Uppsala Master's Theses
in Computing Science 270
Examensarbete MN2
2004-03-25
ISSN 1100-1836

Loading XML Schema based data sources into an Object-Relational Database System

Luca Scheuring

Information Technology
Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

Abstract

This report describes the development of an XML Schema data loader that loads XML Schema based data into an Amos II database whose schema has been generated from the data's XML Schema definition. The Waterloo Benchmark XBench is used as test case for the loader. The translator that maps XML Schema definitions into the data model of Amos II was developed in a previous project.

Supervisor and Examiner: Prof. Tore Risch

Passed:

Contents

1 Introduction

1.1 Project Description

XML files are becoming more and more used to provide data exchange over heterogeneous and distributed systems like many of today's Internet applications such as web services. In such an environment it is important to have systems that combine data from many sources and present them in a form that is suitable for a particular user or application. Being an object-oriented multi-database system, Amos II is suitable to achieve this goal.

XML Schema is an XML-based language for defining data types mainly used in XML-based data exchange files on the web. XML Schema definitions can be represented in Amos II by creating appropriate types and functions. In a previous project, such a schema translation program was developed that can parse XML Schema specifications and translate them to corresponding schema definitions in Amos II [JH03]. The purpose of this project was to develop a general wrapper for XML Schema based data that can load any XML data source that is an instance of a translated XML Schema into Amos II. This wrapper should be implemented as set of Java functions that can then be called within Amos II to import data according to the translated schema. The Waterloo XML Benchmark called XBench is used as a test case for the loader.

1.2 Amos II

Amos II is an object-oriented multi-database system developed by the Uppsala Database Laboratory UDBL. Amos II allows executing functional queries over a set of databases distributed over the Internet. The query language is called AmosQL, which has many similarities to the object-oriented parts of the SQL-99 standard. Amos II can be used as well as a stand-alone main-memory object-relational DBMS [RJK00]. The interaction with the Java programming language is realised with two kinds of interfaces called the callin and the callout interfaces [ER00]. With the callout interface, the programmer can define foreign Amos II functions in Java. These foreign functions can then be used in queries like normal Amos II functions. With the callin interface, Java functions are allowed to access and modify the Amos II database by calling AmosQL statements or executing AmosQL functions. There are two ways to call Amos II from Java with the callin interface: The embedded query interface which allows passing strings that contain AmosQL statements and the fast-path interface which allows executing predefined Amos II functions by calling corresponding Java methods. The fast-path interface is much more faster than the embedded interface.

These Java Interfaces allow the implementation of wrappers that can access external data. There already exist a set of wrappers for Amos II, for example wrappers to relational databases [AMWR].

1.3 XML Schema

As XML Documents are stored as plain text, they can be read and be understood by a human reader (if the tag names are meaningful, of course). Let's consider the following piece of XML data:

```
<countries>
       <country id="1">
         <name>United States</name>
         <exchange_rate>1</exchange_rate>
         <currency>Dollars</currency>
     </country>
        <country id="2">
         <name>Sweden</name>
         <exchange_rate>8.54477</exchange_rate>
         <currency>Krona</currency>
     </country>
     <country id="3">
          <name>Switzerland</name>
          <exchange_rate>1.51645</exchange_rate>
          <currency>Francs</currency>
     </country>
<countries>
```

Since the meaning of that data is obvious for a human reader, it is not the case for a program that must deal with such data. We need to define a grammar, which allows an application to know the structure of a document. This definition is called Document Type Definition DTD [W3C00a]. For our example, the DTD looks like:

```
<!ELEMENT countries (country+)>
<!ELEMENT country (name, exchange_rate, currency)>
<!ATTLIST country id CDATA #REQUIRED>
<!ELEMENT currency (#PCDATA)>
<!ELEMENT exchange_rate (#PCDATA)>
<!ELEMENT name (#PCDATA)>
```

DTDs have several flaws. One drawback is that DTDs use a different syntax than XML. Further, DTDs suffer from having very limited capabilities for specifying datatypes. XML Schemas, a W3C recommendation, are a tremendous advancement over DTDs. XML Schema allows defining sophisticated structures including the possibility of derive types using restrictions or extensions. XML Schema has a rich set of built in datatypes, which can be adapted to one's own needs. The XML Schema for our example could look like:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
     <xs:element name="countries">
          <xs:complexType>
               <xs:sequence>
                    <xs:element ref="country" minOccurs="3" maxOccurs="100"/>
               </xs:sequence>
          </xs:complexType>
     </xs:element>
     <xs:element name="country">
          <xs:complexType>
               <xs:sequence>
                    <xs:element ref="name"/>
                    <xs:element ref="exchange_rate"/>
                    <xs:element ref="currency"/>
               </xs:sequence>
               <xs:attribute name="id" type="xs:long" use="required"/>
          </xs:complexType>
     </xs:element>
     <xs:element name="currency" type="xs:string"/>
     <xs:element name="exchange_rate" type="xs:float"/>
     <xs:element name="name" type="xs:string"/>
</xs:schema>
```

There exist XML Schema verifiers that can check whether a given document satisfies a particular XMLSchema specification. For more information about XML Schema refer to [W3C00b].

1.4 XML and Databases

An XML file itself can be called a database in a strict sense of term since it actually is a collection of data. Together with surrounding features like schemas (DTDs, XML Schemas), query languages (XQuery, XPath) and programming interfaces (SAX, DOM), it can be called as "sort of" a Database System (DBS). But it lacks many of the things found in a real database system: efficient storage, indexes, security, transactions, data-integrity, multi-user access and so on [BOU03].

1.4.1 Text versus data centric documents

Perhaps the most important factor in choosing a database is whether you are using the database to store text centric (sometimes also called document centric) or data centric XML content. Data centric documents are documents that use XML as a data transport. They are designed for machine consumption and the fact that XML is used at all is usually superfluous. That is, it is not important to the application or the database that the data is, for some length of time, stored in an XML document. Examples of data centric documents are sales orders, flight schedules, scientific data, and stock quotes. Data centric documents have normally a highly regular structure and no mixed content (that is, if the content of an element type can consist of both text and further nested elements, like <A>123<B>xyz</B>789</A>).

Text centric documents are usually documents that are designed for human consumption. Examples are books, email and help-files. They are characterized by less regular or irregular structure and lots of mixed content. The order in which elements occur is almost always significant [BOU03].

1.4.2 Mapping Document Schemas to Database Schemas

One way to achieve the properties like efficient storage or indexes is to store an XML file in a "normal" database. This means that the document schema has to be mapped to an appropriate database schema.

1.4.2.1 Table Based Mapping

For data centric documents, a table based mapping in a relational database is often a good solution. XML document can be modelled as a single table or a set of tables, depending on the structure of the XML schema. It is also possible to store text centric documents in a relational database. But this mapping is less straightforward since lots of structure and order information has to be added to relations, which leads to overhead [BOU01].

1.4.2.2 Object Relational Mapping

Object-oriented databases can model an XML schema as a tree of objects that are specific to the data in the document. In this model, element types with attributes, element content, or mixed content are generally modeled as classes. Element types with simple content as well as attributes are modeled as

properties. To map an object-oriented schema to an object-relational database, traditional object-relational mapping techniques can be used. That is, classes are mapped to tables, properties are mapped to columns, and object-valued properties are mapped to primary key / foreign key pairs.

It is important to understand that the object model used in this mapping is not the Document Object Model (DOM). The DOM models the document itself and is the same for all XML documents, while the modeling described above models the data in the document and is different for each set of XML documents [BOU03]. The project described in this paper is based on an object-relational mapping, done by a schema importer described in section 1.5.3. See section 1.5.1 for an example where the actual DOM model is mapped to Amos II.

1.4.3 Native XML Databases

For storing text centric documents, (object) relational databases are often inefficient. If a schema has a deep structure but the actual structure in the XML document varies, this results in either a large number of columns with null values (which wastes space) or a large number of tables (which is inefficient). In this case, a native XML database may be the better choice. It stores the entire XML document on a single place on the disk (depends on the actual implementation of the XML database). This increases the retrieval speed especially if whole parts of the document tree and not just singles values are requested. But the increased speed applies only when retrieving data in the order it is stored on disk. If one want to retrieve a different view of the data, such as a list of customers and the sales orders that apply to them, performance will probably be worse than in a (object) relational database. A native XML database also allows the use of XML queries like XPath and XQuery [BOU03].

1.5 Previous and Related Work

1.5.1 General XML Wrapper

There exists a general Amos II wrapper for XML data documented in [ROD02]. It will be referred to as AXE in this project. The approach of AXE is to map the DOM [W3C02] to an appropriate Amos II schema. This mapping can be done in a straightforward manner since Amos II is an object-relational database. This allows preserving the whole structure of an XML document when storing it in Amos II. AXE is capable to evaluate XPath [W3C99] expressions. The using of the DOM as database schema leads to a lot of overhead in the database. Thus, the focus of AXE is on text centric documents where the preservation of the entire structure is important. AXE is not suitable to wrap big data centric documents in Amos II because of the overhead (see 5.3). Another drawback is that all data is stored as character strings in Amos II and therefore no type information is available.

1.5.2 Translating DTDs into the data model of Amos II

The other approach is to map the data model of a particular XML schema into the data model of Amos II. [LRK01] describes how a DTD can be translated to a corresponding Amos II schema. Two kinds of transformation rules are presented: Rules applied on DTDs and rules applied while reading XML data. This allows to dynamically extending the Amos II schema.

As mentioned in section 1.3, DTDs have many disadvantages compared to XML Schema. Hence the need for translating the more powerful XML Schemas to Amos II has emerged. This has lead to a project described in the next section.

1.5.3 Amos II XML Schema Importer (AXSI)

AXSI [JH03] parses XML Schemas and generates appropriate type and function definitions in Amos II. XML Schema simple types like *string*, *float* or *boolean* are mapped to the corresponding types in Amos II. Some XML Schema types don't have an exactly matching type in Amos II. When loading XML data, this can lead to problems (see section 4 for examples). AXSI maps most of the XML types that don't have an exactly matching type in Amos II to *charstring*. The main transformation rules of AXSI are:

- All globally defined XML elements are mapped to an equally named type in Amos II.
- Locally defined XML elements that have attributes or nested elements as the contents are mapped to an equally named type in Amos II.
- Locally defined XML elements that have neither attributes nor nested elements are mapped to property functions of the Amos II type that represents the parent XML element.
- Attributes of an XML element are mapped to property functions of the Amos II type that represent the XML element.

All types in Amos II are created under the root type *XML* and all type and function names have the prefix *XS_*. Further AXSI supports the *extension* feature of XML Schema, which allows defining elements based on existing ones. Since Amos II is an object-relational database, type inheritance is perfectly suitable for representing XML elements that have been defined by derivation (using the *extension* feature). See section 3.5 for further information about inheritance.

AXSI is mainly focused to be used in conjunction with data centric documents. That means, that AXSI is meant to import XML Schemas of data centric documents. However, also schemas of text centric documents can be processed. But in that case, the following restrictions have to be taken into consideration:

- Mixed content is not supported.
- The order of subsequent elements is not preserved.

1.5.4 XQuery Translator

There exists a translator for transforming XQuery expressions into AmosQL statements [HIL04]. However, since XQuery works on path expressions over the document structure, XML Schema information is not needed when evaluating XQuery expressions and the XQuery translator therefore does not use any XML Schema.

1.5.5 XBench

XBench is a family of benchmarks for XML DBMSs. It was developed by the Waterloo University. Since more and more data from various application domains like biology or e-business is stored in XML format, one may have to deal with lots and potentially very big XML files. Researchers in both industry and academia have been focusing on efficiently storing, manipulating, and retrieving XML documents. XBench captures different XML applications characteristics. These applications are categorized as data centric or text centric and the corresponding databases can consist of single documents or multiple documents.

In data centric (DC) applications, the database stores data that are captured in XML even though the original data may not be in XML. Examples include e-commerce catalog data or transactional data that is captured as XML.

Text centric (TC) applications manage actual text documents. Examples include book collections in a digital library, or news article archives. The single document (SD) case covers those databases, such as an e-commerce catalog, that consists of a single document with complex structures (deep nested elements), while the multiple document case covers those databases that contain a set of XML documents, such as an archive of news documents or transactional data [XBEN03]. XBench provides a database generator which can produce databases for these four cases: DC/SD, DC/MD, TC/SD, and TC/MD. The size of these databases can be chosen in the range of 10 MB to 10 GB. For this project, all tests have been done with the 10 MB versions of the databases.

The following table gives an overview of the four databases.

Overview of the XBench databases

Category	Filename	Schema	Size	Description
TC/SD	dictionary.xml	TCSD.xsd	10 MB	Entries from The Collaborative International Dictionary of English and from The Oxford English Dictionary
TC/MD	article1.xml - article26.xml total 26 files	TCMD.xsd	8 KB-1 MB per file total 11 MB	Articles from the Reuters News Corpus and part of the Springer Digital Library
DC/SD	catalog.xml	DCSD.xsd	10 MB	A typical book catalogue with entries like title, type, author, price and size about each book.
DC/MD	customer.xml	DCMDCust.xsd	3 MB	Customer information such as name, contact information and account balance.
	address.xml	DCMDAddr.xsd	1.7 MB	Address information for customer.
	item.xml	DCMDItem.xsd	1.1 MB	Item (book) information, such as item title, subject, publisher, and price.
	author.xml	DCMDAuth.xsd	130 KB	Author information, such as author name and contact information.
	country.xml	DCMDCoun.xsd	11 KB	Country information including name, currency and exchange rate.
	order1.xml - order2592.xml total 2592 files	DCMDOrd.xsd	1-3 KB per file total 4.3 MB	Purchase order information such as order date, price and status.

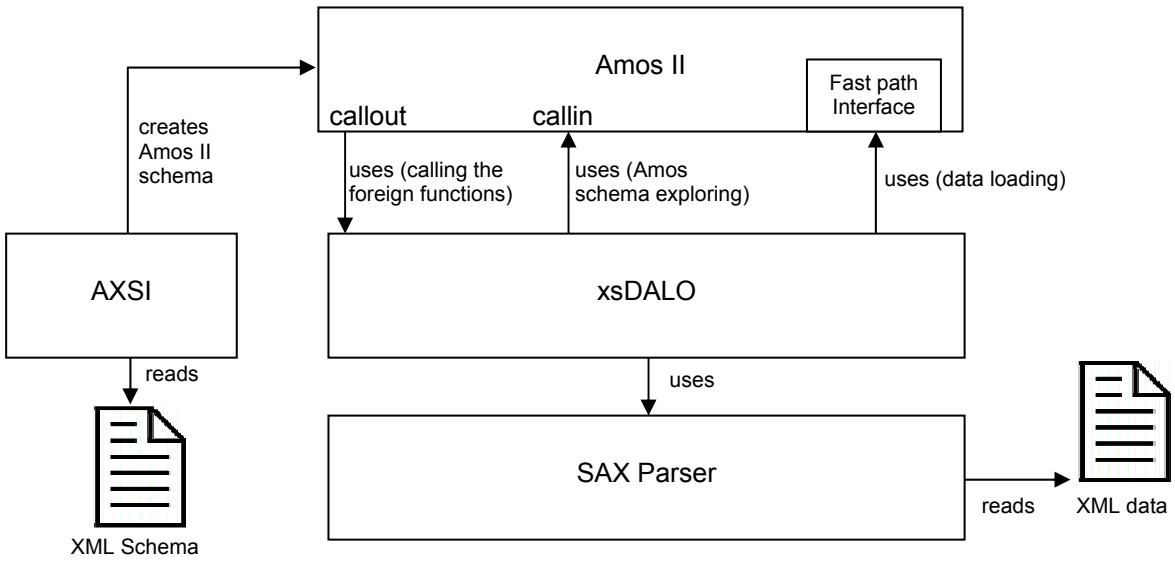
The total size of all files belonging to the DC/MD category is also 10 MB. This category represents a typical collection of business data of an online bookshop.

The workload of the benchmark consists of a set of queries for each of the four databases. Refer to section 3.4 for further information.

2 Architecture and Implementation of the Data Loader

2.1 Architecture

This schema gives a basic overview of the system and shows how the different components work together. The XML data loader developed in this project is called xsDALO (XML Schema Data Loader).



2.2 Implementation

2.2.1 Accessing the Amos II schema

Before parsing of an XML file starts, xsDALO explores the Amos II schema that was generated before by AXSI. The exploring of the schema is possible since AXSI creates all types under the root type named *XML*. Further, all type names and functions created by AXSI have the prefix *XS*. xsDALO creates an internal representation of the relevant part of the Amos II schema. For each Amos II type, an instance of the class *AmosTypeHandler* is created. And each of these instances encapsulates a hashtable that stores all functions (function name is the key, result type is the corresponding value in the hashtable) that can be applied to the represented Amos II type.

2.2.2 SAX

The SAX API [SAXPR] has been chosen for the XML parser of this project. SAX is a simple API for XML that allows parsing a file sequentially and generating events when elements start and end or when attributes or text content was found. SAX was originally a Java-only API (now there are versions for several programming language environments other than Java) and can be seen as a "de facto" standard. There exists another XML interface, called the Document Object Model (DOM) [W3C03]. There is also a Java API for DOM. DOM was not suitable for this project since DOM needs to load the whole XML file in memory. With big XML files, this leads to problems. With SAX and its sequentially nature of parsing, the actual size of the files doesn't matter.

After the Amos II schema has been explored, parsing starts. While parsing, a stack is used to keep track of the document's structure. To convert text content passed by SAX into the appropriate type in Amos II, different implementations of the abstract class *AbstractContentParser* exist. For example, there is a class called *BooleanContentParser*. It is responsible for converting text like *true, false, 1* or *0* into an Amos II boolean type. This design makes xsDALO extensible since other data types can be supported by adding new implementations of *AbstractContentParser.*

2.3 Handling of XML IDs and IDREFs

XML IDs and IDREFs are used to make references within a document. In the schema, an attribute or element can be designated as an ID. But since any name can be chosen for such attributes or elements, special information is needed that allows xsDALO identifying a value as an ID. This is achieved by the convention that AXSI models the ID value not as charstring but as a special type called *XS__ID*. The actual ID is stored in the property function named *XS_VALUE*. IDREF fields can be recognized by xsDALO since their result type is always *XML*. For each ID found during parsing, an entry in a hashtable is done (ID name is key, the OID of the Amos II object attached to that ID is the value in the hashtable). For each IDREF found, an entry is put on a stack containing the IDREF string and the OID of the object holding that IDREF. When parsing is done, all IDREFs get resolved by storing a pointer to the object that is attached to a certain ID in the IDREF field.

Refer to the Javadoc included in the software distribution for further information about the implementation.

3 Using the Implementation

3.1 Function Reference

To use xsDALO, four foreign functions have to be defined in Amos II:

```
JavaAMOS 1> create function loadXMLfromURI(charstring)->charstring as foreign
"JAVA:xsdalo.DataLoaderMain/loadURI";

JavaAMOS 1> create function loadXMLfromDIR(charstring)->charstring as foreign
"JAVA:xsdalo.DataLoaderMain/loadDIR";

JavaAMOS 1> create function refreshAmos IISchema()->charstring as foreign
"JAVA:xsdalo.DataLoaderMain/refreshAmos IISchema";

JavaAMOS 1> create function setLogLevel(integer)->charstring as foreign
"JAVA:xsdalo.DataLoaderMain/setLogLevel";
```

`loadXMLFromURI(charstring URI) -> charstring`	
Description	Loads XML Schema based data into the Amos II database.
Parameter	URI is the location of the XML Data. This can be a local file or a location on the network.
Result	If the data was imported successfully, the message "Loading <URI> done." is returned.

`loadXMLFromDIR(charstring path) -> charstring`	
Description	Loads all XML Schema based data sources that are found in a given directory into the Amos II database.
Parameter	Path of the local directory where the XML files are stored.
Result	If all files were imported successfully, the message "Loaded <Number of files> file(s)." is returned.

`refreshAmosSchema() -> charstring`	
Description	To refresh the schema information in xsDALO before loading a file. Useful when things have been added to the loaded schema when AXSI is run again.
Parameter	(Void)
Result	If command was successful, the message "Amos II Schema will be re-imported before loading the next data source." is returned.

`setLogLevel(integer level) -> charstring`	
Description	Sets the log level of the XML Schema Data Loader.
Parameter	Level specifies the new level to be set. Valid levels are: 1 (ERROR) Errors that stop the parser from loading XML data. 2 (WARNING) Events that keep the parser running but some information from the XML data was not imported properly. 3 (INFO) Informational messages about the loading process. 4 (DEBUG) Verbose information that can be used for debugging. Attention: This may cause a huge amount of log messages and slowdown the data loader.
Result	If the level was set successfully, the message "Log level changed." appears. "Invalid Log Level. Log Level unchanged" is returned if the given level was not valid.

3.2 Settings

The following settings can be done in the global config section at the beginning of the file *xsdalo.DataLoaderMain.java:*

Variable Name	Description	Type	Default Value
defaultLogLevel	The default Log Level.	Integer	`3 (INFO)`
LOG_PREFIX	The prefix of log messages. "%" is replaced by the current log level.	String	`"xsDALO:log(%) "`
XML_READER_IMPL	The SAX parser.	String	`"org.apache.crimson.parser.XMLReaderImpl"`
XML_FILE_EXTENSION	The extension of XML files.	String	`".xml"`
XML_ROOT_TYPE	The supertype of all Amos II types generated by AXSI.	String	`"XML"`
NAME_PREFIX	The prefix of all Amos II type- and function names generated by AXSI.	String	`"XS_"`
ID_TYPE_NAME	The name of the Amos II type that represents XML IDs.	String	`"XS__ID"`
ID_VALUE_NAME	The name of the function that stores the ID value of the "XS__ID" type.	String	`"XS_VALUE"`

3.3 Usage examples

The XML Schema Data Loader xsDALO was tested using Sun's Java version 1.3.1 and "Amos II Beta Release 5, v17". Java 1.4.x is not supported at the moment since SAX does not work properly with Java 1.4.x. Make sure that xsdalo.jar and the jar-file containing the SAX parser (e.g. crimson.jar) are included in the classpath.

Before data is loaded, make sure that the corresponding XML Schema was imported into Amos II using the XML Schema Importer AXSI (see section 1.5.3). The XML data must be both well formed and validated against the XML Schema to make xsDALO work properly. See section 4 for problems and restrictions.

After the foreign functions (see section 3.1) are defined, xsDALO can be used in the following manner:

Change the log level to your needs:

```
JavaAMOS 1> setLogLevel(3);
"Log level changed."
```

Loading data from a local XML file:

```
JavaAMOS 2> loadXMLFromURI("test.xml");
xsDALO:log(3) Start parsing "test.xml".
xsDALO:log(3) Importing Amos II Schema.
xsDALO:log (3) Importing Amos II Schema done.
xsDALO:log (3) Start resolving IDREFS. 91 references to update.
xsDALO:log (3) Done resolving IDREFS. 91 references resolved successfully.
xsDALO:log(3) Parsing "test.xml" done.
"Loading test.xml done."
1.006 s
```

Loading data from a network location:

```
JavaAMOS 2> loadXMLFromURI("http://www.anyhost.com/cgi-bin/test.cgi");
xsDALO:log(3) Start parsing "http://www.anyhost.com/cgi-bin/test.cgi".
xsDALO:log(3) Importing Amos II Schema.
xsDALO:log(3) Importing Amos II Schema done.
xsDALO:log(3) Parsing "http://www.anyhost.com/cgi-bin/test.cgi" done.
"Loading http://www.anyhost.com/cgi-bin/test.cgi done."
1.006 s
```

Loading all XML files from a local directory:

```
JavaAMOS 2> loadXMLFromDIR("orders");
xsDALO:log(3) Start parsing "order1.xml".
xsDALO:log(3) Importing Amos II Schema.
xsDALO:log(3) Importing Amos II Schema done.
xsDALO:log(3) Parsing "order1.xml" done.
xsDALO:log(3) Start parsing "order2.xml".
xsDALO:log(3) Parsing "order2.xml" done.
"Loaded 2 files."
0.543 s
```

Normally, xsDALO imports the database schema from Amos II once when the first XML data file is loaded. The following command forces xsDALO to import the schema again when the next data source gets to be loaded:

```
JavaAMOS 2> refreshAmosSchema();
"Amos II Schema will be re-imported before loading the next data source."
```

There is a demonstration script (xsDALO.bat) provided in the xsDALO distribution. It runs both on Windows 2000 and Windows XP. The script lets the user choose one out of the four XBench test cases or an inheritance test case. Then it runs JavaAMOS, creates the needed foreign functions, imports the XML Schema for the chosen test case into Amos II using AXSI and finally loads the corresponding XML data with xsDALO.

3.4 Querying the XBench data

The workload of the XBench XML benchmark is classified along two dimensions: by functionality and by document class [YAO02]. All queries are expressed in natural language, followed by the corresponding XQuery [BCF03] expression. Since an XQuery wrapper for Amos II isn't available yet, the XQuery expressions needed to be translated into AmosQL manually. As mentioned in section 1.5.3, using AXSI and xsDALO, the Amos II representation of an XML Schema does not allow preserving the entire document structure. Mixed content is not supported and the order in sequences of elements is lost (in fact the order is not really lost since the SAX events occur in the order as the elements appear in the document. Subsequent occurrences of elements are stored in bags within Amos II. Since Amos II preserves the order in bags, the element's order in the bag is the same as in the XML document. But since bags are unordered by concept, one should not rely on the order). Due to the mentioned limitations, some queries are not possible to translate.

Queries	Functionality	Document Class / Notes
Q1	Top level exact match	TC/SD
Q2	Deep level exact match	DC/SD
Q3	Function application	TC/MD
Q4	Relative ordered access	Not possible since element order is not preserved using AXSI and xsDALO.
Q5	Absolute ordered access	Not possible since order is not preserved using AXSI and xsDALO.
Q6	Existential quantifier	DC/SD
Q7	Universal quantifier	DC/SD
Q8	Regular path expressions (unknown element name)	Not possible since a tree based representation of the data is needed to evaluate path expressions.
Q9	Regular path expressions (unknown subpaths)	Not possible since a tree based representation of the data is needed to evaluate path expressions.
Q10	Sorting by string types	TC/SD
Q11	Sorting by non string types	DC/MD

Q12	Document structure preserving	Not possible since document structure isn't entirely preserved using xsDALO.
Q13	Document structure transforming	Not possible since document structure isn't entirely preserved using xsDALO.
Q14	Missing elements	DC/MD If an element is represented as a property function of an Amos II type, it is not possible to decide whether an element is missing or has null value.
Q15	Empty (null) values	TC/MD But it is not possible to decide whether an element is missing or has null value.
Q16	Retrieve individual docs	Not possible since document structure isn't entirely preserved using xsDALO.
Q17	Uni-gram search	DC/MD Only possible if the range of the search consists of one field (not of a part of the XML tree).
Q18	N-gram search	Only possible if the range of the search consists of one field (not of a part of the XML tree).
Q19	References and joins	DC/MD
Q20	Data type Cast	DC/SD Cast is not needed since types in Amos II are based on the type information found in the XML Schema.

Of each query class, one query has been executed over one of the four database classes if possible. The query specifications in natural language as well as the corresponding AmosQL statements can be found in Appendix A. Since the data in the XBench database consist mostly of random characters, the result of the queries isn't showed. All query statements are available in the xsDALO software distribution in the "AmosQL"-directory.

Since AXE (see section 1.5.1) supports the execution of xpath expressions, the queries that are based on the document's structure or need ordered access, could alternatively be executed using the AXE project.

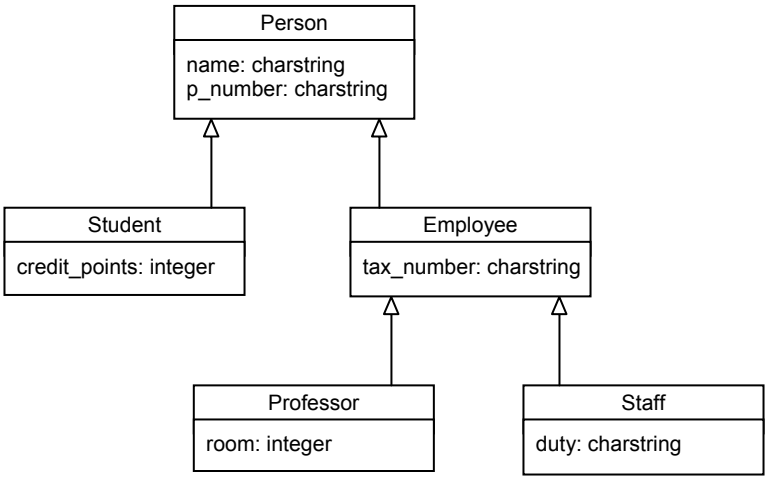
3.5 Inheritance in the XML Schema

3.5.1 Test case

XML Schema introduces new features that increase the level of reusability and extensibility of schema definitions. For example, a schema author can create an element type that extends or restricts an existing one. This principle, called inheritance, allows the user to develop XML Schemas that best suit their needs, without building an entirely new vocabulary from scratch.

AXSI supports the *extension* feature of the XML Schema language. Since extensions of XML types are not used in the schemas of the Waterloo Benchmark, an own test case was developed to test and demonstrate the behavior of xsDALO when inheritance occurs in the Amos II schema.

The test case consists of an extremely simplified people directory of a fictive university. The directory contains of entries for professors, students and personnel. This UML diagram shows the chosen type hierarchy:



This cutout of the XML Schema shows the definition of the type *student*:

```
<xs:element name="student">
     <xs:complexType>
          <xs:complexContent>
               <xs:extension base="person">
                    <xs:sequence>
                         <xs:element name="credit_points" type="xs:int" maxOccurs="1"/>
                    </xs:sequence>
               </xs:extension>
          </xs:complexContent>
     </xs:complexType>
</xs:element>
```

AXSI generates the following type definitions in Amos II:

```
create type XML;
create type XS_person under XML;
create type XS_employee under XS_person;
create type XS_staff under XS_employee;
create type XS_professor under XS_employee;
create type XS_student under XS_person;
```

This test database can be loaded using the demonstrations script of the software distribution.

3.5.2 Implementation in xsDALO

Before xsDALO starts parsing an XML document, it imports the Amos II schema using the AmosQL functions *allsubtypes(Type t)* and *allfunctions(Type t)* provided by the callin interface (see section 2). The problem is that *allfunctions(Type t)* doesn't return the functions that type *t* has inherited from a supertype. So for each type, all functions of its supertypes must be added to that type's function

lookup table. This is done by traversing the type hierarchy in a bottom-up manner till the *XML* type is reached, which is the supertype of all types generated by AXSI.

Consider the people directory database of the test case presented in the previous section. When the *p_number* of a student needs to be set, one may use the *addFunction*[1] in the following way:

addFunction("XS_p_number", tuple1, tuple2);

Note that only the function name is specified, not the whole signature. This makes use of Amos II' late binding capability. But since late binding needs time, it is more efficient to pass the whole signature of a function to *addFunction*:

*addFunction("**XS_person**.XS_p_number->charstring", tuple1, tuple2);*[2]

For high performance xsDALO therefore keeps track of the most specific type of each function to update and selects the appropriate resolvent based on this. In the latter case, the most specific type is *XS_person* (see UML diagram on page 17).

Note that the following function call would lead to an error since this signature is not recognized by Amos II:

*addFunction("**XS_student**.XS_p_number->charstring", tuple1, tuple2);*

[1] The fastpath Java interface method *addFunction* adds an element to a bag stored as the result of an Amos II function. In this case here, *tuple1* is the argument and *tuple2* is the result of the function *XS_p_number*.

[2] *XS_person.XS_p_number->charstring* is the full signature of the function *XS_p_number* since it specifies both argument and result type. The *XS_*-prefix in the function- and type names are determined by AXSI (see section 1.5.3).

4 Problems and Restrictions

There exist several problems and restrictions. Some are caused by limitations of the Amos II system, others by AXSI's way of translating a schema to Amos II. The following list is by no means complete, but it shows what type of restrictions exists and what kind of problems one have to consider. Some of them occur when loading the XBench data.

Description	*Long* type of XML Schema cannot be truly mapped to an Amos II type.
Cause	*Long* is a 64 bit signed integer. Amos II only supports 32 bit integer.
Behaviour of current implementation	AXSI maps *long* types to integer in Amos II. If xsDALO encounters a *long* value in the XML document outside the range that is supported in Amos II, a warning is printed out and the element is skipped.
Workaround or solution	Unless there is no 64 bit integer support available in Amos II, *long* types can simply be replaced by a string type. But this makes e.g. mathematical operations impossible.

Description	Element names that are only distinguishable by having different capital letters cannot be represented in Amos II.
Cause	Type- and function names in Amos II are case insensitive.
Behaviour of current implementation	If e.g. the types of two different XML elements (indistinguishable in Amos II) are different, parsing errors can occur since only one representation exists in Amos II and therefore maybe not all data can be mapped properly.
Workaround or solution	Element names that are only distinguishable by having different capital letters should be avoided in XML Schema. One solution would be to use a sort of a hash function to create the type names in Amos II. But this would make querying more complicated since a user had to be aware of the hash function in order to write correct query statements.

Description	Date values before the year 1970 and after 2038 cannot be stored.
Cause	The Amos II types *date* and *datetime* cannot store date values outside this range.
Behaviour of current implementation	A warning is printed out and the element is skipped.
Workaround or solution	Define these types as strings in the XML Schema.

Description	Mixed content is not supported.
Cause	AXSI does not support XML elements with mixed content.
Behaviour of current implementation	Some of the content gets lost.
Workaround or solution	There is no workaround.

Description	Sequences of XML elements in an instance document are ordered. But order preservation is not supported by AXSI and xsDALO.
Cause	Sequences are stored as a bag of objects in Amos II.
Behaviour of current implementation	The objects are added to the bag according to the occurrence of the corresponding elements in the XML file. Since the order of elements inside a bag normally doesn't change, the order is preserved but one cannot rely on it since bags are unordered by concept.
Workaround or solution	Don't rely on the order of objects in bags. The XML Schema could be extended by adding a position attribute to the elements whose order is important. Alternatively, the behaviour of AXSI could be changed (e.g. using vectors instead of bags to store sequences).

Description	If two equally named and locally defined elements have attributes or nested elements, data can get lost or cast errors may occur when loading XML data.
Cause	AXSI creates an Amos II type for each locally defined element if it has attributes or nested elements. If two such elements have the same name but not the same attribute (types) or nested elements, problems can occur when functions get redefined.
Behaviour of current implementation	See Appendix B for an example.
Workaround or solution	Use different element names in the XML Schema if that problem occurs.

There exist a lot of similar problems like:

- Numbers as elements names are allowed in XML Schema but not possible in Amos II.
- Element names containing the "-" character cannot be represented with Amos II types having the same name since this character is not allowed in Amos II.
- Time zone information is lost when importing date values with time zone info into Amos II.
- If the name of an attribute is equal to the name of a local element nested in the element having that attribute, data is lost or cast errors may occur since both the attribute and the local element are represented with the same property function in Amos II.

Since these problems have similar causes and also similar workarounds or solutions than the ones described in detail above, they are not discussed here in more detail.

5 Performance Measurement and Storage Efficiency

5.1 Test environment

Hardware:	Pentium M Processor 1.4 GHz, 640 MB RAM
OS:	Windows XP Professional Edition
Amos II version:	Beta Release 5, v18
Java version:	Java 2 Standard Edition (build 1.3.1_09-b03)
Settings:	Amos II logging turned off, log level of xsDALO set to WARNING.

5.2 Modifying the Waterloo Schemas

To measure how efficient XML documents are stored in Amos II using AXSI and xsDALO, the four different XBench test databases have been loaded and then dumped to disk using the *save* command. The file sizes of these images can be taken as a measurement of the database size. The initial image size of Amos II (~ 3 MB) has been subtracted from the measured value. This gives us the effective database size. The overhead that is generated when storing XML files in Amos II is calculated as follows:

$$Overhead = \frac{effective\ image\ size\ -\ size\ of\ XML\ file(s)}{size\ of\ XML\ file(s)}$$

The size of the XML file(s) is about 10 MB in all four cases. See section 1.5.5 for details.

It turned out that the Overhead is very big (up to 160%). The main reason behind it is that the XML Schemas that provided by XBench have a very flat design. Almost all element types are defined globally; the tree structure is achieved by inserting references to other element types.

The opposite approach is – sometimes called Russian Doll design – to define the tree structure by nesting the elements. All nested element types are defined locally in this case. Normally it is recommended to use the first approach. If element types are defined globally, their definition can be reused. This makes the schemas more extensible. The Russian Doll design can sometimes also be disturbing for a human reader if deep structures exist [VLIS02]. But the flat schema design has a flaw in conjunction with AXSI: Since every globally defined element type can become root element in a corresponding instance document, an Amos II type has to be created for each of this element types. So if for example one has an XML schema describing persons, an Amos II type is created for each of the fields like name, address and so on. For each of these Amos II types, a property function must be defined to store the actual value. The more straightforward way would be defining an Amos II type *person* and add property functions for name and address, which reduces the database size drastically.

For this project, all Waterloo schemas have been modified by to investigate possible space gains. This was done by encapsulating as much elements as possible in the Waterloo schemas (partial Russian Doll model). In the modified schemas, all leaf elements that don't have attributes are defined locally.

For these elements, AXSI doesn't create an own type in Amos II but a property function for the parent element. This also simplifies the queries (e.g. *select XS_NAME(p) from XS_PERSON p* instead of *select XS_NAME(XS_NAME(p)) from XS_PERSON p*). How modifying the schema reduces the database size can be seen in the following table.

		TC/SD	TC/MD	DC/SD	DC/MD
original XML Schema	effective database size	25.6 MB	17.6 MB	27 MB	26.4 MB
	Needed time to load file(s)	31.3 sec	6.4 sec	30.8 sec	40.6 sec
	Overhead	149%	59%	157%	161%
modified XML Schema	effective database size	15.3 MB	15.2 MB	18.8 MB	14.11
	Needed time to load file(s)	18.8 sec	3.5 sec	19.3 sec	27.8 sec
	Overhead	48%	37%	78%	40%

Note: When loading the TC/SD test case, all character content of elements with mixed content model is skipped and therefore not stored in Amos II. The time measurements indicated in the table are average values built from the five test runs.

Obviously modifying the schema leads to a drastic reduction of the database size. The time needed to parse the XML files decrease in the same manner since less Amos II operations have to be done (less instances need be created).

5.3 Comparison with AXE

AXE uses a completely different approach to store XML documents in Amos II. Instead of translating the XML Schema to an Amos II schema as it is done in AXSI, AXE loads all XML files using the same Amos II schema. This schema models the tree like structure of an XML document, the DOM. So AXE focuses on the structure and order of XML data while AXSI and xsDALO concentrate more on the data and datatypes. As mentioned in section 3.4 some applications or users need access to the order of an XML document. Others may need support for mixed content. AXE and xsDALO can therefore be seen as two complementary projects. Considering this fact, some comparisons between the two projects seem appropriate.

For this test, three subparts of the DC/MD database have been chosen since loading big XML files with AXE needs a tremendous amount of time.

		address.xml	customer.xml	item.xml
xsDALO	Overhead using xsDALO	52%	38%	131%
	Needed time	7.3 sec	4.3 sec	1.6 sec
AXE	Overhead using AXE	355%	240%	285%
	Needed time	376 sec	681 sec	123 sec

Note: The AXE version used for this test is an optimized one (clustering is used for space reduction). Schema validation in AXE was turned off. For xsDALO, the modified versions of the XML Schemas have been used for generating the Amos II schema.

5.4 Clustering

Functions in Amos II can be clustered by creating multiple result stored functions, and then each individual function can be defined as a derived function. For example, let's assume that a *person* is defined as follows in the XML Schema:

```
<xs:element name="person">
     <xs:complexType>
          <xs:sequence>
               <xs:element name="name" type="xs:string"/>
               <xs:element name="address" type="xs:string"/>
               <xs:element name="phone" type="xs:string"/>
          </xs:sequence>
          <xs:attribute name="id" type="xs:long" use="required"/>
     </xs:complexType>
</xs:element>
```

AXSI would generates this types and function:

```
create type XS_person under XML;
create function XS_name(XS_person)->charstring as stored;
create function XS_address(XS_person)->charstring as stored;
create function XS_phone(XS_person)->charstring as stored;
create function XS_id(XS_person)->integer as stored;
```

To cluster the properties of *persons* one could instead define:

```
create type XS_person under XML;
create function XS_personprops(XS_person p) ->
<charstring XS_name, charstring XS_address, charstring XS_phone, integer XS_id> as stored;
create function XS_name(XS_person p) -> charstring nm as
select nm from charstring addr, charstring ph, integer id
where XS_personprops(p) = <nm,addr,ph,id>;
create function XS_address(XS_person p) -> charstring addr as
select addr from charstring nm, charstring ph, integer id
where XS_personprops(p) = <nm,addr,ph,id>;
create function XS_phone(XS_person p) -> charstring ph as
select ph from charstring nm, charstring addr, integer id
where XS_personprops(p) = <nm,addr,ph,id>;
create function XS_id(XS_person p) -> charstring id as
select id from charstring nm, charstring addr, charstring ph
where XS_personprops(p) = <nm,addr,ph,id>;
```

Clustering does not improve the execution time performance significantly in a main memory DBMS such as Amos II. However, clustering can decrease the database size considerably [FJK03]. Clustering is not supported by AXSI and therefore not implemented in xsDALO at the moment. This could be done as a future work since clustering would both reduce the database size and accelerate the parsing process since less callin operations are needed (one *addfunction* call to set all properties instead one call per property).

6 Conclusions and Future Work

This project has shown that AXSI and xsDALO provide a suitable way of storing data centric document in Amos II. However, for text centric documents with mixed content or relevant element order, further work has to be done or a different approach has to be chosen.

Another conclusion is that the design of the XML schema (either flat or in a Russian Doll manner) determines the resulting Amos II schema and creates significantly different storage overhead. This could be avoided if one would combine the schema importer and the data loader to one program. Having access to both the schema and the instance document at the same time, one could make decisions whether an element type needs to be represented as an own Amos II type or not.

Storing data centric documents in Amos II using XML Schema information creates significantly less overhead than using a pure DOM-based schema as in AXE.

The problems and restrictions shown in section 4 make clear that the processes of schema importing and data loading need human supervision and cannot be done completely automatically.

6.1 Comparison with state-of-the-art implementations

There exist native XML databases as presented in section 1.4.3. While they are more suitable for storing and accessing text centric documents, storing and querying data centric documents in Amos II can be faster and more efficient. With its mediator functionality and the Java interfaces, Amos II is perfectly suitable for XML handling in conjunction with for example web services.

All big database products like Oracle or DB2 nowadays provide support for XML data [ORA02]. If XML can be handled directly by the database, it is called an XML enabled database. Sometimes third party software, called middleware, is needed (xsDALO can also be seen as middleware). This approach allows the use of (eventually modified) SQL to query the XML data.

6.2 Future Work

The direction of future work could be towards addressing the following issues:

- Improve support for text centric documents, e.g. by using vectors to preserve order and by supporting mixed content. This would lead to changes in both the AXSI and xsDALO implementation.
- Implement clustering for faster parsing and more efficient storing in Amos II. Again, both AXSI and xsDALO would need to be changed.
- Create a more sophisticated test case for inheritance support within AXSI and xsDALO.

7 References

[AMWR] http://user.it.uu.se/~udbl/amos/wrappers.html

[BCF03] Boag S., Chamberlin D., Fernandez M., Florescu D., Robie J., Siméon J. (2003): XQuery 1.0: An XML Query Language http://www.w3.org/TR/2003/WD-xquery-20031112

[BOU01] Bourret R. (2001): Mapping DTDs to databases http://www.xml.com/pub/a/2001/05/09/dtdtodbs.html?page=3

[BOU03] Bourret R. (2003): XML and Databases http://www.rpbourret.com/xml/XMLAndDatabases.htm

[ER00] Elin, D. / Risch, T. (2000): Amos II Java Interfaces.

[FJK03] Flodin S., Josifovski V., Katchaounov T., Risch T., Sköld M., Werner M. (2003): Amos II User's Manual, Clustering http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html#62005

[HIL04] Hilka T. (2004): Translating XQuery expressions to Functional Queries in a Mediator Database System http://user.it.uu.se/~udbl/publ/XQueryTrans.pdf

[JH03] Johansson T., Heggbrenna R. (2003): Importing XML Schema into an Object-Oriented Database Mediator System http://user.it.uu.se/~udbl/publ/AXSI.pdf

[LRK01] Lin H., Risch T., Katchaounov (2001): Adaptive Data Mediation over XML Data http://user.it.uu.se/~torer/publ/jass01.pdf

[ORA02] Oracle XML DB http://www.cs.utah.edu/classes/cs5530/oracle/doc/B10501_01/appdev.920/a96620/xdb01int.htm

[RJK00] Risch, T. / Josifovski, V. / Katchaounov, T. (2000): AMOS II Concepts. http://user.it.uu.se/~udbl/amos/doc/amos_concepts.html)

[ROD02] Roduner C. (2002): Accessing XML data from an Object-Relational Database ftp://ftp.csd.uu.se/pub/papers/masters-theses/0235-roduner.pdf

[SAXPR] The SAX Project http://www.saxproject.org/

[VLIS02] van der Vlist E. (2002): XML Schema

[W3C00a] Bray, T. / Maler, E. / Paoli, J. / Sperberg-McQueen, C. M. (2000): Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/2000/REC-xml-20001006

[W3C00b] Fallside, D. (2001): XML Schema, W3C Recommendation http://www.w3.org/TR/xmlschema-0/

[W3C02] Le Hégaret P., Whitmer R., Wood L. (2002): Document Object Model (DOM) http://www.w3.org/DOM/

[W3C03] Le Hégaret P. (2003): Document Object Model (DOM)

http://www.w3.org/DOM/

[W3C99] Clark J., DeRose S. (1999): XML Path Language (XPath)
http://www.w3.org/TR/xpath

[XBEN03] Yao B., Özsu M. T. (2003): XBench – A family of benchmarks for XML DMBSs
http://db.uwaterloo.ca/~ddbms/projects/xbench/

[YAO02] Yao, B. (2003): XBench Workload
http://db.uwaterloo.ca/~ddbms/projects/XBench/Workload.html

Appendix A: Queries

Query No.	1	Document Class	TC/SD
Query Text	Return the id of the entry that has matching headword ("the").		

AmosQL	`select xs_value(xs_id(e))`
	`from xs_e e`
	`where xs_hw(xs_hwg(e)) = "the";`		

Query No.	2	Document Class	DC/SD
Query Text	Find the title of the item, which has matching author first name (Bent).		

AmosQL	`select xs_title(i)`
	`from xs_item i`
	`where xs_first_name(xs_name(xs_author(xs_authors(i)))) = "Bent";`		

Query No.	3	Document Class	TC/MD
Query Text	Group articles by date and calculate the total number of articles in each group.		

AmosQL	`create function getDistinctDates()-> bag of date as`
	`  select distinct(xs_date(xs_dateline(xs_prolog(a))))`
	`  from xs_article a;`
	
	`create function articlesPerDate(date d)-> <date, integer> as`
	`  select d, count(a)`
	`  from xs_article a`
	`  where xs_date(xs_dateline(xs_prolog(a))) = d;`
	
	`articlesPerDate(getDistinctDates());`		

Query No.	6	Document Class	DC/SD
Query Text	Return item information where some authors are from certain country (Canada)		

AmosQL	`select xs_value(xs_id(i)), xs_title(i)`
	`from xs_item i`
	`where xs_name_of_country(`
	`  xs_mailing_address(xs_contact_information(xs_author(xs_authors(i))))`
	`) = "Canada";`		

Query No.	7	Document Class	DC/SD
Query Text	Return item information where all its authors are from certain country (Canada).		

AmosQL	`select xs_value(xs_id(i)), xs_title(i)`
	`from xs_item i`
	`where notany(`
	`  select x`
	`  from xs_item x`
	`  where x=i and xs_name_of_country(xs_mailing_address(`
	`  xs_contact_information(xs_author(xs_authors(i))))) != "Canada"`
	`);`		

Query No.	10	Document Class	TC/SD
Query Text	List the words and their pronunciation, alphabetically, quoted in a certain year (1900).		
AmosQL			

```
in(
     sort(
          (
               select xs_hw(xs_hwg(e)), xs_pr(xs_hwg(e))
               from xs_e e
               where xs_qd(xs_q(xs_qp(xs_s(xs_ss(e))))) = 1990
          ),
          1,
          "inc"
     )
);
```

Query No.	11	Document Class	DC/MD
Query Text	List the orders (order id, order date and order total), with total amount larger than a certain number (11000.0), in descending order by total amount.		
AmosQL			

```
create function compare_order(xs_order o1, xs_order o2) -> boolean as
     select TRUE
     where xs_total(o1) < xs_total(o2);

create function cast_to_xs_order(object o) -> xs_order as
     select o;

select xs_id(o), xs_order_date(o), xs_total(o)
from xs_order o
where o = (
     (cast_to_xs_order(
          in(
               sort(
                    (
                         select o
                         from xs_order o
                         where xs_total(o) > 11000
                    ),
                    'compare_order')
               )
          )
     )
);
```

Query No.	14	Document Class	DC/MD
Query Text	List the ids of orders that only have one order line.		
AmosQL			

```
select xs_id(o)
from xs_order o
where count(xs_order_lines(o)) = 1;
```

Query No.	15	Document Class	TC/MD
Query Text	List author names whose contact elements are empty in articles.		
AmosQL			

```
select xs_name(a)
from xs_author a
where (notany(xs_email(xs_contact(a))) and notany(xs_phone(xs_contact(a))));
```

Query No.	17	Document Class	DC/MD
Query Text	Return the ids of authors whose biographies contain a certain word ('hockey').		

AmosQL	`select xs_id(a)`
	`from xs_author a`
	`where like(xs_biography(a), "*hockey*");`		

Query No.	19	Document Class	DC/MD
Query Text	For a particular order with id attribute value (7), get its customer name and phone, and its order status.		

AmosQL	`select xs_first_name(c), xs_last_name(c), xs_phone_number(c), xs_order_status(o)`
	`from xs_order o, xs_customer c`
	`where xs_id(o) = 7 and xs_customer_id(o) = xs_id(c);`		

Query No.	20	Document Class	DC/SD
Query Text	Retrieve the item title of items whose size (length*width*height) is bigger than certain number (500000).		

AmosQL	`select xs_title(i)`
	`from xs_item i`
	`where (`
	`  xs_length(xs_length(xs_size_of_book(xs_attributes(i)))) *`
	`  xs_width(xs_width(xs_size_of_book(xs_attributes(i)))) *`
	`  xs_height(xs_height(xs_size_of_book(xs_attributes(i))))`
	`) > 500000;`		

Appendix B: Problem Demonstration

This is a demonstration of one of the problems described in section 4. Consider the following element definitions (part of an XML Schema):

```
<xs:element name="book">
     <xs:complexType>
          <xs:all>
               <xs:element name="title" type="xs:string"/>
               <xs:element name="author" type="xs:string"/>
               <xs:element name="detail_information">
                    <xs:complexType>
                         <xs:all>
                              <xs:element name="id" type="xs:string"/>
                              <xs:element name="language" type="xs:string"/>
                         </xs:all>
                    </xs:complexType>
               </xs:element>
          </xs:all>
     </xs:complexType>
</xs:element>
<xs:element name="cd">
     <xs:complexType>
          <xs:all>
               <xs:element name="title" type="xs:string"/>
               <xs:element name="interpret" type="xs:string"/>
               <xs:element name="detail_information">
                    <xs:complexType>
                         <xs:all>
                              <xs:element name="id" type="xs:int"/>
                              <xs:element name="language" type="xs:string"/>
                         </xs:all>
                    </xs:complexType>
               </xs:element>
          </xs:all>
     </xs:complexType>
</xs:element>
```

The element *detail_information* is defined locally in the XML Schema, not as a global type. Therefore this element name can be used several times in the same schema. Both element definitions for *book* and *cd* contain the element *detail_information*. But the type of the element *id* is different in the two cases. AXSI generates the following Amos II schema:

```
create type XML;
create type XS_book under XML;
create type XS_detail_information under XML;
create type XS_cd under XML;
create function XS_title(XS_book)->charstring as stored;
create function XS_author(XS_book)->charstring as stored;
create function XS_detail_information(XS_book)->XS_detail_information as stored;
create function XS_id(XS_detail_information)->charstring as stored;
create function XS_language(XS_detail_information)->charstring as stored;
create function XS_title(XS_cd)->charstring as stored;
create function XS_interpret(XS_cd)->charstring as stored;
create function XS_detail_information(XS_cd)->XS_detail_information as stored;
create function XS_id(XS_detail_information)->integer as stored;   ← function XS_id gets redefined
create function XS_language(XS_detail_information)->charstring as stored;
```

As one can see, the function *XS_id* gets redefined. If now xsDALO parses a corresponding instance document, a parse error will occur when xsDALO tries to store the ID of a book that contains non-numerical characters.

One of the XBench schemas (DC/SD) contains equally named and locally defined elements (*contact_information*) that have a different content. But in this particular case it isn't really a problem since the elements that are nested in these two equally named elements have different names. So AXSI generates something like a union of the two element definitions. However, when parsing a corresponding instance document, some fields in Amos II will stay nil which sometimes is unnatural.