Abbildung von XML-Dokumenten auf SQL:2003-konforme Datentypen

von
Dipl.-Math. Kai Schweinsberg

Dissertation
vorgelegt am Fachbereich Elektrotechnik/Informatik
der Universität Kassel
zur Erlangung des Doktorgrades der Naturwissenschaften
(Dr. rer. nat.)

Kassel 2012

Erstgutachter: Prof. Dr. Lutz Wegner, Universität Kassel

Zweitgutachter: Prof. Dr. Klaus Küspert, Friedrich-Schiller-Universität Jena

Datum der Disputation: 15. Juni 2012

Erklärung

Hiermit versichere ich, dass ich die vorliegende Dissertation selbstständig und ohne unerlaubte Hilfe angefertigt und andere als die in der Dissertation angegebenen Hilfsmittel nicht benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen sind, habe ich als solche kenntlich gemacht. Kein Teil dieser Arbeit ist in einem anderen Promotions- oder Habilitationsverfahren verwendet worden.

Kassel, 5. April 2012

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Jim Gray schreibt in seinem Vorwort zu dem Buch *Data on the Web: From Relations to Semistructured Data and XML* [ABS00], dass es in den vergangenen Jahren zu einer Kollision dreier Kulturen gekommen ist: „Alles ist eine Tabelle", „Alles ist ein Objekt" und „Alles ist ein Dokument". Diese drei Paradigmen repräsentieren die relationalen Datenbanken, die objektorientierten Programmiersprachen und die Web-Technologien.

Der XML-Standard spannt eine Brücke zwischen diesen Betrachtungsweisen und hat sich so zu einem etablierten Standard-Datenaustauschformat entwickelt. Zunehmend zeigt sich jedoch der Bedarf, XML-Dokumente nicht nur zum Transfer zu verwenden, sondern sie auch persistent abzulegen. Dies kann unter anderem in speziellen XML- oder relationalen Datenbanken geschehen.

Relationale Datenbanken setzen dabei bisher auf zwei grundsätzlich verschiedene Verfahren: Die XML-Dokumente werden entweder unverändert als binäre oder Zeichenkettenobjekte (BLOBs oder CLOBs) gespeichert oder aber aufgespalten, sodass sie in herkömmlichen relationalen Tabellen normalisiert abgelegt werden können (so genanntes „Flachklopfen" oder „Schreddern" der hierarchischen Struktur).

Aufbauend auf den Erfahrungen des Lehrstuhls von Prof. Wegner mit komplexen Objektstrukturen, insbesondere dem NF²-Datenmodell[1] [Weg89, Weg91, The96, Tha99], wird in dieser Arbeit ein neuer Ansatz verfolgt, der einen Mittelweg zwischen den bisherigen Lösungen darstellt und die Möglichkeiten des weiterentwickelten SQL-Standards aufgreift. SQL:2003 bietet komplexe Struktur- und Kollektionstypen (Tupel, Felder, Listen, Mengen, Multimengen), die es erlauben, XML-Dokumente derart auf relationale Strukturen abzubilden, dass der hierarchische Aufbau erhalten bleibt.

1. Non-First-Normal-Form

Dies bietet zwei Vorteile: Einerseits stehen bewährte Technologien, die aus dem Bereich der relationalen Datenbanken stammen, uneingeschränkt zur Verfügung. Hierzu gehören Funktionen, wie Mehrbenutzerfähigkeit, Transaktionskonzept und ACID-Eigenschaften, die durch spezielle XML-Datenbanken meist nicht in vollem Umfang unterstützt werden. Andererseits lässt sich mit Hilfe der SQL:2003-Typen die inhärente Baumstruktur der XML-Dokumente bewahren, sodass es nicht erforderlich ist, diese im Bedarfsfall durch Joins aus den meist normalisierten und auf mehrere Tabellen verteilten Tupeln zusammenzusetzen.

1.2 Ziele dieser Arbeit

In dieser Arbeit sollen zunächst die grundsätzlichen Fragen zu passenden, effizienten Abbildungsformen von XML-Dokumenten auf SQL:2003-konforme Datentypen geklärt werden. Auf dieser Basis soll ein geeignetes, umkehrbares Umsetzungsverfahren entworfen werden, das schließlich im Rahmen einer prototypischen Applikation implementiert und analysiert werden kann.

Da die Unterstützung von SQL:2003 in den kommerziellen Datenbankmanagementsystemen (DBMS), wie zum Beispiel Oracle, Informix oder DB2, bisher nur unvollständig ist, muss vor der Implementierung untersucht werden, inwieweit die von den einzelnen Systemen unterstützte Untermenge genügt, um eine derartige Abbildung von XML-Dokumenten durchführen zu können. Es ist das Ziel, die prototypische Applikation auf Basis eines ausgereiften relationalen DBMS zu entwickeln. Daher muss das entworfene Abbildungsverfahren gegebenenfalls angepasst werden, sodass es mit diesem System zusammenarbeitet.

Von Interesse sind schließlich auch die Vor- und Nachteile des hier vorgeschlagenen Ansatzes im Vergleich zu den bisherigen Verfahren sowie der Zeit- und Speicherbedarf, den die prototypische Applikation benötigt.

1.3 Gliederung

Die vorliegende Arbeit gliedert sich wie folgt: In Kapitel 2 finden sich eine Einführung in die Grundlagen von XML, in die Dokumentschemabeschreibungssprachen DTD und XML-Schema sowie eine kurze Besprechung des XPath-Datenmodells. Daran schließt sich in Kapitel 3 die Behandlung von objektrelationalen Datenbanken und des SQL-Standards an. Hierbei sind insbesondere der Funktionsumfang der SQL:2003-Norm und die von den Datenbankmanagementsystemen Oracle, DB2 und Informix unterstützten Fähigkeiten von Interesse.

Nach diesen beiden Grundlagenkapiteln widmet sich das Kapitel 4 ausführlich der Speicherung von XML-Dokumenten. Es werden verschiedene Möglichkeiten besprochen, wie XML-Daten abgelegt werden können, und es wird ver-

sucht, zu einer systematischen Klassifizierung der einzelnen Varianten zu kommen. Besondere Aufmerksamkeit erfährt im Zusammengang mit den modellbasierten Verfahren der Bereich der Kennzeichnungsschemata.

In Kapitel 5 wird das entworfene Abbildungsverfahren vorgestellt und gezeigt, wie dies im Rahmen einer prototypischen Applikation implementiert wurde. Hier werden auch die Gründe genannt, wieso als Datenbank das Informix-System gewählt wurde. Mit Hilfe der Abbildungsanwendung wurden Untersuchungen des nötigen Zeitbedarfs und des erforderlichen Arbeits- und Datenbankspeichers durchgeführt. Die erhaltenen Ergebnisse sind in Kapitel 6 dargestellt. Die Arbeit endet mit einer Zusammenfassung sowie einem Ausblick auf mögliche Weiterentwicklungen des Verfahrens.

2 XML

2.1 Grundlagen

In diesem Kapitel soll knapp das Wichtigste zu XML zusammengefasst werden für die Leser, die die Entwicklung, die in zahlreichen Tutorien [Weg09, W3S11] und Lehrbüchern [Har04, Ray01, Eck99, Sch02] ausführlich beschrieben wird, nur bruchstückartig kennen. Eine vollständige, detaillierte Beschreibung kann im Rahmen dieser Dissertation nicht gegeben werden. Auf die entsprechenden Normen des W3C (World Wide Web Consortium) wird an den zugehörigen Stellen hingewiesen.

XML ist die Abkürzung für Extensible Markup Language und wurde vom W3C mittlerweile in Version 1.1 standardisiert [W3C06a]. XML ist eine Metasprache, die es ermöglicht, Markup-Sprachen für spezielle Einsatzbereiche zu definieren.

Markup-Sprachen (im Deutschen Auszeichnungssprachen) regeln das Einstreuen zusätzlicher Markierungen (engl. „tags“) in den Nutztext. Diese Auszeichnungen dienen der Interpretation der Daten oder der Verarbeitung, etwa als Formatierungsvorschrift. Historisch waren sie Anweisungen an den Drucker für den Satz der Texte. Im Englischen wird oft auch von einer „tagged language“ als Synonym für eine Markup Language (ML) gesprochen.

Wichtig ist, dass XML nicht mit HTML (Hypertext Markup Language) gleichgesetzt oder als eine Fortentwicklung von HTML angesehen werden kann. XML ist im Gegensatz zu HTML keine Markup-Sprache, sondern eine Metabeschreibungssprache, das heißt, eine Sprache, die die Definition anderer Markup-Sprachen, wie zum Beispiel HTML, ermöglicht.

2.1.1 Entwicklungsgeschichte und heutige Bedeutung

Die Entwicklung von XML begann Ende der 1990er-Jahre. Mehrere Organisationen und führende Unternehmen der Software-Industrie schlossen sich zusammen, um einen Standard für die Entwicklung von Markup-Sprachen zu definie-

ren. Diese Bestrebungen mündeten schließlich 1998 in der ersten W3C Recommendation zu XML.

Grundlegende Konzepte von XML wurden von der bereits 1986 standardisierten Standard Generalized Markup Language (SGML) übernommen, die deutlich umfangreicher als XML ist und somit eine Art Obermenge darstellt. Aufgrund der hohen Komplexität von SGML wurde die Entwicklung von Software zur Verarbeitung dieser Sprache erschwert. Dies hat dazu geführt, dass SGML nie breite Akzeptanz in der Praxis gefunden hat. XML hat im Gegensatz zu SGML eine strengere Syntax, die Parsern die Interpretation des Codes erleichtert.

Bankwesen	
ACORD XML	XML für Versicherungen
FIXML	Financial Information Exchange Protocol Markup Language
FPML	Financial Product Markup Language
FUNDSML	Funds Markup Language
Biowissenschaften	
AGAVE	Architecture for Genomic Annotation, Visualization and Exchange
BSML	Bioinformatic Sequence Markup Language
CML	Chemical Markup Language
Publikationen	
SportML	Sport Markup Language
NewsML	News Markup Language
XBITS	XML Book Industry Transaction Standards
XPRL	Extensible Public Relations Language
Verschiedene	
LandML	Land Development Markup Language
MODA-ML	Middleware Tools and Documents to enhance the Textile/Clothing Supply Chain through XML
MatML	Materials Property Data Markup Language
JXDM	Global Justice XML Data Model
ebXML	Electronic Business using extensible Markup Language

Tab. 2–1 *Überblick über verschiedene XML-Dialekte (aus [Bia07])*

XML wird heute in zwei großen Bereichen eingesetzt. Zum einen zur Entwicklung von Beschreibungssprachen für unterschiedlichste Einsatzzwecke. Hier ist zuallererst HTML zu nennen, das in der neueren, XML-basierten Version als XHTML bezeichnet wird und zur Gestaltung von Web-Seiten dient. Daneben sind als kleine Auswahl weiterer XML-basierter Beschreibungssprachen SVG (Scalable Vector Graphics) [W3C11] zur Beschreibung zweidimensionaler Vektorgrafiken, MathML zur Darstellung mathematischer Formeln und komplexer

Ausdrücke oder auch RSS (Really Simple Syndication) [RAB09] für die Verbreitung von Nachrichten im Internet zu nennen.

In diesen Bereich gehören auch die Schemabeschreibungssprache XML-Schema, mit deren Hilfe die Struktur von XML-Dokumenten definiert werden kann, sowie die Extensible Stylesheet Language (XSL), die zur Transformation von XML-Dokumenten eingesetzt wird.

Der zweite große Anwendungsbereich von XML ist die Festlegung von Formaten zum Datenaustausch zwischen Anwendungen in heterogenen Netzen. Beispiele sind hier die Standards SOAP (Simple Object Access Protocol) und XML-RPC, mit denen Remote Procedure Calls (RPC) durchgeführt und die zugehörigen Daten übermittelt werden können.

Neben diesen eher „klassischen“ Beispielen für den Einsatz von XML hat die Sprache mittlerweile auch in der Wirtschaft eine hohe Bedeutung erzielt. Die Tabelle 2–1, entnommen aus [Bia07], gibt einen Überblick über wirtschaftlich besonders bedeutende XML-Dialekte.

2.1.2 Dokumentstruktur

Ein XML-Dokument ist grundsätzlich ein Text mit eingestreuten Markierungen, die den sehr einfachen XML-Syntaxregeln genügen müssen. Als Codierung können diverse standardisierte Zeichensätze verwendet werden. Üblich sind der Unicode-Zeichensatz (meist in der UTF-8-Variante) und der besonders in Europa verbreitete Zeichensatz ISO-8859-1.

Sofern es sich nicht um UTF-8 handelt, muss die eingesetzte Codierung zu Beginn eines XML-Dokuments im Prolog in der so genannten XML-Deklaration angegeben werden. Im Anhang der XML-Spezifikation wird ein Verfahren beschrieben, wie die XML-Deklaration ausgewertet und die angegebene Codierung ohne zusätzliche externe Informationen erkannt werden kann. Die XML-Deklaration enthält ebenfalls die eingesetzte Version des XML-Standards (`<?xml version="1.0" encoding="ISO-8859-1"?>`).

Wichtigster Bestandteil eines XML-Dokuments sind die Elemente, die eine hierarchische Struktur bilden. Dadurch lässt sich jedes XML-Dokument als Baum wiedergeben. Dieser Baum muss aus genau einer Wurzel, dem Root-Element, bestehen. Im Dokument werden die Elemente mit Hilfe von zusammengehörigen Tag-Paaren dargestellt, die den Namen des Elements in spitzen Klammern enthalten. Jedes Tag-Paar besteht aus einem Start- und einem End-Tag (`<Element>...</Element>`). Falls ein Element keine weiteren Unterelemente und keinen Textinhalt besitzt, kann das Tag-Paar durch ein einzelnes Tag ersetzt werden (`<Element/>`). Man spricht dann von einem so genannten leeren Element.

Für den Namen eines Elements sind nicht alle Unicode-Zeichen erlaubt. So sind zum Beispiel spitze Klammern, Schrägstriche, Umlaute und eine Reihe weiterer Sonderzeichen nicht zugelassen. Für das erste Zeichen eines Elementnamens

gelten weitere Einschränkungen. Die Namen dürfen unter anderem nicht mit einer Ziffer, einem Bindestrich oder einem Punkt beginnen.

Die Elemente können beliebig tief ineinander verschachtelt werden. Hierbei ist nur zu beachten, dass die Reihenfolge der Start- und End-Tags korrekt eingehalten wird. Auch eine Begrenzung der Anzahl des Elemente pro Verschachtelungsebene existiert nicht.

Die Elemente bestimmen somit die einem XML-Dokument zugrunde liegende Struktur. Informationen können darin aufgenommen werden, indem Text zwischen die Elemente eingestreut wird. Der Text kann an beliebiger Stelle (innerhalb des Wurzelelements) angegeben werden. Das heißt, es sind sowohl Elemente erlaubt, deren Inhalt nur aus Text besteht (`<Element>Textinhalt</Element>`), als auch Elemente mit gemischtem Inhalt, die zwischen dem Text weitere Unterelemente in beliebiger Reihenfolge enthalten (`<ElementA>Dies <ElementB>ist </ElementB> der Textinhalt</ElementA>`).

Außer als Textinhalt können die Informationen eines XML-Dokuments auch mit Hilfe von Attributen angegeben werden. Jedes Attribut gehört stets zu einem Element und besteht aus den Angaben Attributname und Attributwert. Die Codierung der Attribute erfolgt im Start-Tag eines Elements (`<Element Attribut1="Wert1" Attribut2="Wert2">...</Element>`). Jedes Element kann beliebig viele Attribute besitzen, sofern die verwendeten Attributnamen innerhalb dieses Elements verschieden sind. Die Reihenfolge, in der die Attribute angegeben werden, ist irrelevant. Für die Attributnamen gelten die gleichen Regeln wie für die Namen der Elemente.

Bei der Gestaltung eines XML-Dokuments gibt es für den Einsatz von Textinhalt und Attributen keine klaren Verwendungsvorschriften. In vielen Situationen können die Informationen sowohl über Text als auch mit Hilfe von Attributen angegeben werden. Dem Einsatz von Attributen ist der Vorzug zu geben, wenn es um kurze Angaben geht und diese auf der jeweiligen Ebene maximal einmal vorkommen. Dadurch lassen sich Hierarchiestufen einsparen und das Dokument wird übersichtlicher. Sollen jedoch auf einer Ebene mehrere gleichwertige Informationen angegeben werden, so müssen diese in Form von mehreren Unterelementen in das Dokument aufgenommen werden. Die Daten werden dann meist als Textinhalt dieser Unterelemente im XML-Dokument codiert.

Damit sind die drei wichtigsten Bestandteile eines XML-Dokuments beschrieben: Elemente, Attribute und Text. Daneben existieren noch die bereits beschriebene XML-Deklaration sowie Kommentare und Verarbeitungsanweisungen.

Mit Hilfe von Kommentaren können innerhalb eines XML-Dokuments Bemerkungen angegeben werden. Diese dienen meist dazu, Zusatzinformationen für Entwickler anzugeben, die die Struktur des Dokuments oder noch zu implementierende Erweiterungen beschreiben. Kommentare bestehen aus Text, der durch eine Start- und Endmarkierung begrenzt wird (`<!-- Kommentar -->`). Der

Text ist nicht Bestandteil des eigentlichen XML-Dokuments und dem Parser steht es frei, den Inhalt des Kommentars an die Anwendung weiterzugeben oder nicht. Kommentare können an jeder Stelle des Dokuments erscheinen, nur nicht innerhalb der Start- und End-Tags der Elemente.

Verarbeitungsanweisungen (engl. „processing instructions“) ermöglichen es, Steueranweisungen für externe Anwendungen innerhalb eines XML-Dokuments abzulegen. Sie bestehen aus dem Namen der Anwendung, an die die Anweisung gerichtet ist („Target“) und einem Datenteil, der weitere Informationen enthält (`<?Name Daten?>`). Obwohl ihre Syntax der XML-Deklaration ähnelt, sind sie von dieser zu unterscheiden.

Grundsätzlich ist fragwürdig, inwiefern die Verwendung von Verarbeitungsanweisungen sinnvoll ist, da diese die von XML propagierte Trennung von Inhalt und Darstellung bzw. Verarbeitung aufweichen. Es gibt jedoch Anwendungsfälle, in denen ihr Einsatz zweckmäßig ist. Hierzu gehört beispielsweise die Angabe eines XSL-Stylesheets, mit dem ein XML-Dokument in HTML transformiert werden soll, um es in einem Web-Browser darstellen zu können.

Im Jahr 1999 wurde vom W3C eine Spezifikation veröffentlicht, die den XML-Standard um Namensräume (engl. „namespaces“) erweitert [W3C06b]. Wenn man in einem XML-Dokument verschiedene XML-Vokabulare verwendet, kann es zu Konflikten kommen, da die Element- und Attributnamen nicht mehr eindeutig sind. Namensräume beseitigen diese Probleme, indem sie eindeutige Element- und Attributnamen generieren.

Dazu wird jedem XML-Vokabular ein Namensraum zugeordnet, der über eine URI identifiziert wird. Um Elemente aus diesem Namensraum zu verwenden, muss in dem jeweiligen XML-Dokument ein Präfix definiert und mit der zugehörigen URI verknüpft werden. Das Präfix muss allen Elementen, durch einen Doppelpunkt abgetrennt, vorangestellt werden. Zusätzlich kann auch ein Standard-Namensraum festgelegt werden, in den alle Elemente fallen, bei denen kein Präfix angegeben ist.

Die in Abschnitt 2.2.1 besprochenen DTDs, die der Festlegung der Struktur von XML-Dokumenten dienen, können nur sehr eingeschränkt zusammen mit Namensräumen eingesetzt werden. Die Ursache liegt darin, dass an der DTD-Syntax nach Einführung der Namensräume keine Änderungen vorgenommen wurden. Erst XML-Schema (siehe Abschnitt 2.2.2), eine Weiterentwicklung der DTD, bietet vollständige Unterstützung für den Einsatz von Namensräumen.

Wie schlecht entworfene relationale Datenbanken können auch XML-Dokumente redundante Informationen enthalten. Die Datenbanktheorie hat verschiedene Normalformen entwickelt, um dem entgegenzuwirken. [AL02], [VLL04], [Sch05] und [Are06] zeigen Möglichkeiten auf, wie sich die Begriffe Schlüsseleigenschaft und funktionale Abhängigkeit auf XML-Dokumente übertragen und sich daraus Normalformen für XML-Dokumente entwerfen lassen.

2.1.3 Klassifikation

XML-Dokumente lassen sich nach ihrer Struktur und den enthaltenen Daten klassifizieren. Man unterscheidet dokumentenzentrierte[2] und datenzentrierte Dokumente. Datenzentrierte Dokumente haben eine regelmäßige Struktur, in der die Reihenfolge der einzelnen Elemente meist keine Bedeutung hat. Beispiele für diesen Typ sind Preislisten, Bestellungen oder Rechnungen.

Dokumente, bei denen jedoch die Reihenfolge der einzelnen Elemente von entscheidender Bedeutung ist und die sich aus längeren Textpassagen mit eingestreuten Elementen, also aus gemischtem Inhalt, zusammensetzen, werden dokumentenzentriert genannt. Hierzu zählen unter anderem Bücher und die meisten XHTML-Seiten.

Häufig sind datenzentrierte Dokumente für die Verarbeitung durch Maschinen vorgesehen, wohingegen sich dokumentenzentrierte Dokumente eher an Menschen richten, die deren Inhalt lesen. In vielen Fällen lässt sich allerdings keine klare Einteilung vornehmen, da die Dokumente Bestandteile von beiden Kategorien enthalten. Diese Mischformen werden als semistrukturiert bezeichnet. Hierunter fällt zum Beispiel ein Brief, der einerseits durch Anschrift, Absender, Betreff und Datum datenzentriert ist, andererseits jedoch durch den Brieftext, der in Absätze unterteilt ist und Hervorhebungen enthalten kann, auch dokumentenzentrierte Eigenschaften besitzt. Basierend auf den Angaben aus [BR03] gibt Tabelle 2–2 einige Merkmale von dokumenten- und datenzentrierten Dokumenten an.

		dokumentenzentriert	**datenzentriert**
Daten	**Struktur**	unregelmäßig	regelmäßig
	Schema	kann fehlen	vordefiniert
	Inhalt	Elemente mit gemischtem Inhalt	Elemente mit reinem Textinhalt und Attribute
	Element-reihenfolge	wichtig	frei
Operationen	**Navigation**	hierarchisch und sequenziell	Anfragen mit Sortierung, Aggregation und Verbundanweisungen
	Suche	Volltext in Elementinhalten	Elementtextinhalte und Attributwerte
	Suchergebnis	Extraktion von Dokumenten und Dokumentfragmenten	Erstellung von Ergebnisdokumenten aus Anfragedaten durch Restrukturierung
	Änderungen	Dokumentfragmente	Elementtextinhalte und Attributwerte

Tab. 2–2 *Klassifikation von XML-Dokumenten*

2. An Stelle des Begriffs „dokumentenzentriert“ wird häufig auch „textzentriert“ oder „textorientiert“ verwendet. Statt „datenzentriert“ heißt es entsprechend auch „datenorientiert“.

2.2 Dokumentschemata

Ein XML-Dokument, das die im vorangegangen Abschnitt beschriebene grundsätzliche Struktur einhält, bezeichnet man als wohlgeformt (engl. „well-formed"). Wohlgeformtheit ist die Voraussetzung, dass ein Dokument von einem XML-Parser erfolgreich eingelesen und verarbeitet werden kann. Verstöße gegen die Regeln der Wohlgeformtheit müssen vom Parser an die Anwendung gemeldet werden. Es darf nicht versucht werden, die Fehler zu reparieren oder zu ignorieren, da die komplette Struktur des XML-Dokuments nicht mehr eindeutig zu interpretieren ist.

Neben der Wohlgeformtheit eines Dokuments existiert noch der Begriff der Gültigkeit eines Dokuments. Jedes gültige (engl. „valid") Dokument ist wohlgeformt, aber nicht notwendigerweise umgekehrt. Gültigkeit ist also im Vergleich zu Wohlgeformtheit die stärkere Bedingung. Ein gültiges Dokument muss wohlgeformt sein, einen Verweis auf eine Grammatik enthalten und auch dieser Grammatik genügen.

Zur Angabe einer Grammatik wird eine Schemadefinitionssprache benötigt. Dazu wird meist eine DTD (Dokumenttypdefinition, engl. „document type definition") oder ein XML-Schema verwendet. Die Grammatik legt die für die jeweilige Anwendung zu erfüllende Syntax eines Dokuments fest. In der Grammatik können die erlaubten Elemente, deren Reihenfolge, Häufigkeit und Schachtelung sowie die möglichen Attribute und die Datentypen der Texte und Attributewerte festgelegt werden.

2.2.1 DTD

Die Festlegung der Syntax und Semantik einer DTD ist Teil des XML-Standards. Aus diesem Grund wird die Verarbeitung von DTDs von fast allen XML-Parsern unterstützt. Dies hat dazu geführt, dass DTDs gegenüber anderen Schemadefinitionssprachen häufig der Vorzug gegeben wird.

Die DTD-Syntax, die in XML-Dokumenten verwendet wird, hat ihren Ursprung im SGML-Standard und ist der dort beschriebenen SGML-DTD-Syntax sehr ähnlich. Dies hat zur Folge, dass die Syntax der DTD-Anweisungen nicht XML-konform ist. Dies ist ein großer Nachteil der DTDs.

Die Anweisungen einer DTD erlauben es, einen Satz von Regeln anzugeben, der die Struktur einer bestimmten Klasse von XML-Dokumenten beschreibt. Für die Verknüpfung des XML-Dokuments mit diesen Angaben existieren zwei Möglichkeiten. Entweder wird die DTD zu Beginn des XML-Dokuments in dieses direkt eingebettet oder aber die DTD wird in eine separate Datei ausgelagert, sodass man diese über die Angabe des Dateinamens referenzieren kann. Dies vermeidet Redundanz, da sich mehrere XML-Dokumente auf die gleiche Schemadefinition beziehen können.

Die Angabe der DTD bzw. die Referenz auf die separate Datei bezeichnet man als Dokumenttypdeklaration. Sie muss innerhalb des XML-Dokuments vor dem Wurzelelement stehen. Die Syntax lautet

```
<!DOCTYPE Wurzelelement [ ... ]>
```

bei direkter Angabe der DTD (innerhalb der eckigen Klammern) und

```
<!DOCTYPE Wurzelelement SYSTEM "Dateiname">
```

falls die DTD nicht Bestandteil des XML-Dokuments ist.

Die in einer DTD erlaubten Anweisungen bezeichnet man als Markup-Deklarationen. Die wichtigsten beiden sind die Elementtyp- und die Attributlistendeklarationen.

Eine Elementtypdeklarion (`<!ELEMENT Name Definition>`) dient der Definition eines erlaubten Elements und des darin zugelassenen Inhalts. Der Inhalt kann dahingehend eingeschränkt werden, dass nur bestimmte Unterelemente auftreten dürfen oder die Reihenfolge und Häufigkeit der Unterelemente festgelegt wird. Bei der Angabe der Häufigkeit des Auftretens eines Unterelements ist es nicht möglich, eine bestimmte Anzahl oder eine bestimmte Mindest- und Maximalmenge anzugeben. Stattdessen kann entweder die Häufigkeit beliebig sein oder aber man fordert das genau einmalige, das mindestens einmalige oder das maximal einmalige Auftreten eines Elements.

Auch kann eine Menge von Unterelementen angegeben werden, aus denen nur eines ausgewählt werden darf. Daneben bestehen die Möglichkeiten, ein beliebiges in der DTD definiertes Element als Unterelement zuzulassen oder zu fordern, dass ein Element leer sein muss.

Falls neben Unterelementen auch Text als Inhalt eines Elements zugelassen werden soll, so steht für diesen nur der allgemeine Datentyp `PCDATA` (engl. „parsed character data“) zur Verfügung. Eine Einschränkung des Textinhalts auf bestimmte Datentypen, wie zum Beispiel Ganz- oder Dezimalzahlen, ist nicht möglich. Ebensowenig kann der Text auf bestimmte Zeichenketten begrenzt werden. Sobald in einem Element Text als Inhalt erlaubt wird, kann nicht mehr die Reihenfolge und Häufigkeit von ebenfalls zugelassenen Unterelementen vorgegeben werden. Man spricht von einem Element mit gemischtem Inhalt (engl. „mixed content“), falls sowohl Unterelemente als auch Text in diesem auftreten.

Die Attribute eines Elements werden mit Hilfe der Attributlistendeklaration (`<!ATTLIST Name1 Typ1 DefaultDecl1 Name2 Typ2 DefaultDecl2 ...>`) definiert. Für jedes Attribut kann der Name, der Typ des Attributwerts sowie eine so genannte Default-Declaration angegeben werden. Die Default-Declaration legt fest, ob das Attribut im XML-Dokument weggelassen werden darf und falls ja, ob es dann mit einem Vorgabewert belegt wird.

Meist verwendeter Datentyp für die Attributwerte ist `CDATA` (engl. „character data“). Daneben existieren die Typen `ID` und `IDREF`, die es gestatten, eine Art

Fremdschlüsselbeziehung zu definieren. Die Werte *aller* Attribute vom Typ `ID` müssen über das komplette XML-Dokument eindeutig sein. Attribute vom Typ `IDREF` müssen einen Wert besitzen, der mit dem Wert eines `ID`-Attributs übereinstimmt. Somit übernehmen `ID`-Attribute eine Rolle ähnlich den Schlüsseln in Relationen. Diese Funktion wird aber leider dadurch eingeschränkt, dass die Werte den Regeln für die Bildung von Element- und Attributnamen (siehe Abschnitt 2.1.2) genügen müssen, wodurch rein numerische Werte, etwa eine sich als Schlüssel anbietende Kundennummer oder bei Büchern eine ISBN, nicht zugelassen sind. Begrenzt werden die Referenzbeziehungen zusätzlich dadurch, dass keine Paare von `ID`- und `IDREF`-Attributen gebildet werden können, die sich aufeinander beziehen müssen.

Der Vollständigkeit halber seien noch die Attributwerttypen `ENTITY` und `NMTOKEN` erwähnt. Ein Attribut vom Typ `ENTITY` ermöglicht den Verweis auf externe Ressourcen, wie zum Beispiel einzubindende Bilddateien. Attribute, die den Typ `NMTOKEN` (engl. „name token“) besitzen, schränken die zugelassenen Werte auf Zeichenketten ein, deren Zeichen nur aus einer bestimmten Teilmenge des Unicode-Zeichensatzes stammen dürfen. Die zugelassenen Zeichen entsprechen den in Element- und Attributnamen möglichen Zeichen, nur dass für das erste Zeichen keine zusätzlichen Einschränkungen gelten.

Zu den beschriebenen Typen `IDREF`, `ENTITY` und `NMTOKEN` existieren die korrespondierenden „Plural-Typen“ `IDREFS`, `ENTITIES` und `NMTOKENS`. Diese erlauben die Aufzählung mehrerer, durch Leerzeichen getrennter Werte des jeweiligen Typs.

Neben den bisher erwähnten Markup-Deklarationen für Elemente und Attribute können in einer DTD auch Entitäts- und Notationsdeklarationen verwendet werden. Entitätsdeklarationen dienen der Definition von Bezeichnern für bestimmte Zeichenfolgen. Die Bezeichner können dann je nach Definition als Abkürzung in XML-Dokumenten oder in DTDs angegeben werden. Der Parser ersetzt sie durch die in der Entitätsdeklaration festgelegte Zeichenfolge. Es ist auch möglich, die Zeichenfolge in eine externe Datei auszulagern und in der Entitätsdeklaration den Dateinamen als Referenz anzugeben.

Der Zweck von Notationsdeklarationen ist es, Formate für externe Daten festzulegen, die nicht vom XML-Parser selbst verarbeitet werden. Die definierten Notationen dienen dann in Entitätsdeklarationen zur Benennung des Formats einer externen Datei.

2.2.2 XML-Schema

XML-Schema ist eine im Mai 2001 vom W3C standardisierte Schemadefinitionssprache [W3C04d, W3C04e]. Sie dient wie eine DTD zur Festlegung einer Dokumentstruktur, die XML-Dokumente erfüllen müssen, um bezüglich dieses Schemas gültig zu sein. Im Gegensatz zu DTDs, deren Syntax und Semantik im XML-

Standard festgelegt wird, wird XML-Schema in einer eigenständigen Spezifikation definiert.

Eine Instanz eines XML-Schemas bezeichnet man als XML-Schema-Definition (XSD). Auffälligster Unterschied zu einer DTD ist, dass eine XML-Schema-Definition selbst wieder ein gültiges XML-Dokument ist. Darüberhinaus erlaubt ein XML-Schema im Vergleich zu einer DTD deutlich komplexere Festlegungen der gewünschten XML-Dokumentstruktur.

Elemente und Attribute lassen sich mit Hilfe der XML-Schema-Elemente `element` und `attribute` definieren. Ihnen wird ein benannter oder unbenannter Typ zugeordnet. Der Typ legt bei Elementen deren Inhalt (Unterelemente und Text), bei Attributen den Datentyp des Attributwerts fest.

Man unterscheidet zwischen so genannten einfachen und komplexen Typen. Einfache Typen werden mit `simpleType` definiert. Sie besitzen weder Unterelemente noch Attribute. Komplexe Typen (`complexType`) kann man nochmals unterteilen, je nachdem ob sie einfachen oder komplexen Inhalt besitzen. Ein komplexer Typ mit einfachem Inhalt (`simpleContent`) hat keine Unterelemente, kann aber selbst Attribute haben. Komplexe Typen mit komplexem Inhalt (`complexContent`) erlauben Unterelemente und gegebenenfalls auch Attribute.

Die Grundlage des Typsystems von XML-Schema wird durch eine Vielzahl von Basistypen gebildet. Die Typhierarchie ist in Abbildung 2–1 dargestellt. Man erkennt, dass neben den aus DTDs bekannten Typen eine Reihe weiterer spezieller Typen vorhanden sind. Dadurch lassen sich Elemente und Attribute definieren, deren Inhalt beispielsweise ein Wahrheitswert, eine Datumsangabe, eine Dezimalzahl oder eine positive ganze Zahl sein muss.

Diese Basistypen können über Ableitungen in ihren Wertebereichen weiter spezialisiert werden. So kann unter anderem durch so genannte Facetten bei Zeichenketten die minimale und maximale Länge angegeben oder bei Ganzzahltypen der kleinste und größte zugelassene Wert festgelegt werden. Daneben ist es auch möglich, die erlaubten Werte explizit aufzuzählen oder einen regulären Ausdruck anzugeben.

Auch bei der Angabe der Elementverschachtelung bietet XML-Schema gegenüber einer DTD eine Reihe von Erweiterungen. So ist es möglich, die Reihenfolge und Anzahl der Unterelemente vorzugeben (`sequence`). Bei der Anzahl kann das minimale und maximale Auftreten exakt festgelegt werden. Mittels `choice` lässt sich eine Liste von Elementen aufführen, aus denen ein Element ausgewählt werden muss. Die Verwendung von `all` erlaubt die Definition einer Gruppe von Elementen, die alle jeweils genau einmal als Unterelement vorkommen müssen, allerdings ist die Reihenfolge beliebig.

Bei Elementen mit gemischtem Inhalt kann die Reihenfolge der eingestreuten Unterelemente und die Anzahl ihres Auftretens genau festgelegt werden. Bei einer DTD ist keine derartige Vorgabe möglich. Mit Hilfe der XML-Schema-Elemente

key und keyref lassen sich Schlüsseleigenschaften und Fremdschlüsselbeziehungen modellieren.

Alle Typdefinitionsmöglichkeiten können im Rahmen von Erweiterungen (engl. „extensions“) und Einschränkungen (engl. „restrictions“) miteinander kombiniert werden, wodurch sich ein komplexes Typsystem ergibt, das weit mehr Typvarianten als eine DTD bietet.

Abb. 2–1 *Hierarchie der eingebauten Datentypen von XML-Schema (nach [W3C04e])*

2.3 XPath-Datenmodell

An dieser Stelle soll das Datenmodell von XPath kurz vorgestellt werden, da es eine Sichtweise auf XML-Dokumente bietet, die auch in den nachfolgenden Kapiteln verwendet wird. Eine vollständige Darstellung von XPath und der zugehörigen Syntax der Pfadausdrücke findet sich unter anderem in [W3C10b].

XPath ist eine vom W3C mittlerweile in Version 2.0 standardisierte Anfragesprache zur Adressierung von Teilen eines XML-Dokuments. Die Sprache dient

als Grundlage einer Reihe weiterer W3C-Standards, wie zum Beispiel XQuery [LS04, EM05, W3C10c, Muj11], XSLT [W3C07] und XLink [W3C10a].

Das Datenmodell von XPath formalisiert die bereits in Abschnitt 2.1.2 angesprochene Darstellung eines XML-Dokuments als geordneten Baum [W3C10d]. XPath stellt das komplette Dokument als Baumstruktur dar. Dabei können die folgenden sieben Knotentypen auftreten:

- Wurzelknoten
- Elementknoten
- Textknoten
- Attributknoten
- Namensraumknoten
- Verarbeitungsanweisungsknoten
- Kommentarknoten

Der Wurzelknoten und die Elementknoten haben eine Liste[3] von Kinderknoten. Jeder Knoten außer dem Wurzelknoten hat genau einen Elternknoten. Jeder Elternknoten ist entweder Elementknoten oder der Wurzelknoten.

Wurzelknoten

Der Wurzelknoten kommt in der Darstellung eines XML-Dokuments als Baumstruktur genau einmal vor. Er stellt die Wurzel dieses Baums dar. Kinder des Wurzelknotens sind genau ein Elementknoten, der das Wurzelelement des XML-Dokuments repräsentiert, sowie Verarbeitungsanweisungs- und Kommentarknoten für die Verarbeitungsanweisungen und Kommentare, die im XML-Dokument außerhalb des XML-Wurzelelements erscheinen. Der Wurzelknoten wird häufig auch als Dokumentknoten bezeichnet.

Elementknoten

Jedes Element des XML-Dokuments wird durch einen Elementknoten dargestellt. Kinder eines Elementknotens können Element-, Kommentar-, Verarbeitungsanweisungs- und Textknoten sein.

Textknoten

Textknoten dienen der Aufnahme des Textinhalts der Elemente. Dabei soll jeder Textknoten den maximal möglichen Textinhalt aufnehmen, sodass zwei Textknoten, die Kinder desselben Elternknotens sind, niemals direkt aufeinander folgen. Textknoten besitzen keine Kinderknoten.

3. Unter einer Liste soll hier und im Folgenden stets eine *geordnete* Aufzählung von Elementen verstanden werden, unter einer Menge eine *ungeordnete* Zusammenfassung von Elementen.

Attributknoten

Jedes Attribut eines Elements wird durch einen Attributknoten modelliert. Diese Attributknoten bekommen den Elementknoten als Elternknoten zugeordnet. Attribute, die in der DTD oder dem XML-Schema deklariert sind, im XML-Dokument jedoch nicht angegeben sind, werden nicht durch einen Attributknoten repräsentiert.

Namensraumknoten

Jeder Namensraum, der im Bereich eines Elements deklariert ist, wird durch einen Namensraumknoten dargestellt, dessen Elternknoten der jeweilige Elementknoten ist.

Verarbeitungsanweisungsknoten

Dieser Knotentyp wird zur Modellierung von Verarbeitungsanweisungen verwendet. Da die XML-Deklaration – trotz ihrer Ähnlichkeit – selbst keine Verarbeitungsanweisung ist, wird dieser auch kein Verarbeitungsanweisungsknoten zugeordnet.

Kommentarknoten

Kommentarknoten dienen der Repräsentation der im XML-Dokument enthaltenen Kommentare.

Das XPath-Datenmodell definiert eine Reihenfolge unter den Knoten eines XML-Dokuments, die so genannte Dokumentenordnung: Die Knoten werden entsprechend des Auftretens des ersten Zeichens ihrer jeweiligen XML-Darstellung – bei Elementen also das Start-Tag – im XML-Dokument angeordnet. Somit liegen Elementknoten vor den zugehörigen Unterelementknoten. Namensraumknoten erscheinen vor Attributknoten, die wiederum vor den Unterelementknoten angeordnet werden. Innerhalb der Namensraum- und Attributknoten wird keine verbindliche Reihenfolge festgelegt. Die Dokumentenordnung entspricht der Knotenabfolge, die man erhält, wenn man die Knoten des XML-Dokuments in der Baumdarstellung in Präorder-Reihenfolge[4] durchläuft.

Zentraler Bestandteil von XPath sind die so genannten Lokalisierungspfade. Sie bestehen aus einzelnen, durch einen Schrägstrich (/) voneinander getrennten Teilen. Jeder Teil bildet einen Lokalisierungsschritt, der eine Knotenmenge selektiert. Der erste Lokalisierungsschritt führt die Selektion bezogen auf einen Kontextknoten durch. Für jeden weiteren Schritt ist jeder in der zuvor ausgewählten Knotenmenge enthaltene Knoten Kontextknoten für die Selektion. Die durch den

4. Bei der Präorder-Reihenfolge (auch Vorordnung oder Hauptreihenfolge genannt) wird zunächst die Wurzel betrachtet, danach werden die geordneten Teilbäume durchlaufen.

letzten Lokalisierungsschritt selektierte Knotenmenge ist das Ergebnis des vollständigen Lokalisierungspfades.

Jeder Lokalisierungsschritt kann sich aus einer Achse, einem Knotentest und einem Prädikat zusammensetzen. Die Syntax lautet:

```
Achse::Knotentest[Prädikat]
```

Die Achse gibt die gewünschte Beziehung an, in der die zu selektierenden Knoten zum Kontextknoten stehen sollen. XPath kennt zwölf Achsen, deren Definition sich an der Baumdarstellung eines XML-Dokuments orientiert. Es existieren die folgenden Achsen:

- `child`
  Diese Achse enthält alle Kinder, also die direkten Nachfahren, des Kontextknotens.
- `descendant`
  Alle direkten und indirekten Nachfahren des Kontextknotens gehören zur `descendant`-Achse. Also die Kinder selbst, deren Kinder wiederum usw. Attribut- und Namensraumknoten gehören nicht zu dieser Achse.
- `parent`
  Sofern ein Elternknoten des Kontextknotens existiert, so wird dieser durch die `parent`-Achse ausgewählt.
- `ancestor`
  Die `ancestor`-Achse besteht aus allen Knoten auf dem Pfad vom Kontextknoten zur Wurzel des Baums. Der Kontextknoten selbst ist nicht enthalten. Diese Achse enthält stets den Wurzelknoten, es sei denn der Kontextknoten ist selbst der Wurzelknoten.
- `following-sibling`
  Die Achse wählt alle in der Dokumentenordnung nach dem Kontextknoten stehenden Kinder des Elternknotens des Kontextknotens, also die nachfolgenden Geschwister, aus. Auf Attribut- oder Namensraumknoten angewandt ist diese Achse leer.
- `preceding-sibling`
  Die `preceding-sibling`-Achse entspricht der `following-sibling`-Achse bis auf die Tatsache, dass hier statt der nachfolgenden, die vorangehenden Geschwister selektiert werden.
- `following`
  Alle in Dokumentenordnung nach dem Kontextknoten folgenden Knoten gehören zu dieser Achse. Die Nachfahren des Kontextknotens sowie Attribut- und Namensraumknoten sind nicht enthalten.
- `preceding`
  Diese Achse ist das Gegenstück zur `following`-Achse. Hier werden die vorangehenden Knoten des Kontextknotens in Dokumentenordnung selektiert.

- `attribute`
  Die Attribute des Kontextknotens gehören zu der `attribute`-Achse. Falls der Kontextknoten keine Attribute hat oder kein Elementknoten ist, ist diese Achse leer.
- `namespace`
  Sofern der Kontextknoten ein Elementknoten ist und einen Namensraumknoten besitzt, so wird dieser durch die `namespace`-Achse selektiert.
- `self`
  Die `self`-Achse enthält stets den Kontextknoten selbst.
- `descendant-or-self`
  Diese Achse entspricht der `descendant`-Achse, wobei zusätzlich der Kontextknoten selektiert wird.
- `ancestor-or-self`
  Die `ancestor`-Achse, erweitert um den Kontextknoten, ergibt die durch die `ancestor-or-self`-Achse ausgewählte Knotenmenge. Somit enthält diese Achse immer den Wurzelknoten.

Abbildung 2–2 stellt die Achsen in einem Diagramm dar. Wenn man alle Knotentypen außer den Elementknoten unbetrachtet lässt, so stellt man fest, dass die `ancestor`-, `descendant`-, `following`-, `preceding`- und `self`-Achsen zusammen den vollständigen Baum selektieren. Die Achsen überlappen sich dabei nicht, bilden also eine vollständige Partitionierung. Diese Eigenschaft ist unabhängig vom Kontextknoten, auf den man sich bezieht.

Für einige Achsenbezeichner gibt es Abkürzungen: `..` steht für die `parent`-Achse, `//` für die `descendant`-Achse, `.` für die `self`-Achse und `@` für die `attribute`-Achse. Wird keine Achse angegeben, so wird die `child`-Achse verwendet.

Der zweite Bestandteil des Lokalisierungsschritts ist der Knotentest. Mit ihm lassen sich die durch die Achse selektierten Knoten weiter einschränken, sodass nur die Element- oder Attributknoten erhalten bleiben, die den angegebenen Namen besitzen. Mit dem Prädikat, das den dritten Bestandteil des Lokalisierungsschritts bildet, können weitere Bedingungen formuliert werden, die der Knoten erfüllen muss, um in die Ergebnisknotenmenge aufgenommen zu werden. Für die Formulierung des Prädikats steht eine Reihe von Funktionen zur Verfügung, mit denen sich Knotenmengen, Zeichenketten und Zahlen zu komplexen Ausdrücken verbinden lassen. Auf die genaue Darstellung wird an dieser Stelle verzichtet und erneut auf [W3C10b] verwiesen.

Häufig wird im Zusammenhang mit der XML-Baumdarstellung auch von einem DOM-Baum gesprochen. Hauptziel des Document Object Models (DOM) ist die Definition eines Standards zur plattform- und sprachenunabhängigen Darstellung und Bearbeitung von XML- und HTML-Dokumenten [W3C04a]. Dazu legt DOM eine Menge von Klassen und zugehörigen Methoden und Attributen fest, um auf die einzelnen Dokumentbestandteile zuzugreifen, sie zu ändern und

zwischen ihnen zu navigieren. Zur Repräsentation eines XML-Dokuments verwendet DOM ebenfalls eine Baumstruktur. Diese ist im Wesentlichen mit der oben beschriebenen XPath-Baumdarstellung identisch. Es gibt in der Definition jedoch einige formale Unterschiede, auf deren Darstellung hier verzichtet werden soll. In dieser Arbeit wird das XPath-Datenmodell verwendet.

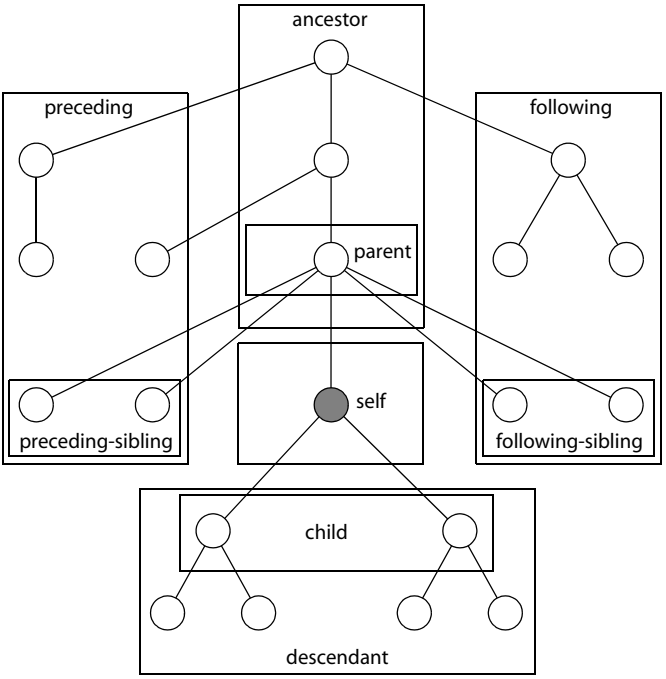


Abb. 2–2 *Darstellung der XPath-Achsen*

Die mehrfache Definition der Baumstruktur in verschiedenen W3C-Standards hat historische Ursachen. DOM wurde zunächst in Web-Browsern im Rahmen der Skript-Sprache JavaScript verwendet, um dynamisch HTML-Seiten zu manipulieren. In Verbindung mit XML wurde DOM erst später eingesetzt. Anstatt auf die bereits vorliegende Festlegung im Document Object Model zu verweisen, wurde leider in den XPath-Standard erneut die Definition einer Baumdarstellung aufgenommen. Die beste Lösung wäre vermutlich die Abtrennung und Auslagerung der Baumstrukturdefinition in den XML-Information-Set-Standard [W3C04b] gewesen, in dem grundlegende Begrifflichkeiten im Zusammenhang mit XML-Dokumenten festgelegt werden.

3 Objektrelationale Datenbanken und SQL:2003

3.1 Entwicklung des SQL-Standards

SQL (Structured Query Language) ist eine Abfragesprache für relationale Datenbankmanagementsysteme. Die Anfänge von SQL sind eng mit dem von E. F. Codd entwickelten relationalen Datenmodell verknüpft, das das theoretische Fundament relationaler Datenbanken bildet. In 1970 veröffentlichte Codd während seiner Tätigkeit für IBM seine einflussreiche Ausarbeitung mit dem Titel „A Relational Model of Data for Large Shared Data Banks“ [Cod70]. Diese Arbeit legte den Grundstein für die heute allseits akzeptierten relationalen Datenbanken.

Aus den theoretischen Überlegungen entstand in den 1970er Jahren in IBMs Forschungszentrum in San José ein Datenbankmanagementsystem mit der Bezeichnung „System R“. Dieses System entsprach nicht vollständig Codds Ideen, insbesondere wurde an Stelle der von Codd verwendeten Sprache „Alpha“ die Sprache SEQUEL verwendet. SEQUEL ist die Abkürzung für Structured English Query Language und wurde von den IBM-Mitarbeitern Donald D. Chamberlin und Raymond F. Boyce entwickelt. Später wurde das Akronym SEQUEL durch SQL ersetzt, da es zu Problemen mit einem gleichnamigen Warenzeichen einer englischen Flugzeugfirma kam.

System R selbst wurde nie ein Produkt für Kunden, obwohl es recht vollständig und ausgereift war. Die Erkenntnisse daraus flossen aber in das System SQL/DS (ab 1981) ein, das der Vorgänger der heutigen IBM DB2 Datenbankprodukte ist. Die erste kommerziell verfügbare Implementierung von SQL wurde 1979 von der amerikanischen Firma Relational Software auf den Markt gebracht. Das Produkt hatte den Namen „Oracle V2“. Der Zusatz „V2“ hatte marketingtechnische Gründe, eigentlich war es die erste Version der Software. Die Kunden sollten nur keine Bedenken wegen des Kaufs einer Version 1 bekommen. Der große Erfolg dieses DBMS führte schließlich 1983 dazu, dass sich das Unternehmen in „Oracle“ umbenannte.

Die Standardisierung von SQL erfolgte 1986 durch das American National Standards Institute (ANSI) und im Anschluss daran 1987 durch die International Standards Organization (ISO) [ISO87]. Aufgrund des Jahres der Veröffentlichung wird dieser erste SQL-Standard auch als SQL-86 bzw. SQL-87 bezeichnet. Die Standardisierung, deren einzelne Schritte in Tabelle 3–1 aufgelistet sind, unterstützte nachhaltig den kommerziellen Erfolg von SQL.

Jahr	Bezeichnungen
1986	SQL-86 (SQL-87)
1989	SQL-89
1992	SQL-92 (SQL2)
1999	SQL:1999 (SQL3)
2003	SQL:2003
2006	SQL:2006 (nur Teil 14: SQL/XML)
2008	SQL:2008
2011	SQL:2011

Tab. 3–1 *Entwicklungsstufen des SQL-Standards*

Eine kleinere Überarbeitung [ISO89] gab es 1989 (SQL-89) sowie eine größere Revision im Jahr 1992 [ISO92]. Diese SQL-Version trägt den Namen SQL-92 oder auch SQL2. SQL-92 führte unter anderem neue Datentypen, die Unterstützung für zusätzliche Zeichensätze, weitere skalare Funktionen und Mengenoperationen ein. Neben einer Vielzahl weiterer neuer Fähigkeiten ist insbesondere die Erweiterung der Möglichkeiten zur Schemadefinition und -manipulation zu nennen. So ist nun erstmals die Änderung der Schemata mittels `ALTER` sowie das Löschen mittels `DROP` standardisiert.

Nächster Entwicklungsschritt der SQL-Sprache war die 1999 veröffentlichte Version SQL:1999, die inoffiziell auch als SQL3 bezeichnet wird. Zu den bedeutensten neuen Fähigkeiten zählt die Erweiterung der Menge der Basisdatentypen (`BOOLEAN`, `BLOB`, `CLOB`), die Standardisierung von Triggern und die Einführung von Anweisungen zur Steuerung des Programmablaufs. Außerdem wurde SQL um objektrelationale Fähigkeiten erweitert, indem die Verwendung von nicht-skalaren Typen und der Einsatz von objektorientierten Funktionen eingeführt wurde. SQL:1999 brachte auch einen neuen Aufbau des Standards. Statt der bisherigen ebenenorientierten Struktur wurde eine Aufspaltung in acht separate Teilstandards vorgenommen [ISO99]. Eine gründliche Einführung in den Sprachkern von SQL:1999 geben W. Panny und A. Taudes in [PT00]. Auch sei hier auf [Mel02] hingewiesen, in dem J. Melton, der für Oracle arbeitet und aktiv an der Entwicklung des SQL-Standards mitgewirkt hat, fortgeschrittene SQL-Konstrukte behandelt.

Vier Jahre später wurde eine erneute Überarbeitung und Erweiterung des SQL-Standards fertiggestellt [ISO03]. Er trägt die Bezeichnung SQL:2003. Wichtigste Neuerungen sind dabei neben kleineren Änderungen am bereits vorhandenen Funktionsumfang die eingeführten XML-bezogenen Funktionen sowie die Unterstützung der Programmiersprache Java. Des Weiteren wurde die Verwendung von Sequenzen und Spalten mit Identitäts- und automatisch generierten Werten standardisiert sowie der Multimengentyp und tabellenwertige Funktionen eingeführt. Erwähnenswert ist auch die komfortable, neue Variante des Kommandos `CREATE TABLE`, mit der eine Tabellendefinition mittels einer SQL-Abfrage vorgenommen werden kann. [Pis03] und [MKF03] geben einen guten Überblick über die Neuerungen.

Der Aufbau des Standards wurde wiederum geändert. Einige Teile wurden gestrichen, andere wurden geteilt oder sind neu hinzugekommen. Als Ergebnis entstand schließlich eine Struktur, die aus neun Teilen besteht, die allerdings nicht fortlaufend nummeriert sind. Die Titel der einzelnen Teilstandards von SQL:1999 und SQL:2003 sind in Tabelle 3–2 aufgeführt. Einige Teilstandards erreichten nicht das Ende des Entwicklungsprozesses oder wurden nach vorübergehender Auslagerung wieder in andere Teile integriert.

Teil	Titel	SQL:1999	SQL:2003
1	Framework (SQL/Framework)	X	X
2	Foundation (SQL/Foundation)	X	X
3	Call Level Interface (SQL/CLI)	X	X
4	Persistent Stored Modules (SQL/PSM)	X	X
5	Host Language Bindings (SQL/Bindings)	X	
6	XA Specialization (SQL/Transaction)		
7	Temporal (SQL/Temporal)		
8	Extended Objects (SQL/Objects)		
9	Management of External Data (SQL/MED)	X	X
10	Object Language Bindings (SQL/OLB)	X	X
11	Information and Definition Schemas (SQL/Schemata)		X
12	Replication (SQL/Replication)		
13	SQL Routines and Types Using the Java Programming Language (SQL/JRT)	X	X
14	XML-Related Specifications (SQL/XML)		X

Tab. 3–2 *Aufbau der Standards SQL:1999 und SQL:2003*

Im Jahr 2006 wurde SQL:2006 verabschiedet. Hierbei handelt es sich nicht um eine vollständige Revision von SQL:2003, sondern nur um eine Überarbeitung des Teils 14, der sich mit SQL/XML beschäftigt [ISO06]. Dieser wurde unter anderem um Definitionen ergänzt, wie XQuery-Abfragen innerhalb von SQL-

Anweisungen eingesetzt werden können. SQL/XML wird in dieser Arbeit in Abschnitt 4.4 vorgestellt.

Einige interessante Neuerungen des im Juli 2008 veröffentlichten Standards SQL:2008 [ISO08] sind: die Standardisierung des bereits von einigen Datenbanken unterstützten Kommandos `TRUNCATE TABLE` zur schnellen Leerung einer Tabelle, die Erweiterung der Befehle `CASE` und `MERGE`, die Möglichkeit den Datentyp einer Spalte zu ändern und die Einführung von `INSTEAD-OF`-Triggern.

Vorläufiger Abschluss der Entwicklung bildet der im Dezember 2011 verabschiedete SQL:2011-Standard [ISO11]. Dieser bietet unter anderem Erweiterungen im Bereich der Fensterrang- und Fensteraggregatfunktionen, Möglichkeiten die Ausgabe von Einfüge-, Änderungs- und Löschkommandos direkt in Abfragen zu verwenden und Unterstützung für temporäre Tabellen. Daneben wurden eine Reihe kleinerer Veränderungen, Korrekturen und Klarstellungen in den neuen Standard aufgenommen.

Die beschriebene rasante Weiterentwicklung des SQL-Standards zeigt, dass SQL heute weit mehr als nur eine Abfragesprache für relationale Datenbanken ist. Die Kommandos von SQL lassen sich in die folgenden Kategorien unterteilen:

- DDL – Data Definition Language
  Hierunter fallen Kommandos zum Anlegen (`CREATE`), Löschen (`DROP`) und Ändern (`ALTER`) von Tabellenschemata und Objekttypdefinitionen.
- DML – Data Manipulation Language
  Mit den DML-Befehlen können Daten abgefragt (`SELECT`), eingefügt (`INSERT`), geändert (`UPDATE`) oder gelöscht (`DELETE`) werden.
- DCL – Data Control Language
  Diese Kategorie besteht aus den Kommandos `GRANT` und `REVOKE` zur Vergabe und zum Entzug von Benutzerrechten.
- TCL – Transaction Control Language
  Mit den Befehlen `COMMIT` und `ROLLBACK`, die zur TCL gehören, kann die Transaktionsverarbeitung gesteuert werden.

Vergessen werden darf auch nicht, dass der SQL-Standard zusätzlich zur Definition der eigentlichen Sprache SQL auch Vorgaben für die Einbettung der SQL-Kommandos in Host-Sprachen (Teil 10, SQL/OLB), für die Integration externer Daten (Teil 9, SQL/MED) sowie für den Zugriff auf API-Ebene (Teil 3, SQL/CLI) macht. Dadurch ist ein sehr komplexes Gesamtwerk mit über 3700 Seiten entstanden.

Leider ist der SQL-Standard nicht frei verfügbar. Im Gegensatz zu den Spezifikationen des W3C sind die meisten Veröffentlichungen von ANSI und ISO nur käuflich zu erwerben. Der Preis für die CD-ROM mit allen Teilen des Standards beträgt etwa 200 Euro. Dies führt dazu, dass es für viele Anwender nicht möglich ist, einen direkten Blick in den Standard zu werfen. Stattdessen muss häufig auf

Informationen aus zweiter oder dritter Hand oder auf frei verfügbare, allerdings noch unfertige Entwürfe des SQL-Standards zurückgegriffen werden.

Ein anderer großer Kritikpunkt ist, dass die einzelnen SQL-Implementierungen der verschiedenen Hersteller inkonsistent und meist nicht untereinander kompatibel sind. Die Unterschiede beginnen bereits bei so grundlegenden Dingen wie der Repräsentation von Uhrzeit und Datum, einfachen Verknüpfungen von Zeichenketten und der Behandlung von Null-Werten.

Ein weiteres Beispiel ist der Teilstandard SQL/PSM (Persistent Stored Modules). Dieser definiert imperative Sprachkonstrukte, um unter anderem das Verhalten benutzerdefinierter Datentypen festzulegen. Die hieraus entstandenen Produkte sind PL/SQL (Procedural Language SQL) bei Oracle, T-SQL (Transact-SQL) bei Microsoft und SPL (Stored Procedure Language) bei Informix. Alle genannten Sprachen sind aber weitgehend nicht normgerecht und untereinander nicht kompatibel.

Ursache für diese Probleme sind zum einen der Umfang und die Komplexität des SQL-Standards. Dies hat zur Folge, dass viele Hersteller nicht den kompletten Standard implementieren oder die Entwicklung neuer Funktionen erst weit nach Veröffentlichung des Standards fertiggestellt wird. Damit die einzelnen SQL-Implementierungen nicht zu weit voneinander abweichen und es zumindest Einheitlichkeit bei den Basisfunktionen gibt, wurde in SQL-92 eine Unterteilung in die Ebenen Entry SQL, Intermediate SQL und Full SQL vorgenommen.

In SQL:2003 wurde ein minimaler Funktionsumfang mit der Bezeichnung Core SQL eingeführt, der von jeder konformen Implementierung mindestens zu unterstützen ist. Zusätzlich wurden themenbezogene Packages mit einzelnen so genannten Features festgelegt, die zusätzliche Funktionen beschreiben, die aber nicht unbedingt unterstützt werden müssen. Durch diese Aufteilung des Funktionsumfangs wird gewährleistet, dass die verschiedenen Datenbankmanagementsysteme besser miteinander verglichen und schneller Aussagen darüber getroffen werden können, ob bestimmte Anwendungen über Datenbankgrenzen hinweg kompatibel sind.

Eine weitere Ursache für die mangelnde Verträglichkeit liegt neben der Komplexität des Standards und manchmal ungenauer semantischer Beschreibungen einzelner Funktionen auch darin, dass viele Fähigkeiten bereits lange Zeit von den Datenbankherstellern implementiert worden sind, bevor sie in den SQL-Standard Einzug gehalten haben. Dies führte dazu, dass die verschiedenen Datenbanken zwar ähnliche Funktionen anboten, aber die Syntax der zugehörigen SQL-Befehle sehr unterschiedlich war.

Zum Zeitpunkt der Standardisierung dieser Funktionen musste eine Einigung auf eine gemeinsame Definition von Syntax und Semantik stattfinden. Dies brachte erhebliche Probleme und führte häufig bei großen Abweichungen zwischen Standard und eigener Implementierung bei den Herstellern dazu, dass der Standard nicht umgesetzt wurde. Eine Verwendung des SQL-Standards hätte

meist den Verlust der Rückwärtskompatibilität bedeutet, sodass bestehende Datenbankanwendungen hätten umgeschrieben werden müssen. Zum Teil sind die dadurch entstandenen Abweichungen vom Standard bis heute nicht korrigiert worden.

Zuletzt soll noch auf einen weiteren Kritikpunkt des SQL-Standards hingewiesen werden. Der Standard spart die in der ANSI-SPARC-Architektur so genannte interne Ebene völlig aus. Diese auch physikalische Schicht bezeichnete Ebene umfasst die Art und die Form der Speicherung der Daten. Hierzu zählen unter anderem die Indexe und die Datenablage auf der Festplatte. Einerseits gibt es gute Gründe, dies nicht im SQL-Standard festzulegen, da dieser im Wesentlichen das Verhalten der Datenbank aus Anwendungssicht definiert. Andererseits hätte die Standardisierung einiger Funktionsabläufe der internen Ebene den Vorteil, dass Datenbankadministratoren nicht mehr nur Spezialisten für die Verwaltung und Einrichtung genau eines Datenbankprodukts wären, auch wenn man vielleicht nicht die Homogenität der Linux-Welt erreicht, wo sich Systemadministratoren in der Regel leicht von einer Distribution auf eine andere umstellen können.

3.2 Objektrelationales Typsystem von Standard-SQL

In diesem Abschnitt wird das objektrelationale Typsystem von SQL beschrieben, das mit den Versionen SQL:1999 und SQL:2003 eingeführt wurde. Im Mittelpunkt stehen dabei die Kollektionstypen, die die Definition von mengen- und listenwertigen Attributen ermöglichen. In diesem Zusammenhang wird insbesondere auch auf die Objekt- und Referenztypen eingegangen.

Andere Erweiterungen von SQL, wie zum Beispiel die Unterstützung für große Datentypen (`BLOB`, `CLOB`), OLAP-Funktionalitäten [Pet07], Multimedia- und Geometriekomponenten, die häufig ebenfalls unter den Begriff der Objektrelationalität fallen, werden hier nicht betrachtet, und es wird auf die ausführliche Behandlung in [Pet03], [Tür03] oder [TS06] verwiesen.

3.2.1 SQL:1999

Da das objektrelationale Typsystem von SQL:2003 bereits recht komplex ist, sollen hier zunächst die mit SQL:1999 eingeführten objektrelationalen Eigenschaften beschrieben werden. Dies erleichtert zunächst die Darstellung und schafft eine Grundlage, um im nächsten Abschnitt die mit SQL:2003 eingeführten Erweiterungen zu behandeln.

Es gibt zwei Einstiegspunkte in das Typsystem von SQL:1999. Dies sind zum einen die Tupeltabellen und zum anderen die so genannten typisierten Tabellen. Tupeltabellen sind traditionelle Tabellen im Sinne von SQL-92 und bilden den Kern des relationalen Teils des Typsystems. Im Relationenmodell präsentiert eine

Relation eine *Menge* von Tupeln. Im Gegensatz dazu gilt für die Tupeltabellen von SQL:1999, so wie bereits seit den Anfängen von SQL üblich, dass sie eine *Multimenge* von Tupeln darstellen.

Mit SQL:1999 wurden die Tupeltabellen insofern erweitert, dass als Typen der Tabellenspalten neben den Basisdatentypen auch Tupel-, Array-, Objekt- oder Referenztypen verwendet werden können. Für Tupel-, Array- und Referenztypen werden die folgenden unbenannten Typkonstruktoren benutzt:

- Tupeltypkonstruktor `ROW`
- Arraytypkonstruktor `ARRAY`
- Referenztypkonstruktor `REF`

Zur Nutzung dieser Typen wurden neben den Konstruktoren auch zugehörige neue Operatoren eingeführt. Für den Tupeltyp existiert die Zugriffsmöglichkeit auf ein Tupelfeld mittels des aus vielen Programmiersprachen bekannten Punktoperators. Weiterhin ist der Vergleich zweier Tupel über die üblichen Vergleichsprädikate (<, <=, >, >=, =, <>) möglich, sofern die Tupel die gleiche Anzahl an Feldern aufweisen und die Feldtypen paarweise vergleichbar sind. Das Ergebnis des Vergleichs zweier Tupel ergibt sich aus der Zurückführung auf den fortlaufenden paarweisen Vergleich der Tupelfelder beginnend mit dem ersten Feldpaar.

Bei den in SQL:1999 eingeführten Arraytypen muss die maximale Kardinalität bei Erzeugung des Typs angegeben werden. Ein Arraytyp stellt eine begrenzte, geordnete Multimenge dar, deren Elemente denselben Typ aufweisen. Wie bei Arrays im Allgemeinen üblich, ist auch in SQL:1999 der direkte Zugriff auf ein Element über den Positionsindex vorgesehen. Daneben kann mittels des Operators `UNNEST` ein Array in eine Tabelle transformiert werden. Die Funktion `CARDINALITY` ermittelt die Kardinalität eines Arrays. Zwei Arrays können auf Gleichheit getestet werden, sofern ihre Elementtypen vergleichbar sind. Schließlich ist es möglich, zwei Arrays über den Operator || miteinander zu verknüpfen. Änderungen von Arrays sind sowohl nur auf einzelnen Elementen als auch auf dem gesamten Array erlaubt.

Eine Instanz eines Referenztyps wird in SQL als Referenz, Objektidentifikator oder Objekt-ID (OID) bezeichnet. Diese Referenz kann auf ein Objekt verweisen, dessen Typ bei der Definition des Referenztyps angegeben werden muss. Zum Umgang mit Referenztypen stehen folgende Operatoren zur Verfügung: Wie auch in anderen Programmiersprachen können Referenzen mittels des Pfeiloperators (`->`) dereferenziert werden. Mit Hilfe der Funktion `DEREF` kann eine Referenz komplett aufgelöst werden, dies liefert die Transformation des referenzierten Objekts in ein Tupel. Referenzen können verglichen werden, falls die referenzierten Objekte aus derselben Typhierarchie stammen. Gleichheit gilt genau dann, wenn die Referenzen identisch sind.

Objekttypen werden in SQL:1999 über das Kommando `CREATE TYPE` erzeugt. Bei der Definition können die Attribute und Methoden des Objekttyps

definiert werden. Die Bildung einer Typhierarchie mit einem Wurzeltyp und mehreren Subtypen ist möglich. Bei Bedarf kann ein Objekttyp mittels der Klausel `NOT INSTANTIABLE` zu einem nicht-instanziierbaren Typ erklärt werden, sodass er einer abstrakten Klasse in objektorientierten Programmiersprachen entspricht.

In SQL:1999 ist der Zugriff auf die Attribute eines Objekts aufgrund der Kapselung nicht direkt möglich. Deshalb werden für jedes Attribut automatisch eine Observer- und eine Mutator-Methode erzeugt, über die die Attributwerte gelesen und geändert werden können. Zusätzlich kann mittels des Prädikats `IS OF` der Typ eines Objekts ermittelt und auch temporär innerhalb einer Typhierarchie über die Klausel `AS` oder `TREAT AS` angepasst werden. Um Objektwerte miteinander vergleichen zu können, muss für sie eine benutzerdefinierte Ordnung festgelegt werden.

Die Subtypbildung geschieht über das Schlüsselwort `UNDER`. Jeder Subtyp erbt alle Attribute und Methoden des Supertyps. Wie üblich, können auch in SQL:1999 geerbte Methoden überschrieben werden. Jeder Subtyp kann nur genau einen Supertyp besitzen, das heißt, Mehrfachvererbung ist nicht zugelassen. Es gilt das Prinzip des Substituierbarkeit. Somit kann an allen Stellen, an denen eine Instanz des Supertyps erwartet wird, auch eine Instanz eines Subtyps verwendet werden kann. Dies betrifft sowohl Parameter und Rückgabewerte von Prozeduren als auch Zuweisungen an Variablen, Objektattribute und Tabellenspalten.

Wenn man die Instanzen der Objekttypen von SQL:1999 streng unter objektorientierter Betrachtungsweise beurteilt, handelt es sich bei ihnen nicht um Objekte, sondern nur um Instanzen strukturierter Typen. Hauptgrund ist, dass die Instanzen von sich aus keine OID als eindeutige Referenz und damit keine Objektidentität besitzen. Eine Referenzierung ist nur möglich, falls die Instanzen als Zeilen einer typisierten Tabelle auftreten und mit einem OID-Attribut assoziiert werden. Dabei kann angegeben werden, ob der Referenztyp eine system- oder benutzerdefinierte Repräsentation aufweisen soll oder aus Attributen des Objekttyps gebildet wird. Unter objektorientierten Gesichtspunkten widerspricht die benutzerdefinierte Darstellung und die Erzeugung der OID aus Attributen jedoch der Vorstellung, dass die OID ein vom Objektinhalt unabhängiger Eigenschaftswert ist, der die Identität des Objekts festlegt.

Auf die Details und Besonderheiten der Methodendefinition und -implementierung soll hier nicht weiter eingegangen werden. Anzumerken ist, dass – vergleichbar mit anderen objektorientierten Programmiersprachen – drei Arten von Methoden unterschieden werden: Konstruktormethoden, die eine neue Instanz eines Objekttyps initialisieren, Instanzmethoden, die auf einer Instanz eines Objekttyps aufgerufen werden sowie statische Methoden, die unabhängig von einer Instanz des Objekttyps verwendet werden können. Das Prinzip der Methodenüberladung wird unterstützt.

Der Einsatz eines Objekttyps als Datentyp einer Spalte einer Tupeltabelle ist jedoch nur eine Verwendungsmöglichkeit. Die Objekttypen bilden ebenfalls die Grundlage für die so genannten typisierten Tabellen, die den zweiten Einstiegspunkt in das Typsystem von SQL:1999 bilden. Eine typisierte Tabelle ist eine Objekttabelle, deren Aufbau durch einen Objekttyp bestimmt wird. Die Spalten einer typisierten Tabelle sind die Attribute des Objekttyps, wobei die OID als erste Spalte hinzugefügt wird. Die Zeilen einer typisierten Tabelle repräsentieren die einzelnen Objektinstanzen. Auf den Zeilen können die Objektmethoden aufgerufen werden. Des Weiteren sind sie über die OID referenzierbar. Im Gegensatz zu Tupeltabellen repräsentieren typisierte Tabellen keine Multimengen, sondern Mengen. Die Mengeneigenschaft wird durch die enthaltene OID sichergestellt.

Analog zur Bildung von Objekthierarchien ist die Bildung von Tabellenhierarchien mit Wurzel- und Subtabellen möglich. Auch bei den Tabellenhierarchien wird das Prinzip der Substituierbarkeit verwendet, das heißt, an Stellen, an denen eine Instanz eines Supertyps erwartet wird, kann auch ein speziellerer Typ der Hierarchie, also ein Subtyp, eingesetzt werden.

Vergleichbar mit der Definition von Tupelsichten bei Tupeltabellen können bei typisierten Tabellen typisierte Sichten bzw. Objektsichten gebildet werden. Objektsichten sind virtuelle Objekttabellen, die sowohl objekterhaltend als auch objekterzeugend definiert werden können. Aus typisierten Sichten wiederum ist der Aufbau von Sichtenhierarchien möglich. Auf die Details wird hier verzichtet und auf [Tür03] verwiesen.

Zusammenfassend ergibt sich für SQL:1999 das in Abbildung 3–1 dargestellte Typsystem. Der dunkel gekennzeichnete Bereich auf der linken Seite umfasst den relationalen Teil des Typsystems: Tupeltabellen sind Multimengen, die aus einzelnen Tupeln bzw. Zeilen bestehen, deren Spalten jeweils einem Basisdatentyp zugeordnet sind. Über die traditionelle relationale Sichtweise hinaus geht bereits die Möglichkeit, den Spalten einer Tabelle auch einen Referenz-, Array-, Tupel- oder Objekttyp zuzuordnen. Es ist jedoch in SQL:1999 noch keine vollständige Orthogonalität zwischen diesen Typen gegeben, da weder eine direkte noch indirekte Schachtelung von Arraytypen zugelassen ist.

Der objektorientierte Teil von SQL:1999 ist auf Abbildung 3–1 oben rechts dargestellt: typisierte Tabellen, die eine Menge von Objekten aufnehmen. Die Attribute der Objekte können als Typ einen Basisdatentyp, einen Referenz-, Array-, Tupel- oder auch einen Objekttyp aufweisen. Bei der Definition muss allerdings darauf geachtet werden, dass implizit keine geschachtelten Arraytypen entstehen. Die Möglichkeit der Erzeugung von Typ- und Tabellenhierarchien mittels Subtyp- bzw. Subtabellenbildung ist in der Abbildung durch entsprechende selbstreferenzierende Pfeile wiedergegeben.

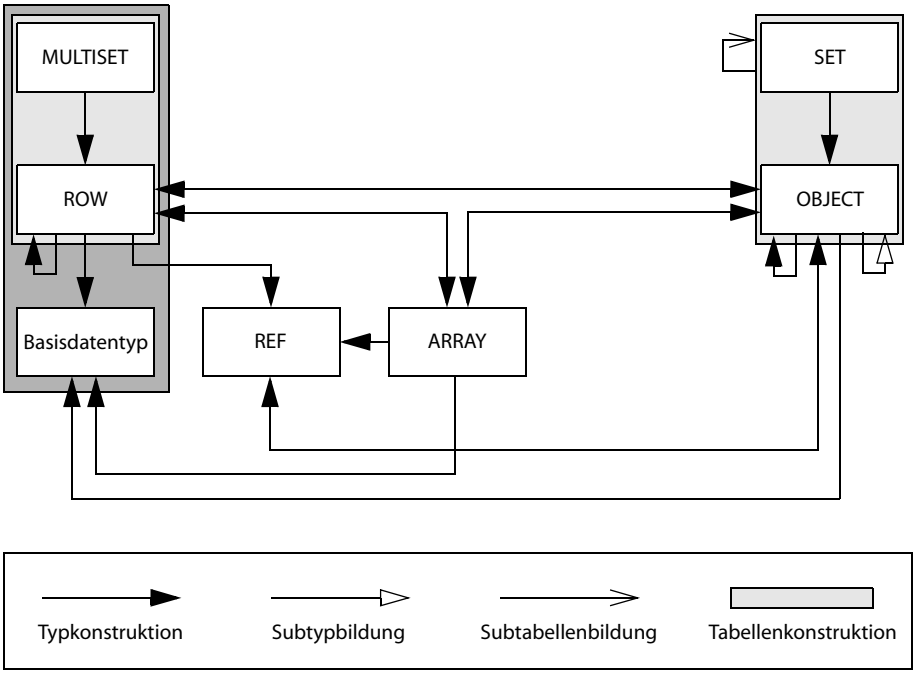


Abb. 3–1 *Objektrelationales Typsystem von SQL:1999 (nach [Tür03])*

An dieser Stelle sei darauf hingewiesen, dass die häufig gelesene Bemerkung, Codd hätte Objekt-IDs als „second great blunder" für das relationale Modell bezeichnet, nicht korrekt ist. Die Aussage stammt aus dem „Third Manifesto" von Darwen und Date [DD00]. Gittens [Git05] stellt klar, dass Codd sich in [Cod79] jedoch explizit für den Bedarf nach eindeutigen und permanenten IDs für Datenbankentitäten ausgesprochen hat. Diese Stellvertreter sollen nach Codd einer so genannten „E domain" entstammen und haben die Eigenschaften von Objekt-IDs.[5]

3.2.2 SQL:2003

Nachdem im vorherigen Abschnitt das objektrelationale Typsystem von SQL:1999 vorgestellt wurde, werden in diesem Abschnitt die mit der Einführung von SQL:2003 hinzugekommenen Erweiterungen dargestellt.

Wichtigste Neuerung ist die Einführung von Multimengen, also ungeordneten Kollektionen, die Duplikate enthalten dürfen. Multimengentypen werden

5. Das Gleichsetzen von Relationenvariablen mit Klassen wird von Date und Darwen als „first great blunder" bezeichnet. Gittens zeigt, dass auch diese Aussage nicht auf Codd zurückgeführt werden kann.

mit dem Schlüsselwort MULTISET definiert. Für Multimengen existieren verschiedene Konstruktoren, mit denen eine leere Multimenge, eine Multimenge aus angegebenen Werten oder auch eine Multimenge aus dem Ergebnis einer SQL-Abfrage erzeugt werden kann. Auch die Umkehrung ist möglich: mit der Funktion UNNEST lässt sich eine Multimenge in eine Tabelle transformieren.

Weitere Operationen auf Multimengen sind die Kardinalitätsermittlung mit CARDINALITY sowie die Vereinigung (UNION), die Differenz (EXCEPT) und der Durchschnitt (INTERSECT) zweier Multimengen.[6] Mittels des Prädikats MEMBER OF bzw. SUBMULTISET OF lässt sich feststellen, ob ein Element oder eine Multimenge in einer anderen Multimenge enthalten ist. Daneben existiert eine Testmöglichkeit, ob eine Multimenge eine Menge ist. Falls nein, kann sie durch Duplikatselimination zu einer solchen gemacht werden. Falls zwei Multimengen vergleichbare Elementtypen aufweisen, können sie ebenfalls verglichen werden. Gleichheit gilt genau dann, wenn beide Multimengen die gleichen Elemente in der jeweils gleichen Kardinalität besitzen.

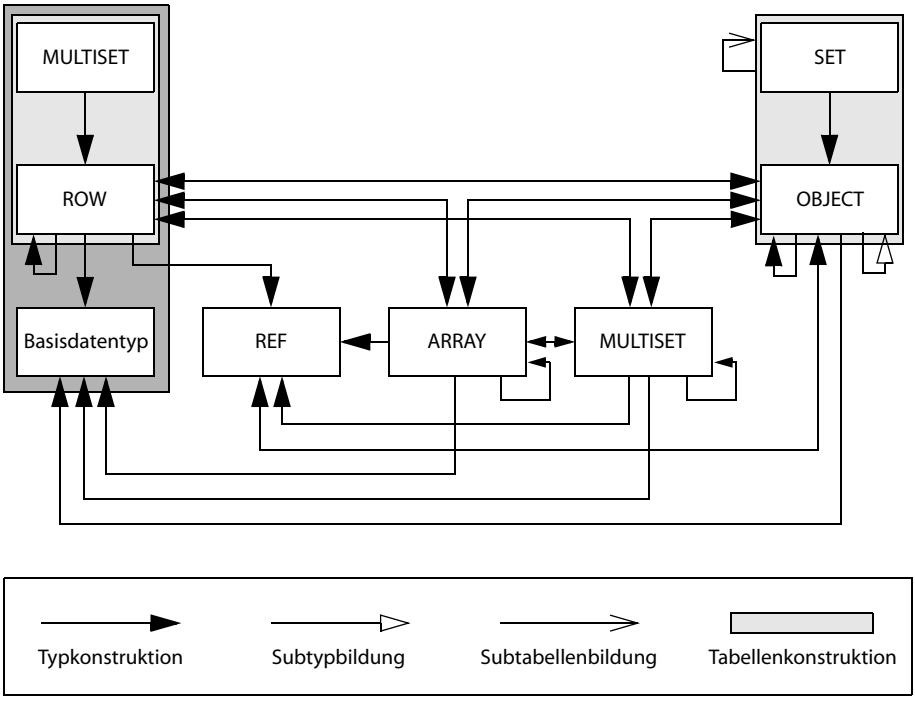


Abb. 3–2 *Objektrelationales Typsystem von SQL:2003 (nach [Tür03])*

Mit SQL:2003 wurde die Einschränkung aufgehoben, dass Arraytypen nicht geschachtelt werden dürfen. Die beliebige Kombination von Array- und auch

6. Für Multimengen gibt es zwei verschiedene Definitionen der Vereinigung, die beide Nachteile haben. Eine Erläuterung mit Beispielen findet sich in [Weg11, S. 30f.].

Multimengentypen ist nun möglich. Die bereits in einigen Datenbankimplementierungen vorhandenen Listen- und Mengentypkonstruktoren `LIST` und `SET` wurden nicht in den SQL:2003-Standard aufgenommen.

Abbildung 3–2 gibt einen abschließenden Überblick über das mit SQL:2003 eingeführte objektrelationale Typsystem. An dieser Stelle soll auf die Dissertation von Lufter [Luf05] hingewiesen werden, der in seiner umfassenden Arbeit das objektrelationale Typsystem von SQL:2003 untersucht und Vorschläge macht, wie die Kollektionsunterstützung ausgebaut werden kann. Dazu gibt er an, welche Erweiterungen an der Sprachspezifikation des SQL-Standards vorgenommen werden müssen, um dies einfach und mit größtmöglicher Orthogonalität zu erreichen. Daneben entwirft er auch ein Konzept, das die Festlegung von Integritätsbedingungen für komplexe Datenstrukturen ermöglicht.

3.3 Objektrelationales Typsystem von Datenbankimplementierungen

In diesem Abschnitt soll der objektrelationale Teil des Typsystems einiger Datenbankimplementierungen dargestellt und mit dem zuvor beschriebenen Typsystem von SQL:2003 verglichen werden. Die Unterschiede, Besonderheiten und Einschränkungen der einzelnen Datenbanken sollen dabei herausgearbeitet werden. Dazu wurden die Datenbanken Oracle, IBM DB2 sowie IBM Informix ausgewählt, da diese zum einen einen relativ hohen Verbreitungsgrad aufweisen und zum anderen Unterstützung für objektrelationale Typen anbieten. Die ebenfalls häufig eingesetzten Datenbankmanagementsysteme Microsoft SQL Server sowie die Open-Source-Datenbank MySQL werden hier nicht untersucht, da beide kaum objektrelationale Typen unterstützen. [KW07] liefert eine allgemeine Untersuchung bezüglich der Konformität verschiedener DBMS in Bezug auf den SQL:2003-Standard.

Die meisten Datenbankmanagementsysteme bieten in ihren aktuellen Versionen auch eine direkte Unterstützung für die Speicherung von XML-Daten. Die dabei eingesetzten Verfahrensweisen und deren Verwendung werden in diesem Abschnitt ausgeklammert. Das sich anschließende Kapitel 4 widmet sich den Fragen der Abbildungs- und Speichervarianten von XML-Daten sowie der spezialisierten XML-Funktionalitäten der Systeme.

Die folgenden drei Unterabschnitte behandeln nacheinander die objektrelationalen Typsysteme der Datenbanken Oracle, DB2 und Informix.

3.3.1 Oracle

Einige der im vorherigen Abschnitt vorgestellten objektrelationalen Eigenschaften werden von Oracle bereits seit den Versionen 8 und 8i aus den Jahren 1997 und 1999 in gleicher oder ähnlicher Form unterstützt [Stü00]. In der Nach-

folgeversion 9i (2001) wurden größere, in den Versionen 10g (2003) und 11g (2007) nur noch kleinere Erweiterungen vorgenommen. [Tür03] beschreibt den Stand von Oracle 9i, [LZ03] und [Feu03] geben einen Überblick über die objektrelationen Erweiterungen, die mit Version 10g hinzugekommen sind. [Ora11b] dokumentiert den aktuellen Funktionsumfang des objektrelationalen Typsystems von Oracle 11g Release 2 (2009), der im Folgenden dargestellt wird.

Das Datenbankmanagementsystem Oracle unterstützt neben den üblichen Basisdatentypen Referenz-, Array-, Tabellen- und Objekttypen. Der Referenztypkonstruktor ist in Oracle der einzige unbenannte Typkonstruktor. Er arbeitet wie in SQL:2003 vorgesehen mit dem Schlüsselwort `REF`, wobei die Syntax jedoch leicht vom Standard abweicht. Die Dereferenzierung ist explizit über die Funktion `DEREF` oder implizit über eine Punktnotation möglich. Über das Prädikat `IS DANGLING` lassen sich ungültige Referenzen ermitteln, für die in der Datenbank das ursprüngliche Bezugsobjekt nicht mehr vorhanden ist.

Oracle kennt nicht den in SQL:2003 definierten unbenannten Arraytyp. Stattdessen müssen alle Arraytypen in Oracle benannt sein. Der Arraytypkonstruktor hat die Bezeichnung `VARRAY` (im Gegensatz zu `ARRAY` bei SQL:2003). `VARRAY` steht für variables Array, dennoch ist auch in Oracle die maximale Kardinalität eines Arrays bei Definition anzugeben. Seit Oracle 10g ist jedoch ein Kommando verfügbar, um die maximale Arraygröße auch nachträglich noch zu erhöhen.

Leider ist es in Oracle innerhalb von SQL-Abfragen nicht erlaubt, die einzelnen Elemente eines Arrays über ihren Positionsindex auszulesen oder zu ändern. Zugriff auf die Elemente ist nur innerhalb von PL/SQL-Blöcken möglich. Dazu bietet PL/SQL eine Reihe von kollektionsspezifischen Methoden an, mit denen zum Beispiel der Wert eines bestimmten Elements ermittelt, ein Element gelöscht, das Array erweitert oder die Kardinalität abgefragt werden kann. Aufgrund der Beschränkung dieser Methoden auf PL/SQL-Blöcke ist es bei Einsatz von Oracle erforderlich, für alle im Betrieb der Datenbank vorkommenden Operationen auf den vorhandenen Arraytypen Prozeduren zu definieren, die die gewünschten Aktionen ausführen. Spontane Änderungen von Arrays sind nur mit größerem Aufwand möglich.

An Stelle des in SQL:2003 eingeführten unbenannten Multimengentyps `MULTISET` tritt in Oracle ein benannter, benutzerdefinierter Tabellentyp, dessen Konstruktor das Schlüsselwort `TABLE` verwendet. Der Tabellentyp repräsentiert eine unbegrenzte, ungeordnete Multimenge, deren Elemente denselben Tupeltyp aufweisen. Mit dieser Konstruktion lassen sich in Oracle die geschachtelten Relationen des NF^2-Modells [AB84] darstellen. Bei der Definition einer geschachtelten Tabelle muss die Klausel `STORE AS` angegeben werden, um den Namen einer internen Tabelle festzulegen, in der der Inhalt der tabellenwertigen Spalte abgelegt wird. Auf diese interne Tabelle kann nicht direkt zugegriffen werden, ihr Name wird für die Definition von Indexen benötigt.

Um innerhalb von SQL-Abfragen auf den Inhalt von geschachtelten benutzerdefinierten Tabellentypen zuzugreifen, ist in Oracle der `TABLE`-Operator erforderlich. Dieser ist mit der Funktion `UNNEST` in SQL:2003 vergleichbar, die die Umwandlung eines Multimengentyps in eine Tabelle durchführt. Die umgekehrte Transformation ist in Oracle über den Operator `CAST` in Verbindung mit dem Schlüsselwort `MULTISET` möglich. Die Verwendung des Begriffs `MULTISET` überrascht an dieser Stelle, da Oracle keinen echten Multimengentyp unterstützt. Mit der Version 10g von Oracle wurden erweiterte Mengenoperationen auf Tabellentypen, Vergleichsmöglichkeiten zwischen Tabellentypinstanzen sowie eine Funktion zum Entfernen doppelter Elemente eingeführt.

Die Objekttypen von Oracle entsprechen im Wesentlichen denen von SQL:2003. Streng genommen handelt es sich auch bei ihnen eigentlich um strukturierte Typen. Die Syntax der Objekttypdefinition ist erheblich von derjenigen von SQL:2003 verschieden. Die Bildung von Subtypen und das Prinzip der Substituierbarkeit werden unterstützt. Gegenseitige, zyklische Beziehungen, die in Standard-SQL nicht zugelassen sind, sind in Oracle erlaubt, indem Vorwärtsdeklarationen eingesetzt werden.

Der direkte Zugriff auf die Attribute eines Objekts geschieht durch die Punktnotation. Der Test eines Objekts auf einen bestimmten Typ und die temporäre Typanpassung sind konform zum SQL-Standard. Um Objekte ordnen und miteinander vergleichen zu können, bietet Oracle die Möglichkeit der Angabe benutzerdefinierter Abbildungs- und Ordnungsmethoden.

Im Bereich der Objektmethoden unterstützt Oracle das Überladen und Überschreiben von Methoden sowie die Unterteilung in Instanz- und statische Methoden. Hierbei weicht die Syntax wiederum von Standard-SQL ab. In Oracle können Methoden nicht nur als objektspezifische Funktionen, sondern auch als objektspezifische Prozeduren angelegt werden. Bei den Prozeduren entfällt die Rückgabe eines Werts über die `RETURN`-Klausel. Außerdem können Prozeduren nicht innerhalb von SQL-Kommandos eingesetzt werden. Sie sind nur in einem PL/SQL-Block aufrufbar.

Kleinere Unterschiede gibt es auch beim Aufruf der Methoden. Oracle verlangt beim Methodenaufruf sowohl für Instanz- als auch statische Methoden die Angabe eines Punktes zwischen Objekt bzw. Objekttyp und Methodenname. Standard-SQL erfordert bei statischen Methoden jedoch den Einsatz von `::` zwischen Typ und Methode. Außerdem sollte beachtet werden, dass in Oracle auch parameterlose Methoden mit Klammern hinter dem Methodennamen aufgerufen werden müssen.

Das Prinzip der typisierten bzw. Objekttabellen wird von Oracle unterstützt, wobei die Einschränkungen gelten, dass eine eigene Syntax eingesetzt wird und es nicht möglich ist, Subtabellen und Tabellenhierarchien zu definieren. Die Zeilen einer typisierten Tabelle, also die Objekte, können über eine OID referenziert

werden. Diese wird entweder automatisch vom System erzeugt oder aus einem Primärschlüssel abgeleitet.

Das auch in SQL:2003 vorhandene Konzept der typisierten Sichten bzw. Objektsichten sowie die Bildung von Subsichten und Sichtenhierarchien wird von Oracle angeboten, allerdings wird auch hierbei eine eigene Syntax eingesetzt. Einen Tupeltyp, wie ihn SQL:2003 definiert, bietet Oracle nicht. Diese Einschränkung kann umgangen werden, indem stattdessen ein Objekttyp verwendet wird.

Abbildung 3–3 gibt zusammenfassend einen Überblick über das objektrelationale Typsystem von Oracle. Die von Oracle gebotene Orthogonalität zwischen den Kollektionstypkonstruktoren ist darin gut zu erkennen.

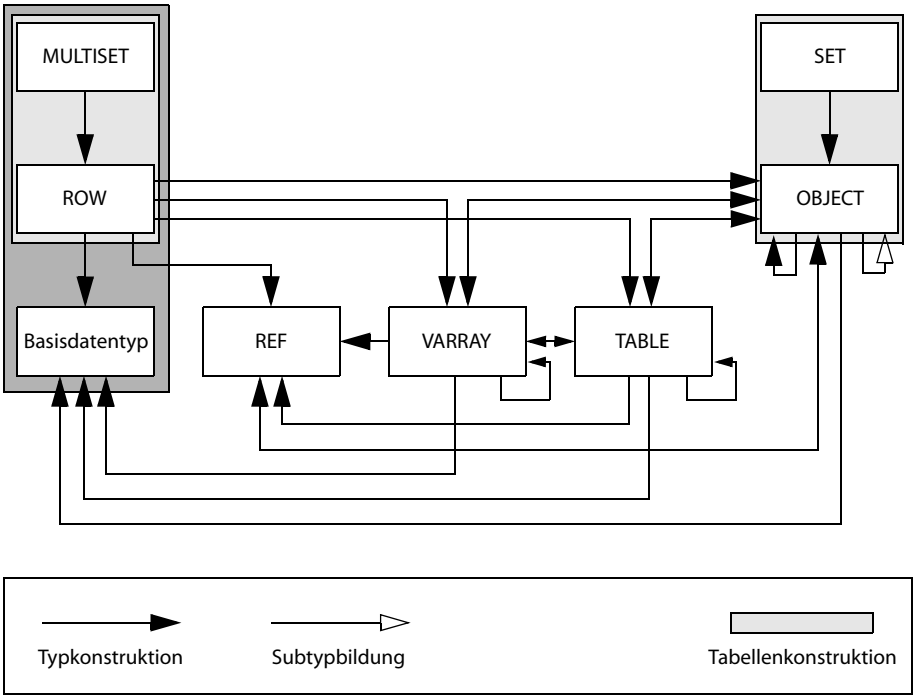


Abb. 3–3 *Objektrelationales Typsystem von Oracle (nach [Tür03])*

3.3.2 DB2

Die Anfänge des Datenbankmanagamentsystems IBM DB2 gehen auf das in den 1970er Jahren von IBM entwickelte System R zurück. Ein Großteil der objektrelationalen Fähigkeiten von DB2 wurde mit der Version 7 aus dem Jahr 2001 eingeführt. Das Konzept der strukturierten Typen hat seinen Ursprung sogar schon in Version 5.2 von 1998 [Zei01a, Zei01b]. Das objektrelationale Typsystem von DB2 Version 8.1 (2002) wird in [Tür03] beschrieben. Die Nachfolgeversionen bis

zur Version 9.7 (2010) brachten in diesem Bereich keine entscheidenden Erweiterungen. Details können [IBM11a] entnommen werden.

Referenztypen werden von DB2 unterstützt. Der zugehörige Konstruktor generiert, wie auch in Standard-SQL vorgesehen, unbenannte Typen. Ziel der Referenzen können Instanzen von strukturierten Typen bzw. Objekttypen sein. Alle von SQL:2003 vorgesehenen Operationen auf Referenztypen werden angeboten, das heißt, Vergleiche von Referenzen sind möglich, die Verfolgung einer Referenz geschieht über den Pfeiloperator und die Auflösung wird durch die Funktion `DEREF` realisiert.

Auch bei den Objekttypen hält sich DB2 weitgehend an die von Standard-SQL definierten Konzepte. Die zugehörige Syntax ist ähnlich zu Standard-SQL, jedoch gibt es Abweichungen. DB2 bietet im Rahmen der Vererbung die Bildung von Subtypen. Allerdings wird die Klausel `FINAL`, die in SQL:2003, Oracle und vielen anderen Programmiersprachen festlegt, dass ein Typ keine Subtypen besitzen darf, nicht unterstützt. Die Substituierbarkeit von Typen ist möglich. Zyklische Beziehungen zwischen Objekttypen sind im Einklang mit dem Standard im Gegensatz zu Oracle nicht zugelassen.

Der Zugriff auf die Attribute und Methoden eines Objekts geschieht in DB2 durch eine etwas exotische Zwei-Punkte-Notation. Derartige Zugriffe können zu einem Pfadausdruck verkettet werden. Der Vergleich von zwei Objekten ist nicht erlaubt, ein Operator zur temporären Typanpassung ist vorhanden.

Bezüglich der Objektmethoden gibt es die Besonderheit in DB2, dass die Überladung von Methoden sowie die Erzeugung statischer Methoden nicht unterstützt werden. Des Weiteren ist zu beachten, dass Methoden nur lesend auf die SQL-Daten zugreifen und diese nicht ändern dürfen. Daneben gilt auch die Einschränkung, dass die Attribute des aufrufenden Objekts nicht geändert werden können.

Typisierte Tabellen und die Erzeugung von Subtabellen und Tabellenhierarchien sind in DB2 möglich. Die wesentlichen Eigenschaften stimmen mit denen von Standard-SQL überein. Die Spalten einer typisierten Tabelle setzen sich aus der OID als erste Spalte sowie den Attributen des zugeordneten Objekttyps zusammen. OIDs müssen stets vom Benutzer – gegebenenfalls unter Zuhilfenahme der angebotenen Funktion `GENERATE_UNIQUE` – beim Einfügen von Instanzen in eine typisierte Tabelle erzeugt werden. Die automatische systemgesteuerte OID-Generierung bietet DB2 nicht an. Typisierte Sichten, Subsichten und Sichtenhierarchien können ähnlich zum SQL-Standard definiert werden.

Abbildung 3–4 zeigt das gesamte von DB2 angebotene objektrelationale Typsystem. Im Vergleich zu SQL:2003 und Oracle werden große Unterschiede sichtbar. DB2 bietet keine Kollektionstypen. Dadurch ist die Bildung von Listen- und (Multi-)Mengentypen nicht möglich, sodass beispielsweise die Erzeugung von geschachtelten Tabellen nicht zu realisieren ist. Dies führt bei Verwendung von DB2 im objektrelationalen Umfeld zu erheblichen Einschränkungen.

Positiv muss bemerkt werden, dass seit Version 9.5 ein Arraytyp zur Verfügung steht, der allerdings nur innerhalb von benutzerdefinierten Prozeduren eingesetzt werden kann. Eine Verwendung als Datentyp für Spalten von Tupeltabellen ist nicht erlaubt.

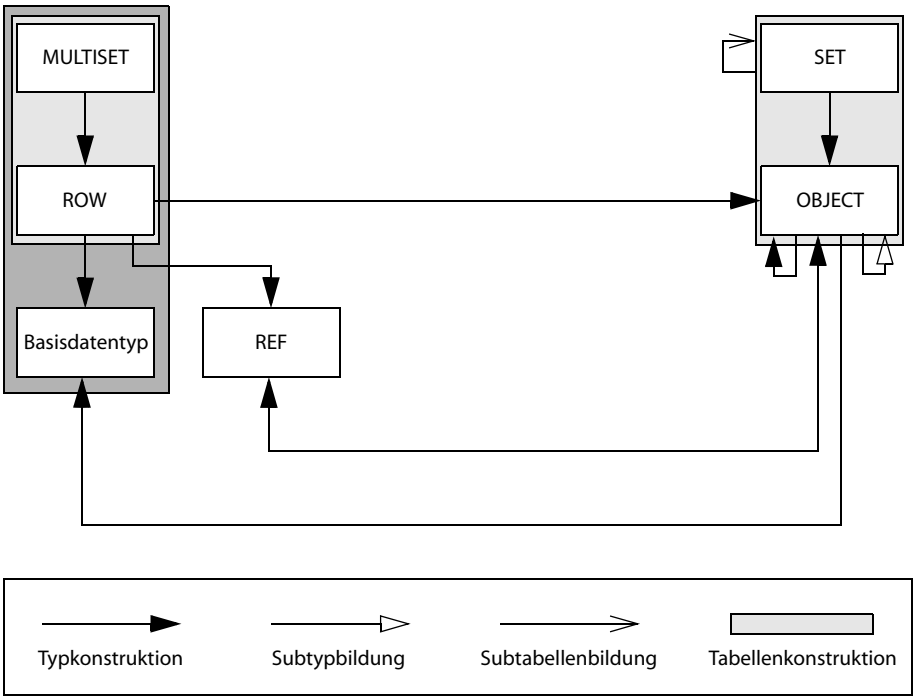


Abb. 3–4 *Objektrelationales Typsystem von DB2 (nach [Tür03])*

3.3.3 Informix

Informix ist die Abkürzung für „Information on Unix“. Die erste Version der Informix-Datenbank erschien 1985 unter der Bezeichnung Informix-SQL. Im Jahre 2001 wurde die Datenbanktechnologie und die Marke Informix durch IBM aufgekauft. Nachdem zunächst Informix mit DB2 zusammengeführt werden sollte, hat sich IBM dazu entschieden, sowohl die eigene Datenbank DB2 als auch Informix weiterzuentwickeln. DB2 und Informix verwenden in einigen Bereichen die gleiche Codebasis, neue Technologien werden meist in beiden Systemen eingeführt. In [Tür03] werden die Fähigkeiten der Version 9.3 aus dem Jahr 2003 beschrieben. Seit dieser Version hat es keine Erweiterungen des objektrelationalen Typsystems von Informix gegeben. Nachfolgend sind die Versionen 9.4 (2003), 10.0 (2005), 11.1 (2007), 11.5 (2008) und schließlich 11.7 (2010) erschienen. Die Details des aktuellen Funktionsumfangs der Datenbank, deren

vollständiger Name zur Zeit Informix Dynamic Server (IDS) lautet, lassen sich in [IBM11b] nachlesen.

Informix unterstützt den im SQL-Standard vorgesehenen unbenannten Tupeltypkonstruktor `ROW`. Der Zugriff auf die einzelnen Tupelfelder geschieht über den üblichen Punktoperator. Der Vergleich von Tupeln ist unter den Voraussetzungen gleiche Feldanzahl und paarweise vergleichbare Typen möglich. Dies entspricht ebenfalls dem SQL-Standard. Innerhalb von SQL-Kommandos können Tupel nur komplett geändert werden. Einzelne Tupelfelder sind nur über den Umweg einer benutzerdefinierten Routine zu ändern, die in der Programmiersprache SPL (Stored Procedure Language) formuliert werden muss.

Im Bereich der Kollektionstypen bietet Informix die Auswahl zwischen den drei unbenannten Konstruktoren `SET`, `MULTISET` und `LIST`. Wie die Bezeichnungen vermuten lassen, erzeugt `SET` einen Mengentyp, `MULTISET` einen Multimengentyp und `LIST` liefert einen Listentyp. Allen Typen gemeinsam ist die Eigenschaft, dass `NULL` als Element nicht zugelassen ist. Dies muss bei der Definition stets durch die notwendige Klausel `NOT NULL` angegeben werden. Als Elementtypen sind bei diesen Kollektionen fast alle Basis- und benutzerdefinierten Datentypen zugelassen. Aus diesem Grund zeichnet sich Informix durch große Typorthogonalität aus. Leider unterstützt Informix nicht den von SQL:2003 definierten Arraytypkonstruktor, sodass als Alternative der `LIST`-Typ genutzt werden muss, der jedoch nicht den direkten Zugriff auf ein bestimmtes Element erlaubt.

Die Syntax der Kollektionstypkonstruktoren fällt durch den unüblichen Einsatz von geschweiften Klammern und doppelten Anführungszeichen auf. Als Operatoren auf Kollektionstypen sind der Elementtest mittels `IN`, die Feststellung der Kardinalität über die Funktion `CARDINALITY` sowie die Transformation einer Kollektion in eine Tabelle mit Hilfe des `TABLE`-Operators vorhanden. Für die Änderung von Kollektionen gilt das oben für den Tupeltyp Beschriebene analog: Einzelne Elemente können nicht direkt, sondern nur im Rahmen einer SPL-Prozedur geändert werden.

Informix bietet neben den bereits erwähnten unbenannten Tupeltypen auch benannte Tupeltypen. Dies steht im Kontrast zu Standard-SQL und den anderen bisher vorgestellten Datenbankmanagementsystemen. Benannte Tupeltypen bestehen aus mindestens einem Attribut. Für den Typ der Tupelfelder gilt Orthogonalität, das heißt, es kann ein beliebiger Tupel-, Kollektions- oder Basisdatentyp eingesetzt werden. Die benannten Tupeltypen erlauben durch die Bildung von Wurzel- und Subtypen die Erzeugung von Typhierarchien. Dies erweckt den Anschein, die Tupeltypen von Informix wären mit den aus anderen Datenbanken bekannten Objekttypen vergleichbar. Dies ist jedoch nicht der Fall, da die Tupeltypen keine Methoden und auch keine OID besitzen können. Deshalb sind sie auch nicht referenzierbar.

Bei der Vererbung gilt für Tupeltypen das zu Erwartende: Subtupeltypen erben alle Attribute des Supertupeltyps und können zusätzlich weitere Attribute

aufweisen. Jeder Subtyp kann nur genau einen Supertyp besitzen, deshalb ist eine Mehrfachvererbung nicht möglich. Das Prinzip der Substituierbarkeit wird von Informix für Tupeltypen unterstützt.

Typisierte Tabellen werden angeboten, allerdings sind diese keine Objekttabellen, da sie auf Tupeltypen basieren. Die Zeilen sind demzufolge Tupel und keine Objekte. Die Bildung von Wurzel- und Subtabellen, also von Tabellenhierarchien, ist möglich. Des Weiteren können typisierte Sichten erzeugt werden, die analog zu den typisierten Tabellen Tupelsichten und keine Objektsichten sind. Sichthierarchien kennt Informix nicht.

Abschließend liefert Abbildung 3–5 einen Überblick über das objektrelationale Typsystem von Informix. Man erkennt die vollständige Orthogonalität zwischen den Typen `ROW`, `SET`, `LIST` und `MULTISET`. Dies ist ein großer Vorteil des Informix-Datenbankmanagementsystems. Jedoch gibt es den entscheidenden Nachteil, dass keine Objekttypen angeboten werden. An Stelle der Objekttypen treten bei Informix Tupeltypen, die zwar Attribute aber keine Methoden aufweisen können. Dadurch ist die in der Objektorientierung geforderte Kapselung von Daten und Programmcode nicht möglich. Programmcode muss bei Einsatz von Informix in benutzerdefinierten Routinen abgelegt werden, die allerdings keine Verbindung zu den Tupeltypen aufweisen, in deren Zusammenhang sie eingesetzt werden.

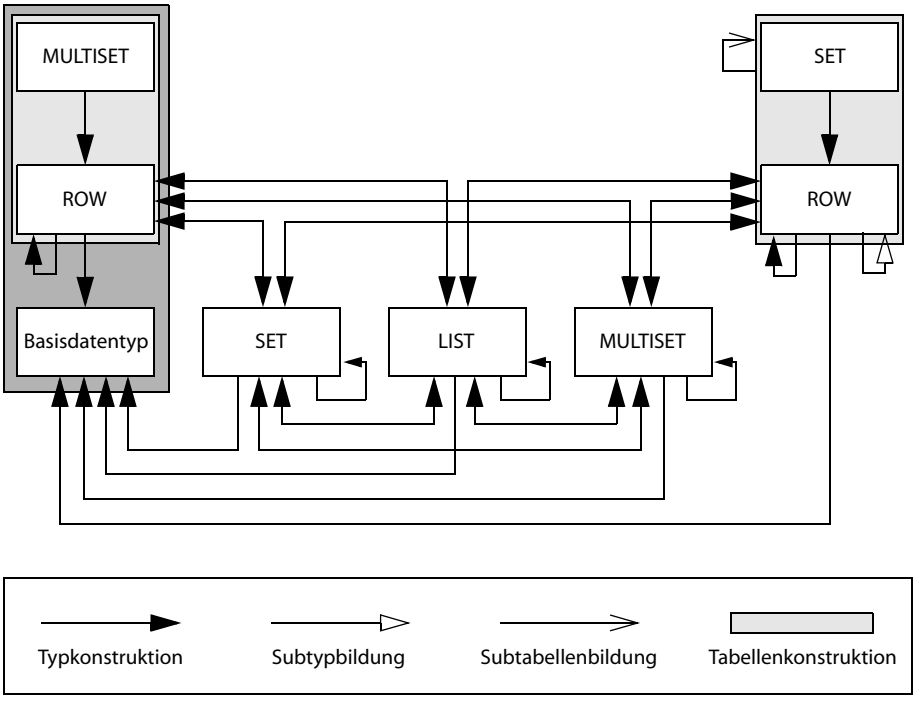


Abb. 3–5 *Objektrelationales Typsystem von Informix (nach [Tür03])*

3.3.4 Vergleich und Zusammenfassung

Die Tabelle 3–3 gibt einen zusammenfassenden Überblick über die Unterstützung der einzelnen benannten und unbenannten objektrelationalen Datentypen im SQL-Standard und in den verschiedenen SQL-Dialekten der Datenbankmanagementsysteme Oracle, DB2 und Informix.

Es fällt auf, dass die Auswahl der angebotenen Datentypen sehr unterschiedlich ist. Kein durch SQL:2003 definierter Datentyp wird in allen drei untersuchten DBMS standardkonform implementiert. Dies macht deutlich, dass man bei Einsatz der objektrelationen Fähigkeiten einer Datenbank stark an das jeweils eingesetzte System gebunden bleibt. Ein Umstieg ist nur mit großem Aufwand möglich, da nicht nur die Syntax der benutzten SQL-Kommandos angepasst werden muss, sondern der gesamte Aufbau der Datenstrukturen muss überarbeitet und auf den vorhandenen Funktionsumfang zugeschnitten werden.

Bei Betrachtung von Tabelle 3–3 erkennt man bei der Datenbank Informix deutlich die weitreichende Unterstützung im Bereich der Tupel- und Kollektionstypen und die fehlende Implementierung von Objekt- und damit auch von Referenztypen. Deshalb ist die Einordnung von Informix als objektrelationale Datenbank nicht ganz korrekt. Eine besser zutreffende Bezeichnung ist erweiterte relationale Datenbank. Bei DB2 sind Objekt- und Referenztypen vorhanden, jedoch fehlt jegliche Unterstützung für Kollektionen und Tupel.

Datentyp	SQL:1999	SQL:2003	Oracle	DB2	Informix
ROW (unbenannt)	X	X			X
ROW (benannt)					X
SET (unbenannt)					X
MULTISET (unbenannt)		X			X
TABLE (benannt)			X		
LIST (unbenannt)					X
ARRAY (unbenannt)	X	X			
ARRAY (benannt)			X		
OBJECT (benannt)	X	X	X	X	
REF (unbenannt)	X	X	X	X	

Tab. 3–3 *Vergleich der Unterstützung objektrelationaler Datentypen*

Bei Oracle liegt eine ausgewogenere Auswahl der einsetzbaren objektrelationalen Typen vor. Wie im SQL-Standard gefordert, können Objekt- und Referenztypen – allerdings mit nicht standardkonformer Syntax – deklariert werden. Arraytypen werden unterstützt, jedoch handelt es sich im Gegensatz zu SQL:2003 um einen benannten Konstruktor. Statt des Multimengentyps muss bei Einsatz von Oracle auf den Typ `TABLE` ausgewichen werden, der im Wesentlichen die gleiche Funktionalität zur Verfügung stellt.

4 Speicherung von XML-Dokumenten

In diesem Kapitel werden Speicherverfahren für XML-Dokumente besprochen. Da XML ein äußerst flexibles Datenformat ist, kann der Aufbau der Dokumente sehr verschieden sein. Dies erlaubt einerseits unterschiedliche Speichervarianten, andererseits ist es schwierig, ein Verfahren zu entwickeln, dass unabhängig vom Aufbau der XML-Dokumente stets eine optimale Speicherung und Wiederherstellung („Publishing") ermöglicht.

Das Kapitel gliedert sich wie folgt: Zunächst wird eine Unterteilung der möglichen Speicherverfahren in drei Haupttypen vorgenommen. Daran schließt sich in den Abschnitten 4.1 bis 4.3 eine Behandlung der jeweiligen Besonderheiten der einzelnen Methoden an, die durch Beispiele ergänzt wird. Der letzte Abschnitt des Kapitels gibt einen Überblick über die SQL-Spracherweiterung SQL/XML, die den Zugriff auf XML-Daten, die in relationalen Datenbanken abgelegt sind, standardisiert.

Kategorisierung der Speicherverfahren

Es lassen sich drei grundsätzliche Speicherungsarten unterscheiden:

- Textbasierte Verfahren
  Hierbei werden komplette XML-Dokumente als Text abgelegt.
- Strukturbasierte Verfahren
  Diese Art der Speicherung untersucht den Aufbau der XML-Dokumente und speichert den Inhalt in Datenbanktabellen, die einen zu der jeweiligen Dokumentstruktur passenden Aufbau besitzen.
- Modellbasierte Verfahren
  Bei modellbasierten Verfahren wird ein Modell gewählt, das den allgemeinen baumartigen Aufbau von XML-Dokumenten widerspiegelt, das aber nicht die spezielle Struktur eines Dokuments für die Speicherung berücksichtigt.

Die beiden zuletzt genannten Verfahren werden von einigen Autoren mit anderen Bezeichnungen versehen. So nennt Lausen in [Lau05] das strukturbasierte Ver-

fahren inhaltsbasiert, wohingegen die modellbasierte Variante als strukturorientiert bezeichnet wird.

4.1 Textbasierte Verfahren

Textbasierte Verfahren speichern vollständige XML-Dokumente als Zeichenkette. Vorteil der textbasierten Speicherung ist, dass das XML-Dokument als Ganzes abgelegt wird und somit unverändert erhalten bleibt. Zur Ablage eines Dokuments ist die Angabe einer Schemadefinition nicht erforderlich. Prinzipiell können auch Fragmente eines XML-Dokuments oder sogar nicht wohlgeformte Dokumente gespeichert werden. Eventuell vorhandene digitale Signaturen bleiben gültig, da die Daten unverändert (bytegenau) wieder ausgelesen werden können.

Diesen Vorteilen steht aber auch eine Reihe von Nachteilen gegenüber. Das Verfahren ist sehr „grobgranular“, das heißt, XML-Dokumente können meist nur komplett geschrieben und wieder vollständig ausgelesen werden. Die direkte Änderung von Teilen eines Dokuments wird im Allgemeinen nicht unterstützt, da es zu Verschiebungen im weiter hinten liegenden Dokumentteil kommt. Der Zugriff auf ein bestimmtes Element eines XML-Dokuments ist nicht direkt möglich. Deshalb bieten diese Verfahren für die meisten Anwendungen, die umfangreiche XML-Dokumente einsetzen, nur eine unzureichende Performanz. Mit Hilfe von Volltextindizierungen kann eine Leistungssteigerung erreicht werden.

4.1.1 Textdateien im Dateisystem

Die einfachste Möglichkeit der textbasierten Speicherung ist die Ablage der XML-Dokumente in Textdateien im Dateisystem eines Fileservers (siehe Abb. 4–1). Dieses Verfahren stellt keine hohen Ansprüche an die Anwendungen, die auf die Daten zugreifen möchten. Der einfache Lese- und Schreibzugriff auf die Dateien des Servers genügt.

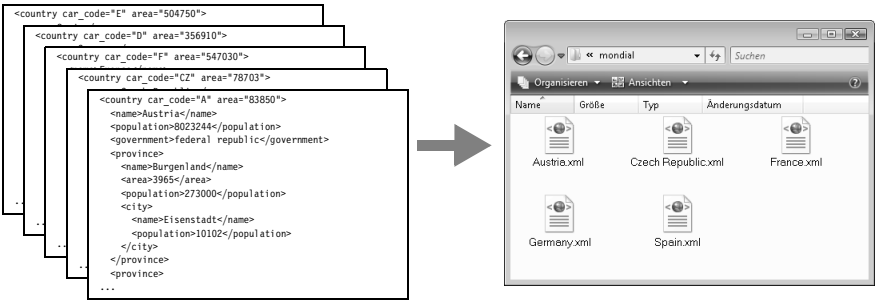


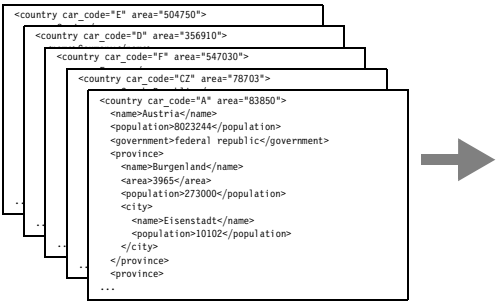
Abb. 4–1 *Textbasierte Speicherung im Dateisystem*

Gravierender Nachteil bei dieser Speicherart ist, dass die üblichen, von Datenbankmanagementsystemen bekannten Funktionalitäten nicht bereitstehen. Es wird keine Unterstützung für die Konsistenzerhaltung bei Mehrbenutzerzugriff, kein Transaktionskonzept und auch keine Hilfe bei der Durchführung von Datensicherungen im laufenden Betrieb angeboten. Unterstützung für die Suche und den Zugriff auf bestimmte Teile der XML-Dokumente ist nur möglich, wenn zusätzliche Hilfsdateien vorgehalten werden, die die Datendateien indizieren. Die Aktualität dieser Dateien muss gewährleistet sein.

Das Verfahren stellt einen Rückschritt gegenüber der heute durch den Einsatz von relationalen Datenbanken erreichten Sicherheit und Stabilität dar. Die textbasierte Speicherung im Dateisystem sollte nur für sehr überschaubare Datenmengen im Einzelnutzerbetrieb eingesetzt werden. Die ACID-Eigenschaften (Atomicity, Consistency, Isolation, Durability) können nicht garantiert werden, Anwendungs- und Implementierungsunabhängigkeit sind nicht gegeben.

4.1.2 LOB-Spalten in Datenbanktabellen

Anspruchsvollere Anwendungen nutzen aus den oben genannten Gründen eine Datenbank, um ein komplettes Dokument in einem Attribut einer Datenbanktabelle als Zeichenkette abzulegen. Dabei können weitere Tabellenattribute verwendet werden, um zusätzliche Informationen zum jeweiligen XML-Dokument zu speichern. Abbildung 4–2 illustriert das Vorgehen.[7]



MONDIAL

ID	XML-Daten
Austria	<country car_code="A" area="83850"> <name>Austria</name> …
Czech Republic	<country car_code="CZ" area="78703"> <name>Czech Republic</name> …
France	<country car_code="F" area="547030"> <name>France</name> …
Germany	<country car_code="D" area="356910"> <name>Germany</name> …
Spain	<country car_code="E" area="504750"> <name>Spain</name> …

Abb. 4–2 *Textbasierte Speicherung in einer Datenbanktabelle*

Als Attributtyp für die XML-Daten muss ein spezieller Typ für große Objekte (LOB, engl. „large object“) verwendet werden, der auch sehr lange Texte aufnehmen kann. In SQL:2003 und den meisten Datenbankmanagementsystemen hat

7. Dieses und die folgenden Beispiele verwenden Daten der geografischen Datenbank „Mondial“ [May99, May10]. Die zugehörige XML-Datei `mondial.xml` wird in Kapitel 6 für die Leistungsuntersuchungen eingesetzt. Aufbau und Inhalt werden dort in Abschnitt 6.2 besprochen.

dieser die Bezeichnung CLOB (Character Large Object). Alternativ kann auch der Datentyp BLOB (Binary Large Object) eingesetzt werden. Tabelle 4–1 zeigt die LOB-Datentypen von Oracle.

Datentyp	Verwendung
CLOB	Lange Zeichenketten, wobei die einzelnen Zeichen durch ein oder mehrere Bytes repräsentiert werden
NCLOB	Wie CLOB, jedoch speziell für den Unicode-Zeichensatz
BLOB	Binäre unstrukturierte Daten ohne Zeichenkettensemantik

Tab. 4–1 *LOB-Datentypen bei Oracle*

Die maximale Größe der LOB-Typen beträgt bei den meisten Systemen mindestens 2 GB. Bei Oracle liegt die Grenze bei 4 GB, seit Version 11g ist mit Hilfe spezieller Parameter eine weitere Vergrößerung möglich.

Das Ablegen und Wiederauslesen der XML-Dokumente aus LOB-Attributen einer relationalen Tabelle kann nur als Ganzes geschehen. Der Zugriff auf Teildokumente, einzelne Elemente oder Attribute ist nicht möglich. Dies ist einer der größten Nachteile der textbasierten Speicherung in relationalen Datenbanken und hat zur Folge, dass XML-Dokumente nicht direkt in der Datenbank geändert werden können. Das vollständige Dokument muss im ungünstigsten Fall bei jeder Modifikation zur Anwendung übertragen, dort verändert und schließlich zurücktransferiert werden. Dies schränkt die Effizienz mit zunehmender Dokumentgröße stark ein.

Für Archiv- und reine Recherchezwecke in dokumentenzentrierten Anwendungen ist die textbasierte Speicherung in relationalen Datenbanken jedoch gut geeignet, da einmal abgelegte XML-Dokumente nicht mehr geändert werden müssen. Dabei sollte ein entsprechender Index zur Volltextsuche zur Verfügung stehen. Von Vorteil ist dabei, dass eine bytegenaue Speicherung möglich ist, sodass zum Beispiel Whitespace-Zeichen[8] in unveränderter Form erhalten bleiben. Dies kann zu Beweiszwecken oder aufgrund des Einsatzes digitaler Signaturen erforderlich sein.

Bei der Verwendung von LOB-Typen muss außerdem beachtet werden, dass diese im Gegensatz zu normalen Zeichenkettentypen (zum Beispiel VARCHAR2 bei Oracle mit maximal 4000 Zeichen) nicht den direkten Vergleich zweier Zeichenketten oder die Suche nach einem Teilstring mit Hilfe des LIKE-Operators erlauben.

Um die zuvor dargestellten Unzulänglichkeiten der LOB-Typen auszugleichen, bieten einige Datenbanken spezielle Zusatzfunktionalitäten an. So sind neben allgemeinen Funktionen zur Behandlung von Zeichenketten häufig Proze-

8. Zu den Whitespace-Zeichen gehören nach dem XML-Standard das Leer- und das Tabulatorzeichen, der Wagenrücklauf (CR) und der Zeilenvorschub (LF).

duren verfügbar, mit denen die abzulegenden XML-Dokumente vor der Speicherung auf Wohlgeformtheit oder auch auf Gültigkeit bezüglich eines angegebenen Schemas getestet werden können. Zusätzlich ist es möglich, einzelne Bestandteile eines XML-Dokuments bereits auf Datenbankseite zu extrahieren oder zu ändern, ohne das komplette Dokument zunächst zum Client zu übertragen. Diese Zugriffe werden beschleunigt, sofern eine textuelle Indizierung der XML-Dokumente vorhanden ist. [Dop07] untersucht, wie derartige Indexe möglichst platzsparend gestaltet und an die speziellen Eigenschaften von XML-Dokumenten angepasst werden können.

Am Beispiel der Oracle-Erweiterung OracleText sollen im Folgenden einige derartige Funktionalitäten vorgestellt werden. OracleText unterstützt die schnelle indexbasierte Textsuche, bei der auch XPath-Ausdrücke angegeben werden können. Zur Nutzung der Funktionalität müssen insbesondere die notwendigen Indexe für die entsprechenden Zeichenkettenattribute vorab definiert werden.

Bei OracleText steht die Funktion `CONTAINS` im Mittelpunkt. Sie kann in der `WHERE`-Klausel einer SQL-`SELECT`-Abfrage eingesetzt werden, um nach einem bestimmten Muster in einem Textattribut zu suchen. `CONTAINS` liefert einen so genannten Score-Wert zwischen 0 und 100, der die Relevanz des Vorkommens des gesuchten Musters repräsentiert. Dabei bedeutet 0, dass das Muster nicht gefunden wurde, wohingegen 100 für einen exakten Treffer steht. Um auch komplexere Suchen zu ermöglichen, können bei der Angabe des Musters verschiedene Operatoren genutzt werden, die hier kurz vorgestellt werden sollen.

- Zu den logischen Operatoren gehören unter anderem die Und-, Oder- und Nicht-Verknüpfungen, mit denen die im gesuchten Text vorkommenden Wörter festgelegt werden können.
- Mit Hilfe der Funktion `NEAR` können Wörter im Text gesucht werden, die in einem anzugebenden Maximalabstand gemeinsam auftreten müssen.
- Die linguistischen Funktionen `stem` und `soundex` helfen Stammformen zu finden und ähnlich klingende Wörter zu ermitteln. Ihr Einsatz erlaubt im Vergleich zu den üblichen Wildcard-Zeichen deutlich mächtigere Suchmöglichkeiten.
- Bei der Suche kann ein Schwellenwert festgelegt werden, den die gefundenen Texte in Bezug auf die Ähnlichkeit zu dem gesuchten Ausdruck nicht unterschreiten dürfen.
- Ein Thesaurus und zugehörige Operatoren stehen zur Verfügung, um auch sinnverwandte Wörter in die Suche einzubeziehen.

Speziell für den Umgang mit XML-Dokumenten stehen in OracleText die folgenden Operatoren zur Verfügung:

- Der `WITHIN`-Operator erlaubt es, die Suche auf einen Teilbereich einzuschränken. Dies kann ein Satz, ein Absatz, ein XML-Element (zwischen Start- und End-Tag) oder auch ein Attributwert eines XML-Elements sein.
- Die `HASPATH`-Funktion wird genutzt, um zu testen, ob in einem XML-Dokument ein über einen XPath-Ausdruck spezifiziertes Element existiert. `HASPATH` ermöglicht ebenfalls die Prüfung, ob das angegebene Element einen bestimmten Wert als Textinhalt aufweist. Hierbei ist nur der Test auf Gleichheit und Ungleichheit möglich.
- Der `INPATH`-Operator erlaubt ähnlich wie `WITHIN` die Einschränkung des Suchbereichs. Die Angabe erfolgt hier über einen XPath-Ausdruck. Dadurch ist eine stärkere Eingrenzung möglich.

Durch diese auf XML zugeschnittenen Zusatzfunktionalitäten ist der Umgang mit XML-Dokumenten im Rahmen von OracleText leichter möglich. OracleText bleibt jedoch in erster Linie eine Werkzeugsammlung zur Suche in großen Texten, die hauptsächlich für Information-Retrieval-Anwendungen eingesetzt wird.

4.2 Strukturbasierte Verfahren

Strukturbasierte Verfahren können eingesetzt werden, um eine Menge von XML-Dokumenten zu speichern, die der gleichen Schemadefinition genügen. Dazu wird aus der in der DTD oder dem XML-Schema definierten Dokumentstruktur ein Datenmodell entworfen, unter das sich alle Dokumentinstanzen subsumieren lassen. Aus der Strukturbeschreibung ist unter anderem bekannt, an welchen Stellen atomare Werte und wo Listen oder Mengen von Werten zu speichern sind.

Das Datenmodell zielt im Allgemeinen darauf ab, die Daten in einer relationalen oder objektrelationalen Datenbank abzulegen. Dazu wird die Struktur der XML-Dokumente in einem relationalen Datenmodell durch eine Menge von Entitäten, Relationen und Attributen repräsentiert. Die Datenbankobjekte können entweder manuell oder automatisch durch einen Abbildungsalgorithmus erzeugt werden. Beim Zugriff auf die Daten kann die Umsetzung zwischen XML-Struktur und internem Datenmodell durch ein Modul des Datenbankmanagementsystems oder durch ein separates Middleware-System erfolgen.

Strukturbasierte Verfahren eignen sich insbesondere für datenzentrierte Dokumente. Sie bieten die Möglichkeit, den je nach Anwendung unterschiedlichen Aufbau der Daten optimal zu berücksichtigen. Dabei können zusätzlich die Zugriffshäufigkeiten auf einzelne Werte einbezogen werden. Dadurch erhält man ein Datenbankschema, das möglichst effiziente Zugriffe für die jeweilige Applikation erlaubt.

Die spezielle Anpassung an den jeweiligen Einsatz ist andererseits auch einer der größten Nachteile der strukturbasierten Speicherung. Für jeden Anwendungsfall muss ein geeignetes Datenbankschema entwickelt werden. Hierbei kön-

nen zwar allgemeingültige Regeln zu Hilfe genommen werden, wie die Umsetzung erfolgen sollte, jedoch ist häufig eine zusätzliche manuelle Optimierung erforderlich. Dabei müssen meist auch jeweils passende Algorithmen entworfen werden, mit denen sich die ursprünglichen XML-Dokumente aus den gespeicherten Daten rekonstruieren lassen.

Ein weiteres Problem tritt auf, falls Veränderungen an der Schemadefinition der XML-Dokumente vorgenommen werden („Schemaevolution", [KT05, XMY10]). Erlaubt man beispielsweise für einen Datenwert zukünftig an Stelle eines atomaren Eintrags einen Kollektionstyp, so kann dies weitreichende Veränderungen an dem Datenbankschema sowie Konvertierungen bereits vorhandener Daten nach sich ziehen [KGH10, GLQ11].

Die strukturbasierte Speicherung von XML-Dokumenten kann auf verschiedenen Wegen umgesetzt werden. In dem sich anschließenden Abschnitt 4.2.1 wird ein Verfahren vorgestellt, das ausschließlich herkömmliche relationale Tabellen mit atomaren Attributen verwendet. Die hierarchische Baumstruktur wird dazu vollständig „flachgeklopft" und in Einzelteile zerlegt. Aus diesem Grund wird das Vorgehen meist auch als Schreddern bezeichnet.

Alternativ kann zumindest ein Teil der Hierarchie erhalten bleiben, indem geschachtelte Tabellen benutzt werden. Mit den so genannten NF^2-Tabellen können einige der XML-Schemastrukturen gut umgesetzt werden. Dieses Verfahren wird in Abschnitt 4.2.2 dargestellt.

4.2.1 Schreddern

Die Bezeichnung Schreddern für dieses Verfahren rührt daher, dass die in einem XML-Dokument enthaltenen Daten zerlegt und anschließend in kleinen „Schnipseln" in mehreren relationalen Datenbanktabellen gespeichert werden. Das Schreddern von XML-Dokumenten beginnt mit einem Schritt, in dem die nötigen Strukturen in der Datenbank angelegt werden. Dieser Prozess kann entweder manuell erfolgen oder durch einen von der Datenbank bereitgestellten Automatismus. Im Idealfall werden damit auch die Voraussetzungen geschaffen, um die Daten nach der Zerlegung wieder zusammensetzen und als XML-Dokument ausgeben zu können. [KCD03] gibt eine Einführung in die Technik des Schredderns.

Beim Schreddern werden häufig Zusatzangaben in die Datenbanktabellen eingefügt, die in den XML-Dokumenten nicht enthalten sind. Dies kann zum Beispiel eine ID sein, die zum korrekten Wiederzusammensetzen des XML-Dokuments benötigt wird. Schon im XML-Dokument vorhandene ID-Attribute zu verwenden, ist meist nicht machbar, da diese – wenn sie denn überhaupt vorhanden sind – nur innerhalb eines einzelnen Dokuments eindeutig sind, in den Tabellen jedoch die Daten vieler XML-Dokumente abgelegt werden sollen.

Falls bei der jeweiligen Applikation die Reihenfolge, in der die Unterelemente eines Elements angeordnet sind, von Bedeutung ist, so muss beim Prozess des

Schredderns eine Ordnungszahl erzeugt werden. Diese Ordnungszahl wird dann als Attribut einer Datenbanktabelle abgelegt und beim Wiederzusammensetzen des XML-Dokuments verwendet.[9] Werden XML-Dokumente in der Datenbank geändert, so muss darauf geachtet werden, dass die Ordnungszahlen entsprechend angepasst werden. Bei großen Dokumenten kann dies leicht zu Performanzproblemen führen.

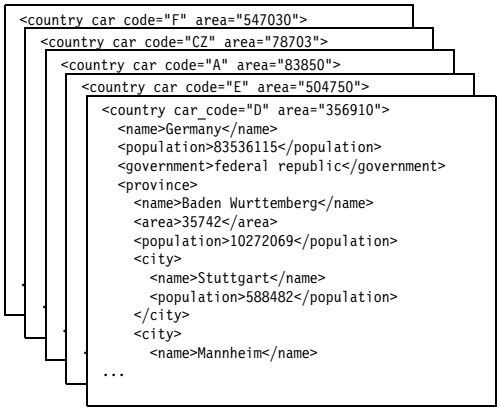
Beim Erstellen des Datenbankschemas muss zwischen zwei grundsätzlichen Entwurfsalternativen unterschieden werden: die Aufteilung der Informationen in getrennte Tabellen oder die Zusammenfassung der Daten mehrerer Elemente in einer Tabelle. Im Allgemeinen beginnt dieser Prozess mit dem Wurzelelement des XML-Dokuments, für das eine Tabelle erzeugt wird. Anschließend muss für jedes mögliche Kindelement die Entscheidung getroffen werden, ob die Angaben des Kindelements ebenfalls in diese Tabelle aufgenommen werden oder ob eine neue Tabelle begonnen wird. Im letzteren Fall muss in den XML-Daten ein Schlüssel gefunden werden, mit dem ein Join zwischen den Tabellen möglich ist. Falls kein derartiger Schlüssel verfügbar ist, muss eine entsprechende ID, wie oben beschrieben, künstlich erzeugt werden.

Die einfachste Regel für die Entscheidung, neue oder bisherige Tabelle, lautet: Eine neue Tabelle wird immer dann begonnen, wenn ein Element mehr als einmal unter einem Elternelement vorhanden sein kann. Falls ein Element genau einmal unter einem Elternelement auftritt, wird die bisherige Tabelle benutzt. Bei optionalen Elementen und an Stellen, an denen Varianten erlaubt sind, sind beide Vorgehensweisen denkbar. Fehlende Elemente werden jeweils durch einen Null-Wert dargestellt. Die Aufnahme der Angaben in die bisherige Tabelle spart beim Zusammensetzen einen Join, hat aber den Nachteil, dass einige Spalten der Tabelle unter Umständen nur dünn belegt sind, was wiederum zu größerem Speicherplatzbedarf führt. Deshalb muss die Entscheidung von der jeweiligen Verwendung abhängig gemacht werden.

Nachdem der Entwurfsprozess von der Wurzel bis zu allen Blattelementen durchgeführt wurde, kann sich eine Konsolidierungsphase anschließen. Dabei werden Tabellen zusammengefasst, die die gleichen Informationen speichern, aber zu Elementen an verschiedenen Positionen des XML-Dokuments gehören. Beim Zusammenfassen muss darauf geachtet werden, dass ein korrekter Join mit den Tabellen, die die verschiedenen Elternelemente aufnehmen, möglich ist. In [TS06, S. 486ff.] findet sich eine ausführliche Darstellung des kompletten Umsetzungsprozesses. Die Autoren weisen darauf hin, dass dabei auch in der jeweiligen Datenbank verfügbare, objektrelationale Datentypen eingesetzt werden können. [Bou05a] zeigt Beispiele, wie sich verschiedene DTD-Strukturen in relationale

9. Die Eigenschaft, dass bei einem XML-Dokument die Reihenfolge aller Elemente und auch alle weiteren Informationen bis auf irrelevante Whitespaces erhalten bleiben, wird von Oracle als „DOM Fidelity“ bezeichnet.

Tabellen umsetzen lassen. Barbaso et al. [BFM05] konzentrieren sich auf die Frage, wie sich informationserhaltende Abbildungsschemata algorithmisch erzeugen lassen und definieren äquivalenzbewahrende Transformationen auf diesen. [CCS05] weist darauf hin, dass bei der Entwicklung des Abbildungsschemas neben dem logischen Entwurf auch gleichzeitig die physikalische Realisierung auf dem eingesetzten DBMS beachtet werden sollte.



COUNTRY

ID	name	population	government	@car_code	@area
1001	Germany	83536115	federal republic	D	356910
1002	Spain	39181114	parliamentary monarchy	E	504750
1003	Austria	8023244	federal republic	A	83850
…	…	…	…	…	…

PROVINCE

ID	name	area	population	countryID
2001	Baden Wurttemberg	35742	10272069	1001
2002	Bayern	70546	11921944	1001
2003	Berlin	889	3472009	1001
2004	Brandenburg	29480	2536747	1001
2005	Bremen	404	680000	1001
2006	Hamburg	755	1705872	1001
2007	Hessen	21115	5980693	1001
…	…	…	…	…
2017	Andalusia	87600	7053043	1002
2018	Aragon	47720	1183576	1002
2019	Asturias	10604	1083388	1002
…	…	…	…	…

CITY

ID	name	population	provinceID
3001	Stuttgart	588482	2001
3002	Mannheim	316223	2001
3003	Karlsruhe	277011	2001
3004	Freiburg	198496	2001
3005	Heidelberg	138964	2001
3006	Heilbronn	122253	2001
3007	Pforzheim	117960	2001
3008	Ulm	115123	2001
3009	Reutlingen	107782	2001
3010	Munich	1244676	2002
3011	Nurnberg	495845	2002
…	…	…	…

Abb. 4–3 *Schreddern eines XML-Dokuments*

Nicht in allen Fällen ist eine problemlose Erzeugung eines passenden Datenbankschemas möglich. Eine Schwierigkeit stellt die exakte Abbildung der in XML-Schema möglichen Datentypen auf die in der jeweiligen relationen Datenbank vorhandenen Typen dar. Hier sind insbesondere die in XML erlaubten Zeichenketten beliebiger Länge zu nennen. Besondere Beachtung erfordert auch die Abbildung von variablen Strukturen, die unter anderem bei Verwendung des XML-Schema-Elements `choice` auftreten können. Außerdem nicht leicht umzusetzen sind polymorphe Folgen (zum Beispiel Folgen, die verschiedene sich wiederholende Unterfolgen enthalten können), gemischter Inhalt (Text und Unterelemente in beliebiger Reihenfolge), rekursive Schemata oder Elemente, die beliebige Unterelemente aufweisen können (`any`).

Auch wenn der Umgang mit DTDs aufgrund der begrenzteren Möglichkeiten im Vergleich zu XML-Schema in den meisten Fällen leichter ist, treten dennoch einige der hier angesprochenen Schwierigkeiten auch im Zusammenhang mit DTDs auf. In manchen Fällen erscheint ein hybrides Verfahren, das grundsätzlich eine strukturbasierte Herangehensweise nutzt, aber dennoch einige Teile textbasiert ablegt, eine gute Lösung.

Zum Abschluss dieses Abschnitts soll noch ein Beispiel für das Schreddern von XML-Dokumenten gegeben werden. In Abbildung 4–3 werden für die Speicherung der `country`-XML-Dokumente drei relationale Tabellen verwendet. Jede Tabelle besitzt ein ID-Primärschlüsselattribut, das für die Verknüpfung der Tabellen über Fremdschlüssel benötigt wird. Auf Ordnungszahlen wird verzichtet, da die Reihenfolge der Unterelemente hier ohne Bedeutung ist. Das heißt jedoch auch, dass das Datenbankschema nicht die exakte Rekonstruktion der XML-Dokumente erlaubt.

[STH99] stellt ein ähnliches Verfahren vor, das zu einer gegebenen DTD ein relationales Schema erzeugt. Dazu werden komplexe Elementdefinitionen vereinfacht. Die Autoren weisen darauf hin, dass jedes in einer DTD deklarierte Element Wurzel eines auf diesem Schema basierenden XML-Dokuments sein kann. Um die Anzahl der nötigen relationalen Tabellen zu reduzieren, werden Unterelemente, die maximal einmal innerhalb eines Elements verwendet werden können, in die Tabelle des zugehörigen Elternelements integriert („inlining“). Die Verknüpfung der Einträge erfolgt über IDs und Fremdschlüsselbeziehungen. [LYW05] bescheinigt einer Erweiterung dieses Ansatzes, die zusätzlich eine Pfadtabelle verwendet [ZLZ01], eine hohe Verarbeitungsleistung. Um das Datenbankschema zu optimieren, können repräsentative Beispieldaten und das zu erwartende Abfrageaufkommen berücksichtigt werden [Kul10]. Auch kann eine Klassifizierung der abzulegenden XML-Dokumente nach ihrer Struktur und die Erzeugung von eigenen Relationen für jede so gebildete Dokumentklasse vorteilhaft sein, da die zum Wiederzusammensetzen der Dokumente erforderlichen Joins auf kleineren Tabellen ausgeführt werden können [LCM04, AMN11].

Mit der Umsetzung von XPath-Ausdrücken in SQL-Abfragen, die auf einem durch Schreddern entstandenen Datenbankschema arbeiten, beschäftigt sich [FYL09]. Dabei werden insbesondere auch rekursiv aufgebaute DTDs betrachtet. [ACL07] untersucht mehrwertige Abbildungsverfahren, bei denen ein XML-Elementtyp auf eine von mehreren möglichen Relationen abgebildet wird.

4.2.2 NF²-Tabellen

Wegner und Ahmad zeigen in [WA01], wie sich hierarchische Dokumente, insbesondere XML-Dokumente, auf Tabellen des geschachtelten relationalen Datenmodells abbilden lassen. Die Tabellen sollen dabei der so genannten Non-First-Normal-Form (NF²) genügen [SS86]. Die Arbeiten am NF²-Datenmodell begannen Mitte der 1980er Jahre. Forschungsgegenstände waren die Entwicklung zugehöriger komplexer Abfragesprachen, die Definition von Normalformen sowie Untersuchungen zu Nest- und Unnest-Operationen, mit denen unter bestimmten Voraussetzungen geschachtelte Tabellen in herkömmliche, flache relationale Tabellen und umgekehrt transformiert werden können.

Einige Wissenschaftler lehnten das NF²-Datenmodell zunächst ab, da sie darin einen Rückschritt zum hierarchischen Modell sahen. Um die Nähe zum relationalen Modell zu betonen, entstand am wissenschaftlichen Zentrum von IBM in Heidelberg für den Prototyp AIM-P die an SQL angelehnte Sprache HDBL [LKD88, Pis89], weitere Abfragesprachen finden sich im Werk von Roth, Korth und Silberschatz [RKS88] sowie in [Rot86]. Seit SQL:99 existiert auch eine standardisierte, deklarative Abfragesprache für das NF²-Modell, die die zugehörigen Typen (Mengen, Listen und Tupel) und deren beliebige Schachtelung unterstützt. Ein eher navigierendes Vorgehen, ähnlich dem Cursor im hierarchischen Modell, verfolgten Wegner et al. im visuellen Editor ESCHER [KTW90].[10]

Allerdings gibt es nicht nur genau eine Möglichkeit, die XML-Baumstruktur in das NF²-Datenmodell umzusetzen. Das prinzipielle Vorgehen ist jedoch stets gleich: die hierarchische Elementschachtelung der XML-Dokumente wird in hierarchisch geschachtelte Typen des NF²-Modells transformiert. Tabelle 4–2 zeigt hierzu ein Beispiel. Es basiert auf den Daten des Beispiels aus Abbildung 4–3. Falls es bei der jeweiligen Anwendung von Bedeutung ist, dass die Reihenfolge der Tupel erhalten bleibt, so müssen die Tupel im NF²-Modell nicht als Menge (Symbol: {}), sondern als Liste (Symbol: <>) abgelegt werden. Eine zusätzliche Ordnungszahl zur Sicherung der Position ist dann nicht erforderlich.

10. Einige Autoren unterscheiden zwischen NF²- und eNF²-Modell. Das ursprüngliche NF²-Modell enthielt nur geschachtelte Relationen. Erst später wurden zusätzliche Typkonstruktoren und deren freie Kombinierbarkeit eingeführt [DKA86]. Diese Erweiterung wurde als eNF²-Modell („extended NF²“) bezeichnet. In dieser Arbeit wird keine Unterscheidung der beiden Modelle vorgenommen und unter dem Begriff NF² ein Modell verstanden, das insbesondere auch orthogonal einsetzbare Listen- und Mengentypen umfasst.

{}COUNTRY									
					{}PROVINCE				
								{}CITY	
name	population	government	@car_code	@area	name	area	population	name	population
Germany	83536115	federal republic	D	356910	Baden Wurttemberg	35742	10272069	Stuttgart	588482
								Mannheim	316223
								Karlsruhe	277011
								Freiburg	198496
								Heidelberg	138964
								Heilbronn	122253
								Pforzheim	117960
								Ulm	115123
								Reutlingen	107782
					Bayern	70546	11921944	Munich	1244676
								Nurnberg	495845
								…	
					…				
…									

Tab. 4–2 *NF²-Darstellung eines XML-Dokuments*

Im Falle von optionalen Attributen im XML-Dokument wird man an den entsprechenden Stellen bei fehlenden Attributwerten Null-Werte speichern. Gleiches gilt beim Auftreten von Varianten, bei denen im XML-Dokument einer von mehreren möglichen Elementtypen stehen kann. Die nicht benötigten Attribute der NF²-Tabelle werden mit Null-Werten gefüllt. Die Verwendung von Null-Werten an Stelle von leeren Zeichenketten oder Repräsentationen der leeren Menge ist dabei zwingend erforderlich, da ansonsten nicht zwischen nicht vorhandenen und leeren Elementen unterschieden werden kann. Sofern vom eingesetzten DBMS unterstützt, können für die Modellierung auch variante Strukturen eingesetzt werden, siehe hierzu [KD95].

Beachtet werden muss auch hier, dass nicht alle in XML-Schema möglichen Datentypen exakt auf SQL-Typen abgebildet werden können. Einige Schwierigkeiten lassen sich durch den Einsatz von Check-Klauseln oder Constraints, die die notwendigen Einschränkungen der Basistypen vornehmen, ausgleichen. Konflikte können ebenfalls aufgrund der im XML-Dokument verwendeten Bezeichner entstehen. Hier kann es Komplikationen wegen des Einsatzes bestimmter Sonderzeichen oder auch aufgrund der in XML möglichen beliebig langen Attribut- und Elementnamen geben. Die meisten Datenbanken erlauben nur eine begrenzte Zeichenzahl für Bezeichner.

Die Speicherung von Elementen mit so genanntem „mixed content“, bei denen Unterelemente und Textinhalt gemischt auftreten können, ist nur mit erhöhtem Aufwand möglich. Es muss sichergestellt werden, dass die Reihenfolge der einzelnen Textfragmente und Elemente gewahrt wird. Eine Möglichkeit ist die Verwendung einer Liste von Tupeln, wobei die Tupel aus einer Komponente für den Textinhalt und aus je einer Komponente für jeden möglichen Elementtyp bestehen.[11] Zur Speicherung wird in jedem Tupel nur eine Komponente belegt, alle anderen enthalten Null-Werte.[12] Abbildung 4–4 verdeutlicht das Vorgehen an einem Beispiel, in dem ein Element mit gemischtem Inhalt auf eine Liste von Tupeln abgebildet wird.[13] Je nach Anwendung kann als Alternative auch der gesamte gemischte Inhalt als eine einzelne (atomare) Zeichenkette abgelegt werden. Nachteil ist dann jedoch, dass die Zeichenkette geparst werden muss, falls einzelne Teile bearbeitet werden sollen.

Ein weiteres Problem bei der Abbildung stellen rekursive Datendefinitionen dar. Diese lassen sich meist nicht direkt in NF²-Tabellen umsetzen. Als Alternative kann auf die Verwendung von Primär- und Fremdschlüsseln bzw. Referenzen

11. Die Komponente mit dem Textinhalt wurde hier mit dem Bezeichner `TEXT` versehen. In der Praxis wird man einen anderen, in XML nicht als Elementname zugelassenen Bezeichner verwenden müssen, um Konflikte mit gleichnamigen Elementen zu verhindern.
12. Der Null-Wert wird hier durch das Literal – dargestellt.
13. Das hier gezeigte Beispiel verzichtet auf eine exakte Umsetzung der im Text des XML-Dokuments enthaltenen Whitespaces. Bei Bedarf können diese unverändert in die NF²-Tabelle übernommen werden.

bei objektrelationalen Datenbanken ausgewichen werden. Da hierbei jedoch die Schachtelung aufgegeben und stattdessen eine flache Struktur eingesetzt wird, stellt dies einen Bruch mit dem eigentlichen hierarchischen Speicherkonzept dar.

```
<description>
  The <u>MONDIAL database</u> has been compiled
  from the <i>CIA World Factbook</i> and other
  geographical Web data sources.
</description>
```

<>description		
TEXT	**u**	**i**
The	—	—
—	MONDIAL database	—
has been compiled from the	—	—
—	—	CIA World Factbook
and other geographical Web data sources.	—	—

Abb. 4–4 *NF²-Darstellung von gemischtem Inhalt*

Zu beachten sind auch hier die Schwierigkeiten, die durch den in Schemadefinitionen erlaubten Datentyp `any` entstehen können. Da Elemente, die diesen Typ besitzen, als Inhalt beliebige Elemente oder Text aufweisen können, genügt die Schemadefinition nicht, um die Struktur der für die Abbildung nötigen NF²-Tabelle zu kennen. Die erforderlichen Typdefinitionen sind von der jeweiligen Instanz des XML-Dokuments abhängig. Einen Ausweg stellt wiederum der Einsatz von atomaren (unstrukturierten) Typen dar, die die Dokumentteile aufnehmen, deren Struktur variabel ist.

Zuletzt sei auf die Möglichkeit von Oracle und anderer DBMS hingewiesen, automatisch eine Abbildung zwischen XML-Dokumenten und Datenbanktabellen zu erzeugen, die der zuvor beschriebenen NF²-Darstellung ähnelt.[14] Hierzu ist es wie bei allen strukturbasierten Verfahren erforderlich, dass die Schemadefinition der abzulegenden Dokumente bekannt ist. Das Schema wird bei der Datenbank zunächst registriert. Dieser Prozess erzeugt automatisch passende, aufeinander aufbauende Objektdefinitionen sowie eine objektrelationale Tabelle, die die XML-Dokumente als Objekte aufnehmen kann. Die Umsetzung zwischen den Datentypen und Bezeichnern von XML und SQL kann ebenfalls ohne Benut-

14. In Abschnitt 6.5 wird dieses Umsetzungsverfahren in Verbindung mit der Oracle-Datenbank eingesetzt, um Vergleichsmessungen durchzuführen.

zereingriff erfolgen. Es besteht jedoch die Möglichkeit, durch entsprechende Anmerkungen im zu registrierenden XML-Schema in diese Standardzuordnungen manuell einzugreifen.

Da dem Datenbankmanagementsystem das Schema, dem die XML-Dokumente genügen, bekannt ist, können bei der Bearbeitung von Anfragen passende Optimierungen vorgenommen werden, um die Verarbeitungsleistung zu steigern. Eine automatische Umsetzung von XQuery-Anweisungen in entsprechende SQL-Abfragen ist möglich. Als Nachteil dieses Verfahrens entstehen durch die maschinelle Erzeugung häufig große Datenbankschemata, deren zugehörige Tabellen nur schwach gefüllt sind und viele Null-Werte enthalten. [LN02] gibt algebraische Operationen an, mit denen XML-Dokumente, die in NF^2-Relationen gespeichert sind, abgefragt und transformiert werden können.

4.3 Modellbasierte Verfahren

Modellbasierte Verfahren nutzen die XML-Baumstruktur aus und unterstützen deshalb gut den navigierenden Zugriff auf die Dokumente. Für die Elemente, Attribute und sonstigen Teile eines XML-Dokuments sowie die Verknüpfungen zwischen ihnen wird ein Datenbankschema angelegt, das die jeweiligen Inhalte aufnimmt. Das Schema ist dabei unabhängig von der konkreten Anwendung und den Schemadefinitionen, die den Dokumenten zugrunde liegen. Es werden nur die generellen Eigenschaften von wohlgeformten XML-Dokumenten vorausgesetzt. Abbildung 4–5 zeigt die Baumdarstellung eines XML-Dokuments. Element- und Attributknoten sind als Rechtecke, Textknoten als Ellipsen wiedergegeben.

4.3.1 Kantentabelle

Häufig verwenden die modellbasierten Verfahren eine so genannte Kantentabelle, die die Baumstruktur abbildet [FK99a, FK99b]. Dabei muss beachtet werden, dass XML-Dokumente geordnete Bäume sind und somit die Reihenfolge der Knoten festgehalten werden muss. Dies kann geschehen, indem alle Knoten durch Nummerierung in Dokumentenordnung mit einer ID versehen werden und für jeden Knoten im Baum die ID des direkten Vorfahren, also des Elternknotens, sowie eine Ordnungszahl gespeichert werden. Tabelle 4–3 zeigt eine derartige Kantentabelle, deren Inhalt auf dem Beispiel aus Abbildung 4–5 basiert.

Neben den Angaben `KnotenID`, `VorfahrID` und `Ordnung` muss zusätzlich für jeden Knoten der Name sowie der Typ festgehalten werden. Es muss erkennbar sein, ob es sich um einen Element-, Attributknoten oder sonstigen Bestandteil eines XML-Dokuments handelt. Um die Unterscheidung einfach zu halten, kann – wie auch in XPath-Ausdrücken – das @-Zeichen den Attributnamen vorangestellt werden. Bei Attributknoten kann auf die Speicherung der Ordnungszahl

verzichtet werden, weil nach XML-Standard keine Reihenfolge zwischen Attributen definiert ist. Das Attribut `DocID` ermöglicht es, falls mehr als ein Dokument in der Tabelle abgespeichert wird, die einzelnen Einträge den verschiedenen Dokumenten zuzuordnen.

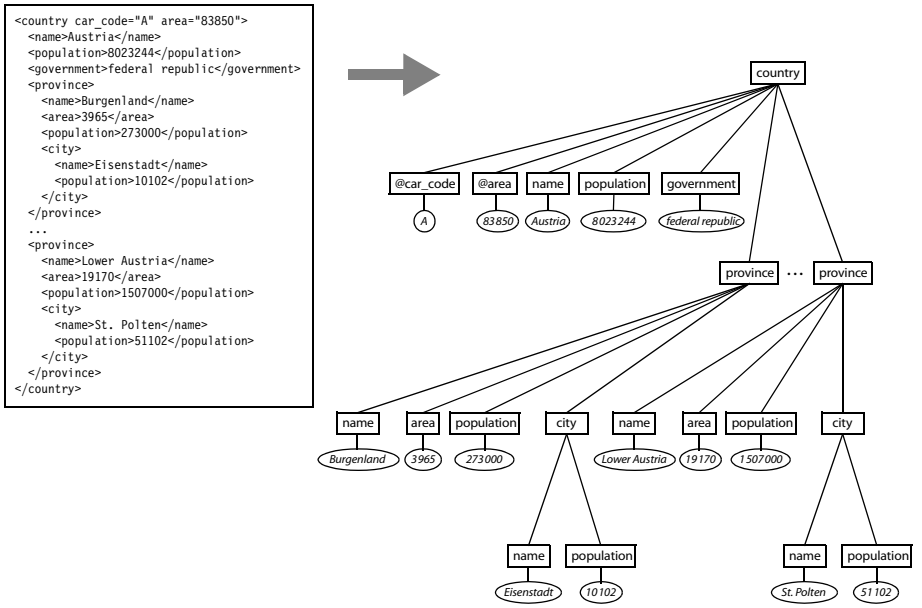


Abb. 4–5 *Darstellung eines XML-Dokuments in Baumstruktur*

Die in Abbildung 4–5 als Ellipsen dargestellten Textknoten werden nicht als eigenständige Knoten in der Kantentabelle aufgelistet. Um Einträge zu sparen, wird der Textinhalt eines Elements direkt in dem Attribut `Wert` des zugehörigen Elementknoteneintrags abgelegt.

Diese Strategie kann jedoch bei Elementen mit gemischtem Inhalt nicht angewandt werden, da ihr Textinhalt aus mehreren separaten Teilen besteht, die nicht einfach zusammengefasst werden können. Hier müssen die einzelnen Textteile und natürlich auch die eingestreuten Elemente als eigenständige Textknoteneinträge mit zugehöriger Ordnungszahl in die Kantentabelle aufgenommen werden. Dies wird in Abbildung 4–6 anhand eines Beispiels demonstriert.[15]

Bedeutendster Vorteil der modellbasierten Verfahren ist, dass im Gegensatz zum strukturbasierten Ansatz auch XML-Dokumente abgelegt werden können, deren Schemadefinition nicht bekannt ist und deren Aufbau nicht einheitlich ist.

Andererseits treten auch eine Reihe von Nachteilen auf. Da sämtliche Informationen in nur einer Tabelle gespeichert sind, ist eine schnelle, effiziente Auswertung von XPath-Ausdrücken ohne Zusatzinformationen nicht möglich. Ein

15. Auch in diesem Beispiel wird auf eine exakte Umsetzung der Whitespaces verzichtet.

weiteres Problem liegt in den verschiedenen Datentypen, die innerhalb eines XML-Dokuments auftreten können. Alle im XML-Dokument vorkommenden Datentypen werden auf nur einen Attributtyp der Kantentabelle abgebildet. Dies führt zu Umsetzungsproblemen sowie zum Verlust der ursprünglichen Typinformationen. Die Schwierigkeiten können durch die Verwendung von separaten Hilfstabellen oder zusätzlichen Tabellenspalten umgangen werden.

DocID	KnotenID	VorfahrID	Ordnung	Knotenname	Wert
1802	1	—	—	country	—
1802	2	1	—	@car_code	A
1802	3	1	—	@area	83850
1802	4	1	1	name	Austria
1802	5	1	2	population	8023244
1802	6	1	3	government	federal republic
1802	7	1	4	province	—
1802	8	7	1	name	Burgenland
1802	9	7	2	area	3965
1802	10	7	3	population	273000
1802	11	7	4	city	—
1802	12	11	1	name	Eisenstadt
1802	13	11	2	population	10102
…	…	…	…	…	…
1802	63	1	12	province	—
1802	64	63	1	name	Lower Austria
1802	65	63	2	area	19170
1802	66	63	3	population	1507000
1802	67	63	4	city	—
1802	68	67	1	name	St. Polten
1802	69	67	2	population	51102

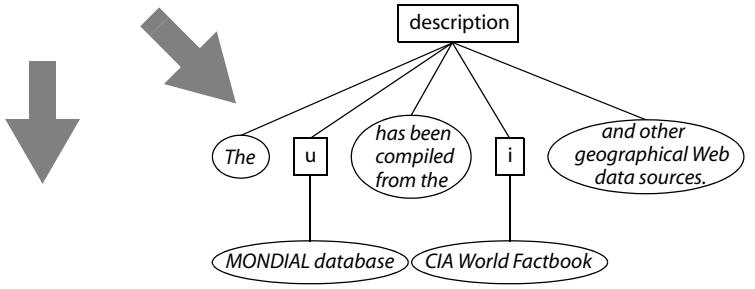
Tab. 4–3 *Kantentabelle*

Da alle zu speichernden XML-Dokumente, unabhängig ihres jeweiligen Schemas, in nur einer Kantentabelle abgelegt werden, kann diese eine sehr große Anzahl an Einträgen aufweisen. Dies führt unter Umständen zu Performanzproblemen. Diesen kann durch Aufspaltung der Kantentabelle begegnet werden. Statt nur einer Tabelle werden jeweils getrennte Tabellen für Element-, Attribut-, Textknoten und die sonstigen Bestandteile eines XML-Dokuments verwendet.

Ein weiteres Problem tritt beim Einfügen eines neuen Knotens zwischen zwei vorhandenen Geschwisterknoten auf. In diesem Fall müssen die Ordnungszahlen der bestehenden Einträge der nachfolgenden Geschwisterknoten erhöht werden.

Des Weiteren bietet die oben angegebene Kantentabelle keine Unterstützung für die einfache Feststellung der Verwandtschaftsbeziehung zwischen zwei Knoten. Dies ist insbesondere bei der Auswertung von XPath-Ausdrücken von Nachteil.

```
<description>
  The <u>MONDIAL database</u> has been compiled
  from the <i>CIA World Factbook</i> and other
  geographical Web data sources.
</description>
```



DocID	KnotenID	VorfahrID	Ordnung	Knotenname	Wert
1	1	—	—	description	—
1	2	1	1	TEXT	The
1	3	1	2	u	MONDIAL database
1	4	1	3	TEXT	has been compiled from the
1	5	1	4	i	CIA World Factbook
1	6	1	5	TEXT	and other geographical Web data sources.

Abb. 4–6 *Gemischter Inhalt: Baumdarstellung und zugehörige Kantentabelle*

Die beiden zuletzt genannten Nachteile haben ihre Ursache in dem verwendeten Kennzeichnungsschema. Dieses besteht hier aus den Attributen `KnotenID`, `VorfahrID` und `Ordnung`. Diese drei Attribute dienen der Speicherung der Baumstruktur der XML-Dokumente. Es existieren eine Reihe weiterer Kennzeichnungsschemata, die durch eine geschickte Knotenkennzeichnung die Nachteile zu überwinden versuchen. Einige ausgewählte Schemata werden im folgenden Abschnitt vorgestellt.

Eine Variation der oben beschriebenen Kantentabelle stellt das Verfahren XRel [YAS01] dar, bei dem die Knoten in vier relationalen Tabellen gespeichert werden. Die Aufteilung auf die Tabellen erfolgt nach dem Knotentyp. Je eine Tabelle wird zur Ablage der Elemente, Attribute und Texte genutzt. Für jeden Knoten wird die Start- und Endposition im XML-Dokument sowie eine Pfad-ID festgehalten. Die vierte Tabelle speichert alle im Dokument auftretenden Pfade und ordnet diesen jeweils eine ID zu.

Bei dem in [JLW02] vorgeschlagenen XParent-Verfahren handelt es sich um eine weitere Abwandlung der kantenorientierten Modellierungen. XParent verwendet ebenfalls vier relationale Tabellen. Texte und Attribute werden in einer gemeinsamen Datentabelle abgelegt. Für Elemente und Pfade steht jeweils eine eigene Tabelle zur Verfügung. Daneben werden die Eltern-Kind-Verknüpfungen in einer separaten Tabelle festgehalten. Im Gegensatz zum XRel-Verfahren werden Start- und Endpositionen der einzelnen Knoten im XML-Dokument nicht gespeichert.

4.3.2 Kennzeichnungsschemata

Zur modellbasierten Speicherung von XML-Dokumenten wird ein Kennzeichnungsschema (engl. „labeling scheme“) benötigt [TVB02, OR10]. Ein derartiges Schema weist allen Knoten, aus denen sich ein XML-Dokument in der Baumstruktur zusammensetzt, eine nach einem bestimmten System aufgebaute Kennzeichnung (Label) zu. Über diese Labels lässt sich die ursprüngliche Struktur des Dokuments wiederherstellen. Die Labels erlauben Aussagen über Beziehungen zwischen Knoten, wie zum Beispiel Eltern-Kind- und Vorfahr-Nachfahr-Beziehungen, sowie über die Dokumentreihenfolge der Elemente.

Bei der Auswahl eines Labeling-Schemas muss zwischen dem optimalen Einsatz für Abfragen und dem zusätzlich eingesetzten Aufwand, um beliebige Updates gut zu unterstützen, abgewägt werden. Ideal ist ein Verfahren, dass möglichst unkompliziert und doch gleichzeitig effizient ist. Es sollte sowohl gut für Dokumente eingesetzt werden können, die viel geändert werden, als auch für solche geeignet sein, die im Wesentlichen statisch sind. Falls neue Knoten eingefügt werden, sollte nach Möglichkeit die Umbenennung bestehender Knoten nicht erforderlich sein. Ein Kennzeichnungsschema mit dieser Eigenschaft bezeichnet man als robustes Schema. Ein Labeling-Schema sollte auch unabhängig von der Anzahl und den Positionen der Änderungen und Einfügungen effizient bleiben.

In den folgenden Unterabschnitten werden einige weit verbreitete Verfahren vorgestellt. Erster hier besprochener Vertreter ist das relativ einfach gehaltene Kennzeichnungsschema, das auf Prä- und Postordnungsrängen basiert. Häufig anzutreffen ist ebenfalls das Dewey-Kennzeichnungsschema, das jedem Knoten ein Label gibt, das den Pfad des Knotens ab der Dokumentwurzel repräsentiert. Dies erlaubt die einfache Bestimmung des niedrigsten gemeinsamen Vorfahren (engl. „lowest common ancestor“, LCA) einer Knotenmenge. Das Eingrenzungskennzeichnungsschema unterstützt dagegen weder die direkte Berechnung des LCA noch die sofortige Bestimmung, ob zwei Knoten Geschwister sind. Ein Nachteil des Dewey-Schemas bei dynamischen Dokumenten, in denen häufig Knoten eingefügt und gelöscht werden, ist jedoch, dass bei Einfügungen eine aufwendige Umkennzeichnung der bestehenden Knoten nötig werden kann. Dies

versucht das auf Dewey basierende Verfahren Ordpath, das der Microsoft SQL-Server nutzt, durch einige Tricks bei der Knotenbenennung zu verhindern.

Schließlich gibt es noch Codierungsverfahren, die ein relativ neuer Vorschlag sind und spezielle Optimierungen für die Unterstützung von Updates auf XML-Dokumenten bieten. Näheres hierzu in Abschnitt 4.3.2.7.

Die grundsätzliche Problematik, dass einige Labeling-Schemata eher für statische und andere für dynamische Dokumente geeignet sind, wird vermutlich nicht beseitigt werden können. Einige Autoren schlagen deshalb den gleichzeitigen Einsatz mehrerer Kennzeichnungssysteme vor, um ein für das jeweilige Dokument besonders geeignetes Verfahren auswählen zu können. Diese Entscheidung wird im Wesentlichen aufgrund der Update- oder Abfragehäufigkeit des einzelnen Dokuments getroffen. Weiterhin kann die Dokumentgröße einbezogen werden und ob das Dokument häufig serialisiert werden muss. Dabei muss allerdings beachtet werden, dass sich diese Eigenschaften im Laufe der Zeit ändern können und deshalb unter Umständen ein Wechsel des Kennzeichnungsschemas erforderlich wird.

Neben diesen Unsicherheiten, das optimale Labeling-Schema auszuwählen, kommt als Nachteil dieses Ansatzes hinzu, dass das Gesamtsystem mit verschiedenen Kennzeichnungsverfahren arbeiten muss. Dies erfordert einen höheren Wartungsaufwand und macht die Abfrage- und Updatelogik deutlich komplizierter. Aus diesem Grund sollte ein Verfahren gewählt werden, dass unabhängig von der Updatehäufigkeit und dem speziellen Aufbau der Dokumente eine gute Leistungsfähigkeit zeigt. [XLW09] schlagen ein dynamisches Dewey-Kennzeichnungsschema (DDE) vor.

4.3.2.1 Prä- und Postordnungskennzeichnungsschema

Das Prä- und Postordnungskennzeichnungsschema nutzt, wie aus der Bezeichnung des Verfahrens klar ersichtlich, den Prä- und Postordnungsrang der Knoten in der Baumdarstellung eines XML-Dokuments, um die Beziehungen der Knoten untereinander zu modellieren.

Unter dem Präordnungsrang eines Knotens versteht man die Position des Knotens im Baum, wenn man diesen in Präordnung durchläuft. Dies ist die Knotenfolge, in der man die Knoten bei einer Tiefensuche im Baum erstmals besucht. Die Präordnung entspricht der Dokumentenordnung, also der Abfolge, in der die Start-Tags der Elemente und die Attribute im zugehörigen XML-Dokument erscheinen. Deshalb wird die Präordnung benötigt, um ein in Baumstruktur vorliegendes XML-Dokument zu serialisieren.

Der Postordnungsrang eines Knotens bezeichnet entsprechend die Position des Knotens, wenn man den Baum in Postordnung durchläuft. Diese Reihenfolge erhält man, wenn man die Knoten so auflistet, wie sie bei einer Tiefensuche im

Baum verlassen werden. Dies ist der Zeitpunkt, zu dem die Bearbeitung des Knotens selbst und aller Kinder erledigt ist.

Abbildung 4–7 zeigt einen Baum, der mit Labels nach dem Prä- und Postordnungskennzeichnungsschema beschriftet ist. Wie man sieht, setzt sich jedes Label (a_1,a_2) eines Knotens A aus der Ordnungszahl a_1 bei Durchlauf in Präordnung und der Ordnungszahl a_2 bei Durchlauf in Postordnung zusammen. Die Labels lassen sich nutzen, um Verwandtschaftsbeziehungen zwischen Knoten zu prüfen.

Vorfahr-Nachfahr-Beziehung: Seien (a_1,a_2) und (b_1,b_2) die Labels der Knoten A und B. B ist Nachfahr von A genau dann, wenn $b_1 > a_1$ und $b_2 < a_2$.[16]

Für die Ermittlung von Eltern-Kind- und Geschwisterbeziehungen sowie des niedrigsten gemeinsamen Vorfahren zweier Knoten existiert beim Prä- und Postordnungskennzeichnungsschema leider kein entsprechend einfacher Zusammenhang.

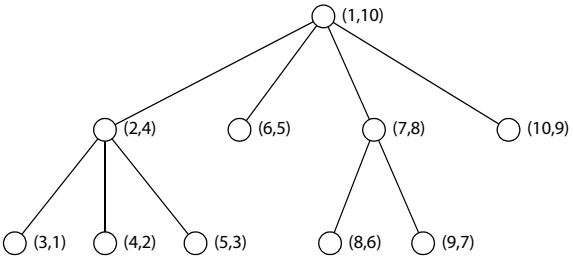


Abb. 4–7 *Prä- und Postordnungskennzeichnungsschema*

Die Eigenschaften dieses Labeling-Schemas kann man sehr gut darstellen, wenn man die Knoten nach ihren Prä- und Postordnungsrängen in ein zweidimensionales Koordinatensystem einträgt. Dies wird in Abbildung 4–8 gezeigt. In der Abbildung wurde durch den Knoten mit dem Label (7,8) eine horizontale und eine vertikale Hilfslinie gezeichnet. Diese beiden Linien teilen das Koordinatensystem in vier Quadranten. Im linken oberen Quadranten liegen die Vorfahren des Knotens mit dem Label (7,8), im rechten unteren dessen Nachfahren.

Im rechten oberen Quadranten befinden sich die Knoten, die im zugehörigen XML-Dokument in Dokumentenordnung nach dem Knoten mit dem Label (7,8) liegen, ausgenommen Nachfahren des Knotens selbst. Dementsprechend enthält der linke untere Quadrant alle in Dokumentenordnung zuvor stehenden Knoten, außer wiederum den Vorfahren des Knotens selbst. Die vier Quadranten finden sich auch im XPath-Datenmodell wieder. Sie entsprechen den Achsen mit den Bezeichnungen `ancestor`, `following`, `preceding` und `descendant`.

16. Da diese Eigenschaft zuerst von Dietz in [Die82] beschrieben wurde, wird das Prä- und Postordnungskennzeichnungsschema in der Literatur zum Teil auch als Dietz' Kennzeichnungsschema bezeichnet.

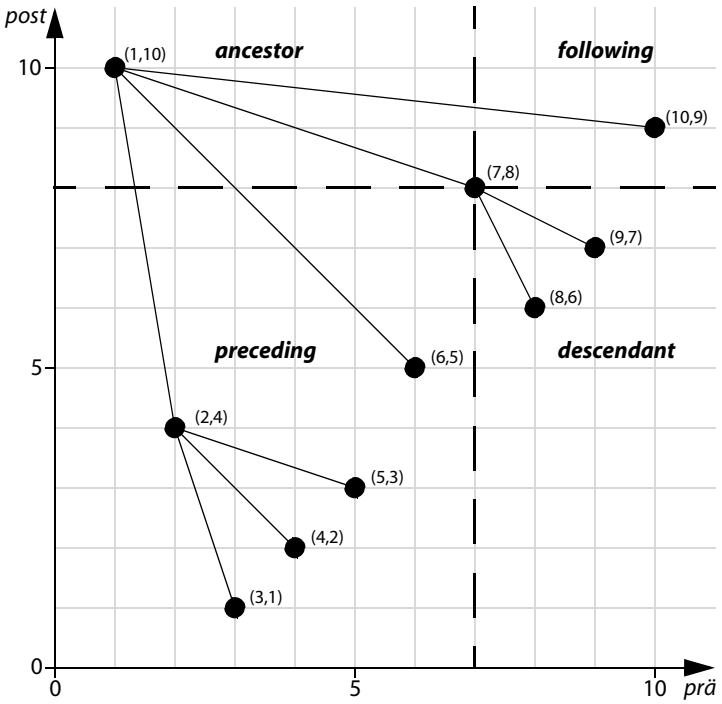


Abb. 4–8 *Prä- und Postordnungskennzeichnungsschema: Knotenbeziehungen*

Das Verhalten des Prä- und Postordnungskennzeichnungsschemas bei Einfügeoperationen ist in Abbildung 4–9 dargestellt. Man erkennt, dass alle Knoten, die – bezogen auf die Dokumentenordnung – nach dem neuen Knoten stehen, sowie alle Vorfahren ein neues Label erhalten müssen. Somit handelt es sich hier nicht um ein robustes Kennzeichnungsschema. Es eignet sich deshalb nur bedingt für Dokumente, die häufig Änderungen unterliegen.

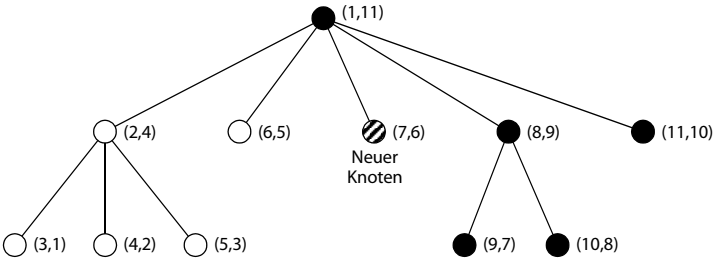


Abb. 4–9 *Prä- und Postordnungskennzeichnungsschema: Einfügeoperation*

Einige Autoren erweitern das Kennzeichnungsschema, indem in die Labels neben dem Prä- und Postordnungsrang des Knotens selbst zusätzlich die Präordnung

des zugehörigen Elternknotens aufgenommen wird. Diese Zusatzangabe ermöglicht unter anderem die effizientere Ermittlung aller Kinder eines Knotens.

[GK03] und [GKT04] zeigen, wie sich im Prä- und Postordnungskennzeichnungsschema codierte XML-Dokumente mit Standard-SQL abfragen lassen und dass eine Leistungssteigerung durch Einführung eines so genannten Staircase-Join-Operators möglich ist. Dieser Operator dient der effizienten Bildung von Knotenmengen, die auf den XPath-Achsen basieren, indem die dabei genutzten Algorithmen Duplikate durch Ausnutzen der Baumstruktur verhindern. Eine Implementierung auf Basis des DBMS MonetDB findet sich in [BGK06].

Aber auch ohne spezielle Join-Operatoren für XML-Strukturen lassen sich mit Standarddatenbanken XPath-Abfragen gut umsetzen, wie Grust et al. zeigen [GRT07]. Sie setzen dazu partitionierte B-Baum-Indexe, Aggregatfunktionen zur Duplikateliminierung und die in den Datenbankmanagementsystemen bereits implementierten Abfrageoptimierungstechniken ein.

4.3.2.2 Eingrenzungskennzeichnungsschema

Ein bekannter Vertreter der intervallbasierten Bezeichnungssysteme ist das Eingrenzungskennzeichnungsschema (engl. „containment labeling scheme“), also ein Schema, das auf der Identifikation der Knoten durch Eingrenzung über Werte basiert.

Dazu werden für das Label jedes Knotens drei Angaben benötigt: Start, Ende und Ebene. Start- und Endwert definieren dabei ein Intervall, das die Intervalle aller Nachfahren umfasst. Der Ebenenwert gibt die Ebene des Knotens im XML-Baum an.[17] Der Ebenenwert ist nicht zwingend erforderlich. Er wird den Labels hinzugefügt, um Verwandtschaftsbeziehungen leichter ermitteln zu können.

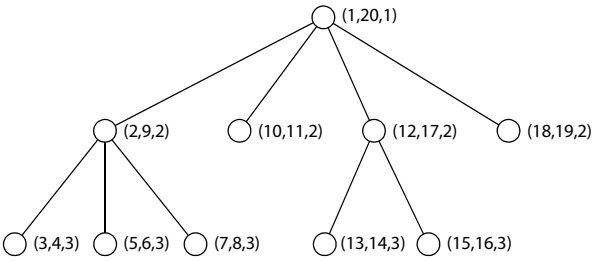


Abb. 4–10 *Eingrenzungskennzeichnungsschema*

Abbildung 4–10 zeigt zur Verdeutlichung ein nach dem Eingrenzungskennzeichnungsschema beschrifteten XML-Baum. Im Beispiel ist der Knoten mit dem Label (15,16,3) Nachfahr des Knotens mit dem Label (1,20,1), da das Intervall

17. Es existieren Varianten dieses Kennzeichnungsschemas, die an Stelle des Endwerts die Größe des Intervalls angeben.

[15;16] im Intervall [1;20] enthalten ist. Falls sich zusätzlich die Ebenenwerte nur um eins unterscheiden, so handelt es sich um einen direkten Nachfahr, also um einen Kindknoten.

Seien nun (a_1,a_2,a_3) und (b_1,b_2,b_3) die Labels der Knoten A und B eines nach dem Eingrenzungskennzeichnungsschema beschrifteten Baums. Allgemein gilt:

Vorfahr-Nachfahr-Beziehung: B ist Nachfahr von A genau dann, wenn $[b_1;b_2] \subset [a_1;a_2]$.

Eltern-Kind-Beziehung: B ist Kind von A genau dann, wenn $[b_1;b_2] \subset [a_1;a_2]$ und $b_3 = a_3 + 1$.

Durch Sortierung der Knoten nach den Startwerten ihrer Labels, erhält man leicht die Knoten in Dokumentenordnung. Problematisch sind jedoch Einfügeoperationen im Baum. Alle Vorfahren eines neuen Knotens und im Bezug auf die Dokumentenordnung weiter hinten stehenden Knoten müssen neu gekennzeichnet werden. Abbildung 4–11 zeigt dazu ein Beispiel. Li und Moon haben in [LM01] vorgeschlagen, Lücken in der Nummerierung vorzusehen, sodass eine Neukennzeichnung vermieden werden kann. Dies löst das Problem jedoch nur bedingt, da nach einigen Einfügeoperationen die Lücken aufgebraucht sind, sodass dennoch eine Neuvergabe der Labels erforderlich ist. Außerdem muss beachtet werden, dass durch die Lücken die Labels nicht so kompakt sind, was eine Erhöhung des Speicherbedarfs zur Folge hat.

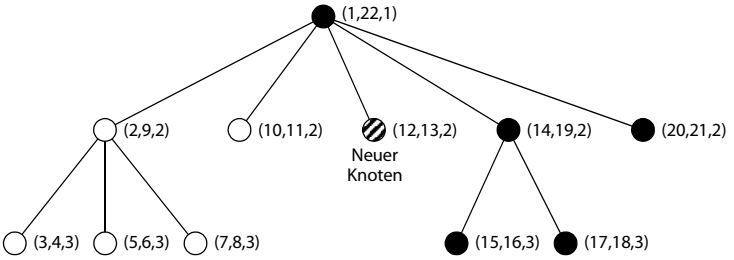


Abb. 4–11 *Einfügeoperation beim Eingrenzungskennzeichnungsschema*

In [AYU03] regen Amagasa et al. an, die ganzzahligen Werte in den Labels durch Gleitkommazahlen zu ersetzen. Dies verhindert ebenfalls den Zwang zur Neukennzeichnung von Knoten nach Einfügeoperationen. Allerdings ist das Verfahren durch die Genauigkeit der verwendeten Gleitkommazahlen begrenzt. Nachdem eine gewisse Anzahl neuer Knoten eingefügt wurde, ist es auch hierbei unvermeidbar, eine Neukennzeichnung vorzunehmen. [ZLF11] schlägt ein ähnliches Verfahren vor, das auf der lexikografischen Ordnung und der Speicherung von ganzzahligen Werten mit Hilfe einer festen Anzahl von Bits basiert. [SLF05] zeigt neben einem Algorithmus zur Durchführung der Neukennzeichnung ver-

schiedene Join-Verfahren, die auf dem Eingrenzungskennzeichnungsschema aufbauen.

Zuletzt sei auch die in [ZND01] und [AJK02] vorgestellte Variante des Eingrenzungskennzeichnungsschemas erwähnt. Die Autoren verwenden als Start- bzw. Endwert die Position des ersten bzw. letzten Wortes der Textrepräsentation des Knotens im zugehörigen XML-Dokument. An den grundsätzlichen Eigenschaften des Verfahrens ändert sich dadurch nichts. [BKS02] zeigt einen Algorithmus, der, basierend auf diesem Kennzeichnungsschema, XPath-Ausdrücke auswertet und die dabei auftretenden Zwischenergebnisse möglichst klein hält. Auch sei an dieser Stelle auf den in [AC11] besprochenen Ansatz hingewiesen, bei dem ein Eingrenzungskennzeichnungsschema in Verbindung mit einem strukturbasierten Abbildungsverfahren eingesetzt wird, das den auftretenden Elementtypen verschiedene Relationen zuordnet. [Lu10] befasst sich mit einer Erweiterung, die die Besonderheiten von rekursiv definierten DTDs berücksichtigt.

4.3.2.3 Dewey-Kennzeichnungsschema

Beim Dewey-Labeling-Schema[18] werden Kennzeichnungen verwendet, die durch das Anhängen einer Ordnungsziffer an die Kennzeichnung des Elternknotens entstehen. Die einzelnen Komponenten werden jeweils durch einen Punkt voneinander getrennt. Die Ordnungsziffer bezeichnet die Position des Knotens unterhalb des jeweiligen Elternknotens, gibt also an, das wievielte Kind ein Knoten ist. Der Wurzelknoten erhält das Label 1. Kennzeichnungsschemata, denen diese Systematik zugrunde liegt, werden auch Präfixkennzeichnungsschemata genannt. Abbildung 4–12 zeigt einen XML-Baum mit Dewey-Labels.

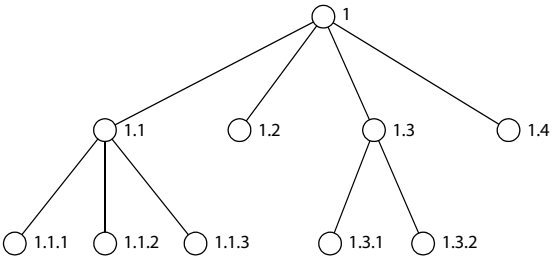


Abb. 4–12 *Dewey-Kennzeichnungsschema*

Bei diesem Kennzeichnungssystem erhält man das Label des Elternknotens durch einfaches Abschneiden der letzten Komponente eines Knotenlabels. Die Ebene im

18. Der Name Dewey geht auf den amerikanischen Bibliothekar Melvin Dewey zurück, der 1876 die Dewey Decimal Classification (DDC) entwickelte, eine Dezimalklassifikation zur Ordnung von Bibliotheksbeständen.

Baum, in der sich ein Knoten befindet, wird nicht explizit abgelegt. Die Ebene ist identisch mit der Anzahl der Komponenten eines Labels.

Dewey-Ordnung

Auf der Menge der Knoten lässt sich mit Hilfe der Dewey-Labels die Dewey-Ordnung definieren.

Definition: Seien $a_1.a_2...a_m$ und $b_1.b_2...b_n$ Dewey-Labels der Knoten A und B. $A <_D B$ gilt genau dann, wenn

- $m < n$ und $a_1 = b_1, a_2 = b_2, ..., a_m = b_m$ oder
- $\exists\, k \leq \min(m, n)$, sodass $a_1 = b_1, a_2 = b_2, ..., a_{k-1} = b_{k-1}$ und $a_k < b_k$.

Die so definierte Dewey-Ordnung entspricht somit der lexikografischen Ordnung. Für zwei Knoten A und B mit $A \neq B$ gilt stets $A <_D B$ oder $B <_D A$.

Dewey-Labels besitzen die nachfolgenden Eigenschaften. Seien $a_1.a_2...a_m$ und $b_1.b_2...b_n$ wiederum die Dewey-Labels der Knoten A und B.

Vorfahr-Nachfahr-Beziehung: B ist Nachfahr von A genau dann, wenn $m < n$ und $a_1 = b_1, a_2 = b_2, ..., a_m = b_m$, mit anderen Worten $a_1.a_2...a_m$ ist Präfix von $b_1.b_2...b_n$.

Eltern-Kind-Beziehung: B ist Kind von A genau dann, wenn B Nachfahr von A und $m = n - 1$, mit anderen Worten das Elternlabel von $b_1.b_2...b_n$ ist $a_1.a_2...a_m$.

Dokumentenordnung: Genau dann wenn $B <_D A$, steht B in Bezug auf die Dokumentenordnung vor A.

Geschwisterbeziehung: A und B sind Geschwister genau dann, wenn die Elternlabels von A und B übereinstimmen.

Niedrigster gemeinsamer Vorfahr: Der Knoten C mit dem Label $c_1.c_2...c_p$ ist niedrigster gemeinsamer Vorfahr von A und B, falls C Vorfahr von A und B ist und $p + 1 = \min(m, n)$ oder $a_{p+1} \neq b_{p+1}$ gilt.

Damit wird im Gegensatz zum Eingrenzungskennzeichnungsschema die einfache Berechnung des niedrigsten gemeinsamen Vorfahren unterstützt. Die Ermittlung des niedrigsten gemeinsamen Vorfahren ist hier äquivalent zur Suche nach dem längsten gemeinsamen Präfix der beteiligten Labels.

Das Einfügen von Knoten in einen XML-Baum, der mit dem Dewey-Labeling-Schema versehen ist, führt ähnlich wie beim Eingrenzungskennzeichnungsschema zu hohem Aufwand durch Neukennzeichnung von Knoten. Wie Abbildung 4–13 zeigt, müssen alle folgenden Geschwister des neuen Knotens sowie alle deren Nachfahren mit neuen Labels versehen werden.

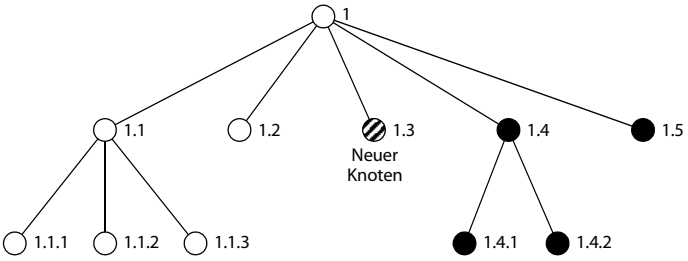


Abb. 4–13 *Einfügeoperation beim Dewey-Kennzeichnungsschema*

Bei Verwendung des Dewey-Kennzeichnungsschemas in einer relationalen Knotentabelle (analog zur Kantentabelle 4–3) wird man häufig in einem zusätzlichen Attribut den eindeutigen Pfad jedes Knotens zur Wurzel ablegen. Viele Autoren verwenden hierzu den absoluten Pfad in XPath-Notation und ersetzen darin die Schrägstriche (/) durch Nummernzeichen (#).

Durch dieses Pfadattribut lassen sich XPath-Anfragen sehr viel effizienter auswerten. Ohne das Attribut benötigt man eine große Anzahl an Self-Joins, da die Verknüpfungen über die Knotenlabels und die Knotennamen von Eltern- und Kindknoten erfolgen müssen. Die Anzahl der erforderlichen Joins ist proportional zur Länge des Suchpfades.

Die Effizienz lässt sich zusätzlich steigern, indem der Pfad der Knoten nicht direkt, sondern invertiert, also in Richtung vom Knoten zur Wurzel abgelegt wird. Dies hat Vorteile bei der Auswertung von XPath-Ausdrücken, die sich auf die `descendant-or-self`-Achse (Abkürzung: //) beziehen. Zum Beispiel muss bei dem XPath-Ausdruck `//city/name` zur Selektion der Namen aller Städte unabhängig von den Provinzen und Ländern, in denen sie liegen, im Pfadattribut der Knotentabelle nach Zeichenketten gesucht werden, die auf `/city/name` enden. In SQL würde dies in der `WHERE`-Klausel mit dem Prädikat `LIKE` in Verbindung mit dem Jokerzeichen `%` zu Beginn der Zeichenkette formuliert werden.

Eine derartige Suche mit beliebigen Präfixen, also eine Postfixsuche, wird von den meisten Indexen nicht gut unterstützt. Viel effizienter ist eine Präfixsuche, hier die Suche nach allen Zeichenketten, die mit `name/city/` beginnen.

Georgiadis und Vassalos verwenden das Dewey-Kennzeichnungsschema zur Speicherung von XML-Dokumenten in relationalen Tabellen und prüfen, wie sich XPath-Ausdrücke in passende SQL-Anweisungen transformieren lassen [GV07]. Sie zeigen Optimierungsmöglichkeiten bezüglich der dabei eingesetzten Join-Algorithmen auf. [MWD06] weist darauf hin, dass sich bei der Umsetzung von XQuery-Anweisungen in SQL die Anzahl der erforderlichen Joins durch paarweise Betrachtung und Auswertung der enthaltenen Join-Bedingungen minimieren lässt. Durch Auftrennung der Dewey-Labels in eine Eltern- und eine Kindkomponente erreicht man Vorteile bei der Auswertung von Eltern-Kind- sowie

Geschwisterabfragen, da sich die notwendigen Joins effizienter formulieren lassen [MOK10].

Eine vorgeschlagene Erweiterung des Dewey-Kennzeichnungsschemas trägt die Bezeichnung Extended Dewey [LLC05]. Bei diesem Verfahren wird die jeweils letzte Labelkomponente so vergeben, dass sich aus dieser mit Hilfe der Modulo-Funktion die Knotenbezeichnung ableiten lässt. Dazu muss eine Schemadefinition, zum Beispiel in Form einer DTD, vorliegen. Ähnlich arbeitet auch das in [MM10] vorgeschlagene Kennzeichnungsschema, bei dem sich die Knotenlabels nur in Verbindung mit einer so genannten Zusammenfassung, die die Struktur des XML-Dokuments repräsentiert, interpretieren lassen.

Das Problem der Neukennzeichnung beim Einfügen von Knoten wird durch Extended Dewey nicht gelöst. In [SZA06] wird die Modifikation IFDewey („Insert-Friendly Dewey“) beschrieben, bei der zunächst nur ungerade Zahlen bei der Labelvergabe benutzt werden. Dies macht Neukennzeichnungen beim Einfügen von Knoten unnötig. Die Details sind bis auf die Möglichkeit der Ermittlung der Knotenbezeichnung aus dem Label wie bei Extended Dewey mit dem im folgenden Abschnitt beschriebenen Ordpath-Verfahren vergleichbar.

4.3.2.4 Ordpath

Ordpath [OOP04] steht als Abkürzung für „ordinal path“ und ist eine Variante des Dewey-Kennzeichnungsschemas.[19] Microsoft hat auf Ordpath ein US-Patent [OOP05] und setzt das Verfahren bei seinem Datenbankmanagementsystem SQL-Server ein. Ordpath ermöglicht es, bei Einfügeoperationen auf die Neukennzeichnung von Knoten zu verzichten. Dies wird erreicht, indem bei der ersten Kennzeichnung des XML-Baums nur Dewey-Labels mit ungeraden Zahlen verwendet werden. Die geraden Zahlen werden für Einfügeoperationen genutzt.

Wenn ein Knoten zwischen zwei Knoten eingefügt werden soll, deren letzte Komponenten zwei aufeinanderfolgende ungerade Zahlen sind, so erhält man das neue Label, indem man als letzte Komponente die zwischen den ungeraden Zahlen liegende gerade Zahl verwendet und zusätzlich als weitere Komponente eine 1 anhängt. Der neue Knoten zwischen den Knoten mit den Labels 1.3 und 1.5 erhält zum Beispiel das Label 1.4.1. Die Abbildungen 4–14 und 4–15 verdeutlichen das Verfahren. Ein weiterer Knoten der zwischen den Knoten 1.4.1 und 1.5 eingefügt wird, bekommt das Label 1.4.3. Ein Knoten, der schließlich zwischen 1.4.1 und 1.4.3 stehen soll, erhält 1.4.2.1 als Label. Einfügungen vor den Knoten, deren Label auf .1 enden, werden über negative Zahlen realisiert.

Im Gegensatz zu Verfahren, die einige Labels reservieren und dadurch nur eine begrenzte Menge an neuen Knoten ohne Neukennzeichnung aufnehmen

19. Einige Autoren bezeichnen auch das Label des Dewey-Kennzeichnungsschemas als Ordpath. Hier ist mit diesem Begriff jedoch die Weiterentwicklung des Schemas gemeint.

können, erlaubt Ordpath prinzipiell beliebig viele Einfügeoperationen ohne Neuvergabe von Labels, da sich die freigehaltenen Labels nicht aufbrauchen.

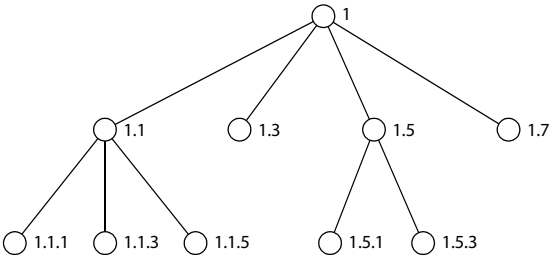


Abb. 4–14 *Ordpath-Kennzeichnungsschema*

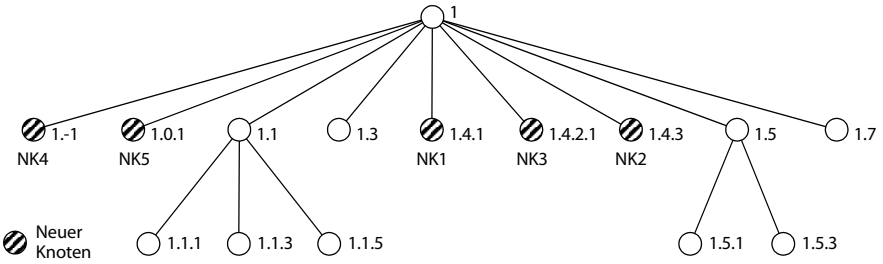


Abb. 4–15 *Einfügeoperationen beim Ordpath-Kennzeichnungsschema*

Die Ebene eines Knotens im Baum lässt sich bei Ordpath nicht mehr durch Zählen der Komponenten des Labels ermitteln. Grundsätzlich besteht jedes Label aus einer variablen Anzahl an geraden und ungeraden Zahlen, auf die als letzte Komponente stets eine ungerade Zahl folgt. Die Verarbeitung von Ordpath-Labels ist im Vergleich zu Dewey-Labels deutlich komplizierter, da jede Label-Komponente in Abhängigkeit davon, ob die Zahl gerade oder ungerade ist, unterschiedlich behandelt werden muss. Dies hat einen negativen Einfluss auf die Verarbeitungsgeschwindigkeit von Abfragen, da unter anderem die Ermittlung des niedrigsten gemeinsamen Vorfahren nicht mehr so einfach möglich ist. Beispielsweise ist der niedrigste gemeinsame Vorfahr der Knoten mit den Labels 1.2.6.3 und 1.2.6.5.7 nicht, wie aufgrund der gemeinsamen Präfixe zu erwarten wäre, 1.2.6, sondern 1.

4.3.2.5 Dynamic Dewey

Dynamic Dewey (DDE) ist ein Kennzeichnungsschema, das in [XLW09] vorgestellt wird und das auf dem in Abschnitt 4.3.2.3 beschriebenen Dewey-Verfahren basiert.

Die Ausgangskennzeichnung entspricht dem Dewey-Kennzeichnungsschema. Die Labels haben das Format $a_1.a_2...a_m$, wobei $a_1.a_2...a_{m-1}$ das Label des Elternknotens ist und a_m die Position des Knotens unterhalb des Elternknotens.

Nun wird jedoch beim DDE-Labeling-Schema auf der Menge der Labels nicht die lexikografische Ordnung verwendet, sondern eine eigene DDE-Ordnung. Diese definiert sich über eine DDE-Quasiordnung wie folgt:

Definition DDE-Quasiordnung: Seien $a_1.a_2...a_m$ und $b_1.b_2...b_n$ die DDE-Labels der Knoten A und B. $A \leq_{DDE} B$ gilt genau dann, wenn

- $m \leq n$ und $\frac{a_1}{b_1} = \frac{a_2}{b_2} = ... = \frac{a_m}{b_m}$ oder
- $\exists\, k \leq \min(m, n)$, sodass $\frac{a_1}{b_1} = \frac{a_2}{b_2} = ... = \frac{a_{k-1}}{b_{k-1}}$ und $a_k \cdot b_1 < b_k \cdot a_1$.

Es lässt sich leicht zeigen, dass die so definierte Quasiordnung reflexiv und transitiv ist, das heißt, für drei beliebige Knoten A, B und C gilt stets $A \leq_{DDE} A$ sowie falls $A \leq_{DDE} B$ und $B \leq_{DDE} C$, so folgt daraus $A \leq_{DDE} C$.

Um nun die DDE-Ordnung zu definieren, muss zunächst eine Äquivalenzrelation eingeführt werden. Seien hierzu A und B zwei mit DDE-Labels versehene Knoten.

Definition DDE-Äquivalenzrelation: A und B sind äquivalent genau dann, wenn $A \leq_{DDE} B$ und $B \leq_{DDE} A$. Man schreibt in diesem Fall $A =_{DDE} B$.

Definition DDE-Ordnung: $A <_{DDE} B$ gilt genau dann, wenn $A \leq_{DDE} B$ und $A \neq_{DDE} B$.

Mit diesen Definitionen gelten für zwei Knoten A und B mit den DDE-Labels $a_1.a_2...a_m$ und $b_1.b_2...b_n$ die folgenden Eigenschaften:

Vorfahr-Nachfahr-Beziehung: B ist Nachfahr von A genau dann, wenn $m < n$ und

$$\frac{a_1}{b_1} = \frac{a_2}{b_2} = ... = \frac{a_m}{b_m}.$$

Eltern-Kind-Beziehung: B ist Kind von A genau dann, wenn B Nachfahr von A und $m = n - 1$ gilt.

Dokumentenordnung: Genau dann wenn $B <_{DDE} A$, steht B in Bezug auf die Dokumentenordnung vor A.

Geschwisterbeziehung: A und B sind Geschwister genau dann, wenn $m = n$ und

$$\frac{a_1}{b_1} = \frac{a_2}{b_2} = ... = \frac{a_{m-1}}{b_{m-1}}.$$

Niedrigster gemeinsamer Vorfahr: Der Knoten C mit dem Label $c_1.c_2...c_p$ ist niedrigster gemeinsamer Vorfahr von A und B, falls C Vorfahr von A und B ist und $p + 1 = \min(m, n)$ oder $a_{p+1} \cdot b_1 \neq b_{p+1} \cdot a_1$ gilt.

Ziel des Dynamic-Dewey-Kennzeichnungsschemas ist, neue Knoten in den XML-Baum einfügen zu können, ohne die Labels der bisherigen Knoten ändern zu müssen. Um dies zu erreichen, muss zunächst die Addition zweier DDE-Labels definiert werden:

Definition DDE-Label-Addition: Seien A und B zwei Knoten, deren zugehörige DDE-Labels $a_1.a_2...a_m$ und $b_1.b_2...b_m$ die gleiche Anzahl an Komponenten besitzen. Die Addition der Labels wird wie folgt definiert: $(a_1.a_2...a_m) + (b_1.b_2...b_m) = (a_1 + b_1).(a_2 + b_2)...(a_m + b_m)$. Der zugehörige Knoten erhält die Bezeichnung $A + B$.

Die so definierte Addition weist die folgende interessante Eigenschaft auf:

Satz: Seien $a_1.a_2...a_m$ und $b_1.b_2...b_m$ die DDE-Labels der Geschwisterknoten A und B mit $A <_{DDE} B$. Dann gilt: $A <_{DDE} (A + B) <_{DDE} B$.

Abbildung 4–16 zeigt einen mit dem Dynamic-Dewey-Labeling-Schema gekennzeichneten Baum, in dem neue Knoten eingefügt wurden. Man sieht, dass dabei die Labels der bestehenden Knoten unverändert bleiben.

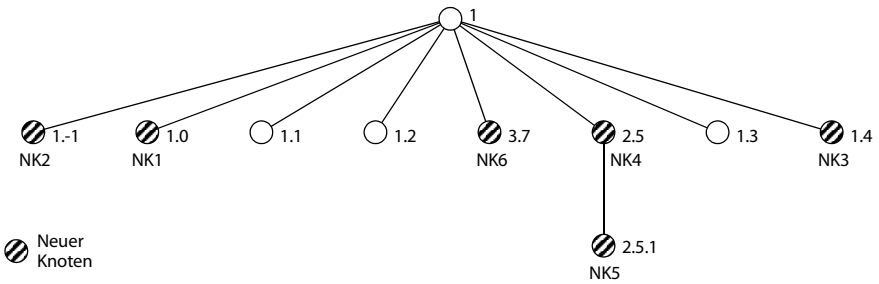


Abb. 4–16 *Einfügeoperationen beim Dynamic-Dewey-Kennzeichnungsschema*

Beim Einfügen können allgemein vier verschiedene Fälle auftreten:

- **Einfügen ganz links:**
  Sei der Knoten A mit dem Label $a_1.a_2...a_n$ das erste Kind eines Knotens. Fügt man einen Knoten links davon als neues erstes Kind ein, dann erhält dieser das Label $a_1.a_2...(a_n\text{-}1)$. Dabei kann die letzte Komponente des Labels auch Null oder eine negative ganze Zahl sein. In Abb. 4–16 tritt dies bei den neuen Knoten *NK1* und *NK2* auf.
- **Einfügen ganz rechts:**
  Sei der Knoten A mit dem Label $a_1.a_2...a_n$ das letzte Kind eines Knotens. Fügt man einen Knoten rechts davon als neues letztes Kind ein, dann erhält dieser

das Label $a_1.a_2...(a_n+1)$. Der neue Knoten *NK3* in Abb. 4–16 gibt hierfür ein Beispiel.

- **Einfügen zwischen zwei Geschwistern:**
  Seien $a_1.a_2...a_m$ und $b_1.b_2...b_m$ die Labels der Geschwisterknoten A und B. Ein zwischen ihnen einzufügender neuer Knoten C erhält das Label $(a_1 + b_1).(a_2 + b_2)...(a_m + b_m)$. Es gilt also $C = A + B$. In Abb. 4–16 wird dies anhand der neuen Knoten *NK4* und *NK6* gezeigt.
- **Einfügen unterhalb eines Blattknotens:**
  Sei der Knoten A mit dem Label $a_1.a_2...a_n$ ein Blattknoten. Fügt man einen Knoten unterhalb davon als neues Kind ein, dann erhält dieser das Label $a_1.a_2...a_n.1$. Dieser Fall wird in Abb. 4–16 bei Knoten *NK5* dargestellt.

Unter Verwendung des Dynamic-Dewey-Kennzeichnungsschemas ist es somit möglich, an beliebigen Stellen innerhalb des Baums Knoten einzufügen, ohne Änderungen an den Labels der im Baum bereits vorhandenen Knoten vornehmen zu müssen.

Xu et al. beweisen in [XLW09] formal die Korrektheit dieses Verfahrens. Außerdem stellen sie das Kennzeichnungsschema Compact DDE (CDDE) vor, das eine Weiterentwicklung von Dynamic Dewey darstellt. CDDE erlaubt als erste Komponente eines Labels auch negative Zahlen. Dies führt zu kompakteren Labels und damit zu einer höheren Verarbeitungsgeschwindigkeit.

Die wichtigsten Änderungen von CDDE gegenüber DDE sind eine veränderte Label-Addition sowie die Definition einer Label-Erweiterungsoperation, die ein Label für einen neu einzufügenden Knoten unterhalb eines Blattknotens erzeugt. Da CDDE abgesehen von der kompakteren Form der Labels keine neuen Aspekte aufzeigt, soll an dieser Stelle auf die Besprechung der Details verzichtet und auf die Darstellung des Verfahrens in [XLW09] verwiesen werden.

4.3.2.6 Primkennzeichnungsschema

Das in [WLH04] beschriebene Primkennzeichnungsschema (engl. „prime labeling scheme“) verwendet einen mathematischen Ansatz, um die Knoten in einem XML-Baum derart zu kennzeichnen, dass bei Einfügeoperationen keine Änderungen an den Labels der bestehenden Knoten vorgenommen werden müssen.

Dazu wird jedem Knoten zunächst eine eindeutige Primzahl, das so genannte Self-Label zugeordnet. Aus diesem Self-Label wird das eigentliche Label des Knotens gebildet, indem das Self-Label mit dem Label des Elternknotens multipliziert wird. Aus einem Label lässt sich somit – da alle Self-Labels verschieden sind – durch Faktorisierung der Pfad des jeweiligen Knotens ermitteln.

Seien A und B zwei Knoten mit den Self-Labels a und b und den Labels $label(A)$ und $label(B)$. Im Primkennzeichnungsschema gelten dann die folgenden Eigenschaften:

Vorfahr-Nachfahr-Beziehung: *B* ist Nachfahr von *A* genau dann, wenn $A \neq B$ und $label(B) \bmod label(A) = 0$.

Eltern-Kind-Beziehung: *B* ist Kind von *A* genau dann, wenn $label(A) = label(B) / b$.

Um in diesem Kennzeichnungssystem die Reihenfolge der Knoten, also die Dokumentenordnung festhalten zu können, werden simultane Kongruenzen verwendet. Dazu wird eine Zahl *SC* ermittelt, mit Hilfe derer aus dem Self-Label jedes Knotens seine Ordnungszahl ermittelt werden kann. Die dazugehörige Formel lautet: *SC* mod *Self-Label* = *Ordnungszahl*. Die Ordnungszahl gibt hier die Position des Knotens in Dokumentenordnung an und wird ab Null gezählt. Da alle Self-Labels prim sind, existiert nach dem Chinesischen Restsatz eine Zahl *SC*, die diese Eigenschaft besitzt. Für die Speicherung der Dokumentenordnung muss nur *SC* abgelegt werden.

Abbildung 4–17 verdeutlicht die Funktionsweise des Primkennzeichnungsschemas anhand eines Beispiels. Nachdem ein Knoten eingefügt wurde, ergibt sich die in Abbildung 4–18 dargestellte Struktur. In beiden Abbildungen sind die Self-Labels in den Kreisen, die die Knoten symbolisieren, angegeben. Die Labels stehen jeweils rechts daneben.

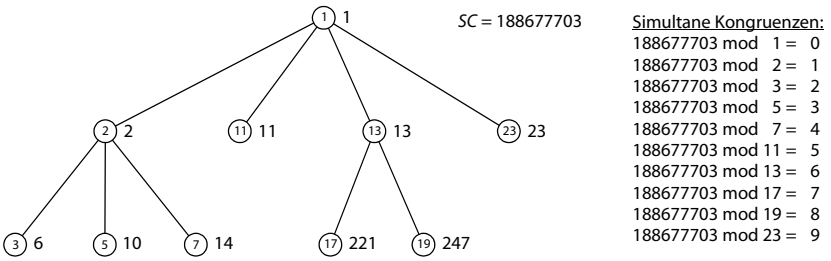


Abb. 4–17 *Primkennzeichnungsschema*

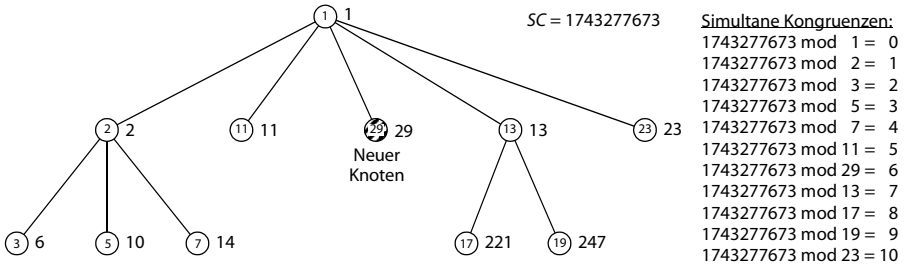


Abb. 4–18 *Einfügeoperation beim Primkennzeichnungsschema*

Damit der mathematische Aufwand zur Lösung der simultanen Kongruenzen nicht zu groß wird, können Gruppen zu je fünf Knoten gebildet werden, deren

Ordnung jeweils über ein System simultaner Kongruenzen festgehalten wird. Dennoch bleibt die Aktualisierung dieser Systeme bei Einfüge- und Löschoperationen aufwendig.

4.3.2.7 Codierungsverfahren

Codierungsverfahren setzen auf einem der zuvor beschriebenen Kennzeichnungsschemata auf und transformieren die Labels dieser Systeme in ein dynamisches Format. Bei dieser Umwandlung soll die Ordnung der Originallabels erhalten bleiben und der Speicherbedarf möglichst klein sein.

QED (Dynamic Quaternary Encoding) ist ein derartiges Codierungsverfahren, bei dem die Labels in so genannte QED-Codes gewandelt werden [LL05]. Ein QED-Code besteht aus einer Folge von Elementen aus der Menge {1, 2, 3}, die auf 2 oder 3 endet. Jedes Element dieser Folge wird mit zwei Bits dargestellt.

Die Bezeichnung „dynamic“ im Namen dieses Verfahrens rührt daher, dass aufgrund des Aufbaus der Codes zwischen je zwei Codes ein weiterer eingefügt werden kann. Dazu wird auf der Menge der QED-Codes die lexikografische Ordnung verwendet. Die Länge der Codes wächst im ungünstigsten Fall bei jeder Einfügung um zwei Bits. Soll zum Beispiel vor dem Code 32 eingefügt werden, so wird dazu der Code 312 genutzt. Ein erneutes Einfügen vor 312 führt zum Code 3112.

Bei Einsatz der QED-Codierung führt das schnelle Längenwachstum zu erhöhtem Speicherbedarf und beeinträchtigt die Abfrageleistung. Das Verfahren CDBS (Compact Dynamic Binary String) versucht, dies zu vermeiden, indem nur die beiden Elemente 0 und 1 eingesetzt werden, sodass je Element nur ein Bit zur Codierung nötig ist [LLH06]. Dadurch erhält man kompaktere Darstellungen, die effizienter verarbeitet werden können. Nachteil bei CDBS ist, dass es zu einem Überlauf kommen kann, wenn, wie üblich, eine feste Anzahl an Bits zur Speicherung der Codelänge vorgesehen ist. Ein weiteres Verfahren ist Vector-Label [XBL07]. Diese Codierung ist zwar weniger kompakt als die QED-Darstellung, allerdings wächst die Länge der Codes bei ungünstigen Einfügungen weniger stark.

Nachteile der meisten Codierungsverfahren sind die Kosten, die durch die Bearbeitung und den Vergleich von Codes variabler Länge entstehen. Des Weiteren ist besonders bei großen XML-Dokumenten der Speicherbedarf der Codes zu berücksichtigen. Wird ein Codierungsverfahren in Verbindung mit Dewey-Labels eingesetzt, so muss jede Label-Komponente einzeln codiert werden. Der erhöhte Aufwand bei der Verarbeitung der Labels macht Codierungsverfahren insbesondere bei Dokumenten, die nur selten geändert werden, weniger attraktiv.

4.3.3 IBM DB2 pureXML

Seit der Version 9 von IBM DB2 bietet die Datenbank weitgehende XML-Unterstützung [PN06].[20] Die neuen XML-Funktionalitäten werden von IBM unter dem Oberbegriff „pureXML“ zusammengefasst. In allen wesentlichen Komponenten der Datenbank wurde eine eigene Verarbeitung von XML-Daten implementiert. So werden beispielsweise XQuery-Ausdrücke nicht in SQL umgesetzt, sondern direkt ausgewertet.[21]

DB2 wird von IBM als hybride Datenbank bezeichnet, da die relationale und XML-wertige Speicherung gleichrangig nebeneinander stehen. Damit bietet sich die Möglichkeit, Teile eines Datenmodells relational und andere als XML-Fragmente abzulegen. Dabei sollten insbesondere Strukturen, die eine hohe Variabilität zeigen oder nur dünn besiedelt sind, XML-wertig gespeichert werden. [MLC07] stellt ein Advisor-Tool vor, das bei diesem Entwurfsprozess helfen kann.

Hier ist die Art und Weise, wie XML-Daten von DB2 gespeichert werden, von besonderem Interesse. DB2 bietet den Spaltentyp `XML`, der in einer relationen Tabelle genutzt werden kann und einen nativen XML-Speicher zur Verfügung stellt. Wie zu erwarten, kann die Tabelle neben dem XML-Datentyp noch weitere herkömmliche oder XML-wertige Attribute besitzen.

Grundsätzlich existiert für die Größe der speicherbaren XML-Dokumente keine Begrenzung. Allerdings bestand bei Einführung der Version 9 zunächst die Einschränkung, dass XML-Dokumente maximal 2 GB groß sein dürfen, da die eingesetzten Client-Server-Protokolle keine größeren Datentransfers erlaubten.

Wenn ein XML-Dokument in eine Tabelle eingefügt oder wieder ausgelesen wird, wandelt DB2 die interne Darstellung in die übliche Textrepräsentation von XML um. Die durch den nativen XML-Speicher implementierte interne Darstellung soll im Folgenden näher erläutert werden.

[NL05] und [BCH06] beschreiben Details des Verfahrens, das seine Ursprünge in dem experimentellen Prototypen System RX [BCJ05] hat. XML-Dokumente, die dem Datenbankserver übergeben werden, werden zunächst mittels eines SAX-Parsers auf Wohlgeformtheit und gegebenenfalls auch anhand eines XML-Schemas validiert. Dabei wird in bekannter Weise ein DOM-Baum erzeugt. In diesem Baum werden alle Element- und Attributnamen sowie die URIs der Namensräume durch Integerwerte ersetzt (siehe Abb. 4–19). Die Zuordnung dieser Werte zu den ursprünglichen Namen geschieht über die Datenbanktabelle `SYSXMLSTRINGS`, die für diesen Zweck dem DB2 Data Dictionary hinzugefügt wurde.

---

20. In der Entwicklungsphase wurde die Version 9 von DB2 mit dem Codenamen „Viper“ bezeichnet.
21. Oracle bietet ebenfalls native Unterstützung für XQuery-Ausdrücke [LKA05, LKW07].

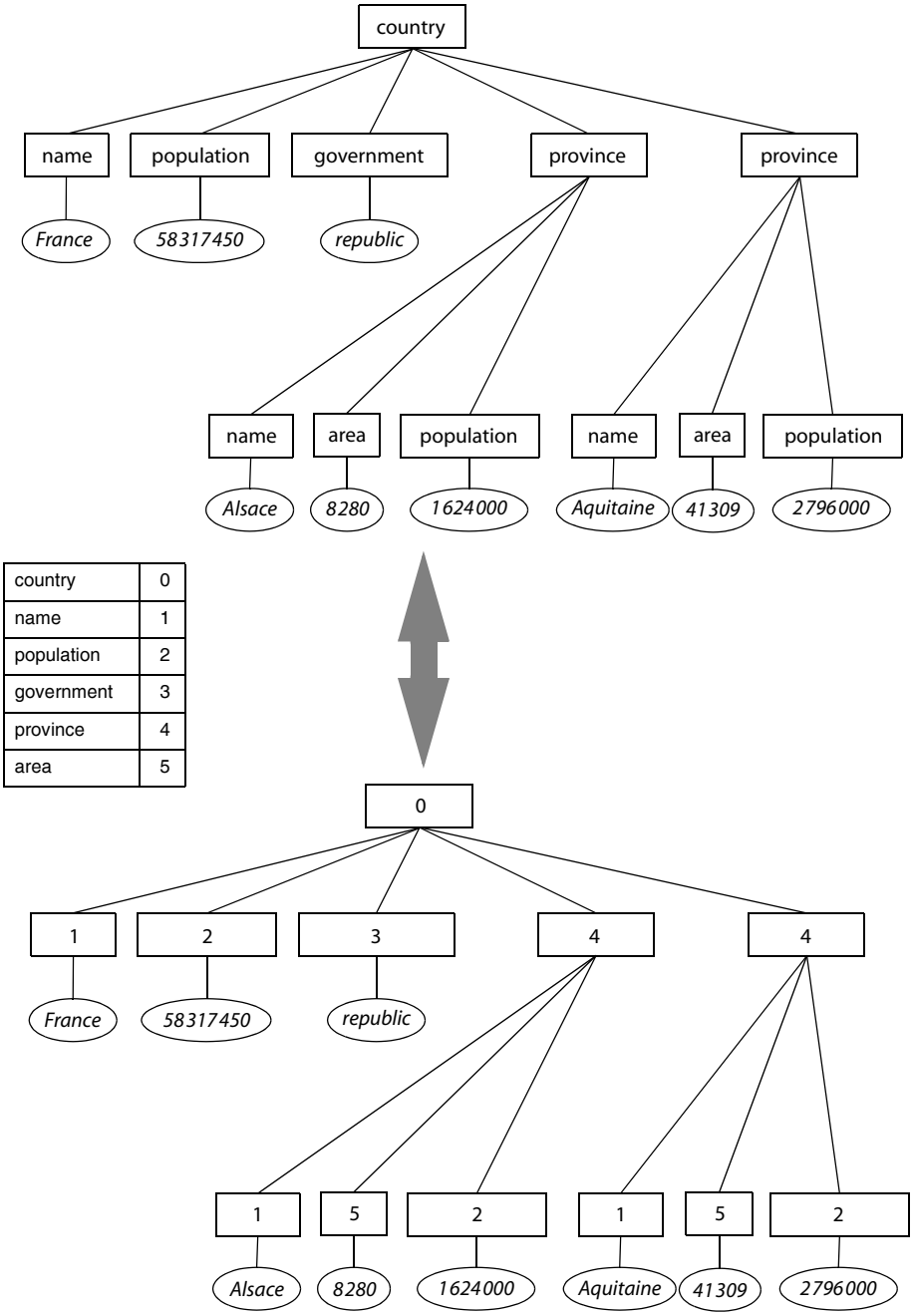


Abb. 4–19 *Baumrepräsentation und Namensersetzung in DB2 pureXML*

Auch wenn einige Namen mehrmals im XML-Dokument vorhanden sind, so wird nur ein Eintrag in die Zuordnungstabelle aufgenommen. Somit führen weitere Dokumente, die demselben XML-Schema genügen, auch nicht zu zusätzlichen Einträgen. Die Autoren geben an, dass aufgrund dieser Maßnahme, die einer Art tokenbasierter Datenkompression entspricht, die Tabelle im Allgemeinen nicht sehr umfangreich wird. Ein spezieller Cache sorgt zusätzlich für einen schnellen Zugriff.

Die eigentliche Speicherung der Dokumente geschieht wie auch bei relationalen Tabellen in einem Tablespace, der in Seiten aufgeteilt ist. Diese können zwischen 4 und 32 KB groß sein. Falls ein XML-Dokument nicht auf eine Seite passt, so können Teilbäume auf separate Seiten ausgelagert werden. Die Verknüpfung der Teilbäume geschieht über einen so genannten Regionsindex. Ziel des Speicheralgorithmus von DB2 ist es, ein Dokument auf so wenig Seiten wie möglich abzulegen. Zur besseren Ausnutzung der Seiten können auch Regionen verschiedener Dokumente auf einer Seite gespeichert werden. Abbildung 4–20 verdeutlicht das Verfahren.

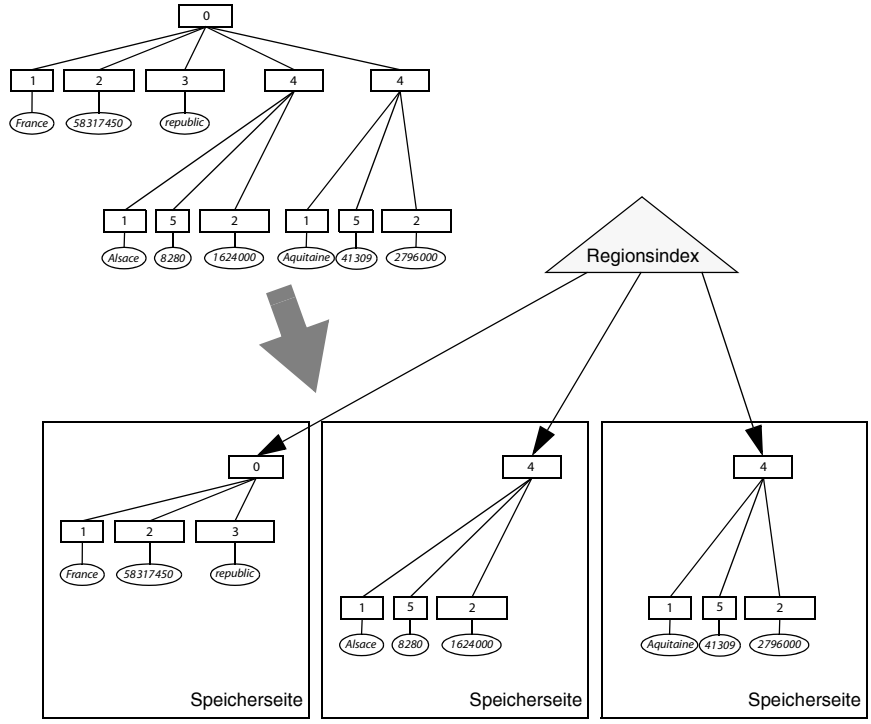


Abb. 4–20 *Funktionsweise des Regionsindex*

Der Regionsindex spart bei großen XML-Dokumenten Hauptspeicherplatz, da über den Index der benötigte Teilbaum ermittelt und nur dieser in den Speicher

geladen werden kann. Ein Einlesen des kompletten Dokuments ist somit in vielen Fällen nicht erforderlich.

Über die Speicherstruktur der Teilbäume auf den Seiten werden in [NL05] und [BCH06] keine genauen Angaben gemacht, auch nicht über den Aufbau des Regionsindex. Der Aufbau scheint eine gewisse Ähnlichkeit zu einem im Hauptspeicher abgelegten DOM-Baum zu besitzen. Zu jedem Knoten werden zusätzlich Typangaben festgehalten, sofern es sich um ein validiertes XML-Dokument handelt.

Um die Zugriffsgeschwindigkeit auf XML-Dokumente zu steigern, bietet DB2 zwei Arten von Indexen an: Volltextindexe sowie pfadspezifische Indexe. Zur Definition eines pfadspezifischen Index muss ein XPath-Ausdruck angegeben werden, der die zu indizierenden Knoten der in einer Tabellenspalte abgelegten XML-Dokumente definiert. Je indiziertem Wert werden im Index eine Pfad-ID, der Wert des Knotens, eine Row-ID sowie eine Node-ID festgehalten. Die Pfad-ID identifiziert den Pfad zum jeweiligen Knoten, die Row-ID beschreibt die Tabellenzeile, in der das zugehörige XML-Dokument gespeichert ist, und die Node-ID erlaubt die Ermittlung des betreffenden Knotens und der Region bzw. des Teilbaums, zu der dieser gehört. Bei der Indizierung werden Namespaces entsprechend berücksichtigt. Die Auswahl der Indexe kann mit Hilfe des in [EAZ08] beschriebenen Index-Advisor-Tools erfolgen.

Mit Hilfe eines Volltextindex kann die schnelle Textsuche in XML-Dokumenten ermöglicht werden. Dies ist hilfreich, wenn im Voraus nicht bekannt ist, nach welchen Attributen und Elementen häufig gesucht wird. Volltextindexe auf XML-Dokumenten entsprechen den gleichnamigen Indexen auf textwertigen Spalten relationaler Tabellen. In DB2 trägt die auch für XML-Dokumente zur Volltextsuche eingesetzte Erweiterung die Bezeichnung Net Search Extender (NSE). Die hierbei durch DB2 zur Verfügung gestellten Funktionalitäten sind mit denen von OracleText vergleichbar (siehe Abschnitt 4.1.2). Insbesondere können ein Thesaurus, Fuzzy-Suche, gewichtete Suche sowie die Sortierung nach Relevanz eingesetzt werden.

4.3.4 Native XML-Datenbank eXist

Der Begriff der nativen XML-Datenbank ist nicht klar definiert. Grundsätzlich versteht man darunter eine Datenbank, deren Schnittstelle zum Anwender derart gestaltet ist, dass XML-Dokumente abgelegt, wieder abgerufen und auch Teildokumente über eine Abfragesprache selektiert werden können. Einige Autoren, wie zum Beispiel Bourret in [Bou05b], erlauben dabei, dass als Datenspeicher im Hintergrund auch eine relationale Datenbank eingesetzt werden kann.

Hier soll der Begriff der nativen XML-Datenbank enger verstanden werden und nur Datenbanken umfassen, deren meist proprietäre Speichertechnik speziell auf die Anforderungen des XML-Datenmodells zugeschnitten ist und bei denen

keine Umsetzung in ein anderes Datenmodell zur Speicherung der Dokumente vorgenommen wird.

Bourret nennt in [Bou10] über 20 native XML-Datenbanken, die auf eigenen Speichertechniken aufbauen. Tamino war eines der ersten kommerziellen Produkte, das vom Hersteller, der Software AG, selbst als native XML-Datenbank bezeichnet wurde.[22] Unter den Open-Source-Lösungen, die aktuell noch gepflegt und weiterentwickelt werden, sind besonders die Projekte eXist, Sedna sowie BaseX der Universität Konstanz zu erwähnen.

Da die XML-Speichertechniken speziell für die jeweiligen Produkte entworfen wurden, unterscheiden sie sich dementsprechend stark voneinander. Bei den meisten kommerziellen Angeboten sind die technischen Details nicht öffentlich dokumentiert und demzufolge nicht bekannt. Aus diesen Gründen soll in diesem Abschnitt exemplarisch die Speichertechnik eines Produkts vorgestellt werden. Der Autor hat sich für die Open-Source-Datenbank eXist entschieden.

Kernaufgabe bei der Entwicklung eines geeigneten Verfahrens zur Speicherung von XML-Dokumenten in einem proprietären Format ist der Entwurf eines effizienten Kennzeichnungsschemas. Ohne ein derartiges Schema muss bei jeder Suche ein kompletter Durchlauf durch die Baumstruktur der XML-Dokumente vorgenommen werden. Dies ist insbesondere bei großen Dokumenten, die nicht vollständig in den Arbeitsspeicher geladen werden können, ungeeignet. Deshalb muss das Kennzeichnungsschema eine einfache Navigation im Baum unterstützen. Zusätzlich wird man nicht auf eine Indexstruktur verzichten können, die die Suche nach häufig abgefragten Pfaden beschleunigt.

In eXist erhält jeder Knoten eine Kennzeichnung, die aus der ID des zugehörigen Dokuments sowie einem Knoten-Label besteht. Ein Großteil der Anfragebearbeitung kann bereits durch Verwendung dieser beiden Angaben erledigt werden. Nur bei Bedarf wird auf die eigentlichen Daten des Knotens zugegriffen. Das eingesetzte Kennzeichnungsschema zur Vergabe der Knoten-Labels basiert auf den Arbeiten von Lee und Shin [LYY96, SJJ98], die als Grundlage noch SGML-Dokumente genutzt haben. Zur Modellierung eines Dokuments verwenden sie einen vollständigen k-Baum. k ist hierbei die maximale Anzahl an Kindknoten in der Baumdarstellung des Dokuments. Ein k-Baum ist ein Baum, dessen Knoten jeweils maximal k Kindknoten besitzen. Vollständig bedeutet hier, dass jede nichtleere Ebene die maximale Knotenzahl aufweist.[23] Das heißt, dass alle inneren Knoten genau k Kindknoten haben und alle Blattknoten die gleiche Tiefe aufweisen.

22. [CRZ03] liefert eine detaillierte technische Beschreibung des Tamino-Systems.
23. Es existiert auch eine abweichende Definition des Begriffs der Vollständigkeit in Bezug auf k-Bäume. Einige Autoren verstehen darunter, dass der Baum k-gleichverzweigt ist, das heißt, jeder Knoten hat entweder keine oder k Kindknoten.

Den Knoten des vollständigen k-Baums werden nun in Ebenenordnung aufsteigende Labels zugewiesen. Ebenenordnung bedeutet, dass der Baum von der Wurzel bis zu den Blättern ebenenweise von links nach rechts durchlaufen wird. Da in der XML-Baumdarstellung nicht jeder Knoten k Kindknoten aufweist, werden virtuelle Knoten eingefügt, um einen vollständigen k-Baum zu erhalten. Somit sind in dem Baum einige Knoten und Labels enthalten, die keine Informationen des dargestellten XML-Dokuments tragen, sondern nur als Platzhalter dienen. Das Verfahren wird auch als Ebenenordnungskennzeichnungsschema bezeichnet. Abbildung 4–21 zeigt ein XML-Dokument, das auf einen vollständigen 2-Baum abgebildet wurde. Die virtuellen Knoten sind gestrichelt dargestellt.

```
<province>
  <name>Hessen</name>
  <city>
    <name>Kassel</name>
    <population>201789</population>
  </city>
</province>
```

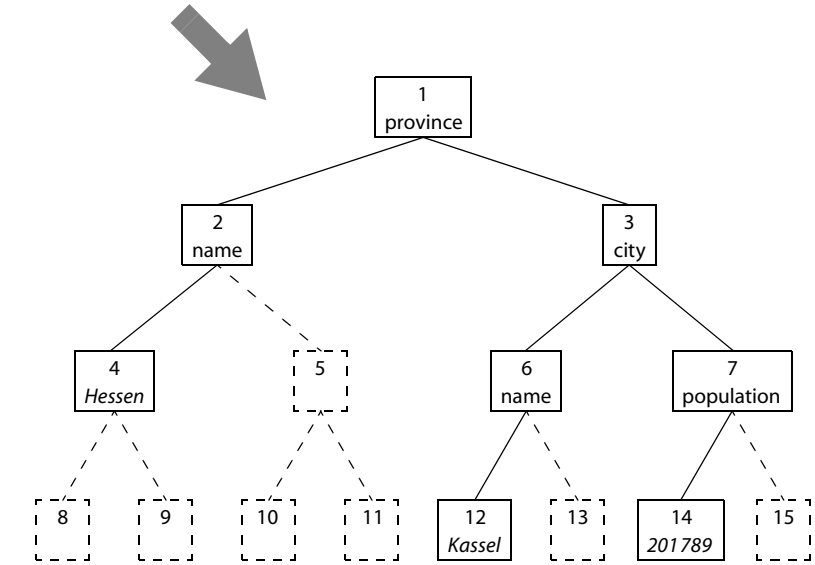


Abb. 4–21 *Ebenenordnungskennzeichnungsschema*

Die durch dieses Kennzeichnungsschema erzeugten Labels haben einige interessante Eigenschaften. Sei A ein Knoten mit dem Label a, der zu einem vollständigen k-Baum gehört.

Eltern-Kind-Beziehung: Sei B der Elternknoten von A. Für sein Label b gilt:

$$b = \left\lfloor \frac{a-2}{k} + 1 \right\rfloor$$

Sei der Knoten C_i das i-te Kind des Knotens A. Für das zugehörige Label c_i gilt:

$$c_i = k \cdot (a - 1) + i + 1$$

Durch diese Beziehungen lässt sich sehr gut in der Baumdarstellung entlang der XPath-Achsen navigieren. Das Speichern von zusätzlichen Angaben, die die Verknüpfungen zwischen den Knoten festlegen, ist nicht erforderlich. Alle nötigen Angaben sind in den Knotenlabels enthalten.

Nachteil des Verfahrens ist die Forderung, dass es sich um einen vollständigen k-Baum handeln muss, wodurch die Labels sehr schnell wachsen. Selbst bei Verwendung von 8-Byte-Integerzahlen können einige, in der Praxis vorkommende, große Dokumente nicht mit diesem Kennzeichnungsschema modelliert werden. So genügen zum Beispiel 8-Byte-Integerzahlen nicht für einen 10-Baum der Höhe 20.[24]

Um diese Einschränkungen zu beseitigen, kann man auf die Vollständigkeit des Baums verzichten und stattdessen die Anzahl der möglichen Kindknoten pro Ebene berechnen. Das heißt, dass wenn sich zwei Knoten auf der gleichen Ebene des Baums befinden, haben sie stets die gleiche Anzahl an Kindknoten. Die Kindknoten können wiederum virtuell sein, sofern an der entsprechenden Stelle nicht alle Knoten zur Modellierung der Struktur des jeweiligen XML-Dokuments benötigt werden.

Es kommt dadurch nicht mehr zu dem negativen Effekt, dass der komplette Baum stark anwächst, nur weil es auf einer Dokumentebene mehr Kindknoten je Element gibt als auf den anderen Ebenen. Der Datenbank eXist genügen durch diese Modifikation 8-Byte-Integerzahlen als Knotenlabels. Die Beziehungen zwischen den Knoten lassen sich weiterhin über die Knotenlabels ermitteln. Die dazu erforderlichen Formeln sind allerdings etwas komplizierter als die für vollständige k-Bäume oben angegebenen. Zusätzlich ist es für jeden Baum erforderlich, die Anzahl der Kindknoten pro Ebene in einem Array abzulegen, da diese Angaben für die Berechnungen verwendet werden müssen. Für das Beispieldokument aus Abbildung 4–21 wird in Abbildung 4–22 die so modifizierte Baumstruktur dargestellt.

Die Datenbank eXist speichert die Element-, Attribut- und sonstigen Knoten aller abgelegten XML-Dokumente in einer einzigen Datei. Diese hat den Namen `dom.dbx` und enthält für jeden Knoten einen Eintrag, der sein Label und seinen Inhalt speichert. Die Einträge werden zu Seiten zusammengefasst. Die Datei enthält zusätzlich einen Index in Form eines B^+-Baums, über den anhand der Dokument-ID und des gewünschten Knotenlabels auf einen Eintrag zugegriffen werden kann. Der Verweis erfolgt durch Seitennummer und Offset.

---

24. Die Höhe eines Baums ist definiert als die maximale Pfadlänge von der Wurzel zu den Blattknoten. Ein Baum der Höhe 20 hat also die maximale Pfadlänge 20 und besteht somit aus 21 Ebenen.

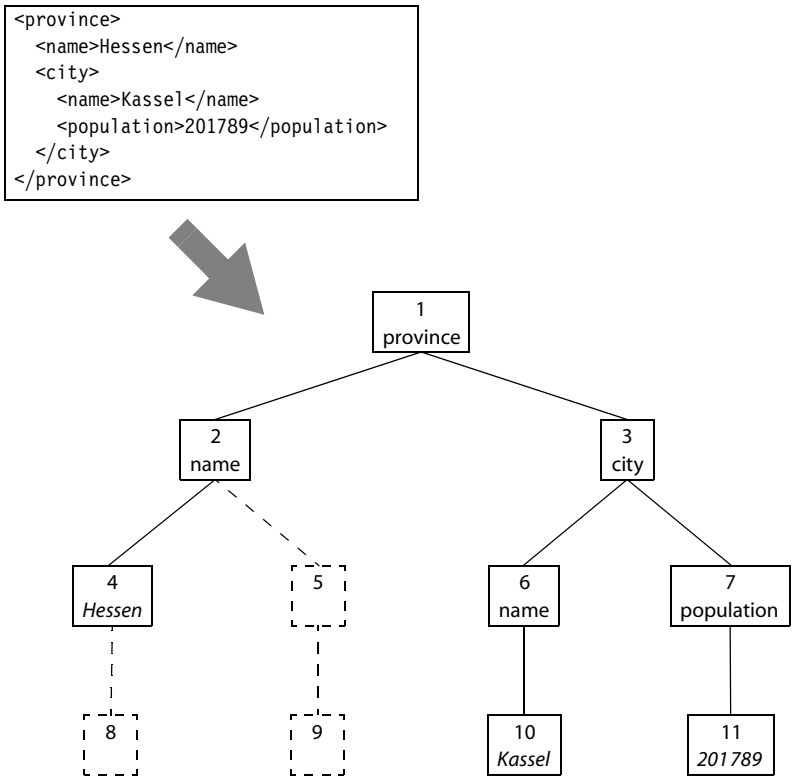


Abb. 4–22 *Modifiziertes Ebenenordnungskennzeichnungsschema*

Damit der Index nicht zu groß wird, werden in diesen nur die XML-Wurzelelemente und die jeweils direkt darunter liegenden Knoten aufgenommen. Der Zugriff auf alle weiteren Knoten geschieht durch Einstieg beim nächsten verfügbaren Vorfahren und anschließendes Durchlaufen des Baums in Dokumentenordnung. Dafür ist nur ein sequenzieller Zugriff auf die Datei erforderlich, da die Knoteneinträge in Dokumentenordnung abgelegt werden und auch mehrere Seiten, die zu dem gleichen Dokument gehören, nach Möglichkeit hintereinander gespeichert werden. Dadurch können abgelegte Dokumente auch effizient wieder serialisiert werden.

Neben der zentralen Datei `dom.dbx` besteht die Speicherarchitektur von eXist aus drei weiteren Dateien: `collections.dbx`, `elements.dbx` sowie `words.dbx`. Mit Hilfe der Datei `collections.dbx` werden hierarchisch strukturierte Dokumentgruppen verwaltet und die gespeicherten Dokumente diesen zugeordnet. Die Datei `elements.dbx` bildet einen Index aller Element- und Attributnamen. Sie dient dazu, um anhand des Namens alle Dokumente und zugehörigen Element- bzw. Attributknoten zu finden, die einen bestimmten Namen besitzen. Um auch

eine schnelle Suche über die in den Dokumenten vorkommenden Texte und Attributwerte durchführen zu können, wird die Datei `words.dbx` eingesetzt, die eine Volltextsuche ermöglicht. Die Volltextsuche kann zur Performanzsteigerung komplett oder auch nur für einige Dokumentteile abgeschaltet werden.

4.4 SQL/XML

In den letzten Jahren werden immer häufiger SQL-basierte Datenbanken genutzt, um XML-Dokumente abzulegen. Dabei tritt die Frage auf, wie sich XML-Dokumente in der Datenbank abfragen lassen. SQL beherrscht nur das komplette Auslesen von Attributen oder aber die textuelle Manipulation der Dokumente. Für XML wiederum existiert die Abfragesprache XQuery, die jedoch keine relationalen Tabellen und Spalten kennt. Die Lücke zwischen diesen beiden Bereichen lässt sich nicht einfach überbrücken, wenn Integration und nicht nur Interaktion erreicht werden soll. Grundproblem ist, dass XML ein geordnetes, inhärent hierarchisches Datenmodell darstellt, SQL jedoch die Sprache für das relationale Datenmodell ist und Elemente aus Relationenalgebra und -kalkül vereint [Dav03].

[SSB01] und auch Datenbankhersteller, unter anderem Microsoft[25] und Oracle, haben versucht, den gleichzeitigen Umgang mit XML- und relationalen Daten zu vereinfachen, indem Erweiterungen der Abfragesprache SQL vorgenommen wurden. Standardisiert wurden diese Bestrebungen durch die Aufnahme des neuen Teilstandards SQL/XML in die SQL:2003-Norm [ISO03]. In den Jahren 2006 und 2008 wurden Überarbeitungen und Erweiterungen vorgenommen [ISO06, ISO08]. Umfassende Darstellungen finden sich in den Arbeiten von Melton et al. [EM04, Mel05 und MB06] sowie in [Tür03]. SQL/XML stellt Funktionalitäten für die folgenden Bereiche zur Verfügung:

- XML-Darstellung von Abfrageergebnissen auf relationalen Daten
- Speicherung und Verarbeitung von XML-Daten in relationalen Datenbanken
- Abfrage von XML-Daten und Ausgabe der Ergebnisse im XML- oder relationalen Format
- Abbildung von relationalen Daten und Bezeichnern nach XML und zurück

In den folgenden Unterabschnitten wird SQL/XML näher besprochen. An dieser Stelle soll noch angemerkt werden, dass SQL/XML keine Vorgaben zum internen Speicherformat der XML-Daten in der relationalen Datenbank macht. Aus diesem Grund sind sowohl textbasierte Verfahren als auch modell- und strukturbasierte Vorgehensweisen denkbar. In der Praxis trifft man häufig auf eine textbasierte Speicherung, da diese am leichtesten zu implementieren ist. Für die Zukunft

25. Microsoft benutzt für seine in den SQL-Server eingebauten proprietären XML-Erweiterungen die Bezeichnung SQLXML (ohne Schrägstrich).

wäre eine Erweiterung des SQL/XML-Standards wünschenswert, sodass auch Update-Funktionalitäten eingeführt werden und Unterstützung für die Volltextsuche angeboten wird.

4.4.1 Der Datentyp XML

Besondere Bedeutung kommt in SQL/XML dem neu definierten Datentyp `XML` zu. Dieser Datentyp kann verwendet werden, um einen XML-Wert in einer Spalte einer relationalen Tabelle abzuspeichern. Unter einem XML-Wert versteht man hier nicht nur ein vollständiges XML-Dokument, sondern auch Fragmente. Damit fallen auch ein XML-Teilbaum (ein Element und die eventuell darin enthaltenen Unterelemente) oder ein XML-Wald (mehrere XML-Teilbäume hintereinander) darunter. SQL/XML verwendet im Umgang mit XML-Daten die mit XPath eingeführte und in Abschnitt 2.3 beschriebene Baumstruktur.

Durch die Angabe zusätzlicher Bedingungen kann der Datentyp `XML` genauer spezifiziert werden, etwa wenn als zulässige Werte nur vollständige XML-Dokumente erlaubt sein sollen. Dies geschieht über den primären und den sekundären Typmodifizierer. Für den primären Typmodifizierer sind die Angaben `SEQUENCE`, `CONTENT` oder `DOCUMENT` möglich. `SEQUENCE` ist der allgemeinste Typ. Er kann entweder `NULL` oder eine beliebige Sequenz im Sinne des XQuery-Datenmodells [W3C10d] aufnehmen. Unter einer derartigen Sequenz versteht man eine geordnete, unbeschränkte Folge von Knoten und atomaren Werten.

Der Typmodifizierer `CONTENT` liefert eine Spezialisierung von `SEQUENCE`. Außer `NULL` sind nur einelementige Sequenzen zugelassen, die aus einem Wurzel- bzw. Dokumentknoten bestehen. Unterhalb dieses Dokumentknotens dürfen beliebige andere Knoten angeordnet sein, also auch zum Beispiel mehr als ein Elementknoten, sodass nicht wohlgeformte XML-Dokumente, die mehr als ein XML-Wurzelelement aufweisen, erlaubt sind. Einen Schritt weiter geht der Typmodifizierer `DOCUMENT`. Er spezialisiert den Typ `CONTENT` dahingehend, dass unterhalb des Dokumentknotens genau ein Elementknoten vorhanden sein muss.

Der sekundäre Typmodifizierer des XML-Typs kann die Werte `UNTYPED`, `ANY` oder `XMLSCHEMA` aufweisen. Er kann nur verwendet werden, wenn der primäre Typmodifizierer `CONTENT` oder `DOCUMENT` lautet. Bei `SEQUENCE` ist keine weitere Typangabe möglich. Durch die Angabe `XMLSCHEMA` gefolgt von der Bezeichnung eines XML-Schemas wird festgelegt, dass die Instanzen des XML-Typs erfolgreich gegen dieses XML-Schema validiert werden müssen. Bei dieser Überprüfung erhält das XML-Dokument so genannte Typanmerkungen aus dem XML-Schema, die die Datentypen der einzelnen Elemente und Attribute genauer beschreiben. Der Typmodifizierer `UNTYPED` verhindert, dass ein XML-Wert diese Typanmerkungen besitzt. `ANY` gestattet es, dass die XML-Werte keine, einige oder vollständige Typanmerkungen aufweisen können.

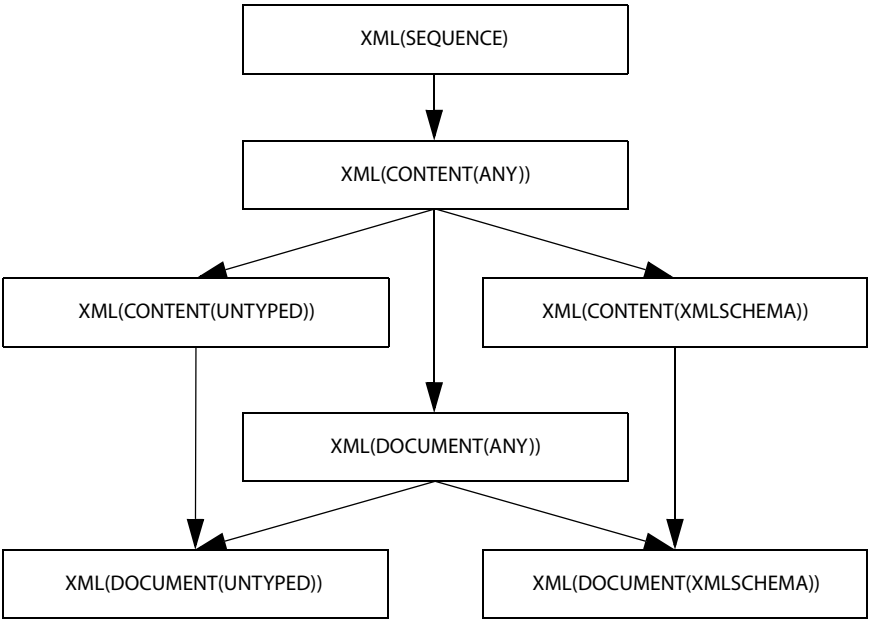


Abb. 4–23 *Primäre und sekundäre Typmodifizierer für den XML-Typ (nach [Mel05])*

Interessant ist, dass die Validierung nicht nur bei Verwendung des primären Typmodifizierers `DOCUMENT` möglich ist, sondern auch bei `CONTENT`. Dies bedeutet, dass die Validierung mit Hilfe eines XML-Schemas und die Verwendung der Typanmerkungen, zumindest eingeschränkt, auch bei nicht wohlgeformten XML-Dokumenten durchgeführt werden kann. Abbildung 4–23 (nach [Mel05]) gibt einen Überblick über die Typhierarchie, die durch den XML-Typ in Verbindung mit dem primären und sekundären Typmodifizierer aufgespannt wird.

4.4.2 XML-Funktionen

Ein Schwerpunkt von SQL/XML liegt auf der Transformation relationaler Daten in XML-Werte. Dazu bietet SQL/XML eine Reihe so genannter Publishing-Funktionen, die innerhalb von SQL-Kommandos eingesetzt werden können, um XML-Werte aus herkömmlichen relationalen Datenbanktabellen zu generieren. Die Arbeitsweise dieser Funktionen wird in der folgenden Aufzählung kurz beschrieben:

- `XMLELEMENT` dient der Erzeugung eines XML-Elements und in Verbindung mit `XMLATTRIBUTES` auch der Generierung der zugehörgen Attribute. Der Inhalt des Elements und die Attributwerte werden durch Wertausdrücke angegeben. Diese Wertausdrücke sind mit den direkt in der `SELECT`-Klausel verwendbaren

Ausdrücken vergleichbar. XML-Namensräume können dabei durch die Klausel XMLNAMESPACES deklariert werden.
- XMLFOREST kann benutzt werden, um mit nur einem Funktionsaufruf mehrere XML-Elemente, also einen XML-Wald, zu generieren.
- XMLCONCAT ermöglicht mehrere XML-Inhalte zu einem XML-Wald zu verknüpfen.
- XMLDOCUMENT, XMLTEXT, XMLCOMMENT und XMLPI werden benutzt, um einen Dokument-, Text-, Kommentar- bzw. Verarbeitungsanweisungsknoten zu erstellen.
- XMLAGG erlaubt in Verbindung mit der Klausel GROUP BY die Aggregation von XML-Werten. Die Aggregation innerhalb einer Gruppe findet statt, indem die enthaltenen Werte zu einem XML-Wald konkateniert werden.

Neben diesen Publishing-Funktionen existieren noch eine Reihe weiterer Funktionen und Prädikate, die den Umgang mit XML-Werten erleichtern. Diese sind:

- XMLPARSE und XMLVALIDATE werden zur Prüfung eines XML-Werts auf Wohlgeformtheit und Gültigkeit bezüglich eines Schemas verwendet. Bei der Validierung wird der XML-Wert mit Typanmerkungen versehen.
- XMLQUERY führt einen XQuery-Ausdruck aus und liefert einen XML-Wert als Ergebnis zurück.
- XMLEXISTS definiert ein Prädikat, das in der WHERE-Klausel von SQL-Anweisungen genutzt werden kann, um das Vorhandensein eines oder mehrerer durch einen XQuery-Ausdruck definierter Knoten in einem XML-Dokument zu testen.
- IS CONTENT und IS DOCUMENT sind Prädikate, mit denen festgestellt werden kann, ob ein XML-Wert vom Typ XML(CONTENT(ANY)) bzw. XML(DOCUMENT(ANY)) vorliegt.
- IS VALID dient zur Prüfung, ob ein XML-Wert einem XML-Schema genügt.
- XMLTABLE wandelt einen Teil eines XML-Dokuments, der durch einen XQuery-Ausdruck angegeben wird, in eine relationale Tabelle um. Der Inhalt der Spalten wird über XPath-Ausdrücke definiert, die Spaltentypen müssen explizit genannt werden.
- XMLCAST ist die Erweiterung der vom SQL-Standard bekannten Typumwandlung mittels CAST auf XML-Werte.
- XMLSERIALIZE konvertiert einen XML-Wert in eine Zeichenkette (CHAR, VARCHAR, CLOB oder BLOB).

Der Umgang mit den XML-Schemata wird in SQL/XML nicht genau festgelegt. XML-Schemata können in einem implementierungsabhängigen Verfahren beim DBMS registriert werden, sodass sie eine ID bekommen und die URI des Namensraums der Datenbank bekannt ist. Die registrierten Schemata können dann vom System zur Validierung der XML-Werte herangezogen werden. In

[Mel05] spricht sich Melton dafür aus, XML-Schemata grundsätzlich zu registrieren und nur registrierte Schemata für die Validierung zu verwenden. Anderenfalls kann es vorkommen, dass XML-Werte, die in der Datenbank bereits abgelegt sind, nicht mehr validiert werden können, da zwischenzeitlich das zugehörige Schema gelöscht wurde. Bei Registrierung kann die Datenbank dies verhindern und auch Änderungen an XML-Schemata unterbinden, die zugehörige vorhandene XML-Werte invalidieren würden.

4.4.3 Abbildungen zwischen SQL und XML

SQL/XML definiert auch eine Reihe so genannter Mappings von SQL nach XML und umgekehrt. Für jede Transformation ist zunächst eine Anpassung der verwendeten Zeichensätze durchzuführen. Auf Seite der relationalen Datenbanken können hierbei verschiedene Zeichensätze verwendet werden, auf XML-Seite geht der Standard vom Einsatz des Unicode-Zeichensatzes aus.

Zusätzlich ist die Wandlung von SQL-Bezeichnern in XML-Namen notwendig, da in SQL-Bezeichnern Sonderzeichen, wie `<`, `>` und `&`, enthalten sein dürfen, deren Vorkommen in XML-Namen nicht zugelassen ist. SQL/XML ersetzt diese Symbole durch eine Zeichenkette der Form `_x####_`, wobei `####` durch die Position des Zeichens im Unicode-Zeichensatz in hexadezimaler Notation substituiert wird. Diese Transformation kann bei Wandlung von XML nach SQL wieder rückgängig gemacht werden, allerdings werden dabei alle Zeichenketten der Art `_x####_` durch das jeweilige Unicode-Zeichen ersetzt. Es kann also nicht erkannt werden, ob die jeweilige Zeichenkette wirklich ein Unicode-Zeichen repräsentiert oder der XML-Name tatsächlich diese Zeichenfolge enthalten soll.

Neben diesem Mapping zwischen SQL-Bezeichnern und XML-Namen wird von SQL/XML auch eine Abbildung von SQL-Datentypen auf XML-Schema-Typen definiert. Durch den Einsatz von einfachen und komplexen XML-Datentypen sowie die Verwendung von Einschränkungen und Facetten ist eine sehr exakte Nachbildung sowohl der Basisdatentypen als auch der mengenwertigen Datentypen von SQL möglich. Zusätzlich wird der ursprüngliche SQL-Datentyp im XML-Schema durch Elemente angegeben, die aus einem eigens hierfür definierten SQL/XML-Namensraum stammen.

Bei der Wandlung von relationen Daten nach XML müssen auch Konvertierungen bei den Werten selbst durchgeführt werden. Dies steht in engem Zusammenhang mit der Transformation der Datentypen von SQL nach XML. Als Beispiele sind hier zu nennen, dass Dezimalwerte in SQL im Gegensatz zu XML stets einen Dezimalpunkt enthalten und dass in Zeichenketten die Symbole `<` und `>` bei der Transformation nach XML durch die vordefinierten Entitäten `<` und `>` ersetzt werden müssen. Des Weiteren bestehen bei Datums-, Zeit- und Intervallangaben größere Unterschiede in der Repräsentation der Werte. Zusätzlich definiert SQL/XML eine Zuordnungsvorschrift von atomaren XQuery-Daten-

typen zu SQL-Datentypen, um XQuery-Abfragen in Verbindung mit SQL einsetzen zu können.

Ein weiterer Bereich der Mappingdefinitionen von SQL/XML befasst sich mit der Repräsentation von relationalen Tabellen durch XML-Dokumente. Eine relationale Tabelle wird durch zwei XML-Dokumente dargestellt. Hierbei beschreibt ein XML-Schema-Dokument den Aufbau der Tabelle und die verwendeten Datentypen der Spalten. Das andere XML-Dokument nimmt den Inhalt der relationalen Tabelle auf, indem für jede Tabellenzeile ein XML-Element erzeugt wird, das für die einzelnen Spalten jeweils ein Unterelement mit dem zugehörigen Datenwert enthält. Die Abbildungsart lässt erkennen, dass hier ein datenbankorientiertes Verfahren gewählt wurde, bei dem Daten und Metadaten voneinander getrennt abgelegt werden. Für SQL/XML gelten strukturierte, Referenz- und `XML(SEQUENCE)`-Typen als „unmappable“. Spalten, die einen dieser Typen aufweisen, werden bei der Umwandlung ausgelassen und ignoriert.

Abbildung 4–24 zeigt zur Verdeutlichung des Transformationsprozesses eine Beispieltabelle und das zugehörige Dokument, das die Daten der Tabelle im XML-Format aufnimmt. SQL/XML weitet diese Repräsentation sogar auf komplette SQL-Schemata und SQL-Kataloge aus, sodass es möglich ist, eine komplette relationale Datenbank durch XML-Dokumente darzustellen.

CONTINENT

name	area
Europe	9562488
Asia	45095292
Australia	8503474
Africa	30254708
America	39872000

```
<CONTINENT>
   <row>
      <name>Europe</name><area>9562488</area>
   </row>
   <row>
      <name>Asia</name><area>45095292</area>
   </row>
   <row>
      <name>Australia</name><area>8503474</area>
   </row>
   <row>
      <name>Africa</name><area>30254708</area>
   </row>
   <row>
      <name>America</name><area>39872000</area>
   </row>
</CONTINENT>
```

Abb. 4–24 *XML-Repräsentation einer relationalen Tabelle*

Im Standard findet sich der Hinweis, dass durch SQL/XML für die Umwandlung von relationalen Tabellen, Schemata und Katalogen in XML-Dokumente sowie für die Abbildung von XML-Namen auf SQL-Bezeichner und umgekehrt keine Syntax definiert wird, die es erlaubt, diesen Transformationsprozess aufzurufen. Die Festlegungen in der SQL/XML-Norm sollen nur ein grundsätzliches Verfahren definieren, das Anwendungen bei Bedarf einsetzen können und das durch andere Standarddokumente referenziert werden kann.

5 Das Abbildungsverfahren und seine prototypische Implementierung

Nachdem in den vorangegangenen Kapiteln die Grundlagen von XML und objektrelationalen Datenbanken behandelt wurden, steht hier die Entwicklung und prototypische Implementierung eines bijektiven Abbildungsverfahrens von XML-Dokumenten auf SQL:2003-konforme Datentypen im Mittelpunkt. In Kapitel 6 folgen anschließend Messungen, die die Leistung des Verfahrens in Bezug auf Ausführungsgeschwindigkeit und Speicherbedarf untersuchen. Hierzu gehört auch die experimentelle Prüfung der Auswirkungen verschiedener Parameter auf die Effizienz der Methode.

Das aktuelle Kapitel gliedert sich wie folgt: Zunächst werden die Anforderungen an die einzusetzende Datenbank erarbeitet und mit den Möglichkeiten der in Kapitel 3 vorgestellten Datenbanken verglichen. Auf dieser Grundlage wird der Auswahlprozess einer geeigneten Datenbank und einer passenden Implementierungsumgebung beschrieben. Dabei wird das Konzept verfeinert und die mit der gewählten Datenbank tatsächlich realisierbaren Eigenschaften erarbeitet.

Es folgt die Behandlung der Datenstrukturen, die für die Abbildung entwickelt wurden, und die Besprechung der dabei aufgetretenen möglichen Designalternativen. Das Kapitel schließt mit der Vorstellung der Java-Applikation, die die Umsetzung der XML-Dokumente in objektrelationale Datenstrukturen und umgekehrt prototypisch vornimmt.

5.1 Eigenschaften und Anforderungen an das Abbildungsverfahren

Im Folgenden werden die grundsätzlichen Anforderungen vorgestellt, die das Abbildungsverfahren von XML-Dokumenten auf SQL:2003-konforme Datentypen und die dazu prototypisch zu entwickelnde Anwendung erfüllen sollen. Daraus ergeben sich als direkte Folgerung die notwendigen Eigenschaften des zu diesem Zweck einzusetzenden Datenbankmanagementsystems.

Eines der Hauptmerkmale von XML-Dateien sind die in der Dokumentstruktur enthaltenen Kollektionen, die an verschiedenen Stellen in Erscheinung treten, unter anderem in der Baumstruktur der Elemente eines Dokuments. Jedes Element kann beliebig viele Unterelemente besitzen. Die jeweiligen Unterelemente bilden eine geordnete Kollektion.

Daneben treten Kollektionen auch in Bezug auf die zu einem Element gehörenden Attribute auf. Jedes Element kann eine beliebige Anzahl an Attributen besitzen. Die Menge der Attribute je Element stellt wiederum eine Kollektion dar. Hierbei handelt es sich um eine ungeordnete Kollektion, da der XML-Standard keine Reihenfolge zwischen den Attributen definiert.

Um diese gegebene „natürliche“ Struktur eines XML-Dokuments möglichst wenig zu verändern, sollen diese Kollektionen bei der Abbildung auf SQL:2003-konforme Datentypen erhalten bleiben. Dies kann mit Hilfe eines modellbasierten Speicherverfahrens erreicht werden, das auf den im SQL:2003-Standard zur Verfügung stehenden Datentypen `LIST`, `SET`, `MULTISET` und `ROW` aufbaut. Durch den modellbasierten Ansatz entsteht ein Datenmodell, das sich für XML-Dokumente mit beliebigem XML-Schema eignet. Einzige Anforderung an die zu speichernden Dokumente ist, dass diese wohlgeformte XML-Dateien darstellen.

Ohne Kollektionstypen lässt sich der hierarchische Aufbau von XML-Dokumenten nur eingeschränkt abbilden. So findet zum Beispiel beim so genannten Schreddern (vgl. Abschnitt 4.2.1) eine Umsetzung in mehrere flache relationale Tabellen statt. Die Beziehung, in der die einzelnen Einträge zueinander stehen, zeigt sich erst, wenn diese durch die enthaltenen Fremdschlüssel miteinander verknüpft werden.

Bei textbasierten Speicherverfahren verhält es sich ähnlich. Einerseits haben sie den Vorteil, dass die im XML-Standard vorgesehene Repräsentation der Dokumente unverändert erhalten bleibt, andererseits ist jedoch die Baumstruktur der Daten nur nach einem aufwendigen Parsen des Textes zu erkennen.

Speicherverfahren, die eine Kantentabelle oder ein trickreiches Kennzeichnungsschema verwenden, liefern eine Struktur, die zwar geeignet ist, das ursprüngliche XML-Dokument wiederherzustellen, jedoch sind auch hier die hierarchischen Beziehungen zwischen den Elementen nicht direkt sichtbar. Es müssen wiederum Primär- und Fremdschlüssel in Beziehung gesetzt werden bzw. mehr oder weniger anspruchsvolle Vergleiche der Kennzeichnungslabels durchgeführt werden, um die Verwandtschaftsbeziehungen der Knoten zu erhalten.

Ein Verfahren dagegen, das direkt die Eltern-Kind-Relationen festhält, indem die Menge der Kindknoten bei jedem Elternknoten als Kollektion abgelegt wird, erlaubt es, den ursprünglichen Aufbau von XML-Dokumenten sehr viel direkter abzubilden.

Grundsätzlich ist daneben auch eine weitere sehr einfache Speicherung der Knotenbeziehungen denkbar: Anstatt je Knoten die Menge der Kindknoten als

Kollektion festzuhalten, ist es auch möglich, zu jedem Knoten den zugehörigen Elternknoten abzulegen. Dazu sind keine Kollektionen erforderlich.

Hauptunterschied ist, dass die Richtung, in der die Kanten des Baums modelliert werden, hierbei nicht von der Wurzel zu den Blattknoten führt, sondern umgekehrt, also von den Blattknoten zur Wurzel des Baums. Vorzuziehen ist die Richtung hin zu den Blattknoten, da diese der „natürlichen Baumrichtung" entspricht: zuerst kommt die Wurzel, danach die Blätter. Eine doppelte Verknüpfung zwischen Eltern- und Kindknoten ist abzulehnen, da dies zu Redundanz führt und somit das Auftreten von Inkonsistenzen ermöglicht, was unbedingt vermieden werden sollte.

Diese Argumentation wird auch von der Mehrzahl der in der Praxis auftretenden Abfragen auf XML-Dokumenten unterstützt. Meist werden Kindelemente unterhalb eines vorgegebenen absoluten Pfades, der von der Wurzel ausgeht, gesucht. Nur in seltenen Fällen ist es erforderlich, zu einem gegebenen Knoten direkt den Elternknoten zu bestimmen.

Die Gründe für eine Speicherung der Baumkanten in Richtung zu den Blättern werden auch besonders deutlich an der zweiten Stelle, an der Verknüpfungen zwischen Eltern- und Kindknoten im XML-DOM-Baum in Erscheinung treten. Für jedes Element müssen die zugehörigen Attribute gespeichert werden. Die Attribute bilden eine Menge von Name-Wert-Paaren. Die Reihenfolge ist weder definiert noch von Bedeutung. Jedes Name-Wert-Paar selbst stellt einen strukturierten Typ bzw. einen `ROW`-Typ dar, der sich aus den beiden Angaben Attributname und Attributwert zusammensetzt. Die Attribute eines Elements bilden somit eine ungeordnete Kollektion gleichartiger strukturierter `ROW`-Typen.

Eine Speicherstruktur, die nicht zu jedem Element die Attribute festhält, sondern zu auftretenden Attributen das zugehörige Element, bietet in der Mehrzahl der Fälle nur Nachteile. Meist interessiert ein bestimmtes Attribut eines vorgegeben Elements. Eine Anwendung, in der ein Attribut gegeben ist und dann die Ermittlung des zugehörigen Elements, also des Elternknotens, erforderlich ist, tritt in der Praxis nur selten auf.

Neben den Verbindungen zu den jeweiligen Unterelementen und der Menge der Attribute müssen je Element auch die in diesem enthaltenen Texte abgelegt werden. In vielen XML-Dokumenten tritt pro Element nur ein Textblock auf. Es ist jedoch auch möglich, dass sich der Text eines Elements auf mehrere Blöcke aufteilt, die jeweils durch Kindelemente unterbrochen werden. Hierbei ist es erforderlich, nicht nur den Inhalt der einzelnen Texte selbst zu speichern, sondern auch die Reihenfolge der Blöcke und in welcher Abfolge die Kindelemente einzuordnen sind. Die Textblöcke lassen sich hierbei als spezielle Kindelemente auffassen und auch analog zu diesen ablegen. Dabei muss sichergestellt werden, dass die Menge der Verknüpfungen zu den Kindelementen und Textblöcken als geordnete Aufzählung, mit anderen Worten als Liste, modelliert wird, damit die

ursprüngliche XML-Dokumentstruktur unverändert wiederhergestellt werden kann, also DOM-Fidelity gegeben ist.

Bei der Behandlung der Element- und Attributnamen sowie der Attributwerte muss beachtet werden, dass diese nach dem XML-Standard prinzipiell eine beliebige Länge aufweisen können. Da diese Forderung in der Praxis meist zu Schwierigkeiten bei der Abbildung der XML-Dokumente in alternative Speicherstrukturen führt, ist es akzeptabel, hier eine maximale Länge dieser Zeichenketten vorzugeben. Attribut- und Elementnamen, die länger als etwa 30 Zeichen sind, treten nur äußerst selten auf. Jedoch sollten die Attributwerte nicht so stark begrenzt werden. Hier ist es eher ratsam, auch relativ lange Zeichenketten zuzulassen. Eine Obergrenze von mindestens 1000 Zeichen erscheint angemessen.

Bei Texten dagegen muss es möglich sein, auch sehr lange Zeichenketten abzulegen. Eine Beschränkung auf eine bestimmte Maximallänge sollte nicht erfolgen. Dies gilt sowohl für jeden einzelnen Textblock als auch für die Gesamtlänge aller Texte eines Elements. Nur so lassen sich auch komplette im XML-Format vorliegende Bücher, deren Inhalt nur schwach in Kapitel und einzelne Unterabschnitte strukturiert ist, ablegen.

Zielsetzung dieser Arbeit ist die Abbildung von XML-Dokumenten auf SQL:2003-konforme Datentypen. Dies beinhaltet insbesondere die ausschließliche Verwendung von standardisierten Typen und Objekten. Der Einsatz von proprietären Erweiterungen und Sonderfunktionalitäten einzelner Datenbankmanagementsysteme soll nach Möglichkeit vermieden werden.

Dies erlaubt eine, zumindest teilweise gegebene, Portabilität des Abbildungsverfahrens über Datenbankgrenzen hinweg. Die Verwendung von standardisierten SQL-Konstrukten gestattet außerdem den direkten Zugriff auf die in der Datenbank abgelegten Datenstrukturen, deren Manipulation und Änderung. Des Weiteren ist es unter diesen Voraussetzungen möglich, leicht auf die Daten aus Programmiersprachen heraus zuzugreifen. Dabei können bewährte, standardisierte Zugriffsmethoden, wie zum Beispiel JDBC, eingesetzt werden.

Beim Umgang mit den in der Datenbank hinterlegten Strukturen können gespeicherte Prozeduren (engl. „stored procedures“) unterstützend zum Einsatz kommen. Mit dieser Technik können abgelegte Werte direkt im Datenbankmanagementsystem verarbeitet werden, ohne die Daten zunächst zum Client zu transferieren. Gespeicherte Prozeduren eignen sich insbesondere für Operationen, die häufig benötigt werden oder die zur Sicherstellung der Konsistenz des Datenbestands erforderlich sind.

Die Prozeduren können auch zum Einsatz kommen, um komplette XML-Dokumente oder auch nur Dokumentfragmente aus der Datenbank auszulesen und zum Client zu übertragen. Der Prozess des Auslesens der einzelnen Elementknoten und die Serialisierung des Dokuments zur XML-Textdarstellung kann von diesen Prozeduren übernommen werden und muss damit nicht von einem Applikationsserver oder vom Client selbst durchgeführt werden.

Ein Problem beim Einsatz von gespeicherten Prozeduren ist, dass bei den meisten DBMS Einschränkungen bezüglich der Durchführung bestimmter Operationen und des Einsatzes mancher SQL-Kommandos existieren. Nachteilig ist ebenfalls, dass keine leichte Umsetzung der Prozeduren von einem Datenbankmanagementsystem auf ein anderes möglich ist, da die jeweils verwendeten Programmiersprachen untereinander nicht kompatibel sind. Werden statt der gespeicherten Prozeduren SQL-Anweisungen auf dem Client eingesetzt, so lassen sich diese deutlich problemloser an ein anderes DBMS anpassen.

In diesem Zusammenhang ist auch zu prüfen, ob es sinnvoll ist, an Stelle von einfachen Mengen, Zeilen und atomaren Werten, Objekte zu speichern. Durch Objektfunktionalitäten und -methoden sowie Vererbung, Objektidentität und Objektreferenz erhalten die gespeicherten Instanzen selbst gewisse Fähigkeiten, die es ihnen erlauben, Berechnungen und Verknüpfungen durchzuführen. Hierbei kann es sich zum Beispiel um die Transformation der Darstellungsweise (u. a. Serialisierung als Zeichenkette) handeln. Der Einsatz von Objekten sollte mit den meisten Datenbankmanagementsystemen möglich sein, da die objektrelationalen Funktionen bereits seit längerer Zeit Bestandteil des SQL-Standards sind.

Da ein ausgereiftes Datenbankmanagementsystem als Grundlage für die Entwicklung des Abbildungsverfahrens verwendet werden soll, stehen natürlich sämtliche unterstützten Funktionalitäten zur Verfügung. Dies gestattet insbesondere die Nutzung des Transaktionskonzepts der Datenbank und garantiert somit die ACID-Eigenschaften. Dadurch lässt sich sicherstellen, dass alle Manipulationen an den abgelegten Datenstrukturen geordnet ablaufen. Es kann dadurch verhindert werden, dass Änderungen nur teilweise durchgeführt werden und es somit zu einem inkonsistenten Datenbestand kommt.

Alternativ kann diese Funktionalität auch durch ein Middleware-System bzw. einen Applikationsserver bereitgestellt werden. Allerdings ist der Aufwand bei der Nutzung der Datenbankfunktionen geringer, und es kann auch davon ausgegangen werden, dass diese Systeme ausgereifter und fehlerfreier sind als eine neu zu entwickelnde derartige Zwischenkomponente.

Auf Grundlage dieses Anforderungskatalogs, den das Abbildungsverfahren für XML-Dokumente erfüllen soll, lässt sich nun untersuchen, welches DBMS für diese Aufgabe am besten geeignet erscheint. Nachdem das zu nutzende System ausgewählt wurde, kann auf dessen Basis ein detailliertes Datenmodell entworfen werden und die genauen Funktionen der einzelnen Komponenten festgelegt werden. Zusammenfassend ergeben sich die folgenden Anforderungen an die einzusetzende Datenbank:

- Kompatibilität zum SQL:2003-Standard
- Unterstützung der Kollektionstypen `LIST`, `SET` und `MULTISET` und des Strukturtyps `ROW`

- Weitreichende Verknüpfungsmöglichkeiten zwischen den Kollektionstypen untereinander und dem Strukturtyp (Typ-Orthogonalität)
- Objektrelationale Eigenschaften: Objekttypen und -tabellen, Methoden, Vererbung, Objektidentitäten und -referenzen
- Stored Procedures und eine zugehörige Datenbanksprache, die auch mit Kollektionstypen umgehen kann
- Leichter Zugriff aus Programmiersprachen heraus, zum Beispiel in Java über JDBC als Programmierschnittstelle
- Ausgereifte Abfrageverarbeitung, die über einen Optimierer möglichst effiziente Ausführungspläne erstellt
- Transaktionsunterstützung, ACID-Eigenschaften
- Gute Skalierbarkeit und Unterstützung auch großer Datenmengen
- Einfache Administrierbarkeit, Robustheit, Zuverlässigkeit sowie gute Software-Pflege durch den Hersteller

5.2 Auswahl einer geeigneten Datenbank

Im vorherigen Abschnitt wurden die Anforderungen an das zu entwickelnde Abbildungsverfahren von XML-Dokumenten auf SQL:2003-konforme Datentypen besprochen. Auf dieser Basis soll nun eine geeignete Datenbank ausgewählt werden, die im Rahmen der prototypischen Implementierung eingesetzt wird.

In Kapitel 3 findet sich die ausführliche Untersuchung verschiedener Datenbankmanagementsysteme hinsichtlich ihrer jeweiligen Unterstützung von Kollektionstypen. Die Ergebnisse wurden in Abschnitt 3.3.4 zusammengefasst. Wenn man auf dieser Grundlage nach einer geeigneten Datenbank sucht, die möglichst viele der in Abschnitt 5.1 aufgelisteten Eigenschaften erfüllt, kommt man zunächst zu dem Ergebnis, dass die meisten Spezialdatenbanken und Prototypen ungeeignet sind. Derartige Datenbankentwicklungen sind stets für einen bestimmten Anwendungszweck optimiert. Die hier geforderten Fähigkeiten decken jedoch ein breites Spektrum an Eigenschaften ab, das diese Produkte nicht vollständig erfüllen können. Daher gelangt man recht schnell zu der Erkenntnis, dass sich eines der kommerziellen Produkte der großen Datenbankhersteller möglicherweise am besten für die gestellte Aufgabe eignet.

Große Open-Source-Datenbanken bieten in den meisten Fällen keinen mit kommerziellen Produkten vergleichbar großen Funktionsumfang. So unterstützt MySQL in der Version 5.5 keinerlei Kollektionstypen. PostgreSQL steht etwas besser da: Version 9.1 bietet sowohl einen Array- als auch einen Composite-Typ. Diese sind mit den Typen `LIST` und `ROW` vergleichbar. Allerdings stehen die Typen Menge und Multimenge in PostgreSQL nicht zur Verfügung.

Aus den genannten Überlegungen heraus beschränken sich die weiteren Untersuchungen auf die Datenbanken DB2, Oracle und Informix. Diese drei Sys-

teme werden in den folgenden Abschnitten auf ihre Eignung für die gestellte Aufgabe hin untersucht.

5.2.1 Beurteilung von DB2

Die Datenbank DB2 genügt den gestellten Anforderungen nicht. Wie in Abschnitt 3.3.2 ausführlich dargelegt, bietet die Datenbank keine Kollektionstypen, die im Rahmen von Tabellendefinitionen eingesetzt werden können. Somit stehen weder Listen noch Mengen und auch keine Multimengen zur Verfügung. Dies gilt auch weiterhin für die zur Zeit aktuelle Version 9.7 der DB2-Datenbank.

In Tabellen ist der Einsatz von strukturierten Typen, also `ROW`- bzw. Objekttypen, möglich, die sich aus Basisdatentypen und auch aus strukturierten Typen zusammensetzen lassen. Mit dieser Möglichkeit kann zwar eine Schachtelung erzeugt werden, allerdings bleibt diese flach, da darin keine mengenwertigen Typen zur Verfügung stehen.

Grundsätzlich unterstützt zwar DB2 mengenwertige Typen, genauer `ARRAY`-Typen. Allerdings lassen sich diese nicht als Typen von Tabellenspalten verwenden. `ARRAY`-Typen können nur für die Verarbeitung von Daten innerhalb von gespeicherten Prozeduren, die in der Sprache SPL geschrieben sind, genutzt werden.

Aus diesen Gründen ist bei Einsatz der IBM-Datenbank DB2 keine Abbildung von XML-Dokumenten auf kollektionswertige Typen von SQL:2003 möglich. Sollen XML-Dokumente mit DB2 abgelegt werden, so muss entweder auf das in Abschnitt 4.3.3 beschriebene DB2-eigene, proprietäre Verfahren zurückgegriffen oder eine Transformation vorgenommen werden, sodass die XML-Daten flach gespeichert werden können. Dies genügt dann aber nicht den in Abschnitt 5.1 beschriebenen Anforderungen an das zu entwickelnde Abbildungsverfahren.

5.2.2 Beurteilung von Oracle

Zunächst fiel die Entscheidung, die Abbildung von XML-Dokumenten auf SQL:2003-konforme Datentypen mit Hilfe der Datenbank Oracle zu implementieren. Gründe dafür waren, dass die Datenbank im Fachgebiet von Prof. Wegner an der Universität Kassel bereits eingesetzt wurde und der Autor mit der Verwaltung der Datenbank vertraut war.

Es war auch durchaus so, dass sich Oracle grundsätzlich für die gestellte Aufgabe eignete. Wie in Abschnitt 3.3.1 dargestellt, bietet die Datenbank eine relativ große Auswahl an Kollektionstypen. Unter Verwendung von Oracle konnte eine erste Applikation entwickelt werden, die die gewünschte Umsetzung prototypisch durchführt.

Im Zuge der Weiterentwicklung der Anwendung wurde die Eignung anderer DBMS eingehend untersucht. Dabei fiel die sehr fortgeschrittene Unterstützung von Kollektionstypen bei Informix auf, die insbesondere auch den weitreichenden orthogonalen Einsatz dieser Typen erlaubt. Hier bietet Informix eine deutlich bessere Unterstützung als Oracle.

Weitere Argumente, die schließlich dazu führten, die Entwicklung des Abbildungsverfahrens nicht mehr auf Grundlage von Oracle, sondern in Verbindung mit Informix durchzuführen, sollen im Folgenden kurz skizziert werden. Beim direkten Vergleich von Oracle und Informix fällt auf, dass die Installation und die Konfiguration von Informix leichter möglich ist. Dies bedeutet jedoch nicht, dass die Datenbank weniger Möglichkeiten als Oracle bietet. Die gesamten Verwaltungsfunktionen erscheinen jedoch einheitlicher und auch besser bedienbar. Die Suche nach Konfigurationsfehlern gelang ebenfalls besser.

Ein Grund für diesen Unterschied zwischen den beiden Systemen könnte darin zu suchen sein, dass Oracle im Zuge der fortschreitenden Entwicklung der Datenbank um immer neue Funktionen erweitert wurde. Diese Module sind meist derart mit dem Kernsystem verzahnt, dass es nicht möglich ist, diese nicht zu installieren. Viele dieser Module sind somit untrennbar mit der Datenbank verbunden, oder ihre Nicht-Verwendung würde eine Reihe manueller, schwer zu überschauender und deshalb fehlerträchtiger Eingriffe in das System erforderlich machen. Dennoch sind diese Zusatzpakete in der Mehrzahl der Fälle nicht derart integriert, dass sie einen echten Bestandteil des Kernsystems bilden. Deshalb bieten die Module häufig separate Konfigurationsmöglichkeiten und besitzen eigene Verwaltungsdaten, was die Einstellung zusätzlich erschwert.

In den letzten Jahren wurden hier zwar einige Verbesserungen erreicht, unter anderem bietet der so genannte Enterprise Manager mittlerweile eine relativ einheitliche, grafische Oberfläche zur Verwaltung des gesamten Systems. Nachteil ist jedoch, dass diese Konfigurationsoberfläche selbst wiederum ein mehr oder weniger eigenständiges Modul bildet, das über Schnittstellen mit den bisherigen Verwaltungsmöglichkeiten interagiert. Dies führt zu dem Effekt, dass sich das System gut verwalten lässt, solange keine unerwarteten Schwierigkeiten auftreten. Sind diese jedoch vorhanden, entsteht durch die Oberfläche nur eine weitere Stelle, an der die Fehler zu suchen sind und die eigentliche Ursache kann zunächst verdeckt bleiben.

In Bezug auf die weiteren Anforderungen an die Datenbank, wie Robustheit, Zuverlässigkeit, Transaktionsunterstützung und ACID-Eigenschaften, sind keine wesentlichen Unterschiede zwischen Oracle und Informix zu erkennen. Auch werden von beiden Systemen Programmierschnittstellen zum Zugriff auf die Datenbank angeboten.

5.2.3 Beurteilung von Informix

Wie schon im vorangegangenen Abschnitt angedeutet, stellte sich heraus, dass das Datenbankmanagementsystem Informix in Bezug auf die hier zu erstellende Applikation am besten geeignet ist. Hauptgrund ist die breite Unterstützung von Kollektionstypen. Es stehen die Datentypen `LIST`, `SET`, `MULTISET` und `ROW` zur Verfügung, unter denen Informix vollständige Orthogonalität bietet.

Darüberhinaus ist Informix gut zu administrieren. Die Verwaltung kann mit Hilfe einer Reihe von Kommandozeilentools durchgeführt werden. Diese erlauben durch ein einheitliches, gut strukturiertes Bedienkonzept eine rasche Durchführung der meisten Administrationsaufgaben. Daneben steht noch ein grafisches, web-basiertes Tool zur Verfügung, mit dessen Hilfe einige Aufgaben komfortabler durchgeführt werden können.

Das gesamte DBMS macht einen übersichtlichen Eindruck. Es existieren nicht derart viele Zusatzpakete wie es bei Oracle der Fall ist. Dies erleichtert den Verwaltungsaufwand deutlich. In Bezug auf Stabilität und Zuverlässigkeit kann das System mit den von Oracle bekannten Werten verglichen werden.

Es fällt jedoch auf, dass Informix keinen Objekttyp anbietet. Dies bedeutet auch, dass keine Objektmethoden, keine Objektidentitäten und Referenzen auf Objekte existieren. Dies ist jedoch für die hier zu implementierende Applikation kein gravierender Nachteil, da für die Abbildung ausschließlich relationale Funktionalitäten, erweitert um Kollektionstypen, benötigt werden. Die Programmiersprache SPL, mit der in Informix gespeicherte Prozeduren erstellt werden, bietet aufgrund der fehlenden Objektunterstützung keine Möglichkeit, Daten und Code zu kapseln. Ansonsten ist sie jedoch vom Funktionsumfang her mit der Oracle-Variante PL/SQL vergleichbar.

Aus all diesen Überlegungen heraus erweist sich Informix als das für die Aufgabe am besten geeignete System. Einige durchgeführte praktische Versuche bestätigten diese Erkenntnis und führten zum Entschluss, die Entwicklung des prototypischen Abbildungsverfahrens für XML-Dokumente auf Basis des Informix-Systems fortzusetzen. Da das Abbildungsverfahren nicht nur theoretisch entworfen, sondern auch tatsächlich implementiert werden soll, ist diese Festlegung von entscheidender Bedeutung. Auf dieser Grundlage können nun die Details des Verfahrens festgelegt und die dazu passenden, von Informix angebotenen Datentypen und anderen Funktionalitäten ausgewählt werden. Ein späterer Wechsel des Datenbankmanagementsystems ist wegen der nötigen Änderungen am Typsystem und den Zugriffsfunktionen aufgrund der abweichenden SQL:2003-Implementierungen nur mit größerem Aufwand möglich.

In den sich anschließenden Abschnitten wird zunächst die Implementierungsumgebung vorgestellt, bevor danach das Abbildungsverfahren selbst und schließlich die entwickelte prototypische Applikation zum Speichern und Wiederauslesen von XML-Dokumenten detailliert besprochen werden.

5.3 Implementierungsumgebung

An dieser Stelle wird das Entwicklungsumfeld beschrieben, das einerseits aus dem Datenbankserver und andererseits aus den Software-Technologien besteht, die eingesetzt wurden, um die Applikation zu implementieren.

5.3.1 Datenbankserver

In den vorangegangenen Abschnitten wurde der Auswahlprozess erläutert, der zu der Entscheidung führte, die Datenbank Informix der Herstellers IBM zu verwenden.

Die exakte Bezeichnung der eingesetzten Datenbank ist Informix Dynamic Server (IDS). Im Oktober 2010 wurde die Version 11.7 veröffentlicht. Die Datenbank existiert in den Editionen Express, Workgroup, Developer und Enterprise. Die Varianten unterscheiden sich durch die Anzahl der unterstützten Prozessoren und die Größe des möglichen Arbeits- und Datenbankspeichers. Außerdem gibt es noch Abweichungen bei Zusatzfunktionen, wie zum Beispiel Unterstützung für Hochverfügbarkeitslösungen und bestimmte Replikationsfunktionen. Für die Implementierung wurde die Enterprise Edition, die den größten Funktionsumfang bietet, als 64-Bit-Applikation eingesetzt. Zu Beginn der Entwicklung wurde die Datenbankversion 11.5, zuletzt Version 11.7 genutzt. Die Unterschiede zwischen den beiden Versionen liegen im Wesentlichen in den Bereichen Cluster-Einsatz, Data-Warehouse-Funktionen sowie Debugging und Administration. Die neuen Fähigkeiten der Version 11.7 haben jedoch keinen Einfluss auf den Einsatz im Rahmen der Implementierung des Abbildungsverfahrens.

Das Datenbanksystem wurde auf dem Betriebssystem Windows Server 2008 R2 Enterprise (64 Bit) eingerichtet. Die technischen Details des Datenbankservers finden sich in Abschnitt 6.1, dort insbesondere in Tabelle 6–1. Für die Konfiguration und Verwaltung der Informix-Datenbank können die mitgelieferten Kommandozeilentools eingesetzt werden. Etwas komfortabler ist die Nutzung des Tools Informix Server Administrator (ISA) sowie dessen Nachfolgers OpenAdmin Tool (OAT). Beide Werkzeuge arbeiten auf Basis eines Web-Servers und ermöglichen die Konfiguration der Datenbank über eine grafische Oberfläche, die in einem Web-Browser dargestellt wird. Durch diese Architektur ist auch der problemlose Zugriff auf die Verwaltungsfunktionen von anderen Computern aus möglich.

5.3.2 Implementierungswerkzeuge

Die Transformationsapplikation wurde mit Hilfe der Programmiersprache Java entwickelt, wobei das Java Development Kit (JDK) in der Version 6 eingesetzt wurde. Als grafische Entwicklungsumgebung wurde die IDE Eclipse verwendet.

Sowohl beim JDK als auch bei der Entwicklungsumgebung wurde die jeweilige 64-Bit-Version genutzt.

Um auf die Datenbank zugreifen zu können, ist ein entsprechender Schnittstellentreiber notwenig. Für Java-Applikationen wird in der Mehrzahl der Fälle die JDBC-Schnittstelle eingesetzt. JDBC ist eine 1997 von Sun definierte Menge an Klassen, die der Interaktion mit (inbesondere relationen) Datenbanken dienen. Für die jeweils genutzte Datenbank muss ein passender JDBC-Datenbanktreiber verwendet werden. Dieser wird meist vom Hersteller der Datenbank oder einem Drittanbieter entwickelt. Gegenüber den von Sun standardisierten Klassen bieten die Treiber meist noch einige Erweiterungen, die auf die speziellen Bedürfnisse und Möglichkeiten der jeweiligen Datenbank zugeschnitten sind.

Für die vorliegende Arbeit wurde der Original-Informix-JDBC-Treiber in der Version 3.70 verwendet. Es handelt sich dabei um einen Typ-4-Treiber. Dies bedeutet, dass es sich bei dem JDBC-Treiber um einen so genannten Pure Java Driver handelt, der die Kommandos direkt in die jeweiligen Befehle des Datenbankmanagementsystems übersetzt und über das Netzwerk versendet. Bei diesem Treibertyp ist deshalb keine Installation zusätzlicher Programmbibliotheken auf dem Client erforderlich.

Mit Hilfe des JDBC-Treibers kann die Java-Applikation Verbindungen zur Datenbank aufbauen, SQL-Kommandos versenden und die zurückgelieferten Ergebnisse abrufen. Gespeicherte Prozeduren können ausgeführt und mehrmals benötigte Kommandos vorbereitet werden. Dies bedeutet, dass der Datenbank die Struktur des Befehls bekanntgemacht wird, ohne jedoch die konkreten Abfrageparameter zu setzen. Diese Parameter werden erst beim eigentlichen Ausführen der Abfrage festgelegt und können sich von Ausführung zu Ausführung ändern. Durch dieses Verfahren kann die Datenbank die Abfrage schneller bearbeiten.

Software-Übersicht	
Datenbankmanagementsystem	Informix Dynamic Server (IDS) 11.50.FC6 bzw. 11.70.FC1 (64 Bit)
Web-Verwaltungstools	Informix Server Administrator (ISA) 1.60.TC4 OpenAdmin Tool (OAT) 2.70
Entwicklungsumgebung	Eclipse Classic 3.6.1 (64 Bit)
Java-SDK	Java JDK 6 (Build 1.6.0_23-b05, 64 Bit)
Datenbankschnittstellentreiber	Informix JDBC 3.70.JC3
XML-Bibliothek	Xerces2 Java 2.9.1

Tab. 5–1 *Übersicht über die eingesetzte Software*

Schließlich wird noch die XML-Bibliothek Xerces2 für Java in der Version 2.9.1 eingesetzt. Mit Hilfe von Xerces können XML-Dokumente geparst und in einen DOM-Baum überführt werden. Die Bibliothek unterstützt auch die Generierung eines neuen XML-Dokuments, indem ein DOM-Baum im Speicher schrittweise

zusammengesetzt wird. Dabei werden Funktionen angeboten, mit denen Elemente, Attribute und andere Objekte, die in einem XML-Dokument vorkommen, erzeugt und in den DOM-Baum eingefügt werden können. Der im Speicher erzeugte DOM-Baum lässt sich am Ende mittels Java-Bibliotheksmethoden serialisieren und somit in ein XML-Dokument überführen.

Tabelle 5–1 gibt einen zusammenfassenden Überblick über die eingesetzte Software und die jeweils genutzten Programmversionen. In Abbildung 5–1 ist die Implementierungsumgebung schematisch dargestellt.

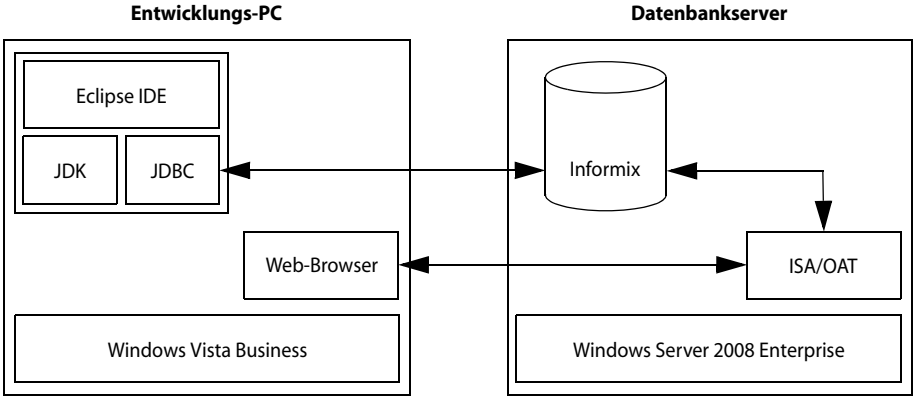


Abb. 5–1 *Schematische Darstellung der Implementierungsumgebung*

5.4 Datenstrukturen

Dieser Abschnitt widmet sich den Datenstrukturen, die zur Speicherung der XML-Dokumente entworfen wurden. Es werden die untersuchten relationalen Schemata sowie die darin auftretenden Typen beschrieben. Begonnen werden soll mit der Skizzierung der einzelnen Entwicklungsentscheidungen, die schließlich zu den eingesetzten Datenstrukturen geführt haben.

5.4.1 Designalternativen

Das Abbildungsverfahren wurde mit der Zielsetzung entwickelt, die Baumstruktur von XML-Dokumenten möglichst direkt abzubilden. Dazu sollten relationale Tabellen mit kollektionswertigen Typen zum Einsatz kommen, die von bestehenden Datenbankmanagementsystemen unterstützt werden.

5.4.1.1 Objektschachtelung

Die festgelegten Rahmenbedingungen führten zu der Überlegung, die XML-Struktur durch geschachtelte Tabellen, also NF^2-Tabellen, zu modellieren. Dieses

Vorgehen erscheint vielversprechend, da es zu einer logisch konsequenten Abbildung der XML-Dokumente führt. Tabelle 5–2 zeigt eine derartige Datenstruktur. Als Datenbasis dient das in Abbildung 4–3 oben links gezeigte XML-Dokumentfragment des `country`-Elements „Germany“. Zur besseren Übersichtlichkeit wurde auf die Wiedergabe der Spalten, die die XML-Attribute und den Textinhalt der Elemente aufnehmen, verzichtet. Die Tabelle kann Objekte oder Tupel aufnehmen, die aus einem Namen und aus einer Liste von Unterobjekten bzw. -tupeln, hier Unterelementen, bestehen.[26] Für jede dieser beiden Angaben steht eine Spalte zur Verfügung, wobei `NAME` vom Typ `VARCHAR` oder einem anderen Texttyp ist. `SUBELEMENTS` kann selbst eine Liste mit Objekten speichern, die wiederum aus einem Namen und einer Unterelementliste bestehen. Bezogen auf die XML-Welt bildet jedes Objekt ein XML-Element ab, von dem hier in diesem Beispiel nur der Elementname und die zugehörigen Unterelemente gezeigt werden.[27]

NAME	**<>SUBELEMENTS**		
country	**NAME**	**<>SUBELEMENTS**	
	name	—	
	population	—	
	government	—	
	province	**NAME**	**<>SUBELEMENTS**
		name	—
		area	—
		population	—
		city	**NAME** / **<>SUBELEMENTS**
			name / —
			population / —
		city	…
		…	…
	province	…	
	…	…	

Tab. 5–2 *Objektschachtelung*

Aufgrund der Vorgabe, dass das zu entwickelnde Abbildungsverfahren ein existierendes, ausgereiftes relationales DBMS nutzen soll, musste untersucht werden, inwiefern sich derartige geschachtelte Tabellen umsetzen lassen. Um geschachtelte Tabellen in einer Datenbank anlegen zu können, müssen bei den meisten Systemen zunächst passende Typdefinitionen vorgenommen werden. Die hier

26. Im Folgenden wird auf die explizite Nennung des Begriffs „Tupel“ verzichtet. Wenn von Objekten gesprochen wird, soll dies auch Tupel mit einschließen.
27. Das Nichtvorhandensein von Unterelementen kann entweder durch einen Null-Wert oder durch eine leere Liste in der Spalte `SUBELEMENTS` dargestellt werden.

benötigten Typen sind rekursiv aufgebaut, da, wie oben gezeigt, die Grundeigenschaft von XML-Dokumenten, dass jedes Element eine Menge von Elementen als Unterelemente enthalten kann, umgesetzt werden muss.

In einer DTD oder einem XML-Schema ist es möglich, die Struktur der zugehörigen XML-Dokumente derart festzulegen, dass beliebig tief geschachtelte Elementstrukturen möglich sind. Die maximale Verschachtelungstiefe ist somit aus der DTD nicht erkennbar. Eine rekursiv definierte DTD und ein zugehöriges XML-Dokument werden in Abbildung 5–2 gezeigt.[28]

```
<!ELEMENT ElementA (ElementA*)>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE ElementA SYSTEM "rekursiv.dtd">
<ElementA>
  <ElementA>
    <ElementA>
      <ElementA></ElementA>
      <ElementA></ElementA>
    </ElementA>
    <ElementA></ElementA>
  </ElementA>
</ElementA>
```

Abb. 5–2 *Rekursiv definierte XML-Dokumentstruktur*

Bei der Abbildung derartiger XML-Dokumente auf geschachtelte Objekte relationaler Tabellen ist es zwingend erforderlich, auch die zugehörigen Datenbanktypen rekursiv zu definieren. Wäre dagegen die maximale Verschachtelungstiefe beschränkt und im Voraus bekannt, käme als Alternative die Abbildung mit Hilfe einer zuvor definierten Menge unterschiedlicher Objekttypen in Frage. Diese Typen könnten dann ineinander geschachtelt werden, ohne dass eine Rekursion bei der Definition erforderlich wäre. Jeder dieser Typen würde für eine zuvor festgelegte Verschachtelungsebene eingesetzt.

Es ist offensichtlich, dass diese Lösung für das zu implementierende Abbildungsverfahren ungeeignet ist, da es zu einer zu starken Beschränkung der abbildbaren Dokumente führt. Aus diesem Grund ist die Nutzung rekursiver Objekt- oder Tupeltypen erforderlich.

Die Frage ist, inwiefern SQL rekursive Typdefinitionen zulässt. Der SQL:2003-Standard [ISO03] beschreibt in Teil 2 auf Seite 632 ff. in Abschnitt 11.41 die Definition benutzerdefinierter Datentypen. Die Attributdefinition wird separat in Abschnitt 11.42 ab Seite 648 behandelt. Dort wird festgelegt, dass der Datentyp eines Attributs eines benutzerdefinierten Typs nicht auf dem Datentyp des zu definierenden benutzerdefinierten Datentyps basieren darf. Somit ist die

28. Laut [Cho02] sind mehr als die Hälfte der in der Praxis vorkommenden DTDs rekursiv aufgebaut.

Selbstenthaltung von Datentypen, also eine Rekursion in der Typhierarchie, ausgeschlossen.

Dem SQL-Standard entsprechend werden direkt und indirekt rekursive Typdefinitionen von den allermeisten relationalen Datenbanken nicht unterstützt. Dies betrifft auch Informix, DB2 und Oracle. In Oracle erhält man zum Beispiel beim Versuch einen derartigen Typ zu erzeugen die folgende Fehlermeldung:

```
ORA-04055: Aborted: ... formed a non-REF mutually-dependent cycle with ...
```

Als Konsequenz ergibt sich, dass das zu implementierende Abbildungsverfahren aufgrund der fehlenden Unterstützung rekursiver Datenstrukturen in SQL unter Verwendung geschachtelter Objekte nicht umsetzbar ist.

5.4.1.2 Objektreferenzen

Die am Ende des vorherigen Abschnitts genannte Fehlermeldung der Oracle-Datenbank gibt einen Hinweis, wie die Einschränkung, dass keine rekursiven Typdefinitionen zulässig sind, umgangen werden kann. Es ist möglich, die in einem Objekt enthaltenen eingeschachtelten Objekte durch Referenzen, also Zeiger, auf die eigentlichen Objekte zu ersetzen.

Durch die Referenzen entsteht eine Typhierarchie, die nicht im eigentlichen Sinne geschachtelt ist, sondern jedes Objekt kann rekursive Verweise auf Objekte des eigenen Typs enthalten. Die Objektinstanzen selbst sind nicht ineinander enthalten, also hierarchisch angeordnet, sondern auf gleicher Ebene abgelegt. Die Hierarchie entsteht erst durch Verknüpfung der Objekte über die abgelegten Referenzen. Tabelle 5–3 zeigt zur Verdeutlichung ein Beispiel, das die gleiche Datenbasis wie Tabelle 5–2 verwendet. Auf die Wiedergabe von Attributen und Textinhalten wurde wiederum verzichtet.

Objektidentitäten

In dem Beispiel werden allerdings keine echten Objektidentitäten und -referenzen eingesetzt. Stattdessen wurden eindeutige IDs, die den jeweiligen Datensatz identifizieren sowie Fremdschlüssel genutzt. Echte Objektidentitäten werden direkt vom Datenbankmanagementsystem erzeugt und in einem meist proprietären Format des jeweiligen Herstellers gespeichert und bei der Ausgabe entsprechend dargestellt. Das System garantiert in diesem Fall, dass sich die Identität eines Objekts während seiner gesamten Existenz nicht ändert und auch, dass diese nicht geändert werden kann. Wichtig ist, darauf hinzuweisen, dass die ID nicht aus den Attributwerten des Objekts errechnet werden kann, da die Änderung der Attribute nicht zu einer Änderung der ID führen darf. Die Eigenschaften Unveränderlichkeit und Eindeutigkeit der ID müssen stets gewahrt bleiben. Diese ID bzw. dieser Identifikator wird dann auch Objekt-ID, abgekürzt OID, genannt. Nach

dem Löschen eines Objekts sollte die Datendank die frei gewordene ID keinem anderen Objekt zuweisen, sondern für jedes neue Objekt immer eine neue, möglichst bisher nicht genutzte ID verwenden.

ID	NAME	<>SUBELEMENTREFS
1	country	< 2, 3, 4, 5, 20, … >
2	name	—
3	population	—
4	government	—
5	province	< 6, 7, 8, 9, 12, … >
6	name	—
7	area	—
8	population	—
9	city	< 10, 11 >
10	name	—
11	population	—
12	city	…
…	…	…
20	province	…
…	…	…

Tab. 5–3 *Verknüpfung der Objekte durch Referenzen*

Einige DBMS erzeugen die OID mit Hilfe der physikalischen Adresse des jeweiligen Objekts. Dabei muss jedoch beachtet werden, dass sich die physikalische Adresse ändert, wenn das Objekt im Speicher umzieht, weil zum Beispiel die Objektattribute größer geworden sind und nicht mehr an der bisherigen Position abgelegt werden können. Bei diesem Vorgang muss die OID unverändert bleiben. Dies lässt sich dadurch gewährleisten, dass an dem alten Speicherplatz ein Verweis abgelegt wird, der auf die neue Position zeigt. Der Vorteil des Verfahrens, die ID an die physikalische Adresse des Objekts zu knüpfen, liegt darin, die Objekte schnell anhand ihrer ID ohne zusätzliche Indexe zu finden.

Eine weit verbreitete Alternative zur Erzeugung der OID über die physikalische Adresse des Objekts ist der Einsatz großer Integer-Werte als IDs. In diesem Fall kann das Auffinden eines Objekts anhand seiner ID mit Hilfe einer Hash-Tabelle erfolgen.

Manche Systeme zeigen die OID nach außen hin nicht an, da diese meist nur für die Verknüpfungen innerhalb der abgelegten Daten des Systems von Bedeutung ist. Wie schon oben erwähnt, wurde in Tabelle 5–3 darauf verzichtet, „echte“ OIDs zu verwenden. An deren Stelle werden kleine eindeutige Integer-Werte genutzt, mit deren Hilfe die Objektreferenzen in Form von Fremdschlüsselbeziehungen dargestellt werden.

Objekte und Tupel

Zu Beginn der Entwicklung von objektorientierten Datenmodellen wurde häufig das Paradigma angewandt: „Alles ist ein Objekt." Dies führte dazu, dass auch einfache Werte, wie Zahlen oder Zeichenketten, in der Datenbank als Objekt abgelegt wurden. Alle diese Objekte bekamen damit automatisch eine Objekt-ID.

Vorteil dieses Vorgehens ist, dass gleiche Werte unterscheidbar bleiben, etwa zwei Zahlen oder Namen. Dies kann sinnvoll sein, wenn sich die Werte auf zwei verschiedene Einheiten beziehen. Dabei handelt es sich jedoch mehr um einen theoretischen Vorteil. Praktisch führt dies zu einem relativ hohen Aufwand zur Erzeugung und weiteren Verwaltung der Objekt-IDs.

Fast alle aktuellen objektorientieren Datenbankmanagementsysteme erlauben auch die Speicherung einfacher Werte, die keine Objekte sind. Hierdurch lassen sich Leistungssteigerungen erzielen und auch der Umgang mit diesen Werten gestaltet sich unkomplizierter.

Neben der Unterscheidung zwischen Objekten und einfachen Werten muss auch zwischen Objekttabellen und herkömmlichen Tupeltabellen differenziert werden. Bei Objekttabellen stellt jeder Eintrag eine Objektinstanz dar, die neben einer Objekt-ID gewisse Eigenschaften und Methoden besitzen kann. Bei Tupeltabellen hingegen entspricht jeder Eintrag bzw. jede Zeile einem Datensatz, einem Record, der bestimmte Attribute umfasst. Jedes Attribut wiederum kann entweder einen einfachen Datentyp aufweisen oder aber selbst ein Objekt mit eigenen Eigenschaften und Methoden darstellen.

5.4.1.3 Individuelle Elementobjekttypen

Für die folgende Betrachtung wird davon ausgegangen, dass Referenzen an Stelle unmittelbarer Objektschachtelungen eingesetzt werden, da, wie zuvor erläutert, nur so eine Implementierung des Abbildungsverfahrens unter den festgelegten Rahmenbedingungen möglich ist. Unabhängig davon, ob echte Objektreferenzen oder Fremdschlüsselverknüpfungen eingesetzt werden, kann ein an das jeweils zu speichernde XML-Element angepasster oder ein generischer Objekttyp eingesetzt werden.

Prinzipiell ist die Nutzung speziell zugeschnittener Objekttypen aufwendiger, da beim Speichern eines XML-Dokuments, das einer bisher nicht verwendeten DTD oder XML-Schema genügt, in der Datenbank für alle vorkommenden Elementarten entsprechende, angepasste Datenbank-Objekttypen erzeugt werden müssen. Diese Objekttypen besitzen dann passende Komponenten, die für die jeweiligen Attribute vorgesehen sind. Dabei werden den Komponenten Datentypen zugewiesen, sodass sie zu den Attributdefinitionen passen. Bei der Generierung der Objekttypen kann zusätzlich berücksichtigt werden, dass nicht jeder Elementtyp Unterelemente haben kann und dass einige Elementtypen Einschrän-

kungen bezüglich der Anzahl und der Typen der zugelassenen Unterelemente aufweisen können. Auf diese und weitere in der DTD oder dem XML-Schema festgelegte Struktureigenschaften können die Objekttypen abgestimmt werden.

Aufgrund der Verwendung derartiger nicht-generischer Objekttypen handelt es sich um ein strukturbasiertes Verfahren, bei dem eine gewisse Sicherheit erreicht wird, dass in der Datenbank nur XML-Dokumente abgelegt werden können, die einer zuvor registrierten DTD bzw. Schema genügen. Allerdings handelt es sich hierbei nur um eine grobe Prüfung, da auf diese Weise nicht alle Bedingungen, die in einer DTD oder einem Schema festlegbar sind, nachgebildet werden können.

TYPE = country			
ID	**car_code**	**area**	**<>SUBELEMENTREFS**
1	D	356910	< 2, 3, 4, 5, 6, 21, … >
TYPE = name			
ID	**TEXT**		
2	Germany		
TYPE = population			
ID	**year**	**TEXT**	
3	95	83536115	
TYPE = government			
ID	**TEXT**		
4	federal republic		
TYPE = encompassed			
ID	**continent**	**percentage**	
5	europe	100	
TYP = province			
ID	**<>SUBELEMENTREFS**		
6	< 7, 8, 9, 10, 13, … >		
TYPE = name			
ID	**TEXT**		
7	Baden Wurttemberg		
…			

Tab. 5–4 *Abbildung auf individuelle Objekttypen*

In Tabelle 5–4 wird eine Speicherstruktur dargestellt, die die beschriebenen individuellen Objekttypen einsetzt.[29] Es handelt sich um eine Objekttabelle (vgl. Abschnitt 3.2.1), der bei der Definition mitgegeben wird, welche Objekttypen in

29. Die Datengrundlage wurde gegenüber Abb. 4–3 geringfügig geändert, damit die Abbildung von Elementen mit Attributen besser dargestellt werden kann.

ihr abgelegt werden können. Im Allgemeinen lässt sich nur ein Objekttyp festlegen, für den die Tabelle vorgesehen ist. Indem man die individuellen Objekttypen für die einzelnen Elementtypen als Subtypen dieses Superobjekts definiert, können alle diese so deklarierten Objekttypen in der Tabelle gespeichert werden.

Hierbei wird der Name der einzelnen Elemente nicht mehr als Spalte bzw. Komponente der jeweiligen Objekte angesehen. Der Name ergibt sich stattdessen direkt aus dem Typ des Objekts. Dieser wird vom DBMS auf eine proprietäre Weise gespeichert. Der Typ der Objekte kann beim Auslesen der Tabelle abgefragt werden. Aus diesem Typ ergeben sich dann auch die jeweils vorhandenen weiteren Komponenten des Objekts.

Elementattribute werden in passenden, typgerechten Objektkomponenten abgelegt. Sofern Unterelemente vorhanden sein können, ist eine atomare oder listenwertige Komponente zur Aufnahme der Unterelementreferenzen vorgesehen. Dieses Attribut hat in Tabelle 5–4 die Bezeichnung `SUBELEMENTREFS`. Ob der Typ atomar oder listenwertig ist, wird anhand der Schemadefinition entschieden. Gibt es nur maximal ein Unterelement, so wird ein atomarer Typ genutzt, bei mehr Unterelementen ein kollektionswertiger. Falls das jeweilige Element textuellen Inhalt besitzt, so wird dieser in einer Komponente mit der Bezeichnung `TEXT` abgelegt. Diese Komponente ist nur vorhanden, falls es laut der Definition des Elements in DTD bzw. Schema zulässig ist, in diesem Element Text abzulegen.

Elemente mit gemischtem Inhalt können mit dem hier vorgestellten Verfahren nicht eindeutig gespeichert werden. Der Grund liegt darin, dass je Element nur ein Text abgelegt werden kann und deshalb nicht festgehalten wird, an welchen Positionen dieser durch eingeschobene Unterelemente zu unterbrechen ist. Eine mögliche Lösung für diese Einschränkung ist, die einzelnen Textblöcke der Elemente mit gemischtem Inhalt separat in Form von eigenständigen Unterelementen abzulegen. Auf diese Idee wird in Abschnitt 5.4.2.4 ausführlicher eingegangen.

5.4.1.4 Generischer Elementobjekttyp

Die Alternative zur Generierung individueller Typen für jeden auftretenden Elementtyp besteht darin, nur einen generischen Elementtyp zu verwenden, der alle aufzunehmenden Daten speichern kann. Die gegebenenfalls zum XML-Dokument vorhandene DTD oder das Schema wird dabei nicht beachtet und muss demzufolge dem Datenbankmanagementsystem auch nicht bekannt gemacht werden. Bei diesem modellbasierten Ansatz wird ein allgemeiner Objekttyp eingesetzt, der eine beliebige Anzahl an Attributen und Elementreferenzen aufnehmen kann. Für jedes Attribut muss dabei ein Paar bestehend auf Attributname und -wert abgelegt werden. Hierzu eignet sich ein mengenwertiger Typ, dessen Einzelelemente aus einem strukturierten Typ bestehen. Dieser strukturierte Typ muss zwei Komponenten aufweisen, die jeweils eine Zeichenkette speichern kön-

nen. Ein in dieser Form aufgebauter Typ zur Aufnahme der Attribute lässt sich in den meisten objektrelationalen Datenbanken, die kollektionswertige Typen unterstützen, problemlos deklarieren.

Das Attribut zur Aufnahme der Elementreferenzen muss hier stets ein Listentyp sein, da keine Sonderbehandlung für Elementtypen erfolgt, die kein oder nur genau ein Unterelement besitzen können. Der Textinhalt der Elemente wird, wie beim vorherigen Verfahren, in einer separaten Spalte als Zeichenkette abgelegt. Tabelle 5–5 zeigt eine Beispiel, dem die gleichen Daten wie im vorherigen Abschnitt zugrunde liegen.

ID	NAME	{}ATTRIBUTES		TEXT	<>SUBELEMENTREFS
		NAME	VALUE		
1	country	car_code	D	—	< 2, 3, 4, 5, 6, 21, … >
		area	356910		
2	name	—		Germany	—
3	population	year	95	83536115	—
4	government	—		federal republic	—
5	encompassed	continent	europe	—	—
		percentage	100		
6	province	—		—	< 7, 8, 9, 10, 13, … >
7	name	—		Baden Wurttemberg	—
…	…	…		…	…

Tab. 5–5 *Abbildung auf generischen Objekttyp*

Man erkennt auch hier, dass eine unveränderte Abbildung und Rekonstruktion von Elementen mit gemischtem Inhalt nicht möglich ist, da nur ein Textfeld pro Element zur Verfügung steht. Wie sich dieses Verfahren leicht erweitern lässt, sodass auch Elemente mit gemischtem Inhalt aus der Datenbank rekonstruiert werden können, wird im nächsten Abschnitt besprochen.

Ein Vorteil dieser Variante ist, dass auch unbekannte XML-Dokumente, also Dateien, zu denen keine DTD und kein Schema vorliegt, in der Datenbank abgelegt werden können. Dadurch entfällt der vor der Speicherung durchzuführende Schritt, in dem die Schemadefinition bei der Datenbank bekannt gemacht werden muss. Diese Registrierung der verschiedenen verwendeten Schemata erzeugt sehr viele Objekttypen, die unter Umständen aufgrund von Typvergleichen und -konvertierungen zu Performanzproblemen führen können.

Als gravierendster Nachteil der Nutzung nur eines generischen Objekttyps für alle Elemente ist zu nennen, dass dieses Verfahren keine direkte Unterstützung für die Validierung des Dokuments gegenüber der Schemadefinition ermöglicht. Die Typen und die Anzahl der Unterelemente und auch die auftretenden Attribute eines Elements unterliegen keinerlei Restriktionen.

Um dies auszugleichen, können „externe“ Zusätze eingesetzt werden, mit deren Hilfe die nötigen Prüfungen durchgeführt werden. Ein Ansatz ist es, an Stelle des direkten Zugriffs auf die Elementtabelle gespeicherte Prozeduren zu nutzen. Dies bedeutet, dass für jede Änderung an den Dokumentdaten eine Stored Procedure aufgerufen wird. Diese kann prüfen, ob die gewünschten Operationen zulässig sind oder ob sie zu einer Verletzung der Schemadefinition führen. Je nach Ergebnis dieses Tests wird die Änderung ausgeführt oder zurückgewiesen.

Als Alternative hierzu können Datenbanktrigger eingesetzt werden. Dies bedeutet, dass weiterhin direkte Änderungen an der Elementtabelle durchgeführt werden dürfen, jedoch bei jedem Schreibzugriff automatisch über einen Trigger eine Routine ausgeführt wird, die überprüft, ob durch die Änderungen die Gültigkeit des Dokuments bezüglich eines festgelegten Schemas weiterhin gegeben ist. Falls nein, kann die Routine unterbinden, dass die Daten der Tabelle geändert werden können.

Bei theoretischer Betrachtung stellen derartige Trigger ein gut einzusetzendes Instrument zur Sicherstellung der Gültigkeit der abgelegten XML-Dokumente dar. Bei der konkreten Implementierung gerät man jedoch häufig an Grenzen, da das jeweilige DBMS bestimmte Operationen oder Zugriffe auf andere Tabellenteile innerhalb von Triggerroutinen nicht unterstützt. Des Weiteren sind zusätzliche Überlegungen erforderlich, wie die Validierung möglichst effektiv durchführt werden kann. Die triviale Lösung, stets das geänderte Dokument komplett zu validieren, führt bei häufigen Änderungen auf umfangreichen Dokumenten zu großen Performanzschwierigkeiten. Sinnvoller erscheint es, nur die Auswirkungen der konkreten Änderung zu untersuchen und zu bewerten, ob dies einen Einfluss auf die Gültigkeit des Dokuments hat oder nicht.

Balmin et al. untersuchen dies in [BPV04]. Mit dem von ihnen entwickelten Verfahren ist in Verbindung mit einer DTD eine inkrementelle Validierung beim Einfügen, Löschen oder Umbenennen von Elementen in $O(m \log n)$ möglich. Dabei gibt n die Größe des XML-Dokuments und m die Anzahl der Änderungen an. Ihr Algorithmus benötigt eine Hilfsstruktur der Größe $O(n)$.

5.4.2 Genaue Beschreibung des Abbildungsverfahrens

In den vorangegangenen Abschnitten wurden verschiedene Designalternativen vorgestellt, wie XML-Dokumente in objektrelationalen Datenbanken gespeichert werden können. Die einzelnen Varianten wurden dabei hinsichtlich ihrer Realisierbarkeit untersucht und die jeweiligen Vor- und Nachteile erläutert. Dabei stellt sich heraus, dass die Verwendung von Objektreferenzen zusammen mit einem generischen Elementobjekttyp am geeignetsten erscheint.

In diesem Abschnitt werden nun das Abbildungsverfahren und die dabei eingesetzten Datenstrukturen im Detail beschrieben. Da Informix keine Objekttabellen unterstützt, wird zur prototypischen Implementierung auf eine typisierte

Tabelle ausgewichen. Eine typisierte Tabelle basiert auf einem benannten Tupeltyp, wobei in jeder Tabellenzeile eine Instanz dieses Typs gespeichert werden kann. Da es sich um eine typisierte Tabelle und nicht um eine Objekttabelle handelt, stehen keine Objektidentifikatoren und -referenzen zur Verfügung. Die Verknüpfungen müssen deshalb über eine Fremdschlüsselbeziehung erfolgen.

5.4.2.1 Speicherung mehrerer Dokumente

Da es nicht sinnvoll ist, für jedes zu speichernde Dokument eine eigene Datenbanktabelle zu nutzen, müssen Maßnahmen getroffen werden, um die Einträge einer Elementtabelle den verschiedenen Dokumenten, zu denen sie gehören, zuordnen zu können. Dies lässt sich realisieren, indem Tabelle 5–5, um eine `DOCID`-Spalte ergänzt wird, mit deren Hilfe alle Tupel identifiziert werden können, die zu einem XML-Dokument gehören. Um Verwechslungen zu vermeiden, wird die bisherige ID-Spalte nun `NODEID` genannt. `DOCID` und `NODEID` bilden zusammen den Schlüssel der Tabelle.

5.4.2.2 Erzeugung der nötigen Typen

In Abbildung 5–3 sind die erforderlichen SQL-Typdefinitionskommandos für Informix wiedergegeben. Die Typdefinitionen beginnen mit der Erzeugung des benannten Tupeltyps `ATTRIBUTE`, der aus den Spalten `NAME` und `VALUE` besteht. Dieser Tupeltyp wird innerhalb des Tupeltyps `ELEMENT` genutzt, um die kollektionswertige Komponente `ATTRIBUTES` zu definieren. `ATTRIBUTES` ist definiert als eine Menge, die aus nicht-leeren `ATTRIBUTE`-Elementen besteht.

```
CREATE ROW TYPE ATTRIBUTE (
   NAME     VARCHAR(30),
   VALUE    LVARCHAR(30000)
);

CREATE ROW TYPE ELEMENT (
   DOCID          INT,
   NODEID         INT,
   NAME           VARCHAR(30),
   ATTRIBUTES     SET(ATTRIBUTE NOT NULL),
   TEXT           LVARCHAR(30000),
   SUBELEMENTREFS LIST(INT NOT NULL)
);

CREATE TABLE ELEMENTS OF TYPE ELEMENT(
   UNIQUE (DOCID,NODEID)
);
```

Abb. 5–3 *Typdefinitionsanweisungen*

Daneben enthält der Tupeltyp ELEMENT noch Spalten zur Aufnahme der DOCID und der NODEID, die wie oben erläutert eingesetzt werden. Die Spalten NAME und TEXT dienen der Aufnahme des Namens und des textuellen Inhalts des Elements. Die Spalte NAME wird auch genutzt, um zu unterscheiden, ob das jeweilige Tupel ein normales XML-Element oder einen speziellen Knoten, wie zum Beispiel einen Kommentar, repräsentiert. Die Unterscheidung erfolgt durch reservierte Bezeichner, bei einem Kommentar wird die Zeichenkette !COMMENT verwendet. Damit dies nicht zu Konflikten mit den in den XML-Dokumenten vorkommenden Elementnamen führt, beginnen alle reservierten Bezeichner mit einem Ausrufungszeichen. Für tatsächliche Elementnamen schließt der XML-Standard dagegen die Verwendung von Ausrufungszeichen aus. Somit ist eine sichere Unterscheidung gewährleistet.

SUBELEMENTREFS ist ein listenwertiges Attribut. Die einzelnen Komponenten dieser Liste sind ganze Zahlen, mit denen die Verknüpfung zu den Unterelementen über die NODEID hergestellt wird. Es handelt sich um eine Liste, somit wird auch die Reihenfolge der Unterelemente festgehalten.

```
...
<city is_country_cap="yes" country="L">
  <name>Luxembourg</name>
  <longitude>6.08</longitude>
  <latitude>49.4</latitude>
  <population year="87">76600</population>
</city>
...
```

DOCID	NODEID	NAME	{}ATTRIBUTES		TEXT	<>SUBELEMENTREFS
			NAME	VALUE		
...	...	...	...		...	...
815	32	city	is_country_cap	yes	—	< 33, 34, 35, 36 >
			country	L		
815	33	name	—		Luxembourg	—
815	34	longitude	—		6.08	—
815	35	latitude	—		49.4	—
815	36	population	year	87	76600	—
...	...	...	...		...	...

Abb. 5–4 *Abbildung von Elementen und Attributen*

Mit Hilfe des so definierten Tupeltyps ELEMENT wird eine typisierte Tabelle angelegt. Diese trägt die Bezeichnung ELEMENTS und besitzt einen Index, der eine schnelle Suche nach der Kombination (DOCID, NODEID) ermöglicht und sicherstellt, dass beide Werte zusammen jede Zeile eindeutig identifizieren.[30] Die Tabelle ELEMENTS wird, wie in Abbildung 5–4 dargestellt, genutzt, um XML-

Dokumente abzubilden. Solange es sich ausschließlich um Elemente mit Attributen und einfachem Textinhalt handelt, sollte das Vorgehen klar sein. Die Behandlung von Elementen mit gemischtem Inhalt und andere Sonderformen werden im Folgenden besprochen.

5.4.2.3 Behandlung von langen Texten

Der Text von Elementen, die keine Unterelemente besitzen, wird bis zu einer maximalen Länge von 30000 Zeichen in der hierfür vorgesehenen Spalte `TEXT` gespeichert. Da der XML-Standard keine Obergrenze für die maximal mögliche Textlänge vorsieht, sollte es auch möglich sein, Texte, die länger als 30000 Zeichen sind, abzuspeichern.

Hierzu wird der Text in Teile aufgespalten, die jeweils maximal 30000 Zeichen umfassen. Die einzelnen Textteile werden als eigenständige Unterknoten des Elements gespeichert, zu dem der Text gehört. Diese Unterknoten erhalten zur Kennzeichnung in der Spalte `NAME` die Bezeichnung `!TEXT`. Das Element, das den Text enthält, führt in der Spalte `SUBELEMENTREFS` die Node-IDs der einzelnen Textteile als Liste auf, sodass sich der ursprüngliche Text wieder in der richtigen Reihenfolge zusammensetzen lässt. Die Spalte `TEXT` beim übergeordneten Element bleibt bei diesen langen Texten leer und wird nicht verwendet. Abbildung 5–5 gibt ein Beispiel, um das Verfahren zu verdeutlichen.

Die Festlegung der Textfragmentgröße auf maximal 30000 Zeichen hat ihren Grund darin, dass der Informix-Typ `LVARCHAR` eine Maximalgröße von 32739 Zeichen zulässt, wobei die Tabellezeile bzw. der Row-Type insgesamt nicht länger als 32767 Bytes sein darf. Unter diesen Bedingungen wäre aktuell für die Spalte `TEXT` eine maximale Länge von 30553 Zeichen möglich. Um in Zukunft bei Bedarf den Typ `ELEMENT` problemlos um weitere Spalten erweitern zu können, wurde die Maximallänge auf 30000 Zeichen beschränkt.

Prinzipiell hätte auch an Stelle des `LVARCHAR`-Typs ein `CLOB`- oder `BLOB`-Typ eingesetzt werden können. Damit wäre die Speicherung deutlich längerer Zeichenketten (bis zu 4 TB) ohne Aufteilung in einzelne Fragmente möglich. Nachteil dabei ist, dass diese Large-Object-Typen weniger flexibel eingesetzt werden können und dass das Ablegen und Auslesen dieser Zeichenketten zusätzlichen Aufwands bedarf. Nicht zuletzt verringert der Einsatz dieser Typen die Performanz, insbesondere vor dem Hintergrund, dass nur ein kleiner Bruchteil der

30. Neben diesem Index ist es je nach Anwendung sinnvoll, weitere Indexe anzulegen. Dabei kann es sich zum Beispiel um einen Index handeln, der das Attribut `NAME` enthält, um eine schnelle Suche nach bestimmten Knotentypen oder Elementnamen zu erlauben. Denkbar ist ebenfalls ein Volltextindex über das Attribut `TEXT`. Ein Index über die XML-Attributnamen oder -werte ist in Informix nicht direkt realisierbar, da keine Indexe auf kollektionswertigen Attributen unterstützt werden. Dazu wäre eine Konstruktion mit Hilfstabelle erforderlich.

üblichen XML-Dokumente Zeichenketten enthält, die länger als 30000 Zeichen sind.

```
...
<Absatz>Es war einmal eine alte Geis, die hatte sieben junge Geislein, und hatte
sie lieb, wie eine Mutter ihre Kinder lieb hat. Eines Tages wollte sie in den
Wald gehen und Futter holen, da rief sie alle sieben herbei und sprach "liebe
Kinder, ich will hinaus in den Wald, seid auf eurer Hut vor dem Wolf, wenn er
herein kommt, so frißt er Euch alle mit Haut und Haar. Der Bösewicht verstellt
sich oft, aber an seiner rauhen Stimme und an seinen schwarzen Füßen werdet ihr
ihn gleich erkennen." Die Geislein sagten, "liebe Mutter, wir wollen uns schon
in Acht nehmen, Ihr könnt ohne Sorge fortgehen." Da meckerte die Alte und machte
sich getrost auf den Weg.</Absatz>
...
```

DOCID	NODEID	NAME	{}ATTRIBUTES		TEXT	<>SUBELEMENTREFS
			NAME	VALUE		
...	...	...	...		...	...
4710	21	Absatz	—		—	< 22, 23, 24, 25, 26, 27, 28 >
4710	22	!TEXT	—		Es war einmal eine alte Geis, die hatte sieben junge Geislein, und hatte sie lieb, wie eine Mutter i	—
4710	23	!TEXT	—		hre Kinder lieb hat. Eines Tages wollte sie in den Wald gehen und Futter holen, da rief sie alle sie	—
...	...	...	...		...	...
4710	28	!TEXT	—		kerte die Alte und machte sich getrost auf den Weg.	—
...	...	...	...		...	...

Abb. 5–5 *Abbildung von langen Texten*

5.4.2.4 Elemente mit gemischtem Inhalt

Elemente, die sowohl Unterelemente als auch Textinhalt aufweisen („mixed content“), müssen gesondert behandelt werden. Grundsätzlich ist es denkbar, den Textinhalt des Elements in der Spalte `TEXT` abzulegen und über die `SUBELEMENTREFS`-Spalte die Verknüpfung zu den Unterelementen vorzunehmen. Dies bringt jedoch das Problem mit sich, dass nicht festgelegt werden kann, an welchen Stellen der Text zwischen die Unterelemente einzufügen ist.

Aus diesem Grund muss bei Elementen mit gemischtem Inhalt stets der Textinhalt – wie auch im Fall von langen Texten – in einem oder, wenn nötig, auch mehreren Unterknoten gespeichert werden. Wie in Abbildung 5–6 gezeigt, bekommen die Unterknoten wiederum in der `NAME`-Spalte die Kennzeichnung `!TEXT` und die Verknüpfung erfolgt über die `SUBELEMENTREFS`-Spalte des übergeordneten Knotens.[31]

```
...
<letterBody>
  <salutation>Dear Mr. <name>Robert Smith</name>.</salutation>Your order of
  <quantity>1</quantity> <productName>Baby Monitor</productName> shipped from
  our warehouse on <shipDate>1999-05-21</shipDate>.
</letterBody>
...
```

DOCID	NODEID	NAME	{}ATTRIBUTES		TEXT	<>SUBELEMENTREFS
			NAME	VALUE		
…	…	…	…		…	…
4711	42	letterBody	—		—	< 43, 47, 48, 49, 50, 51, 52, 53 >
4711	43	salutation	—		—	< 44, 45, 46 >
4711	44	!TEXT	—		Dear Mr.	—
4711	45	name	—		Robert Smith	—
4711	46	!TEXT	—		.	—
4711	47	!TEXT	—		Your order of	—
4711	48	quantity	—		1	—
4711	49	!TEXT	—		*Leerzeichen*	—
4711	50	productName	—		Baby Monitor	—
4711	51	!TEXT	—		shipped from our warehouse on	—
4711	52	shipDate	—		1999-05-21	—
4711	53	!TEXT	—		.	—
…	…	…	…		…	…

Abb. 5–6 *Abbildung von Elementen mit gemischtem Inhalt*

5.4.2.5 Element- und Attributnamen sowie Attributwerte

Gemäß der in Abbildung 5–3 wiedergegebenen Typdefinitionen dürfen Element- und Attributnamen maximal 30 Zeichen lang sein. Für die Mehrzahl der Anwendungen sollte dies genügen. Es ist aber problemlos möglich, die Maximallänge des eingesetzten `VARCHAR`-Typs auf bis zu 255 Zeichen zu erhöhen. Der XML-Standard sieht zwar keine Begrenzung der Länge der Element- und Attributnamen vor, jedoch sollte eine Länge von 255 Zeichen in den allermeisten Fällen genügen.

Für Attributwerte ist eine Obergrenze von 30000 Zeichen vorgesehen.[32] Auch dies bedeutet in der Praxis keine Einschränkung. Konflikte aufgrund

31. Der in Abb. 5–6 genutzte Beispieltext stammt aus [W3C04c, Abschnitt 2.5.2].

bestimmter Sonderzeichen, die in den Attributwerten vorkommen, sind nicht zu erwarten. In XML-Dokumenten können als Begrenzungszeichen der Attributwerte sowohl einfache als auch doppelte Anführungszeichen (jeweils paarweise) genutzt werden. Da in der Datenbank nur der eigentliche Attributwert abgelegt wird, kann beim Auslesen und Serialisieren des Dokuments nicht sichergestellt werden, ob das gleiche Begrenzungszeichen wie im ursprünglichen XML-Dokument genutzt wird. Somit ist eine bytegenaue Wiederherstellung des Dokuments nicht möglich, aber auch nicht erstrebenswert. In diesem Zusammenhang soll auch nochmals darauf hingewiesen werden, dass der XML-Standard keine Reihenfolge unter den Attributen eines Elements definiert. Aus diesem Grund wird zur Ablage der Attribute an Stelle einer Liste eine einfache Menge genutzt. Somit kann es nach dem Serialisieren zu Unterschieden in der Attributreihenfolge kommen. Auf die Möglichkeit, dies durch eine kanonische Form für XML-Dokumente zu verhindern, wird in Abschnitt 5.5.2 eingegangen.

5.4.2.6 Dokumentwurzel, XML-Version, Standalone-Attribut und Zeichensatz

Bekanntlich besitzen XML-Dokumente einen noch oberhalb des XML-Wurzelelements angesiedelten Dokumentwurzelknoten. Für jedes in der Tabelle zu speichernde XML-Dokument muss deshalb ein zugehöriges Dokumentwurzeltupel erzeugt werden. Dieses Tupel bildet die Wurzel der Knotenhierarchie und speichert zusätzliche Angaben, die für das jeweilige Dokument von Bedeutung sind. Dazu zählen die verwendete XML-Version, der Wert des Attributs `standalone` sowie der zur Codierung eingesetzte Zeichensatz.

Diese Werte sind erforderlich, um bei einem Wiederauslesen des Dokuments aus der Datenbank den ursprünglichen Inhalt der XML-Datei möglichst exakt zu rekonstruieren. Als Zeichensatz wird in der Datenbank stets UTF-8 eingesetzt. Sollte ein Dokument in einer anderen Codierung abgespeichert sein, so erfolgt durch den XML-Parser die Konvertierung nach UTF-8. Die ursprüngliche Zeichencodierung wird abgespeichert, sodass beim Wiederauslesen des Dokuments eine Rückkonvertierung vorgenommen werden kann.

Der Dokumentwurzeleintrag wird in der Tabelle `ELEMENTS` mit dem Wert `!DOCROOT` im Attribut `NAME` gekennzeichnet. XML-Version, Standalone-Angabe und originale Zeichencodierung werden als Pseudoattribute in der Spalte `ATTRIBUTES` abgelegt. Diese Attribute erhalten die Bezeichner `XML_VERSION`, `STANDALONE` und `ORIG_ENCODING`. In Abbildung 5–7 ist hierzu ein Beispiel dargestellt.

32. Technisch wäre in Informix eine Maximallänge von 32733 Zeichen möglich. Aus Gründen der Einheitlichkeit wurde die gleiche Maximallänge wie bei der Spalte `TEXT` verwendet.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE mondial SYSTEM "mondial.dtd">
<mondial>
  <country car_code="AL" area="28750">
    ...
  </country>
  ...
```

DOCID	NODEID	NAME	{}ATTRIBUTES		TEXT	<>SUBELEMENTREFS
			NAME	VALUE		
4712	1	!DOCROOT	XML_VERSION	1.0	—	< 2, 3 >
			ORIG_ENCODING	UTF-8		
			STANDALONE	no		
4712	2	!DOCTYPE	SYSTEM	mondial.dtd	—	—
4712	3	mondial	—		—	< 4, … >
4712	4	country	car_code	AL	—	…
			area	28750		
…	…	…	…		…	…

Abb. 5–7 *Abbildung von Dokumentwurzel und Zusatzangaben*

In gleicher Weise ist es denkbar, weitere Angaben zu einem Dokument abzuspeichern. Dabei könnte es sich zum Beispiel um den ursprünglichen Dateinamen des Dokuments oder um einen Zeitstempel handeln, der das Erzeugungsdatum oder den Zeitpunkt des Ablegens in die Datenbank angibt.

Das Dokumentwurzeltupel stellt über die Spalte `SUBELEMENTREFS` die Verknüpfung zum XML-Wurzelelement und ggf. weiteren Knotentypen her, die auf dieser Ebene verwendet werden können. Hierbei kann es sich um Kommentare, Verarbeitungsanweisungen oder um eine Dokumenttypangabe handeln.

5.4.2.7 Kommentare, Verarbeitungsanweisungen, CDATA-Sektionen und Dokumenttypangaben

Kommentare, Verarbeitungsanweisungen und Dokumenttypangaben werden ebenfalls als Tupel der Tabelle `ELEMENTS` gespeichert. Sie werden durch die Bezeichner `!COMMENT`, `!PROCINSTR` und `!DOCTYPE` in der `NAME`-Spalte gekennzeichnet. Unterknoten können in allen drei Fällen nicht auftreten. Die Speicherung der zugehörigen Daten wird im Folgenden beschrieben, Abbildung 5–8 zeigt ein Beispiel.

Kommentare bestehen nur aus einem Textfeld, das den eigentlichen Kommentar darstellt. Dieser wird in der Spalte `TEXT` gespeichert. Verarbeitungsanweisungen setzen sich aus den beiden Angaben „Ziel“ und „Daten“ zusammen. Die

zugehörigen Werte werden in Form von Attributen modelliert. Die Attributbezeichner sind entsprechend TARGET und DATA.

```
...
<!-- Stylesheet einbinden -->
<?xml-stylesheet type="text/xsl" href="Formeln.xsl"?>
<Gleichung>
  <![CDATA[x+y<z]]>
</Gleichung>
```

DOCID	NODEID	NAME	{}ATTRIBUTES		TEXT	<>SUBELEMENTREFS
			NAME	VALUE		
...	...	...	...		...	...
4713	2	!COMMENT	—		Stylesheet einbinden	—
4713	3	!PROCINSTR	TARGET	xml-stylesheet	—	—
			DATA	type="text/xsl" href="Formeln.xsl"		
4713	4	Gleichung	—		—	< 5 >
4713	5	!CDATA	—		x+y<z	—

Abb. 5–8 *Kommentare, Verarbeitungsanweisungen und CDATA-Sektionen*

Sowohl Kommentare als auch der Ziel- und Datenteil der Verarbeitungsanweisungen sind auf eine maximale Länge von jeweils 30000 Zeichen begrenzt. Dies sollte für alle Anwendungen genügen. Gegebenenfalls lässt sich das Abbildungsmodell erweitern, um auch längere Zeichenketten zu unterstützen.

Mit Hilfe einer Dokumenttypangabe kann ein XML-Dokument mit einer DTD verknüpft werden. Die zugehörige DTD kann dabei entweder durch einen Dateipfad, unter dem sie zu finden ist, angegeben werden oder durch Nennung eines allgemeinen („öffentlichen“) Identifikators. Diese beiden Varianten werden durch die Schlüsselwörter SYSTEM und PUBLIC unterschieden. Die Angaben werden dementsprechend entweder als Attribut mit dem Bezeichner SYSTEM oder PUBLIC abgelegt. Abbildung 5–7 gibt ein Beispiel. Mit Hilfe einer Dokumenttypangabe kann alternativ auch direkt eine DTD angeführt werden. Man spricht in diesem Fall von einer internen DTD. Die Speicherung interner DTDs wird momentan nicht unterstützt. Denkbar ist, eine interne DTD in der Spalte TEXT abzulegen.

CDATA-Sektionen sind mit Textblöcken vergleichbar, werden jedoch im Gegensatz zu diesen stets als eigenständige Tabellentupel abgebildet. Dabei trägt die NAME-Spalte den Bezeichner !CDATA. Der Inhalt der CDATA-Sektion wird in der Spalte TEXT abgelegt. Sollte diese Spalte nicht genügen (Maximallänge 30000 Zeichen), so werden weitere CDATA-Tupel angelegt, sodass prinzipiell beliebig lange CDATA-Sektionen gespeichert werden können.

Falls eine CDATA-Sektion in mehrere Tupel aufgeteilt werden muss, so kann ohne Zusatzmaßnahmen beim Serialisieren des XML-Dokuments nicht zweifelsfrei erkannt werden, ob es sich dabei ursprünglich um nur eine Sektion handelte oder ob schon zu Beginn getrennte Sektionen vorlagen. Dies stellt jedoch keinen Informationsverlust dar, da der eigentliche Inhalt der Sektionen unverändert bleibt und es für Anwendungen unbedeutend sein sollte, in wie viele CDATA-Abschnitte der Inhalt aufgeteilt ist.

Sollte wider Erwarten eine exakte Rekonstruktion der CDATA-Sektionen notwendig sein, so kann dies durch zusätzliche Markierungen sichergesellt werden. Auf deren Implementierung, wie auch auf die Unterstützung sehr langer Zeichenketten in Kommentaren und Verarbeitungsanweisungen, wurde verzichtet, da diese den Grundgedanken des Abbildungsverfahrens durch technische Details unnötig verkomplizieren.

5.4.2.8 Entitäten und Namensräume

Entitäten haben verschiedene Einsatzzwecke. Zum einen können mit ihnen häufiger benötigte Teile einer DTD oder eines XML-Dokuments ausgelagert werden und über Schlüsselbegriffe (Entitätsreferenzen) mehrfach eingebunden werden. Hierdurch lässt sich eine rudimentäre Modularisierung der Dokumente erreichen. Zum anderen werden Entitäten auch eingesetzt, um Zeichen, die nicht direkt dargestellt oder eingegeben werden können, über ihre Position im Unicode-Zeichensatz oder durch vordefinierte Entitäten zu referenzieren. Man spricht in diesem Fall von zahlen- bzw. vordefinierten Zeichenentitäten.

In der aktuellen Implementierung werden alle Entitätsreferenzen durch den Wert der jeweiligen Entität ersetzt. Dadurch geht die Information, dass es sich bei dem jeweiligen Fragment ursprünglich um eine Entität gehandelt hat, verloren. Das Dokument selbst bleibt jedoch vollständig.

Zukünftig ist es wünschenswert, die Behandlung von Entitäten auszubauen, sodass die Entitätsdefinitionen und auch die Referenzen auf diese im Dokument erhalten bleiben. Dies lässt sich durch Erweiterung der existierenden Datenstruktur realisieren, indem spezielle Tupel eingeführt werden, die zum Beispiel in der `NAME`-Spalte die Bezeichner `!ENTITY` und `!ENTITYREF` tragen. Momentan wurde darauf verzichtet, da die beim Wiederauslesen der XML-Dokumente eingesetzte Schnittstelle, die DOM-API Level 2, das Editieren von Entitäten nicht spezifiziert. Hier müsste eine Eigenimplementierung erfolgen. Die Funktionsweise des Abbildungsverfahrens lässt sich jedoch auch ohne die vollständige Unterstützung von Entitäten hinreichend gut prototypisch demonstrieren, sodass die Implementierung zurückgestellt wurde.

Die Unterstützung für Namensräume ist gegeben. Sie werden in der Datenbank abgelegt und beim Auslesen unverändert wiederhergestellt. Eine besondere Behandlung der Namensraumpräfixe und der zugehörigen URIs erfolgt nicht.

Dies ist nicht erforderlich, da die Verwendung von Namensräumen nachträglich derart in den XML-Standard eingefügt wurde, dass auch Parser, die Namensräume nicht kennen, weiterhin problemlos funktionieren. In der Datenbanktabelle werden die Namensraumpräfixe zusammen mit den Elementnamen gespeichert. Für zukünftige Weiterentwicklungen ist zu untersuchen, ob es vorteilhaft ist, die Präfixe und URIs sowie die Attribute, über die die Namensraumfestlegungen erfolgen, gesondert zu behandeln.

5.5 Implementierungsdetails

In den bisherigen Abschnitten dieses Kapitels wurden zunächst die Anforderungen an das Abbildungsverfahren aufgestellt und eine geeignete Datenbank ausgewählt. Daraufhin wurden die Implementierungsumgebung und schließlich der Kern des Abbildungsverfahrens und die dabei eingesetzten Datenstrukturen vorgestellt. An dieser Stelle sollen nun die Details der Implementierung beschrieben werden. Begonnen wird mit der Besprechung des Java-Programms `XML2DB`, das für das Ablegen und Wiederauslesen der XML-Dokumente zuständig ist.

An dieser Stelle soll auch auf das Informix-Tool `DBAccess` hingewiesen werden, das in Abbildung 5–9 gezeigt wird. Mit Hilfe dieses Werkzeugs können SQL-Kommandos direkt auf der Datenbank ausgeführt und die zurückgelieferten Tupel betrachtet werden. Die Anwendung ist in der Entwicklungsphase sehr hilfreich, um schnell Klarheit über die Struktur der Datenbank und die gespeicherten Werte zu erhalten. In der hier wiedergegebenen Abbildung sieht man das Tupel, das die Dokumentwurzel des Dokuments `mondial.xml` darstellt.

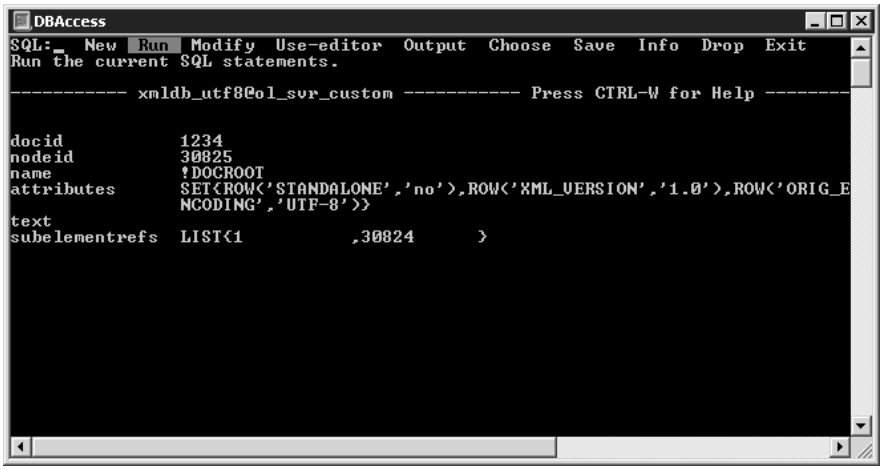


Abb. 5–9 *Ausgabe eines Tupels der Tabelle ELEMENTS mit dem Informix-Tool DBAccess*

5.5.1 Das Tool XML2DB

Um den Einsatz der in Abschnitt 5.4 entwickelten Datenstrukturen im praktischen Einsatz testen und demonstrieren zu können, wurde ein passendes Tool implementiert. Es handelt sich hierbei um das in Java programmierte Kommandozeilentool `XML2DB`.

`XML2DB` hat zwei grundlegende Funktionen: zum einen die Speicherung eines XML-Dokuments in der für den Prototyp ausgewählten Datenbank Informix und zum anderen das Wiederauslesen und die Rekonstruktion der ursprünglichen XML-Datei. Diese zuletzt genannte Funktionalität müsste eigentlich von einem Tool mit dem Namen `DB2XML` zur Verfügung gestellt werden. Da jedoch beide Transformationsrichtungen einige gemeinsame Funktionen verwenden, wurde auf die separate Implementierung zweier Tools verzichtet. Die jeweils gewünschte Transformationsrichtung lässt sich über einen Kommandozeilenparameter auswählen.

Das Tool `XML2DB` besteht aus neun Klassen. Die Abbildungen 5–10 und 5–11 zeigen das zugehörige Klassendiagramm. Die einzelnen Klassen haben die folgenden Aufgaben:

- `XML2DB`
  Hierbei handelt es sich um die Klasse mit der `main`-Methode, in der die Kommandozeilenparameter ausgewertet werden und anschließend die `Inserter`- oder `Serializer`-Klasse instanziiert wird.
- `Inserter`
  Diese Klasse stellt die zum Einfügen eines Dokuments in die Datenbank nötigen Funktionen zur Verfügung.
- `Serializer`
  Das Wiederauslesen und Serialisieren eines Dokuments wird von der Klasse `Serializer` erledigt.
- `DBConnection`
  Die Klasse `DBConnection` kümmert sich um das Herstellen und Beenden der Datenbankverbindung sowie um die Behandlung von vorbereiteten SQL-Kommandos.
- `AttributeRowType`
  Diese Hilfsklasse wird benötigt, um die Java-Datenstruktur mit den XML-Attributen als Objekt an die Datenbank übergeben zu können.
- `InsertStatistics`, `SerializeStatistics`, `Stopwatch` und `MemoryUsage`
  Die Klassen `InsertStatistics` und `SerializeStatistics` liefern statistische Informationen zum Einfüge- bzw. Ausleseprozess. Dazu besitzen sie diverse Zähler sowie Zeitmesser vom Typ `Stopwatch`. Der Arbeitsspeicherbedarf wird mit Hilfe der Klasse `MemoryUsage` ermittelt.

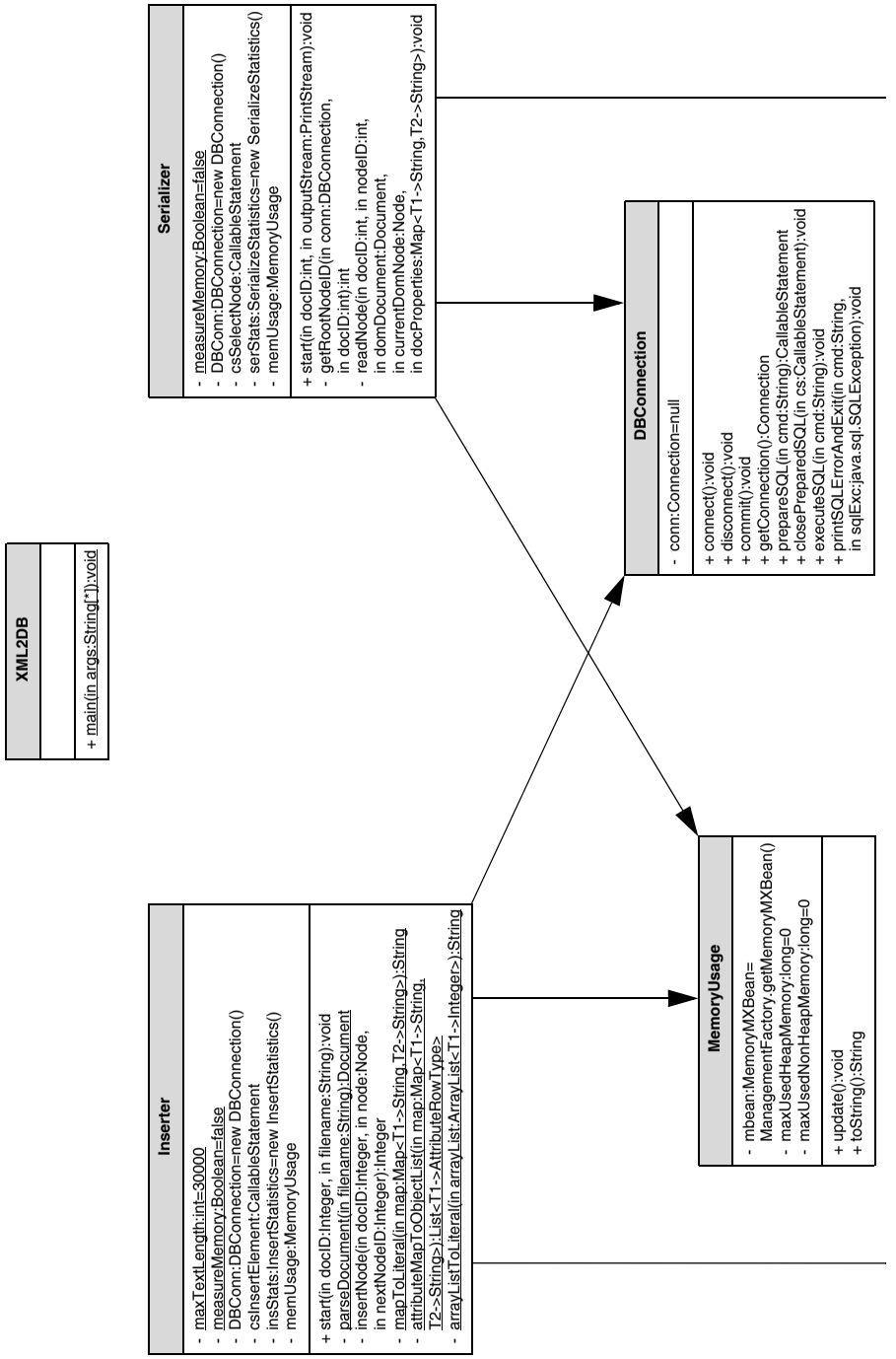


Abb. 5–10 *Klassendiagramm zum Tool XML2DB*

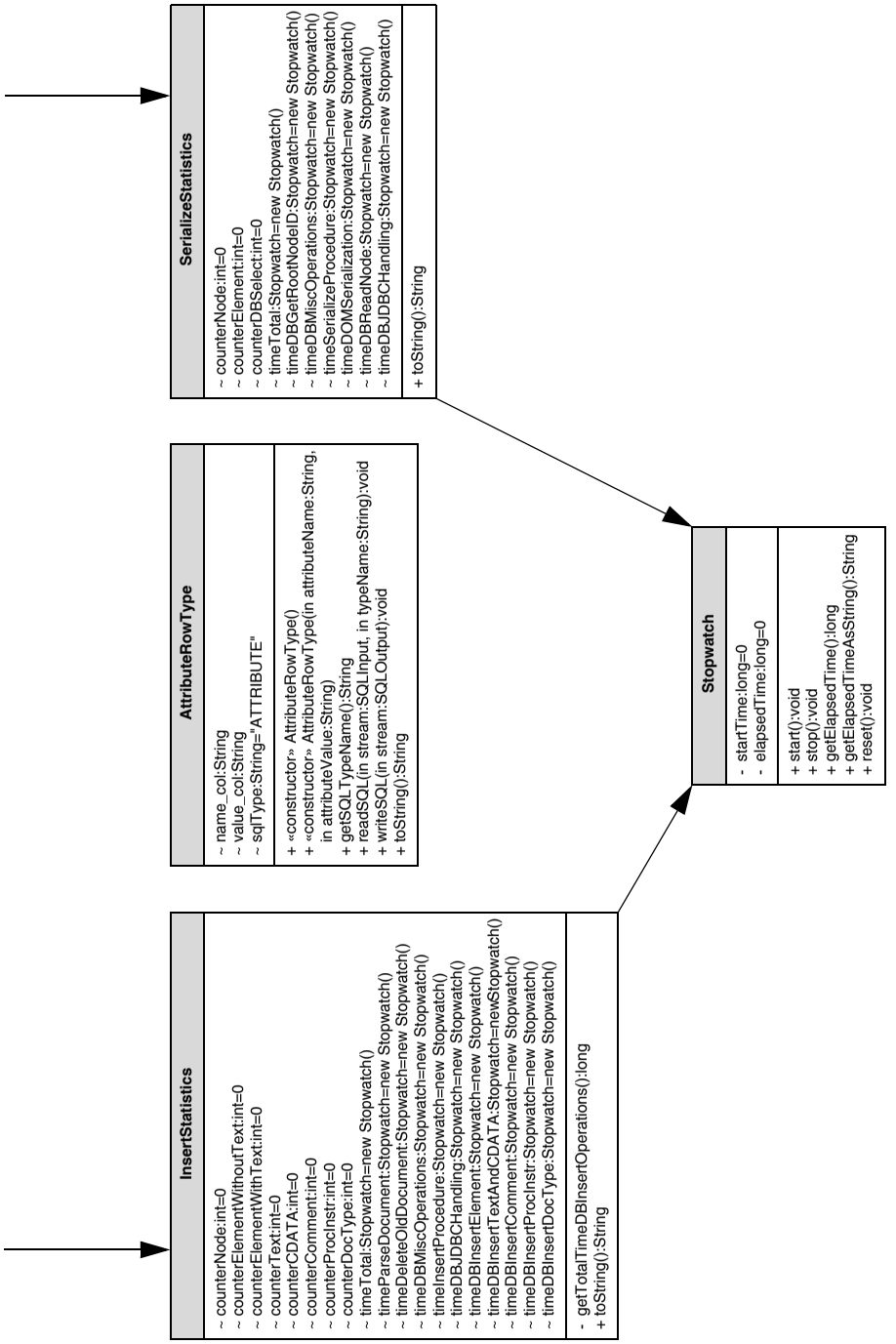


Abb. 5–11 *Klassendiagramm zum Tool XML2DB (Fortsetzung)*

5.5.1.1 Einfügen in die Datenbank

Zum Einfügen eines XML-Dokuments in die Datenbank muss das Tool `XML2DB` wie folgt aufgerufen werden:

```
XML2DB i <DOCID> <XML-Datei>
```

Bei diesem Kommando kennzeichnet der erste Parameter `i`, dass ein Dokument in der Datenbank abgelegt werden soll („insert“). `DOCID` gibt die ID an, die dem Dokument in der Datenbank zugeordnet wird und `XML-Datei` legt das abzuspeichernde XML-Dokument über seinen Dateinamen fest.

Nachdem die Kommandozeilenparameter geprüft und ausgewertet sind, wird das angegebene XML-Dokument durch eine Methode der Klasse `Inserter` geparst. Als Parser wird der Apache Xerces DOM-Parser eingesetzt. Dieser erzeugt aus dem XML-Dokument einen DOM-Baum im Arbeitsspeicher. Der Parser wird mit der Option aufgerufen, die die Erkennung von so genannten „ignorable Whitespaces“ ermöglicht und diese nicht im DOM-Baum aufnimmt. Ignorierbare Whitespaces dienen nur der Formatierung des Dokuments. Sie sind allerdings von normalen Whitespaces nur zu unterscheiden, sofern dem XML-Dokument eine DTD zugrundeliegt. Die Erkennung ist dadurch möglich, dass die Whitespaces als Text an Stellen im Dokument erscheinen, an denen sie nach der DTD eigentlich nicht zulässig sind.[33]

Entitätsreferenzen im XML-Dokument werden beim Parsen durch den Text der jeweiligen Entität ersetzt. Falls das Dokument fehlerfrei eingelesen werden kann und somit wohlgeformt ist und auch bezüglich einer eventuell einbezogenen DTD oder eines XML-Schemas gültig ist, wird die Verbindung zur Informix-Datenbank mit Hilfe der Klasse `DBConnection` aufgebaut. Hierzu wird der Informix-JDBC-Treiber eingesetzt, der neben den Standardfunktionen eines derartigen Treibers einige Informix-spezifische Besonderheiten unterstützt. Die verwendeten Klassen und Methoden von JDBC sollen an dieser Stelle nicht näher erläutert werden. Es wird auf die Java-API-Referenz [Ora11a] und die Informix-JDBC-Dokumentation [IBM11b] verwiesen.

Der Zeichensatz von Client und Server wird auf UTF-8 festgelegt. Die zu verwendende Server-Instanz, die Datenbank, Benutzername und Passwort werden bei der Verbindungsherstellung angegeben. Diese Werte sind momentan fest eingetragen, könnten aber auch problemlos über die Kommandozeile übergeben werden.

---

33. Wird ein XML-Schema verwendet, so dürfen ebenfalls Whitespaces zur Formatierung eingesetzt werden, ohne die Gültigkeit des Dokuments zu verletzen, auch wenn diese Zeichen der festgelegten Syntax widersprechen. Bei Einsatz des Xerces-Parsers ist für das aufrufende Programm die Unterscheidung zwischen normalem Text und ignorierbaren Whitespaces nicht möglich. Der Java-SAX-Parser erlaubt diese Erkennung. Allerdings gilt dies nur in Verbindung mit einer DTD, nicht jedoch bei Vorliegen eines XML-Schemas.

Autocommit wird deaktiviert, sodass alle durchzuführenden Datenbankoperationen in einer Transaktion zusammengefasst werden können. Erst nachdem das komplette Dokument eingefügt wurde, wird die Transaktion abgeschlossen und die Änderungen permanent gemacht. Sollte es während des Vorgangs zu einem Abbruch kommen, werden die bisher durchgeführten Operationen rückgängig gemacht, um so zu verhindern, dass sich unvollständige Dokumente in der Datenbank befinden.

Alternativ ist es auch möglich, Autocommit einzuschalten, also jede einzelne Änderung als separate, sofort abzuschließende Transaktion aufzufassen. Dies hat den Vorteil, dass keine langen Transaktionen entstehen, die einen entsprechend großen Datenbank-Log-Speicher benötigen.[34] Nachteil ist, dass die Transaktionsunterstützung in diesem Fall durch einen externen Applikationsserver übernommen werden muss, um weiterhin eine konsistente Datenbasis zu gewährleisten. Des Weiteren hat sich experimentell gezeigt, dass das Einschalten von Autocommit zu einer schlechteren Performanz führt.

Jedes abgelegte Dokument wird über eine vom Anwender anzugebende `DOCID` identifiziert. Somit darf es in der `ELEMENTS`-Tabelle aus Konsistenzgründen niemals zwei Dokumente unter der gleichen `DOCID` geben. Deshalb muss vor der Ablage eines neuen XML-Dokuments überprüft werden, ob die zu nutzende `DOCID` bereits verwendet wird. In diesem Fall ist das Einfügen zurückzuweisen oder aber das alte Dokument aus der Datenbank zu entfernen.

Aus Gründen der Vereinfachung wird im Prototyp momentan so vorgegangen, dass vor dem Speichern eines neuen Dokuments unter einer bestimmten `DOCID` immer zunächst alle Einträge zu dieser `DOCID` gelöscht werden. An dieser Stelle könnte noch eine Sicherheitsabfrage eingefügt werden, sodass das bisherige Dokument nur nach Rückfrage entfernt wird. Zum Testen der Funktionalität von `XML2DB` ist das aktuelle Vorgehen jedoch ausreichend.

Generell ist anzumerken, dass neben der `DOCID` zukünftig zusätzliche Angaben zu einem XML-Dokument gespeichert werden sollten, um einen leichteren Zugriff auf ein gesuchtes Dokument zu ermöglichen. Dabei kann es sich unter anderem um einen Dateinamen, um das Speicherdatum oder um eine Beschreibung handeln. Über diese Zusatzinformationen lässt sich dann leicht das gewünschte XML-Dokument selektieren. Die eindeutige Identifikation der Dokumente erfolgt jedoch weiterhin über die `DOCID`.

Bevor die Bearbeitung der DOM-Knoten beginnt, muss noch das `INSERT`-Kommando vorbereitet werden. Durch diese Aktion wird der Aufbau des `INSERT`-Befehls der Datenbank bekannt gemacht und durch Platzhalter diejenigen Stellen gekennzeichnet, an denen variable Parameter verwendet werden. Dieses Vorgehen ist empfehlenswert, sofern mehrere gleichstrukturierte Kommandos an eine

34. Falls der Log-Speicher zu klein ist, liefert Informix die Fehlermeldung „Long transaction aborted“.

Datenbank übergeben werden sollen. Bei der jeweiligen konkreten Befehlsausführung müssen die Platzhalter nur noch mit Werten gefüllt werden. Der eingesetzte INSERT-Befehl lautet hier:

```
INSERT INTO ELEMENTS
            (DOCID, NODEID, NAME, ATTRIBUTES, TEXT, SUBELEMENTREFS)
            VALUES (?, ?, ?, ?, ?, ?)
```

Damit sind alle Initialisierungen abgeschlossen, und das eigentliche Einfügen der Tupel des jeweiligen XML-Dokuments kann begonnen werden. Hierzu wird die rekursiv arbeitende Methode `insertNode(docID, node, nextNodeID)` der Klasse `Inserter` genutzt. Die drei Parameter geben die `DOCID`, den zu bearbeitenden Knoten sowie die nächste zu vergebende Node-ID an. Zentraler Bestandteil der Methode ist eine große Switch-Anweisung, die anhand des Knotentyps entscheidet, wie weiter zu verfahren ist.

Falls es sich um einen Elementknoten handelt, so wird als erster Schritt ein `HashMap`-Objekt initialisiert und mit den Paaren aus Name und Wert der Attribute des jeweiligen Elements gefüllt.[35] Anschließend müssen die Kindknoten untersucht werden. Wenn die Menge der Kindknoten nur aus maximal einem Textknoten besteht und der Text dieses Knotens nicht länger als 30000 Zeichen ist, so muss für den Textknoten kein eigenes Tupel in die `ELEMENTS`-Tabelle eingefügt werden. Der Text kann dann direkt in die `TEXT`-Spalte geschrieben werden.

Sofern das aktuell zu bearbeitende Element mehr als einen Textknoten besitzt oder der Inhalt des einzigen Textknotens länger als 30000 Zeichen ist, so müssen hierfür zusätzliche Einträge in die Datenbanktabelle geschrieben werden. Dazu werden zunächst die eventuell vorhandenen langen Text- und CDATA-Knoten in mehrere Einzelknoten aufgesplittet.

Nun werden die Kindknoten der Reihe nach durchlaufen und für jeden dieser Knoten wird die Funktion `insertNode` erneut rekursiv aufgerufen. Die Methode liefert jeweils als Rückgabewert die Node-ID des eingefügten Knotens. Diese Knoten-IDs werden mit Hilfe eines Objekts vom Typ `ArrayList` gesammelt. Nach jedem erfolgreich eingefügten Knoten, bei dem die Methode eine Node-ID zurückliefert, wird diese ID um eins hochgezählt und dieser neue Wert als `nextNodeID` beim nächsten Aufruf an `insertNode` übergeben.

Nachdem die Methode `insertNode` für alle Kindknoten ausgeführt wurde, kann für den Knoten, für den die Methode ursprünglich aufgerufen wurde, ein entsprechendes Tupel in die Tabelle `ELEMENTS` eingefügt werden.[36] Dabei werden

35. Alternativ kann auch ein anderer Array- oder Listentyp eingesetzt werden. Hier wird ein `HashMap`-Objekt genutzt, da dieser Typ die assoziative Struktur der Name-Wert-Paare der Attributmenge gut modelliert.
36. Das Einfügen des Tupels für den Elternknoten erst nach Bearbeitung aller Kindknoten bietet den Vorteil, dass eine Update-Anweisung eingespart wird. Alternativ könnte auch zunächst das Elternknotentupel eingefügt, dann die Kindknoten bearbeitet und als letzter Schritt das Elternknotentupel mit den Referenzen auf die Kindknoten aktualisiert werden.

der aktuelle Wert der Variable `nextNodeID` in der Spalte `NODEID` und der Name des Elements in der Spalte `NAME` abgelegt. Der Textinhalt des Elements wird in der Spalte `TEXT` gespeichert, sofern die oben genannten Bedingungen für die Speicherung des Textes im Elementtupel selbst erfüllt sind.

In der Spalte `ATTRIBUTES` werden die Attribute des Elements gesichert. Hierzu muss die zuvor gefüllte Objektinstanz vom Typ `HashMap` in ein Literal konvertiert werden. Dazu wurde die Methode `mapToLiteral` entwickelt, die die nötige Umwandlung in eine Zeichenkette durchführt und dabei Vorkehrungen trifft, sodass in allen Fällen ein gültiges Informix-`SET`-Literal entsteht.

Leider ist die Länge der Informix-Literale auf 32 739 Bytes beschränkt. Beim Versuch längere Literale zu verwenden, gibt Informix die Fehlermeldung „invalid collection literal“ aus. Die maximal mögliche Länge scheint nicht dokumentiert zu sein. Sie wurde experimentell ermittelt und entspricht der maximal möglichen Größe einer Zeichenkette, die in einer Spalte vom Typ `LVARCHAR` gespeichert werden kann.

Für die meisten Dokumente stellt die Obergrenze kein Problem dar. Dennoch sollten Vorkehrungen getroffen werden, um auch Elemente mit einer sehr großen Anzahl an Attributen ablegen zu können oder solche, die mehrere sehr lange Attributwerte aufweisen. Die Lösung findet sich in der Verwendung von Objektstrukturen, die direkt an den JDBC-Treiber übergeben werden. Dazu musste die passende Hilfsklasse `AttributeRowType` zur Verfügung gestellt werden, die das `SQLData`-Interface implementiert und die notwendigen Konvertierungen vornimmt.

Die Nutzung derartiger Objekte stellt prinzipiell gegenüber den Literalen eine elegantere Alternative dar, Daten an den JDBC-Treiber zu übergeben. Leider hat sich jedoch herausgestellt, dass die Verarbeitung deutlich langsamer ist.[37] Die Interpretation des Literals in der Datenbank ist überraschenderweise effizienter als die Konvertierung durch den JDBC-Treiber. Aus diesem Grund wird folgende Strategie eingesetzt: Sollte das Attributeliteral maximal 32 739 Bytes groß sein, so wird die Übergabe darüber durchgeführt. Anderenfalls wird auf den Einsatz von Objekten ausgewichen.

Die Nutzung von Objekten hat neben der geringeren Verarbeitungsgeschwindigkeit noch einen weiteren Nachteil. Es kommt bei der Übergabe des Attributnamens, der in der Datenbank den Datentyp `VARCHAR(30)` aufweist, zu dem Effekt, dass alle Zeichenketten auf 30 Zeichen durch Leerzeichen am Ende aufgefüllt werden. Hier scheint es sich um einen unerwünschten Effekt des JDBC-Treibers zu handeln. Die zusätzlichen Leerzeichen am Ende belegen unnötigen Speicherplatz in der Datenbank und müssen beim Wiedereinlesen mit Hilfe der `trim`-Methode entfernt werden. Da die Attributübergabe über Objekte nur in seltenen Fällen erfolgt, lässt sich dieses unerwünschte Verhalten tolerieren.

37. Die Literal- ist gegenüber der Objektübergabe bis zu 10-mal schneller.

Vor der Ausführung des INSERT-Kommandos muss zuletzt noch die Spalte SUBELEMENTREFS gefüllt werden. Hierzu werden die gesammelten IDs der Unterknoten ausgewertet. Falls es keine gibt, so wird SUBELEMENTREFS auf NULL gesetzt.[38] Anderenfalls wird ein Informix-LIST-Literal erzeugt, dass die Knoten-IDs, durch Kommas getrennt, als Zeichenkette enthält. Auch hier muss die maximal zulässige Literallänge von 32739 Bytes berücksichtigt werden. Bei sehr großen Listen wird deshalb das ArrayList-Objekt direkt ohne String-Konvertierung an den JDBC-Treiber übergeben.[39] Dabei tritt wiederum der Nachteil auf, dass die Ausführung des INSERT-Befehls in diesen Fällen deutlich länger dauert.

Der beschriebene Ablauf zum Einfügen eines Elementknotens kann fast unverändert für den Dokumentwurzelknoten übernommen werden. Beim Dokumentwurzelknoten wird jedoch die NAME-Spalte auf !DOCROOT gesetzt und die ATTRIBUTES-Spalte wird mit den Angaben zu XML-Version, Zeichensatz und Standalone-Eigenschaft gefüllt.

Die Behandlung der weiteren fünf Knotentypen (Text, CDATA-Sektion, Kommentar, Verarbeitungsanweisung, Dokumenttypangabe) soll hier nicht besprochen werden. Das Vorgehen ergibt sich aus dem für Elementknoten beschriebenen sowie aus der Darstellung der entsprechenden Datenstrukturen in Abschnitt 5.4.

Beim Einfügen des Dokuments wird der DOM-Baum in einem Tiefendurchlauf (engl. „depth-first traversal“) traversiert. Da der jeweilige Knoten selbst erst nach der Behandlung aller Kindknoten in die Tabelle eingefügt wird, handelt es sich genauer um einen Post-Order-Tiefendurchlauf. Dies bietet den Vorteil, dass nur ein INSERT-Kommando je Elementknoten nötig ist. Die Knoten-IDs werden ebenfalls in der Reihenfolge dieses Durchlaufs vergeben. Daraus ergibt sich, dass der Dokumentwurzelknoten stets die größte Knoten-ID unter den Knoten-IDs eines Dokuments aufweist.

5.5.1.2 Auslesen aus der Datenbank

Wenn ein Dokument aus der Datenbank ausgelesen werden soll, so muss XML2DB wie folgt aufgerufen werden:

```
XML2DB s <DOCID> [<XML-Datei>]
```

38. Semantisch passender wäre es, bei nicht vorhandenen Unterknoten keinen Null-Wert, sondern eine leere Liste abzuspeichern. Experimentell wurde jedoch festgestellt, dass das INSERT-Kommando beim Einfügen der leeren Liste an Stelle des Null-Werts um etwa 10 % langsamer ausgeführt wird. Deshalb wird bei der Implementierung der Null-Wert verwendet. Es ist dabei überraschend, dass die Übergabe des Null-Werts als Zeichenkette vom Wert NULL schneller abgearbeitet wird als der explizite Aufruf der hierfür vorgesehenen Methode setNull.
39. Die Grenze wurde bei 1000 Einträgen, also 1000 Unterknoten, gezogen.

In diesem Kommando kennzeichnet die Option `s`, dass ein Dokument ausgegeben, also serialisiert, werden soll. Das XML-Dokument wird über seine `DOCID` spezifiziert. Ohne weitere Parameter erfolgt die Ausgabe des Dokuments auf der Standardausgabe. Wird ein Dateiname als dritter Parameter übergeben, so wird das Dokument in diese Datei geschrieben.

Erster Schritt bei der Bearbeitung des Kommandos durch die Klasse `Serializer` ist die Herstellung der Verbindung zur Informix-Datenbank sowie die Prüfung, ob ein Dokument mit der angegebenen ID in der Tabelle `ELEMENTS` existiert. Dabei wird nach dem Dokumentwurzeltupel gesucht. Falls dieses existiert, so wird die `NODEID` des Dokumentwurzelknotens ausgelesen.

Die folgende SQL-`SELECT`-Abfrage wird für das Auslesen des Dokuments aus der Datenbank eingesetzt. Diese wird vorbereitet, da sie für jeden Knoten einmal aufgerufen werden muss.

```
SELECT NAME, ATTRIBUTES, TEXT, SUBELEMENTREFS
FROM ELEMENTS
WHERE DOCID = ? AND NODEID = ?
```

Um den Auslesevorgang zu beginnen, wird ein leerer DOM-Baum erzeugt und anschließend die rekursiv arbeitende Prozedur `readNode` aufgerufen. In dieser Prozedur werden mit Hilfe der oben genannten `SELECT`-Anweisung die Daten des jeweils aktuellen Knotens aus der `ELEMENTS`-Tabelle ermittelt und aufbereitet. Der ausgelesene Wert der `NAME`-Spalte bestimmt das weitere Vorgehen. In mehreren `IF-THEN-ELSE`-Konstrukten erfolgen die nötigen Fallunterscheidungen.[40]

Sofern aufgrund der `NAME`-Spalte zu erkennen ist, dass es sich um ein normales Element handelt, so muss ein entsprechendes Element für den DOM-Baum erzeugt, die Attribute eingetragen und das Element an den bisher aktuellen Knoten des DOM-Baums mit Hilfe der Methode `appendChild` angehängt werden. Anschließend wird das neue Element zum aktuellen Knoten des bisher aufgebauten DOM-Baums. Die Verarbeitung der anderen Knotentypen erfolgt grundsätzlich analog, allerdings müssen die in der `ATTRIBUTES`-Spalte abgelegten Zusatzinformationen dem Typ des Knotens entsprechend behandelt werden.

Falls der jeweilige Knoten Unterknoten besitzt, müssen diese am Ende der Prozedur `readNode` ausgewertet werden. Mit dem oben genannten Informix-`SELECT`-Kommando wurde die listenwertige Spalte `SUBELEMENTREFS` komplett ausgelesen. Für Fälle, in denen nur ein bestimmter Eintrag und nicht die vollständige Liste selektiert werden muss, wurde die gespeicherte Prozedur `GetSubElementRef` entwickelt, deren Definition in Abbildung 5–12 wiedergegeben ist. Die Prozedur ermöglicht die direkte Selektion eines bestimmten Listeneintrags eines angegebenen Dokumentknotens. Aus Performanzgründen sollte die gespei-

40. Bei Nutzung der neuen Java-Version 1.7 kann stattdessen auch eine Switch-Anweisung genutzt werden, da in dieser nun auch Zeichenkettenvergleiche möglich sind.

cherte Prozedur jedoch nicht eingesetzt werden, wenn alle Listeneinträge – wie hier beim vollständigen Serialisieren des Dokuments – benötigt werden.

```
CREATE FUNCTION GetSubelementRef(ParDocID INT, ParNodeID INT,
   ParSubElementRefIndex INT) RETURNING INT;

   DEFINE RefList LIST(INT NOT NULL);
   DEFINE i INT;
   DEFINE Ref INT;

   SELECT SUBELEMENTREFS INTO RefList
   FROM ELEMENTS
   WHERE DOCID = ParDocID AND NODEID = ParNodeID;

   IF RefList IS NOT NULL THEN
      LET i = 1;
      FOREACH Cursor1 FOR
         SELECT * INTO Ref FROM TABLE(RefList)
         IF i = ParSubElementRefIndex THEN
            RETURN Ref;
         END IF;
         LET i = i + 1;
      END FOREACH;
   END IF;

   RETURN -1;

END FUNCTION;
```

Abb. 5–12 *Definition der Prozedur „GetSubelementRef"*

Um die Unterknoten eines Knotens zu verarbeiten, wird für jeden in der Spalte `SUBELEMENTREFS` abgelegten Verweis in einer Schleife der Reihe nach die Prozedur `readNode` rekursiv aufgerufen. Auf diese Weise werden die Unterknoten in den DOM-Baum eingefügt. Die sich so ergebende Abfolge der Knoten des XML-Dokuments entspricht wiederum einem Tiefendurchlauf des Baums.

Nachdem die Prozedur `readNode` und alle ihre rekursiven Selbstaufrufe abgearbeitet sind, ist der DOM-Baum des auszulesenden Dokuments vollständig aufgebaut und die Verbindung zur Informix-Datenbank kann getrennt werden. Um ein XML-Dokument zu erhalten, muss der Baum nur noch serialisiert werden.

Für die Serialisierung wird die Klasse `Transformer` aus dem Java-Paket `javax.xml.transform` genutzt. Dabei kann unter anderem bestimmt werden, ob die Ausgabe durch Einrückungen gut lesbar formatiert werden soll. Des Weiteren werden der Klasse die aus dem Dokumentwurzelknoten ausgelesenen Eigenschaften XML-Version, Zeichensatz und Standalone-Attribut übergeben, sodass auch diese bei der Serialisierung dem Originaldokument entsprechend wiederhergestellt werden. Außerdem müssen an dieser Stelle die Festlegungen aus dem Dokumenttypknoten explizit eingebracht werden, da diese Angaben beim Seriali-

sieren nicht direkt aus dem entsprechenden DOM-Baumknoten übernommen werden.

5.5.2 Änderungen der XML-Dokumentstruktur

Wird ein XML-Dokument in der Datenbank abgelegt und anschließend wieder ausgelesen und serialisiert, kommt es unter Umständen zu Abweichungen gegenüber dem Originaldokument. Von den Änderungen ist die Form des Dokuments und nicht der eigentliche Inhalt betroffen. Im Folgenden werden die möglichen Unterschiede besprochen.

Da der XML-Standard nur die Reihenfolge der Elemente festschreibt und keine Abfolge der Attribute eines Elements definiert, wurde auch das Datenmodell derart gestaltet, dass für die Speicherung der Attribute an Stelle einer geordneten Liste eine ungeordnete Menge eingesetzt wird. Deshalb kann es dazu kommen, dass das aus der Datenbank wieder ausgelesene Dokument die Attribute in einer anderen Abfolge anordnet, als sie im ursprünglichen Dokument vorgekommen sind. Dies führt dazu, dass die Dokumente zwar logisch gleich sind, bei zeichenweisem Vergleich treten jedoch Unterschiede auf.

Entsprechendes gilt für die Anführungszeichen, die Attributwerte begrenzen. Der XML-Standard sieht hierfür sowohl einfache als auch doppelte Anführungszeichen vor. Wichtig ist nur, dass sie paarweise auftreten und dass innerhalb eines Attributwerts, der durch doppelte Anführungszeichen begrenzt wird, keine weiteren doppelten Anführungszeichen genutzt werden, es sei denn, sie werden durch die zugehörige vordefinierte Zeichenentität ausgedrückt. Dies gilt analog für einfache Anführungszeichen.

Das genutzte Datenmodell benötigt diese Begrenzungszeichen beim Speichern von Attributwerten nicht. Demzufolge werden diese auch nicht in der Datenbank abgelegt. Beim Wiederauslesen des Dokuments kann deshalb nicht bestimmt werden, ob bei den einzelnen Attributwerten ursprünglich doppelte oder einfache Anführungszeichen verwendet wurden. Aus diesem Grund werden beim Serialisieren stets doppelte Anführungszeichen genutzt. Eventuelle doppelte Anführungszeichen, die Teil des Attributwerts sind, werden durch die Entität `&quot;` ausgedrückt. Einfache Anführungszeichen im Attributwert bleiben unangetastet.

Innerhalb der Datenbank werden die XML-Dokumente stets im Unicode-Zeichensatz codiert. Hierzu wird beim Einfügen in die Datenbank eine entsprechende Konvertierung durchgeführt. Der Originalzeichensatz wird in der Datenbank gespeichert, sodass beim Serialisieren eine Rückkonvertierung möglich ist. Somit treten bezüglich der Zeichencodierung keine Unterschiede zwischen Original- und aus der Datenbank wieder ausgelesenem Dokument auf.

In Bezug auf Whitespaces werden folgende Änderungen durchgeführt: Prinzipiell werden Whitespaces, also Leerzeichen, Tabulatoren und Zeilenvorschübe, die in Textknoten auftreten, unverändert gespeichert und bei der Serialisierung

wiederhergestellt. Dies gilt allerdings nicht für die ignorierbaren Whitespaces, die nur der Formatierung des Dokuments dienen. Derartige Whitespaces werden entfernt.

Daneben findet man Whitespaces auch als Trennzeichen, zum Beispiel zwischen zwei Attributangaben im Start-Tag eines Elements. An dieser Stelle findet eine Vereinheitlichung statt, sodass bei der Ausgabe der XML-Dokumente aus der Datenbank stets genau ein Leerzeichen zwischen zwei Attributangaben steht. Ähnliches gilt unter anderem auch für Verarbeitungsanweisungen und Dokumenttypangaben.

Bei der Behandlung der Entitäten werden die Entitätsreferenzen durch den jeweiligen Text ersetzt, sodass die eigentliche Entität nicht in die Datenbank aufgenommen wird und somit auch nicht im wieder serialisierten XML-Dokument erscheinen kann. Diese Veränderung des Dokuments hat ihre Ursache in der hier eingesetzten Implementierung des Abbildungsverfahrens. Da grundsätzlich keine DTDs – weder interne noch externe – in die Datenbank aufgenommen werden, kommt es bei Nichtersetzung der Entitätsreferenzen dazu, dass aus der Datenbank ein nicht gültiges XML-Dokument zurückgeliefert wird, da nicht definierte Entitätsreferenzen enthalten sind. Dieses Problem kann durch die Entitätsersetzung nicht auftreten, und die Dokumente behalten grundsätzlich ihren vollständigen Inhalt, auch wenn natürlich die Angabe, dass es sich ursprünglich um Referenzen gehandelt hat, verlorengeht. Sollte dies bei einer konkreten Anwendung zu Schwierigkeiten führen, muss das Datenmodell entsprechend erweitert werden, um diese Zusatzangaben aufnehmen zu können.

Es existiert eine Reihe verschiedener Tools[41], mit denen eine Prüfung möglich ist, ob zwei XML-Dokumente identisch sind. Diese Applikationen können dabei die oben beschriebenen möglicherweise auftretenden Abweichungen ignorieren und trotz zeichenbezogener Ungleichheit die logische Übereinstimmung zweier Dokumente feststellen.

Das W3C hat sich der Problematik der logisch identischen, aber dennoch zeichenweise verschiedenen XML-Dokumente angenommen. Es existiert der Standard „Canonical XML“ [W3C01], der eine Normalisierungsvorschrift festlegt, nach der XML-Dokumente transformiert werden müssen, sodass logisch gleiche Dokumente auch tatsächlich zeichenweise übereinstimmen.[42] Der Standard legt unter anderem hierzu die folgenden Konvertierungen fest:

41. Als Beispiele sind die XML-Diff-and-Patch-Tools von Mouat [Mou02] und Microsoft [Raj02] zu nennen.
42. In diesem Zusammenhang wird häufig die Abkürzung „C14n“ benutzt, die für Canonicalization steht. Die Zahl 14 bezeichnet die Anzahl der Buchstaben zwischen C und N.

- Als Zeichencodierung wird UTF-8 verwendet.
- Alle Whitespaces innerhalb des Dokumentwurzelelements bleiben unverändert erhalten, Zeichenumbrüche werden einheitlich mit dem Unicode-Zeichen `U+000A` dargestellt.
- CDATA-Sektionen sowie Entitätsreferenzen werden durch den enthaltenen bzw. referenzierten Text ersetzt.
- Attribute werden bezüglich der auftretenden Whitespaces normalisiert und die Werte einheitlich durch doppelte Anführungszeichen begrenzt.
- Bestimmte Sonderzeichen in Attributwerten und Textabschnitten werden durch vordefinierte Entitätsreferenzen ersetzt.
- Die Namensraumdeklarationen und Attribute jedes Elements werden alphabetisch sortiert. Unbenutzte Namensraumdeklarationen werden entfernt.
- Die Kurznotation für leere Elemente wird nicht genutzt.
- Nicht angegebene Attribute mit Default-Werten werden zum Dokument hinzugefügt.
- Die XML-Deklaration und eine eventuell vorhandene DTD-Angabe sind nicht Teil der kanonischen Form eines XML-Dokuments.

Für die Durchführung der Konvertierung in Canonical XML stehen diverse Tools zur Verfügung. Es existieren für diese Aufgabe auch freie Java-Bibliotheken.

Im Zusammenhang mit kryptografischen Signaturen kommt dem Standard Canonical XML besondere Bedeutung zu. Eine Signatur signiert stets eine bestimmte Zeichen- bzw. Bytefolge. Sofern es sich bei der zu signierenden Datei um ein XML-Dokument handelt, bleibt die Signatur nur so lange gültig, wie auch keinerlei Änderungen am Dokument durchgeführt werden. Durch verschiedene Darstellungen des logisch gleichen XML-Dokuments kann es deshalb dazu kommen, dass die Signatur nicht mehr als gültig erkannt wird. Diesem Problem kann begegnet werden, indem vor der Signierung und bei jeder Prüfung der Signatur eine Konvertierung des Dokuments gemäß dem Canonical-XML-Standard durchgeführt wird.

Grundsätzlich ist es also nicht unüblich, dass sich logisch gleiche Dokumente bei zeichenweiser Betrachtung unterscheiden. Bei dem hier entwickelten Abbildungsverfahren treten derartige Unterschiede ebenfalls auf. Diese lassen sich jedoch gut beherrschen, und es ist nicht erforderlich, eine zeichengetreue Abbildung durch die zusätzliche Speicherung weiterer Angaben zu erzwingen. Dies würde der möglichst direkten Umsetzung der logischen Struktur im Wege stehen und das komplette Verfahren unnötig technisch kompliziert machen.

6 Leistungsbetrachtungen

Wie bei den meisten Anwendungen in der Informatik ist es nicht nur von Bedeutung, dass der Algorithmus grundsätzlich funktioniert, sondern auch, dass seine Ausführung in einer akzeptablen Zeit bei einem beherrschbaren Speicherbedarf möglich ist. Nur dann kann der Algorithmus, hier das vorgeschlagene Abbildungsverfahren, auch praktisch eingesetzt werden.

In diesem Kapitel werden die Ergebnisse von Leistungsmessungen besprochen, die durchgeführt wurden, um sowohl den Zeit- als auch den Speicherbedarf des Abbildungsverfahrens zu analysieren. Dabei wurde auch versucht zu ermitteln, welche Auswirkungen bestimmte Konfigurationseinstellungen auf die Verarbeitungsleistung haben und welche Vor- und Nachteile sich dadurch ergeben.

Das Kapitel gliedert sich wie folgt: Im ersten Abschnitt werden die technischen Daten des verwendeten Datenbankservers dargestellt. Daran schließt sich ein kurzer Überblick über die XML-Dokumente an, auf deren Basis die Messungen durchgeführt wurden. Hierzu werden die jeweiligen Besonderheiten der einzelnen Dokumente sowie einige statistische Werte angeführt.

In den Abschnitten 6.3 und 6.4 werden schließlich die Ergebnisse der Leistungsuntersuchungen getrennt nach Zeit- und Speicherbedarf behandelt. Das Kapitel endet mit einer vergleichenden Untersuchung, bei der die Oracle-Datenbank verwendet wird.

6.1 Technische Daten der eingesetzten Hardware

Für die Leistungsmessungen wurde die Informix-Datenbank in der Version 11.70 auf einem Server installiert, der auf dem Betriebssystem Windows Server 2008 R2 in der 64-Bit-Variante basiert. Der Server verwendet eine Vier-Kern-Intel-Xeon-CPU mit einer Taktfrequenz von 2,4 GHz. Es stehen 465 GB Festplattenplatz und 16 GB Arbeitsspeicher zur Verfügung. Tabelle 6–1 fasst die technischen Daten des Servers zusammen.

Komponenten	
Betriebssystem	Windows Server 2008 R2 Enterprise Service Pack 1
Systemtyp	64 Bit
CPU	Intel Xeon X3430
Taktfrequenz	2,4 GHz
Arbeitsspeicher	16 GB (DDR3-1333)
Festplatten	2 x 465 GB (7200 RPM) als RAID 1

Tab. 6–1 *Technische Daten des eingesetzten Datenbankservers*

6.2 Testdokumente

In diesem Abschnitt werden vier ausgewählte XML-Dokumente vorgestellt, die für die Durchführung der Speicher- und Zeitbedarfsanalysen genutzt wurden. Jedes der Dokumente weist im Aufbau einige Besonderheiten auf, mit denen die Auswirkungen dieser Unterschiede auf das Abbildungsverfahren untersucht werden können. Mit diesen Dokumenten wurden auch die ersten Tests durchgeführt, um die einwandfreie Funktion des implementierten Algorithmus zeigen zu können. Daneben wurden noch weitere Dokumente zur Überprüfung eingesetzt, die hier jedoch nicht vorgestellt werden sollen.

Das XML-Dokument „mondial.xml“

Die geografische Datenbank „Mondial“ wurde im Jahr 1998 am Institut für Programmstrukturen und Datenorganisation der Universität Karlsruhe entwickelt und im Rahmen der dort stattfindenden Übungen zu SQL genutzt. Für die Erstellung wurden verschiedene frei im Internet verfügbare Datenquellen verwendet, unter anderem das so genannte CIA World Factbook[43]. 1999 wurde hierzu von Wolfgang May ein technischer Report mit dem Titel „Information Extraction and Integration with Florid: The Mondial Case Study“ [May99] veröffentlicht. Die Mondial-Datenbank ist für Zwecke der Forschung und Lehre unter der Bedingung der Angabe der Herkunft frei nutzbar.

Aus dieser Datenbank wurde das XML-Dokument `mondial.xml` erzeugt. Die Datei kann zusammen mit der zugehörigen DTD von der Web-Seite des Instituts für Informatik an der Universität Göttingen, dessen Leiter Wolfgang May zur Zeit ist, heruntergeladen werden [May10]. Neben der XML-Variante stehen auch eine relationale Version sowie die Formate F-Logic und RDF/OWL zur Verfügung. Im Jahr 2009 wurden einige inkrementelle Anpassungen der Daten vorge-

43. `https://www.cia.gov/library/publications/the-world-factbook/`

nommen, um unter anderem die politischen Veränderungen seit 1998 einfließen zu lassen. Hier wird diese aktualisierte Version verwendet.

Das XML-Dokument `mondial.xml` besteht aus insgesamt 30822 Elementen, davon 238 `country`-Elemente, die wiederum als Nachfahren unter anderem 3111 `city`-Elemente enthalten. Neben Städten und Ländern sind auch Angaben zu Bergen, Seen, Flüssen, Kontinenten, Inseln, Wüsten und vielem mehr enthalten. Die Tabelle 6–2 zeigt einige statistische Angaben zu der XML-Datei. Abbildung 6–1 gibt zur Verdeutlichung der Struktur einen kurzen Ausschnitt aus dem XML-Dokument und der zugehörigen DTD wieder.

Das XML-Dokument „mondial.xml“	
Dateigröße	1810233 Byte (1,73 MB)
Gesamtzahl der Elemente	30822 Elemente
Gesamtzahl der Attribute	30015 Attribute
Anzahl der verschiedenen Elemente	48 Elemente
Anzahl der verschiedenen Attribute	22 Attribute
Maximale Anzahl an Attributen eines Elements	5 Attribute
Maximale Element-Verschachtelungstiefe	5 Ebenen
Maximale Anzahl an Kindelementen eines Elements	1357 Elemente
Gesamtlänge aller Attributwerte	449005 Zeichen
Gesamtlänge aller Textknoteninhalte	138688 Zeichen
Gesamtlänge aller ignorierbaren Whitespaces	330625 Zeichen

Tab. 6–2 *Statistische Angaben zum XML-Dokument „mondial.xml“*[44]

Das Dokument `mondial.xml` repräsentiert ein typisches datenzentriertes XML-Dokument. Es gibt eine relativ ausgewogene Mischung von Elementen und Attributen, maximal sind fünf Elemente ineinander geschachtelt. Es treten bei einem Element bis zu 1357 Kindelemente auf. Der Textinhalt der Elemente und die Attributwerte bestehen aus kurzen Angaben, die auch in ähnlicher Form in den Spalten einer relationalen Tabelle abgelegt sein könnten. Mit einer Größe von 1,73 MB zählt das Dokument eher zu den größeren XML-Dateien. Aufgrund der Formatierung des Dokuments tritt eine recht hohe Anzahl an ignorierbaren Whitespaces auf.

44. Die maximale Element-Verschachtelungstiefe gibt die maximale Anzahl an Elementebenen im XML-Baum an und entspricht damit der üblichen Definition der Baumtiefe im Bereich Algorithmen und Datenstrukturen. Ein XML-Dokument, das nur aus einem Wurzelelement besteht, besitzt die maximale Element-Verschachtelungstiefe 1.

```
<!ELEMENT mondial (country*,continent*,organization*,
                   sea*,river*,lake*,island*,mountain*,desert*)>
<!ELEMENT country (name,population?,
                   population_growth?,infant_mortality?,gdp_total?,gdp_agri?,gdp_ind?,
                   gdp_serv?,inflation?,(indep_date|dependent)?,government?,encompassed*,
                   ethnicgroups*,religions*,languages*,border*,(province+|city+))>
<!ATTLIST country car_code ID #IMPLIED
                  area CDATA #IMPLIED
                  capital IDREF #IMPLIED
                  memberships IDREFS #IMPLIED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT area (#PCDATA)>
<!ELEMENT population (#PCDATA)>
<!-- note that population is also a subelement of city -->
<!ATTLIST population year CDATA #IMPLIED>
<!ELEMENT population_growth (#PCDATA)>
<!ELEMENT infant_mortality (#PCDATA)>
<!ELEMENT gdp_total (#PCDATA)>
<!ELEMENT gdp_ind (#PCDATA)>
<!ELEMENT gdp_agri (#PCDATA)>
...
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mondial SYSTEM "mondial.dtd">
<mondial>
   <country car_code="AL" area="28750" capital="cty-cid-cia-Albania-Tirane"
            memberships="org-BSEC org-CE org-CCC org-ECE org-EBRD org-FAO org-IAEA ...">
      <name>Albania</name>
      <population>3249136</population>
      <population_growth>1.34</population_growth>
      <infant_mortality>49.2</infant_mortality>
      <gdp_total>4100</gdp_total>
      <gdp_agri>55</gdp_agri>
      <inflation>16</inflation>
      <indep_date>1912-11-28</indep_date>
      <government>emerging democracy</government>
      <encompassed continent="europe" percentage="100"/>
      <ethnicgroups percentage="3">Greeks</ethnicgroups>
      <ethnicgroups percentage="95">Albanian</ethnicgroups>
      <religions percentage="70">Muslim</religions>
      <religions percentage="10">Roman Catholic</religions>
      <religions percentage="20">Christian Orthodox</religions>
      <border country="GR" length="282"/>
      <border country="MK" length="151"/>
      <border country="MNE" length="172"/>
      <border country="KOS" length="112"/>
      <city id="cty-cid-cia-Albania-Tirane" is_country_cap="yes" country="AL">
         <name>Tirane</name>
         <longitude>19.8</longitude>
         <latitude>41.3</latitude>
         <population year="87">192000</population>
      </city>
      <city id="stadt-Shkoder-AL-AL" country="AL">
         ...
```

Abb. 6–1 *Ausschnitt aus der DTD und dem XML-Dokument zur Mondial-Datenbank*

Das XML-Dokument „bibel.xml“

Das XML-Dokument `bibel.xml` beinhaltet den Text der Lutherbibel von 1546. Die Datei genügt der Auszeichnungssprache Zefania XML, mit der Bibeltexte strukturiert abgelegt werden können. Die Definition von Zefania XML liegt als XML-Schema vor. Die XML-Datei und das zugehörige Schema stehen auf dem Web-Portal SourceForge zum freien Download zur Verfügung.[45] In der Tabelle 6–3 sind statistische Angaben zum XML-Dokument zusammengestellt.

Das XML-Dokument „bibel.xml“	
Dateigröße	1120635 Byte (1,07 MB)
Gesamtzahl der Elemente	7379 Elemente
Gesamtzahl der Attribute	7424 Attribute
Anzahl der verschiedenen Elemente	17 Elemente
Anzahl der verschiedenen Attribute	11 Attribute
Maximale Anzahl an Attributen eines Elements	5 Attribute
Maximale Element-Verschachtelungstiefe	4 Ebenen
Maximale Anzahl an Kindelementen eines Elements	80 Elemente
Gesamtlänge aller Attributwerte	12817 Zeichen
Gesamtlänge aller Textknoteninhalte	890551 Zeichen
Gesamtlänge aller ignorierbaren Whitespaces	29983 Zeichen

Tab. 6–3 *Statistische Angaben zum XML-Dokument „bibel.xml“*

Bei der Datei `bibel.xml` handelt es sich um einen typischen Vertreter der so genannten dokumenten- oder textzentrierten XML-Dokumente. Die Struktur wird im Wesentlichen durch lange Textinhalte geprägt, deren zugehörige Elementknoten nur eine recht geringe Verschachtelungstiefe aufweisen. Fast 80 % aller Zeichen des Dokuments gehören zu einem der Textknoten.

Das XML-Dokument „purchaseOrder.xml“

Das Dokument `purchaseOrder.xml` wird in [W3C04c] unter der Bezeichnung `po.xml` als Beispiel verwendet.[46] Mit nur 1039 Zeichen ist dieses Testdokument sehr kurz und dient der Prüfung des Verhaltens des Abbildungsverfahrens bei kleinen XML-Dateien.

Die Struktur des Dokuments ist datenzentriert, alle Angaben sind einzeln in den verschiedenen Elementen und Attributen abgelegt. Je Element existiert maxi-

45. Originaldateiname: `sf_lu1546_rev1.zip`; Downloadlink: `http://sourceforge.net/projects/zefania-sharp/files/Zefania%20XML%20Modules%20%28old%29/Bibles%20GER/Luther%20%28original%29/`
46. Zur Überraschung des Autors enthält das im W3C-Dokument wiedergegebene Beispiel einen Syntaxfehler in einem der schließenden Element-Tags.

mal ein Attribut. Die statistischen Angaben dieses Dokuments sind in der unten stehenden Tabelle 6–4 aufgelistet.

Das XML-Dokument „purchaseOrder.xml“	
Dateigröße	1039 Byte (1 KB)
Gesamtzahl der Elemente	25 Elemente
Gesamtzahl der Attribute	5 Attribute
Anzahl der verschiedenen Elemente	15 Elemente
Anzahl der verschiedenen Attribute	3 Attribute
Maximale Anzahl an Attributen eines Elements	1 Attribut
Maximale Element-Verschachtelungstiefe	4 Ebenen
Maximale Anzahl an Kindelementen eines Elements	5 Elemente
Gesamtlänge aller Attributwerte	26 Zeichen
Gesamtlänge aller Textknoteninhalte	181 Zeichen
Gesamtlänge aller ignorierbaren Whitespaces	207 Zeichen

Tab. 6–4 *Statistische Angaben zum XML-Dokument „purchaseOrder.xml“*

Das XML-Dokument „SigmodRecord.xml“

Die Datei `SigmodRecord.xml` enthält ein Verzeichnis von Artikeln aus den ACM SIGMOD Record Zeitschriften. Das Dokument ist knapp ein halbes Megabyte groß und steht stellvertretend für mittelgroße datenzentrierte XML-Dokumente. Es existieren verschiedene Varianten dieses XML-Artikelverzeichnisses. Die hier verwendete stammt aus dem XML Data Repository der University of Washington, das eine Sammlung verschiedener XML-Dokumente für Forschungszwecke bereithält.[47] Bei Betrachtung der in Tabelle 6–5 zusammengestellten Dokumenteigenschaften fällt besonders auf, dass nur sehr wenige Elementtypen und nur ein einziger Attributtyp genutzt werden.

Zusammenfassend stehen für die in den nächsten beiden Abschnitten beschriebenen Zeit- und Speicherbedarfsanalysen vier Testdokumente zur Verfügung: neben dem eher großen Dokument `mondial.xml`, ein sehr kleines (`purchaseOrder.xml`) und ein mittelgroßes Dokument, das sich dadurch auszeichnet, dass es nur einen Attributtyp verwendet (`SigmodRecord.xml`). Diese drei Dokumente sind datenzentriert. Für die Untersuchung des Verhaltens bei dokumentenzentrierten Dateien wird das XML-Dokument `bibel.xml` genutzt.

47. `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`

Das XML-Dokument „SigmodRecord.xml“	
Dateigröße	478336 Byte (467 KB)
Gesamtzahl der Elemente	11526 Elemente
Gesamtzahl der Attribute	3737 Attribute
Anzahl der verschiedenen Elemente	11 Elemente
Anzahl der verschiedenen Attribute	1 Attribut
Maximale Anzahl an Attributen eines Elements	1 Attribut
Maximale Element-Verschachtelungstiefe	6 Ebenen
Maximale Anzahl an Kindelementen eines Elements	89 Elemente
Gesamtlänge aller Attributwerte	7474 Zeichen
Gesamtlänge aller Textknoteninhalte	146015 Zeichen
Gesamtlänge aller ignorierbaren Whitespaces	70545 Zeichen

Tab. 6–5 *Statistische Angaben zum XML-Dokument „SigmodRecord.xml“*

6.3 Analyse des zeitlichen Programmablaufs

In diesem Abschnitt werden der Ablauf und die Ergebnisse der durchgeführten Zeitmessungen beschrieben. Die Untersuchungen dienen dazu, einen Überblick über die Verteilung der Ausführungszeiten auf die einzelnen Bearbeitungsschritte zu erhalten. Der sich anschließende Abschnitt 6.3.1 widmet sich dem Einfüge-, der Abschnitt 6.3.2 dem Auslesevorgang.

Die nötigen Messungen wurden ausgeführt, indem das Programm `XML2DB` auf dem Datenbankserver selbst gestartet wurde, sodass die Datenübertragung über das Netzwerk keinen Einfluss auf die Ergebnisse haben kann. Auf dem Server liefen keine anderen Anwendungen, außer systemeigene. Um die Zeitmessungen direkt aus dem Programm `XML2DB` heraus durchführen zu können, wurde die Klasse `Stopwatch` implementiert. `Stopwatch` verwendet die Methode `nanoTime` der `System`-Klasse, die den aktuellen Wert des präzisesten verfügbaren Systemzeitgebers liefert. Die Angabe erfolgt in Nanosekunden, dies bedeutet aber nicht, dass die Zeitmessung auch nanosekundengenau erfolgt. Auf den meisten Systemen erreicht man eine Genauigkeit im Mikrosekundenbereich. Die gemessenen Zeiten werden am Programmende auf der Standardausgabe angezeigt.

Um möglichst aussagekräftige Messergebnisse zu erhalten, wurde das Programm jeweils zehnmal mit dem gleichen XML-Dokument gestartet. Für jede Ausführung wurden die Bearbeitungszeiten festgehalten und anschließend die Mittelwerte der Gesamt- und Teilausführungszeiten ermittelt. Diese Mittelwerte sind in den Tabellen in den folgenden Unterabschnitten angegeben.

6.3.1 Einfügevorgang

Die Schwankungen der Gesamtausführungszeiten beim Einfügen liegen im Allgemeinen bei unter ±4 %. In Einzelfällen kann es insbesondere bei sehr kurzen Dokumenten, deren absolute Verarbeitungszeit klein ist, zu Abweichungen von bis zu 17 % kommen. Die Ursache für diese relativ große Bandbreite der Messwerte liegt darin, dass ein modernes Multitasking-Betriebssystem wie Windows Server 2008 im Hintergrund permanent Prozesse erzeugt, um bestimmte Wartungsaufgaben durchzuführen. Eine Einflussmöglichkeit auf diese dadurch ausgelösten Prozesswechsel besteht nicht.

6.3.1.1 Die einzelnen Verarbeitungsschritte

Beim Programm `XML2DB` ist von Interesse, wie sich die Ausführungszeit auf die verschiedenen Bearbeitungsschritte aufteilt. Hierzu wurden Untersuchungen mit den in Abschnitt 6.2 vorgestellten XML-Dokumenten durchgeführt. Die Ergebnisse werden im Folgenden besprochen. In der Tabelle 6–6 ist zunächst dargestellt, wie hoch die Gesamtausführungszeit bei den vier XML-Dokumenten ist und wie sich die benötigte Zeit auf die einzelnen Programmabschnitte aufteilt.

	mondial.xml mit DTD	bibel.xml mit DTD	purchaseOrder.xml mit DTD	SigmodRecord.xml mit DTD
Gesamtzeit	18,275 s	6,820 s	0,515 s	7,913 s
Parsen	0,939 s	0,760 s	0,352 s	1,340 s
Altes Dokument löschen	1,030 s	0,300 s	0,028 s	0,363 s
Sonstige DB-Operationen	0,080 s	0,075 s	0,084 s	0,084 s
Dokument einfügen	16,227 s	5,685 s	0,051 s	6,127 s

Tab. 6–6 *Benötigte Ausführungszeit beim Einfügen bei verschiedenen XML-Dokumenten*

In der Tabelle sind jeweils neben der Gesamtausführungszeit die Zeiten angegeben, die für die Schritte „Parsen“, „Altes Dokument löschen“, „Sonstige DB-Operationen“ sowie „Dokument einfügen“ benötigt werden.

Der Schritt „Parsen“ umfasst die Instanziierung des Xerces-DOM-Parsers sowie das eigentliche Parsen des XML-Dokuments und die damit zusammenhängende Erzeugung des DOM-Baums im Arbeitsspeicher. „Altes Dokument löschen“ bedeutet, dass ein Dokument, das eventuell unter der zu nutzenden Dokument-ID bereits in der Datenbank gespeichert ist, gelöscht wird, bevor das neue Dokument unter dieser ID eingefügt werden kann. Der Zeitbedarf dieses Vorgangs hängt entscheidend davon ab, ob die Dokument-ID zum Zeitpunkt des Einfügens des neuen Dokuments bereits benutzt wird und davon, wie umfangreich das zu entfernende Dokument ist. In der Tabelle 6–6 bezieht sich die Zeitangabe auf ein Dokument, das identisch mit dem neu einzufügenden Dokument ist.

Unter dem Schritt „Sonstige DB-Operationen“ sind der Aufbau der JDBC-Verbindung zur Informix-Datenbank, die Vorbereitung der SQL-Kommandos sowie einige kurze abschließende Tätigkeiten am Ende des Programmablaufs zusammengefasst. Die letzte Zeile der Tabelle gibt schließlich die Zeit für das eigentliche Einfügen des Dokuments an. Hierzu gehören das rekursive Durchlaufen des DOM-Baums, die Auswertung der gelesenen Werte in den Knoten sowie das Einfügen der entsprechenden Zeilen in die Datenbank.

Man erkennt anhand der Tabelle, dass der Schritt „Dokument einfügen“ bei drei der vier untersuchten XML-Dokumente den größten Teil der Bearbeitungszeit einnimmt. In Abbildung 6–2 ist die prozentuale Verteilung der Ausführungszeit auf die einzelnen Schritte für alle vier Dokumente grafisch dargestellt.

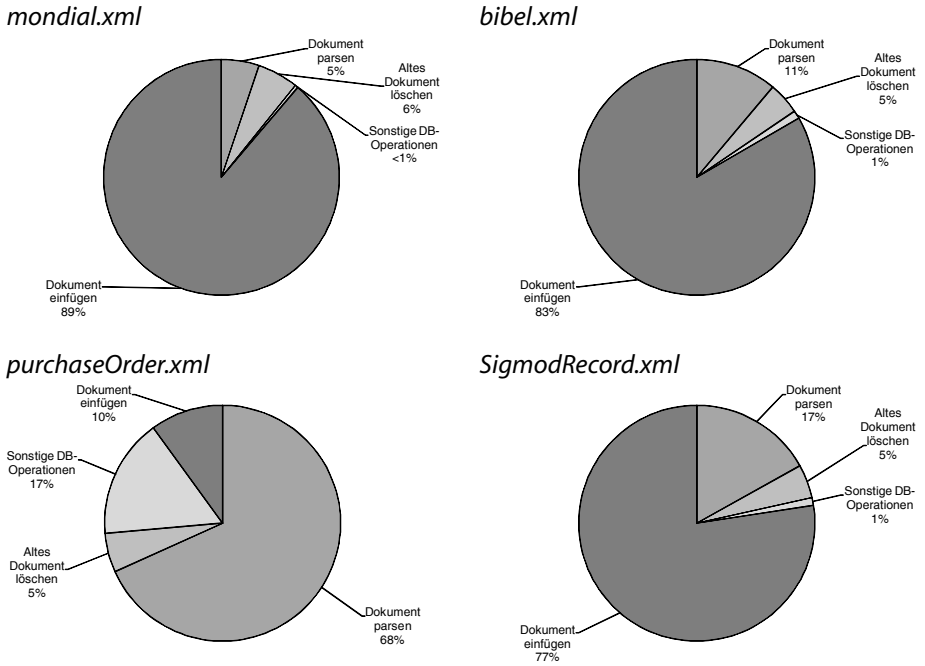


Abb. 6–2 *Prozentuale Verteilung der Ausführungszeit beim Einfügen*

Für die Dokumente `mondial.xml`, `bibel.xml` und `SigmodRecord.xml` ergibt sich eine ähnliche Aufteilung der Zeiten: zwischen 5 und 17 % der Zeit wird für das Parsen des jeweiligen Dokuments benötigt, das Löschen des alten Dokuments benötigt 5 bis 6 % und die sonstigen Datenbankoperationen maximal 1 % der Ausführungszeit. Bis zu 89 % der Bearbeitungszeit entfällt auf den eigentlichen Einfügevorgang, der die Hauptaufgabe des Programms darstellt.

Für das Dokument `purchaseOrder.xml` lässt sich aus Abbildung 6–2 leicht ablesen, dass es hier zu einer deutlich abweichenden Verteilung zwischen den einzelnen Verarbeitungsschritten kommt. Zusammen mit Tabelle 6–6 erkennt man,

dass für die Verschiebung im Wesentlichen der Schritt „Dokument parsen“ verantwortlich ist. Beim Dokument `purchaseOrder.xml` handelt es sich um ein sehr kleines Dokument. Der Vorgang des Parsens wird jedoch nicht im gleichen Verhältnis kürzer, wie das XML-Dokument im Vergleich zu den anderen Dokumenten kleiner ist. Die Ursache ist hierbei in der internen Verarbeitung durch den Xerces-Parser zu suchen. Auch bei einem sehr kleinen Dokument müssen Initialisierungen durchgeführt, Objekte instanziiert sowie die Datei gesucht und zum Lesen geöffnet werden. Diese Operationen benötigen hier einen größeren Zeitanteil.

6.3.1.2 Analyse der Einfügeroutine

Im Folgenden soll der Verarbeitungsschritt „Dokument einfügen“ genauer analysiert werden. Dieser Schritt wurde im vorangegangen Abschnitt als derjenige identifiziert, der bei drei der vier Testdokumente den größten Anteil der Ausführungszeit benötigt.

Der Schritt „Dokument einfügen“ besteht im Wesentlichen aus einer Methode, die sich selbst aufruft, um den DOM-Baum rekursiv zu durchlaufen. In Abhängigkeit der Knoten, auf die die Routine trifft, werden entsprechende Zeilen in die Datenbanktabelle eingefügt. Hierzu werden passende SQL-`INSERT`-Kommandos erzeugt. Von Interesse ist nun, wie viel der Zeit, die der Schritt „Dokument einfügen“ benötigt, auf die Verarbeitung innerhalb des Java-Programms entfällt und wie groß die Anteile sind, in denen auf die Ausführung der Kommandos durch die Informix-Datenbank gewartet wird. Zu dieser Datenbankbearbeitungszeit gehört auch die Zeit, die für die Übergabe der Werte an den JDBC-Treiber benötigt wird (JDBC-Handling).

Die entsprechenden Zeiten wurden wiederum für die vier XML-Testdokumente ermittelt. Sie sind der Tabelle 6–7 zu entnehmen. In der letzten Tabellenzeile ist der Anteil der Zeit, in der auf die Datenbank gewartet wird, bezogen auf die Gesamtzeit, die die Einfügeroutine benötigt, angegeben. Bei drei der vier XML-Dokumente wird über 85 % der Zeit mit der Ausführung der SQL-Kommandos verbracht. Beim Dokument `purchaseOrder.xml` ist dieser Wert niedriger, da das Dokument sehr klein ist und somit das Verhältnis zwischen einmalig und mehrmals ausgeführten Operationen nicht dem der größeren Dokumente entspricht.

Festzuhalten bleibt, dass das Programm `XML2DB` selbst – außer bei sehr kleinen Dokumenten – nur einen geringen Teil der Rechenzeit benötigt und dass die Datenbankoperationen den Hauptteil der Ausführungszeit beanspruchen.

	mondial.xml mit DTD	bibel.xml mit DTD	purchaseOrder.xml mit DTD	SigmodRecord.xml mit DTD
Dokument einfügen	16,227 s	5,685 s	0,051 s	6,127 s
DB-Einfügeoperationen	14,946 s	4,884 s	0,036 s	5,514 s
– davon für JDBC-Handling	1,203 s	0,513 s	0,020 s	0,686 s
Anteil DB-Einfügeoperationen	92,1 %	85,9 %	70,6 %	90,0 %

Tab. 6–7 *Zeitanteil der Datenbankeinfügeoperationen*

6.3.1.3 Verteilung der Datenbankbearbeitungszeit

Die Frage ist, wie sich die Zeit für die Datenbankeinfügeoperationen zusammensetzt. Die hierzu ermittelten Werte sind in Tabelle 6–8 dargestellt.

	mondial.xml mit DTD	bibel.xml mit DTD	purchaseOrder.xml mit DTD	SigmodRecord.xml mit DTD
DOM-Knoten	51397	14494	46	19912
Datenbankeinträge	30825	7384	27	11529
Elementeinträge	30823	7380	26	11527
– davon mit Text	20572	7110	19	8383
Text- und CDATA-Einträge	0	0	0	0
Kommentareinträge	1	3	0	1
Proz.instr.-Einträge	0	0	0	0
DocType-Einträge	1	1	1	1
DB-Einfügeoperationen	14,946 s	4,884 s	0,036 s	5,514 s
– davon für Elemente	14,929 s	4,866 s	0,022 s	5,503 s
– davon für Texte/CDATA	0,000 s	0,000 s	0,000 s	0,000 s
– davon für Kommentare	0,000 s	0,016 s	0,000 s	0,011 s
– davon für Proz.instr.	0,000 s	0,000 s	0,000 s	0,000 s
– davon für DocType-Ang.	0,016 s	0,001 s	0,015 s	0,001 s

Tab. 6–8 *Knotenzahl, Eintragstypen und jeweiliger Zeitbedarf*

Im oberen Teil der Tabelle sind statistische Daten zu den vier untersuchten XML-Dokumenten angegeben. Hierzu gehört die Gesamtzahl der Knoten im DOM-Baum sowie die Anzahl der Datenbankeinträge, die für die verschiedenen Knotentypen erzeugt werden. Interessant ist hierbei insbesondere die Zahl der Elementeinträge[48] und die genannte Anzahl an Elementeinträgen mit Text. Darunter sind Datenbankeinträge zu verstehen, die für ein Element erzeugt werden, und

48. Der Eintrag für die DOM-Wurzel wird hier zu den Elementeinträgen gezählt.

die den Textinhalt dieses Elements direkt mit aufnehmen. Dies ist immer dann möglich, wenn es sich nicht um ein Element mit gemischtem Inhalt handelt und die Textlänge nicht zu groß ist.[49] Die Aufnahme des Textinhalts in den Elementeintrag selbst hat den großen Vorteil, dass mit einem Datenbankeintrag zwei DOM-Knoten repräsentiert werden: der Elementknoten und der zugehörige Textknoten.

Die unteren Zeilen der Tabelle 6–8 geben an, wie viel Zeit für das Einfügen der verschiedenen Datenbankeinträge benötigt wird.[50] Es erfolgt hierbei die gleiche Unterteilung wie bei der Auflistung der Anzahl der zugehörigen Einträge. Interessante Werte ergeben sich nur für die Elementeinträge, da die sonstigen Eintragstypen in den hier genutzten XML-Dokumenten nur in sehr geringer Anzahl vorkommen.

Bei allen vier Dokumenten enthalten mindestens zwei Drittel der Elementeinträge auch Text. Die sonstigen Elementeinträge enthalten keinen Text, weil es sich um leere Elemente oder um Elemente handelt, die ausschließlich Unterelemente aufweisen. Eigenständige Einträge für Textinhalte müssen nicht erzeugt werden. Dies hat zur Folge, dass fast die gesamte Zeit, die für Datenbankeinfügeoperationen aufgewandt wird, auf das Einfügen der Elementeinträge entfällt.

6.3.1.4 Zeitbedarf je DOM-Knoten

Zur besseren Vergleichbarkeit der Messergebnisse für die vier XML-Dokumente wurde der Zeitbedarf je DOM-Knoten sowie je XML-Element und je Datenbankeinfügeoperation ermittelt. Die Werte sind in der unten stehenden Tabelle 6–9 angegeben.

Zeitbedarf	mondial.xml mit DTD	bibel.xml mit DTD	purchaseOrder.xml mit DTD	SigmodRecord.xml mit DTD
je XML-Element	0,529 ms	0,780 ms	5,224 ms	0,539 ms
je DB-Einfügeoperation	0,529 ms	0,780 ms	5,030 ms	0,539 ms
je DOM-Knoten	0,317 ms	0,397 ms	2,952 ms	0,312 ms

Tab. 6–9 *Zeitbedarf beim Einfügen je XML-Element, je DB-Einfügeoperation und je DOM-Knoten*

Als Basis für die Werte aus der Tabelle wurde jeweils die Zeitdauer für das Einfügen des Dokuments ohne Parsen und ohne Löschen des alten Dokuments genutzt. Die letzte Zeile der Tabelle, die die jeweilige Zeitdauer je DOM-Knoten nennt, hat die größte Aussagekraft. Bei drei der vier Dokumente liegt diese Zeit um etwa 0,35 ms. Es handelt sich hierbei um einen recht stabilen Wert, der von dem konkreten XML-Dokument relativ unabhängig ist. Dass das `purchaseOr-`

49. Details hierzu finden sich in Abschnitt 5.4.2.
50. Die Zeiten enthalten jeweils auch die anteilige Zeit für das JDBC-Handling.

der-Dokument abweicht, liegt, wie bereits zuvor erläutert, an der Kürze der Datei mit nur 46 DOM-Knoten. Alle Messergebnisse aus Tabelle 6–9 sind in Abbildung 6–3 nochmals grafisch aufbereitet.

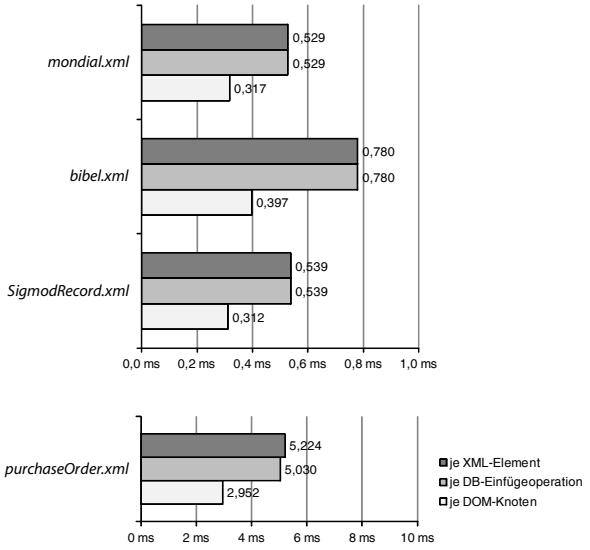


Abb. 6–3 *Zeitbedarf beim Einfügen je XML-Element, je DB-Einfügeoperation und je DOM-Knoten*

Untersuchung des Zusammenhangs von DOM-Knotenanzahl und Zeitbedarf anhand einer auf dem purchaseOrder-Dokument basierenden Testreihe

Um den Einfluss der Anzahl der DOM-Knoten auf den Zeitbedarf beim Einfügen in die Datenbank genauer untersuchen zu können, wurde eine passende Messreihe durchgeführt. Die dazu nötigen Testdokumente sind entstanden, indem, basierend auf dem Dokument `purchaseOrder.xml`, das XML-Fragment unterhalb des Wurzelelements durch Anhängen einer identischen Kopie schrittweise verdoppelt wurde. Für alle diese Dokumente wurde der mittlere Zeitbedarf der einzelnen Verarbeitungsschritte beim Einfügen in die Datenbank ermittelt. In Abbildung 6–4 ist der Zeitbedarf der Datenbankeinfügeoperationen in Abhängigkeit von der Anzahl der DOM-Knoten wiedergegeben. Der Graph nähert sich mit steigender Knotenanzahl einem lineraren Verlauf an. Das Verhalten des Zeitbedarfs je DOM-Knoten[51] ist in Abbildung 6–5 dargestellt. Die auftretenden Skalierungseffekte zeigen sich deutlich: ab etwa 2000 DOM-Knoten wird ein Zeitbedarf von weniger als 0,5 ms je DOM-Knoten erreicht.

51. Basis ist hier wie zuvor die gesamte Zeitdauer des Einfügens ohne Parsen und Löschen des alten Dokuments.

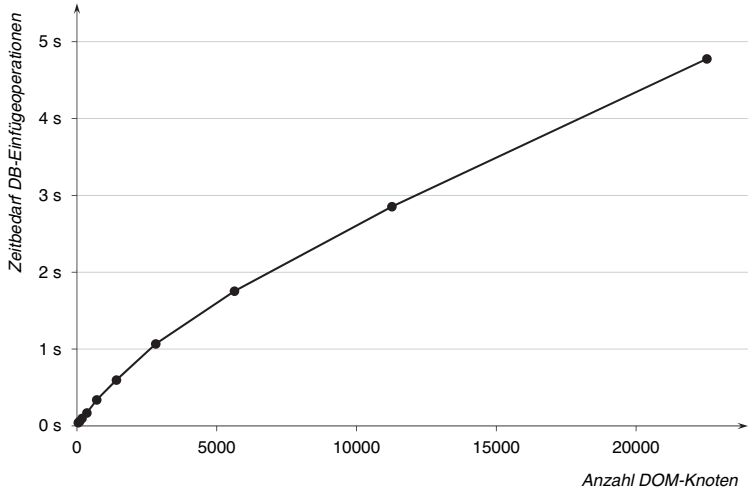


Abb. 6–4 *Zeitbedarf der Datenbankeinfügeoperationen*

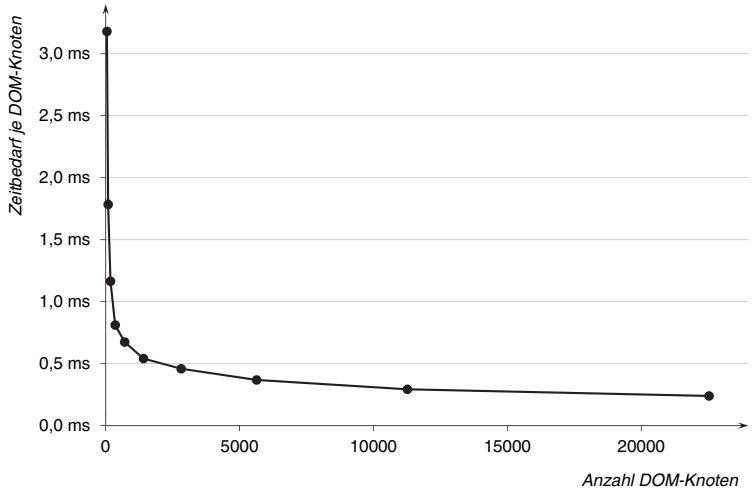


Abb. 6–5 *Zeitbedarf je DOM-Knoten beim Einfügen*

6.3.1.5 Einfluss der Einbindung einer DTD

Zum Abschluss des Abschnitts 6.3.1 wird an dieser Stelle der Einfluss untersucht, den die Verwendung bzw. Nichtverwendung einer DTD auf das Abbildungsverfahren hat.

Als Vorbemerkung sei darauf hingewiesen, dass das Abbildungsprogramm `XML2DB` so eingestellt ist, dass erkannte ignorierbare Whitespaces beim Einfügen

der Zeilen in die ELEMENTS-Tabelle übergangen und somit nicht abgespeichert werden. Falls ein XML-Dokument ohne eingebundene DTD vorliegt, so können keine ignorierbaren Whitespaces erkannt werden. Das bedeutet, dass diese Zeichenfolgen als reguläre Texte auf Einträge in der ELEMENTS-Tabelle abgebildet werden.

Zu den vier in Abschnitt 6.2 vorgestellten Testdokumenten ist jeweils eine passende DTD vorhanden. Den dort angegebenen statistischen Daten und auch den bisherigen Messungen dieses Abschnitts liegen eingebundene DTDs zugrunde.

Für die beiden Testdokumente `mondial.xml` und `bibel.xml` wurden die gleichen Zeitmessungen ein weiteres Mal durchgeführt, wobei keine DTD eingebunden wurde. Die dabei erhaltenen Ergebnisse sind in den Tabellen 6–10 und 6–11 den bisherigen Resultaten gegenübergestellt.

	mondial.xml mit DTD	mondial.xml ohne DTD	bibel.xml mit DTD	bibel.xml ohne DTD
DOM-Knoten	51397	88385	14494	22140
Datenbankeinträge	30825	67813	7384	15030
Elementeinträge	30823	30823	7380	7380
– davon mit Text	20572	20572	7110	7110
Text- und CDATA-Einträge	0	36989	0	7647
Kommentareinträge	1	1	3	3
Proz.instr.-Einträge	0	0	0	0
DocType-Einträge	1	0	1	0
DB-Einfügeoperationen	14,946 s	23,432 s	4,884 s	7,415 s
– davon für Elemente	14,929 s	12,733 s	4,866 s	4,216 s
– davon für Texte/CDATA	0,000 s	10,698 s	0,000 s	3,188 s
– davon für Kommentare	0,000 s	0,001 s	0,016 s	0,011 s
– davon für Proz.instr.	0,000 s	0,000 s	0,000 s	0,000 s
– davon für DocType-Ang.	0,016 s	0,000 s	0,001 s	0,000 s

Tab. 6–10 *Vergleich von Knotenzahl, Eintragstypen und jeweiligem Zeitbedarf beim Einfügen mit und ohne DTD*

Die Tabelle 6–10 zeigt, dass die Gesamtzahl der DOM-Knoten bei Nichteinbindung einer DTD deutlich höher liegt. Beim Dokument `mondial.xml` ist sie etwa 72 %, beim Dokument `bibel.xml` etwa 53 % höher. Die zusätzlichen Knoten repräsentieren die ignorierbaren Whitespaces als Zeichenketten und werden in der Datenbank als eigenständige Texteinträge dargestellt. Die Differenz von einem Knoten zwischen der Anzahl der hinzugekommenen Knoten und der Anzahl der Texteinträge erklärt sich durch den Wegfall der Dokumenttypangabe, die nur bei Verwendung einer DTD vorhanden ist.

Die Zeit, die auf die Datenbankeinfügeoperationen entfällt, hat sich bei beiden Dokumenten bei Nichtnutzung der DTD erhöht. Der Zeitbedarf ist um etwa 57 % bzw. 52 % gestiegen. Ursache ist, dass deutlich mehr Einträge in die Datenbank eingefügt werden müssen.

Zeitbedarf	mondial.xml mit DTD	mondial.xml ohne DTD	bibel.xml mit DTD	bibel.xml ohne DTD
je XML-Element	0,529 ms	0,802 ms	0,780 ms	1,116 ms
je DB-Einfügeoperation	0,529 ms	0,365 ms	0,780 ms	0,548 ms
je DOM-Knoten	0,317 ms	0,280 ms	0,397 ms	0,372 ms

Tab. 6–11 *Vergleich des Zeitbedarfs beim Einfügen mit und ohne DTD*

In der Tabelle 6–11 wird der Zeitbedarf je XML-Element, je DB-Einfügeoperation und je DOM-Knoten für die beiden Testdokumente jeweils mit und ohne DTD angegeben. Als Basis dient wiederum die insgesamt benötigte Zeit ohne Parsen und ohne Löschen des alten Dokuments. Da die Anzahl der XML-Elemente unabhängig von der Verwendung einer DTD konstant bleibt, kommt es hier beim Nichteinbinden der DTD zu einer Erhöhung des Zeitbedarfs je XML-Element. Die Werte je DB-Einfügeoperation und je DOM-Knoten fallen, wenn keine DTD eingesetzt wird. Ursache für diese Änderungen sind Skalierungseffekte, die bewirken, dass diese Zeiten sinken, je häufiger Einfügeoperationen in die Datenbank vorgenommen werden. Gewisse einmalig auszuführende Aktionen, wie das Herstellen und Beenden der Datenbankverbindung, fallen bei einer höheren Anzahl an Einfügeoperationen weniger stark ins Gewicht.

Da auf die Speicherung der ignorierbaren Whitespaces unter Umständen je nach XML-Dokument ein beachtlicher Anteil des Gesamtzeitbedarfs entfällt, sollte stets eine DTD eingebunden und damit auf die Speicherung dieser Zeichenketten verzichtet werden. Für die allermeisten Anwendungen haben diese Whitespaces keine Bedeutung, und es kommt nicht zu Problemen, wenn diese ignoriert und nicht abgespeichert werden.

6.3.2 Auslesevorgang

Nach der Besprechung des Zeitbedarfs beim Einfügen eines Dokuments beschäftigt sich dieser Abschnitt mit der Untersuchung des Auslesevorgangs. Die folgenden Unterabschnitte und die zugehörigen Tabellen und Diagramme sind ähnlich wie zuvor beim Einfügevorgang strukturiert.

Die Schwankungen der Gesamtausführungszeiten liegen beim mehrmaligen Ausführen des Auslesevorgangs auf Basis des gleichen XML-Dokuments unter ±2 %. Bei sehr kurzen Testdokumenten, wie zum Beispiel `purchaseOrder.xml`, kann es jedoch auch zu stärkeren Abweichungen von bis zu 10 % kommen.

6.3.2.1 Die einzelnen Verarbeitungsschritte

In Tabelle 6–12 sind die Ergebnisse der Messungen für die vier Testdokumente zusammengefasst. Dabei wird die Gesamtausführungszeit sowie die auf die vier Teilschritte „Root-Node-ID ermitteln“, „Sonstige DB-Operationen“, „Dokument auslesen“ und „DOM-Serialisierung“ jeweils entfallene Zeit angegeben.

	mondial.xml mit DTD	bibel.xml mit DTD	purchaseOrder.xml mit DTD	SigmodRecord.xml mit DTD
Gesamtzeit	26,686 s	9,104 s	0,422 s	9,054 s
Root-Node-ID ermitteln	0,218 s	0,075 s	0,029 s	0,097 s
Sonstige DB-Operationen	0,263 s	0,245 s	0,263 s	0,271 s
Dokument auslesen	25,891 s	8,540 s	0,106 s	8,440 s
DOM-Serialisierung	0,313 s	0,244 s	0,024 s	0,246 s

Tab. 6–12 *Benötigte Ausführungszeit beim Auslesen bei verschiedenen XML-Dokumenten*

Der Ausleseprozess beginnt mit dem Schritt „Root-Node-ID ermitteln“. Hierbei wird anhand der Dokument-ID die Knoten-ID des Dokumentwurzelknotens aus der Datenbank ermittelt. Unter dem Begriff „Sonstige DB-Operationen“ sind das Herstellen und Trennen der Verbindung zur Datenbank sowie das Vorbereiten des SQL-Kommandos zusammengefasst.

Der Bearbeitungsschritt „Dokument auslesen“ ist der Kern des Auslesevorgangs. Hierzu gehört die Initialisierung des DOM-Baums sowie das Auslesen der einzelnen Tabellenzeilen aus der Datenbank und der rekursive Aufbau der Baumstruktur. Dieser Schritt wird im folgenden Unterabschnitt noch näher untersucht. Zuletzt erfolgt die so genannte DOM-Serialisierung. Darunter versteht man die Umwandlung des im Hauptspeicher befindlichen DOM-Baums in ein XML-Dokument, das anschließend in einer Datei gespeichert oder auf der Standardausgabe angezeigt wird.

In Abbildung 6–6 sind die Messergebnisse nochmals grafisch aufbereitet, um den Anteil der einzelnen Schritte an der Gesamtausführungszeit zu verdeutlichen. Es fällt auf, dass die drei Dokumente `mondial.xml`, `bibel.xml` und `SigmodRecord.xml` eine vergleichbare Aufteilung der Zeitanteile zeigen: Etwa 1 % des Zeitbedarfs entfällt auf die Ermittlung der Wurzel-Knoten-ID, bis zu 3 % auf die sonstigen DB-Operationen und mindestens 93 % auf das eigentliche Auslesen und den Aufbau des Dokuments. Die abschließende DOM-Serialisierung benötigt bis zu 3 % der Ausführungszeit.

Beim Dokument `purchaseOrder.xml` ergibt sich auch beim Auslesevorgang eine deutlich abweichende Aufteilung der Zeitanteile. Der Kernauslesevorgang beansprucht nur ein Viertel der Gesamtzeit, wohingegen die sonstigen Datenbankoperationen einen Anteil von 62 % aufweisen. Diese Verschiebung lässt sich wiederum mit der Kürze dieses Testdokuments erklären, wodurch es beim Ausle-

seprozess zu einem deutlich anderen Verhältnis zwischen einmalig und mehrmals auszuführenden Operationen kommt.

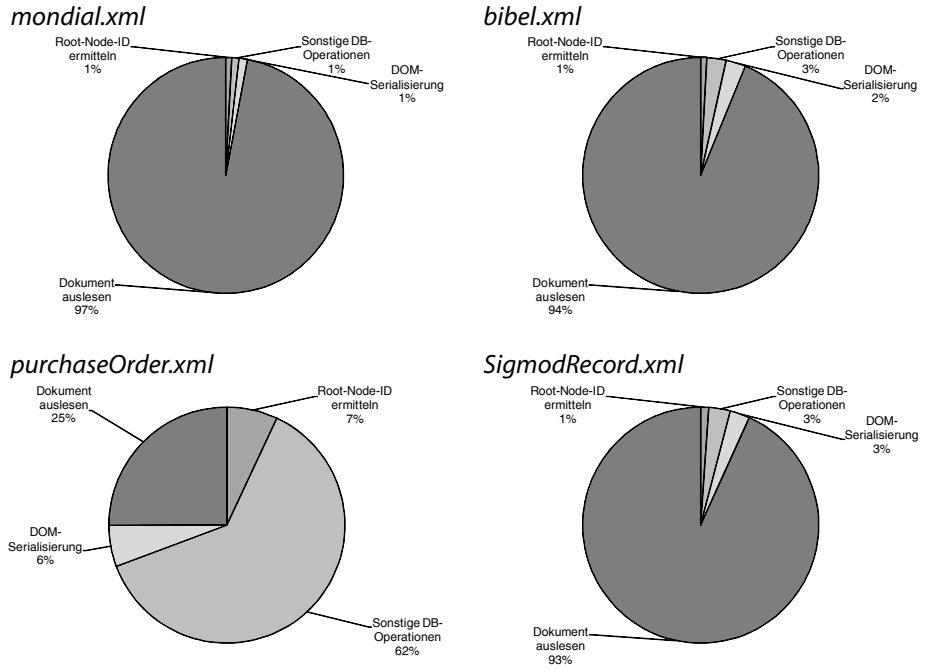


Abb. 6–6 *Prozentuale Verteilung der Ausführungszeit beim Auslesen*

6.3.2.2 Analyse der Ausleseroutine

Im vorhergehenden Abschnitt wurde gezeigt, dass der Schritt „Dokument auslesen“ bei drei der vier XML-Testdokumente den größten Zeitanteil benötigt. Zu diesem Schritt zählen das Auslesen der jeweils benötigten Zeile aus der Datenbanktabelle `ELEMENTS` mittels SQL-`SELECT`-Kommando, deren Auswertung und Behandlung entsprechend des Werts in der Spalte `NAME` sowie die Erweiterung des DOM-Baums um einen passenden Knoten.

Dabei ist nun von Interesse, wie groß der Zeitanteil ist, den das Programm `XML2DB` selbst für die Bearbeitung benötigt, und wie viel Zeit auf die Ausführung der SQL-Kommandos entfällt. Hierzu wurde der Zeitbedarf der Datenbankausleseoperationen getrennt erfasst. Die Messergebnisse finden sich in Tabelle 6–13.

Der Zeitbedarf für das so genannte JDBC-Handling wurde zusätzlich separat gemessen. Hierzu zählen die Befehle, mit denen Werte an das vorbereitete SQL-Kommando übergeben werden, und Operationen, mit deren Hilfe die Ergebnisse der Datenbankabfragen übertragen werden. Der Zeitbedarf für das JDBC-Handling wird zu der Zeit, die die Datenbank für die SQL-Ausführung benötigt,

gerechnet, da der JDBC-Treiber für die Bearbeitung dieser Operationen mit der Informix-Datenbank kommunizieren muss.

In der letzten Zeile der Tabelle 6–13 ist der Zeitanteil der Datenbankoperationen am Ausleseschritt angegeben. Man erkennt, dass bei allen Dokumenten (außer `purchaseOrder.xml`) der weitaus größte Anteil auf diese Datenbankbearbeitungszeit entfällt. Das heißt, die Ausführung der weiteren Operationen des Programms `XML2DB` benötigt nur relativ wenig Rechenzeit.

	mondial.xml mit DTD	bibel.xml mit DTD	purchaseOrder.xml mit DTD	SigmodRecord.xml mit DTD
Dokument auslesen	25,891 s	8,540 s	0,106 s	8,440 s
DB-Ausleseoperationen	24,130 s	7,568 s	0,062 s	7,793 s
– davon für JDBC-Handling	6,513 s	2,214 s	0,043 s	2,577 s
Anteil DB-Ausleseoperationen	93,2 %	88,6 %	58,5 %	92,3 %

Tab. 6–13 *Zeitanteil der Datenbankausleseoperationen*

6.3.2.3 Zeitbedarf je DOM-Knoten

Um die Auslesezeiten der vier Testdokumente besser vergleichen zu können, wurde die benötigte Ausführungszeit je XML-Element, je Datenbankleseoperation und je DOM-Knoten ermittelt. Die Werte sind in Tabelle 6–14 angegeben. Als Basiswert wurde die benötigte Zeit ohne den Schritt der DOM-Serialisierung genutzt.

Zeitbedarf	mondial.xml mit DTD	bibel.xml mit DTD	purchaseOrder.xml mit DTD	SigmodRecord.xml mit DTD
je XML-Element	0,856 ms	1,201 ms	15,305 ms	0,764 ms
je DB-Leseoperation	0,856 ms	1,200 ms	14,738 ms	0,764 ms
je DOM-Knoten	0,513 ms	0,611 ms	8,651 ms	0,442 ms

Tab. 6–14 *Zeitbedarf beim Auslesen je XML-Element, je DB-Leseoperation und je DOM-Knoten*

Es ergibt sich somit ein Zeitbedarf von etwa 0,4 bis 0,6 ms je im Dokument enthaltenen DOM-Knoten. Damit liegen die Auslesezeiten leicht über den jeweiligen Einfügezeiten (siehe Tab. 6–9). Da beim Dokument `purchaseOrder.xml` mit nur 46 DOM-Knoten keine Skalierungseffekte Wirkung zeigen können, weicht der Zeitbedarf bei dieser Testdatei deutlich nach oben ab. Einen grafischen Überblick über die erhaltenen Messwerte gibt Abbildung 6–7.

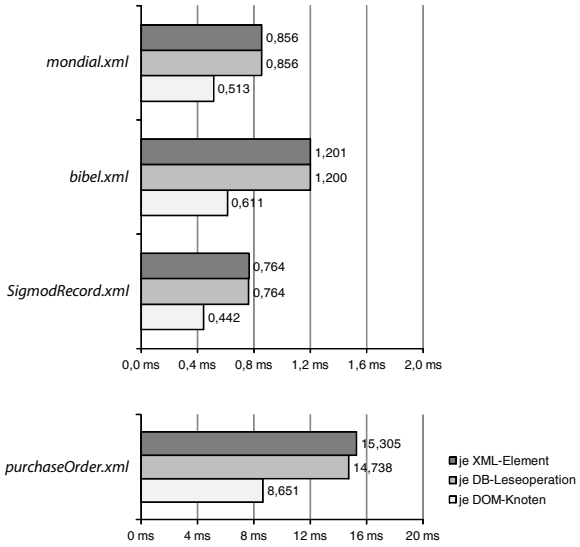


Abb. 6–7 *Zeitbedarf beim Auslesen je XML-Element, je DB-Leseoperation und je DOM-Knoten*

Untersuchung des Zusammenhangs von DOM-Knotenanzahl und Zeitbedarf anhand der purchaseOrder-Testdokumentreihe

Analog zu Abschnitt 6.3.1.4 wurde auch für den Auslesevorgang der Zusammenhang zwischen der Anzahl der DOM-Knoten und dem Zeitbedarf der Datenbankausleseoperationen sowie dem Zeitbedarf je DOM-Knoten[52] untersucht. Dazu wurde erneut die auf dem Dokument `purchaseOrder.xml` basierende Testdokumentreihe verwendet. Die erhaltenen Ergebnisse, die in den Abbildungen 6–8 und 6–9 dargestellt werden, sind mit den in Abschnitt 6.3.1.4 gezeigten vergleichbar. Mit steigender Knotenanzahl nähert sich der Graph in Abbildung 6–8, der den Zeitbedarf der DB-Ausleseoperationen zeigt, einem lineraren Verlauf an. Der Zeitbedarf je DOM-Knoten in Abbildung 6–9 erreicht ab etwa 6000 Knoten einen relativ stabilen Wert von weniger als 0,6 ms.

52. Grundlage ist die Gesamtdauer des Auslesens ohne Berücksichtigung der Zeit für die DOM-Serialisierung.

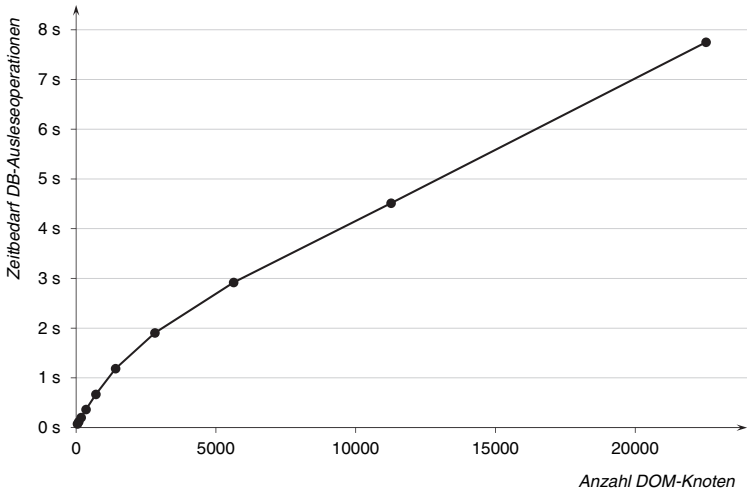


Abb. 6–8 *Zeitbedarf der Datenbankausleseoperationen*

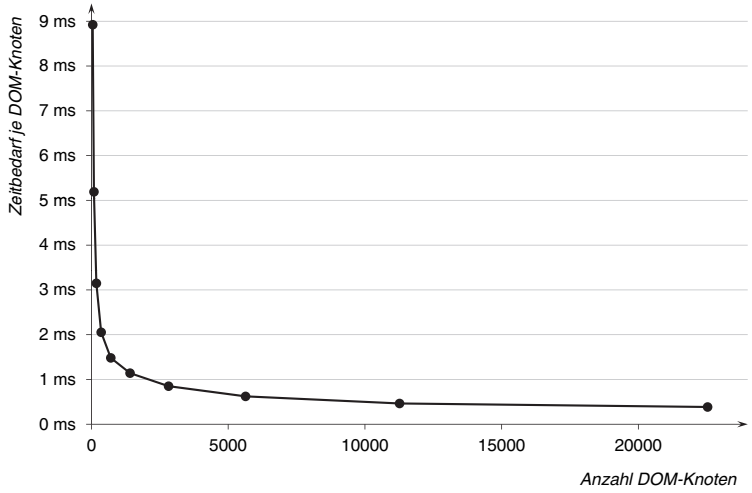


Abb. 6–9 *Zeitbedarf je DOM-Knoten beim Auslesen*

6.3.2.4 Einfluss der Einbindung einer DTD

Zum Abschluss der Zeitbedarfsmessungen soll nun auch beim Auslesevorgang der Einfluss einer DTD auf die Ausführungsgeschwindigkeit untersucht werden. Hierzu wurden die Testdokumente `mondial.xml` und `bibel.xml` zusätzlich ohne DTD-Einbindung in der Datenbank abgelegt und wieder ausgelesen. Die Tabelle 6–15 gibt die dabei erhaltenen Ergebnisse sowie die bisherigen Werte an.

Zeitbedarf	mondial.xml mit DTD	mondial.xml ohne DTD	bibel.xml mit DTD	bibel.xml ohne DTD
je XML-Element	0,856 ms	1,389 ms	1,201 ms	1,656 ms
je DB-Leseoperation	0,856 ms	0,631 ms	1,200 ms	0,813 ms
je DOM-Knoten	0,513 ms	0,485 ms	0,611 ms	0,552 ms

Tab. 6–15 *Vergleich des Zeitbedarfs beim Auslesen mit und ohne DTD*

Bei der Nichtnutzung einer DTD benötigt der gesamte Ausleseprozess mehr Zeit, umgerechnet auf einen einzelnen DOM-Knoten jedoch liegen die Werte nah beieinander. Der Zeitbedarf von etwa 0,5 ms je DOM-Knoten ist somit ein relativ stabiler Wert. Wie schon zuvor darauf hingewiesen, sollte dennoch nach Möglichkeit stets eine DTD genutzt werden, da so die unnötige Speicherung von Whitespaces verhindert wird.

6.4 Analyse des Speicherbedarfs

Neben der Ausführungszeit, die das Abbildungsverfahren benötigt, interessiert auch die für den Einfüge- und Auslesevorgang erforderliche Arbeitsspeichergröße sowie der belegte Speicherplatz in der Datenbank. Der jeweilige Speicherbedarf wird in den folgenden Unterabschnitten anhand der in Abschnitt 6.2 vorgestellten Testdokumente untersucht.

6.4.1 Arbeitsspeicherbedarf

Um die Effizienz des Abbildungsverfahrens besser beurteilen zu können, wurden Untersuchungen durchgeführt, die den Bedarf an Arbeitsspeicher während des Einfüge- und Auslesevorgangs analysieren. Hierbei sind bei einem Java-Programm sowohl die Größe des Heap- als auch die des so genannten Non-Heap-Speichers von Interesse.

Im Heap-Speicher werden alle Klasseninstanzen und Felder abgelegt. Der Non-Heap-Speicher wird verwendet, um interne Daten der Java Virtual Machine und Klassenstrukturen zu speichern. Hierzu zählen unter anderem Konstanten, statische Feld- und Methodendaten sowie der Code der Methoden und Konstruktoren.

Die Speichermessungen wurden durchgeführt, indem während der Verarbeitung zyklisch über das `MemoryMXBean`-Interface die zum jeweiligen Zeitpunkt belegte Menge an Heap- und Non-Heap-Speicher ermittelt wurde. Vor jeder Abfrage des Speicherbedarfs wurde eine Garbage Collection ausgelöst, um sicherzustellen, dass nur aktuell tatsächlich belegter Speicher berücksichtigt wird. Von den einzelnen Messwerten wurde schließlich das Maximum bestimmt.

Die Messungen des Arbeitsspeicherbedarfs für das Einfügen und Auslesen wurden für jedes der vier Testdokumente zehnmal durchgeführt. Die jeweiligen Mittelwerte dieser Messergebnisse sind in den folgenden Tabellen eingetragen. Die ermittelten minimalen und maximalen Messwerte weichen von dem zugehörigen Mittelwert in allen Fällen um weniger als 0,4 % ab.

6.4.1.1 Einfügen

Die Tabelle 6–16 zeigt die erhaltenen Werte für den Arbeitsspeicherbedarf beim Einfügen der vier Testdokumente. Zur besseren Vergleichbarkeit wurden in die Tabelle zusätzlich die Anzahl der DOM-Knoten und die Dateigröße der Dokumente aufgenommen. Des Weiteren wurde das Verhältnis der Größe des belegten Arbeitsspeichers zur Dokumentgröße sowie die Anzahl der Bytes aus dem Heap-Speicher errechnet, die auf einen DOM-Knoten entfallen.

	mondial.xml mit DTD	bibel.xml mit DTD	purchaseOrder.xml mit DTD	SigmodRecord.xml mit DTD
Anzahl DOM-Knoten	51397	14494	46	19912
Dokumentgröße	1768 KB	1094 KB	1 KB	467 KB
Max. belegter Heap-Speicher	12625 KB	5473 KB	1770 KB	4000 KB
Max. belegter Non-Heap-Speicher	9057 KB	8772 KB	8038 KB	9211 KB
Max. belegter Speicher	21682 KB	14246 KB	9809 KB	13211 KB
Faktor Größe des belegten Speichers zu Dokumentgröße	12,26	13,02	9667,10	28,28
Heap-Speicher je DOM-Knoten	252 B	387 B	39410 B	206 B

Tab. 6–16 *Arbeitsspeicherbedarf beim Einfügen*

Bei der Analyse des Speicherbedarfs lässt sich feststellen, dass die Größe des Non-Heap-Speichers nur wenig von der Dokumentgröße abhängt. Die Dokumentgröße beeinflusst jedoch direkt die Größe des benötigten Heap-Speichers. Eine Sonderstellung nimmt das Dokument `purchaseOrder.xml` ein, da es im Vergleich zu den anderen Testdokumenten sehr klein ist und aus nur 46 DOM-Knoten besteht.

Wie schon bei der Untersuchung des Zeitbedarfs soll auch hier überprüft werden, welche Auswirkungen die Nutzung bzw. Nichtnutzung einer DTD hat. Dazu sind in Tabelle 6–17 die Messergebnisse für die Dokumente `mondial.xml` und `bibel.xml` mit und ohne DTD-Einbeziehung einander gegenübergestellt.

	mondial.xml mit DTD	mondial.xml ohne DTD	bibel.xml mit DTD	bibel.xml ohne DTD
Anzahl DOM-Knoten	51397	88385	14494	22140
Dokumentgröße	1768 KB	1768 KB	1094 KB	1094 KB
Max. belegter Heap-Speicher	12625 KB	16699 KB	5473 KB	6191 KB
Max. belegter Non-Heap-Speicher	9057 KB	8811 KB	8772 KB	8610 KB
Max. belegter Speicher	21682 KB	25510 KB	14246 KB	14801 KB
Faktor Größe des belegten Speichers zu Dokumentgröße	12,26	14,43	13,02	13,52
Heap-Speicher je DOM-Knoten	252 B	193 B	387 B	286 B

Tab. 6–17 *Vergleich des Arbeitsspeicherbedarfs beim Einfügen mit und ohne DTD-Nutzung*

Wenn keine DTD genutzt wird, müssen aufgrund der nicht möglichen Erkennung von ignorierbaren Whitespaces deutlich mehr DOM-Knoten verarbeitet werden. Dies führt zu einem höheren Speicherbedarf beim Einfügen. Der Anstieg verläuft aufgrund von Skalierungseffekten jedoch nicht linear, wie man insbesondere an der Größe des belegten Heap-Speichers je DOM-Knoten feststellt.

Der weitaus größte Teil des Speichers wird kurz nach dem Start des Einfügevorgangs belegt. Hauptgrund ist das Parsen des kompletten XML-Dokuments zu Beginn und die damit verbundene Erzeugung des DOM-Baums im Arbeitsspeicher. Während der sukzessiven Verarbeitung der einzelnen DOM-Knoten steigt der Speicherbedarf nur noch wenig an.

Untersuchung des Zusammenhangs von DOM-Knotenanzahl und Arbeitsspeicherbedarf anhand der purchaseOrder-Testdokumentreihe

Um den Zusammenhang zwischen der Anzahl der DOM-Knoten und dem benötigten Arbeitsspeicher beim Einfügen genauer untersuchen zu können, wurde – wie bereits in Abschnitt 6.3 für den Zeitbedarf – auf Basis des Dokuments `purchaseOrder.xml` eine Messreihe durchgeführt. Die einzelnen Testdokumente sind wiederum durch schrittweise Verdopplung des XML-Fragments unter dem Wurzelelement entstanden. Die Ergebnisse sind in den Abbildungen 6–10 und 6–11 wiedergegeben. Sie bestätigen die bereits genannten Erkenntnisse: Der Non-Heap-Speicherbedarf ist nahezu unabhängig von der Größe des einzufügenden Dokuments, wohingegen der Heap-Speicherbedarf mit zunehmender DOM-Knotenanzahl von dieser fast linear abhängig ist. In Abbildung 6–11 erkennt man dies daran, dass der belegte Heap-Speicher je DOM-Knoten ab etwa 20000 Knoten weitestgehend konstant bleibt.

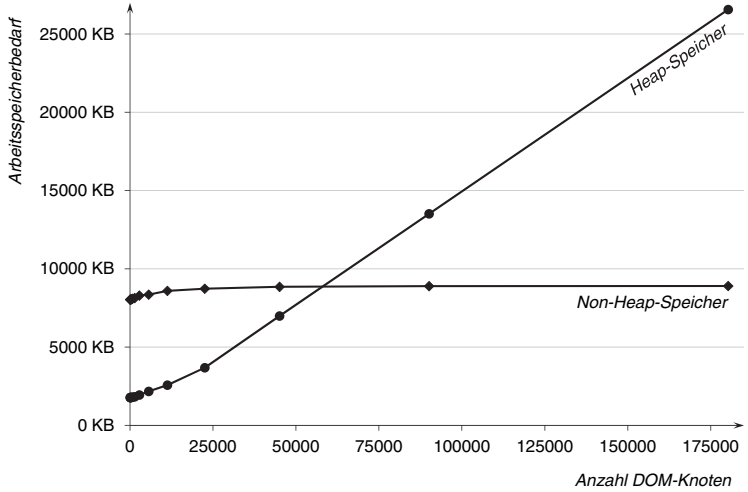


Abb. 6–10 *Heap- und Non-Heap-Speicherbedarf beim Einfügen*

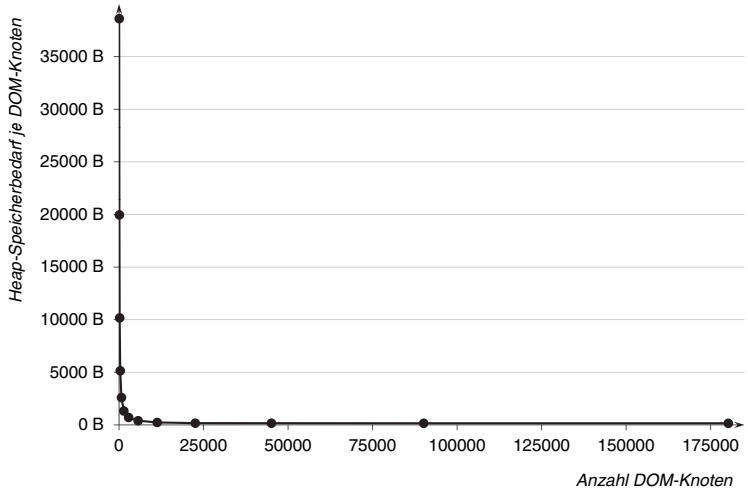


Abb. 6–11 *Heap-Speicherbedarf je DOM-Knoten beim Einfügen*

6.4.1.2 Auslesen

Für den Auslesevorgang wurden entsprechende Arbeitsspeichermessungen durchgeführt. Die Ergebnisse finden sich in den Tabellen 6–18 und 6–19. Sie ähneln stark den beim Einfügen ermittelten Werten. Es zeigt sich auch hier, dass die Größe des belegten Non-Heap-Speichers kaum von der Dokumentgröße beeinflusst wird. Dagegen hängt der benötigte Heap-Speicher von der Größe des Dokuments und der Anzahl der enthaltenen DOM-Knoten ab. Der Heap-

Speicherbedarf je DOM-Knoten schwankt um etwa 300 Byte.[53] Im Gegensatz zum Einfügevorgang steigt hier der Speicherbedarf im Laufe der Verarbeitung sukzessive mit der Erzeugung der einzelnen Knoten des DOM-Baums an.

	mondial.xml mit DTD	bibel.xml mit DTD	purchaseOrder.xml mit DTD	SigmodRecord.xml mit DTD
Anzahl DOM-Knoten	51397	14494	46	19912
Dokumentgröße	1768 KB	1094 KB	1 KB	467 KB
Max. belegter Heap-Speicher	12635 KB	5291 KB	1648 KB	3747 KB
Max. belegter Non-Heap-Speicher	8429 KB	8311 KB	7538 KB	8239 KB
Max. belegter Speicher	21063 KB	13602 KB	9187 KB	11986 KB
Faktor Größe des belegten Speichers zu Dokumentgröße	11,91	12,43	9053,93	25,66
Heap-Speicher je DOM-Knoten	252 B	374 B	36695 B	193 B

Tab. 6–18 *Arbeitsspeicherbedarf beim Auslesen*

In Tabelle 6–19 erkennt man die Auswirkungen des Einsatzes einer DTD auf den Arbeitsspeicherbedarf beim Auslesen. Es ist auch hier festzustellen, dass bei Nichtnutzung einer DTD zwar ein höherer Arbeitsspeicherbedarf vorhanden ist, jedoch die Größe des benötigten Heap-Speichers je DOM-Knoten aufgrund von Skalierungseffekten sinkt.

	mondial.xml mit DTD	mondial.xml ohne DTD	bibel.xml mit DTD	bibel.xml ohne DTD
Anzahl DOM-Knoten	51397	88385	14494	22140
Dokumentgröße	1768 KB	1768 KB	1094 KB	1094 KB
Max. belegter Heap-Speicher	12635 KB	16300 KB	5291 KB	5933 KB
Max. belegter Non-Heap-Speicher	8429 KB	8410 KB	8311 KB	8285 KB
Max. belegter Speicher	21063 KB	24710 KB	13602 KB	14218 KB
Faktor Größe des belegten Speichers zu Dokumentgröße	11,91	13,98	12,43	12,99
Heap-Speicher je DOM-Knoten	252 B	189 B	374 B	274 B

Tab. 6–19 *Vergleich des Arbeitsspeicherbedarfs beim Auslesen mit und ohne DTD-Nutzung*

Untersuchung des Zusammenhangs von DOM-Knotenanzahl und Arbeitsspeicherbedarf anhand der purchaseOrder-Testdokumentreihe

Auch beim Auslesevorgang wurde der Arbeitsspeicherbedarf mit Hilfe der auf der Datei `purchaseOrder.xml` basierenden Testdokumentreihe analysiert. Die gewonnenen Ergebnisse bestätigen die oben genannten Zusammenhänge. Auf die

53. Auch hier stellt wiederum das sehr kleine Dokument `purchaseOrder.xml` eine Ausnahme dar.

Wiedergabe der entsprechenden Graphen wird hier verzichtet, da sie den in Abschnitt 6.4.1.1 bei der Untersuchung des Arbeitsspeicherbedarfs beim Einfügen gezeigten Abbildungen 6–10 und 6–11 sehr ähnlich sind. Tendenziell ist der Speicherbedarf etwas niedriger als beim Einfügen: Es werden bis zu 550 KB weniger Heap-Speicher und zwischen 300 und 650 KB weniger Non-Heap-Speicher belegt.

6.4.2 Datenbankspeicherbedarf

Werden XML-Dokumente nach dem beschriebenen Abbildungsverfahren in der Informix-Datenbank abgelegt, ist von Interesse, wie sich der dazu nötige Datenbankspeicherbedarf im Vergleich zur Originalgröße des XML-Dokuments verhält. Um diese Frage zu beantworten, wurden die aus den vorangegangenen Abschnitten bekannten Testdokumente verwendet, um den jeweils belegten Datenbankspeicher zu untersuchen. In den Tabellen 6–20 und 6–21 sind die dabei ermittelten Kennzahlen dargestellt.

	mondial.xml mit DTD	bibel.xml mit DTD	purchaseOrder.xml mit DTD	SigmodRecord.xml mit DTD
Seitengröße	4 KB	4 KB	4 KB	4 KB
Belegte Seiten für Daten	3067	1090	4	981
– darin unbenutzte Bytes	5714 KB	1973 KB	7 KB	1861 KB
Belegte Seiten für Index	163	41	2	63
Effektiver Gesamtspeicherbedarf	7206 KB	2551 KB	17 KB	2315 KB
Faktor Datenbank- zu Dateigröße	4,08	2,33	16,42	4,95

Tab. 6–20 *Datenbankspeicherbedarf ohne Kompression und mit DTD*

	mondial.xml ohne DTD	bibel.xml ohne DTD
Seitengröße	4 KB	4 KB
Belegte Seiten für Daten	5689	1606
– darin unbenutzte Bytes	10640 KB	2925 KB
Belegte Seiten für Index	358	81
Effektiver Gesamtspeicherbedarf	13548 KB	3823 KB
Faktor Datenbank- zu Dateigröße	7,66	3,49

Tab. 6–21 *Datenbankspeicherbedarf ohne Kompression und ohne DTD*

Die Seitengröße von Informix beträgt unter Windows 4 KB und unter den meisten UNIX-Systemen 2 KB. Eine Änderung der Seitengröße ist nicht möglich. Die belegten Seiten unterteilen sich in Seiten, die für die Speicherung der Tabellenzeilen benötigt werden, und solchen, die der Aufnahme des Index dienen. Der effek-

tive Gesamtspeicherbedarf errechnet sich aus der Anzahl der belegten Seiten abzüglich der darin enthaltenen unbenutzten Bytes, die hier durch Verschnitt an den Seitengrenzen entstehen. In der letzten Zeile der Tabellen wird jeweils der Faktor angegeben, der das Verhältnis des Datenbankspeicherbedarfs zur ursprünglichen XML-Dokumentgröße beschreibt.

Wie man anhand dieser Faktoren feststellt, ist der Speicherbedarf in der Datenbank mindestens etwa doppelt so groß wie im XML-Format. Je nach Dokument kann die Speicherbedarfsvergrößerung auch darüber liegen, bleibt jedoch grundsätzlich in einem vertretbaren Rahmen. Der hohe Faktor beim Dokument `purchaseOrder.xml` ist auch hier mit den fehlenden Skalierungseffekten aufgrund der Kürze des Dokuments zu erklären.

Man erkennt, dass das einfache XML-Format somit trotz aller Ausführlichkeit – zum Beispiel enthalten Start- und End-Tag jeweils fast den gleichen Text – auch gewisse Vorteile in Bezug auf den Speicherbedarf bietet. Grund ist, dass Verwaltungsangaben, die zum Beispiel Belegungs-, Typ- und Längeninformationen beinhalten, nicht erforderlich sind. Natürlich entsteht dadurch auch der Nachteil, dass auf einzelne Teile einer XML-Datei nur nach Parsen des Dokuments zugegriffen werden kann.

Informix bietet die Möglichkeit, Tabellen komprimiert abzuspeichern. Der Kompressionsvorgang führt dazu, dass sich der Zeitbedarf der Datenbankoperationen leicht verändert. Eine Untersuchung auf Basis der Testdokumente ergab, dass der Zeitbedarf der DB-Einfügeoperationen durch die Kompression um durchschnittlich 3,6 % sinkt. Beim Auslesen waren nur beim Dokument `purchaseOrder.xml` Auswirkungen messbar. Hier stieg die Ausführungszeit der DB-Ausleseoperationen um fast 8 % an. Bei allen anderen Dokumenten blieb sie nahezu unverändert.

Im Folgenden soll das von Informix genutzte Kompressionsverfahren, das an den Ziv-Lempel-Algorithmus erinnert, kurz erläutert werden. Die in der zu komprimierenden Tabelle enthaltenen Zeilen werden zunächst unabhängig von den Spaltenbegrenzungen betrachtet, um darin sich wiederholende Muster zu suchen. Anschließend wird ein Komprimierungswörterverzeichnis aufgebaut, das den Mustern Symbole zuordnet. Statt der ursprünglichen Tabellenzeilen werden diese Symbole und natürlich das Kompressionsverzeichnis gespeichert.[54]

Das Komprimierungswörterverzeichnis wird auf Grundlage einer Stichprobe von mindestens 2000 Tabellenzeilen erstellt und kann maximal 3840 Muster („Wörter“) mit einer Länge zwischen 2 und 15 Byte enthalten. Jedes dieser Muster wird durch ein 12 Bit langes Symbol repräsentiert. Die Verzeichnisse werden

54. Eine ausführlichere Beschreibung des Kompressionsverfahrens findet sich in [IBM11b] sowie auf der folgenden Web-Seite von IBM: `https://www.ibm.com/developerworks/data/library/techarticle/dm-0904idsoptimization/`

in einer separaten Systemtabelle abgelegt, laut Informix-Benutzerhandbuch ist typischerweise von einer Größe von etwa 100 KB auszugehen.

Um festzustellen, wie gut die Kompression in Zusammenhang mit dem entwickelten Abbildungsverfahren für XML-Dokumente arbeitet, wurden die Untersuchungen zum Datenbankspeicherbedarf erneut mit einer komprimierten `ELEMENTS`-Tabelle durchgeführt. Die Ergebnisse sind in den Tabellen 6–22 und 6–23 zusammengefasst. Das Dokument `purchaseOrder.xml` kann nicht komprimiert werden, da die Tabelle hierbei nur 27 Zeilen enthält und für die Komprimierung mindestens 2000 Zeilen benötigt werden.

In den Tabellen 6–22 und 6–23 wurde eine Zeile ergänzt, die die jeweilige Größe des Komprimierungswörterverzeichnisses angibt. Dabei ist zu beachten, dass die Größe dieses Verzeichnisses nicht nur von dem abgelegten XML-Dokument abhängig ist, sondern auch von den zufällig ausgewählten Tabellenzeilen, auf deren Basis die zu komprimierenden Muster ausgewählt werden. Dies kann als Folge auch dazu führen, dass die Anzahl der belegten Datenseiten von Einfügevorgang zu Einfügevorgang schwankt. Erfahrungsgemäß bleibt dieser Wert jedoch nahezu konstant und weicht maximal um eine Seite ab.

	mondial.xml mit DTD	bibel.xml mit DTD	purchaseOrder.xml mit DTD	SigmodRecord.xml mit DTD
Seitengröße	4 KB	4 KB	Komprimierung nicht möglich	4 KB
Belegte Seiten für Daten	872	338		234
– darin unbenutzte Bytes	1686 KB	644 KB		459 KB
Belegte Seiten für Index	163	41		63
Größe des Kompr.wörterverz.	72 KB	82 KB		63 KB
Effektiver Gesamtspeicherbedarf	2526 KB	954 KB		793 KB
Faktor Datenbank- zu Dateigröße	1,43	0,87		1,70

Tab. 6–22 *Datenbankspeicherbedarf mit Kompression und mit DTD*

	mondial.xml ohne DTD	bibel.xml ohne DTD
Seitengröße	4 KB	4 KB
Belegte Seiten für Daten	1569	492
– darin unbenutzte Bytes	3031 KB	933 KB
Belegte Seiten für Index	358	81
Größe des Kompr.wörterverz.	69 KB	77 KB
Effektiver Gesamtspeicherbedarf	4745 KB	1436 KB
Faktor Datenbank- zu Dateigröße	2,68	1,31

Tab. 6–23 *Datenbankspeicherbedarf mit Kompression und ohne DTD*

Wie man anhand der Faktoren in der jeweiligen letzten Zeile der beiden Tabellen erkennt, hat sich der Datenbankspeicherbedarf durch die Verwendung einer komprimierten Tabelle deutlich reduziert. Beim Dokument `mondial.xml` verringert sich bei Nutzung einer DTD der Datenbankspeicherbedarf durch die Kompression von etwa 7,0 MB auf 2,5 MB. Das heißt, der Bedarf entspricht nun nicht mehr der 4,1-fachen, sondern nur noch der 1,4-fachen XML-Dateigröße. Entsprechendes gilt bei Nichtnutzung einer DTD: Hier verringert sich der Datenbankspeicherbedarf aufgrund der Kompression von 13,2 MB auf 4,6 MB, was der 2,7-fachen statt bisher der 7,7-fachen XML-Dateigröße entspricht.

Zusammenfassend ist für die Testdokumente festzustellen, dass der Speicherbedarf durch die Kompression um etwa zwei Drittel kleiner wird. Hier zeigt sich, dass sich die Inhalte von XML-Dateien gut komprimieren lassen, da sie meist Texte und nur sehr selten binäre Daten enthalten. Der Speicherbedarf in der Datenbank ist damit in den meisten Fällen nur wenig größer oder sogar geringfügig kleiner als die zugehörige unkomprimierte XML-Textdatei.

Untersuchung des Zusammenhangs von Dateigröße und Datenbankspeicherbedarf anhand der purchaseOrder-Testdokumentreihe

	Dateigröße	ohne Kompression		mit Kompression	
		Datenbank-speicherbedarf	Faktor Datenbank-zu Dateigröße	Datenbank-speicherbedarf	Faktor Datenbank-zu Dateigröße
1	1 KB	17 KB	17,32	Komprimierung nicht möglich	
2	2 KB	21 KB	11,43		
3	4 KB	29 KB	8,22		
4	7 KB	54 KB	7,68		
5	14 KB	91 KB	6,54		
6	28 KB	165 KB	5,97		
7	55 KB	313 KB	5,67		
8	110 KB	609 KB	5,53	285 KB	2,58
9	220 KB	1206 KB	5,47	464 KB	2,11
10	441 KB	2396 KB	5,44	812 KB	1,84
11	881 KB	4779 KB	5,42	1572 KB	1,78
12	1762 KB	9549 KB	5,42	2968 KB	1,68
13	3524 KB	19089 KB	5,42	5875 KB	1,67
14	7048 KB	38161 KB	5,41	11596 KB	1,65
15	14096 KB	76309 KB	5,41	23139 KB	1,64
16	28192 KB	152606 KB	5,41	46938 KB	1,66

Tab. 6–24 *Datenbankspeicherbedarf bei verschiedenen Dateigrößen*

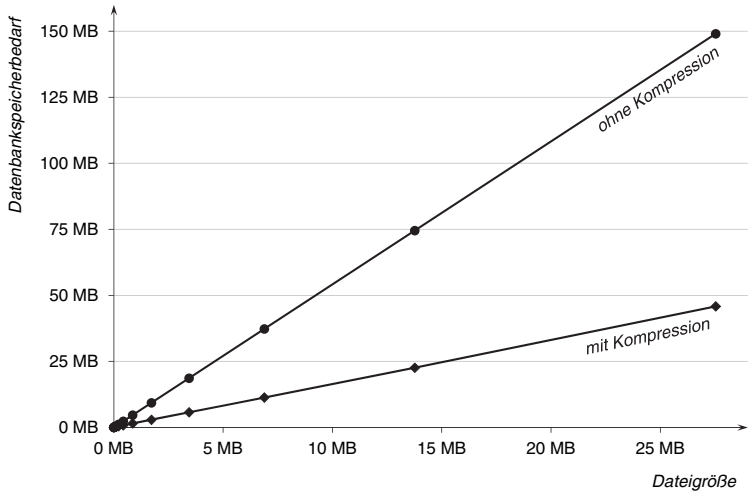


Abb. 6–12 *Datenbankspeicherbedarf in Abhängigkeit der Dateigröße*

Zum Abschluss der Leistungsuntersuchungen wurde die auf dem Dokument `purchaseOrder.xml` basierende Testdokumentreihe verwendet, um auch den Zusammenhang von XML-Dateigröße und benötigtem Datenbankspeicher auf diese Weise genauer zu untersuchen. Die erhaltenen Ergebnisse sind in der oben gezeigten Tabelle 6–24 aufgelistet. Es wurden sowohl Messungen mit als auch ohne Datenbankkompression durchgeführt. Dabei ist zu beachten, dass Informix nur Tabellen komprimieren kann, die aus mindestens 2000 Zeilen bestehen. Aus diesem Grund enthält ein Teil der Tabelle keine Werte.

Interessant sind die in der Tabelle angegebenen Faktoren, die das Verhältnis von Datenbankspeichergröße zu Dateigröße zeigen. Mit zunehmendem Umfang des gespeicherten XML-Dokuments verringern sich die Faktoren und nähern sich hier den Werten 5,4 bzw. 1,6 an, sodass die in Abbildung 6–12 gezeigten Graphen einen fast linearen Verlauf aufweisen.[55]

6.5 Oracle-Vergleichsuntersuchungen

Um die in den vorangegangenen Abschnitten vorgestellten Messergebnisse besser einordnen zu können, wurden Vergleichsmessungen auf Basis der Oracle-Datenbank durchgeführt.

Oracle unterstützt verschiedene eingebaute Speicherverfahren für XML-Dokumente. Diese tragen die Bezeichnungen *Structured (Object-Relational) Sto-*

55. Hierbei muss beachtet werden, dass sich auch die XML-Textdateien gut komprimieren lassen. Bei der purchaseOrder-Testdokumentreihe erreicht man bei einer Standard-ZIP-Komprimierung eine Reduzierung der Dokumentgröße auf zwei Drittel bis ein Zweihundertstel der Ausgangsgröße.

rage, *Unstructured (CLOB) Storage* sowie *Binary XML Storage*.[56] Hier wird die strukturierte, objektrelationale Speicherung genutzt, da diese dem implementierten Verfahren `XML2DB` am ähnlichsten ist. Dabei werden die einzelnen Elemente in mehreren ineinander geschachtelten Tabellen abgelegt. Das Speicherverfahren ist damit mit den in Abschnitt 4.2.2 beschriebenen NF^2-Tabellen vergleichbar. Wie bei `XML2DB` und auch beim Schreddern (siehe Abschnitt 4.2.1) wird das XML-Dokument in einzelne Tabelleneinträge aufgespalten. Oracle nutzt dabei Zusatzangaben, um DOM-Fidelity zu gewährleisten. Die geschachtelten Tabellen („nested tables“) werden intern als separate Tabellen abgelegt, die jedoch nicht direkt, sondern nur durch Zugriff auf die übergeordnete Tabelle abgefragt werden können. Die Syntax der dazu nötigen SQL-Anweisungen ist relativ aufwendig und nicht sehr intuitiv.

Für die Messungen wurde die Oracle-Datenbank in der Version 11g Release 1 eingesetzt. Die dabei genutzte Hardware ist etwas leistungsschwächer als die zuvor für die Informix-Messungen verwendete. Es handelt sich um einen Server mit Pentium-4-CPU, einer Taktfrequenz von 3,2 GHz, 2 GB Arbeits- und 149 GB Festplattenspeicher. Als Betriebssystem kommt Microsoft Windows Server 2003 R2 (32 Bit) zum Einsatz.

Die Vergleichsuntersuchungen wurden auf Basis des Dokuments `mondial.xml` durchgeführt. Um dieses XML-Dokument objektrelational in Oracle ablegen zu können, muss zunächst die Dokumentstruktur bei der Datenbank registriert werden. Da Oracle hierbei keine DTD-Dokumente unterstützt, wurde die bisher genutzte DTD in ein Schemadokument umgeschrieben. Die Registrierung erfolgt durch ein PL/SQL-Skript, das die zugehörige Methode des `DBMS_XMLSCHEMA`-Pakets aufruft. Durch die Registrierung werden in der Datenbank passende Typen für die verschiedenen Elementdefinitionen des XML-Schemas angelegt. Weiterhin wird eine passende Tabelle vom Typ `XMLType` erzeugt, die geschachtelte Untertabellen enthält, um die durch das XML-Schema vorgegebene hierarchische Struktur abzubilden. Für das Mondial-Dokument sind dazu 34 Untertabellen erforderlich.

Durch ein passendes `INSERT`-Kommando kann das Dokument `mondial.xml` anschließend in der Datenbank abgelegt werden. Dabei wurde das Oracle-Tool SQL-Developer in der Version 3.1 eingesetzt.[57] Mit diesem können SQL-Anweisungen und PL/SQL-Skripte auf der Oracle-Datenbank ausgeführt werden. Die Funktion `BFILENAME` ermöglicht dem `INSERT`-Kommando den Zugriff auf das Dateisystem des Servers, auf dem die in der Datenbank zu speichernde XML-

56. Details zu den Varianten und eine Gegenüberstellung der Vor- und Nachteile finden sich in der Oracle-Dokumentation [Ora11b, XML DB Developer's Guide, Seite 1-17 ff.].
57. Beschreibung des Programms und Downloadmöglichkeit unter: `http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html`

Datei abgelegt ist. Das gespeicherte XML-Dokument kann in der Datenbank über eine Objektreferenz angesprochen und selektiert werden.

Am Dokument `mondial.xml` mussten zwei Änderungen durchgeführt werden, damit ein problemloses Einfügen in die Datenbank möglich war: Zum einen wurde die Dokumenttypangabe entfernt, da die Validierung des XML-Dokuments ohne plausiblen Grund abgebrochen wurde. Ursache ist wahrscheinlich ein Programmierfehler im zugehörigen Oracle-Modul, das für die XML-Gültigkeitsprüfung zuständig ist. Dies stellt jedoch keine schwerwiegende Einschränkung dar, weil durch die Registrierung des XML-Schemas nur Dokumente abgelegt werden können, die einen zu den erzeugten Datenbankstrukturen passenden Aufbau besitzen. Als zweite Änderung am Dokument `mondial.xml` musste ein Attributwert gekürzt werden, da dieser mit 2316 Zeichen die experimentell ermittelte Obergrenze von 2000 Zeichen überschritten hat.

Zum Auslesen des XML-Dokuments genügt ein `SELECT`-Kommando, das auf die XML-wertige Datenbanktabelle zugreift. Durch die Methode `getClobVal` erhält man das wieder serialisierte XML-Dokument als `CLOB`-Wert. Um diesen in einer Datei abzuspeichern, kann die Funktion `CLOB2FILE` aus dem Paket `DBMS_XSLPROCESSOR` verwendet werden. Diese Funktion muss dazu nur einmal aufgerufen werden und arbeitet deutlich schneller als andere Lösungsansätze, bei denen die XML-Datei blockweise geschrieben wird.

In Tabelle 6–25 sind die Ausführungsgeschwindigkeiten für das Einfügen, Auslesen und Löschen des Dokuments `mondial.xml` angegeben. Die genannten Zeiten sind Mittelwerte auf Grundlage zehnmaliger Durchführung der jeweiligen Aktion.

Schritt	Ausführungsdauer
XML-Dokument einfügen	3,563 s
XML-Dokument auslesen	5,973 s
XML-Dokument löschen	1,947 s

Tab. 6–25 *Oracle-Vergleichswerte für das Mondial-Dokument*

Das Löschen und Einfügen des Dokuments `mondial.xml` dauert somit auf der Oracle-Datenbank insgesamt etwa 5,5 s, das Auslesen etwa 6,0 s. Das Verfahren `XML2DB` benötigt auf der Informix-Datenbank hierfür 18,3 s bzw. 26,7 s, ist also um den Faktor 3,3 bzw. 4,5 langsamer. Ein Grund hierfür ist die Zugriffsart auf die Datenbank: `XML2DB` läuft außerhalb der eigentlichen Datenbank und verwendet eine JDBC-Verbindung, um SQL-Kommandos ausführen zu können. Bei dem von Oracle genutzten Verfahren erfolgt jedoch ein viel direkterer Zugriff, da das Einfügen und Auslesen durch ein in die Datenbank eingebautes Modul durchgeführt wird. Ein derartiges Vorgehen ist nur dem Hersteller der Datenbank möglich, da nur dieser die dafür nötigen Änderungen am Datenbankkern selbst vornehmen kann.

7 Zusammenfassung und Ausblick

7.1 Bewertung

In dieser Arbeit wurde ein Verfahren entwickelt, mit dem XML-Dokumente in relationalen Datenbanken unter Verwendung SQL:2003-konformer Datentypen gespeichert und wieder ausgelesen werden können. Mit Hilfe einer prototypischen Applikation wurde das Vorgehen demonstriert und seine prinzipielle Einsatzfähigkeit gezeigt.

Beim Entwurf des Abbildungsverfahrens wurde besondere Aufmerksamkeit auf die Einsatzmöglichkeit in Verbindung mit einem existierenden, ausgereiften relationalen Datenbankmanagementsystem gelegt. Dazu musste untersucht werden, inwieweit die verschiedenen Systeme den SQL:2003-Standard unterstützen. Die Wahl fiel auf das Informix-DBMS von IBM, das für die gestellte Aufgabe am geeignetsten erschien und auf das die Abbildungssystematik angepasst wurde.

Durch den Einsatz von kollektionswertigen Datentypen konnte erreicht werden, dass die hierarchische Struktur von XML-Dokumenten erhalten bleibt und keine Verteilung der Daten auf mehrere Tabellen nötig ist. Das bedeutet, dass kein Schreddern und keine vollständige Normalisierung der Strukturen erforderlich ist. Das Abbildungsverfahren ist generisch und lässt sich unabhängig vom Vorliegen einer Dokumentenbeschreibung (DTD oder XML-Schema) einsetzen. Es müssen keine an den jeweiligen Dokumentaufbau angepassten Datenbankstrukturen erzeugt werden. Dies hat den Vorteil, dass auch verschiedene Dokumente problemlos nebeneinander abgelegt werden können.

Um die Leistungsfähigkeit des Verfahrens beurteilen zu können, wurde nach Abschluss der Implementierung eine Reihe von Untersuchungen des Speicher- und Zeitbedarfs vorgenommen. Die dabei erhaltenen Ergebnisse werden ausführlich in Kapitel 6 dargestellt. Zusammenfassend lässt sich sagen, dass das Abbildungsverfahren nicht überragend schnell und speicherarm arbeitet. Mögliche Ursachen sind die hohe Anzahl an SQL-Kommandos, die je XML-Dokument ausgeführt werden, sowie eine vermutete Ineffizienz bei der Verarbeitung und Speicherung objektrelationaler Typen. Jedoch bleiben der Speicher- und Zeitbe-

darf beherrschbar. Durch Anpassungen der Datenbankmanagementsysteme und der Hardware sind hier sicherlich noch Steigerungen, insbesondere bei der Ausführungsgeschwindigkeit, möglich.

Das Verfahren kann grundsätzlich nicht mit Spezialdatenbanken, die häufig auch die Bezeichnung NoSQL-Datenbanken („Not only SQL“) tragen, verglichen werden. Diese brechen mit der Tradition der herkömmlichen relationalen Datenbanken, indem sie kein Tabellenschema aufweisen und darauf ausgelegt sind, ganze Dokumente oder Schlüssel-Wert-Paare möglichst rasch zu speichern und wieder auszulesen. Um dies mit sehr hoher Performanz erledigen zu können, wird die Datenhaltung meist redundant verteilt, und es existieren keine oder nur schwache Konsistenzgarantien („eventual consistency“). Joins werden im Allgemeinen vermieden, die Daten sind nicht normalisiert. ACID-Eigenschaften werden nicht angeboten oder nur teilweise durch eine zusätzliche Middleware zur Verfügung gestellt.

Somit werden bei diesen Spezialdatenbanken zur Leistungssteigerung eine Reihe an Nachteilen in Kauf genommen, die ihren Einsatz für geschäftskritische Daten höchst fragwürdig erscheinen lassen. Ein Verzicht auf die sichere, isolierte Ausführung paralleler Transaktionen ist häufig nicht möglich. Es muss garantiert sein, dass einmal geschriebene und bestätigte Datenänderungen auch sofort bei allen neu begonnenen Transaktionen sichtbar sind. Anderenfalls kommt es zu Inkonsistenzen, die bei transaktionsorientierten Anwendungen nicht tolerierbar sind. NoSQL-Datenbanken eignen sich daher für Einsatzzwecke, in denen es mehr auf die Geschwindigkeit als auf die sofortige, konsistente Sichtbarkeit von Änderungen ankommt. Sie eignen sich daher gut für Social-Web-Applikationen, in denen es kein Problem ist, wenn die ein oder andere Meldung oder Statusangabe erst nach und nach an alle verteilten Datenbankserver und die mit ihnen verbundenen Clients propagiert wird.

7.2 Mögliche Erweiterungen

Zum Schluss der Arbeit sollen an dieser Stelle einige denkbare Erweiterungen des Abbildungsverfahrens vorgestellt werden. In Abschnitt 5.4 wurden bereits kleinere mögliche Weiterentwicklungen erwähnt. Dazu zählen der Erhalt der Entitäten und der zugehörigen Referenzen, die vollständige Unterstützung für Namensräume und Namensraumpräfixe sowie die Speicherung zusätzlicher dokumentexterner Daten, zum Beispiel Erstellungsdatum und Autor.

Daneben sind Fortschritte im Zusammenhang mit DTD- und XML-Schema-Dokumenten wünschenswert. Momentan werden die Angaben, die für die Einbindung eines dieser Beschreibungsdokumente zuständig sind, zwar abgespeichert und beim Auslesen wiederhergestellt, allerdings wird die DTD bzw. das XML-Schema selbst nicht in der Datenbank abgelegt. Für XML-Schema-Dokumente ist eine einfache Lösung denkbar: Das Schema kann, da es selbst ein XML-

Dokument darstellt, problemlos mit dem bisherigen Abbildungsverfahren in der Datenbank gespeichert werden. Es ist nur eine Erweiterung nötig, um die Verknüpfung zwischen Dokument und Schema zu ermöglichen und die mehrfache Speicherung des gleichen Schemas zu verhindern. Für eine interne oder externe DTD ist etwas mehr Aufwand erforderlich, da diese nicht dem XML-Format genügen. Entweder kann die DTD komplett als Text abgelegt werden oder aber das Dokument wird in die einzelnen Definitionen aufgebrochen, sodass diese zeilenweise in der Tabelle `ELEMENTS` gespeichert werden können. Die Erkennung dieser speziellen Tupel kann über die `NAME`-Spalte erfolgen. Alternativ ist auch die kanonische Umsetzung einer DTD in ein XML-Schema möglich, das dann als XML-Dokument mit dem Abbildungsverfahren in der Datenbank abgelegt wird.[58]

Für den praktischen Einsatz des Abbildungsverfahrens wäre es von Vorteil, wenn gespeicherte Prozeduren in der Datenbank zur Verfügung stünden, die Änderungen an gespeicherten Dokumenten in der Datenbank erlauben, ohne diese zunächst vollständig auszulesen und neu speichern zu müssen. Prinzipiell können derartige Änderungen auch direkt durch SQL-Kommandos durchgeführt werden. Hier besteht allerdings die Gefahr, dass die resultierenden Strukturen inkonsistent sind. Dies lässt sich durch den Einsatz von gespeicherten Prozeduren oder durch die Verwendung eines zwischengeschalteten Applikationsservers, der die eigentlichen Änderungen durchführt, verhindern. Weiterhin ist der Einsatz von Datenbanktriggern denkbar, die bei Änderungen, die mittels SQL direkt ausgeführt werden, auslösen, und prüfen, ob diese Inkonsistenzen verursachen und dann gegebenenfalls die Änderungsoperation abbrechen können.

In diesem Zusammenhang ist allerdings noch zu untersuchen, inwiefern sich diese Anforderung mit Hilfe von Triggern lösen lässt, da die möglichen Operationen, die durch die Kommandos eines Triggers durchgeführt werden können, bei den meisten DBMS einer Reihe von Einschränkungen unterliegen. Bestehen durchzuführende Dokumentänderungen aus mehreren Teiloperationen, sollten diese in einer Transaktion zusammengefasst werden, um garantieren zu können, dass alle Teile vollständig oder – bei einem Abbruch im Fehlerfall – gar nicht ausgeführt werden.

Im Mehrbenutzerbetrieb muss zusätzlich sichergestellt werden, dass die nötigen Sperren auf den zu ändernden Dokumenten gesetzt werden. Dabei sollte möglichst restriktiv mit den Sperren umgegangen werden, das heißt, nach Möglichkeit nur Schreib- statt Lesesperren und eine geringe Granularität verwendet werden, um die Nebenläufigkeit nicht unnötig einzuschränken. Zu dem Themenkomplex Transaktionen auf XML-Dokumenten und deren gemeinsame Bearbei-

58. Die Umsetzung in ein XML-Schema kann zum Beispiel mit dem von Y. Koike auf den Web-Seiten des W3C zur Verfügung gestellten Tool `dtd2xsd` durchgeführt werden. Siehe: `http://www.w3.org/2000/04/schema_hack/`

tung existiert bereits eine Reihe an Arbeiten [GBS02, DH04, HH04, HKM04, Sik08]. Von Interesse ist, wie sich die dort vorgeschlagenen Algorithmen in Verbindung mit dem hier entwickelten Abbildungsverfahren einsetzen lassen.

Werden Änderungen an XML-Dokumenten in der Datenbank vorgenommen, tritt die Frage auf, ob das resultierende Dokument weiterhin einer bestimmten DTD bzw. XML-Schema genügt. Dazu könnte das Dokument vollständig geparst werden, um anschließend entscheiden zu können, ob es weiterhin gültig ist. Dieses Vorgehen erscheint aufwendig. Günstiger wäre es, bereits eine Aussage treffen zu können, wenn nur die Änderungsoperationen selbst betrachtet werden. In [BPV04] wird diese Art der inkrementellen Validierung vorgeschlagen, die im Rahmen einer Erweiterung des Abbildungsverfahrens berücksichtigt werden sollte.[59]

Um die Ausführungszeit des Abbildungsverfahrens zu senken, sollte getestet werden, ob eine Parallelisierung der Datenbankzugriffe sinnvoll ist. Dies bedeutet, dass beim Einfügen eines XML-Dokuments nach dem Parsen parallel durch mehrere Threads die Tupel in die Datenbanktabelle eingefügt werden. Dabei ist ein Geschwindigkeitszuwachs zu erwarten, da so die JDBC-Operationen und die Datenbankzugriffe nebenläufig ausgeführt werden können.

Ein weiterer Bereich, der in Zukunft bearbeitet werden sollte, sind Möglichkeiten der Umsetzung von XQuery-Ausdrücken in SQL-Kommandos, die auf den Strukturen des entwickelten Abbildungsverfahrens die gewünschten Ergebnisse liefern. Zunächst müssen dazu grundsätzliche Strategien der Transformation untersucht werden, sodass später eine automatische Umsetzung möglich ist. Dabei ist zu erwarten, dass weitere Optimierungspotenziale erkannt werden, mit denen die Abfragen durch zusätzliche Indexe, Hilfstabellen oder Strukturänderungen beschleunigt werden können.

59. Siehe hierzu auch die Ausführungen am Ende von Abschnitt 5.4.1.4.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

Alle hier im Literaturverzeichnis und an anderen Stellen in dieser Arbeit angeführten Internet-Links wurden am 3. April 2012 auf ihre Gültigkeit überprüft.

AB84 S. Abiteboul und N. Bidoit. *Non first normal form relations to represent hierarchically organized data*. Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS), Seiten 191-200, ACM, 1984.

ABS00 S. Abiteboul, P. Bunemann und D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

AC11 M. Atay und A. Chebotko. *Schema-Based XML-to-SQL Query Translation Using Interval Encoding*. Proceedings of the 8th International Conference on Information Technology: New Generations (ITNG), Seiten 84-89, IEEE, 2011.

ACL07 M. Atay, A. Chebotko, S. Lu und F. Fotouhi. *XML-to-SQL Query Mapping in the Presence of Multi-valued Schema Mappings and Recursive XML Schemas*. Proceedings of the 18th International Conference on Database and Expert Systems Applications (DEXA), Seiten 603-616, LNCS 4653, Springer, 2007.

AJK02 S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava und Y. Wu. *Structural Joins: A Primitive for Efficient XML Query Pattern Matching*. Proceedings of the 18th International Conference on Data Engineering (ICDE), Seiten 141-152, IEEE, 2002.

AL02 M. Arenas und L. Libkin. *A Normal Form for XML Documents*. Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), Seiten 85-96, ACM, 2002.

AMN11 A. Algergawy, M. Mesiti, R. Nayak und G. Saake. *XML Data Clustering: An Overview*. ACM Computing Surveys, Ausgabe 43 (2011) 4, Artikel 25, Oktober 2011.

Are06 M. Arenas. *Normalization Theory for XML*. ACM SIGMOD Record, Ausgabe 35 (2006) 4, Seiten 57-64, Dezember 2006.

AYU03 T. Amagasa, M. Yoshikawa und S. Uemura. *QRS: A Robust Numbering Scheme for XML Documents*. Proceedings of the 19th International Conference on Data Engineering (ICDE), Seiten 705-707, IEEE, 2003.

BCH06 K. Beyer, R. Cochrane, M. Hvizdos, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, R. Lyle, M. Nicola, F. Özcan, H. Pirahesh, N. Seemann, A. Singh, T. Truong, R. C. van der Linden, B. Vickery, C. Zhang und G. Zhang. *DB2 goes hybrid: Integrating native XML und XQuery with relational data and SQL*. IBM Systems Journal, Ausgabe 45 (2006) 2, Seiten 271-298, 2006.

BCJ05 K. Beyer, R. J. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, B. Lyle, F. Özcan, H. Pirahesh, N. Seemann, T. Truong, B. van der Linden und B. Vickery. *System RX: One Part Relational, One Part XML*. Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, Seiten 347-358, ACM, 2005.

BFM05 D. Barbosa, J. Freire und A. O. Mendelzon. *Designing Information-Preserving Mapping Schemes for XML*. Proceedings of the 31st International Conference on Very Large Data Bases (VLDB), Seiten 109-120, ACM, 2005.

BGK06 P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger und J. Teubner. *MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine*. Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, Seiten 479-490, ACM, 2006.

Bia07 B. Bialek. *Strukturwandel – Tabellen und XML-Objekte in derselben SQL-Datenbank*. Magazin für Computertechnik, Ausgabe 20/2007, Seiten 148-153, September 2007.

BKS02 N. Bruno, N. Koudas und D. Srivastava. *Holistic Twig Joins: Optimal XML Pattern Matching*. Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Seiten 310-321, ACM, 2002.

Bou05a R. P. Bourret. *Mapping DTDs to Databases*. September 2005. `http://www.rpbourret.com/xml/DTDToDatabase.htm`

Bou05b R. P. Bourret. *XML and Databases*. September 2005. http://www.rpbourret.com/xml/XMLAndDatabases.htm

Bou10 R. P. Bourret. *XML Database Products*. Juni 2010. http://www.rpbourret.com/xml/XMLDatabaseProds.htm

BPV04 A. Balmin, Y. Papakonstantinou und V. Vianu. *Incremental Validation of XML Documents*. ACM Transactions on Database Systems, Ausgabe 29 (2004) 4, Seiten 710-751, Dezember 2004.

BR03 T. Böhme und E. Rahm. *Benchmarking von XML-Datenbanksystemen*. In: E. Rahm und G. Vossen (Hrsg.). *Web & Datenbanken – Konzepte, Architekturen, Anwendungen*. Seiten 437-460, Dpunkt-Verlag, 2003.

CCS05 S. Chaudhuri, Z. Chen, K. Shim und Y. Wu. *Storing XML (with XSD) in SQL Databases: Interplay of Logical und Physical Designs*. IEEE Transactions on Knowledge and Data Engineering, Ausgabe 17 (2005) 12, Seiten 1595-1609, Dezember 2005.

Cho02 B. Choi. *What Are Real DTDs Like*. Proceedings of the 5th International Workshop on the Web and Databases (WebDB), Seiten 43-48, Informal Proceedings, 2002.

Cod70 E. F. Codd. *A Relational Model of Data for Large Shared Data Banks*. Communications of the ACM, Ausgabe 13 (1970) 6, Seiten 377-387, Juni 1970.

Cod79 E. F. Codd. *Extending the Database Relational Model to Capture more Meaning*. ACM Transactions on Database Systems, Ausgabe 4 (1979) 4, Seiten 397-434, Dezember 1979.

CRZ03 A. B. Chaudhri, A. Rashid und R. Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems*. Addison-Wesley, 2003.

Dav03 M. M. David. *ANSI SQL Hierarchical Processing Can Fully Integrate Native XML*. ACM SIGMOD Record, Ausgabe 32 (2003) 1, Seiten 41-46, März 2003.

DD00 C. J. Date und H. Darwen. *Foundation for Future Database Systems: The Third Manifesto*. Addison-Wesley, 2000.

DH04 S. Dekeyser und J. Hidders. *Conflict Scheduling of Transactions on XML Documents*. Proceedings of the 15th Australasian Database Conference (ADC), Seiten 93-101, CRPIT 27, Australian Computer Society, 2004.

Die82 P. F. Dietz. *Maintaining order in a linked list*. Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC), Seiten 122-127, ACM, 1982.

DKA86 P. Dadam, K. Küspert, F. Andersen, H. Blanken, R. Erbe, J. Günauer, V. Lum, P. Pistor und G. Walch. *A DBMS Prototype to Support Extended NF² Relations: An Integrated View on Flat Tables and Hierarchies*. Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, ACM SIGMOD Record, Ausgabe 15 (1986) 2, Seiten 356-367, Juni 1986.

Dop07 P. Dopichaj. *Space-Efficient Indexing of XML Documents for Content-Only Retrieval*. Datenbank-Spektrum, Ausgabe 7 (2007) 23, Seiten 21-28, November 2007.

EAZ08 I. Elghandour, A. Aboulnaga, D. C. Zilio, F. Chiang, A. Balmin, K. Beyer und C. Zuzarte. *An XML Index Advisor*. Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, Seiten 1267-1270, ACM, 2008.

Eck99 R. Eckstein. *XML Pocket Reference*. O'Reilly, 1999.

EM04 A. Eisenberg und J. Melton. *Advancements in SQL/XML*. ACM SIGMOD Record, Ausgabe 33 (2004) 3, Seiten 79-86, September 2004.

EM05 A. Eisenberg und J. Melton. *XQuery 1.0 is Nearing Completion*. ACM SIGMOD Record, Ausgabe 34 (2005) 4, Seiten 78-84, Dezember 2005.

Feu03 S. Feuerstein. *Taking Up Collections*. Oracle Magazine, Ausgabe September/Oktober 2003. `http://www.oracle.com/technetwork/issue-archive/o53plsql-083350.html`

FK99a D. Florescu und D. Kossmann. *Storing and Querying XML Data using an RDBMS*. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, Seiten 27-34, 1999.

FK99b D. Florescu und D. Kossmann. *A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database*. Rapport de recherche, INRIA, Mai 1999.

FYL09 W. Fan, J. X. Yu, J. Li, B. Ding und L. Qin. *Query translation from XPath to SQL in the presence of recursive DTDs*. VLDB Journal, Ausgabe 18 (2009) 4, Seiten 857-883, August 2009.

GBS02 T. Grabs, K. Böhm und H.-J. Schek. *XMLTM: Efficient Transaction Management for XML Documents*. Proceedings of the 2002 ACM International Conference on Information and Knowledge Management (CIKM), Seiten 142-152, ACM, 2002.

Git05 M. Gittens. *A Codd inspired amendment to my critical reading of the Third Manifesto*. November 2005. `http://www.gittens.nl/gittens/ATTM.pdf`

GK03 T. Grust und M. van Keulen. *Tree Awareness for Relational DBMS Kernels: Staircase Join*. LNCS 2818, Seiten 231-246, Springer, 2003.

GKT04 T. Grust, M. van Keulen und J. Teubner. *Accelerating XPath Evaluation in Any RDBMS*. ACM Transactions on Database Systems, Ausgabe 29 (2004) 1, Seiten 91-131, März 2004.

GLQ11 P. Genevès, N. Layaida und V. Quint. *Impact of XML Schema Evolution*. ACM Transactions on Internet Technology, Ausgabe 11 (2011) 1, Artikel 4, Juli 2011.

GRT07 T. Grust, J. Rittinger und J. Teubner. *Why Off-the-Shelf RDBMSs are Better at XPath Than You Might Expect*. Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, Seiten 949-957, ACM, 2007.

GV07 H. Georgiadis und V. Vassalos. *XPath on Steroids: Exploiting Relational Engines for XPath Performance*. Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, Seiten 317-328, ACM, 2007.

Har04 E. R. Harold. *XML 1.1 Bible*. Wiley & Sons, 2004.

HH04 M. P. Haustein und T. Härder. *A Lock Manager for Collaborative Processing of Natively Stored XML Documents*. Proceedings of the 19th Brazilian Symposium on Databases (SBBD), Seiten 230-244, UnB, 2004.

HKM04 S. Helmer, C.-C. Kanne und G. Moerkotte. *Evaluating Lock-based Protocols for Cooperation on XML Documents*. ACM SIGMOD Record, Ausgabe 33 (2004) 1, Seiten 58-63, März 2004.

IBM11a IBM. *IBM DB2 Information Center Version 9.7*. 2011. `http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp`

IBM11b IBM. *IBM Informix Dynamic Server Information Center Version 11.7*. 2011. `http://publib.boulder.ibm.com/infocenter/idshelp/v117/index.jsp`

ISO87 ISO 9075:1987. *International Standard: Information processing systems – Database language – SQL*. ISO, 1987.

ISO89 ISO/IEC 9075:1989. *International Standard: Information processing systems – Database language SQL with integrity enhancement*. ISO, April 1989.

ISO92 ISO/IEC 9075:1992. *International Standard: Information technology – Database languages – SQL*. ISO, November 1992.

ISO99 ISO/IEC 9075(1-5, 9, 10, 13):1999. *International Standard: Information technology – Database languages – SQL*. ISO, Dezember 1999.

ISO03 ISO/IEC 9075(1-4, 9-11, 13, 14):2003. *International Standard: Information technology – Database languages – SQL*. ISO, Dezember 2003.

ISO06 ISO/IEC 9075-14:2006. *International Standard: Information technology – Database languages – SQL – Part 14: XML-Related Specifications (SQL/XML)*. ISO, Juni 2006.

ISO08 ISO/IEC 9075(1-4, 9-11, 13, 14):2008. *International Standard: Information technology – Database languages – SQL*. ISO, Juli 2008.

ISO11 ISO/IEC 9075(1-4, 9-11, 13, 14):2011. *International Standard: Information technology – Database languages – SQL*. ISO, Dezember 2011.

JLW02 H. Jiang, H. Lu, W. Wang und J. X. Yu. *Path Materialization Revisited: An Efficient Storage Model for XML Data*. Australian Computer Science Communications, Ausgabe 24 (2002) 2, Seiten 85-94, Januar/Februar 2002.

KCD03 H. Katz, D. D. Chamberlin, D. Draper, M. Fernandez, M. Kay, J. Robie, M. Rys, J. Simeon, J. Tivy und P. Wadler. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley, 2003.

KD95 C. Kalus und P. Dadam. *Flexible Relations - Operational Support of Variant Relational Structures*. Proceedings of the 21th International Conference on Very Large Data Bases (VLDB), Seiten 539-550, Morgan Kaufmann, 1995.

KGH10 M. Kwietniewski, J. Gryz, S. Hazlewood und P. van Run. *Transforming XML Documents as Schemas Evolve*. Proceedings of the VLDB Endowment, Ausgabe 3 (2010) 1-2, Seiten 1577-1580, September 2010.

KT05 M. Klettke und T. Tiedt: *XML-Schemaevolution und inkrementelle Validierung*. Berliner XML-Tage 2005, Seiten 111-116, Humboldt-Universität zu Berlin, 2005.

KTW90 K. Küspert, J. Teuhola und L. Wegner. *Design Issues and First Experiences with a Visual Database Editor for the Extended NF²-Data Model*. Proceedings of the 23rd Hawaii International Conference on System Sciences (HICSS), Seiten 308-317, IEEE, 1990.

Kul10 L. Kulic. *Adaptability in XML-to-Relational Mapping Strategies*. Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Seiten 1674-1679, ACM, 2010.

KW07 P. S. Kaliappan und U. Wloka. *Investigations to the Conformance about Oracle, DB2, MS SQL Server, Sybase with respect to SQL:2003 Standard*. Datenbank-Spektrum, Ausgabe 7 (2007) 23, Seiten 38-44, November 2007.

Lau05 G. Lausen. *Datenbanken – Grundlagen und XML-Technologien*. Spektrum Akademischer Verlag, 2005.

LCM04 W. Lian, D. W. Cheung, N. Mamoulis und S.-M. Yiu. *An Efficient and Scalable Algorithm for Clustering XML Documents by Structure*. IEEE Transactions on Knowledge and Data Engineering, Ausgabe 16 (2004) 1, Seiten 82-96, Januar 2004.

LKA05 Z. H. Liu, M. Krishnaprasad und V. Arora. *Native XQuery Processing in Oracle XMLDB*. Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, Seiten 828-833, ACM, 2005.

LKD88 V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch und M. Wallrath. *Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions*. Proceedings of the 14th International Conference on Very Large Data Bases (VLDB), Seiten 294-305, Morgan Kaufmann, 1988.

LKW07 Z. H. Liu, M. Krishnaprasad, J. W. Warner, R. Angrish und V. Arora. *Effective and Efficient Update of XML in RDBMS*. Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, Seiten 925-936, ACM, 2007.

LL05 C. Li und T. W. Ling. *QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates*. Proceedings of the 2005 ACM International Conference on Information and Knowledge Management (CIKM), Seiten 501-508, ACM, 2005.

LLC05 J. Lu, T. W. Ling, C.-Y. Chan und T. Chen. *From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching*. Proceedings of the 31st International Conference on Very Large Data Bases (VLDB), Seiten 193-204, ACM, 2005.

LLH06 C. Li, T. W. Ling und M. Hu. *Efficient Processing of Updates in Dynamic XML Data*. Proceedings of the 22nd International Conference on Data Engineering (ICDE), IEEE, 2006.

LM01 Q. Li und B. Moon. *Indexing and Querying XML Data for Regular Path Expressions*. Proceedings of the 27th International Conference on Very Large Data Bases (VLDB), Seiten 361-370, Morgan Kaufmann, 2001.

LN02 L. H. Lam und W. Ng. *Querying XML Data Based on Nested Relational Sequence Model*. Proceedings of the 11th International World Wide Web Conference, Poster Session, ACM, 2002.

LS04 W. Lehner und H. Schöning. *»XQuery« – ein Überblick*. Datenbank-Spektrum, Ausgabe 4 (2004) 11, Seiten 23-32, November 2004.

Lu10 Y. Lu. *A New Relational Mapping Method of Graph-structured XML*. Proceedings of the 2010 International Conference on Computer Application and System Modeling (ICCASM), Seiten 229-231, IEEE, 2010.

Luf05 J. Lufter. *Unterstützung komplexer Datenstrukturen in SQL-Norm und objektrelationalen Datenbanksystemen*. Dissertation, Friedrich-Schiller-Universität Jena, 2005.

LYW05 H. Lu, J. X. Yu, G. Wang, S. Zheng, H. Jiang, G. Yu und A. Zhou. *What Makes the Differences: Benchmarking XML Database Implementations*. ACM Transactions on Internet Technology, Ausgabe 5 (2005) 1, Seiten 154-194, Februar 2005.

LYY96 Y. K. Lee, S.-J. Yoo, K. Yoon und P. B. Berra. *Index Structures for Structured Documents*. Proceedings of the 1st ACM International Conference on Digital Libraries, Seiten 91-99, ACM, 1996.

LZ03 G. Lee und F. Zemke. *SQL 2003 Standard Support in Oracle Database 10g*. Oracle White Paper, November 2003.

May99 W. May. *Information Extraction and Integration with Florid: The Mondial Case Study*. Technischer Report Nr. 131, Institut für Informatik, Universität Freiburg, Dezember 1999.

May10 Web-Seite zur Mondial-Datenbank der Arbeitsgruppe Datenbanken und Informationssysteme von Prof. W. May an der Universität Göttingen.
http://www.dbis.informatik.uni-goettingen.de/Mondial/

MB06 J. Melton und S. Buxton. *Querying XML – XQuery, XPath, and SQL/XML in Context*. Morgan Kaufmann, 2006.

Mel02 J. Melton. *Advanced SQL:1999: Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann, 2002.

Mel05 J. Melton. *SQL/XML*. Präsentation für ISO/IEC, JTC1 SC32 N1632, Dezember 2005.

MKF03 J.-E. Michels, K. Kulkarni, C. M. Farrar, A. Eisenberg und N. Mattos. *The SQL Standard*. it – Information Technology, Ausgabe 45 (2003) 1, Seiten 30-38, Januar 2003.

MLC07 M. M. Moro, L. Lim und Y.-C. Chang. *Schema Advisor for Hybrid Relational-XML DBMS*. Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, Seiten 959-970, ACM, 2007.

MM10 S. Mohammad und P. Martin. *LLS: Level-based Labeling Scheme for XML Databases*. Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research (CASCON), Seiten 115-127, ACM, 2010.

MOK10 M. Maghaydah, M. A. Orgun und I. Khazali. *Optimizing XML Twig Queries in Relational Systems*. Proceedings of the 14th International Database Engineering and Applications Symposium (IDEAS), Seiten 123-129, ACM, 2010.

Mou02 A. Mouat. *XML Diff and Patch Utilities*. Juni 2002.
http://sourceforge.net/projects/diffxml/

Muj11 T. N. Mujawar. *A Survey of XQuery: An XML Query Language*. Proceedings of the International Conference and Workshop on Emerging Trends in Technology (ICWET), Seiten 569-570, ACM, 2011.

MWD06 M. Mani, S. Wang, D. Dougherty und E. A. Rundensteiner. *Join Minimization in XML-to-SQL Translation: An Algebraic Approach*. ACM SIGMOD Record, Ausgabe 35 (2006) 1, Seiten 20-25, März 2006.

NL05 M. Nicola und B. van der Linden. *Native XML Support in DB2 Universal Database*. Proceedings of the 31st International Conference on Very Large Data Bases (VLDB), Seiten 1164-1174, ACM, 2005.

OOP04 P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller und N. Westbury. *ORDPATHs: Insert-Friendly XML Node Labels*. Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, Seiten 903-908, ACM, 2004.

OOP05 P. E. O'Neil, E. J. O'Neil, S. Pal, G. Schaller, I. Cseri. J. A. Blakeley, N. C. Westbury, S. Agarwal und F. S. Terek. *System and Method for relational representation of hierarchical data*. US-Patent 6889226 B2, Mai 2005.

OR10 M. F. O'Connor und M. Roantree. *Desirable Properties for XML Update Mechanisms*. Proceedings of the 2010 EDBT/ICDT Workshops, ACM, 2010.

Ora11a Oracle. *Java Platform, Standard Edition 6. API Specification*. 2011. `http://docs.oracle.com/javase/6/docs/api/`

Ora11b Oracle. *Oracle Database 11g Release 2 (11.2) Documentation Library*. 2011. `http://www.oracle.com/pls/db112/homepage`

Pet03 D. Petkovic. *SQL objektorientiert*. Addison-Wesley, 2003.

Pet07 D. Petkovic. *Die Unterstützung von SQL/OLAP im SQL-Standard und in relationalen Datenbanksystemen*. Datenbank-Spektrum, Ausgabe 7 (2007) 23, Seiten 29-37, November 2007.

Pis89 P. Pistor: *Variante Strukturen in HDBL*. In: H.-D. Ehrich, G. Engels, M. Gogolla und G. Saake (Hrsg.). *Abstracts Workshop Grundlagen von Datenbanken (GvD)*. Informatik-Bericht Nr. 89-02, Technische Universität Braunschweig, 1989.

Pis03 P. Pistor. *Neuerungen in SQL2003*. Fachvortrag an der Fachhochschule Heidelberg, Juni 2003.

PN06 M. Päßler und M. Nicola. *Native XML-Unterstützung in DB2 Viper*. Datenbank-Spektrum, Ausgabe 6 (2006) 17, Seiten 42-47, Mai 2006.

PT00 W. Panny und A. Taudes. *Einführung in den Sprachkern von SQL-99*. Springer, 2000.

RAB09 RSS Advisory Board. *RSS 2.0 Specification*. März 2009. `http://www.rssboard.org/rss-specification`

Raj02 N. Rajpal. *Using the XML Diff and Patch Tool in Your Applications*. Microsoft Corporation, August 2002. http://msdn.microsoft.com/en-us/library/aa302294.aspx

Ray01 E. T. Ray. *Einführung in XML*. O'Reilly, 2001.

RKS88 M. A. Roth, H. F. Korth und A. Silberschatz. *Extended algebra and calculus for nested relational databases*. ACM Transactions on Database Systems, Ausgabe 13 (1988) 4, Seiten 390-417, Dezember 1988.

Rot86 M. A. Roth. *Theory of non-first normal form relational databases*. Ph. D. Dissertation, Department of Computer Science, University of Texas, 1986.

Sch02 H. Schöning. *XML und Datenbanken*. Hanser, 2002.

Sch05 K.-D. Schewe. *Redundancy, Dependencies and Normal Forms for XML Databases*. Proceedings of the 16th Australasian Database Conference (ADC), Seiten 7-16, CRPIT 39, Australian Computer Society, 2005.

Sik08 A. Sikorski. *Collaborative processing of XML documents in Internet*. Theoretical and Applied Informatics, Ausgabe 20 (2008) 4, Seiten 329-345, 2008.

SJJ98 D. Shin, H. Jang und H. Jin. *BUS: An Effective Indexing and Retrieval Scheme in Structured Documents*. Proceedings of the 3rd ACM International Conference on Digital Libraries, Seiten 235-243, ACM, 1998.

SLF05 W. M. Shui, F. Lam, D. K. Fisher und R. K. Wong. *Querying and Maintaining Ordered XML Data using Relational Databases*. Proceedings of the 16th Australasian Database Conference (ADC), Seiten 85-94, CRPIT 39, Australian Computer Society, 2005.

SS86 H.-J. Schek und M. H. Scholl. *The Relational Model with Relation-Valued Attributes*. Information Systems, Ausgabe 11 (1986) 2, Seiten 137-147, 1986.

SSB01 J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh und B. Reinwald. *Efficiently Publishing Relational Data as XML Documents*. VLDB Journal, Ausgabe 10 (2001) 2-3, Seiten 133-154, September 2001.

STH99 J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt und J. Naughton. *Relational Databases for Querying XML Documents: Limitations und Opportunities*. Proceedings of the 25th International Conference on Very Large Data Bases (VLDB), Seiten 302-314, Morgan Kaufmann, 1999.

Stü00 G. Stürner. *Oracle8i – Der objekt-relationale Datenbank Server*. DBMS Publishing, 2000.

SZA06 S. Soltan, A. Zarnani, R. AliMohammadzadeh und M. Rahgozar. *IFDewey: A New Insert-Friendly Labeling Schema for XML Data*. Proceedings of the World Academy of Science, Engineering and Technology (WASET), Ausgabe 13, Seiten 116-118, Mai 2006.

Tha99 J. Thamm. *Visualisierungsverfahren zur Interaktion mit objekt-relationalen Datenbanken*. Dissertation, Universität Kassel, 1999. http://nbn-resolving.de/urn:nbn:de:hebis:34-149

The96 S. Thelemann. *Semantische Anreicherung eines Datenmodells für komplexe Objekte*. Dissertation, Universität Kassel, 1996. http://nbn-resolving.de/urn:nbn:de:hebis:34-2006092014633

TS06 C. Türker und G. Saake. *Objektrelationale Datenbanken*. Dpunkt-Verlag, 2006.

Tür03 C. Türker. *SQL:1999 und SQL:2003 – Objektrelationales SQL, SQLJ und SQL/XML*. Dpunkt-Verlag, 2003.

TVB02 I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita und C. Zhang. *Storing and Querying Ordered XML Using a Relational Database System*. Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Seiten 204-215, ACM, 2002.

VLL04 M. W. Vincent, J. Liu und C. Liu. *Strong Functional Dependencies and Their Application to Normal Forms in XML*. ACM Transactions on Database Systems, Ausgabe 29 (2004) 3, Seiten 445-462, September 2004.

W3C01 World Wide Web Consortium. *Canonical XML Version 1.0*. W3C Recommendation, März 2001. http://www.w3.org/TR/2001/REC-xml-c14n-20010315

W3C04a World Wide Web Consortium. *Document Object Model (DOM) Level 3 Core Specification Version 1.0*. W3C Recommendation, April 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/

W3C04b World Wide Web Consortium. *XML Information Set (Second Edition)*. W3C Recommendation, Februar 2004. http://www.w3.org/TR/2004/REC-xml-infoset-20040204/

W3C04c World Wide Web Consortium. *XML Schema – Part 0: Primer (Second Edition)*. W3C Recommendation, Oktober 2004. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/

W3C04d World Wide Web Consortium. *XML Schema – Part 1: Structures (Second Edition)*. W3C Recommendation, Oktober 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

W3C04e World Wide Web Consortium. *XML Schema – Part 2: Datatypes (Second Edition)*. W3C Recommendation, Oktober 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

W3C06a World Wide Web Consortium. *Extensible Markup Language (XML) Version 1.1 (Second Edition)*. W3C Recommendation, September 2006. http://www.w3.org/TR/2006/REC-xml11-20060816/

W3C06b World Wide Web Consortium. *Namespaces in XML 1.1 (Second Edition)*. W3C Recommendation, August 2006. http://www.w3.org/TR/2006/REC-xml-names11-20060816

W3C07 World Wide Web Consortium. *XSL Transformations (XSLT) Version 2.0*. W3C Recommendation, Januar 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/

W3C10a World Wide Web Consortium. *XML Linking Language (XLink) Version 1.1*. W3C Recommendation, Mai 2010. http://www.w3.org/TR/2010/REC-xlink11-20100506/

W3C10b World Wide Web Consortium. *XML Path Language (XPath) 2.0 (Second Edition)*. W3C Recommendation, Dezember 2010. http://www.w3.org/TR/2010/REC-xpath20-20101214/

W3C10c World Wide Web Consortium. *XQuery 1.0: An XML Query Language (Second Edition)*. W3C Recommendation, Dezember 2010. http://www.w3.org/TR/2010/REC-xquery-20101214/

W3C10d World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)*. W3C Recommendation, Dezember 2010. http://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/

W3C11 World Wide Web Consortium. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. W3C Recommendation, August 2011. http://www.w3.org/TR/2011/REC-SVG11-20110816/

W3S11 W3Schools. *XML Tutorial*. `http://www.w3schools.com/xml/default.asp`

WA01 L. Wegner und M. Ahmad. *Orthogonality in DBMS Design: the XML Approach*. Proceedings of the 10th International Workshop on Foundations of Models for Information Integration (FMII), University of Manchester, 2001.

Weg89 L. Wegner. *ESCHER – Interactive, Visual Handling of Complex Objects in the Extended NF²-Database Model*. Proceedings of the IFIP TC-2 Working Conference on Visual Database Systems, Seiten 277-297, Elsevier North-Holland, 1989.

Weg91 L. Wegner. *Let the Fingers Do the Walking: Object Manipulation in an NF² Database Editor*. Proceedings of the Symposium on New Results and New Trends in Computer Science, Seiten 337-358, LNCS 555, Springer, 1991.

Weg09 L. Wegner. *Einführung in XML*. Vorlesungsskript, Universität Kassel, 2009.

Weg11 L. Wegner. *Datenbanken I*. Vorlesungsskript, Universität Kassel, 2011.

WLH04 X. Wu, M. L. Lee und W. Hsu. *A Prime Number Labeling Scheme for Dynamic Ordered XML Trees*. Proceedings of the 20th International Conference on Data Engineering (ICDE), Seiten 66-78, IEEE, 2004.

XBL07 L. Xu, Z. Bao und T. W. Ling. *A Dynamic Labeling Scheme using Vectors*. Proceedings of the 18th International Conference on Database and Expert Systems Applications (DEXA), Seiten 130-140, LNCS 4653, Springer, 2007.

XLW09 L. Xu, T. W. Ling, H. Wu und Z. Bao. *DDE: From Dewey to a Fully Dynamic XML Labeling Scheme*. Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, Seiten 719-730, ACM, 2009.

XMY10 Y. Xu, S. Ma, S. Yi und Y. Yan. *From XML Schema to Relations: A Incremental Approach to XML Storage*. Proceedings of the 2010 International Conference on Computational Intelligence and Software Engineering (CiSE), IEEE, 2010.

YAS01 M. Yoshikawa, T. Amagasa, T. Shimura und S. Uemura. *XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases*. ACM Transactions on Internet Technology, Ausgabe 1 (2001) 1, Seiten 110-141, August 2001.

Zei01a K. Zeidenstein. *DB2's object-relational highlights: User-defined structured types und object views in DB2*. IBM Corporation, Oktober 2001. http://www.ibm.com/developerworks/db2/library/techarticle/zeidenstein/0108zeidenstein.html

Zei01b K. Zeidenstein. *DB2's object-relational highlights: Store and invoke structured type objects*. IBM Corporation, Oktober 2001. http://www.ibm.com/developerworks/db2/library/techarticle/zeidenstein/0109zeidenstein.html

ZLF11 C. Zhuang, Z. Lin und S. Feng. *Insert-friendly XML Containment Labeling Scheme*. Proceedings of the 2011 ACM International Conference on Information and Knowledge Management (CIKM), Seiten 2449-2452, ACM, 2011.

ZLZ01 A. Zhou, H. Lu, S. Zheng, Y. Liang, L. Zhang, W. Ji und Z. Tian. *VXMLR: A Visual XML-Relational Database System*. Proceedings of the 27th International Conference on Very Large Data Bases (VLDB), Seiten 719-720, Morgan Kaufmann, 2001.

ZND01 C. Zhang, J. Naughton, D. DeWitt, Q. Luo und G. Lohmann. *On Supporting Containment Queries in Relational Database Management Systems*. Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Seiten 425-436, ACM, 2001.