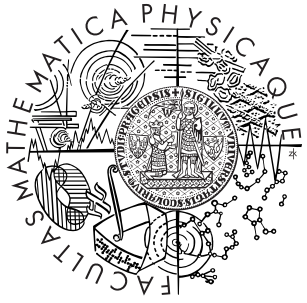
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Luboš Kulič

Adaptability in XML-to-Relational Mapping Strategies

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Irena Mlýnková, Ph.D.
Studijní program: Informatika

2009

Na tomto místě bych rád poděkoval vedoucí mé práce RNDr. Ireně Mlýnkové, Ph.D. za zajímavé téma, mnoho informací, které mi pomohly začít s jeho řešením a v neposlední řadě za bezpočet připomínek k textu práce i algoritmům, které napomohly dostat práci do finálního stavu.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 17. 4. 2009 Luboš Kulič

Contents

Název práce: Adaptability in XML-to-Relational Mapping Strategies
Autor: Luboš Kulič
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Irena Mlýnková, Ph.D.
e-mail vedoucího: irena.mlynkova@mff.cuni.cz
Abstrakt:
Jednou z možností jak pracovat s XML dokumenty je využití (objektově-)relačních databází. Nejdůležitějším úkolem je v tomto případě nalezení optimálního mapování mezi XML a databází, tedy způsobu jak ukládat XML data do relací. Aktuálně nejefektivnější řešení, tzv. adaptivní metody, prohledávají prostor možných řešení a vybírají to, které nejlépe vyhovuje příkladům dokumentů a dotazů. V této práci řešíme problém mapování XML do relací pomocí heuristiky zvané Optimalizace pomocí kolonií mravenců (ACO). Navržený algoritmus jsme také adaptovali pro použití v dynamické verzi problému. Vlastnosti obou algoritmů jsou ověřeny v řadě experimentů, ze kterých vyplývá, že algoritmy založené na ACO jsou nejen vhodné pro daný problém, ale umožňují i řešení jeho dynamické verze.
Klíčová slova: XML, ukládání XML v databázích, XML Schema, dynamické adaptivní metody

Title: Adaptability in XML-to-Relational Mapping Strategies
Author: Luboš Kulič
Department: Department of Software Engineering
Supervisor: RNDr. Irena Mlýnková, Ph.D.
Supervisor's e-mail address: irena.mlynkova@mff.cuni.cz
Abstract:
One of the ways how to manage XML documents is to exploit tools and functions offered by (object-)relational database systems. The key aim of such techniques is to find the optimal mapping strategy, i.e. the way the XML data are stored into relations. Currently the most efficient approaches, so-called adaptive methods, search a space of possible mappings and choose the one which suits the given sample data and query workload the most. In the thesis we exploit a general heuristic method called Ant Colony Optimization (ACO) to solve the XML-to-Relational mapping problem. We also adapt the algorithm so it can be used on a dynamic variant of the problem. The algorithms are evaluated in a set of experiments with a conclusion that the ACO-based algorithms are suitable for the problem and can be even used as a basis of a dynamic mapping mechanism.
Keywords: XML, storing XML in databases, XML Schema, dynamic adaptation

Chapter 1

Introduction

Since its birth, XML[1] has become a frequently used format for representing, exchanging and manipulating data both in traditional applications and on the Internet. Because of that, there is naturally a need for efficient and reliable methods for managing and storing XML data. Many solutions of this task have been proposed using different approaches.

In this thesis we first review the most important methods and their advantages and disadvantages. Since the most usable are in our opinion the methods using (Object-)Relational Database Management Systems (simply because relational databases have a long practical and theoretical history and Relational Database Management Systems are wide spread), we focus mainly on them.

Of course, when using a relational database to store XML documents a mapping from the XML definition to a relation one has to be found. This can be accomplished using various methods, the most promising ones are so called *adaptive* (or *flexible*) techniques. These methods exploit various information about the target application (such as a sample set of XML documents or queries) to create a schema most suitable for it.

The main goal of this thesis is to propose an adaptive mapping selection algorithm which would address selected drawbacks of the existing methods. This solution will be then evaluated by a set of experiments.

[1] standardized in [W3C: XML]

1.1 Thesis organization

The rest of the thesis is organized as follows: Chapter 2 summarizes used technologies and terms.

Chapter 3 introduces the problem of XML-to-Relational mapping and gives a motivation for the thesis. The problem is then further analyzed in Chapter 4.

Chapter 5 describes the proposed algorithm for solving the XML-to-Relational mapping problem. Chapter 6 then discusses the possibility to adapt the proposed algorithm to the dynamic variant of the problem.

In Chapter 7 we summarize the most important implementation-related topics.

Our experiments with the proposed algorithms are described in Chapter 8.

Finally, Chapter 9 concludes the thesis and gives an outlook on future work.

Chapter 2

Background Technologies

In this chapter we provide a brief description of the technologies used in the rest of the thesis – especially XML, DTD, XML Schema, XPath and the Relational model. It serves primary as a summary of used terms and definitions, the reader is given references to detailed information on the topics in corresponding sections. Note that we simplified the description in some cases on purpose – we did not want to bother the reader with specific parts of the technologies, which are not used in the rest of the thesis anyway.

2.1 XML

XML (Extensible Markup Language), standardized in [W3C: XML], is a general-purpose markup metalanguage, i.e. a specification for creating custom languages including custom mark-up elements. The main components of the XML language are *elements* – they are denoted by a mark (the name of the element, called also tag) in angle-brackets, the element could be either empty or have some content (called children) – other elements (subelements), attributes, textual data and some others. An example of an XML fragment with an element called `item` is given in Figure 2.1a. It has two attributes (`id` and `featured`) and four child elements (`location`, `quantity`, `name` and `incategory`).

When a document fulfills some basic syntactic rules (the starting and ending parts of its elements are always properly paired and do not cross other elements'

```
<item id="item2"
   featured="yes">
  <location>
     United States
  </location>
  <quantity>1</quantity>
  <name>subtle</name>
  <incategory
   category="category1"/>
  <incategory
   category="category2"/>
</item>
```

(a) `item` XML Element

```
<!ELEMENT item (location, quantity,
name, incategory+)>
<!ATTLIST item id ID #REQUIRED
featured CDATA #IMPLIED>
<!ELEMENT location (#PCDATA)>
<!ELEMENT quantity(#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT incategory EMPTY>
```

(b) The DTD for `item`

Figure 2.1: An XML Element with its schema in DTD

```
<xs:complexType name="item">
  <xs:sequence>
    <xs:element name="location" type="xs:string"/>
    <xs:element name="quantity" type="xs:string"/>
    <xs:element name="name" type="xs:string"/>
    <xs:element maxOccurs="unbounded" name="incategory" type="incategory"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:ID" use="required"/>
  <xs:attribute name="featured" type="xs:string"/>
</xs:complexType>
```

Figure 2.2: The XML Schema definition for `item`

definitions, the whole document is enclosed by exactly one *root* element etc.), it is called a *well-formed* XML document.

It is also possible to further restrict the structure of a set of documents by a *schema* for them. A schema is a set of constraints on both the structure and textual content of an XML document. It can be defined in various languages, probably the two most common are DTD and XML Schema (see Section 2.2 resp. 2.3). If a document fulfills these constraints, it is *valid* against this schema.

2.2 DTD

DTD (Document Type Definition), defined directly in [W3C: XML], is a method of declaring a structure of XML documents. It uses element declarations which specify the allowed set of elements in a valid document together with the specification whether and how many times a particular element can be contained in the other ones. Another declaration is an attribute-list which specifies the set of allowed attributes for every element including a type of such attribute. An example of a DTD fragment for the element `item` is given in Figure 2.1b.

2.3 XML Schema

XML Schema [W3C: XML Schema] is a much more sophisticated language for defining schema of XML documents (such a definition of a schema in XML Schema language is called XML Schema Definition, or XSD). Besides declarations of elements, their subelements and attributes it allows to define quite a complex type system. The types can be derived from each other and reused (shared) by many elements. The following sections summarize the most important components of XML Schema used in this thesis. An example of an XSD fragment for the element `item` is given in Figure 2.2.

Note that the syntax of XSD is based on XML, i.e. we use XML elements with proper tags to define elements (...) in the target XML document. Every XSD must have exactly one root element called `schema` which cannot appear in the XSD on any other position.

```
<auction:item xmlns:loc="www.locations.org/loclist"
       xmlns:name="www.namedirectory.org/names"
       xmlns:auction="www.sothebys.com/auctions"
       id="item2" featured="yes">
  <loc:location>United States</location>
  <auction:quantity>1</quantity>
  <name:name>subtle</name>
  <auction:incategory category="category1"/>
  <auction:incategory category="category2"/>
</item>
```

Figure 2.3: XML Element with Namespaces

2.3.1 Namespaces

A *namespace* is a space where the tags of all contained elements (and in the context of one element also the names of the attributes) are unique. Namespaces enable to define an XML Schema using sets of marks. An example of an XML fragment with the `item` element rewritten to use namespaces is given in Figure 2.3.

2.3.2 Elements

Elements are the basic components of an XML document, in XML Schema they are defined using element `xs:element`. They associate a tag (given in the `name` attribute of `xs:element`) with a data type (see Section 2.3.4). There are basically two types of elements:

- *Global* or *globally defined elements* are defined as children of the root element `schema` and are visible in the whole XSD.
- *Local* or *locally defined elements* are defined as a part of a definition of some complex type (see Section 2.3.4) and can be used only in its context.

The global elements have two functions besides the definition of a structure of the document. They can be used as root elements of an XML document valid against

this XSD and they can be referenced in the XSD. A reference means a definition of a (local) element which does not contain any tag name or type but only a `ref` attribute which tells a global element definition should be used in this place.

2.3.3 Attributes

Attributes are defined in special subelements of the element definition (see Figure 2.2). They again associate a name with a type, in this case only a simple one.

2.3.4 Data Types

Data types are used in XML Schema to define a structure of the XML Document as well as a format of the textual values. They can be classified according to two main aspects – what they define (see Section 2.3.4) and where in the schema they are defined (see Section 2.3.4).

Simple and Complex Types

A *Simple type* is a set of constraints on a textual value, it can only contain some text and not any elements or attributes. A simple type can be either *built-in* (i.e. defined in the XML Schema specification [W3C: XML Schema], for example `xs:string`, `xs:integer`, ...) or user-defined. The user-defined types are derived from another simple type (either built-in or another user-defined one) by:

- *Restriction* – restricting the values, for example minimum/maximum length, allowed characters etc. The type which is restricted is denoted by an attribute `base`.
- *List* – creates a type which can contain a list of values of some (simple) type.
- *Union* – creates a type which can contain values from any of the values allowed by the types in the union.

A *Complex type* is a set of declarations of attributes and content of element. The content can be defined using one of the following constructs:

- *Sequence* – creates a sequence of elements (or possible nested constructs from this list) with a strictly defined order.
- *Choice* – corresponding element in the XML document can contain only one subelement which is chosen from the ones contained in the choice definition.
- *All* – a sequence of elements in any order.
- *Simple content* – contains a restriction of a simple type or an extension of such type by a set of attributes.
- *Complex content* – is a derivation of a complex type by a restriction (the new type is a subset of the original one) or an extension (the new type contains both the original and the new one).

Note that in *sequence* and *all* definition, the minimum and maximum number of occurrences of the particular children can be set using attributes `minOccurs` and `maxOccurs`.

Another construct in the XML Schema is a *Group* which contains a set of elements (either in a *sequence*, *all* or a *choice*). It is defined globally (i.e. as a subelement of the root element `schema`) and can be referenced in the schema definition in a way similar to referencing global elements – by inserting an element with the tag `group` and the attribute `ref` with the name of the referenced group. Groups can be seen as some special (globally defined) data types, but instead of defining a type of an element, they define a set of elements.

Global and Local Types

A data type can be defined *locally* (as a subelement of an element definition) or *globally*, i.e. as a subelement of the root element `schema`, in which case the type has to be named. The globally defined (or global) types can be reused (shared) by more than one element. An example of a global complex type `categories` with a subelement `category` with a local type is given in Figure 2.4.

```
<xs:complexType name="categories">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" name="category" >
      <xs:complexType>
        <xs:sequence>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="description" type="description"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="required"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Figure 2.4: Global type with an element with a local type

2.4 DOM

DOM (Document Object Model) [W3C: DOM] is a platform- and language-independent interface for representing and interacting with documents in XML. In DOM, the XML document is represented as a graph in the application memory which consists of *nodes* which represent elements, textual content, attributes and other XML constructs. The fact that the whole XML document is stored in the memory makes the DOM usage problematic for large documents. On the other hand, it allows fast traversing and manipulating of the graph (and thus the document).

2.5 Querying XML

Since XML is used widely for storing data, there is a natural need for querying XML documents and there is a couple of languages specified. The most important ones are *XQuery* [W3C: XQuery] and its subset, *XPath* [W3C: XPath].

2.5.1 XPath

XML Path Language (XPath) is a language used especially for selecting parts of an XML document (mainly sets of nodes). The main construct of XPath language is a path – it denotes a path in a model of XML data. This is a graph of nodes corresponding to elements, textual nodes, attributes and other parts of the XML document. The graph has some specifics which make it different from the DOM graph, for example the root of the graph is a special node (denoted by `/`) different from the root of the document.

An expression in the XPath language is then a regular expression which addresses such a path. The expression can either start in the root node, then it represents an *absolute* path, or it can be a *relative* path which starts in a *context* node given together with the query. An example of an XPath query is given in Figure 2.5. It assumes an XML document whose fragment is in Figure 2.1a and retrieves all `items` in the document which have a specification of a category (i.e. childrent `category`).

```
//item[incategory]
```

Figure 2.5: An XPath query

2.6 Relational Model

Relational model is probably the most widely used database model (if we count Object-Relational model also as a relational one). It was firstly proposed in [Codd (1969), Codd (1970)]. It describes a database as a collection of predicates on predicate variables with constraints on the possible values. The content of the database is a finite model of it, i.e. set of *relations*.

In practice, the relations are often called *tables* and their attributes are referred as *columns*. The actual data values are stored in the tables as a set of *rows*. Also the existing database management systems (DBMS) use *SQL* (the latest version is standardized in [SQL:2008]) as a query and data definition language. Note that the model of SQL is slightly different from the original relational model. An

example of an SQL script creating a relational table called `item` is given in Figure 2.6.

```
CREATE TABLE item (
ID integer PRIMARY KEY,
FEATURED char(1),
LOCATION varchar(100),
QUANTITY integer,
NAME varchar(50)
)
```

Figure 2.6: SQL Create table

Chapter 3

Managing XML Data

As we briefly mentioned in Chapter 1, there have been proposed many solutions for storing and managing XML data and these solutions use a couple of different approaches. These approaches are summarized in the following list with their pros and cons and comparison of main attributes such as effectiveness, round tripping etc.

- *filesystem based methods* – these methods store XML documents as text files in a filesystem, to process the data, content of the files has to be loaded using some other method such as DOM or SAX.

  + **:** simple storing of data, no mapping mechanism needed,

  + **:** stored data are easily accessible and readable by humans

  + **:** maximum level of round tripping (the same document character by character)

  − **:** data cannot be processed without additional technique which requires reading content of the files to other structures and, hence, processing is often slow

- *(O)RDBMS-based methods* – this approach uses an (Object-)Relational Database Management System to store the data, which has to be properly mapped into DB structures.

  + **:** usage of wide-spread, reliable and efficient technology

+ : solutions do not have to be built from scratch

− : efficiency is highly dependent on the mapping from XML data to relational tables, designing of such mapping is not trivial and, generally, it has to consider a lot of factors

− : queries over XML data have to be translated to SQL, this translation can also be nontrivial especially for advanced query languages like XQuery

- *methods using object-oriented approach* – these methods store directed graph of objects, use special indexes to access them.

  − : new query evaluation methods and tools have to be developed

- *native XML methods* – these methods are proposed from the beginning for XML and the storage strategy is fully optimized for its structure.

  + : the methods respect natural tree structure of XML data

  + : easy and efficient querying

  + : flexibility

  − : the whole solution, i.e. data structures, data management system and tools, has to be built from scratch

The native XML methods are by intuition the most efficient and in fact they would probably perform better than other solutions. However, in context of practical world, their disadvantages cannot be ignored. Not only a new technology has to be developed (or purchased, however the market is still very young [Harold (2007)]), but also analysts, developers and sometimes power users have to learn and adapt to the new principles and tools. And native XML technologies have neither very long theoretical nor practical history, so the technology is not yet really mature and verified.

On the contrary, relational (or object-relational) databases have been theoretically studied for a long time, widely used in real-world applications and they have become the most popular way of storing any kind of data. Modern database management systems are reliable, efficient and scalable, they have highly optimized

query processors and provide concurrency control, crash recovery and many other features.

For these reasons, (O)RDBMS-based ways to process XML data are very popular and in our opinion they will still be important XML management systems for at least a couple of next years, if not longer. Thus there is still a reason for studying and improving these methods.

The main aspect of methods using relational databases is how they transform or map a XML tree into relations. Generally there are three different approaches to creating such a mapping:

- *Generic* – these methods use universal mapping regardless of an existing schema of the stored data.
- *User-defined* – this approach avoids automatic creating of the mapping by leaving all the decisions to the user.
- *Schema-driven* – methods which employ information of a schema of the XML data (described in DTD, XML Schema etc.[1]) to create mapping optimized for the given situation.

Naturally, all of these methods have their pros and cons. *Generic* methods are the most universal and simple ones and they are suitable for situations, where there is no schema given for stored documents, or the schema may vary a lot over the time. However, the created relational schema cannot be optimal for all documents and will probably be inefficient.

User-defined approach, on the other hand, leaves all the logic on the user, so this method can theoretically achieve the best outcome. But it requires very skilled user in both XML and database technologies and also creating a relational schema for large and complex documents is not easy.

The last mentioned approach, *schema-driven* methods, seems to be a good compromise in a situation of a known schema (or schemas) of stored documents. It can be fully automated and thus it does not require any knowledge or help from the user. It takes advantage of a given schema of the data and tries to optimize the

[1] list of languages to define schema of XML data can be found for example in [Wikipedia: XML schema]

mapping for it. This optimization starts with some initial relational schema, such as a trivial schema, where there is a separate table for each XML element, and tries to improve it using a set of transformations such as inlining and outlining, splitting and merging types, etc.

The aim of these methods is to produce the most suitable and efficient schema. So the question is, how to determine the quality of a particular schema. There are generally two ways – *fixed* techniques, which tell which mapping is better using general rules and heuristics (such as that the number of tables should not be too small but neither too large) and *adaptive* (or *flexible*), which use additional information about the application and its intended usage to optimize the result to a particular case. The latter approach is apparently the most promising and, hence, suitable for further improvements and it is discussed in the rest of this thesis.

3.1 Adaptive XML-to-Relational Mapping

The main principle of adaptive schema driven methods is not only the awareness of a structure of stored data but also employing other information about the target application, such as a sample of XML data, queries or various statistics. This information is used in the process of creating the target relational schema to evaluate the cost of a particular result according to given set of data and queries and thus to tune the created relational schema to a particular situation and usage.

Advantages of this approach are obvious – it takes into account the intended usage and so the created schema is customized to it. Hence, in an average case, querying over such a schema is more efficient. On the other hand, there are two principal problems with this approach. First, the space of possible relational schemas generated using various transformations is very large, possibly infinite, and it was proven (for example in [Zheng et al. (2003)] or [Xiao-ling et al. (2002)]), that even for a simple set of transformations the problem is NP-hard. Therefore various heuristics and terminal conditions are used and, hence, only a suboptimal solution can be found.

Second, the adaptation of the schema to the application is usually done only once at the beginning and the quality of the result is consequently highly depen-

dent on how good the given samples are. But of course giving a good sample of data or even queries before the application is even created can be hard and the real usage may change quite a lot, which causes the created schema to be less efficient or even worse then a result of fixed methods.

3.2 Related Work

The problem of XML-to-Relational mapping and adaptive methods in particular has been studied in a number of papers. To start, [Mlýnková, Pokorný (2006)] gives a comprehensive list of various methods, their principles and discusses their pros and cons. In this section we provide a brief summary of adaptive methods with focus on their advantages and disadvantages and their possible improvements.

In [Klettke, Mayer (2000)] the authors use a hybrid approach, where some parts (fragments) of an XML document are stored in a classical way using relational tables for elements and some fragments using a special *XML datatype*. This proposed datatype provides path expressions evaluation as well as fulltext operation on `#PCDATA` parts of the stored fragment. Which fragments of the given XML document would be stored into this special type is determined using a simple weight computing mechanism which takes into account the DTD structure, sample data and queries. The main idea is that for the well structured document parts storing in relations is suitable, but for the semi-structured ones it is better to use the *XML datatype*.

This approach, however quite simple, can easily be deployed since most modern database management systems contain an XML type which corresponds to that defined in the paper [Klettke, Mayer (2000)]. And even in more sophisticated algorithms, the idea of storing an XML fragment in a special single attribute can be used.

In [Ramanath et al. (2003)], a *FlexMap* framework is proposed which enhances the *LegoDB* system from [Bohannon et al. (2001), Bohannon et al. (2002)]. It uses a greedy heuristic and proposes a comprehensive set of schema transformations – Inlining and Outlining, Type Split and Merge, Commutativity and Associativity, Union Distribution and Factorization, Splitting and Merging repetitions

and Simplifying unions[2]. Nevertheless, the set of actually implemented and used transformations is much smaller.

Another feature of the proposed solution is usage of *StatiX* [Freire et al. (2002)], an XML Schema-aware framework, for collecting statistics of the input schema and sample data. These statistics are stored in the schema (resp. its internal representation) and preserved while applying transformations so that they can be used (together with sample queries) to get the cost of a given schema. Also the statistics are computed on fully decomposed schema (i.e. schema where all possible split operations have been performed) and then updated at each XML-to-XML transformation. Hence every XML document has to be fully processed only once.

[Zheng et al. (2003)] uses a broader set of transformations – a combination of transformations similar to those in the previous approaches. The search for (sub)optimum is done using the Hill Climbing heuristic. The authors conducted a set of performance experiments (sample queries were obtained from XMark XML benchmark [Schmidt et al. (2001)]), which tested not only the performance of the proposed algorithm itself, but also an impact of initial schema. They conclude, that this choice is crucial for a good result of their algorithm.

Authors of this paper also propose quite complex cost function for their solution. It exploits sample data statistics gathered at the beginning of the algorithm and computes or estimates other necessary variables. All of these are then used to determine the cost of a query by simulating the cost of joining relational tables corresponding to particular schema fragments.

In [Xiao-ling et al. (2002)] so called *Adjustable and Adaptable Method (AAM)* is proposed. This method searches for a (sub)optimal mapping using principles of genetic algorithms, which bring some level of randomness into decisions (in creating initial population, computing cost, mutation between populations etc.). The method does not use any sample set of queries or documents, instead only probabilities of retrieval of particular elements are used.

[Atay et al. (2007)] propose schema-aware mapping algorithm which simplifies DTDs and then maps a DTD graph (graph constructed from the DTD by making

[2]For detailed summary of schema transformations and an explanation of their function see Section 4.5

nodes from all elements and join them in a hierarchy as defined in the DTD) into a relational schema by inlining child nodes. Although the algorithm is of *fixed* type, it provides some interesting features, especially the capability to reconstruct the original XML document with its order and also support for more efficient evaluation of queries with preceding/descending axes. Two efficient data loading algorithms were proposed as well.

3.3 Motivation

As we have shown in previous sections, there is a couple of approaches to mapping XML to relational schema and the adaptive schema-driven methods seem to be the most promising ones. There is a number of papers and implementations based on these adaptive principles, however all of them have both advantages and disadvantages.

Especially because of the disadvantages, we think there is still a space for improvements of these methods and in this thesis we propose a method which addresses some of these drawbacks while still taking advantage of the adaptive approach.

Here are in our view the most important drawbacks in current solutions and naturally these are the domains which we want to find an improvement in and, thus, form goals for our work:

- *Choice of heuristic* – as already found out by [Mlýnková (2008)], most of the proposed solutions use only a basic greedy search strategy, which has its disadvantages (especially the possibility of getting stuck in local suboptimum). So our first aim is to use a more sophisticated one.

- *Set of schema transformations* – existing algorithms use only a subset of proposed schema transformations. It could be interesting to evaluate the results of a solution which uses more of them and to compare their impact on the result.

- *Dynamic adaptation* – the main problem of all adaptive techniques is that the schema is adapted to future usage only once at the beginning. One

of our goals is to study a possibility of a mechanism which would enable dynamic adaptation of used database schema.

Chapter 4

Analysis

In this chapter we formalize the XML-to-Relational mapping problem, provide the analysis of our way to solve it and describe the design and architecture of our prototype implementation of proposed solution called *AntMap*.

4.1 Adaptive XML-to-Relational Mapping Problem

As introduced in Section 3.1, adaptive methods map an XML schema to relations using various information about the application, usually a sample set of documents or statistics counted from them and a set of queries representing the usage of the data. The problem can be generally formalized as follows:

- *Input:* an initial XML schema S_{init}, a set of sample data and queries D_{sample} and a cost function f_{cost} which determines the efficiency of given relational schema R considering the set D_{sample}.
- *Output:* an optimal relational schema R_{opt} which minimizes the cost function, i.e. the value of $f_{cost}(R_{opt}, D_{sample})$ is minimal.

In practice, the adaptive algorithms have some other common features and can be generalized as a simple process:

- *Input:* S_{init}, D_{sample}, f_{cost}
- *Additional parameters:*
  - a set of XML schema transformations $Trans = \{t_1, t_2, \ldots, t_{|Trans|}\}$ where every $t_i \in Trans$ transforms an XML schema S_1 to another XML schema $S_2 = t_i(S)$
  - a fixed mapping strategy f_{map} capable of transforming the given XML schema S to a relational schema R
- *Mapping selection process:*
  - Search Σ – the space of possible transformed schemas, it can be defined as:

$$\Sigma = \{S_{init}\} \cup \{S_i | S_i = t_{i_1}(t_{i_2}(\cdots(t_{i_n}(S_{init})))) \wedge t_{i_j} \in Trans\}$$

  - Find $S_{opt} \in \Sigma$ such that $f_{cost}(f_{map}(S_{opt}), D_{sample})$ is minimal.

The most important aspect of adaptive algorithms is that while the set $Trans$ is always finite, the set of possible solutions Σ can be infinite or, at least, very large. As already mentioned in Section 3.1, the problem was proven to be NP-hard even for a small set of transformations. Results of the mapping algorithms are therefore usually only suboptimal.

Consequently the most interesting feature of any flexible algorithm is how it searches Σ for the solution, how effective and time-consuming the search algorithm is and how good (i.e. how close to the optimum) the found solution is. The search algorithm used in our solution is described in detail in Chapter 5. However, the mapping system has many other aspects and problems whose solution is not trivial. The rest of this chapter discusses them in detail.

4.2 Internal Schema Representation – *NSchema*

In the description of a general mapping algorithm it is assumed that both the f_{map} function and the schema transformations work with the XML schema whose

actual form is not further specified. Various algorithms represent the schema differently and sometimes the schema is even transformed before the main search algorithm is applied to enable simple translation to a relational schema. However, most of these representations use some form of a graph which corresponds to the nature of as an XML schema.

In our view, there are two principle variants of such a representation:

- Use one of the defined XML schema format directly.
- Define the graph independently and then create it from an input format (or possibly enable the system to create the internal graph from various schema formats).

The advantage of defining own schema representation is a control over it, however usage of a standard form enables to use already implemented and verified tools and technologies. Our solution tries to include features of both of these approaches – we decided to base our representation on XML Schema format. This format keeps the natural structure of the schema while being in fact an XML document so we can represent it as a DOM graph and use standardized APIs for parsing and tracing which are implemented for various environments. Also it is a more general format than for example DTD, which can be transformed to it. However, we define some transformations and constraints on the schema which ease other parts of the system while technically they still keep the schema in a DOM tree.

The mapping function f_{map} should be well defined on every instance of our schema representation and we want it to be as simple as possible – every logic which influences the actual relational schema should be kept in the mapping search algorithm, i.e. the internal schema representation should directly correspond to the relational storage. Because of that, the input schema needs to be somehow transformed, or normalized, so that it can be directly mapped to relations and all schema transformations have to produce only this normalized form. To achieve that we employ a concept of *p-schemas* introduced in [Bohannon et al. (2001), Bohannon et al. (2002)], we call our representation *NSchema* (as Normalized Schema). NSchema fulfills all requirements for simple fixed mapping to relations and it is also enriched by some features which simplify the search algorithm.

The requirements for the NSchema come from the NSchema-to-Relational translation algorithm (see Section 4.4) and are defined on the basis the requirements on *p-schema* (as described in [Freire, Siméon (2003)]):

- Every global element in the schema uses named (i.e. globally defined) type.
- There are no element references.
- There are no shared globally defined complex types and groups, i.e. every globally defined complex type can be used by only one (or possibly none) element and every group can be referenced only once.
- Every type defines a structure that can be directly mapped into a relation. This means that any type definition (both local and global) can be a simple type definition or, in case of complex type, can contain either:
  - Subelements with a simple (either built-in or user-defined) type.
  - Complex regular expression (e.g. repetitions of elements, unions etc.) made of items which do not directly hold values (i.e. the textual content). So every such complex expression can contain only elements with a globally defined complex type and/or group references.

Such a schema can be easily mapped to relations by simply creating a relation for every globally defined complex type and for every group, for details see Section 4.4.

The most important and also most difficult to understand is the last constraint. From the point of view of the relational database world it is, however, quite natural so let us rephrase it this way.

Say we have a one-to-many relation between two entities A and B, which corresponds to a repetition of some item in the NSchema. It is natural that it would not be modeled using only one relational table since it would require a duplication of the data in the database – let a be an instance of the entity A which is in the relation with $b_1, b_2, \ldots, b_n$, instances of B. If we model this situation using only one table, it would have to contain n rows, each of which would have to contain all the attributes of a and then attributes of some b_i.

This data duplication is avoided in most database schemas (and in general algorithms and guidances for creating a relational schema) for two main reasons:

ID A	name A	address A	ID B	name B	function B
1	Firm ltd.	New York	5	John Smith	CEO
1	Firm ltd.	New York	8	David Gates	programmer
1	Firm ltd.	New York	15	Michael Door	analyst

Table 4.1: Single table for entities A, B

ID A	name A	address A
1	Firm ltd.	New York

(a) Table **A**

ID B	ID A	name B	function B
5	1	John Smith	CEO
8	1	David Gates	programmer
15	1	Michael Door	analyst

(b) Table **B**

Table 4.2: Separate tables for A, B

- Inefficiency – data representing the same instance of an entity are contained more then once in the database.
- Updatability – when some attribute of the entity changes, it has to be updated in all records containing the attributes of the given instance.

The natural way to model such a situation is to create two separate tables – for both entities A and B. This way every instance of B can contain a reference to appropriate instance of A and no data are duplicated. And this is exactly what we achieve with the constraint – when the complex regular expression is modeled using a group or globally defined type reference, the resulting relational schema will also contain only a reference between two tables and data are not duplicated. An example of this situation is given in Tables 4.1 (single table) and 4.2 (separate tables).

4.3 Schema Normalization

In Section 4.2 we defined NSchema and constraints it should fulfill, here we provide a method which is used to transform the original XML Schema definition into NSchema. It is based on a normalization algorithm sketched in

[Freire, Siméon (2003)], however it is adjusted to usage with a DOM representation of an XML Schema document (i.e. we exploit the fact that a schema in the XML Schema format is still an XML document) and elaborated in more detail.

The contract for the normalization algorithm consists of four goals:

- Remove local type definitions in global elements.
- Remove element references.
- Remove shared globally defined types and groups.
- Change the structure of defined types so that they can be directly mapped into relations.

Algorithms solving all of these goals are described in the following sections.

4.3.1 Removing of Local Type Definitions in Global Elements

Removing of local type definitions from global elements is quite simple, the process is described in Algorithm 4.1.

Algorithm 4.1 Removing local type definitions from global elements

```
GlobalElements ← All globally defined elements
for all ge ∈ GlobalElements do
  if ge has a local type definition (denote it T_ge) then
    Insert a new globally defined type T with the same content as T_ge
    Remove the local type definition from ge, add an attribute denoting that
    ge is of type T
  end if
end for
```

4.3.2 Removing of Element References

After local type definitions in global elements have been removed, we can remove element references. The global elements are exactly the only ones which can be referenced. Algorithm 4.2 describes how to do that.

Algorithm 4.2 Removing of Element References

ElementsWithRef ← All element definitions which reference a (global) element
for all e in *ElementsWithRef* **do**
  $g \leftarrow$ the global element being referenced by e
  $T \leftarrow$ the type of g
  Change e so that it uses type T instead of reference to g
end for

Since all local type definitions have been removed from the global elements, g must have a globally defined type.

Note that while the algorithm removes the references to global elements, the global element definitions have to remain in the schema because their function is not only to be used as reference but also can act as a root element of an XML document valid against the schema.

Also note that the algorithm is defined as a simple loop over all elements with a reference (which have to be somehow obtained, for example using an XPath query), it can be nevertheless also implemented as a simple recursion over the nodes of the DOM graph of the schema and possibly merged with other parts of the normalization algorithm.

4.3.3 Normalization of Type Structure

The normalization of the type structure, described in Algorithms 4.3, 4.4 and 4.5, ensures the possibility to directly translate a normalized schema into a relational one. To explain the particular steps, we use an example XML Schema definition derived from the DTD of XMark benchmark data [XMark] (which has also been used for the experiments, see Chapter 8).

Algorithm 4.3 Normalization of the type structure

GlobalTypes ← All globally defined complex types and groups
for all $T \in$ *GlobalTypes* **do**
  Normalize the type/group definition, i.e. apply Algorithm 4.4 on T.
end for

Algorithm 4.4 Normalization of a single node

Input: T – a part of the definition of some type or group
for all $T' \in$ Child elements of T **do**
Recursively process T', i.e. apply Algorithm 4.4 to T'
end for
Normalize the T node itself according to the type of the node:

- *Atomic type* (i.e. `string`): Let T unchanged
- *User-defined simple type*: Let T unchanged
- *Optional element declaration* (i.e. an element with attributes `minOccurs` $= 0$ and `maxOccurs` $= 1$): Let T unchanged
- *Sequence*: Let T unchanged
- *All*: Change T to a *Sequence* of the child items. See Figure 4.1 for an example.
- *Choice* (called also *Union*): Create a new group definition G_c for every possible choice c of the union. Then change the original choice to a choice of group references to every newly created group G_c.

Solve possible repetition, i.e. apply Algorithm 4.5 to T

Algorithm 4.5 Solving repetitions

Input: T – a part of the definition of some type or group
if T is a *Repetition*, i.e. an element or other construct with attribute `maxOccurs` > 1 **then**
Insert a new group definition G, containing T without the `min/maxOccurs` attributes
Replace T in the schema with g – a reference to G
Add the `min/maxOccurs` attributes from T to g.
An example is given in Figure 4.3
end if

```
<xs:complexType name="regions">
<xs:all>
  <xs:element name="africa"
      type="africa"/>
  <xs:element name="asia"
      type="asia"/>
  <xs:element name="australia"
      type="australia"/>
  <xs:element name="europe"
      type="europe"/>
</xs:all>
</xs:complexType>
```

(a) Before normalization

```
<xs:complexType name="regions">
<xs:sequence>
  <xs:element name="africa"
      type="africa"/>
  <xs:element name="asia"
      type="asia"/>
  <xs:element name="australia"
      type="australia"/>
  <xs:element name="europe"
      type="europe"/>
</xs:sequence>
</xs:complexType>
```

(b) Normalized

Figure 4.1: Normalization of *All*

The normalization algorithm basically processes all globally defined complex types and groups, for every one of them it processes its whole definition recursively and normalizes every node.

Note that all steps of the normalization except of solving *All* keep the set of the XML documents valid against the schema unchanged. The change from an unordered sequence (i.e. *All*) to an ordered one is caused by the fact that the schema of the XML Schema does not allow *All* to contain groups. This constraint makes it impossible to normalize the content of *All* properly and also to use some of the schema transformations. However, XML documents valid against S_{init} can be transformed to suit the normalized schema – the elements in the *All* construct have to be just reordered to fit the order of the definition in the schema.

When inserting a new group definition containing only single item, for example while solving a repetition in Algorithm 4.5, the group will in fact be defined as a sequence of this one item. This is also required by the schema of XML Schema because a group cannot directly contain an element.

Also note that there are some XML Schema constructs omitted in the algorithm. These are especially attributes and the Simple content, which can be, however, handled simply the same way as subelements (or the algorithm in Section 4.3.1 can be adjusted to also remove attribute definitions and change them

```
<xs:complexType name="description">
  <xs:choice>
    <xs:element name="text"
       type="xs:string"/>
    <xs:element name="parlist"
     type="parlist"/>
  </xs:choice>
</xs:complexType>
```

(a) Before normalization

```
<xs:complexType name="description">
  <xs:choice>
  <xs:group ref="parlist_choice"/>
  <xs:group ref="text_choice"/>
</xs:choice>
</xs:complexType>

<xs:group name="parlist_choice">
  <xs:sequence>
    <xs:element name="parlist"
     type="parlist"/>
  </xs:sequence>
</xs:group>

<xs:group name="text_choice">
  <xs:sequence>
    <xs:element name="text"
     type="xs:string"/>
  </xs:sequence>
</xs:group>
```

(b) Normalized

Figure 4.2: Normalization of *Choice*

```
<xs:complexType name="categories">
  <xs:sequence>
    <xs:element maxOccurs="unbounded"
     name="category" type="category"/>
  </xs:sequence>
</xs:complexType>
```

(a) Before normalization

```
<xs:complexType name="categories">
  <xs:sequence>
   <xs:group maxOccurs="unbounded"
    ref="category_rep"/>
  </xs:sequence>
</xs:complexType>

<xs:group name="category_rep">
  <xs:sequence>
    <xs:element name="category"
     type="category"/>
  </xs:sequence>
</xs:group>
```

(b) Normalized

Figure 4.3: Solving *Repetitions*

to subelements). Other feature is so-called Complex content, i.e. a way to derive a complex type as an extension or restriction of an existing type. The restriction can be quite simply solved by replacing the derived type by its parent type, however the extension would need a deeper analysis, so we omit the complex content completely as it is out of scope of this thesis. For the same reason we forbid the input schema to contain any wildcards, i.e. the element *any*.

4.3.4 Removing of Shared Global Types and Groups

The removal of the shared globally defined types is necessary for the possibility of gathering schema statistics once at the beginning and then only updating them while performing a schema transformation [Ramanath et al. (2003), Bohannon et al. (2001), Bohannon et al. (2002)].

Since our representation of the normalized XML Schema is (after the normalization) used in the rest of the mapping process and since almost all of its parts often work with globally defined types and groups, we hold their list in a special structure of the NSchema. And while we extract this list from the schema DOM graph, we also count and remember how many times every type or group is used

(i.e. how many group references to the particular group are in the schema and how many elements with a particular type are defined in the schema).

These usage counts enable us to remove shared types or groups very simply. We just loop over the list of global types and groups and for those with more than one usage we copy them as many times as their usage count is. Every usage of the type (or reference to the group) is then changed to one of the copies. An example is given in Figure 4.4.

Note that this step can be performed after the rest of the normalization process but it can be also done as the first step and all other parts of the normalization can be simply adjusted so that they do not create shared types.

4.3.5 Type and Group Names Handling

Every time any of the presented algorithms inserts a new globally defined type or group into the schema, it uses a simple algorithm (depicted in Algorithm 4.6) to determine a final name of it.

Algorithm 4.6 Determining the name of a newly inserted type or group

Input: n a desired name for the type
$counter \leftarrow 1$
$newName \leftarrow n$
while There is a globally defined type or a group with name = new_name **do**
  $newName \leftarrow$ concatenate$(n, counter)$
  $counter \leftarrow counter + 1$
end while
return newName

The desired name for the type is determined by the situation which leeds to the insertion of a new type, usually the name of the element with the type (or contained in the group) is used. A suffix denoting the particular situation is then added (for example "_rep" for repetition, "_choice" for unions etc.).

```
<xs:complexType name="item">
  <xs:sequence>
    <xs:element name="location"
     type="xs:string"/>
    <xs:element name="quantity"
     type="xs:string"/>
    <!-- other definitions -->
  </xs:sequence>
</xs:complexType>

<xs:complexType name="africa">
  <xs:sequence>
    <xs:element name="item"
     type="item"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="asia">
  <xs:sequence>
    <xs:element name="item"
     type="item"/>
  </xs:sequence>
</xs:complexType>
```

(a) Before normalization

```
<xs:complexType name="item">
  <xs:sequence>
    <xs:element name="location"
     type="xs:string"/>
    <xs:element name="quantity"
     type="xs:string"/>
    <!-- other definitions -->
  </xs:sequence>
</xs:complexType>

<xs:complexType name="africa">
  <xs:sequence>
    <xs:element name="item"
     type="item"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="item_2">
  <xs:sequence>
    <xs:element name="location"
     type="xs:string"/>
    <xs:element name="quantity"
     type="xs:string"/>
    <!-- other definitions -->
  </xs:sequence>
</xs:complexType>

<xs:complexType name="asia">
  <xs:sequence>
    <xs:element name="item"
     type="item_2"/>
  </xs:sequence>
</xs:complexType>
```

(b) Normalized

Figure 4.4: Removing shared types and groups

4.4 NSchema to Relational Schema Translation

Once we have a normalized XML schema, NSchema, the translation to relational schema is quite straightforward:

- Create a relation R_X for every globally defined type and group X.
- For each relation R_X create a key that identifies the corresponding element.
- For each relation R_X create a foreign key P_X to all relations R_{P_X} such that P_X is a parent type or group of X.
- Create a column in R_X for every simple type element a inside the type or group X.
  - *Built-in (atomic) type* – translate to an equivalent SQL type.
  - *User-defined simple type* – translate its basis to an SQL type.
- If the data type is contained within an optional type then the corresponding column can contain a null value.

A globally defined type or group P_X is considered as a parent type of X if the definition of P_X contains an element with a type X or a group reference to X.

Every user-defined simple type is derived from a built-in type (possibly in more than one step) by some restrictions or extensions. We call this built-in type a basis of the user-defined type and use it in the translation. Note that similarly to Section 4.3.3, we do not handle possible user defined simple types derived by an extension.

Before the created relational schema can be loaded into the database, the tables have to be topologically sorted according to parent/child relationships (i.e. according to foreign keys).

4.5 Schema Transformations

The mapping selection algorithm (see Chapter 5) uses a set of schema transformations to derive variations of S_{init}. The possible transformations proposed in various papers (see Section 3.2) are summarized in the following list:

- *Inlining & Outlining* – The inline and outline transformations change the place where an element e stored – either directly in a table T corresponding to the parent type of e (*inlined*) or in a separate table (*outlined*). Because of the way of creating a relational schema for the NSchema (see Section 4.4), inlined elements are those which have a locally defined type or a simple type. Those which have a globally defined type or are members of a group, which is referenced from T, are outlined. There are two variants of the outline transformation:
  - Local type elements outlining – this transformation is applicable on elements with a locally defined type, it creates a new globally defined type with the same definition and changes e so it uses the new type.
  - Simple elements outlining – this variant outlines elements with a simple type (either user-defined or built-in) and also sets of such elements. It does that by creating a new group containing the outlined elements.

  An example of inlinlined and outlined element `category` is given in Figure 4.5.

- *Commutativity & Associativity* – these transformations alter the structure of the schema and the order of the contained items. Associativity groups different elements into a single relational table, while commutativity changes the order of elements and, thus, can change which elements are grouped by associativity. The same effect can be achieved by applying the simple elements outlining though, so we do not use these two transformations. An example of associativity is given in Figure 4.6.

- *Splitting & Merging types or groups* – Split breakes a shared type or a group (i.e. a type or group used by more than one type definition) into separate definitions while merge does the exact opposite. Because of the way we gather and manage schema statistics (see Section 4.7), we perform all possible split operations at the beginning (see Section 4.3.4) and only allow merge in the algorithm.

- *Simplifying unions* – this transformation exploits the fact that a union is always contained in a sequence of optional elements. This, of course, extends the set of valid documents but as the transformed schema is used only to

create a relational one this does not cause any problem. After the simplifying, the types of the elements of the union can be inlined which can improve the cost of the whole schema. An example of this transformation is given in Figure 4.7.

4.6 Cost Estimation

An important part of the mapping selection process is the cost function, f_{cost}. It is used to represent a RDBMS engine which creates an execution plan for every query and is usually capable of evaluating it by a cost value. So naive but quite illustrative way to get a cost of some transformed schema would be to translate it into a relational one, load that with sample XML documents, translate sample XML queries to SQL and get a sum of their cost. When the best schema according to this cost is found, it would be then also the best one in the real usage.

Nevertheless, this process would be quite time-consuming and probably render the whole algorithm useless. Only the load of the sample data can take a couple of seconds or even minutes and it would have to be done for the evaluation of every constructed schema and the number of schemas constructed during the search can be very large.

An improvement of this method is presented in [Ramanath et al. (2003)] and previous publications on *LegoDB* ([Bohannon et al. (2001), Bohannon et al. (2002)]). Instead of loading data to a database, the presented systems count various statistics about the document (such as a number of instances of particular element) and store it directly in their schema representation. These statistics are then updated with every schema transformation. When a schema is to be evaluated, the statistics are translated into appropriate relational statistics for the derived relational schema and they are then used (together with sample queries) as an input for a relational optimizer to obtain cost.

Another approach is to avoid the usage of relational optimizer or database completely and define a true cost function, which simulates the relational optimizer and derives the cost of a query directly from the schema and statistics. This approach obviously does not produce so accurate costs as the use of a real

```
<xs:complexType name="categories">
  <xs:sequence>
    <xs:element maxOccurs="unbounded"  name="category" type="category"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="category">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="description" type="description"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:ID" use="required"/>
</xs:complexType>
```

(a) Outlined `category`

```
<xs:complexType name="categories">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" name="category" >
      <xs:complexType>
        <xs:sequence>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="description" type="description"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="required"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

(b) Inlined `category`

Figure 4.5: Inlined and outlined element

```
<xs:complexType name="category">
  <xs:sequence>
    <xs:element name="name"
        type="xs:string"/>

    <xs:sequence>
      <xs:element name="text"
          type="text"/>
      <xs:element name="parlist"
          type="parlist"/>
    </xs:sequence>
  </xs:sequence>
  <xs:attribute name="id"
    type="xs:ID" use="required"/>
</xs:complexType>
```

(a)

```
<xs:complexType name="category">
  <xs:sequence>
    <xs:sequence>
      <xs:element name="name"
          type="xs:string"/>
      <xs:element name="text"
          type="text"/>
    </xs:sequence>

    <xs:element name="parlist"
          type="parlist"/>
  </xs:sequence>
  <xs:attribute name="id"
    type="xs:ID" use="required"/>
</xs:complexType>
```

(b)

Figure 4.6: Associativity

```
<xs:complexType name="desc">
  <xs:choice>
    <xs:element name="text"
          type="text"/>
    <xs:element name="parlist"
          type="parlist"/>
  </xs:choice>
</xs:complexType>
```

(a)

```
<xs:complexType name="desc">
  <xs:sequence>
    <xs:element name="text"
     type="text"
        minOccurs="0" maxOccurs="1"/>
    <xs:element name="parlist"
     type="parlist"
        minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
```

(b)

Figure 4.7: Simplyfying unions

database management system, nevertheless, it has some advantages. First, an artificial cost function does not depend on a particular RDBMS system and, hence, it can be general enough to be a satisfactory simulation of a set of such systems. Thus the schema can be optimized regardless the given target RDBMS.

Second, a cost function implemented as a part of the mapping system itself is usually much more efficient and, hence, the whole algorithm can either run much faster or search a much larger subspace of Σ.

In our solution, we have chosen to define an artificial cost function and after detailed analysis we decided to use the one presented in [Zheng et al. (2003)]. It exploits statistics about the schema and simulates the join of the relational tables necessary for evaluating the query.

The original f_{cost} is defined over a relational schema R and a sample set D_{sample} which consists of a set of sample XML documents X_{sample} and a set of sample XML queries Q_{sample}:

$$f_{cost} : R \times D_{sample} \rightarrow \mathbb{R}_0^+ \tag{4.1}$$

Thus it takes a relational schema and sets of sample documents and queries and returns a cost as a (non-negative) real number. We made some adjustments to the definition of f_{cost}, so our cost function is defined as

$$f_{cost} : S \times Q_{sample} \rightarrow \mathbb{N}_0^+ \tag{4.2}$$

where S is a normalized XML Schema (NSchema) which also contains statistics (see Section 4.7), $Q_{sample} = \{(Q_i, w_i)_{i=1,2,\ldots,n}\}$ is a set of sample queries (Q_i) with assigned weights (w_i). Note that in our solution we set weights of all queries to 1 (i.e. we omit the weighting of queries completely).

4.6.1 f_{cost} Calculation

The calculation of the cost function, as defined in [Zheng et al. (2003)], uses a couple of variables and information explained in the following list. The model is based on f_i – a schema fragment rooted in element E_i. In our situation, the relevant fragments for the calculation of the cost are the definitions of global types and groups because all elements in such a definition are then contained in a single relational table.

- $|E_i|$ – the number of instances of element E_i from the schema in X_{sample}
- C_i – the width of the field corresponding to an element E_i with a simple value
- *Fan-out*(i, j) – the average number of instances of E_j in E_i (where E_j is a subelement of E_i) derived from X_{sample}, i.e. $|E_j|/|E_i|$
- $|D_i|$ – the number of instances of all elements in the schema fragment f_i
- $|f_i|$ – the size of all elements in the schema fragment f_i
- $Sel(e_1/e_2/\ldots/e_k)$ – the selectivity of a simple path $e_1/e_2/\ldots/e_k$
- Sel_i – the selectivity of the path from the root of the schema to the fragment f_i

The function is then calculated as follows:

$$f_{cost}(S, Q_{sample}) = \sum_{i=1}^{n} w_i \cdot \text{cost}(Q_i, S) \tag{4.3}$$

where $\text{cost}(Q_i, S)$ is a cost of evaluating a query Q_i on the schema S. In [Zheng et al. (2003)] it is defined for a query Q_i which accesses $\{f_{i1}, \ldots, f_{im}\}$, a set of m fragments in S:

$$\text{cost}(Q_i, S) = \begin{cases} |f_{i1}| & \text{when } m = 1 \\ \sum_{j,k}(|f_{ij}| \cdot Sel_j + \delta \cdot \frac{(|E_{ij}|+|E_{ik}|)}{2}) & \text{when } m > 1 \end{cases} \tag{4.4}$$

where f_j, f_k are two joined fragments while evaluating the query and δ is a coefficient. To simulate common hash join of the corresponding relational tables, δ is set to 3 [Zheng et al. (2003)].

Note that we assume, similarly as the authors of [Zheng et al. (2003)], that the queries are defined using only simple path expressions and that every path starts at the root of the schema. Since complex regular expressions can be translated into a set of simple path expressions exploiting the schema, this assumption is not too restrictive.

Consequently, if a simple path query $Q_i = e_1/e_2/\ldots/e_k$ can be rewritten using corresponding fragments (which are globally defined types or groups) as $f_{i1}/f_{i2}/\ldots/f_{im}/$, then fragments f_{ix}, f_{iy} are joined only if $y = x + 1$. Thus

$$\text{cost}(Q_i, S) = \begin{cases} |f_{i1}| & \text{when } m = 1 \\ \sum_{j=1}^{m-1}(|f_{ij}| \cdot Sel_j + \delta \cdot \frac{(|E_{ij}+|E_{ij+1}|)}{2}) & \text{when } m > 1 \end{cases} \tag{4.5}$$

4.6.2 Calculation of Variables for f_{cost}

The f_{cost} equation (4.5) has a couple of variables which have to be computed. The most important observation is that most of the variables can be computed only once, at the beginning, and they do not change after any transformation except of merging of shared types. This observation (based on a similar observation in [Bohannon et al. (2001), Bohannon et al. (2002)]) has only one precondition – the statistics (and the derived variables) have to be obtained on a fully decomposed schema, i.e. a schema where there is no shared type or group. But we have defined NSchema exactly this way, so we can safely use the observation and, hence, only count most of the variables at the start of the mapping process.

Let us show this observation for a schema S and particular variables:

- $|E_i|$ depends only on X_{sample}, it can only change with a merge transformation, but in that case the new value can be derived:
  - When merging two types with definitions in elements E_i, E_j, denote the resulting element E_m. The value of $|E_m|$ can be counted as $|E_m| = |E_i| + |E_j|$
- C_i is given by the definition of $|E_i|$ in S, so it even does not depend on X_{sample}
- *Fan-out*(i, j) depends only on $|E_i|, |E_j|$ so again it does not change with any transformation except of merge, when the new value can be derived from the old ones.
- Sel_i is the same regardless of any transformation except of the merge since the path from the root of S to E_i is the same. Only the elements on it may

change the type which they are contained in, but this is only a different way to define a schema for the same document.

What changes with a different schema is the set of types or groups accessed when evaluating the query since the definitions of the types change. And consequently, $|D_i|$ and $|f_i|$ also change as the set of the descendant elements of these types changes as well. These variables are thus computed at the time of evaluating the cost of S using following formulas:

$$|D_i| = \sum_{E_j \in f_i} |E_j| \tag{4.6}$$

$$|f_i| = \sum_{E_j \in f_i} |E_j| \cdot C_j \tag{4.7}$$

Note that the values of $|D_i|$ and $|f_i|$ only have to be computed for the fragments corresponding to globally defined types or groups which are accessed by any of the queries in Q_{sample}.

The value of Sel_i is estimated using a *Markov table* [Zheng et al. (2003), Aboulnaga et al. (2001)] from the selectivity of a single step (i.e. a path with length 2) which is equal to the *Fan-out* between the two elements. As introduced in [Aboulnaga et al. (2001)], the selectivity of a path of the length n can be estimated using already known selectivities of paths of the lengths $m < n$:

$$Sel(e_1/e_2/\ldots/e_n) = Sel(e_1/e_2/\ldots/e_m) \cdot \prod_{i=1}^{n-m} \frac{Sel(e_{i+1}/e_{i+2}/\ldots/e_{i+m})}{Sel(e_{i+1}/e_{i+2}/\ldots/e_{i+m-1})} \tag{4.8}$$

When we set $m = 2$, we get:

$$Sel(e_1/e_2/\ldots/e_n) = Sel(e_1/e_2) \cdot \prod_{i=1}^{n-2} \frac{Sel(e_{i+1}/e_{i+2})}{Sel(e_{i+1})} \tag{4.9}$$

Of course, the selectivity of a single element path is 1, i.e. $Sel(e_{i+1}) = 1$. And, as we already mentioned, the selectivity of a path of two element is equal to the

Fan-out of these nodes, i.e. $Sel(e_i/e_j) = Fan\text{-}out(i,j)$. So we get:

$$Sel(e_1/e_2/\ldots/e_n) = Fan\text{-}out(e_1/e_2) \cdot \prod_{i=1}^{n-2} Fan\text{-}out(e_{i+1}/e_{i+2})$$

$$Sel(e_1/e_2/\ldots/e_n) = \prod_{i=1}^{n-1} Fan\text{-}out(e_i/e_{i+1}) \tag{4.10}$$

Finally, we can express $Sel(e_1/e_2/\ldots/e_n)$ using $Sel(e_1/e_2/\ldots/e_{n-1})$ as:

$$Sel(e_1/e_2/\ldots/e_n) = Sel(e_1/e_2/\ldots/e_{n-1}) \cdot Fan\text{-}out(e_{n-1}/e_n) \tag{4.11}$$

Using equation (4.11) we can simply determine the selectivities of all schema elements. We trace the whole schema from the root element and count Sel_i using selectivity of the parent node E_{p_i} and the $Fan\text{-}out(i,p_i)$. Note that in this algorithm we consider two nodes e_1, e_2 in a parent-child relationship when they fulfill one of these conditions:

- e_1 is a parent node of e_2 in the schema XML document, i.e. the definition of e_1 directly contains the definition e_2
- e_1 contains as a child a definition of an element using type with definition in node e_2
- e_1 contains as a child a reference to a group with definition in node e_2

4.7 Gathering Sample XML Data Statistics

The cost function used in our solution, as introduced in Section 4.6, does not need to access sample documents, it only uses simple statistics from them. We obtain these statistics before the mapping selection algorithm starts and store them directly into the NSchema. Since the statistics are needed on an element basis, we count them for every particular element and the result is then stored directly to the element representation. This way the cost function algorithm can acquire the statistics as it traces the schema.

The statistics gathering is done in two steps which are sketched in Algorithms 4.7 and 4.9. The first one is called on every sample XML document, it traces it and counts occurrences of every element in the document (denoted as $count(e)$).

The second one is then called on the root document element of the NSchema. It processes the whole schema and propagates the statistics to the schema nodes which does not directly appear in XML documents. These are especially various constructions used to denote a type and a structure of elements (e.g. schema elements like `complexType`, `sequence`, `choice` etc.).

Algorithm 4.7 Gathering schema elements statistics

Input: XML document D, NSchema S
$root \leftarrow$ The root element of D
$T_{root} \leftarrow$ The type of $root$ according to S {All globally defined elements in S must have a globally defined type}
Apply Algorithm 4.8 on $(root, T_{root})$

Algorithm 4.8 Gathering statistics from a single element

Input: element e, parent type or group T_p
if e has a globally defined type T_e **then**
  $e_s \leftarrow$ a root of the definition of T_e
  $T'_p \leftarrow T_e$
else if e is a reference to a group G_e **then**
  $e_s \leftarrow$ a root of the definition of G_e
  $T'_p \leftarrow G_e$
else
  find e_s – a schema element in the definition of T_p corresponding to e
  $T'_p \leftarrow T_p$
end if
$count(e_s) \leftarrow count(e_s) + 1$
for all $e \in$ Children of e **do**
  Apply Algorithm 4.8 on (e, T'_p)
end for

Algorithm 4.9 Consolidation of the statistics – processing of a single element

Input: NSchema element e, C_p, a value of $count(p)$, where p is the parent NSchema element of e
if $count(e)$ is not defined **then**
  $count(e) \leftarrow C_p$
end if
if e is a reference to a group G **then**
  Apply Algorithm 4.9 on $(G, count(e))$
end if
$Children \leftarrow$ All child elements of e
for all $ch \in Children$ **do**
  Apply Algorithm 4.9 on $(ch, count(e)$
end for

4.8 Loading XML Data to a Relational Schema

The loading of the XML data to created relational schema is not truly a part of the XML-to-Relational Mapping Problem, however in a real-world system for storing XML data using RDBMS it cannot be left out. So in this section we provide a brief discussion about the possible load algorithm.

The first idea is to use an algorithm similar to statistics gathering (Algorithm 4.7). When we have a corresponding schema element e_s for the XML document element e and also a parent type or group T_p, we can determine the exact relational table and its column which e should be stored in – the name of the table is the same as the name of T_p and the column name is the name of e. Of course we would have to enhance the algorithm so it also generates and tracks the identification of the instances of type/group T_p and its parent type/group. With that the loading algorithm would be able to construct the whole tuple corresponding to an instance of type/group T_p including a key of the appropriate tuple in the parent type.

4.9 AntMap System Architecture

Figure 4.8 shows a high-level architecture of our solution, called *AntMap*. The key algorithms and procedures used in the application are detailed in previous sections and Chapter 5, here we provide only a brief summary.

The main application module, called simply *Application*, is responsible for managing the whole run of the application and for a communication of other modules.

The application run starts by the *Input Parser* module which takes input XML Schema definition and sample XML documents and queries, parses them and produces NSchema with data statistics and our internal representation of queries.

These structures are then passed to the *Schema Transformator*, which is a module providing the key function of the whole application – it searches for the optimal mapping of the given NSchema to a relational schema. It uses *Cost Estimator* to determine the quality of the particular schemas.

The resulting transformed NSchema is then translated to a relational schema by the *Relational Translator* which is capable of exporting it as a set of SQL CREATE statements in proper order.

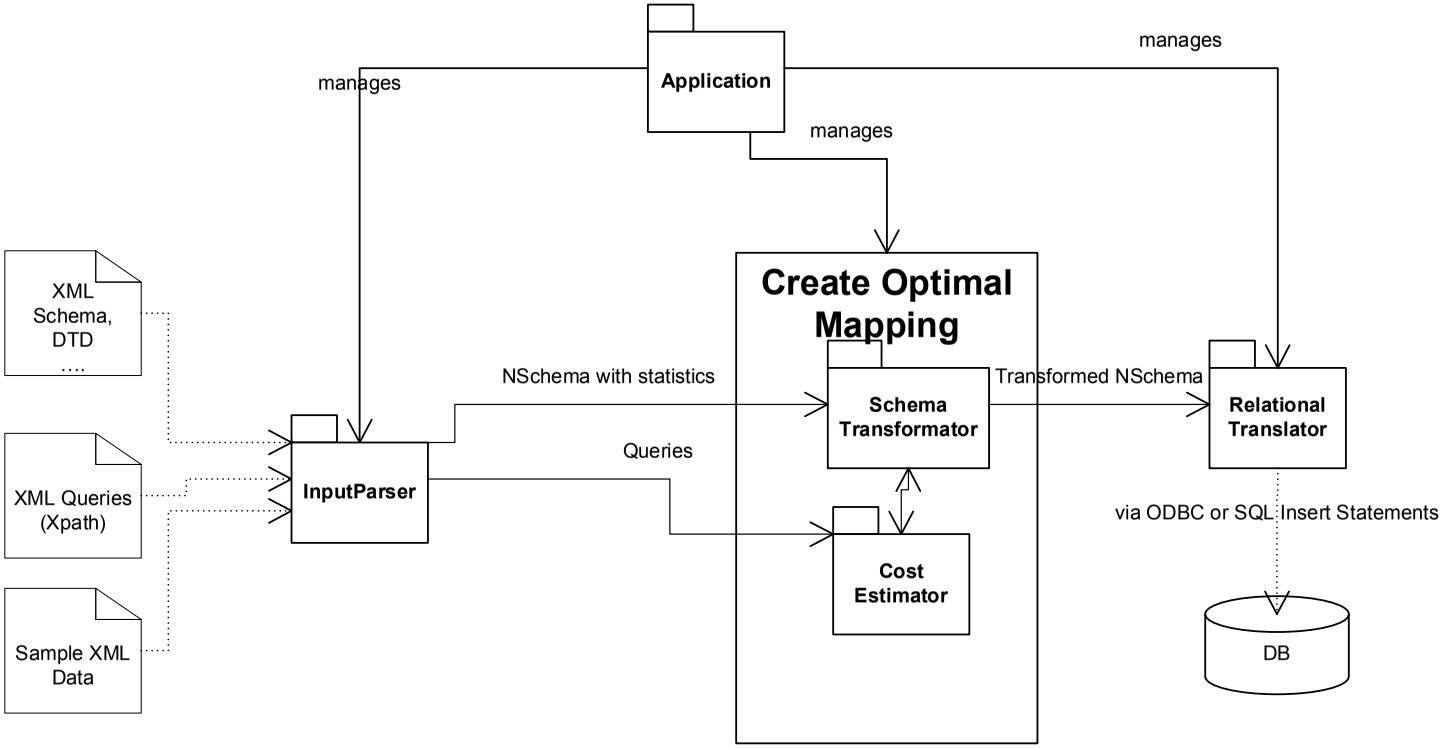


Figure 4.8: AntMap design overview

Chapter 5

Mapping Selection Algorithm

5.1 Choice of Heuristic

The proposed solutions of the XML-to-Relational mapping problem often use quite simple heuristics (e.g. greedy search) and thus suffer from their problems. One of the most painful one is the possibility of getting stuck in some local optimum. Thus we choose a more promising heuristic called Ant Colony Optimization, resp. its variant called Ant Colony System, both of which are described in detail in the following sections.

This heuristic not only solves the local suboptimum problem but has some other nice features. From its nature this algorithm is capable of exploring quite wide area of the search space but also tries to use information about good solutions explored so far to give the search some direction. Also this heuristic has been adjusted and applied to dynamic optimization problems which is especially important for our problem. As mentioned in Section 3.3, in this thesis we want to explore the possibility to adapt the schema dynamically.

5.2 Ant Colony Optimization Metaheuristic

Ant Colony Optimization (ACO) is a family of algorithms for solving various computational problems. First of these algorithms was initially proposed in

[Dorigo et al. (1991)] for the Traveling Salesman Problem and since then there has been many other variants and applications. The main idea is inspired by the behavior of ants looking for food [Dorigo et al. (2006)] – every ant spreads a substance called *pheromone* on its way to and from the food source, other ants notice the presence of pheromone and tend to follow paths with higher concentration of the pheromone. This way the ants in fact communicate or share information they have learnt.

The ACO has been formalized as a metaheuristic (i.e. general-purpose algorithm which can be applied to different problems) for Combinatorial optimization problems. A Combinatorial optimization problem (COP) can be modeled as a set $P = (\mathbf{Sigma}, \Omega, f)$ where

- **Sigma** is a search space defined over a finite set of variables.
- Ω is a set of constraints over these variables.
- $f : \mathbf{Sigma} \to \mathbb{R}_0^+$ is an objective function to be minimized

Using this model, the ACO algorithm can be described as a search for an optimal path in so-called construction graph which is build from a set of solution components. A solution component means an assignment of a particular value to a particular variable.

Algorithm 5.1 General ACO Algorithm

```
Initialization – i.e. creating a set of artificial ants and initialization of pheromones
while final termination condition do
  repeat
    Build partial solutions
    Choose one solution
  until each ant has constructed a solution
  Update path pheromones
end while
```

See Algorithm 5.1 for an overview of the ACO algorithm, important parts of it are detailed in the following list. Note that the outer loop (while loop) is called *iteration* and the inner one (repeat-until loop) is called step.

- Build partial solutions – in every step an ant extends its partial solution by adding some feasible solution component (i.e. the path in the construction graph is extended by adding an edge to it). It first constructs all possible partial solutions which can be derived from the actual one. Then every such partial solution is assigned a probability depending on the quality of it and a pheromone value for the corresponding step (i.e. for the corresponding edge in the construction graph).

- Choose one solution – the ant chooses from the possible steps randomly however with respect to counted probabilities, so the best possible step is most likely to be chosen, however the ant can also choose another one.

- Update path pheromones – after all ants have constructed their solutions, the pheromones of the paths in the construction graph are updated, usually in two steps:

  - Decreasing pheromone values of all paths – this simulates the evaporation of the pheromones of the real ants.
  - Increasing pheromone values on paths leading to good solutions (or letting every ant to increase every pheromone on its path by an amount depending on the quality of constructed solution).

- Final termination condition – can vary depending on the problem, usually it means a predefined number of iterations or reaching a defined threshold quality of the constructed solution.

5.3 Ant Colony System

In our proposed solution we choose *Ant Colony System (ACS)*, a variant of the ACO algorithm introduced in [Dorigo, Gambardella (1997)]. Its main contribution is the introduction of a so called local pheromone update – this is performed after every step of every ant and it diversifies the search performed by subsequent ants so that it is less likely that different ants will construct the same solution. This is quite important in our problem because we implement the whole algorithm based on ACO because we want to improve variants with simple greedy

techniques. If some ants will follow same paths, it reduces the advantage of the algorithm. Another argument for ACS variant is the fact that convergence to the optimal solution has been proved for this variant of ACO (as mentioned in [Dorigo et al. (2006)]).

5.4 Main Mapping Algorithm Using ACS

The structure of the main application algorithm using ACS is presented in Algorithm 5.2, in the next sections we analyze all its steps in detail.

Algorithm 5.2 Main application algorithm

```
Initiate the algorithm
while Final condition do
  Position each ant on a starting schema
  repeat
    Perform a state transition (i.e. build partial solutions, choose one)
    Perform local pheromone updating
  until Iteration termination condition
  Perform global pheromone updating
end while
```

5.4.1 State Transition

First, every ant a (positioned on a schema S_a) constructs all possible steps to take from its current position, i.e. the set of solutions $Sol = \{S|S = t_i(S_a) \wedge t_i \in Trans\}$. Then every such solution is evaluated – its cost $f_{cost}(S)$ is counted and it is given a probability according to it. In original ACO/ACS the probability is counted as defined in equation (5.1):

$$\frac{\tau_{ST}^{\alpha} \cdot \eta_{ST}^{\beta}}{\sum_{U \in Sol} \tau_{SU}^{\alpha} \cdot \eta_{SU}^{\beta}} \tag{5.1}$$

where

- τ_{ST} means the pheromone value for edge ST.

- η_{ST} is a measure of quality of a given step, in our case the difference between costs of the schemas S and T (the lesser $f_{cost}(T)$, the better the step is).
- α and β are parameters which control the relevance of the quality of the step versus the pheromone value.

Note that for the purpose of the implementation of the algorithm the probability itself is not relevant, we only need the *ratio* of probabilities of all possible steps so we can only count the numerator part of the fraction (5.1).

An ant in a schema S then chooses a schema T to move to using so-called *pseudorandom proportional* rule. It depends on a random variable q uniformly distributed over $\langle 0, 1\rangle$ and a parameter q_0 which controls how often the ants explore new possibilities and how often they follow the best possible way. The resulting schema is then chosen as follows:

$$T = \begin{cases} \arg\max_{U \in Sol} \tau_{SU}^{\alpha} \cdot \eta_{SU}^{\beta} & \text{if } q \leq q_0 \\ ACO & \text{otherwise} \end{cases} \tag{5.2}$$

where ACO means the solution is chosen according to rule presented by original ACO – as a random variable with distribution given in equation (5.1). In other words, when $q \leq q_0$, the ant takes the step to the best resulting solution possible, otherwise the schema is chosen in a standard ACO way.

5.4.2 Local Pheromone Updating

After every step performed by an ant the pheromone of the last used edge is updated according to the following rule:

$$\tau_{ST} \leftarrow (1 - \rho) \cdot \tau_{ST} + \rho \cdot \tau_0 \tag{5.3}$$

where

- $0 < \rho < 1$ is a parameter of evaporating for the local pheromone updating.
- τ_0 is the initial pheromone level. Note that the authors of the ACS algorithm also proposed 2 other versions of the formula (5.4), however, the one

presented here has been observed both performing better or equal than the other ones and the simplest so we focus on it.

The local pheromone update prevents possible following ants to take the same step and so diversifies the set of constructed solutions.

5.4.3 Global Pheromone Updating

Only the best Ant a_{best} (i.e. the ant with the best solution from the beginning of the run of the algorithm) deposits a pheromone along its path in this phase. The value of the pheromone for the edge between schemas S and T – τ_{ST} – is counted as follows:

$$\tau_{ST} \leftarrow (1 - \varphi) \cdot \tau_{ST} + \varphi \cdot \Delta\tau_{ST} \tag{5.4}$$

where

- $\Delta\tau_{ST} = \begin{cases} C/f_{cost}(a_{best}) & \text{when edge } ST \text{ belongs to the global best solution found} \\ 0 & \text{otherwise} \end{cases}$
- $f_{cost}(a_{best})$ is the cost of the best solution found
- C is a constant
- $0 < \varphi < 1$ is the pheromone evaporation parameter for the global pheromone updating.

5.4.4 Termination Condition for One Iteration

The ACS algorithm is, in general, defined for combinatorial optimization problems. It assumes the problem is defined over a set of variables and the solution is built from a set of solution components. This situation does not exactly correspond to our problem, because every schema we produce by performing a transformation on some of its nodes is in fact a possible solution of the problem. This forces us to define our own iteration termination condition.

We can define the algorithm to perform only one step (or a constant number of steps) in each iteration, it however seems like a more natural choice to use a

number of iterations dependent on the scale of the problem. This will correspond to the original idea of the ACO metaheuristic where the ants perform steps till they construct the whole solution and this number of steps is of course equal to the size of the construction graph.

We can redefine our problem as a COP to fit ACO better, the COP model can be defined for example this way:

- $\mathbf{S} = \Sigma$, a search space where the decision variables are particular schema nodes, the values assigned to these variables are chains of transformations performed on the node.
- Ω contains only one constraint – any chain of transformation assigned to a node can contain only transformations which can be applied on given node transformed by previous applied transformations.
- $f = f_{cost}$

This definition, however, still does not really divide the solution into a set of components all of which have to be built during an iteration, but it can help us to define the number of iteration dependent on the size of the problem. We can see the components as an assignment of transformations to particular nodes and so we grant every ant a number of steps dependent on the number of types in the initial schema (i.e. the number of nodes which can be transformed) and leave it to every ant whether it would use some transformation on every type or leave some types untransformed while others would contain a chain of transformations.

5.4.5 Placing Ants to a Starting Position

In the original ACO and ACS algorithms, the ant are placed randomly in the construction graph. However, in our situation we cannot actually construct the graph, we build it by performing the transformation.

So we start with all ants in S_{init}. After every iteration, ants $a_1, a_2, \ldots, a_m$ reach positions $S_1, S_2, \ldots, S_m$. The ants with a good solution need to continue their search. But for those with bad solutions it would be better to search from a different point. We exploit both of these ideas and let a half of the ants in their positions, while the rest is placed on the original starting position, i.e. S_{init}.

Another situation is when an ant reaches a schema which cannot be further transformed, we called such an ant dead and remove it from the set of ants for the rest of the iteration. Before the next one, a new ant is created and placed on S_{init}.

5.4.6 Termination Condition for the Whole Algorithm

Another important part of the algorithm design is the final termination condition, i.e. the condition which stops performing iterations and ends the whole algorithm. Usually, there is a predefined maximal number of algorithm iterations and/or a threshold value of f_{cost} which ends the algorithm.

The first idea is to set the number of iterations to a number which depends on the size of the problem, in our case the number of types of S_{init}. Our experiments however showed, that the final solution is found much earlier, so we have proposed additional condition.

The second condition is inspired by the greedy algorithms – they end when they cannot improve the solution any more. Similarly to that, we stop the algorithm when an improving solution has not been found for some number of iterations. We set this number proportionally to the maximum number of iterations defined above.

5.4.7 Parameter Settings

The algorithm presented in previous section has a couple of parameters. We summarize them in the following list together with an explanation of their setting:

- α, β – parameters which influence the determination of the quality of possible steps, in particular the bias between the information from other ants (pheromone) and f_{cost}. We use the settings proposed in [Dorigo, Gambardella (1997)] and, hence, set $\alpha = 1$ and $\beta = 2$.
- ρ, φ – pheromone evaporation coefficients which ensure that some solution from the beginning of the algorithm does not prevent the ants from exploring other possibilities. We set both of these to 0.1, as do authors of [Dorigo, Gambardella (1997)].

- q_0 – parameter for the state transition decision, it diversifies the search and prevents the ants from taking the same steps. The authors of [Dorigo, Gambardella (1997)] set $q_0 = 0.9$ however in our situation this caused many ants to construct exactly the same solutions, so we have performed some experiments (see Section 8.5) and finally set it to 0.5.

- C – a constant which influences the amount of pheromone deposited. Authors of [Dorigo, Gambardella (1997)] simply use 1 but experiments showed that this is not suitable for our situation as the pheromone value is too low (since the costs can be very high). We set it to a cost of the initial schema so that the deposited pheromone value is always greater than 1.

- τ_0 – an initial value of the pheromone, the authors of [Dorigo, Gambardella (1997)] propose a value of

  $$\frac{1}{f_{cost}(S_{est})}$$

  where S_{est} is a solution found by another method. We in fact adjust this equation to

  $$\frac{C}{f_{cost}(S_{init})}$$

  as S_{init} is also a solution and used equation is consistent with the (global) pheromone updating.

- The number of ants – we set it to 10, i.e. the value proposed in [Dorigo, Gambardella (1997)].

Chapter 6

Dynamic Adaptation

As mentioned in Section 3.3, the main problem of all adaptive techniques is that the schema is adapted to future usage only once at the beginning. When the application changes or when the usage differs from samples given by the user at the time of creating database schema, the resulting efficiency may considerably worsen. Consequently, there is a need for a dynamic schema adaptation mechanism.

Note that such a mechanism is not an easy task because the idea naturally implies database schema reconstruction (at least to some extent) which could be very expensive in both time and space. Nevertheless, when we exploit the idea of gradual small (or local) changes, the performance gain could outweigh the cost of reconstructing the schema. The idea is based on a presumption that a real application and its usage would probably also evolve gradually and in cases of radical changes the underlaying data storage have to be reconstructed anyway (as it usually happens in the case of pure database-based applications).

The ACO heuristic has been successfully applied to some dynamic problems and, hence, our mapping selection algorithm using this heuristic can be adjusted to handle dynamic adaptation problems as well.

There are in principle two different possibilities how the situation can change:

1. Change of the input XML schema
2. Change of sample data and/or queries

While the first one may be relevant in some situations, it would require a reconstruction of the whole relational schema or, at least, the data and query mapping mechanism anyway. In our view, the second case is more often since the real usage of an application will usually differ from the predicted one given to the algorithm. And the usage will probably change also while the application is being used.

For our problem, this change means a change of X_{sample}, which implies a change in data statistics, and a possible change of Q_{sample}. Both of these changes influence the computation of the value of f_{cost}.

The adaptation of an ACO algorithm to some dynamic problem is discussed for example in [Angus, Hendtlass (2005)] and [Eyckelhof, Snoek (2002)]. While the papers explore different types of the dynamic adaptation of TSP, they both contain basically the same ideas:

- The ants in the ACO algorithm are capable of adapting to a new situation from the nature of the algorithm – even when some very good solution has been found, some of the ants will still explore different paths because of the state transition rule (see Section 5.4.1).
- When some route in the construction graph has very high values of the pheromones, the exploration of new paths is less probable. This is especially problematic in case of dynamic adaptation. Both papers propose a similar solution of that – when the situation changes, the pheromone values are normalized to remove large differences between the pheromone values, while still preserving the order of the edges according to them.

Using the ideas presented in the previous list, the adaptation of our algorithm is quite simple. We only need to store the information about pheromone values and when the adaptation algorithm is used again with a different set D_{sample}, we use these values instead of the default value τ_0. Before the algorithm is started, the pheromone values are normalized to lower the differences. The normalization rules proposed in [Angus, Hendtlass (2005)] and [Eyckelhof, Snoek (2002)] are presented in equations (6.1) and (6.2):

$$\tau_{ij} \leftarrow \frac{\tau_{ij}}{\tau_{max}} \tag{6.1}$$

$$\tau_{ij} \leftarrow \tau_0 \cdot (1 + \log \frac{\tau_{ij}}{\tau_0}) \tag{6.2}$$

where τ_0 is the initial pheromone level and τ_{max} is the maximal pheromone value set so far. The first one seems to be more effective while still keeping the order of the pheromone values unchanged, so it seems like the more promising choice.

After this normalization the paths of the previous solution are still preferred but new good solutions are likely to be found near the original one. This suits the problem of dynamic schema adaptation in case of gradual and rather small changes of the application. And also note that this way we can obtain not only the new schema, but also an information about the changes from the last one (i.e. the different steps taken in the path), which could be then used for the reconstruction of the relational schema.

While the main mapping selection algorithm can be adjusted to the dynamic version of the problem quite easily, there are other aspects to be considered. Most importantly, even with a slight change in the resulting relational schema, the schema has to be altered and the stored data have to be either moved and adjusted on the way or reimported from the original XML documents. Both of these tasks are difficult to perform automatically and can be quite expensive. Hence the algorithm should also exploit the cost of the schema reconstruction and decide, whether it is worth the benefit of the newly adapted schema. Designing such an algorithm needs a further detailed analysis.

Chapter 7

Implementation

We have implemented the most important parts of proposed algorithm in the Java language (version 1.6)[1] so we can evaluate our solution in experiments (see Chapter 8).

Since we wanted to focus on the algorithm itself, we have used as many of various libraries and tools as possible. In particular, we used Xerces Java XML parser and APIs[2] and Xalan APIs[3] for XPath querying. We keep the schema in a DOM graph trough the whole algorithm and we use standard DOM API to work with it together with XPath expressions to obtain specific nodes from the graph.

7.1 Schema Normalization

The normalization is implemented as a set of DOM graph visitors which have shown as very suitable for the problem. We have implemented a generic XML Schema definition visitor which handles the types of the schema elements and then only reimplemented the particular methods to do a particular part of normalization process.

Other method considered was usage of a standard XML transformation tool – XSLT[W3C: XSLT]. The reason for such a solution is a usage of well defined and

[1] `http://java.sun.com/`
[2] `http://xerces.apache.org/xerces2-j/`
[3] `http://xml.apache.org/xalan-j/`

known technology which is suited for transforming XML documents. Nevertheless, our solution using DOM graphs and visitors enables us to construct advanced structures to help the work with the schema. And more importantly, in XSLT many of the normalization tasks would have to be rather cumbersome.

7.2 Pheromone Handling

Pheromone values for the algorithm have to be stored for every possible step, i.e. for every two schemas S_1, S_2 such that $\exists t_i \in Trans : S_2 = t_i(S_1)$. Nevertheless, to construct a structure which hold all these values from the beginning would be expensive as well as useless since the algorithm usually does not evaluate all possible steps. Hence, we implemented a map (specifically a hash map) which is capable of storing and retrieving the pheromones and in case it does not contain a value for the given schema pair, it returns a default value τ_0.

Of course, the problem with such a map (or any other structure for holding pheromones) is how to index it. In other words, a way to encode the whole schema. Another possible way would possibly be to index the steps by the chains of transformations together with the elements they have been applied on. This would however require proper analysis of equality between two chains of transformations applied on the same initial schema which can be more expensive than the encoding of the schema.

We choose to encode every particular transformed schema into a unique identifier called *schema code*. *Schema code* is a string obtained by a pre-order traversal of the schema DOM representation. The process is sketched in Algorithm 7.1, it is applied on the root of the encoded schema. It uses a lookup table $f_{text} : \{\text{text values in S}\} \rightarrow \mathbb{N}^+$ for all textual values (including the XML tags and attribute names) present in the schema. f_{text} can be extracted from the initial schema since no transformation changes any textual value. To save memory, a hash of the resulting string is used instead of the code itself.

Algorithm 7.1 Schema Encoding

Input: e an schema node, f_{text}
Output: a code of e
$c \leftarrow$ an empty string
for all $ch \in$ Children and attributes of e **do**
  Apply Algorithm 7.1 on ch
  Append the result to c
end for
if e is an attribute **then**
  Append f_{text}(name of e) to c
  Append f_{text}(value of e) to c
else if e is an element **then**
  Append f_{text}(tag of e) to c
else if e is a text node **then**
  Append f_{text}(value of e) to c
end if
return c

Chapter 8

Experiments

To test how the proposed solution of the adaptive XML-to-Relational mapping problem performs and also to fine tune some of the parameters of the algorithm, we conducted a sequence of experiments.

8.1 Experimental Setup

The machine used for the experiments has following specifications:

- Dual core 1.66 GHz CPU
- 3 GB RAM with no page file on the disk
- 5400 rpm hard disk

The algorithm has been implemented in the Java language, the heap space for the virtual machine has been set to 1 GB.

The data used for the experiments were obtained from [XMark], an XML-Benchmark project which provides a tool for generating XML documents according to a predefined DTD. We transformed the DTD to a corresponding XML Schema Definition and used it as S_{init}. A set of sample documents were generated (a total size about 8 MB) to obtain schema statistics and four sets of XPath queries were derived from the ones provided by [XMark]. The first three of them

	Workload-1	Workload-2	Workload-3	Combined Workload
Cost of the initial schema	42 510 422	12 978 202	59 736 854	128 820 398

Table 8.1: Initial costs according to query workloads

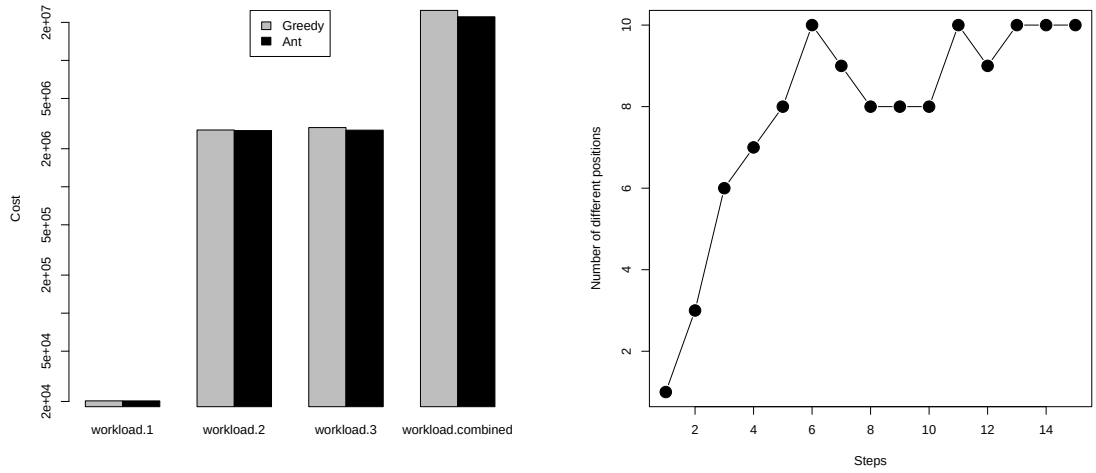
contain a couple of queries related to a subset of entities in the schema, while the fourth one combines the queries from the other three sets.

The costs of the initial schema according to these workloads are presented in Table 8.1.

We have also implemented a simple Greedy algorithm proposed in [Bohannon et al. (2001), Bohannon et al. (2002)] so we can compare the results of our algorithm to another one. In the following sections we refer to these two algorithms simply as Greedy and Ant.

8.2 Overall Performance

First we ran the algorithms on our four workloads to compare them. The graph of resulting costs is depicted in Figure 8.1a. It shows, that although the Ant algorithm gives better results on all of the workloads, the difference between it and the Greedy one is not big. In fact it varies from almost 0% to about 11% and grows with the complexity of the query workload. This supports a hypothesis, that the Greedy algorithm works very well since by taking the best step in its iteration it in fact does not block other good steps from choosing in the following iterations. This is true especially when the queries focus on only a subset of the schema. When the workload become more complex, the Greedy algorithm starts to miss some possibilites and the Ant algorithm becomes better.



(a) Overall performance (b) Diversification of search

Figure 8.1: Performance and diversification of search

8.3 Diversification of the Search

An important feature of the Ant algorithm (especially since we choose the ACS variant of ACO) is the fact that it searches a wide subspace of Σ. To see whether different ants really search different possible paths, we ran the algorithm on one of the workloads and observe the paths created by particular ants. Figure 8.1b summarizes the number of ants in different positions in the first 15 steps of the algorithm. We can see that from the start of the algorithm the ants follow different paths and after a couple of steps about 8 of them are in different places on an average.

Another observation can be made from the behavior of the Ant algorithm – it usually finds more than one solution with the best cost. This can be important in a real usage of the algorithm. Although the variants have the same cost, other characteristics may differ, for example the readability of the relational schema for the users. And the cost model in the algorithm cannot simulate a particular RDBMS absolutely, so tests of different variants in the database can give different real costs.

8.4 Impact of the Set of Transformations

We implemented most of the transformations proposed in related papers (which usually focus on only a subset of them) so we tried to find out how the choice of the transformations influences the resulting cost. Figure 8.2 shows results of both Greedy and Ant using a different sets of transformations: all but the Simplifying unions, all but the Type merge, all but the Outline of simple elements and finally only Inline and Outline of local types were used. Note that workload-3 was used for the whole experiment.

We can see that the difference is not really remarkable which supports the hypothesis that the inlining and outlining are the most important transformations. However, the result can also be caused by f_{cost} function model.

Also note that while Greedy and Ant give almost the same results with the restricted set of transformations, the difference grows with the size of the set. This indicates that the more possibilities to transform the schema the better the Ant algorithm is.

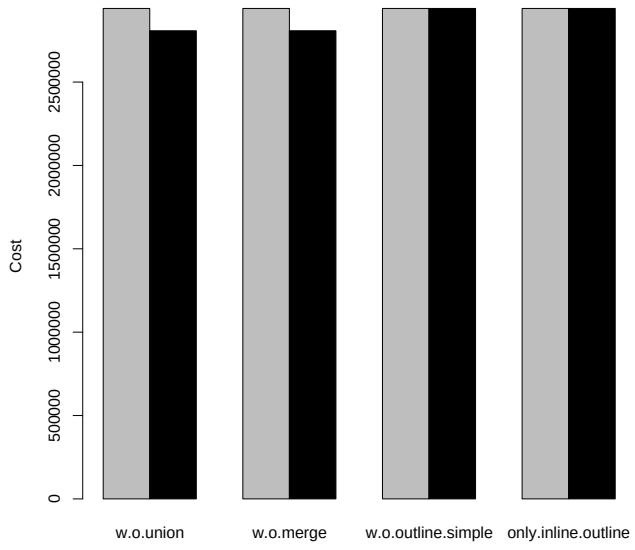


Figure 8.2: Impact of the number of transformations

8.5 Impact of q_0

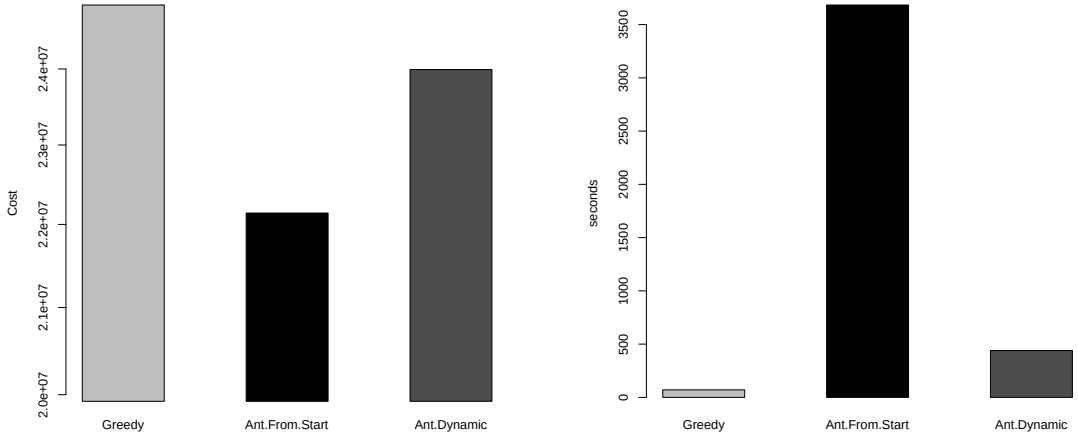
Our Ant algorithm has quite important parameter q_0 which controls how much the ants explore and how much they follow the best possible way. We ran Ant algorithm on workload-3 with a different setting of parameter q_0. Although the experiment showed that the lesser q_0 is the better the result is, the result is not quite conclusive. The difference between the best and the worst result was under 1‰. Nevertheless, there is no disadvantage in setting a lower value and, hence, we used a value $q_0 = 0.5$ for the rest of the experiments since the differences may enlarge with a more complex workload.

8.6 Dynamic Adaptation

Finally we conducted a simple experiment to verify the possibility and benefits of the dynamic adaptation mechanism described in Chapter 6. We let the Ant algorithm to find a solution on workload-3 and then we changed the workload to combined workload. But this time we provided the already used pheromone map to the algorithm (before that we performed the pheromone normalization, see Chapter 6).

Figure 8.3 shows our results. The Subfigure 8.3a compares the resulting costs on the combined workload for Greedy, Ant which has run from scratch and an Ant which used the dynamic mechanism. It can be seen, that the dynamic Ant found a worse solution than the standard one (by about 8%), but it is still better than the one from Greedy (by about 4%). Note that the y scale of Subfigure 8.3a is logarithmic. The second graph (Figure 8.3b) is much more important in our view. It compares the times of the runs of these three algorithms. As we can see, the time of the dynamic Ant is significantly better than the one of the standard Ant algorithm. In fact, it is only 11% of the standard one, i.e. it is almost 10 times better.

This confirms a hypothesis that the ACO-based algorithms for the adaptive XML-to-Relational mapping problems are definitely promising and can be used to solve the dynamic variation of the problem as well.



(a) Cost (b) Time

Figure 8.3: Dynamic version of Ant algorithm

Chapter 9

Conclusions and Future work

In this thesis we analyzed various methods of mapping XML data to a relational schema and conclude that the adaptive schema-driven methods are the most promising ones. Nevertheless, all of these approaches have some drawbacks and we proposed a solution addressing selected ones of them.

The first goal was a choice of more sophisticated heuristic which would address the flaws of the ones used in most of the papers – variants of a simple greedy algorithm. We designed an algorithm based on Ant Colony Optimization and our experiments have shown, that it in fact gives better mappings.

The difference between our algorithm and a simple greedy one is not vast, nevertheless, the problem is quite suitable for the greedy heuristic since it does not really block possible steps by taking the best ones. And, as can be seen from our experiments, the lag between our algorithm and the greedy one grows with the complexity of the query workload used to estimate the cost.

Our algorithm also usually finds more than one solution with the same cost, which can be useful in a practical usage of the mapping system since no cost model can simulate all database management systems perfectly.

Second goal was to evaluate the influence of the selected set of schema transformations on the result. We implemented most of the transformations presented in related papers and conducted an experiment to discover the behavior with different sets. It has shown that although more transformations enable the algorithms to find a better solution, the difference is not radical. We can however see from

the experiment that the more complex situation (i.e. more transformations) suits our algorithm better, i.e. the difference between its result and the result of the greedy algorithm enlarges with the number of transformations.

Finally, we wanted to explore the possibility of a dynamic schema adaptation. Since the ACO algorithms are from their nature well suited to dynamic problems, we could adapt our algorithm to the dynamic version of our problem. We performed a simple experiment to evaluate the dynamic version of the algorithm and the results are promising.

We ran the dynamic version of the algorithm on one query workload first and than again on a changed one. Then we compared its results with both the greedy algorithm and the original version of our algorithm which ran only on the changed workload. Although the dynamic algorithm did not find the best solution, it was still better than the one from the greedy algorithm. And more importantly, the runtime of the dynamic algorithm was almost 10 times better than the time of the original algorithm.

Our future work will focus on further enhancing our algorithm based on ACO. There is a couple of variants of Ant Colony Optimization and it can be interesting to evaluate their ideas in our situation.

But in our view the most important feature enabled by the usage of ACO is the dynamic adaptation. And the need for a dynamic adaptation is simultaneously the most painful problem of the existing XML-to-Relational mapping algorithms. Consequently, further analysis of possible solutions of the dynamic adaptation problem will be our most important future goal.

Bibliography

[Aboulnaga et al. (2001)] A. Aboulnaga, A. R. Alameldeen, J. F. Naughton: *Estimating the Selectivity of XML Path Expressions for Internet Scale Applications*, Proc. of the International Conference on Very Large Data Bases, pages 591-600, Roma, Italy, September 2001.

[Amer-Yahia et al. (2004)] S. Amer-Yahia, F. Du, J. Freire: *A Comprehensive Solution to the XML-to-Relational Mapping Problem*, Proc. of the 6th annual ACM international workshop on Web information and data management, pages 31-38, Washington DC, USA, 2004. `www.cs.utah.edu/$\sim$juliana/pub/shrex-widm2004.pdf`

[Angus, Hendtlass (2005)] D.Angus, T. Hendtlass: *Dynamic Ant Colony Optimisation*, Applied Intelligence vol. 23, pages 33-38, 2005.

[Atay et al. (2007)] M. Atay, A. Chebotko, D. Liu, S. Lu, F. Fotouhi: *Efficient schema-based XML-to-Relational data mapping*, Information Systems, v.32 n.3, pages 458-476, May, 2007.

[Bohannon et al. (2001)] P. Bohannon, J. Freire, P. Roy, and J. Siméon: *From XML Schema to Relations: A Cost-based Approach to XML Storage*, Technical report, Bell Laboratories, 2001.

[Bohannon et al. (2002)] P. Bohannon, J. Freire, P. Roy, and J. Siméon: *From XML Schema to Relations: A Cost-based Approach to XML Storage*, In ICDE '02: Proceedings of the 18th International Conference on Data Engineering, page 64, Washington, DC, USA, 2002.

[Christ, Rundensteiner (2002)] S. Christ, E. A. Rundensteiner: *X-Cube: A fleXible XML Mapping System Powered by XQuery*, Technical Report WPI-CS-TR-02-18, Worcester Polytechnic Institute, 2002.

[Codd (1969)] E.F. Codd: *Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks*, IBM Research Report RJ599, 1969.

[Codd (1970)] E.F. Codd: *A Relational Model of Data for Large Shared Data Banks*, Communications of the ACM 13, no. 6, pages 377—387, 1970.

[Dorigo, Gambardella (1997)] M. Dorigo, L. M. Gambardella: *Ant colony system: A cooperative learning approach to the traveling salesman problem*, IEEE Transactions on Evolutionary Computation, vol. 1, no. 1, pages 53–66, 1997.

[Dorigo et al. (1991)] M. Dorigo, V. Maniezzo, A. Colorni: *Positive Feedback as a Search Strategy*, Technical Report No. 91-016, Politecnico di Milano, Italy, 1991.

[Dorigo et al. (2006)] M. Dorigo, M. Birattari, T. Stutzle: *Ant Colony Optimization - Artificial Ants as a Computational Intelligence Technique*, Technical Report No. TR/IRIDIA/2006-023, IRIDIA, Bruxelles, Belgium, September 2006. `http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2006-023r001.pdf`

[Eyckelhof, Snoek (2002)] C. J. Eyckelhof, M. Snoek: *Ant Systems for a Dynamic TSP: Ants caught in a traffic jam*, Ant Algorithms – Proc. of ANTS 2002 – Third International Workshop, vol. 2463, pages 88–99, 2002.

[Freire et al. (2002)] J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Siméon: *Statix: Making XML count.* In Proc. of SIGMOD, 2002.

[Freire, Siméon (2003)] J. Freire, J. Siméon: *Adaptive XML Shredding: Architecture, Implementation, and Challenges*, Proceedings of the VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop DTWeb on Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web-Revised Papers,pPages: 104 - 116, 2003.

[Harold (2007)] E.Harold: *The State of Native XML Databases*, August 2007. `http://cafe.elharo.com/xml/the-state-of-native-xml-databases`

[Klettke, Mayer (2000)] M. Klettke, H. Mayer: *XML and Object-Relational Database Systems – Enhancing Structural Mappings Based On Statistics*, Lecture Notes in Computer Science, volume 1997, pages 151-164, 2000. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.1766

[Mlýnková, Pokorný (2006)] I. Mlýnková, J. Pokorný: *Adaptability of Methods for Processing XML Data using Relational Databases – the State of the Art and Open Problems*, Technical report 9/2006, Charles University, Prague, Czech Republic, October 2006. http://kocour.ms.mff.cuni.cz/~mlynkova/doc/tr2006-9.pdf

[Mlýnková (2008)] I. Mlýnková: *Standing on the Shoulders of Ants: Towards More Efficient XML-to-Relational Mapping Strategies*, XANTEC '08: Proceedings of the 3rd International Workshop on XML Data Management Tools and Techniques of DEXA '08: 19th International Conference on Database and Expert Systems Applications, pages 279 - 283, Turin, Italy, September 2008.

[Ramanath et al. (2003)] M. Ramanath, J. Freire, J. R. Haritsa, P. Roy: *Searching for Efficient XML-to-Relational Mappings*, XSym 2003: Proc. Proc. of 1st International XML Database Symposium, volume 2824, pages 19-36, Berlin, Germany, 2003. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.7412

[Schmidt et al. (2001)] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, R. Busse: *The XML Benchmark Project*, Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001. http://monetdb.cwi.nl/xml/

[Simanovsky (2008)] A. A. Simanovsky: *Data Schema Evolution Support in XML-Relational Database Systems*, Programming and Computer Software, Vol. 34, No. 1, pages 16–26, 2008.

[SQL:2008] *SQL:2008*, ISO standards ISO/IEC 9075(1-4,9-11,13,14):2008.

[Xiao-ling et al. (2002)] W. Xiao-ling, L. Jin-feng, D. Yi-sheng: *An Adaptable and Adjustable Mapping from XML Data to Tables in RDB*, In Proc. of the VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop DTWeb, pages 117-130, Springer-Verlag, London, UK, 2003.

[XMark] *XMark: The XML-Benchmark Project*, http://monetdb.cwi.nl/xml

[W3C: DOM] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, L. Wood: *Document Object Model (DOM) Level 1 Specification*, W3C Recommendation, October 1998.

[W3C: XML] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau: *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, W3C Recommendation, August 2006. http://www.w3.org/TR/REC-xml

[W3C: XML Schema] D. C. Fallside, P. Walmsley: *XML Schema Part 0: Primer Second Edition*, W3C Recommendation, October 2004. http://www.w3.org/TR/xmlschema-0

[W3C: XPath] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, J. Siméon: *XML Path Language (XPath) 2.0*, W3C Recommendation, January 2007.

[W3C: XQuery] S. Boag, D. Chamberlin, M. F. Fernández, D.a Florescu, J. Robie, J. Siméon: *XQuery 1.0: An XML Query Language*, W3C Recommendation, January 2007.

[W3C: XSLT] J. Clark: *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation, November 1999. November1999

[Wikipedia: Genetic algorithm] *Genetic algorithm*, Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Genetic_algorithms

[Wikipedia: XML schema] *XML schema*, Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Xml__schema

[Zheng et al. (2003)] S. Zheng, J.-R. Wen, H. Lu: *Cost-driven Storage Schema Selection for XML*, Proc. of DASFAA 2003, pages 337-344, Kyoto, Japan, 2003. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.3079

Appendix A

Supplied CD

We included our implementation *AntMap* on the supplied CD together with an electronic form of this document and a couple of other resources.

The structure of the CD is described in the following list:

- The root folder – contains an electronic form of this document called DiplomaThesis.pdf and also an example of input XMS Schema, a normalized version of it and the result of the mapping selection algorithm in the form of both transormed XML Schema and create scripts for a relational schema.
- Implementation
  - AntXMLMap – contains the implementation.
    - doc – contains a Javadoc code reference.
    - experiments – contains various resources and sample files used for the experiments.
    - lib – library files necessary for the work of the application (when compiling from source).
    - src – contains all source files of the implementation.
    - unittest – source files of the unit tests.
    - unittest_data – resources used by the unittests.
  - Documents – contains some documents about the implementation – diagrams of the application, notes etc.

There is a compiled runnable version of the application in the folder AntXMLMap, called AntMap.jar. Its usage is, however, limited since it does not take any attributes but only reads the schema definition in "experiments/auction.xsd", normalizes it ("auction_normalized.xsd") and finds a (sub)optimal mapping of it ("result_schema.sql"). It uses the list of sample documents given in the file "sample_data.txt" and sample queries from file "XPath_query_01.txt".

The set of experiments is also included in a runnable form, called Experiments.jar. Note that for both of these jars the JVM heap size has to be set to 1G. To simplify the run of the applications, batch commands are provided as run.bat and runExperiments.bat.