Accessing XML Data from an Object-Relational Mediator Database

A semester thesis paper

by Christof Roduner

Advisor and Supervisor

Prof. Tore Risch

December 4, 2002

Thesis Register Number 235
ISSN 1100-1836

Uppsala Database Laboratory
Uppsala University
P.O. Box 513
S-751 20 Uppsala
Sweden

Contents

1 Introduction

The following paper is the finishing report on a term project exploring an approach to store arbitrary XML documents in the object-oriented database management system AMOS II [RJK00].

AMOS II has a functional data model and is based on the relationally complete object-oriented query language AmosQL, which is similar to the OO parts of SQL-99. An AMOS II database system can integrate data of different type (e.g. relational databases) into its own OO database using a *wrapper*. This results in a common data model and query language for heterogeneous data.

Apart from serving as a standalone database server for applications, many independent AMOS II systems can also interoperate over a communication network such as the internet. In this design, an application can access data stored in an AMOS II database through other AMOS II databases called mediators. A mediator offers a high-level abstraction of the underlying data sources by combining them in the way required by the application. This greatly simplifies the use of heterogeneous data sources at an application level. In addition to this, each mediator system can also manage data of its own.

The AMOS II database system is implemented in C and available for the Windows platform. An AMOS II system's database resides in main-memory and is saved to disk only when requested explicitly. AMOS II is extensible through the use of *foreign functions*. Foreign functions are implemented by the user in some external programming language (Java, C and LISP are currently supported) and can then be used as new data types or operators in the AMOS II system.

The goal of the project presented in this paper was to use Java to write extensions to the AMOS II system in order to integrate XML documents into the database. These XML Extensions consist of three modules that can be called from the AMOS II environment:

- Builder module (wrapper) to parse XML documents using SAX technology and store them as objects in the AMOS II database
- Evaluator module to query an XML document stored in the database with an XPath expression.
- Flattener module to traverse an object graph representing a document or parts of it in order to rebuild the original text representation of the XML data.

1.1 The roots of XML and databases

As this paper is dealing with the issue of bringing traditional database technology and XML together, it is worth taking a look at the relationship between these two technologies. In this section, we will first discuss their origin and benefits offered. Later, we will look more closely at what these two technologies have in common and what makes them different.

Databases have proven an extremely valuable technology for handling large amounts of data and have therefore become an essential tool in software development. They provide features like multi-user access, consistency, integrity, security and distribution that are important building blocks of today's software solutions.

With the rapid growth of the Internet during the 1990ies there was an increasing need for a data format to publish and exchange information between different platforms (operating systems, programming languages, vendors etc.). This led to the standardization of the

Extensible Markup Language (XML) by the World Wide Web Consortium (W3C) in 1998. XML is a metalanguage (i.e. a language to describe other languages with) that meets these challenges by providing means to easily define schemas for documents and to create document instances. Together with its surrounding Web technologies (e.g. HTTP) and other widely accepted standards (e.g. Unicode), XML allows for the encapsulation of information in documents in order to share it across the Web. In this way, information can be exchanged between systems that wouldn't be able to communicate otherwise. Since its introduction by the W3C, XML has gained attention throughout the industry and is now widely used in various ways in software projects.

1.2 XML as a database

A question that arises frequently in discussions about XML is, if XML is actually a database. In the narrow definition of the term, an XML document is a database, namely a collection of logically related data and a description of this data. However, in a broader definition, a database also consists of the many features a Database Management System (DBMS) provides. Of these, XML and its related technologies only offer some:

DBMS function	Corresponding XML technology
storage	XML documents
schemas	DTDs, XML Schema
data retrieval	XPath, XML Query, XQL,
API	SAX, DOM, JDOM, dom4j etc.
views	XSLT
concurrency control	not supported
transactions	not supported
enforcing integrity constraints	not supported
backup and recovery	not supported
security and authorization	not supported

1.3 The need for XML enabled databases

As we saw in the previous subsection, XML and its related technologies lack some important features that are commonly used in software development. There are many application scenarios in which it would be advantageous to rely on these features that are typically provided by a DBMS. Some of these scenarios include:

- Simple Object Access Protocol (SOAP) and Electronic Business using XML (ebXML): An application might need to keep track of transmitted messages in order to guarantee non-repudiation. Such a system needs querying facilities over XML documents that are usually found in DBMS.
- Content Management Systems (CMS): Solutions to publish content on the Web usually use XML to store data in a device-independent way. Depending on the requested output-format, these XML documents are then transformed to HTML, WML or PDF using XSLT technology. Efficient solutions require more sophisticated ways for storing XML content than just flat files.
- Message-Oriented Middleware (MOM): A middleware for exchanging messages asynchronously typically uses XML as its message format. Once published, these XML message have to be queued until they can be delivered to the subscriber. Because the messages need to be queried and processed concurrently, storing

them in flat XML files is not an option for anything other than very light load. Products like XmlBlaster[1] therefore use XML databases.

To satisfy these needs, most commercial database vendors offer extensions to their database systems that support the storage of XML documents. Thus, the advantages of both XML and database technology can be combined.

[1] XmlBlaster (http://www.xmlblaster.org) uses the Apache Xindice (http://xml.apache.org/xindice) native XML database.

2 Storing XML documents in databases

As the aim of this project is to store XML documents in the existing AMOS II database, we shall briefly look at some simple methods commonly used to achieve this in relational database environments. Both of the following ideas are outlined in [Bour99a].

2.1 *Table-Based Mapping*

This approach uses a mapping of simple XML documents to one or more tables:

```
<A>
  <B>
    <C>ccc</C>
    <D f=”111”>ddd</D>
    <E >eee</E>
  </B>
  <B>
    <C>fff</C>
    <D f=”222”>ggg</D>
    <E>hhh</E>
  </B>
  <F>
    <G>iii</G>
  </F>
</A>
```

Table B

docId	*C*	*D*	*f*	*E*
635	ccc	ddd	111	eee
635	fff	ggg	222	hhh
…	…	…		…

Table F

docId	*G*
635	iii
…	…

The advantage of this mapping is its simplicity that leads to good performance even in large-scale environments. However, arbitrary XML structures cannot be expressed with this approach, i.e. further nesting of elements is not possible.

2.2 *Object-Relational mapping*

The Object-Relational approach is typically used in XML-enabled relational databases or middleware tools (e.g. mapping of Entity Beans in EJB to an RDBMS). With this method, an XML document is treated as a tree structure. This results in an object graph representing the document, which is similar to object graphs in object-oriented programming languages. The object graph is then stored in the database using standard object persistence methods.

To create the database schema associated with a certain type of XML documents, the corresponding DTD is first mapped to classes:

DTD

```
<!ELEMENT A (B | C)>
<!ATTLIST A F CDATA #REQUIRED>
<!ELEMENT B (#PCDATA)>
<!ELEMENT C (D, E)>
<!ELEMENT D (#PCDATA)>
<!ELEMENT E (#PCDATA)>
```

Classes

```
class A {
  String b;
  String f;
  C c;
}

class C {
  String d;
  String e;
}
```

Attributes and elements that contain only PCDATA are mapped to scalar properties in the classes. Elements that contain other elements are mapped to classes and properties of the according type.

As a second step, the resulting classes are then mapped to tables in the database schema:

```
Classes                        Tables

class A {                      Table A (
  String b;                      Column b    # Nullable
  String f;                      Column f    # Not nullable
  C c;                           Column c_fk # Nullable
}                              )

class C {                      Table C (
  String d;                      Column c_pk # Not nullable
  String e;                      Column d    # Not nullable
}                                Column e    # Not nullable
                               )
```

With this mapping, a foreign key is used to express the object reference.

There are of course many more mappings of XML data to a relational database. They are mostly based on a graph modelling the underlying XML document. Some of these approaches are presented in [FK99a] and [FK99b].

2.3 Drawbacks

There are several drawbacks to the two approaches presented above:

- The mappings may lead to a large number of tables in the database. Although the mappings try to avoid introducing new tables whenever possible, the number of tables is proportional to the number of elements defined in the DTD. (A mapping with a constant number of tables is presented in [FK99a]. However, performance evaluation has shown that this slows down querying.)
- The relational approach is not very intuitive in the context of XML data. This makes it somewhat inelegant to formulate queries.

There are two characteristics that all of these approaches have in common:

- The database schema is dependent on the DTD of the documents stored. This might be problematic in scenarios where documents of many different types (i.e. DTDs) are to be stored in the same database. There is a potential for conflicts occurring between elements with the same name but different syntax and semantics in different DTDs.
- Fidelity to the original document is not preserved. This is because the above methods only model a document’s elements and attributes. Other items of an XML document like comments and processing instructions are omitted.

Both characteristics are the result of the mappings being data-centric rather than document-centric. Data-centric documents are usually more structured and more fine-grained with the order of the elements being unimportant. They are mainly intended for processing through software. Document-centric documents, however, are usually intended for human consumption. Their structure is less regular and the order of their elements matters. Processing instructions are often essential, e.g. to inform a processor to apply a certain stylesheet to a document. Examples of data-centric documents include air travel information, stock quotes and orders. Typical document-centric documents include user manuals, news articles and web pages.

A more natural mapping can be accomplished by using an underlying object-oriented database system. AMOS II has already been used for this as depicted in [KLR01]. However, that object-oriented database representation did not fully preserve the corresponding XML document either. By contrast, the representation used in this work is an object-oriented database representation with a one-to-one mapping to the original XML document.

2.4 Document preserving storage

Because document-centric applications require documents to be stored with the original structure preserved (i.e. including items other than elements and attributes), the mappings presented so far cannot be used. Thus, other mappings have to be applied. The XML community has developed several data models to store an XML document. Some of these models are more complete than others and are therefore more or less suitable for storing a document without losing too many of its features. What document-preserving approaches have in common is that they provide a logical model for an XML document rather than for the data in the document. As the internal models of such systems are based on XML, they are usually referred to as native XML databases.

The term “native XML database” (NXD) has never been defined formally. One possible definition that was developed by the members of the XML:DB initiative[2] reads as follows:

> A native XML database…
>
> - Defines a (logical) model for an XML document – as opposed to the data in that document – and stores and retrieves documents according to that model. At a minimum, the model must include elements, attributes, PCDATA, and document order. Examples of such models are the XPath data model, the XML Infoset, and the models implied by the DOM and the events in SAX 1.0.
> - Has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage.
> - Is not required to have any particular underlying physical storage model. For example, it can be built on a relational, hierarchical, or object-oriented database, or use a proprietary storage format such as indexed, compressed files.

With this definition in mind, the software in this project was developed as a small step exploring the possibilities of extending AMOS II to provide some of the capabilities found in native XML databases.

[2] The aim of the XML:DB initiative (http://www.xmldb.org) is to develop standards and specifications for XML databases. This includes the promotion of a standardized programming interface for XML databases (XML:DB API) and an update language for XML documents (XUpdate). Several products such as the Tamino XML Database and Apache Xindice provide support for the XML:DB API.

3 Representing XML documents in AMOS II

Being a truly object-oriented system, AMOS II is almost ideally suited for mapping an XML data model to a persistent structure in the database. In this project, it was decided to go for the probably most straightforward approach of representing XML structures, namely using a DOM-like database schema in the underlying AMOS II system. The decision in favour of DOM was made for several reasons: First, the DOM data model is widely used and thus well documented. The implementation presented in this paper allows users that are familiar with the concepts of DOM to navigate intuitively through an XML document using simple AMOS II commands. Second, DOM is supported by an abundance of tools readily available. In this project, such a tool is used for performing XPath evaluation, thus providing path-oriented queries over XML documents stored in AMOS II. Third, the implementation of a more or less straightforward DOM data model in AMOS II serves as a reference point for future work and improvement.

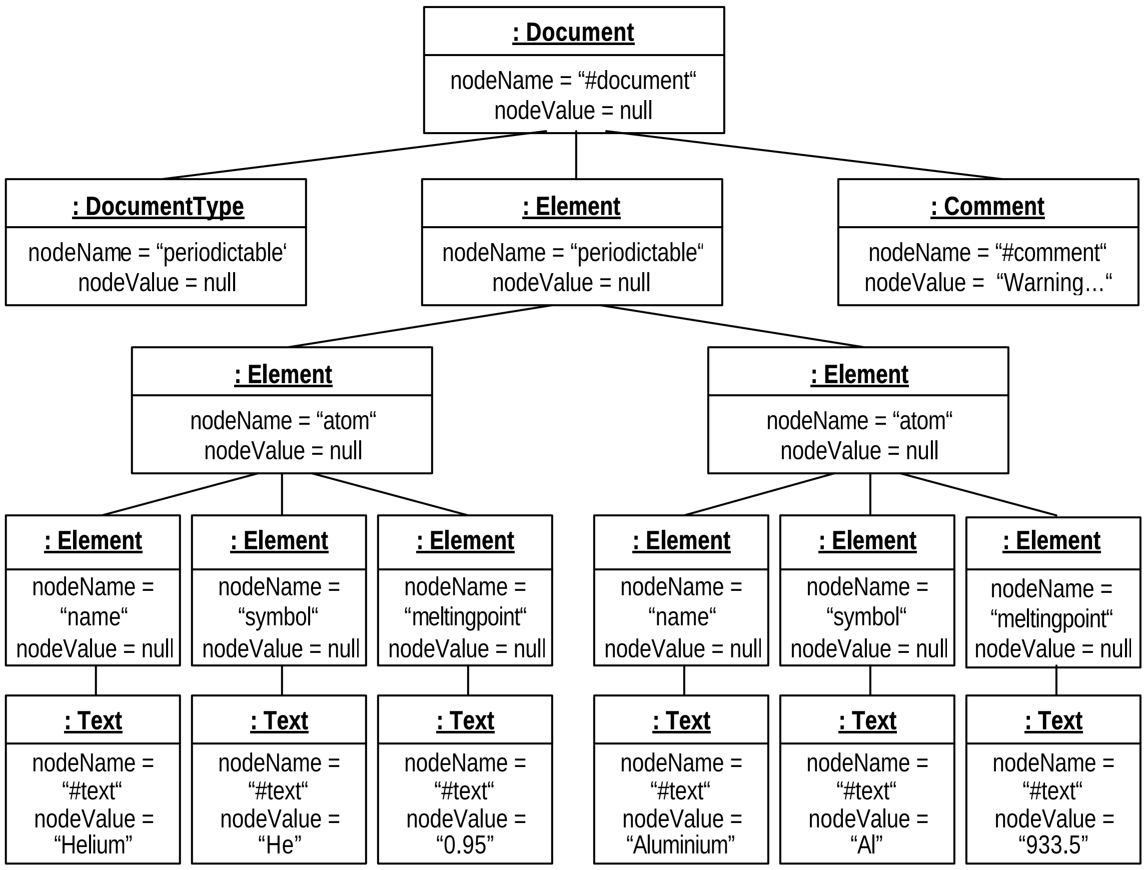
3.1 The Document Object Model

The Document Object Model (DOM) [W3C00b] is an application programming interface (API) for valid HTML and well-formed XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated. DOM allows an application to create documents, navigate their structure and add, modify and delete elements, attributes and content. It is important to note that DOM is not an implementation carrying out these tasks. DOM is just a specification that is originally defined in Object Management Group (OMG) IDL. Additionally, language bindings for Java and ECMAScript are provided by the W3C.

The DOM models the document that it represents with a tree-like object graph. Consider the following well-formed XML document:

```
<?xml version=”1.0”?>
<!DOCTYPE periodictable SYSTEM 'periodic.dtd'>
<periodictable>
  <atom>
    <name>Helium</name>
    <symbol>He</symbol>
    <meltingpoint units=”Kelvin”>0.95</meltingpoint>
  </atom>
  <atom>
    <name>Aluminium</name>
    <symbol>Al</symbol>
    <meltingpoint units=”Kelvin”>933.5</meltingpoint>
  </atom>
</periodictable>
<!-- Warning: This periodic table is incomplete. -->
```

DOM exposes this document as the following tree structure:



The object graph in DOM is composed of so-called nodes as shown above. Each node is of a certain node type (*Document*, *Element*, *Text* etc.) with twelve possible node types (interfaces) in total. Depending on its type, a node can have children that are stored in an ordered list. All the specific node types like *Document*, *Element*, *Text* etc. are descendants of the general node type *Node*. This interface defines some common properties and methods, notably the properties *nodeName* and *nodeValue* as shown in the diagram above.

The following table summarizes the twelve node types and their possible child nodes as found in the DOM specification:

Node Type	**Allowed Children**	**Description**
Document	Element, ProcessingInstruction, Comment, DocumentType	Represents the whole XML document and is the root node of the document tree. Each document contains zero or one node for the document type, one node for the root element and zero or more comments or processing instructions.
DocumentFragment	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference	Provides an abstraction of a subtree in the XML document. Used to extract or insert portions of a document.
DocumentType	none	Represents the <!DOCTYPE …> declaration and provides an interface to the list of entities (e.g. &euro;) defined for the document.
EntityReference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference	Represents a reference to an entity other than a reference to a predefined entity or a character. (An XML processor needs not provide EntityReference objects and

		can just expand entities instead.)
Element	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference	Represents an element (e.g. <person>…</person>) in an XML document.
Attr	Text, EntityReference	Represents an attribute in an Element node (e.g. <person *name="Henric"*>…</person>).
ProcessingInstruction	none	Represents a processing instruction (e.g. <?xml-stylesheet href="mystyle.css" type="text/css"?>). Note that the XML declaration (<?xml version="1.0"?>) is not a processing instruction as processing instructions must not begin with the sequence <?xml. [W3C00a]
Comment	none	Represents a comment (e.g. <!-- ignore -->)
Text	none	Represents the textual content ("character data") in an Element node (e.g. "Henric" in <name>Henric</name>). Special characters (e.g. & and <) that are represented in the original document by a predefined entity reference or character reference are replaced by the actual characters they stand for.
CDATASection	none	Represents a section of text containing character sequences (like < or &) that would otherwise be regarded as markup (e.g. <![CDATA[if (a > 10 && b) invoke();]]>)
Entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference	Represents a (parsed or unparsed) entity declared in the document's DTD (e.g. <!ENTITY euro "&#x20AC">).
Notation	none	Represents a notation declared in the document's DTD (e.g. <!NOTATION PNG SYSTEM "http://www.w3.org/TR/REC-png">)

Please note that the four node types *DocumentFragement*, *Attr*, *Entity* and *Notation* are not considered part of the document tree. They are not in any other node's children list and thus not present in the example above. Because these nodes are not children of the nodes they are attached to, they are accessible through special properties of *Element* and *Document* nodes.

Please also note that the XML declaration (*<?xml version="1.0"?>*) is not a processing instruction and thus not present in the document tree above.

Besides the original DOM specification presented by the W3C, there are a number of alternative models [Sos01] that slightly differ from the W3C model. Products like JDOM[3] and dom4j[4] try to speed up and simplify DOM functionality in Java through the definition of a somewhat modified object model than the one originally proposed by the W3C.

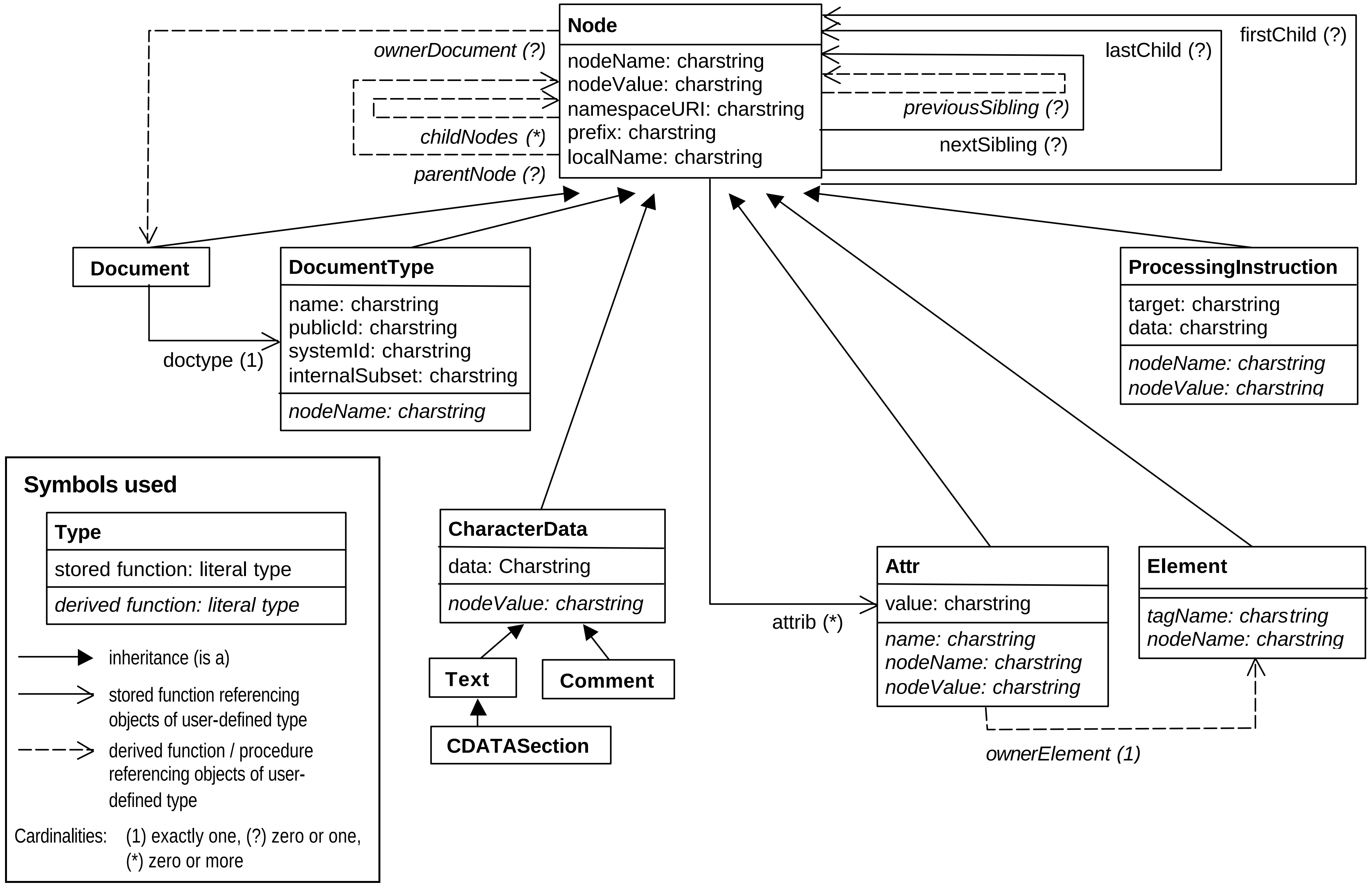
[3] http://www.jdom.org
[4] http://www.dom4j.org

3.2 Database Schema

Due to the object-oriented nature of the AMOS II system, it was possible to map the DOM Level 2 data model to a database schema in a rather straightforward way (see diagram below). However, the three node types *EntityReference*, *Entity* and *Notation* have been omitted in the database schema. Entity references are not represented in the database for efficiency reasons. It would be highly inefficient to create a database object for every single occurrence of an entity reference. Thus, entity references are always expanded and merged with the surrounding *Text* nodes. This design decision implies that the value of creating *Entity* nodes in the database would be very limited: If there are no references to entities, then the entities themselves contain no valuable information. Finally, *Notation* nodes are omitted because they're very uncommon and hardly ever used in applications.

The general (simplified) overview of the database schema is shown in the following diagram.

Database Schema



One important goal during the design of the database schema was to implement as much as possible directly in the AmosQL. Thus, all of the crucial properties of the DOM Level 2 data model specified by [W3C00b] have been implemented in the database using either stored functions, derived functions or procedures in AMOS II. The following tables summarize these properties. If the semantics of a property matches the semantics defined in [W3C00b], then the name of the property is identical wherever possible. Please refer to the implementation in AmosQL for full details.

Node

Function / Procedure	**Return type**	**Description**
nodeName	Charstring	value depending on node type (see respective table)
nodeValue	Charstring	value depending on node type (see respective table)
namespaceURI	Charstring	see types *Element* and *Attr*, null for other types
prefix	Charstring	see types *Element* and *Attr*, null for other types
localName	Charstring	see types *Element* and *Attr*, null for other types
childNodes	bag of Node	the child nodes of this node
parentNode	Node	the parent node of this node
firstChild	Node	the first child node of this node
lastChild	Node	the last child node of this node
previousSibling	Node	the node immediately preceding this node
nextSibling	Node	the node immediately following this node
ownerDocument	Document	the document containing this node
attrib[5]	bag of Attr	see type *Element*, null for other types

Document

Function / Procedure	**Return type**	**Description**
nodeName	Charstring	fixed value "#document"
nodeValue	Charstring	null
doctype	DocumentType	The document type declaration associated with this document.

DocumentType

Function / Procedure	**Return type**	**Description**
nodeName	Charstring	same value as function *name*
nodeValue	Charstring	null
name	Charstring	the name of the DTD; i.e. the name immediately following the *DOCTYPE* keyword
publicId	Charstring	the public identifier of the internal subset
systemId	Charstring	the system identifier of the internal subset
internalSubset	Charstring	the internal subset of the DTD as a string

ProcessingInstruction

Function / Procedure	**Return type**	**Description**
nodeName	Charstring	same value as function *target*
nodeValue	Charstring	same value as function *value*
target	Charstring	the target of the processing instruction
data	Charstring	the content of the processing instruction (string starting with first non-whitespace character immediately after the target)

[5] The DOM specification calls this property "attributes". As there is a predefined function by that name in AMOS II, the identifier has been modified to "attrib".

Text

Function / Procedure	Return type	Description
nodeName	Charstring	fixed value “#text”
nodeValue	Charstring	same value as *data*
data	Charstring	the string contained in this node

CDATASection

Function / Procedure	Return type	Description
nodeName	Charstring	fixed value “#cdata-section”
nodeValue	Charstring	same value as *data*
data	Charstring	the string contained in this node

Comment

Function / Procedure	Return type	Description
nodeName	Charstring	fixed value “#comment”
nodeValue	Charstring	same value as *data*
data	Charstring	the string contained in this node

Element

Function / Procedure	Return type	Description
nodeName	Charstring	same value as *tagName*
nodeValue	Charstring	same value as *data*
namespaceURI	Charstring	the namespace URI of this element
prefix	Charstring	the namespace prefix of this element
localName	Charstring	the local name of this element
tagName	Charstring	the qualified name of this element (i.e. *localName* or a concatenation of *prefix*, a colon and *localName*)
attrib	bag of Attr	the attributes of this element

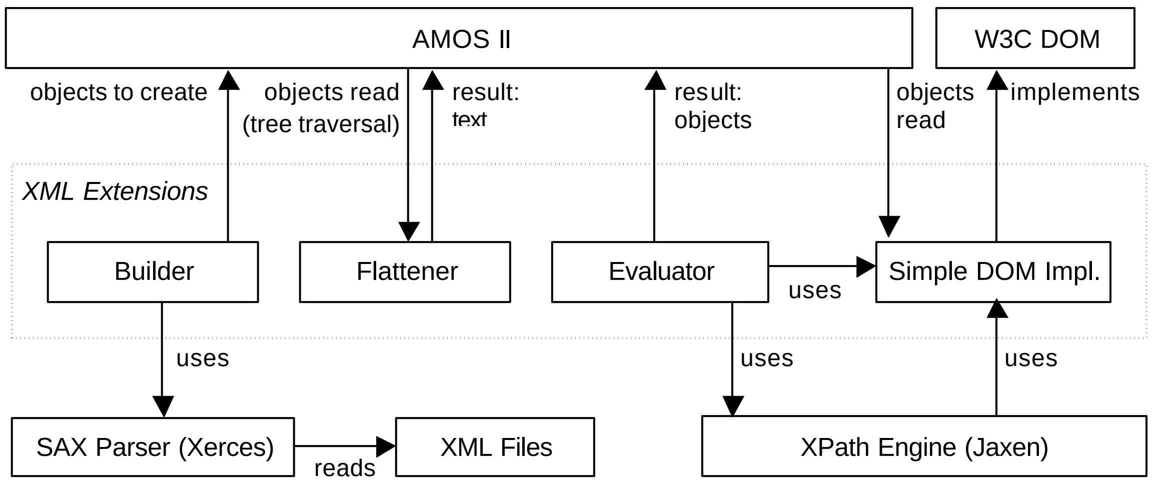
Attrib

Function / Procedure	Return type	Description
nodeName	Charstring	same value as *name*
nodeValue	Charstring	same value as *value*
namespaceURI	Charstring	the namespace URI of this attribute
prefix	Charstring	the namespace prefix of this attribute
localName	Charstring	the local name of this attribute
name	Charstring	the qualified name of this attribute (i.e. *localName* or a concatenation of *prefix*, a colon and *localName*)
value	Charstring	the value assigned to this element
ownerElement	Element	the element owning this attribute

In DOM, the base type *Node* defines some properties like *namespaceURI* that are not actually used by its descendants (e.g. *ProcessingInstruction*). These properties are not repeated in the descendants’s tables above although they’re inherited, of course.

4 Implementation of the Extensions

4.1 General Architecture



The XML Extensions implemented in this project are divided into the four modules *Builder*, *Evaluator*, *Flattener* and *Simple DOM Implementation*. The Builder module is used to import XML files into the database (wrapper). The Flattener traverses a subtree indicated by a root node (e.g. an entire document) and emits the subtree in its flat string representation. The Evaluator takes a document and an XPath expression and returns the matching nodes. Finally, the Simple DOM Implementation is used as a glue layer between the database and third party tools (currently Jaxen). These four modules will be discussed in detail in the following sections.

4.2 Importing XML documents: The Builder

4.2.1 Simple API for XML

The XML Wrapper uses the Simple API for XML (SAX[6]) for importing XML documents into the database. SAX is an event-driven way of reading XML documents. This means that the user of a SAX parser has to implement some interfaces that define a set of parsing events. Before starting the parse, the implementation is passed to the parser for callback. As the parser then reads the file, it emits a stream of so called parse events by invoking the respective methods of the registered implementation. The parser is sometimes referred to as the producer, while the implementation provided by the user is referred to as the consumer.

Unlike other XML technologies, SAX acts like a serial I/O stream. Data is seen as it streams in, but there is no way to go back to an earlier position or forward to a later position. The granularity with which events are processed by the application can be defined by implementing only a certain subset of the interfaces.

[6] http://www.saxproject.org

The following listing shows an XML file and the corresponding event stream:

```
XML file                            SAX Event Stream

<?xml version=”1.0”?>               startDocument
<record type=”cd”>                  startElement: record
  <artist>                            attributes: type “cd”
    The White Stripes               startElement: artist
  </artist>                         characters: “The White Stripes”
</record>                           endElement: artist
                                    endElement: record
                                    endDocument
```

The advantage of the SAX technology is that it is fast and, unlike a DOM tree, requires very little memory. However, many operations on XML documents cannot be carried out if only an event stream is present. XPath for example needs access to a tree modelling the document in order to evaluate expressions.

4.2.2 Java Implementation

The DOM representation of an XML file in the database is created by the *Builder* class of the XML Extensions. It is an implementation of the SAX parsing event interfaces *ContentHandler*, *LexicalHandler*, *DeclHandler*, *DTDHandler* and *ErrorHandler*.

To build the tree structure of the DOM representation by processing the event stream, the *Builder* class has to keep track of the current position in the tree. Whenever a *startElement*, *processingInstruction* or *comment* event occurs, a new node of the respective type is attached as a child node to the *Element* node last created. To deal with this, the *Builder* class uses an internal stack: Every time a *startElement* event is reported, the newly created *Element* node is pushed onto the stack. Thus, events that are triggered later create child nodes of the topmost *Element* on the stack. Whenever an *endElement* event occurs, the topmost *Element* is popped from the stack.

SAX also requires special handling of *character* events to create a *Text* node in the DOM representation. SAX may read characters in chunks and report a continuous string through multiple events. A single *Text* node may thus be the result of several calls to the SAX event *character*. The *Builder* class handles this by appending strings reported by the *character* event to an internal buffer. This buffer is written out to the database as a *Text* node only upon notification of a *startElement*, *endElement*, *processingInstruction* or *comment* event. Only then it is sure that there cannot be any more *character* events that refer to the same *Text* node.

4.2.3 Parsing Issues

The wrapper developed in this project uses the Xerces-J[7] SAX parser by the Apache XML Project[8]. As most parsers, it supports two different modes of operation called validating and non-validating mode. The wrapper provides a switch for the user to decide which mode to use. The main difference between the two modes is that with validation, the parser is *required* to read the document’s DTD. If it cannot fetch the DTD, an error occurs. In non-validating mode, however, the parser *can* read the DTD. Whether it actually reads the DTD (i.e. whether the external DTD file can be fetched) influences the parsing behaviour. For the wrapper, the following aspects that are linked to the parsing mode are important:

[7] http://xml.apache.org/xerces2-j
[8] http://xml.apache.org

Checking against model
Checking against the model means that the parser makes sure that a document complies with its DTD, i.e. that it is valid and not just well-formed [W3C00a]. Elements for example are checked if they occur in an allowed context. If an XML document is invalid, the errors are reported, but the document is still parsed. Xerces performs model checking only if the user explicitly requires validating mode. If the DTD cannot be read in validating mode, the document is still parsed, but a warning is reported.

Ignorable Whitespaces
Ignorable whitespaces are used in XML documents solely for the convenience of a human reader. In the following XML document, the ignorable whitespaces are marked with a dotted line:

```
<?xml version="1.0"?>

<!DOCTYPE catalog [
<!ELEMENT catalog (record*)>
<!ELEMENT record (#PCDATA)>
]>

<catalog>
........<record>The White Stripes</record>
........<record>Flaming Lips     </record>
</catalog>
```

In this fragment, the definition of the *catalog* element specifies that it only consists of *record* elements and must not contain character data. Thus, the dotted whitespaces before the *<record>* tag cannot be character data. They're only provided to make the document more readable and are thus ignorable to applications. However, the trailing whitespaces after the string "Flaming Lips" up to the *</record>* tag are not ignorable because the *record* element is defined as containing character data. Because ignorable whitespaces are meaningless for most applications, the wrapper just discards them. Storing them in the database would be an unnecessary waste of space. However, if the parser cannot fetch a document's DTD, then it cannot decide if a whitespace sequence is ignorable or not. In this case, the wrapper stores all whitespaces in the database. To avoid this situation, it is recommended to carefully check the error and warning messages issued by the parser.

Attribute Defaulting
DTDs allow users to specify default values for attributes. When a parser then encounters an unspecified attribute in an XML file, it defaults it to the value as defined in the DTD. Attribute defaulting is only possible if the parser has access to the DTD, of course. In a DOM representation, the node type *Attr* usually has a property called *specified.* This property indicates whether the attribute was specified in the original document or defaulted by a parser. Unfortunately, this information is not provided by SAX parsers. For this reason, there is no such property stored in the database.

Entity Expansion
Whenever Xerces has access to the DTD, it expands all general entity references recursively. But when the DTD cannot be read, the references cannot be resolved. For an unresolvable reference outside an attribute value, the parser reports this to the application with the *skippedEntity* event. However, for an unresolvable reference in an attribute value, SAX provides no way to report this to the application. This means that the application will simply not see the unresolvable entity reference. As this limitation causes some loss of information anyway, the wrapper does not store unresolvable entities in the database at all. This decision can also be justified by the high overhead that storing of single entity references would generate.

For the various reasons outlined in this section, it is generally recommended to use the wrapper in validating mode.

4.3 Querying documents: The Evaluator

The XML Path Language (XPath) [W3C99a] is a language for addressing parts of an XML document. Like DOM, the underlying data model of XPath defines an XML document as a tree of nodes. However, the two models differ in many ways. The XPath language is commonly used by both XSLT and XPointer.

In the project presented in this paper, the XPath language is used to retrieve portions of an XML document stored in the database. Together with a reference to a document or any part of it, an XPath expression can be passed to the evaluation function. This function then returns all the nodes that match the XPath expression.

4.3.1 The Jaxen Framework

To evaluate XPath expressions, the XML Extensions make use of the Jaxen[9] framework. This package provides a universal XPath engine capable of evaluating expressions across different underlying object models. The models currently supported by Jaxen are DOM, JDOM, dom4j and JavaBeans. Jaxen can be flexibly extended to support additional object models by implementing a predefined interface. The project has its roots in an XPath engine for both JDOM and dom4j. It was decided to factor out the common functionality and to provide a general framework for XPath evaluation.

4.3.2 The Simple DOM Implementation

Jaxen requires the underlying documents to be accessible through the DOM API. For this reason, the XML extensions contain a simple DOM implementation. This module exhibits the documents stored in the AMOS II database through the standard DOM API as defined by the W3C. However, the W3C specification of DOM is rather large and requires the implementation of more than 100 methods. Due to the limited time available for this project, only the methods required by Jaxen have been implemented. Writing access to the document, for example, is not possible. All the methods not directly needed by Jaxen are implemented with placeholders that simply issue an error message on invocation. Still, this simple implementation of DOM provides all the necessary operations to carry out basic read-only tasks with the documents stored in AMOS II.

The simple DOM implementation consists of twelve classes in total. The following code sample illustrates how a W3C DOM object that represents an XML document stored in AMOS II can be created in Java when called as a foreign function from AMOS II:

```
public void someAmosForeignFunction(
  CallContext cxt, Tuple tpl)
  throws AmosException {

      Oid oid = tpl.getOidElem(0);
      org.w3c.Document doc = new SimpleDocumentImpl(oid);
      ...
}
```

[9] http://www.jaxen.org

The simple DOM implementation uses lazy initialization if possible. That is, data is only fetched from the AMOS II database when needed. When a property of the DOM object in Java is accessed, its value is transparently loaded from the database.

Unlike DOM implementations generally used, the solution developed in this project does not suffer from excessive memory consumption in the Java application. DOM tools like Jaxen usually operate on the complete DOM tree of a document in the Java memory. Depending on the size of the document processed, memory consumption can be relatively high. By contrast, the DOM implementation used in this project builds the DOM tree step by step: The DOM tree grows only as data is fetched from the database, because it is actually needed by the application.

4.3.3 Putting it together: Jaxen and the Simple DOM Implementation

With these two building blocks, it is rather easy to apply XPath querying to the XML documents stored in the AMOS II database. The following code section selects all nodes in a document *doc* matching a given XPath expression and emits them to the AMOS II query executor.

```
// create object representing XPath expression
org.jaxen.XPath xpath =
  new org.jaxen.dom.DOMXPath(“/sample/xpath/expression”);

// query document (doc is of class SimpleDocumentImpl)
Iterator result = xpath.selectNodes(doc).iterator();

while (result.hasNext()) {

      // fetch node matching XPath expression
      SimpleNodeImpl node = (SimpleNodeImpl)result.next();

      // return oid of node to AMOS II
      tpl.setElem(2, node.getId());
      cxt.emit(tpl);
}
```

4.4 Exporting nodes: The Flattener

To export an XML document or a fragment of it to a string representation, the *Flattener* module is used. This string representation produced by it can be written to a file in order to recreate the original XML file outside the database. The module is implemented by the *Flattener* class.

To build the string representation, the module uses a simple pre-order traversal algorithm. Like the *Evaluator*, it operates on the Simple DOM Implementation. The module can be passed a single node indicating the root node of a subtree. An entire XML file can be reconstructed by calling the *Flattener* module with the root node of the document as a parameter. Flattening of subtrees is also useful to print the result of an XPath evaluation.

Please note that the resulting string representation of the XML document might not be physically equivalent[10] to the original XML file. This is, again, due to a limitation of the SAX

[10] However, the XML documents might still be logically equivalent in a given application context. Further information regarding an operator to test for document equivalence can be found in [W3C01].

parsing technology. As discussed earlier, SAX cannot supply the *Specified* property of an attribute. Therefore, there is no reliable way to reconstruct the original document by the sole means provided by SAX.

The same difficulty applies to the representation of an empty element. When parsed with SAX, the two sequences *<element></element> and <element/>* generate exactly the same event stream. Hence, in the output of the *Flattener* module, you will always find two tags denoting an empty element, no matter what the original sequence was.

A similar but more serious problem occurs with formatting the output of the *Flattener* module. XML files are usually indented to make them more readable for humans. The idea of indenting is to add ignorable whitespaces to the document. However, as discussed above, indenting cannot be applied to a document without prior knowledge of its grammar (DTD). If whitespaces are mechanically added to a document without considering its DTD, the semantics of the document might change, as not all created whitespaces are ignorable. In the context of long term storage (typically in a database), problems with external DTDs arise: In order to recreate a string representation equivalent to the original XML file, the whole DTD has to be accessible. Because external subsets might not be accessible in the future, it might be advantageous to store them in the database.

Because the software developed in this project does not store external DTD subsets in the database, it is not possible to reconstruct a pretty printed string representation equivalent to the original document. The *Flattener* module thus provides the option of building the string representation with pretty printing either switched on or off. With pretty printing switched off, the resulting string is more difficult to read for humans. But at the same time, it is guaranteed not to differ from the original document because of output formatting.

5 Using the XML Extensions in AMOS II

5.1 Function Reference

The XML Extensions are accessible in AMOS II through the following foreign functions:

`store(Charstring uri, Integer validation, Integer cdata) -> Node`		
Imports an XML document into the AMOS II database. A reference to the stored document is returned.		
Parameters	`uri`	A string containing a local filename or a URL to the XML file that is to be imported.
	`validation`	The number 0 or 1 to indicate if validation is switched on (1) or off (0) during parsing of the document.
	`cdata`	The number 0 or 1 to indicate if CDATA sections are stored as *CDATASection* nodes (1). A value of 0 indicates that CDATA sections should be stored as normal *Text* nodes and, if possible, be merged with adjacent text.

`evaluateNS(Node subtree, Charstring expression, Vector bindings) -> bag of Node`		
Evaluates an XPath expression in the context of a certain subtree (e.g. a whole document) and returns all matching nodes.		
Parameters	`subtree`	The root node of the context in which the expression is to be evaluated.
	`expression`	Any valid XPath expression.
	`bindings`	Namespace bindings used during evaluation. The vector consists of interleaving prefixes and namespace URIs. E.g.: {“html”, “http://www.w3.org/1999/xhtml”, “env”, “http://schemas.xmlsoap.org/soap/envelope”}

`evaluate(Node subtree, Charstring expression) -> bag of Node`
Evaluates an XPath expression in the context of a certain subtree (e.g. a whole document) and returns all matching nodes. This function is just an abbreviation for the function *evaluateNS* with an empty *bindings* vector.

`flatten(Node subtree, Integer pretty) -> Charstring`		
Returns the string representation of an XML document or fragment.		
Parameters	`subtree`	The root node of the subtree to be converted to its string representation.

	`pretty`	The number 0 or 1 to indicate if the output should be formatted to be more readable by humans. Please note: Pretty printing might change the XML data in a way that does not retain equivalence to the original XML file.

`pretty(Node subtree) -> Charstring`
Returns the string representation of an XML document or fragment. This function is just an abbreviation for the function *flatten* with pretty printing switched on.

`verbose(Integer mode) -> Boolean`		
The functions of the XML Extensions collect data on the internal steps carried out. If verbose mode is switched on, this data is displayed at the end of each function call. Please note that the times displayed in the brackets are accumulated and not per-call times.		
Parameters	`mode`	The number 0 or 1 indicating if extra information is shown at the end of each function call.

5.2 Usage examples

The following examples use an XML document called *example.xml*[11] with the following content modelling an imaginary online shop for records:

```
<?xml version="1.0"?>
<!DOCTYPE catalog SYSTEM 'example.dtd'>
<?xml-stylesheet href="webpage.xsl" type="text/xsl"?>

<!-- data from an imaginary record shop -->

<catalog xmlns:html="http://www.w3.org/1999/xhtml">

  <entry type="cd">
    <artist>Kent</artist>
    <title>Vapen &amp; Ammunition</title>
    <style>Pop/Rock</style>
    <price>17</price>
    <comment user="Lena">Best album <html:I>ever</html:I>! Check it out!</comment>
    <comment user="Magnus">After all the hype: Rather disappointing.</comment>
  </entry>

  <entry type="cd">
    <artist>Suede</artist>
    <title>Coming Up</title>
    <style>Pop/Rock</style>
    <price>19</price>
    <comment user="Sofia">Still <html:B>great</html:B> - even without Bernard.</comment>
  </entry>

  <entry type="lp">
    <artist>Tricky</artist>
    <title>Blockback</title>
    <style>Triphop</style>
    <price>25</price>
  </entry>

  <entry type="cd">
    <artist>White Stripes</artist>
```

[11] The file can be found in the *samples* directory of the source distribution.

```
    <title>White Blood Cells</title>
    <style>Independent</style>
    <price>19</price>
    <comment user="Lena">Hotel Yorba rocks!</comment>
  </entry>

</catalog>
```

5.2.1 Storing and retrieving a document

- First of all, we switch on verbose mode and import an XML file into the database with both validation and CDATA processing enabled:

  ```
  JavaAMOS 1> verbose(1);
  JavaAMOS 2> set :doc = store("c:\xml\example.xml", 1, 1);
  ```

  The system will report the number of nodes created and the time needed.

- We then read the whole document from the database and display its string representation in a nicely formatted way:

  ```
  JavaAMOS 3> pretty(:doc);
  ```

5.2.2 Exploring the DOM tree

- We now explore the DOM tree of the XML document stored in the previous example:

  ```
  JavaAMOS 4> nodeName(childNodes(:doc));
  "catalog"
  "xml-stylesheet"
  "#comment"
  "catalog"
  ```

  These are the four topmost nodes in the document. The first occurrence of *catalog* stems from the document type declaration, while the second occurrence is the root element of the document.

- We continue by listing the child nodes of the root element:

  ```
  JavaAMOS 5> nodeName(childNodes(childNode(:doc, 3)));
  "entry"
  "entry"
  "entry"
  "entry"
  ```

  The root node only contains *entry* elements.

- To reduce the typing, we first select the first *entry* element into the variable *:e*. We then display its child nodes again and select the first child into the variable *:artist*.

  ```
  JavaAMOS 6> set :e = childNode(childNode(:doc, 3), 0);
  JavaAMOS 7> nodeName(childNodes(:e));
  "artist"
  ```

```
"title"
"style"
"price"
"comment"
"comment"
JavaAMOS 8> set :artist = childNode(:e, 0);
```

- Now we display the name and value of the child nodes found under the element *artist*:

```
JavaAMOS 9> select nodeName(c), nodeValue(c) from Node c
              where c = childNodes(:artist);
<"#text","Kent">
```

- We then fetch the last element under the *entry* element into the variable *:c* and print it.

```
JavaAMOS 10> set :c = lastChild(:e);
JavaAMOS 11> pretty(:c);
"<comment user='Magnus'>
  After all the hype: Rather disappointing.
</comment>
"
```

- Then we select the node immediately preceding the last *comment* element into the variable *:c* and print its attributes:

```
JavaAMOS 12> set :c = previousSibling(:c);
JavaAMOS 13> select nodeName(a), nodeValue(a) from Attr a
               where a = attrib(:c);
<"user","Lena">
```

- Finally, we retrieve the parent node of the *comment* element and the document containing the *comment* element.

```
JavaAMOS 14> nodeName(parentNode(:c));
"entry"
JavaAMOS 15> ownerDocument(:c);
#[OID 713]
```

5.2.3 Querying the document with XPath

- In this example, we extract all the *comment* elements from the document stored in the example above and print the resulting nodes:

```
JavaAMOS 16> flatten(evaluate(:doc, "/catalog/entry/comment"),
0);
"<comment user='Magnus'>After all the hype: Rather
disappointing.</comment>"
"<comment user='Lena'>Best album <html:I>ever</html:I>! Check it
out!</comment>"
"<comment user='Sofia'>Still <html:B>great</html:B> - even
without Bernard.</comment>
"<comment user='Lena'>Hotel Yorba rocks!</comment>"
```

- We then select only the comment by Magnus:

  ```
  JavaAMOS 17> pretty(evaluate(:doc,
                 "/catalog/entry/comment[@user='Magnus']/text()"));
  "After all the hype: Rather disappointing."
  ```

- Next, we select the titles of all records selling for the same price as the album by Suede. This time, we use the unabbreviated XPath syntax:

  ```
  JavaAMOS 18> pretty(evaluate(:doc,
  "/child::catalog/child::entry[price=/child::catalog/child::entry
  [artist='Suede']/child::price/text()]/child::title/text()"));
  "Coming Up"
  "White Blood Cells"
  ```

- Now we are interested to know the records with more than one user commenting on:

  ```
  JavaAMOS 20> pretty(evaluate(:doc,
  "/catalog/entry[count(comment) > 1]/title/text()"));
  "Vapen &amp; Ammunition"
  ```

- In this example, we evaluate an XPath expression that is not in the context of an entire document, but of an individual *Text* node. We first fetch the last *Text* node of Lena’s comment into the variable *:t*. Then we select all the *Element* nodes on the same level as the node stored in *t:*.

  ```
  JavaAMOS 21> set :t = evaluate(:doc,
  "/catalog/entry[artist='Kent']/comment[@user='Lena']/text()[posi
  tion()=last()]");
  JavaAMOS 22> pretty(evaluate(:t, "../*"));
  "<html:I>
    ever
  </html:I>"
  ```

- Finally, we select all the elements in the *http://www.w3.org/1999/xhtml* namespace:

  ```
  JavaAMOS 23> pretty(evaluateNS(:doc, "//pref:*", {"pref",
  "http://www.w3.org/1999/xhtml"}));
  "<html:I>
    ever
  </html:I>"
  "<html:B>
    great
  </html:B>"
  ```

6 Performance Measurements

6.1 Evolution of the system

The first implementation of the system used a slightly different approach of representing the parent-child-relationship in XML documents. The idea was then, to store all the child nodes in an indexed vector property of the parent node. The following AmosQL listing illustrates the initial implementation:

```
create function childNodes(Node) -> vector of Node as stored;

create function nextSibling(Node n) -> Node s as
  select s from integer i, integer j, Node p where
    childNodes(p)[i] = n and
    j = i + 1 and
    childNodes(p)[j] = s;
```

To measure the performance of the system, the XML document *periodic.xml*[12] was imported into the database. This 114 KB document containing the periodic table of the elements consists of roughly 1900 element nodes, 1900 text nodes and 900 attributes. In the context of this document, the XPath expression */PERIODIC_TABLE/ATOM/NAME* was then evaluated. Total runtime for evaluation was 135.154 s with the following details[13]:

```
Calls to getBasicProperties:     1899 (5874 ms)
Calls to getExtendedNextSibling: 1898 (127877 ms)
Calls to getExtendedPrevSibling: 0 (0 ms)
Calls to getExtendedParentNode:  0 (0 ms)
Calls to attrib:                 0 (0 ms)
Calls to getExtendedChildNodes:  114 (1144 ms)
Calls to ownerDocument:          0 (0 ms)
```

XPath evaluation with Jaxen obviously requires numerous database calls to fetch the next sibling of a node. To improve performance, it was decided to give up vector storage and use a linked list with a stored *nextSibling* property instead. The database schema was modified to reflect this change and performance was measured again. Total runtime decreased to 9.804 s with the following details:

```
Calls to getBasicProperties:     1899 (6869 ms)
Calls to getExtendedNextSibling: 1898 (1253 ms)
Calls to getExtendedPrevSibling: 0 (0 ms)
Calls to getExtendedParentNode:  0 (0 ms)
Calls to attrib:                 0 (0 ms)
Calls to getExtendedChildNodes:  114 (1332 ms)
Calls to ownerDocument:          0 (0 ms)
```

Changing the datastructure from a vector to a linked list also sped up the document's loading time. Originally, a parent node's child nodes vector was built by appending nodes with the following statement:

```
set childNodes(parent) = concat(childNodes(parent), {child});
```

With the new datastructure, *nextSibling* references are added instead. Performance evaluation showed that loading speed had nearly doubled by changing from a vector to the linked list, too.

[12] See *samples* directory in the source distribution.
[13] The first number in a row indicates the number of calls to a function, while the number in the bracket sums up the time that was needed for these calls.

The times listed above show that calling *getBasicProperties* now took 6.9 s. This function was implemented as follows:

```
create function getBasicProperties(Node n) -> vector as
begin
  declare charstring c1, charstring c2, charstring c3,
          charstring c4, charstring c5, charstring c6;
  set c1 = nodeName(n);
  if some(nodeValue(n)) then set c2 = nodeValue(n) else set c2 = NIL;
  set c3 = lower(name(typeof(n)));
  if some(namespaceURI(n)) then set c4 = namespaceURI(n) else set c4 = NIL;
  if some(prefix(n)) then set c5 = prefix(n) else set c5 = NIL;
  if some(localName(n)) then set c6 = localName(n) else set c6 = NIL;
  result {c1, c2, c3, c4, c5, c6};
end;
```

The idea of having a *getBasicProperty* function was to reduce the number of database calls from Java. However, it turned out that calling *getBasicProperty* is slower than calling the six single functions separately. Measurements showed that calling *getBasicProperty* 110 times takes 3885 ms, while calling every single of the six functions (*nodeName, nodeValue, typeof, namespaceURI, prefix, localName*) 110 times takes 1292 ms in total. Moreover, analysis of the XPath engine showed, that it would be sufficient just to fetch *localName* and *namespaceURI* in most cases. Clustering of these properties was not considered because many of them are usually not assigned to a node.

These findings led to giving up the idea of a property that gets the six values in a single database call. Instead, they are now fetched in many individual calls whenever needed. As a result of this, evaluation time of the XPath expression decreased from 9.804 s to 3.565 s with the following details:

```
Calls to nodeName:               0 (0 ms)
Calls to nodeValue:              0 (0 ms)
Calls to nodeType:               0 (0 ms)
Calls to namespaceURI:           1897 (482 ms)
Calls to prefix:                 0 (0 ms)
Calls to localName:              1897 (410 ms)
Calls to getExtendedNextSibling: 1898 (1192 ms)
Calls to getExtendedPrevSibling: 0 (0 ms)
Calls to getExtendedParentNode:  0 (0 ms)
Calls to attrib:                 0 (0 ms)
Calls to getExtendedChildNodes:  114 (1331 ms)
Calls to ownerDocument:          0 (0 ms)
```

At this time, the Java implementation did not use lazy initialization of node lists yet. With this functionality added, performance again increased with a total evaluation time of 1.883 s now. The times needed in detail are as follows:

```
Calls to nodeName:               0 (0 ms)
Calls to nodeValue:              0 (0 ms)
Calls to getNodeType:            0 (0 ms)
Calls to namespaceURI:           1897 (431 ms)
Calls to prefix:                 0 (0 ms)
Calls to localName:              1897 (220 ms)
Calls to getExtendedNextSibling: 1898 (1132 ms)
Calls to getExtendedPrevSibling: 0 (0 ms)
Calls to getExtendedParentNode:  0 (0 ms)
Calls to attrib:                 0 (0 ms)
Calls to getExtendedFirstChild:  114 (40 ms)
Calls to getExtendedChildNodes:  0 (0 ms)
Calls to ownerDocument:          0 (0 ms)
```

6.2 Results

The following data summarizes the runtime behaviour of the system. It was collected on an Intel Pentium III machine with 500 MHz processor speed and 256 MB main memory. The software environment consisted of Windows 2000 (SP2) and the Java VM 1.4.0_01. Again, the 114 KB XML document *periodic.xml* consisting of 1897 *Element* nodes, 1888 *Text* nodes and 936 *Attr* nodes was imported.

- Loading of the document[14]: 2.9 s.

  This time consists of 742 ms needed by Xe rces to parse the document and the following times for database calls:

```
Calls to create_Element: 1897 (1142 ms)
Calls to create_Attr:    936 (350 ms)
Calls to create_Text:    1887 (712 ms)
```

- Space requirements: 1.68 MB

- Query time for expression */PERIODIC_TABLE/ATOM[SYMBOL='Cu']*: 1.9 s
  The following database calls were needed:

```
Calls to nodeName:               0 (0 ms)
Calls to nodeValue:              112 (40 ms)
Calls to getNodeType:            0 (0 ms)
Calls to namespaceURI:           1897 (451 ms)
Calls to prefix:                 0 (0 ms)
Calls to localName:              1897 (251 ms)
Calls to getExtendedNextSibling: 1898 (931 ms)
Calls to getExtendedPrevSibling: 0 (0 ms)
Calls to getExtendedParentNode:  0 (0 ms)
Calls to attrib:                 0 (0 ms)
Calls to getExtendedFirstChild:  226 (70 ms)
Calls to getExtendedChildNodes:  112 (20 ms)
Calls to ownerDocument:          0 (0 ms)
```

- Query time for expression */PERIODIC_TABLE/ATOM*: 0.12 s
  The following database calls were needed:

```
Calls to nodeName:               0 (0 ms)
Calls to nodeValue:              0 (0 ms)
Calls to getNodeType:            0 (0 ms)
Calls to namespaceURI:           113 (0 ms)
Calls to prefix:                 0 (0 ms)
Calls to localName:              113 (20 ms)
Calls to getExtendedNextSibling: 114 (40 ms)
Calls to getExtendedPrevSibling: 0 (0 ms)
Calls to getExtendedParentNode:  0 (0 ms)
Calls to attrib:                 0 (0 ms)
Calls to getExtendedFirstChild:  2 (0 ms)
Calls to getExtendedChildNodes:  0 (0 ms)
Calls to ownerDocument:          0 (0 ms)
```

[14] Average of 10 individual measurements.

7 Conclusion and Future Work

In this project, a fully functioning set of XML extensions to the AMOS II database have been developed. With AMOS II, XML documents of any (possibly unknown) schema can be integrated in a distributed environment that consists of many heterogeneous data sources. Imported documents or parts of them can be queried by using an XPath evaluation module. This module provides support for the full XPath language as specified by the W3C. The results of these XPath queries can be exported back to a string representation that could be written to an XML file. These features are available in AMOS II through foreign functions and can thus be easily integrated in any AmosQL expression.

In addition to this, a simple implementation of the W3C DOM API has been developed. It is now possible for Java programmers to access XML data stored in AMOS II through the most widely used XML API. Any existing Java library operating on XML data can be combined easily with AMOS II, as long as it supports the W3C DOM API. Because the XML data is stored in a DOM-like database schema, no reparsing of documents is needed. An advantage of the simple DOM implementation is, that it does not need to keep the whole document tree in memory. A Java application starts by using just a single DOM object (e.g. the root node of the document) that is linked to an AMOS II database object. This initial tree consisting of a single node only grows, when the application really needs to access more data in the document. Then, the requested nodes are read from the database and added to the in-memory DOM tree in Java. This process of growing is transparent to the programmer, as it happens in the background.

A problem with the current implementation is the space consumption of XML files in the AMOS II database. One reason for this is the DOM-like database schema, in which even simple primitive types like strings and numbers (e.g. *12* in *<price>12</price>*) are stored as complex objects (nodes) with an own database identifier (OID).

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

- Reduce the space requirements in AMOS II. As space requirement is linked to performance, extensive analysis would be needed to assess the tradeoffs. The current system supports the storage of the complete XML document (document-centric) and provides features like namespace evaluation in XPath. A more lightweight solution omitting for example support for comments, processing instructions, namespaces etc. might reduce space requirements, because a simple schema could be used.
- The evaluation of XPath expressions could be sped up by an own implementation of the W3C XPath specifications. XPath evaluation is currently done by Jaxen, which is not optimized for the AMOS II environment, of course.
- At the moment, the simple DOM implementation only allows read-access to the XML data stored in AMOS II. It could be easily extended to provide write-access as well.
- Support for more standard technologies like XQuery, XUpdate and especially the XML DB API could be added.

References

[ABS00]	Abiteboul, S. / Buneman, P. / Suciu, D. (2000): Data on the Web. From Relations to Semistructured Data and XML.
[Bour99a]	Bourret, R. (2001): Mapping DTDs to Databases. (Available at: http://www.xml.com/pub/a/2001/05/09/dtdtodbs.html)
[Bour99b]	Bourret, R. (2001): XML and Databases. (Available at: http://www.rpbourret.com/xml/XMLAndDatabases.htm)
[Bro02]	Brownell, D. (2002): SAX 2. Processing XML Efficiently with Java.
[EN00]	Elmasri, R. / Navathe, S.B. (2000): Fundamentals of Database Systems.
[ER00]	Elin, D. / Risch, T. (2000): Amos II Java Interfaces.
[FK99a]	Florescu, D. / Kossmann, D. (1999): Storing and Querying XML Data using an RDMBS.
[FK99b]	Florescu, D. / Kossmann, D. (1999): A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database.
[Flo00]	Flodin, S. / Josifovski, V. / Katchaounov, T. / Risch, T. / Sköld, M. / Werner, M. (2000): Amos II User's Manual. (Available at: http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html)
[Har02]	Harold, E. R. (2002): Processing XML with Java. (Available at: http://cafeconleche.org/books/xmljava)
[KLR01]	Katchaounov, T. / Lin, H. / Risch, T. (2001): Adaptive Data Mediation over XML Data. (Available at: http://user.it.uu.se/~torer/publ/jass01.pdf)
[Ris01]	Risch, T. (2001): AMOS II Active Mediators for Information Integration (Whitepaper). (Available at: http://user.it.uu.se/~udbl/amos/amoswhite.html)
[RJK00]	Risch, T. / Josifovski, V. / Katchaounov, T. (2000): AMOS II Concepts. (Available at: http://user.it.uu.se/~udbl/amos/doc/amos_concepts.html)
[Sos01]	Sosnoski, D. (2001): XML in Java: Document Models, Part 1: Performance. (Available at: http://www-106.ibm.com/developerworks/xml/library/x-injava/index.html)
[ST02]	Salminen, A. / Tompa F. W. (2002): Requirements for XML Document Database Systems. In: E.V. Munson (Ed.), Proceedings of the ACM Symposium on Document Engineering (DocEng '01), pp. 85-94, New York: ACM Press. (Available at: http://db.uwaterloo.ca/~fwtompa/.papers/xmldb-desiderata.pdf)

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

[W3C00b] Byrne, S. / Champion, M. / Le Hégaret, P. /Le Hors, A. / Nicol, G. / Robie, J. / Wood, L. (2000) : Document Object Model (DOM) Level 2 Core Specification (Version 1.0).
(Available at : http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113)

[W3C01] Boyer, J. (2001): Canonical XML Version 1.0.
(Available at: http://www.w3.org/TR/xml-c14n)

[W3C99a] Clark, J. / DeRose, S. (1999): XML Path Language (XPath) Version 1.0.
(Available at: http://www.w3.org/TR/xpath)

[W3C99b] Bray, T. / Hollander, D. / Layman, A. (1999): Namespaces in XML.