Effiziente XML-Speicherung und XQuery-Auswertung mit Oracle9i

Diplomarbeit

Universität Rostock, Fachbereich Informatik

vorgelegt von:
Below, Björn

geboren am:
25.02.1975 in Cottbus

Gutachter:
Prof. Dr. rer. nat. habil. Andreas Heuer,
Prof. Dr. rer. nat. Clemens H. Cap

Betreuer:
Dr.-Ing. Holger Meyer,
Dipl.-Inf. Denny Priebe

Abgabedatum: 19.03.2003

Zusammenfassung

Als Standardformat für semistrukturierte Daten und deren Austausch findet XML eine zunehmende Verbreitung in den unterschiedlichsten Bereichen. Viele Datenbanksysteme unterstützen bereits die Archivierung von XML-Daten. Um auf Informationen im XML-Format zugreifen zu können, wurden eine Reihe von Anfragesprachen entwickelt. Eine davon ist XQuery, die sich mit großer Wahrscheinlichkeit als Standard etablieren wird.
In dieser Arbeit wird eine Technik zur direkten Abbildung der hierarchischen XML-Dokumentstruktur auf objektrelationale Strukturen des Datenbanksystems Oracle9i entwickelt. Es wird ein generisches Speichermodell für XML-Dokumente ohne festes Schema betrachtet, das Änderungsoperationen und Anfrageauswertung gleichermaßen unterstützt. Zu diesem Zweck wird ein Satz von Methoden entwickelt, der auf Seiten des Datenbank-Servers zur Verfügung gestellt werden soll.

Abstract

XML became a standard format for semi-structured data and their exchange between different applications. Over the years it has reached a wide dissemination. Many database systems support a data type for storing documents that use XML to hold their data. To query XML data sources the development of a query language became very important. Latest result of this evolution is XQuery, which seems to become a standard for retrieving and interpreting such information.
Scope of work is to find a mapping between the hierarchical structures of XML documents and the object relational structures of the database system Oracle9i. A generic storage model is considered that should be able to store documents independent of a schema description. Both operations for update and retrieval should be highly supported by this model. A set of methods is discussed that shall be placed inside the database at a client application's disposal to serve this purpose.

CR-Klassifikation

E.1 DATA STRUCTURES
E.2 DATA STORAGE REPRESENTATIONS
H.2 DATABASE MANAGEMENT
H.2.1 Logical Design
H.2.3 Languages
H.2.4 Systems
I.7 DOCUMENT AND TEXT PROCESSING
I.7.1 Document and Text Editing
I.7.2 Document Preparation

Key Words

XML, Query Language, XQuery, Storage Model, Database, Oracle9i

Inhaltsverzeichnis

Abkürzungsverzeichnis

BLOB	Binary Large Object
CLOB	Character Large Object
DB	Datenbank
DBMS	Datenbank Management System
DOM	Document Object Model
DTD	Document Type Definition
FLWR	FOR-LET-WHERE-RETURN
HTML	Hypertext Markup Language
IBM	Internal Business Machines Corporation
ID	Identifikator
ISO	International Standards Organization
LOB	Large Object
OID	Objekt Identifikator
PI	Processing Instruction
SGML	Standard Generalized Markup Language
SQL	Structured Query Language
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Kapitel 1

Einführung und Motivation

Bei der Erstellung, Speicherung und Verarbeitung von Dokumenten gilt es, einen wesentlichen Punkt zu beachten. Dokumente bestehen nicht nur aus Inhalt wie Text, Tabellen und Bildern. Eine sehr wichtige Rolle spielen auch Informationen zur visuellen Darstellung dieser Dinge, das Layout. Außerdem besitzen Dokumente eine Struktur. Die Struktur beschreibt die Aufteilung in beispielsweise Kapitel und Abschnitte. Zusätzlich können bestimmte Wörter und Satzteile mit logischen Informationen versehen werden. Ein Textteil kann als Zitat ausgezeichnet werden, ein Wort als Personenname und ähnliche Dinge. Diese Zusammenhänge sind für einen Leser sofort ersichtlich, ein verarbeitender Computer kann sie nicht ohne weiteres erkennen. Die Idee, solche Informationen mit abzuspeichern, führte zur Entwicklung einer Sprache, mit der derartige Auszeichnungen beschrieben werden können und die 1986 veröffentlicht wurde - die *Structured Generalized Markup Language* (SGML). Der Sinn einer sogenannten Metasprache wie SGML besteht darin, dass mit ihrer Hilfe die Regeln und Symbole einer anderen Sprache definiert werden können. Im Laufe der Jahre hat sich SGML als zu komplex herausgestellt. Einige Teile der Spezifikation wurden nicht benutzt, unterstützende Software neigte zu Fehlern. Aus diesem Grund wurde 1996 unter der Schirmherrschaft des World Wide Web Consortium (W3C) die *Extensible Markup Language* (XML) [W3C00] entwickelt. XML stellt eine Teilmenge von SGML dar und hat sich als sehr viel flexibler erwiesen. Als eine zusätzliche Motivation kann die Beseitigung der nicht vorhandenen Erweiterbarkeit der *Hypertext Markup Language* (HTML), einer mittels SGML definierten Sprache, angesehen werden. Ende 1997 wurde die Version 1.0 der XML-Spezifikation zu einem offiziellen Standard erklärt und im Web vorgestellt.

In den vergangenen Jahren hat sich XML als Standardformat für semistrukturierte Daten und deren Austausch zwischen den verschiedensten Anwendungen etabliert. XML bietet als Datenformat eine Reihe von Vortei-

len wie Flexibilität und Plattformunabhängigkeit. Dies ist der Grund für seine weite Verbreitung in den unterschiedlichsten Bereichen. Nicht immer ist es jedoch ausreichend, nur die Dokumente als solches abrufbar zu halten. Neben effizienten Strukturen zur Speicherung sind auch Möglichkeiten des Zugriffs auf Teile oder die Gesamtheit der archivierten Daten erforderlich. Daher existieren mittlerweile eine Reihe von XML-Anfragesprachen, darunter auch der sich abzeichnende Standard XQuery (kurz für XML Query). Mittels XQuery können sowohl dokumentzentrierte Anfragen nach der Dokumentstruktur als auch datenzentrierte Anfragen an die strukturierten Anteile von Dokumenten umgesetzt werden.

Mit der zunehmenden Verbreitung von XML als Datenformat steigt natürlich auch die Anzahl der Dokumente, die im XML-Format vorliegen. Aus diesem Grund ist es erforderlich, Möglichkeiten zu finden, die anfallenden Mengen von Dokumenten effizient zu archivieren. Ein XML-Datentyp wird daher bereits von vielen Datenbanksystemen unterstützt. Allerdings werden in den meisten Fällen die XML-Dokumente als *Large Object* (LOB) innerhalb der Datenbank bzw. als Datei im Dateisystem abgespeichert. Beim letzteren Ansatz enthält die Datenbank nur die Verweise auf die entsprechenden Dokumente. Um bei dieser Speicherungstechnik Anfragen zu ermöglichen, muss zusätzlich ein Index angelegt werden. SQL-Anfragen liefern dann aber nur das gesamte Dokument. Konkrete Anfragen an Teile des Inhalts werden nicht unterstützt.

Ein Beispiel für ein Datenbanksystem mit XML-Unterstützung ist DB2 von IBM. Dort wird mit dem XML-Extender [IBM99] ein Tool zur Verfügung gestellt, mit dessen Hilfe man zum einen die Möglichkeit hat, komplette XML-Dokumente zu archivieren. Zum anderen können aber auch die reinen Daten extrahiert und in der Datenbank abgelegt werden.
Als neuere Entwicklung kann an dieser Stelle Infonyte-DB genannt werden[1]. Infonyte verspricht ein performantes und transaktionssicheres XML-Warehouse, ohne ein vordefiniertes Schema zu erfordern. Die gängigen XML-Standards zur Verarbeitung, Änderung, Validierung und zum Durchsuchen werden unterstützt.
Auch Oracle bietet seit der Version 9i einen XML-Datentyp an. Die in der vorliegenden Arbeit diskutierten Speicherstrukturen sollen mit den Möglichkeiten dieses Datenbanksystems prototypisch umgesetzt werden. Deshalb ist der vorhandene XML-Typ zwar nicht hauptsächlicher Gegenstand dieser Diplomarbeit, wird aber trotzdem innerhalb eines eigenen Abschnitts näher betrachtet. Es wird untersucht, weshalb die Entwicklung eines eigenen Modells unter Oracle sinnvoll ist.

[1] http://www.infonyte.com

1.1 Aufgabenstellung

Ziel dieser Arbeit ist die Konzeptionierung und Entwicklung einer generischen Speicherungsstruktur für beliebige XML-Dokumente unabhängig von einer vorhandenen XML-Schema-Beschreibung bzw. *Document Type Definition* (DTD). Unter Ausnutzung der objektrelationalen Möglichkeiten des Datenbank Management Systems (DBMS) Oracle9i soll das Speicherungsmodell Zugriffe sowohl auf den Inhalt als auch auf Informationen bezüglich der Struktur der Dokumente erlauben. Diesbezüglich müssen Anfrageauswertung und Änderungsoperationen gleichermaßen effizient unterstützt werden. Die Konzeptionierung der Anfrageauswertung bezüglich des zu entwickelnden XML-Datentyps soll auf der Anfragesprache XQuery basieren und als Ergebnis einen Satz von Methoden liefern, der auf Datenbankseite bereitgestellt wird.

1.2 Einteilung der Arbeit

In Kapitel 2 werden drei Ansätze vorgestellt, mit denen sich XML-Daten in Datenbanken abspeichern lassen. Damit soll ein Grundverständnis für das zu entwickelnde Speichermodell vermittelt werden. Zusätzlich werden zwei spezielle Verfahren genannt und ihre Vor- und Nachteile in Bezug auf Anfragen und Änderungsoperationen diskutiert.
Kapitel 3 liefert einen Einblick in das Datenbanksystem Oracle9i, unter dem die zu entwickelnde Speicherstruktur realisiert wird. Ziel der Betrachtungen soll jedoch keine Einführung in die relationalen Aspekte eines DBMS sein. Vielmehr geht es um die objektrelationalen Konzepte, die speziell von Oracle unterstützt werden und sich für die Abspeicherung von XML-Daten nutzen lassen. Ein weiterer Abschnitt beschäftigt sich mit dem von Oracle angebotenen XML-Datentyp. Damit soll insbesondere motiviert werden, warum es sinnvoll ist, ein eigenes Modell zu entwickeln. Den Abschluss des Kapitels bildet ein Abschnitt, in dem vertiefende Literatur zu den behandelten Themen aufgeführt wird. Die angegebenen Veröffentlichungen sind entweder Grundlage der in dieser Arbeit vorgestellten Lösungsansätze oder lassen sich für eine weiterführende Recherche nutzen.
Inhalt des 4. Kapitels ist die Diskussion eines konkreten Modells zum Abspeichern von XML-Dokumenten. Als Ausgangspunkt dient dabei das XML Information Set, ein Datenmodell für XML-Daten, mit dessen Hilfe relevante Informationen bezüglich eines Dokumentes identifiziert werden sollen. Die Umsetzung der einzelnen Teile und Informationen eines XML-Dokumentes in Strukturen der Datenbank wird ausführlich besprochen und bewertet.

Das folgende Kapitel 5 beschäftigt sich mit der Anfrageauswertung bezüglich eines abgespeicherten Dokumentes. Voraussetzung dafür sind die Anfragesprache XQuery und, da deren Pfadausdrücke darauf basieren, die XML-Navigationssprache XPath. Beide werden zu Beginn des Kapitels vorgestellt. Darauf aufbauend werden im Anschluss der auf Datenbankseite umzusetzende Sprachumfang und die daraus resultierenden Methoden, mit denen ein Zugriff auf die Daten ermöglicht wird, diskutiert. Den Abschluss bildet ein Abschnitt, der über weiterführende Literatur informiert.
Ein sehr wichtiger Aspekt ist die Möglichkeit, abgespeicherte Dokumente nachträglich zu verändern. Trotzdem dies momentan noch keine Berücksichtigung innerhalb von XQuery findet, sollte ein Speichermodell derartige Operationen unterstützen. Dieser Ansatz ist Schwerpunkt von Kapitel 6. Dort werden grundsätzliche Änderungsoperationen und deren Umsetzung basierend auf der gewählten Speicherstruktur vorgestellt.
Das letzte Kapitel liefert eine Zusammenfassung der vorliegenden Arbeit sowie einen Ausblick auf noch zu behandelnde Themen.
Der Anhang enthält ein Beispieldokument, das innerhalb der Arbeit zur Erklärung der betrachteten Techniken benutzt wird. Außerdem sind die verwendete Entwicklungsumgebung, unter der Teile der diskutierten Themen prototypisch implementiert worden sind, die ausführlichen Definitionen aller Objekttypen und die Quelltexte der bereits implementierten Teile der Konzeption enthalten.

Kapitel 2

Speicherung von XML-Dokumenten

Zur Speicherung von XML-Dokumenten existieren eine Reihe von Techniken. Diese werden mit ihren Eigenschaften vorgestellt sowie Vor- und Nachteile diskutiert. Im Anschluss daran werden zwei Verfahren beschrieben, mit denen sowohl der Inhalt von Dokumenten als auch die Struktur abgespeichert werden können.

2.1 Speicherungstechniken

Will man große Mengen von XML-Dokumenten auf Dauer archivieren, bieten sich verschiedene Ansätze, um die vorhandenen Daten zu sichern. Ein wichtiger Aspekt bei der Wahl einer Technik ist der Grad der Strukturierung der vorliegenden Informationen. Geht es hauptsächlich um den Inhalt eines Dokumentes? Sollen nur ganze Dokumente oder auch Teile davon abrufbar sein? Ist eventuell sogar eine Beschreibung der Struktur einer Reihe von gleichartigen Dokumenten in Form eines Schemas oder einer DTD vorhanden? Ausgehend von diesen Fragen kann man unter verschiedenen Techniken wählen. Im Folgenden wird eine Einteilung der gängigsten Methoden vorgenommen. Dabei lassen sich drei Verfahren unterscheiden, die kurz angesprochen werden.

- Dokumente können als ganze Datei im Filesystem oder als LOB innerhalb einer Datenbank gespeichert werden[1].
- Bei der Speicherung wird die Struktur der Dokumente berücksichtigt.

[1] zum Beispiel als *Character Large Object* (CLOB)

- Es erfolgt eine Abbildung der Struktur der Dokumente auf Datenbankstrukturen.

Eine ausführlichere Beschreibung der aufgezeigten Ansätze wird in [KM02] gegeben.

2.1.1 Speicherung als ganze Datei

Bei diesem Ansatz bleiben die Dokumente im Originalzustand erhalten. Sie sind jederzeit verfügbar und es ist keine Umwandlung nötig. Da für die Beantwortung von Anfragen die gespeicherten Dokumente sequentiell durchsucht werden müssen, ist das Anlegen eines Indexes zur Performance-Steigerung sinnvoll. Eine Indizierung kann mithilfe der aus dem *Information Retrieval* bekannten Verfahren erfolgen (siehe hierzu u.a. [BR99]). Dabei werden Informationen über Inhalte und Strukturen der XML-Dokumente extrahiert.
Anhand des Indexes können Anfragen, die vollständige Dokumente als Ergebnis erwarten, gut realisiert werden. Für die Rückgabe von Teilen von XML-Dokumenten ist das zusätzliche Parsen der Dokumente erforderlich.
Diese Speicherungsform wird unter anderem von Oracle9i unterstützt. Sie eignet sich besonders für dokumentzentrierte XML-Anwendungen.

2.1.2 Speicherung der Struktur

Bei dieser Methode versucht man, sowohl Inhalt der Dokumente als auch Informationen zur Dokumentstruktur in eine allgemeine Speicherungsstruktur abzubilden. Ziel ist die Generierung eines festen Datenbankschemas anhand der Graphenstruktur der Dokumente, meist innerhalb einer relationalen Datenbank. Mithilfe des festen Datenbankschemas soll die Speicherung verschiedener Klassen von XML-Dokumenten unabhängig von einer vorhandenen DTD bzw. einer XML-Schema-Beschreibung möglich sein.
Diese Herangehensweise wird im Verlauf der vorliegenden Arbeit dazu verwendet, ein Konzept für eine effiziente XML-Speicherung in Oracle9i zu entwickeln.
Eine Rekonstruktion der nach dieser Methode abgelegten Dokumente ist aufgrund der vollständigen Abbildung der Graphenstruktur realisierbar, kann aber in Abhängigkeit von der Anzahl der verwendeten Relationen sehr aufwändig werden. Anfragen an die abgespeicherten Daten sind mithilfe von XML-Anfragesprachen wie XQuery oder XPath möglich. Allerdings ist dann eine Abbildung in SQL basierend auf den gewählten Speicherungsstrukturen erforderlich.

Diese Art der Speicherungsform eignet sich sowohl für daten- als auch für dokumentzentrierte Anwendungen.

2.1.3 Abbildung auf Datenbankstruktur

Voraussetzung für diese Methode ist eine vorhandene XML-Struktur entweder in Form einer DTD oder eines XML-Schemas. Anhand der Strukturbeschreibung werden dann mithilfe von sogenannten *Mapping*-Regeln pro Klasse von Dokumenten separate Relationen generiert. Beispielsweise können Tag-Namen und Attributnamen auf Attribute von Tabellen abgebildet werden. Im Gegensatz dazu kann man die verwendeten Regeln aber auch derart modifizieren, dass nur bestimmte Informationen aus den XML-Dokumenten in die Datenbankrelationen übernommen werden. Dieser Vorgang wird als Schreddern der Daten bezeichnet.
Der Nachteil des Schredderns besteht in jedem Falle darin, dass es Anfragen geben kann, deren Beantwortung anhand des kompletten Dokumentes möglich ist, nicht aber mithilfe der in der Datenbank abgelegten Informationen. Außerdem entfällt die Möglichkeit, die ursprünglichen Dokumente zu rekonstruieren. Eine Änderung des Schemas der XML-Dokumente zieht eine Anpassung des Datenbankschemas nach sich.
Diese Form der Speicherung eignet sich am ehesten für datenzentrierte Anwendungen.

In den folgenden Abschnitten werden verschiedene Speicherungsmodelle für XML-Dokumente vorgestellt.

2.2 Indizierung von XML-Dokumenten

In der Arbeit von Guido Rost [Ros01] wird eine Implementierung der XML-Anfragesprache XQuery vorgestellt. Diese basiert auf einer Speicherungsmethode, bei der der Inhalt eines XML-Dokumentes vollständig indiziert und dann in ein festes Datenbankschema eingefügt wird. Ausgangspunkt für diese Art der Speicherung ist die in [SYU] vorgestellte Methode, die bereits für die Implementierung von Quilt [CRF00] genutzt wurde, einer früheren XML-Anfragesprache. Verwendet wird ein generisches Datenbankschema, das beliebige XML-Dokumente aufnehmen kann.

2.2.1 Kodierung

Dabei erhält jedes Wort eines Textknotens eine fortlaufende, ganzzahlige Nummer beginnend bei Eins. Worte, die in Attributen vorkommen, werden nicht berücksichtigt. Im Anschluss daran wird jeder Knoten des Dokumentes als Paar von Integerwerten (start,ende) aufgefasst.

Textknoten

"start" ist der Index des ersten Wortes des aktuellen Textknotens.
"ende" ist der Index des letzten Wortes.

Elementknoten

Bei Elementen werden sowohl "start" als auch "ende" wiederum als Paare definiert.
"start" eines Elementknotens setzt sich dann aus dem Index des letzten Wortes vor dem öffnenden Knoten (erster Wert des Paares) und der Nummer des öffnenden Elementes in der Reihe aufeinanderfolgender Elemente (zweiter Wert des Paares) zusammen.
"ende" eines Elementknotens setzt sich aus dem Index des letzten Wortes vor dem schließenden Knoten (erster Wert) und der Nummer des schließenden Elementes in der Reihe aufeinanderfolgender Elemente (zweiter Wert) zusammen.
Zur Verdeutlichung soll ein kurzes Beispiel dienen (Ausgangspunkt ist das Beispieldokument siehe Anhang A.1):

```
<buecher>
  <buch einband="paperback">
    <titel>The[1] Lord[2] Of[3] The[4] Rings[5]</titel>
    <autor>J.R.R.[6] Tolkien[7]</autor>
    <verlag>Harper[8] Collins[9]</verlag>
  </buch>
  ...
```

Wie beschrieben wurden nur die Wörter durchnummeriert. Dann wird der Textknoten, der Inhalt des <titel>-Elementes ist, durch das Wertepaar (1,5) kodiert. Das <titel>-Element selbst erhält die Kodierung ((0,3),(5,1)).

Attributknoten

Bei Attributen werden genau wie bei den Elementknoten sowohl "start" als auch "ende" als Paare definiert. Allerdings erhalten beide Werte dieselbe Kodierung, da sich Anfang und Ende von Attributen nicht unterscheiden.

"start" und "ende" eines Attributknotens entsprechen beide dem Start-Wert des dazugehörigen Elementknotens. Falls ein Element mehrere Attribute besitzt, erhalten alle diese Kodierung, da die Reihenfolge innerhalb der Attributmenge keine Rolle spielt.

2.2.2 Datenbankschema

Die derart kodierten Dokumente werden dann auf folgende Relationen abgebildet:

- eine Relation "documents" enthält den Namen der Dokumente und zu jedem Dokument einen Identifikator,
- eine Relation "path" enthält alle im Dokument vorkommenden Pfade und zu jedem Pfad einen Identifikator; außerdem wird der Dokument-Identifikator als Fremdschlüssel mit aufgeführt,
- eine Relation "element" enthält Angaben zur Position des Elementes innerhalb des Dokumentes sowie Dokument- und Pfad-Identifikator als Fremdschlüssel,
- in eine Relation "text" werden der Inhalt der Textknoten und deren Position gespeichert; außerdem Dokument- und Pfad-Identifikator als Fremdschlüssel,
- eine Relation "attribute" dient der Aufnahme der Attributwerte und der Position der Attribute; auch hier werden Dokument- und Pfad-Identifikator als Fremdschlüssel verwendet.

2.2.3 Vor- und Nachteile

Ein wichtiger Vorteil ergibt sich für das Datenbankschema zur Aufnahme der Dokumente. Ändert sich nachträglich die Struktur der Dokumente, müssen die Relationen nicht angepasst werden.
Dies gilt jedoch nicht für die gespeicherten Dokumente. Durch die gewählte Kodierung gestalten sich nachträgliche Änderungen äußerst aufwändig. Allein durch das Einfügen eines neuen Textknotens innerhalb des Dokumentes ändert sich die Nummer für alle folgenden Wörter und damit die Kodierung aller weiteren Knoten.
Zusätzlich lassen sich vollständige Pfadausdrücke (siehe Kapitel 5), mit denen Teile von Dokumenten adressiert werden können, nur in sehr komplexe SQL-Anweisungen umsetzen. Diese Einschränkung des Speicherungsmodells wird auch in [Ros01] erwähnt.

2.3 Abbildung der Baumstruktur

Ein zweites Verfahren, mit dem sich ebenfalls Informationen zur Dokumentstruktur abbilden lassen, nutzt die Baumstruktur eines XML-Dokumentes. Abbildung 2.1 zeigt eine derartige Baumstruktur für das Beispieldokument.

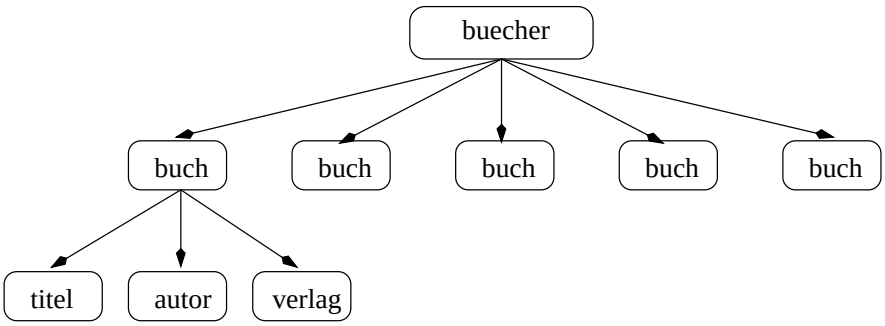


Abbildung 2.1: Baumstruktur eines XML-Dokumentes

Die im Folgenden beschriebene Art der Speicherung wurde in [Bra98] vorgestellt und wird auch in [KM02] erwähnt.
Ein Knoten des Baumes erhält jeweils einen eindeutigen Identifikator, der ihn von allen anderen Knoten unterscheidet. Ausgangspunkt für die Speicherung der Graphenstruktur sind Informationen zum jeweiligen Vorgänger eines Knotens innerhalb des Baumes. In Bezug auf das Beispiel bedeutet das, dass zum Elementknoten <titel> (Abb. 2.1) ein zusätzlicher Verweis auf dessen Vater (das erste <buch>-Element) abgelegt wird. Der Verweis kann aus der ID des entsprechenden Vorgängers bestehen.

2.3.1 Datenbankschema

Knoten haben unterschiedliche Typen, es gibt u.a. Elementknoten, Attributknoten und *Processing Instructions*. Jeder Knotentyp wird in einer eigenen Relation abgespeichert. Eine Relation dient beispielsweise der Aufnahme von Elementen, d.h. jedes enthaltene Tupel stellt ein Element eines XML-Dokumentes dar.

- Eine **Doc-ID** wird als Verweis auf das entsprechende Dokument benötigt, falls mehrere Dokumente abgespeichert werden sollen.
- Die Spalte **Element-Name** enthält als Werte die Tag-Namen der Elemente.

- **ID** dient zum einen der eindeutigen Identifikation eines Knotens. Zum anderen werden über dieses Attribut die Verwandtschaftsbeziehungen zwischen den einzelnen Knoten realisiert.
- **Vorgänger** enthält die ID des Knotens, dessen Subknoten durch das aktuelle Tupel repräsentiert wird.
- Durch die **Kind-Nr.** wird die Ordnung innerhalb der Menge von Subknoten des darüberliegenden Elementes festgelegt. Sie kennzeichnet die Position des aktuellen Knotens in dieser Ordnung.
- **Wert** enthält die Textdaten, also den Inhalt eines XML-Elementes. Für das <titel>-Element wäre das z.B. der String "The Lord Of The Rings".

In einer weiteren Tabelle werden alle Attribute gesammelt.

- Die **Doc-ID** wird wiederum als Verweis auf das entsprechende Dokument benötigt.
- **Element-ID** enthält die ID des Elementes, zu dem das aktuelle Attribut gehört.
- **Attributname** und
- **Attributwert** erklären sich von selbst.

Weitere Informationen, wie z.B. Kommentare und *Processing Instructions* werden ebenfalls in eigenen Relationen abgespeichert. Damit lassen sich die grundlegenden Informationen eines XML-Dokumentes abbilden.

Doc-ID	Name	ID	Vater	Kind-Nr.	Wert
doc_1	buecher	elem_1		1	
doc_1	buch	elem_2	elem_1	1	
doc_1	titel	elem_3	elem_2	1	The Lord Of ...
doc_1	autor	elem_4	elem_2	2	J.R.R. Tolkien
doc_1	verlag	elem_5	elem_2	3	Harper Collins
doc_1	buch	elem_6	elem_1	2	
doc_1	titel	elem_7	elem_6	1	Datenbanken ...
...					

Tabelle 2.1: Tabelle zum Abspeichern von Elementen

Zu diesem Modell existieren verschiedene Erweiterungen. Unter anderem können die Werte von Attributen und Elementen in zusätzlichen Tabellen

Doc-ID	Elem-ID	Name	Wert
doc_1	elem_2	einband	paperback
doc_1	elem_6	einband	gebunden
...			

Tabelle 2.2: Tabelle zum Abspeichern von Attributen

in Abhängigkeit von ihren Typen (String, Integer, ...) abgespeichert werden. In den entsprechenden "Wert"-Spalten werden sie dann ebenfalls durch eindeutige Identifikatoren referenziert. Somit können Werte im richtigen Format abgelegt werden, was beispielsweise typabhängige Vergleichsoperationen ermöglicht.

2.3.2 Vor- und Nachteile

Wie auch beim in Abschnitt 2.2 vorgestellten Verfahren besteht ein großer Vorteil darin, dass Dokumente unabhängig von einer DTD bzw. Schema-Beschreibung abgelegt werden können. Falls sich die Struktur einer Klasse von Dokumenten ändert, hat das keinerlei Auswirkungen auf die Relationen, in denen die Daten gespeichert werden.
Dazu kommt, dass bei diesem Speichermodell nachträgliche Update-Operationen möglich sind, da keine statische Kodierung verwendet wird, die einem Neueinfügen bzw. Löschen von Elementen im Weg steht.
Allerdings muss man auch mit diesem Modell Restriktionen in Kauf nehmen. Es gibt zum Beispiel keine Möglichkeit, *mixed content* abzubilden, d.h. direkte Nachfahren eines Elementknotens sind sowohl andere Elemente als auch Textknoten. Gerade dies als wesentlicher Bestandteil von XML-Dokumenten sollte aber in jedem Fall durch eine Speicherungsstruktur realisierbar sein.
Außerdem lassen sich Anfragen, die mehrere Elemente oder Attribute enthalten, nur sehr ineffizient umsetzen, weil dabei eine Menge von Verbund-Operationen über den betreffenden Relationen durchgeführt werden müssen.

Kapitel 3

Oracle9i

Ziel dieser Arbeit soll es sein, eine Speicherungsstruktur für beliebige XML-Dokumente zu finden und innerhalb eines konkreten Datenbanksystems umzusetzen. Als DBMS fungiert dabei Oracle9i. Es gilt, die dort vorhandenen objektrelationalen Eigenschaften bei der Entwicklung des Modells sinnvoll auszunutzen. Die Vorstellung dieser objektrelationalen Eigenschaften von Oracle9i soll hauptsächlich Inhalt dieses Kapitels sein.
Da Oracle selbst einen Datentyp anbietet, mit dessen Hilfe XML-Daten innerhalb der Datenbank abgelegt werden können und der auch einen Zugriff auf die abgespeicherten Informationen gewährleistet, wird in Abschnitt 3.2 die vorhandene Lösung kurz vorgestellt.

3.1 Objektrelationale Eigenschaften des Datenbanksystems

Ein objektrelationales Datenbanksystem wie Oracle9i[1] erweitert die Eigenschaften eines herkömmlichen relationalen Datenbanksystems um objektorientierte Konzepte und Strukturen. Dies sind beispielsweise komplexe Objekte, Typen und Klassen sowie Vererbung. Das grundlegende Konstrukt bleibt aber weiterhin die Relation oder Tabelle.
Dieser Abschnitt orientiert sich in erster Linie an den in [Ora02a] vorgestellten Konzepten. Nähere Erläuterungen und Beispiele zu den Themen Vererbung und Polymorphismus finden sich in [Ora01].

[1] Bereits der Vorgänger Oracle8 (siehe beispielsweise [CHRS98] und dort insbesondere Kapitel 13) unterstützt objektorientierte Konzepte. Damit ist er die erste Version von Oracle, die als objektrelationale Datenbank gelten kann. Allerdings ist die Modellierung von Typvererbung, Substituierbarkeit und Polymorphismus dort noch nicht möglich.

Objektorientiertheit impliziert in erster Linie die Beschreibung von Systemen als eine Ansammlung von Objekten. Diese besitzen Eigenschaften und Methoden. Eigenschaften repräsentieren die Struktur der Objekte, sie werden durch Attribute dargestellt. Methoden bieten die Möglichkeit, das Verhalten von Objekten zu definieren bzw. Operationen auf den Attributen eines Objektes durchzuführen. Dies lässt sich zum Modellieren komplexer Zusammenhänge der realen Welt nutzen. Dabei werden gleichartige Objekte zu Klassen zusammengefasst, wobei eine Klasse die Beschreibung eines Objekttyps repräsentiert.

Im Folgenden wird näher auf die objektrelationalen Eigenschaften eingegangen, die Oracle9i zur Verfügung stellt. Ein kurzer Überblick über Konzepte wie Objekttypen, Kollektionstypen und Vererbung soll als Grundlage für die spätere Umsetzung eines Speicherungsmodells für XML-Dokumente dienen.

3.1.1 Objekttypen

Ein Objekttyp ist ein nutzerdefinierter, zusammengesetzter Datentyp. Er besteht im Wesentlichen aus zwei Komponenten:

- Attribute beschreiben die Eigenschaften eines Objektes. Wertebereiche können dabei sowohl built-in Datentypen wie VARCHAR2 und NUMBER sein als auch wiederum Objekttypen.
- Objekte können Methoden besitzen, die Operationen auf den Attributen eines Objektes durchführen. Da die Attribute eines Objekttyps grundsätzlich **public** sind, ein Zugriff also ohne Einschränkung möglich ist, sind Methoden optional. Allerdings kann durch eine Methode ein spezielles Verhalten eines Objektes nachgebildet werden. Eine Anwendung muss eine solche Operation nicht selbst implementieren, sondern kann die entsprechende Methode des Objektes aufrufen. Ändert sich das Verhalten, muss nur die Methode selbst, nicht aber jede Anwendung, die die Methode nutzt, geändert werden.

Bei der Definition eines Objekttyps werden Spezifikation und *Body* unterschieden. Erstere bildet die Schnittstelle für Applikationen. Innerhalb der Spezifikation werden eine Menge von Attributen und Methoden deklariert. Folgendes Beispiel zeigt die Definition eines Objekttyps person_type, dessen Instanzen Personen darstellen.

```
CREATE OR REPLACE TYPE person_type AS OBJECT (
   vorname    VARCHAR2(20),
   nachname   VARCHAR2(20),
   alter      NUMBER,
   adresse    adress_typ,
   MEMBER FUNCTION get_name RETURN VARCHAR2)
NOT FINAL;
```

Eine Person besitzt einen Vornamen, einen Nachnamen, ein Alter und eine Adresse, wobei die Adresse an anderer Stelle wiederum als Objekttyp definiert worden ist. Der Zusatz NOT FINAL am Ende der Definition gibt an, dass von diesem Objekttyp weitere Typen abgeleitet werden können.
Ein Verweis auf einen anderen Objekttyp kann aber nur dann erfolgen, wenn dieser Typ bereits existiert, d.h. er muss im Vorfeld zumindest definiert worden sein. Das Erzeugen von Platzhaltern (unvollständigen Typen), die später ersetzt werden können, ist möglich. Definition und Deklaration sind somit voneinander trennbar.
Im obigen Beispiel wurde außerdem eine Funktion definiert, die den Namen der Person liefert. Innerhalb der Spezifikation erfolgt nur die Angabe der sogenannten CALL SPEC der Methode. Eine derartige *Call Specification* gibt der Datenbank einen sicheren Einstiegspunkt für die Ausführung der genannten Methode an und definiert Ein- und Ausgabeparameter.
Funktionen bzw. Prozeduren (Methoden ohne Rückgabewerte) werden nicht innerhalb der Spezifikation implementiert. Ihre Implementierung erfolgt im *Body*, dem zweiten Teil des Objekttyps.

```
CREATE OR REPLACE TYPE BODY person_type AS
   MEMBER FUNCTION get_name RETURN VARCHAR2 IS
   BEGIN
   ...
   END;
END;
```

Für die Implementierung der Methoden eines Objekttyps innerhalb des *Body* stellt Oracle PL/SQL zur Verfügung, eine prozedurale Erweiterung zur Einbettung von SQL. Informationen dazu findet man in [Ora02b].
Es besteht aber auch die Möglichkeit, die Methoden in Java zu schreiben, sie zu kompilieren und anschließend in die Datenbank zu laden. In diesem Fall wird der *Body* bei der Definition eines Objekttyps weggelassen. Es erfolgt nur die Angabe der CALL SPEC innerhalb der Spezifikation. Das Laden der Java-Klassen in die Datenbank startet man von der Kommandozeile aus mittels

```
loadjava -user nutzername/passwort Beispieldatei.class.
```

Als MEMBER deklarierte Methoden benötigen für die Ausführung eine Instanz des Objekttyps, d.h. sie können von einer Anwendung nur in Verbindung mit einem spezifizierten Objekt aufgerufen werden und ermöglichen dann den Zugriff auf die Daten des betreffenden Objektes.
Im Gegensatz dazu gibt es auch STATIC Methoden, die keine Instanz des Objekttyps erfordern und mit deren Hilfe allgemeine Operationen den gesamten Objekttyp betreffend ausgeführt werden können.

Objekttypen können auf verschiedene Arten eingesetzt werden. Wie im Beispiel beschrieben, kann man sie innerhalb der Deklaration eines anderen zusammengesetzten Datentyps verwenden, um komplexe Eigenschaften darzustellen, die der betreffende zusammengesetzte Typ besitzt.
Analog dazu besteht auch die Möglichkeit, den Objekttyp beim Erzeugen einer Tabelle als Attributbereich eines Attributes unter anderen einfachen oder ebenfalls komplexen Attributen anzugeben. Ein solches Objekt wird als column object bezeichnet.

```
CREATE TABLE familie_table (
  familienname VARCHAR2(20) PRIMARY KEY,
  vater        person_type,
  mutter       person_type,
  ...);
```

Zusätzlich ist es möglich, Objekttabellen zu erzeugen. Das sind Relationen, in denen jede Zeile (Tupel) ein Objekt repräsentiert (row object). Der Tupeltyp entspricht dann direkt dem Objekttyp. Eine Objekttabelle kann somit als einspaltige Tabelle aufgefasst werden. Der Zugriff auf die einzelnen Spalten (Attribute des Objekttyps) ist aber ebenfalls möglich.

```
CREATE TABLE person_table OF person_type (
  nachname PRIMARY KEY)
OBJECT IDENTIFIER IS PRIMARY KEY;
```

Jede Instanz eines Objekttyps besitzt eine eindeutige Objekt-ID (OID). Diese kann vom System generiert werden. Das ist die Default-Einstellung, wenn keine Angabe gemacht wird. Es kann auch mittels OBJECT IDENTIFIER IS SYSTEM GENERATED explizit vereinbart werden. Andererseits ist, wie das Beispiel zeigt, die Nutzung des Primärschlüssels für die Erzeugung der OID möglich. Dies ist unter Umständen günstiger, da die vom System vergebene ID 16 Byte groß ist[2].

[2]Bei der Deklaration der Objekttypen und -tabellen für die XML-Speicherung hat sich allerdings herausgestellt, dass Objekte, bei denen der Primärschlüssel als OID benutzt wird, nicht referenziert werden können. Referenzen werden im weiteren Verlauf noch vorgestellt und sind wesentlicher Bestandteil der Umsetzung des Speichermodells.

3.1.2 Kollektionstypen

In Oracle9i gibt es zwei Kollektionstypen zum Modellieren von 1:n-Beziehungen zwischen Objekten bzw. mehrwertigen Attributen. Will man beispielsweise in einer relationen Datenbank ausdrücken, dass eine Person mehrere Telefonnummern besitzt, muss man diese Information in einer separaten Tabelle speichern, um Daten (in diesem Fall alle anderen Attribute einer Person) nicht redundant abzulegen. Die Möglichkeit einer Tabelle in einer Tabelle, womit eine derartige Beziehung modelliert werden kann, gibt es bei relationalen Datenbanken nicht.
Oracle bietet hierfür Arrays und geschachtelte Tabellen.

- **VARRAY**[3]
  Ein Array ist eine geordnete Sammlung von Elementen desselben Typs. Auf jedes einzelne Element kann über seine Position direkt zugegriffen werden.
  Die Definition eines VARRAY erfordert die Angabe, wie viele Elemente maximal aufgenommen werden sollen. Die Anzahl bestimmt die Größe des Arrays. Ein Array wird in Oracle9i als abgeschlossenes Objekt abgespeichert (z.B. als *Binary Large Object* - BLOB).

- **Nested Table**
  Eine geschachtelte Tabelle kann ebenfalls eine beliebige Anzahl von Elementen desselben Typs aufnehmen. Allerdings ist die Menge der Elemente nicht geordnet. Wie bei "normalen" Tabellen hat man bei einer Nested Table die Möglichkeit, SQL-Operationen (Insert, Delete, Select, ...) auszuführen. Die Elemente eines Nested Table-Attributes werden intern in einer separaten Tabelle abgespeichert.

Kollektionstypen werden auf folgende Weise definiert.

```
CREATE TYPE hobbies AS VARRAY(10) OF VARCHAR2(20);
```

Durch diese Anweisung wird ein Listentyp erzeugt. Ein Objekt dieses Typs kann bis zu zehn Elemente aufnehmen, jedes Element eine Zeichenkette von maximal 20 Zeichen Länge.

```
CREATE TYPE adress_liste AS TABLE OF adress_typ;
```

Derart definierte Objekte repräsentieren geschachtelte Tabellen. Eine Instanz dieses Typs kann beliebig viele Adressen aufnehmen.

[3] steht für Variable Array, da die Anzahl der Elemente variabel sein kann

Das Erzeugen von Kollektionstypen, deren Elemente selbst wieder Kollektionstypen sind, ist möglich.

Wie Objekttypen können auch Kollektionstypen für verschiedene Zwecke eingesetzt werden, beispielsweise als Datentyp für die Spalten von relationalen Tabellen oder als Attribut eines Objekttyps.
Wann sollte man nun eine geschachtelte Tabelle und wann ein VARRAY nutzen? Diese Frage lässt sich anhand der Eigenschaften der beiden genannten Kollektionstypen beantworten. Ist eine Ordnung innerhalb der Elemente einer Kollektion erforderlich, sollte man sich für die Liste entscheiden. Ein VARRAY ist sehr effizient, wenn es darum geht, die Sammlung als Ganzes zu manipulieren und es nicht nötig ist, Anfragen über einzelne Elemente zu stellen. Außerdem kann ein VARRAY genutzt werden, wenn die Anzahl der Einträge bereits vorher feststeht bzw. ein oberes Limit für die Größe der Liste bekannt ist. Dies hat eine effizientere Speichernutzung zur Folge.
Demgegenüber sollte man die geschachtelte Tabelle wählen, wenn eine Reihenfolge der Elemente keine Rolle spielt oder man bei der Definition des Typs noch nicht bestimmen kann, wie groß die Anzahl der aufzunehmenden Elemente sein wird. Sollen effiziente Anfragen auf einzelne Teile der Kollektion möglich sein, muss man sich ebenfalls für eine Nested Table entscheiden.

3.1.3 Referenzen

Zum Modellieren von Beziehungen zwischen Objekten können Referenzen verwendet werden. Eine Referenz ist ein logischer Zeiger auf ein row object und bietet eine einfache Möglichkeit zum Navigieren zwischen Objekten. Der Vorteil eines derartigen Konzepts liegt auf der Hand. Statt den eigentlichen Objekten werden Verweise auf diese abgespeichert. Damit vermeidet man das Ablegen von redundanten Informationen. Ein Objekt mit all seinen Informationen ist wirklich nur einmal vorhanden. Es kann aber beliebig oft von anderen Stellen aus auf dieses Objekt verwiesen werden.
Das folgende Beispiel zeigt eine Erweiterung des Typs person_type um Vater und Mutter.

```
ALTER TYPE person_type
   ADD ATTRIBUTE (
   vater      REF person_type,
   mutter     REF person_type)
CASCADE;
```

Innerhalb einer Objekttabelle dieses Typs kann dann ein Tupel Referenzen auf andere Personen-Tupel (bzw. -objekte) beinhalten. Der Zugriff auf die Informationen der referenzierten Objekte erfolgt mittels der Punkt-Notation.

Im weiteren Verlauf dieser Arbeit wird es eine Reihe von Beispielen geben, die dies veranschaulichen (siehe u.a. Unterabschnitt 5.3.1). Deshalb soll an dieser Stelle darauf verzichtet werden.
Oracle stellt zum Testen, ob eine Referenz auf ein Objekt *dangling* (deutsch: "baumelnd") ist, das Prädikat IS DANGLING zur Verfügung. Damit ist es möglich, ein Objekt zu identifizieren, auf das nicht mehr zugegriffen werden kann, weil es beispielsweise gelöscht wurde oder sich die Zugriffsrechte geändert haben.

3.1.4 Vererbung

Eines der wichtigsten Konzepte der Objektorientierung ist die Vererbung. Durch die Typvererbung in Oracle9i können IS-A-Beziehungen dargestellt, d.h. Spezialisierungen von Objekten ausgedrückt werden.

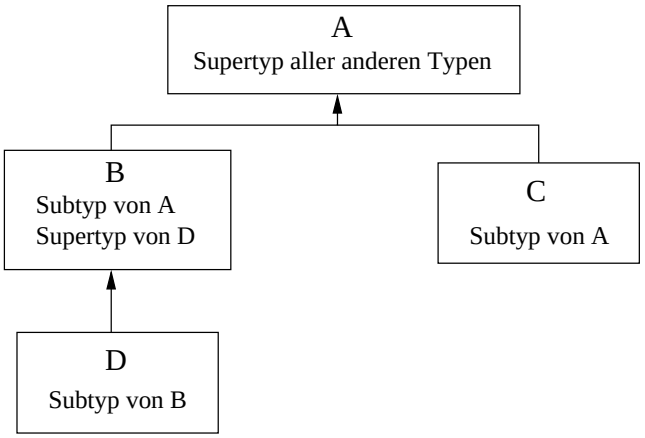


Abbildung 3.1: Typhierarchie bei Objekttypen

Von einem definierten Typ (dann auch Supertyp genannt) werden weitere Typen (Subtypen) abgeleitet. Die abgeleiteten Typen erben alle Attribute und Methoden des Supertyps. Dabei werden spätere Änderungen innerhalb des Supertyps ebenfalls berücksichtigt. Zusätzlich zu den geerbten Eigenschaften können bei den abgeleiteten Typen neue Attribute definiert und weitere Methoden hinzugefügt werden. Auch die Neudefinition ererbter Methoden ist möglich.
Eine Methode kann so unterschiedliche Formen annehmen. Wird beispielsweise die Implementierung einer geerbten Methode modifiziert, muss beim Aufruf der Methode für ein Objekt geprüft werden, von welchem Typ (Sub- oder Supertyp) das betreffende Objekt ist. Danach wird entschieden, welche der verschiedenen Implementierungen im jeweiligen Fall verwendet wird.

Damit kann ein unterschiedliches Verhalten von Objekten bezüglich ein und desselben Kontextes ausgedrückt werden. Dies wird als Polymorphismus bezeichnet.

Bei der Erzeugung eines Objekttyps durch Ableitung von einem anderen Typ wird die Phrase AS OBJECT weggelassen. Dafür verwendet man als Schlüsselwort UNDER innerhalb der Anweisung zum Erzeugen des Typs. Folgendes SQL-Statement erzeugt einen Subtyp zum Typ Person.

```
CREATE TYPE student_type UNDER person_type (
  matrikel_nummer NUMBER);
```

Damit ist ein Student eine spezielle Person, d.h. er erbt alle Eigenschaften und Methoden, die eine Person auch hat, besitzt aber zusätzlich noch ein Attribut "matrikel_nummer". Instanzen vom Typ Person haben dieses Attribut nicht. Neue Attribute dürfen dabei keinen Namen erhalten, der bereits für ein Attribut des Supertyps bzw. dessen Supertyps usw. verwendet wurde.

FINAL/NOT FINAL

Um das Ableiten von einem Objekttyp zu ermöglichen, muss bei der Deklaration das Schlüsselwort NOT FINAL verwendet werden. Der Default-Wert ist FINAL. Ein späteres Ändern ist möglich, allerdings darf ein Typ noch keine Subtypen besitzen, wenn sein Wert auf FINAL gesetzt werden soll.
Gleiches kann verwendet werden, falls die Methode eines Supertyps nicht vom abgeleiteten Typ überschrieben werden soll. Per Default ist das Verändern der Implementierung aber erlaubt.

INSTANTIABLE/NOT INSTANTIABLE

Wird ein Objekttyp als NOT INSTANTIABLE deklariert, gilt er als abstrakter Typ. Von einem derart deklarierten Typen darf es keine Instanz geben, d.h. es kann kein Objekt erzeugt werden. Der Typ wird in diesem Fall ausschließlich als Ausgangspunkt für das Ableiten von spezialisierten Untertypen verwendet.
Dasselbe gilt für Methoden. Damit kann die Implementierung einer Methode im jeweiligen Untertyp erzwungen werden. Im Supertyp wird dann nur ein Platzhalter angegeben. Falls jeder abgeleitete Typ die Methode ohnehin überschreibt, macht eine Implementierung an dieser Stelle keinen Sinn.
Falls die Methode eines Typs als NOT INSTANTIABLE deklariert ist, muss dasselbe für den Typen selbst gelten.

overloading vs. overriding

Im Gegensatz zu Attributen können für einen abgeleiteten Typ Methoden definiert werden, die den selben Namen wie geerbte Methoden haben. Besitzt ein Subtyp mehrere Methoden vom selben Namen, wird dies als overloading bezeichnet. In derartigen Fällen wird in Abhängigkeit von der Signatur, beispielsweise anhand der Parameter, entschieden, welche der Methoden aufgerufen wird.
Wird eine vererbte Methode neu definiert, um ihr Verhalten anzupassen, sozusagen zu spezialisieren, heißt das overriding. In diesem Fall muss es explizit mittels

```
OVERRIDE MEMBER FUNCTION get_name();
```

angegeben werden. Ruft eine Instanz des Subtyps diese Methode auf, wird die überschriebene Version ausgeführt.

Abschließend zum Thema Vererbung soll hier noch die Möglichkeit genannt werden, Objektinstanzen eines Subtyps dort zu verwenden, wo eine Instanz des Supertyps erwartet wird[4]. Dies gilt für alle Einsatzbereiche von Objekttypen. Soll ein Attribut eine Referenz auf einen Objekttyp enthalten, kann auch auf die Instanzen von Subtypen verwiesen werden. Ein Attribut kann nicht nur Objekte eines bestimmten Typs aufnehmen, sondern auch Objekte all seiner Subtypen. Besonders wichtig für die Umsetzung des in Kapitel 4 beschriebenen Speicherungsmodells für XML-Dokumente ist, dass eine Objekttabelle von einem Objekttyp auch Instanzen der Subtypen dieses Typs aufnehmen kann. Insbesondere ist es dabei gleichgültig, ob die Subtypen zusätzliche Attribute besitzen.

3.1.5 Wichtige Operatoren

Zum Abschluss des Abschnitts über die objektrelationalen Möglichkeiten, die das Datenbanksystem Oracle9i bietet, sollen hier noch SQL-Operatoren und -Funktionen aufgeführt werden, die für die Umsetzung bzw. Nutzung bestimmter Konzepte erforderlich sind.

- **REF**
  Die REF-Funktion liefert einen Zeiger auf das Objekt eines bestimmten Objekttyps und all seiner abgeleiteten Untertypen.

[4]Umgekehrtes gilt nicht! Wird eine Instanz des Supertyps eingesetzt, wo ein Objekt des Subtyps erwartet wird, erfolgt eine Fehlermeldung.

- **DEREF**
  Diese Funktion liefert bezüglich eines REF die Instanz des entsprechenden Objektes.

- **VALUE**
  Diese Funktion erwartet als Parameter einen Tabellen-Alias und liefert Instanzen des betreffenden Objekttyps und all seiner abgeleiteten Subtypen, die in der entsprechenden Tabelle gespeichert sind, zurück.

- **TREAT**
  Mittels der TREAT-Funktion wird dafür gesorgt, dass eine Instanz eines Supertyps wie eine Instanz eines davon abgeleiteten Subtyps behandelt wird. Damit wird der Zugriff auf Attribute und Methoden, die nur der Subtyp besitzt, ermöglicht. Ist eine Person beispielsweise ein Student, wird die Person als Student zurückgegeben. Ansonsten liefert TREAT den Wert NULL.

- **IS OF**
  Dieses Prädikat kann zum Testen benutzt werden, ob ein Objekt von einem bestimmten Objekttyp bzw. einem von diesem Typ abgeleiteten Subtyp ist.

- **ONLY**
  Mit Hilfe dieses Operators kann eine Auswahl auf einen bestimmten Objekttyp beschränkt werden. Auch alle Subtypen des entsprechenden Typs sind ausgeschlossen.

3.2 XML-Speicherung in Oracle9i - XMLType

In diesem Abschnitt soll eine in Oracle9i angebotene Lösung zur Abspeicherung von XML-Daten betrachtet werden. Seit Oracle9i Release 1 (9.0.1) gibt es innerhalb des DBMS einen neuen *built-in*-Datentyp zur Aufnahme von XML-Dokumenten: XMLType.
Instanzen dieses Objekttyps repräsentieren XML-Dokumente. Genau wie bei den Objekttypen (siehe Unterabschnitt 3.1.1) ist es möglich, Tabellen von diesem Typ zu erzeugen. Eine derartige Tabelle besitzt dann eine Spalte, jedes Tupel der Relation stellt ein XML-Dokument dar. Ebenso besteht aber auch die Möglichkeit, XMLType als Wertebereich für einzelne Tabellenspalten zu nutzen.
XMLType-Daten können auf zwei verschiedene Arten in einer Oracle-Datenbank gehalten werden. Existiert für ein abzuspeicherndes Dokument keine

XML-Schema-Beschreibung, wird es in jedem Fall als CLOB in der Datenbank abgelegt[5]. Andererseits können die Dokumente auch "aufgebrochen" und strukturiert in objektrelationalen Tabellen abgespeichert werden. Diese Methode wird in Unterabschnitt 3.2.1 noch ausführlich erläutert. Die Kombination beider Techniken wird ebenfalls angeboten. In diesem Fall wird ein Teil der Daten strukturiert, der Rest des Dokumentes als CLOB abgelegt. Diese Form der Speicherung nennt man *hybrid*.
Abbildung 3.2 verdeutlicht die verschiedenen Methoden. In Unterabschnitt 3.2.3 werden Vor- und Nachteile der beiden Methoden genannt.

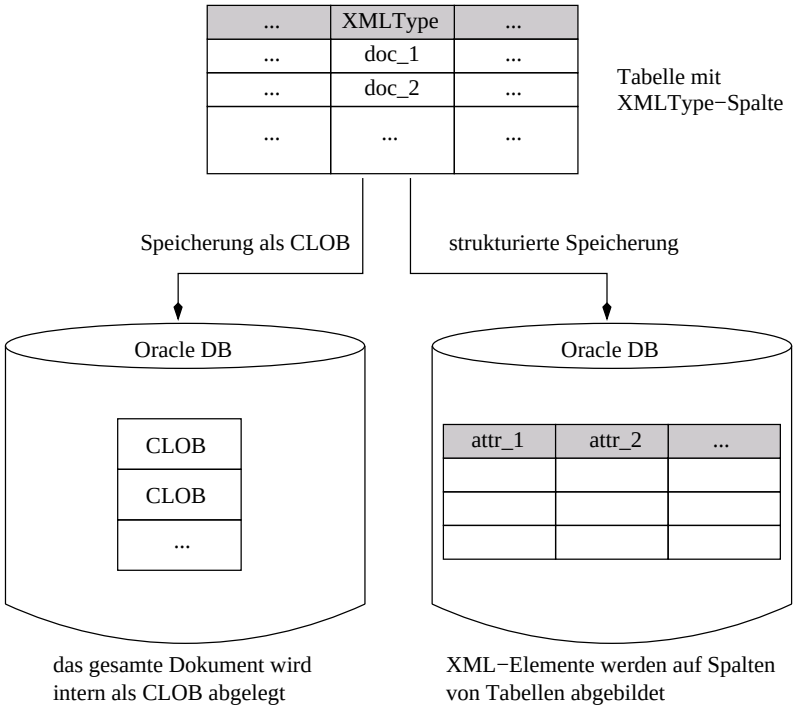


Abbildung 3.2: Speicherung von XML-Daten in Oracle9i

3.2.1 Objektrelationales Mapping

Ein XML-Schema wird dazu benutzt, für eine Klasse von XML-Dokumenten die Struktur der Dokumente zu vereinbaren. Eine Schema-Beschreibung enthält Informationen über Elemente und Attribute, die in einem Dokument enthalten sein dürfen. Es wird festgelegt, welche Subelemente ein XML-Element besitzt, zusätzlich dazu deren Anzahl und Reihenfolge bestimmt. Mithilfe

[5] Auch bei vorhandenem Schema kann man diese Art der Speicherung wählen.

eines Schemas wird definiert, ob ein Element leer ist oder Text beinhaltet, welche Datentypen Elemente und Attribute besitzen und ob es Defaultwerte oder unveränderbare Werte gibt.
Existiert eine solche Schema-Beschreibung für Dokumente, die in der Datenbank gespeichert werden sollen, müssen diese nicht als CLOB abgelegt werden. Anhand des XML-Schemas ist dann eine strukturierte Speicherung möglich.
Ein vorhandenes Schema muss der Datenbank bekannt gemacht werden. Dazu nutzt man die Methode *registerSchema()*. Beim Registrieren eines Schemas innerhalb der Datenbank geschieht Folgendes:

- Oracle erzeugt passende SQL-Objekttypen, die ein strukturiertes Speichern ermöglichen.
- Es werden *default*-XMLType-Tabellen für alle Wurzelelemente erzeugt.
- Optional werden Java-Klassen generiert, deren Methoden den Zugriff auf die im Schema deklarierten Elemente und Attribute optimieren sollen.

Ein XML-Schema wird in Oracle selbst als Instanz von XMLType abgelegt.

Die Objekttypen für eine strukturierte Speicherung werden auf folgende Weise erzeugt. Ein XML-Dokument besteht aus einer Sammlung von Elementen und Attributen. Elemente können entweder komplex sein, d.h. sie enthalten Subelemente und Attribute, oder sie haben skalare Werte.
Für jeden innerhalb des Schemas deklarierten komplexen Typ wird ein korrespondierender Objekttyp erzeugt. Dabei bildet Oracle jedes Subelement und jedes Attribut des komplexen Typs in ein Attribut des Objekttyps ab. Ist ein Subelement selbst wieder ein komplexer Typ, wird als Datentyp für das betreffende Attribut der entsprechende Objekttyp verwendet. Für Subelemente mit einfachem Typ und Attribute werden die passenden SQL-Datentypen eingesetzt. So wird zum Beispiel für die Typen *integer*, *double* und *decimal* der SQL-Typ NUMBER gewählt.
Subelemente, die laut Schema-Definition mehr als einmal auftauchen dürfen (maxOccurs > 1), werden als Kollektion abgebildet (zu Kollektionstypen siehe Unterabschnitt 3.1.2). Wenn nichts anderes angegeben wird, nutzt Oracle ein VARRAY zur Speicherung.

```
<complexType name="buch" xdb:SQLType="buch_t">
  <sequence>
     <element name="titel"   type="string"/>
     <element name="autor"   type="string"/>
```

```
    <element name="verlag"  type="string"/>
  </sequence>
  <attribute name="einband" type="string" minOccurs="0"/>
</sequence>
```

So könnte ein XML-Element <buch> definiert werden, das die Subelemente <titel>, <autor> und <verlag> und optional ein Attribut "einband" besitzt. Oracle würde daraus folgenden Objekttypen machen.

```
CREATE TYPE buch_t AS OBJECT (
  SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
  titel      VARCHAR2(4000),
  autor      VARCHAR2(4000),
  verlag     VARCHAR2(4000),
  einband    VARCHAR2(4000));
```

Wurde im Schema keine Angabe gemacht, wie der resultierende Objekttyp heißen soll (xdb:SQLType=...), verwendet Oracle vom System generierte Bezeichner.
Weitere Informationen wie Kommentare oder Namensraum-Deklarationen werden nicht abgebildet. Um trotzdem zu ermöglichen, dass ein rekonstruiertes Dokument, das strukturiert abgespeichert war, identisch zum Originaldokument ist, werden diese Daten gesondert abgelegt. Das im Beispiel vorhandene zusätzliche Attribut SYS_XDBPD$ enthält alle Informationen, die nicht mithilfe der anderen Attribute gespeichert werden können.

3.2.2 Verwendung von XMLType

Folgendes Beispiel generiert eine Tabelle, die eine Spalte vom Typ XMLType besitzt, und fügt das Beispieldokument ein.

```
CREATE TABLE xml_documents (
  doc_id    NUMBER,
  doc_name  VARCHAR2(100),
  xml_doc   XMLTYPE);

INSERT INTO xml_documents
VALUES(1,
       'buch_katalog.xml',
       XMLType(<?xml version="1.0"?>
               <buecher>
                 <buch einband="paperback">
                   <titel>The Lord Of The Rings</titel>
                   ...
               </buecher>));
```

Beim Einfügen eines Dokumentes wird überprüft, ob es sich um ein wohlgeformtes Dokument handelt. Ein XML-Dokument wird als wohlgeformt bezeichnet, wenn es den Regeln der XML-Spezifikation [W3C00] entspricht. Bei nicht korrekter Dokument-Syntax erhält man eine Fehlermeldung und das Einfügen wird abgebrochen.
Um auf dem Inhalt der abgespeicherten XML-Dokumente zu operieren, bietet XMLType eine Reihe von *built-in* MEMBER-Methoden. Mit ihnen ist es möglich, XML-Daten zu erzeugen und zu extrahieren. Alle Funktionen nutzen den standardmäßig im DBMS vorhandenen C-Parser zum Parsen bzw. Validieren. Um Informationen zu extrahieren, werden XPath-Pfadausrücke verwendet. XML-Anfragesprachen werden im nächsten Kapitel noch ausführlich behandelt. Zum momentanen Verständnis soll an dieser Stelle nur gesagt werden, dass mithilfe eines Pfadausdrucks die Navigation durch einen XML-Baum und die Auswahl einer Menge von Knoten des Baumes ermöglicht wird.
Tabelle 3.1 enthält eine Auswahl der vorhandenen Methoden. Eine vollständige Liste, die konkrete Syntax und ausführliche Beschreibungen der angebotenen Funktionen und Prozeduren können in [Ora02h] nachgelesen werden.

Methode	**Erklärung**
XMLType()	Dies ist der Default-Konstruktor. Mithilfe dieser Methode wird eine XMLType-Instanz erzeugt. Übergabeparameter kann ein CLOB bzw. eine Zeichenkette sein. Die Funktion wurde bereits innerhalb des Beispiels beim Einfügen des Dokumentes benutzt.
existsNode()	Diese Methode erwartet als Parameter ein XMLType-Objekt und einen Pfadausdruck. Sie liefert true, wenn die durch den Pfadausdruck spezifizierte Knotenmenge nicht leer ist.
extract()	Hiermit wird eine Menge von Knoten als Ergebnis eines Pfadausdruckes zurückgegeben. Eingabeparameter sind ein XMLType-Objekt und ein Pfadausdruck. Das Ergebnis ist ebenfalls eine Instanz des XMLType-Typs.
getStringVal()	Die Methode liefert den Wert einer XMLType-Instanz als Zeichenkette zurück. Analog dazu gibt es die Methoden *getClobVal()* und *getNumVal()*.
getRootElement()	Diese Funktion liefert das Wurzelelement einer Instanz. Falls es sich um ein XML-Fragment handelt, d.h. es gibt mehr als ein Root-Element, wird NULL zurückgegeben.

Tabelle 3.1: Auswahl von XMLType-Methoden

Die Verwendung der Methoden soll anhand eines Beispiels erläutert werden. Folgende SQL-Anweisung nutzt die Methode *extract()*, um den Titel des ersten <buch>-Knoten des Dokumentes zu erhalten. Mithilfe der Funktion *getStringVal()* wird der Knoten und sein Inhalt als Zeichenkette zurückgege-

ben.

```
SELECT a.xml_doc.extract('/buecher/buch[1]/titel').getStringVal()
FROM xml_documents;

<titel>The Lord Of The Rings</titel>
```

Das nächste Beispiel liefert alle Titel aller vorhandenen Bücher[6].

```
SELECT a.xml_doc.extract('/buecher/buch/titel').getStringVal()
FROM xml_documents;

<titel>The Lord Of The Rings</titel>
<titel>Datenbanken: Implementierungstechniken</titel>
<titel>The Hobbit</titel>
<titel>LaTeX - kurz & gut</titel>
```

Ein Ändern der vorhandenen XML-Daten ist ebenfalls möglich. Oracle9i stellt hierzu die Funktion *updateXML()* zur Verfügung. Damit können einzelne Knotenwerte geändert werden. Die Methode wird innerhalb einer SQL-UPDATE-Anweisung benutzt. Sie verlangt als Parameter eine Instanz, einen Pfadausdruck, mit dem der Knoten identifiziert wird, und den neuen Wert des zu ändernden Knotens.

```
UPDATE xml_documents a
SET a.xml_doc.updateXML(a.xml_doc,
                        '/buecher/buch/autor/text()',
                        'Tolkien');
```

Im Beispiel werden alle vorhandenen Autoren des Dokumentes in "Tolkien" abgeändert.
XML-Dokumente, die als CLOB in der Datenbank abgespeichert sind, können als Ganzes gelöscht werden. Dazu verwendet man die DELETE-Anweisung. Ob es eine Möglichkeit gibt, nur Teile eines Dokumentes zu entfernen, war aus der Oracle-Dokumentation nicht ersichtlich.

3.2.3 Strukturiert vs. Unstrukturiert

Was macht nun den Vorteil einer strukturierten Speicherung gegenüber der mittels eines CLOB aus?

[6] Dies sollte es zumindest tun. Allerdings liefert Oracle auf diese Anfrage nur einen Knoten zurück, nämlich den ersten passenden. Die restlichen Bücher sind jedoch grundsätzlich erreichbar, indem man innerhalb des Pfadausdrucks den Index für das jeweilige Buch verwendet: buch[n].

In erster Linie bringt das Aufspalten der Dokumente und das Abbilden der komplexen Typen in SQL-Objekttypen einen Performance-Gewinn mit sich. XPath-Ausdrücke als Übergabeparameter der XMLType-Methoden werden in SQL-Anweisungen umgesetzt, die direkt auf den einzelnen Objekten operieren. Im Gegensatz dazu muss ein Dokument, das als CLOB in der Datenbank liegt, für jede Operation, die darauf ausgeführt wird, erneut geparst und in eine DOM-Struktur überführt werden.
Jede Update-Operation auf einem als CLOB abgespeicherten Dokument erfordert ein komplettes Neuschreiben. Bei der strukturierten Speicherung kann auf die zu ändernden Elemente bzw. Attribute individuell zugegriffen werden, ohne, dass sich für den Rest des Dokumentes etwas ändert.
Zusätzlich wird der benötigte Speicherplatz geringer. Da ein XML-Schema vorhanden ist, in dem alle Elemente deklariert sind, ist es nicht nötig, die Tag-Namen explizit abzuspeichern. Es reicht aus, wenn der Inhalt der einzelnen Elemente in der Datenbank gehalten wird.
Dokumente können neben ihrer Wohlgeformtheit auch auf ihre Gültigkeit entsprechend einer Schema-Definition validiert, sprich überprüft werden.
Wenn es allerdings darum geht, Dokumente als Ganzes abzuspeichern und auch nur ganze Dokumente als Ergebnis einer Anfrage zu erhalten, ist die Speicherung als CLOB vorzuziehen. Es gibt keinerlei Datenverlust und die Kosten für die Rekonstruktion eines Dokumentes sind gering. Bei der strukturierten Speicherung dagegen gehen zum Beispiel *Whitespaces* verloren.

3.3 Weiterführende Literatur

Die Verbindung von Datenbanken und objektorientierten Konzepten und Strukturen wird detailliert in [Heu97] beschrieben.
Zum Thema *Java Stored Procedures* findet man umfangreiche Informationen in der Oracle-Dokumentation, beispielsweise in den Dokumenten [Ora02c], [Ora02d] und [Ora02e].
Eine vollständige Beschreibung der *Structured Query Language* (kurz: SQL), die dazu benötigt wird, die in einer Datenbank abgespeicherten Informationen zu verwalten, bietet [Ora02f]. Die Referenz orientiert sich bei den enthaltenen Ausführungen am SQL99 Standard [ISO99].
Eine ausführliche Beschreibung, wie XML-Daten in Oracle9i behandelt werden können, wird in [Ora02g] gegeben.
Wie eine XML-Schema Definition aussieht, wird in [W3C01c] anhand von Beispielen ausführlich erläutert.

Kapitel 4

Entwicklung eines Speichermodells

Ziel dieses Kapitels ist es, ein Speichermodell zu entwickeln, mit dem eine direkte Abbildung der hierarchischen Struktur von XML-Dokumenten auf objektrelationale Strukturen in Oracle9i möglich ist.
Aufgrund des im letzten Kapitel im Abschnitt 3.2 vorgestellten Datentyps zur Behandlung von XML-Daten in Oracle stellt sich natürlich die Frage, warum man ein derartiges Speichermodell überhaupt neu "erfinden" muss. Hauptsächliches Manko des vorhandenen Ansatzes ist es, dass für eine strukturierte Speicherung in jedem Fall ein XML-Schema vorhanden sein muss. Anderenfalls ist nur eine Speicherung als *Character Large Object* möglich. Außerdem würde, wie bereits in Kapitel 2 erwähnt, jede Änderung des Schemas ein Update der aus der XML-Schema-Beschreibung resultierenden Tabellen und Typen erfordern. Deshalb geht es in erster Linie darum, eine Struktur zu finden, die beliebige, von einem Schema unabhängige Dokumente aufnehmen kann.
Desweiteren soll das Modell in der Lage sein, auf Anfragen der XML-Anfragesprache XQuery zu reagieren. Diese enthält bezüglich den durch XPath beschriebenen Pfadausdrücken einige zusätzliche Eigenschaften wie z.B. Variablenbindungen.

Bei der Entwicklung einer Speicherungsstruktur für XML-Dokumente sollte man sich in erster Linie an den Informationen orientieren, die ein Dokument enthalten kann. Dazu kommen mögliche Strukturen, die es abzubilden gilt. Zusätzlich muss darauf geachtet werden, das zu entwickelnde Modell derart zu wählen, dass auf den abgespeicherten Daten effizient operiert werden kann. Es nützt nicht sehr viel, wenn die einmal in einer Datenbank abgelegten Informationen nur unter großem Aufwand abgefragt bzw. nicht mehr geän-

dert werden können. Wie bereits in der Einleitung erwähnt, soll diesbezüglich noch untersucht werden, ob sich Anfragen, die in der XML-Anfragesprache XQuery gestellt werden, auf die Strukturen des Modells abbilden lassen. Dies ist aber Inhalt eines späteren Kapitels.

4.1 Datenmodell - XML Information Set

Im Folgenden soll diskutiert werden, aus welchen Teilen sich ein XML-Dokument im Allgemeinen zusammensetzt und wie diese untereinander verbunden sind. Das XML Information Set bietet eine Grundlage für diese Herangehensweise. Welche dieser Informationen bei der Entwicklung des Datenbankschemas berücksichtigt werden, wird im Anschluss geklärt.

Das XML Information Set (kurz Infoset) definiert ein Datenmodell für XML-Dokumente. Es setzt sich laut Spezifikation [W3C01b] aus verschiedenen Informationseinheiten zusammen. Eine Informationseinheit ist dabei als abstrakte Beschreibung eines bestimmten Teils (beispielsweise eines Elementes) eines XML-Dokumentes zu sehen. Ein solches sogenanntes *Information Item* besitzt eine Menge von Eigenschaften, die den entsprechenden Teil des Dokumentes charakterisieren. Eine Eigenschaft kann ein einfacher Wert sein. Andererseits ist es aber auch möglich, dass ein Attribut einer weiteren Informationseinheit oder auch einer Menge von Informationseinheiten entspricht. Damit können strukturelle Beziehungen modelliert werden.
Um das Verständnis zu erleichtern, kann das Infoset auch als Baum aufgefasst werden. Die Knoten des Baumes werden dann durch die Informationseinheiten gebildet.
Innerhalb eines Information Sets sind bis zu elf verschiedene Typen von Informationseinheiten denkbar:

- Document Information Item,
- Element Information Item,
- Attribute Information Item,
- Processing Instruction Information Item,
- Unexpanded Entity Reference Information Item,
- Character Information Item,
- Comment Information Item,

- Document Type Declaration Information Item,
- Unparsed Entity Information Item,
- Notation Information Item und
- Namespace Information Item.

In Hinsicht auf die Entwicklung eines Speicherungsmodells für beliebige XML-Dokumente sollen mit dieser Arbeit nicht alle möglichen Teile eines Dokumentes umgesetzt werden. Die Diskussion beschränkt sich in erster Linie auf die wesentlichen Bestandteile wie Elemente, Attribute und Inhalt. Aufgrund dessen wird in diesem Abschnitt auch nur auf die entsprechenden Informationseinheiten näher eingegangen. Zusätzlich sind nicht alle Eigenschaften, die ein *Information Item* besitzt, wichtig für die Entwicklung einer Speicherungsstruktur. Ziel dieses Abschnitt ist es vielmehr, einen Satz von unabdingbaren Informationen festzustellen. Alle weiteren Informationseinheiten und die vollständige Menge der Eigenschaften können in der oben genannten Spezifikation nachgeschlagen werden.

4.1.1 Document Information Item

Die Wurzel des Infosets wird in jedem Fall durch genau ein *Document Information Item* gebildet. Es repräsentiert den Einstiegspunkt in ein XML-Dokument.
Wichtigste Eigenschaft ist das **[children]**-Attribut, dessen Wert eine Liste von Informationseinheiten ist. Diese Informationseinheiten sind die direkten Nachfahren des *Document Information Items* innerhalb des Baumes. Genau einer dieser Nachfahren ist ein *Element Information Item*. Dies entspricht dem *top-level*-Element eines XML-Dokumentes. Dieses Element wird auch als *document element* bezeichnet und ist gleichzeitig Wert der **[document element]**-Eigenschaft.

4.1.2 Element Information Item

Für jedes Element innerhalb eines XML-Dokumentes existiert ein *Element Information Item*. Das **[parent]**-Attribut enthält die Informationseinheit, die innerhalb des Baumes der Vater des betreffenden *Element Information Items* ist (für das *document element* ist dies z.B. das *Document Information Item*). Außerdem gibt es auch hier eine **[children]**-Eigenschaft, die alle direkten Nachkommen enthält. Diese Nachkommen können weitere Element-Informationseinheiten, Kommentar-Informationseinheiten, *Processing Instruction*-Informationseinheiten oder Informationseinheiten für Zeichendaten sein.

Weitere Eigenschaften beinhalten Daten zum Namensraum des jeweiligen Elementes, wie z.B. den Namen des *namespace* und das dazugehörende Präfix. Namensräume werden aber in dieser Arbeit nicht näher betrachtet.
Wichtig ist hingegen noch der **[local name]**, womit der lokale Teil des Elementnamens charakterisiert wird, und die **[attributes]**-Eigenschaft. Letztere enthält eine ungeordnete Zusammenstellung von Informationseinheiten für jedes Attribut, das das Element besitzt. (Bei Attributen spielt im Gegensatz zu den in **[children]** enthaltenen Nachkommen die Ordnung keine Rolle.) Besitzt ein Element keine Attribute, ist diese Liste leer.
Anhand eines Beispieldokumentes (siehe Anhang A.1) soll dieses Prinzip verdeutlicht werden:
Das Element-Tag <buecher> des Beispiels ist das *document element* des Dokumentes und als solches Wert des **[document element]**-Attributes des *Document Information Items*. Der **[local name]** ist "buecher", **[parent]** enthält das *Document Information Item*. Die Informationseinheiten zu den fünf <buch>-Elementen bilden gemeinsam die **[children]**-Liste. **[attributes]** ist leer, da das betreffende Element keine Attribute besitzt.

4.1.3 Attribute Information Item

Eine Informationseinheit für ein Attribut besitzt eine Reihe von Eigenschaften wie **[local name]**, **[namespace name]** und **[prefix]**, die genau wie die korrespondierenden Eigenschaften bei *Element Information Items* interpretiert werden. Zusätzlich gibt es ein **[normalized value]**, was den eigentlichen Wert des Attributes enthält und eine Eigenschaft **[attribute type]**, die den Typ charakterisiert, der für das betreffende Attribut deklariert worden ist.
Statt **[parent]** gibt es die Information **[owner element]**, deren Wert das *Element Information Item* ist, in dessen **[attributes]**-Liste das betreffende Attribut vorkommt.

Abbildung 4.1 verdeutlicht die Beziehungen zwischen Informationseinheiten für Elemente und Attribute.

4.1.4 Character Information Item

Um Inhalt von Elementen darzustellen, werden *Character Information Items* verwendet. Dabei wird jedes Zeichen als allein stehende Informationseinheit betrachtet. Im Wesentlichen besitzt eine Zeichen-Informationseinheit zwei Eigenschaften: den **[charakter code]**, dessen Wert der ISO-10646-Zeichencode

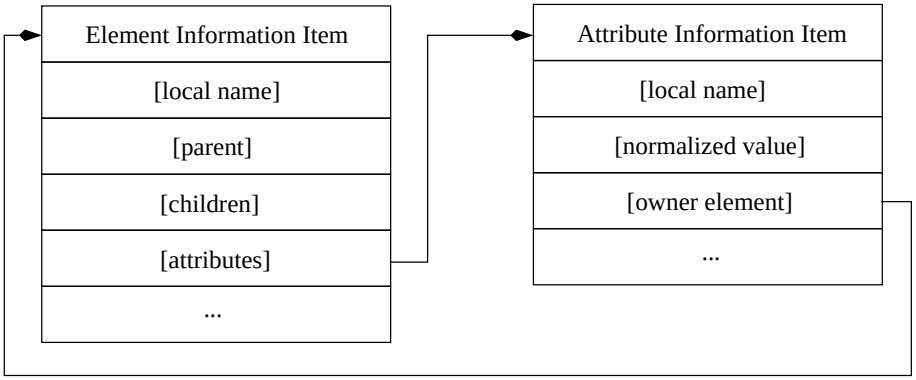


Abbildung 4.1: Element- und Attribut-Informationseinheit

des entsprechenden Zeichens ist[1], und **[parent]**. Letzteres Attribut enthält die Informationseinheit des Elementes, zu dessen Inhalt das Zeichen gehört.

4.2 Speichermodell

Ausgehend vom beschriebenen Datenmodell, können folgende Informationen als relevant für die Speicherung von XML-Dokumenten betrachtet werden:

- Ein Dokument kann als Menge von Knoten aufgefasst werden.
- Es gibt verschiedene Knoten innerhalb eines Dokumentes, nämlich Elementknoten, Attributknoten, Textknoten (einzelne Zeichen - siehe XML Infoset - werden als Zeichenkette aufgefasst), Kommentarknoten, *Processing Instructions* und *Namespace*-Knoten. Ein Dokument als Ganzes wird durch einen Dokumentknoten repräsentiert.
- Ein Dokumentknoten hat eine Menge von Kindern, nur eines dieser Kinder darf vom Typ Elementknoten sein.
- Elemente können Attribute besitzen, eine Reihe von geordneten Subelementen und/oder Text beinhalten bzw. auch leer sein.

Die einzelnen Knoten eines Dokumentes sollen auf Objekte einer Oracle-Datenbank abgebildet werden. Das bedeutet, die Eigenschaften eines Knotens werden Attribute des entsprechenden Objekttyps. Außerdem kann es Objekt-Methoden geben, die auf den Informationen der Knoten operieren.

[1]Der Wertebereich reicht von 0 bis #x10FFFF, wobei nicht jeder Wert ein erlaubter XML-Zeichencode ist. Der Zeichencode für den Buchstaben “a“ beträgt beispielsweise 0x0061.

4.2.1 Modellierung eines allgemeinen Knotentyps

Die verschiedenen Knotentypen unterscheiden sich durch ihre Eigenschaften. Es existieren aber auch Attribute, die jeder Knoten besitzt. Deshalb soll zuerst ein Objekttyp definiert werden, mit dem eine allgemeine Beschreibung von Knoten vorgenommen werden kann. Von diesem werden dann im Folgenden alle speziellen Knotentypen abgeleitet. Sie erben sozusagen die allgemeinen Eigenschaften und erhalten zusätzliche, nur für den entsprechenden Typ zutreffende Attribute und Methoden.
Tabelle 4.1 listet die Attribute auf, die für alle Knotentypen gelten.

Attribut	Beschreibung
node_id	Dies ist ein eindeutiger Identifikator. Damit ist es möglich, Knoten voneinander zu unterscheiden.
node_value	Mit diesem Attribut wird der Wert eines Knotens abgespeichert. Bei Elementknoten ist das beispielsweise der Tag-Name.
node_type	Um die einzelnen Knotentypen voneinander unterscheiden zu können, ist es nötig, die Information über den Typ explizit mit abzulegen. Eine Unterscheidung kann zwar innerhalb der Datenbank anhand des Objekttyps erfolgen. Erhält aber eine Anwendung eine Menge von Knoten als Ergebnis einer Anfrage, steht diese Möglichkeit nicht mehr zur Verfügung. Für das Zusammensetzen von XML-Fragmenten auf Client-Seite muß der Typ eines Knotens aber bekannt sein.
position	Ein Knoten kann Teil einer Menge von Kindknoten eines Elementes sein. Die Menge ist laut Datenmodell geordnet. Die Reihenfolge innerhalb der Subknoten wird durch dieses Attribut ausgedrückt.
parent_node	Dieses Attribut enthält einen Verweis auf das übergeordnete Element, also den Vaterknoten. Damit werden Verwandtschaftsbeziehungen, sprich die Kanten innerhalb eines Dokumentbaumes modelliert (siehe Unterabschnitt 4.2.6).
document	Falls mehrere Dokumente abgespeichert werden, muss es möglich sein, einen Knoten dem jeweiligen Dokument zuordnen zu können. Jeder Knoten besitzt deshalb einen Verweis auf ein Dokument.

Tabelle 4.1: Allgemeine Knoten-Attribute

Zum Objekttyp gehören außerdem eine Reihe von Objekt-Methoden, mit deren Hilfe auf den angegebenen Attributen operiert werden kann. Die Beschreibung einer prototypisch implementierten Auswahl dieser Methoden findet sich im Anhang.

Oracle bietet zum Generieren von eindeutigen, aufeinanderfolgenden Werten ein Datenbankobjekt SEQUENCE an. Derartig erzeugte Werte werden oft auch als Primärschlüssel einer Relation verwendet. Das Attribut node_id ist

vom Typ NUMBER. Der Zugriff auf Sequenz-Werte ist mithilfe zweier Funktionen möglich:

- **NEXTVAL** liefert den nächsten Wert und dient zu Beginn auch der Initialisierung,
- **CURRVAL** gewährt den Zugriff auf den aktuell vergebenen Wert.

Zur Darstellung von Verwandtschaftsbeziehungen werden Referenzen benutzt (siehe Unterabschnitt 3.1.3). Ohne diese Möglichkeit müsste mindestens die Knoten-ID zusätzlich abgespeichert werden, um zu zeigen, dass ein Knoten Subknoten eines anderen ist. Um Zugriff auf die restlichen Informationen des übergeordneten Knotens zu erhalten, wären dann Verbundoperationen unvermeidbar. Durch die Verwendung von Oracle-Referenzen können die Attribute des referenzierten Objektes mittels der Punkt-Notation direkt adressiert werden. Das Ablegen redundanter Informationen ist nicht nötig.
Vom definierten Typ kann eine Relation erzeugt werden, die der Abspeicherung aller möglichen Knotentypen dient. Werden alle Knoten eines Dokumentes in nur einer Tabelle abgelegt und durch Referenzen miteinander verknüpft, entfallen aufwendige Verbundoperationen über mehrere Relationen. Dadurch, dass Oracle das Einsetzen von Subtypen anstelle eines Supertyps erlaubt, verringert sich die Menge der abzuspeichernden Daten erheblich. In ein und derselben Tabelle können Knoten abgelegt werden, die weniger Attribute besitzen als andere, ohne dass bei ihnen die restlichen Eigenschaften mit NULL-Werten aufgefüllt werden müssen.

Im Folgenden sollen die verschiedenen Knotentypen und ihre entsprechenden Eigenschaften einzeln diskutiert werden. Von der Betrachtung ausgeschlossen werden hier Kommentare, *Processing Instructions* und *Namespaces*. Die Modellierung dieser Knotentypen kann aber analog erfolgen.
Die Definition der Objekttypen ist in Anhang C nachzulesen.

4.2.2 Modellierung von Dokumentknoten

In Hinsicht auf die Anfragebearbeitung ist es vorteilhaft, Dokumentknoten direkt mit abzuspeichern. Es gibt Pfadausdrücke, die als Ergebnis den Dokumentknoten liefern. Dieser kann auch Ausgangspunkt für weitere Anfragen sein.
Dokumentknoten erben alle Attribute des allgemeinen Knotentyps. Denkbar sind zusätzliche Attribute, wie zum Beispiel Informationen über ein vorhandenes Schema bzw. eine DTD. Dies würde die Auswahl von Dokumenten, die zu einer gemeinsamen Klasse gehören, ermöglichen. In Bezug auf das

hier vorgestellte Modell reichen aber die Attribute des allgemeinen Typs aus (siehe Tab. 4.2).

Attribut	Beschreibung
node_id	eindeutiger Identifikator, durch Sequenz erzeugt
node_value	Name des Dokumentes
node_type	doc
position	nicht nötig, bei allen Dokumenten konstanter Wert 1
parent_node	NULL
document	NULL

Tabelle 4.2: Attribute von Dokumentknoten

4.2.3 Modellierung von Attributknoten

Im XML Infoset werden Attribute als eigenständige Informationseinheit dargestellt. Ein Attribut gehört zu genau einem Element. Dabei kann ein Element mehrere Attribute besitzen. Attribute werden deshalb direkt als Eigenschaft eines Elementknotens abgespeichert. Diese Variante erspart das explizite Modellieren von Verwandtschaftsbeziehungen über Referenzen. Damit wird die Menge der zu speichernden Daten erheblich reduziert. Tabelle 4.3 listet die benötigten Informationen auf.

Attribut	Beschreibung
attr_name	Name des Attributes
attr_value	Wert des Attributes
attr_type	Typ des Attributwertes

Tabelle 4.3: Eigenschaften von Element-Attributen

Da ein Element mehrere Attribute besitzen kann, werden sie als Kollektion abgespeichert. Hierzu wird eine geschachtelte Tabelle verwendet. Die Reihenfolge spielt keine Rolle, die Anzahl der Einträge steht nicht fest und kann pro XML-Element variieren.

4.2.4 Modellierung von Textknoten

Das in Abschnitt 2.3 vorgstellte Speichermodell behandelt Text nicht als eigenständige Knotenart sondern als Inhalt von XML-Elementen. Das Problem dieses Ansatzes liegt in der Darstellung von *mixed content*. Ein Element, dessen Kinder andere Elemente und Zeichenketten sein können, ist auf diese Weise nicht umsetzbar.

Eine Lösung besteht darin, Text ebenfalls als Knoten zu modellieren, wie es auch im XML Infoset beschrieben wird. Genau wie ein Elementknoten besitzt auch ein Textknoten ein Attribut mit einem Verweis auf den darüberliegenden Vater.
Textknoten erben alle Attribute des allgemeinen Knotentyps. Zusätzliche Informationen sind nicht notwendig, da mittels der vorhandenen alle Eigenschaften eines Textknotens ausgedrückt werden können (siehe Tab. 4.4).

Attribut	**Beschreibung**
node_id	eindeutiger Identifikator, durch Sequenz erzeugt
node_value	Inhalt des übergeordneten Elementes
node_type	text
position	1, falls Element nur Inhalt besitzt; sonst wird durch diese Angabe die Position innerhalb der Subknoten des übergeordneten Elementes bestimmt (*mixed content*)
parent_node	Verweis auf das übergeordnete Element
document	Verweis auf das Dokument, zu dem der Textknoten gehört

Tabelle 4.4: Attribute von Textknoten

4.2.5 Modellierung von Elementen

Elemente können, zusätzlich zu den Eigenschaften des allgemeinen Knotentyps, eine ungeordnete Menge von Element-Attributen besitzen. Die Modellierung dieser Menge wurde bereits beschrieben. Tabelle 4.5 beschreibt die Eigenschaften von Elementknoten.

Die Möglichkeiten, die das Speichermodell bietet, sollen nun noch einmal zusammengefasst dargestellt werden.

- Das *main*-Element eines Dokumentes ist der Knoten der Relation, der im Attribut "parent_node" eine Referenz auf einen Dokumentknoten enthält.
- Alle anderen Elemente beinhalten an dieser Stelle einen Verweis auf den übergeordneten Elementknoten. Dies modelliert die Abbildung von Elementen, die 0, 1 oder mehr Subelemente besitzen.
- Eine Ordnung innerhalb der Subelemente wird durch das Attribut "position" gewährleistet.

Attribut	Beschreibung
node_id	eindeutiger Identifikator, durch Sequenz erzeugt
node_value	Tag-Name des Elementes
node_type	element
position	kennzeichnet in jedem Fall die Position innerhalb der Subknoten des übergeordneten Knotens
parent_node	Verweis auf das übergeordnete Element, beim *main*-Element verweist dieses Attribut auf den Dokumentknoten und ist somit identisch zum "document"-Attribut
document	Verweis auf das Dokument, zu dem der Elementknoten gehört
attr_list	enthält eine geschachtelte Tabelle zur Aufnahme der zum Element gehörenden XML-Attribute

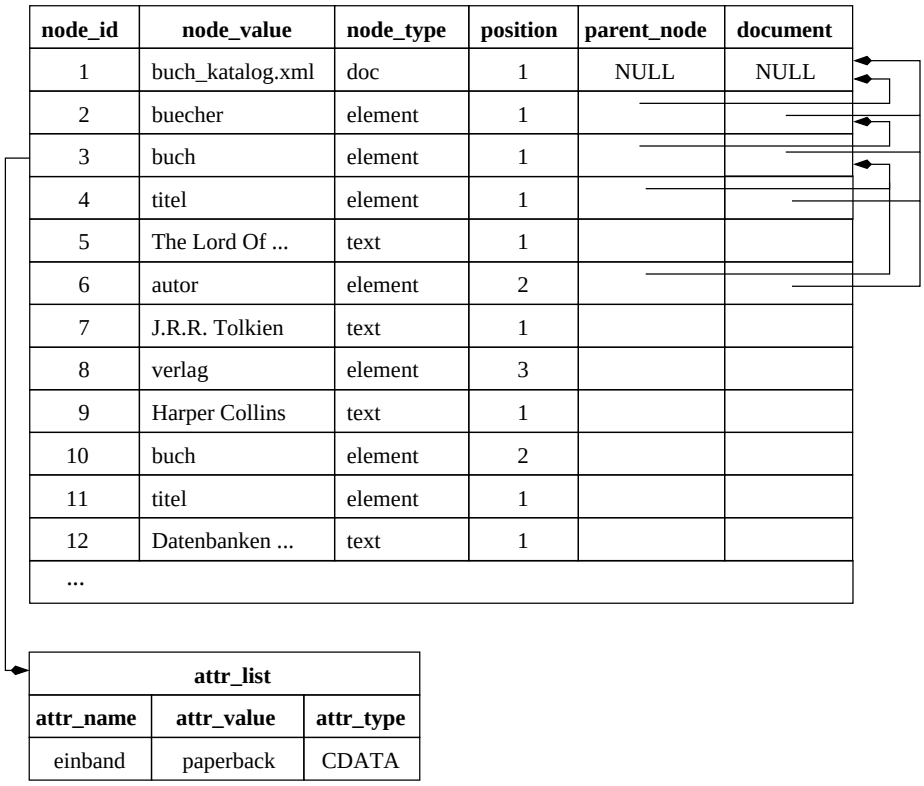
Tabelle 4.5: Eigenschaften von Elementknoten

- Da grundsätzlich jeder Knotentyp der Relation einen anderen Knoten als Vaterknoten referenzieren kann, ist die Darstellung von *mixed content* möglich. Beim Erzeugen einer solchen Referenz muss darauf geachtet werden, dass nur auf Elementknoten verwiesen wird!
- Ein Elementknoten kann 0, 1 oder mehr Attribute enthalten. Attribute werden in geschachtelten Tabellen abgespeichert. Diese Eigenschaft besitzen nur Elementknoten.
- Auf Elemente, die nur Text beinhalten, wird nur von einem Textknoten aus verwiesen.
- Die Darstellung von leeren Elementen ist ebenfalls möglich: kein Knoten verweist auf diesen Elementknoten.

4.2.6 Verwandtschaft zwischen Knoten

Ein wesentlicher Aspekt bei der Modellierung von Strukturen sind die Verwandtschaftsbeziehungen zwischen den einzelnen Knoten eines Dokumentes. Abbildung 4.2 zeigt die Verwendung von Referenzen zur Darstellung von Kanten zwischen zwei Knoten für das diskutierte Speichermodell.

Es gibt die verschiedensten Möglichkeiten, Beziehungen innerhalb eines Baumes auszudrücken. Das XML Infoset benutzt zwei davon. Ein Knoten kann eine Menge von Kindern haben und einen übergeordneten Knoten (den Vaterknoten) besitzen.
Zusätzlich ist das Ablegen von Informationen über die Geschwister eines Knotens realisierbar. Zu einem Knoten kann es sowohl einen Verweis auf den



node_id	node_value	node_type	position	parent_node	document
1	buch_katalog.xml	doc	1	NULL	NULL
2	buecher	element	1		
3	buch	element	1		
4	titel	element	1		
5	The Lord Of ...	text	1		
6	autor	element	2		
7	J.R.R. Tolkien	text	1		
8	verlag	element	3		
9	Harper Collins	text	1		
10	buch	element	2		
11	titel	element	1		
12	Datenbanken ...	text	1		
...					

attr_list		
attr_name	attr_value	attr_type
einband	paperback	CDATA

Abbildung 4.2: Relation mit Referenzen und geschachtelter Tabelle

linken Bruder als auch auf den rechten Bruder geben. Wird zum Beispiel bei der Darstellung von Subknoten statt einer Menge von Kindern nur eine Referenz auf das linkeste Kind verwendet, sind derartige Angaben unabdingbar. Natürlich steigt mit dem Grad der Vernetzung zwischen den einzelnen Knoten auch die Performance für Anfragen. Für die Abbildung einer bestimmten Anfrage kann die passendste Beziehung ausgewählt werden. Beispielsweise ist es günstiger, bei der Suche nach einem Geschwisterknoten direkt den Verweis auf den jeweiligen Bruder zu nutzen, als den Umweg über den Vater in Verbindung mit einer entsprechenden Positionsangabe verwenden zu müssen.
Je vernetzter ein solcher Baum ist, desto umständlicher gestalten sich aber auch Änderungsoperationen. Alle Referenzen auf verwandte Knoten müssen gewartet werden. Allein bei Rückverweisen, wie sie auch im XML Infoset vorhanden sind, leidet die Effizienz eines Updates. So muss zum Beispiel beim Löschen eines Knotens nicht nur der Knoten selbst angefasst werden, sondern alle Knoten, die Verweise auf diesen beinhalten.
Bei einem Speichermodell, das sowohl bei Anfragen als auch bei Änderungen effizient sein soll, ist es deshalb wichtig, einen Mittelweg zu finden. Dies

bedeutet letztendlich, dass nur eine der Vater-Kind-Beziehungen des XML Infosets im Modell verwendet wird - die Referenz auf den Vaterknoten. Für diese Wahl existieren mehrere Gründe.

Komplexität der Objekttypen

Eine Menge von Kindern lässt sich in Oracle9i ausdrücken, indem ein Knoten als Attribut eine Kollektion von Referenzen auf Knoten besitzt. Als Kollektionstyp würde man sicher eine geschachtelte Tabelle wählen, da die Position schon bei den einzelnen Knoten abgespeichert ist und der separate Zugriff auf jedes Element der Sammlung möglich sein soll. Ein einzelner Verweis, wenn nämlich vom jeweiligen Subknoten aus der Vater referenziert wird, lässt sich allerdings sehr viel einfacher verwalten. Zusätzlich verringert sich dadurch die Menge der zu speichernden Informationen.

Einfügen eines Dokumentes

Innerhalb eines Attributes kann nur auf ein Objekt verwiesen werden, dass es bereits in der Datenbank gibt. In der Regel wird ein Vaterknoten vor seinen Kindern eingefügt. Dies bedeutet, dass zusätzlich zum Insert eine weitere Operation nötig wird: ein Update der Kindreferenzen, nachdem auch die Subelemente eingefügt wurden.
Der Verweis auf einen Vaterknoten kann jedoch sofort angelegt werden, da der Knoten bereits existiert.

Pfadausdrücke

Das Abbilden eines Pfadausdrucks in eine SQL-Anfrage gestaltet sich über die Referenz auf den Vaterknoten sehr viel einfacher. Über die Punktnotation kann direkt auf Attribute des betreffenden Knotens zugegriffen werden. Eine SQL-Anweisung, die auf Elemente einer Kollektion zugreift, ist ungleich komplizierter.

Kapitel 5

Serverseitige Anfrageauswertung

Immer mehr Anwendungen und Informationsquellen nutzen XML als Datenformat zum Archivieren, Austauschen und Darstellen ihrer Daten. Damit erhöht sich zwangsläufig die Notwendigkeit einer Anfragesprache, mit deren Hilfe derart abgelegte Informationen effizient nach bestimmten Kriterien durchsucht und extrahiert werden können.
Im Mittelpunkt der folgenden Ausführungen steht die Anfragesprache XQuery als Quasi-Standard, wenn es darum geht, Daten aus XML-Dokumenten zu gewinnen. Ein kurzer Sprachüberblick ist Inhalt von Abschnitt 5.2.
Da Pfadausdrücke einen wesentlichen Bestandteil der Sprache XQuery ausmachen und diese in erster Linie auf den *Location Paths* in XPath 2.0 basieren, wird XPath in einem separaten Abschnitt behandelt.
Um auch auf das beschriebene Speichermodell mittels XQuery zugreifen zu können, muss eine Abbildung gefunden werden, die es ermöglicht, entsprechende XQuery-Anfragen in SQL-Anweisungen umzusetzen. Mit dieser Thematik beschäftigt sich Abschnitt 5.3.

5.1 Pfadausdrücke mittels XPath

XPath ist eine Sprachentwicklung des W3C und liegt mittlerweile als XPath 2.0 vor. Mittels XPath ist es möglich, Teile von XML-Dokumenten (Knoten bzw. ganze Teilbäume) zu adressieren. Knoten innerhalb eines Dokumentes können dabei durch XPath-Ausdrücke anhand ihrer Position, ihres Typs bzw. des Inhalts referenziert werden. Das Ergebnis eines Ausdruckes kann wieder Ausgangspunkt einer erneuten Anfrage sein.
Im Folgenden wird ein Überblick über die Funktionalitäten der Sprache gegeben. Dabei orientiert sich dieser Abschnitt hauptsächlich an [HM02]. Weitere Literaturangaben folgen am Ende dieses Kapitels.

XPath operiert auf der abstrakten logischen Struktur eines XML-Dokumentes. Dabei werden Pfad-Notationen ähnlich wie bei URLs zum Navigieren durch die hierarchische Struktur genutzt.
Das Datenmodell, das XPath (und auch XQuery) zugrunde liegt, basiert auf dem XML Information Set, das bereits in Abschnitt 4.1 beschrieben wurde. Ein Dokument wird als Baum von Knoten aufgefasst. Dabei können Knoten wiederum andere Knoten enthalten. Ein Wurzelknoten *root* enthält alle weiteren im Dokument vorkommenden Knoten. Aus Sicht von XPath existieren sieben verschiedene Knotenarten:

- der Wurzelknoten *root* (ist gleichbedeutend mit dem Dokumentknoten),
- Elementknoten,
- Textknoten,
- Attributknoten,
- Kommentarknoten,
- *Processing Instruction*-Knoten und
- *Namespace*-Knoten.

Der Wurzelknoten enthält das gesamte Dokument.

5.1.1 Location Paths

Der am häufigsten verwendete XPath-Ausdruck ist der *Location Path*. Dieser setzt sich aus einem oder mehreren Schritten, sogenannten *Location Steps* zusammen. Mit deren Hilfe können Knoten von XML-Dokumenten identifiziert und der Kontextknotenliste hinzugefügt werden. Die Kontextknotenliste eines Schrittes ist Ausgangspunkt für den nächsten *Location Step*. Jeder enthaltene Knoten wird dabei als Kontextknoten für die zu treffende Auswahl betrachtet. Die resultierende Ergebnismenge eines *Location Paths* kann leer sein oder einen bzw. mehrere Knoten enthalten.
Jeder Schritt eines Pfadausdrucks ist wie folgt definiert.

```
axisname::nodetest[predicate]
```

Die Achse beschreibt die Beziehung zwischen dem Kontextknoten und den Knoten, die durch den jeweiligen Schritt auszuwählen sind. Mittels dieser Angabe wird festgelegt, in welche Richtung der Baum ausgehend vom gerade

Achsenbezeichnung	Erklärung
child axis	betrifft die Kindknoten, also die direkten Nachkommen des Kontextknotens
parent axis	betrifft den direkten Vorgänger des aktuellen Knotens
self axis	damit ist der Kontextknoten selbst gemeint
attribute axis	ermöglicht den Zugriff auf die Attributknoten des Kontextknotens
descendant-or-self axis	betrifft alle Nachkommen und den Kontextknoten selbst
ancestor axis	betrifft alle Elementknoten, die den Kontextknoten enthalten, also den Vater des aktuellen Knotens, dessen Vorgänger usw.
following-sibling axis	damit sind alle Geschwisterknoten, die in Dokumentordnung nach dem Kontextknoten kommen, gemeint
preceding-sibling axis	selektiert alle Geschwisterknoten, die im Dokument vor dem Kontextknoten vorkommen
following axis	dies ermöglicht den Zugriff auf alle Knoten, die im Dokument nach dem aktuellen Knoten kommen
preceding axis	betrifft alle Knoten, die im Dokument vor dem Kontextknoten auftreten
descendant axis	greift auf alle Nachkommen (nicht nur die direkten) des aktuellen Knotens zu
ancestor-or-self axis	betrifft den Kontextknoten selbst und alle Elementknoten, die den Kontextknoten enthalten

Tabelle 5.1: Achsen in XPath-Pfadausdrücken

aktuellen Knoten überprüft wird. Tabelle 5.1 listet die in XPath möglichen Achsen und die dadurch auswählbaren Knoten auf.

Der Knotentest bestimmt, welche Knoten entlang der ausgewählten Achse eingesammelt werden. Welche Möglichkeiten es dabei gibt, wird später noch erläutert. Achse und Knotentest werden durch :: voneinander getrennt.
Die selektierte Knotenmenge kann man durch Prädikate weiter einschränken. An jeden Schritt eines Pfadausdrucks können Bedingungen geknüpft werden, anhand derer überprüft wird, ob ein ausgewählter Knoten zur Ergebnismenge gehören soll. Die Angabe erfolgt in eckigen Klammern.
Ein Prädikat besteht aus einem Boolschen Ausdruck, der für jeden Knoten der Kontextknotenliste getestet wird. Nur, wenn der Ausdruck true liefert, verbleibt der Knoten in der Liste. Innerhalb des Boolschen Ausdrucks können die gängigen relationalen Operatoren (=, !=, <, >, <= und >=) verwendet werden. Die Kombination mehrerer Prädikate ist über Boolsche Verknüpfung mittels and und or möglich. Ist ein angegebenes Prädikat kein Boolscher Ausdruck, erfolgt eine Konvertierung. Knotenmengen entsprechen true, wenn die Knotenmenge nicht leer ist, sonst false. Zeichenketten werden analog behan-

delt.

Die Verschmelzung von Achsenangabe und Knotentest führt zu einer verkürzten Form von *Location Steps*. Allerdings lassen sich damit nicht alle in Tabelle 5.1 beschriebenen Knotenmengen erreichen. Nur die Elemente entlang folgender Achsen können ausgewählt werden: *child*-Achse, *parent*-Achse, *self*-Achse, *attribute*-Achse und *descendant-or-self*-Achse.

5.1.2 Einfache Pfadausdrücke

Zur Angabe eines einfachen Pfadausdruckes, der nur aus einem *Location Step* besteht, existieren folgende Möglichkeiten.

- *Root Location Path*
  Die einfachste Variante eines Pfadausdrucks besteht in der Angabe von /. Damit wird der Wurzelknoten eines Dokumentes ausgewählt.

- *Child Element Location Step*
  Hierbei wird ein einzelner Elementname angegeben. Damit werden alle Elemente ausgewählt, die den spezifizierten Namen besitzen. Die Auswahl ist abhängig vom jeweiligen Kontextknoten. Existiert als direkter Nachkomme des aktuellen Knotens kein Element mit dem angegebenen Namen, ist die Ergebnismenge leer.

- *Attribute Location Step*
  Ein derartiger Schritt dient der Auswahl eines Attributknotens in Abhängigkeit vom Kontextknoten. Die Angabe erfolgt mithilfe des @-Zeichens gefolgt vom Namen des Attributes.

- *comment()*
  Diese Funktion dient der Auswahl von Kommentaren in Bezug auf den aktuellen Knoten. Jeder Kommentar wird dabei einzeln betrachtet.

- *text()*
  Diese Funktion wählt jeden Textknoten bezüglich des Kontextknotens aus. Als Textknoten wird dabei die gesamte Zeichenfolge bis zum nächsten auftretenden Element-Tag angesehen (kann also auch Entity Referenzen und CDATA-Abschnitte enthalten).

- *processing-instruction()*
  Ohne Parameter werden alle *Processing Instructions* (PI) des Kontextknotens selektiert. Anderenfalls werden nur die durch den Parameter spezifizierten PIs ausgewählt.

- *Wildcards*
  $*$ ermöglicht die gleichzeitige Auswahl verschiedener Elementknoten.
  node() ermöglicht die Auswahl aller Knotentypen ohne Einschränkung (also auch Text, Kommentare usw.).
  @$*$ ermöglicht den Zugriff auf alle Attributknoten in Abhängigkeit vom Kontextknoten.

5.1.3 Komplexe Pfadausdrücke

Es besteht die Möglichkeit, mehrere *Location Steps* durch | miteinander zu kombinieren. Damit wird die Angabe von Alternativen realisiert. Es können mehrere verschiedene Element- bzw. Attributtypen ausgewählt werden. Im Gegensatz zu der Nutzung von *Wildcards* werden aber nicht alle Knoten in die Ergebnismenge aufgenommen.
Ein *Location Path* entsteht, wenn mehrere *Location Steps* durch / miteinander kombiniert werden. Jeder Schritt wird dann in Abhängigkeit vom vorherigen ausgeführt. Beginnt ein Pfad mit /, dient der Wurzelknoten als Ausgangspunkt. Das folgende Beispiel zeigt einen zusammengesetzten XPath-Pfadausdruck und die resultierende Ergebnismenge.

```
/buecher/buch[@einband="paperback"]/titel

<titel>The Lord Of The Rings</titel>
<titel>The Hobbit</titel>
```

Soll sich die Auswahl auf alle Nachkommen des aktuellen Knotens beziehen, wird // verwendet. Der nächste Schritt wird dann nicht nur auf den direkten Nachfolger innerhalb des Baumes angewendet, sondern auf alle Knoten des unter dem Kontextknoten liegenden Teilbaumes. Beginnt der Pfadausdruck mit //, bezieht sich die Suche auf alle Nachfahren des Wurzelknotens.
Mittels . kann der aktuelle Knoten referenziert werden. Der direkte Vorgänger (Vater) des Kontextknotens wird durch .. ausgewählt.

5.1.4 Typen und Funktionen

Location Paths können nicht nur Mengen von Knoten (*node set*) identifizieren, sondern auch andere Typen von Rückgabewerten ergeben: *number*, *string* und *boolean*.

- **Number**
  Alle Zahlen bei XPath sind Fließkommazahlen. Zur Manipulation von

Zahlenwerten werden die gängigen Basisoperationen zur Verfügung gestellt: Addition, Substraktion, Multiplikation, Division div (da / bereits vergeben ist) und mod zur Ermittlung des ganzzahligen Rests.

- **String**
  Eine Zeichenkette ist geordnete Menge von Unicode-Zeichen. Eingeschlossen werden sie in doppelte bzw. einfache Hochkommata (je nachdem, welche Form eventuell bereits innerhalb der Zeichenkette verwendet wird).

- **Boolean**
  Mögliche Wahrheitswerte sind true und false. In der Regel wird einer dieser beiden Werte zurückgegeben, wenn zwei Objekte miteinander verglichen werden. Eine Kombination ist mittels and und or möglich. Verwendung finden Boolsche Werte in erster Linie bei Prädikaten.

XPath-Funktionen operieren auf den vier angesprochenen Typen. Sie können sowohl bei Prädikaten als auch in Pfadausdrücken verwendet werden.

- Funktionen können auf Knotenmengen operieren bzw. Informationen über Knotenmengen liefern. Ein Beispiel dafür ist die Funktion count(), die die Anzahl der Knoten der als Parameter übergebenen Knotenmenge zurückgibt.

- Funktionen können Informationen über Zeichenketten zurückgeben bzw. diese manipulieren. Ein Beispiel für eine String-Funktion ist contains(). Die Funktion erwartet zwei Argumente und liefert den Wert true, wenn das erste Argument das zweite enthält.

- Beispiel für eine Boolsche Funktion ist not(), die den Eingabeparameter negiert.

- Andere Funktionen können dazu verwendet werden, auf Zahlenwerten zu operieren. Die Funktion round() rundet beispielsweise ein Argument auf den am dichtesten liegenden ganzzahligen Wert.

5.2 XQuery

Dieser Abschnitt orientiert sich in erster Linie an der Spezifikation des W3C [W3C02e].
XQuery kann als Erweiterung von XPath 2.0 angesehen werden. Jeder Ausdruck, der sowohl in XPath 2.0 als auch in XQuery 1.0 syntaktisch korrekt

ist und erfolgreich ausgeführt werden kann, liefert in beiden Fällen dasselbe Ergebnis. Zusätzlich dazu existieren aber noch weitere Ausdrücke, die komplexere Anfragen ermöglichen.
Abgeleitet wurde XQuery von der Anfragesprache Quilt, die selbst eine Reihe von Eigenschaften anderer Sprachen in sich vereint, so u.a. von XPath 1.0 [W3C99], XQL [RLS], XML-QL [DFFLS], SQL [ISO99] und OQL [Cat96].

XQuery ist funktional, das bedeutet, jede Anfrage wird durch einen Ausdruck repräsentiert. Ausdrücke sind somit grundlegend für die Anfragesprache. XQuery ist darüber hinaus orthogonal. Das Ergebnis einer Anfrage kann wiederum als Eingabe für eine neue Anfrage verwendet werden.
XQuery-Ausdrücke können aus Schlüsselwörtern, Symbolen und Operanden konstruiert werden, wobei es möglich ist, als Operanden wiederum Ausdrücke zu verwenden. Der Wert eines Ausdrucks ist in jedem Fall eine Liste (auch als Sequenz bezeichnet) von null oder mehr atomaren Werten bzw. Knoten. XQuery operiert auf denselben sieben Knotentypen wie auch XPath (siehe Abschnitt 5.1). Dabei besitzt jeder Knoten eine eindeutige Identität. Aus einem Knoten können bei Bedarf die ihn betreffenden Informationen (wie Name, Wert usw.) extrahiert werden.
Im Folgenden werden die wichtigsten Ausdrücke vorgestellt.

5.2.1 Pfadausdrücke

Hier werden noch einmal die wesentlichen Informationen in Bezug auf Pfadausdrücke zusammengefasst.
Pfadausdrücke erlauben die Navigation durch ein Dokument entlang eines Pfades. Ein Pfad besteht dabei aus einer Sequenz von einem oder mehreren Schritten, die durch / voneinander getrennt werden. Ergebnis eines jeden Schrittes ist eine Knotenliste. Diese Knoten, geordnet in Abhängigkeit von der Dokumentreihenfolge, dienen als Ausgangspunkt für den nächsten Schritt. Zusätzlich ist die Angabe von Prädikaten zum Verfeinern der Ergebnismenge möglich. Ergebnis eines Pfadausdrucks ist in jedem Fall eine Sequenz von Knoten bzw. atomaren Werten.
Für Pfadausdrücke gibt es sowohl eine ausführliche als auch eine verkürzte Schreibweise. Das folgende Beispiel soll beides demonstrieren. Die angegebenen Ausdrücke haben dieselbe Bedeutung.

```
child::buch[attribute::einband="paperback"]

buch[@einband="paperback"]
```

Ausführliche Informationen zu Pfadausdrücken können in Abschnitt 5.1 nachgelesen werden.
Für XQuery gibt es bezüglich der verwendeten Achsen folgende Einschränkung. Es werden nur die *child*-Achse, die *descendant*-Achse, die *parent*-Achse, die *attribute*-Achse, die *self*-Achse und die *descendant-or-self*-Achse unterstützt.
Einen sehr wichtigen Zusatz stellen Variablenbindungen dar, die innerhalb von XQuery-Pfadausdrücken in verschiedenen Zusammenhängen vorkommen können. Beispielsweise kann ein Pfad eine Variable enthalten, an die in einem vorherigen Ausdruck eine Menge von Knoten bzw. atomaren Werten gebunden wurde. Dieses Prinzip wird später in Verbindung mit den sogenannten FLWR-Ausdrücken noch ausführlicher behandelt (siehe Unterabschnitt 5.2.3).

5.2.2 Konstruktoren

Konstrukoren können zur Erzeugung neuer XML-Elemente verwendet werden. Damit wird es möglich, Ergebnisse selbst wieder im XML-Format zu generieren. Konstruktoren können auch innerhalb anderer XQuery-Ausdrücke eingesetzt werden.
Man unterscheidet *element constructors* und *computed constructors*. Bei ersteren erfolgt die Angabe von Tag-Namen und Inhalt bzw. Werten (bei Attributen) durch Konstanten. Ein Element-Konstruktor entspricht dann gewöhnlicher XML-Notation. Andererseits können die Informationen auch Ergebnis von weiteren XQuery-Ausdrücken sein. In diesem Fall werden geschweifte Klammern {} verwendet.

```
<book>{ buch[@einband="paperback"] }</book>
```

Bei *computed constructors* werden die Schlüsselwörter "element", "attribute" bzw. "document" zum Erzeugen der jeweiligen Knotentypen verwendet. Folgendes Beispiel generiert ein neues XML-Element.

```
element buch { "Ein neues Buch!" }

<buch>Ein neues Buch!</buch>
```

5.2.3 FLWR-Ausdrücke

FLWR-Ausdrücke (gesprochen wie das englische "flower") werden analog zu Select-From-Where-Blöcken in SQL verwendet. Sie bilden das Grundgerüst

von XQuery-Ausdrücken. Mit ihrer Hilfe können Verknüpfungen (*Joins*) zwischen mehreren Dokumenten realisiert und Daten restrukturiert werden.
Die Kürzel stehen für FOR, LET, WHERE und RETURN.

FOR-Klausel

Die FOR-Klausel bindet eine oder mehrere Variablen an eine Liste von Werten, die durch einen XQuery-Ausdruck erzeugt wurde. In der Regel werden Pfadausdrücke verwendet. Über die gebundenen Werte wird dann iteriert. Ein Beispiel soll dies verdeutlichen.
Der Ausdruck

```
FOR $i IN (<one/>,<two/>,<three/>)
RETURN <ergebnis>$i</ergebnis>
```

erzeugt folgendes Resultat:

```
<ergebnis><one/></ergebnis>
<ergebnis><two/></ergebnis>
<ergebnis><three/></ergebnis>.
```

LET-Klausel

Die LET-Klausel bindet ebenfalls eine oder mehrere Variablen an eine Liste von Werten. Über diese Werte wird aber nicht iteriert.
Der Ausdruck

```
LET $i IN (<one/>,<two/>,<three/>)
RETURN <ergebnis>$i</ergebnis>
```

erzeugt das Resultat

```
<ergebnis><one/>
          <two/>
          <three/></ergebnis>.
```

In der Regel wird dies für mengenorientierte Funktionen wie count() und avg() benutzt.

WHERE-Klausel

Die WHERE-Klausel dient zum Filtern der durch die FOR- bzw. LET-Klausel spezifizierten Knotenmenge. Sie enthält ein oder mehrere Prädikate. Mithilfe von Vergleichsoperatoren wird die resultierende Knotenmenge eingeschränkt.

RETURN-Klausel

Das Anfrageergebnis wird dann in der RETURN-Klausel generiert. Sie enthält einen oder mehrere Element-Konstruktoren und/oder Referenzen auf Variablen. Sie wird für jeden Knoten, der vorher durch die FOR/LET/WHERE-Klauseln selektiert wurde, ausgeführt.

5.2.4 Operatoren und Funktionen

XQuery bietet die verschiedensten Operatoren.

- arithmetische Operatoren
  Es werden die gängigen arithmetischen Operationen Addition, Subtraktion, Multiplikation, Division und Ermittlung des ganzzahligen Rests unterstützt.

- Vergleichsoperatoren
  Der Vergleich einzelner Werte ist mittels eq möglich. Die gängigen relationalen Operatoren (=, !=, <, >, <= und >=) können ebenfalls verwendet werden. Mit ihrer Hilfe wird auch der Vergleich von Sequenzen von Werten realisiert.
  Knoten werden durch is bzw. isnot miteinander verglichen.

- logische Operatoren
  Zum Verknüpfen boolscher Ausdrücke werden and bzw. or verwendet.

Zusätzlich besitzt XQuery eine Reihe von *built-in*-Funktionen wie count(), max() und avg(). Nutzerdefinierte Funktionen sind ebenfalls möglich.
An dieser Stelle sollte außerdem noch eine sogenannte Input-Funktion genannt werden. Als Einstiegspunkt für die Suche innerhalb eines Dokumentes kann man document("String") verwenden. Argument ist der Verweis auf ein XML-Dokument. Die Funktion liefert den Wurzelknoten des betreffenden Dokumentes zurück.

5.2.5 Bedingungen

Bedingte Ausdrücke bei XQuery besitzen das von Programmiersprachen bekannte IF-THEN-ELSE-Format. Mit ihrer Hilfe kann die Struktur eines Ergebnisses anhand von Bedingungen beeinflusst werden. Ist der Testausdruck hinter IF gleich true, wird für das Ergebnis der Ausdruck nach dem THEN ausgewertet. Ansonsten wird der Ausdruck nach dem ELSE verwendet.

5.2.6 Quantoren

Es existieren zwei Quantoren in XQuery, der Existenzquantor SOME und der Allquantor EVERY. Quantoren können dazu benutzt werden, auf Mengen von Knoten Existenzbedingungen anzuwenden. Die Bedingung wird anhand eines Testausdrucks überprüft, der dem Schlüsselwort SATISFIES folgt.

```
EVERY $i IN document("buch_katalog.xml")/buecher/buch SATISFIES $i/titel
```

Der Ausdruck liefert true, wenn jedes Buch des Beispieldokumentes einen Titel besitzt.
Somit gilt: bei einem mit SOME quantifizierten Ausdruck muss es mindestens einen Knoten innerhalb der Knotenmenge geben, der dem Testausdruck genügt. Ist der Ausdruck mit EVERY quantifiziert, müssen alle Knoten der Knotenmenge dem Prädikat genügen, um als Resultat den Rückgabewert true zu erhalten.

5.2.7 Datentypen

Von XQuery werden sowohl Standard-Datentypen (das Typsystem basiert auf XML-Schema) als auch nutzerdefinierte Datentypen unterstützt.
In der Regel nutzt man Datentypen als Ein- und Ausgabeparameter bei der Deklaration von Funktionen. Ansonsten erfordern folgende Ausdrücke die Angabe eines Datentyps.

- INSTANCE OF
  Dieser Ausdruck liefert true, wenn der Wert des ersten Operanden zum angegebenen Typ (zweiter Operand) passt.

  ```
  5 INSTANCE OF xsd:integer
  ```

- TYPESWITCH
  Steht der Datentyp eines Eingabewertes erst zur Laufzeit fest, kann mittels TYPESWITCH in Abhängigkeit vom Typ ein bestimmter Ausdruck berechnet werden.

- CAST AS
  CAST AS dient zum Konvertieren eines Eingabewertes in einen bestimmten Datentyp. Dabei müssen allerdings die in [W3C02d] angegebenen möglichen Kombinationen von Eingabe- und Zieltypen berücksichtigt werden.

- TREAT AS
  Hiermit wird sichergestellt, dass ein Eingabewert den richtigen Datentyp besitzt. Im Gegensatz zu CAST AS wird der Typ dabei allerdings nicht verändert.

5.3 Anfragemethoden

Im Folgenden wird untersucht, ob das in Kapitel 4 vorgestellte Speichermodell geeignet ist, die Beantwortung von XQuery-Anfragen effizient zu unterstützen. Eine Implementierung des vollständigen Sprachumfangs von XQuery ist Inhalt der Arbeit von Guido Rost [Ros01]. Sie ist nicht Thema dieser Diplomarbeit. Relevant für die Anfragebearbeitung auf Seiten des Datenbank-Servers sind vielmehr die verwendeten Pfadausdrücke, mit denen Informationen aus XML-Dokumenten (und damit aus der Datenbank) gewonnen werden können. Alle anderen XQuery-Ausdrücke werden dazu verwendet, die extrahierten Informationen weiterzuverarbeiten. Abbildung 5.1 stellt die Interaktion zwischen einem Client und dem Datenbank-Server graphisch dar.

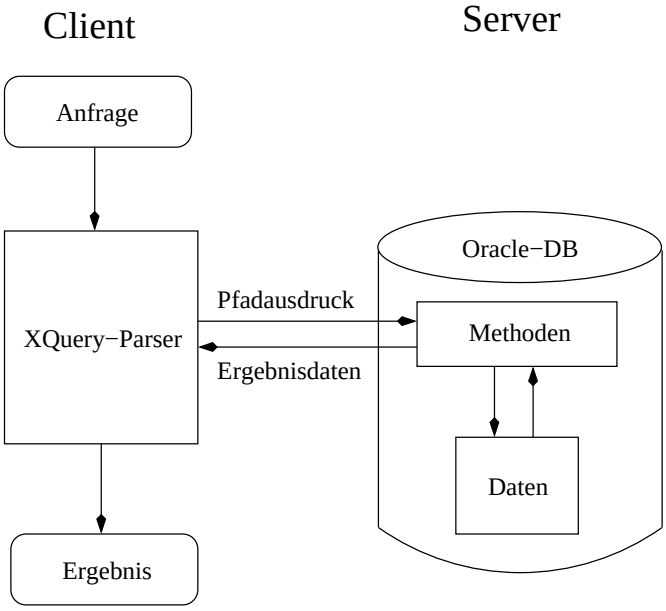


Abbildung 5.1: Datenaustausch bei XQuery-Anfragebearbeitung

Findet ein Anfrageprozessor innerhalb eines XQuery-Moduls einen Pfadausdruck, wird dieser zusammen mit eventuell bereits vorhandenen Variablenbindungen an den Datenbank-Server geschickt, indem die entsprechende Anfragemethode aufgerufen wird.

Damit unterscheidet sich die Anfragebearbeitung von dem in [Ros01] vorgestellten Verfahren. Dort werden Pfadausdrücke auf Client-Seite in die einzelnen *Location Steps* zerlegt. Jeder Schritt wird separat in eine SQL-Anweisung bezüglich des dort gewählten Speicherungsmodells (siehe Abschnitt 2.2) abgebildet. Die Anweisung wird an die Datenbank geschickt. Ergebnis ist eine Liste von Kontextknoten, die auf Seiten des Clients für die Bearbeitung des nächsten *Location Steps* verwendet wird.
Der Nachteil dieser Herangehensweise ist, dass im Allgemeinen sehr viele SQL-Anfragen erzeugt und an den Server zur Bearbeitung geschickt werden. Dies geht eindeutig zu Lasten der Performance.
Deshalb ist es günstiger, die Auswertung von Pfadausdrücken vollständig auf Seiten des Datenbank-Servers zu implementieren.

5.3.1 Beantwortung von Pfadausdrücken

Zur Beantwortung von Pfadausdrücken wird vom allgemeinen Knotentyp "node_type" die Objekt-Methode PROCESS_QUERY() zur Verfügung gestellt. Die Methode bildet einen *Location Path* in Abhängigkeit von den gewählten Speicherstrukturen (siehe Abschnitt 4.2) in eine SQL-Anweisung ab. Als Eingabeparameter wird in jedem Fall ein Pfadausdruck erwartet. Falls dieser eine Variable enthält, an die in einer vorhergehenden Anfrage bereits eine Menge von Knoten gebunden wurde, müssen der Name der Variablen und die Menge der Knoten ebenfalls mit übergeben werden.
Um die zu übermittelnde Datenmenge so gering wie möglich zu halten, werden als Ergebnis keine vollständigen Knotenobjekte an die aufrufende Client-Applikation gesendet. Als Antwort ist ein Identifikator, mit dem ein Knotenobjekt eindeutig selektiert werden kann, ausreichend. Wenn das Ergebnis beispielsweise an eine Variable gebunden werden soll, die Teil eines weiteren Pfadausdrucks ist, genügen die IDs. Alle weiteren Informationen werden weder vom Client noch vom Server benötigt. Der Client übergibt das Ergebnis ohne Änderung an die Datenbank-Methode. Diese kann bei Bedarf die entsprechende Information anhand der ID aus der Datenbank extrahieren.
Für die vorliegende Arbeit werden Pfadausdrücke in abgekürzter Notation betrachtet, d.h. die Achsenangabe innerhalb eines *Location Steps* fehlt.
Die Bearbeitung eines Pfadausdrucks erfolgt von hinten nach vorn, beginnend mit dem letzten *Location Step*. Dies ist der Schritt, durch den letztendlich die Ergebnisknoten des gesamten *Location Paths* identifiziert werden. Die Abbildung eines Pfadausdrucks in eine SQL-Anweisung entspricht dann der Auswahl der "node_id" der Zielknoten.

```
SELECT a.node_id
FROM document_table a
```

Die WHERE-Klausel der SQL-Anweisung wird anhand des Pfadausdruckes zusammengesetzt. Für jeden *Location Step* wird die WHERE-Klausel um entsprechende Bedingungen erweitert. Dabei erfolgt die Angabe der Bedingungen immer in Abhängigkeit von den Zielknoten.
Jeder Knotentest resultiert in einem Vergleich mit dem Wert des SQL-Attributs "node_value" eines Knotenobjektes der Datenbank. Für jeden Schritt wird der Bedingung ein weiteres "parent_node" vorangestellt. Ausgehend von den Blattknoten navigiert man damit rückwärts durch den Baum. Die Bedingung der WHERE-Klausel, in die der (n-2)-te *Location Step* abgebildet wird, bezieht sich somit auf den Vaterknoten des Vaterknotens des Zielknotens.

```
n-ter     Schritt: node_value = "Elementname"
(n-1)-ter Schritt: parent_node.node_value = "Elementname"
(n-2)-ter Schritt: parent_node.parent_node.node_value = "Elementname"
...
```

Zur Umsetzung der document()-Funktion wird der String innerhalb der Klammern mit dem Wert des "node_value"-Attributs des Dokumentknotens verglichen. Enthält der Pfadausdruck *Wildcards*, wird für den aktuellen *Location Step* kein Vergleich mit "node_value" spezifiziert. Somit sind alle Knotenobjekte zutreffend.
Enthält der Knotentest des aktuellen Schrittes eine Variable, wird die WHERE-Klausel um folgende Bedingung erweitert.

```
node_id IN ("Menge von IDs")
```

Location Step-Prädikate lassen sich ebenfalls einfach umsetzen. Für den aktuellen Schritt wird eine weitere Bedingung hinzugefügt. Dazu muss die SQL-Anweisung geschachtelt werden. Mithilfe des inneren SELECT wird eine Menge von Knoten (bzw. deren IDs) ausgewählt, die dem Prädikat entsprechen.

```
node_id IN (SELECT node_id FROM ...))
```

Das folgende Beispiel soll das Prinzip des vorgestellten Verfahrens verdeutlichen.

```
document("buch_katalog.xml")[1]/buecher[2]/buch[3][@einband="paperback"][4]/titel[5]

SELECT a.node_id
FROM document_table a
WHERE

[5]  a.node_value = 'titel' AND
```

```
[3]  a.parent_node.node_value = 'buch' AND

[4]  a.parent_node.node_id IN (
              SELECT b.node_id
              FROM document_table b,
                   TABLE(TREAT(VALUE(b) AS element_type).attr_list) t
              WHERE t.attr_name = 'einband' AND
                    t.attr_value = 'paperback') AND

[2]  a.parent_node.parent_node.node_value = 'buecher' AND

[1]  a.parent_node.parent_node.parent_node.node_value = 'buch_katalog.xml'
```

Damit ist eine sehr einfache und intuitive Umsetzung von Pfadausdrücken in SQL möglich. Sind Prädikate vorhanden, steigt die Komplexität der SQL-Anweisung, insbesondere, da jeder *Location Step* eine Bedingung besitzen bzw. es auch mehrere Prädikate pro Schritt geben kann.
Der Einfachheit halber wird das Ergebnis eines Pfadausdrucks auf Seiten der Datenbank serialisiert. Dies bedeutet, dass die ermittelten Knoten-Identifikatoren durch Kommata voneinander getrennt als Zeichenkette abgelegt werden. Der String wird von der Methode als Rückgabewert an die Client-Applikation gesendet. Diese wird das Ergebnis in den meisten Fällen als Parameter für einen erneuten Funktionsaufruf verwenden. Die Knoten-IDs werden im Allgemeinen dazu genutzt, Knoten-Objekte zu identifizieren. Zu diesem Zweck sind sie Teil der WHERE-Klausel einer SQL-Anfrage, die im Verlauf des Prozesses der Anfragebearbeitung dynamisch generiert wird. Geht man davon aus, ist die Wahl des Typs "String" für das Ergebnis eines Pfadausdrucks sehr sinnvoll, da die SQL-Anweisung ebenfalls eine Zeichenkette ist und die Knoten-IDs ohne weitere Konvertierung hinzugefügt werden können. Selbst der Separator erweist sich dabei als hilfreich, da bei einer Anweisung

```
WHERE a.node_id IN (...)
```

die Angaben innerhalb der Klammern ohnehin durch Kommata getrennt werden müssen.
Grundsätzlich lassen sich alle möglichen Pfadausdrücke auf diese Weise modellieren. Die Ausnahme bilden *Location Paths*, die // enthalten.
Folgender Pfadausdruck ist noch auf einfache Weise umsetzbar.

```
document("buch_katalog.xml")//buch/autor
```

```
SELECT a.node_id
FROM document_table a
WHERE a.node_value = 'autor' AND
      a.parent_node.node_value = 'buch' AND
      a.document.node_value = 'buch_katalog.xml'
```

Befindet sich // nicht unmittelbar nach der document()-Funktion, lässt sich der betreffende *Location Path* nicht ohne weiteres abbilden. Dies liegt daran, dass jeder Knoten nur seinen unmittelbaren Vorgänger kennt.
Eine umständliche Möglichkeit der Realisierung ergibt sich, wenn die Schachtelungstiefe des XML-Dokumentes bekannt ist. Die Information kann dazu genutzt werden, alle möglichen Fälle innerhalb der WHERE-Klausel anzugeben (sie werden dann durch OR voneinander getrennt). Für eine maximale Baumtiefe von 5 ergibt sich dann folgende Anweisung.

```
document("buch_katalog.xml")/buecher//autor

SELECT a.node_id
FROM document_table a
WHERE a.node_value = 'autor' AND

      ((a.parent_node.node_value = 'buecher' AND
       a.parent_node^2.node_value = 'buch_katalog.xml') OR

      (a.parent_node^2.node_value = 'buecher' AND
       a.parent_node^3.node_value = 'buch_katalog.xml') OR

      ...

      (a.parent_node^4.node_value = 'buecher' AND
       a.parent_node^5.node_value = 'buch_katalog.xml'))
```

(Aus Gründen der Übersichtlichkeit ist folgende abkürzende Schreibweise gewählt worden: "parent_node^2" entspricht "parent_node.parent_node".)
Die WHERE-Klausel wird demnach so oft um die entsprechende Bedingung erweitert (jeweils mit zusätzlichem "parent_node"), wie die Schachtelungstiefe des Dokumentes beträgt.

5.3.2 Informationen zu den Knotenobjekten

Unter Umständen benötigt eine Anwendung außer der "node_id" weitere Informationen eines Knotens. Das ist zum Beispiel der Fall, wenn ein XML-Dokument vollständig oder in Teilen rekonstruiert werden soll. In der Datenbank werden diesbezüglich zusätzliche Methoden zur Verfügung gestellt (siehe Tab. 5.2), die durch die Anwendung aufgerufen werden können. Diese Methoden sind innerhalb der jeweiligen Knotentypen als STATIC definiert und erfordern deshalb beim Aufruf die Angabe des entsprechenden Objekttyps.
Oracle bietet zudem Techniken, vollständige Objekte an eine aufrufende Anwendung zu senden. Zu diesem Zweck werden die Objekttypen auf Java-Klassen abgebildet. Die benötigten Klassen kann man vorher definieren. An-

dererseits kann auch die bereits vorhandene STRUCT-Klasse verwendet werden. Die Abbildung von Referenzen, VARRAYS und geschachtelten Tabellen ist ebenfalls möglich. Für nähere diesbezügliche Informationen sei an dieser Stelle auf [Ora02e] und, speziell für die Generierung von Java-Klassen, auf [Ora02i] verwiesen.

Methode	Beschreibung
GET_VALUE()	liefert den Wert eines Knotenobjektes (Attribut "node_value")
GET_TYPE()	liefert den Typ eines Knotenobjektes (Attribut "node_type")
GET_POSITION()	liefert die Position eines Knotenobjektes (Attribut "position")
GET_PARENT_NODE()	liefert die ID des Vaterelementes eines Knotenobjektes (Zugriff auf dessen "node_id" über Attribut "parent_node")
GET_CHILDREN()	liefert die IDs der Subelemente eines Knotenobjektes, Methode kann nur für Element-Knoten aufgerufen werden
GET_ATTRIBUTES()	liefert die XML-Attribute eines Knotenobjektes, Methode kann nur für Element-Knoten aufgerufen werden
GET_DOCUMENT()	liefert das Dokument, zu dem ein Knotenobjekt gehört (Zugriff auf dessen "node_value" über Attribut "document")

Tabelle 5.2: Methoden für den Zugriff auf Objektinformationen

Die in Tabelle 5.2 aufgelisteten Methoden bieten dem gegenüber aber den Vorteil, dass Informationen zu den vorhandenen Knoten einzeln und nach Bedarf abrufbar sind. Dies reduziert wiederum die Menge der auszutauschenden Daten zwischen Client und Server. Zudem ist auch die Übertragung mehrerer bzw. aller Informationen denkbar, wie das folgende Beispiel zeigt. Damit ist die Verwendung von Objekt-Methoden flexibler und unter Umständen auch performanter als ein *Type Mapping*.

```
SELECT node_type.GET_VALUE("ID"),
       node_type.GET_POSITION("ID"),
       node_type.GET_PARENT("ID"),
       element_type.GET_CHILDREN("ID") FROM DUAL;
```

Einzig die Methode GET_ATTRIBUTES() sollte über *Type Mapping* realisiert werden, da die XML-Attribute eines Elementknotens ungeordnet in einer geschachtelten Tabelle abgelegt sind. Ein Auslesen Attribut für Attribut ist nicht sinnvoll, da man anhand eines erhaltenen Attributes nicht entscheiden

kann, welches als nächstes angefordert werden muss. In diesem Fall bleibt also nur das Senden eines kompletten "attr_ntab"-Objektes übrig.

5.4 Weiterführende Literatur

Als Einstieg in XPath kann das Web-Tutorial [W3School] verwendet werden. Tiefergehende Informationen erhält man in [W3C02a]. Dort findet sich die detaillierte Beschreibung der XPath-Grammatik.
Folgende Dokumente sind sowohl für XPath als auch für XQuery relevant. Das Datenmodell, und damit die Informationen, auf die ein Anfrage-Prozessor zugreifen kann, wird in [W3C02b] behandelt. In [W3C02c] findet man Informationen sowohl zur statischen als auch zur dynamischen Semantik. Eine Bibliothek von Funktionen und Operatoren, die von XPath und XQuery unterstützt werden, ist in [W3C02d] beschrieben.
Die XQuery-Spezifikation des W3C umfasst außer [W3C02e] noch weitere Dokumente, die für eine tiefere Recherche dienen können. In [W3C01a] werden Anforderungen aufgeführt, die von einer XML-Anfragesprache erfüllt werden sollten. Eine Reihe von Beispielanfragen, mit deren Hilfe der Umgang mit XQuery veranschaulicht werden kann, enthält [W3C02f]. Zu den skizzierten Beispielen, die auf einem Satz vordefinierter Daten operieren, sind auch die zu erwartenden Ergebnisse mit angegeben.
Außer der Umsetzung von Guido Rost [Ros01] können im Internet eine Reihe weiterer Implementierungen gefunden werden, wie Tabelle 5.3 zeigt.

Anbieter	Web-Adresse
X-Hive	http://www.x-hive.com/xquery/
Fatdog	http://www.fatdog.com
Kawa-XQuery	http://www.gnu.org/software/kawa/xquery/
Kweelt	http://db.cis.upenn.edu/Kweelt/
Lucent	http://db.bell-labs.com/galax/
Microsoft	http://131.107.228.20
Openlink Software	http://demo.openlinksw.com:8391/xquery/demo.vsp
QuiP (Software AG)	http://www.softwareag.com/developer/guip.default.htm
XQuench	http://sourceforge.net/projects/xquench/

Tabelle 5.3: XQuery-Implementierungen im Web

Die angegebenen Verweise sind [BW] entnommen. Dort findet man zusätzlich eine kurze Einführung zu XQuery.
Informationen zum Typsystem basierend auf XML-Schema können in den Dokumenten [W3C01c], [W3C01d] und [W3C01e] des W3C gefunden werden.

Kapitel 6

Update-Operationen

Die aktuelle Version der XML-Anfragesprache XQuery sieht keine Update-Operationen vor, mit denen Dokumente nachträglich geändert werden können.
Wie eine Erweiterung des aktuellen XQuery-Sprachumfangs um Manipulationsoperationen aussehen kann, ist in der Arbeit von Birger Hänsel [Hae02] beschrieben. Aufgrund des dort verwendeten Speicherungsmodells kann die vorgestellte Sprachabbildung hier nicht umgesetzt werden. Ausgehend von den aufgeführten Basisoperationen wird deshalb in diesem Abschnitt ein Satz von Methoden definiert, der auf Datenbankseite zum Zweck der Manipulation der abgelegten Daten bereitgestellt werden soll.
Folgende Grundoperationen können als Erweiterung der Semantik von XQuery aufgestellt werden:

- **Insert** zum Einfügen von Knoten,
- **Delete** zum Löschen einzelner Knoten bzw. ganzer Dokumentteile,
- **Rename** zum Ändern von Knotenwerten,
- **Replace** zum Ersetzen eines Knotens durch einen anderen und
- **Move** zum Verschieben von Knoten bzw. ganzer Dokumentteile.

Im Folgenden werden die oben aufgeführten Basisoperationen einer näheren Betrachtung bezüglich des gewählten Speicherungsmodells unterzogen.
Alle Abschnitte dieses Kapitels beschränken sich auf die in dieser Arbeit diskutierten Knotentypen. Die angegebenen Verfahren können aber ohne weiteres auf Kommentare, Processing Instructions und Namespace-Knoten angewendet werden.

6.1 Insert

Die Operation **Insert** dient dem nachträglichen Einfügen einzelner Knoten. In Bezug auf die in dieser Arbeit behandelten Knotentypen, muss zwischen dem Einfügen eines zusätzlichen Attributes zu einem vorhandenen Elementknoten, dem Einfügen eines Knotens als Blatt an einen vorhandenen Elementknoten und dem Einfügen eines Elementes innerhalb des Dokumentbaumes unterschieden werden. Außer beim Einfügen von Attributen ist eine Positionsangabe erforderlich, anhand derer die Position innerhalb der eventuell bereits vorhandenen Subelemente des Kontextknotens bestimmt wird.
Dabei wird der Kontextknoten (bzw. eine Menge von Kontextknoten, falls das einzufügende Objekt gleichzeitig an verschiedene Knoten angehängt werden soll) durch eine ID spezifiziert, die Ergebnis einer vorherigen Anfrage sein kann.

6.1.1 Einfügen eines Knotens als Blatt

Vom Knoten selbst sind als Informationen nur noch der Wert und der Typ erforderlich.
Wenn Subelemente vorhanden sind, müssen deren Positionsangaben angepasst werden. Alle Knoten, deren Position größer bzw. gleich der Position des einzufügenden Knotens ist, rücken um eine Stelle nach hinten (Abb. 6.1).

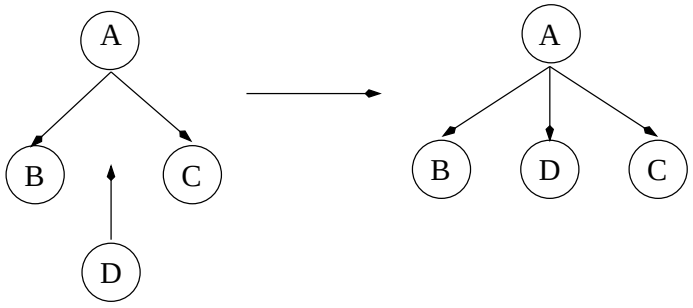


Abbildung 6.1: Einfügen eines Knotens als Blatt

Zur Sicherheit sollte hier noch überprüft werden, ob die angegebene Position sinnvoll ist und beim Einfügen des Knotens an dieser Stelle keine Lücken in der Ordnung der Subelemente entstehen. Eine Lösung besteht zum Beispiel darin, die Anzahl der vorhandenen Subelemente mittels MAX abzufragen, um zu testen, ob die angegebene Position kleiner oder gleich dem Wert MAX + 1 ist. Ist das nicht der Fall, wird der Knoten an der Stelle MAX + 1 eingefügt. Ansonsten wird das Anpassen der Positionen durch folgende Operation realisiert.

```
UPDATE document_table a
SET a.position = a.position + 1
WHERE a.parent_node.node_id = "ID" AND
      a.position >= "Position"
```

Danach kann der neue Knoten durch Aufruf des jeweiligen Typ-Konstruktors in die Relation eingefügt werden. Folgendes Beispiel fügt einen Textknoten ein.

```
INSERT INTO document_table
VALUES(text_type(node_seq.NEXTVAL,
                 "Knotenname",
                 'text',
                 "Position",
                 (SELECT REF(a)
                    FROM document_table a
                    WHERE a.node_id = "ID"),
                 (SELECT b.document
                    FROM document_table b
                    WHERE b.node_id = "ID")))
```

Die Funktion NEXTVAL liefert einen eindeutigen noch nicht vergebenen Wert für die ID des neuen Knotens. Für den Verweis auf das entsprechende Dokument wird das vom Vaterknoten referenzierte Dokument verwendet.

Besonderheit beim Einfügen von Elementknoten

Einige XML-Elemente besitzen Attribute. Diese werden erst nachträglich zu einem bestehenden Element hinzugefügt, damit nicht zwischen dem Insert von Elementen mit und dem von Elementen ohne Attribut unterschieden werden muss.
Oracle unterscheidet zwei Fälle von NULL-Werten. Zum einen kann ein Objekt NULL sein, zum anderen können alle Attribute des Objektes NULL-Werte besitzen. Nur im zweiten Fall ist ein nachträgliches Ändern der Attributwerte möglich. Dies bedeutet konkret, dass bei einem Element, das beim Einfügen (anstelle einer Liste von Attributen) den Wert NULL übergeben bekommt, später kein Attribut mehr hinzugefügt werden kann. Das betreffende Objekt "Nested Table" hat den Wert NULL. Deshalb wird der Konstruktor des Attribut-Typs aufgerufen und ein Dummy-Attribut eingefügt. Dieses wird gleich im Anschluss gelöscht. Die geschachtelte Tabelle ist dann aber in der Lage, nachträglich eine beliebige Anzahl von Tupeln aufzunehmen.
Zum Einfügen eines Elementes müssen also folgende Operationen durchgeführt werden.

```
INSERT INTO document_table
VALUES(element_type(node_seq.NEXTVAL,
                    "Knotenname",
                    "Knotentyp",
                    "Position",
                    (SELECT REF(a)
                       FROM document_table a
                       WHERE a.node_id = "ID"),
                    (SELECT b.document
                       FROM document_table b
                       WHERE b.node_id = "ID"),
                    attr_ntab(attr_type(NULL,NULL,NULL))))
```

Zum Löschen des Dummy-Attributs ist die ID des gerade eingefügten Knotens erforderlich. Diese erhält man, indem man den aktuellen Wert der für die IDs benutzten Sequenz erfragt. Da eine derartige Anfrage von Oracle nicht innerhalb einer "Where"-Klausel erlaubt wird, muss sie separat gestellt und das Ergebnis als Parameter beim anschließenden Delete verwendet werden.

```
SELECT node_seq.CURRVAL FROM DUAL

DELETE
FROM TABLE(SELECT TREAT(VALUE(a) AS element_type).attr_list
           FROM document_table a
           WHERE a.node_id = "ID")
```

Da die ID des gerade eingefügten Knotens die größte bisher vergebene ID ist, kann sie allerdings auch über die MAX-Funktion selektiert werden. In diesem Fall ist ein Schachteln der Anfragen möglich.
Auch hier wird das Dokument über das Attribut "document" des Vaters ermittelt. Dies funktioniert jedoch nicht, wenn es sich beim einzufügenden Knoten um das *top level*-Element des Dokumentes handelt. In diesem besonderen Fall beinhaltet das "document"-Attribut des Vaterknotens keinen Verweis, weil es sich um den Dokumentknoten selbst handelt. Bei einer Implementierung muss dies berücksichtigt werden. Um das Einfügen von Elementknoten allgemeingültig umzusetzen, sollte man den Namen des Dokumentes beim Aufruf der Methode übergeben. Die innere SELECT-Anweisung sieht dann wie folgt aus.

```
...
(SELECT REF(a) FROM document_table a WHERE a.node_value = "doc-name"),
...
```

6.1.2 Einfügen innerhalb des Dokumentbaumes

Ein Einfügen eines Knotens innerhalb eines Dokumentbaumes bedeutet zum einen das Auftrennen einer bestehenden Kante, das Einfügen des neuen Knotens an diese Position und das Anhängen des bisherigen Subelementes an den neu eingefügten Knoten. Eine zweite Möglichkeit besteht im Einfügen einer zusätzlichen Hierarchieebene in den Baum. Dabei wird der neue Knoten einziges Kind des übergeordneten Knotens. Alle bisherigen Subknoten werden an den neuen Knoten angehängt. Abbildung 6.2 veranschaulicht beide Varianten.
Diese Form des Insert ist nur für Elementknoten zulässig. Alle anderen Knotentypen dürfen keine Child-Elemente besitzen.

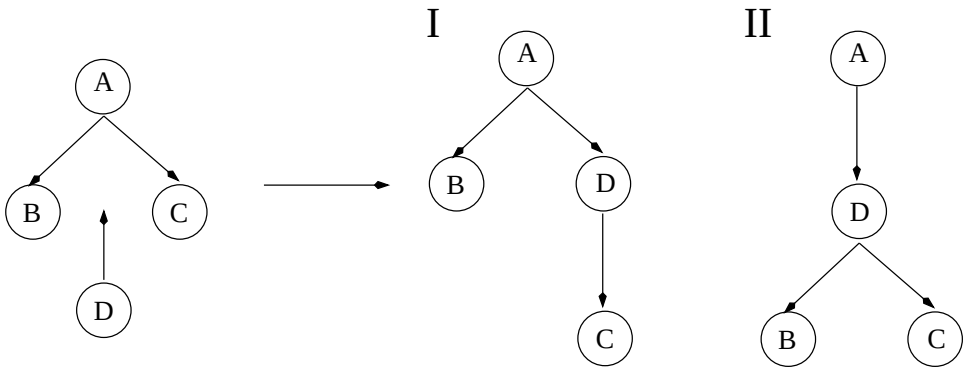


Abbildung 6.2: Einfügen eines Knotens innerhalb des Baumes

Verschieben eines Subelementes

Wird beim Aufruf der Methode eine Position übergeben, kann davon ausgegangen werden, dass der neue Knoten an einer bestimmten Stelle im Baum eingehängt werden soll. Nach dem Einfügen des neuen Elementes werden nur die Informationen des Knotens angepasst, der sich ursprünglich an dieser Stelle befunden hat (Abb. 6.2: I). Die Änderung beschränkt sich dabei auf das Anpassen der "parent_node"-Referenz und das Setzen der Position auf den Wert 1, da der Knoten nun einziger Subknoten des neu eingefügten Knotens ist.

```
UPDATE document_table a
SET a.position = 1,
    a.parent_node = (SELECT REF(b)
                     FROM document_table b
                     WHERE b.node_id = (SELECT MAX(c.node_id)
                                        FROM document_table c))
```

```
WHERE a.parent_node.node_id = "ID" AND
      a.position = "Position" AND
      NOT a.node_id = (SELECT MAX(d.node_id) FROM document_table d)
```

Verschieben aller Subelemente

Bei dieser Variante des Einfügens eines Knotens innerhalb eines Dokumentbaumes werden alle Subknoten des Kontextknotens um eine Ebene nach unten verschoben. Nach dem Einfügen des neuen Elementes, das an der Position 1 unterhalb des Kontextknotens eingehängt wird, ist nur das Anpassen der "parent_node"-Referenzen der ursprünglichen Subknoten erforderlich (Abb. 6.2: II). Die Ordnung innerhalb dieser Knoten bleibt gleich.

```
UPDATE document_table a
SET a.parent_node = (SELECT REF(b)
                     FROM document_table b
                     WHERE b.node_id = (SELECT MAX(c.node_id)
                                        FROM document_table c))
WHERE a.parent_node.node_id = "ID" AND
      NOT a.node_id = (SELECT MAX(d.node_id) FROM document_table d)
```

6.1.3 Einfügen eines Attributes

Zum Einfügen eines neuen Attributes sind folgende Parameter erforderlich: Eine Menge von Knoten-IDs identifiziert die Knoten, bei denen das neue Attribut eingefügt werden soll, und das Attribut selbst, bestehend aus Attributname, Attributwert und Typ des Attributwertes.
Für jeden Knoten der Knotenmenge wird dann die folgende Aktion durchgeführt.

```
INSERT
INTO TABLE(SELECT TREAT(VALUE(a) AS element_type).attr_list
           FROM document_table a
           WHERE a.node_id = "ID")
VALUES("Name", "Wert", "Typ")
```

6.2 Delete

Eine Methode **Delete** muss zum einen die Möglichkeit bieten, einzelne Knoten der abgespeicherten Dokumente zu entfernen. Zum anderen müssen auch ganze Teilbäume gelöscht werden können. Beide Ansätze sollen unabhängig voneinander behandelt werden. Zusätzlich sind Attribute aufgrund ihrer abweichenden Speicherung gesondert zu betrachten.

Zum Löschen einer Menge von Knoten wird die ID der Knoten benötigt. Daher sollte jedem Delete eine Anfrage vorausgehen, bei der mittels eines *Location Path* eine Menge von Knoten-IDs spezifiziert wird. Diese Menge ist Eingabeparameter der in Abhängigkeit von der geplanten Aktion aufzurufenden Methode.

6.2.1 Löschen einzelner Knoten

Handelt es sich beim zu löschenden Knoten um einen Blattknoten des Dokumentbaumes, kann der spezifizierte Knoten einfach entfernt werden, nachdem die Positionen seiner Geschwister angepasst worden sind. Im anderen Fall gestaltet sich das Löschen etwas aufwändiger, da eventuell vorhandene Subelemente an der Stelle des zu löschenden Knotens eingehängt werden müssen. Abbildung 6.3 soll diesen Ansatz verdeutlichen. Folgende Schritte realisieren das Löschen eines einzelnen Knotens innerhalb des Dokumentbaumes.

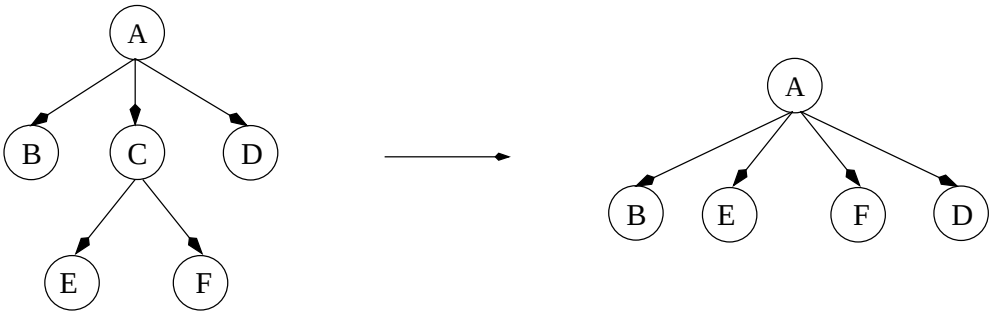


Abbildung 6.3: Löschen eines Knotens innerhalb des Baumes

Bei eventuell vorhandenen Subknoten müssen die Positionen derart geändert werden, dass der neue Wert des ersten Subknotens der Position des zu löschenden Knotens entspricht.

```
UPDATE document_table a
SET a.position = a.position - 1 + (SELECT b.position
                                   FROM document_table b
                                   WHERE b.node_id = "ID")
WHERE a.parent_node.node_id = "ID")
```

Alle Geschwister des zu löschenden Knotens, die in der Ordnung der Menge der Subelemente nach diesem Knoten kommen, werden um die Anzahl der Subelemente des Knotens nach hinten geschoben.

```
UPDATE document_table a
SET a.position = a.position - 1 + (SELECT COUNT(*)
                                   FROM document_table b
                                   WHERE b.parent_node.node_id = "ID")
WHERE a.parent_node.node_id = (SELECT c.parent_node.node_id
                               FROM document_table c
                               WHERE c.node_id = "ID") AND
      a.position > (SELECT d.position
                    FROM document_table d
                    WHERE d.node_id = "ID")
```

Die Subknoten des zu entfernenden Knotens werden nun innerhalb des Dokumentbaumes verschoben. Nach dieser Aktion nehmen sie die Stelle ihres Vaterelementes ein. Dazu müssen nur die im Attribut "parent_node" enthaltenen Referenzen angepasst werden.

```
UPDATE document_table a
SET a.parent_node = (SELECT REF(b)
                     FROM document_table b
                     WHERE b.node_id = (SELECT c.parent_node.node_id
                                        FROM document_table c
                                        WHERE c.node_id = "ID"))
WHERE a.parent_node.node_id = "ID"
```

Wurde bei Aufruf der Methode eine Menge von Knoten übergeben, müssen die bis hierhin beschriebenen Teilschritte für jeden Knoten einzeln ausgeführt werden. Danach können mit einer Operation alle Knoten gelöscht werden.

```
DELETE
FROM document_table a
WHERE a.node_id IN ("Menge von IDs")
```

Es stellt sich nun die Frage, in wie weit dies eine sinnvolle Operation ist. Gehört ein Textknoten grundsätzlich zum entsprechenden Elementknoten und wird deshalb in jedem Falle mit entfernt? Was passiert dann, wenn ein zu löschender Knoten sowohl Elemente als auch Text als Subknoten besitzt (mixed content)? In diesen Fällen ist das Löschen eines einzelnen Knotens eher unsinnig. Im Gegensatz dazu kann es aber schon Anwendungsfälle geben, in denen eine Hierarchieebene aus einem Dokumentbaum entfernt werden soll. Für ein Element, das nur Subelemente besitzt, wird also unter Umständen eine derartige Operation benötigt.

Etwas unkomplizierter gestaltet sich das Löschen ganzer Teilbäume, das im Folgenden beschrieben wird.

6.2.2 Löschen von Teilbäumen

Eine Methode zum Entfernen ganzer XML-Fragmente benötigt als Parameter den Wurzelknoten des zu löschenden Teilbaumens. Das eigentliche Löschen verläuft dann in drei Schritten.

Als erstes müssen für jeden zu löschenden Knoten die Positionsinformationen seiner Geschwister angepasst werden. Alle Knoten, die in der Ordnung der Menge der Subelemente nach dem zu löschenden Knoten kommen, rücken dabei um eine Position nach vorn.

```
UPDATE document_table a
SET a.position = a.position - 1
WHERE a.parent_node.node_id = (SELECT b.parent_node.node_id
                               FROM document_table b
                               WHERE b.node_id = "ID") AND
      a.position > (SELECT c.position
                    FROM document_table c
                    WHERE c.node_id = "ID")
```

Danach kann mittels einer einzigen Operation das eigentliche Löschen vorgenommen werden.

```
DELETE
FROM document_table a
WHERE a.node_id IN ("Menge von IDs")
```

Im Gegensatz zum Löschen einzelner Knoten verbleiben die durch die vorangegangenen Aktionen nicht mehr erreichbaren Dokumentteile nicht in der Datenbank, sondern werden ebenfalls entfernt. Dazu stellt Oracle die Funktion IS DANGLING zur Verfügung, mit der getestet werden kann, ob referenzierte Objekte noch existent sind. Kaskadierend werden nun alle nicht mehr erreichbaren Knoten gelöscht. Folgende Operationen werden solange ausgeführt, bis die Selektion der ersten Operation die leere Menge als Ergebnis liefert.

```
SELECT a.node_id
FROM document_table a
WHERE a.parent_node IS DANGLING

DELETE
FROM document_table a
WHERE a.node_id IN ("Menge von IDs")
```

Die Reihenfolge der beschriebenen Schritte ist wesentlich, da für das Anpassen der Positionen der zu löschende Knoten noch vorhanden sein muss, für das Entfernen des darunterliegenden Teilbaumes aber bereits fehlen sollte.

Das Löschen eines gesamten Dokumentbaumes als Spezialfall kann mittels einer einzigen Anweisung umgesetzt werden. Alleiniger Parameter ist dabei der Name des zu löschenden Dokumentes.

```
DELETE
FROM document_table a
WHERE a.document.node_value = ‘‘Dokumentname‘‘ OR
      a.node_value = ‘‘Dokumentname‘‘
```

6.2.3 Löschen von Attributen

Das Entfernen von Attributen lässt sich aufgrund des gewählten Speicherungmodells sehr einfach umsetzen. Die Methode zum Löschen erhält dabei als Parameter eine Menge der Knoten, bei denen das betreffende Attribut gelöscht werden soll sowie den Namen des zu löschenden Attributes.
Für jeden durch eine ID spezifizierten Knoten wird die folgende Aktion separat durchgeführt. Dies ist erforderlich, da pro Elementknoten auf einer geschachtelten Tabelle operiert wird. Das gleichzeitige Bearbeiten von mehreren Tabellen ist nicht möglich.

```
DELETE
FROM TABLE(SELECT TREAT(VALUE(b) AS element_type).attr_list
           FROM document_table b
           WHERE b.node_id = ‘‘ID‘‘) a
WHERE a.attr_name = ‘‘Attributname‘‘
```

Eventuell kann die Menge der Knoten-IDs vorher eingeschränkt werden, indem mittels einer weiteren Anfrage nur die IDs von Knoten abgefragt werden, die das betreffende Attribut auch besitzen. Letztendlich sollte man aber den *Location Path* der Anfrage, die die ursprüngliche ID-Menge liefert, derart wählen, dass dieser zusätzliche Schritt vermieden werden kann.
Eine weitere Variante des Löschens von Attributen besteht darin, nur Attribute des betreffenden Namens zu entfernen, die einen speziellen Wert aufweisen. In diesem Fall wird die “Where“-Klausel durch ein

```
AND a.attr_value = ‘‘Attributwert‘‘
```

erweitert.

6.3 Rename

Mittels eines Rename wird der Wert eines Knotens geändert. Dies bezieht sich auf den Tag-Namen von Elementknoten und den Inhalt eines Textknotens. Bei Attributen kann sowohl der Name des Attributes als auch der Wert geändert werden.

6.3.1 Ändern von Knotenwerten

Die Änderung von Knotenwerten lässt sich in Bezug auf das gewählte Speicherungsmodell sehr einfach umsetzen.
Durch Aufruf der Query-Methode und Übergabe eines Pfadausdrucks wird der Kontextknoten, dessen Wert geändert werden soll, spezifiziert. Eingabeparameter einer Rename-Methode ist die ID des Kontextknotens und der neue Wert. Die Änderung wird dann durch Ausführen der folgenden Operation erreicht.

```
UPDATE document_table a
SET a.node_value = “neuer Wert“
WHERE a.node_id IN (“Menge von IDs“)
```

6.3.2 Ändern von Attributen

Sollen Attributwerte geändert werden, sind zur Identifizierung die ID des Kontextknotens und der Name des entsprechenden Attributes erforderlich. Handelt es sich um eine Menge von Kontextknoten, muss wie beim Löschen von Attributen jeder Knoten als separates Objekt behandelt werden. Folgende Operation ändert den Namen des bezeichneten Attributes.

```
UPDATE TABLE(SELECT TREAT(VALUE(a) AS element_type).attr_list
             FROM document_table a
             WHERE a.node_id = “ID“) t
SET t.attr_name = “neuer Wert“
WHERE t.attr_name = “Attributname“
```

Analog erfolgt das Ändern eines Attributwertes durch folgende Operation. Wird der Wert eines Attributes geändert, sollte zur Sicherheit der neue Typ des Wertes mit angegeben werden.

```
UPDATE TABLE(SELECT TREAT(VALUE(a) AS element_type).attr_list
             FROM document_table a
             WHERE a.node_id = “ID“) t
SET t.attr_value = “neuer Wert“,
    t.attr_type = “neuer Typ“
WHERE t.attr_name = “Attributname“
```

6.4 Replace

Ein **Replace** entspricht in der Regel dem Einfügen eines neuen Knotens und dem anschließenden Löschen des zu ersetzenden Knotens. Dabei existieren verschiedene Möglichkeiten, die durch die bereits beschriebenen Operationen umgesetzt werden können.

- Das Ersetzen eines Textknotens durch einen anderen Textknoten entspricht dem Ändern des Wertes des betreffenden Knotens, da die ID des Knotens ohne Einschränkung beibehalten werden kann.

- Das Ersetzen eines Elementknotens durch einen anderen Elementknoten entspricht dem Ändern des Wertes des betreffenden Knotens, wenn der darunterliegende Teilbaum bestehen bleiben soll. Es entfällt das Anpassen der "parent_node"-Referenzen der Subknoten, da die ID des Knotens dabei nicht geändert wird. Die Attribute des alten Elementes werden gelöscht, eventuell neue Attribute eingefügt.
  Im Falle, dass der darunter liegende Teilbaum nicht erhalten bleiben soll, entspricht die Operation dem Einfügen eines neuen Elementknotens an der Position des alten und dem anschließenden Löschen eines XML-Fragmentes, wobei der Kontextknoten dieser Operation der alte Elementknoten ist.

- Ein Elementknoten kann nur durch einen Textknoten ersetzt werden, wenn keine Subelemente vorhanden sind bzw. wenn alle vorhandenen Subelemente ebenfalls gelöscht werden. Es ist also das Einfügen eines Textknotens und das Löschen eines XML-Fragmentes erforderlich.

- Das Ersetzen eines Textknotens durch einen Elementknoten wird durch das Einfügen eines neuen Elementes und das anschließende Löschen des Textknotens realisiert.

Für alle weiteren Knotentypen, die in dieser Arbeit nicht separat behandelt werden, gestalten sich die nötigen Operationen analog.

6.5 Move

Das Verschieben von Objekten entspricht im Allgemeinen dem nochmaligen Einfügen des zu verschiebenden Objektes und dem anschließenden Löschen an der alten Position. In Bezug auf die beschriebene Speicherungsstruktur ist das Ausführen dieser Operation sehr viel einfacher zu bewerkstelligen, indem die Referenzen der zu verschiebenden Objekte angepasst werden. Zusätzlich

ist eine Änderung sowohl der Ordnung innerhalb der ehemaligen als auch der neuen Geschwister erforderlich.

6.5.1 Verschieben innerhalb des Dokumentes

Eingabeparameter ist die ID des zu verschiebenden Knotens, die ID des Knotens, an den der zu verschiebende Knoten angehängt werden soll, und die Position innerhalb der Subknoten des neuen Vaterknotens. Um beide IDs im weiteren Verlauf unterscheiden zu können, werden die ID des zu verschiebenden Knotens mit "Move-ID" und die des zukünftigen Vaters mit "Destination-ID" bezeichnet. Beim Verschieben eines Knotens innerhalb eines Dokumentes sind eventuell vorhandene Kinder von den auszuführenden Operationen nicht betroffen. Deshalb gestaltet sich ein Move bei allen Knotentypen gleich.

Als erstes müssen die Positionsinformationen der Geschwister des zu verschiebenden Knotens entsprechend geändert werden. Bei allen Knoten, die einen größeren Wert besitzen als der zu verschiebende Knoten, wird die Position um 1 vermindert.

```
UPDATE document_table a
SET a.position = a.position - 1
WHERE a.parent_node.node_id = (SELECT b.parent_node.node_id
                               FROM document_table b
                               WHERE b.node_id = "Move-ID") AND
      a.position > (SELECT c.position
                    FROM document_table c
                    WHERE c.node_id = "Move-ID")
```

Danach werden die Informationen der zukünftigen Geschwister angepasst. Allen Knoten, bei denen der Wert der Position größer bzw. gleich dem übergebenen Positionswert ist, rücken um eine Stelle nach hinten.

```
UPDATE document_table a
SET a.position = a.position + 1
WHERE a.parent_node.node_id = "Destination-ID" AND
      a.position >= "Position"
```

Zum Ende erfolgt das eigentliche "Umschaufeln" des zu verschiebenden Knotens. Dabei wird nicht nur die "parent_node"-Referenz angepasst, sondern auch die aktuelle Position eingetragen.

```
UPDATE document_table a
SET a.position = "Position",
    a.parent_node = (SELECT REF(b)
                     FROM document_table b
                     WHERE b.node_id = "Destination-ID")
WHERE a.node_id = "Move-ID"
```

6.5.2 Verschieben zwischen Dokumenten

Beim Verschieben zwischen verschiedenen Dokumenten erfolgen dieselben Aktionen wie beim Verschieben innerhalb eines Dokumentes.
Im Anschluss daran müssen allerdings noch für den gesamten verschobenen Teilbaum die "document"-Referenzen angepasst werden. Dazu wird festgestellt, bei welchen Knoten das "document"-Attribut nicht auf dasselbe Dokument verweist wie das des Vaters. Bei diesen Knoten wird im Folgenden die entsprechende Information geändert. Die beiden Schritte werden solange wiederholt, bis die Ergebnismenge der ersten Operation leer ist.

```
SELECT a.node_id
FROM document_table a
WHERE NOT a.document.node_id = a.parent_node.document.node_id

UPDATE document_table a
SET a.document = a.parent_node.document
WHERE a.node_id IN ("Menge von IDs")
```

6.6 Optimierung der Änderungs-Operationen

Die einleitend erwähnten Basisoperationen lassen sich für das gewählte Speichermodell ohne Einschränkung umsetzen. Der Aufwand für das nachträgliche Ändern von XML-Daten ist gering. Alle Operationen können lokal für den jeweiligen Knoten durchgeführt werden und haben unter Umständen noch Auswirkung auf die Geschwisterknoten des Kontextknotens. In der Regel bedeutet dies aber lediglich, dass deren Positionsattribute angepasst werden müssen. Damit bietet das vorgestellte Speichermodell einen großen Vorteil gegenüber der Methode, die in der Arbeit von Guido Rost [Ros01] für die Implementierung von XQuery gewählt wurde. Dort beeinflusste eine derartige Operation aufgrund der Kodierung das gesamte Dokument.
Update-Operationen lassen sich durch wenige SQL-Anweisungen ausdrücken. Welche Schritte im Einzelnen nötig sind, wurde in den vorhergehenden Abschnitten ausführlich erläutert.
Zusätzlich kann man ausgehend von den beschriebenen Verfahren atomare Methoden definieren. Dies sind Anweisungen, die im Verlauf der Update-Operationen immer wieder verwendet werden. Beispiele dafür sind das Ändern der Positionen und das Anpassen der "parent_node"-Attribute. Diese Schritte werden in fast jeder Update-Methode verwendet und jedesmal neu implementiert. Verallgemeinert und mit den entsprechenden Eingabe-Parametern versehen, können sie einmalig als Objekt-Methoden definiert

werden. Eine Update-Operation lässt sich dann durch Aufrufe dieser atomaren Methoden (siehe Tab. 6.1) umsetzen.

Methode	Beschreibung
INSERT_DOC_NODE()	fügt einen Dokumentknoten ein
INSERT_TEXT_NODE()	fügt einen Textknoten ein
INSERT_ELEMENT_NODE()	fügt einen Elementknoten ein und initialisiert dessen "attr_list"-Attribut
INSERT_ATTRIBUTE()	fügt zu einem Elementknoten ein XML-Attribut ein
UPDATE_POSITION()	passt die Positionen einer Menge von Knoten an
UPDATE_PARENT_NODE()	verschiebt eine Menge von Knoten unter einen neuen Vaterknoten
UPDATE_NODE_VALUE()	ändert den Wert eines Knotens
UPDATE_ATTRIBUT_NAME()	ändert den Namen eines bestimmten Attributes eines Elementknotens
UPDATE_ATTRIBUTE_VALUE()	ändert den Wert (und gleichzeitig den Typ) eines bestimmten Attributes eines Elementknotens
DELETE_DOCUMENT()	löscht ein vollständiges Dokument aus der Relation
DELETE_NODES()	löscht eine Menge von Knoten aus der Relation
GET_DANGLING_NODES()	liefert die IDs einer Menge von Knoten, deren Attribut "parent_node" ins Leere verweist

Tabelle 6.1: Atomare Objekt-Methoden

Weitere Methoden, die dem Zugriff auf die einzelnen Objektinformationen dienen, wurden bereits im Abschnitt 5.3 aufgeführt.
Die beschriebene Herangehensweise bedeutet jedoch nicht zwangsläufig, dass die Komplexität der Änderungsoperationen sinkt. Es ist durchaus möglich, dass ein derartiges Verallgemeinern der Anweisungen zusätzliche Schritte erforderlich macht. Das Anpassen der Positionen einer Menge von Knoten erfordert beispielsweise drei Parameter. Benötigt werden in allen Fällen der Wert, um den die Positionen erhöht (bzw. vermindert) werden müssen, die ID des Knotens, bei dessen Kindern die Änderung erfolgen soll sowie die Angabe, bei welchen dieser Subknoten die Position verändert wird. Im Einzelfall können diese Angaben durch Schachtelung direkt innerhalb der Anweisung selektiert werden. Um die Werte jedoch als Parameter an eine entsprechende Methode übergeben zu können, müssen sie im Vorfeld selbst durch eine Anweisung (bzw. einen Methodenaufruf) spezifiziert werden.
Außerdem ist das Trennen von gleichzeitig durchführbaren Schritten erforderlich. Unter Umständen kann eine Implementierung dadurch erheblich an Umfang zunehmen.

Kapitel 7

Schlußbetrachtungen

7.1 Zusammenfassung

In dieser Arbeit wurde ein allgemeines Speichermodell zur Archivierung von XML-Dokumenten entwickelt. Unter allgemein ist die Unabhängigkeit von einem vorgegebenen Schema zu verstehen. Der Vorteil dieses Ansatzes besteht darin, dass Dokumente verschiedener Klassen mithilfe derselben Strukturen abgelegt werden können. Das entwickelte Modell wurde in Kapitel 4 vorgestellt. Mit ihm lassen sich Daten und Dokumentstrukturen gleichermaßen darstellen. Ausgangspunkt war das XML InfoSet, ein Datenmodell, mit dessen Hilfe für die Speicherung relevante Strukturinformationen zusammengestellt wurden. So setzt sich ein Dokument aus verschiedenen Knoten zusammen, die jeweils spezielle Attribute besitzen. Zwischen den Knoten gibt es Verwandtschaftsbeziehungen, mit denen sich die Zusammenhänge einzelner Teile eines XML-Dokumentes darstellen lassen. In der vorliegenden Arbeit wurden hierfür Verweise auf das jeweilige Vaterelement verwendet. XML-Attribute wurden nicht als eigenständige Knoten umgesetzt. Sie wurden direkt als Teil des entsprechenden *Owner*-Elementes abgelegt. Dadurch wird die zu speichernde Datenmenge reduziert.
Zur Modellierung des XML-Datentyps wurden eine Reihe von objektrelationalen Eigenschaften des DBMS Oracle9i verwendet. Durch ihre effiziente Ausnutzung wurde eine Optimierung der Speicherstrukturen erreicht. Beispielsweise basieren die entwickelten Speicherungsstrukturen zu einem Teil auf Referenzen, durch die eine intuitive Darstellung der Verwandtschaftsbeziehungen zwischen Knoten möglich ist. Durch die Verwendung solcher direkter Verweise entfällt die Speicherung redundanter Informationen wie beispielsweise eines Identifikators. Außerdem gestaltet sich der Zugriff auf verwandte Objekte einfacher, da keine Verbundoperationen genutzt werden

müssen.
Auch das Prinzip der Typvererbung, das seit der Version 9i Bestandteil von Oracle ist, spielte eine wichtige Rolle bei der Umsetzung der unterschiedlichen Knotenarten eines Dokumentes. So wurde ein allgemeiner Knotentyp definiert, von dem alle weiteren speziellen Knoten abgeleitet werden. Diese Herangehensweise bietet die Möglichkeit, alle Knoten in einer Relation abzuspeichern, ohne bei Objekten mit weniger Attributen die nicht benötigten Spalten mit NULL-Werten auffüllen zu müssen.
Es wurde ein Satz von Methoden ermittelt und ausführlich beschrieben (siehe Abschnitt 5.3 und Kapitel 6), der innerhalb der Datenbank sowohl für eine Anfragebearbeitung als auch für Änderungsoperationen zur Verfügung gestellt wird. Als ein Schwerpunkt dieser Betrachtungen muss der effiziente Datenaustausch zwischen einem Client und dem Datenbankserver genannt werden. So wurde beispielsweise festgestellt, dass es genügt, nur XQuery-Pfadausdrücke zur Beantwortung an die Datenbank zu schicken. Die restliche Bearbeitung einer XQuery-Anfrage erfolgt auf Seiten des Client. Die Abbildung eines Pfadausdruckes in eine SQL-Anweisung wird vollständig innerhalb des Datenbankservers vorgenommen. Ergebnis einer derartigen Anfrage ist eine Menge von Knoten-IDs. Braucht die Client-Anwendung weitere Informationen, können zu diesem Zweck zusätzlich entworfene Methoden aufgerufen werden. Damit wird erreicht, dass nur ein Austausch der Daten zwischen Client und Server erfolgt, die eine Anwendung wirklich benötigt.
Obwohl die momentane Spezifikation von XQuery noch keine Änderungsoperationen vorsieht, wurde die Bereitstellung derartiger Methoden ebenfalls diskutiert (Kapitel 6). Dabei wurde festgestellt, dass sich das entwickelte Speichermodell ohne Einschränkung für die Implementierung unabdingbarer Basisoperationen wie dem nachträglichen Einfügen, Löschen bzw. Ändern von XML-Daten eignet. Ein großer Vorteil besteht darin, dass Änderungen lokal durchgeführt werden können, ohne das restliche Dokument zu beeinflussen.
Abschließend soll noch erwähnt werden, dass, wenn zu einer Klasse von Dokumenten eine Beschreibung in Form eines Schemas existiert, der von Oracle angebotene XMLType verwendet werden kann. Oracle bietet effiziente Speicher- und Zugriffsmethoden auf XML-Daten. Zum Zwecke der Anfragebearbeitung wird eine XPath-Anfrageschnittstelle zur Verfügung gestellt.

7.2 Ausblick

Bei der Speicherung von XML-Dokumenten mithilfe des entwickelten Modells werden alle Informationen innerhalb einer einzigen Relation abgespeichert.

Bei vielen bzw. großen Dokumenten kann diese sehr umfangreich werden. Das in Anhang A.1 enthaltene Dokument resultiert beispielsweise bereits in 33 Tupeln. Für sehr große Dokumentmengen können deshalb die Daten auch in verschiedenen Tabellen abgelegt werden. Eine Trennung kann unter anderem nach Klassen von Dokumenten erfolgen. Zusätzlich ist eine Aufteilung nach Knotentypen denkbar, da die verwendeten Referenzen auch zwischen verschiedenen Relationen möglich sind und daher diese Art der Modellierung von Verwandtschaftsbeziehungen nicht verändert werden muss.
Um Dokumente vollständig abspeichern zu können, wird die Umsetzung der restlichen Knotentypen empfohlen: *Processing Instructions*, Kommentare und *Namespaces*. Dabei muss auf die Implementierung zusätzlicher Attribute bei bestehenden Knotentypen geachtet werden. Das XML InfoSet beschreibt unter anderem die Eigenschaft **[prefix]**, die dann innerhalb des Knotentyps für XML-Elemente umgesetzt werden muss.
Die Beantwortung von Pfadausdrücken bietet ein weites Feld für weiterführende Arbeiten. Es können zum Beispiel noch weitere im Sprachumfang von XPath bzw. XQuery vorhandene Selektionskriterien für Knoten innerhalb der *Location Step*-Prädikate umgesetzt werden. Insbesondere ist noch nach Möglichkeiten der effizienten Umsetzung von // zu suchen. Unter Umständen muss dafür das Speicherungsmodells ergänzt werden.
In Bezug auf die Rückgabewerte von Objektmethoden können noch weiterführende Ansätze betrachtet werden. Hierzu zählt beispielsweise die Generierung von Java-Klassen für das sogenannte *Type Mapping*, worunter die Abbildung von SQL-Objekttypen und geschachtelten Tabellen in Java-Typen zu verstehen ist. Interessant ist auch die Frage, ob es in Oracle9i möglich ist, Objekte zu generieren, die als solche nicht in der Datenbank vorhanden sind. Damit können Informationen von verschiedenen Tupeln zu einem Objekt zusammengefasst werden.
Zusätzlich kann die Implementierung der vorgestellten Update-Operationen fortgesetzt werden. Einen Ausgangspunkt bietet die in dieser Arbeit vorgenommene Untersuchung der Vor- und Nachteile einer Realisierung durch atomare Methoden.
Das Speichermodell ist in jedem Fall offen für Erweiterungen. So können die bestehenden Knotentypen um zusätzliche Attribute ergänzt werden. Als Beispiel soll hier ein Zusatz für Dokumentknoten genannt werden. Eine Information, die das diskutierte Modell noch nicht besitzt, die für einen XML-Datentyp aber sehr wohl interessant sein kann, ist die Speicherung von Schema-Daten. Anhand eines entsprechenden Attributes wäre es möglich, bei der Selektion von Elementen dokumentübergreifend all diejenigen zu erfragen, deren Dokumente zu einer gemeinsamen Klasse gehören.

Literaturverzeichnis

[BR99] Ricardo Baeza-Yates, Berthier Ribeiro-Neto. Modern Information Retrieval. Addison-Wesley, Harlow, England. 1999.

[Bra98] Neil Bradley. The XML Companion. Addison-Wesley, March 1998.

[BW] Bas de Bakker, Irsan Widarto. An Introduction to XQuery. http:www.perfectxml.com/articles/xml/xquery.asp.

[Cat96] Rick Cattell et al. The Object Database Standard: ODMG-93, Release 1.2. Morgan Kaufmann Publishers, San Francisco, 1996.

[CHRS98] Andreas Christiansen, Michael Höding, Claus Rautenstrauch, Gunter Saake. Oracle8 effizient einsetzen - Aufbau, Entwicklung, Verteilung und Betrieb leistungsfähiger Oracle8-Anwendungen. Addison-Wesley, Bonn 1998.

[CRF00] Don Chamberlin, Jonathan Robie, Daniela Florescu. Quilt: an XML Query Language for Heterogeneous Data Sources. In Lecture Notes in Computer Science. Springer-Verlag, 2000.

[DFFLS] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, Dan Suciu. A Query Language for XML. http://www.research.att.com/ mff/files/final.html.

[Hae02] Birger Hänsel. Änderungsoperationen in XML-Anfragesprachen. Diplomarbeit, Universität Rostock, Juni 2002.

[Heu97] Andreas Heuer. Objektorientierte Datenbanken - Konzepte, Modelle, Standards und Systeme. 2., aktualisierte Auflage, Addison-Wesley, Bonn 1997.

[HM02] Elliotte Rusty Harold, W. Scott Means. XML in a Nutshell - A Desktop Quick Reference. O'Reilly, Januar 2001.

[IBM99] XML Extender - Administration and Programming. Version 7. IBM Corporation, 1999.

[ISO86] ISO (International Organization for Standardization). ISO 8879:1986(E). Information processing - Text and Office Systems - Standard Generalized Markup Language (SGML). First edition - 1986-10-15. International Organization for Standardization, 1986.

[ISO99] ISO (International Organization for Standardization). ISO/IEC 9075:1999. Information Technology-Database Language SQL. International Organization for Standardization, 1999.

[KM02] Meike Klettke, Holger Meyer. XML und Datenbanken - Konzepte, Sprachen und Systeme. dpunkt, 2002.

[Ora01] Inheritance, substitutability and polymorphism in Oracle9i. White Paper. Note: 146398.1. 25. June 2001.

[Ora02a] Oracle9i Application Developer's Guide - Object-Relational Features. Release 2 (9.2). Part No. A96594-01. Oracle Corporation, March 2002.

[Ora02b] PL/SQL User's Guide and Reference. Release 2 (9.2). Part No. A96624-01. Oracle Corporation, March 2002.

[Ora02c] Oracle9i Java Developer's Guide. Release 2 (9.2). Part No. A96656-01. Oracle Corporation, March 2002.

[Ora02d] Oracle9i Java Stored Procedures Developer's Guide. Release 2 (9.2). Part No. A96659-01. Oracle Corporation, March 2002.

[Ora02e] Oracle9i JDBC Developer's Guide and Reference. Release 2 (9.2). Part No. A96654-01. Oracle Corporation, March 2002.

[Ora02f] Oracle9i SQL Reference. Release 2 (9.2). Part No. A96540-01. Oracle Corporation, March 2002.

[Ora02g] Oracle9i XML Database Developer's Guide - Oracle XML DB. Release 2 (9.2). Part No. A96620-01. Oracle Corporation, March 2002.

[Ora02h] Oracle9i XML API Reference - XDK and Oracle XML DB. Release 2 (9.2). Part No. A96616-01. Oracle Corporation, March 2002.

[Ora02i] Oracle9i JPublisher User's Guide. Release 2 (9.2). Part No. A96658-01. Oracle Corporation, March 2002.

[RLS] J. Robie, J. Lapp, D. Schach. XML Query Language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[Ros01] Guido Rost. Zur Implementierung von XQuery auf einem Objekt-relationalem XML-Speicher. Diplomarbeit, Universität Rostock, Dezember 2001.

[SYU] Takeyuki Shimura, Masatoshi Yoshikawa, Shunsuke Uemura. Storage and Retrieval of XML Documents using Object-Relational Databases.

[W3C00] Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation. 6. October 2000.
http://www.w3.org/TR/2000/REC-xml-20001006.

[W3C01a] XML Query Requirements. W3C Working Draft. 15. February 2001.
http://www.w3.org/TR/2001/WD-xmlquery-reg-20010215.

[W3C01b] XML Information Set. W3C Recommendation. 24. October 2001.
http://www.w3.org/TR/2001/REC-xml-infoset-20011024.

[W3C01c] XML Schema Part 0: Primer. W3C Recommendation. 2. May 2001.
http://www.w3.org/TR/2001/REC-xmlschema-0-20010502.

[W3C01d] XML Schema Part 1: Structures. W3C Recommendation. 2. May 2001.
http://www.w3.org/TR/2001/REC-xmlschema-1-20010502.

[W3C01e] XML Schema Part 2: Datatypes. W3C Recommendation. 2. May 2001.
http://www.w3.org/TR/2001/REC-xmlschema-2-20010502.

[W3C02a] XML Path Language (XPath) 2.0. W3C Working Draft. 16. August 2002.
http://www.w3.org/TR/2002/WD-xpath20-20020816.

[W3C02b] XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft. 16. August 2002.
http://www.w3.org/TR/2002/WD-query-datamodel-20020816.

[W3C02c] XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft. 16. August 2002. http://www.w3.org/TR/2002/WD-query-semantics-20020816.

[W3C02d] XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft. 16. August 2002. http://www.w3.org/TR/2002/WD-xquery-operators-20020816.

[W3C02e] XQuery 1.0: An XML Query Language. W3C Working Draft. 16. August 2002. http://www.w3.org/TR/2002/WD-xquery-20020816.

[W3C02f] XML Query Use Cases. W3C Working Draft. 16. August 2002. http://www.w3.org/TR/2002/WD-xmlquery-use-cases-20020816.

[W3C99] XML Path Language (XPath) Version 1.0. W3C Recommendation. 16. November 1999. http://www.w3.org/TR/xpath.html.

[W3School] Learn XPath. Web-Tutorial. http://www.w3schools.com/xpath.

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang A

Beispiel

A.1 Beispieldokument

buch-katalog.xml

```
<?xml version="1.0"?>
<buecher>
  <buch einband="paperback">
    <titel>The Lord Of The Rings</titel>
    <autor>J.R.R. Tolkien</autor>
    <verlag>Harper Collins</verlag>
  </buch>
  <buch einband="gebunden">
    <titel>Datenbanken: Implementierungstechniken</titel>
    <autor>Andreas Heuer</autor>
    <autor>Gunter Saake</autor>
    <verlag>mitp</verlag>
  </buch>
  <buch inhalt="leer"/>
  <buch einband="paperback">
    <titel>The Hobbit</titel>
    <autor>J.R.R. Tolkien</autor>
    <verlag>Harper Collins</verlag>
  </buch>
  <buch inhalt="mixed content">
    ein Buch mit dem Titel
    <titel>LaTeX - kurz & gut</titel>
    von
    <autor>Matthias Kalle Dalheimer</autor>
  </buch>
</buecher>
```

A.2 Beispiel-Relation

Aus Gründen der Übersicht wurde zur Darstellung der Verweise auf den jeweiligen Vaterknoten bzw. den Dokumentknoten die Knoten-IDs gewählt.

node_id	node_value	node_type	position	parent_node	document
1	buch_katalog.xml	doc	1	NULL	NULL
2	buecher	element	1	1	1
3	buch	element	1	2	1
4	titel	element	1	3	1
5	The Lord Of ...	text	1	4	1
6	autor	element	2	3	1
7	J.R.R. Tolkien	text	1	6	1
8	verlag	element	3	3	1
9	Harper Collins	text	1	8	1
10	buch	element	2	2	1
11	titel	element	1	10	1
12	Datenbanken ...	text	1	11	1
13	autor	element	2	10	1
14	Andreas Heuer	text	1	13	1
15	autor	element	3	10	1
16	Gunter Saake	text	1	15	1
17	verlag	element	4	10	1
18	mitp	text	1	17	1
19	buch	element	3	2	1
20	buch	element	4	2	1
21	titel	element	1	20	1
22	The Hobbit	text	1	21	1
23	autor	element	2	20	1
24	J.R.R. Tolkien	text	1	23	1
25	verlag	element	3	20	1
26	Harper Collins	text	1	25	1
27	buch	element	5	2	1
28	ein Buch mit ...	text	1	27	1
29	titel	element	2	27	1
30	LaTeX ...	text	1	29	1
31	von	text	3	27	1
32	autor	element	4	27	1
33	M. K. Dalheimer	text	1	32	1

attr_list

attr_name	attr_value	attr_type
einband	paperback	CDATA

attr_list

attr_name	attr_value	attr_type
einband	gebunden	CDATA

attr_list

attr_name	attr_value	attr_type
inhalt	leer	CDATA

attr_list

attr_name	attr_value	attr_type
einband	paperback	CDATA

attr_list

attr_name	attr_value	attr_type
inhalt	mixed content	CDATA

Abbildung A.1: Relation mit vollständigem Beispieldokument

A.3 Baumdarstellung des Beispiels

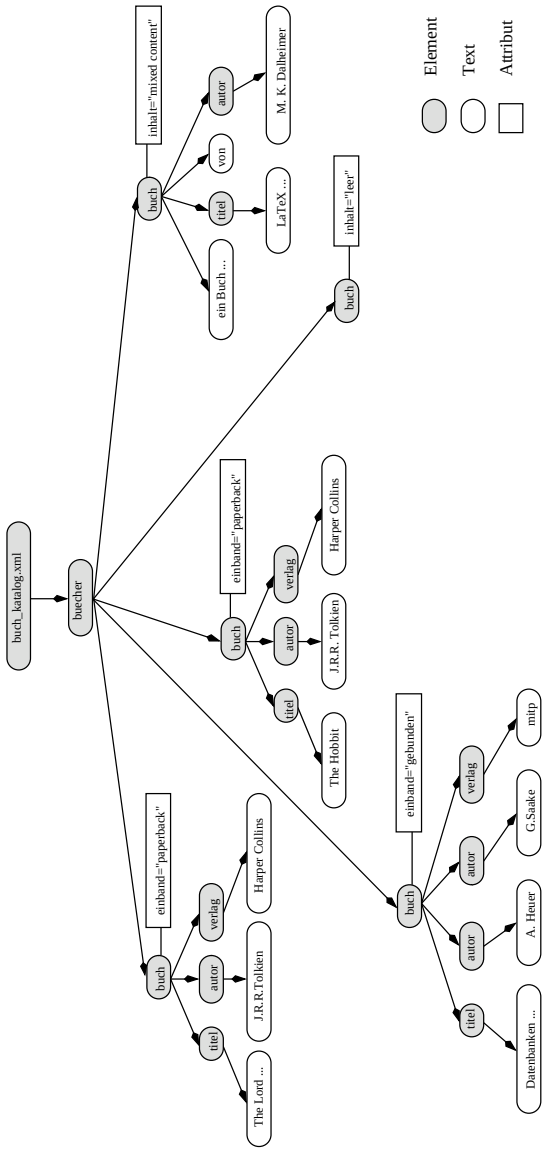


Abbildung A.2: Vollständiger Baum des Beispieldokumentes

Anhang B

Entwicklungsumgebung

Für die prototypische Umsetzung des diskutierten Speicherungsmodells und die Implementierung der damit verbundenen Methoden wurde folgende Entwicklungsumgebung verwendet:

- Betriebssystem SuSE Linux 8.0 (Kernel 2.4.18),
- Oracle9i Database Standard Edition Release 2 (9.2.0.1.0) für Unix-Systeme,
- Oracle Enterprise Manager 9.2.0.1.0,
- JDK 1.3.1,
- Oracle JDBC 2.0 Thin Driver,
- Oracle SQL.

Anhang C

Objekttyp-Definitionen

C.1 node_type und document_table

Folgender Objekttyp modelliert einen allgemeinen Knoten mit den relevanten Informationen und den für alle Knotentypen geltenden Methoden.

```
CREATE TYPE node_type AS OBJECT (
  node_id      NUMBER,
  node_value   VARCHAR(4000),
  node_type    VARCHAR2(10),
  position     NUMBER,
  parent_node  REF node_type,
  document     REF node_type,
  STATIC FUNCTION PROCESS_QUERY (location_path VARCHAR2,
                                 variable VARCHAR2,
                                 nodeSet VARCHAR2) RETURN VARCHAR2
    AS LANGUAGE JAVA
    NAME 'Query.processQuery(java.lang.String,
                             java.lang.String,
                             java.lang.String)
          return java.lang.String',
  STATIC FUNCTION GET_VALUE (node VARCHAR2) RETURN VARCHAR2
    AS LANGUAGE JAVA
    NAME 'GetNodeInfo.getValue(java.lang.String)
          return java.lang.String',
  STATIC FUNCTION GET_TYPE (node VARCHAR2) RETURN VARCHAR2
    AS LANGUAGE JAVA
    NAME 'GetNodeInfo.getType(java.lang.String)
          return java.lang.String',
  STATIC FUNCTION GET_POSITION (node VARCHAR2) RETURN NUMBER
    AS LANGUAGE JAVA
    NAME 'GetNodeInfo.getPosition(java.lang.String)
          return int',
  STATIC FUNCTION GET_PARENT_NODE (node VARCHAR2) RETURN VARCHAR2
```

```
    AS LANGUAGE JAVA
    NAME 'GetNodeInfo.getParentNode(java.lang.String)
          return java.lang.String',
  STATIC FUNCTION GET_DOCUMENT (node VARCHAR2) RETURN VARCHAR2
    AS LANGUAGE JAVA
    NAME 'GetNodeInfo.getDocument(java.lang.String)
          return java.lang.String',
  STATIC PROCEDURE DELETE_FRAGMENT (nodeSet VARCHAR2)
    AS LANGUAGE JAVA
    NAME 'Delete.deleteFragment(java.lang.String)');
NOT FINAL;
```

Zur Aufnahme aller Knotenobjekte (node_type und alle davon abgeleiteten Subtypen) wird folgende Objekt-Tabelle generiert.

```
CREATE TABLE document_table OF node_type
(node_id PRIMARY KEY);
```

C.2 Definition doc_type

Folgender von node_type abgeleiteter Objekttyp dient der Modellierung von Dokumentknoten. Der Typ erbt alle allgemeinen Knotenattribute und Methoden. Zusätzlich werden weitere Methoden definiert, die nur für Dokumentknoten aufgerufen werden können.

```
CREATE OR REPLACE TYPE doc_type UNDER node_type (
  STATIC PROCEDURE INSERT_DOC_NODE (name VARCHAR2)
    AS LANGUAGE JAVA
    NAME 'XMLInsert.InsertDocNode(java.lang.String)');
```

C.3 Definition text_type

Folgender von node_type abgeleiteter Objekttyp dient der Modellierung von Textknoten. Der Typ erbt alle allgemeinen Knotenattribute und Methoden. Zusätzlich werden weitere Methoden definiert, die nur für Textknoten aufgerufen werden können.

```
CREATE OR REPLACE TYPE text_type UNDER node_type (
  STATIC PROCEDURE INSERT_TEXT_NODE (name VARCHAR2,
                                     position NUMBER,
                                     parent VARCHAR2,
                                     document VARCHAR2)
    AS LANGUAGE JAVA
    NAME 'XMLInsert.InsertTextNode(java.lang.String,
```

```
                                    int,
                                    java.lang.String,
                                    java.lang.String)');
```

C.4 Definition attr_type und att_ntab

Folgender Objekttyp dient der Modellierung von Attributen. Zusätzlich werden Attribute in geschachtelten Tabellen gesammelt. Jeweils eine solche Nested Table ist dann Attribut eines Element-Knotens.

```
CREATE TYPE attr_type AS OBJECT (
  attr_name   VARCHAR(4000),
  attr_value  VARCHAR(4000),
  attr_type   VARCHAR(4000));

CREATE TYPE attr_ntab AS TABLE OF attr_type;
```

C.5 Definition element_type

Folgender von node_type abgeleiteter Objekttyp dient der Modellierung von XML-Elementen. Der Typ besitzt ein neues Attribut zur Aufnahme von Element-Attributen. Es werden zusätzliche Methoden vereinbart, die nur für Elementknoten aufgerufen werden können.

```
CREATE OR REPLACE TYPE element_type UNDER node_type (
  attr_list  attr_ntab,
  STATIC FUNCTION GET_CHILDREN (node VARCHAR2) RETURN VARCHAR2
    AS LANGUAGE JAVA
    NAME 'GetNodeInfo.getChildren(java.lang.String)
          return java.lang.String',
  STATIC PROCEDURE INSERT_ELEMENT_NODE (name VARCHAR2,
                                        position NUMBER,
                                        parent VARCHAR2,
                                        document VARCHAR2)
    AS LANGUAGE JAVA
    NAME 'XMLInsert.InsertElementNode(java.lang.String,
                                      int,
                                      java.lang.String,
                                      java.lang.String)',
  STATIC PROCEDURE INSERT_ATTRIBUTE (name VARCHAR2,
                                     value VARCHAR2,
                                     type VARCHAR2,
                                     parent VARCHAR2)
    AS LANGUAGE JAVA
    NAME 'XMLInsert.InsertAttribute(java.lang.String,
```

```
                                java.lang.String,
                                java.lang.String,
                                java.lang.String)');
```

Anhang D

Objekt-Methoden

D.1 Methoden zur Informationsgewinnung

Java-Klasse Query Die Methoden dieser Klasse dienen der Beantwortung von Pfadausdrücken. Von einer Anwendung wird die Funktion processQuery() aufgerufen.
Die Implementierung realisiert folgende Dinge:

- Pfadausdrücke mit mehreren *Location Steps*
- *Wildcards*
- document()-Funktion
- text()-Funktion
- Variablenbindungen (ein Schritt enthält statt eines Knotentests eine Variable, Variablen in Prädikaten müssen gesondert behandelt werden)
- Prädikate:
  - Elemente, die ein bestimmtes Attribut besitzen
  - Elemente, die ein Attribut mit einem bestimmten Wert besitzen
  - Elemente, die ein bestimmtes Subelement besitzen

```
import java.sql.*;
import java.io.*;
import oracle.sql.*;
import oracle.jdbc.*;

public class Query
{
  private static String var = new String("");
  private static String nodes = new String("");
  private static String where_clausel = new String("");
  private static String condition     = new String("");

  public static String processQuery(String location_path, String variable, String nodeSet)
  {
```

```
  where_clausel = "WHERE ";
  condition     = "p.";
  String result = new String("");
  String pfad   = new String(location_path);
  var = variable;
  nodes = nodeSet;
  System.out.println(pfad);

  // solange der Location Path nicht leer ist, extrahiere nächsten Location Step
  while(!(pfad.equals("")))
  {
    pfad = processPath(pfad);
  }

  // falls Location Path leer gewesen ist, muss die WHERE-Klausel komplett leer sein
  if(where_clausel.equals("WHERE "))
    where_clausel = "";
  // ansonsten wird das letzte AND abgeschnitten, da nichts mehr angefügt wird
  else
    where_clausel = where_clausel.substring(0,where_clausel.length()-5);

  String sql = "SELECT p.node_id FROM test_table p " + where_clausel;
  try
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");
    PreparedStatement pstmt = conn.prepareStatement(sql);

    ResultSet rs = pstmt.executeQuery();
    while(rs.next())
    {
      result += rs.getString(1) + ",";
    }
    rs.close();
    pstmt.close();
  }
  catch (SQLException e)
  {
    System.err.println(e.getMessage());
  }

  if(result.indexOf(",") != -1)
    result = result.substring(0,result.lastIndexOf(","));
  return result;
}


private static String processPath(String pfad)
{
  String step;

  // der nächste Step reicht vom letzten Slash bis zum Ende des Location Path
  // und wird vom Location Path abgeschnitten
  if(!(pfad.lastIndexOf("/") == -1))
  {
    step = pfad.substring(pfad.lastIndexOf("/")+1,pfad.length());
    pfad = pfad.substring(0,pfad.lastIndexOf("/"));
  }
  // falls Location Path nur noch aus einem Step besteht
  else
  {
    step = pfad;
    pfad = "";
  }

  processStep(step);
  return pfad;
}


private static void processStep(String step)
{
  String predicate = "";
  String name      = "";
  String value     = "";

  // Sonderbehandlung der Location Step-Funktion document()
  if(step.startsWith("document("))
    step = step.substring(step.indexOf("")+1, step.lastIndexOf(""));

  // Sonderbehandlung der Location Step-Funktion text()
  if(step.equals("text()"))
    where_clausel += condition + "node_type = 'text' AND ";
```

```
    // Sonderbehandlung Variablenbindung
    if(step.equals(var))
    {
      where_clausel += condition + "node_id IN (" + nodes + ") AND ";
    }

    // Sonderbehandlung der Location Steps mit Prädikat
    if(step.indexOf("[") != -1)
    {
      predicate = step.substring(step.indexOf("[")+1,step.indexOf("]"));
      name  = "";
      value = "";
      step = step.substring(0,step.indexOf("["));

      // Behandlung von Attribut-Prädikaten
      if(predicate.startsWith("@"))
      {
        where_clausel += condition + "node_id IN (SELECT p.node_id FROM test_table p";
        where_clausel += ", TABLE(TREAT(VALUE(p) AS element_type).attr_list) t ";

        // Attribut mit Wert
        if(predicate.indexOf("=") != -1)
        {
          name  = predicate.substring(predicate.indexOf("@")+1,predicate.indexOf("="));
          value = predicate.substring(predicate.indexOf("")+1,predicate.lastIndexOf(""));

          where_clausel += "WHERE t.attr_name = '";
          where_clausel += name;
          where_clausel += "' AND t.attr_value = '";
          where_clausel += value;
       }
        // Attribut ohne Wert
        else
        {
          name  = predicate.substring(predicate.indexOf("@")+1,predicate.length());
          where_clausel += "WHERE t.attr_name = '";
          where_clausel += name;
        }
        where_clausel += "') AND ";
      }
      // Behandlung von Child-Element-Prädikaten
      else
      {
        where_clausel += condition + "node_id IN (SELECT p.parent_node.node_id FROM test_table p";
        where_clausel += " WHERE p.node_value = '" + predicate + "') AND ";
      }
    }

    // wenn Wildcard, wird kein node_value in aktueller Bedingung angegeben
    // ansonsten wird WHERE-Klausel um aktuelle node_value-Bedingung erweitert
    if(!(step.equals("*")) && !(step.equals(var)) && !(step.equals("text()")))
      where_clausel += condition + "node_value = '" + step + "' AND ";

    // wird die Methode das nächste Mal aufgerufen, bezieht sich der Location Step
    // auf das Vaterelement des gerade aktuellen Elementes
    condition += "parent_node.";
  }
}
```

Java-Klasse GetNodeInfo Die Methoden dieser Klasse dienen dem Auslesen der Informationen eines Knotenobjektes.

```
import java.sql.*;
import java.io.*;
import oracle.sql.*;
import oracle.jdbc.*;

public class GetNodeInfo
{

  /**
   * Methode zum Auslesen des Attributs "node_value"
   */
  public static String getValue(String node) throws SQLException
  {
    String result = null;
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");

    String sql = "SELECT a.node_value FROM test_table a WHERE a.node_id = " + node;

    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sql);

    while(rs.next())
    {
      result = rs.getString(1);
    }

    rs.close();
    stmt.close();
    conn.close();

    return result;
  }

  /**
   * Methode zum Auslesen des Attributs "position"
   */
  public static int getPosition(String node) throws SQLException
  {
    int result = 0;
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");

    String sql = "SELECT a.position FROM test_table a WHERE a.node_id = " + node;

    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sql);

    while(rs.next())
    {
      result = rs.getInt(1);
    }

    rs.close();
    stmt.close();
    conn.close();

    return result;
  }

  /**
   * Methode zum Auslesen des Attributs "node_type"
   */
  public static String getType(String node) throws SQLException
  {
    String result = null;
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");

    String sql = "SELECT a.node_type FROM test_table a WHERE a.node_id = " + node;

    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sql);

    while(rs.next())
    {
      result = rs.getString(1);
    }

    rs.close();
    stmt.close();
    conn.close();
```

```
    return result;
  }

  /**
   * Methode liefert die ID des Vaterknotens des aktuellen Knotens
   */
  public static String getParentNode(String node) throws SQLException
  {
    String result = null;
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");

    String sql = "SELECT a.parent_node.node_id FROM test_table a WHERE a.node_id = " + node;

    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sql);

    while(rs.next())
    {
      result = rs.getString(1);
    }

    rs.close();
    stmt.close();
    conn.close();

    return result;
  }

  /**
   * Methode liefert die IDs der Subknoten des aktuellen Knotens
   */
  public static String getChildren(String node) throws SQLException
  {
    String result = new String("");
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");

    String sql = "SELECT a.node_id FROM test_table a WHERE a.parent_node.node_id = " + node;

    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sql);

    while(rs.next())
    {
      result += rs.getString(1) + ",";
    }

    rs.close();
    stmt.close();
    conn.close();

    if(result.indexOf(",") != -1)
      result = result.substring(0,result.lastIndexOf(","));

    return result;
  }

  /**
   * Methode liefert Dokument, zu dem aktueller Knoten gehört
   */
  public static String getDocument(String node) throws SQLException
  {
    String result = null;
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");

    String sql = "SELECT a.document.node_value FROM test_table a WHERE a.node_id = " + node;

    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sql);

    while(rs.next())
    {
      result = rs.getString(1);
    }

    rs.close();
    stmt.close();
    conn.close();

    return result;
  }
}
```

D.2 Methoden zum Einfügen

Java-Klasse XMLInsert Die Funktionen dieser Klasse zeigen beispielhaft die Implementierung atomarer Methoden zum Einfügen von Objektknoten.

```
import java.sql.*;
import java.io.*;
import oracle.sql.*;
import oracle.jdbc.*;

public class XMLInsert
{

  /**
   * Methode wird zum Einfügen eines Dokumentknotens aufgerufen
   */
  public static void InsertDocNode(String name) throws SQLException
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");

    String sql = "INSERT INTO test_table VALUES(doc_type(node_seq.NEXTVAL,?,?,1,NULL,NULL))";

    PreparedStatement pstmt = conn.prepareStatement(sql);
    pstmt.setString(1,name);
    pstmt.setString(2,"doc");
    pstmt.execute();
    pstmt.close();
    conn.close();
  }

  /**
   * Methode wird zum Einfügen eines Textknotens aufgerufen
   */
  public static void InsertTextNode(String name, int position, String parent, String document) throws SQLException
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");

    String sql = "INSERT INTO test_table VALUES(text_type(node_seq.NEXTVAL,?,?,?," +
                 "(SELECT REF(a) FROM test_table a WHERE a.node_id = ?),"          +
                 "(SELECT REF(a) FROM test_table a WHERE a.node_value = ?)))";

    PreparedStatement pstmt = conn.prepareStatement(sql);
    pstmt.setString(1,name);
    pstmt.setString(2,"text");
    pstmt.setInt(3,position);
    pstmt.setString(4,parent);
    pstmt.setString(5,document);
    pstmt.execute();
    pstmt.close();
    conn.close();
  }

  /**
   * Methode wird zum Einfügen eines Elementknotens aufgerufen
   */
  public static void InsertElementNode(String name, int position, String parent, String document) throws SQLException
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");

    String sql = "INSERT INTO test_table VALUES(element_type(node_seq.NEXTVAL,?,?,?," +
                 "(SELECT REF(a) FROM test_table a WHERE a.node_id = ?),"             +
                 "(SELECT REF(a) FROM test_table a WHERE a.node_value = ?),"         +
                 "attr_ntab(attr_type(NULL,NULL,NULL))))";

    PreparedStatement pstmt = conn.prepareStatement(sql);
    pstmt.setString(1,name);
    pstmt.setString(2,"element");
    pstmt.setInt(3,position);
    pstmt.setString(4,parent);
    pstmt.setString(5,document);
    pstmt.execute();

    sql = "DELETE FROM TABLE(SELECT TREAT(VALUE(a) AS element_type).attr_list " +
                           "FROM test_table a "                                 +
                           "WHERE a.node_id = (SELECT MAX(b.node_id) FROM test_table b))";
    pstmt = conn.prepareStatement(sql);
    pstmt.execute();
```

```
    pstmt.close();
    conn.close();
  }

  /**
   * Methode wird zum Einfügen eines Attributes aufgerufen
   */
  public static void InsertAttribute(String name, String value, String type, String parent) throws SQLException
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");

    String sql = "INSERT INTO TABLE(SELECT TREAT(VALUE(a) AS element_type).attr_list " +
                                    "FROM test_table a "                               +
                                    "WHERE a.node_id = ?) "                            +
                          "VALUES (?,?,?)";

    PreparedStatement pstmt = conn.prepareStatement(sql);
    pstmt.setString(1,parent);
    pstmt.setString(2,name);
    pstmt.setString(3,value);
    pstmt.setString(4,type);
    pstmt.execute();
    pstmt.close();
    conn.close();
  }
}
```

D.3 Methoden zum Löschen

Java-Klasse Delete Die Methoden dieser Klasse dienen dem Löschen eines XML-Elementes und des darunter liegenden Teilbaumes. Von der Anwendung wird die Funktion deleteFragment() aufgerufen. Die Implementierung orientiert sich an der Beschreibung in Abschnitt 6.2.2. Es erfolgt kein Aufruf atomarer Methoden.

```
import java.sql.*;
import java.io.*;
import oracle.sql.*;
import oracle.jdbc.*;

public class Delete
{
  public static void deleteFragment(String nodeSet)
  {
    String nodes = new String(nodeSet);
    if(!(nodeSet == null) && !(nodeSet.equals("")))
    {
      while(!(nodes.equals("")))
      {
String node = new String("");
if(nodes.indexOf(",") != -1)
       {
  node = nodes.substring(nodes.lastIndexOf(",")+1,nodes.length());
         nodes = nodes.substring(0,nodes.lastIndexOf(","));
}
       else
{
  node = nodes;
  nodes = "";
       }
       updatePosition(node);
      }
      deleteNodes(nodeSet);
      deleteDanglingRefs();
    }
  }

  private static void updatePosition(String node)
  {
    try
    {
      Connection conn = DriverManager.getConnection("jdbc:default:connection:");
      String sql = new String("UPDATE test_table a SET a.position = a.position - 1 WHERE a.parent_node.node_id = ");
      sql += "(SELECT b.parent_node.node_id FROM test_table b WHERE b.node_id = " + node + ")";
      sql += " AND a.position > ";
      sql += "(SELECT c.position FROM test_table c WHERE c.node_id = " + node + ")";
      sql += " AND NOT a.node_id = " + node;

      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.execute();
      pstmt.close();

      conn.close();
    }
    catch (SQLException e)
  }

  private static void deleteNodes(String nodes)
  {
    try
    {
      Connection conn = DriverManager.getConnection("jdbc:default:connection:");

      String sql = new String("DELETE FROM test_table p WHERE p.node_id IN (" + nodes + ")");

      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.execute();
      pstmt.close();
      conn.close();
    }
    catch (SQLException e)
  }

  private static void deleteDanglingRefs()
```

```
  {
    String result = null;
    try
    {
      Connection conn = DriverManager.getConnection("jdbc:default:connection:");

      do
      {
result = "";
        PreparedStatement pstmt;
        pstmt = conn.prepareStatement("SELECT p.node_id FROM test_table p WHERE p.parent_node IS DANGLING");
        ResultSet rst = pstmt.executeQuery();
        while(rst.next())
        {
  result += rst.getString(1) + ",";
        }
        rst.close();
        pstmt.close();

        if(!(result == null))
        {
          if(result.indexOf(",") != -1)
            result = result.substring(0,result.lastIndexOf(","));
  deleteNodes(result);
        }
      }
      while(!(result == null));
      conn.close();
    }
    catch (SQLException e)
  }
}
```

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Berlin, den 19.03.2003

Thesen

1. XML hat sich als Standardformat für semistrukturierte Daten und deren Austausch etabliert.

2. Die wachsende Verbreitung von Dokumenten im XML-Format erfordert die Entwicklung effizienter Speicherungstechniken zur Archivierung. Viele Datenbanken bieten deshalb einen XML-Datentyp an. Zusätzlich sind Anfragesprachen erforderlich, mit deren Hilfe die in den Dokumenten vorhandenen Informationen extrahiert werden können. XQuery ist eine solche XML-Anfragesprache.

3. Ein Beispiel für eine Datenbank mit XML-Unterstützung ist Oracle9i. Insbesondere, wenn eine Schema-Beschreibung zu einer Klasse von Dokumenten existiert, stellt der dort angebotene XMLType eine effiziente Alternative zum Archivieren von XML-Dokumenten dar. Anfragen und Operationen zum Ändern werden über eine XPath-Anfrageschnittstelle zur Verfügung gestellt.

4. Ein generisches Speicherungsmodell bietet den Vorteil, dass bei der Änderung der Schema-Beschreibung einer Klasse von Dokumenten, die in der Datenbank gespeichert wurden, das Datenbankschema konstant bleibt.

5. Die Unterstützung von Update-Operationen muss als wesentlicher Bestandteil bei der Konzeptionierung eines Speicherungsmodells berücksichtigt werden. Nur dadurch ist ein nachträgliches Ändern von Teilen der abgespeicherten Dokumente möglich.

6. Das XML Information Set kann dazu verwendet werden, Informationen bezüglich der Struktur von XML-Dokumenten, die für eine Speicherung relevant sind, herzuleiten.

7. Mithilfe von objektrelationalen Konzepten lassen sich effiziente Speicherungsstrukturen zum Abspeichern von XML-Dokumenten umsetzen.

8. Zum Modellieren von Verwandtschaftbeziehungen zwischen den Knoten eines XML-Dokumentes ist es ausreichend, einen Verweis auf den übergeordneten Knoten abzuspeichern. Damit lassen sich Anfrageoperationen effizient umsetzen. Der Aufwand, mit dem Änderungsoperationen durchgeführt werden müssen, minimiert sich.

9. Verwandtschaftsbeziehungen lassen sich in Oracle9i durch Referenzen modellieren.

10. Werden XML-Attribute direkt als Eigenschaft des jeweiligen Elementes abgelegt, wird eine Reduzierung der zu speichernden Datenmenge erreicht, da Verwandtschaftsbeziehungen zwischen Elementen und Attributen nicht explizit modelliert werden müssen.

11. XQuery-Pfadausdrücke dienen der Informationsgewinnung. Alle anderen Bestandteile der XML-Anfragesprache dienen der Weiterverarbeitung der gewonnenen Informationen. Aus diesem Grund sind nur Pfadausdrücke relevant für eine Anfragebearbeitung auf Seiten eines Datenbankservers.

12. Mithilfe des in dieser Arbeit entwickelten Speicherungsmodells lassen sich beliebige Dokumente in einer Oracle9i-Datenbank ablegen. Die diskutierten Operationen wurden teilweise prototypisch implementiert und können zur Beantwortung von Anfragen bzw. zum Ändern der gespeicherten Daten verwendet werden.

13. Das entwickelte Speichermodell ist offen für ergänzende Erweiterungen. Je nach Bedarf können beispielsweise zu den einzelnen Knotentypen eines XML-Dokumentes weitere Eigenschaften in Form von zusätzlichen Attributen vereinbart werden.