the focus is on exploiting additional properties of the pre/post plane to speedup XPath query evaluation and the Staircase join operator is proposed for this purpose. The focus of [Gru02, GvKT03] is on efficiently supporting the basic operations in a path expression and is complementary to the XML-to-SQL query translation issue.

In Oracle XML DB [OXD] and IBM DB2 XML Extender [DB2], a schema-oblivious way of storing XML data is provided, where the entire XML document is stored using the CLOB data type. Since evaluating XML queries in this case is similar to XML query processing in a native XML database and does not involve XML-to-SQL query translation, we do not discuss this approach any further.

2.2.2 Query Translation

In STORED [DFS99], an algorithm is outlined for translating an input STORED query into SQL. The algorithm uses inversion rules to create a single canonical data instance, intuitively corresponding to a schema. The structural component of the STORED query is then evaluated on this instance to obtain a set of results, for each of which a SQL query is generated incorporating the rest of the STORED query.

In [FK99], a brief overview of how to translate the basic operations in a path expression query to SQL is provided. The operations described are (1) returning an element with its children, (2) selections on values, (3) pattern matching, (4) optional predicates, (5) predicates on attribute names and (6) regular path queries which can be translated into recursive SQL queries.

The binary association method [SKWW00] deals with translating OQL-like queries into SQL. The class of queries they consider roughly corresponds to branching path expression queries in XQuery.

In XRel [YASU01], a core part of XPath called XPathCore is identified and a detailed algorithm for translating such queries into SQL is provided. Since with each element, a path id corresponding to the root-to-leaf path is stored, a simple path expression query like book/section/title gets efficiently evaluated. Instead of performing a join for each

step of the path expression, all elements with a matching path id are extracted. Similar optimizations are proposed for branching path expression queries exploiting both path ids and the interval encoding. We examine this in more detail in Section 2.2.3.

In [TVB+02], algorithms for translating order based path expression queries into SQL are provided. They provide translation procedures for each axis in XPath, as well as for positional predicates. Given a path expression, the algorithm translates one axis at a time in sequence.

The dynamic intervals approach [DTCO03] deals with a larger fragment of XQuery with arbitrarily nested FLWR expressions, element constructors and built-in functions including structural comparisons. The core idea is to begin with static intervals for each element and construct dynamic intervals for XML elements constructed in the query. Several new operators are proposed to efficiently implement the generated SQL queries inside the relational engine. These operators are highly specialized and are similar to operators present in a native XML engine.

2.2.3 Summary and Open Problems

The various schema-oblivious storage techniques can be broadly classified as:

1. Id-based: each element is associated with a unique id and the tree structure of the XML document is preserved by maintaining a foreign key to the parent.

2. Interval-based: each element is associated with a region representing the subtree under it.

3. Path-based: each element is associated with a path id representing the root-to-leaf path in addition to an interval-based or id-based representation.

We organize the rest of the discussion by considering different classes of queries.

Reconstructing an XML sub-tree

This problem is largely solved. In the schema-oblivious scenario, the sub-tree corresponding to an XML element could potentially span all tables in the database. Hence, while solutions that store all the XML data in only one table need to process just that table, other solutions will need to access all tables in the database.

For id-based solutions, a recursive SQL query can be used to reconstruct a sub-tree. For interval-based solutions, a non-recursive query with interval predicates is sufficient.

Simple Path Expression Queries

We refer to the class of path expression queries without predicates as simple path expression queries. For interval-based solutions, evaluating simple path expressions entails performing a range join for each step of the path expression. For example the query book/author/name translates into a three-way join. For id-based solutions, each parent-child(/) step translates into an equijoin, whereas recursion in the path expression (through //) requires a recursive SQL query. For path-based solutions, the path id can be used to avoid performing one join per step of the path expression.

Path Expression Queries With Predicates

Predicates can be existential path expression predicates, or positional predicates. The latter is dealt with in [TVB+02, YASU01]. We focus on the former for the rest of the section.

For id-based and interval-based solutions, a straightforward method for query translation is to perform one join per step in the path expression [DFS99, FK99, ZND+01]. With path ids, however, it is conceivable that certain joins can be skipped, just as they

can be skipped for some simple path expressions. A detailed algorithm for doing so is proposed in [YASU01]. That algorithm is correct for nonrecursive data sets — it turns out that it does not give the correct result when the input XML data has an ancestor and descendant element with the same tag name.

More Complex XQuery queries

The only published work that we are aware of that deals with more general XQuery queries is [DTCO03]. The main focus of the paper is on issues such as structural equality in FLWR where clauses, full compositionality of XML query expressions (in particular, the possibility of nesting FLWR expressions within functions), and the need for constructed XML documents representing intermediate query results. As mentioned earlier, special purpose relational operators are proposed for better performance. We note that without these operators, the performance of their translation is likely to be inferior even for simple path expressions. As an example, using their technique, the path expression /site/people is translated to an SQL query involving five temporary relations created using the *With* clause in SQL99, three of which involve correlated subqueries. To conclude, excepting [DTCO03], all prior work has been on translating path expression queries into SQL. Using the approach proposed by [DTCO03], we observe that with respect to function, a large fragment of XQuery can be handled using dynamic intervals in a schema-oblivious fashion. However, without modifications to the relational engine, its performance may not be acceptable.

2.3 Schema-Based XML Storage

In this section, we discuss approaches to storing XML in relational systems that make use of a schema for the XML data in order to choose a good relational schema. The main problems to be addressed in this subspace are

1. Relational schema selection — given an XML schema (or DTD), how should we choose a good relational schema and XML-to-relational mapping.

2. Query translation — having chosen an XML-to-relational mapping, how should we translate XML queries into SQL.

2.3.1 Relational Schema Selection

In [STZ+99], three techniques for using a DTD to choose a relational schema are proposed — basic inlining, shared inlining, and hybrid inlining. The main idea is to inline all elements that occur at most once per parent element in the parent relation itself. This is extended to handle recursive DTDs.

In [LC00], a constraint preserving algorithm for transforming an XML DTD to a relational schema is presented. The authors chose the hybrid inlining algorithm from [STZ+99] and showed how semantic constraints can be generated.

In [BFRS02], the problem of choosing a good relational schema is viewed as an optimization problem: given an XML schema, an XML query workload, and statistics over the XML data choose the relational schema that maximizes query performance. They give a greedy heuristic for this purpose.

In [HSJJ02, ML03], the theory of regular tree grammars is used to choose a relational schema for a given XML schema.

In [CDZ02], a storage mapping that takes into account the key and foreign key constraints present in an XML schema is presented.

There has been some work on using object-relational DBMS to store XML documents. In [KM00, RP02], parts of the XML document are stored using an XML ADT. The focus of these papers is to determine *which* parts of the DTD must be mapped to relations and which parts must be mapped to the XML ADT.

In Oracle XML DB [OXD], an annotated XML Schema is used to define how the XML data is mapped into relations. If the XML Schema is not annotated, XML DB uses a default algorithm to decide the relational schema based on the XML Schema. This algorithm handles recursive XML schemas.

A similar approach is made in Microsoft SQL Server 2000 SQLXML [SXM] and IBM DB2 XML Extender [DB2], but they only handle non-recursive XML schemas.

2.3.2 Query Translation

In [STZ+99], the general approach to translating XML-QL queries into SQL is illustrated through examples without any algorithmic details.

As discussed in [SSK+01], it is possible to use techniques from the XML publishing domain in the XML storage domain. To see this, notice that once XML data is shredded into relations, we can view the resulting data as if it were pre-existing relational data. Now by defining a reconstruction view that mirrors the XML-to-relational mapping used to shred the data, the query translation algorithms in the XML publishing domain are directly applicable. Indeed, this is the approach adopted in [BFRS02]. While this approach has the advantage that solutions for XML publishing can be directly applied to the schema-based XML storage scenario, it has one important drawback. In the XML

storage scenario, the data in the RDBMS originates from an XML document and there is some semantic information associated with this (like the tree structure of the data and the presence of a unique parent for each element). This semantic information can be used by the XML-to-SQL translation algorithm to generate efficient SQL queries. By using solutions from the XML publishing scenario, we are potentially making the use of this semantic information harder. We discuss this in more detail with an example in Section 2.3.3.

Note that even the schema-oblivious subspace can be dealt with in an analogous manner as mentioned in [SSK+01]. However, in this case, the reconstruction view is fairly complex — for example, the reconstruction view for the Edge approach is an XQuery query involving recursive functions [SSK+01]. Since there was no published query translation algorithm for recursive XML view schemas (Section 2.1.4) prior to the work we present in Chapter 3, this approach for the schema-oblivious scenario needs to be explored further.

In [TVB+02], as we mentioned in Section 2.2.2, the focus is on supporting order-based queries. The authors give an algorithm for the schema-oblivious scenario, and briefly mention how the ideas can be applied with any existing schema-based approach.

In Oracle XML DB [OXD], Microsoft SQL Server 2000 SQLXML [SXM] and IBM DB2 XML Extender [DB2], the XML Publishing and Schema-Based XML Storage scenarios are handled in an identical manner. So, the description of their approaches for the XML Publishing scenario presented in Section 2.1.3 holds for the Schema-Based XML Storage scenario. To summarize, XML DB supports branching path expression queries with the child and attribute axes, while SQLXML supports the parent and self axes as well. XML Extender does not support any XML query language. Instead, it provides user-defined functions to manipulate XML data.

In [KCN03], the problem of finding optimal relational decompositions for XML workloads is considered in a formal perspective. Using three XML-to-SQL query translation algorithms for path expression queries over a particular family of XML schemas, the interaction between the choice of a good relational decomposition and a good query translation algorithm is studied. The authors showed that the query translation algorithm and the cost model used play a vital role not just in the choice of a good decomposition, but also in the complexity of finding the optimal choice.

2.3.3 Discussion and Open Problems

Prior to the work presented in this thesis, there was no published query translation algorithm for the schema-based XML storage scenario. One alternative is to reduce this problem to XML publishing (using reconstruction views). Hence, from a functionality perspective, whatever is open in the XML publishing case is open here also. In particular, the entire problem was open when the input XML schema is recursive. Even for non-recursive XML schemas, a lot of interesting questions arise when the XML schema is not a tree. For example, if there is recursion in an XPath query through //, the straightforward approach of enumerating all satisfying paths using the schema and handling them one at a time is no longer an efficient approach. If we wish to reduce the problem to XML publishing, the only way to use an existing solution is to unfold the DAG schema into an equivalent tree schema.

In this thesis, we address some of the above issues. In Chapter 3, we present a generic algorithm to translate path expression queries into SQL that works for a large class of XML-to-Relational mappings. This algorithm handles recursion in the XML schema and XML query. It also addresses some of the issues that arise for recursive queries over

non-recursive schema.

From a performance perspective, just like in the XML Publishing scenario, there has been no focus on the quality of the final SQL queries produced. We now examine the translation problem from a performance perspective and show how there is a lot of scope for improving the quality of the SQL queries.

Goals of XML-to-SQL Query Translation

When an XML document is shredded into relations, there is inherent semantic information associated with the relation instances given that the source is XML. For example, consider the XML schema shown in Figure 5. One candidate relational decomposition is also shown in the figure. The mapping is illustrated through annotations on the XML schema. Each node is annotated with the corresponding relation name. Leaf nodes are annotated with the corresponding relational column as well. Parent-child relationships are represented using id and parentid columns. The figure element has two potential parents in the schema. In order to distinguish between them, a parentcode field is present in the Figure relation. In this case, notice that there is inherent semantics associated with the columns parentid and parentcode given that they represent the manner in which the tree structure of the XML document is preserved.

Given this semantics, when an XML query is posed, there are several equivalent SQL queries, which are not necessarily equivalent without the extra semantics that come from knowing that the relations came from shredding XML. Consider the following query: find captions for all figures in top level sections. This can be posed as an XPath query $XQ =$ /book/section/figure/caption. There are two equivalent ways in which we could translate XQ into SQL. They are shown below.

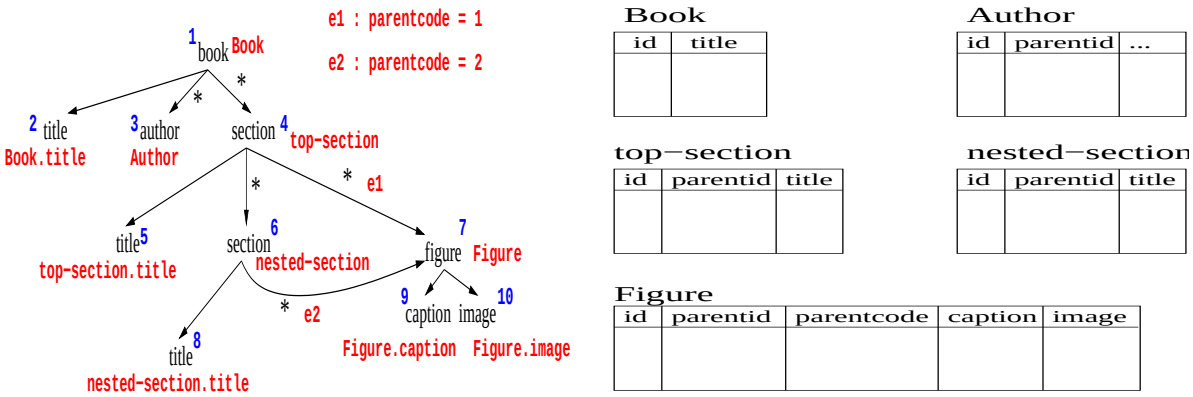

Figure 5: Sample XML-to-Relational mapping schema

```
SQ1:                          SQ2:
select caption                select caption
from figure                   from   figure f, top-section ts, book b
where parentcode=1            where  f.parentcode=1 and f.parentid=ts.id
                                     and ts.parentid=b.id
```

While SQ1 merely performs a scan on the figure table, SQ2 roughly performs a join for each step of the path expression. SQ2 is what we would obtain by adapting techniques from XML publishing. Queries SQ1 and SQ2 are equivalent only because of the semantics associated with the parentcode and parentid columns and would not be equivalent otherwise.

Now, since the XML-to-SQL translation algorithm is aware of the semantics of the XML-relational mapping, it is better placed than the relational optimizer to find the best SQL translation. Hence, from a performance perspective, the problem of effectively exploiting the XML schema and the XML-relational mapping during query translation is an interesting direction to pursue.

In Chapter 4, we present our approach for translating path expression queries into efficient SQL queries by making intelligent use of the additional semantic information.

Enhancing Schema-Based solutions with Intervals/Path-ids

All the schema-based solutions proposed in published literature have been id-based. In the schema-oblivious scenario, it has been shown that using intervals and path-ids can be helpful in XML-to-SQL query translation. The problem of augmenting the schema-based solutions with some sort of intervals and/or path-ids is an interesting open problem. Note that while any id-based storage scheme can be easily augmented by adding either a path-id column or an interval for each element, developing query translation algorithms that use *both* the schema information and the interval/path information is non-trivial.

2.4 Summary

To conclude, we refer again to the summary in Table 1. From that table, we see that the community has made varying degrees of progress for different subproblems in the XML to SQL query translation domain. We next summarize this progress, in terms of functionality.

- In the XML-Publishing scenario, techniques have been proposed for handling complex query languages like XQuery and XSLT over tree XML view schema. However, very little progress has been made on handling recursive XML view schema. Even for tree XML view schema, the subset of XQuery handled by current solutions is not clear.

- In the schema-oblivious XML storage scenario, excepting [DTCO03], the focus has been on path expression queries.

- In the schema-based XML storage scenario, prior to the work presented in this thesis, there was no published query translation algorithm. The only approach

known to us is through a reduction to the XML publishing scenario.

In this thesis, we address some of the open issues for the schema-based XML storage and XML publishing scenarios. In particular, we present an algorithm for handling recursive XML schemas and also present techniques for using semantic information to generate more efficient SQL queries. For the latter, we need to develop different techniques for the two scenarios, as semantic information is available in different ways in the two cases.

Chapter 3

Schema-based XML storage: Recursive Schemas and Queries

In this chapter, we consider the XML-to-SQL query translation problem for schema-based XML storage. As we saw in the previous chapter, prior to the work presented in this chapter, there was no published XML-to-SQL query translation algorithm that handled recursive XML schemas. Even for non-recursive schema, recursive XML queries were handled in a simplistic fashion.

We consider the translation of path expression queries into SQL in the presence of recursion in the schema and queries. We present an algorithm that performs this translation over a general class of XML-to-Relational mappings, which includes all techniques proposed in literature. Some of the salient features of this algorithm are: (i) It translates a path expression query into a single SQL query, irrespective of how complex the XML schema is, (ii) It uses the "with" clause in SQL99 to handle recursive queries even over non-recursive schemas, (iii) It reconstructs recursive XML subtrees with a single SQL query and (iv) It shows that the support for linear recursion in SQL99 is sufficient for handling path expression queries over arbitrarily complex recursive XML schema.

We first describe the class of XML-to-Relational mappings in Section 3.1. We then present the algorithm to translate path expression queries into SQL in Section 3.2 and discuss related work in Section 3.3.

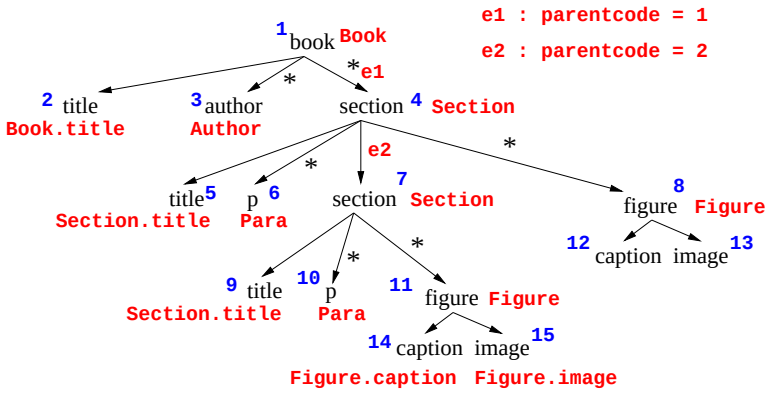

Figure 6: Sample XML-to-Relational mapping schema

3.1 Formal Model

In order to express our translation techniques, we need a representation for XML to Relational mappings. Any reasonable representation would serve our purpose; for concreteness, in this section we present a formal way to represent XML to Relational mappings that covers all the lossless mapping techniques proposed in existing literature.

3.1.1 XML Schema Graph

An XML schema can be viewed as a directed graph $\mathcal{SG} = (V, E)$, where V is the set of vertices and E is the set of edges. The vertices correspond to elements and attributes and the edges represent containment (parent-child) relationships. The vertices are labeled with the name of the element or attribute. The edges have an additional multiplicity label that can take a value from $\{?, *, +, \epsilon\}$. A sample non-recursive schema graph is given in Figure 6. With each schema node, we associate an integer to identify the node. If the schema graph is a tree, then we call it a *Tree* schema graph. If it is acyclic, we call it a *DAG* schema graph (directed acyclic graph). Otherwise, it is a *recursive* schema graph.

3.1.2 XML to Relational Mappings

We represent the mapping between XML elements and relational columns through annotations on the schema graph. For example, one way of mapping the XML schema in Figure 6 into relations results in the following relational schema.

- **Book** (*id*, title, ...)
- **Author** (*id*, parentid, ...)
- **Section** (*id*, parentid, parentcode, title, ...)
- **Para** (*id*, parentid, ...)
- **Figure** (*id*, parentid, caption, image, ...)

The annotations on the schema graph in Figure 6 correspond to this decomposition. Each non-leaf (internal) node in the schema is associated with a relation name (shown next to the node). Each leaf node is associated with a column name as well. The relational schema into which we shred the XML data is the set of relations that occur in the node annotations. Each relation has an id field, which is the primary key. In addition, parentid and parentcode fields are included as required to preserve document structure.

A node annotation for a leaf node n, *Annot*(n), is of the form $R.C$, where R denotes a relation and C denotes a column in R. A node annotation for a non-leaf node n, *Annot*(n), is of the form R indicating a relation name R. If a node n in the schema has multiple in-coming edges, then each of these edges is annotated with a condition of the form *parentcode* = *val*, indicating a code for the parent of an element matching n in the document. For a relational column $R.C$, we define *LeafNodes(R.C)* to be the set of leaf schema nodes annotated with $R.C$.

While the above description defines the syntax of an XML-to-Relational mapping, we next describe the associated semantics. A shredding algorithm uses the XML-to-Relational mapping to convert XML data into relational data. We say that the shredding algorithm *respects* a mapping if it satisfies the following properties:

- The shredding algorithm actually shreds the XML data into relations based on the annotations of the mapping.
- All the XML data is completely shredded into relations and no part of the XML data is stored multiple times.
- No data, other than what which is present in the XML document, is inserted into the relations mentioned in the mapping.
- Enough information is maintained in the relational data to enable reconstruction of the original XML data.

Every decomposition scheme we have encountered in the literature satisfies the above properties.

We say that the *lossless from XML* constraint is satisfied if all the relational data was loaded by a shredding algorithm that respects the XML-to-Relational mapping. This implies that the relational data set resulted from the lossless shredding of an XML document that conformed to a given XML schema.

For example, consider the following shredding algorithm A that satisfies the above properties. Given an XML document D_1 conforming to the schema in Figure 6, Algorithm A creates (and inserts into the RDBMS) relational tuples in the following fashion. The algorithm processes the elements in the order in which they appear in the document. For the root `book` element, a tuple is inserted into the `Book` relation. The value of the `title` column of this tuple is set to the value of the *title* subelement. Then for each `author` child

element, a tuple is inserted into the `Author` relation with the `parentid` column having the value of the `id` field of the `book` tuple. Similarly, for each `section` child element, a tuple is inserted into the `Section` relation with the `parentid` column having the value of the `id` field of the `book` tuple. Also, the `parentcode` column's value for the tuples corresponding to the `section` elements is set to 1 to capture the edge annotation `e1`. The value of the `title` column is set to the value of the corresponding `title` elements (corresponding to schema node 5). For each child element of the `section` elements, the same process is continued recursively and tuples are inserted into the corresponding relations (`Para, Section` and `Figure`).

The above definition of the "lossless from XML" constraint is complete if we formally define the notion of when a shredding algorithm respects a mapping. We introduce some terminology for this purpose.

With every path $p = <n_1, \ldots, n_k>$, we associate an SQL query, $SQL(p)$ as given in Figure 7. Intuitively, the SQL query retrieves from the relational shredding the information that appeared in portions of the original document that match the path p. With a leaf (schema) node l, we associate a root-to-leaf SQL query, $RtoL(l)$ as follows. Let the root-to-leaf paths to l be $p_1, \ldots, p_m$. Then, $RtoL(l) = \cup_{i=1}^{m} SQL(p_i)$. The union operation here preserves duplicates. If the mapping schema is recursive, the number of root-to-leaf paths will be infinite for certain leaf nodes and the $RtoL$ query for such nodes is the union of infinitely many queries.

For example, for the schema in Figure 6, $RtoL(9)$ is given below.

```
select S2.title
from   Book B, Section S1, Section S2
where  B.id = S1.parentid and S1.parentcode = 1
   and S1.id = S2.parentid and S2.parentcode = 2
```

```
procedure SQL(path p)
begin
 add Annot(n_1) to the From clause
 for (i from 2 to k) do
   Let e be the edge from n_{i-1} to n_i
   if (Annot(n_i) is different from Annot(n_{i-1})) then
     add Annot(n_i) to the From clause
     add Annot(n_{i-1}).id = Annot(n_i).parentid
         to the Where clause
   else if (Annot(e) is of the form parentcode = val) then
     add Annot(n_i) to the From clause
   if (Annot(e) is of the form C = val) then
      /* C may be parentcode */
     Let the last relation added to the From clause be R
     add R.C = val to the Where clause
 Add Annot(n_k) = R.C to the Select clause
 /* if there are multiple instances of the relation R,
   use the last instance */
end
```

Figure 7: SQL Query associated with a path p in the XML schema

Again, intuitively, $RtoL(l)$ retrieves from the relations the information that would be found in the original XML document by starting at the route and traversing all paths that match l.

A shredding algorithm respects an XML-to-Relational mapping,T,if for every collection of XML documents conforming to the given XML schema, the algorithm loads data into the relational database such that the following properties hold.

P1: For each root-to-leaf path p, $SQL(p)$ returns the values of all elements that satisfy p.

P2: For every relational column $R.C$ with $LeafNodes(R.C) \neq \phi$, let Q be the SQL query: "select $R.C$ from R". Then, $Q = \bigcup_{l \in LeafNodes(R.C)} RtoL(l)$

P3: For each path $p \in$ T ending in a leaf node, let P denote the set of root-to-leaf paths $p' \in$ T such that the relation names annotating the nodes in p match the

annotations for some suffix of p'. Then, $SQL(p) \subseteq \bigcup_{p' \in P} SQL(p')$.

All the above comparisons are under multiset semantics. Every shredding algorithm we have encountered in the literature, including the shredder we presented above, satisfies the above properties.

Notice how the shredding algorithm A needs to be validated just once to make sure that it shreds data respecting any given XML-to-Relational mapping. Once this is done, as long as all the data (in the relations appearing in the mapping) is loaded by the algorithm A , the "lossless from XML" constraint is guaranteed to hold.

In general, we allow two additional features in the mapping: selection conditions as edge annotations and presence of dummy nodes. Any edge e from n_1 to n_2 may have an optional annotation of the form $C = val$, where C is a column in the relation $Annot(n_1)$. In XML documents, certain elements may be introduced just to group elements that appear beneath them. We refer to such schema nodes as *dummy* nodes. For example, we could have a dummy Sections node in-between nodes 1 and 4 to group together all the sections in a book. An algorithm that shreds this document into relations need not take any action on finding a dummy node. We can detect that a shredding algorithm has considered a node n to be a dummy node by the fact that (1) n is a non-leaf node, (2) n is annotated with the same relation as its parent, (3) each in-coming edge is labeled ϵ and, (4) each in-coming edge has a null annotation. For ease of exposition, we assume that any non-leaf node that is not a dummy node has an elemid attribute that uniquely identifies an element within an XML document.

3.1.3 Path Expression Queries

A simple path expression (SPE) can be denoted as "$s_1\ l_1\ s_2\ l_2 \ldots\ s_k\ l_k$," where each of the l_i is a tag name and each of the s_i is either / (denoting a parent-child traversal) or // (denoting an ancestor-descendant traversal). Each $s_i\ l_i$ pair is a navigation step of the path expression and k is the number of steps in the query.

A generalized simple path expression (GSPE) can be denoted as "$p_1\ p_2 \ldots\ p_k$" where each p_i is of the form $p_i^1|p_i^2 \ldots p_i^{k_i}$ $(k_i \geq 1)$. Here, each p_i^j is a simple path expression. Each p_i thus denotes a disjunction of simple path expressions. Also, the special tag name '*' matches any tag name in a GSPE query.

The result of a generalized path expression is the *set* of all nodes that match the path expression query. This is similar to XPath semantics, where the result is an ordered list of matching nodes. We assume an unordered data model in this paper and discuss how our algorithm can be modified to work in an ordered model in Section 3.2.4. There are two possible ways to return the set of matching nodes:

- **Select** mode: For leaf nodes, this corresponds to returning the values of the elements. For non-leaf nodes, we return the value of the corresponding elemid attributes.
- **Reconstruct** mode: For leaf nodes, this corresponds to returning the values of the elements. For non-leaf nodes, we reconstruct the subtree rooted at the element.

3.2 Query Translation Over Recursive XML schemas

In this section, we present an XML-to-SQL query translation algorithm over recursive mapping schemas for the class of generalized simple path expression (GSPE) queries

procedure XML_to_SQL_translation(Q, S)
begin
1. Perform PathId using the mapping S and query Q.
   Let S_{CP} be the resultant cross-product schema.
2. Construct a SQL query corresponding to S_{CP}.

Figure 8: Query Translation Algorithm handling recursion in XML schema and query

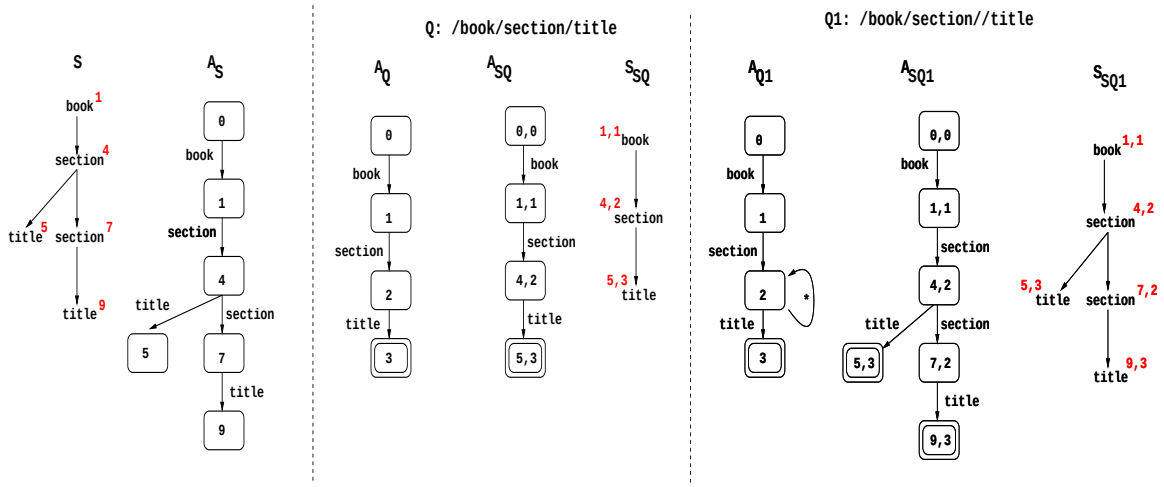

Figure 9: Example to illustrate PathId

defined in Section 3.1.3. We will assume the "Select" mode in this section and present our solution for the "Reconstruct" mode in Section 3.2.4.

Evaluating a path expression query over an XML-to-Relational mapping can be viewed as a two stage process: (i) use the XML query to identify the paths in the XML schema graph that satisfy the query, and (ii) use the annotations from the XML-to-Relational mapping to construct an equivalent relational query. We refer to these stages as the PathId and SQLGen stages respectively. We explain the two stages in the next two subsections.

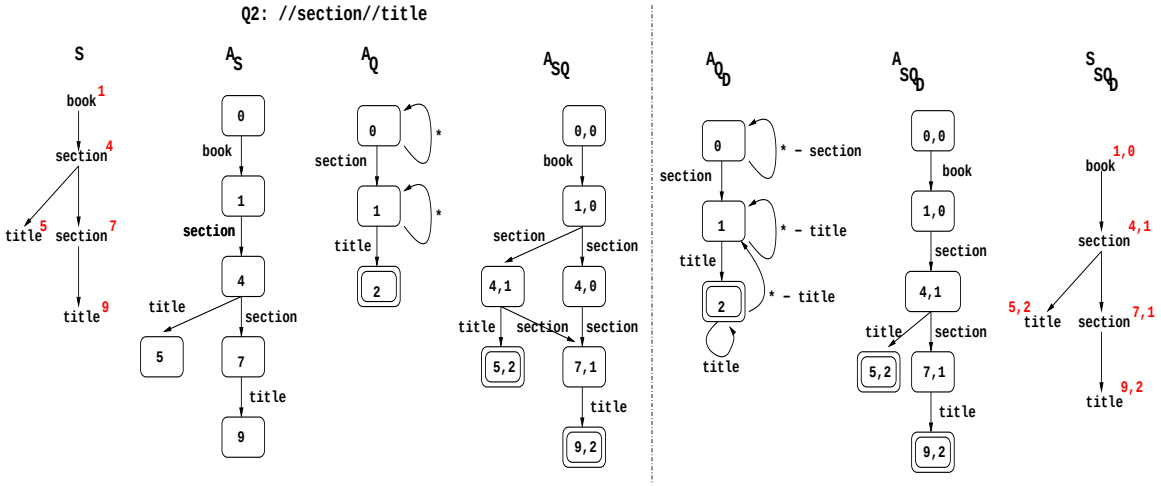

Figure 10: Example to illustrate duplicate counting in PathId

3.2.1 PathId stage

In the PathId stage, we execute the GSPE query $Q = p_1 \dots p_k$ on a schema graph and identify the satisfying paths in the schema graph. Since the mapping schema may be recursive, the number of paths may be infinite, so we cannot enumerate all the possible matching paths. Even when the mapping schema is non-recursive, for DAG schema graphs, it is possible for the number of matching paths to be exponential in the size of the mapping schema and the query. So, we should not attempt to enumerate all complete paths. Instead, just like the DAG schema graph represents shared information across multiple paths in a compact fashion, we represent the matching paths as a graph. This will allow us to handle recursive and non-recursive mapping schemas in a unified fashion. As an added benefit, we shall see later how preserving the relationship across multiple paths that existed in the original mapping schema will help us in the SQLGen stage.

Consider the evaluation of a query Q over a mapping schema S. We treat the mapping schema as an automaton A_S and the query as an automaton A_Q. We construct the cross-product automaton A_{SQ} from A_S and A_Q. We eliminate all the dead-states in A_{SQ} and

the resulting automaton has all the matching paths in it. This approach is similar to the one proposed in [FS98] for evaluating regular path queries over graph schemas. We illustrate the main idea with an example and explain the parts where our algorithm differs from the one in [FS98]. The reader is referred to [FS98] for more details.

Consider the schema S given in Figure 9, which is a part of the schema in Figure 6. The corresponding automaton A_S is shown next to it. Similarly, the query $Q =$ /book/section/title is translated into the automaton A_Q, where state 3 corresponding to the title element in Q is the accepting state. We construct the cross-product automaton A_{SQ} and remove the dead states. The resulting automaton A_{SQ} is shown in the figure. A state with number (i, j) in A_{SQ} represents a combination of state i in A_S with state j in A_Q. Since state 3 in A_Q is an accepting state, all states with state number $(i, 3)$ are accepting states in A_{SQ} (in this case just $(5, 3)$). Notice how A_{SQ} has simulated the query over the mapping schema and identified the single matching path. The state numbers in A_{SQ} illustrate exactly how each path matched the query. In general, A_S and A_Q are non-deterministic, and as a result A_{SQ} is also a non-deterministic automaton. This cross-product automaton can then be viewed as a mapping schema S_{SQ}. The node (edge) annotations for S_{SQ} are the same as the underlying annotations in S.

The PathId stage for the query $Q1 =$ /book/section//title is also shown in the figure. Notice how the // operation in the query translates into a self-loop on node 2 in A_{Q1}. Also, there are two matching paths in the schema for this query. So, there are two root-to-leaf paths in the cross-product automaton A_{SQ1}.

For purposes of exposition, we assume that all accepting states in S_{SQ} correspond to a leaf node in the original schema. If an accepting state $s \in S_{SQ}$ corresponds to a non-leaf node $n \in S$, we add the state corresponding to the elemid child of n as a final state in S_{SQ} (instead of s). Informally, this corresponds to returning the elemid's of non-leaf nodes as

the result of the query. This corresponds to the "Select" mode described in Section 3.1.3. We will present our solution for the "Reconstruct" mode in Section 3.2.4.

Handling XPath semantics

According to XPath semantics, the result of a path expression query is a duplicate-eliminated sequence of nodes. So, even if an element has multiple derivations with respect to the query, it should appear in the query result only once. For example, consider the evaluation of query $Q2$ = //section//title. The cross-product automaton A_{SQ} for this query is given in Figure 10. Notice how there are two matching paths in A_{SQ} for the title node under the second-level section (node 9). This is due to the fact that either of the section nodes in S (4 or 7), can match the //section part of Q_1. For both the cases, the //title part of the query is matched by the title node (node 9) in S. As a result, the /section/section/title path in the schema is replicated twice in A_{SQ}. So, if we construct a SQL query based on this cross-product automaton, we may get duplicate results. But, according to XPath semantics, we should return each satisfying element exactly once. In this section, we explain our approach to handling this issue.

We first examine what the primary reason for the presence of duplicate paths in A_{SQ} is and how we can avoid it. Going back to the above example, we see that the two paths for book/section/section/title have the following property: the first component of the nodes occurring in the two paths are identical, while the second component differs in (at least) one place. In other words, a single path in the schema gets duplicated, once with each of the two different derivations for the query. So, if the query automaton is a DFA, then the cross-product automaton will not have any schema path duplicated.

For an SPE query with k steps, we have an algorithm to construct an equivalent DFA with $k + 1$ states. We explain this algorithm using the query Q_2. The resulting

procedure PathId(Q,S)
begin
1. Let A_S be the automaton corresponding to S
2. If (Q is an SPE query) then
3. Let A_{Q_D} be the DFA corresponding to Q
4. return the cross-product automaton A_{SQ_D}
5. Else // Q is a GSPE query
6. Let A_Q be the NFA corresponding to Q
7. Let A_Q^2 be the automaton that accepts all strings with two or more accepting paths in A_Q
8. Compute the cross-product automaton A_{SQ^2}
9. If (A_{SQ^2} is empty) then
10. return the cross-product automaton A_{SQ}
11. Else
12. Convert A_Q into a DFA A_{Q_D}
13. If (A_{Q_D} does not have an exponential increase in size)
14. return the cross-product automaton A_{SQ_D}
15. Else
16. return the cross-product automaton A_{SQ}
17. // a distinct clause needs to be added in this case
18. // to the final SQL query
end

Figure 11: PathId Algorithm

deterministic automaton A_{Q_D} is shown in Figure 10. We partition the query into blocks such that each block has a leading // and there is no occurrence of // in that block. In this case, there are two blocks, one each for //section and //title. Then, we process these blocks from left to right. For the first block //section, we create a start state (state 0) and add a transition to state 1 on the label section. For any other label, since there is a leading //, we add a transition into state 0 itself (the start state of the current block). In general, when there are multiple steps in a block, there may be a partial match with the current string and we may have to transition not to the start state but to some intermediate state. This can be found by identifying the longest suffix that matches the current set of labels and is similar to the Knuth-Morris-Pratt string matching algorithm [CLR90]. State 1 is the final state for this block and will act as the

start state for the next block. We repeat the process for //title and add state 2 and a transition from 1 to 2 on title. We also add transitions on other labels for state 1. Since this is the last block, we also compute the transitions from state 2 and set state 2 as the final state for A_{Q_D}.

LEMMA **1** *For an SPE query Q having k steps, the equivalent DFA having $k+1$ states can be computed in $O(k)$ time.*

Proof: We show the correctness of the above algorithm by induction on the number of descendant axes k in the query. Without loss of generality, we assume that the query starts with a // (otherwise, we remove the prefix of the query before the first occurrence of // and add the corresponding states to the beginning of the resulting DFA). For the case $k = 1$, the correctness follows from the correctness of the Knuth-Morris-Pratt algorithm.

Assume that the result holds for a query with k descendant axes. We show that the algorithm constructs the correct DFA for a query with $k+1$ descendant axes. Let the query be of the form Q_1Q_2, where Q_1 is the leading block. Let the equivalent DFA constructed by the above algorithm be A_Q. Note that there is a path p from the start state to the accepting state in A_Q that matches the query Q' obtained by replacing all occurrences of // with /. Divide the states in path p into two subsets, p_1 and p_2, corresponding to Q_1 and Q_2 respectively. By construction, there are no transitions from any state in p_2 to any state in p_1. So, from our induction hypothesis, the part of DFA A_Q corresponding to states in p_2 is correct for query Q_2. Moreover, the states in p_1 make sure that any string accepted by DFA A_Q has a prefix matching Q_1 before it entered any state in p_2. The correctness of the DFA A_Q follows from this.

The running time of the algorithm follows from the fact that the processing time for

each block of size k_i is $O(k_i)$. □

On the other hand, for GSPE queries, there are scenarios when the smallest equivalent DFA is exponential in the size of the query. In this case, we use the following approach. Let A_Q denote the NFA corresponding to the query Q. We first compute an NFA A_Q^2 that accepts all input strings that have two or more accepting paths in A_Q. Then, we compute the cross-product automaton A_{SQ^2} between A_S and A_Q^2. If this automaton is empty, then it means that the cross-product automaton A_{SQ} obtained from the original query and schema automata (A_Q and A_S respectively) will not have any duplicate schema paths and we use A_{SQ} as the output of the PathId stage. On the other hand, if A_{SQ^2} is not empty, then we have two options: (1) convert A_Q into a DFA A_{Q_D} and compute cross-product between A_S and A_{Q_D} or (2) apply a *distinct* clause for the query obtained from A_{SQ}. We choose one of these options based on whether there is a size explosion when we convert A_Q into a DFA[1].

The PathId algorithm along with the above modifications to handle the semantics of XPath is given in Figure 11.

Analysis of PathId stage

In this section, we present an analysis of the number of states in the resulting cross-product automaton and the running time of the above algorithm.

Let s and e be the number of nodes in the schema and k be the number of steps in the query. Then the number of states in A_S is $n_s = s + 1$ and the number of states in A_Q is $n_q = k + 1$. For an SPE query, the number of states in $A_{Q_D} = k + 1$.

LEMMA **2** *The number of states in the cross-product automaton A_{SQ} is no greater than*

[1]This can be achieved by placing a bound on the number of states explored in the NFA-to-DFA conversion

$n_s * n_q$.

Proof: Each state in A_{SQ} corresponds to a pair of states from the schema automaton and the query automaton. The upper bound follows from this. □

LEMMA **3** *If S is a tree schema and Q is an SPE query, then the number of states in A_{SQ_D} is no greater than n_s.*

Proof: Since S is a tree schema, there is a unique path p from the start state of A_S to every state $s \in A_S$. Also, the query automaton is deterministic implying that there is at most one state in A_Q matching the path p. So, every state in A_S may contribute at most one state to A_{SQ_D}. The upper bound on the number of states in A_{SQ_D} follows from this. □

For an SPE query Q, for every label x, let $\mathsf{ChildOccur}(x, Q)$ denote the number of occurrences of the pattern $/x$ in Q. For example, for the query Q = /section//section//title, ChildOccur(section,Q) = 1 and ChildOccur(title,Q) = 0. Let $\mathsf{MaxChildOccur}(Q)$ denote the maximum across all values for ChildOccur(x, Q) over all labels. In this case, MaxChildOccur$(Q) = 1$.

Let $\mathsf{DescendantSteps}(Q)$ denote the number of // steps in Q. For the above example, DescendantSteps$(Q) = 2$. Notice how DescendantSteps(Q) + MaxChildOccur$(Q) \leq n_q$

LEMMA **4** *For an SPE query Q, the number of states in the cross-product automaton A_{SQ_D} is no greater than n_s* (DescendantSteps(Q) + MaxChildOccur(Q)).*

Proof: Consider a state $s \in A_S$. Since the automaton A_S is constructed from an XML schema, all in-coming edges have the same label l (the element name of the corresponding schema node). Suppose the schema state s creates a state in A_{SQ_D} of the form (s, q). For this state to have an incoming edge, the corresponding query state q must have an

incoming edge with transition l. A query state will have such an edge if it corresponds to a step in the query with element name l or if the corresponding axis is //. The value (DescendantSteps(Q) + MaxChildOccur(Q) is an upper bound on the number of such states. □

Let us now consider the running time of the various steps in the PathId stage.

PROPOSITION **1** *The running time of the PathId stage for an SPE query is $O(n_s^2 * n_q^2)$, while for a GSPE query, the running time is $O(n_s^2 * n_q^4)$.*

Proof: From Lemma 1, we see that the DFA corresponding to a query Q can be computed in time $O(n_q)$. Combining this with the facts that the automaton A_Q^2 can be computed in $O(n_q^4)$ and that the cross-product automaton of two state machines with n_1 and n_2 states respectively can be computed in $O(n_1^2 * n_2^2)$, the results follow. □

From a correctness viewpoint, we have following result that the mapping schema resulting from the PathId stage captures all the schema paths that satisfy the query.

PROPOSITION **2** *Given a mapping schema S and query Q, the output of the PathId stage, S_{SQ}, has exactly all the matching schema paths in it. When Q is an SPE query, each matching path in S appears only once in S_{SQ}.*

Proof: We need to prove that every path in S that matches the query appears in S_{SQ} and that there are no extra accepting paths in S_{SQ}.

We first prove the former claim. Suppose a path $p \in S$ matches the query. From the way the automaton A_S is constructed, this implies that there is a corresponding path $p' \in A_S$. Let $p' = \{n_1, n_2, \ldots, n_k\}$. Since the path p matches the query, we argue that there is a path $q' \in A_{SQ} = \{(n_1, q_1), (n_2, q_2), \ldots, (n_k, q_k)\}$. This follows from the way the cross-product automaton A_{SQ} is constructed. Since the path p satisfies the query,

it means that there is an accepting path $p'' \in A_Q$ such that the labels along this path are the same as the element names along p. The path $q' \in A_{SQ}$ represents the path obtained by combining the paths p' and p''. By construction, every path $q' \in A_{SQ}$ has a corresponding path $q \in S_{SQ}$. Hence, we have shown that every path in S that matches the query appears in S_{SQ}.

For the latter claim, let $q \in S_{SQ}$ be a root-to-leaf path. Then, there is a corresponding path $q' \in A_{SQ} = \{(n_1, q_1), (n_2, q_2), \ldots, (n_k, q_k)\}$ that is an accepting path. From the way the cross-product automaton is constructed, this implies that the path $p' \in A_S = \{n_1, n_2, \ldots, n_k\}$ matches the query Q. This in turn implies that there is a corresponding path in S that matches the query.

Finally, when Q is an SPE query, the query automaton is a DFA. Hence, every matching schema path p will appear only once in A_{SQ}. □

3.2.2 SQLGen stage

Once we have identified all matching paths in the schema S corresponding to query Q, we have a cross-product schema S_{SQ} with all the matching paths encoded in it. Informally, the union of all root-to-leaf paths in S_{SQ} corresponds to the query result. A simple algorithm to generate an SQL query corresponding to Q is to return $RQ = \bigcup RtoL(l)$ over all leaf nodes in S_{SQ}. While this is a good algorithm when S_{SQ} is a tree, it does not suffice when S_{SQ} is a DAG or is recursive. If S_{SQ} is a DAG, then the number of matching paths may be exponential. Moreover, by unfolding a DAG we may also be missing shared computation in the form of common subexpressions in the final SQL query. So, we need to somehow reflect the DAG structure of S_{SQ} in the SQL query. Similarly, if S_{SQ} is recursive, then RQ is the union of infinite queries. In this section, we

procedure SQLGen(S_{SQ})
begin
1. Identify strongly connected components (SCCs) in S_{SQ}
2. Let C be the set of SCCs
3. Merge adjacent components in C that are acyclic
   if one of them dominates the other
4. foreach ($c \in C$ in top-down topological order) do
5. if (c is not recursive) then
6. generate the query for c using SQLForDAG(c)
7. else
8. generate the query for c using SQLForRecursive(c)
   // a relational query $T(n)$ is associated with
   // each leaf node n now
  endFor
9. Let finalQ be $\cup_n$ is a leaf node "select * from $T(n)$"
10. If (duplicate elimination is required) then
11. Output the query "select distinct(*) from finalQ"
12. else output the query "select * from finalQ"
end

Figure 12: SQLGen Algorithm

show how using the support for linear recursion in SQL99 (*with* operator) along with the *outer union* approach, we can construct the equivalent (finite!) SQL query for a recursive cross-product schema.

In order to illustrate our algorithm for handling complex cross-product schema, we use the schema graph S in Figure 13. Notice how this schema has a DAG part and a recursive part. The edge annotations are omitted for clarity. We use the shorthand i to denote the schema node corresponding to element Ei and refer to the elemid node as node 11. We explain the algorithm by running through the evaluation of the query $Q = \ /E0//E10$ on the schema graph S in Figure 13. The PathId stage will result in a cross-product schema S_{SQ} identical to S. Notice how since $E10$ is a leaf node, we add the elemid attribute node to S_{SQ} and make that the accepting state.

The outline of the algorithm is given in Figure 12. We first identify the components

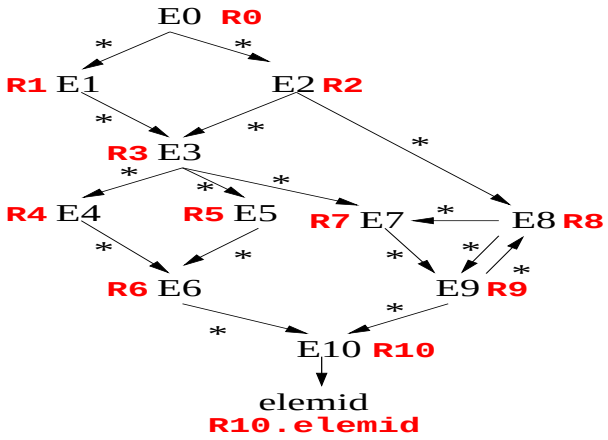

Figure 13: Sample recursive schema to explain the SQLGen algorithm

in S_{SQ} that are recursive. The rest of the nodes are grouped into a set of non-recursive components. We perform this computation by first identifying the strongly connected components in S_{SQ} (step 1) and then merging adjacent non-recursive components wherever possible (step 3). Recall that a component c_1 dominates component c_2 if every path from the root to a node in c_2 passes through some node in c_1. After the first 3 steps in Figure 12, there are three components in C. They are $c_1 = \{0, 1, 2, 3, 4, 5, 6\}$, $c_2 = \{7, 8, 9\}$ and $c_3 = \{10, 11\}$ (Here node 11 refers to the elemid node). We then process these components in top-down topological order, namely c_1 followed by c_2 followed by c_3. For each component, we generate the appropriate relational queries. In the process, we associate a temporary relation $T(n)$ with every schema node n that is either a leaf node or has a child node in a different component. Once we have processed all components, we generate the final relational query in steps 10-12 using the temporary relations defined earlier.

The algorithm for generating SQL queries corresponding to a non-recursive and a recursive component are given in Figures 14 and 15 respectively. We discuss these in the next two subsections.

procedure SQLFromDAG(c)
begin
1.Let N be the set of nodes in c with either
    a parent or a child in a different component
2.Add any node in c to N if it corresponds
    to a leaf node in S.
3.Add all nodes in c with $>$ one in-coming edge to N
4.Add all nodes in c with $>$ one out-going edge to N
5.With each node $n \in N$, associate a unique temporary
    relation $T(n)$
6.foreach ($n \in N$ in top-down topological order) do
7. //generate SQL fragment to populate $T(n)$
8. foreach (in-coming edge e into n) do
9.   Backtrack along e till either a node $m \in N$
      or a node $m \notin c$ is obtained.
10.   Let the unique m to n path be p
11.   Generate SQL(p) using $T(m)$ as the relation
      corresponding to m
12.   //Other node and edge annotations in S_{SQ} are
      //same as underlying ones in the mapping S
13.   Call this query SQL(e)
14. $T(n)$ is defined as the union of all the SQL(e)
15.endFor
end

Figure 14: SQLGen Algorithm for a DAG Component

Handling a non-recursive component

For non-recursive components, a straightforward approach is to translate each path in the DAG component into a SQL query and take the union of all these queries. However, the number of paths can be exponential in the size of the component. The question arises whether there is any way in which we can at least guarantee a query that is polynomial in the size of the DAG component. We show that this is impossible if we only consider relational queries involving the select, project, join and union operations (SPJU queries). We formalize this claim as follows. Let C_1 denote the class of relational queries whose relational algebra expression has the select, project, join and union operators. Let the size

of a query $SQ \in C_1$, RelInst(SQ), be the number of relation instances in the relational algebra expression. Then we have the following result.

PROPOSITION **3** *There is a family of mapping schemas SG such that, for each schema $S_i \in SG$, there is a simple path expression query p_i that has the following property. No relational query $SQ \in C_1$, whose size is polynomial in the size of S_i and p_i, is a correct translation for p_i.*

Proof: The proof is based on the fact that there are instances of acyclic Deterministic Finite Automata (DFA) whose minimum equivalent regular expression is superpolynomial in length [EZ76].

In [EZ76], a *half-complete graph* is defined as a graph with n nodes that has a distinctly labeled directed arc running from every node to every higher numbered node. Let N be the size (in terms of number of alphabetic symbols) of any equivalent regular expression. It is shown that $(n-2)^{(2/3)(log((1/3)log(n-2))-1)} \leq N \leq (2n+1)^{\lceil logn \rceil}$

Given a half-complete graph G, we can construct a mapping schema SG as follows. For each edge $e \in G$, add two vertices v_e^1 and v_e^2 to SG with the same label. Let the relation name annotating these nodes be e. For every pair of edges $e_1 = <v_1, v_2>$ and $e_2 = <v_2, v_3>$ in G, add edges $<v_{e_1}^1, v_{e_2}^1>$ and $<v_{e_1}^2, v_{e_2}^2>$ in SG. Add four more nodes n_0, n_1, n_2 and n_3 to SG with new labels (say root, subtree1, subtree2 and leaf respectively). Annotate these nodes with corresponding relation names. The leaf node n_3 is annotated with the column name `leaf.id`. Add edges from n_0 to n_1 and n_2. Let v_1 and v_2 be the lowest and highest numbered vertices in G. Add edges from n_1 to node v_e^1, where e is an outgoing edge from v_1. Add an annotation on these edges of the form "parentcode = 1". Add similar outgoing edges from n_2. Similarly, add edges from nodes v_e^i to $leaf$, where e is an incoming edge to n_2.

Consider the path expression query /root/subtree1//leaf. This query returns all root-to-leaf paths in the schema SG that appear in one of the two subtrees. Since each node has a different relation name, it follows that every equivalent SPJU SQL query corresponds to a regular expression solution for the paths in G. The proposition follows from the bounds on N. □

It turns out that we can use the *with* clause to solve this problem. Even though the *with* clause was primarily introduced for supporting recursive queries, it also provides us with a mechanism for creating temporary relations in a SQL query. So, whenever there is some computation that can be shared by multiple paths, we create a temporary relation corresponding to this shared computation, which can be used repeatedly in the rest of the query. Notice how creating temporary relations in the query allows us to reduce the size of the generated SQL query from (potentially) exponential in the size of the component to a guaranteed polynomial bound.

Component c_1 is non-recursive and an example of a DAG component. We use the algorithm for generating the relational query corresponding to a DAG component given in Figure 14. We associate temporary relations with any node that is either a leaf node (part of the final query result), has a parent or child in a different component, or represents shared computation (multiple incoming/outgoing edges). For component c_1, the set N is $N = \{2, 3, 6\}$. So, we generate SQL *with* clauses for three temporary relations corresponding to $T(2), T(3)$ and $T(6)$ in that order. The query corresponding to $T(3)$ is given below.

```
with T3 as (
  select R3.*
  from R0, R1, R3
  where R0.id = R1.parentid and R1.id=R3.parentid and R3.parentcode=1
union all
  select R3.*
```

```
    from T2, R3
    where T2.id=R3.parentid and R3.parentcode=2
)
```

Notice how the query is the union of two subqueries, one corresponding to each in-coming edge into node 3. Also note how we use $T2$ in the definition of $T3$, as node $2 \in N$ and has a temporary relation associated with it. In a similar fashion, the query for $T6$ will have $T3$ in it. This illustrates how shared computation can be efficiently reflected in the relational query. We would like to point out that the use of the *with* clause has two benefits. Firstly, it avoids the potential size blowup for complex DAG schema. Secondly, it represents the shared computation across different root-to-leaf paths explicitly. The relational optimizer can choose from the two options of either sharing computation across different fragments in the final execution plan or unfolding the with clause into the union of several conjunctive queries. In fact, we know of one commercial RDBMS whose optimizer does this exploration. On the other hand, if we did not use the *with* clause in the SQL query, then the relational optimizer has the additional task of finding common subexpressions, which is known to be a difficult task.

Handling a recursive component

Let us now look at how to generate the relational query for a recursive component. This algorithm is given in Figure 15. For each recursive component c, we associate a temporary relation T_R whose schema is the outer-union of the schemas of relations annotating some node in T_R and generate a recursive query for T_R as follows. A recursive query has two parts, an initialization part and a recursive part. The initialization part for the query defining T_R (steps 2-6) captures all incoming edges into c from a different component. For the component c_2, there are two such edges $(2, 8)$ and $(3, 7)$ and the initialization part will be the union of two conjunctive queries, one for each incoming edge. The recursion

procedure SQLFromRecursive(c)
begin
1. Let T_R be a temporary relation whose schema is the
   outer union of all relations in c
   //Construct the initialization query for T_R
2. foreach (in-coming edge e into c from node $n \notin c$) do
3. Let $n' \in c$ be the target of e
4. Let Q_e be the query:
   select R2.*, $id(n)$
   from $T(n)$ R1, $Annot(n')$ R2
   where $Annot(e)$ and R2.parentid = R1.id"
5. Null pad Q_e appropriately to reflect outer-union schema
6. Let Q_{init} be $\cup Q_e$ over all in-coming edges e
   //Construct the recursive part for T_R
7. foreach (edge e with both end-points in c) do
8. Let e be from n_1 to n_2
9. if (e corresponds to a join edge) then
10. Let Q_e be the query:
    select R2.*, id(n_2)
    from T_R R1, $Annot(n_2)$ R2
    where R1.schemanode = id(n_1) and
    R2.parentid = R1.id and $Annot(e)$
11. else
    //e corresponds to a selection or n_2 is a dummy node
12. Let Q_e be the query:
    select R1.*, id(n_2)
    from T_R R1
    where R1.schemanode = id(n_1) and $Annot(e)$
13. Null pad Q_e appropriately
14. Let Q_{rec} be $\cup Q_e$ where the union is taken over
    edges e with both end-points in c
15. T_R is a recursive query defined with Q_{init} as the
    initialization condition and Q_{rec} as the
    recursive component
16. With each node $n \in c$ we associate the query $T(n)$:
    select * from T_R where schemanode = id(n)
end

Figure 15: SQLGen Algorithm for a Recursive Component

in the component c is captured by the recursive part of the definition of T_R. Each edge in c is translated into a query as shown in steps 8-13 and the recursive part of the query defining T_R is the union across all edges within the component. For the component c_2, there are four edges and so the recursive query Q_{rec} is the union of four recursive queries. In this case, all four edges are join edges. For example, the edge $(8, 7)$ will translate to the following query:

```
select R7.*, id(7)
from TR, R7
where R7.parentid=TR.id and TR.schemanode=id(8) and R7.parentcode=8
```

Notice how the condition TR.schemanode ensures that the parent tuple corresponds to schema node 8 and the other conditions capture the annotations on the edge. By projecting the id of the child node, we ensure that the queries for outgoing edges from node 7 can be correctly constructed.

Returning to the example query, finally, component c_3 is non-recursive and we generate the equivalent relational query using the algorithm in Figure 14. The complete SQL query is given in Figure 16.

If S_{SQ} has two root-to-leaf paths matching the same path in S, then duplicate elimination is required and we add the distinct clause (recall discussion in previous section, step 16 in Figure 11). In such a scenario, for each leaf node $n \in S_{SQ}$, the id of n and the key column of R, where $Annot(n) = R.C$ also have to be projected along with $Annot(n)$ while creating the temporary relations $T(n)$.

We now show that the SQL query output by the XML_to_SQL algorithm is a correct translation of the input XML query Q.

THEOREM **1** *Given an XML-to-Relational mapping T, a path expression query Q and the guarantee that the "lossless from XML" constraint is satisfied, the XML_to_SQL algorithm outputs a correct equivalent SQL query.*

```
//query for component c1
    with T2 as (
      select R2.*
      from R0, R2
      where R0.id = R2.parentid
    ),
    with T3 as (
      select R3.*
      from R0, R1, R3
      where R0.id = R1.parentid and
      R1.id=R3.parentid and R3.parentCode=1
    union all
      select R3.*
      from T2, R3
      where T2.id=R3.parentid and R3.parentCode=2
    ),
    with T6 as (
      select R6.*
      from T3, R4, R6
      where T3.id = R4.parentid and
      R4.id=R6.parentid and R6.parentCode=4
    union all
      select R6.*
      from T3, R5, R6
      where T3.id = R5.parentid and
      R5.id=R6.parentid and R6.parentCode=5
    ),
// query for component c2
    with TC2 as (
    (
      select R7.*, id(7)
      from T3, R7
      where T3.id=R7.parentid and R7.parentCode=3
    union all
      select R8.*, id(8)
      from T2, R8
      where T2.id=R8.parentid and R8.parentCode=2
    )
    union all
    (
      select R7.*, id(7)
      from TC2, R7
      where R7.parentid=TC2.id and TC2.schemanode=id(8)
      and R7.parentid=8
    union all
      select R8.*, id(8)
      from TC2, R8
      where R8.parentid=TC2.id and TC2.schemanode=id(9)
      and R8.parentid=9
    union all
      select R9.*, id(9)
      from TC2, R9
      where R9.parentid=TC2.id and TC2.schemanode=id(7)
      and R9.parentid=7
    union all
      select R9.*, id(9)
      from TC2, R9
      where R9.parentid=TC2.id and TC2.schemanode=id(8)
      and R9.parentid=8
    )
    ),
    with T7 as (
      select *
      from TC2
      where schemanode=id(7)
    ),
    with T8 as (
      select *
      from TC2
      where schemanode=id(8)
    ),
    with T9 as (
      select *
      from TC2
      where schemanode=id(9)
    ),
// query for component c3
    with T10 as (
      select R10.*
      from T6, R10
      where T6.id = R10.parentid and R10.parentCode=6
    union all
      select R10.*
      from T9, R10
      where T9.id = R10.parentid and R10.parentCode=9
    ),
    with T11 as (
      select elemid
      from T10
    ),
// the final query
    select elemid
    from T11
```

Figure 16: SQL query output by the XML_to_SQL algorithm for $Q = /E0//E10$

Proof: From Proposition 2, we know that the result of the PathId stage, S_{SQ}, has all the schema paths that match the query. Since the "lossless from XML" constraint holds, it follows that $RQ = \bigcup_{\text{l}}$ is a leaf node in $_{S_{SQ}}$ $RtoL(l)$ is a correct SQL translation for Q. We need to show that the query RQ_1 output by the SQLGen algorithm is equivalent to RQ.

Let us first consider the case when S_{SQ} is acyclic. In this case, notice that the only difference between RQ and RQ_1 is that the latter uses *with* clauses to group together common computation as indicated by the mapping schema. In other words, if we unroll the with clauses in RQ_1 (replace all occurrences of the temporary relations Ti with their actual definitions), we obtain RQ. Hence, the two queries are equivalent.

Consider the case when the mapping S_{SQ} is recursive. The main difference between RQ and RQ_1 in this case is the use of the *with* clause in RQ_1 to represent the infinite paths in a more compact fashion. The only other difference between RQ_1 and RQ is the use of the outer-union schema for recursive components. The use of the *schemanode* field takes care of the fact that only root-to-leaf paths present in the mapping are captured by the corresponding SQL query. The equivalence between RQ and RQ_1 follows directly from the fact that recursive chain datalog programs are equivalent to the union of the infinite conjunctive queries obtained by unrolling them. □

We next analyse the running time of the SQLGen stage of the algorithm. For a recursive component C, let N_C denote the number of columns in the outer union schema for C. This is the sum of the number of columns over all relations annotating some node in C. For a mapping schema S, let $N_C^{max}(S)$ denote the maximum across all values for N_C over all recursive components in S.

LEMMA **5** *For a mapping schema T and query Q, let the output of the PathId stage,*

Clique size (n)	Time Taken (ms)
5	6
10	19
20	80

Table 2: Execution time of translation algorithm

A_{SQ} have V nodes and E edges. The equivalent SQL query can be obtained using the SQLGen algorithm in $O(E * N_C^{max}(A_{SQ}))$ time.

Proof: In the processing of a DAG component, each edge is processed a constant number of times. The same is true for a recursive component, but there is a potential overhead due to the outer union schema. The running time follows from this. □

3.2.3 A Note on Query Translation Time

The reader may wonder what constants are hiding behind the asymptotic bounds on the running time of the query translation algorithm and whether this is really practical. This section shows that on some synthetic queries, even those constructed to be worst case, the running time of the translation algorithm is reasonable.

We implemented the above algorithm for evaluating SPE queries over a generic XML-to-Relational mapping. Using the XMark benchmark schema [XMa] and SPE query fragments that appear in the associated query test suite, we evaluated the equivalent SQL queries using the above query translation algorithm. The XML-SQL query translation process took less than 6ms for each SPE query. The XML-to-Relational mapping schema has 101 nodes. We also observed that in all cases, the size of the cross-product schema was less than 100 nodes (the size of the schema).

In order to test the running time of the algorithm under extreme scenarios, when the

cross-product schema may have $n_s * n_q$ states, we used a more complex XML schema. This mapping schema was a *complete* graph of n nodes and all transitions were on a single label x. We then measured the running time of the query translation for the query //x//x//x//x//x, which has 5 steps. The cross-product automaton has approximately $4n$ states and $4n^2$ transitions. The running time for different values of n are given in Table 2.

Notice how while the running time shows a quadratic growth due to the quadratic increase in the number of transitions in the schema, it is still small for reasonable clique sizes. The size of every recursive component that we have seen in real-world DTDs has been less than 10. So we believe that the running time of our translation algorithm will be small in practice.

3.2.4 Extensions to the Algorithm

In this section, we describe how the above algorithm can be extended to reconstruct XML subtrees and handle order in XPath semantics.

Reconstructing XML subtrees

In [FMS01, SSB+00], algorithms were presented for reconstructing XML subtrees when the mapping schema is a tree. In this section, we describe how to handle the reconstruction of a recursive component and a DAG component.

Notice that the SQLGen algorithm for handling a recursive component in Figure 15 actually reconstructs the XML data corresponding to the entire recursive component. But, what is missing in order to reconstruct the XML subtree is structural information about the different elements. Recall that in [FMS01, SSB+00], this could be determined statically as for a tree XML schema, the number of distinct root-to-leaf paths is fixed.

On the other hand, for recursive components, we need to construct the root-to-leaf path dynamically. Notice that the schemanode column in relation T_R keeps track of the schema node corresponding to the tuple. We maintain an additional rtol column that keeps track of the path from the root of the subtree being constructed. This is similar to the approach proposed in [TVB+02] for constructing dewey numbers dynamically.

In order to handle a DAG component, we have two options. One option is to unroll the DAG into a tree and apply prior techniques. If this may lead to a size explosion, we could reconstruct a DAG directly by keeping track of the root-to-leaf path as mentioned above.

Handling order in XPath semantics

According to XPath semantics, the results of a path expression query have to be returned in document order. In order to support this, the schema-based shredding of XML into relations will need to maintain the relative position among sibling XML elements in some form. This was the primary focus of [TVB+02], where solutions were proposed to handle order in XML for an arbitrary query translation algorithm. Hence, in particular, their techniques can be integrated with our algorithm in a straightforward fashion.

3.3 Related Work

A detailed description of the existing published work on XML-to-SQL query translation was given in Chapter 2. In this section, we look at related work done in a context other than XML-to-SQL query translation.

There has been work on optimizing queries in a semi-structured framework [BDFS97, FS98, MW99] using graph schemas. These techniques are similar to the PathId stage

of query translation, and we adapted the cross-product automaton technique proposed in [FS98], for the PathId algorithm in Section 3.2.1.

The Hy$^+$ database visualization system uses GraphLog, a novel graphical language. In [Eig94], a GraphLog-to-SQL translation algorithm is presented that makes use of the support for inline views and recursion in SQL. This is similar to our use of the "with" clause in translating path expression queries into SQL (for DAG and recursive schema). The data model and query language used in the Hy$^+$ system are different from the XML counterparts. So, the solutions proposed in [Eig94] are not directly applicable in the context of XML-to-SQL query translation.

3.4 Summary

We presented a generic algorithm to translate path expression queries to SQL in the presence of recursion in the XML schema and queries. This algorithm is applicable over a wide class of techniques for schema-based shredding of XML into relations. We also showed how the *with* clause in SQL99 is useful in XML-to-SQL query translation over DAG XML schema and how the support for linear recursion in SQL99 is sufficient for translating path expression queries into a single SQL query over an arbitrary (recursive) XML-to-Relational mapping.

Chapter 4

Mapping-aware Query Translation

In the previous chapter, we presented an algorithm for translating path expression queries into SQL over (non)recursive XML-to-Relational mappings. While this algorithm works correctly and also has some optimizations to generate efficient SQL queries (like the use of the *with* clause for DAG schemas), it still has one major drawback. The final SQL query generation phase (SQLGen stage) is top-down. As a result, the XML hierarchy gets reflected in the final SQL query causing even simple XML queries to produce fairly complex SQL queries. This problem is aggravated when the XML query is recursive. We briefly discussed this problem in Chapter 2.3.3. In the example we used there, SQ2 is the SQL query that will be output by existing XML-to-SQL query translation algorithms, including the XML_to_SQL algorithm we presented in Chapter 3. On the other hand, SQ1 is a much simpler equivalent SQL query. By replacing a three-way join query with a scan-based query a significant performance improvement can be achieved.

In the schema-based approach to shredding XML into relations, we have a simple guarantee about the input relational data. We know that the relational data set resulted from the lossless shredding of one or more XML documents that conformed to a given XML schema. Notice that the "lossless from XML" constraint defined in Chapter 3.1.2 corresponds to this notion. Of the three properties that hold when the "lossless from XML" constraint is satisfied (properties P1-P3 in Chapter 3.1.2), the XML_to_SQL algorithm made use of just one property, namely, property P1. It did not take advantage of

the fact that the other two properties also hold. While property P1 suffices for correctly translating path expression queries into SQL, we show how by exploiting the fact that the other two properties hold, we can translate XML queries into efficient SQL.

The basic idea is that since all the three properties hold, the XML-to-Relational mapping fully captures the entire relational schema. By using this information in an intelligent fashion, we obtain a mapping_aware translation algorithm that eliminates computation in the relational query made redundant by the mapping information. In this chapter, we extend the XML_to_SQL algorithm to illustrate how the "lossless from XML" constraint can be fully exploited to generate efficient SQL queries.

The rest of this chapter is organized as follows. We first present some more example scenarios in Section 4.1 to illustrate the various optimizations that can be performed in the query translation process. We present the mapping_aware algorithm in Section 4.2 for tree schema. This algorithm is an extension of the XML_to_SQL algorithm and uses the mapping information to avoid generating redundant computation in the final relational query. We then present the algorithm for recursive XML schema in Section 4.3. Finally, we present experimental results to show the performance improvements achieved by the mapping_aware algorithm.

4.1 Motivation for mapping_aware techniques

We next illustrate how an XML-to-SQL translation algorithm can use the mapping information and generate more efficient relational queries. We first give an example situation, in which the query is recursive (has the wildcard //) and then present another example where the XML schema is recursive as well.

Part of the XMark benchmark [XMa] XML schema is given in Figure 17. The XMark

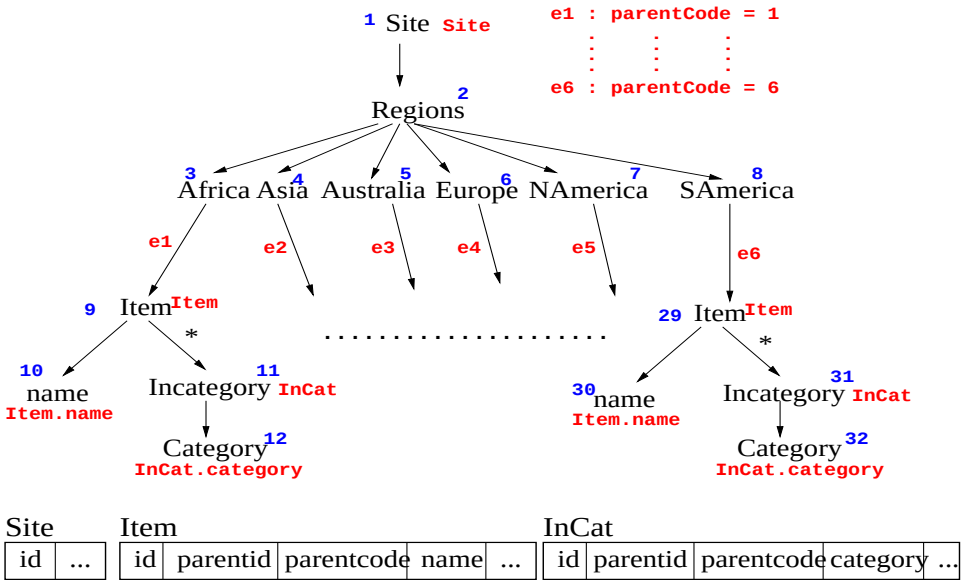

Figure 17: XMark benchmark schema and a sample relational decomposition

benchmark models an auction site and it has been widely used in research literature to evaluate XML data management strategies [BKTT04, CAYLS02, CJLP03, DT03a, DTCO03, FHR+02, FSC+03, GSBS03, GST04, JLWO03, MS03, TH02]. One way of mapping the XML schema into relations is given in the figure. Consider the evaluation of the following query Q_1, which returns all the item categories: //Item/InCategory/Category.

Consider the following simple algorithm for handling queries with wildcards (//) [JMS02]. Identify all paths in the schema that satisfy the query. For each path, generate a relational query by joining all relations appearing in this path. The final query is the union of the queries over all satisfying paths (six paths for Q_1). This algorithm will result in the following SQL query SQ_1^1.

select C.category

from Site S, Item I, InCat C

where S.id = I.siteid and I.id = C.itemid and I.continent='africa'

union all ... (6 queries one for each continent except Antarctica)

The XML_to_SQL algorithm presented in the previous chapter and several other algorithms proposed in literature [FMS01, SKS+01] will also result in a similar relational query. On the other hand, looking at the XML-to-Relational mapping, we see that there are six nodes in the schema whose values are stored in the column `InCat.Category` and the query Q_1 selects all of them. Since we know that the "lossless from XML" constraint is satisfied, we can translate Q_1 into the following simpler optimized SQL query OQ_1.

```
select  category
from    InCat C
```

Notice how we are able to replace a query SQ_1 that was the union of six queries, each with 2 joins, by a simpler scan query OQ_1. The two queries SQ_1 and OQ_1 are equivalent given that the "lossless from XML" constraint holds under this particular XML-to-Relational mapping.

Let us now consider another example scenario that involves a recursive XML schema. We saw how to translate the query $Q = //E10$ on the schema graph in Figure 13. The final relational query SQ was given in Figure 16 and it is a fairly complex recursive relational query. Even though the XML query is fairly simple, since the XML schema is fairly complex, the final SQL query is also complex. The main reason for this is the fact that the SQLGen stage is top-down, hence the complexity of the XML schema is reflected in the final SQL query.

Again, by using the mapping information, we know that the relation `R10` matches the path expression "//E10". Since the "lossless from XML" constraint holds, the following query OQ will be equivalent to SQ under this particular mapping and, as a result, a correct translation for Q.

```
select  elemid
```

from R10

Notice how this SQL query is much simpler than the query output by the XML_to_SQL algorithm and the associated performance benefits are obvious.

At this point the reader may wonder if XML query minimization will be helpful in the above scenario. Note that the above path query is already minimal (and so will be all the other example queries we use throughout this thesis), it is the translation that caused the problem, not the original XML query. All the work on XML query minimization [AYCLS01, FFM03, Ram02] and on containment and equivalence of path expression queries [DT01, MS02, Woo02] is complementary to the focus of our work and can be used as the first stage to minimize the input XML queries.

In both of the above examples, we saw that by using the mapping information we can eliminate recursive sub-components in the SQL query, combine multiple query components and eliminate redundant joins. We were able to this by exploiting the fact that the "lossless from XML" constraint holds. To keep things simple, we gave examples where the entire query matched a single relation. In general, only parts of the query generated by current algorithms, including our XML_to_SQL algorithm may be implied by the mapping. For example, for a XML query with //, we may be able to perform these optimizations to varying degrees across different satisfying paths. We may be able to eliminate some recursive sub-queries across one path (as they are implied by the mapping) and certain others across a different path. In the next section, we extend our XML_to_SQL algorithm to use the mapping information and perform such optimizations for the class of path expression queries over a given XML-to-Relational mapping.

procedure mapping_aware_translation(Q, S)
begin
1. Perform PathId using the mapping S and query Q.
   Let S_{CP} be the resultant cross-product schema.
2. Prune S_{CP} making use of the "lossless from XML" constraint .
3. Translate the pruned S_{CP} into SQL.

Figure 18: Query Translation Algorithm using the "lossless from XML" constraint

4.2 Exploiting the "lossless from XML" constraint for Tree XML Schemas

In this section, we show how we can use the "lossless from XML" constraint while translating queries over a tree schema in order to generate a query that may be more efficient than the corresponding naively generated query. We extend this to directed acyclic graph (DAG) and recursive XML schema in Section 4.3.

The outline of this algorithm is given in Figure 18. The output of the PathId stage is pruned making use of the "lossless from XML" constraint and this pruned cross-product schema is the input to the SQLGen stage. The PathId stage remains unchanged from the XML_to_SQL algorithm. We explain the pruning stage and the SQLGen stage in this section.

4.2.1 Basic Idea behind the Algorithm

Consider the set of paths P in the schema that end in a node annotated with the column $R.C$. Since the XML-to-Relational mapping satisfies the "lossless from XML" constraint, we know that every tuple in the relation R corresponds to the value of (exactly) one element in the XML document. In other words, each tuple in relation R will appear in the result of the SQL query corresponding to (exactly) one root-to-leaf path in the

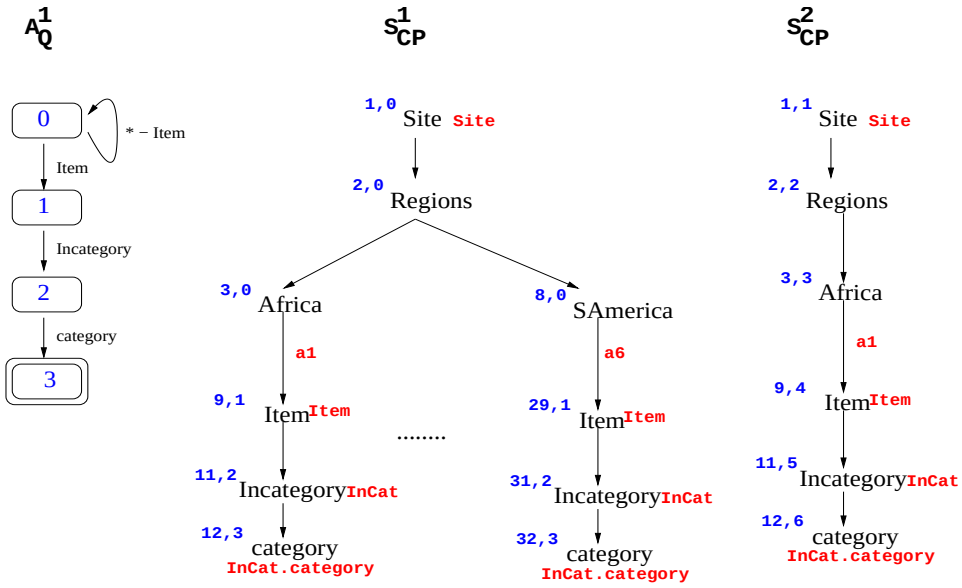

Figure 19: Result of PathId stage for Q_1 and Q_2

XML schema. Notice that this gives us two guarantees: (i) No two root-to-leaf paths will have a common tuple in their results and (ii) All the root-to-leaf paths combined together correspond to a scan of the column $R.C$.

Let us look at some example queries to see how we can use the above information to prune the cross-product schema. In the process, we identify two important concepts that form the core of our algorithm. We use the mapping schema in Figure 17 in the following discussion, and will discuss how the algorithm works informally in these specific examples before turning to specifying the full algorithm.

Consider the example query Q_2, /Site/Regions/Africa/Item/Incategory/Category. The output of the PathId stage is the cross-product schema S^2_{CP} given in Figure 19. We use the first component of the node identifiers in the cross-product automaton to identify the nodes in the following discussion. There is a single path $p =<1, 2, 3, 9, 11, 12>$ in S^2_{CP} and issuing the query $SQL(p)$ is a correct translation for Q_1. Our goal in this case is to find a shorter suffix p' of p such that $SQL(p)$ is an equivalent translation under the given

mapping.

Now, for any suffix p' of p, $SQL(p')$ will certainly contain all the results for Q_1. We also know that the "lossless from XML" constraint is satisfied. So, if we ensure that $SQL(p')$ does not have any results corresponding to some other path in the mapping schema, then we are done. We do this as follows: We first start with the smallest suffix $p' = \{<12>\}$. The equivalent query will be a scan of the InCat.category column. We notice that there is a path $q = \{<32>\}$ in the original schema S that has the same annotation. So, $SQL(p')$ will also return results corresponding to the path q. Since the queries corresponding to the paths p' and q have common results, we say that they are in *conflict* with each other. Now, since q does not have a corresponding path in S^1_{CP}, we know that it is not a part of the query result. This implies that for $p' = \{<12>\}$, $SQL(p')$ will have results corresponding to this path q as well.

So, we go up one level and set $p' = \{<11, 12>\}$. The same conflict persists with the path $q = \{<31, 32>\}$ and we have to increment p' by another level. Repeating this, we get to $p' = \{<3, 9, 11, 12>\}$. Now we see that the corresponding path $q = \{<8, 29, 31, 32>\}$ is not in conflict with p' due to the difference in the parentCode condition (edge annotations e_1 and e_6). In fact, there is no other path in conflict with p' elsewhere in the schema. So, the query $SQL(p')$ will be equivalent to $SQL(p)$ and it is a correct translation for Q_1. This query is given below:

```
select   C.eid
from     Item I, InCat C
where    I.id = C.parentid and I.parentCode=1
```

Contrast this with $SQL(p)$, which will be the relational query output by existing algorithms that do not use the "lossless from XML" constraint. $SQL(p)$ has an additional

join with the Site relation, which has been removed using the mapping information. This leads us to the first important concept that we will use in developing an algorithm to exploit the "from XML" constraint while doing path expression to SQL translation.

Concept 1: For every path p in the result automaton S_{CP}, we need to identify a suffix p' that has the following property: SQL(p') will not return results corresponding to any path not appearing in the query result.

Now let us revisit query Q_1 from Section 4.1. The output of the PathId stage is the cross-product schema S^1_{CP} given in Figure 19. There are six satisfying paths in the schema (we will denote these p_1 to p_6). We need to find the shortest suffix for each of these paths that will together result in a correct SQL query. While we can handle each path independently in a fashion similar to query Q_1, we perform an additional optimization — we combine the queries for different paths whenever possible. In addition to grouping the SQL queries for paths with similar relational joins, this optimization also allows us to eliminate longer prefixes as we will see in this example.

Consider the path $p_1 =<1, 2, 3, 9, 11, 12>$. We start with the suffixes $p'_1 = \{<12>\}$ for this path. The path $q = \{<1, 2, 8, 29, 31, 32>\}$ is in conflict with p'_1. So, $SQL(p'_1)$ will have results corresponding to the path q. But this time q appears in S^1_{CP}, which means that it is a part of the query result (categories of South American items are a part of the query result). The corresponding suffix is $q'_1 = \{<32>\}$. At this point, if we issue SQL queries for the two paths p'_1 and q'_1 separately, then we will get duplicate results. All the item categories will be returned twice. So, we need to go further up the tree for both the paths. On the other hand, if we issue a common query for the two paths, then we need not worry about the common results across these paths. In this case, since the two paths p'_1 and q'_1 have the same relation sequence (scan of the `InCat.Category column`), we can combine the queries for these two paths. Similarly, the other four schema nodes

that have the annotation `InCat.category` are also part of the query result. So, the suffix $p'_1 = \{<12>\}$ suffices for the path p_1. We reach a similar decision for the other five paths. Finally, combining the queries for the six paths, we obtain the final relational query SQ_1^2 (see Section 4.1) that is a scan of the `InCat.Category` column.

Notice how by using the "lossless from XML" constraint, we are able to replace a query SQ_1^1 that was the union of six queries, each with 2 joins, by a simpler scan query SQ_1^2.

Concept 2: Suppose we are considering suffixes p' and q' for paths p and q respectively. We need not worry about the queries corresponding to the two suffixes generating common results as long as we issue a combined single SQL query for them.

From the above discussion, we see that by making sure that the above two concepts are satisfied we can find the required prefix for every path in the cross-product schema. Notice that we are able to do this only because the "lossless from XML" constraint holds. This constraint implies that there is a "one-to-one" correspondence between the relational data and the XML data. So, if we know that for a path p and a suffix p', SQL(p') satisfies concept 1, we are then guaranteed that it does not return any extra results. This holds because the mapping completely captures the relational data, i.e., all the root-to-leaf queries together represent the entire relational data. Similarly, concept 2 holds because the relational data stores values corresponding to each XML element separately, i.e., the result of no two root-to-leaf queries will have the value of the same XML element.

On the other hand, if we did not use the fact that the "lossless from XML" constraint holds for this instance, things will be a lot different. For example, suppose the join between `Item` and `InCat` relations was not a key foreign-key join. Then, the "lossless from XML" constraint may no longer be valid — for example, $RtoL(12)$ may return some tuples in the `InCat` relation multiple times as they join with several tuples in the `Item`

relation. So, SQ_1^2 will no longer be a correct translation for query Q_1. The fact that the "lossless from XML" constraint is satisfied makes it a lot simpler to design a good query translation algorithm.

In the above examples, notice that we used the notions of combinability and conflict in the context of two paths in the mapping. We formally define these notions and then describe the pruning and SQLGen stages of the algorithm.

4.2.2 Terminology

We next introduce some terminology that will be used in the full specification of the translation algorithm. Whenever we refer to paths, we mean paths in the schema graph that end in leaf nodes.

Let $p = <n_1, \ldots, n_k>$ be a path in the schema graph, ending in a leaf node. We refer to n_k as $p.last$. Let RelSeq(p) denote the sequence of relations joined in SQL(p) in a top-down order. For example, for the path $p = <1, 2, 3, 9, 11, 12>$, RelSeq(p) = <Site,Item,InCat>.

We say that two paths p_1 and p_2 are *combinable* if the corresponding relation sequences $RelSeq(p_1)$ and $RelSeq(p_2)$ are the same and $Annot(p_1.last)$ and $Annot(p_2.last)$ are the same. Note that combinability is an equivalence relation. Combinable paths are useful in identifying when we can rewrite a union query, say (SQL(p_1) union all SQL(p_2)), as a SQL query without unions. For example, let $p' = <1, 2, 8, 29, 31, 32>$. Then the paths p and p' are combinable. From the "lossless from XML" constraint we know that the resulting query does not have to retain any duplicate results. This allows us to combine the two queries even if they have overlapping results.

Given two paths p_1 and p_2, we define when the two paths are in *conflict*. Intuitively,

the two paths are in conflict if the result of SQL(p_1) and SQL(p_2) will have common results. Here, by common results, we refer to the two queries returning the value of a common element in the original XML document. For example, the paths p and $p^{'}$ are not in conflict as they return the categories of Africa items and South America items respectively. So, while the results of SQL(p) and SQL($p^{'}$) may have common values, these will be the values of different elements in the original XML document. On the other hand, consider $p^{''} =<29, 31, 32>$. The two paths p and $p^{''}$ are in conflict as the corresponding SQL queries overlap: the categories of africa items are common to the two query results.

Given two paths, p_1 and p_2, we say that they are in conflict if the following conditions hold.

- $RelSeq(p_1)$ is a suffix of $RelSeq(p_2)$ or vice versa.
- Without loss of generality, let $RelSeq(p_1)$ be a suffix of $RelSeq(p_2)$. Let p_3 be the longest suffix of p_2 such that $RelSeq(p_1) = RelSeq(p_3)$. Let $RelSeq(p_1) = RelSeq(p_3) =<R_1, \ldots, R_k>$. Then,

  there is no column $R_i.C$ such that $SQL(p_1)$ has a selection $R_i.C = val_1$, and $SQL(p_3)$ has a selection $R_i.C = val_3$, where $val_1 \neq val_3$.

The former condition checks if the two paths differ in the sequence of relations joined. If each sequence has a join not present in the other, they will not generate common results. We know this from the "lossless from XML" constraint. The latter condition checks if a conflicting edge annotation is present on any relation across the two paths; if so they will not generate common results and are not in conflict.

If p_1 is not in conflict with p_2, then we say that p_1 is *safe* from p_2. In the presence of the "lossless from XML" constraint, we know that no two root-to-leaf paths p_1 and p_2

procedure Pruning(S_{CP},S)
begin
1. Let PathSet = {<n> |n is a leaf node in S_{CP}}.
2. do
3. foreach ($p \in$ PathSet)
4. Let Conflict(p) denote the set of root-to-leaf paths in S that are in conflict with CurrPath(p)
5. If ($\exists p' \in Conflict(p)$ that does not match the query)
6. Increment p by one level
7. endFor
8. while (some path was modified in the previous iteration)
9. do
10. Let p and q be two paths in PathSet that are in conflict
11. If p and q are not combinable
12. Let RelSeq(p) be no longer than RelSeq(q)
13. Increment p by one level
14. while (some path was modified in the previous iteration)
15. Return $PathSet$
end

Figure 20: Pruning stage for tree XML schema

have common results. In other words, root-to-leaf paths are always safe from each other. Observe that this may not be true if the integrity constraint is not satisfied.

4.2.3 The Pruning Stage

We present the pruning algorithm for translating path expression queries in Figure 20. Recall that the result of the PathId stage is S_{CP}, which represents all the matching paths in S. For each path, instead of constructing the SQL query from the root of the schema, our goal is to start bottom-up and stop at the lowest possible level.

So, for every path $p \in S_{CP}$, we start with just the leaf node and keep going up until we find a suffix p' that satisfies the following property:
For every path p'' in conflict with p'

- p'' appears in the cross-product schema S_{CP} and

- The queries corresponding to p' and p'' are combinable.

For every path p, steps 2-8 make sure that conflicts with paths not in the query result are resolved. Steps 9-14 make sure that duplicate results are not produced, i.e, conflicts with paths appearing in the query result are resolved, if the corresponding relation sequences are not combinable. We do the above computation for all paths in S_{CP} and stop when we have found the required suffixes for all of them. Since, each iteration in both the while loops will increment the length of the path, they will terminate in at most d iterations (combined), where d is the length of the longest path in S_{CP}.

In the pruning stage, for each path $p \in SCP$, we have removed some prefix of p and have the suffix path p' in PathSet instead of p. For correctness, we need to argue that the same results are returned if we use p' instead of p. Since the "lossless from XML" constraint is satisfied, it can be easily shown that every suffix p'' of p will return a superset of results, i.e., $SQL(p) \subseteq SQL(p'')$. So, we need to make sure that extra results are not returned. In the above algorithm, through additional checks we make sure that this does not happen, i.e., tuples corresponding to schema paths not matching the query are not returned and tuples corresponding to schema paths matching the query are returned exactly once. The first while loop in the algorithm takes care of the former and the second while loop takes care of the latter. These checks make use of the fact that the "lossless from XML" constraint is satisfied.

While incrementing a path by one level, we make the following optimization. If the edge we traverse to go up one level has an edge annotation on it, we split the operation of going up one level into two parts: (1) use the edge annotation to see if that suffices and (2) go up to the parent node, if necessary. By doing this optimization, we may be able to save a join operation. We will later show an example in Section SQLGen that

uses this optimization.

4.2.4 The SQLGen Stage

The result of the pruning stage is a set of paths, `PathSet` in the cross-product schema S_{CP}. We partition this set based on the relation sequence and issue a SQL query for each equivalence class created. The final query is the union of the queries corresponding to all the partitions.

Suppose a single equivalence class C had two paths p_1 and p_2 in it. Since p_1 and p_2 are combinable, SQL(p_1) and SQL(p_2) involve the same relations. Thus, the from clause of the grouped query $SQL(C)$ contains these relations. Let C_{common} denote the set of conditions that are common to both SQL(p_1) and SQL(p_2). Let C_i denote the conditions corresponding to p_i that are not present in C_{common}. The where clause has the condition (C_{common} and (C_1 or C_2)). This solution generalizes when we have to combine more than two paths.

We have the following theorem about the correctness of the above query translation algorithm.

THEOREM **2** *Given a tree XML-to-Relational mapping $\mathcal{T}$, a path expression query P and the guarantee that the "lossless from XML" constraint is satisfied, the mapping_aware algorithm outputs a correct equivalent SQL query.*

Proof: The result of the PathId stage is a set of paths, `PathSet`, in the cross-product schema S_{CP}. From Proposition 2 in Chapter 3, we know that the set S_{CP} has exactly all the schema paths matching the query. So, the union of the queries corresponding to all root-to-leaf paths in S_{CP} is a correct SQL translation. In the pruning stage, for each path p, we remove some prefix of p and have the suffix path p' in PathSet instead of p.

For correctness, we need to show that the same results are returned if we use p' instead of p.

Let $P_1, P_2, \ldots, P_k$ be the sets of paths obtained by partitioning PathSet based on combinability of the paths. We show that for each partition $P_i = \{p_i^1, p_i^2, \ldots, p_i^l\}$, the SQL query $SQL(P_i)$ constructed in the SQLGen stage returns the same results as $\cup_{j=1}^{l} RtoL(p_i^j.last)$.

First, we show that $SQL(P_i) \supseteq \cup_{j=1}^{l} RtoL(p_i^j.last)$ (under multiset semantics). Since the "lossless from XML" constraint is satisfied, we show that every suffix p'' of a root-to-leaf path p will return a superset of results, i.e., $SQL(p'') \supseteq SQL(p)$. Let $R.C$ be the annotation for $p.last$. Note that the prefix only adds additional selection and join conditions. So, any tuple from relation R that appears in the result of $SQL(p)$ will also appear in the result of $SQL(p'')$. Moreover, since the "lossless from XML" constraint holds, each tuple from relation R appears at most once in the result of $SQL(p)$. Hence it follows that $SQL(p'') \supseteq SQL(p)$. Since, this holds for each path in P_i individually, we have $SQL(P_i) \supseteq \cup_{j=1}^{l} RtoL(p_i^j.last)$.

We next show that $SQL(P_i) \subseteq \cup_{j=1}^{l} RtoL(p_i^j.last)$ (under multiset semantics). Suppose a tuple t from relation R appears in the result of $SQL(P_i)$. Since the "lossless from XML" constraint is satisfied, this tuple t will appear in the result of $SQL(q)$ for some path q in the XML schema ending in a node annotated with a column $R.C$. Depending on whether the path q matches the query and, if so, how the pruning stage handled the path, there are three possibilities: (i) q does not match the query, (ii) q matches the query but the corresponding suffix does not appear in P_i (iii) q matches the query and the corresponding suffix appears in P_i. Scenario (i) cannot happen as steps 2-8 (the first while loop) in the algorithm in Figure 20 ensure that the results returned by every

suffix query are non-overlapping with all the schema paths not satisfying the query. Similarly, scenario (ii) cannot happen as the second while loop (steps 9-14) ensure that the results returned by every suffix query are non-overlapping with schema paths belonging to a different partition in PathSet. So scenario (iii) is the only option that implies that $SQL(P_i) \subseteq \cup_{j=1}^{l} RtoL(p_i^j.last)$.

We have shown that for each partition P_i the SQL query constructed by the mapping_aware algorithm is equivalent to the $\cup_{j=1}^{l} RtoL(p_i^j.last)$. Since the mapping_aware algorithm does not perform any optimizations across partitions, the correctness of the algorithm for the entire query follows. □

In the example queries we saw earlier, the second `while` loop is trivially satisfied — for Q_1 all the six paths in PathSet were combinable, while for Q_2 there was only one path in PathSet. We next present an example scenario where we find two paths that appear in the result, but are not combinable.

We use the example XML schema and mapping in Figure 21 in the following discussion. All `a` elements are stored in relation `R1`. The child elements are all stored in relation `R2`, with the `pc` column distinguishing between `b,c` and `d` children. Similarly, the children of `b` elements are stored in relation `R3`, with the `pc` column distinguishing between `x` and `y` children. The children of `c` and `d` elements are all stored in relation `R3`. Since all the children are `x` elements in these two cases, the `pc` column is not specified in these cases. An important point to note here is that since the value of the column is not specified for tuples corresponding to schema nodes 56 and 57, any value in the corresponding domain (including 1,2 and null) is allowed. As we will see later, this difference in the information available about the column `R3.pc` needs to be handled correctly when we attempt to find

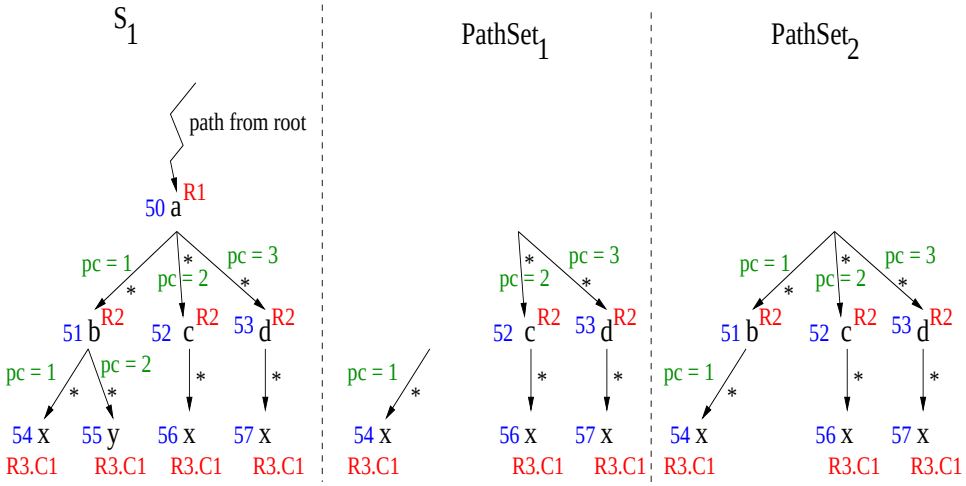

Figure 21: Example mapping S_1 to explain the pruning algorithm

the correct suffix for each path.

Notice that we are only concerned with the subtree rooted at element `a` and this is only a portion of the entire schema. Let us assume that the element name `x` does not occur anywhere else in the XML schema. Also, no other leaf node in the schema is annotated with the column `R3.C1`.

Consider the evaluation of query Q_3 that returns all `x` elements, //x. The PathId stage will identify the three paths p_1, p_2 and p_3 ending in nodes 54,56 and 57 respectively. Suppose we execute the algorithm in Figure 20. After the first while loop (steps 2-8), we observe that the suffixes are $p'_1 = <51, 54>, p'_2 = <50, 52, 56>, p'_3 = <50, 53, 57>$. The only conflicting path p_4 ends in node 55. For p_1, the annotation on edge $<51, 54>$ makes it safe from p_4. So, going up one level was sufficient. Note that the edge annotation was sufficient in this case, and we did not need the join with relation R_1 ($annot(51)$). For p_2 and p_3, there was no annotation on the edges $<52, 56>$ and $<53, 57>$. So, we had to go up one more level till node 51. The annotation on edges $<50, 52>$ and $<50, 53>$ are different from the one on edge $<50, 51>$. Hence by going up two levels, p'_2 and p'_3 become safe from p_4.

Let us observe what happens if we skip the next while loop. The resulting PathSet is shown in Figure 21 as PathSet_1 and the corresponding SQL query SQ_3^1 is

```
select  R3.C1
from    R3
where   R3.pc = 1
union all
select  R3.C1
from    R2,R3
where   R2.id = R3.parentid and R2.pc IN {2,3}
```

Notice how values of `x` elements corresponding to schema nodes 56 and 57 may appear twice in the query result. This happens when the `pc` column of these tuples has a value of 1, which is valid as the XML-to-Relational mapping conveys no information about the value of the `pc` column of these tuples. So, the above query is not a correct translation for Q_1. While it returns the correct set of results, it may return duplicates.

In order to avoid generating duplicate results, we go up the schema further (steps 9-14 in algorithm 20) resulting in the set of paths PathSet_2 shown in Figure 21. This will result in the following correct SQL query SQ_3^2:

```
select  R3.C1
from    R2,R3
where   R2.id = R3.parentid and (R2.pc IN {2,3} or
                (R2.pc = 1 and R3.pc = 1))
```

The above example illustrates the point that we should make sure that two paths p' and q' that appear in the query result are not in conflict, if they are not combinable (i.e., if we are going to construct the queries for them separately). The second while loop (steps 9-14) in the algorithm takes care of this.

4.2.5 Some Alternative Solutions

We would like to point out that there are two other alternative solutions to the problem in the above example: (i) generate and then remove duplicates or (ii) consider a different definition of combinability that will generate more complex SQL queries. We explain these two options next.

Generate and Eliminate Duplicates

Here, we apply a distinct clause finally when duplicate results may be generated in this fashion. In this case, the final SQL query will be SQ_3^3:

```
with temp(id,value) as (
    select  R3.id, R3.C1
    from    R3
    where R3.pc = 1
    union all
    select  R3,id, R3.C1
    from    R2,R3
    where R2.id = R3.parentid and R2.pc IN {2,3}
)
select   distinct(id,value)
from    temp
```

If we follow this approach, we do not need the second while loop (steps 9-14) in the pruning algorithm in Figure 20. However, we need to replace it with a check to see if the distinct clause is required. This modification is shown in Figure 22.

In this approach, notice how we are generating duplicates and then eliminating them. The assumption here is that this is likely to be a cheaper operation than following the

10. Let p and q be two paths in PathSet that are in conflict
11. If p and q are not combinable
12. DupEliminationRequired = true

Figure 22: Modified pruning stage

conservative approach, where additional joins are introduced. Moreover, situations when duplicate elimination is required in the above fashion are unlikely to be common in practice.

We can perform duplicate elimination in the XML storage scenario in a fairly simple fashion: using the *key* field of the relation corresponding to the leaf nodes(`id` field) suffices. If the query returns results from multiple relations, we apply a distinct clause to each of them separately.

In contrast, a fairly complex mechanism is required to perform duplicate-elimination in the XML publishing scenario [KKN04].

Changing the Definition of Combinability

Here, in order to handle the problem with PathSet_1, we modify the definition of combinability of relations. Recall that according to our definition of combinability in Section 4.2.2, we combine the queries for two paths if they are on the same relation sequence. Suppose, we extend this to say that we will combine the queries even when one sequence is a suffix of the other. Then, the following SQL query SQ_3^4 will be produced:

```
select  R3.id
from    R3
where   R3.pc = 1 or exists (
            select*
            from R2
```

```
        whereR2.id = R3.parentid and R2.pc IN {2,3}
)
```

Notice how the common suffix appears in the main SQL query and the extra prefix appears as a nested query. This translation works because the relational data satisfies the "lossless from XML" constraint. As a result, we know that no tuple from relation R3 will appear multiple times in the query result. So, using a nested subquery for the join with relation R2 is correct. Notice how the structure of the final SQL query is considerably modified with this definition of combinability — nested queries are introduced even for simple path expression queries without predicates.

Discussion

From the above example, we notice that there are two important factors in designing a mapping_aware algorithm for tree schema: (i) whether we want to avoid generating duplicate results (and, of course, removing them eventually) and (ii) the class of SQL queries generated. We saw that for tree schemas there are at least three mapping_aware algorithms possible for simple path expression queries based on the answers to the above two questions.

Notice how we are able to generate three different equivalent queries SQ_3^2, SQ_3^3 and SQ_3^4 for the same XML query Q_3. Identifying that these three queries are equivalent and choosing among them will be non-trivial, if we did not use the additional knowledge that the "lossless from XML" constraint is satisfied.

At this point, the reader may wonder if scenarios like the one discussed above are likely to be common in practice. We believe the answer is "no" — in a lot of actual scenarios the three algorithms will result in similar queries. Nevertheless, since we make minimal

assumptions about the XML-to-Relational mapping and do not interpret or place any restrictions on the semantics of different relational columns (like the `pc` column in the above example), the above mapping is a valid mapping. So, our algorithm has to (and does) handle these inputs correctly.

4.3 Exploiting the "lossless from XML" constraint for complex XML Schemas

In this section, we extend the techniques for exploiting the "lossless from XML" constraint to more complex XML schemas: directed acyclic graph (DAG) and recursive schemas. The outline of the algorithm remains the same as for tree XML schema (Figure 18). We need to augment the pruning stage to address some additional challenges that arise for complex XML schemas such as extending the notion of combinability to handle more complex XML schemas.

4.3.1 Combinability for Complex Schema

Consider the mapping shown in Figure 23. Suppose the query result is the set of paths shown in the figure. First of all, as shown in Chapter 3.2.1, enumerating all the matching paths may be expensive, as the number of paths may be exponential in the size of the DAG schema. So, the XML_to_SQL algorithm in uses the *with* clause in SQL99 to represent common computation present in the DAG schema. For example, the paths $<11, 14, 21, 24>$, $<11, 14, 21, 25>$, $<11, 15, 21, 24>$ and $<11, 15, 21, 25>$ share some common computation as represented by the common schema nodes across them. So, the SQL fragment generated will be

with temp_21 as (

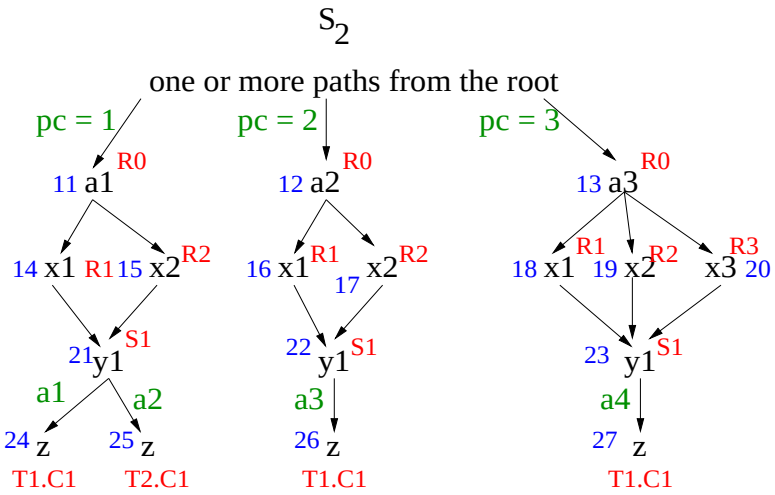

Figure 23: Example mapping S_2 to explain issues in defining combinability for complex schema

```
    select S1.*
    from R0,R1,S1
    where R0.id = R1.parentid and R0.parentcode = 1
          and R1.id = S1.parentid
    union all
    select S1.*
    from R0,R2,S1
    where R0.id = R2.parentid and R0.parentcode = 1
          and R2.id = S1.parentid
)
select  T1.C1
from    temp_21, T1
where   temp_21.id = T1.parentid and a1
union all
select  T2.C1
from    temp_21, T2
```

where temp_21.id = T2.parentid and a2

Notice how the *with* clause is used to group together paths that have common computation. In a similar fashion, the *with* clause is used to handle the other two subtrees rooted at nodes 12 and 13 also. The final query result is the union of the three queries.

Let us revisit the definition of combinability of paths in this context. Suppose we want to combine the SQL queries corresponding to the subtrees rooted at nodes 11 and 12. This implies that we will have a single SQL query for the six paths. The following problem arises: while the structure of the *with* clause ($R0 \bowtie R1 \bowtie S1 \bigcup R0 \bowtie R2 \bowtie S1$) is common for the two subtrees, the subtree rooted at node 11 has two suffixes, edges $<21, 24>$ and $<21, 25>$. On the other hand, the subtree rooted at node 12 has only one suffix, edge $<22, 26>$. So, if we combine the *with* clause for the two subtrees in a simple fashion, we will generate two spurious paths $<12, 16, 22, 25>$ and $<12, 17, 22, 25>$. In order to avoid this, we need to somehow differentiate between tuples corresponding to nodes 21 and 22. One way to do this is to use the annotations on the incoming edges to the nodes 11 and 12. This helps us in differentiating between tuples corresponding to nodes 21 and 22 and the resulting query is:

```
with temp_21_22 as (
    select S1.*,R0.parentcode
    from R0,R1,S1
    where R0.id = R1.parentid and R0.parentcode = 1
          and R1.id = S1.parentid
    union all
    select S1.*,R0.parentcode
    from R0,R2,S1
```

```
    whereR0.id = R2.parentid and R0.parentcode = 1
          and R2.id = S1.parentid
)
select   T1.C1
from     temp_21_22, T1
where    temp_21_22.id = T1.parentid and
          ((a1 and temp_21_22.parentcode = 1)
            or (a3 and temp_21_22.parentcode = 2))
union all
select   T2.C1
from     temp_21_22, T2
where    temp_21_22.id = T2.parentid and a2
          and temp_21_22.parentcode = 1
```

Notice how if the annotations `a1` and `a3` are different, we again need to differentiate between tuples corresponding to schema node 21 and 22.

Suppose we want to combine the SQL queries corresponding to the subtrees rooted at nodes 12 and 13 instead. In this case, notice how the path $<13, 20, 23, 27>$ has no equivalent path in the other subtree. So, we need to be careful and ensure that we do not create a spurious path $<13, 20, 23, 26>$ (through the join $R0 \bowtie R3 \bowtie S1 \bowtie T1$).

To summarize, from the above discussion we observe that for DAG schemas, we have a lot of options in defining when and how we combine the queries corresponding to different subtrees. Notice that whenever we have subtrees that have a different set of relation sequences (such as subtrees rooted at nodes 11,12 and 13), we have a choice: either combine the queries by maintaining some more state information or construct the

queries separately. Whenever we need to maintain state information, we are also making the *where* clause of the SQL query complex. More importantly the relational query optimizer may have problems in optimizing the final SQL query efficiently, as it has no way of interpreting the semantics of this additional state information.

In this thesis, we use a simple definition for combinability over complex schemas: one that requires minimal additional state information. Intuitively, we combine two subtrees if they are similar: the joins involved in each part of the resultant query are identical (the selection conditions can be different). This definition of combinability extends naturally to recursive schema also. Notice that this is a generalization of the definition of combinability in Section 4.2.2 for single paths. We next define this formally as follows.

A graph path gp corresponds to a subgraph of the schema. It is a concise way of representing a large number (possibly infinite) paths. Let Paths(gp) denote the set of paths represented by the graph path.

For a graph path gp, let Template(gp) denote the corresponding graph constructed based on the relational annotations. The template graph represents the structure of the SQL query corresponding to the subtree. For example, Template(11) will be a graph with five nodes, one for each node in the subtree (since each edge traversal corresponds to a join operation). Each node will be labelled with the name of the corresponding relation. In this case, the nodes will be labelled with relation names R0,R1,R2,T1 and T2. For each edge $e \in gp$ that corresponds to a join between two relations, we add a corresponding edge in the template graph. Note that the template graph is similar to the original subtree; if the latter is recursive the template graph is recursive as well.

We say that two graph paths gp_1 and gp_2 are combinable if the corresponding templates are isomorphic.

procedure Pruning(S_{CP},S)
begin
1. Let PathSet = {<n> |n is a leaf node in S_{CP}}.
2. do
3. foreach ($p \in$ PathSet)
4. Let Conflict(p) denote the set of root-to-leaf paths in S that are in conflict with p
5. If ($\exists p' \in Conflict(p)$ that does not match the query)
6. Increment p by one level
7. endFor
8. while (some path was modified in the previous iteration)
9. do
10. Let p and q be two (graph) paths in S_{CP} that are not combinable
11. If $\exists$ path $q' \in q$ such that q' is in conflict with p
12. Let p' be the path $\in p$ that is in conflict with q'
13. Increment p' by one level
14. while (some path was modified in the previous iteration)
15. Return $PathSet$
end

Figure 24: Pruning stage for recursive XML schema

Just like the tree schema case, the above definition makes use of the fact that as the "lossless from XML" constraint is satisfied, we can combine any two graph paths even if they have overlapping results. Since the result of a path expression query returns the values of all matching XML elements (exactly once), and there is a "one-to-one" correspondence between XML elements and relational tuples, this implies that no relational tuple will appear multiple times in the result of any equivalent SQL query.

4.3.2 The Pruning Stage

The pruning algorithm for recursive mappings is shown in Figure 24. While it looks very similar to the tree schema case, there are some important differences. These include the definition of combinability and conflict, and how we keep track of all the matching paths in PathSet.

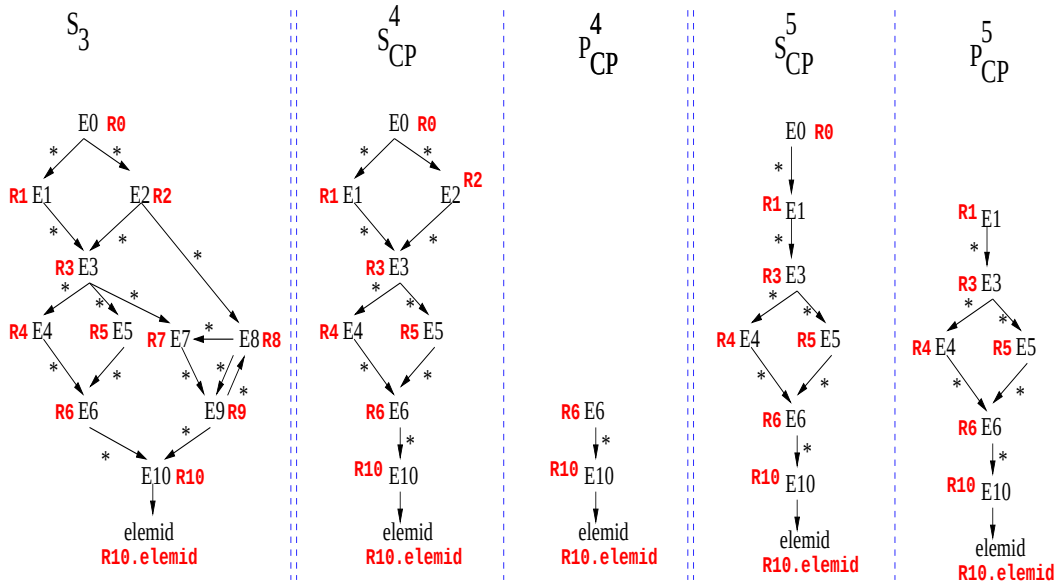

Figure 25: Examples to illustrate the mapping_aware algorithm

We say that a path p is in *conflict* with a graph path gp if p is in conflict with some path $q \in \text{Paths}(gp)$.

Another important difference is that the set of matching paths is maintained as subgraphs of the cross-product schema (as there may be infinitely many of them if we enumerate them). Contrast this with the algorithm for tree schema where we explicitly keep track of all the paths. We describe how we increment the graph paths by one level (steps 6 and 13) through some examples later in this section.

Also note that in order to check the condition in Steps 4-5, we need to enumerate all the paths that do not appear in the query result. Since the query automaton for simple path expression queries is a deterministic finite automaton, we can do this efficiently.

The SQLGen stage is similar to the original algorithm proposed in Chapter 3.2.2, with a slight modification. We combine the queries corresponding to two different graph paths in PathSet, if they are combinable. To do this, we partition the graph paths based on combinability (similar to Section 4.2.4) and construct the SQL query for each

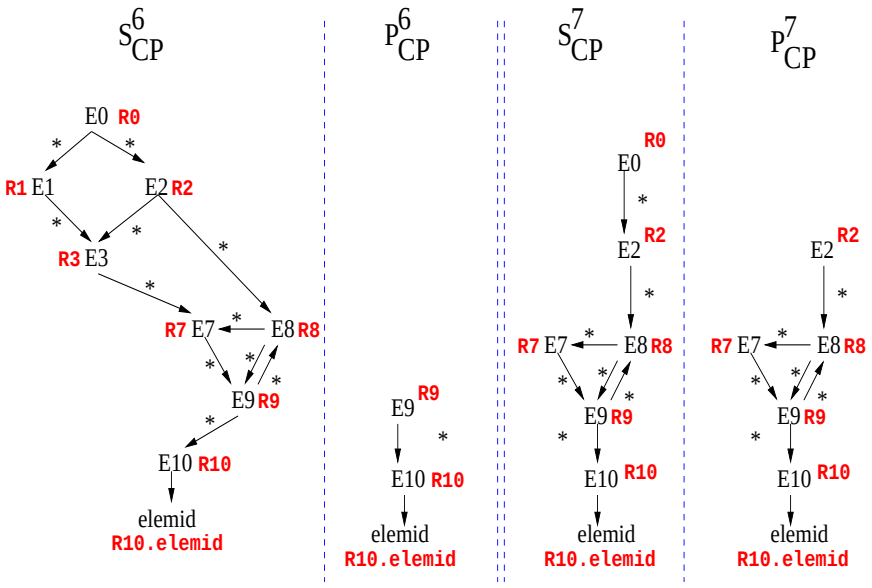

Figure 26: Examples to illustrate the mapping_aware algorithm

equivalence class in a fashion similar to the algorithm in Chapter 3.2.2.

THEOREM **3** *Given an XML-to-Relational mapping $\mathcal{T}$, a path expression query P and the guarantee that the "lossless from XML" constraint is satisfied, the mapping_aware algorithm outputs a correct equivalent SQL query.*

Proof: The proof is similar to the proof of Theorem 2 and relies on the fact that the pruning stage only removes the prefix of a path when it is safe to do so. In particular, the algorithm makes sure that for the current suffix path under consideration there is no overlap in results with schema paths not matching the query or with paths in PathSet that are not combinable. Combining this with the fact that the "lossless from XML" constraint holds, we see that the pruning stage does not alter the results. Then the correctness of the mapping_aware algorithm follows from the correctness of the original XML_to_SQL algorithm (Theorem 1 from Chapter 3). □

We now explain some example query evaluations over the XML schema S_3 in Figure 25 (same schema we saw earlier in Figure 13 in Chapter 3).

Consider query Q_4 =/E0//E6/E10/elemid. After the PathId stage, we obtain the cross product mapping S^4_{CP} shown in Figure 25. If we directly translate this into SQL, we will get a complex query involving two *with* clauses, corresponding to elements `E3` and `E6`. On the other hand, by using the pruning algorithm in Figure 24, we obtain the pruned mapping P^4_{CP}. The corresponding SQL query is fairly simple. Let us look at how the algorithm worked in this case. We start with the single path p = `elemid` in PathSet. Since, path p_1 =<E0,E2,E3,E7,E9,E10,elemid> does not appear in the query result and is in conflict with p, we increment p by one level. The same conflict persists and so we go up one more level. Now, p =<E6,E10,elemid> and p_1 is no longer in conflict with p (they have different relation sequences now). So, we have completed steps 2-8 of the algorithm. Now since there is only one path left, steps 9-14 can be skipped and we return PathSet as the result.

Let us now consider Q_5 =/E0//E1//E6/E10/elemid. The result of the PathId stage, S^5_{CP}, is shown in the figure. Notice that there are two satisfying paths. Let us see what happens in the pruning stage. We start with the single path p = `elemid` in PathSet. Just like the previous query, we need to go two levels higher and p =<E6,E10,elemid>. Notice how while p_1 is no longer in conflict with p, the path p_2 =<E0,E2,E3,E4,E6,E10,elemid> is in conflict with p. Also, p_2 is not in the query result. So, we need to increment p by one more level. Element E6 has two parent nodes and so we go up along both paths. In the process, a single graph path p gets split into two graph paths p' and p'', rooted at nodes E4 and E5 respectively. p' is still in conflict with p_2, while p'' is in conflict with p_3 =<E0,E2,E3,E5,E6,E10,elemid>. So, we increment the paths, p' and p'', by one level each. The two paths are merged into one (say p rooted at E3), but the conflict with p_2

and p_3 persist. Finally, when we increment p by one more level to get the graph P^5_{CP} in the figure. This graph path is safe from both p_2 and p_3 (join with relation R1, instead of relation R2). Also, there are no other conflicting paths. Hence, P^5_{CP} is the result of the pruning stage.

We now consider some example queries that match recursive parts of the schema. Consider query $Q_6 =$ /E0//E9//E10/elemid. The set of matching paths S^6_{CP} is shown in Figure 26. The result of the pruning stage P^6_{CP} is shown in the figure. Notice how the join between relations $R9$ and $R10$ suffices to make the path safe from all the paths not satisfying the query.

We consider query $Q_7 =$ /E0/E2/E8//E10/elemid to illustrate what happens when we need to go up the schema and enter a recursive component. The result of PathId S^7_{CP} is shown in the figure. Notice how the edge <E3,E7> does not match the query. So, query results corresponding to all the paths that pass through this edge need to be avoided in the final SQL query. The pruned schema P^7_{CP} is shown in the figure. Notice how we have to go up the recursive component during step 6 of the algorithm. We start with `elemid` and go up two levels till E9. The next time we have to increment a level, we enter the recursive component (comprising of nodes E7,E8 and E9). Here, we use a simple algorithm to go up the schema: include the entire recursive component in one step. Finally, when we add the element E2, we can stop. In this particular case, we managed to save a single join operation with relation R0.

4.3.3 Schema-Oblivious Storage

The examples in the preceding sections may give the erroneous assumption that the optimizations discussed in this chapter depend somehow upon the relational schema into

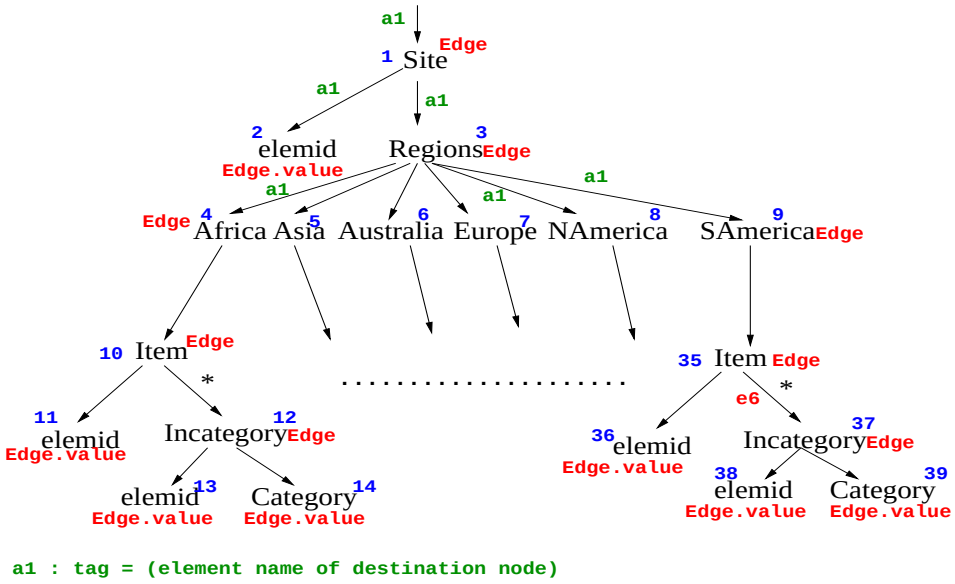

Figure 27: XMark schema mapped to the Edge relation

which the documents are shredded reflecting a good deal of the XML schema for the document being shredded. In this section we show that this is not true — in fact, the "lossless from XML" constraint is useful even when the relational schema is generic and reflects nothing of the XML schema (a scenario we term "schema-oblivous storage.")

In schema-oblivious XML storage, the relational schema is fixed independent of the XML schema. This option may be chosen either because the XML schema may is unavailable during data load time or due to the fact that the XML schema changes frequently.

The Edge approach [FK99] is one example of schema-oblivious storage. Here, the input XML document is viewed as a graph and each edge of the graph is represented as a tuple in a single table. This *Edge* relation has 5 columns, ID, PARENTID, TAG, ORDER AND VALUE.

During query translation time, let us assume that an XML schema is either given or has been inferred from the XML documents loaded into the system. For example, a sample XML-to-Relational mapping is shown in Figure 27. All the nodes are annotated

with the same relation name `Edge`. All the edges have similar annotations. For example, an edge $e = u \rightarrow v$ has the annotation "tag = element_name(v)". Notice each edge traversal will translate into a join operation.

Since this input scenario satisfies the "lossless from XML" constraint, the query translation algorithms presented earlier in this chapter are applicable. For example, the query Q_8 =/Site//Item//Category will translate into the following two-way join query over the `Edge` relation.

```
select   E2.value
from     Edge E1, Edge E2
where    E1.tag = 'InCategory' and E2.tag = 'Category'
         and E1.id = E2.parentid
```

The above query was obtained by exploiting the "lossless from XML" constraint. In contrast, if we use just the XML schema information, we will identify the six matching paths and the equivalent SQL query will be the union of six queries, one corresponding to each matching schema path (similar to query SQ_1^1 in Section 4.1). Each of these queries will be a join between six copies of the Edge relation. If we do not use the XML schema information at all (like the algorithm proposed in [FK99]), the resulting SQL query will be a recursive SQL query (due to the // in the XML query).

The above example illustrates how the "lossless from XML" constraint can be used to generate efficient SQL queries in a wide spectrum of scenarios: ranging from schema-based to schema-oblivious techniques.

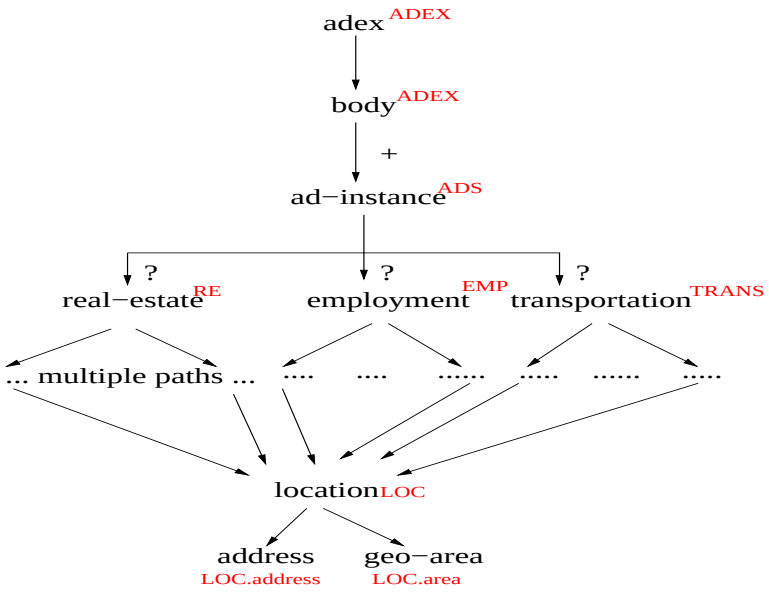

Figure 28: Part of ADEX XML schema

ADEX(adid, . . .)	Metadata for each advertisement
ADCONTACTS(personid, name, title, . . .)	Contact People appearing in ads
ADS(id, adid, instanceno, category)	Instance of a particular ad
RE(id, category, . . .)	Information about real-estate ads
TRANS(id, category, . . .)	Information about transportation ads
EMP(id, category, . . .)	Information about employment ads
LOC(id, address, zipcode, area)	Location information for all ads
JOBCAT(id, category,. . .)	Category info for employment ads
JOBREF(id, reference,. . .)	Reference info for employment ads

Table 3: Part of Relational Schema for ADEX dataset

SITE(id,...)	Information about the auction site
ITEM(id,siteid,continent,...)	Items available for auction
INCATEGORY(id, itemid, category,...)	List of categories for each item
CATEGORY(id,name,...)	Information about all categories
OPENAUCTION(id,itemid,...)	Information about currently active auctions
BIDDER(id, auctionid, personid,...)	Bidder information for open auctions

Table 4: Part of Relational Schema for XMark dataset

	Queries	Speedup (Cold buffer)	Speedup (Warm buffer)
A1	Get the number of open-house ads in the campus area	1.22	1.15
A2	Get the number of real-estate ads in the campus area	2.73	3.25
A3	Get the addresses of ads in the campus area	27.05	31.1
X1	Get the number of items in a particular category	2.69	5.56
X2	For a particular person, get categories of items for which (s)he made a bid	5.35	13.20

Table 5: Relative performance improvement obtained by the mapping_aware algorithm

4.4 Experimental Study

The examples in this chapter showed that while published algorithms often translate XML queries into fairly complex SQL queries, often there are equivalent queries that looks much simpler. The mapping_aware algorithm we presented in the previous sections achieve this goal. An important question to answer at this point is whether the associated performance gains are substantial. In order to demonstrate that this improvement can be sizable in practice, we performed an experimental study using two datasets: a synthetic ADEX dataset conforming to a standard advertisement schema [Ade] and a dataset from the XMark Benchmark [XMa]. Since all the prior techniques were applicable only for non-recursive XML schema, we only consider XML queries that correspond to non-recursive parts of the XML schema.

The ADEX dataset conforms to the standard DTD being developed by the Newspaper Association of America Classified Advertising Standards Task Force [Ade]. This standard is intended to pave the way for the aggregation of classified ads among publishers on the Internet, as well as to enhance the development of classified processing systems. Part of the ADEX schema is shown in Figure 28. We generated synthetic data conforming to the ADEX schema. This generated data consists of 100K advertisements and 200 publications, and is approximately 150 MB. The XMark Benchmark [XMa] schema contains information about an auction database and we used the standard 100 MB dataset defined

in the benchmark. A brief description of the relations in the relational schema for the ADEX and XMark datasets is given in Tables 3 and 4 respectively. For both scenarios, we built indexes on all columns that appeared in a query. We ran the experiments using the IBM DB2 database on a Linux workstation with an Intel 800 MHZ Pentium processor and 256 MB of main memory. The buffer pool was set to 32 MB.

We compare the execution times we measured for the queries in Table 5. The queries labeled Ai are on the advertisement dataset, while those labeled Xi are on the XMark dataset. For each XML query, we generated relational queries using prior published algorithms and used the best timing for comparison with our approach, the mapping_aware algorithm. The speedups obtained in execution times are given in the table. The XML queries and the SQL queries used in these experiments are given in Appendix A.

The relative improvement in performance ranges from 1.15 to 31. In general, by using the mapping information we do no worse than any of the prior strategies; so the relative performance is always greater than or equal to 1. We found that the actual performance improvement depends on two main factors: (i) the number of satisfying paths that can be merged together due to the fact that they have the same relation sequence and (ii) the length of the prefix that can be eliminated.

For example, the wild card in query A1 had two satisfying paths, while that in A2 and A3 had seven and twenty satisfying paths respectively. The response times show that as the number of satisfying paths for a wild card increases, the benefit obtained by our approach also increases considerably.

Similarly, queries X1 and X2 on the XMark dataset also had significant speedups ranging from a factor of 2.7 to a factor of 13.2. The maximum speedup was smaller in these cases relative to the ADEX dataset as the maximum number of satisfying paths for a wild card is six for the XMark schema.

To summarize, we see that using our mapping_aware translation algorithm, we get significant performance benefits. This improvement is markedly higher when the XML query has wild cards in it.

4.5 Summary

We looked at the XML-to-SQL query translation problem from a performance perspective. We showed how due to the mismatch between the XML and relational data models, the SQL queries produced in XML-to-SQL query translation are often unnecessarily complex, even for simple input XML queries. By using the simple fact that the "lossless from XML" constraint holds for the XML storage scenario, we presented an algorithm that uses the mapping information in an intelligent fashion to generate efficient SQL queries. The algorithm translates path expression queries over recursive XML-to-Relational mappings and the quality of the final SQL queries is significantly better than those output by prior query translation algorithms in many cases.

Chapter 5

Generating Efficient SQL Queries in the Publishing Scenario

As we discussed in Chapter 4, the SQL queries produced in XML-to-SQL query translation are often unnecessarily complex, even for simple input XML queries. In many cases, there is a much simpler equivalent SQL query. This observation motivated us to search for techniques that make use of readily available semantic information to improve the quality of the generated SQL. For the XML storage scenario, we showed in Chapter 4 how we can fully exploit the "lossless from XML" constraint to use the mapping information in an intelligent fashion and generate efficient SQL queries.

An obvious question is whether the same techniques can be extended to the XML Publishing scenario as well. Unfortunately, the answer is "no" as the relational data is no longer dependent on the XML view definition. Recall that in the publishing scenario, an XML view of pre-existing relational data is exported. As a result, the "lossless from XML" constraint is not guaranteed to hold. So, in general, we cannot guarantee that the optimizations in Chapter 4 are correct if applied directly to the XML Publishing scenario.

However, in some cases, we can perform similar optimizations in the publishing scenario if we are given information about what relational integrity constraints hold on the underlying relational data. We present an example scenario in Section 5.1 to illustrate

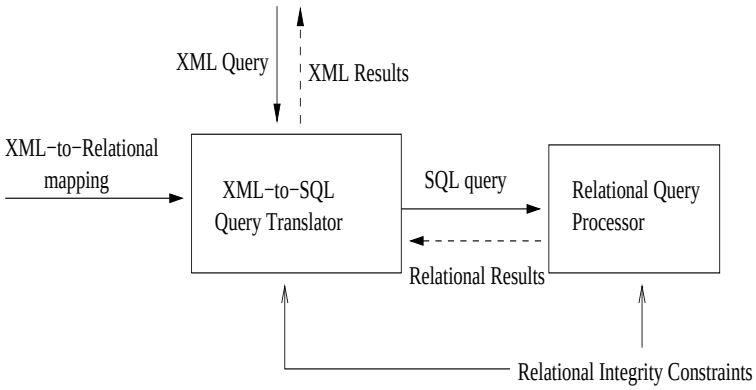

Figure 29: Stages in using an RDBMS to evaluate an XML query

this idea. By using the relational integrity constraints, it is possible to find simpler equivalent SQL queries during XML-to-SQL query translation. We follow this approach in this chapter.

To understand the alternatives for how we can do this, consider the different stages in the translation process as shown in Figure 29. Given an XML-to-Relational mapping, some relational integrity constraints, and an XML query, the XML-to-SQL query translator generates an equivalent SQL query and hands it over to the relational query processor. The relational query processor optimizes and executes the query, and returns the results to the query translator, which adds the appropriate XML tags to the results and returns them to the user. There are two important points to note here: (i) As the XML-to-Relational mapping and relational integrity constraints are valid across multiple query invocations, they are shown separately, and (ii) We have made no assumptions about whether the XML-to-SQL query translator is inside an RDBMS or in middleware. This is the reason for using the term Relational Query Processor instead of RDBMS for the box on the right.

There are two logical extremes in approaches toward obtaining efficient SQL queries for XML workloads. One could generate suboptimal SQL queries using a fairly naive

translation algorithm, and then optimize the resulting SQL queries (SQL Optimization); or one could use a more intelligent query translation algorithm and attempt to generate efficient SQL queries directly (Intelligent Query Translation).

In Section 5.3, we will show that if we take the SQL Optimization approach, then in order to obtain efficient SQL queries we have to solve the relational query minimization problem under bag semantics. The techniques for query minimization in the published literature rely on algorithms for query containment or query equivalence. Unfortunately, these problems become intractable in even simple scenarios, making the SQL Optimization approach impractical. In view of this problem, we need to find a way to generate good SQL queries that does not require the solution of these intractable problems during actual query translation.

In response to this goal, we propose that Intelligent Query Translation should be used instead of SQL Optimization, and propose a translation approach that relies upon three main ideas. First, we identify a class of tree XML-to-relational mappings called *bijective* mappings. Bijective mappings cover a large class of the mappings we have encountered in print, and they have the desirable property that they can be optimized using containment and equivalence algorithms under set semantics instead of multiset semantics. They subsume the class of "lossless from XML" constraint-preserving mappings that we saw in the XML storage scenario.

Second, we observe that for a given XML schema over a given relational schema, the SQL queries generated from XML queries are not arbitrary. That is, the XML-to-Relational mapping determines the class of SQL queries that are likely to be output by the XML-to-SQL query translation algorithm, which in turn fixes the class of queries that need to be minimized. Since the XML-to-Relational mapping and the underlying relational integrity constraints are independent of the query being optimized, we can

use them to precompute some useful information, and then use this information during the runtime query translation. This way, we can move the potentially expensive task of reasoning about integrity constraints to the precomputation phase, keeping the run time overhead small.

Third, the conjunctive queries produced by XML to SQL translation are mainly *chain* queries of the form

$$R(x_n) : -R_1(x_1, x_2), R_2(x_2, x_3), \ldots, R_{n-1}(x_{n-1}, x_n)$$

As we will show, in the XML to SQL translation domain, exploiting integrity constraints enables the minimization of such queries by removing a prefix of the relational predicates. We refer to this as *prefix elimination*. This turns out to be more tractable than general conjunctive query minimization. Notice that this is similar to the prefix-elimination optimization we performed in the mapping_aware algorithm in Chapter 4.

We show that by exploiting the above three ideas, the XML-to-SQL query translation problem can be solved in polynomial time for path expression queries over bijective tree mappings. Our proof works by presenting a query translation algorithm that solves the problem with the required efficiency. Our algorithm works correctly even over non-bijective mappings; it identifies the bijective portions of the mapping and performs more efficient query translation in those parts. This translation algorithm produces SQL queries that in many cases are far more efficient than those produced by previously published translation algorithms.

The rest of the chapter is organized as follows. In Section 5.1, we present an example scenario to illustrate the problems with published XML-to-Relational translation algorithms. Next we define the query translation problem in Section 5.2. Then, in Section 5.3, we present some of the known complexity results we bump into if we attempt

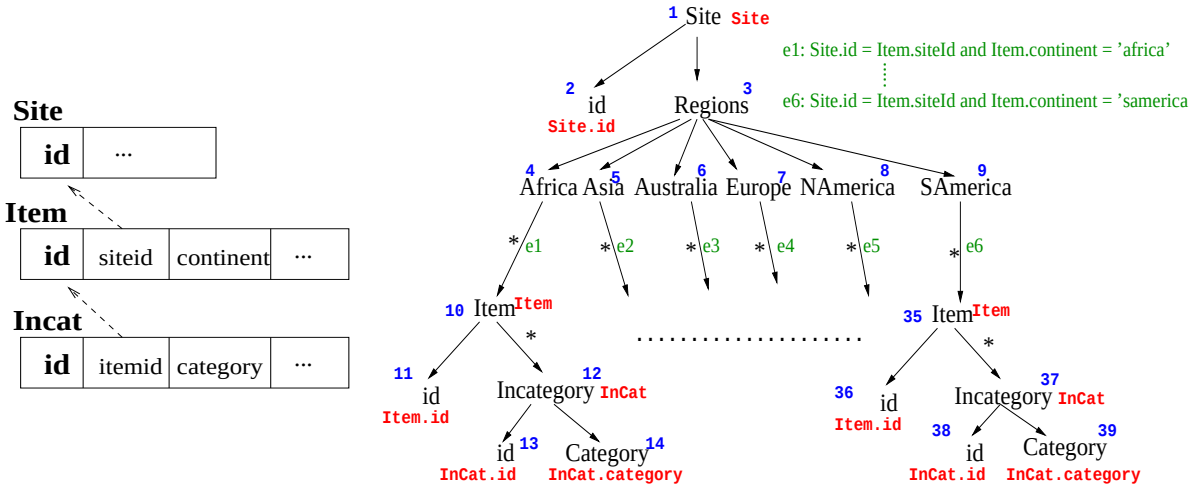

Figure 30: Sample relational schema and corresponding XML view

to minimize the SQL queries after generating them. We describe our strategy for more intelligent query translation in Section 5.4. A more formal description of the various components of this approach is presented in Section 5.5.

5.1 Motivation

In this section, we revisit the example discussed in Section 4.1 from an XML publishing viewpoint. We show how even simple XML queries can give rise to fairly complex SQL queries if we use published translation algorithms. On the other hand, we can identify simpler equivalent SQL queries by making use of the relational integrity constraints on the underlying relational data.

Part of a sample relational schema for an auction database is shown in Figure 30. The figure also shows one way of exporting this data as XML. The example XML schema is part of the XMark benchmark [XMa] schema. Each node in the XML schema is annotated with a table name, to indicate the relational table that corresponds to the element represented by the node. Each leaf node has a column name next to it, which

indicates the column in which the value of corresponding element is stored.

Consider the evaluation of the following query Q_1, which returns all the item categories: //Item/InCategory/Category.

As described in Chapter 4.1, previously published algorithms will result in the following SQL query SQ_1.

select C.category
from Site S, Item I, InCat C
where S.id = I.siteid and I.id = C.itemid and I.continent='africa'
union all ... (6 queries one for each continent except Antarctica)

Suppose furthermore that the underlying relational schema has the following domain integrity constraint (in addition to the key and foreign key constraints shown in the figure): the column `Item.continent` has only six potential values {asia, africa, australia, europe, namerica, samerica}.

For the above query, we have found through experimentation that the optimizers in current relational systems will use foreign key constraints to eliminate some redundant joins. For instance, the join between Site and Item can be removed. Though the join between Item and InCat is a key-foreign key join, it cannot be removed due to the condition on Item.continent. Thus the query as rewritten by a relational optimizer becomes the new query SQ_1^1:

select C.category
from Item I, InCat C
where I.id = C.itemid and I.continent = 'africa'
union all ... (6 queries one for each continent except Antarctica)

We have seen that existing commercial RDBMS optimizers convert SQ_1 to SQ_1^1. A reasonable question is whether the XML to SQL translation routines proposed in SilkRoute [FMS01] and Xperanto [SSB+00] do better. We find that by merging common subexpressions, they generate a better initial query than SQ_1. But, interestingly, if you feed the queries that they generate to a relational optimizer, the resulting final query is once again SQ_1^1. So, no matter whether we use a naive XML to relational translation, or these more sophisticated translation schemes, in the end the RDBMS will evaluate SQ_1^1.

As we pointed out in Chapter 4.1, the XML query Q_1 has no redundant parts in it, and so XML query minimization [AYCLS01, FFM03, Ram02] will not help in this case.

Unfortunately, SQ_1^1 is far from optimal, since all of these queries are equivalent to the even simpler OQ_1 given below:

```
select  category
from    InCat C
```

The equivalence between the queries SQ_1, SQ_1^1 and OQ_1 holds under the key, foreign key and domain constraints mentioned above. Notice how we are able to replace a query SQ_1^1, which was the union of six queries each with a join, by a single scan query OQ_1.

Notice how we were able to simplify the query SQ_1 to OQ_1 by using the relational integrity constraints. This is similar to what we achieved in Chapter 4.1 for the XML storage scenario, though by a different mechanism. Notice that there is one important difference between the XML storage and publishing scenarios: the lack of the "lossless from XML" guarantee in the latter case. For example, suppose the domain constraint on the `Item.continent` column allowed a seventh value as well (Antarctica). In this case, items corresponding to this continent are not present in the XML view and so OQ_1 is

not a correct translation. The minimal equivalent SQL query is OQ_1^1 given below:

```
select   C.category
from     Item I, InCat C
where    I.id = C.itemid and
         I.continent IN {'africa','asia','australia','europe','namerica','samerica'}
```

In the rest of the chapter, we look at two different ways of attempting to automatically generate these better queries: SQL Optimization, and Intelligent Query Translation. In the former approach, SQL queries are generated in a straightforward fashion and then optimized using the relational integrity constraints. In the latter approach, we use the constraint information during the XML-to-SQL query translation process itself.

5.2 Problem Definition

In this section, we present a formal description of the XML-to-SQL query translation problem for the XML publishing scenario.

Class of XML views

For concreteness, we need to provide some mechanism for representing how an XML schema is mapped to a relational schema. In this chapter, we use the simple approach of defining an XML view with annotations on the XML schema nodes and edges. A non-leaf node is annotated with a relation name, while a leaf node is annotated with the name of a relational column. Each edge $e = (u \rightarrow v)$ is annotated with a conjunctive query, where the relations allowed in the query are the relational annotations of nodes on the path from the root of the graph to node v. A *simple* XML view is one in which each of the edge annotations involves at most one join condition.

We illustrate this approach to defining views with an example. Consider the relational schema and the corresponding XML view definition in Figure 30. Consider a top-down traversal of this schema, which illustrates how an XML document can be constructed from underlying relational data. The Site element is the root of the document and it has an id child whose value is the value of Site.id attribute. A Regions child element is created within the Site element and six subelements are created within Regions, one for each continent. Within each continent element, the information about the items in that continent are exported. For example, consider the element Africa. The annotation on the outgoing edge $(4, 10)$ indicates that for each tuple in the Item relation corresponding to this continent and satisfying the join condition, an Item subelement is created. For each such item, its id is exported as an id child element and the categories to which the item belongs is represented as incategory subelements. The annotation on edge $(10, 12)$ is a join condition Item.id = InCategory.itemId (not shown in figure). This view definition is an example of a *simple* view definition as each edge annotation has at most one join condition.

To formally describe these schema annotations, consider the following way of defining XML views over relational databases. In this method, proposed in [SSB+00], a default view is defined over the relational database and the XML view is defined as a query (an XQuery query in [SSB+00]) over the default view. The default view maps each relation R to an element with name R. Each row of the relation R is mapped to a row element under element R. This row element has one child element for each column of relation R. The XQuery query (over the default view) corresponding to the annotations in Figure 30 is shown in Figure 31. Since the view might correspond to a number of XML documents, we have added an extra root element called AuctionDocuments.

The following edge annotations are allowed in a simple XML view definition. Here R.C

```
1.  create view auction_data as (
2.  <AuctionDocuments>
3.     for $site in view("default")/SITE/row
4.     return
5.        <Site >
6.           <id>$site/id < /id>
7.           <Regions>
8.              <Africa>
9.                 for $item in view("default")/ITEM/row
10.                where $site/id = $item/siteid and $item/continent = 'africa'
11.                return
12.                   <Item >
13.                      <id>$item/id < /id>
14.                      for $incat in view("default")/InCat/row
15.                      where $site/id = $incat/itemid
16.                      return
17.                         <Incategory>
18.                            <id> $incat/id < /id>
19.                            <Category> $incat/category < /Category>
                            < /Incategory>
20.                   < /Item>
21.             < /Africa>
22.             <Asia>
⋮                    ⋮
    </AuctionDocuments> )
```

Figure 31: View Definition expressed as an XQuery query over the default view

represents column C of relation R.

1. $<R>$: an initial annotation.

2. $<R.C$ op value$>$: a selection condition on column $R.C$.

3. $<R_p.C_p, R_c.C_c>$: an equijoin between the two columns.

4. Null annotation (to allow dummy elements).

The annotation on an edge e is called *annot*(e). The relations in the edge annotations are not allowed to be arbitrary. We associate a relation $Rel(n)$ with every schema node

n in a top down fashion as follows. Let the (unique) in-coming edge into n be $e_{p,n}$. If $annot(e_{p,n})$ is

1. an initialization condition $<R>$,then $Rel(n) = R$.

2. a selection condition involving the column $R.C$, then $Rel(n) = R$. Here, we must have that $Rel(p) = R$.

3. a join condition of the form $<R_p.C_p, R_c.C_c>$, then $Rel(n) = R_c$. Here, we must have that $Rel(p') = R_p$ for some ancestor p' of node n.

4. a null annotation, then $Rel(n) = Rel(p)$.

Node annotations are allowed only on leaf nodes. For a node n, $annot(n) = R.C$ is a column in the relation $Rel(n) = R$. This annotation indicates that the values in the column $R.C$ are exported (in the XML view) as values of elements corresponding to the schema node n. We note here that we can extend the above class of mappings to allow extra features like, for instance, internal nodes having mixed content. Our techniques extend easily to cover such extensions.

Class of XML queries

In this chapter we focus on a simple but useful class of queries: simple path expressions. A simple path expression can be denoted as "$s_1\ l_1\ s_2\ l_2 \ldots\ s_k\ l_k$," where each of the l_i is a tag name and each of the s_i is either / (denoting a parent-child traversal) or // (denoting an ancestor-descendant traversal).

Class of SQL queries

For a path expression query over a tree XML view, the equivalent relational query output by published translation algorithms can be viewed as the union of several conjunctive queries. So, we consider this class of queries with a simple extension: a disjunction of selection conditions is allowed for each conjunctive query.

Semantics of translation

For concreteness, we need to define what is meant by a translation of an XML query to a SQL query. That is, we need to define when a SQL query is considered a correct translation for a given path expression query Q under a mapping $\mathcal{T}$. Our approach to defining these semantics is to present a straightforward translation algorithm that returns the query *baseline*(Q). Any SQL query SQ that is equivalent to *baseline*(Q) under the given relational integrity constraints is a correct SQL translation for the XML query Q.

For a leaf node n in the schema, we define the root-to-leaf query $rtol(n)$ as the SQL query obtained by (conjunctively) combining the annotations on the edges of the root-to-leaf path of n and projecting the annotation of node n. For example, $rtol(14)$ is the query

```
select   C.category
from     Site S, Item I, InCat C
where    S.id = I.siteid and I.id = C.itemid and I.continent='africa'
```

Given a tree XML-to-Relational mapping $\mathcal{T}$ and a simple path expression query Q, let $S = \{n_1, n_2, \ldots, n_k\}$ denote the set of nodes in $\mathcal{T}$ that match the query Q. Then a *baseline* query translation algorithm is to return the SQL query $\bigcup_{n \in S} rtol(n)$. Let *baseline*(Q) denote this query.

XML-to-SQL Query Translation Problem:

Finally, we are able to define what we mean by the XML-to-SQL translation problem: Given an XML-to-Relational mapping $\mathcal{T}$, a simple path expression query Q and integrity constraints on the underlying relational schema, find the equivalent SQL query with minimum cost.

The above definition is precise modulo the interpretation of the phrase "minimum cost." Different problems will result with different cost metrics. A reasonable cost metric is the traditional metric for conjunctive query minimization — that is, the cost of a query is the number of relational conjuncts. Let us denote this metric RelCount.

5.3 The SQL Optimization approach

The scenario we presented in Section 5.1 showed that query minimization is a core issue in generating efficient SQL queries for XML workloads. For the class of path expression queries over a tree XML-to-Relational mapping, recall that *baseline*(Q) is a union of conjunctive queries. Hence, we need to minimize a union of conjunctive queries under multiset semantics in the presence of relational integrity constraints. In this section, we first discuss prior work on relational query minimization and then discuss the impact on the SQL Optimization approach.

5.3.1 Previous work on Relational Query Minimization

Most, if not all, techniques in the published literature for minimizing relational queries are based on algorithms for query containment or query equivalence. We next present some known results about the complexity of these problems.

- The containment, equivalence and minimization problems for conjunctive queries under set semantics are NP-complete [ASU79, CM77].
- The containment problem for conjunctive queries under multiset semantics is π_2^P-hard [CV93].
- The equivalence problem for conjunctive queries under multiset semantics is same as graph isomorphism [CV93].
- The containment and equivalence problems for monotonic relational expressions under set semantics is π_2^P-complete [SY80].
- The containment problem for union of conjunctive queries is undecidable under multiset semantics [IR95].

There has also been a lot of work on the use of constraints in query optimization of relational queries [DPT99, JK82, ZO97]. In [JK82], the query containment problem under functional dependencies and inclusion dependencies is studied. In [SO87], a scheme for utilizing semantic integrity constraints in query optimization, using a graph theoretic approach, is presented. In [ZO93], a necessary and sufficient condition for the IC-RFT problem (does a conjunctive query always produce an empty result under a given set of implication constraints) is presented and in [ZO97] the results are extended when referential constraints are also allowed. Polynomial equivalence to other problems like the query containment problem are also proved.

More recently, the chase and backchase algorithm was introduced in [DPT99] motivated by logical redundancy and physical independence in mediator-like components.

This approach brings together use of indexes, use of materialized views, semantic optimization and join/scan minimization and allows non-trivial use of indexes and materialized views through the use of semantic constraints. In [DT01], the authors present a generalization of the classical chase algorithm for embedded dependencies [BV84] to a richer class of constraints known as Disjunctive Embedded Dependencies (DEDs).

5.3.2 Impact on the SQL Optimization approach

While a lot of research has been done on relational query optimization in the presence of constraints, there are some mismatches with what we need in the XML-to-SQL query translation scenario.

- Most of the prior work is on reasoning under set semantics. On the other hand, we need to optimize relational queries under multi-set semantics. We are not aware of any published algorithm for minimizing union of conjunctive queries under multi-set semantics (both in the absence and presence of integrity constraints).

- Even under set semantics, the running time of these algorithms are exponential in the size of the input (relational schema, constraints and query). Incurring this overhead on a per-query basis may be expensive in practice.

- The class of constraints handled by different approaches vary considerably and no single technique dominates the others.

By a simple reduction, we have the following result.

PROPOSITION 4 *Solving the XML-to-SQL Query Translation problem using the SQL Optimization approach for a tree XML view under the metric RelCount is at least as hard as minimizing a union of conjunctive queries under multiset semantics.*

Proof: Let $Q = \bigcup_{i=1}^{k} Q_i$ be the union of conjunctive queries that needs to be minimized. We first consider the case when there is a single distinguished variable in Q. Consider the query $Q' = \bigcup_{i=1}^{k} R(x)Q_i$, where R is a new relation and x is a new variable, neither of which appear in the query Q. Minimizing the query Q' is the same as minimizing the original query Q. We construct an instance of the XML-to-SQL query translation problem that mirrors minimizing Q'.

The XML schema graph has $k+1$ nodes, a root node n with k children (say $n_1, \ldots, n_k$). The root node n has label l_1 and is annotated with relation name R. All the other nodes have label l_2. Each edge $<n, n_i>$ is annotated with the query Q_i. The leaf node n_i is annotated with the column name corresponding to the distinguished variable in Q_i. For the input XML query $/l_1/l_2$, the *baseline* SQL query is Q'. The resulting XML-to-SQL query translation problem is equivalent to minimizing Q'.

In general, if the query Q had m distinguished variables, we modify the reduction by adding m children to each node n_i. All the new nodes are given the label l_3. Then the query $/l_1/l_2/l_3$ completes the reduction. □

5.4 Intelligent Query Translation

In this section, we present our approach to generating SQL queries that are often more efficient than those generated by existing translation algorithms. We are able to do so by focusing on a tractable yet important subpart of the problem space. This section is somewhat complex; we begin with an overview of our approach and then explain the main components of our approach. A more formal description is presented in the following section (Section 5.5).

5.4.1 Outline of our approach

As we saw in the previous section, the SQL Optimization approach has three main problems: (i) lack of techniques for query minimization under multi-set semantics, (ii) high overhead for reasoning using constraints even under set semantics and (iii) variety of techniques for different class of constraints. In the Intelligent Query Translation approach, we circumvent each of these problems in the following fashion.

While reasoning about query minimization under multi-set semantics can be a lot different from reasoning under set semantics, there are scenarios where the two notions are similar. In our approach, we identify a class of views that, informally speaking, have the property that the target relational data is exported *exactly* once in the XML view. We refer to such views as *bijective*, and describe this concept in more detail in Section 5.4.2. Such mappings have the desirable property that they can be optimized using containment and equivalence algorithms under set semantics instead of multiset semantics. In our approach, we identify parts of the mapping that are bijective and apply our optimizations to those parts.

In order to address problems (ii) and (iii), we adopt the following strategy. By observing that the XML-to-Relational mapping and the underlying relational integrity constraints remain constant across multiple query invocations, we compute some summary information in a precomputation phase. In this precomputation phase, we make use of an algorithm for reasoning about conjunctive query containment under set semantics (say $\mathcal{A}$). Then, when we need to translate an XML query into SQL, we use this summary information in the run-time query translation phase. This way, the potentially expensive part of reasoning using integrity constraints is moved to a (offline) phase and the run-time overhead is kept small. In addition, we can make use of different algorithms

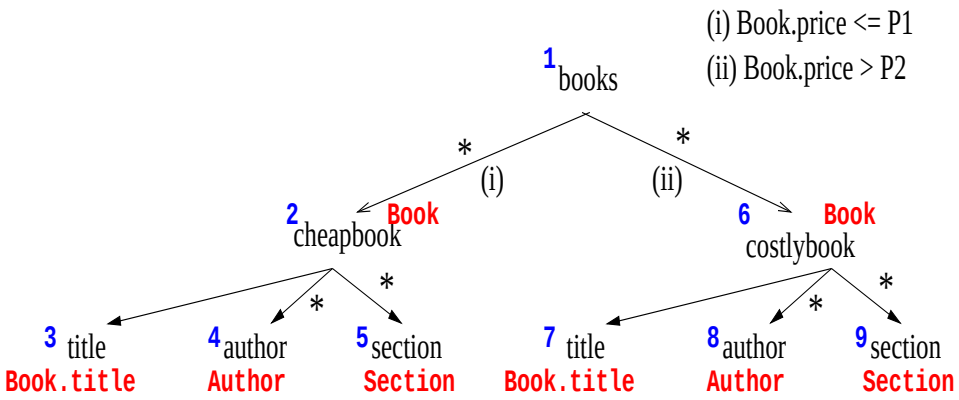

Figure 32: Example view to illustrate (non)bijective mappings

for $\mathcal{A}$ that work for varying classes of relational integrity constraints. This is especially useful as we can choose algorithm $\mathcal{A}$ based on the class of relational integrity constraints that are applicable for the current relational schema.

Since, we are going to use some summary information during the run-time query translation process, we need to relax the optimality metric that we hope to achieve. As we have seen in Section 5.1, optimizing SQL generated by XML to SQL translation frequently involves eliminating unnecessary prefixes in the SQL queries. Motivated by this observation, we define a different notion of minimality for generated SQL queries — one where we would like to maximize the length of the prefix eliminated for each matching path in the schema. We define this metric, *PrefixMetric* in Section 5.4.3.

Using the above techniques, we developed a constraint_aware approach to efficiently translate path expression queries into SQL. We describe the main components of our approach informally in the following subsections. A more formal description of our approach is presented in Section 5.5.

5.4.2 Bijective mappings

Consider the XML schema shown in Figure 32, which represents information about a collection of books. The XML view has created a simple hierarchy, partitioning the books into cheap and costly books by the relationship of their prices to two constants P_1 and P_2.

Let us now consider three possible scenarios: $P_1 = P_2$, $P_1 < P_2$ and $P_1 > P_2$. If $P_1 = P_2$, then the XML view has information about all the books exactly once, while if $P_1 < P_2$ the XML view has information about only certain books. On the other hand, when $P_1 > P_2$, the XML view has information about the books in the price range $\{P_2 - P_1\}$ twice.

The scenario when $P_1 = P_2$ corresponds to an interesting and common class of mappings, one in which there is a one-to-one correspondence between the XML view data and the underlying relational data. We refer to this class of mappings as *bijective*. These mappings have the property that the query results of two root-to-leaf path queries do not have any common results, so the corresponding SQL queries can be merged without worrying about preserving counts of duplicates.

For example, the *rtol* queries for nodes 3 and 7 returning the titles of cheapbooks and costlybooks will not have any common results when the mapping is bijective. This simple observation makes the query minimization process a lot simpler as we can use algorithms for query minimization under set semantics instead of multiset semantics.

Notice that whether an XML view definition is *bijective* or not is a property of the view, and that one cannot determine if an XML view definition is bijective by simply examining the relational schema without the mapping. So, while one can easily use this information during the query translation process (where we know about the XML view),

in order to perform similar optimizations after the SQL query has been generated, the appropriate module (be it the relational optimizer or some other module) needs to know about properties of the XML view. This means that if existing relational optimizers are to be extended to handle optimizations based upon bijective views, they need to be extended to understand XML views, which is not very attractive.

5.4.3 Prefix Elimination Optimality

We define the cost metric *PrefixMetric*$(SQ, \mathcal{T})$ to be the number of nodes in the XML-to-Relational mapping $\mathcal{T}$ that correspond to the SQL query SQ. For example, consider the query SQ_1 in Section 5.1. The fragment of this query identifying items in Africa corresponds to the sequence of nodes $<1, 3, 4, 10, 12, 14>$, and so the cost is six. Since there are six such fragments in SQ_1, the total cost *PrefixMetric*$(SQ_1, \mathcal{T})$ is 36. Similarly, the cost for each fragment of query SQ_1^1 is four and the total cost *PrefixMetric*$(SQ_1^1, \mathcal{T})$ is 24. For query OQ_1, the total cost *PrefixMetric*$(OQ_1, \mathcal{T})$ is six.

By definition, the cost of any SQL query that does not correspond to a path in the mapping is undefined.

Notice that the definition of the *PrefixMetric* metric restricts the class of equivalent SQL queries considered. For example, we are only interested in finding equivalent queries that are in some sense "syntactically" contained in some conjunctive query fragment in *baseline*(Q). While this misses opportunities to find equivalent queries that involve materialized views or cached query results or eliminating intermediate relations in the conjunctive query, it is still general enough to cover a large number of interesting scenarios.

5.4.4 The query translation algorithm

In this section, we briefly explain the main components of our query translation algorithm using examples. The algorithm has two parts: an (offline) precomputation phase, in which summary information is computed; and a run time phase when the actual query translation occurs.

Precomputation Phase

Here, we make use of the fact that the XML-to-Relational mapping and the relational integrity constraints are valid across multiple queries and use them to precompute some summary information. The information that we precompute is related to properties of the root-to-leaf queries we discussed in connection with the semantics of translation in Section 5.2.

For a given node in the XML schema, it may be possible to eliminate a prefix of its corresponding root to leaf query. The actual prefix that can be eliminated for a leaf node varies depending on the subset of schema nodes selected by the query. We define the notion of **L**east **D**istinguishing **A**ncestors (LDAs) to capture this. For each pair of leaf nodes (u, v), we compute $\text{LDA}(u, v) = w$. Intuitively, w is the lowest ancestor of u such that if node u matches a given XML query, it is sufficient to issue the query from $w - u$ (instead of the root to leaf query for u) without returning any results corresponding to node v. In order to create the query for a node u, it suffices to pick the highest ancestor among $\text{LDA}(u, v)$ over all leaf nodes v not matching the query.

For example, for the schema in Figure 30, $\text{LDA}(14, 39) = 4$ and $\text{LDA}(39, 14) = 9$. In other words, if node 14 matches a query and node 39 does not, then it suffices to issue the query corresponding to the path $\{4, 10, 12, 14\}$ in order to return the results

corresponding to node 14. This query is shown below.

```
select IC.category
from Item I, Incat IC
where I.id = IC.itemid and I.continent = 'africa'
```

In our precomputation phase, for every pair of non-leaf nodes u, v that have the same annotation, we compute LDA(u, v). In addition, we identify the parts of the XML view definition that are bijective. In our running example, the entire XML view is bijective.

Run-time Query Translation

We use the following query on the mapping schema in Figure 30 to illustrate the translation algorithm.

Q: //Item/InCategory/Category

We first execute Q on the schema graph and identify the satisfying nodes: $S = \{14, 19, 24, 29, 34, 39\}$. For each node $n \in S$, issuing $rtol(n)$ is a correct translation. Our goal is to find the smallest suffix of each such query. Consider the leaf node $n_1 = 14$. We need to identify the lowest ancestor a_1 of n_1 such that it suffices to output the query for the path $<a_1, \ldots, n_1>$. In order to find a_1, we look at the other nodes with the same annotation, namely $C = \{19, 24, 29, 34, 39\}$ and compute $LDA(14, x), \forall x \in (C - S)$. The highest node among these corresponds to a_1. In this particular case, $(C - S)$ is empty, so we do not have to look at the LDA values. As a result, for node 14 it suffices to issue the scan query corresponding to the leaf node. We obtain a similar scan query for the other five schema nodes in S. Since the six scan queries are on the same relation, we merge them and issue a single query OQ given below:

```
select C.eid from InCat C
```

On the other hand, using existing algorithms we would have obtained a relational query SQ that is the union of six queries, each with two joins (similar to the example query SQ_1 in Section 5.1).

Analysis

In this section, we show that the constraint_aware algorithm always outputs a correct equivalent SQL query. In addition, for bijective tree XML views, it outputs the optimal SQL query under metric *PrefixMetric*. We give the proofs of these theorems in Section 5.5.4 after formally describing the complete algorithm.

THEOREM **4** *Given a tree XML-to-Relational mapping $\mathcal{T}$ along with the integrity constraints that hold on the underlying relational schema, and a path expression query P, the constraint_aware algorithm outputs a correct equivalent SQL query in polynomial time.*

We would like to point out that our algorithm performs XML-to-SQL query translation correctly even when part of the mapping is not bijective or when the conjunctive query containment algorithm $\mathcal{A}$ is sound but not complete for the class of relational constraints that are applicable. Note that the running time of algorithm $\mathcal{A}$ does not impact the complexity of the translation algorithm, since $\mathcal{A}$ is run once as a precomputation step, not on a per-query basis during translation.

Let Q_1 be a conjunctive query and Q_2 be a union of conjunctive queries. Let UQC denote the problem: is $Q_1 \subseteq Q_2$ under set semantics? and DUP denote the problem: Are the results of Q_1 duplicate-free?

Suppose $\mathcal{C}$ is the class of integrity constraints that hold on the relational schema and $\mathcal{A}$ and $\mathcal{A}'$ are sound and complete algorithms for the UQC and DUP problems over this class of constraints. Examples of such algorithms and a description of the corresponding

class of integrity constraints can be found in [DPT99, ZO97]. See Section 5.5.2 for a summary of the techniques in [DPT99] and how we use them in our approach. In such cases, our algorithm actually outputs the optimal query under metric *PrefixMetric*.

THEOREM **5** *Given sound and complete algorithms $\mathcal{A}$ and $\mathcal{A}'$ for the UQC and DUP problems over the class $\mathcal{C}$, the XML-to-SQL Query Translation problem for a bijective tree XML view under metric* PrefixMetric *can be solved in polynomial time.*

5.5 The constraint_aware approach

In this section, we formally describe the various components of our approach. We start by describing some terminology used in the formalization followed by the actual description of the two main components: precomputation phase and run-time query translation phase.

5.5.1 Terminology

Most of the properties we talk about address leaf nodes in the schema that are annotated with the same relational column. We define *nodes(R.C)* to be the set of leaf nodes annotated with $R.C$. We call two leaf nodes *column-compatible* if they are annotated with the same relational column. We refer to the annotation of node n as $annot(n)$.

Recall that we defined $rtol(n)$ to be the root-to-leaf query for a node n. We generalize this notion to an arbitrary sequence of nodes as follows. A *node sequence* $NS =< n_1, n_2, \ldots, n_k >$ is a sequence of nodes in the schema graph that corresponds to a path starting from the node $n_1 = NS.first$ and terminating in the leaf node $n_k = NS.last$. The relational query Query(NS) is obtained by combining the conditions on the edges of the sequence and projecting $annot(n_k)$. The relational query keyQuery(NS) is the

same as Query(NS), except that the key column(s) of $Rel(n_k)$ is (are) also projected. Query(NS) and keyQuery(NS) are always conjunctive queries. Just like Query(NS) corresponds to $rtol(n)$, we refer to keyQuery(NS) as $keyrtol(n)$.

Let RelSeq(NS) denote the sequence of relations joined in Query(NS), in a bottom-up order. For example, for NS =<1, 3, 4, 10, 12>, RelSeq(NS) = <Site,Item,InCat>.

Two node sequences NS_1 and NS_2 are said to be *combinable* if the corresponding relation sequences $RelSeq(NS_1)$ and $RelSeq(NS_2)$ are the same, the join conditions are on the same set of columns for each pair of relations, and $NS_1.last$ and $NS_2.last$ are column-compatible. In other words, the two relation sequences are identical modulo the selection conditions.

5.5.2 Precomputation Phase

Recall that, in the precomputation phase we identify the parts of the XML-to-Relational mapping that are bijective and also compute LDA information. We formally define these two notions in the next two subsections and then describe how we compute this information.

Bijective column mappings

For a relational column $R.C$, let KeyProject(R.C) denote the query "select R.key, R.C from R" and NodeKeyProject(R.C) denote the query $\bigcup_{n \in nodes(R.C)}$ keyrtol(n). Here, $R.key$ denotes the key column(s) of R. We will make use of the following definitions:

DEFINITION **1** *For a relational column R.C,*

- *If $KeyProject(R.C) \subseteq NodeKeyProject(R.C)$, then R.C is At-least-once mapped*
- *If $KeyProject(R.C) \supseteq NodeKeyProject(R.C)$, then R.C is At-most-once mapped*

- *If* $KeyProject(R.C) = NodeKeyProject(R.C)$*, then* $R.C$ *is bijectively mapped*

In the preceding definition, the containment operations are under multi-set semantics.

Informally, if all the values in the column $R.C$ appear in the XML view "exactly once", then the relational column is bijectively mapped. In order to check this under multi-set semantics, we use the key field(s) of the relation R.

An XML-to-Relational mapping $\mathcal{T}$ is bijectively mapped if each of the relational columns annotating some leaf node in $\mathcal{T}$ is bijectively mapped.

Notice how the definition of when a relational column is bijectively mapped corresponds to the properties P1 and P2 that hold when the "lossless from XML" constraint is satisfied (Chapter 3.1.2). This implies that all mappings that satisfy the "lossless from XML" constraint are bijectively mapped.

Lowest Distinguishing Ancestor

Let u and v be two column-compatible leaf nodes in the schema. Let node sequence $NS =< n_1, n_2, \ldots, n_k >$, where $n_1 = \text{root}(\mathcal{T})$ and $n_k = u$, represent the root-to-leaf path in to u.

DEFINITION **2** *The node* n_j *is a* distinguishing ancestor *for* u *with respect to* v *if the intersection of the results of the two queries, keyQuery*$(< n_j, \ldots, n_k >)$ *and keyrtol(v), is empty.*

If n_j is a *distinguishing ancestor* for u with respect to v, then we write $u \mid^{n_j} v$. Thus, for the above example, 4 is a distinguishing ancestor of 14 with respect to every other *column-compatible* node. In other words, issuing the query from 4 to 14, we will obtain all the results corresponding to node 14 and no result corresponding to any other column-compatible node (such as node 39).

Observe that the *distinguishing ancestor* relation is not a symmetric relation. For example, in the annotated schema graph shown in Figure 30, consider schema nodes 14 and 39, which are column-compatible. Now, 14 $|^4$ 39 is true. Notice that node 4 is an ancestor of node 14 but not an ancestor of node 39. So, 39 $|^4$ 14 is false.

DEFINITION **3** *The* lowest distinguishing ancestor *for u with respect to v, $u \,||\, v$, is the lowest ancestor w of u such that $u \,|^w\, v$.*

We represent this as $w = lda(u, v)$ or $w = u \,||\, v$. The *lda* relation is not symmetric. For example, $14 \,||\, 39 = 4 \neq 39 \,||\, 14$.

Using these definitions, and our previously defined notion of a bijective column mapping, we have the following lemma that aids in the identification of lowest distinguishing ancestors:

LEMMA **6** *Let u and v be two column-compatible nodes in the schema graph $\mathcal{T}$, where $annot(u) = annot(v) = R.C$ and $R.C$ is bijectively mapped. Then $u \,|^{root(\mathcal{T})}\, v$ holds.*

Proof: For $u \,|^{\mathrm{root}(\mathcal{T})}\, v$ to hold, we need to show that the intersection of the results of keyQuery(<root(T),...,u>) and keyrtol(v) is empty. Note that the former query is exactly the query keyrtol(u). Since $R.C$ is bijectively mapped, it satisfies the At-most-once property. So we have that the interesection of the results of keyrtol(u) and keyrtol(v) is empty. □

Notice how the notion of conflict used in the mapping_aware algorithm in Chapter 4 is similar to the notion of distinguishing ancestors. In the case of XML storage, we had the additional guarantee that the mapping completely described the relational data (follows from property P3 in Chapter 3.1.2). So, we are able to use a simple technique during run-time query translation to decide when two schema paths were in conflict.

In contrast, for the XML publishing scenario, we have to use the relational integrity constraints to check if two schema paths are in conflict. Since this can be an expensive operation, we move this to a precomputation stage. The resulting summary information is the *lda* information.

Computing Summary Information from the Constraints

Given an XML-to-Relational mapping $\mathcal{T}$ and the integrity constraints that hold on the underlying relational schema, we precompute the following information

- For each relational column $R.C$, is $R.C$ *bijective*?
- For every pair of column-compatible nodes (u, v),
  $u \mid\mid v$ and $v \mid\mid u$.

In this computation, we use procedures for solving the following problems on conjunctive queries in the presence of constraints.

UQC: Given a conjunctive query Q_1 and a union of conjunctive queries Q_2, is $Q_1 \subseteq Q_2$ under set semantics?

EQI: Is the intersection of two given conjunctive queries empty?

DUP: Are the results of a given conjunctive query duplicate-free?

We first describe how we compute the summary information given solutions for the above three problems. We then describe one way of developing procedures for these three problems using the techniques proposed in [DT01].

Identifying Bijective Column Mappings:

To check if a relational column $R.C$ is bijectively mapped, we need to check if $R.C$ is both At-least-once and At-most-once mapped.

```
procedure IsColumnBijective(R.C)
begin
    Let S ← nodes(R.C)
    For each node u ∈ S,
       If keyrtol(u) can have duplicates
          return false
    For each pair of nodes u, v ∈ S
       If keyrtol(u) ∩ keyrtol(v) ≠ φ
          return false
    If keyProject(R.C) ⊆ NodeKeyProject(R.C)
       return false
    return true
end
```

Figure 33: Algorithm to check if a relational column is bijectively mapped

Let us first look at the At-least-once property. Recall from Definition 1 that for this we need to check if $KeyProject(R.C) \subseteq NodeKeyProject(R.C)$ under multiset semantics. Since the key columns are also projected, the left hand side query has no duplicates. So, performing conjunctive query containment under set semantics will suffice.

Similarly, to check whether the At-most-once property is satisfied by $R.C$, we need to check if $KeyProject(R.C) \supseteq NodeKeyProject(R.C)$ under multiset semantics. Notice that this containment holds trivially under set semantics. Moreover, since the left hand side query has no duplicates, it suffices to check if the right hand side query also has no duplicates. Recall that $NodeKeyProject(R.C) = \bigcup_{u \in nodes(R.C)} keyrtol(u)$. Duplicates can be produced either in a single component of the union or across two different components. For the former case, we check if, for each $u \in nodes(R.C)$, any result tuple in $keyrtol(u)$ has two or more valuations. For the latter case, we need to check if $keyrtol(u)$ and $keyrtol(v)$ have a non-null intersection, for all node pairs $u, v \in nodes(R.C)$.

The algorithm to determine if a relational column is bijectively mapped is given in Figure 33.

```
procedure ldaComputation(u,v)
begin
    Let currAncestor = u
    while (true) do
       Let NS =<n_j, ..., n_k = u>
       If keyQuery(NS) ∩ keyrtol(v) = φ
          return currAncestor
       If (currAncestor = root(T))
          return null
             // this occurs if At-most-once condition is violated
       currAncestor = parent(currAncestor)
end
```

Figure 34: Algorithm to compute *lda* information

Computing least distinguishing ancestor information: The other summary information that we compute is the *lda* information. If we have a subroutine that verifies whether or not $u \mid^w v$ holds, then we can process the ancestors of u in a bottom-up fashion, and obtain $u \mid\mid v$. By definition, to check if $u \mid^w v$ is true we need to check if the intersection of the results of the queries keyQuery($<n_j, \ldots, n_k = u>$) and $keyrtol(v)$ is empty, where $n_j = w$. The algorithm is given in Figure 34.

Solutions to the UQC,EQI and DUP problems: One way of developing procedures for these three problems is to adapt the chase and query containment algorithms proposed in [DT01] to design procedures for the UQC, EQI and DUP problems. We describe this approach below. In general, any algorithm for conjunctive query containment under set semantics can be used to develop procedures for the above three problems.

In [DT01], the authors present an algorithm for conjunctive query containment (under set semantics) in the presence of a class of constraints known as **D**isjunctive **E**mbedded **D**ependencies (DEDs). We first review their results.

Definition **4** *The general form of Disjunctive Embedded Dependencies (DEDs) is the following*

$$\forall x_1 \ldots x_n [\phi(x_1, \ldots, x_n) \rightarrow$$
$$\vee_{i=1}^{l} \exists z_{i,1}, \ldots, z_{i,k_i} \psi_i(x_1, \ldots, x_n, z_{i,1}, \ldots, z_{i,k_i})]$$

where ϕ, ψ_i are conjunctions of relational atoms of the form $R(w_1, \ldots, w_l)$ and equality atoms of the form $w = w'$, where $w_1, \ldots, w_l, w, w'$ are variables.

The class of DEDs covers a rich set of constraints including functional dependencies, inclusion dependencies, multi-relational inclusion dependencies and uniqueness constraints, and a restricted class of domain constraints. For example, the key condition on the ITEM relation in our example in Section 5.1, can be expressed as

$$\forall x_1, x_2, x_3, x_4, x_5 [Item(x_1, x_2, x_3) \wedge Item(x_1, x_4, x_5)$$
$$\rightarrow (x_2 = x_4) \wedge (x_3 = x_5)]$$

The paper [DT01] presents the following results:

1. They present a generalization of the classical chase algorithm for embedded dependencies [BV84] to the class of DEDs. While the result of the classical chase procedure is a chase sequence, in the presence of DEDs, the result is a chase *tree*. More details can be found in [DT01].

2. Given conjunctive queries Q_1, Q_2, and the set D of DEDs, assume that the chase of Q_1 with D terminates. Then we have (1) Q_1 is equivalent to the union of the leaves of the chase tree, and (2) Q_1 is contained in Q_2 under D if and only if there is a containment mapping from Q_2 into every leaf $L \in chase_D(Q_1)$.

We now briefly explain our procedures for the UQC, EQI and DUP problems.

UQC: We need to verify whether $Q_1 \subseteq Q_2$, where Q_1 is a conjunctive query and $Q_2 = \bigcup Q_2^i$. We then have the following result: assume that the chase of Q_1 with D terminates.

Then we have that Q_1 is contained in Q_2 under D if for every leaf $L \in chase_D(Q_1)$, there is a containment mapping from some Q_2^i into L.

EQI: In order to check if the intersection of two conjunctive queries Q_1 and Q_2 is empty, we need to check if the query $Q = Q_1 \wedge Q_2$ has an empty result, i.e., $Q_1 \wedge Q_2 \subseteq \phi$. Here the distinguished (i.e., the projected) variables of Q_1 and Q_2 are equated and become the distinguished variables of Q.

DUP: In order to check whether a conjunctive query Q produces duplicate results, we take two copies of Q, Q_1 and Q_2 and construct the query $Q^{'} = Q_1 \wedge Q_2$. As above, the distinguished (i.e., the projected) variables of Q_1 and Q_2 are equated and become the distinguished variables of $Q^{'}$. Then we compute $chase(Q^{'})$ and check if for every conjunct in $Q^{'}$ that was originally from Q_1, the values in the key columns are the same as the values in the corresponding conjunct from Q_2. If so, multiple evaluations for a single tuple are not possible.

5.5.3 Run-Time Query Translation Algorithm

The run-time query translation algorithm is outlined in Figure 35. Given a path expression query Q, we first identify the parts of the schema that match the query. Let S denote the set of matching schema nodes. For purposes of exposition, we assume that S consists only of leaf nodes, leaving the handling of non-leaf nodes to Section 5.6.1.

We then partition the set S into two sets based on whether the corresponding relational column is bijectively mapped. For the set S_{nonbij}, we construct the root-to-leaf queries just like prior algorithms. On the other hand, for the set S_{bij} we utilize the summary information to eliminate parts of the query that are redundant. This is a two stage process: first we find the longest prefix that can be eliminated for each node $n \in S_{bij}$

procedure constraint_aware(Q)
begin
Let $S \leftarrow PathId(Q)$
Partition S into S_{bij} and S_{nonbij}
based on whether $annot(n)$ is *bijective*
$SQL_{nonbij} = \bigcup_{n \in S_{nonbij}} rtol(n)$
Prefix-Elimination(S_{bij})
$SQL_{bij} =$ SQLGen(S_{bij})
Return $SQL_{bij} \bigcup SQL_{nonbij}$
end

Figure 35: constraint_aware query translation algorithm for path expression queries

(*Prefix-Elimination*); then we construct the SQL query using the prefix-eliminated set of nodes (SQLGen). Finally, we union the queries corresponding to the bijective and non-bijective nodes.

We next describe the prefix-elimination and SQLGen stages.

Eliminating Redundant Prefixes

The *Prefix-Elimination* algorithm is given in Figure 36. We use the pre-computed information about least distinguishing ancestors in this computation. Instead of taking the naive approach of issuing the full query for each of these nodes and taking their union, we wish, at the very least, to be able to issue a smaller query for each node $n \in S$. Thus, we want to find the lowest ancestor a such that $Query(< a, \ldots, n >)$ returns the correct answer, that is, where the prefix of $rtol(n)$ from root($\mathcal{T}$) to a can be safely eliminated. There are two conditions to check here:

- a must distinguish n from all column-compatible nodes not in S. This computation corresponds to the *for* loop in Figure 36.

- For each column-compatible node $n_1 \in S$, either the two queries are combinable or

procedure Prefix-Elimination(S)
begin
1. for each node $n \in S$ do
2. Let $Schema(n)$ denote the set of schema nodes mapped to the same column as n
3. Let $Conflict(n) \leftarrow Schema(n) - S$
4. Let LDA_Set(n) denote set of $n \mid\mid x$ for every node x in Conflict(n)
5. C_lda(n) = highest node in LDA_Set(n)
6. While true do
7. If $(\exists n, n_1 \in S)$, such that RelSeq(C_lda(n),n) and RelSeq(C_lda(n_1),n_1) are not combinable and $n \mid\mid n_1$ is a strict ancestor of C_lda(n)
8. Then
9. Increment C_lda(n) by one node
10. Else
11. Break
end

Figure 36: Prefix-Elimination phase

a distinguishes n from n_1. This corresponds to the *while* loop in Figure 36.

The *while* loop is an iterative process that will terminate in at most $(k * d)$ iterations, where $k = |S|$ and d is the maximum depth (in the XML schema) among all nodes in S. At the end of this process we have the prefix eliminated node sequence for every node in S.

In the while loop, if there are multiple pairs of nodes satisfying the condition in step 7, we choose the pair $<n, n_1>$ such that RelSeq(C_lda(n),n) is the shortest relation sequence. This way, we keep all the relation sequences corresponding to nodes still being processed of roughly the same length. This is necessary for proving the optimality of this algorithm under the metric, *PrefixMetric*.

SQLGen Stage

We next construct the optimized SQL query by taking the prefix-eliminated set of nodes and grouping multiple paths that involve the same sequence of relations. Let $\mathcal{NS}$ = {<C_lda(n),...,n >: $n \in PathId(Q)$}. Recall that combinability of node sequences is an equivalence relation. We partition $\mathcal{NS}$ based on combinability and construct a SQL query for each equivalence class created. The final SQL query is the union of the queries across all equivalence classes. Notice that all the queries in an equivalence class have the same relation sequence and differ only in the selection conditions. This operation is correct under multi-set semantics because it is only applied to columns that are bijectively mapped.

5.5.4 Analysis

THEOREM 4 *Given a tree XML-to-Relational mapping $\mathcal{T}$ along with the integrity constraints that hold on the underlying relational schema, and a path expression query P, the constraint_aware approach outputs a correct equivalent SQL query. The running time of the query translation algorithm is polynomial in the size of the input, while the running time of the precomputation stage may be exponential in the size of the input.*

Proof: For a given path expression query Q, let Q_1 be the SQL query obtained by applying the constraint_aware-translation algorithm. Let Q_2 be the SQL query obtained by applying the *baseline* query translation algorithm. Recall that $Q_2 = \bigcup_{n \in S} rtol(n)$, where $S = PathId(Q)$. We show that $Q_1 = Q_2$ under multiset semantics, which proves that the constraint_aware approach always outputs a correct equivalent SQL query.

From the algorithm in Figure 35, we have

$$Q_1 = SQL_{bij} \bigcup SQL_{nonbij}$$

Similarly, Q_2 can be written as

$$Q_2 = \bigcup_{n \in S_{bij}} rtol(n) \quad \bigcup \quad \bigcup_{n \in S_{nonbij}} rtol(n)$$

By definition, we have

$$SQL_{nonbij} = \bigcup_{n \in S_{nonbij}} rtol(n)$$

So, we need to show that

$$SQL_{bij} = \bigcup_{n \in S_{bij}} rtol(n)$$

Since each relational column being projected in these two queries is *bijectively* mapped, it suffices if we prove that

$$keySQL_{bij} = \bigcup_{n \in S_{bij}} keyrtol(n)$$

Here $keySQL_{bij}$ refers to the query SQL_{bij} augmented with the appropriate key column(s) in the projection clause. We show this equality by proving containment in both the directions. In the following discussion, we assume that no two nodes in S are annotated with different column names from the same relation. Otherwise, we partition S based on the annotations and the following proof shows that the equality holds for each partition.

1. $keySQL_{bij} \supseteq \bigcup_{n \in S_{bij}} keyrtol(n)$**:** If we show that every tuple t appearing in the result of the RHS query also appears in the result of the LHS query, the containment result holds under set semantics. We first show this to be true. Let t belong to relation R and $R.C$ be the *bijective* column being projected. Let n_1 be the schema node, such that, t occurs in the result of $keyrtol(n_1)$. The existence of a unique schema node n_1 follows from the fact that $R.C$ is bijectively mapped. Consider the same node n_1 in the LHS query, $keySQL_{bij}$. Since node $n_1 \in S_{bij}$, at the end of the Prefix-Elimination stage in the constraint_aware algorithm, we have

a node sequence $NS =< Conflict\_lda(n_1), \ldots, n_1 >$ corresponding to node n_1. In the Grouping phase, we create a basic SQL query (say Q_3) corresponding to RelSeq(NS). It can be seen that tuple t occurs in the result of query Q_3. This is due to the fact that the relation sequence corresponding to Q_3 is a suffix of the relation sequence corresponding to $keyrtol(n_1)$. Moreover, the where clause in Q_3 is of the form C_{NS} *or* (conditions corresponding to other nodes with same relation sequence as n_1). Notice that C_{NS} is a subset of the where clause of $keyrtol(n_1)$. As a result, tuple t occurs in the result of query Q_3. This implies that t occurs in the result of $keySQL_{bij}$. Hence, we have the required containment result under set semantics.

Since $R.C$ is *bijectively* mapped, t appears exactly once in the result of the RHS query. So, we have the containment result under multiset semantics.

2. $keySQL_{bij} \subseteq \bigcup_{n \in S_{bij}} keyrtol(n)$**:** Here, if we show that every tuple t occuring in the result of the LHS query also occurs in the result of the RHS query, we prove the containment under set semantics. Since the column under consideration is *bijectively* mapped, there are no duplicates in the result of the RHS query. So, if we also show that no tuple will appear multiple times in the result of the LHS query, we prove the containment under multiset semantics.

   We first show that if a tuple t occurs in the result of the LHS query, it also occurs in the result of the RHS query. Since the corresponding column is *bijectively* mapped, there is some schema node n, such that, $keyrtol(n)$ has t in its result set. We next show that $n \in S_{bij}$, i.e., the root-to-leaf path corresponding to n matches the query. Assume that the tuple is produced by a basic SQL query Q_4 in $keySQL_{bij}$. Let the where clause of Q_4 be of the form C_{common} and (C_{n_1} or C_{n_2} or $\ldots or$ or C_{n_k}).

There is a condition C_{n_i} that was satisfied by some evaluation making tuple t appear in the result. Let n_i be the corresponding schema node. Now t appears in the results of both the prefix-eliminated query for n_i and $keyrtol(n)$. This implies that either (i) $n_i = n$ or (ii) $n \in S_{bij}$ and the prefix-eliminated queries of n_i and n are combineable. Otherwise, the *lda* for n_i would have been a higher ancestor (steps 1-5 of the prefix-elimination algorithm (Figure 36). In either case, we have that $n \in S_{bij}$. Combining this with the fact that t appears in the result of $keyrtol(n)$, we have that t appears in the result of $\bigcup_{n \in S_{bij}} keyrtol(n)$.

We next show that no tuple appears multiple times in the result of the LHS query. Assume the contrary. Suppose a tuple t appears more than once in the result. Since the relational column being projected is bijectively mapped, a single basic SQL query in $keySQL_{bij}$ will not produce duplicates. So, there are two basic SQL queries in $keySQL_{bij}$ that have t in their result set. Let these queries be Q_5 and Q_6. As in the previous case, we can find the corresponding schema nodes n_5 and n_6 that cause t to appear in the result of Q_5 and Q_6 respectively. Let NS_5 and NS_6 be the corresponding prefix-eliminated node sequences. Since the column is *bijectively* mapped, there is some schema node n, such that, $keyrtol(n)$ has t in its result set. Using the same argument as in the previous case, we see that $n \in S_{bij}$. Let NS be the prefix-eliminated node sequence for n. The two node sequences NS and NS_5 are combineable, as otherwise this pair violates the condition in step 7 of the prefix-elimination algorithm (Figure 36). Similarly, the node sequences NS and NS_6 are also combineable. Since combinability is an equivalence relation, this implies that NS_5 and NS_6 are also combineable. This contradicts the assumption that n_5 and n_6 belong to two different basic SQL queries. So, there are no duplicates in the

LHS query.

We have shown that $Q_1 = Q_2$ under multiset semantics, which proves that the constraint_aware approach always outputs a correct equivalent SQL query.

The running time of the algorithm is polynomial as the while loop terminates in a polynomial number of iterations and the cost of each step in the algorithm is polynomial. The precomputation phase may have an exponential cost since the subroutines for the UQC, EQI and DUP problems may have exponential running times. □

THEOREM **5** *Given sound and complete algorithms $\mathcal{A}$ and $\mathcal{A}'$ for the UQC and DUP problems over the class $\mathcal{C}$, the XML-to-SQL Query Translation problem for a bijective tree XML view under metric* PrefixMetric *can be solved in polynomial time.*

Proof: We need to show that the SQL query output by the constraint_aware algorithm is optimal under metric *PrefixMetric*. In other words, if we show that for each node $n \in S$, the constraint_aware algorithm identifies the lowest ancestor till which we need to go up, we have proved the optimality of the algorithm.

Since we have sound and complete algorithms for UQC and DUP, the *lda* computation is also complete. Consider a node $n \in S$. Let C_lda(n) be the final ancestor chosen by the constraint_aware algorithm. Let Q_1 be a correct SQL query and Q_1^n be the corresponding fragment that returns results corresponding to the root-to-leaf path ending in n. Suppose Q_1^n corresponds to stopping at the ancestor n_1. We argue that n_1 is not a proper descendant of C_lda(n). Otherwise, Q_1^n will return results corresponding to some leaf node $\notin S$ or not combinable with RelSeq(C_lda(n),n). The former implies that Q_1 does not return the correct result and the latter implies that Q_1 returns duplicate results. This contradicts the assumption that Q_1 is a correct equivalent SQL

query. Hence, n_1 is at least as high as C_lda(n), implying that the query output by the constraint_aware algorithm is optimal under metric *PrefixMetric*. □

5.6 Extensions to More General Cases

In this section, we discuss how the methods discussed to up to this point extend to more general situations. Note that our optimization techniques will never generate an incorrect query — they will either not apply (in which case we will generate the naive query) or they will apply and will generate a query expected to be more efficient than the naive query. Hence the discussion here outlines techniques that allow us to apply optimizations to more queries.

5.6.1 Path Expression Queries Involving Non-Leaf Nodes

In our discussion in Section 5.5.3 on translating path expression queries, we assumed that the query matches a set of leaf nodes in the schema. If the result includes non-leaf nodes as well, then there are two alternative ways of returning the resulting XML elements corresponding to the non-leaf nodes.

1. For each non-leaf element, we can return an identifier or representative subelement. In this case, each non-leaf node n in the schema is associated with a child leaf node n_c. If n appears in a query result, then the corresponding n_c elements are returned instead. For example, we can associate the key field(s) of the corresponding relations with each non-leaf node.

2. For each non-leaf element, we can return the entire subtree rooted at this element. The problem of efficiently constructing entire subtrees of XML documents has been

considered in [FMS01, SSB+00]. We leave the interesting problem of combining our algorithm with one of these algorithms for future work.

5.6.2 Beyond Path Expressions

Our techniques can be extended in a straightforward way to handle branching path expression queries; because that extension does not provide any additional insight, we do not discuss that extension here.

We now briefly describe how to extend constraint_aware translation to more general queries. A path expression query corresponds to a single For clause in XQuery. Consider an XQuery that has several of these For clauses and (optional) Where clauses. A natural way of applying our techniques is to perform constraint_aware translation for each of the individual path expressions, and then combine the resulting queries with appropriate join conditions. For example, consider a query XQ involving two path expressions p_1 and p_2 with a join condition between them. We apply our constraint_aware translation on p_1 and p_2 individually to obtain relational queries Q_1 and Q_2 respectively. Note that Q_1 and Q_2 are the union of k_1 and k_2 queries respectively. We generate the query $Q = Q_1 \bowtie Q_2$ as the SQL query corresponding to XQ. If $k_1 > 1$ or $k_2 > 1$, then we could have generated the final SQL query in a number of other ways. For example, we could have distributed the unions over the join and generated the query Q' that is the union of $k_1 * k_2$ queries. Choosing the best query from amongst these (possibly exponential) alternatives is also an interesting area for future work.

5.6.3 Beyond Bijective Mappings

Recall that our technique optimizes the SQL query corresponding to bijective parts of the mapping. It constructs the *baseline* query for the non-bijective parts of the mapping. While we expect bijectively mapped columns to be common, we have extended our algorithm to perform efficient XML to SQL query translation when the At-least-once condition is violated but the At-most-once condition is satisfied. We outline the main idea here with an example.

Let us look at the scenario when a relational column $R.C$ satisfies the At-most-once condition but violates the At-least-once condition. For example, consider the example in Figure 30. While the XMark XML schema contains information about items in six continents, in reality, there is actually a seventh continent (Antarctica). So, it is reasonable to assume that the relational schema has an integrity constraint on *Item.continent* allowing seven potential values. In this case, parts of the relational data are not present in the XML view, namely the items corresponding to Antarctica. Now while SQ_1 and SQ_1^1 are correct SQL queries for Q_1, OQ_1 is not. The best query in this scenario as discussed in Section 5.1 is a variation of SQ_1^1 that combines all the six queries into one, since they are on the same sequence of relations. This query is given below.

```
select   C.category
from     Item I, InCat C
where    I.id = C.itemid and
         I.continent IN {'africa','asia','australia','europe','namerica','samerica'}
```

Notice how we were able to group together the six paths corresponding to different

continents. This was possible due to the fact that InCat.category satisfied the At-most-Once condition. As a result, the rtol queries corresponding to any two column-compatible schema nodes mapped to InCat.category will not have any common results. So, we can translate the unions to a disjunction. In other words, we can perform the SQLGen phase without any change.

On the other hand, we need to be careful in the prefix-elimination stage. We cannot eliminate any prefix below the continent nodes due to one missing continent in the XML schema. To account for this fact, we have to augment the prefix-elimination stage. We do this as follows: $S = nodes(InCat.category) = \{14, 19, 24, 29, 34, 39\}$. For each schema node $n \in S$, we compute the lowest schema node below which the prefix cannot be eliminated (since the column is not completely exported). Let us call this *lowest required ancestor* (lra(n)). For example, lra(14) = 4 and lra(39) = 9. This ensures that the selection condition on Item.continent is always present in the query.

The lra computation is another summary information that we precompute for schema nodes corresponding to relational columns that violate the At-least-once condition. The prefix-elimination algorithm in Figure 36 has to be modified so that in the `for` loop, for each node $n \in S$, lra(n) is added to LDA_Set(n). No other change needs to be made to the rest of the algorithm.

5.7 Summary

We have considered the problem of generating efficient SQL queries for XML workloads and showed that published translation algorithms can generate SQL queries that are suboptimal. We consider the problem of where to add the intelligence in order to obtain optimized SQL queries using integrity constraint information. Our results argue that

the quality of the resulting SQL should be a concern of the translation algorithm itself, rather being left in the hands of a traditional relational optimizer. This is because many "easy" opportunities for optimization are apparent only when the XML view definition and relational integrity constraints are considered simultaneously. These opportunities vanish by the time the relational optimizer is presented with SQL.

Chapter 6

Conclusions and Future Work

In this thesis, we looked at the XML-to-SQL query translation problem that arises when queries are posed over an (logical) XML view of data stored (physically) in an RDBMS. The main contributions of this thesis are:

- We presented the XML_to_SQL algorithm for translating path expression queries into SQL, even in the presence of recursion in the XML schema and XML query. This algorithm demonstrates that a path expression query can be translated into a single SQL query, irrespective of how complex the XML schema is.

- We showed how the quality of the final SQL queries can be improved if we make use of the "lossless from XML" constraint, i.e., the entire relational data resulted from the lossless shredding of XML documents conforming to the given XML schema. We extended the above algorithm to exploit the presence of this constraint in the XML storage scenario and generate efficient SQL queries.

- For the XML publishing scenario, we showed how relational integrity constraints can be used to generate efficient SQL queries. Since using these constraints at query translation time requires solutions to intractable problems, we proposed an approach that relies on precomputation. The associated query translation algorithm translates path expression queries over tree XML schema into efficient SQL queries.

Future Work

The following issues are open for future work.

- Extending our algorithms to translate more complex FLWOR XQuery queries is an interesting open problem. In particular, extending the mapping_aware algorithm for XML storage and the constraint_aware algorithm for XML publishing to more complex XML queries is an interesting avenue for future research.
- For the XML publishing scenario, the constraint_aware algorithm uses the constraint information to optimize queries over bijective parts of the XML view. It also handles the scenario where the At-least-once condition is violated. But, if the At-most-once condition is violated, then the algorithm does not perform any optimizations during query translation. Extending the algorithm to use the constraint information even in this case is future work.
- As we saw in Chapter 2, three different techniques are used while designing relational decompositions to capture the hierarchical nature of the XML data — id-based, interval-based and path-based. In this thesis, we focussed on the id-based techniques. Investigating the XML-to-SQL query translation problem in the presence of all three techniques is open.
- For the XML storage scenario, we presented an approach that generates efficient SQL queries without resorting to the use of relational integrity constraints. An alternative approach is to generate complex SQL queries and then minimize them using the relational integrity constraints. Identifying the right combination of the class of relational integrity constraints generated in this scenario, the class of relational queries that need to be minimized and efficient algorithms for doing the same is another interesting direction to pursue.

Bibliography

[Ade] Naa classified advertising standards task force. http://www.naa.org/technology/clsstdtf/.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AKJP+02] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural Joins: A Primitive For Efficient XML Query Pattern Matching. In *Proceedings of the 18th International Conference on Data Engineering*, 2002.

[ASU79] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalence among relational expressions. *SIAM Journal of Computing*, 8(2):218–246, 1979.

[AYCLS01] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of tree pattern queries. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 497–508, 2001.

[BCF+02] Michael Benedikt, Chee Yong Chan, Wenfei Fan, Rajeev Rastogi, Shihui Zheng, and Aoying Zhou. DTD-Directed Publishing with Attribute Translation Grammars. In *Very Large Data Bases (VLDB) Conference*, pages 838–849, 2002.

[BDFS97] Peter Buneman, Susan B. Davidson, Mary F. Fernandez, and Dan Suciu. Adding Structure to Unstructured Data. In *Proceedings of 6th International Conference on Database Theory*, volume 1186 of *Lecture Notes in Computer Science*, pages 336–350, 1997.

[BFRS02] Philip Bohannon, Juliana Freire, Prasan Roy, and Jerome Simeon. From XML Schema to Relations: A Cost-based Approach to XML Storage. In *Proceedings of the 18th International Conference on Data Engineering*, pages 64–75, 2002.

[BGK+02] Philip Bohannon, Sumit Ganguly, Henry F. Korth, P. P. S. Narayan, and Pradeep Shenoy. Optimizing View Queries in ROLEX to Support Navigable Tree Results. In *Very Large Data Bases (VLDB) Conference*, pages 119–130, 2002.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 310–321, 2002.

[BKTT04] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang-Chiew Tan. Archiving scientific data. *ACM Transactions on Database Systems*, 29(1):2–42, 2004.

[BV84] Catriel Beeri and Moshe Vardi. A Proof Procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, 1984.

[CAYLS02] SungRan Cho, Sihem Amer-Yahia, Laks V. S. Lakshmanan, and Divesh Srivastava. Optimizing the Secure Evaluation of Twig Queries. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB)*, pages 490–501, 2002.

[CDZ02] Yi Chen, Susan B. Davidson, and Yifeng Zheng. Constraint preserving XML Storage in Relations. In *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB)*, pages 7–12, 2002.

[Cho02] Byron. Choi. What Are Real DTDs Like. In *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB)*, pages 43–48, 2002.

[CJLP03] Zhimin Chen, H. V. Jagadish, Laks V. S. Lakshmanan, and Stelios Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB)*, pages 237–248, 2003.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Databases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90. ACM Press, 1977.

[CV93] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of Real Conjunctive Queries. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 59–70, 1993.

[CVZ+02] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB)*, pages 263–274, 2002.

[DB2] DB2 XML Extender. http://www.ibm.com/software/data/db2/extenders/xmlext.

[DFS99] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 431–442, 1999.

[DPT99] Alin Deutsch, Lucian Popa, and Val Tannen. Physical Data Independence, Constraints, and Optimization with Universal Plans. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 459–470, 1999.

[DT01] Alin Deutsch and Val Tannen. Containment and Integrity Constraints for XPath Fragments. In *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases (KRDB 2001)*, volume 45 of *CEUR Workshop Proceedings*. Technical University of Aachen (RWTH), 2001.

[DT03a] Alin Deutsch and Val Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB)*, pages 201–212, 2003.

[DT03b] Alin Deutsch and Val Tannen. Reformulation of XML Queries and Constraints. In *Proceedings of 9th International Conference on Database Theory*, volume 2572 of *Lecture Notes in Computer Science*, pages 225–241, 2003.

[DTCO03] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Ozsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*, pages 623–634, 2003.

[Eig94] Frank Ch. Eigler. Translating GraphLog to SQL. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1994.

[EM02] Andrew Eisenberg and Jim Melton. SQL/XML is Making Good Progress. *SIGMOD Record*, 31(2):101–108, 2002.

[EZ76] Andrzej Ehrenfeucht and Paul Zeiger. Complexity Measures for Regular Expressions. *Journal of Computer and System Sciences (JCSS)*, 12(2):134–146, 1976.

[FFM03] Sergio Flesca, Filippo Furfaro, and Elio Masciari. On the minimization of Xpath queries. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB)*, pages 153–164, 2003.

[FHR+02] Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jerome Simeon. StatiX: making XML count. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 181–191, 2002.

[FK99] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[FMS01] Mary F. Fernandez, Atsuyuki Morishima, and Dan Suciu. Efficient Evaluation of XML Middle-ware Queries. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 103–114, 2001.

[FS98] Mary F. Fernandez and Dan Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 14–23, 1998.

[FSC+03] Mary F. Fernandez, Jerome Simeon, Byron Choi, Amelie Marian, and Gargi Sur. Implementing Xquery 1.0: The Galax Experience. In *Proceedings of*

29th International Conference on Very Large Data Bases (VLDB), pages 1077–1080, 2003.

[FTS00] Mary Fernandez, Wang-Chiew Tan, and Dan Suciu. SilkRoute: Trading between Relations and XML. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications networking*, pages 723–745. North-Holland Publishing Co., 2000.

[Gru02] Torsten Grust. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120, 2002.

[GSBS03] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*, pages 16–27, 2003.

[GST04] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proceedings of 30th International Conference on Very Large Data Bases (VLDB)*, 2004.

[GvKT03] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB)*, pages 524–535, 2003.

[HSJJ02] Sun Hongwei, Zhang Shusheng, Zhou Jingtao, and Wang Jing. Constraints-Preserving Mapping Algorithm from XML-Schema to Relational Schema. In *Engineering and Deployment of Cooperative Information Systems: First International Conference (EDCIS)*, volume 2480 of *Lecture Notes in Computer Science*, 2002.

[INC] INCITS H2.3 Task Group. http://www.sqlx.org.

[IR95] Yannis E. Ioannidis and Raghu Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *ACM Transactions on Database Systems*, 20(3):288–324, 1995.

[JK82] David S. Johnson and Anthony C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. In *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 164–169, 1982.

[JLWO03] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proceedings of the 19th International Conference on Data Engineering*, pages 253–263, 2003.

[JMS02] Sushant Jain, Ratul Mahajan, and Dan Suciu. Translating XSLT programs to Efficient SQL queries. In *Proceedings of the eleventh international conference on World Wide Web*, pages 616–626, 2002.

[KCN03] Rajasekar Krishnamurthy, Venkatesan T. Chakaravarthy, and Jeffrey F. Naughton. On the Difficulty of Finding Optimal Relational Decompositions for XML Workloads: A Complexity Theoretic Perspective. In *Proceedings of the 9th International Conference on Database Theory (ICDT)*, pages 267–281, 2003.

[KKN04] Rajasekar Krishnamurthy, Raghav Kaushik, and Jeffrey F. Naughton. Unraveling the Duplicate-Elimination Problem in XML-to-SQL Query Translation. In *Proceedings of the Seventh International Workshop on the Web and Databases (WebDB)*, 2004.

[KM00] Meike Klettke and Holger Meyer. XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics. In *Proceedings of the Third International Workshop on the Web and Databases (WebDB)*, pages 63–68, 2000.

[LBN03] Chengkai Li, Philip Bohannon, and P. P. S. Narayan. Composing XSL transformations with XML publishing views. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*, pages 515–526, 2003.

[LC00] Dongwon Lee and Wesley W. Chu. Constraints-preserving Transformation from XML Document Type Definition to Relational Schema. In *In Proceedings of the 19th International Conference on Conceptual Modeling*, volume 1920 of *Lecture Notes in Computer Science*, pages 323–338, 2000.

[LM01] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB)*, pages 361–370, 2001.

[MFK01] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB)*, pages 241–250, 2001.

[ML03] Murali Mani and Dongwon Lee. XML to Relational Conversion Using Theory of Regular Tree Grammars. In *Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web, VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop DTWeb. Revised Papers*, volume 2590 of *Lecture Notes in Computer Science*, pages 81–103, 2003.

[MS02] Gerome Miklau and Dan Suciu. Containment and Equivalence for an XPath Fragment. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 65–76, 2002.

[MS03] Amelie Marian and Jerome Simeon. Projecting XML Documents. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB)*, pages 213–224, 2003.

[MW99] Jason McHugh and Jennifer Widom. Compile-Time Path Expansion in Lore. In *Workshop on Query Processing for SemiStructured Data and Non-Standard Data Formats*, 1999.

[OXD] Oracle XML DB. http://otn.oracle.com/tech/xml/xmldb.

[Ram02] Prakash Ramanan. Efficient algorithms for minimizing tree pattern queries. In *SIGMOD*, 2002.

[RP02] Kanda Runapongsa and Jignesh M. Patel. Storing and Querying XML Data in Object-Relational DBMSs. In *XML-Based Data Management and Multimedia Engineering - EDBT 2002 Workshops XMLDM, MDDE, and YRWS, Revised Papers*, volume 2490 of *Lecture Notes in Computer Science*, pages 266–285, 2002.

[SKS+01] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene J. Shekita, Catalina Fan, and John Funderburk. Querying XML Views of Relational Data. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB)*, pages 261–270, 2001.

[SKWW00] Albrecht Schmidt, Martin L. Kersten, Menzo Windhouwer, and Florian Waas. Efficient Relational Storage and Retrieval of XML Documents. In *Proceedings of the Third International Workshop on the Web and Databases (WebDB)*, pages 47–52, 2000.

[SO87] Sreekumar T. Shenoy and Z. Meral Ozsoyoglu. A System for Semantic Query Optimization. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference (SIGMOD)*, pages 181–195, 1987.

[SSB+00] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB)*, pages 65–76, 2000.

[SSK+01] Jayavel Shanmugasundaram, Eugene J. Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Stratis Viglas, Jeffrey F. Naughton, and Igor Tatarinov. A General Technique for Querying XML Documents using a Relational Database System. *SIGMOD Record*, 30(3):20–26, 2001.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB)*, pages 302–314, 1999.

[SXM] SQLXML and XML Mapping Technologies. http://msdn.microsoft.com/sqlxml/default.asp.

[SY80] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM (JACM)*, 27(4):633–655, 1980.

[TH02] Pankaj M. Tolani and Jayant R. Haritsa. XGRIND: A Query-Friendly XML Compressor. In *Proceedings of the 18th International Conference on Data Engineering*, pages 225–234, 2002.

[TVB+02] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 204–215, 2002.

[W3C] W3c: World wide web consortium. http://www.w3.org/.

[WJLY03] Wei Wang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. PBiTree Coding and Efficient Processing of Containment Joins. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, pages 391–402, 2003.

[Woo02] Peter T. Wood. Containment for XPath Fragments under DTD Constraints. In *Proceedings of the 9th International Conference on Database Theory (ICDT 2003)*, volume 2572 of *Lecture Notes in Computer Science*, pages 300–314, 2002.

[XMa] Xmark: The XML benchmark project. http://monetdb.cwi.nl/xml.

[XTa] XML for Tables. http://www.alphaworks.ibm.com/tech/xtable.

[YASU01] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology (TOIT)*, 1(1):110–141, 2001.

[ZND+01] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD 2001 Electronic Proceedings*, 2001.

[ZO93] Xubo Zhang and Z. Meral Ozsoyoglu. On Efficient Reasoning with Implication Constraints. In *Proceedings of Third International Conference on Deductive and Object-Oriented Databases*, volume 760 of *Lecture Notes in Computer Science*, pages 236–252, 1993.

[ZO97] Xubo Zhang and Z. Meral Ozsoyoglu. Implication and Referential Constraints: A New Formal Reasoning. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 9(6):894–910, 1997.

[ZPR02] Xin Zhang, Bradford Pielech, and Elke A. Rundensteiner. Honey, I Shrunk the XQuery! – An XML Algebra Optimization Approach. In *Fourth ACM CIKM International Workshop on Web Information and Data Management (WIDM)*, pages 15–22, 2002.

Appendix A

Queries used in the experiments in Chapter 4.4

In this section, we present the details for the queries used in the experiments in Chapter 4.4. For each query Q, we give the XML query (XQ), the SQL query generated by exploiting the "lossless from XML" constraint (OQ) and the SQL query generated if the "lossless from XML" constraint is not used (NQ).

Query A1: Get the number of open-house ads in the campus area

XA1: count(//ad-instance//open-house/location[geo-area='area1'])

OA1: select count(*) from RE, LOC where (RE.category = 2 or RE.category = 3) and LOC.id = RE.id and LOC.area = 'area1'

NA1: select count(*) from RE, LOC, ADS, ADEX where RE.category = 2 and LOC.id = RE.id and LOC.area = 'area1' and RE.id = ADS.id and ADS.adid = ADEX.adid UNION ALL select count(*) from RE, LOC, ADS, ADINSTANCE where RE.category = 3 and LOC.id = RE.id and LOC.area = 'area1' and RE.id = ADS.id and ADS.adid = ADEX.adid

Query A2: Get the number of real-estate ads in the campus area

XA2: count(//ad-instance//real-estate//location[geo-area='area1'])

OA2: select count(*) from RE, LOC where LOC.id = RE.id and LOC.area = 'area1'

NA2: select count(*) from RE, LOC, ADS, ADEX where RE.category = 1 and LOC.id = RE.id and LOC.area = 'area1' and RE.id = ADS.id and ADS.adid = ADEX.adid UNION ALL select count(*) from RE, LOC, ADS, ADEX where RE.category = 2 and LOC.id = RE.id and LOC.area = 'area1' and RE.id = ADS.id and ADS.adid = ADEX.adid UNION ALL select count(*) from RE, LOC, ADS, ADEX where RE.category = 3 and LOC.id = RE.id and LOC.area = 'area1' and RE.id = ADS.id and ADS.adid = ADEX.adid UNION ALL select count(*) from RE, LOC, ADS, ADEX where RE.category = 4 and LOC.id = RE.id and LOC.area = 'area1' and RE.id = ADS.id and ADS.adid = ADEX.adid UNION ALL select count(*) from RE, LOC, ADS, ADEX where RE.category = 5 and LOC.id = RE.id and LOC.area = 'area1' and RE.id = ADS.id and ADS.adid = ADEX.adid UNION ALL select count(*) from RE, LOC, ADS, ADEX where RE.category = 6 and LOC.id = RE.id and LOC.area = 'area1' and RE.id = ADS.id and ADS.adid = ADEX.adid UNION ALL select count(*) from RE, LOC, ADS, ADEX where RE.category = 7 and LOC.id = RE.id and LOC.area = 'area1' and RE.id = ADS.id and ADS.adid = ADEX.adid

Query A3: Get the addresses of ads in the campus area

XA3: //ad-instance//location[geo-area='area1']/address

OA3: select address from LOC where area = 'area1'

NA3: with TEMP(address) as (LOC.address from RE, LOC, ADS, ADEX where RE.category = 1 and LOC.id = RE.id and RE.id = ADS.id and ADS.adid = ADEX.adid and LOC.area = 'area1' UNION ALL LOC.address from RE, LOC, ADS, ADEX where RE.category = 2 and LOC.id = RE.id and RE.id = ADS.id and ADS.adid = ADEX.adid and LOC.area = 'area1' UNION ALL LOC.address from RE, LOC, ADS, ADEX where RE.category = 3 and LOC.id = RE.id and RE.id = ADS.id and ADS.adid = ADEX.adid and LOC.area = 'area1' UNION ALL LOC.address from RE, LOC, ADS, ADEX where RE.category = 4 and LOC.id = RE.id and RE.id = ADS.id and ADS.adid = ADEX.adid and LOC.area = 'area1' UNION

ALL LOC.address from RE, LOC, ADS, ADEX where RE.category = 5 and LOC.id = RE.id and RE.id = ADS.id and ADS.adid = ADEX.adid and LOC.area = 'area1' UNION ALL LOC.address from RE, LOC, ADS, ADEX where RE.category = 6 and LOC.id = RE.id and RE.id = ADS.id and ADS.adid = ADEX.adid and LOC.area = 'area1' UNION ALL LOC.address from RE, LOC, ADS, ADEX where RE.category = 7 and LOC.id = RE.id and RE.id = ADS.id and ADS.adid = ADEX.adid and LOC.area = 'area1' UNION ALL LOC.address from TRANS, LOC, ADS, ADEX where TRANS.category = 9 and LOC.id = TRANS.id and TRANS.id = ADS.id and ADS.adid = ADEX.adid and LOC.area = 'area1' UNION ALL LOC.address from EMP, LOC, ADS, ADEX where EMP.category = 1 and LOC.id = EMP.id and EMP.id = ADS.id and ADS.adid = ADEX.adid and LOC.area = 'area1' UNION ALL LOC.address from EMP, LOC, ADS, ADEX where EMP.category = 2 and LOC.id = EMP.id and EMP.id = ADS.id and ADS.adid = ADEX.adid and LOC.area = 'area1' UNION ALL LOC.address from EMP, LOC, ADS, ADEX where EMP.category = 3 and LOC.id = EMP.id and EMP.id = ADS.id and ADS.adid = ADEX.adid and LOC.area = 'area1' UNION ALL LOC.address from EMP, LOC, ADS, ADEX where EMP.category = 4 and LOC.id = EMP.id and EMP.id = ADS.id and ADS.adid = ADEX.adid and LOC.area = 'area1') select address from TEMP

Query X1: Get the number of items in a particular category

XX1:

```
let $cat := //categories/category[name = 'pie'],
    $item := /site//item/Incategory[category = $cat/id]
return count($item)
```

OX1: select count(*) from INCATEGORY, CATEGORY where INCATEGORY.category = CATEGORY.id and CATEGORY.name = 'pie'

NX1$_1$: select count(*) from SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and INCATEGORY.category = CATEGORY.id and CATEGORY.name = 'pie' and ITEM.continent = 'africa' UNION ALL select count(*) from SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and INCATEGORY.category = CATEGORY.id and CATEGORY.name = 'pie' and ITEM.continent = 'asia' UNION ALL select count(*) from SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and INCATEGORY.category = CATEGORY.id and CATEGORY.name = 'pie' and ITEM.continent = 'australia' UNION ALL select count(*) from SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and INCATEGORY.category = CATEGORY.id and CATEGORY.name = 'pie' and ITEM.continent = 'europe' UNION ALL select count(*) from SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and INCATEGORY.category = CATEGORY.id and CATEGORY.name = 'pie' and ITEM.continent = 'namerica' UNION ALL select count(*) from SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and INCATEGORY.category = CATEGORY.id and CATEGORY.name = 'pie' and ITEM.continent = 'samerica'

NX1$_2$: with TEMP(category) as (select INCATEGORY.category from SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and ITEM.continent = 'africa' UNION ALL select INCATEGORY.category from SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and ITEM.continent = 'asia' UNION ALL select INCATEGORY.category from SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and ITEM.continent = 'australia' UNION ALL select INCATEGORY.category from SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and ITEM.continent = 'europe' UNION ALL select INCATEGORY.category

from SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and ITEM.continent = 'namerica' UNION ALL select INCATEGORY.category from SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and ITEM.continent = 'samerica') select count(*) from TEMP, CATEGORY where TEMP.category = CATEGORY.id and CATEGORY.name = 'pie'

Query X2: For a particular person, get categories of items for which (s)he made a bid

XX2:

let $bids := /site/open_auctions/open_auction/bidder[personref='person4892'],
$itemcat := //item[id=$bids/itemref/@item]/incategory/category,
$cat := distinct-values(//categories/category[id=$itemcat]/name)
return $cat

OX2: select distinct(CATEGORY.name) from OPENAUCTION, BIDDER, ITEM, INCATEGORY, CATEGORY where BIDDER.auctionid = OPENAUCTION.id and OPENAUCTION.itemid = ITEM.id and ITEM.id = INCATEGORY.itemid and INCATEGORY.category = CATEGORY.id and BIDDER.personref = 'person4892'

NX2$_1$: with TEMP(name) as (select distinct(CATEGORY.name) from OPENAUCTION, BIDDER, SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and BIDDER.auctionid = OPENAUCTION.id and OPENAUCTION.itemid = ITEM.id and ITEM.id = INCATEGORY.itemid and INCATEGORY.category = CATEGORY.id and BIDDER.personref = 'person4892' and ITEM.continent = 'africa' UNION ALL select distinct(CATEGORY.name) from OPENAUCTION, BIDDER, SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and BIDDER.auctionid = OPENAUCTION.id and OPENAUCTION.itemid

= ITEM.id and ITEM.id = INCATEGORY.itemid and INCATEGORY.category = CATEGORY.id and BIDDER.personref = 'person4892' and ITEM.continent = 'asia' UNION ALL select distinct(CATEGORY.name) from OPENAUCTION, BIDDER, SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and BIDDER.auctionid = OPENAUCTION.id and OPENAUCTION.itemid = ITEM.id and ITEM.id = INCATEGORY.itemid and INCATEGORY.category = CATEGORY.id and BIDDER.personref = 'person4892' and ITEM.continent = 'australia' UNION ALL select distinct(CATEGORY.name) from OPENAUCTION, BIDDER, SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and BIDDER.auctionid = OPENAUCTION.id and OPENAUCTION.itemid = ITEM.id and ITEM.id = INCATEGORY.itemid and INCATEGORY.category = CATEGORY.id and BIDDER.personref = 'person4892' and ITEM.continent = 'europe' UNION ALL select distinct(CATEGORY.name) from OPENAUCTION, BIDDER, SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and BIDDER.auctionid = OPENAUCTION.id and OPENAUCTION.itemid = ITEM.id and ITEM.id = INCATEGORY.itemid and INCATEGORY.category = CATEGORY.id and BIDDER.personref = 'person4892' and ITEM.continent = 'namerica' UNION ALL select distinct(CATEGORY.name) from OPENAUCTION, BIDDER, SITE, ITEM, INCATEGORY, CATEGORY where SITE.id = ITEM.siteid and BIDDER.auctionid = OPENAUCTION.id and OPENAUCTION.itemid = ITEM.id and ITEM.id = INCATEGORY.itemid and INCATEGORY.category = CATEGORY.id and BIDDER.personref = 'person4892' and ITEM.continent = 'samerica') select distinct(name) from TEMP

NX2$_2$: with TEMP(category,id) as (select INCATEGORY.category, ITEM.id from SITE, ITEM, INCATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and ITEM.continent = 'africa' UNION ALL select INCATEGORY.category, ITEM.id from SITE, ITEM, INCATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and ITEM.continent = 'asia' UNION ALL select INCATEGORY.category, ITEM.id

from SITE, ITEM, INCATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and ITEM.continent = 'australia' UNION ALL select INCATEGORY.category, ITEM.id from SITE, ITEM, INCATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and ITEM.continent = 'europe' UNION ALL select INCATEGORY.category, ITEM.id from SITE, ITEM, INCATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and ITEM.continent = 'namerica' UNION ALL select INCATEGORY.category, ITEM.id from SITE, ITEM, INCATEGORY where SITE.id = ITEM.siteid and ITEM.id = INCATEGORY.itemid and ITEM.continent = 'samerica') select distinct(CATEGORY.name) from OPENAUCTION, BIDDER, TEMP, CATEGORY where BIDDER.auctionid = OPENAUCTION.id and OPENAUCTION.itemid = TEMP.id and TEMP.category = CATEGORY.id and BIDDER.personref = 'person4892'