XML data exchange via relations

Xiaotong Fu (s1421412)

MSc Project Report
Master of Informatics
School of Informatics
University of Edinburgh

2015

Abstract

The problem of data exchange is that: suppose we have the source data conforming to the source schema, then we want to use the source data to populate another database with another schema following a set of mapping rules. This problem can be divided into two types: relational data exchange and XML data exchange. Although relational data exchange is well developed, XML data exchange is still primitive. Many new methods are proposed for this problem. Among them, 'XML data exchange via relations is a promising way to handle the problem of XML data exchange.

We implemented the main algorithms involved in the method 'XML data exchange via relations based on the paper [13]. Furthermore, we compared this method with the method implemented by a mapping system named ++spicy [22]. The experiments show that our method is practical and very efficient compared to the naive implementation on ++spicy.

Acknowledgements

I would like to thank my supervisor Leonid Libkin. His invaluable guidance and suggestions help me understand and successfully accomplish this project. I also would like to thank Mr. Paolo Guagliardo, who offered me a selfless assistance when exploring the system ++spicy.

Finally, I want to thank my family for their patience and support. Their encouragement reduced my stress.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Table of Contents

1 Introduction **9**
- 1.1 Hypothesis . . . 9
- 1.2 Motivation . . . 9
- 1.3 Results achieved . . . 10
- 1.4 The structure of the remainder . . . 10

2 Preliminaries **11**
- 2.1 Data exchange . . . 11
  - 2.1.1 Relational data exchange setting . . . 11
  - 2.1.2 XML data exchange setting . . . 12
- 2.2 Nested-relational DTD . . . 15
  - 2.2.1 Nested Relations . . . 15
  - 2.2.2 Nested-relational DTD . . . 15
- 2.3 Conjunctive tree query . . . 15
- 2.4 ++Spicy . . . 16

3 Background **17**
- 3.1 Related Work . . . 17
- 3.2 Assumption of the work . . . 18
  - 3.2.1 Restriction . . . 18
  - 3.2.2 Tools . . . 18
- 3.3 Contributions . . . 19

4 Algorithms and Implementation **21**
- 4.1 Algorithm 1: Translating DTDs . . . 21
  - 4.1.1 Creating Relations . . . 22
  - 4.1.2 Adding attributes . . . 22
  - 4.1.3 Implementation . . . 23
- 4.2 Algorithm 2: Translating XML documents . . . 24
  - 4.2.1 Form a tuple . . . 24
  - 4.2.2 Attributes of a tuple . . . 25
  - 4.2.3 Implementation . . . 26
- 4.3 Algorithm 3: Translating Query . . . 26
  - 4.3.1 Translate Tree Pattern . . . 26
  - 4.3.2 Implementation . . . 28
- 4.4 Algorithm 4: Translating Mapping . . . 29

5 Evaluation **31**
5.1 Environment 31
5.2 Procedure of experiments 31
5.3 Results 32
5.4 Analysis 34

6 Conclusion **35**
6.1 Observation and remarks 35
6.2 Unsolved problems 35
6.3 Future work 35

Bibliography **37**

Chapter 1

Introduction

The problem of data exchange can be defined as following: given a source instance D under the source schema S, a target schema S' and a mapping between S and S', we want to find a target instance D' that is in accord with the mapping. More and more efforts are put on studying this subject because nowadays we are dealing with data from a variety of sources. In the past years, most research focus on the relational data exchange. However, since the XML documents are becoming more and more popular, studying the problems of XML data exchange becomes valuable.

1.1 Hypothesis

There are two types of data exchange: relational data exchange and XML data exchange. Since relational database technology has been developed for many years, relational data exchange techniques are well studied and of good performance. Although the research for the XML data exchange received more attention in recent years, the XML data exchange techniques are still naive compared to the relational data exchange.

Therefore, there is a natural way named 'XML Data exchange via relations' to solve the XML data exchange problem: firstly we translate XML data exchange setting to the relational data exchange setting; then we can directly apply a mature relational data exchange engine to the generated relational data exchange setting.

1.2 Motivation

Although the correctness and the restriction of this approach are both proved and discussed in the paper [13], there is a need of implementation and corresponding experiments to figure out the exact performance of this approach.

The objective of this project is: implementing the primitive algorithm of the technique 'XML data exchange via relations' proposed in the paper [13], then compare this

method with a naive XML data exchange approach implemented by a system named ++Spicy.

1.3 Results achieved

The whole approach contains four parts:

- the algorithm of translating XML DTD to relational schema;
- the algorithm of translating XML documents to relations populated with tuples;
- the algorithm of translating XML queries to relational queries; and
- the algorithm of translating XML mappings to relational mappings.

All the four algorithms are successfully programmed by Python.

The relational data exchange setting generated by our implementations can be successfully loaded into the relational data exchange engine ++spicy. The exchange engine ++spicy can properly produce the correct relational target instances.

The experiments show that the approach XML data exchange via relation is more efficient than the approach implemented by the ++Spicy.

1.4 The structure of the remainder

The rest of this paper is organized as following: chapter 2 introduced some basic knowledge in the data exchange field; chapter 3 presented the background of this project and the assumptions we made; the details of the algorithms and the key points of our implementation are given in the chapter 4; the experiments and analysis are reported in the chapter 5; finally, the conclusion is made in the chapter 6.

Chapter 2

Preliminaries

2.1 Data exchange

Data exchange is the problem that can be described as following: given an instance of a source schema and relationship (mappings) between the source schema and target schema, we want to find an instance of the target schema conforming to the mapping and preserving the information from the source instance. Then we can answer our queries [14] [15] on the target instance. Hence, the four ingredients of data exchange are: the schemas, the instances, the mappings, the queries.

2.1.1 Relational data exchange setting

2.1.1.1 Schema and Relations

In the relational data exchange scenario, the **schema** indicates the relational database schema. This is a formal structure that describes how the data are organized in the database. The **instances** mean the relations, which are the collections of data in form of tuples. The structure of relations should conform to its schema. For example, a schema can be Book(title, author, publisher), and the relation according to this schema could be Table 2.1.

Mathematics	Peter	Ace Books
Information Theory	John	Free Press
Art of War	Sun	Titan Books

Table 2.1: Relation

2.1.1.2 Mapping

The relationship between the source and target schema is called mappings. These mappings are a collection of logical formulas, which tell us how to populate the target

schema using the source relations. For example, suppose we have the source schema $\sigma = Book(Title, Author, Publisher)$, and the target schema $\tau = Writer(Name, Work, Year)$. Now we know the Name of Writer are from the Author attribute in the Book relation and the Work of Writer are from the Title of Book. This rule can be represented as a mapping: $\Sigma = Writer(x_1, x_2, z_1) : -Book(x_2, x_1, z_2)$. In this formula, the Name in the Writer and the Author in the Book share the same variable x_1, which means the Name in the Writer can be populated with the Author in the Book. At last, based on the Table 2.1 the target instance is Table 2.2.

Peter	Mathematics	Null
John	Information Theory	Null
Sun	Art of War	Null

Table 2.2: Target Relation

2.1.2 XML data exchange setting

In the XML scenario, the **schema** indicates the DTDs (Document Type Definition), the **instances** are XML documents, the **mappings** are formed by a set of tree patterns, and the **queries** are XQueries.

2.1.2.1 DTD

DTD (Document Type Definition) is to XML what Schema is to relational database. It defines the structure, the elements and the attributes of an XML file. Figure 2.1 is an example of a DTD. (This example is from the paper [13])

```
<!DOCTYPE r [

<!ELEMENT r (book*)>
<!ELEMENT book (author*,subject)>
<!ELEMENT author (name,aff)>
<!ELEMENT name EMPTY>
<!ELEMENT aff EMPTY>

<!ATTLIST book title CDATA #IMPLIED>
<!ATTLIST subject sub CDATA #IMPLIED>
<!ATTLIST name nam CDATA #IMPLIED>
<!ATTLIST aff aff CDATA #IMPLIED>

]>
```

Figure 2.1: Source DTD

2.1.2.2 XML Document

XML Documents are the files written by XML (Extensible Markup Language). They are designed to describe data. The structure of an XML document is defined by its DTD file. The Figure 2.2 is the XML file corresponding to the DTD in the Figure 2.1. An XML file can be easily represented as a tree in Figure 2.3.

```
<?xml version="1.0"?>
- <r>
  - <book title="Algorithm Design">
    - <author>
        <name nam="Kleinberg"/>
        <aff aff="CU"/>
      </author>
    - <author>
        <name nam="Tardos"/>
        <aff aff="CU"/>
      </author>
      <subject sub="CS"/>
    </book>
  - <book title="Algebra">
    - <author>
        <name nam="Hungerford"/>
        <aff aff="SLU"/>
      </author>
      <subject sub="Math"/>
    </book>
</r>
```

Figure 2.2: Source XML document

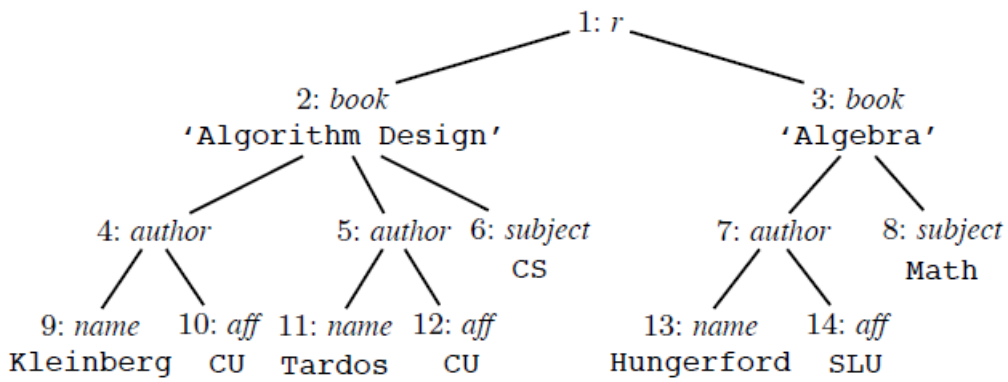


Figure 2.3: Source XML document tree

2.1.2.3 Mapping

The XML data exchange mappings are a set of tree patterns. The tree patterns can be expressed by the **Tree Pattern Formulae**. The **Attribute Formulae** form the basic elements of a tree pattern formula. An attribute formula can be defined as following [6]:

$$\alpha := l | l(@a_1 = x_1, \ldots, @a_n = x_n)$$

where l could be a wildcard or an XML element. $@a_1, @a_2 ... @a_n$ are the attributes of l. Each attribute $@a_i$ is assigned a variable. These variables indicate the relationship among attributes. For example, author$(@name = x_1, @aff = x_2)$ represent an 'author' element with different 'name' and 'aff'.

Based on attribute formula, a tree pattern formula can be defined as this [6]:

$$\varphi := \alpha | \alpha[\varphi, ...\varphi] | //\varphi$$

where α is an attribute formula, φ is a tree pattern. From this expression we can see that φ could be the children ($\alpha[\varphi, \varphi...]$) or descendants ($//\varphi$) of an element α.

$$\begin{aligned} (x,y) = r[book(@title = x)[author[name(@nam = y)]]] \\ \rightarrow (x,y) = r[writer[name(@nam = y), work(@title = x)]]; \end{aligned} \tag{2.1}$$

Considering the XML document in Figure 2.3, for the left side ($(x,y) = r[book(@title = x)[author[name(@nam = y)]]]$) of the mapping 2.1, we can find the tuples $(AlgorithmDesign, Tardos)$, $(AlgorithmDesign, Kleinberg)$ and $(Algebra, Hungerford)$. Hence according to mapping 2.1, the target instance is as Figure 2.4.

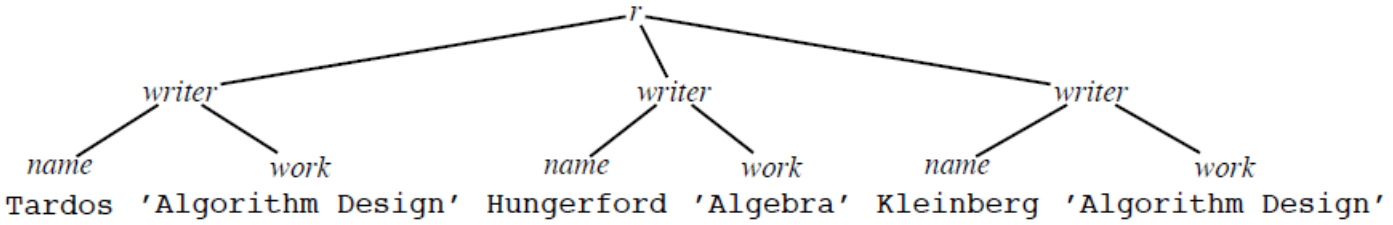


Figure 2.4: Target XML document tree

2.1.2.4 XQuery

XQuery is to XML what SQL is to database tables [3]. It is the language for XML files to query XML data. A tree pattern can be rewritten to an XQuery. For example, a query on XML in figure 2.4 in form of tree pattern:

$$Q(x) = r[writer[name(@nam = x)]] \tag{2.2}$$

can be translated to the XQuery:

$$doc(`writer') / r / writer / name / nam \tag{2.3}$$

2.2 Nested-relational DTD

2.2.1 Nested Relations

Project		
Manager	Detail	
	P_Name	Budget(K)
Joe	P1	40
	P2	30
Sue	P2	30
	P3	20
	P4	30

Figure 2.5: Nested Relation (from [18])

Figure 2.5 is an example of nested relations. From this relation we can see that the 'Detail' relation is a sub-relation of the relation 'Project'. The relation 'Project' has two tuples: the first one with manager Joe; and the second one with manager Sue. Meanwhile, each tuple of them contains several sub-tuples: the tuple with Joe has two tuples, and the tuple with Sue has three tuples.

2.2.2 Nested-relational DTD

Since nested relations can easily be represented by XML documents, we can take this advantage by extending the concept of nested relations and defined the notion of nested-relational DTDs [6]. A nested relational DTD is a DTD that is non-recursive and all rules inside which follow the form:

$$l \rightarrow \hat{l}_0, ...\hat{l}_m$$

where all $l_i's$ are different elements and every $\hat{l}_i$ is one of the following: $\hat{l}_i$, or $\hat{l}_i*$, or $\hat{l}_i+$, or $\hat{l}_i?$ [6].

2.3 Conjunctive tree query

In the relational database theory, there is a class of queries with good properties — the conjunctive queries [12]. Similarly, in the XML scenario, there is a class of XML queries named Conjunctive Tree Queries (CTQ) [16] [9].

A conjunctive tree query should follow this format:

$$Q = \varphi | Q \wedge Q | \exists x Q$$

where φ is a tree pattern as introduced in section 2.1.2.3.

There are two classes of conjunctive tree queries. One is the queries that output tuples of attribute values [7] [8]. Another one is the queries that output XML trees. For example, if we consider the XML tree in Figure 2.4, when we query the names of writers, the results should be three tuples in table 2.3; when we query the patterns that writers with Name and Work, the answer should be three trees in Figure 2.6.

Tardos
Hungerford
Kleinberg

Table 2.3: Answer: Attribute values

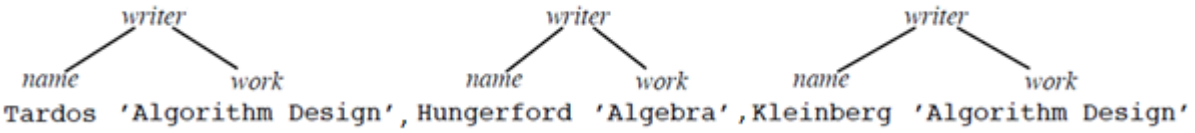


Figure 2.6: Answer: XML trees

2.4 ++Spicy

++Spicy is a mapping system developed at the Universit della Basilicata - Department of Mathematics and Computer Science for the generation and manipulation of schema mappings [2]. It can be used to solve both the relational and XML data exchange problems.

To solve the data exchange problem using ++Spicy, we should configure the source and target database (or XML files) and the mappings. Then in the relational scenario, the ++spicy will generate SQL script; in the XML scenario, the ++spicy will produce XQuery script. Finally, we can execute those scripts either on the ++spicy itself or other external engines such as PostgreSQL or BaseX [1] to get the target instances.

Chapter 3

Background

3.1 Related Work

The problems of data exchange vary according to the types of data model. Most works are involved in the relational model. Several basic notions such as universal solutions, certain answers and their computing complexity are discussed in [15]. A comprehensive overview of data exchange is presented in the survey [20].

However, only in last few years have people come to focus on the XML model. The first paper that formally proposed the problem of XML data exchange and investigated its basic properties is [6]. Then the issues about XML schema mappings are studied in [5] [4]. These work explored the theoretic basis of XML data exchange. On the other hand, some practical works on the algorithm have been done. An XML database system named TIMBER was implemented at the University of Michigan [19]. In addition, other general naive schema mapping system like Clio [23], HePToX [10], and Spicy [11] are all concerned with solving the data exchange in the XML scenario. Their schema mapping algorithms rely on the form 'correspondences' — 'matching's between pairs of source and target attributes. The paper [13] proposed another way to handle XML data exchange. That is using the Shredding [21] and Inlining [17] techniques to convert the XML data exchange problems to the Relational data exchange problems. The correctness of this approach is proved in [13]. Recently the new generation of the system Spicy adopted this idea and combined other methods from [24] [6].

3.2 Assumption of the work

3.2.1 Restriction

3.2.1.1 Well-formed XML format

The input XML files for the XML data exchange we discuss in this project should be well formed. That is, all PCDATA instances are attributes. For example, take the XML document in Figure 2.3, the data 'Algorithm Design' is stored in the attribute of Book element as following:

$$< booktitle = `AlgorithmDesign' >< /book >$$

3.2.1.2 Nested-relational DTD

The DTDs we process in this project are nested-relational DTDs. The reason for this is that if the DTDs in the XML data exchange setting are not nested-relational DTDs, the problem will become coNP-hardness of answering conjunctive queries [6].

3.2.1.3 Conjunctive Tree Query returning attribute values

As mentioned in section 2.3, there are two classes of Conjunctive Tree Queries. In this paper, we only focus on the queries that return the attribute values.

3.2.1.4 Features of Tree patterns

In this project, the tree patterns in the mappings should not use one of the three features:

- The descendant relation
- The wild card
- Patterns that do not start at root

If the tree patterns contain any of above features, the query answering could be coNP-hard [6].

3.2.2 Tools

The programming language: Python The XML processing engine: BaseX [1] The Relational Database Management Engine: PostgreSQL

3.3 Contributions

The main contributions of our work are:

- Implementing the algorithms of ‘XML data exchange via relations’;
- Comparing this method with another method on ++spicy;
- Analysis on the results of the experiments.

Chapter 4

Algorithms and Implementation

The relationship between relational data exchange and XML data exchange can be explained as this graph:

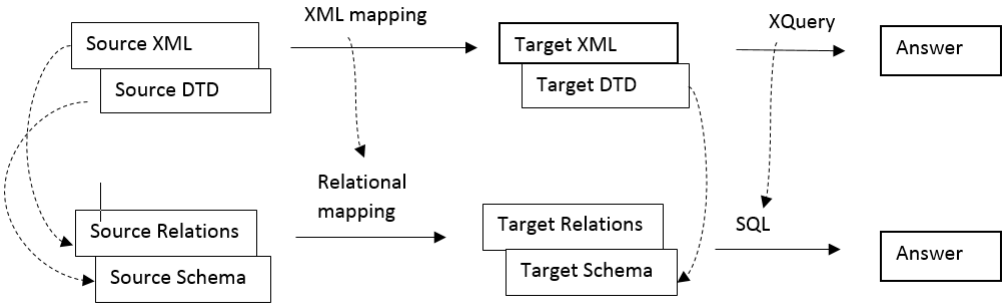


Figure 4.1: Problem Translation

This graph illustrates the translations from the relational data exchange to the XML one. Each dash line stands for a translation. Overall, this procedure involves four algorithms:

- Translation of DTDs
- Translation of XML documents
- Translation of Queries
- Translation of Mappings

4.1 Algorithm 1: Translating DTDs

The key point to translate a DTD to a schema of several relational tables is to decide how many tables we need and what is their attributes. Hence our algorithm aims at overcoming these two difficulties: 1) when should we create a table; 2) what should its attributes be like.

4.1.1 Creating Relations

The idea is enlightened by the paper [17]. The element with a star symbol in the DTD file declares zero of more occurrences of an element. Now we extend this notion a bit more. We convert other elements with non-start symbols to elements with a star. The rules of converting including following steps (examples from [17]):

a. Convert all + operators to *

b. Separate the nested definitions to several plain definitions. The details of converting are as following:

c. Reduce multiple symbols to a single symbol.

d. The sub-elements sharing the same name should be merged to one.

After converting, we create a relation for each element with a star symbol. Besides, we create a relation for the root element although it has no star. Consider the source DTD in Figure 2.1. The elements with a star are: ‘book’, ‘author’. The root is ‘r’. Hence the schema contains three relations: ‘book’, ‘author’, ‘r’.

4.1.2 Adding attributes

When we decided to create a relation for an element *l*, we should define the attributes of that relation. Five types of attributes should be considered:

1. An ID attribute named by the element l itself. It is the primary key of the relation. For example, in Figure 2.1, we want to create a relation 'book' for the element 'book'. Attribute 'bookID' should be added into the relation.

2. The attributes named by the XML attributes of the element l. In Figure 2.1, element 'book' has an XML attribute $@title$, so the attribute 'ATTtitle' should be added into the relation.

3. The ID attribute named by the 'nearest appropriate ancestor' of the element l. The 'nearest appropriate ancestor' of the element l is the closest element with a star in the path from the root element to the element l. In Figure 2.1, the 'nearest appropriate ancestor' of element 'book' is the element 'r'. Then the attribute 'rID' should be added into the relation 'book'.

4. The ID attribute named by a particular set of children of the element l. Each child element l' in this set should satisfy this requirement: l' is not with a star and the element l is the 'nearest appropriate ancestor' of l'. In Figure 2.1, the element 'book' is the 'nearest appropriate ancestor' of the element 'subject' and 'subject' is not an element with a star. So the attribute 'subjectID' should be added into the relation 'book'.

5. The attributes named by the XML attributes of each element satisfying rule d. In Figure 2.1, 'subject' has an attribute 'sub'. Hence we add the attribute 'ATTsub' into the relation 'book'.

Following above rules, we can translate the source DTD in Figure 2.1 to a schema of three relations:

$$r(rID)$$

$$book(bookID, ATTtitle, rID, subjectID, ATTsub)$$

$$author(authorID, bookID, nameID, ATTnam, affID, ATTaff)$$

4.1.3 Implementation

Our implementation of this algorithm is as following:

1. Read the DTD file and store elements as a set of 'Node'. The 'Node' is a customized class with four member variables (Figure 4.2).

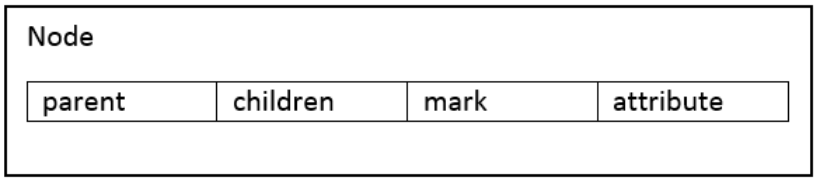


Figure 4.2: Node

The information of the DTD can be expressed properly by this 'Node' data structure. The 'mark' marks the elements with a star. The 'parent' and 'children' variables can represent the relationship among elements and can be used to identify the 'neared appropriate ancestors' by the 'mark' variable.

2. Traverse all Nodes with a star. For each Node, find all attributes belonging to its relation and output the definition of this relation.

4.1.3.1 Difficulties

1. The whole program for this algorithm is quite straightforward except for the codes handling the rule 4 and 5 in section 4.1.2. This is a recursive procedure to: first, find the children without a star; second, look for their XML attributes.

Algorithm 1 Finding Appropriate Children

```
1: function VISIT(child,relation)
2:     if node[child] = Marked then
3:         relation ← node[child] + 'ID'.          ▷ add child+'ID' to relation
4:         relation ← node[child].attributes.
5:         for i ∈ node[child].attributes do VISIT(i)
6: procedure VISITCHILDREN(marked)      ▷ find attributes satisfying rule 4 and 5
7:     for child ∈ node[marked].children do
8:         VISIT(child)
```

2. The translation of target DTDs is slightly different from that of source DTDs. When we generate the target schema, we should eliminate all key constraints, because the target instance is usually incomplete.

4.2 Algorithm 2: Translating XML documents

The procedure of translating XML documents is close to the procedure of translating DTDs. The main problem of translating XML documents is when and how to form a tuple for a relation.

4.2.1 Form a tuple

To form a tuple, we should use the concept of element with a star again. In the context of XML documents, the node with a star means a node in the XML tree that is defined as an element with a star in the DTD file. Take the XML document in Figure 2.3 and the DTD from Figure 2.1. There are two book nodes: Algorithm Design and Algebra. Since the element book has a star in the DTD, two tuples should be created for the nodes Algorithm Design and Algebra.

4.2.2 Attributes of a tuple

When we decide to create a tuple, the key problem is to define the values of the attributes of this tuple. First of all, we should assign a unique number to each node of the XML tree. For example, in Figure 2.3, node 'r' is 1, node Algorithm Design is 2, and Algebra is 3, and so on.

We can define the values of the attributes of a tuple for a node ω by this way:

1. The value of the ID attribute '$\omega ID'$ is the number we assigned before. For the node 'Algorithm Design' in Figure 2.3, we should insert a tuple into the relation 'book'. The value of the attribute 'bookID' of this tuple is '2'.

2. Insert the values of $\omega's$ attributes. For instance, for the node 'book', the value of $@title$ is 'Algorithm Design'. This is the value of the attribute 'ATTtitle' of the tuple.

3. Insert the ID of $\omega's$ 'nearest appropriate ancestor'. For the node 'book', node 'r' is its 'nearest appropriate ancestor', so the value of attribute 'rID' of the tuple is '1'.

4. Insert the IDs of the proper children nodes of the node ω. The node ω should be the 'nearest appropriate ancestor' these children nodes, and they are not nodes with a star. For example, the 'node 4' is a child of the 'node 2' ('Algorithm Design'), but 'node 4' is a node with a star, so it is not a proper child of 'node 2' and cannot be used to define the tuple. The 'node 6', a node of 'subject' type, is a proper child of the node 'Algorithm Design', so we should add value '6' of attribute 'subjectID' to the tuple.

5. Insert the values of the proper childrens XML attributes. In the Figure 2.3,The 'node 6' has an attribute $@sub$ with a XML attribute 'CS', so the value of attribute 'ATTsub' of the tuple is 'CS'.

Following these rules, for the 'node 2' in Figure 2.3, we should insert the tuple $('2', 'AlgorithmDesign', '1', '6', 'CS')$ into the relation $book(bookID, ATTtitle, rID, subjectID, ATTsub)$. For the entire XML document in the Figure 2.3, we can get relations as following:

rID
1

Table 4.1: Relation r

bookID	ATTtile	rID	subjectID	ATTsub
2	Algorithm Design	1	6	CS
3	Algebra	1	8	Math

Table 4.2: Relation book

authorID	bookID	nameID	ATTname	affID	ATTaff
4	2	9	Kleinberg	10	CU
5	2	11	Tardos	12	CU
7	3	13	Hungerford	14	SLU

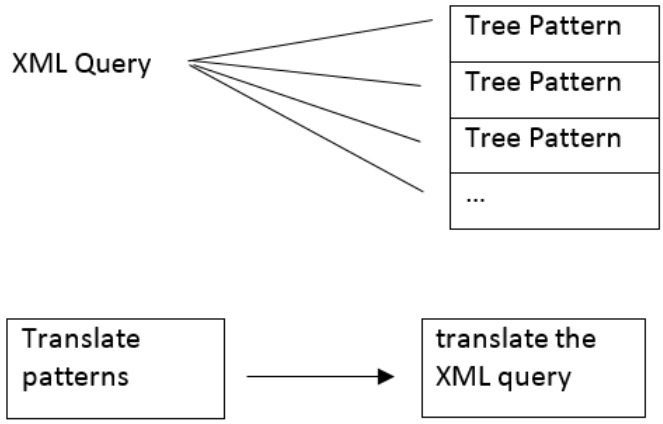
Table 4.3: Relation author

4.2.3 Implementation

We slightly adjust the 'Node' structure in Figure 4.2. In the context of XML document, we do not use 'Node' to store the DTD elements. Instead, we store the nodes in the XML document into the 'Node' structure. A 'Node' contains four member variables: 'parent', 'children', 'mark', and 'attributes'. They are used in the same way as mentioned section 4.1.3.

4.3 Algorithm 3: Translating Query

An XML query consists of several tree patterns. If we can translate these tree patterns, the translation of the entire query will be the combination of the translations of tree patterns.

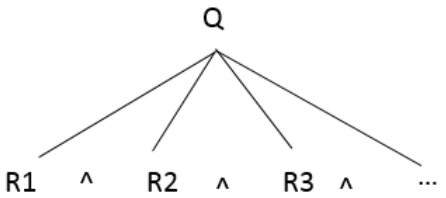


4.3.1 Translate Tree Pattern

In the relational scenario, a conjunctive query is formed by a join of several 'single relation query's. A 'single relation query' is constructed by projection and selection on a single relation. To find the target relational conjunctive query, the main task is to find each 'single relation query'. For example,

$$Q(x,y) = R(x,z) \wedge S(z,y)$$

This query contains two single relations: R and S. If we find those two, we can easily build this query.



Now we introduce the method to find each single relation in the target query. First we regard a tree pattern of a query as a tree. The pattern in the left side of mapping 2.1 can be represented as the following tree:

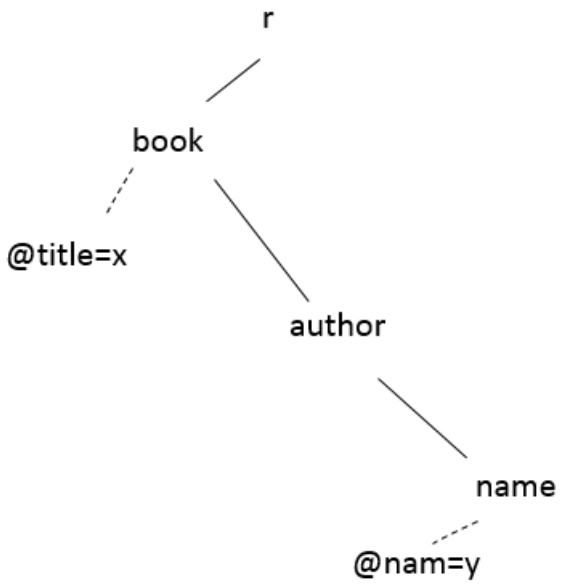


Figure 4.3: Tree Pattern: $(x,y) = r[book(@title = x)[author[name(@nam = y)]]]$

There are a set of nodes *lxv* in which each node has an attribute with a variable. We call this set 'Projection Nodes'. For example, in Figure 4.3, the node 'book' has the attribute '@title' with the variable 'x' and the node 'name' has the attribute '@nam' with the variable 'y'. So the set *lxv* is '$book'$, '$name'$. After collecting all nodes of *lxv*, for each node *v* in *lxv*, we can produce a 'single relation query' for itself or its 'nearest appropriate ancestor'.

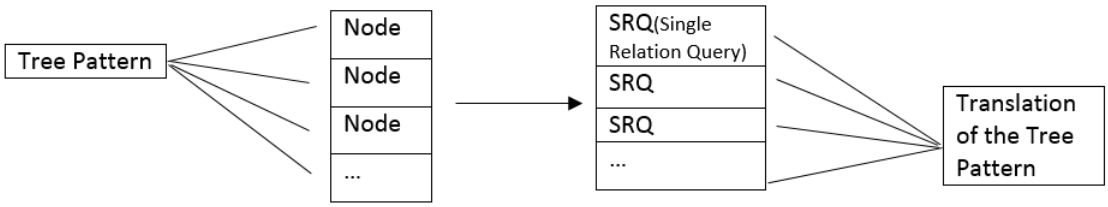


Figure 4.4: translate tree patterns

The attributes of a single relation for node v is assigned variables by this way:

1. Attributes that are assigned variables in the tree pattern keep the variables unchanged. For example, in Figure 4.3, for the node 'book', the attribute $@title$ is assigned a variable 'x', thus the 'ATTtitle' of the relation 'book' should be assigned the variable 'x'.
2. Assign the ID attribute of v a variable with the same name of itself. In Figure 4.3, the ID attribute 'bookID' is assigned a variable 'bookID'.
3. Assign the ID attribute of $v's$ 'nearest appropriate ancestor' a variable with the same name of itself. In Figure 4.3, the node 'book's 'nearest appropriate ancestor' is the node 'r', then the attribute 'rID' of the relation 'book' is assigned a variable 'rID'.
4. Name the rest attributes with a set of distinct variables $\bar{z}$. In Figure 4.3, the rest attributes of the relation 'book' are 'subjectID' and 'ATTsub'. Hence we assign them variables 'z_0' and 'z_1'.

Overall, the tree pattern $r[book(@title = x)[author[name(@nam = y)]]]$ can be translated to this query:

$$Q(x,y) : -book(bookID, x, rID, z_0, z_1), author(authorID, bookID, z_2, z_3, nameID, y)$$

This expression represents the following SQL:

Select t1.ATTtitle, t2.ATTnam
From book t1, author t2
Where t1.bookID=t2.bookID

Notice that is if a node in lxv is not with a star, we produce a 'single relation query' for its 'nearest appropriate ancestor'. For example, in Figure 4.3, the node 'name' is not with a star, then we produce a 'single relation query' for its 'nearest appropriate ancestor' — 'author'.

4.3.2 Implementation

We firstly read the DTD file related to the query and create a set of nodes to store the information of the DTD file. Then we use another node set to store the tree pattern. By

referring the data stored in the two sets of nodes, we can get all information we need to create the target query.

4.4 Algorithm 4: Translating Mapping

An XML mapping is composed of two queries over the source DTD and the target DTD respectively. So the implementation is just applying the Algorithm 3 to the queries on the both sides of the mapping.

Chapter 5

Evaluation

5.1 Environment

The hardware environment on which we run experiments:

- Operation system: Windows 8.1 64bit
- Memory: 4GB
- CPU: i5 3337U 1.8GHz

The software involved:

- XML database management system: BaseX [1]
- Relational database management system: PostgreSQL

The original input data come from the XML mapping examples on the official website of ++spicy [2]. The lager XML dataset is created by inserting more nodes with distinct data.

5.2 Procedure of experiments

The ++spicy system will procedure SQL script and XQuery script for relational and XML data exchange respectively. If we run those scripts on the appropriate database management systems, the target instances will be generated. In order to evaluating the performance of these two methods, we recorded the time they took to produce the target instance.

In addition, since the source XML documents are already existing but the source relations should be built before data exchanging, we should add this building time to the relational data exchanging time.

At last, as the final goal of data exchange is query answering, we also designed a set of queries on the target instances and recorded the time it takes to answer those queries.

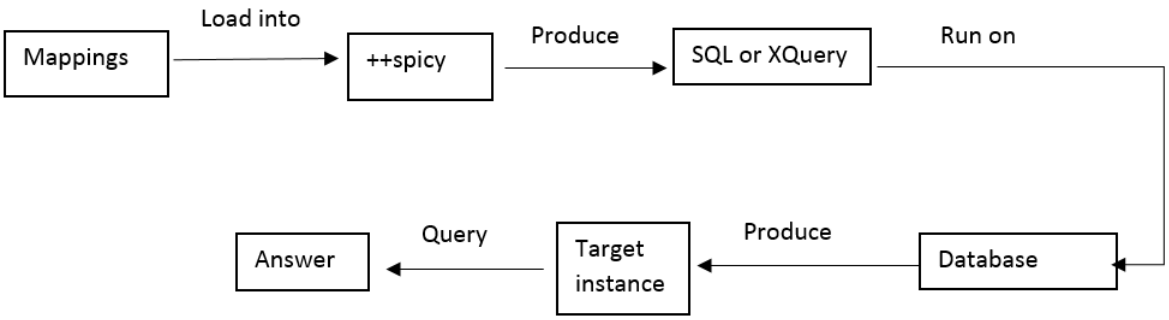


Figure 5.1: stages of experiments

5.3 Results

Figure 5.2 shows the results of experiments on the tiny data.

Content and Size	XML	Relation
Book Publisher/ 15 tuples		Build DB: 214ms
	Generate Target: 51.68ms	Generate Target: 245ms
	Answer Query: 1.44ms	Answer Query: 16ms
Cities & Cars/ 22 tuples		Build DB: 267ms
	Generate Target: 168.62ms	Generate Target: 341ms
	Answer Query: 1.67ms	Answer Query: 11ms
Cities & Names/ 14 tuples		Build DB: 239ms
	Generate Target: 73.84ms	Generate Target: 259ms
	Answer Query: 1.35ms	Answer Query: 32ms
Writer/ 10 tuples		Build DB: 158ms
	Generate Target: 18.44ms	Generate Target: 95ms
	Answer Query: 1.27ms	Answer Query: 16ms

Figure 5.2: Experiments on tiny cases

Figure 5.3 shows the results of experiments on the large data (data based on the livesIn example from [2] and tuples are multiplied by the generator).

Size	XML	Relation
1000 tuples	Generate Target: 1060.83ms	Build DB: 317ms
		Generate Target: 219ms
	Answer Query: 11.34ms	Answer Query: 13ms
5000 tuples	Generate Target: 23859.4ms	Build DB: 1102ms
		Generate Target: 830ms
	Answer Query: 21.79ms	Answer Query: 109ms
10000 tuples	Generate Target: 106541.0ms	Build DB: 1736ms
		Generate Target: 1279ms
	Answer Query: 12.66ms	Answer Query: 136ms
20000 tuples	Generate Target: 474928.75ms	Build DB: 3261ms
		Generate Target: 2509ms
	Answer Query: 25.93ms	Answer Query: 251ms

Figure 5.3: Experiments on large cases

5.4 Analysis

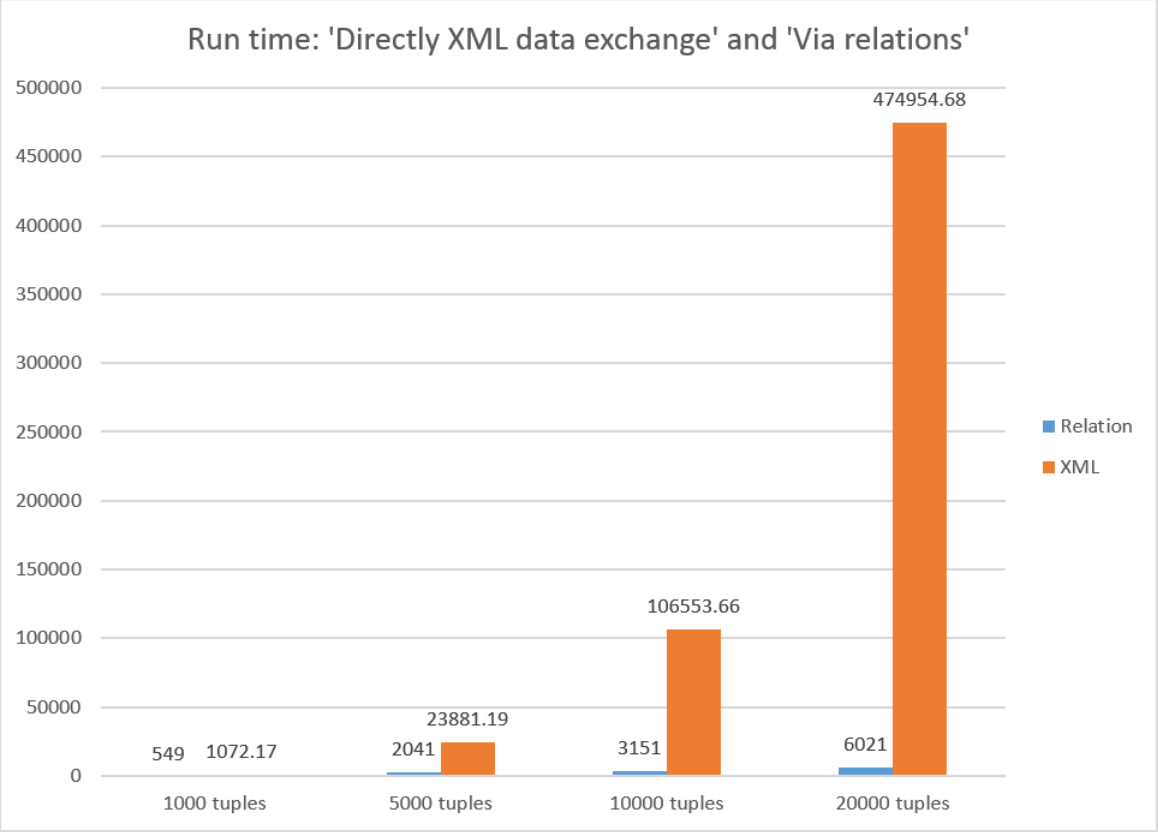


Figure 5.4: 'Directly XML data exchange' and 'Via relations'

According to the study from [13], the performance of the approach 'XML data exchange via relations' is expected to be better than the naive approach of XML data exchange.

From the results of our experiments, in the tiny cases, the naive implementation by ++spicy seems faster than our approach. Nevertheless, this superiority of the 'direct method' is less useful because: 1) in real life, the dataset is rarely as small as that; 2) even for the 'via relations' method with longer run time, the exchange time is still under one second.

On the other hand, the large cases are more meaningful since it is closer to real life and the difference of performance between these two methods is huge. From the Figure 5.4 we can see that in the 1000-tuple case the run time of 'directly XML data exchange' is twice as long as that of our approach. The gap between these two methods increases rapidly with the number of tuples we process. In the 20000-tuple case, the 'via relation' approach is about 70 times faster than the 'direct method'.

Overall, we can say that the approach 'XML data exchange via relations' is as efficient as we expect, and this method is faster than the XML data exchange method implemented by the system ++spicy.

Chapter 6

Conclusion

6.1 Observation and remarks

In this project we successfully implemented the approach 'XML data exchange via relations'. The experiments show that this implementation actually works well and this method is far better than the 'directly XML data exchange' approach implemented by the system ++spicy.

'XML data exchange via relations' takes the advantage of well-studied relational database techniques and provide a practical way to handle the problem of XML data exchange. The mature optimization techniques of relational database model guarantee the superiority of our approach when processing large dataset.

6.2 Unsolved problems

Our implementation is an incomplete version of the approach. The implementation of the algorithm 'translating XML query' only considered the XML queries that return attribute values. Hence the algorithm for the 'XML-to-XML' queries [13] remains unsolved.

6.3 Future work

There are still a host of work worthy to be done in the future:

- It is needed to implement the algorithm for the 'XML-to-XML' queries and do experiments on it.
- XML schema becomes a popular replacement of DTD, we should adjust these algorithms and make them handle the XML schema.

- The syntax of XML mappings in the system ++spicy is complicated and can be improved to be more expressive.

Bibliography

[1] Basex. http://basex.org/. Accessed Aug 9, 2015.

[2] ++spicy. http://www.db.unibas.it/projects/spicy/. Accessed Aug 9, 2015.

[3] w3school. http://www.w3schools.com/. Accessed Aug 9, 2015.

[4] Shun'ichi Amano, Claire David, Leonid Libkin, and Filip Murlak. On the tradeoff between mapping and querying power in xml data exchange. In *Proceedings of the 13th International Conference on Database Theory*, pages 155–164. ACM, 2010.

[5] Shun'ichi Amano, Leonid Libkin, and Filip Murlak. Xml schema mappings. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 33–42. ACM, 2009.

[6] Marcelo Arenas and Leonid Libkin. Xml data exchange: consistency and query answering. *Journal of the ACM (JACM)*, 55(2):7, 2008.

[7] Andrey Balmin and Yannis Papakonstantinou. Storing and querying xml data using denormalized relational databases. *The VLDB JournalThe International Journal on Very Large Data Bases*, 14(1):30–49, 2005.

[8] Pablo Barceló. Logical foundations of relational data exchange. *ACM SIGMOD Record*, 38(1):49–58, 2009.

[9] Henrik Björklund, Wim Martens, and Thomas Schwentick. Conjunctive query containment over trees. In *Database Programming Languages*, pages 66–80. Springer, 2007.

[10] Angela Bonifati, Elaine Chang, Terence Ho, Laks VS Lakshmanan, Rachel Pottinger, and Yongik Chung. Schema mapping and query translation in heterogeneous p2p xml databases. *The VLDB Journal*, 19(2):231–256, 2010.

[11] Angela Bonifati, Giansalvatore Mecca, Alessandro Pappalardo, Salvatore Raunich, and Gianvito Summa. Schema mapping verification: the spicy way. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 85–96. ACM, 2008.

[12] Ashok K Chandra and Philip M Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90. ACM, 1977.

[13] Rada Chirkova, Leonid Libkin, and Juan L Reutter. Tractable xml data exchange via relations. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1629–1638. ACM, 2011.

[14] Claire David, Leonid Libkin, and Filip Murlak. Certain answers for xml queries. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 191–202. ACM, 2010.

[15] Ronald Fagin, Phokion G Kolaitis, Renée J Miller, and Lucian Popa. Data exchange: Semantics and query answering. In *Database TheoryICDT 2003*, pages 207–224. Springer, 2003.

[16] Georg Gottlob, Christoph Koch, and Klaus U Schulz. Conjunctive queries over trees. *Journal of the ACM (JACM)*, 53(2):238–272, 2006.

[17] Jayavel Shanmugasundaram Kristin Tufte Gang He and Chun Zhang David DeWitt Jeffrey Naughton. Relational databases for querying xml documents: Limitations and opportunities. 2008.

[18] HE Hongbo. *Implementation of Nested Relations in a Database Programming Language*. PhD thesis, McGill University, Montreal, 1997.

[19] Hosagrahar V Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks VS Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, et al. Timber: A native xml database. *The VLDB JournalThe International Journal on Very Large Data Bases*, 11(4):274–291, 2002.

[20] Phokion G Kolaitis. Schema mappings, data exchange, and metadata management. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 61–75. ACM, 2005.

[21] Rajasekar Krishnamurthy, Raghav Kaushik, and Jeffrey F Naughton. Xml-to-sql query translation literature: The state of the art and open problems. In *Database and XML Technologies*, pages 1–18. Springer, 2003.

[22] Bruno Marnette, Giansalvatore Mecca, Paolo Papotti, Salvatore Raunich, Donatello Santoro, et al. ++ spicy: an open-source tool for second-generation schema mapping and data exchange. *Clio*, 19:21, 2011.

[23] Lucian Popa, Yannis Velegrakis, Mauricio A Hernández, Renée J Miller, and Ronald Fagin. Translating web data. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 598–609. VLDB Endowment, 2002.

[24] Mark A Roth, Herry F Korth, and Abraham Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems (TODS)*, 13(4):389–417, 1988.