Universität Kaiserslautern
AG Datenbanken und Informationssysteme

Seminar im Wintersemester 2002/2003:
XML und Datenbanken
(http://wwwdbis.informatik.uni-kl.de/courses/seminar/WS0203/)

XML-Middleware

Torsten Lenhart
(t_lenhar@informatik.uni-kl.de)

Inhalt

1. Einleitung

Die Realisierung des reibungslosen und schnellen Datenaustauschs zwischen oft nur lose gekoppelten Systemen ist eine wesentliche Herausforderung bei der Entwicklung von Anwendungs- und Informationssystemen. In diesem Zusammenhang gewinnt XML als Format für den kanonischen Datenaustausch immer mehr Bedeutung. Gleichzeitig werden riesige Mengen von Daten in zumeist relationalen oder objekt-relationalen Datenbanken gehalten, deren interne Struktur zunächst nichts mit XML zu tun hat. Eine Schlüsselaufgabe besteht deshalb in der Entwicklung von Konzepten und Techniken, mit deren Hilfe man Daten aus herkömmlichen Datenbanken in XML abbilden und umgekehrt XML-Daten in herkömmlichen Datenbanken ablegen kann. Systeme und Komponenten, die diese Aufgabenstellung lösen, bezeichnet man als *XML-Middleware*.
Es existieren bereits verschiedene Ansätze zur Realisierung von XML-Middleware, weitere befinden sich zur Zeit in der Entwicklung. Im folgenden werden einige dieser Ansätze bzw. deren exemplarische Umsetzungen vorgestellt.
Kapitel 2 beschäftigt sich mit sogenannten *XML-erweiterten* (*XML-enabled) Datenbanken*. Hierbei handelt es sich um zumeist (objekt-)relationale Datenbanken, die durch den Hersteller oder einen Drittanbieter mit einer SQL-Mappingschicht versehen wurden. In 2.1 wird in diesem Zusammenhang der *DB2 XML Extender* von IBM vorgestellt, 2.2 gibt einen Überblick über die XML-Fähigkeiten des *SQL Server 2000* von Microsoft.
Im dritten Kapitel wird *XTABLES* als Beispiel für ein unabhängiges XML-Middleware-System vorgestellt. Hier wird eine XML-Sicht auf die in der Datenbank abgelegten Daten angeboten, welche mit Hilfe der XML-Anfragesprache XQuery genutzt werden kann, um benutzerdefinierte Sichten zu erstellen und Anfragen auf diesen Sichten durchzuführen.
Kapitel 4 behandelt die sich im Moment noch in der Entwicklung befindliche SQL-Erweiterung *SQL/XML* (Part 14: "XML-related specifications"). Idee hierbei ist es zum einen, den Umgang von relationalen Datenbanksystemen mit XML zu standardisieren, zum anderen die in diesem Zusammenhang angebotene Funktionalität durch SQL-Sprachkonstrukte einheitlich zugänglich zu machen.
Im fünften Kapitel wird die Programmierschnittstelle *XML:DB API* vorgestellt, die einen ähnlich komfortablen Zugang zu nativen XML-Datenbanken ermöglichen soll, wie ihn ODBC und JDBC für relationale Datenbanken bereits bieten.
Im sechsten und letzten Kapitel werden die vorgestellten Ansätze noch einmal zusammengefasst und hinsichtlich ihrer Vor- und Nachteile bewertet.

2. XML-erweiterte Datenbanken

Einer der klassischsten Ansätze, um eine Beziehung zwischen XML-Daten und Datenbanken herzustellen, ist die Erweiterung von zu meist (objekt-)relationalen Datenbanksysteme um eine zusätzliche XML-Mapping-Schicht, welche entweder vom Hersteller selbst oder aber von Drittanbietern bereitgestellt wird. Die Abbildung der Daten von und nach XML unterscheidet sich bei den am Markt erhältlichen Lösungen allerdings mitunter sehr stark und es ist kein einheitliches Konzept erkennbar.

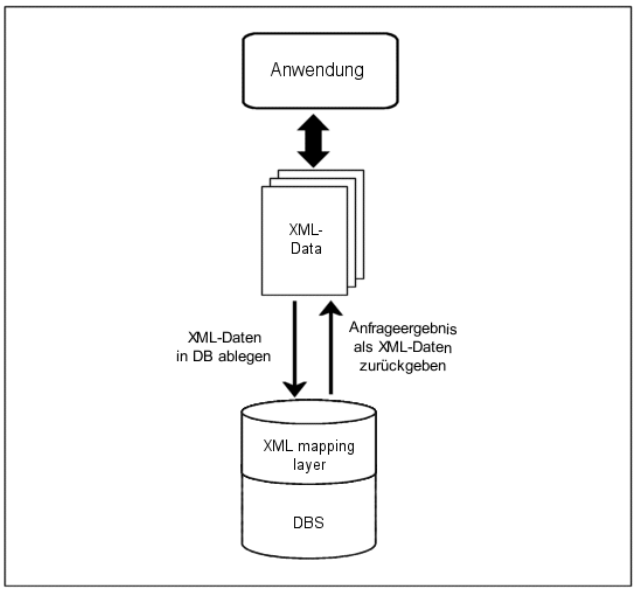

Abb. 1: Aufbau einer XML-erweiterten Datenbank

Im folgenden werden mit dem *XML Extender* für die *DB2 Universal Database* der IBM und dem *SQL Server 2000* von Microsoft zwei Umsetzungen von XML-erweiterten Datenbanken von verschiedenen Herstellern vorgestellt.

2.1 IBM DB2 XML Extender

Durch Einsatz des XML Extenders wird die DB2 Universal Database um die Möglichkeit erweitert, ganze XML-Dokumente in adäquater Weise abzulegen und auf diese komfortabel zuzugreifen. Des weiteren können XML-Dokumente in relationale Daten umgewandelt und umgekehrt aus existierenden relationalen Daten wohlgeformte XML-Dokumente gewonnen werden. Für diese zwei verschiedenen Anwendungsarten werden vom XML Extender zwei grundlegende Konzepte angeboten:

- ***XML Columns:*** Dieses Konzept ermöglicht die Speicherung ganzer XML-Dokumente in DB2, wofür verschiedene neue Datentypen geschaffen wurden. Die Dokumente werden in Spalten abgelegt und können aktualisiert, durchsucht und natürlich auch wieder ausgelesen werden. Um eine schnellere Suche gewährleisten zu können, ist es möglich, bestimmte Elemente und Attribute auf sogenannte *side tables* abzubilden.

- ***XML Collections:*** Mit Hilfe von XML Collections können XML-Dokumentstrukturen auf DB2 Tabellen abgebildet werden. Dies erlaubt die komfortable Komposition von XML-Dokumenten aus Daten, die in DB2 vorgehalten werden. Umgekehrt ist es ausgehend von einem XML-Dokument möglich, seinen Content in DB2 Tabellen abzulegen.

DB2 Datenbanken, die mit Hilfe des XML Extenders für XML *enabled* wurden, enthalten automatisch auch einen XML DTD Repository, in welchem die den XML-Dokumenten

zugeordneten DTDs abgelegt werden. Diese können zur Validation der XML-Dokumente verwendet werden.
Informationen über das Abbildungsschema werden in sogenannten Document Access Definiton (DAD) Files festgehalten. Es handelt sich bei den DADs selbst ebenfalls um XML-Dokumente und sie werden genau wie die DTDs in einem eigenen Repository abgelegt. Struktur und Zweck einer DAD unterscheidet sich abhängig davon, ob sie für den Gebrauch mit XML Columns oder XML Collections vorgesehen ist. Dabei ist es Aufgabe des Benutzers, eine DAD entsprechend seinen Bedürfnissen zu erstellen. Zur Identifikation von XML-Elementen und Attributen kommt dabei im XML Extender *XPath* zum Einsatz.

2.1.1 XML Columns

Wie bereits erwähnt, werden XML Columns verwendet, wenn man ganze XML-Dokumente in einer Datenbank ablegen will. Dies bietet sich vor allem dann an, wenn die Dokumente oft gelesen und selten Updates auf ihnen ausgeführt werden. Hat man sich für diese Methode entschieden, muss man anschließend festlegen, in welcher Tabelle das XML-Dokument gespeichert werden soll. Prinzipiell hat man hier die Möglichkeit eine neue Tabelle mit einer XML Column zu kreieren, ebenso kann man aber auch eine bereits bestehende Tabelle um eine XML Column erweitern. Für XML Columns werden vom XML Extender dabei drei neue benutzerdefinierte Datentypen angeboten:

- *XMLVarchar*: wird verwendet, wenn das XML-Dokument kleiner als 3 KB ist
- *XMLCLOB*: wird verwendet, wenn das XML-Dokument größer als 3 KB ist. Die Maximalgröße liegt hier bei 2 GB
- *XMLFILE*: hierbei handelt es sich um eine Referenz auf ein XML-Dokument, das sich außerhalb der Datenbank befindet

Bei Verwendung mit XML Columns spezifiziert ein DAD-File, welche zusätzlichen Maßnahmen bei der Speicherung eines XML-Dokumentes in einer XML Column vorgenommen werden sollen. Zum einen kann man festlegen, ob das Dokument vor seiner Speicherung auf Konsistenz mit dem entsprechenden DTD überprüft werden soll. Dabei ist darauf zu achten, dass im Falle der Validation die notwendige DTD auch tatsächlich im XML DTD Repository vorliegt. Zum anderen kann festgelegt werden, für welche Elemente und Attribute *side tables* erstellt werden sollen. In einer *side table* wird der Content des Elements bzw. der Wert des Attributs in einem passenden DB2 Datentyp zusammen mit einem Fremdschlüssel abgelegt. Dieser Fremdschlüssel zeigt auf den Primärschlüssel der Tabelle, welche die XML Column mit dem entsprechenden XML-Dokument enthält. Wenn ein Element oder Attribut nur einmal in einem XML-Dokument vorkommen kann, ist diese Vorgehensweise unproblematisch und es können mehrere Elemente und Attribute in einer *side table* zusammengefasst werden. Ist hingegen ein mehrfaches Vorkommen möglich, muss für diese Elemente und Attribute jeweils eine eigene *side table* erstellt werden, die um eine Spalte mit einer Sequenznummer erweitert wird. Abb. X zeigt beispielhaft, wie XML Dokumente in einer XML Column abgelegt und mit Hilfe einer DAD *side tables* erzeugt werden können.
Nachdem eine XML Column angelegt wurde, können XML-Dokumente eingefügt, durchsucht, aktualisiert und ausgelesen werden. Entsprechend diesen vier Funktionalitäten bietet der XML Extender vier Gruppen von benutzerdefinierten Funktionen (*user-defined*

michael.xml:

```
<?xml version="1.0"?>
<person>
    <name>michael</name>
    <haarfarbe>rot</haarfarbe>
</person>
```

thomas.xml:

```
<?xml version="1.0"?>
<person>
       <name>thomas</name>
       <haarfarbe>braun</haarfarbe>
</person>
```

person.dad:

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "c:\dxx\dtd\dad.dtd">
<DAD>
<dtdid>person.dtd</dtdid>
<validation>YES</validation>
<Xcolumn>
    <table name="person_haarfarbe">
        <column name="farbe"
            type="varchar(20)"
            path="/person/haarfarbe"
            multi_occurence="NO" />
    </table>
<Xcolumn>
</DAD>
```

resultierende Tabellen:

Tabelle person1:

id	person
1	XMLVarchar <?xml version="1.0"?> <person> <name>michael</name> <haarfarbe>rot</haarfarbe> <person>
2	XMLVarchar <?xml version="1.0"?> <person> <name>thomas</name> <haarfarbe>braun</haarfarbe> <person>

side table person_haarfarbe:

pid	farbe
1	rot
2	braun

Abb. 2: XML Column Beispiel

functions, UDFs).
Speicherfunktionen (*storage UDFs*) dienen dabei wie der Name schon sagt zum Einfügen von XML-Dokumenten in eine XML Column. Die Einfügefunktion `XMLVarcharFromFile()` wird zum Beispiel benutzt, wenn man eine XML-Datei in eine XML Column vom Typ XMLVarchar einfügen will. Beispiel (a) in Abb. 3 zeigt ein SQL-Statement, in dem `XMLVarcharFromFile()` verwendet wird.

Es existieren zwei unterschiedliche Arten von Updatefunktionen (*update UDFs*): Zum einen solche, die zum Überschreiben eines ganzen Dokumentes dienen. Diese sind den Einfügefunktionen sehr ähnlich. Zum anderen gibt es die gleichnamige Funktion `Update()`, mit deren Hilfe unter Verwendung von XPath der Content einzelner Elemente oder der Wert einzelner Attribute geändert werden kann. Das Update-Granulat ist bei beiden Arten aber immer das gesamte XML-Dokument, weshalb XML Columns nur eingesetzt werden sollten, wenn seltene Updates zu erwarten sind. Auch hier findet sich unter (b) ein Beispiel in Abb.3.
Es gibt eine Fülle von Auslesefunktionen (*retrieval UDFs*), die jedoch hier im Detail nicht vorgestellt werden. Als Beispiel sei die überladene Funktion `Content()` genannt, welche ein XML-Dokument ausliest und zu einem (externen) Dateisystem exportiert. Vergleiche hierzu Beispiel (c) in Abb. 3.

```
(a)     insert into person1 values
        (1, db2xml.xmlvarcharfromfile('c:\michael.xml'));

(b)     update person1 set
        person = db2xml.Update(person,'/person/haarfarbe','blond');

(c)     select db2xml.Content(person,'c:\retrieveperson.xml')
        from person1 where id=1;

(d1)    select db2xml.extractVarchar(person,'/person')
        from person1

(d2)    select
        db2xml.extractVarchars(person,'/person')
        from person1
```

Abb. 3: Beispiele für Anwendung von UDFs

Bleiben noch die Suchfunktionen (*extraction UDFs*), die immer dann Anwendung finden, wenn eine Nutzung der *side tables* nicht weiterhilft oder nicht möglich ist. Auch hier gibt es zwei Arten, die wiederum abhängig davon genutzt werden, ob das gesuchte Element oder Attribut mehrmals vorkommen kann. Skalare Funktionen (*scalar extraction UDFs*) werden bei maximal einmaligen Vorkommen benutzt und liefern daher genau einen Wert zurück. Im anderen Fall kommen Tabellenfunktionen (*table extraction UDFs*) zum Einsatz, welche eine Tabelle mit den entsprechenden Werten zurückliefern. Beide Arten haben grundsätzlich den immer gleichen Aufbau: `extractXXX`(<XML-Dokument>, <XPath des gesuchten Elementes bzw. Attributes>), wobei `XXX` jeweils durch den DB2 Datentyp ersetzt wird, in den der Content bzw. der Wert konvertiert werden soll. Skalare und Tabellenfunktionen unterscheiden sich dadurch, dass bei Tabellenfunktionen für `XXX` die Mehrzahl verwendet wird. (d1) zeigt ein Beispiel für eine skalare Funktion, (d2) für eine Tabellenfunktion.
Im Gegensatz zur Verwendung bei XML Collections ist das Anlegen einer DAD bei XML Columns nicht in jedem Fall notwendig. Will man auf Validation und *side tables* verzichten, kann man sich das Erstellen einer DAD sparen. Hat man sich aber für diese zusätzlichen Funktionalitäten entschieden und eine DAD angelegt, muss diese der entsprechenden XML Column zugewiesen werden. Man spricht hier vom *XML column enabling*. Dabei wird zunächst der DAD-File geparst, anschließend werden die *side tables* angelegt und Trigger kreiert, die die Konsistenz der *side tables* mit der entsprechenden XML Column Tabelle

gewährleisten. Insbesondere sollten direkte Updates auf die side tables in jedem Fall vermieden werden. Schließlich wird die DAD dem DAD Repository hinzugefügt.

2.1.2 XML Collections

XML Collections ermöglichen es, die Struktur eines XML-Dokuments auf DB2 Tabellen abzubilden und umgekehrt. Dadurch können XML-Dokumente relational in der Datenbank gespeichert und aus relationalen Daten XML-Dokumente generiert werden. Diese Methode bietet sich zum Beispiel an, wenn Teile des Dokuments häufig aktualisiert werden müssen und es sich bei der Update-Performanz um einen kritischen Faktor handelt.
Auch bei XML Collections werden DAD-Files benutzt, um das Abbildungsschema festzulegen. Im Gegensatz zur Verwendung mit XML Columns, wo, wenn überhaupt, nur Teile des Dokuments auf *side tables* abgebildet werden, erfolgt hier eine vollständige Abbildung des Dokumentes auf Tabellen. Die DADs können dabei auf zwei Arten realisiert werden: Entweder als DAD mit SQL_stmt oder aber als DAD mit RDB_node.
DADs mit SQL_stmt orientieren sich an der SQL-Syntax. Es ist hervorzuheben, dass sie sich lediglich für die Abbildung relationaler Daten in XML eignen, der umgekehrte Weg ist mit ihrer Hilfe nicht möglich. Kern einer entsprechenden DAD ist ein Element `<sql_stmt>`, welches ein `SELECT`-Statement enthält, das zum Auslesen der abzubildenden Daten dient. Im weiteren Verlauf der DAD wird die Struktur des resultierenden XML-Dokuments spezifiziert und mit den ausgelesenen Daten gefüllt.
DADs mit RDB_node orientieren sich an der XML-Syntax und können im Gegensatz zu DADs mit SQL_stmt für beide Abbildungsrichtungen genutzt werden. Informationen, welche Elemente und Attribute auf welche Tabellen und Spalten gemapt werden, finden sich hier in `<RDB_node>` Elementen. Jeweils ein solches Element wird für jedes Element und Attribut des XML-Dokuments gebraucht. Die DADs für beide möglichen Abbildungsrichtungen unterscheiden sich dabei dahingehend, dass bei Abbildung von XML in die Datenbank im `<RDB_node>`, der die Abbildung eines Elementes in eine Tabelle spezifiziert, auch der Primärschlüssel der resultierenden Tabelle festgelegt werden muss (bzw. das Element oder Attribut, dass zum Primärschlüssel abgebildet wird).
Eine XML Collection bezeichnet genau betrachtet ein Paar aus DAD und der Datenbank, welche die Tabellen enthält, auf die bzw. in die die Abbildung stattfinden soll. Durch *XML Collection enabling* kann eine XML Collection explizit erstellt und ihr ein konkreter Name zugewiesen werden. Eine solche explizite Definition ist aber im allgemeinen nicht nötig, eine direkte Angabe des Paars DAD und Datenbank ist alternativ auch möglich.
Für die Komposition von XML-Dokumenten aus relationalen Daten sind zwei stored procedures definiert. Die Prozedur `dxxGenXML()` erzeugt ein XML-Dokument aus den Tabellen, welche in der als Eingabeparameter übergebenen DAD festgelegt sind. Die Prozedur `dxxRetrieveXML()` arbeitet ganz ähnlich, nur das hier eine XML Collection als Parameter übergeben wird, welche ja wiederum eindeutig eine DAD identifiziert. Bei beiden Prozeduren ist es egal, ob es sich um eine DAD mit SQL_stmt oder DAD mit RDB_node handelt.

Tabelle person:

id	name	alter
1	Andreas	20
2	Christian	30

Tabelle telefon:

pid	telefonnr
1	0631/123456
2	0631/654321

personen.dad:

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "c:\dxx\dtd\dad.dtd">
<DAD>
    <validation>NO</validation>
    <Xcollection>
        <prolog>?xml version="1.0"?</prolog>
        <root_node>
            <element_node name="personen">
                <RDB_node>
                    <table name="person"/>
                    <table name="telefon"/>
                    <condition>
                        person.id=telefon.pid
                    </condition>
                </RDB_node>
                <element_node name="person">
                    <attribute_node name="telefon">
                        <RDB_node>
                            <table name="telefon"/>
                            <column name="telefonnr"/>
                        </RDB_node>
                    </attribute_node>
                    <text_node>
                        <RDB_node>
                            <table name="person"/>
                            <column name="name"/>
                        </RDB_node>
                    </text_node>
                </element_node>
            </element_node>
        </root_node>
    </Xcollection>
</DAD>
```

resultierendes Dokument:

```
<?xml version="1.0"?>
<personen>
    <person telefon="0631/123456">Andreas</person>
    <person telefon="0631/123456">Christian</person>
</personen>
```

Abb. 4: XML Collection mit RDB_Node zur Erzeugung eines XML-Dokuments

Für die Dekomposition eines XML-Dokuments in eine Datenbank stehen ebenfalls zwei Funktionen `dxxShredXML()` und `dxxInsertXML()` zur Verfügung. Auch bei ihnen besteht der einzige Unterschied darin, dass `dxxShredXML()` eine DAD und `dxxInsertXML()` eine Collection als Parameter übernimmt. Die entsprechenden DADs müssen hier aber DADs mit RDB_node sein, da, wie schon oben erwähnt, DADs mit SQL_stmt für die Dekomposition nicht geeignet sind.
(Bemerkung: Alle vier erwähnten stored procedures übernehmen neben der DAD bzw. der Collection auch noch andere Parameter, die hier nicht näher erläutert werden)

2.2 Microsoft SQL Server 2000

Als zweites Beispiel für eine XML-erweiterte Datenbank wird nun im folgenden der SQL Server 2000 von Microsoft vorgestellt. Ebenso wie beim DB2 XML Extender bietet der SQL Server die Möglichkeit, aus relationalen Daten XML-Daten zu generieren. Dazu dient die Erweiterung `FOR XML` des `SELECT`-Statements, die im verwendeten SQL-Dialekt definiert ist. Auch die Dekomposition von XML-Dokumenten in relationale Daten wird unterstützt. Hierzu bietet der SQL Server die UDF `OPENXML`, welche zusammen mit zwei stored procedures für diese Zwecke genutzt werden kann.
Neben diesen Built-In-Features sind im Zuge dreier Updates, den SQLXML (XML für SQL Server 2000) Releases 1.0 – 3.0, weitere Funktionalitäten hinzugekommen.

2.2.1 FOR XML

Prinzipiell kann fast jedes beliebige `SELECT`-Statement durch das Anhängen von `FOR XML` dazu gebracht werden, das resultierende Rowset in XML-Form zurückzuliefern. Allerdings gilt unter anderem die Einschränkung, dass `FOR XML` nicht in verschachtelten `SELECT`-Anweisungen verwendet werden darf. Zusammen mit den Schlüsselwörtern `FOR XML` muss ein Modus spezifiziert werden, welcher das zu verwendende Abbildungsschema festlegt. Es gibt drei solche Modi: `RAW`, `AUTO` und `EXPLICIT`, wobei mit `RAW` die einfachsten und mit `EXPLICIT` die komplexesten Abbildungen möglich sind.
Der `RAW` Modus ist wie eben erwähnt der simpelste der drei angebotenen. Jeder vom `SELECT`-Statement zurückgelieferte Record wird jeweils in einem `<row>`-Element untergebracht, wobei die Spaltenwerte als Attribute übernommen werden.
Durch Verwendung des `AUTO` Modus können auch eingebettete Elemente erzeugt und damit hierarchische Strukturen verwirklicht werden. Für jede Tabelle eines Joins werden dabei separate Elemente erzeugt, welche auch nach den Tabellen benannt werden. Die Einbettung erfolgt dann entsprechend der Nennung im FROM-Teil des `SELECT`-Statements.
Der `EXPLICIT` Modus ermöglicht die komplexeste Erzeugung von XML-Daten, die entsprechenden `SELECT`-Statements können aber mitunter ebenfalls sehr komplex werden. Neben der flexiblen Vorgabe der hierarchischen Struktur lässt sich in diesem Modus bestimmen, ob die Spalte einer Tabelle als Unterelement oder als Attribut dargestellt werden soll. In Abb. 5 wird für jeden Modus ein Beispiel gegeben.
In bezug auf `FOR XML` bleibt anzumerken, dass nicht automatisch wohlgeformte XML-Daten zurückgeliefert werden, da in den meisten Fällen kein Wurzelelement erzeugt wird. Ohne

(Tabelle person, Tabelle telefon wie oben)

- ***FOR XML in RAW-Mode:***

```
SELECT person.name, telefon.telefonnr FROM person, telefon
WHERE person.id=telefon.pid FOR XML RAW
```

resultierendes XML-Fragment:

```
<row name="Andreas" telefonr="0631/123456"/>
<row name="Christian" telefonnr="0631/654321"/>
```

- ***FOR XML in AUTO-Mode:***

```
SELECT person.name, telefon.telefonnr FROM person, telefon
WHERE person.id=telefon.pid FOR XML AUTO
```

resultierendes XML-Fragment:

```
<person name="Andreas">
    <telefon telfonnr="0631/123456"/>
</person>
<person name="Christian">
    <telefon telfonnr="0631/654321"/>
</person>
```

- ***FOR XML in EXPLICIT-Mode:***

```
SELECT 1 as Tag,
       NULL as Parent,
       person.id as [person!1!id],
       NULL as [telefon!2!name]
       NULL as [telefon!2!nummer]
FROM person
UNION ALL
SELECT 2,
       1,
       person.id,
       person.name,
       telefon.telefonr
FROM person, telefon WHERE person.id = telefon.pid
ORDER BY [person!1!id], [telefon!2!name], [telefon!2!nummer]
FOR XML EXPLICIT
```

resultierendes XML-Fragment:

```
<person id="1">
    <telefon name="Andreas" nummer="0631/123456"/>
</person>
<person id="2">
    <telefon name="Christian" nummer="0631/654321"/>
</person>
```

Abb. 5: Beispiele für die Anwendung von FOR XML

explizite Behandlung dieses Umstandes handelt es sich bei den von `FOR XML` zurückgelieferten Daten nur um XML-Fragmente.

2.2.2 OPENXML

SQL Server 2000 führt eine neue Transact-SQL Funktion `OPENXML` ein, die dazu genutzt werden kann, XML-Dokumentstrukturen und XML-Daten auf relationale Tabellen abzubilden. `OPENXML` wird dabei zusammen mit den *stored procedures* `sp_xml_preparedocument` und `sp_xml_removedocument` verwendet. Die Prozedur `sp_xml_preparedocument` liest ein XML-Dokument in den Speicher und wendet Microsofts XML-Parser MSXML auf dieses an. Dabei handelt es sich um einen Preparsing-Vorgang, durch den die Daten für die Verwendung mit OPENXML vorbereitet werden. Schließlich wird ein sogenannter *document handle* zurückgeliefert. Dies ist ein Integer-Wert, der die geparsten Daten im Speicher identifiziert.
`OPENXML` nimmt anschließend diesen *document handle* als Parameter und erzeugt daraus eine tabellenwertige Sicht auf die XML-Daten. Außerdem wird an Hand eines XPath-Patterns angegeben, welche Elemente des Dokuments eingelesen werden sollen. Im optionalen `WITH`-Teil kann die Abbildung genauer spezifiziert werden. Lässt man den `WITH`-Teil ganz weg, wird eine neue Tabelle nach Default-Regeln erzeugt.
Nach der Verwendung von `OPENXML` ist es ratsam, den von den geparsten XML-Daten blockierten Speicher wieder freizugeben. Dies geschieht unter Verwendung der Prozedur `sp_xml_removedocument`, welche wiederum den entsprechenden *document handle* übergeben bekommt. In bezug auf die konkrete Anwendung von OPENXML ist zu erwähnen, dass es sich hierbei technisch gesehen ähnlich wie bei Tabellen und Views um einen *Rowset Provider* handelt, wodurch die Funktion überall dort verwendet werden kann, wo auch Tabellen und Views Verwendung finden. Eine schwerwiegende Restriktion von OPENXML ist aber, dass es im allgemeinen nur auf kleinen Datenmengen anwendbar ist, weil immer das gesamte Dokument im Speicher vorliegen muss.

2.2.3 SQLXML (XML für SQL Server 2000)

Um die Unterstützung von XML weiter zu verbessern, nicht zuletzt aber auch zur Einbindung des SQL Server 2000 in das .Net Konzept, wurden von Microsoft mehrere Web Releases von SQLXML (XML für SQL Server 2000) herausgebracht, die einige interessante Erweiterungen beinhalten.
SQLXML 1.0 wurde im Februar 2001 veröffentlich und führte unter anderem Updategrams und XML Bulk Load ein. Updategrams können genutzt werden, um relationale Daten durch XML über HTTP zu modifizieren. Bei XML Bulk Load handelt es sich um ein COM Objekt, welches zusammen mit XDR-Schemas (mittlerweile auch mit XSD-Schemas) zur effizienten und komfortablen Dekomposition großer Mengen von XML-Daten benutzt werden kann. Es stellt dadurch eine Alternative zu `OPENXML` dar.
Mit dem im Oktober 2001 erschienene SQLXML 2.0 wurden sogenannte *SQLXML Managed Classes* eingeführt, die eine Unterstützung des .Net Frameworks gewährleisten. Konkret bedeutet dies, dass .Net-Entwicklern ermöglicht wird, aus ihren Anwendungen heraus auf die

XML-Features von SQL Server 2000 inklusive der durch SQLXML gebotenen Erweiterungen zuzugreifen. Ab dieser Version wird des weiteren die Unterstützung von *annotated XSD Schemas* angeboten. Durch XSD *annotations* können XSD Schemas auf Tabellen und Spalten abgebildet werden. Dies ermöglicht die Einrichtung von XML-Sichten, welche zusammen mit XPath verwendet werden können, um Anfragen auf relationale Daten zu formulieren, die Records im XML-Format zurückliefern.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sql="urn:schemas-microsoft-com:mapping-schema">
  <xsd:element name="person" sql:relation="Employees" >
   <xsd:complexType>
     <xsd:sequence>
        <xsd:element name="vorname"
                     sql:field="FirstName"
                     type="xsd:string" />
        <xsd:element name="nachname"
                     sql:field="LastName"
                     type="xsd:string" />
     </xsd:sequence>
        <xsd:attribute name="id"
                       sql:field="EmployeeID"
                       type="xsd:integer" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Abb. 6: XML Schema mit *annotations*

Im Rahmen von SQLXML 3.0, welches das aktuellste Release ist und im Februar 2002 erschien, wurde neben der Verbesserung und Erweiterung der Features der früheren Releases eine breite Unterstützung von Web Services eingerichtet. Dies beinhaltet, dass existierende stored procedures, UDFs und Updategrams binnen Sekunden als Web Services zur Verfügung gestellt werden können.

3. XTABLES

Das Middleware-System XTABLES wird zur Zeit im IBM Almaden Research Center in San Jose entwickelt und war früher auch unter dem Namen XPERANTO bekannt. Der Benutzer bekommt hier die Möglichkeit geboten, XML Views auf eine existierende (objekt-)relationale Datenbank zu erzeugen und Anfragen auf diesen Views auszuführen. Wie man später noch sehen wird, kann dabei sowohl für die Erstellung der Views als auch für das Absetzen der Anfragen die XML Anfragesprache *XQuery* verwendet werden. Im Unterschied zu anderen Ansätzen auf diesem Gebiet, muss der Benutzer also lediglich eine Sprache beherrschen.

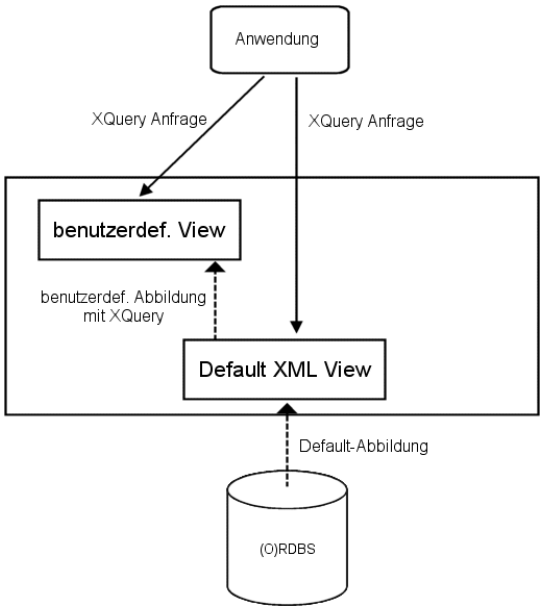

Abb. 7: XTABLES-Architektur (hohe Abstraktionsebene)

Insbesondere sind keine SQL-Kenntnisse erforderlich und die zu Grunde liegende Datenbank bleibt verborgen. Abb. 6 zeigt den Aufbau der XTABLES-Architektur auf hoher Abstraktionsebene.

(Tabelle person, Tabelle telefon wie oben)

Default XML View:

```
<db>
    <person>
        <row>
            <id>1</id> <name>Andreas</name><alter>20</alter>
        </row>
        <row>
            <id>2</id><name>Christian</name><alter>30</alter>
        </row>
    </person>
    <telefon>
        <row>
            <pid>1</pid><telefonnr>0631/123456</telefonnr>
        </row>
        <row>
            <pid>2</pid><telefonnr>0631/654321</telefonnr>
        </row>
    </telefon>
</db>
```

Abb. 8: Beispiel für Default XML View

3.1 XML Views

Nach der Anbindung eines Datenbanksystems wird von XTABLES automatisch ein sogenannter Default XML View erzeugt. Es handelt sich dabei um eine noch recht primitive Abbildung der in der Datenbank enthaltenen Daten in XML. Das Wurzelelement einer solchen Sicht ist immer das Element <db>, für jede Tabelle gibt es ein Top-Level-Element mit gleichem Namen wie die Tabelle selbst. Für jeden Record einer Tabelle existiert ein <row>-Element, welches in das entsprechende Tabellen-Element eingebettet ist. In einem <row>-Element ist wiederum für jede Spalte ein gleichnamiges Element enthalten. In diesen sind schließlich die konkreten Werte für die Spalte als Text eingebettet. Abb. 7 zeigt einen Default XML View auf eine Datenbank, welche die Tabellen person und telefon enthält..
Der Default XML View stellt die unterste, für den Benutzer sichtbare Ebene dar. Mit Hilfe

XQuery zur Erstellung eines benutzerdefinierten Views:

```
create view person as (
<db>
    for $person in view("default")/person/row
    return
        <person>
            <name>$person/name</name>
            <alter>$person/alter</alter>
            <telefon>
                for $telefon in view("default")/telefon/row
                where $order/id=$telefon/id
                return
                    <telefonnummer>$telefon/telefonnr</telefonnummer>
            </telefon>
        </person>
</db>
```

resultierender View:

```
<db>
    <person>
        <name>Andreas</name>
        <alter>20</alter>
        <telefon>
            <telefonnummer>0631/123456</telefonnummer>
        </telefon>
    </person>
    <person>
        <name>Christian</name>
        <alter>30</alter>
        <telefon>
            <telefonnummer>0631/654321</telefonnummer>
        </telefon>
    </person>
</db>
```

Abb. 9: Beispiel für die Erzeugung eines benutzerdefinierten Views

von XQuery können aus diesem neue, benutzerdefinierte Views erzeugt werden. Insbesondere ist es möglich, auf den benutzerdefinierten Sichten selbst wiederum benutzerdefinierte Sichten zu erstellen. Dadurch lassen sich leicht höhere Abstraktionsebenen erreichen und unter Ausschöpfung der Fähigkeiten von XQuery können beliebig komplexe Abbildungen erzeugt werden.
Abb. 8 zeigt die XQuery Anweisung, die den ebenfalls angegebenen benutzerdefinierten View erzeugt.

3.2 Ausführung der Anfragen durch den Query Processor

Wurde eine neue Sicht erzeugt, können vom Benutzer Anfragen auf dieser Sicht ausgeführt werden. Dabei kommt der Query Processor ins Spiel, der auch als das Herz von XTABLES bezeichnet wird. Zunächst konvertiert dieser die übergebene XQuery-Anfrage in eine Zwischenrepräsentation, in das sogenannte XML Query Graph Model (XQGM). In diese Form gebracht ist es relativ leicht möglich, eine komplexe Anfrage auf eine komplexe Sicht in eine einfache Anfrage auf die Default XML Sicht umzubauen. Es müssen dadurch nur die Daten aus der Datenbank ausgelesen werden, die zur Beantwortung der Anfrage tatsächlich benötigt werden, insbesondere müssen die betroffenen Sichten nicht alle einzeln materialisiert werden. Danach wird im sogenannten Computation Pushdown Module aus dem umgebauten XQG-Modell die benötigte SQL-Anfrage erstellt und an die Datenbank übergeben. Direkt an das Tagger Runtime Module werden Informationen gesendet, die festlegen, wie das Anfrageergebnis genau aussehen soll. Die ebenfalls an das Tagger Runtime Module gelieferten Ergebnisse der SQL-Anfrage werden dann entsprechend diesen Informationen konvertiert und schließlich dem Benutzer zurückgegeben. Man beachte, dass bei dieser Vorgehensweise alle speicher- und rechenaufwändigen Aufgaben auf das Datenbanksystem verlagert werden.

4. SQL/XML

Seit Mitte 2000 wird an einer Erweiterung von SQL gearbeitet, die eine Verbindung zwischen SQL und XML herstellen soll. Diese Erweiterung ist als part 14, "XML-related specifications" oder auch prägnanter unter dem Namen SQL/XML bekannt. Das verfolgte Ziel ist hierbei, die Standardisierung verschiedener Aspekte der Interaktion und Integration von SQL und XML. Bis zum gegenwärtigen Zeitpunkt befindet sich das Projekt noch in der Entwicklung, so dass bisher nur einzelne Fragmente der Spezifikation feststehen und auch hier noch einmal Änderungen möglich sind. Mit einer Verabschiedung als internationaler

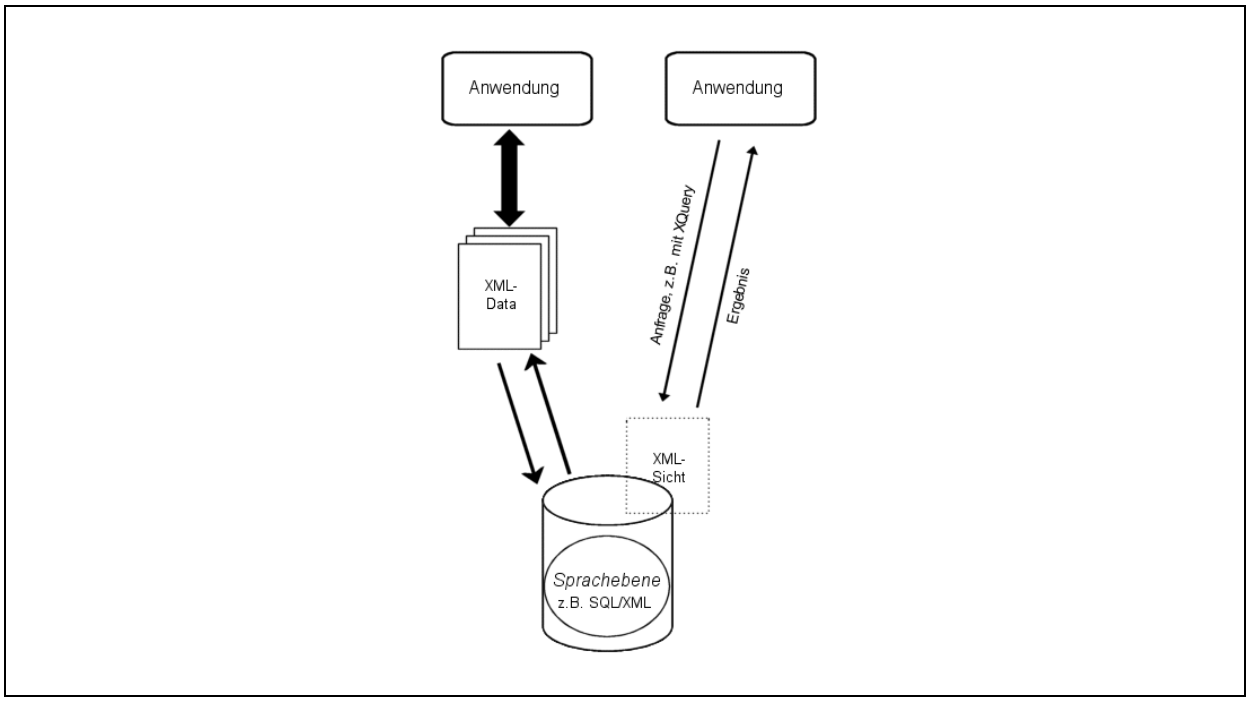

Abb. 10: SQL/XML-Ansatz im Überblick

Standard ist wohl nicht vor Mitte 2003 zu rechnen.
Im folgenden wird ein Überblick über die bisher erarbeiteten Teile der Spezifikation gegeben.

4.1 Infrastruktur

Zunächst gilt es einige grundlegende Regeln zu definieren, die auch als die Infrastruktur von SQL/XML bezeichnet werden.
Eine fundamentale Lücke zwischen SQL und XML besteht zum Beispiel in den verwendeten und verwendbaren Zeichensätzen. SQL kann prinzipiell mit verschiedenen Zeichensätzen verwendet werden. Welcher konkret benutzt wird und wie die interne Darstellung aussieht, ist abhängig von der Implementation. Im Gegensatz dazu ist XML auf Unicode als Zeichensatz festgelegt. Jedes Produkt, dass SQL/XML unterstützt, muss deshalb für alle in Zusammenhang mit SQL verwendeten Zeichensätze eine Abbildung auf Unicode und umgekehrt definieren. Diese Abbildung ist produktspezifisch und wird daher im Standard nicht näher definiert.
In XML gibt es gewisse Zeichen, die in Element- und Attributbezeichnern nicht vorkommen dürfen. Dazu gehört z.B. das Leerzeichen. Deshalb ist hier eine gesonderte Abbildung von Nöten, die zusätzlich zur Abbildung der Zeichensätze geschehen muss, denn Leerzeichen und Unterstriche in Content und Attributwerten sind ja durchaus erlaubt.
SQL/XML legt fest, dass ein solches Mapping auf folgende Art und Weise geschehen soll: Unproblematische Zeichen werden einfach übernommen, für Zeichen, die nicht in XML Namen vorkommen dürfen, wird entweder die Zeichenkette _xHHHH_ oder die Zeichenkette _xHHHHHHHH_ eingefügt. HHHH bzw. HHHHHHHH stehen hierbei für die hexadezimale Unicode-Darstellung des entsprechenden Zeichens. Man beachte, dass wegen seiner Verwendung als erstes Zeichen der Ersetzung, auch der Unterstrich selbst auf diese Weise behandelt werden muss.

Neben obiger Abbildung, die als *partially escaped mapping* bezeichnet wird, wird noch eine zweite Möglichkeit definiert, das *fully escaped mapping*. Hierbei werden auch der Doppelpunkt und das Präfix XML behandelt. Der Doppelpunkt wegen seiner Verwendung als Trennzeichen zwischen Namespace und lokalem Namen, das Präfix XML weil es vom W3C für zukünftige Erweiterungen reserviert ist. Der Doppelpunkt wird auf die selbe Weise wie oben abgebildet, dem Präfix XML wird ganz einfach die Zeichenkette _xFFFF_ vorangestellt, wobei es irrelevant ist, ob es sich bei X, M und L um Groß- oder Kleinbuchstaben handelt.
Die umgekehrte Abbildungsrichtung ergibt sich bei beiden Techniken direkt. Wird eine Zeichenkette in einem abzubildenden XML-Namen gefunden, wird diese im resultierenden XML-Bezeichner durch das Unicode-Zeichen ersetzt, welches von der hexadezimalen Zahl identifiziert wird. Im Spezialfall _xFFFF_ wird diese Zeichenkette einfach weggelassen. Abb. 10 zeigt einige beispielhafte Abbildungen.

SQL <identifier>	***fully escaped XML-Mapping***	***partially escaped XML-Mapping***
person	person	person
"telefon"	telefon	telefon
"Zeichen Test"	Zeichen_x0020_Test	Zeichen_x0020_Test
"Zeichen_Test"	Zeichen_x005F_Test	Zeichen_x005F_Test
"Zeichen:Test"	Zeichen_x003A_Test	Zeichen:Test
xmltest	_xFFFF_xmlcol	xmlcol

Abb. 11: Partially und fully escaped mapping

4.2 Abbildungsregeln

Neben den in der obigen Infrastruktur beschriebenen Abbildungsregeln werden in SQL/XML auch solche für Datentypen definiert. Des weiteren wird eine Standardabbildung von relationalen Tabellen nach XML gegeben. Diese beiden Aspekte sollen im folgenden näher erläutert werden.
SQL bietet eine mehr oder weniger breite Anzahl an vordefinierten Datentypen. Auch für diese sollte eine Abbildungsbeziehung zu XML hergestellt werden. Günstigerweise werden in XML Schema Part 2 einige einfache Datentypen definiert, die als Ziel der Abbildung dienen können. Grundsätzlich wird dabei in SQL/XML nach dem Motto gehandelt, dass für jeden vordefinierten SQL-Datentyp der XML Schema-Datentyp ausgewählt wird, der ihm am nächsten ist. "Am nächsten" bedeutet in diesem Zusammenhang, dass sich alle Werte des SQL-Datentyps repräsentieren lassen und die Zahl der zusätzlich im XML Schema-Datentyp darstellbaren Werte so klein wie möglich ist. Beispielweise wird der SQL-Datentyp INTEGER im allgemeinen auf den gleichnamigen XML Schema-Datentyp abgebildet. *XML Schema facets* können dabei dazu benutzt werden, die akzeptierbaren Werte in XML so weit als möglich auf den tatsächlich benötigten Wertebereich zuzuschneiden. Abb. 6 zeigt ein Beispiel, wie dies bei der Abbildung des SQL-Datentyps INTEGER geschehen kann.
Der Annotation-Mechanismus von SQL Schema wird hier verwendet, um Informationen über den Ursprungsdatentyp zu sichern, die ansonsten verloren gehen würden. Diese Verwendung ist allerdings optional. Man beachte, dass minInclusive und maxInclusive abhängig von der Implementation sind.

```
<xsd:simpleType name="INTEGER">
   <xsd:restriction base="xsd.integer">
      <xsd:maxInclusive value="2157483647"/>
      <xsd:minInclusive value="-2157483647"/>
      <xsd:annotation>
         <sqlxml:sqltype name="INTEGER"/>
      </xsd:annotation>
   </xsd:restriction>
</xsd:simpleType>
```

Abb. 12: Beispiel für Abbildung unter Verwendung von XML Schema

Früher war im Rahmen von SQL/XML lediglich die Abbildung von vordefinierten Datentypen vorgesehen, mittlerweile sind aber auch Abbildungen für verschiedene nicht-vordefinierten Datentypen spezifiziert. Dazu gehören solche für DOMAIN, ARRAY, ROW, MULTISET und distinkte benutzerdefinierte Datentypen. Aus Platzgründen wird an dieser Stelle nicht näher auf diese Abbildungen eingegangen. Der interessierte Leser wird in [16] und [18] nähere Informationen finden.
Es bleibt anzumerken, dass sich die Abbildung von konkreten Werten entsprechend der soeben besprochenen Typabbildung ergibt.
Neben dem Mapping der Datentypen wird in SQL/XML definiert, wie relationale Tabellen in XML-Dokumente abgebildet werden sollen. Dabei ist es möglich, sowohl einzelne Tabellen, als auch alle Tabellen in einem Schema oder einem Katalog abzubilden. Dabei werden immer zwei XML-Dokumente erzeugt: Zum einen ein XML-Dokument, dass die in den Tabellen gespeicherten Daten enthält und zum anderen ein XML Schema Dokument, in das die Metadaten abgebildet werden, welche die Tabelle(n) beschreiben.
Zunächst wird im folgenden die Struktur des XML-Dokuments betrachtet, welches die Daten enthält. Wie wir schon in früheren Kapiteln gesehen haben, sind hier verschiedene Ansätze denkbar. Bei SQL/XML hat man sich für folgende Vorgehensweise entschieden (Bemerkung: wenn im folgenden von "gleichen Namen" die Rede ist, sind immer gleiche Namen in bezug auf die Abbildung in 4.2 gemeint): Die Abbildung von Katalogen, Schemas und Tabellen erfolgt gemäß ihrer Hierarchie. Ein Katalog-Element enthält für jedes Schema ein Subelement. Jedes Schema-Element enthält wiederum Subelemente für alle enthaltenen Tabellen. Soll lediglich eine Tabelle abgebildet werden, entfallen die oberen Hierarchien und das entsprechende Tabellen-Element ist das Root-Element. Analog gilt dies für die Abbildung von Schemas. Alle Elemente sind dabei nach den Katalogen, Schemata und Tabellen benannt, die sie repräsentieren. Für jeden Record einer Tabelle wird ein <row>-Element eingefügt. Für jeden Spaltenwert enthält dieses <row>-Element ein Element mit gleichem Namen wie die Spalte (Bemerkung: Für die bekannten Beispieltabellen person und telefon sähe eine solche Abbildung ganz ähnlich aus, wie ihr Default XML View in Kapitel 3)
Neben den bis jetzt festgelegten Regeln, muss auch noch die Abbildung von NULL-Werten explizit behandelt werden. Hierfür definiert SQL/XML zwei Möglichkeiten, von denen eine vom Benutzer ausgewählt werden muss. Die erste Möglichkeit besteht darin, dass dem betroffenen Element ein Attribut xsi:nil="true" hinzugefügt wird. Bei der zweiten wird das betroffene Element einfach ganz weggelassen.
Im XML Schema Dokument, auf das die Metadaten abgebildet werden, wird zunächst für jeden in den Spalten verwendeten SQL-Datentyp ein XML Schema Datenyp definiert (vgl. 4.3, speziell Abb. X). Danach werden für jede Tabelle zwei XML Schema Typen festgelegt,

(Tabelle person wie oben)

resultierendes XML-Dokument:

```
<person>
    <row>
        <id>1</id><name>Andreas</name><alter>20</alter>
    </row>
    <row>
        <id>2</id><name>Christian</name><alter>30</alter>
    </row>
</person>
```

resultierendes XML Schema:

```
<xsd:schema>
    <xsd:simpleType name="INTEGER">
        <xsd:restriction base="xsd:integer">
            <xsd:maxInclusive value="2157483647"/>
            <xsd:minInclusive value="-2157483648"/>
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType name="CHAR_8">
        <xsd:restriction base="xsd:string">
            <xsd:length value="8"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:complexType name="RowType.katalogname.schemaname.person">
        <xsd:sequence>
            <xsd:element name="id" type="INTEGER"/>
            <xsd:element name="name" type="CHAR_8"/>
            <xsd:element name="alter" type="INTEGER"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="TableType.katalogname.schemaname.person">
        <xsd:sequence>
            <xsd:element name="row"
                type="RowType.katalogname.schemaname.person"
                minOccurs="0"
                maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="person"
        type="TableType.katalogname.schemaname.person"/>
</xsd:schema>
```

Abb. 13: Abbildung einer Tabelle in XML

einen der den Aufbau jeder Zeile beschreibt (RowType) und einen, der die Tabelle selbst beschreibt (TableType). Der RowType besteht dabei aus einer Sequenz von XML Schema Elementen, jeweils eins für jede Spalte, für die dann auch ein zuvor definierter XML Schema Datentyp festgelegt wird. Auch der TableType ist als Sequenz definiert, allerdings besteht er aus einer nicht spezifizierten Anzahl von XML Schema Elementen des Typs RowElement.

Die Definition eines XML Schema Elements vom Typ TableType schließt das Dokument ab, sofern nur eine Tabelle abgebildet werden soll (bei Katalogen oder Schemata müssen auch noch für diese Typen und Elemente definiert werden).
Eine interessante Anwendung der soeben besprochenen Abbildung von Tabellen in XML-Dokumente liegt in der Erstellung eines XML Views auf relationale Daten. Dieser View kann wiederum z.B. als Basis für XQuery verwendet werden.

4.3 XML Datentyp

Ähnlich wie beim IBM DB2 XML Extender (vgl. 2.1) wird auch in SQL/XML ein eigener XML-Datentyp eingeführt. Dieser hat ganz einfach den Namen XML. Werte dieses Datentyps können neben XML-Dokumenten auch Elemente, Element-Wälder, Textknoten und vermischter Content sein. Neben dem Datentyp selbst sind einige Funktionen definiert, die Ausprägungen von XML erzeugen können: XMLELEMENT, XMLFOREST, XMLGEN, XMLCONCAT und XMLAGG. Diese werden auch als *publishing functions* bezeichnet.
Die Funktion XMLELEMENT erzeugt genau ein XML-Element pro Record. Als erstes Argument wird ihr der Name des Elementes, das erzeugt werden soll, übergeben. Das zweite Argument ist optional und wird durch das Schlüsselwort XMLATTRIBUTES(...) eingeleitet. Hier können, wie der Name schon vermuten lässt, die zu erzeugenden Attribute angegeben werden. Anschließend wird der Content des Elements angegeben. Abb. 7 zeigt eine beispielhafte Anwendung von XMLELEMENT.

```
SELECT p.id, XMLELEMENT( NAME="Person",
             XMLATTRIBUTES( p.alter as "Jahre"), p.name)
             AS Ergebnis
FROM person p;
```

id	Ergebnis
1	<Person Jahre="20">Andreas</Person>
2	<Person Jahre="30">Christian</Person>

Abb. 14: beispielhafte Anwendung von XMLELEMENT

Eingebettete Element können durch eingebettete XMLELEMENT-Funktionen erreicht werden, z.B. im obigen Beispiel durch Ersetzung von p.name durch einen erneuten Aufruf von XMLELEMENT. Auch die Erzeugung von gemischtem Content ist mit XMLELEMENT möglich, da ein erneuter Aufruf von XMLELEMENT an jeder Stelle des Contents erfolgen kann.
XMLFOREST arbeitet ähnlich wie XMLELEMENT, allerdings können hier mehrere Elemente pro Record erzeugt werden (ein Wald von Elementen, daher der Name). XMLCONCAT bekommt selbst XML-Elemente als Argumente und liefert diese Elemente konkateniert zurück Durch die Verwendung von XMLGEN kann ein Element unter

Verwendung der XQuery Konstruktorsysntax erzeugt werden, XMLAGG kann zur Aggregation von XML über Tupel hinweg verwendet werden.

4.4 SQL/XML Ausblick

Wie bereits oben erwähnt ist die Entwicklung von SQL/XML noch nicht abgeschlossen. Neben anderen Aspekten sollen noch folgende Dinge spezifiziert werden:
Eine Funktion, die automatisch eine Tabelle, ein Schema oder einen Katalog auf XML abbildet und die entsprechenden Dokumente zurückliefert (XMLEXTRACT). Außerdem eine Funktion, welche XML-Dokumente, die sich als Varchar oder CLOB in der Datenbank befinden, in den Datentyp XML parsen kann (XMLPARSE). Des weiteren ist eine Funktion geplant, die aus einem Wert des Typs XML eine Tabelle mit skalaren Spalten aufbaut. In diesem Zusammenhang sind außerdem Prädikate vorgesehen, die entscheiden, um was es sich bei einem Wert vom Typ XML genau handelt (Dokument, Element, Element-Wald etc.). Nicht zu letzt ist eine Funktion geplant, welche die Validität eines Dokuments oder Elements in bezug auf das entsprechende XML Schema überprüfen soll.

5. XML:DB API

Als letzter behandelter Ansatz wird im folgenden kurz die XML:DB API vorgestellt. Es handelt sich dabei um eine Programmierschnittstelle, die einen ähnlich komfortablen Zugang zu nativen XML-Datenbanken ermöglichen soll, wie ihn ODBC und JDBC für relationale Datenbanken bereits bieten. Die XML:DB API wird von einer Gruppe, bestehend aus zu meist eher kleineren Unternehmen, entwickelt, die unter dem Namen XML:DB Initiative bekannt ist. Die Entwicklung ist noch nicht abgeschlossen, eine Referenz-Implementierung in Java steht allerdings bereits zur Verfügung.

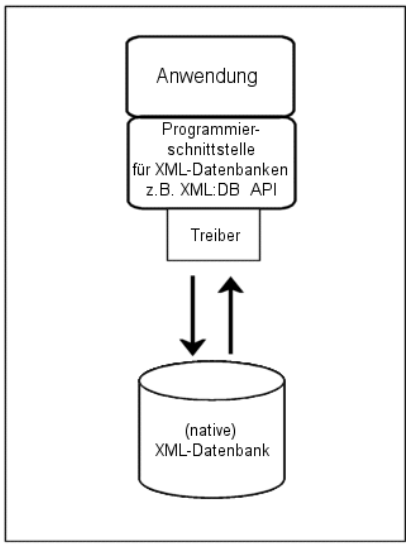

Abb. 15:
XML:DB API Ansatz

Die XML:DB API ist modular aufgebaut, verschiedene Module werden dabei zu sogenannten Core Levels zusammengefasst. Diese Core Levels sind von 0 aufsteigend durchnummeriert,wobei ab Core Level 1 alle Module des jeweiligen Vorgängers enthalten sein müssen. Die Komplexität eines Core Levels steigt also äquivalent mit der Zahl, die es bezeichnet. Konkrete Implementationen können auf Grund der benötigten Module entscheiden, welches Core Level sie verwirklichen wollen (Anmerkung: bisher existieren lediglich Core Level 0 und 1). Grundsätzlich lassen sich drei Arten von Modulen unterscheiden: Das Basis-, die Resource- und die Service-

Module.
Das Basis-Modul ist im Core Level 0 und muss somit von allen Implementationen umgesetzt werden. Es liefert den grundlegenden Rahmen und die Basisschnittstellen für alle anderen Module. Hier ist auch das Collection-Interface definiert. Mit Collections bezeichnet man in nativen XML-Datenbanken Container, in denen XML-Dokumente gespeichert werden. Verglichen mit relationalen Datenbanken könnte man sie grob als Tabellen bezeichnen.
Die in Collections abgelegten Daten werden auch als Ressourcen bezeichnet. Analog zu obigem Vergleich mit relationalen Datenbanken könnte man Ressourcen als Zeilen in einer abelle bezeichnen. Für XML-Daten ist das Interface XMLResource definiert. Dieses ermöglicht textuellen Zugriff auf die zu Grunde liegende Ressource, aber auch die DOM und die SAX API werden hier unterstützt. Des weiteren wird auch zumindest ein anderes Resource-Modul BinaryResource angeboten, das einen Block von Binärdaten oder ein BLOB in der Datenbank repräsentiert.

```
import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;

/**
 * Simple XML:DB API example to query the database.
 */
public class Example1 {
   public static void main(String[] args) throws Exception {
      Collection col = null;
      try {
         String driver = "org.hersteller_x.xmldb.DatabaseImpl";      (1)
         Class c = Class.forName(driver);

         Database database = (Database) c.newInstance();             (2)
         DatabaseManager.registerDatabase(database);                 (3)
         col = DatabaseManager.getCollection                         (4)
         ("xmldb:herstellerx://xmlmovies/movies");

         String xpath = "//movie[@title='Music Man']";
         XPathQueryService service = (XPathQueryService)             (5)
               col.getService("XPathQueryService","1.0");
         ResourceSet resultSet = service.query(xpath);               (6)

         ResourceIterator results = resultSet.getIterator();         (7)
         while (results.hasMoreResources()) {
            Resource res = results.nextResource();
            System.out.println((String) res.getContent());           (8)
         }
      }
      catch (XMLDBException e) {
         System.err.println("XML:DB Exception occured "
               + e.errorCode);
      }
      finally {
         if (col != null) {
            col.close();
         }
      }
   }
}
```

Abb. 16: Beispielanwendung der XML:DB API

Service-Module sind, wie der Name vermuten lässt, Schnittstellen zu angebotenen Diensten, die von der zugrunde liegenden Datenbank befriedigt werden können. Services werden dabei auf Collections definiert. Ein Beispiel für ein solches Service-Modul ist XPathQueryService, welches einen übergebenen XPath-Ausdruck auf einer Collection ausführt. Generell können aber auch für andere Anfragesprachen Service-Module angeboten werden. QueryServices liefern im allgemeinen ein ResourceSet zurück. Auf diesem ResourceSet wird dann ein ResourceIterator definiert, mit dessen Hilfe jede enthaltene Resource iterativ ausgelesen werden kann. Das Beispiel in Abb. 8 soll zur weiteren Verdeutlichung dienen, wobei hier die Referenz-Implementation in Java verwendet wird. Aus die Collection movies in der Datenbank xmlmovies soll hier jeder Film ausgegebn werden, der dessen Titelattribut den Wert "Music Man" besitzt.
Zunächst wird in (1) der benötigte Treiber festgelegt und anschließend ein entsprechendes Objekt erzeugt. Die Implementierung des Treibers ist dabei Sache des Datenbank-Herstellers. Über das Treiber-Objekt wird in (2) ein `Database`-Objekt erzeugt, welches den entsprechende Datenbanktyp repräsentiert. Im `DatabaseManager` werden alle Datenbanktypen registriert, mit denen gearbeitet werden soll (3). Damit ist die Grundlage geschaffen, um mit konkreten Datenbanken des obigen Typs zu arbeiten.
Hier soll mit der Collection movies aus der Datenbank xmlmovies gearbeitet werden. Dazu wird mit Hilfe des `DatabaseManagers` in (4) ein `Collection`-Objekt erzeugt, welches die tatsächlich in der Datenbank vorliegende Collection movies repräsentiert. Auf diesem `Collection`-Objekt wird nun ein `XPathQueryService` definiert (5), der in (6) durch die Methode `query` zum absetzen der zuvor im String `xpath` festgelegten Anfrage verwendet werden kann. Man erhält hier ein `ResourceSet`, welches mit Hilfe eines `ResourceIterators` (7) Ressource für Ressource durchgegangen wird. Jede Ressource wird dabei unter Verwendung der Methode `getContent` in (8) ausgegeben.

6. Zusammenfassung und Bewertung

Wie anfangs erwähnt gibt es auf dem Gebiet der XML-Middleware eine Vielzahl von unterschiedlichen Ansätzen, Philosophien und Vorgehensweisen. Einige wurden hier vorgestellt und sollen abschließend noch einmal zusammengefasst und hinsichtlich ihrer Vor- und Nachteile bewertet werden.
Bei XML-erweiterten Datenbanken wird dem eigentlichen Datenbanksystem eine XML-Mapping-Schicht hinzugefügt, mit deren Hilfe Abbildungen von und nach XML ermöglicht werden. Als Beispiele wurden hier der DB2 XML Extender der IBM und der SQL Server 2000 von Microsoft näher vorgestellt. Der XML Extender unterstützt beide Abbildungsrichtungen und ist auch für komplexe Mappings geeignet, des weiteren wird ein eigener XML-Datentyp angeboten. Auch beim SQL Server 2000 sind beide Abbildungsrichtungen und komplexe Abbildungen möglich. Durch SQLXML-Upgrades wurden hier weitere Erweiterungen eingeführt und die Einbettung in das .Net-Konzept von Microsoft wurde forciert.
Das große Problem von XML-erweiterten Datenbanken besteht aber darin, dass es sich um herstellerspezifische Ansätze handelt, sowohl in bezug auf die Abbildungskonzepte als auch auf die konkret angebotenen Methoden. Tauscht man die zu Grunde liegende Datenbank aus, muss man sich mit völlig neuen Konzepten und Techniken auseinandersetzen, der parallele Einsatz von heterogenen Datenbanken wird erschwert.

Mit XTABLES wurde im 3. Kapitel ein Middleware-System vorgsestellt, welches zwischen Datenbank und Benutzer angesiedelt ist. Es wird zunächst ein Default-XML-View der zu Grunde liegenden Datenbank erzeugt, auf dem dann benutzerdefinierte Sichten erstellt werden können. Mit Hilfe von XQuery können schließlich Anfragen auf diese Sichten formuliert werden. XTABLES hebt sich unter ähnlichen Ansätzen dahingehend hervor, dass die zu Grunde liegende Datenbank dem Benutzer vollkommen verborgen bleibt. Außerdem wird lediglich eine Sprache für die Erzeugung der benutzerdefinierten Views und für das Stellen der Anfragen verwendet, des weiteren werden alle rechen- und speicherintensiven Vorgänge auf die Datenbank verlagert. Ein Rundum-Lösung kann XTABLES jedoch nicht sein, denn es wird lediglich die Abbildung von relationalen Daten in XML unterstützt, nicht aber die umgekehrte Richtung.
Bei SQL/XML handelt es sich um eine Erweiterung von SQL, mit deren Hilfe die Beziehung zwischen SQL und XML standardisiert werden soll. Einige Funktionalitäten der anderen Ansätze können dabei wieder gefunden werden: Auch hier wird ein eigener XML-Datentyp eingeführt (vgl. DB2 XML Extender) und Abbildungen in beide Richtungen werden unterstützt. Des weiteren können XML-Views erzeugt werden, die Abbildung von relationalen Daten in XML entspricht hierbei im wesentlichen dem Mapping, das bei XTABLES zur Erzeugung des Default-XML-Views verwendet wird.
SQL/XML befindet sich zur Zeit noch in der Entwurfsphase. Da SQL weit verbreitet ist, ist auch eine breite Annahme von SQL/XML zu erwarten. Dadurch kann es gelingen, die Beziehung zwischen Datenbank-Welt und XML-Welt weitgehend zu standardisieren.
Die XML:DB API fällt im Vergleich mit den anderen Ansätzen ein bisschen aus dem Rahmen: Zum einen handelt es sich um eine Programmierschnittstelle, zum anderen geht es hier um XML-native und nicht um relationale Datenbanken. Mit Hilfe der XML:DB API soll dabei die gleiche Funktionalität angeboten werden, die ODBC und JDBC für relationale Datenbanken bereits bieten. Die API ist modular aufgebaut und beliebig erweiterbar, wodurch sie nicht auf bestimmte Anfrage- und Update-Sprachen festgelegt ist. Es bleibt allerdings abzuwarten, ob sich die XML-nativen Datenbanken auf längere Zeit halten können, denn je ausgefeilter die Ansätze zur Integration von relationaler Welt und XML werden, desto geringer wird ihre Existenzberechtigung. Außerdem muss angemerkt werden, dass die Entwicklung der XML:DB API seit längerem keine Fortschritte gemacht hat.

Anhang A: Literaturangaben

Gemeinsame Quellen:

[1] Ronald Bourret,
XML and Databases (Last updated Feb. 2002)
http://www.rpbourret.comxml/XMLAndDatabases.htm

[2] Dare Obasanjo,
An exploration of XML in database management systems (2001)

[3] Prof. Dr. Stefan Deßloch,
Skript zur Vorlesung Middleware für verteilte Systeme im SS 2002
http://wwwdbis.informatik.uni-kl.de

IBM DB2 XML Extender:

[4] Luis Ennser, Christophe Delporte, Mikio Oba, K.M. Sunil
Integrating XML with DB2 XML Extender and DB2 Text Extender
IBM Redbook, Dezember 2000
http://www.ibm.com/redbooks

[5] Josephine Chang, Jane Xu
IBM DB2 XML Extender – An end-to-end solution for storing and retrieving XML documents
ICDE '00 Conference, San Diego, February 2000

[6] Tiina Pihlasviita
XML and DB2 (2002)

Microsoft SQL Server 2000:

[7] Rob Vieira
SQL Server 2000/XML Integration – Where the data meets the road
Proceedings of the Aworx conferences

[8] The MSDN Library
Microsoft SQL Server 2000 and subtopics
http://msdn.microsoft.com/

[9] Unbekannt
SQL Server 2000 XML & .NET Integration Features
http://www.perfectxml.com/sqlxml.asp

[10] Darshan Singh
OPENXML
http://www.perfectxml.com/Articles/XML/OPENXML.asp

[11] Darshan Singh
Importing XML into SQL Server 2000
http://www.perfectxml.com/articles/XML/ImportXMLSQL.asp

[12] Darshan Singh
Exporting SQL Data as XML
http://www.perfectxml.com/articles/XML/ExportSQLXML.asp

XTABLES:

[13] C. Fan, J. Funderburk, H. Lam, J. Kiernan, E. Shekita, J. Shanmugasundaram
XTABLES: Bridging Relational Technology and XML
http://www.cs.cornell.edu/People/jai/papers/XTABLES.pdf

[14] M. Carey, J. Kiernan, J. Shanmugasundaram, E. Shekita, S. Subramanian
XPERANTO: A Middleware for Publishing Object-Relational Data as XML-Documents
http://www.cs.cornell.edu/People/jai/papers/XperantoDemo.pdf

[15] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, S. Subramanian
XPERANTO: Publishing Object-Relational Data as XML
http://www-db.cs.wisc.edu/dbseminar/spring01/talks/jai2.pdf

SQL/XML:

[16] Jim Melton (Editor)
(ISO-ANSI Working Draft) XML-Related Specifications (SQL/XML)
WG3:DRS-020 / H2-2002-365 (August 2002)
http://www.sqlx.org/5wd-14-xml-2002-08.pdf

[17] Jim Melton, Andrew Eisenberg
SQL/XML and the SQLX Informal Group of Companies
http://www.acm.org/sigmod/record/issues/0109/standards.pdf

[18] Jim Melton, Andrew Eisenberg
SQL/XML is Making Good Progress
http://www.acm.org/sigmod/record/issues/0206/standards.pdf

XML:DB API:

[19] Kimbro Staken (Editor)
XML:DB API Working Draft
http://www.xmldb.org/xapi/xapi-draft.html

[20] Kimbro Staken
An Introduction to the XML:DB API
http://www.xml.com/pub/a/2002/01/09/xmldb_api.html

[21] Lars Martin
Teamgeist! – Entwicklung neuer Standards für XML-Datenbanken
http://www.linux-magazin.de/Artikel/ausgabe/2001/04/XMLDB/xmldb.html