

XML-RDB Mapping

Kontroverse Strukturen

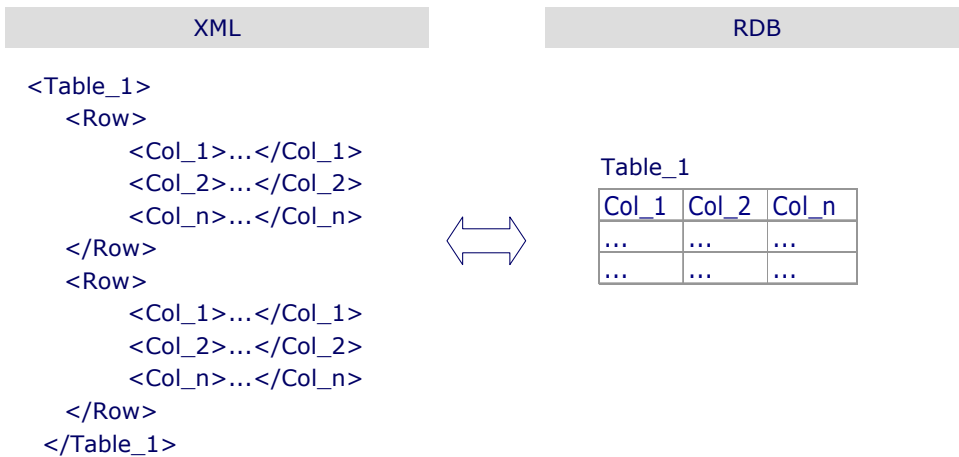
XML

- Hierarchisch nach inhaltlichen Zusammenhängen.
- Beliebige hierarchische Tiefe
- "Single Rooted".
- Strukturinformation und Daten nahe beieinander.
- Veränderung der Strukturen während Lebenszeit der Daten.
- Zusammenhänge zwischen Daten durch Hierarchie gegeben

RDB

- Tabellarisch nach Normalisierung, d.h. Redundanzfreiheit.
- 2-stufige Hierarchie, Tabellen und Spalten.
- Unabhängige Tabellen.
- Strukturinformation und Daten vollständig getrennt.
- Lebenslang gleiche Struktur.
- Datenelemente durch Schlüssel-Beziehungen untereinander verknüpft

Table-Based Mapping



Prinzip

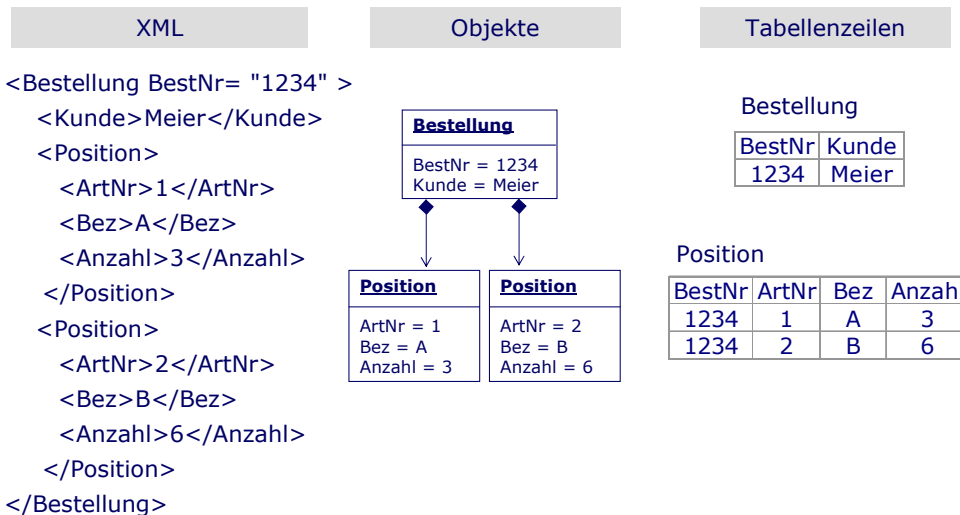
XML-Elemente haben Strukturnamen der RDB-Strukturelemente (zum Beispiel Table, Row, Col). Die XML-Elemente bezeichnen *nicht* die Art der Daten aus der Sicht einer bestimmten Applikation (zum Beispiel Kunde, Bestellung, Artikel).

Eine wesentlicher des Table-Based Mapping ist, dass die eine Darstellung ist aus der anderen leicht erzeugbar ist. Table-Based Mapping eignet sich gut für den Transfer von Daten zwischen Datenbanken und für Import/Export-Aufgaben. XML-Attribute können benutzt werden, um Hilfsinformation über die Daten auszudrücken, beispielsweise über Datentyp, erlaubte Nullwerte, Minimum- und Maximumwert, Schlüsselfunktionen. Im einfachsten Fall können XML-Attribute auch benutzt werden, um die Daten aufzunehmen. Die Attributnamen entsprechen dann den Spaltennamen in der Tabelle. Diese Variante ermöglicht sehr einfache Programme für die Konvertierung zwischen der XML- und der Tabellendarstellung.

Die Flexibilität von XML ist beim Table-Based Mapping stark eingeschränkt: Die Verschachtelungstiefe der Elemente ist exakt 3, pro Zeile sind genau n vordefinierte Spalten erlaubt.

Eine RDB-Struktur ist sehr stark auf Redundanzfreiheit ausgelegt und nicht darauf, wie stark die einzelnen Informationsteile zusammengehören. Eine XML-Struktur beruht auf der hierarchischen Gliederung zusammengehöriger Information. Sobald seitens der RDB mehr als eine Tabelle im Spiel ist, wird offensichtlich, dass die Idee der XML-Struktur im Table-Based Mapping nicht zum Tragen kommt.

Object-Relational Mapping (1)



Prinzip

Anhand der XML-Struktur werden Teilbäume definiert, die zu Datenobjekten werden. Diese Datenobjekte werden ihrerseits zu Tabellenzeilen. Bei bereits bestehenden Datenbanken kann natürlich gerade der umgekehrte Weg eingeschlagen werden.

In diesem Schema können insbesondere Nicht-XML Applikationen und XML-Applikationen sehr gut auf eine gemeinsame Datenbasis zurückgreifen.

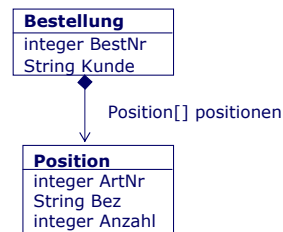
Object-Relational Mapping (2)

DTD

Referenz auf terminales Element	→	Basis-Datentyp als Attribut
Referenz auf zusammengesetztes Element	→	Referenz-Datentyp auf Klasse
Attribut	→	Basis-Datentyp als Member
Zusammengesetztes Element	→	Klasse

Klassendiagramm

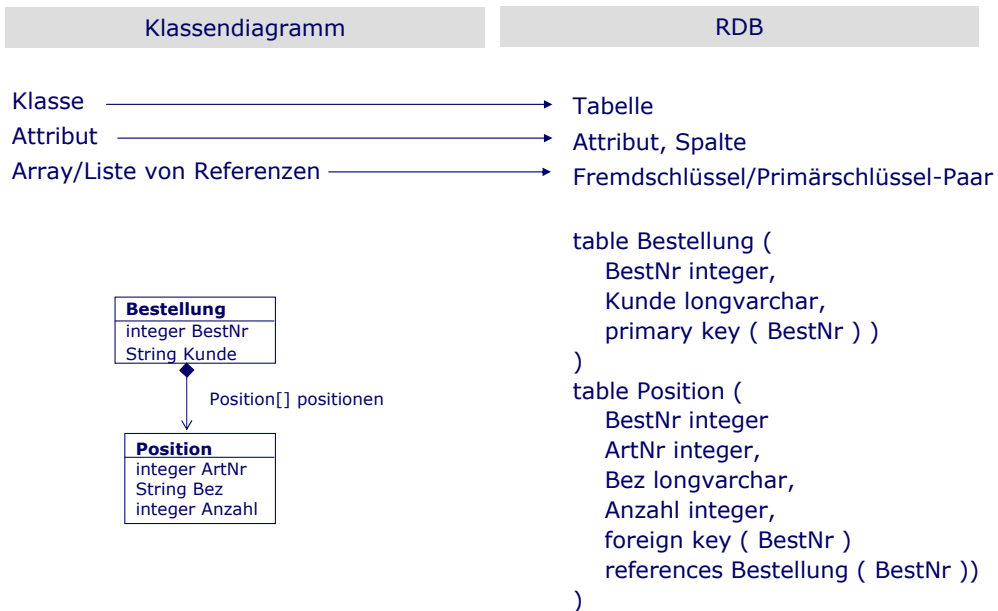
```
<!ELEMENT Bestellung( Kunde, Position+ )>
<!ATTLIST Bestellung BestNr CDATA
    #REQUIRED>
<!ELEMENT Kunde (#PCDATA)>
<!ELEMENT Position (ArtNr, Bez, Anzahl)>
<!ELEMENT ArtNr (#PCDATA)>
<!ELEMENT Bez (#PCDATA)>
<!ELEMENT Anzahl (#PCDATA)>
```



Ein terminales Element kann auch als Klasse modelliert werden, wenn es sich um einen zentralen Begriff in der Applikation handelt oder wenn zu erwarten ist, dass aus dem terminalen Element in Kürze ein zusammengesetztes Element entsteht, weil weitere Informationen festgehalten werden muss. In diesem Beispiel wäre für das Element `Kunde` sicher zum Vornherein eine eigene Klasse angebracht.

Da mit DTD's kein anderer Basis-Datentyp als `#PCDATA` definiert werden kann, muss für die genaue Spezifikation des Datentyps in der OO-Struktur noch irgendwo eine Information eingeholt werden. Wenn anstelle eines DTD ein XMLSchema verwendet wird, ist das Mapping präziser und kann besser automatisiert werden.

Object-Relational Mapping (3)



Prinzip

Auf der RDB-Seite gilt immer noch die Grundsätze der Normalisierung und der Redundanzfreiheit. Diese stehen oft im Gegensatz zu intuitiven Annahmen. Beispielsweise ist in der Tabelle 'Bestellung' keine Liste von Referenzen auf die einzelnen Bestellpositionen mehr zu finden, sondern die Referenzierung geschieht von der Tabelle 'Position' her: Jede Position bezieht sich auf eine bestimmte Bestellung. Würde man ganz hypothetisch annehmen, dass eine Position zu mehreren Bestellung gehören kann, müsste eine dritte Tabelle 'Position-Bestellung' definiert werden, welche Paare von Positionsnummer und Bestellnummer enthält. Diese zeigen, welche Position zu welchen Bestellungen gehört und umgekehrt. Im Klassendiagramm sind nach wie vor nur zwei Klassen im Spiel. Es wird lediglich die Komposition von Bestellung zu Position zu einer Aggregation.

In einer Datenbank werden oft zusätzliche Schlüssel verwendet, welche keine eigentlichen Nutzdaten sind und nur innerhalb *einer lokalen* Datenbank Gültigkeit haben. Lässt man die Schlüssel beim Generieren von XML-Daten weg, entsteht die Schwierigkeit, dass die XML-Daten später nicht mehr eindeutig in die Datenbank übernehmbar sind (weil sie vielleicht leicht modifiziert wurden). Übernimmt man sie in die XML-Daten, sind sie dort für andere als die lokale Datenbank nutzlos -> nicht behebbare Unschönheit.

Die Stufe Klassendiagramm ist konzeptionell. Sie kann, aber muss nicht realisiert werden.

ORM, Sequenzen

DTD	Klassen	Tabellen
<pre><!ELEMENT A (B, C)> <!ELEMENT B (#PCDATA)></pre>	<pre>class A { String b; C c; }</pre>	<pre>table A (b longvarchar, <i>c_key integer</i>)</pre>
<pre><!ELEMENT C (D, E)> <!ELEMENT D (#PCDATA)> <!ELEMENT E (#PCDATA)></pre>	<pre>class C { String d; String e; }</pre>	<pre>table C (<u>c_key integer</u>, d longvarchar, e longvarchar)</pre>

Primärschlüssel in der Tabellendefinition sind unterstrichen, Fremdschlüssel kursiv geschrieben.

ORM, Alternativen

DTD	Klassen	Tabellen
<pre><!ELEMENT A (B C)> <!ELEMENT B (#PCDATA)></pre>	<pre>class A { String b; C c; }</pre>	<pre>table A (b longvarchar null, c_key integer null)</pre>
<pre><!ELEMENT C (D, E)> <!ELEMENT D (#PCDATA)> <!ELEMENT E (#PCDATA)></pre>	<pre>class C { String d; String e; }</pre>	<pre>table C (c_key integer, d longvarchar, e longvarchar)</pre>

Ein Constraint auf Tabelle A ist notwendig:

entweder *b* oder *c_key* muss null sein

Wenn anstelle von
<!ELEMENT A (B | C)> die Definition <!ELEMENT A (B?, C?)> stünde, entfällt einzig der Constraint auf
Tabelle A.

ORM, Iterationen

DTD	Klassen	Tabellen
<pre><!ELEMENT A (B, B, B)> <!ELEMENT B (#PCDATA)></pre>	<pre>class A { String[3] b; }</pre>	<pre>table A (b1 longvarchar, b2 longvarchar, b3 longvarchar)</pre>
<pre><!ELEMENT A (B*)> <!ELEMENT B (#PCDATA)></pre>	<pre>class A { String[] b; }</pre>	<pre>table A (<u>a_key integer</u>) table B (a_key integer, b longvarchar)</pre>

Wenn mindestens *ein* B in A vorkommen muss
(B+) ist ein
entsprechender Constraint auf Tabelle A zu
definieren.

ORM, Untergruppen

DTD	Klassen	Tabellen
<pre><!ELEMENT A (B, (C D*))> <!ELEMENT B (#PCDATA)> <!ELEMENT C (X)> <!ELEMENT D (Y)> <!ELEMENT X (...)> <!ELEMENT Y (...)></pre>	<pre>class A { String b; C c; D[] d; } class C { X x; } class D { Y y; }</pre>	<pre>table A (<u>a_key</u> integer, b longvarchar, c_key integer null, d_key integer null) table C (<u>c_key</u> integer, x_key integer,) table D (a_key integer null, y_key integer,) ...</pre>

Zu beachten ist, dass die Untegruppen (dargestellt durch die Klammerung) im Klassenmodell und im Tabellenmodell "verlorengehen". Im Prinzip wäre es möglich, "anonyme" Klassen und Tabellen einzuführen, welche die Gruppierung repräsentieren. Gerade in der Datenbank ist es jedoch fraglich, ob diese zu einer übersichtlicheren Struktur führen. Sicher ist, dass sie das Prinzip der globalen Normalisierung verletzen würden, das heisst die Anforderung, möglichst *wenig* Tabellen in einer Datenbank zu haben (gleichzeitig jedoch die 3. Normalform zu gewährleisten).

ORM, Mixed Content (1)

- Mixed Content = XML-Elemente und Text in beliebiger Reihenfolge gemäss dem DTD

```
<!ELEMENT M ( #PCDATA | A | B | C )* >
```

- Speziell für Dokumente mit gemischten Fliesstext- und Markup-Elementen, zum Beispiel bei HTML:

```
<P>Text mit <B>fetten</B>oder<I>kursiven</I>Worten</P>
```

- Mixed Content ist eine Abart des DTD

```
<!ELEMENT M ( text | A | B | C )* >  
<!ELEMENT text ( #PCDATA ) >
```

An erster Stelle in der Elementdefinition von 'mixed' muss immer #PCDATA stehen, anschliessend dürfen beliebig viele andere Elemente vorkommen.

ORM, Mixed Content (2)

DTD	Klassen	Tabellen
<pre><!ELEMENT M (#PCDATA A)*> <!ELEMENT A (#PCDATA)></pre>	<pre>class M { String[] pcddata; A[] a; }</pre>	<pre>table M (<u>m_key integer</u>) table PCDATA (m_key integer, pcddata longvarchar) table A (m_key integer, a longvarchar)</pre>

Dieser Ansatz führt bei der Zerlegung vieler und längerer Mixed Content Dokumente relativ rasch zu einer grossen Anzahl Datensätzen. Er ist daher nur dort geeignet, wo XML-Dokumente eher zu kleinen Teilen aus Mixed Content bestehen. In einem Publikations-Verzeichnis könnte zum Beispiel die Kurz-Zusammenfassung (Abstract) in obiger Form abgelegt sein.

Grosse Mixed Content Dokumente werden häufig nur bis zu einer bestimmten Granularität zerlegt. Zum Beispiel kann ein Buch gemäss folgendem DTD

```
<!ELEMENT Buch (Kapitel)* >
<!ELEMENT Kapitel (Abschnitt)* >
<!ELEMENT Abschnitt (PCDATA | Markup)* >
<!ELEMENT Markup ( HTMLTag ) >
```

nur in einzelne Kapitel zerlegt werden anstatt bis zu den einzelnen Textfragmenten (PCDATA) und HTML-Tags. Jedes Kapitel belegt in der Datenbank dann einen Datensatz in einer Tabelle.

Erhaltung der Reihenfolge (1)

- Technisch habe alle XML-Elemente in einem Dokument eine bestimmte Reihenfolge
 - welche zufällig zustandegekommen ist
 - welche der DTD oder das XMLSchema so verlangt
- Logisch kann die Reihenfolge wichtig sein oder unerheblich
 - In datensatzorientierten Dokumenten ist sie meist unwichtig
 - In textorientierten Dokumenten ist zwingend notwendig
- Information über die Reihenfolge muss in die DB eingetragen werden, wenn ein XML-Dokument gespeichert und später in der exakt gleichen Reihenfolge wieder extrahiert werden soll.
- Die Reihenfolge-Information in einer relationalen Datenbank muss ausdrücklin in zusätzlichen Datenfeldern abgelegt werden.

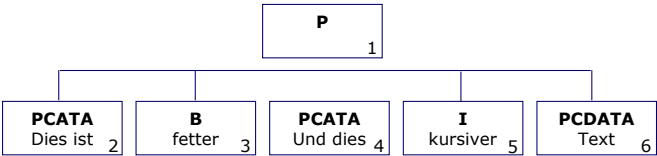
DTDs, welche eine bestimmte Reihenfolge verlangen, sind wesentlich einfacher zu schreiben, als reihenfolge-unabhängige. Oft muss deshalb ein XML-Dokument die Elemente in einer bestimmte Reihenfolge enthalten, obwohl dies von der Semantik her keineswegs notwendig ist. Beispielsweise ist es absolut unerheblich, in welcher Reihenfolge Name, Vorname und Geburtstag einer Person in einer Datenbank oder einem XML Dokument stehen. Der Informationsgehalt ist immer derselbe. Allenfalls ist bei der Darstellung in einem Benützer-Interface auf die Reihenfolge zu achten. Sind in einem Textdokument jedoch Worte oder Sätze umgestellt, wird es wertlos.

Die Reihenfolge der Elemente in einem XML-Dokument ist fix gegeben. Ein XML-Dokument wird ja meist als Datenstrom verschickt oder ist als Datei abgelegt. Werden die Elemente in einer relationalen Datenbank abgelegt, muss die Reihenfolge der erzeugten Datensätze (Tabellenzeilen) keineswegs der Reihenfolge entsprechen, wie sie auf dem Speichermedium abgelegt, oder bei einer SQL-Abfragen wieder ausgegeben werden. Daher muss zwingend ein Wert in einem Reihenfolge-Feld mitgegeben werden, nach dem später bei einer SQL-Abfrage sortiert werden kann.

Erhaltung der Reihenfolge (2)

<!ELEMENT P (#PCDATA|B|I)*>
<!ELEMENT B (#PCDATA) >
<!ELEMENT I (#PCDATA) >

<P>dies ist fetter und dies
<I>kursiver</I> Text</P>



P	
p_key	
1	

PCDATA			
p_key	pcdata	pos	
1	dies ist	2	
1	und dies	4	
1	Text	6	

B		
p_key	b	pos
1	fetter	3

I		
p_key	i	pos
1	kursiver	5

Die Numerierung erfolgt gemäss einer pre-order-Traversierung des Element-Baumes: Ein XML-Parser teilt jedem Element aus dem XML-Verarbeitungsstrom sofort eine fortlaufende Zahl zu.

Mapping von Attributen

- Attribute mit Einfachwerten (CDATA, ID, IDREF, NMTOKEN)

<!ELEMENT A>	class A {	table A (
<!ATTLIST A	String a1;	a1 longvarchar,
a1 CDATA #REQUIRED	String a2;	a2 longvarchar null
a2 CDATA >	}	)

- Attribute mit Mehrfachwerten (IDREFS, NMTOKENS, Aufzählung)

<!ELEMENT A>	class A {	table A (
<!ATTLIST A	String[] a1;	<u>a key integer,</u>
a1 NMTOKENS #REQUIRED	}	)
>		table a1 (
		<u>a_key integer,</u>
		a1 longvarchar,
		pos integer
		)

Fakultative Attribute

Ein fakultatives Attribut mit Einfachwert wird in der Datenbank-Tabelle als Feld mit möglichem Nullwert geführt. Ein fakultatives Attribut mit Mehrfachwert hat, wenn es in einem XML-Element nicht vorkommt, in der entsprechenden Attributtabelle einfach keine Einträge.

Reihenfolge

Die Reihenfolge der Attribute ist per Definition in XML unerheblich. Hingegen kann die Reihenfolge der *Werte* in einem Attribut mit Mehrfachwerten sehr wohl wichtig sein, beispielsweise eine Aufzählung aller Vornamen einer Person, eine Aufzählung von Wochentagen oder Monatsnamen usw. Daher ist ein Attribut 'pos' in der Datenbank vorzusehen, das die im XML-Dokument gegebene Reihenfolge in der Datenbank konserviert.

ID, IDREF und IDREFS Attribute

Eine ID definiert einen eindeutigen Schlüsselwert innerhalb eines XML-Dokumentes. Eine IDREF bezieht sich auf einen eindeutigen und existierenden Schlüsselwert innerhalb eines XML-Dokumentes. Die Abbildung in eine Datenbank könnte daher mit Primärschlüssel-Fremdschlüssel Feldern erfolgen. Sehr häufig enthält aber eine Datenbank die Daten von mehr als einem XML-Dokument und die Eindeutigkeit der Primärschlüssel ist daher für die Datenbank nicht garantiert. Die Lösung besteht darin, dass man in den Datenbank-Tabellen Primärschlüssel und Fremdschlüssel aus zwei Feldern zusammen setzt, nämlich dem ID-Feld respektive dem IDREF-Feld und einem Feld welches das Dokument eindeutig identifiziert. Eine noch einfachere Lösung besteht darin, dass man direkt den Wert des ID und IDREF-Feldes mit einem Präfix erweitert, dass eindeutig für das Dokument ist. Bei Anwendungssoftware, die XML-Dokumente für den Transfer von Daten zwischen Prozessen (Datenbanken) benutzt, wird oft verzichtet auf die Verwendung von ID- und IDREF-Attributen. Man verwendet gewöhnliche CDATA-Attribute, welche von der Semantik her Schlüsselwerte enthalten. Die Generierung, Erkennung und Kontrolle dieser Schlüsselwerte wird komplett den Datenbank-Schichten der beteiligten Applikationen überlassen und nicht den XML-Parsern.

Zusammenfassung

1. Aus komplexen Elementtypen werden Tabellen
2. Aus einfachen Elementtypen werden Tabellenfelder
3. Aus einfachwertigen Attributen werden Felder in der entsprechenden Element-Tabelle
4. Aus mehrfachwertigen Attributen werden eigene Tabellen
5. Die Reihenfolge von Elementen muss in Tabellen mit zusätzlichen Feldern nachgebildet werden
6. Die Beziehungen unter den Elementen sind durch Primärschlüssel / Fremdschlüssel Beziehungen in den Tabellen abzubilden
7. Optionale einfache Elementtypen oder Attribute können durch eine Null-Klausel in der Datenbank-Tabelle abgebildet werden.
8. XML-seitige Default- und Vorgabewerte können in Datenbanken ebenfalls in der Datenstruktur definiert werden.

Das unumstössliche Prinzip innerhalb der Datenbank ist in jedem Fall die Normalisierung der Tabellen.

Unter Umständen kann eine grössere XML-Substruktur in der Datenbank als einziges Feld abgelegt werden. Das vereinfacht die Datenbankstruktur und erhöht ziemlich sicher die Performance der Datenbank, hat jedoch folgende Einschränkungen:

- Die Daten liegen ausschliesslich im XML-Format vor.
- Abfrage der Daten aufgrund bestimmter Bedingungen über den Inhalt ist nur schwer möglich.
- Entnahme einer Teilstruktur ist nur schwer möglich.

Die letzten beiden Einschränkungen versuchen die Datenbank-Hersteller zu umgehen, indem innerhalb von SQL auch XQL- oder XPath-Abfragen auf Tabellenfelder mit XML-Inhalten angeboten werden. Das Angebot ist verführerisch, führt bei grösseren Tabellen jedoch wieder sehr rasch zu Performance-Problemen, weil die XQL- und XPath Abfragen im Gegensatz zu vielen SQL-Abfragen durch den Query-Optimizer nicht optimierbar sind.

Produkte

1. **Oracle XSU** transformiert XML-Dokumente in eine objektrelationale DB-Struktur (verschachtelte Tabellen) und umgekehrt.
2. **DB2** verwendet einen DTD und eine manuell erstellte Data Access Definition (DAD) für die Übersetzung XML-Dokumenten in die Datenbank und umgekehrt.
3. **SQL-Server** verwendet spezielle SQL-Konstrukte für die Extraktion und das Einfügen von XML-Daten. Die Struktur-Information wird dem jeweiligen SQL-Befehl mitgegeben.
4. **Sybase** verwendet Java als serverseitige Programmiersprache. In SQL-Befehlen können Java-Methoden direkt aufgerufen werden. Damit ist auch der Weg frei für das Arbeiten mit XML-Daten über entsprechende Klassen.

Mit der Oracle-Methode kann ein verschachteltes XML-Dokument in eine exakt gleich verschachtelte, objekt-relationale Tabellenstruktur abgebildet werden. Korrespondierende Tabellenfelder und XML-Elemente müssen dabei denselben Namen haben. Die Transformation von Attributen ist nicht möglich. Es wird keine *explizite* Struktur-Information (DTD) verwendet.

DB2 verwendet einen DTD und weil dieser ungenügend Information (zum Beispiel über den Datentyp) enthält, eine ergänzende in XML geschriebene Data Access Definition. Der DAD wird anhand des DTD erzeugt und durch den Administrator fertiggestellt. Das Extrahieren und Einfügen von XML-Daten geschieht über Stored Procedures.

Aufgabe 1

- Erstellen Sie ein Datenbank-Modell aus folgendem DTD
- (Siehe Notiz-Teil dieser Folie)

```
<!ELEMENT catalog (book*)>
```

```
<!ELEMENT book (title,subtitle?,author+,  
                publisher,edition,year,binding?,pages?,isbn)>
```

```
<!ELEMENT title      (#PCDATA)>
```

```
<!ELEMENT subtitle   (#PCDATA)>
```

```
<!ELEMENT author     (#PCDATA)>
```

```
<!ELEMENT publisher   (#PCDATA)>
```

```
<!ELEMENT edition     (#PCDATA)>
```

```
<!ELEMENT year        (#PCDATA)>
```

```
<!ELEMENT binding     (#PCDATA)>
```

```
<!ELEMENT pages       (#PCDATA)>
```

```
<!ELEMENT isbn        (#PCDATA)>
```

Aufgabe 2

- Erstellen Sie ein Datenbank-Modell aus folgendem DTD (Siehe Notiz-Teil dieser Folie)

```
<!ELEMENT catalog (publisher*,author*,book*)>

<!ELEMENT publisher EMPTY>
<!ATTLIST publisher id ID #REQUIRED
                    name CDATA #REQUIRED
                    url CDATA #REQUIRED>

<!ELEMENT author EMPTY>
<!ATTLIST author id ID #REQUIRED
                 firstname CDATA #REQUIRED
                 middlename CDATA #IMPLIED
                 lastname CDATA #REQUIRED>

<!ELEMENT book EMPTY>
<!ATTLIST book id ID #REQUIRED
               title CDATA #REQUIRED
               subtitle CDATA #IMPLIED
               authors IDREFS #REQUIRED
               publisher IDREF #REQUIRED
               edition CDATA "1st edition"
               year CDATA #REQUIRED
               binding (Paperback|Hardcover) #IMPLIED
               pages CDATA #IMPLIED
               isbn CDATA #REQUIRED>
```