Feingranulare Transaktionsisolation in nativen XML-Datenbanksystemen

Michael P. Haustein

Titelfoto
Julia Scheel

Fotoarchiv der
Technischen Universität Kaiserslautern

Feingranulare Transaktionsisolation in nativen XML-Datenbanksystemen

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation
von

Dipl.-Inform. Michael P. Haustein

Dekan des Fachbereichs Informatik
Prof. Dr. H. Hagen

Promotionskommission
Vorsitzender: Prof. Dr. J. Schmitt
Berichterstatter: Prof. Dr. Dr. T. Härder, Prof. Dr. N. Ritter

Datum der wissenschaftlichen Aussprache
22. Dezember 2005

D 386

Für Sandra und unsere Tochter Lena Marie

Danksagung

Zunächst möchte ich meine Eltern erwähnen, die mir nach dem Abitur und der Pflichterfüllung für unser Land das Informatikstudium an der Universität Kaiserslautern ermöglicht haben. Sie haben in dieser Zeit einige finanzielle Opfer erbracht, wofür ihnen mein besonderer Dank gebührt. Weiterhin möchte ich mich besonders bei meiner Frau Sandra bedanken, die viel Verständnis für meine Arbeit aufgebracht und auch die zahlreichen Tabellen im Anhang gegen die Originale im Quellcode abgeglichen hat. Sie trainiert mich mit viel Liebe und weiblicher Intuition in enger Zusammenarbeit mit unserer Tochter Lena Marie täglich in der Kunst der Kommunikation und Stressbewältigung in nahezu allen informatikfremden und nicht weniger wichtigen Aspekten des Lebens.

Diese Arbeit ist während meiner Tätigkeit als wissenschaftlicher Mitarbeiter in der Arbeitsgruppe „Datenbanken und Informationssysteme“ (AG DBIS) von Prof. Dr. Dr. Theo Härder an der Technischen Universität Kaiserslautern von Juli 2002 bis Dezember 2005 entstanden. Mit ein Grund für die Wahl dieser Stelle war die hervorragende Betreuung durch Dr.-Ing. Wolfgang Mahnke und Prof. Dr. Norbert Ritter während meiner Zeit als Student im Sonderforschungsbereich 501 und durch Wolfgang Mahnke während meiner anschließenden Diplomarbeit. Mein besonderer Dank gilt meinem Doktorvater Theo Härder. Ich möchte ihm für alle gewährten Freiheiten danken, welche es mir ermöglichten, meine Ideen im Rahmen des XTC-Projekts umzusetzen. Er kennt wie kein anderer die politischen Netzwerke in der Datenbankszene und weiß sofort, welche Ergebnisse einer Forschungsarbeit weiter zu untersuchen sind oder genug für eine Publikation beinhalten. Ich danke ihm für alle Unterstützung bei meiner Arbeit und den Veröffentlichungen, für die trotz seines vollen Terminkalenders unglaublich schnell redigierten Kapitel meiner Dissertation und hoffe, dass er mir die Kündigung und den Schritt in den harten Alltag der Wirtschaft verzeiht.

Weiterhin möchte ich meinen Kollegen Andreas Bühmann, Prof. Dr. Stefan Deßloch, Philipp Dopichaj, Jürgen Göres, Christian Mathis und Boris Stumm fürs Korrekturlesen und alle Anregungen zur Verbesserung dieser Arbeit danken. Besonders hervorheben möchte ich Christian Mathis, den ich zunächst als Diplomand betreut habe, und der nun als Promotionsstudent in der Arbeitsgruppe das XTC-Projekt fortführt. Auch den ehemaligen Studenten Georges Berscheid, Konstantin Luttenberger, Juliane Neumann, Burkhard Schäfer und Markus Wagner, die mit Projekt- und Diplomarbeiten entscheidende Teile zum Gelingen meiner Arbeit beigetragen haben, gilt es in diesem Zusammenhang zu danken.

Natürlich darf in der Danksagung einer Dissertation der AG DBIS die Huldigung der Tee-Ecke nicht fehlen, die Generationen von Doktoranden in politisch korrektem Smalltalk und gänzlich intuitivem Abblocken jeglicher Versuche von Mobbing geschult hat. Da ich als Student und Mitarbeiter an dieser berüchtigten Tee-Ecke eine Art Generationenwechsel erlebt habe, möchte ich es mir nicht nehmen lassen, auch die großen Namen aller ehemaligen Kollegen zu erwähnen. Hier sind Markus Bon, Marcus Flehmig, Jernej Kovse, Henrik Löser, Ulrich Marder und Hans-Peter Steiert zu nennen.

Für die technische Unterstützung während meiner Arbeit und Klärung aller Fragen in den teilweise krankhaften Abläufen der Verwaltung der Universität möchte ich Lothar Gauß und Manuela Burkart danken, die stets bemüht sind, die Informatiker der Arbeitsgruppe vor der völligen Lebensuntüchtigkeit zu bewahren.

Schließlich gilt mein Dank auch Prof. Dr. Norbert Ritter, der trotz des engen Zeitplans die Aufgaben des Zweitgutachters übernommen hat, und Prof. Dr. Jens Schmitt für den Vorsitz der Prüfungskommission.

Michael Haustein

Kaiserslautern, im Dezember 2005

Kurzfassung

Die *Extensible Markup Language* (XML) ist eine Auszeichnungssprache zur Strukturierung von Textdokumenten. Da diese Dokumente (sog. XML-Dokumente) im Gegensatz zu binären Dateien problemlos zwischen Anwendungen verschiedener Betriebssysteme und Rechnerarchitekturen ausgetauscht werden können, etablierten sich XML und die damit verbundenen Technologien schon kurz nach ihrer Standardisierung 1998 durch das *World Wide Web Consortium* (W3C) als Standard für den Datenaustausch bei der Anwendungsintegration.

Seitdem wird XML nicht nur im Bereich der Systemkommunikation, sondern zunehmend auch für die plattformübergreifende Datenhaltung eingesetzt. Die stetig wachsende Zahl von XML-Dokumenten und die zunehmenden Dokumentgrößen erfordern den Einsatz von Datenbanksystemen, um die Dokumente an einer zentralen Stelle zu verwalten und mit einheitlichen Schnittstellen zu verarbeiten. Die zur Zeit verfügbaren Lösungen sind als Erweiterungen für relationale Datenbanksysteme realisiert oder stellen auf XML-Datenstrukturen zugeschnittene Speicherungsverfahren (native XML-Datenbanksysteme) zur Verfügung. Der erste Ansatz erzwingt die Abbildung der XML-Dokumente auf ein relationales Datenbankschema (ein vollkommen anderes Datenmodell), Systeme des zweiten Ansatzes sind meist nur für den Lesezugriff optimiert und erzwingen bei Datenänderungen die Blockierung des gesamten Dokuments zur Konsistenzsicherung. Beide Ansätze werden den Anforderungen für die Verwaltung von XML-Daten im Mehrbenutzerbetrieb nicht gerecht.

Diese Arbeit untersucht daher insbesondere die folgenden Aspekte der nativen XML-Datenverarbeitung zur Realisierung eines XML-Datenbanksystems, das einen hohen Grad an Parallelität bei der Datenverarbeitung im Mehrbenutzerbetrieb ermöglicht:

- Datenmodelle. XML-Dokumente müssen auf ein Datenmodell abgebildet werden, das vom Datenbanksystem zur Repräsentation, Auswertung und Manipulation der Daten in der Datenbank eingesetzt wird.
- Speicherungsstrukturen. Um die Instanzdaten des verwendeten Datenmodells persistent ablegen zu können, muss ein Datenbanksystem zugeschnittene Speicherungsstrukturen implementieren, die einen schnellen wahlfreien Zugriff und die Datenmodifikation durch parallel ablaufende Transaktionen unterstützen.
- Transaktionsisolation. Im Mehrbenutzerbetrieb muss ein Datenbanksystem für jede laufende Transaktion den logischen Einbenutzerbetrieb simulieren. Dazu sind auf das Datenmodell abgestimmte Synchronisationsmechanismen erforderlich, die konkurrierende Zugriffe voneinander isolieren.
- Anwendungsprogrammierschnittstellen. Zum Zugriff auf die vom Datenbanksystem verwalteten Daten muss für Anwendungsprogramme eine Schnittstelle zur Verfügung gestellt werden, die alle XML-typischen Operationen transaktionssicher implementiert.

Um die Lösungsansätze für die genannten Aspekte zu bewerten, wurde im Rahmen dieser Arbeit prototypisch ein natives XML-Datenbanksystem entwickelt, das die erarbeiteten Konzepte implementiert und detaillierte Leistungsmessungen ermöglicht.

Inhaltsverzeichnis

KAPITEL 1 Einleitung

Es ist ein großer Vorteil im Leben, die Fehler, aus denen man lernen kann, möglichst frühzeitig zu machen.
(Winston Churchill)

Die *Extensible Markup Language* (XML) ist eine *Auszeichnungssprache* für Textdokumente. Auszeichnungssprachen bezeichnen Computersprachen zur Strukturierung (und evtl. auch Formatierung) von Dokumenten. Ursprünglich entstammen Auszeichnungssprachen dem Umfeld der Textverarbeitungssysteme, für die es erforderlich war, zusätzliche Informationen über das Layout an den Textinhalt zu binden. Es wurde jedoch schnell klar, dass es in der automatisierten Verarbeitung von Texten sehr nützlich ist, über die Formatierung hinausgehende Informationen (z. B. Aussagen über den Text selbst) in Textdokumente einzubetten. Dazu wurden allgemeinere Auszeichnungssprachen entwickelt, was zunächst zur *Generalized Markup Language* (GML) für die Strukturierung von Texten in Kapitel, Absätze usw. und schließlich 1986 zur Standardisierung der *Standard Generalized Markup Language* (SGML) [SGML86] führte.

SGML erlaubt dem Anwender sehr viele Freiheitsgrade, was möglicherweise die Entwicklung von Software zur Verarbeitung von SGML-Dokumenten entsprechend aufwändig und teuer macht. Ausgehend von SGML begann daher 1996 eine Arbeitsgruppe des *World Wide Web Consortium* (W3C) mit der Entwicklung von XML. Ziel war es, die Komplexität von SGML zu reduzieren, trotzdem dazu kompatibel zu bleiben und ergänzend zum Bereich der Textverarbeitung ein breites Spektrum von Anwendungsgebieten zu erschließen. 1998 wurde XML schließlich vom W3C als *Recommendation* in der Version 1.0 [BPS98] standardisiert.

1.1 Motivation und Zielsetzung

XML hat durch die bereits im Standard berücksichtigte Erweiterbarkeit schnell eine Vielzahl von Anwendungsgebieten erschlossen. Da sich XML-Dokumente im Gegensatz zu binären Dateien oder proprietären Nachrichtenformaten einfach zwischen verschiedenen Betriebssystemen und Rechnerarchitekturen austauschen lassen und normierte Schnittstellen für fast alle Plattformen verfügbar sind, hat sich XML bereits kurz nach der Standardisierung als Bindeglied in der Anwendungsintegration etablieren können.

Seitdem wird XML nicht nur zum systemübergreifenden Nachrichtenaustausch eingesetzt, sondern erobert zunehmend auch das Gebiet der Datenhaltung [Fl05]. So können mit Hilfe von XML-Dokumenten Daten anwendungsunabhängig verwaltet und mit speziellen Transformationsregeln in jedes beliebige anwendungsspezifische Format umgewandelt werden. Die damit verbundene Problematik der Verwaltung einer großen Datenbasis in einzelnen Dokumenten lehrt uns jedoch die Geschichte, denn vor der Einführung von Datenbanksystemen in den 70er Jahren wurden Daten auf ähnliche Weise in zahlreichen Dateien meist verteilt, inkonsistent und redundant gespeichert. Die langfristige Lösung zur Verwaltung von XML-Dokumenten findet sich folglich im Einsatz von Datenbanksystemen, die in der Lage sind, den zeitgleichen Zugriff mehrerer Benutzer zu koordinieren und dabei die Datenkonsistenz zu sichern.

Dies haben auch Forschungsgruppen und die Hersteller kommerzieller Datenbanksysteme erkannt, sodass Lösungsansätze und Produkte zur Erweiterung der Systeme für die Verarbeitung von XML-Daten publiziert und angeboten werden. Die generelle Problematik all dieser Ansätze liegt darin, dass XML-Dokumenten ein anderes Datenmodell als den (meist relationalen) Datenbanksystemen zugrunde liegt, sodass XML-Dokumente für die Verwaltung in einem Datenbanksystem zunächst in dessen Datenmodell überführt und für die Ausgabe wieder zurück transformiert werden müssen. Diese Transformation ist natürlich in derselben Weise auch für jede XML-Schnittstelle oder -Anfragesprache durchzuführen, da Datenbanksysteme oft nur Anfragesprachen bzgl. ihres eigenen Datenmodells anbieten. Zudem sind innerhalb eines Datenbanksystems alle erforderlichen Maßnahmen zur Transaktionsverwaltung auf das zugrunde liegende Datenmodell ausgerichtet, sodass für die aus externer Sicht durchgeführten Arbeitsschritte auf einem anderen Datenmodell eine Reihe von Nachteilen entstehen [Ph04].

Um diese Probleme in den Griff zu bekommen, werden so genannte *native* XML-Datenbanksysteme entwickelt, deren gesamte Systemarchitektur auf ein XML-Datenmodell abgestimmt ist. Diese Systeme sind allerdings meist bis auf das Speichern, Löschen und Ersetzen ganzer Dokumente auf rein lesenden Zugriff optimiert. Die Modifikation von Dokumentteilen führt in einer Mehrbenutzerumgebung fast immer zur Blockierung des gesamten Dokuments. Mit ein Grund für die durchweg schlechte Unterstützung von nebenläufigen feingranularen Modifikationen ist der Standardisierungsprozess der verbreiteten deklarativen XML-Anfragesprache XQuery, für die trotz der Bemühungen seit 1999 noch immer kein Standard verabschiedet wurde und auch der momentane Entwurf keinerlei Änderungsoperatoren enthält. Somit bieten zur Zeit weder native noch relationale Datenbanksysteme (mit den entsprechenden Erweiterungen) Unterstützung für die Transaktionsverarbeitung mit XML-Daten in einer Mehrbenutzerumgebung, wie dies bspw. für relationale Daten in Datenbanksystemen möglich ist.

Gegenstand dieser Arbeit ist daher die Untersuchung von Lösungsansätzen zur Realisierung eines nativen XML-Datenbanksystems, das einen hohen Grad an Parallelität für die transaktionssichere Verarbeitung von XML-Daten im Mehrbenutzerbetrieb ermöglicht. Dafür sind die folgenden Aspekte von besonderer Bedeutung:

- Datenmodelle.
  Die zu verwaltenden XML-Dokumente müssen auf ein Datenmodell abgebildet werden, das vom Datenbanksystem zur Repräsentation, Auswertung und Manipulation der Daten in der Datenbank eingesetzt wird. Das Datenmodell muss alle XML-typischen Operationen effizient ohne weitere Transformationsschritte unterstützen.

- Speicherungsstrukturen.
  Um die Instanzdaten des verwendeten Datenmodells persistent auf Massenspeichern ablegen zu können, muss ein Datenbanksystem auf XML-Dokumente zugeschnittene Speicherungsstrukturen implementieren, die einen schnellen wahlfreien Zugriff und die Datenmodifikation durch parallel ablaufende Transaktionen unterstützen.

- Transaktionsisolation.
  Im Mehrbenutzerbetrieb muss ein Datenbanksystem für jede laufende Transaktion den logischen Einbenutzerbetrieb simulieren. Dazu sind auf das Datenmodell abgestimmte Synchronisationsmechanismen erforderlich, die konkurrierende Zugriffe voneinander isolieren. Aus Leistungsgründen kann es für die Datenverarbeitung erforderlich sein, diese Eigenschaft für eine höhere Parallelität aufzuweichen, sodass ein Datenbanksystem für Transaktionen verschiedene Isolationsstufen anbieten sollte.

- Anwendungsprogrammierschnittstellen.
  Zum Zugriff auf die vom Datenbanksystem verwalteten Daten muss für Anwendungsprogramme eine Schnittstelle zur Verfügung gestellt werden, die alle XML-typischen Operationen implementiert. Dabei müssen mehrere Operationen innerhalb eines gemeinsamen Transaktionskontexts ausführbar sein, für den die ACID-Eigenschaften [GR93] garantiert werden.

Um die Lösungsansätze für die genannten Aspekte zu bewerten, wurde im Rahmen dieser Arbeit prototypisch ein natives XML-Datenbanksystem entwickelt, das die erarbeiteten Konzepte implementiert und detaillierte Leistungsmessungen ermöglicht.

1.2 Gliederung der Arbeit

Im Folgenden beschäftigen wir uns zunächst mit den Grundlagen zu dieser Arbeit. Dazu erläutert Kapitel 2 die Konzepte der Auszeichnungssprache XML und die damit in Beziehung stehenden Schnittstellen, Anfragesprachen und Technologien. Kapitel 3 beleuchtet die bereits auf dem Gebiet der XML-Datenverarbeitung veröffentlichten Arbeiten. Diese betreffen vor allem die Systemarchitekturen einiger nativer XML-Datenbanksysteme, Nummerierungsschemata, die zur Realisierung von Speicherungsstrukturen benötigt werden, und Ansätze zur Isolation nebenläufiger Transaktionen, die auf XML-Daten operieren.

Nachdem mit der Betrachtung der Grundlagen und der verwandten Arbeiten die zu lösenden Probleme bei der nativen XML-Datenverarbeitung verdeutlicht wurden, beschreibt Kapitel 4 ein *Transaktionsmodell*, das als Basis aller vorgestellten Konzepte dieser Arbeit dient. Das Transaktionsmodell besteht aus vier wesentlichen Komponenten. Zur Repräsentation von XML-Dokumenten wird ein *Datenmodell* eingeführt, auf dem *Basisoperationen* zum Zugriff auf die verwalteten Dokumente definiert werden. Um einzelne Komponenten innerhalb des Datenmodells ansprechen zu können, beinhaltet das Transaktionsmodell ein *Adressierungsverfahren*. Transaktionen, die die Basisoperationen für parallele Zugriffe auf das Datenmodell nutzen, werden mit einem *Isolationsmechanismus* synchronisiert. Die Transaktionsisolation wird detailliert mit der Berücksichtigung verschiedener Konsistenzstufen in Kapitel 5 behandelt.

Um die vorgestellten Konzepte in einer realen Laufzeitumgebung auszuwerten, wurde prototypisch ein natives XML-Datenbanksystem entwickelt. Für das Leistungsverhalten wichtige Details dieser Implementierung sind Inhalt von Kapitel 6.

Nachdem damit alle Aspekte für die Speicherung und den parallelen Zugriff auf XML-Dokumente beschrieben sind, erfolgt in Kapitel 7 deren Auswertung anhand konkreter Messergebnisse. Dazu werden zunächst einige Varianten zur Realisierung der Speicherungsstrukturen untersucht, danach erfolgen Leistungsmessungen für das gesamte System in einer verteilten Testumgebung, um den Einfluss der vorgestellten Isolationsmechanismen auf den Transaktionsdurchsatz zu beurteilen.

Kapitel 8 fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf noch zu untersuchende Fragestellungen.

KAPITEL 2 XML – Grundlagen und Technologien

Fortschritt – das bedeutet, dass wir unsere alten Sorgen gegen neue eintauschen.
(Bertrand Russell)

Dieses Kapitel widmet sich den Grundlagen der Extensible Markup Language [BPS+04] und den damit verbundenen Technologien, die im Rahmen dieser Arbeit relevant sind. Wir betrachten zunächst kurz den grundsätzlichen Aufbau von XML-Dokumenten und die Möglichkeiten, Schemata mit den Schemabeschreibungssprachen *Document Type Definition* und *XML Schema* zu definieren. Schließlich erläutern wir die Kernkonzepte der Anwendungsschnittstellen *SAX*, *DOM*, *XQuery* und einiger Spracherweiterungen sowie XML-basierte Techniken zum Transport der Daten zwischen Anwendungen und Datenbanksystemen.

2.1 XML-Dokumente

Wie einfach XML-Grundkonstrukte anzuwenden sind, zeigt Abbildung 1, die bereits ein vollständiges XML-Dokument mit den Daten eines Bankkunden darstellt. Ein XML-Dokument ist ein einfaches Textdokument, vor dessen eigentlichem Inhalt der *Prolog* steht, der mit der Zeichenfolge *<?xml* eingeleitet und mit *?>* abgeschlossen wird. Im Prolog ist die Version der zugrunde liegenden XML-Spezifikation (hier *1.1*) und die Zeichenkodierung des Dokuments (hier *UTF-8*) angegeben.

```
<?xml version="1.1" encoding="UTF-8"?>
<Bank>
    <Kunden>
        <Kunde id="kd1606">
            <Name>
                <Vorname>Michael</Vorname>
                <Vorname>Peter</Vorname>
                <Nachname>Haustein</Nachname>
            </Name>
            <Adresse>
                <Straße>Gottlieb-Daimler-Straße</Straße>
                <Hausnummer/>
                <PLZ>D-67663</PLZ>
                <Ort>Kaiserslautern</Ort>
            </Adresse>
        </Kunde>
    </Kunden>
</Bank>
```

Abbildung 1: Ein einfaches XML-Dokument

Optional kann der Prolog auch eine Schemabeschreibung enthalten oder referenzieren (siehe Abschnitt 2.2.1), mit der eine Struktur des Dokuments definiert wird. Entspricht diese Struktur den im Folgenden vorgestellten Syntaxregeln, so wird das Dokument als *wohlgeformt* bezeich-

net. Stimmt die Dokumentenstruktur zusätzlich auch noch mit der in einer Strukturbeschreibung festgelegten Vorschrift überein (siehe Abschnitt 2.2), so wird das Dokument bzgl. dieser Beschreibung als *gültig* bezeichnet. Die Überprüfung dieser Eigenschaften durch einen XML-Prozessor nennt man *Validierung*. Ein Ausschnitt aus einem XML-Dokument, der bis auf den fehlenden Prolog wohlgeformt ist, heißt *XML-Fragment*. Die Inhalte von Dokumenten bzw. Fragmenten können weiterhin bzgl. ihrer Struktur klassifiziert werden. Ist ein Dokument wie in Abbildung 1 sehr detailliert und regelmäßig strukturiert, so spricht man von einem *datenorientierten* Dokument. Enthält das Dokument eher unregelmäßige und grob strukturierte Inhalte (bspw. ein Aufsatz mit Formatierungsinformation), so wird dies als *dokumentenorientiert* bezeichnet.

Der Inhalt eines XML-Dokuments wird mit so genannten *Tags* strukturiert, die in spitze Klammern eingeschlossen sind und dadurch *Elemente* innerhalb des Dokuments kennzeichnen. Dabei wird zwischen Groß- und Kleinschreibung der Elementnamen unterschieden. Tags der Gestalt <...> werden als *öffnende Tags* oder *Start-Tags* bezeichnet, Tags der Gestalt </...> als *schließende Tags* oder *End-Tags*.

2.1.1 Elemente

Der Inhalt eines *Elements* wird durch dessen öffnendes und schließendes Tag begrenzt. Jedes XML-Dokument besitzt genau ein *Wurzelelement*, das direkt auf den Prolog folgt. Elemente können neben textuellem Inhalt wiederum Elemente enthalten (so genannte *Kindelemente*), sodass es möglich ist, innerhalb des Dokuments eine beliebige Schachtelungstiefe von Elementen zu erreichen. Umgekehrt hat jedes Element mit Ausnahme des Wurzelelements genau ein *Elternelement*, wodurch ein XML-Dokument eine Baumstruktur beschreibt. XML-Dokumente sind *ordnungserhaltend*, das heißt, die Reihenfolge der Elemente ist relevant und muss bei der Weiterverarbeitung des Dokuments stets sichergestellt werden.

XML-Elemente können zusätzlich mit einem Namensraum (*namespace*) annotiert werden [BHL99], der durch einen Doppelpunkt getrennt vor den eigentlichen Namen geschrieben wird, um semantische Namenskollisionen verschiedener Domänen zu verhindern. *Namespaces* spielen allerdings in dieser Arbeit keine Rolle und werden daher nicht weiter betrachtet.

Besitzt ein Element nur textuellen Inhalt, so spricht man von *Textinhalt,* enthält es ausschließlich eine Liste von Kindelementen, so wird dies als *strukturierter Inhalt* bezeichnet. Ein Element, das sowohl Elemente als auch Texte enthält, wird durch *gemischten Inhalt* klassifiziert. Elemente können jedoch auch ohne Inhalt im Dokument auftreten (siehe *<Hausnummer/>* in Abbildung 1), wobei <.../> als Abkürzung für <...></...> verwendet werden darf.

2.1.2 Texte

Der textuelle Inhalt eines XML-Elements wird durch Niederschrift der entsprechenden Zeichenfolge zwischen dem öffnenden und schließenden Tag des Elements angegeben. Dieser Text kann jedoch wiederum besondere Zeichen der XML-Syntax (wie z. B. die Zeichen < oder >) enthalten. Um solche Textteile als Inhalt eines Elements anzugeben, müssen diese in so genannte *CDATA-Abschnitte* eingebettet werden, die mit *<![CDATA[* beginnen und mit *]]>* enden. Der Inhalt eines CDATA-Abschnitts darf dann von einem XML-Prozessor nicht interpretiert werden. Die folgende Abbildung 2 zeigt deren Anwendung.

```
<Beschreibung>
    <![CDATA[Hier werden die <Klammern> nicht verarbeitet.]]>
</Beschreibung>
```

Abbildung 2: CDATA-Abschnitt

2.1.3 Attribute

Zusätzlich zu den oben beschriebenen Möglichkeiten der Verschachtelung eines Elements mit weiteren Elementen und Texten kann ein XML-Element beliebig viele Attribute besitzen. Diese werden mit Namen und jeweiligem Wert direkt innerhalb des Element-Tags angegeben (z. B. *id="kd1606"* in Abbildung 1).

Der Attributwert wird in einfache oder doppelte Anführungszeichen eingeschlossen und dem Attributnamen mit einem Gleichheitszeichen zugewiesen. Mehrere Attributdefinitionen werden innerhalb des Element-Tags durch Leerzeichen getrennt. Dabei ist zu beachten, dass ein Attribut für ein Element nur einmal definiert werden darf und die Reihenfolge der Attribute im Gegensatz zu der der Elemente irrelevant ist.

2.1.4 Verarbeitungsanweisungen und Kommentare

An beliebigen Stellen (außer zwischen öffnenden und schließenden spitzen Klammern, mit denen Elemente definiert werden) kann in einem XML-Dokument eine Verarbeitungsanweisung stehen, die mit den Zeichen <? und ?> umschlossen wird. Nach dem öffnenden <? folgt zunächst das so genannte *Ziel*, danach der *Anweisungsteil*. Ein Beispiel [BDG+01] für eine Verarbeitungsanweisung ist in Abbildung 3 gegeben.

```
<?xml-stylesheet href="mystyle.css" type="text/css"?>
```

Abbildung 3: Verarbeitungsanweisung in einem XML-Dokument

Eine Anwendung erhält von einem XML-Prozessor die Information *xml-stylesheet* als Target und die Zeichenfolge *href="mystyle.css" type="text/css"* als Anweisungsteil. Die Interpretation und weitere Verarbeitung dieser Information ist Aufgabe der Anwendung und hat nichts mit dem XML-Standard zu tun. Es ist auch nicht spezifiziert, welcher Syntax der Anweisungsteil zu entsprechen hat. Es empfiehlt sich aber zur intuitiveren nachfolgenden Verarbeitung [BDG+01] das auch hier verwendete Attribut-Wert-Muster zu benutzen.

Analog zu Verarbeitungsanweisungen können in ein XML-Dokument Kommentare aufgenommen werden. Sie werden durch die Zeichenfolge <!-- eingeleitet und mit --> abgeschlossen. Kommentare gehören nicht zum Textinhalt eines XML-Dokuments. Der XML-Standard [BPS+04] erlaubt einem XML-Prozessor, Operationen zum Auslesen der Kommentare zur Verfügung zu stellen, schreibt es allerdings nicht vor. Somit können Kommentare von einem standardkonformen Prozessor auch ignoriert werden.

2.1.5 XML-Vokabulare

Da in XML-Dokumenten typischerweise dieselben Element- und Attributnamen sehr oft auftreten (vgl. die Elemente in Abbildung 1 für bspw. 10.000 Kunden), besteht in einem Datenbanksystem eine mögliche Optimierung darin, jedem Element- bzw. Attributtyp einen eindeutigen Zahlenwert zuzuweisen. Zur Speicherung des Dokuments wird dann statt dem eigentlichen Namen des Elements bzw. Attributs der gewählte Zahlenwert verwendet, der sehr viel platzsparender verwaltet werden kann. Die auf diese Weise kodierten und in der Datenbank gespeicherten Bezeichner für Elemente und Attribute bezeichnet man als *Vokabular*.

In manchen Fällen bezeichnet man mit einem XML-Vokabular auch die reinen Element- und Attributnamen (ohne entsprechende Kodierung), wenn Daten von einem anderen Datenmodell (bspw. dem relationalen) in eine XML-Repräsentation transformiert werden sollen [Sc03]. Im Rahmen dieser Arbeit wird jedoch die obige Definition zugrunde gelegt.

2.2 Definition von XML-Dialekten

Die gemeinsame Strukturbeschreibung für eine Klasse von XML-Dokumenten wird als *XML-Dialekt* bezeichnet. Neben Ansätzen wie RelaxNG [CM01] oder Schematron [Sch04] sind die *Document Type Definition* und *XML Schema* am weitesten verbreitet, um XML-Dialekte zu definieren. Daher betrachten wir diese beiden Schemabeschreibungssprachen im Folgenden etwas genauer.

2.2.1 Document Type Definition

Mit einer Document Type Definition (DTD) [BPS+04] kann die Struktur eines XML-Dokuments erzwungen werden. Das hat den Vorteil, dass eine Anwendung, deren XML-Prozessor ein XML-Dokument bzgl. seiner DTD als gültig eingestuft hat, keine Vorkehrungen mehr für abweichende Strukturen treffen muss.

Die Formulierung einer DTD beginnt mit *<!DOCTYPE,* gefolgt vom Namen des Dokumententyps und enthält danach in eckigen Klammern die erlaubte Struktur, welche sich aus Element- und Attributtypen zusammensetzt. Beendet wird die Document Type Definition mit einer schließenden spitzen Klammer.

Jeder Elementtyp wird mit *<!ELEMENT* und einem Namen eingeleitet, danach folgt, in runden Klammern eingeschlossen, das so genannte *Inhaltsmodell*. Ein Element kann aus nicht genauer definierten Daten bestehen (*#PCDATA – parsed character data*), wiederum weitere Elementtypen beinhalten oder durch eine Mischung aus beidem aufgebaut sein. Ein leeres Element (vgl. *Hausnummer* in Abbildung 1) kann mit dem Inhaltsmodell *EMPTY* erzwungen werden. In diesem Fall ist für eine Anwendung nur die Existenz des Elements von Bedeutung. Für die enthaltenen Elementtypen kann deren Kardinalität durch die übliche Notation (? höchstens einmal, + mindestens einmal, * beliebig oft oder auch gar nicht) festgelegt werden. Ohne Angabe einer Kardinalität muss ein angegebener Elementtyp genau einmal vorkommen. Sind die angegebenen Elementtypen durch ein Komma getrennt, so müssen sie in dieser Reihenfolge auftreten (*sequence*), sind sie durch einen senkrechten Strich getrennt, so kann unter den Elementtypen ausgewählt werden (*choice*).

Die für einen Elementtyp zulässigen Attribute werden durch *<!ATTLIST* eingeleitet. Darauf folgt der Name des betroffenen Elementtyps, der Name des Attributs, sein Typ und evtl. dessen Kardinalität und Vorgabewert. Für Attributtypen stehen einfache Zeichenketten (*CDATA*), ein Identifikatortyp (*ID*), eine Referenz bzw. eine Liste von Referenzen auf einen Identifikator (*IDREF* bzw. *IDREFS*), ein einfaches *Name Token* bzw. eine Liste von Name Tokens (*NMTOKEN* bzw. *NMTOKENS* – eine Folge von ausgewählten Zeichen ohne das Leerzeichen) oder ein Aufzählungstyp, dessen mögliche Werte durch senkrechte Striche getrennt sind, zur Verfügung. Da Attribute bzgl. eines Elements eindeutig sein müssen, beschränkt sich die Kardinalität auf „vorhanden“ bzw. „nicht vorhanden“. Um dies festzulegen, existieren die Schlüsselwörter *#REQUIRED* (das Attribut muss angegeben werden), *#IMPLIED* (das Attribut darf fehlen und erhält dann keinen Vorgabewert) und *#FIXED* (das Attribut darf nur den angegebenen Vorgabewert erhalten).

Das Beispieldokument aus Abbildung 1 wird durch die in Abbildung 4 definierte DTD beschrieben. Das Wurzelelement *Bank* muss genau ein Element vom Typ *Kunden* enthalten, das wiederum beliebig viele *Kunde*-Elemente enthalten darf. Ein Element vom Typ *Kunde* besitzt ein ID-Attribut mit Namen *id* und besteht aus den Elementtypen *Name* und *Adresse*. *Name* enthält mindestens einen *Vornamen* und genau einen *Nachnamen*. Die Elemente *Vorname* und *Nachname* enthalten jeweils Zeichenfolgen mit den entsprechenden Werten. Eine *Adresse* besteht aus einer *Straße*, einer optionalen *Hausnummer*, einer Postleitzahl (*PLZ*) und einem *Ort,*

wobei alle Elemente wiederum Zeichenfolgen enthalten. Im XML-Dokument in Abbildung 1 ist die Hausnummer zwar angegebenen, allerdings als leeres Element ohne Inhalt, was durch die DTD nicht verhindert werden kann. Die Tatsache, dass der Wert für eine Postleitzahl in einem gültigen XML-Dokument einem gegebenen Muster entsprechen sollte, kann mit der Document Type Definition ebenfalls nicht festgelegt werden; hier liegen die Stärken von XML Schema (siehe dazu Abschnitt 2.2.2).

```
<!DOCTYPE Bank [
    <!ELEMENT Bank (Kunden)>
    <!ELEMENT Kunden (Kunde*)>
    <!ELEMENT Kunde (Name,Adresse)>
    <!ATTLIST Kunde id ID #REQUIRED>
    <!ELEMENT Name (Vorname+,Nachname)>
    <!ELEMENT Vorname (#PCDATA)>
    <!ELEMENT Nachname (#PCDATA)>
    <!ELEMENT Adresse (Straße,Hausnummer?,PLZ,Ort)>
    <!ELEMENT Straße (#PCDATA)>
    <!ELEMENT Hausnummer (#PCDATA)>
    <!ELEMENT PLZ (#PCDATA)>
    <!ELEMENT Ort (#PCDATA)>
]>
```

Abbildung 4: DTD für das Beispieldokument

Die Document Type Definition kann sowohl innerhalb eines XML-Dokuments (*intern*) als auch in einer separaten Datei (*extern*) stehen. Im ersten Fall wird die gesamte Definition zwischen Prolog und Wurzelelement eingefügt, im zweiten Fall wird an dieser Position eine Referenz auf eine Datei, die eine DTD enthält, angegeben. Angenommen, die in Abbildung 4 eingeführte DTD wird in der Datei *bank.dtd* abgelegt, so zeigt Abbildung 5, wie innerhalb des Beispieldokuments auf diese DTD mit dem Schlüsselwort *SYSTEM* verwiesen werden kann.

```
<?xml version="1.1" encoding="UTF-8"?>
<!DOCTYPE Bank SYSTEM "bank.dtd">
<Bank>
    <Kunden>...</Kunden>
</Bank>
```

Abbildung 5: Referenz auf eine externe Document Type Definition

ID-Attribute

Mit der Verwendung von ID-Attributen können Elemente in einem XML-Dokument eindeutig gekennzeichnet werden. Der Wert eines Attributs, das mittels einer DTD als ID-Attribut ausgewiesen ist, darf dabei innerhalb des gesamten XML-Dokuments höchstens einmal vorkommen, unabhängig vom Elementtyp. Im Beispieldokument aus Abbildung 1 wird für das Element *Kunde* solch ein ID-Attribut mit Namen *id* und dem Wert *kd1606* definiert.

Um den Wert eines ID-Attributs zu referenzieren, wird an einer anderen Stelle im XML-Dokument ein referenzierendes Attribut benutzt, dem die DTD den Typ *IDREF* zuweist. Damit das Dokument von einem XML-Prozessor validiert wird, muss jedes referenzierende Attribut einen Wert besitzen, der innerhalb des Dokuments einem ID-Attribut zugewiesen ist. Der XML-Prozessor sorgt somit für die referentielle Integrität innerhalb des Dokuments. Um dies am Beispiel zu verdeutlichen, erweitern wir in Abbildung 6 das Beispieldokument um Bankkonten, deren Besitzer durch eine ID/IDREF-Beziehung gekennzeichnet werden. Da in unserem Bei-

spiel auch Gemeinschaftskonten mit mehreren Besitzern verwaltet werden sollen, modellieren wir das referenzierende Attribut *Besitzer* mit dem Typ *IDREFS,* der Verweise auf mehrere ID-Attributwerte ermöglicht.

```
<?xml version="1.0" encoding="UTF-8">
<!DOCTYPE Bank [
    <!ELEMENT Bank (Kunden,Konten)>
    <!ELEMENT Kunden (Kunde*)>
    ...
    <!ELEMENT Konten (Konto*)>
    <!ELEMENT Konto (Dispo)>
    <!ATTLIST Konto id       ID     #REQUIRED
                    Besitzer IDREFS #REQUIRED>
    <!ELEMENT Dispo (#PCDATA)
]>
<Bank>
    <Kunden>
        <Kunde id="kd1606">
        ...
        </Kunde>
        <Kunde id="kd1407">
        ...
        </Kunde>
    </Kunden>
    <Konten>
        <Konto id="kto4711" Besitzer="kd1606 kd1407">
            <Dispo>4500.00</Dispo>
        </Konto>
    </Konten>
</Bank>
```

Abbildung 6: ID/IDREFS-Beziehung in XML-Dokumenten

Die referenzierten ID-Werte *kd1606* und *kd1407* werden innerhalb des Attributwerts durch Leerzeichen getrennt. Ein XML-Prozessor muss zur Validierung des Dokuments diese Werte extrahieren und für jeden Einzelwert prüfen, ob ein ID-Attribut mit diesem Wert vorhanden ist.

Mit der ID/IDREF(S)-Beziehung entsteht nun aus der Baumstruktur des ursprünglichen XML-Beispieldokuments durch das referenzierende *Besitzer*-Attribut auf logischer Ebene ein Graph. Dieser Graph wird ausschnittsweise in Abbildung 7 gezeigt, wobei Elemente mit Dreiecken und Attribute mit Kreisen dargestellt werden. Wie diese Graphstruktur in einem nativen XML-Datenbanksystem behandelt werden kann, beschreibt Kapitel 4.1.2 detaillierter.

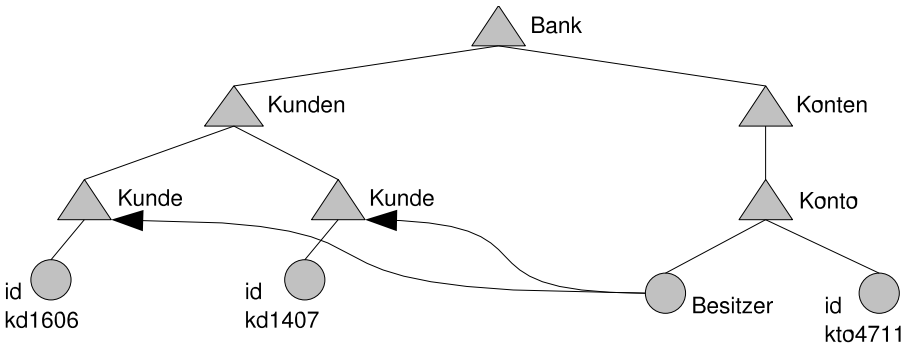


Abbildung 7: Graphstruktur mit der ID/IDREFS-Beziehung

Entities

Neben Elementen, Attributen und Textinhalten kann man in einer DTD auch so genannte *Entities* definieren, die als separate Dateneinheiten aufgefasst und als eine Art Platzhalter im XML-Dokument verwendet werden. Bei der Verarbeitung eines Dokuments ist einer der ersten Schritte eines XML-Prozessors die Auflösung von Entities mit Hilfe deren Definition, bevor die Prüfung der Wohlgeformtheit und die Validierung durchgeführt werden kann. Es werden *Parsed Entities*, die man als XML-Fragmente betrachten kann, von *Unparsed Entities*, die beliebige Datenformate enthalten können, wie z. B. Graphikdateien, unterschieden.

Unparsed Entities können aufgrund ihres beliebigen Formats nur außerhalb des Dokuments, also beispielsweise in einer separaten Datei liegen. Das Format wird mit einer *Notation* spezifiziert. Abbildung 8 zeigt nach [Sc03] am Beispiel eines Passbilds, wie so etwas funktioniert.

```
<!NOTATION gif SYSTEM "http://www.isi.edu/in-notes/iana/assignments/
                       media-types/image/gif">
<!ENTITY Passbild SYSTEM "Bilder/Pass.gif" NDATA gif>
<!ENTITY internal "<Hinweis>Nur für den internen Gebrauch!</Hinweis>">
```

Abbildung 8: Anwendung von Entities und Notation

Das Entity *Passbild* ist ein Bild im GIF-Format, das in der Datei *Bilder/Pass.gif* gespeichert ist. Das Bildformat wird mit der Notation *gif* spezifiziert, was vorausgesetzt, dass die Anwendung die für die Notation angegebene URI richtig interpretieren kann. Parsed Entities können Tags enthalten, die in diesem Fall auch vom XML-Prozessor verarbeitet werden. Ihr Wert muss daher wohlgeformt sein. Ein Beispiel für ein Parsed Entity ist *internal* in Abbildung 8.

Entities werden innerhalb eines XML-Dokuments mit dem Zeichen & referenziert, gefolgt vom Namen des Entity und einem Semikolon. Abbildung 9 zeigt in der ersten Zeile die Anwendung einer Entity-Referenz und in der darauffolgenden Zeile das Ergebnis nach der Ersetzung durch einen XML-Prozessor [Sc03].

```
<Titel>&internal;Strategie</Titel>

<Titel><Hinweis>Nur für den internen Gebrauch!</Hinweis>Strategie</Titel>
```

Abbildung 9: Anwendung der Entity-Referenz

2.2.2 XML Schema

XML erlaubt mit der Document Type Definition die schematische Beschreibung von Dokumentstrukturen. Dieses Konzept ist jedoch eher auf XML-Dokumente mit dokumentenorientiertem Charakter zugeschnitten, da es von SGML [SGML86] übernommen wurde. Für XML-Dokumente als Träger von Anwendungsdaten eignet sich die DTD aufgrund fehlender Datentypen weniger. Zudem ist für die maschinelle Verarbeitung der DTDs weitere Funktionalität zur Verfügung zu stellen, da eine DTD selbst kein wohlgeformtes XML-Dokument darstellt und somit neue Operationen auf dieser Struktur erfordert. Um dieses Dilemma zu beheben, arbeitet das W3C an der Standardisierung von *XML Schema*.

XML-Schema-Dokumente sind selbst gültige XML-Dokumente und unterstützen ein umfangreiches Typkonzept, gegenüber DTDs erweiterte Referenzbeziehungen und den modularen Entwurf von Strukturbeschreibungen. Die XML-Schema-Empfehlung des W3C ist so kom-

plex, dass sie zur besseren Übersicht in drei Teile gegliedert ist. Teil 0 [XSD04a], der „Primer“, enthält lediglich eine Übersicht über die beiden anderen Teile, Teil 1 [XSD04b] spezifiziert die Möglichkeiten der Schemabeschreibung und Teil 2 [XSD04c] die Basisdatentypen. Es ist nicht möglich, an dieser Stelle ausführlich die Empfehlungen des W3C zu erläutern. Daher werden einige Kernkonzepte vorgestellt, die detailliert bspw. in [Wal02] nachgelesen werden können.

Einfache Datentypen

XML Schema stellt eine große Anzahl bereits vordefinierter Basistypen bereit (*decimal*, *string*, *date* etc.). Ein *einfacher Typ* ist ein Datentyp, für den weder Attribute noch Kindelemente definiert sind. Alle vordefinierten Typen stellen somit einfache Typen dar.

Ein selbstdefinierter einfacher Typ basiert auf einem vordefinierten Typ, für den ein *Wertebereich*, ein *Repräsentationsraum* und verschiedene *Aspekte* definiert werden können. Der Wertebereich enthält alle gültigen Werte, die ein Datentyp annehmen kann. Für die Werte des Wertebereichs können zusätzlich verschiedene *Literale* eines Repräsentationsraums festgelegt werden. So ist es bspw. sinnvoll, für Datumsangaben verschiedene Repräsentationen zu erlauben, die alle denselben Wert darstellen. Über Aspekte kann der Datentyp weiter eingeschränkt werden. Hierbei legt man zum Beispiel Unter- und Obergrenzen für Wertebereiche (*minInclusive*, *minExclusive*, *maxInclusive*, *maxExclusive*), Längenbeschränkungen für Zeichenfolgen (*minLength*, *maxLength*) oder die Anzahl von Dezimalstellen und Nachkommastellen eines Zahlentyps (*totalDigits*, *fractionDigits*) fest. Ein einfacher Datentyp wird also immer durch Angabe eines Basisdatentyps und evtl. einer Beschränkung (*restriction*) definiert. XML Schema stellt weiterhin den Mechanismus der Erweiterung (*extension*) zur Verfügung, der dann jedoch immer einen komplexen Typ produziert (nächster Abschnitt).

Zur besseren Anschauung der Verwendung einfacher Typen definieren wir in Abbildung 10 für die zulässigen Werte der Dispokredithöhe der Konten unseres Beispieldokuments den Datentyp *Dispowert*. Dieser beruht auf dem vordefinierten Basistyp *decimal*, besitzt zwei Nachkommastellen und darf Werte zwischen *0,00* und *10.000,00* (jeweils inklusive) annehmen.

```
<xsd:simpleType name="Dispowert">
    <xsd:restriction base="xsd:decimal">
        <xsd:fractionDigits value="2"/>
        <xsd:minInclusive value="0.00"/>
        <xsd:maxInclusive value="10000.00"/>
    </xsd:restriction>
</xsd:simpleType>
```

Abbildung 10: Definition einfacher Datentypen mit XML Schema

Ein weiteres Beispiel für die Anwendung eines Aspekts ist die Definition einer Mustervorlage mit *Pattern* für Postleitzahlen in Abbildung 11. Eine gültige Postleitzahl wird für unser Beispiel als Zeichenfolge realisiert und besteht aus dem führenden *D*, einem Bindestrich und fünf Ziffern, jeweils zwischen 0 und 9.

```
<xsd:simpleType name="PLZWert">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="D-[0-9][0-9][0-9][0-9][0-9]"/>
    </xsd:restriction>
</xsd:simpleType>
```

Abbildung 11: Anwendung des Pattern-Aspekts in XML Schema

Komplexe Datentypen

Komplexe Typen werden für die Spezifikation von Attributen, Kindelementen und Textinhalten von Elementen benötigt. XML Schema unterscheidet in den Empfehlungen zur Dokumentenstruktur [XSD04b] bei komplexen Datentypen zwischen *einfachem Inhalt* für Elemente ohne Kinder, aber mit Attributen und Textinhalt, und *komplexem Inhalt* für Elemente mit beliebigem Inhaltsmodell. Der Inhalt eines komplexen Typs kann durch den Mechanismus der Erweiterung oder Einschränkung definiert werden, die beide eine Spezialisierung beschreiben. Die Erweiterung fügt einer Definition neue Elemente oder Attribute hinzu, die Einschränkung beschränkt die Wertebereiche der neuen oder vom Basistyp übernommenen Komponenten.

Für den einfachen Inhalt des Elements *<Dispo seit="2002-07-01">4500</Dispo>* kann bspw. eine Erweiterung durchgeführt werden (*xsd:extension*). Es entspricht damit dem komplexen Datentyp *DispoTyp* mit einfachem Inhalt in Abbildung 12. Die Höhe des Dispositionskredits basiert auf positiven Integer-Werten (*xsd:positiveInteger*) und wurde um die Definition eines Attributs zur Angabe eines Datums (*xsd:date*) erweitert.

```
<xsd:complexType name="DispoTyp">
    <xsd:simpleContent>
        <xsd:extension base="xsd:positiveInteger">
            <xsd:attribute name="seit" type="xsd:date"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
```

Abbildung 12: Komplexer Datentyp mit einfachem Inhalt in XML Schema

Komplexer Inhalt eines komplexen Typs ermöglicht die Definition beliebiger Inhaltsmodelle. Dazu stehen drei Möglichkeiten zur Verfügung. Bei der Sequenz (*sequence*) müssen die Inhaltsteile in der angegebenen Reihenfolge auftreten. Die Auswahl (*choice*) erlaubt nur einen der angegebenen Teile, wogegen bei der Konjunktion (*all*) alle angegebenen Inhaltsteile höchstens einmal benutzt werden dürfen.

Eine Anwendung des komplexen Inhalts eines komplexen Typs wird in Abbildung 13 für das *Name*-Element unseres Beispieldokuments gezeigt. Jeder komplexe Typ ist auch wieder eine Erweiterung oder Einschränkung eines existierenden Typs, im einfachsten Fall schränkt man *xsd:anyType,* den Basistyp jedes vordefinierten Typs, ein. Das Element *Name* enthält eine Sequenz von zwei Elementen, *Vorname* und *Nachname*, die jeweils vom Typ *xsd:string* sind, wobei maximal drei *Vorname*-Elemente (*maxOccurs="3"*) verwendet werden dürfen.

```
<xsd:complexType name="NameTyp">
    <xsd:complexContent>
        <xsd:restriction base="xsd:anyType">
            <xsd:sequence>
                <xsd:element name="Vorname" type="xsd:string" maxOccurs="3"/>
                <xsd:element name="Nachname" type="xsd:string"/>
            </xsd:sequence>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

Abbildung 13: Komplexer Datentyp mit komplexem Inhalt in XML Schema

Referenzbeziehungen

Die ID/IDREF(S)-Beziehung der Document Type Definition besitzt eine Reihe von Einschränkungen. Sie kann nur für Attributwerte und nicht für Elemente spezifiziert werden, jeder ID-Wert muss stets im gesamten Dokument unabhängig vom besitzenden Element eindeutig sein und die Referenz selbst basiert nicht auf der Wertegleichheit der ID- und IDREF-Attribute, sondern auf der syntaktischen Gleichheit ihrer Zeichenfolgen. XML Schema unterstützt diese Typen zur Kompatibilität zwar auch, löst jedoch diese Beschränkungen mit der Definition von eindeutigen Werten mittels *unique* oder *key* und der Referenz dieser Werte mittels *keyref*. Die Eindeutigkeit von Werten wird dabei mit der typbezogenen Gleichheit überprüft: Zwei Werte können nur gleich sein, wenn sie vom gleichen Typ sind. Beispielsweise sind 45*00* und 45*00,00* gleich, wenn sie vom Typ *decimal* sind, aber ungleich, wenn sie vom Typ *string* sind.

Eindeutige Werte können für einzelne Elemente oder Attribute festgelegt werden, aber auch für beliebige Kombinationen davon. Die Teile des Schemas, für die die Eindeutigkeit gilt, werden mit den Elementen *selector* und *field* spezifiziert. Im *selector*-Element wird dazu ein relativer Pfadausdruck angegeben, der eine Knotenmenge unterhalb des deklarierenden Elements adressiert, das *field*-Element gibt relativ dazu genau einen Knoten an. Bei der Definition der Eindeutigkeit wird zwischen *unique* und *key* unterschieden: *unique* fordert nur die Eindeutigkeit der Werte, wenn sie vorhanden sind, *key* fordert zusätzlich auch deren Existenz.

Für unser Beispieldokument wird in der folgenden Abbildung 14 die Eindeutigkeit des Attributs *id* aller *Kunde*-Elemente innerhalb des *Bank*-Elements (Elternelement der *xsd:key*-Spezifikation) mit dem *key*-Mechanismus von XML Schema realisiert. *Kunden* wird als Kindelement von *Bank* mit komplexem Inhaltsmodell definiert, das eine Sequenz aller *Kunde*-Elemente enthält. Nach der Definition des komplexen Inhalts folgt die Angabe des eindeutigen Attributs mit dem *selector*-Wert *Kunden/Kunde* und dem *field*-Wert *@id*.

```
<xsd:element name="Bank">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="Kunden">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="Kunde">
                            <xsd:complexType>
                                <xsd:attribute name="id" type="xsd:string"/>
                                <xsd:sequence>
                                    ...
                                </xsd:sequence>
                            </xsd:complexType>
                        </xsd:element>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:key name="Kundennummer">
        <xsd:selector xpath="Kunden/Kunde"/>
        <xsd:field xpath="@id"/>
    </xsd:key>
</xsd:element>
```

Abbildung 14: Definition eindeutiger Werte in XML Schema

Die Referenz auf mit *unique* bzw. *key* definierte Werte erfolgt in ähnlicher Weise mit dem *keyref*-Element, wobei mit dem Attribut *refer* der Name des zu referenzierenden *unique*- bzw. *key*-

Elements angegeben wird. Abbildung 15 erweitert unser Beispieldokument um Konten, deren Inhaber aus dem Kundenstamm referenziert werden. Da für ein Konto analog zur Erweiterung in Abschnitt 2.2.1 mehrere Besitzer eingetragen werden sollen, muss der Besitzer eines Kontos in XML Schema als mehrfach auftretendes Element realisiert werden. Hier wird ein Unterschied zur DTD deutlich: Das IDREFS-Konzept, das die mit Leerzeichen separierte Aneinanderreihung von ID-Referenzen innerhalb eines Attributwerts erlaubt, kann in XML Schema nicht mit dem *keyref*-Konzept nachgebildet werden. Dafür bietet XML Schema die Möglichkeit, einen eindeutigen Attributwert als Textinhalt eines Elements zu referenzieren. Die Referenz selbst wird im Beispiel durch das *keyref*-Element eingeleitet, erhält den Namen *Kontobesitzer* und referenziert die in Abbildung 14 definierte *Kundennummer*. Die *selector*- und *field*-Elemente spezifizieren die *Besitzer*-Elemente innerhalb der *Konto*-Elemente und stellen somit die Referenz her.

```
<xsd:element name="Bank">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="Kunden">
                ...
            </xsd:element>
            <xsd:element name="Konten">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="Konto">
                            <xsd:complexType>
                                <xsd:sequence>
                                    <xsd:element
                                        name="Besitzer"
                                        type="xsd:string"
                                        maxOccurs="5"/>
                                </xsd:sequence>
                            </xsd:complexType>
                        </xsd:element>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:key name="Kundennummer">
        ...
    </xsd:key>
    <xsd:keyref name="Kontobesitzer" refer="Kundennummer">
        <xsd:selector xpath="Konten/Konto/Besitzer"/>
        <xsd:field xpath="."/>
    </xsd:keyref>
</xsd:element>
```

Abbildung 15: Referenzierung eindeutiger Werte mit XML Schema

Eine vollständige Übersicht auf das Beispieldokument mit den in diesem Kapitel vorgestellten Erweiterungen für die Document Type Definition und XML Schema zur Kontenverwaltung, sowie die jeweils dazu definierten Schemata können in Anhang A nachgelesen werden.

2.3 Schnittstellen und Anfragesprachen

Zum Zugriff auf XML-Dokumente haben sich für Anwendungsprogramme drei verschiedenartige Typen von Schnittstellen (*Application Programming Interfaces, API*) etabliert. Die *Simple API for XML* (SAX) führt eine ereignisbasierte Verarbeitung der Daten durch, indem das

Dokument einmal sequentiell gelesen wird. Das *Document Object Model* (DOM) baut für das gesamte Dokument im Hauptspeicher der Anwendung eine Baumrepräsentation auf, für die zahlreiche Navigations- und Modifikationsoperationen bereit gestellt werden. Die *XML Query Language* (XQuery) bietet die Möglichkeit, unter Verwendung eines umfangreichen Typsystems deklarative Suchanfragen zu formulieren. Da es noch keinen Standard zur deklarativen Modifikation eines Dokuments gibt, existieren diverse proprietäre Erweiterungsvorschläge.

Jede der Schnittstellen bietet im Anwendungsbereich, für den sie geschaffen wurde, entsprechende Vorteile, sodass sie je nach Problemstellung ausgewählt werden sollten und daher in einem nativen XML-Datenbanksystem durchaus parallel zum Einsatz kommen können.

2.3.1 Simple API for XML (SAX)

Die Entwicklung der Simple API for XML [Br02] begann im Dezember 1997 als Open-Source-Projekt [SAX] kurz nach Veröffentlichung des letzten Entwurfs (*Working Draft*) der XML Spezifikation und wurde hauptsächlich von David Megginson und David Brownell geleitet. SAX wurde nie von einem Gremium der Internetgemeinde standardisiert, hat sich jedoch innerhalb kürzester Zeit als De-facto-Standard etabliert und liegt zur Zeit in der Version 2.0.1 vor.

SAX führt eine ereignisbasierte Verarbeitung für XML-Dokument durch. Dazu stellt die API eine vordefinierte Schnittstelle, den *Content Handler*, bereit. Der Content Handler enthält für jede Komponente eines XML-Dokuments (Start und Ende des Dokuments, der Elemente, Attribute, Texte und Verarbeitungsanweisungen) eine so genannte *Callback-Operation*. Eine konkrete Anwendung implementiert nun eine Content-Handler-Klasse und realisiert in den Rümpfen der Callback-Methoden die Anwendungslogik. Der Content Handler wird zur Laufzeit an den SAX-Parser übergeben. Dieser liest sequentiell das zu verarbeitende XML-Dokument und signalisiert jede gefundene Komponente des Dokuments durch den Aufruf der entsprechenden Callback-Methode. Auf diese Weise wird das gesamte XML-Dokument der Anwendung als Folge von Ereignissen dargestellt. Abbildung 16 zeigt die SAX-Ereignisfolge für den Kundennamen aus unserem Beispieldokument.

```
start element: Name
start element: Vorname
characters: Michael
end element: Vorname
start element: Vorname
characters: Peter
end element: Vorname
start element: Nachname
characters: Haustein
end element: Nachname
end element: Name
```

Abbildung 16: Ereignisfolge bei der SAX-Verarbeitung

Analog zum Content Handler existieren noch weitere vordefinierte Handler-Klassen, mit denen Kommentare, CDATA-Abschnitte und DTDs ausgelesen werden können.

Der Vorteil der SAX-API ist die hohe Geschwindigkeit und der geringe Verbrauch an Hauptspeicherplatz, da das XML-Dokument nur einmal sequentiell durchlaufen wird und nicht die gesamte Dokumentenstruktur im Hauptspeicher materialisiert, sondern lediglich durch Methodenaufrufe an die Anwendung gemeldet wird. Allerdings ist diese strombasierte Verarbeitung auch gleichzeitig ein Nachteil, da ein Zugriff auf bereits verarbeitete Daten nur durch eigene, zusätzlich auf Anwendungsseite implementierte Datenstrukturen, möglich ist.

2.3.2 Document Object Model (DOM)

Das Document Object Model ist eine Sammlung von XML-Empfehlungen des W3C [DOM1, DOM2, DOM3] und liegt aktuell in der Version 3 vor. Versionen werden im DOM-Umfeld als *Level* bezeichnet.

Mit dem Document Object Model stellt eine Schnittstelle einer Anwendung ein XML-Dokument in einer Baumrepräsentation zur Verfügung. Dieser Dokumentenbaum wird dabei vollständig im Hauptspeicher der Anwendung (meist mit Hilfe der SAX-API zum Aufbau) materialisiert. Als Knotentypen des Baums werden dabei alle in Abschnitt 2.1 vorgestellten Komponenten eines XML-Dokuments modelliert.

Zu jedem Knotentyp gibt es eine Schnittstelle, die den typischen navigationsorientierten Zugriff zum Elternknoten, den Geschwistern, dem ersten und letzten Kindknoten, einer Liste aller Kinder und Attribute und dem Knotenwert ermöglicht. Zur Modifikation kann ein neuer Knotenwert gesetzt, ein existierender Knoten gelöscht oder ein neuer Knoten an beliebiger Position eingefügt werden. Zur Suche stehen wenige einfache Operationen zur Verfügung, die den direkten Zugriff auf ein Element über dessen ID-Attribut oder eine Liste von Elementen über deren Namen in einem Teilbaum ermöglichen. Eine Repräsentation unseres Beispieldokuments mit dem Document Object Model ist in Abbildung 17 dargestellt.

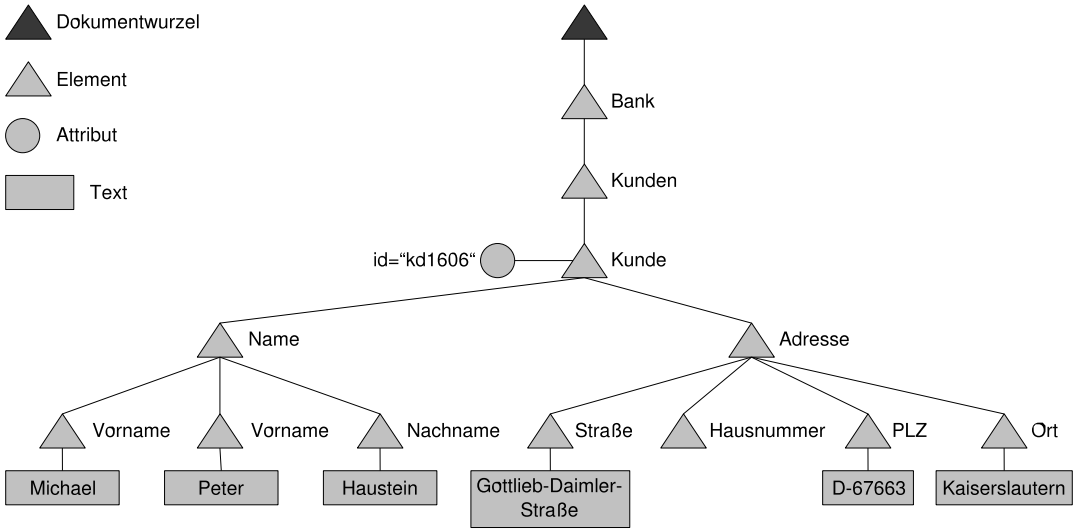


Abbildung 17: DOM-Repräsentation des Beispieldokuments

Im Gegensatz zu SAX bietet DOM jederzeit über die Navigation den Zugriff auf das gesamte XML-Dokument, da dessen Struktur und Inhalt vollständig im Arbeitsspeicher liegen. Diese Tatsache problematisiert jedoch die Verarbeitung größerer Dokumente. Ein XML-Dokument von 20 MB Umfang kann bereits bis zu 400 MB Hauptspeicher belegen [TFW02].

2.3.3 XML Query Language (XQuery)

Die XML Query Language ist ein sehr umfangreiches Standardisierungsprojekt des W3C für eine deklarative XML-Anfragesprache, die sich jedoch trotz langer Bemühungen immer noch im Entwurfsstadium (*W3C Working Draft*) befindet. Momentan umfasst der Entwurf bereits 17 Spezifikationen [XQL]. Die wichtigsten sind neben den Grundlagen der Sprache selbst [XQL05a] die Pfadadressierungssprache XPath 2.0 [XQL05f], die Spezifikation der formalen

Semantik [XQL05b], des Datenmodells [XQL05c] und der definierten Funktionen und Operatoren [XQL05d]. Es existiert sogar ein Vorschlag zur standardisierten Beschreibung einer XQuery-Anfrage in XML selbst [XQL05e], sodass auch XQuery-Anfragen normiert in einem XML-Dokument zwischen Anwendungen ausgetauscht oder nativ in einem XML-Datenbanksystem neben ihrer Auswertung gespeichert werden können. Aufgrund dieses Umfangs können wir hier nur die elementaren Grundlagen von XQuery wiedergeben und verweisen zum detaillierteren Studium auf die Spezifikationen [XQL] und die entsprechende Fachliteratur [LS04].

Das Typsystem von XQuery basiert auf XML Schema (siehe Abschnitt 2.2.2). Ein grundlegender Datentyp ist die Sequenz, die durch eine umklammerte Liste von Werten oder Knoten dargestellt wird, also z. B. (1, 2, 3). Besonders ist dabei, dass eine Sequenz nicht geschachtelt wird, sondern eine Kombination von Sequenzen wieder eine flache Sequenz ergibt (1, (2, 3)) = (1, 2, 3), und dass eine Sequenz mit einem Element als identisch zu diesem Element angesehen wird, also (1) = 1.

Ausdrücke

Eine XQuery-Anfrage besteht aus einem optionalen Prolog, in dem Variablen, Module und Funktionen definiert werden können, und einer Folge von *Ausdrücken*. Solch ein Ausdruck kann ein Literal (z. B. 1 oder *„eins“*), ein arithmetischer Ausdruck, ein Sequenzausdruck, ein Vergleichsausdruck, ein logischer Ausdruck oder ein Pfadausdruck (eine Aneinanderreihung so genannter *Pfadachsen*) sein. Da die in XQuery möglichen Pfadachsen für diese Arbeit von besonderer Bedeutung sind, werden sie im nächsten Abschnitt genauer beschrieben.

Ein weiterer spezieller Ausdruck in XQuery ist der *FLWOR*-Ausdruck (sprich „*Flower*“-Ausdruck). Dieser ermöglicht die Variablenbindung, Iteration über Sequenzen und eine Ergebnisrestrukturierung. Das Akronym FLWOR kürzt die in einem Ausdruck enthaltenen Klauseln *for*, *let*, *where*, *order by* und *return* ab:

- Mit dem Iterator *for* erfolgt eine Bindung des Variablennamens an nacheinander alle Elemente einer Sequenz. Werden mehrere Variablen gebunden, so wird für das Ergebnis über das entsprechende Kartesische Produkt iteriert.
- Mit *let* wird eine Variable an eine vollständige Sequenz gebunden. Aus der Kombination von *for* und *let* ergeben sich Tupel von Variablenbindungen.
- Der optionale Filter *where* wird mit einem Ausdruck kombiniert. Wird dessen logischer Wert zu „wahr“ ausgewertet wird, dann verfolgt ein XQuery-Prozessor die aktuelle betrachtete Variablenbindung weiter.
- Eine Sortierung kann mit *order by* erzwungen werden. Dabei sorgt die Angabe von mit *for* oder *let* gebundenen Variablen für die Auswertung der entsprechenden Klauseln in der definierten Reihenfolge. Die Adressierung eines Werts durch Anhängen eines Pfadausdrucks an die Variable sorgt für eine Sortierung der Ergebniswerte (wie bspw. in SQL).
- Die Klausel *return* wird zusammen mit einem Ausdruck angegeben, der für jedes Tupel ausgewertet wird, das den *where*-Filter erfolgreich passiert hat, und produziert somit das Ergebnis eines FLWOR-Ausdrucks. Damit bei der Verarbeitung zwischen Literalen und auszuwertenden Teilausdrücken unterschieden werden kann, müssen die Teilausdrücke in geschweifte Klammern eingeschlossen werden.

Zum elementaren Zugriff auf ein XML-Dokument wird die Funktion *doc()* benutzt, mit deren Hilfe wir in Abbildung 18 den Aufbau eines einfachen FLWOR-Ausdrucks und den Unterschied zwischen der *for*- und *let*-Klausel verdeutlichen.

```
for $i in doc("bank.xml")//Vorname/text()
return <Ergebnis>{$i}</Ergebnis>

liefert: <Ergebnis>Michael</Ergebnis><Ergebnis>Peter</Ergebnis>


let $i := doc("bank.xml")//Vorname/text()
return <Ergebnis>{$i}</Ergebnis>

liefert: <Ergebnis>MichaelPeter</Ergebnis>
```

Abbildung 18: Unterschied zwischen for- und let-Klausel in XQuery

Angenommen, das Beispieldokument aus Abbildung 1 liegt in der Datei *bank.xml* vor, so liefert der Pfadausdruck *//Vorname/text()* eine Sequenz der Werte der beiden *Vorname*-Elemente. Wird diese Sequenz mit *for* an die Variable *i* gebunden, so erfolgt eine Bindung für jedes Element der Sequenz und die Werte *Michael* und *Peter* werden separat in ein *Ergebnis*-Element geschachtelt. Die *let*-Klausel hat eine Bindung der gesamten Sequenz zur Folge, was eine Konkatenation der Sequenzwerte *MichaelPeter* in das *Ergebnis*-Element bewirkt.

Pfadachsen

XQuery definiert im zentralen Entwurf [XQL05a] für einen Pfadausdruck zwölf so genannte *Pfadachsen* (ohne die hier nicht relevante *Namespace*-Achse), mit deren Hilfe eine Sequenz von Knoten für einen aktuell betrachteten Knoten adressiert werden kann. Diesen aktuell betrachteten Knoten nennt man *Kontextknoten*. Ein Pfadausdruck besteht aus einer Folge von *Pfadschritten*, die jeweils aus einer Achse und einem durch zwei Doppelpunkte getrennten Namen bestehen. Weil die Pfadachsen eine wichtige Rolle in Kapitel 5 spielen, wird deren Semantik nun im Folgenden für einen Kontextknoten *k* genau beschrieben.

- *self::tagname*: Hat k den Elementnamen *tagname*, so ist er in der *Self*-Achse enthalten. Die Abkürzung der Self-Achse ist „.“.
- *attribute::tagname*: Besitzt der Kontextknoten k ein Attribut mit dem Namen *tagname*, so qualifiziert sich dieses Attribut für die *Attribute*-Achse. Abkürzung für die *Attribute*-Achse ist „*@tagname*“.
- *parent::tagname*: Trägt der Elternknoten von k den Namen *tagname*, so wird der Elternknoten in die Ergebnissequenz aufgenommen.
- *ancestor::tagname*: Die Ergebnissequenz enthält alle *Vorfahren* des Kontextknotens mit dem Namen *tagname*. Die Vorfahren lassen sich durch rekursives Ermitteln aller Elternknoten bis zur Dokumentwurzel bestimmen.
- *ancestor-or-self::tagname*: Die Ergebnissequenz besteht aus allen Knoten der *Ancestor*-Achse und zusätzlich dem Kontextknoten selbst, wenn er den Namen *tagname* trägt.
- *preceding::tagname*: Alle Knoten, die den Namen *tagname* tragen, und sich in der sequentiellen Dokumentenordnung vor dem Kontextknoten befinden, allerdings keine Vorfahren von k sind, heißen *Vorgänger* und qualifizieren sich für die Ergebnissequenz.
- *preceding-sibling::tagname*: Die Ergebnissequenz enthält aus der *Preceding*-Achse alle Knoten, die *Geschwister* des Kontextknotens sind, d. h. denselben Elternknoten besitzen.
- *descendant::tagname*: Alle *Nachfahren* des Kontextknotens, die den Namen *tagname* tragen, werden in die Ergebnissequenz aufgenommen. Ein Nachfahre von k ist ein Knoten, zu dem k ein Vorfahre ist. Abkürzung für die *Descendant*-Achse ist „*//tagname*“.

- *descendant-or-self::tagname*: Die Ergebnissequenz enthält alle Knoten der *Descendant*-Achse und zusätzlich den Kontextknoten selbst, wenn er den Namen *tagname* trägt.
- *child::tagname*: Die Ergebnissequenz wird aus allen *Kindknoten* des Kontextknotens konstruiert, die den Namen *tagname* besitzen. Ein Kindknoten von k ist ein Knoten, dessen Elternknoten k selbst ist. Die Abkürzung für die *Child*-Achse ist „*/tagname*“.
- *following::tagname*: Alle Knoten, die den Namen *tagname* tragen, und sich in der sequentiellen Dokumentenordnung hinter dem *Kontextknoten* befinden, allerdings keine Vorfahren von k sind, heißen *Nachfolger* und qualifizieren sich für die Ergebnissequenz.
- *following-sibling::tagname*: Die Ergebnissequenz enthält aus der *Following*-Achse alle Knoten, die Geschwister des Kontextknotens sind.

Die einzelnen Achsen innerhalb eines Ausdrucks werden durch Schrägstriche voneinander getrennt. Der Ausdruck *./descendant::Kunde/attribute::id* liefert bspw. für einen Kontextknoten eine Sequenz der Attribute mit dem Namen *id* aller Nachfahren, die den Namen *Kunde* tragen. Der Ausdruck kann auch mit der Schreibweise *.//Kunde/@id* abgekürzt werden. Eine grafische Übersicht über die wichtigsten XQuery-Pfadachsen zeigt Abbildung 19.

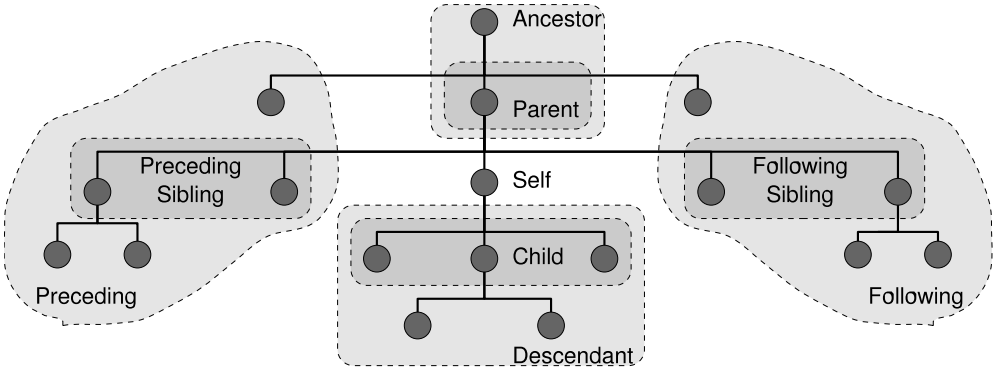


Abbildung 19: XQuery-Pfadachsen

2.3.4 XML:DB-Initiative

Da XQuery keinerlei Funktionalität zur Modifikation eines XML-Dokuments zur Verfügung stellt (ein Ergebnis ist stets eine unabhängige Sequenz), bemüht sich die XML:DB-Initiative [XMLDB] um eine Sprache zur standardisierten Dokumentenaktualisierung und um die Zusammenführung von SAX, DOM und XPath zu einer gemeinsamen Schnittstelle für XML-Datenbanksysteme.

Die XML:DB API [St01] organisiert die Dokumente eines XML-Datenbanksystems in *Kollektionen*. Eine Kollektion enthält wiederum Kollektionen oder *Ressourcen*. Die Ressourcen entsprechen XML-Dokumenten (*XMLResource*) oder beliebigen binären Dateien (*BinaryResource*). Auf XML-Dokumente selbst kann mit der *getContent()*-Methode auf deren Inhalt in Form einer Zeichenkette zugegriffen werden, oder mit Hilfe der *getContentAsSAX()*- bzw. der *getContentAsDOM()*-Methode auf den Inhalt mit der SAX- bzw. DOM-Schnittstelle. Dabei wird im ersten Fall der Methode ein Content Handler übergeben, im zweiten Fall liefert die Methode den Wurzelknoten des Dokuments zurück. Zur Aktualisierung eines XML-Dokuments kann der Methode *setContentAsSAX()* ein Content Handler oder der Methode *setCon-*

tentAsDOM() der Wurzelknoten eines XML-Fragments übergeben werden, mit deren Hilfe der Dokumenteninhalt in das Datenbanksystem übernommen wird.

Zur mengenorientierten Anfrage stehen die *XPathQueryServices* bzw. *XUpdateQueryServices* zur Verfügung, über die ein XPath- bzw. ein *XUpdate*-Ausdruck verschickt werden kann. XUpdate [LM00] definiert eine XML-basierte Sprache zur Modifikation von XML-Dokumenten mit deklarativen Konstrukten. Für die Modifikation stehen Anweisungen zum Einfügen, Anhängen, Aktualisieren, Umbenennen und Entfernen zur Verfügung. Neu einzufügende oder zu aktualisierende Knoten werden mit Erzeugungskonstrukten für Elemente, Attribute, Texte, Verarbeitungsanweisungen und Kommentare gebildet.

Abbildung 20 zeigt das Einfügen eines neuen Kunden in unser Beispieldokument mit einem ID-Attributwert von *kd1407* in XUpdate. Zunächst wird mit dem *xupdate:append*-Element das letzte Kind des Knotens mit dem Pfad */Bank/Kunden* adressiert. Innerhalb des *xupdate:append*-Elements wird mit *xupdate:element* ein neues Element mit dem Namen *Kunde* und einem Attribut mit Namen *id* (*xupdate:attribute*) erzeugt. Liefert die *select*-Anweisung des *xupdate:append*-Elements mehrere Knoten, so wird an jeden dieser Knoten der spezifizierte Inhalt angehängt.

```
<xupdate:append select="/Bank/Kunden" child="last()">
    <xupdate:element name="Kunde">
        <xupdate:attribute name="id">kd1407</xupdate:attribute>
    </xupdate:element>
</xupdate:append>
```

Abbildung 20: Dokumentmodifikation mit XUpdate

Parallel zur XML:DB API und XUpdate läuft die Standardisierung der *Simple XML Data Manipulation Language* (*SiXDML*) [Ob02a] und einer entsprechenden API [Ob02b], mit der die Verwaltung von Kollektionen, Dokumenten und Indexstrukturen durchgeführt werden kann. Da die letzte Veröffentlichung jedoch 2002 stattfand, ist die Zukunft des Projekts in Frage zu stellen. Es wäre jedoch wünschenswert, dass ein Ansatz wie XUpdate zur deklarativen Modifikation von XML-Dokumenten in nächster Zukunft in die XQuery-Empfehlung des W3C aufgenommen wird.

2.3.5 Erweiterungen für XQuery

Ein weiterer Ansatz zur mengenorientierten Modifikation von XML-Dokumenten wird von Patrick Lehti in seiner Diplomarbeit [Le01] mit Erweiterungen für XQuery vorgestellt. Eine Implementierung dieser Lösung wurde im Rahmen des nativen XML-Datenbanksystems eXist (siehe Abschnitt 3.1.1) vorgenommen.

Die Erweiterungen für XQuery ergänzen die Ausdrücke *insert*, *delete*, *replace* und *rename,* um neue Knoten oder XML-Fragmente in ein bestehendes Dokument einfügen oder löschen und vorhandene Knoten ersetzen oder umbenennen zu können. Die betroffenen Dokumententeile können dabei mit der vollen XQuery-Funktionalität spezifiziert werden.

Eine Änderungsoperation wird mit dem Schlüsselwort *update* eingeleitet, danach folgen ein oder mehrere Modifikationsausdrücke, die nacheinander abgearbeitet werden. Ein erstes einfaches Beispiel zum Einfügen neuer Knoten ist nach [Le01] in Abbildung 21 gegeben. Hier wird ein neues *warning*-Element als Geschwisterknoten hinter alle *blood_pressure*-Knoten eingefügt, die einen Kindknoten namens *systolic* mit einem Wert über *180* besitzen.

```
UPDATE
INSERT <warning>High Blood Pressure!</warning>
FOLLOWING //blood_pressure[systolic>180]
```

Abbildung 21: Einfügen neuer Knoten nach [Le01]

Die Modifikationsausdrücke können auch mit den von XQuery bekannten FLWOR-Ausdrücken kombiniert werden. Dies wird als *FLW-Update-Ausdruck* bezeichnet (damals wurden XQuery-Ausdrücke noch mit FLWR abgekürzt) und ist in Abbildung 22 dargestellt. Zunächst werden alle *invoice*-Elemente an die Variable *i* gebunden und nacheinander alle Nachfolger-Elemente *product*, die ein *Name*-Element mit Wert *screwdriver* besitzen, an die Variable *s*. Vor jeden mit *s* adressierten Kontextknoten wird nun ein neues Produkt (der *Hammer*) eingefügt und danach der Kontextknoten gelöscht.

```
UPDATE
FOR $i IN //invoice
LET $s := $i//product[name="screwdriver"]
WHERE not (empty($s))
INSERT
    <product>
        <name>Hammer</name>
        <price>15.40</price>
    </product>
PRECEDING $s
DELETE $s
```

Abbildung 22: Einfügen und Löschen mit FLW-Update-Ausdrücken

2.3.6 SQL/XML

Zur standardisierten Verarbeitung von XML-Daten in relationalen Datenbanksystemen wird ein XML-Dokument oder eine Menge von XML-Fragmenten als einzelner Attributwert in einer Tabelle betrachtet. Dazu spezifiziert SQL/XML als Teil 14 der SQL-Spezifikation SQL:2003 den neuen Datentyp *xml.*

Für diesen Datentyp werden u. a. vier Operatoren und ein Prädikat definiert, mit deren Hilfe XML-Daten für die Nutzung in einem relationalen Datenbanksystem verarbeitet werden können. *XMLParse* liefert für eine übergebene Zeichenfolge mit XML-Daten die entsprechende Repräsentation der Daten als Typ *xml*. Den entgegengesetzten Weg beschreitet der Operator *XMLSerialize*, der einen gegebene XML-Datentyp in eine SQL-Zeichenfolge umwandelt. *XMLRoot* setzt für den Wurzelknoten eines übergebenen XML-Datums einen neuen Wert und *XMLConcat* erzeugt einen neuen XML-Wert durch das Aneinanderhängen zwei übergebener Werte. Das Prädikat *IS DOCUMENT* prüft, ob ein übergebener XML-Wert ein Dokument repräsentiert, d. h., ob der Wert ein einzelnes Wurzelelement beinhaltet.

Ein weiterer Operator *XMLTable* ermöglicht die Bindung von Ergebnissen einer XQuery-Anfrage an relationale Attributwerte zur Konstruktion einer temporären Tabelle aus XML-Daten. Die so in einer Anfrage erzeugte Tabelle kann über einen Verbund mit rein relational gespeicherten Daten kombiniert werden. *XMLTable* ist in der aktuellen Spezifikation *SQL:2003* noch nicht enthalten, wird aber bereits für den Entwurf der folgenden Spezifikation diskutiert. Eine Anwendung des *XMLParse*- und *XMLTable*-Operators ist an Beispielen in Abschnitt 3.1.4 zu sehen, in dem die prototypische Implementierung dieses Standards im Datenbanksystem *System RX* vorgestellt wird.

Um XML-Werte mit SQL-Ausdrücken zu erzeugen, spezifiziert SQL/XML die so genannten *publishing functions*. *XMLElement* erzeugt ein XML-Element, dessen Inhalt durch Texte, die Funktion *XMLAttribute* zur Definition von Attributen oder weiteren *XMLElement*-Aufrufen angegeben werden kann. Abbildung 23 zeigt die Anwendung von *XMLElement* zur Erzeugung des in Abschnitt 2.2.2 diskutierten Elements *<Dispo seit="2002-07-01">4500</Dispo>* aus der für dieses Beispiel fiktiven Tabelle *Konto (ktonr, dispodatum, dispowert,...)*.

```
SELECT ktonr, XMLElement(NAME "Dispo",
                         XMLAttribute(k.dispodatum AS "seit"),
                         k.dispowert)
FROM Konto k
```

Abbildung 23: Anwendung von XMLElement in SQL/XML

Weitere *publishing functions* sind *XMLForest*, die im Gegensatz zu *XMLElement* nicht ein einzelnes XML-Element, sondern eine Sequenz von Elementen erzeugt, und *XMLNamespace* zur Annotation von Elementen mit Namespaces.

2.4 Technologien

Es gibt über 100 weitere Technologien, die auf dem XML-Standard aufbauen[1]. Wir diskutieren im Folgenden die für diese Arbeit relevanten Spezifikationen, die meist in DBMS-Schnittstellen zur Kommunikation mit Anwendungsprogrammen zum Einsatz kommen.

2.4.1 XML Remote Procedure Call (XML-RPC)

XML-RPC ist ein sehr einfacher Standard zum plattformunabhängigen Aufruf von Methoden[2] auf entfernten Systemen [Wi03]. Die aufzurufende Methode sowie die dazu benötigten Parameter werden in XML formuliert; das Ergebnis wird ebenfalls mit einem XML-Dokument dargestellt. Der Transport der Methodenaufrufe sowie der Ergebnisse erfolgt über das HTTP-Protokoll.

Ein einfaches Beispiel für einen RPC-Aufruf ist in Abbildung 24 gegeben. Eingeschachtelt in das *methodCall*-Element wird zunächst der Name der aufzurufenden Methode angegeben. Danach folgt eine Liste der benötigten Parameter. Die übergebenen Parameterwerte können mit einer kleinen Auswahl von Datentypen (*int*, *boolean*, *string*, *double*, *datetime* und *base64*) typisiert werden. Wird kein Typ angegeben, so wird der Parameterwert als Zeichenfolge (*string*) interpretiert. Mehrere Parameter können zudem zu einer Struktur (*<struct>*) oder zu einem Feld (*<array>*) kombiniert werden.

```
<?xml version="1.0">
<methodCall>
    <methodName>examples.getStateName</methodName>
    <params>
        <param>
            <value><int>41</int></value>
        </param>
    </params>
</methodCall>
```

Abbildung 24: Methodenaufruf mit XML-RPC

1 Key XML Specifications and Standards, http://www.zapthink.com

2 Gemäß der UML-Nomenklatur bezeichnen wir in diesem Zusammenhang mit einer Methode die Implementierung einer Schnittstellenoperation.

Die Antwort eines Methodenaufrufs wird in einem ähnlich strukturierten XML-Dokument zurückgeschickt, das die Ergebniswerte oder eine Fehlermeldung (mit Fehlercode und -nachricht) in einem *methodResponse*-Element enthält. Abbildung 25 zeigt im Teil a) das Ergebnisdokument für einen erfolgreichen Methodenaufruf und im Teil b) die Antwort auf einen Fehler.

```
a)  <methodResponse>
        <params>
            <param><value><string>South Dakota</string></value></param>
        </params>
    </methodResponse>

b)  <methodResponse>
        <fault>
            <value>
                <struct>
                    <member>
                        <name>faultCode</name>
                        <value><int>4</int></value>
                    </member>
                    <member>
                        <name>faultString</name>
                        <value><string>Too many parameters.</string></value>
                    </member>
                </struct>
            </value>
        </fault>
    </methodResponse>
```

Abbildung 25: Ergebnisdokumente für XML-RPC-Aufrufe

2.4.2 SOAP

SOAP ist das XML-Nachrichtenformat, mit dem der Datenaustausch im *XML Web Service Framework*[3] des W3C abgewickelt wird. Die ursprüngliche Bezeichnung *Simple Object Access Protocol* der Abkürzung SOAP wird heute nicht mehr verwendet. Analog zu XML-RPC lassen sich mit SOAP auch einfache Methodenaufrufe durchführen. Die SOAP-W3C-Empfehlung ist in drei Teile aufgegliedert: den *Primer* [SOAP03a], das *Messaging Framework* [SOAP03b] und die *Adjuncts* [SOAP03c].

Eine SOAP-Nachricht besteht ähnlich einem gewöhnlichen Brief aus einem Umschlag (*envelope*), der aus einem Kopf (*header*) mit Steuerungsinformationen und einem Körper (*body*) mit dem eigentlichen Nachrichteninhalt besteht. Im Gegensatz zu XML-RPC ist die Übertragung einer SOAP-Nachricht nicht an ein bestimmtes Protokoll gebunden, sondern es existiert ein Regelwerk zur Bindung an verschiedene Protokolle (z. B. *http get* und *post*, *smtp*). Zudem muss die Nachricht nicht direkt von der Anwendung zum verarbeitenden System transportiert werden, sondern der Transport kann über mehrere Stationen erfolgen, wobei im Kopf der SOAP-Nachricht auch Anweisungen für die Zwischenstationen hinterlegt werden können.

Da die SOAP-Spezifikation sehr umfangreich ist und ausreichend Gebrauch von XML-Namespaces macht, hat das erste einfache Beispiel aus [SOAP03a] bereits eine beachtliche Größe (siehe Abbildung 26). Im Kopf der Nachricht ist festgelegt, dass der Empfänger und alle Zwischenstationen die Elemente *reservation* und *passenger* verarbeiten können müssen. Der eigentliche Inhalt dieser Elemente ist anwendungsspezifisch und wird von SOAP nicht festgelegt. Der Körper enthält Angaben über die Reiseroute (*itinerary*) mit Hin- und Rückweg (*departure* und *return*) des Passagiers und Präferenzen zur Unterbringung (*lodging*).

3 Hier nicht näher erläutert, Details siehe unter http://www.w3c.org/2002/ws

SOAP-Nachrichten werden wie oben erwähnt im Web-Service-Umfeld eingesetzt. Mit der *Web Service Description Language* (WSDL) werden angebotene Dienste beschrieben und die dazu erforderlichen SOAP-Nachrichten spezifiziert. Da hierbei auch XML Schema eine Rolle spielt, kann für die Definition der SOAP-Nachrichtendaten das XML-Schema-Typsystem eingesetzt werden.

```
<?xml version='1.0'?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">

    <env:Header>
        <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
                env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
                env:mustUnderstand="true">
            <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe87d</m:reference>
            <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
        </m:reservation>
        <n:passenger xmlns:n="http://mycompany.example.com/employees"
                env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
                env:mustUnderstand="true">
            <n:name>Ake Jogvan Oyvind</n:name>
        </n:passenger>
    </env:Header>

    <env:Body>
        <p:itinerary
         xmlns:p="http://travelcompany.example.org/reservation/travel">
            <p:departure>
                <p:departing>New York</p:departing>
                <p:arriving>Los Angeles</p:arriving>
                <p:departureDate>2001-12-14</p:departureDate>
                <p:departureTime>late afternoon</p:departureTime>
                <p:seatPreference>aisle</p:seatPreference>
            </p:departure>
            <p:return>
                <p:departing>Los Angeles</p:departing>
                <p:arriving>New York</p:arriving>
                <p:departureDate>2001-12-20</p:departureDate>
                <p:departureTime>mid-morning</p:departureTime>
                <p:seatPreference/>
            </p:return>
        </p:itinerary>
        <q:lodging
         xmlns:q="http://travelcompany.example.org/reservation/hotels">
            <q:preference>none</q:preference>
        </q:lodging>
    </env:Body>

</env:Envelope>
```

Abbildung 26: Beispiel-SOAP-Nachricht für eine Reisebuchung

2.4.3 Web Distributed Authoring and Versioning (WebDAV)

WebDAV [WDAV99] ist eine Erweiterung des HTTP-Protokolls um Aktionen zur Organisation von Ressourcen in Kollektionen. Die durchzuführenden Aktionen werden als XML-Dialekt mit dem HTTP-Protokoll, das um zusätzliche Befehle ergänzt wurde, übertragen. Im Unterschied zu HTTP, das nur rein lesenden Zugriff erlaubt, bietet WebDAV auch die Möglichkeit, Kollektionen anzulegen, neue Ressourcen an einen Server zu übertragen, zu modifizieren oder zu löschen. Um Zugriffskonflikte zu vermeiden, kann eine Anwendung Lese- und Schreibsperren auf den vom Server verwalteten Ressourcen anfordern. Zusätzlich können Ressourcen mit Eigenschaftswerten versehen werden, über die Suchanfragen möglich sind.

Das folgende einfache Beispiel aus [WDAV99] in Abbildung 27 zeigt das Kopieren aller Ressourcen im Verzeichnis *container* des Rechners *www.foo.bar* in das Verzeichnis *http://www.foo.bar/othercontainer* mit einem Aufruf an eine WebDAV-Schnittstelle. Das Element *d:keepalive* sorgt dafür, dass alle kopierten Ressourcen auf dem Zielsystem direkt veröffentlicht werden.

```
COPY /container/ HTTP/1.1
Host: www.foo.bar
Destination: http://www.foo.bar/othercontainer/
Content-Type: text/xml; charset="utf-8"

<?xml version="1.0" encoding="utf-8" ?>
<d:propertybehavior xmlns:d="DAV:">
    <d:keepalive>*</d:keepalive>
</d:propertybehavior>
```

Abbildung 27: Kopieren von Ressourcen mit WebDAV

Inzwischen wurden mehrere Ergänzungen zum WebDAV-Protokoll veröffentlicht. [WDAV02] erweitert das Protokoll um umfangreiche Methoden zur Versionierung, [WDAV03] bietet einen neuen Datentyp zum Zugriff auf geordnete und ungeordnete Kollektionen und [WDAV04] ermöglicht die Definition von Zugriffsrechten auf einzelnen Ressourcen.

2.5 Zusammenfassung

Dieses Kapitel gibt einen Überblick über die Grundlagen der Extensible Markup Language und die für diese Arbeit relevanten Technologien im XML-Bereich. Zunächst wurden die elementaren Konstrukte für den Aufbau eines XML-Dokuments betrachtet und ein Beispieldokument eingeführt, das in den nächsten Kapiteln als durchgängiges Beispiel fortlaufend erweitert wird. Zur Definition vorgegebener Strukturen, gegen die XML-Dokumente validiert werden können, wurde die *Document Type Definition* (*DTD*) und *XML Schema* diskutiert. Dabei fand die Definition von Referenzbeziehungen besondere Beachtung. Als Schnittstellen für die Anwendungsprogrammierung haben wir die drei typischen Vertreter *Simple API for XML*, *Document Object Model API* und *XML Query Language* (mit einigen möglichen Erweiterungen) für den ereignisbasierten, navigationsorientierten und deklarativen Zugriff auf XML-Dokumente betrachtet. Eine Erläuterung XML-basierter Technologien zum Datentransport zwischen Anwendungen schließt die Grundlagendiskussion ab.

KAPITEL 3 Verwandte Arbeiten

Das gefährliche an Halbwahrheiten ist,
dass immer die falsche Hälfte geglaubt wird.
(Hans Krailsheimer)

Dieses Kapitel gibt eine Übersicht über veröffentlichte Arbeiten im Bereich der nativen XML-Datenverarbeitung. Bzgl. der in dieser Arbeit behandelten Themen beschränken wir uns dabei auf die grundlegenden Architekturkonzepte existierender XML-Datenbanksysteme, mögliche Nummerierungsschemata zur Identifikation einzelner Knoten in einem XML-Dokument und die publizierten Ansätze zur Transaktionsisolation.

3.1 Native XML-Datenbanksysteme

Es gibt bereits eine Reihe nativer XML-Datenbanksysteme. Da dieses Forschungsgebiet jedoch noch relativ jung ist und noch keine Anfragesprache standardisiert wurde, die auch Dokumentmodifikationen unterstützt (siehe Kapitel 2), sind diese Systeme hauptsächlich auf eine möglichst effiziente Auswertung von reinen Suchanfragen ausgelegt. Die feingranulare transaktionsgeschützte Veränderung von Dokumentinhalten auf der Ebene einzelner Knoten wird meist nicht unterstützt, und wenn, dann durch Abbildung auf ein gröberes Granulat. Dieses Granulat wird durch die eingesetzte Speicherungsstrategie bestimmt, nach denen die im Folgenden diskutierten XML-Datenbanksysteme klassifiziert werden können (Abbildung 28).

Elementbasierte Systeme speichern jedes Element eines Dokuments als separaten Datensatz und erlauben somit (zumindest konzeptionell) eine feingranulare Aktualisierung. Hierzu betrachten wir die Systeme *eXist* und *Sedna*. Teilbaumbasierte XML-Datenbanksysteme fassen mehrere Elemente zu einem Teilbaum zusammen und verwalten diesen in einem gemeinsamen Datensatz. Nach diesem Prinzip sind *System RX*, *Natix* und *OrientX* implementiert, wobei OrientX optional auch eine elementbasierte Speicherung unterstützt. Die dokumentbasierten Systeme schließlich verteilen ein XML-Dokument komprimiert als logische Einheit über mehrere Datenseiten. Als typische Vertreten stellen wir dazu *Xindice* und *Tamino* vor.

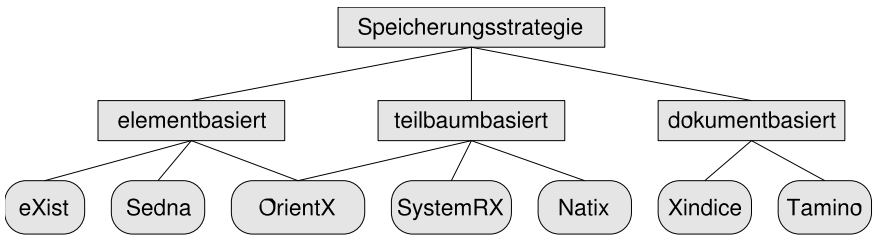


Abbildung 28: Speicherungsstrategien nativer XML-Datenbanksysteme

Zur Transaktionsisolation setzen die Systeme oft ein pessimistisches Sperrverfahren mit einfachen Lese- und Schreibsperren ein. Im Extremfall resultiert dies in Sperren auf Dokumentebene. Um dies genauer zu untersuchen, beschreiben wir einige typische XML-Datenbanksysteme als Vertreter der obigen Klassen bzgl. ihrer Funktionalität, der zugrunde liegenden Architektur, der verwendeten Speicherungsstrukturen und der eingesetzten Techniken zur Gewährleistung von Transaktionseigenschaften.

3.1.1 eXist

Das XML-Datenbanksystem eXist [Me02] wurde im Januar 2001 als Open-source-Projekt[4] ins Leben gerufen. Das Projekt ist vollständig in Java implementiert und wird zur Zeit auch weiterentwickelt. Federführend für die Publikationen ist die Technische Universität Darmstadt.

Funktionalität

eXist organisiert XML-Dokumente in hierarchisch angeordneten Kollektionen, die mit der Strukturierung von Dateien in Dateisystemen vergleichbar sind. Die Speicherung erfolgt schemaunabhängig, d. h., weder die Kollektionen noch die XML-Dokumente selbst sind an ein Schema gebunden.

Das Datenbanksystem kann als alleinstehender Server-Prozess gestartet werden, als Komponente direkt in eine Anwendung integriert werden oder innerhalb eines Applikationsservers laufen. Zum Zugriff auf die gespeicherten Dokumente stehen die Schnittstellen XML-RPC, XML:DB API, SOAP und WebDAV (siehe dazu Kapitel 2) zur Verfügung. Die Anfragen selbst werden in XPath 2.0 und XQuery 1.0 formuliert, wobei XQuery um die von Patrick Lehti vorgeschlagenen Konstrukte zur Datenmanipulation [Le01] erweitert wurde. Da eXist hauptsächlich zur Verwaltung dokumentenorientierter XML-Dokumente eingesetzt werden soll, wurde die XPath-Funktionalität mit den beiden Operatoren &= und |= und einigen Funktionen zur Volltextsuche erweitert, die im Folgenden anhand kleiner Beispielanfragen erläutert werden.

Die erste Anfrage in Abbildung 29a [CRZ03] liefert alle *section*-Elemente eines Dokuments, die die Suchbegriffe *XML* und *database* in der angegebenen Reihenfolge enthalten, wobei zwischen den Begriffen nicht mehr als 50 weitere Wörter stehen dürfen. Falls die Reihenfolge und der Abstand der Suchbegriffe nicht von Bedeutung ist, setzt man die Operatoren &= und |= ein. Die Anfrage in Abbildung 29b benutzt den Operator &= und liefert alle *scene*-Elemente, die ein *speech*-Element enthalten, das wiederum die Elemente *speaker* und *line* enthält. Die Reihenfolge oder der Wortabstand der durch Leerzeichen getrennten Suchbegriffe *witch* bzw. *fenny* und *snake* ist dabei nicht von Bedeutung. Sollen nicht alle Begriffe, sondern mindestens nur jeweils ein Begriff der Anfrage im Ergebnis vorhanden sein, so kann der Operator |= verwendet werden. Zur Suche nach Zeichenfolgenmustern gibt es die Funktion *match-all*. Die letzte Anfrage in Abbildung 29c zeigt deren Anwendung bei der Suche nach allen *speech*-Elementen, die ein *line*-Element enthalten, das im Wert *u. a.* die Wörter *live*, *lives* oder *life* aufweist.

```
a) //section[near(.,'XML database',50)]
b) //scene[speech[speaker &= 'witch' and line &= 'fenny snake']]
c) //speech[match-all(line,'li[fv]e[s]')]
```

Abbildung 29: Erweiterte XPath-Anfragen in eXist

4 http://www.exist-db.org

Architektur

Die Architektur von eXist ist schematisch nach [Me02] in Abbildung 30 dargestellt. XML-Dokumente werden gemäß der DOM-Spezifikation des W3C in einzelne Knoten zerlegt und gespeichert. Der Kern des Datenbanksystems ist von der Speicherungskomponente (nativ oder in einem angekoppelten relationalen Datenbanksystem) vollkommen isoliert. Alle Methoden zum Speichern, Ändern oder Löschen von Dokumenten oder Dokumentteilen implementieren die Operationen der *DBBroker*-Schnittstelle.

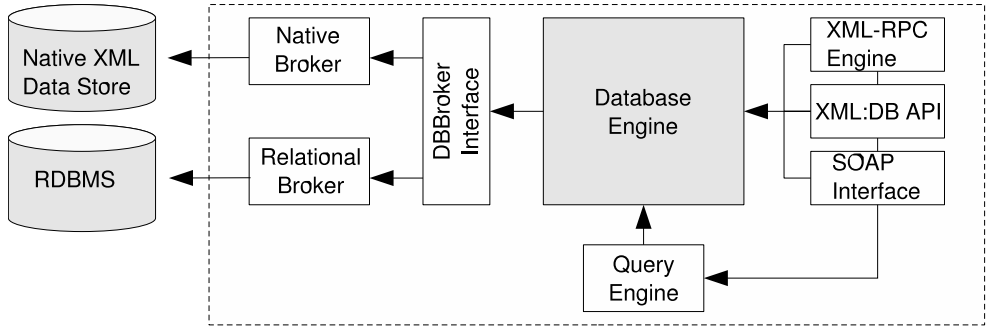


Abbildung 30: Architektur des Datenbanksystems eXist

Die Speicherung der XML-Dokumente in einem relationalen Datenbanksystem wurde vor allem im Anfangsstadium der Entwicklung verwendet, um die Architektur von eXist auf einem zuverlässigen Speichersystem zu evaluieren. Ab Version 0.6 wurde die im folgenden Abschnitt beschriebene native XML-Speicherungskomponente ergänzt.

Speicherungsstrukturen

Zur Speicherung eines XML-Dokuments stellt der *Native XML Data Store* Datenseiten fester Länge bereit, in denen die einzelnen Knoten in Dokumentenordnung (*left-most depth-first*) fortlaufend abgelegt werden. Dies ermöglicht die Rekonstruktion von XML-Fragmenten durch sequenzielles Auslesen der Datenseiten. Die Organisation des XML Data Store nach [Me02] ist in Abbildung 31 zu sehen.

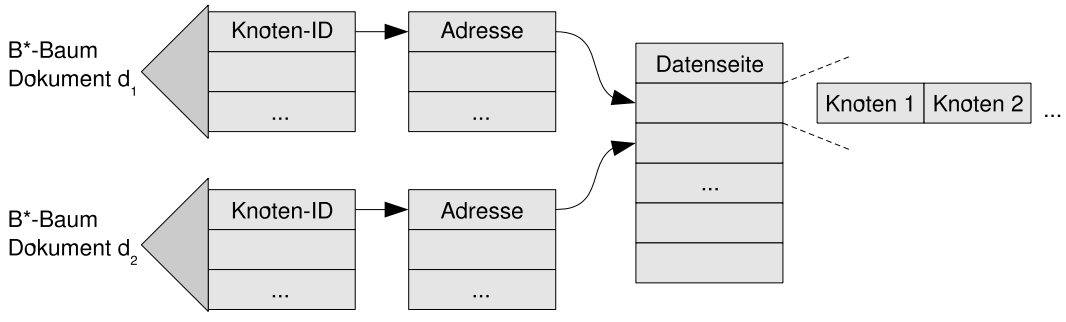


Abbildung 31: Organisation des XML Data Store in eXist

eXist verwendet zur Nummerierung der einzelnen Knoten eine Erweiterung des von Lee et al. [LYY96] vorgeschlagenen k-Wege-Baums (siehe dazu Abschnitt 3.2.1). Dabei wird auf die Vollständigkeit des Baums verzichtet und für jede Ebene des Dokuments eine maximale An-

zahl von Kindern definiert. Man spart damit eine große Anzahl von Platzhalterelementen ein und kann dokumentenorientierte XML-Dokumente, die gewöhnlich durch in der Nähe der Blätter breitere Bäume beschrieben werden, geeigneter nummerieren. Da durch den k-Wege-Baum die maximale Anzahl der Kinder in jeder Ebene des Dokuments bekannt ist, können die IDs der Geschwister-, Kind- und Vorfahrenknoten aus der ID des Kontextknotens berechnet werden, wodurch die Verwaltung zusätzlicher Strukturinformation entfällt. Da im k-Wege-Baum die Knoten innerhalb einer Ebene sequentiell nummeriert werden, die Knoten in den Datenseiten jedoch in Dokumentenordnung vorliegen, ist eine zusätzliche Indirektion über einen Adresszeiger zwischen Knoten-ID und Datenseite erforderlich.

Zur Realisierung der nativen Speicherung werden vier Dateien eingesetzt. Die Datei *dom.dbx* beinhaltet die Datenseiten mit den XML-Knoten. Für Elemente und Attribute wird ein Vokabular eingesetzt, das in *elements.dbx* gespeichert ist. Die Verwaltung der XML-Dokumente, die wie oben beschrieben zu Kollektionen zusammengefasst sind, organisiert *collections.dbx*. Die vierte Datei *words.dbx* fungiert als invertierter Index zur Unterstützung der Textsuche.

Pfadanfragen können mit dem verwendeten Nummerierungsschema effizient ausgewertet werden, da die Lage zweier Knoten zueinander aus deren IDs berechenbar ist. Für die Anfrage */play//speech* werden beispielsweise über den Volltextindex zwei Mengen mit jeweils den Knoten-IDs der *play*- bzw. der *speech*-Elemente gebildet. Für jedes *speech*-Element kann nun berechnet werden, ob dessen Knoten-ID Nachfolger einer ID eines *play*-Elements ist, und in die Ergebnismenge aufgenommen werden muss.

Transaktionsverwaltung

Der *Native XML Data Store* kontrolliert nur, dass einzelne Datensätze (also XML-Knoten) nicht durch parallel laufende Operationen zeitgleich überschrieben werden, was der Strategie des kurzfristigen *page fixing* [GR93] entspricht. eXist unterstützt keine Transaktionsverwaltung und garantiert bis auf die Konsistenzzusicherung, dass die gespeicherten Dokumente wohlgeformt sind, keine der ACID-Eigenschaften.

Bewertung

Das Datenbanksystem eXist eignet sich zur Verwaltung von XML-Dokumenten, die keinen allzu großen Änderungen unterliegen, da das verwendete Nummerierungsschema für die XML-Knoten nur geringe Modifikationen ohne Reorganisation übersteht. Dafür können jedoch strukturelle Beziehungen innerhalb der Daten berechnet werden, was die Wartung zusätzlicher Indexe für die Dokumentenstruktur überflüssig macht und eine effiziente Anfrageverarbeitung ermöglicht.

Die leichtgewichtige Architektur von eXist bietet zwar keine ACID-Transaktionseigenschaften, eignet sich jedoch sehr gut zur direkten Einbettung in Anwendungen mit hauptsächlich lesendem Datenzugriff. Für den italienischen Automobilhersteller FIAT werden beispielsweise technische Fahrzeugdokumentationen als XML-Dokumente einschließlich des eXist-Datenbanksystems und einer Anwendungssoftware auf einer lauffähigen CD an Fahrzeughändler ausgeliefert. Bei einer durchschnittlichen Dokumentationsdatei von 12 MB erreicht eXist auf einem Rechner der Pentium-I-Klasse mit 200 MHz für die Anwendung eine Antwortzeit von drei bis fünf Sekunden [Me02].

3.1.2 Sedna

Das Datenbanksystem Sedna [GFK04] wird am Institut für Systemprogrammierung der Russischen Akademie der Wissenschaften entwickelt. Der Systemkern wird in C++ implementiert, die Anfrageanalyse und -optimierung in Scheme, einem Lisp-Dialekt.

Funktionalität

Zur Anwendungsprogrammierung mit Sedna stehen Schnittstellen für Java, C und Scheme zur Verfügung. Das System ist für mengenorientierte Zugriffe ausgelegt, sodass die Treiber nur die Möglichkeit bieten, XQuery-Anfragen zu formulieren. Zur Modifikation gespeicherter XML-Dokumente wurde XQuery mit einer Änderungsfunktionalität erweitert, die sich an den Vorschlägen von Lehti (siehe Abschnitt 2.3.5) orientiert.

Eine weitere interessante Ergänzung bietet die Einbettung von SQL-Anweisungen in XQuery [SED05]. Dazu stehen die Funktionen *sql:connect* und *sql:execute* zur Verfügung. Abbildung 32a zeigt einen FLWOR-Ausdruck, der über eine ODBC-Verbindung alle Tupel der Tabelle *people* aus einer relationalen Datenbank auswählt, deren Attribut *first* den Wert *Peter* besitzt. Die Ergebnistupel werden mit jeweils einem XML-Element formatiert, das die Attribute eines relationalen Tupels als XML-Attribute besitzt (Abbildung 32b).

```
a)  let $connection := sql:connect("odbc:mysql://localhost/somedb",
                                   "user","pass")
    return sql:execute($connection,"SELECT * FROM people WHERE first='Peter'")

b) <tuple first="Peter" last="Jackson" city="Wellington"/>
```

Abbildung 32: Einbettung von SQL-Anweisungen in XQuery bei Sedna

Architektur

Eine Übersicht über die Architektur von Sedna ist in Abbildung 33 gegeben. Die zentrale Steuerung übernimmt der *Governor*, bei dem alle weiteren Komponenten registriert sind.

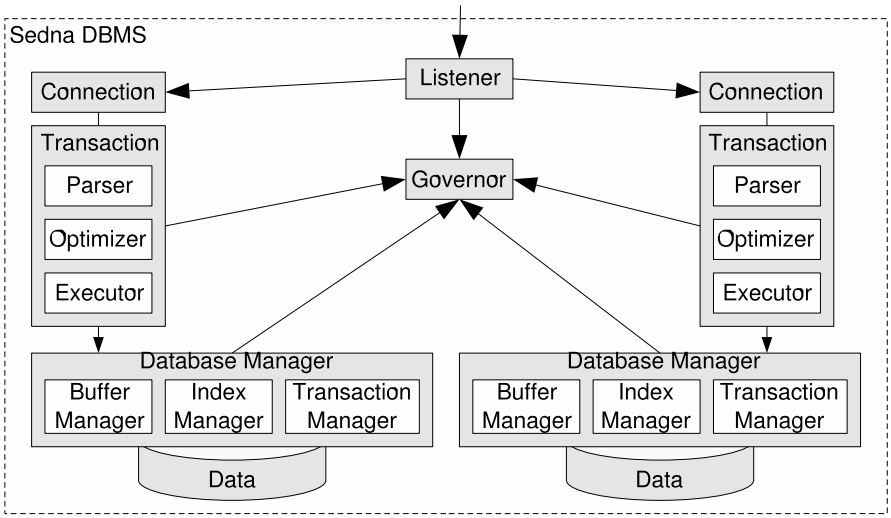


Abbildung 33: Architektur des Datenbanksystems Sedna

Jede Anwendung, die sich mit dem Datenbanksystem verbindet, nimmt zuerst Kontakt mit dem *Listener* auf, der für die Anwendung ein *Connection*-Objekt erzeugt. Der Zugriff auf die eigentlichen Daten des Systems erfolgt über den *Database Manager*, der einen *Buffer Manager*, *Index Manager* und *Transaction Manager* kapselt. Ein Database Manager wird für jede

verwaltete Datenbankinstanz anlegt. Startet eine Anwendung eine Transaktion, so wird dafür ein *Transaction*-Objekt erzeugt, das den *Parser*, *Optimizer* und *Executor* zur Anfrageanalyse, -optimierung und -ausführung enthält. Zur Anfrageverarbeitung wird ein Anfragegraph mit Planoperatoren der *XQuery Core Language* [XQL05b] erstellt, optimiert und ausgeführt.

Speicherungsstrukturen

Die Speicherung eines XML-Dokuments erfolgt in Sedna schemabasiert. Der dazu entwickelte Ansatz wird als *descriptive schema driven storage strategy* bezeichnet und ist in Abbildung 34 dargestellt. Für jedes Dokument wird ein so genannter *beschreibender Graph* angelegt, der nichts anderes als ein DataGuide [GW97] ist und für jeden im Dokument vorkommenden Pfadtyp einen Stellvertreterpfad enthält. Jeder Knoten des XML-Dokuments wird nun entsprechend seines Pfads im beschreibenden Graph in einer verketteten Liste gespeichert. Zur Entkopplung der Knotenadresse von der physischen Speicheradresse wird eine Indirektionstabelle eingeführt, in der logische Knotenadressen mit Hilfe eines *node handle* auf physische abgebildet werden. Die Beziehungen zu Eltern-, Geschwister- und Kindknoten werden mit Verweisen auf deren logische Adressen verwaltet. Um den Speicherplatzbedarf für einen Knoten nicht zu stark anwachsen zu lassen, wird nur eine einstellbare Anzahl von Kindknoten gespeichert. Sind alle reservierten Verweisplätze belegt, so muss zur Bestimmung aller Kinder der *next-sibling*-Verweis des Kindknotens auf dem letzten Verweisplatz iterativ weiterverfolgt werden, bis alle Kinder ermittelt sind.

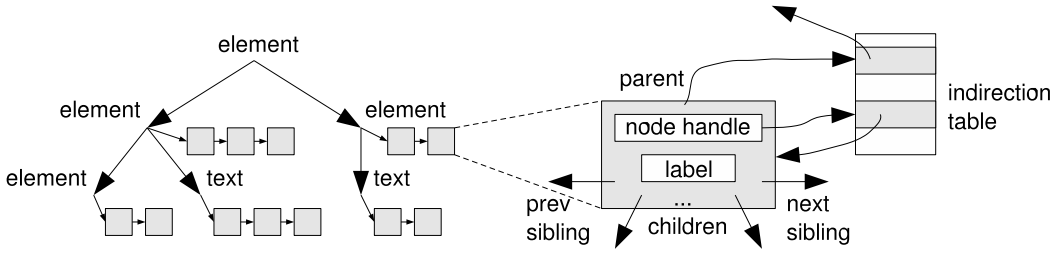


Abbildung 34: Speicherungsstrukturen in Sedna

Zur logischen Adressierung der Dokumentknoten setzt Sedna ein Verfahren ein, das jedem Knoten ein Intervall von Zeichenfolgen zuweist. Dieses kodiert alle enthaltenen Knoten des darunter liegenden Teilbaums und wird zur Verwaltung in der Indirektionstabelle in eine Zahlendarstellung transformiert. Die Zeichenfolgen erlauben die Bestimmung von Nachfahrenbeziehungen und durch beliebige Verlängerung das Einfügen von Knoten an jeder Position im Dokument. Dieses Vorgehen wird mit weiteren Nummerierungsschemata detaillierter in Abschnitt 3.2.8 behandelt.

Transaktionsverwaltung

Zur Transaktionsisolation wird zunächst auf Seitenebene der klassische *Fix-unfix*-Mechanismus [GR93] zur Konsistenzsicherung der Datenseiten angewendet. Zum Schutz der einzelnen XML-Knoten gegen parallele Zugriffe nebenläufiger Transaktionen wird das bekannte pessimistische RX-Sperrprotokoll [HR99] (siehe Abschnitt 3.3.1) eingesetzt, dessen Sperrmodi feingranular auf jedem einzelnen Knoten des XML-Dokuments angefordert werden können. Da jeder Knoten jedoch durch eine Intervallangabe adressiert wird, sind von Sperren nicht nur die Knoten selbst, sondern auch stets die darunter liegenden Teilbäume betroffen.

Bewertung

Sedna ist ein natives XML-Datenbanksystem, das zum Dokumentenzugriff nur die XQuery-Anfragesprache mit Erweiterungen für Modifikationen und Einbettung von SQL-Anweisungen zur Verfügung stellt. Die klassischen prozeduralen Schnittstellen SAX und DOM werden nicht unterstützt. Sedna verwaltet XML-Dokumente feingranular in einem beschreibenden Schemagraph, der im Grunde analog zum DataGuide einen Pfadindex für das Dokument darstellt. Knotenbeziehungen müssen daher mit zusätzlichen Verweisen verwaltet werden. Zur Rekonstruktion von Fragmenten oder gesamten Dokumenten kann keine sequentiellen Verarbeitung der Datenseiten erfolgen, sondern es muss eine Reihe von Navigationsoperationen durchgeführt werden. Nebenläufige Modifikationen sind zwar technisch auf Knotenebene möglich, allerdings werden sie durch das einfache RX-Sperrprotokoll, das nur Teilbäume sperren kann, auf Blattknoten eingeschränkt. Das einfache Sperrprotokoll verhindert ebenso die Realisierung einer externen Navigationsschnittstelle, da der Einstieg einer Anwendung auf dem Wurzelknoten mit einer R-Sperre stets das gesamte Dokument sperrt (siehe Abschnitt 3.3.1).

3.1.3 OrientX

OrientX [MWL+04] wird an der Renmin Universität China entwickelt und ist ein schemabasiertes XML-Datenbanksystem. Die Implementierung des Systems sowie aller Schnittstellen wird in C++ durchgeführt. Bis auf eine generierte Dokumentation der C++-Klassen liegen alle Unterlagen leider nur in Chinesisch vor.

Funktionalität

OrientX verwaltet alle XML-Dokumente in einem globalen Katalog, die Organisation der Dokumente in Kollektionen oder Verzeichnissen wird nicht erwähnt. Ein Dokument kann über seinen eindeutigen Namen adressiert werden. Der Zugriff auf die Dokumente ist mit einer SAX-, DOM- oder XQuery-Schnittstelle möglich, die jeweils die Funktionalitäten der Standards zur Verfügung stellen.

Architektur

Die Gesamtarchitektur des Datenbanksystems OrientX wurde bisher noch nicht publiziert, allerdings die des Speichersubsystems *OrientStore*, das nach [MLL+03] einem klaren Schichtenmodell folgt und in Abbildung 35 dargestellt ist.

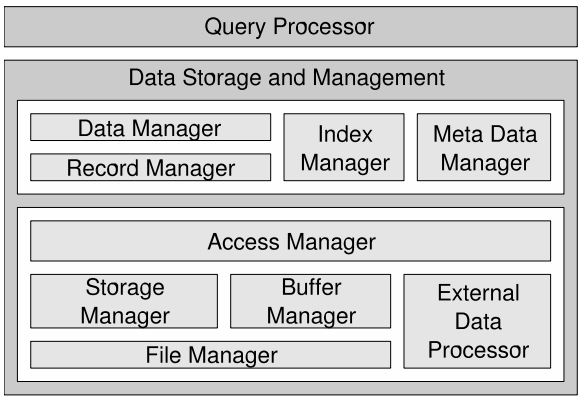


Abbildung 35: Architektur von OrientStore

Der *File Manager* sorgt für den Zugriff auf das Dateisystem und schreibt Datenseiten aus dem *Buffer Manager* in einzelne Dateien. Die Freispeicherverwaltung erledigt der *Storage Manager*; mit dem *External Data Processor* können weitere Datenquellen eingebunden werden. Auf alle Datenseiten bietet der *Access Manager* eine einheitliche Sicht. Der *Record Manager* wandelt die Datensätze zur Speicherung in den Datenseiten in ein internes Format, Im- und Exporte von XML-Dokumenten werden vom *Data Manager* durchgeführt. Der *Index Manager* stellt B*-Bäume zur Indexierung bereit.

Die Schemainformationen zu den gespeicherten XML-Dokumenten werden vom *Meta Data Manager* verwaltet. Dazu wird jedem im Dokument auftretenden Elementnamen über ein Vokabular eine 4-Byte lange ID zugewiesen und ein Schemagraph aufgebaut, der mit einem DataGuide verglichen werden kann. Dieser Graph erlaubt bei XQuery-Anfragen eine Transformation aller relativen Pfade („//“) oder Platzhalter („/*“) in die Menge aller möglichen absoluten Pfade. Des Weiteren wird, wie im folgenden Abschnitt beschrieben, die Schemabeschreibung zur Wahl der Speicherungsstruktur herangezogen.

Speicherungsstrukturen

OrientX unterstützt zwei Varianten [MLL+03] zur Speicherung eines XML-Dokuments: das *Element-Based Clustering* (*EBC*) und das *Logical Partition-Based Clustering* (*LPC*). In [MWL+04] werden die beiden Ansätze als *Clustering Element-Based* und *Clustering Subtree-Based* beschrieben.

EBC speichert jedes XML-Element als eigenständigen Datensatz ab, allerdings werden die Sätze nicht in Dokumentenordnung, sondern nach dem Elementnamen mit eventuellem Textinhalt in einem Cluster organisiert. Daraus folgt, dass zur Rekonstruktion der ursprünglichen Dokumentenstruktur für jedes Element ergänzende Zeiger zwischen Eltern- und Kindknoten verwaltet werden müssen. Im Grunde genommen entspricht EBC einem Textindex, der zusätzliche Strukurinformation trägt.

LPC partitioniert den Schemagraph eines XML-Dokuments in so genannte *semantische Blöcke*. Diese Blöcke werden nach einer sehr einfachen Heuristik bestimmt: Ein Knoten innerhalb eines Schemagraphen bildet die Wurzel eines semantischen Blocks, wenn er die Dokumentwurzel ist oder die Kardinalität * oder + und Kindknoten besitzt. Alle Instanzen eines semantischen Blocks werden in separaten Datensätzen gespeichert. Abbildung 36 verdeutlicht dies an einem Beispiel.

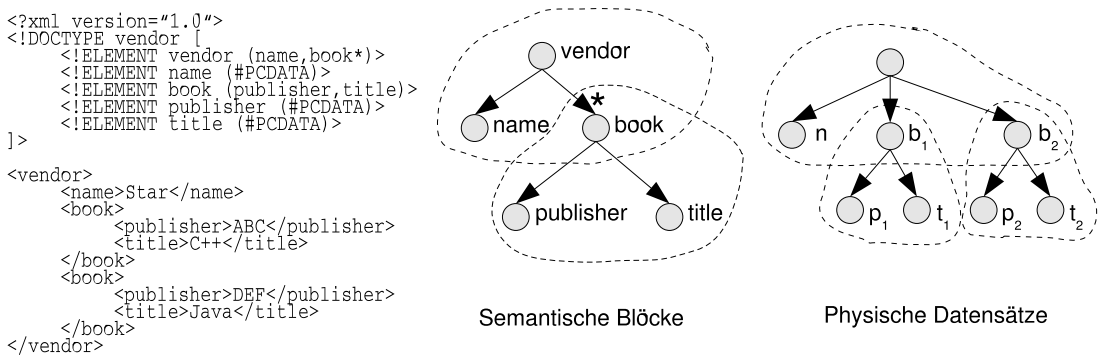


Abbildung 36: Bildung semantischer Blöcke in OrientX

Jedes XML-Element wird beim LPC-Verfahren zusätzlich über den *Index Manager* indexiert. Zur Adressierung eines Knotens wird das in [LM01] vorgeschlagene bereichsbasierte Numme-

rierungsverfahren benutzt (siehe Abschnitt 3.2.11). Dabei kann für zwei gegebene Knoten aus deren ID die Nachfahre-Vorfahre-Beziehung bestimmt werden, was zur Anfrageauswertung auf Knotenmengen ausgenutzt wird.

Transaktionsverwaltung

Konzepte zur Transaktionsverwaltung werden in den Publikationen zu OrientX nicht erwähnt, daher ist anzunehmen, dass eine Isolation nebenläufiger Transaktionen nur auf Datenseitenebene zur Vermeidung inkonsistenter Speicherungsstrukturen durchgeführt wird.

Bewertung

OrientX eignet sich nur zur Verwaltung von XML-Dokumenten, auf denen ausschließlich Suchanfragen durchgeführt werden. Für Änderungsoperationen ist eine Transaktionsverwaltung vorzusehen, die in den Veröffentlichungen nicht erwähnt wird. Zudem erzwingt das Nummerierungsschema häufige Reorganisationen der Dokumente. OrientX stellt die beiden Speicherungsstrukturen EBC und LPC zur Verfügung, die allerdings nicht automatisch bestimmt werden, sondern vom Anwender aufgrund der Dokumentenschemata und der zu erwartenden Anfragetypen festzulegen sind [MWL+04]. Die Vorteile des LPC-Verfahrens [MLL+03] basieren allerdings auf sehr vagen Annahmen. So soll zum einen ein Datensatz sehr viel kleiner als eine Datenseite sein, was jedoch voraussetzt, dass die Schemabeschreibung des XML-Dokuments ausreichende Kardinalitäten vom Typ * oder + enthält und die Struktur der Elemente nicht zu tief und nicht fest vorgeschrieben ist (Kardinalitäten vom Typ 1 erzeugen keine neuen semantischen Blöcke). Zum anderen sollen die semantischen Blöcke Knoten, die in Anfragen häufig zusammen zurückgeliefert werden, in einen Datensatz zusammenfassen. Dies ist jedoch aufgrund der recht trivialen Heuristik zu bezweifeln.

3.1.4 System RX

IBM eröffnet mit System RX [BCJ+05] als erster der großen DBMS-Hersteller den Reigen der Veröffentlichung zu nativen XML-Speicherungslösungen. System RX ist die prototypische Implementierung eines nativen XML-DBMS-Kerns, der nach und nach in das Datenbanksystem DB2 integriert werden wird.

Funktionalität

Da System RX die Infrastruktur der relationalen DB2 nutzt, steht damit eine stabile Plattform zur atomaren und dauerhaften Datenspeicherung zur Verfügung, die es gilt, mit Funktionalität zur Konsistenzsicherung und Transaktionsisolation für XML-Daten anzureichern. IBM bietet dafür den neuen Datentyp *xml*, der wie gewöhnliche SQL-Datentypen zur Definition von Tabellen benutzt werden kann. Allerdings verbirgt sich dahinter keine Funktionsbibliothek, die sämtliche XML-Operationen auf relationale Strukturen abbildet, sondern ein wirklich natives Speicherungsverfahren. Anfragen sind in System RX in XQuery und SQL/XML (siehe Abschnitt 2.3.6) möglich. Zur Beschleunigung der Anfrageauswertung können strukturbasierte, wertbasierte und Volltextindexe angelegt werden.

```
create table bibliographies (id integer, bib xml);
insert into bibliographies values  (1492,
                                    XMLParse('<?xml encoding="UTF-8"?>
                                              <book price="23.98"
                                                <lang>English</lang>...
                                              </book>'));
```

Abbildung 37: Bearbeiten einer Tabelle mit Datentyp *xml* in System RX

Abbildung 37 zeigt zwei SQL-Anweisungen, mit denen zunächst die Tabelle *bibliographies* erzeugt und danach ein Buch als XML-Dokument mit der SQL/XML-Funktion *XMLParse* eingefügt wird.

Auf der so definierten und gefüllten Tabelle können Anfragen nun wahlweise in XQuery oder SQL/XML gestellt werden. Abbildung 38 zeigt jeweils ein Beispiel für die Suche nach allen englischsprachigen Büchern mit einem Preis unter 80. Innerhalb der XQuery-Anfrage (Abbildung 38a) kann auf die Tabellenspalte *bib* mit dem Funktionsaufruf *xmlcolumn('BIBLIOGRAPHIES.BIB')* zugegriffen werden. Für die SQL/XML-Anfrage (Abbildung 38b) wird die Funktion *XMLTable* benutzt, die aus dem Ergebnis einer eingebetteten XQuery-Anfrage eine Relation *T* mit den Datentypen *double* für *T.price* und *xml* für *T.names* konstruiert.

```
a)  for $bib in xmlcolumn('BIBLIOGRAPHIES.BIB')/bib[lang/text()='English'],
        $book in $bib//book
    where $book/@price < 80
    return  <bookInfo>
                {$book/@price, $book/author/name}
            </bookInfo>

b)  select T.price,T.names
    from   bibliographies as B,
           XMLTable('for $bib in $doc/bib[lang/text()="English"],
                         $book in $bib/book
                     where $book/@price > 80
                     return $book'
                passing B.bib as 'doc',
                columns 'price' double path './@price',
                        'names' xml path './author/name') as T
```

Abbildung 38: XQuery- und SQL/XML-Anfrage in System RX

Architektur

Abbildung 39 zeigt die Architektur von System RX. Anfragen werden je nach Typ vom SQL/XML- oder XQuery-Parser analysiert und an einen hybriden *Compiler* übergeben. Da System RX nicht erzwingt, dass jedes XML-Dokument in einer Tabellenspalte einem Schema genügen muss, wird sowohl statische als auch dynamische Typisierung unterstützt. Das bedeutet, dass der *Compiler* u. U. in einem ersten Schritt zunächst Datentypen für einzelne Elemente der Anfrage bestimmen muss (*semantics*). Danach erfolgt ein eventuelles Umschreiben und Optimieren der Anfrage, wodurch in klassischer Weise der Anfrageplan entsteht, mit dessen Hilfe der Code zur Auswertung der Anfrage generiert wird.

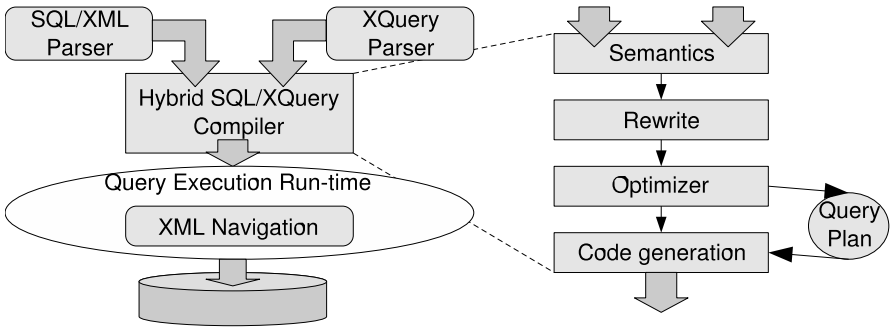


Abbildung 39: Architektur von System RX

Während der Anfrageauswertung (Abarbeitung des generierten Codes) erlaubt die Komponente *XML Navigation* den Zugriff auf die XML-Daten und die Eltern-, Geschwister- und Kindknotenbeziehungen der nativen Speicherungsstrukturen.

Speicherungsstrukturen

Zur Speicherung wird ein XML-Dokument in *Regionen* logisch zusammenhängender Knoten aufgeteilt. Eine Region wird in einer Datenseite als Datensatz abgespeichert. Das genaue Vorgehen zur Bestimmung des logischen Zusammenhangs von Knoten wird in [BCJ+05] allerdings nicht beschrieben. Ein Regionsdatensatz wird innerhalb einer Datenseite mit einem Offset-Verweis adressiert [HR99]; alle Regionen eines XML-Dokuments werden in einem *Regionenindex* verwaltet. Diese Speicherungsstruktur ist in Abbildung 40 dargestellt.

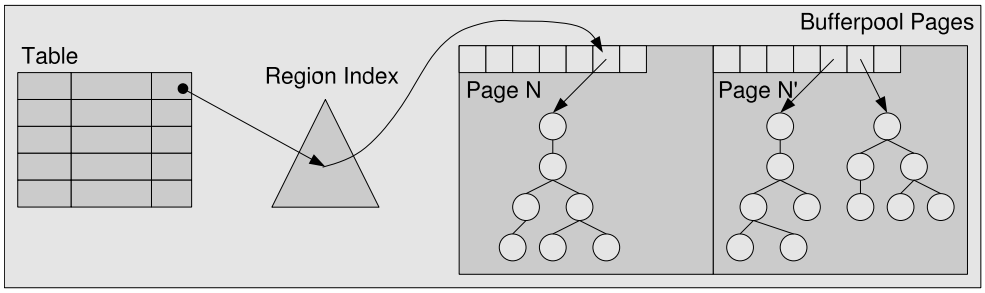


Abbildung 40: Speicherungsstrukturen in System RX

Eine Region enthält selbst alle Informationen über die Eltern-, Geschwister- und Kindbeziehungen der in ihr gespeicherten Knoten. Erfolgt eine Navigation im Dokument über die Regionsgrenzen hinweg, so muss mit einer Indirektion über den Regionenindex die betroffene Region ermittelt werden. Regionen werden initial in Cluster gruppiert, die während der Aktualisierung der Dokumente allerdings fragmentieren können. Um das Cluster wiederherzustellen, muss bei Bedarf ein Reorganisationswerkzeug angestoßen werden.

Transaktionsverwaltung

Zur Transaktionsverwaltung nutzt System RX auf Datenseitenebene die volle Funktionalität der bereits vorhandenen relationalen Infrastruktur. Somit werden alle Dokumentmodifikationen atomar und dauerhaft in die Datenbank eingebracht. Die Isolation nebenläufiger Transaktionen wird mit einem versionsbasierten Sychronisationsmechanismus gewährleistet. Dazu werden die verwalteten Regionen eines XML-Dokuments versioniert, sodass eine Transaktion immer eine bzgl. ihres Startzeitpunkts konsistente Sicht auf das Dokument zu sehen bekommt.

Bewertung

System RX präsentiert die prototypische Implementierung der nativen XML-Speicherungsstrukturen, die in dieser Form auch in das kommerzielle Datenbankprodukt DB2 der IBM einfließen werden. Die Nutzung der vorhandenen relationalen Infrastruktur wird System RX zu einer leistungsfähigen und zuverlässigen Komponente verhelfen.

Wenn die verwalteten Regionen, zu denen mehrere XML-Knoten zusammengefasst werden, sinnvoll gewählt werden, ist auch bzgl. der Transaktionsisolation ein hohes Leistungsvermögen möglich. Dazu ist jedoch eine Anpassung der Regionen an die zu erwartenden Anfragen und evtl. eine nachträgliche Korrektur nötig. Bei hohen Änderungsraten geht sehr schnell die Cluster-Struktur durch die Versionierung und damit die Möglichkeit der sequentiellen Datenver-

arbeitung verloren, sodass vermutlich häufig Reorganisationen während des Systembetriebs durchzuführen sind. Zudem werden die durch die Versionierung möglichen parallelen Änderungen in der nächsten DB2-Version noch nicht ausgenutzt, da bei einer Modifikation eines XML-Dokuments die betroffene Tabellenspalte noch exklusiv gesperrt wird.

3.1.5 Natix

Die Entwicklung von Natix [FHK+03] begann im März 1998 aus einem Projekt der Universität Mannheim mit dem Spektrum-Verlag, der Universität Erlangen und der Universität Passau. In diesem Projekt wurde die multimediale Aufbereitung von Molekulardatenbanken untersucht, worauf sich die Universität Mannheim dazu entschied, zur Datenhaltung ein natives XML-Datenbanksystem zu entwickeln. 1999 wurde das Startup-Unternehmen *data ex machina* ausgegründet, das seitdem die Entwicklung von Natix leitet. Die Ankündigung einer offiziellen Version von Natix ist allerdings seit mehreren Jahren unverändert auf der Webseite zu lesen.

Funktionalität

XML-Dokumente werden von Natix in einer Verzeichnishierarchie verwaltet, die über diverse Schnittstellen den Anwendungssystemen zugänglich gemacht werden. Dazu steht eine HTTP- und eine WebDAV-Schnittstelle zur Verfügung, das Dokumentenverzeichnis kann aber auch mittels eines Treibers in das Dateisystem eingehängt werden. Somit erhalten auch ältere Anwendungen (*Legacy-Applikationen*) Zugriff auf XML-Daten. An den Zugriffspfad der o. g. Schnittstellen können direkt XPath-Anfragen angehängt werden, sodass der Zugriff auf Fragmente bzw. Knotenmengen möglich wird.

Einzelne XML-Dokumente, XML-Fragmente oder Ergebnisse von XPath-Anfragen sind für eine Anwendung als aufbereitetes Textdokument oder über eine DOM- bzw. SAX-Schnittstelle zugreifbar.

Architektur

Die Architektur von Natix setzt sich aus den drei Schichten *Binding Layer*, *Service Layer* und *Storage Layer* zusammen und ist in Abbildung 41 dargestellt. Die beiden Schichten Service Layer und Storage Layer bilden zusammen die *Natix Engine*.

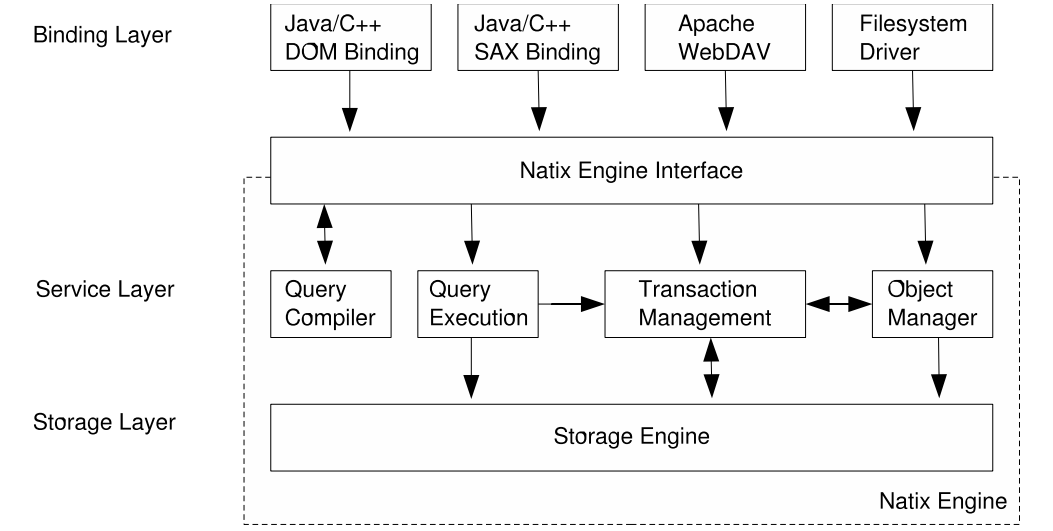


Abbildung 41: Architektur des Datenbanksystems Natix

In der Binding-Layer-Schicht erfolgt die Abbildung der Anwendungsdaten auf die Natix Engine. Hier sind die DOM- und SAX-Schnittstelle, ein WebDAV-Modul für den Apache-Webserver und der Dateisystemtreiber integriert. Anwendungsprogramme greifen direkt auf die Module des Binding Layer zu.

In der Schicht Service Layer sind alle Komponenten angesiedelt, die ergänzende Funktionalität zu den einfachen Speicher- und Ladeoperationen der Speicherungskomponente bieten. Dazu zählen die Anfrageverarbeitung, die Transaktionskontrolle und die Verwaltung von Objekten (z. B. XML-Dokumente bzw. -Fragmente, die aus gespeicherten Knoten rekonstruiert werden).

Persistente Datenstrukturen werden mit der Storage-Layer-Schicht in einzelnen Datenseiten verwaltet. Zur Speicherung der Seiten werden Dateien und direkter Festplattenzugriff (*raw disk access*) unter Linux und Solaris unterstützt.

Speicherungsstrukturen

Die Datenspeicherung in Natix [FHK+02a] erfolgt in Datenseiten fester Länge. Mehrere Seiten werden, wie in Datenbanksystemen üblich [HR99], zu Segmenten zusammengefasst, wobei alle Seiten innerhalb eines Segments die gleiche Länge aufweisen. Unter anderem werden Segmente für B-Bäume und Segmente für XML-Dokumente angeboten. Sobald eine Seite vom Externspeicher in den Hauptspeicher geladen wird, wird der Seite ein *page interpreter object* zugewiesen. Diese Objekte bieten den Komponenten des Datenbanksystems seitentyp-spezifische Methoden zum Zugriff auf die Datenseiten und abstrahieren dabei vom eigentlichen physischen Format der Seite. Die Page-Interpreter-Objekte spielen auch bei der Transaktionsverwaltung eine wichtige Rolle (siehe nächster Abschnitt).

XML-Dokumente werden in Natix als Bäume verwaltet, deren geordnete Knoten die Knoten des XML-Dokuments repräsentieren. Jeder Knoten erhält eine Bezeichnung. Die inneren Baumknoten, die XML-Elemente und -Attribute darstellen, erhalten ihre Bezeichnung aus einem Alphabet Σ_{Tag} (ein XML-Vokabular), das für jedes Segment definiert wird und für alle Dokumente eines Segments gültig ist. Die Blattknoten des Baums, die die textuellen Werte des XML-Dokuments repräsentieren, werden mit den entsprechenden Zeichenfolgen beschriftet. Neue Knoten können als Kind- oder Geschwisterknoten eingefügt und jeder Knoten (einschließlich des durch ihn als Wurzel definierten Teilbaums) gelöscht werden.

Zur Speicherung eines XML-Dokuments wird die logische Baumstruktur auf Sätze abgebildet [FHK+03], die in den Seiten der Segmente gespeichert werden. Da ein XML-Dokument gewöhnlich größer als eine Seite des Datenbanksystems ist, wird die Baumstruktur in mehrere Teilbäume zerlegt, sodass jeder Teilbaum jeweils in einem Satz in einer einzelnen Seite abgelegt werden kann. Dieses Prinzip der Transformation eines logischen Baums in einen physischen ist in Abbildung 42 dargestellt.

Um die strukturellen Beziehungen zwischen den auf Sätze aufgespaltenen Teilbäumen nicht zu verlieren, werden vier weitere Knotentypen eingeführt: *facade aggregate nodes*, *facade literal nodes*, *scaffolding proxy nodes* und *scaffolding helper nodes*. *Facade nodes* sind Knoten, die aus der logischen Baumdarstellung übernommen und auch in dieser Form direkt an eine Anwendung weitergereicht werden. Sie werden unterschieden in *aggregate nodes*, die im Innern des Baums liegen (f_1), und *literal nodes*, die die Blätter des Baums bilden (f_2 bis f_7). Für den strukturellen Zusammenhang zwischen Sätzen werden beim Splitvorgang *scaffolding nodes* erzeugt. Dabei verweisen *proxy nodes* (p_1 und p_2) auf einen Datensatz, der Kindknoten enthält, und *helper nodes* (h_1 und h_2) innerhalb eines Datensatzes auf die eigentlichen Kindknoten.

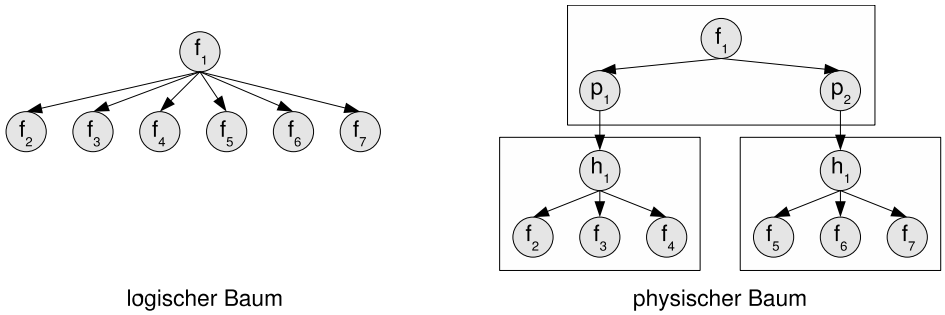


Abbildung 42: Verteilung eines logischen Baums auf Sätze in Natix

Wird nun ein Knotenwert geändert, und der Datensatz, der den Teilbaum mit diesem Knoten enthält, passt durch die Operation nicht mehr in eine Datenseite, so wird der Teilbaum wiederum in mehrere kleinere Teilbäume zerlegt und somit auf mehrere Sätze verteilt. Das System versucht dabei mit einem besonderen Algorithmus [FHK+02b], die entstehenden Teilbäume möglichst gleich groß zu gestalten. Der Benutzer kann den Aufteilungsvorgang beeinflussen, indem er in einer so genannten *Splitmatrix* Elementtypen angibt, die vom Algorithmus möglichst in einen gemeinsamen bzw. in verschiedene Teilbäume verlagert werden sollen.

Zur Unterstützung der Anfrageauswertung wird der *XASR*-Index (*eXtended Access Support Relations*) benutzt [FM00], der analog zum Nummerierungsschema in Abschnitt 3.2.14 alle Knoten des Dokuments in der Reihenfolge *left-most depth-first* durchzählt und dabei jedem Knoten die Nummer beim ersten und letzten Zugriff zuweist. Mit den so indexierten Bereichsangaben können Pfadausdrücke effizient ausgewertet werden.

Transaktionsverwaltung

Natix bietet die Verarbeitung atomarer, dauerhafter und serialisierbarer Transaktionen in einer Mehrbenutzerumgebung an. Konsistenzsicherung wird bis auf die Wohlgeformtheit von XML-Dokumenten nicht unterstützt. Dies wäre beispielsweise durch Schemaspezifikationen für einzelne Segmente denkbar.

Logging- und Recovery-Mechanismen wurden gegenüber den bekannten Maßnahmen aus relational Datenbanksystemen [HR99] für die XML-Verarbeitung erweitert. Für Datenseiten wird physiologisches Logging durchgeführt. Die Log-Sätze werden vom *page interpreter object* gesammelt und bei Freigabe der Seite ausgeschrieben. Da das *page interpreter object* durch das physiologische Logging Kenntnis über die Semantik der aufeinander folgenden Log-Sätze besitzt, kann vor dem Ausschreiben eine Optimierung vorgenommen werden. Werden beispielsweise innerhalb einer Seite ein XML-Teilbaum erzeugt und nacheinander Knoten in diesen Baum eingefügt, so können die einzelnen Log-Sätze dieser Operationen durch einen einzelnen Log-Satz zur Erzeugung des resultierenden Baums ersetzt werden. Dies wird als *subsidiary logging* bezeichnet. Umgekehrt sorgt das *annihilator undo* für eine Optimierung der Kompensation. Wird zum Beispiel ein Teilbaum erzeugt und in diesen eine Menge von Knoten eingefügt, so muss bei einem Transaktionsabbruch nicht jeder Knoten in umgekehrter Reihenfolge einzeln gelöscht, sondern der Teilbaum kann als Ganzes entfernt werden.

Zur Isolation nebenläufiger Transaktionen werden in den Natix-Publikationen vier pessimistische Sperrprotokolle diskutiert, die im Detail in Abschnitt 3.3.2 beschrieben werden. Die präsentierten Ergebnisse basieren jedoch alle auf Simulationen und nicht auf einer konkreten Implementierung in Natix, denn Natix benutzt merkwürdigerweise ein fast klassisches hierarchisches Synchronisationsprotokoll [ScR01]. Dazu wird lediglich das hierarchische Sperrver-

fahren mit speziellen Anwartschaftssperren [HR99] um eine zusätzliche *TOUCH*-Sperre erweitert. Springt eine Transaktion über einen Index direkt in einen Teilbaum ein, so wird zunächst der Pfad bis zur Dokumentwurzel bestimmt. Danach werden auf diesem Pfad (entsprechend dem hierarchischen Protokoll) von der Wurzel bis zum betroffenen Datensatz die erforderlichen Sperren angefordert. Da nun zwischenzeitlich eine parallel laufende Transaktion diesen Pfad hätte verändern können, wird der Pfad beim ersten Durchlauf mit *TOUCH*-Sperren geschützt. Dieses seltsame Vorgehen wird allerdings in einem realen System unweigerlich zu einer sehr hohen Deadlock-Rate führen, da sich „einspringende“ Transaktionen, die TOUCH-Sperren von den Blättern zur Wurzel anfordern, pausenlos mit ändernden Transaktionen, die Sperren von der Wurzel abwärts anfordern, blockieren werden.

Bewertung

Die Konzepte des Datenbanksystems Natix eignen sich für die Verarbeitung von XML-Dokumenten in einer transaktionalen Mehrbenutzerumgebung mit wenigen Änderungstransaktionen. Logging-, Recovery- und Synchronisationsmechanismen, sowie die gängigen XML-Schnittstellen sind vorgesehen. Vollkommen undurchsichtig ist die Implementierung eines fast klassischen hierarchischen Sperrprotokolls trotz der veröffentlichten XML-spezifischen Protokolle. Da die Synchronisation somit auf Datensatzebene durchgeführt wird, hat der Benutzer nur über den Eingriff in die physischen Speicherungsstrukturen mit der Splitmatrix die Möglichkeit, die Granularität der gesperrten Dokumententeile zu beeinflussen, bzw. muss bei systembedingten Umstrukturierungen eine Änderung der Granularität hinnehmen. Die Anforderung der *TOUCH*-Sperren von Baumblättern hinauf zur Wurzel erhöhen die Deadlock-Wahrscheinlichkeit enorm. Über den Einfluss von Änderungsoperationen auf die Indexstrukturen, die durch die Verwaltung von Bereichsangaben häufigen Reorganisationen ausgesetzt sind, wird keine Aussage getroffen.

3.1.6 Xindice

Das Datenbanksystem Xindice [XIN05a, XIN05b] (englisch „zeen-dee-chay“ gesprochen) wird von der Apache Software Foundation seit Dezember 2001 als Open-source-Projekt in Java entwickelt und ist zur Zeit in Version 1.1 verfügbar.

Funktionalität

Xindice kann als eigenständiger Datenbankserver laufen (sowohl als separater Prozess als auch als *Servlet* in einem *J2EE Web Application Server*) oder als Komponente in eine bestehende Anwendung eingebettet werden. Zum Zugriff auf die Daten liegt ein Kommandozeilenwerkzeug, eine XML-RPC-Schnittstelle und eine Implementierung der XML:DB API vor, über die auf XML-Dokumente mit SAX und DOM zugegriffen und XPath-Anfragen abgesetzt werden können. Die Modifikation der Dokumente erfolgt über den XUpdate-Vorschlag der XML:DB-Initiative (siehe Abschnitt 2.3.4). Wird Xindice eingebettet benutzt, so kann auch direkt über die Methoden der Xindice-Bibliotheken mit dem System kommuniziert werden, da die Anwendung und das Datenbanksystem in derselben virtuellen Maschine laufen.

XML-Dokumente werden in *Kollektionen* verwaltet. Dabei können Kollektion Dokumente aber auch wiederum Kollektionen enthalten. Zum Speichern eines XML-Dokuments wird eine Datei, die das Dokument enthält, und optional ein *Dokumentenschlüssel,* mit dem das Dokument adressiert wird, angegeben. Fehlt die Angabe eines Schlüssels, so wird dieser vom System automatisch vergeben. Abbildung 43 zeigt exemplarisch in Zeile 1 das Speichern des XML-Dokuments *fx102a.xml* unter dem Namen *fx102* in die Kollektion */db/data/products* und in Zeile 2 die Rekonstruktion des Dokuments in die Zieldatei *fx102b.xml*.

```
a) xindice add_document -c /db/data/products -f fx102a.xml -n fx102
b) xindice retrieve_document -c /db/data/products -n fx102 -f fx102b.xml
```

Abbildung 43: Datenzugriff unter Xindice

XPath-Anfragen können über eine gesamte Kollektion formuliert werden, aber nicht über Kollektionsgrenzen hinweg. Damit bei der Anfrageauswertung nicht auf alle Dokumente der betroffenen Kollektion zugegriffen werden muss, können für eine Kollektion Indexe auf Elementen und Attributen (spezifiziert durch deren Namen) angelegt werden [XIN05d].

Architektur

Die Implementierung von Xindice entspricht keinem Schichtenmodell, sondern es gibt nur eine Menge von zur Laufzeit interagierenden Klassen, die grob in Pakete für die Anwendungsintegration, den Datenbankkern und die XML-spezifische Verarbeitung untergliedert sind.

Die einzelnen Komponenten des Systems nutzen auch nicht einen einheitlichen Puffermechanismus. So gibt es beispielsweise die Klasse *DocumentCache*, die Methoden zum Zugriff auf Dokumente einer Kollektion als DOM-Bäume zur Verfügung stellt und dabei einen internen Puffer benutzt, allerdings werden z. B. Operationen auf Baumstrukturen direkt im Dateisystem durchgeführt.

Speicherungsstrukturen

Alle Daten werden von Xindice dokumentenbasiert [XIN05c] in Dateien verwaltet, die in Datenseiten fester Länge unterteilt sind. Dazu wird von einem zu speichernden XML-Dokument eine komprimierte Repräsentation als Byte-Folge erzeugt (*compressed DOM*), die als Datensatz gespeichert wird. Datensätze, die größer als eine Datenseite sind, werden über mehrere miteinander verkettete Seiten verteilt. Die Adressierung der Dokumente ist mit einem B*-Baum realisiert, dessen Schlüssel-Wert-Paare aus einem Dokumentenschlüssel und dem Verweis auf die erste Datenseite des Dokumentendatensatzes bestehen. Abbildung 44 zeigt diese Struktur.

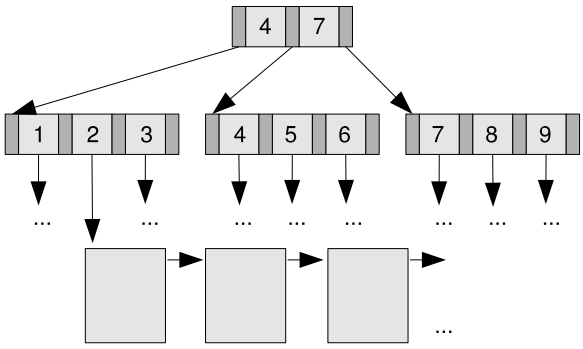


Abbildung 44: Dokumentenorganisation in Xindice

Die Modifikation von Knoten innerhalb eines XML-Dokuments erfolgt stets auf Basis des gesamten Dokuments selbst. Dazu wird aus der *Compressed-DOM*-Repräsentation ein gewöhnlicher DOM-Baum des Dokuments im Hauptspeicher materialisiert und auf diesem die Modifikation vollzogen. Danach wird der DOM-Baum wieder zurück in die *Compressed-DOM*-Variante überführt und das gesamte ursprüngliche XML-Dokument ersetzt.

Transaktionsverwaltung

Xindice stellt keine Mechanismen zur Gewährleistung von Transaktionseigenschaften zur Verfügung. Die Einzelkomponenten des Systems sind lediglich *thread*-sicher, sodass beim nebenläufigen Zugriff keine Inkonsistenzen in den Speicherungsstrukturen auftreten. Die Aufgabenliste der Projektseite beinhaltet noch die Sperrverwaltung (in einem ersten Schritt auf Dokumentenebene) und die Realisierung von Transaktionseigenschaften über mehrere Komponenten des Systems hinweg.

Bewertung

Das XML-Datenbanksystem Xindice eignet sich momentan nur zur Verwaltung einer großen Anzahl sehr kleiner XML-Dokumente in einem Umfeld, das nicht auf Transaktionseigenschaften angewiesen ist. Die Dokumente sind über den B*-Baum schnell adressierbar, werden jedoch zum Zugriff in eine DOM-Repräsentation umgewandelt, was die Verwaltung allzu großer Dokumente ausschließt. Auch die vollständige Ersetzung eines Dokuments bei jeglicher Änderung sollte behoben werden. Als positiv zu bewerten ist der Einsatz des Datenbanksystems als *Servlet*, sodass Xindice nach der Implementierung von Transaktionseigenschaften sicherlich im J2EE-Umfeld positioniert werden wird.

3.1.7 Tamino XML Server

Der Tamino XML Server [SW00, Sc01], kurz Tamino, ist ein kommerzielles XML-Datenbanksystem der Software AG und seit 1999 verfügbar. Der Schwerpunkt des Produkts liegt in der Speicherung von XML-Dokumenten und Anfragen auf Dokumentmengen. Die Software AG hat schon frühzeitig begonnen, die damals aktuelle Anfragesprache XQL (dem Vorläufer von XPath) mit Modifikationsoperatoren zu erweitern.

Funktionalität

XML-Dokumenten werden in Tamino in so genannten *Collections* gespeichert. Alle Dokumente, die ein gleichnamiges Wurzelelement besitzen, sind nochmals zu einem *Doctype* zusammengefasst. Dabei ist es nicht erforderlich, dass die Dokumente einem Schema entsprechen, sie müssen nur wohlgeformt sein. Es ist allerdings möglich, einem *Doctype* ein Schema zuzuweisen. Da die Document Type Definition (siehe Abschnitt 2.2.1) nicht sehr mächtig ist, mussten Schemata vor dem Erscheinen von XML Schema in einer proprietären Sprache definiert werden. Der schematische Aufbau einer Tamino-Datenbank ist in Abbildung 45 dargestellt.

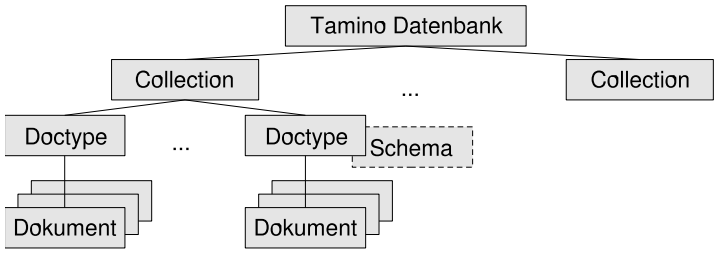


Abbildung 45: Aufbau einer Tamino-Datenbank

Zur Anfrageverarbeitung hat die Software AG die Semantik von XPath zur Verarbeitung der *Collections* erweitert. So bezieht sich die Anfrage „/“ stehts auf die Dokumentmenge innerhalb einer Kollektion, und es muss folglich der Identifikator eines Dokuments zu Beginn einer

Anfrage stehen. Mit „/*/...“ lässt sich mit dieser Erweiterung eine Anfrage auf einer gesamten Kollektion formulieren.

Zur Verbesserung der Textsuche wurde der *contains*-Operator ~= ergänzt. Damit wird nach [Sc03] die in Abbildung 46 gezeigt Funktionalität geboten. Die erste Anfrage liefert alle *Titel*-Elemente einer *Collection*, die das Wort XML enthalten, wobei Groß- und Kleinschreibung nicht beachtet wird. Die zweite Anfrage liefert alle Elemente, die die Zeichenfolge *at* enthalten; der Stern wird als Platzhalter verwendet. Möchte man eine Anfrage für Wörter formulieren, die benachbart stehen sollen (unabhängig von ihrer Reihenfolge), so wird, wie in Abbildung 46c zu sehen ist, das Schlüsselwort NEAR verwendet.

```
a)  /*[//Titel ~= "XML]
b)  /*[. ~= "*at*"]
c)  /*[@Name~="Harald" NEAR "Schöning"]
```

Abbikdung 46: Textsuche mit dem contains-Operator in Tamino

Anfragen werden an Tamino über eine HTTP-Schnittstelle geschickt, was den Zugriff auf XML-Daten über Unternehmensgrenzen hinweg bzgl. der Firewall-Problematik entschärft (lediglich der Zugriff auf einen Webserver-Port muss möglich sein). Dazu stellt die Software AG für alle gängigen Webserver Erweiterungen zur Verfügung, welche die Verbindungen zwischen Webserver und Datenbanksystem verwalten. Die HTTP-Schnittstelle ist WebDAV-fähig, sodass moderne Betriebssysteme die Tamino-Ressourcen auch in Dateisysteme integrieren können. Zusätzlich existiert eine Komponente namens *Extended Transport Service* (*XTS*), mit der auch der direkte Zugriff auf Tamino innerhalb eines Intranets möglich ist.

Architektur

Im Kern von Tamino arbeitet die *XML-Engine*, die hauptsächlich aus der *SQL-Engine* und der *X-Machine* [SW00] besteht. Beide Komponenten arbeiten auf einem gemeinsamen Datenspeicher, sodass es mit der Definition von *DataMaps* möglich ist, XML-Daten auf ein relationales Schema abzubilden, das über eine gewöhnliche SQL-Schnittstelle klassischen Datenbankanwendungen zugänglich gemacht wird. Die *X-Machine* führt die eigentliche XML-Verarbeitung zur Speicherung, Modifikation und Anfrage von Dokumenten durch. Eine Übersicht über die Architektur von Tamino zeigt Abbildung 47.

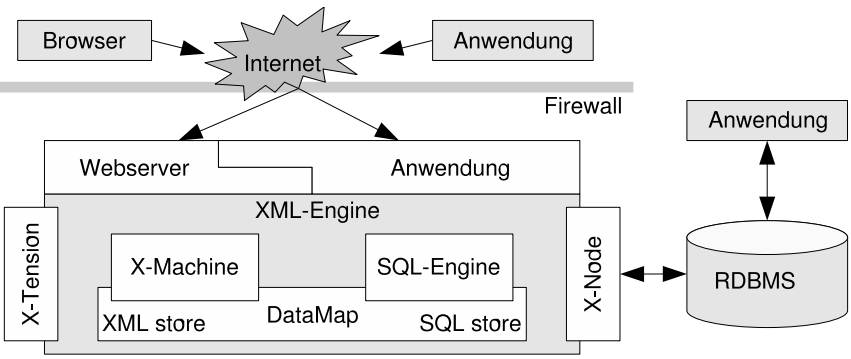


Abbildung 47: Die Architektur von Tamino

Zur Erweiterung des Systems stehen zwei Konzepte zur Verfügung. Zum einen können über *X-Nodes* relationale Datenbanken angekoppelt werden, deren Daten über Abbildungsvorschriften in neue XML-Dokumente überführt oder in bestehende XML-Dokumente integriert werden können. Somit wird es für eine Anwendung völlig transparent, ob sie nun auf nativ gespeicherte Daten zugreift, oder ob die Daten teilweise in externen Datenbanksystemen liegen. Das zweite Erweiterungskonzept erlaubt mit *X-Tensions* die Definition eigener Funktionen, die in Tamino integriert und in Anfragen verwendet werden können. Im Gegensatz zu *X-Nodes*, die nur den Zugriff auf Datenbanksysteme erlauben, ermöglichen *X-Tensions* darüber hinaus auch die Einbindung von ERP- oder Middleware-Systemen. Beide Konzepte stellen die Definition von Propagierungsregeln zur Verfügung, sodass Änderungen an den virtualisierten XML-Dokumenten auch in die Quellsysteme zurückgeschrieben werden können.

Speicherungsstrukturen

Zur Speicherung von XML-Dokumenten in einer Tamino-Datenbank werden von der *X-Machine* zwei persistente Teile angelegt.

Teil 1, der *Data Space,* beinhaltet die eigentlichen XML-Dokumente, die bzgl. der *Doctypes* in Cluster mit spezifischen Kompressionsalgorithmen organisiert werden [Sc03]. Weitere Informationen über die Speicherungsstrukturen für XML-Dokumente werden von der Software AG nicht publiziert.

Der *Index Space* verwaltet als zweiter Teil die Meta-Daten und Indexstrukturen des Systems. Zur Beschleunigung der Anfrageauswertung kann der Benutzer drei Indexarten anlegen. Der *Werteindex* übernimmt die vollständigen Werte von Elementen und Attributen, sodass damit Punkt- und Bereichsanfragen möglich werden. Zur korrekten Erkennung der Datentypen einzelner Werte ist ein Schema für die zu indexierenden Dokumente erforderlich. Der *Textindex* beinhaltet einzelne Wörter und deren relative Lage zur Beantwortung von Nachbarschaftsanfragen mit dem oben beschriebenen *contains*-Operator. Der *Strukturindex* speichert schließlich die Pfade innerhalb von Dokumenten. Er wird zur Anfrageoptimierung herangezogen und kann bei der Schemaevolution benutzt werden, um Dokumente zu identifizieren, die einer erneuten Validierung unterzogen werden müssen.

Transaktionsverwaltung

Alle Verbindungen zwischen Anwendungsprogrammen und Tamino werden in *Sessions* verwaltet. Innerhalb einer Session werden Anweisungen sequentiell verarbeitet. Dabei öffnet jede Anweisung eine Transaktion, innerhalb deren Kontext alle nachfolgenden Anweisungen atomar abgearbeitet werden, bis ein *Commit* oder *Rollback* erfolgt. Das heißt, ein Fehler, der während einer Anweisung auftritt, führt zur Rücksetzung der gesamten Änderungen der Anweisung, jedoch nicht zur Rücksetzung der Transaktion.

Analog zu SQL wird für jede Session eine Isolationsstufe [GR93] für die zukünftig zu startenden Transaktionen gewählt. Änderungsoperationen an XML-Dokumenten resultieren immer in einer exklusiven Sperre für das gesamte Dokument, die durch nebenläufige Transaktionen nur durch die Wahl einer Isolationsstufe, die das Lesen „schmutziger“ Daten zulässt, umgangen werden kann. In diesem Fall tritt durch das Lesen ohne Anforderung einer Lesesperre keine Blockierung auf.

Zur Gewährleistung der dauerhaften Speicherung und der Atomarität von Modifikationen einer Transaktion wird von Tamino der *Journal Space* und der *Log Space* geführt. Der *Journal Space* ist als Ringpuffer organisiert und ermöglicht das Rücksetzen bei einem Transaktionsabbruch durch Benutzerinteraktion (*rollback*) oder einen Systemabsturz. Der *Log Space* enthält die Archivkopien für die Wiederholung aller erfolgreich beendeter Transaktionen.

Bewertung

Der Tamino XML Server der Software AG eignet sich zur Verwaltung von XML-Dokumenten in transaktionalen Arbeitsumgebungen. Die Transaktionsisolation auf Dokumentebene erzwingt ein Operationsmuster, das typischerweise auf einer großen Anzahl eher kleinerer, in Kollektionen zusammengefasster Dokumente durchgeführt wird. Bemerkenswert ist die Möglichkeit, die volle Funktionalität des Systems über HTTP-Aufrufe zu nutzen, was die Positionierung des Systems als unternehmensübergreifender Datenspeicher im Internet erlaubt. Die Erweiterbarkeit mit *X-Nodes* und *X-Tensions* integriert beliebige Datenquellen in virtuelle XML-Dokumente. Auch wenn die Propagierung von Änderungen auf die Datenquellen möglich ist, sollte dies aufgrund der zu erwartenden Blockierungen durch Dokumentensperren nur im Einzelfall oder in niedrigen Isolationsstufen genutzt werden.

3.2 Nummerierungsschemata

Nummerierungsschemata werden zur eindeutigen Adressierung der Knoten eines XML-Dokuments eingesetzt. Dabei lassen sich die publizierten Ansätzen in drei Verfahrensklassen einteilen.

Wertbasierte Nummerierungen ordnen jedem Knoten einen eindeutigen Zahlenwert zu, der sich in einer konkreten Implementierung meist als Integer-Wert mit einer Länge von 4 Bytes darstellen lässt. *Präfixbasierte* Verfahren konstruieren die Adresse eines Knotens, indem die ID des Elternknotens als Präfix in die ID des Kindknotens mit einfließt. Die *bereichsbasierten* Ansätze schließlich weisen jedem Knoten als Adresse einen Wertebereich zu, der alle Wertebereiche der Adressen seiner Nachfahren umschließt. Eine Übersicht über alle im Folgenden diskutierten Ansätze und deren Klassifikation ist in Abbildung 48 gegeben.

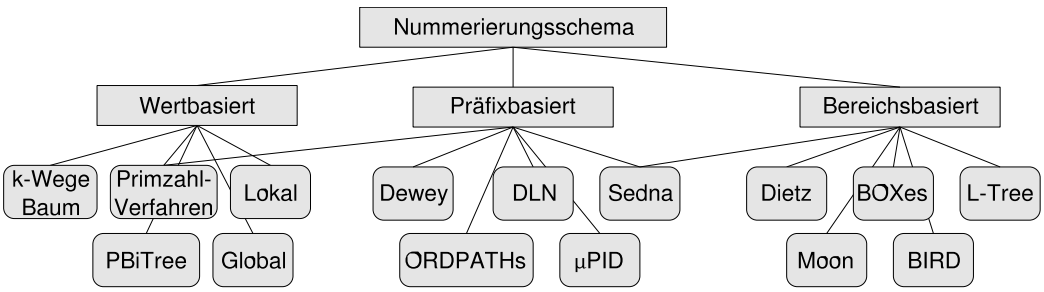


Abbildung 48: Klassifikation der XML-Nummerierungsschemata

Ursprünglich wurden Nummerierungsverfahren zur Beschleunigung von Pfadanfragen auf statischen XML-Dokumenten eingesetzt. Mit dem Einsatz transaktionsorientierter nativer XML-Datenbanksysteme, die die Struktur der Dokumente extrem dynamisieren, und komplexer Anfragesprachen wie XQuery stellen sich jedoch weitere Anforderungen an ein Nummerierungsschema bzgl. derer wir die Verfahren beurteilen werden:

- Das Nummerierungsverfahren sollte stabil gegen Einfüge- und Löschoperationen sein, d. h., die Adresse eines Knotens darf sich nicht aufgrund des Einfügens oder Löschens anderer Knoten ändern.
- Für den Einsatz hierarchischer Sperrprotokolle müssen sich aus der Adresse eines beliebigen Knotens die Adressen aller Vorfahren bis zur Dokumentwurzel berechnen lassen.

- Zur effizienten Rekonstruktion von XML-Fragmenten durch das sequentielle Lesen von Datenseiten müssen die Knotenadressen eine Ordnung definieren, die der Dokumentenordnung der XML-Knoten entspricht.
- Die Auswertung deklarativer Anfragen erfordert für zwei gegebene Knotenadressen die Entscheidbarkeit der *Preceding/Following*- bzw. *Preceding-Sibling/Following-Sibling*-Achsenbeziehung. Die Ermittlung der *Ancestor/Descendant*-Beziehung kann mit der Berechnung aller Vorfahren (siehe oben) durchgeführt werden.

3.2.1 k-Wege-Bäume

Eines der ältesten und einfachsten Verfahren zur Nummerierung von XML-Knoten besteht darin, das XML-Dokument als vollständigen k-Wege-Baum zu betrachten [LYY96]. Der Parameter *k* wird dabei vom Element mit der höchsten Anzahl von Kindknoten bestimmt; Elemente, die weniger Kinder besitzen, müssen mit *virtuellen* Knoten ergänzt werden, um die Vollständigkeit des Baums zu gewährleisten. Abbildung 49 zeigt einen Ausschnitt aus dem erweiterten Beispieldokument von Kapitel 2 und dessen Darstellung als vollständiger 2-Wege-Baum.

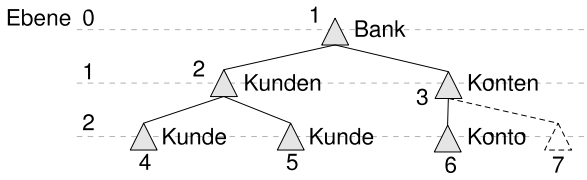


Abbildung 49: Knotennummerierung in einem k-Wege-Baum

Die Nummerierung der Knoten erfolgt durchgängig über alle Ebenen hinweg. Die Berechnung des Elternknotens (und damit rekursiv aller Vorfahren) ist durch $parent(x) = trunc((i-2)/k + 1)$ möglich. Im Gegensatz zu den Behauptungen in der Veröffentlichung ist eine Berechnung der Kindknoten direkt nicht möglich. Zwar liefert die Formel $child(x,i) = k(i-1) + x + 1$ die ID des theoretisch i-ten Kindes von *x*, allerdings erzwingt das Auffüllen mit virtuellen Knoten, dass ein Zugriff auf das Dokument erfolgen muss, um zu überprüfen, ob ein Kind an der berechneten Position auch wirklich existiert. Das Einfügen neuer Knoten in das Dokument ist nur auswirkungslos, falls zufällig eine noch nicht besetzte virtuelle Knotenadresse betroffen ist. Ansonsten muss eine Neunummerierung aller Knoten der *Following*-Achse des eingefügten Knotens erfolgen. Sollten virtuelle Knoten zwischen zwei vorhandenen Knoten zugelassen sein, können Löschoperationen ohne nachfolgende Reorganisation durchgeführt werden. Da die Nummerierung der einzelnen Knoten in Ebenenordnung stattfindet, erfolgt eine Speicherung des Dokuments bzgl. der Knoten-IDs nicht in Dokumentenordnung.

3.2.2 PBiTree

Der Name des PBiTree-Verfahrens [WJL+03] steht als Abkürzung für einen *Perfect Binary Tree*, der als balancierter vollständiger binärer Baum definiert ist. Um die Idee als neu erscheinen zu lassen, haben die Autoren nicht die klassische Nummerierung in Dokumenten- oder Ebenenordnung gewählt, sondern nummerieren die Knoten des binären Baums vom „äußeren linken“ Knoten beginnend (*Zwischenordnung*). Das hat zur Folge, dass auch die Formel zur Berechnung eines Vorfahren von *x* auf Höhe *h* mit $2^{h+1} \cdot trunc(x / 2^{h+1}) + 2^h$ wesentlich wissenschaftlicher erscheint. Um nun die *Ancestor/Descendant*-Beziehung zweier

Knoten des XML-Dokuments aus dem PBiTree zu ermitteln, ist eine Abbildung des Dokuments auf den PBiTree zu finden, die zwei triviale Forderungen zu erfüllen hat. Zum einen müssen zwei Knoten *u* und *v*, die im XML-Dokument identisch sind, auch im PBiTree identisch sein, zum anderen muss *u* im PBiTree ein Vorfahre von *v* sein, genau dann, wenn *u* auch im XML-Dokument ein Vorfahre von *v* ist. Die Nummer innerhalb des PBiTree stellt dann die Adresse des XML-Knotens dar. Abbildung 50 zeigt solch eine mögliche Zuordnung von XML-Knoten zu PBiTree-Knoten. Wie diese Zuordnung effizient für mehrere Millionen Knoten eines großen XML-Dokuments zu finden ist, bleibt allerdings offen.

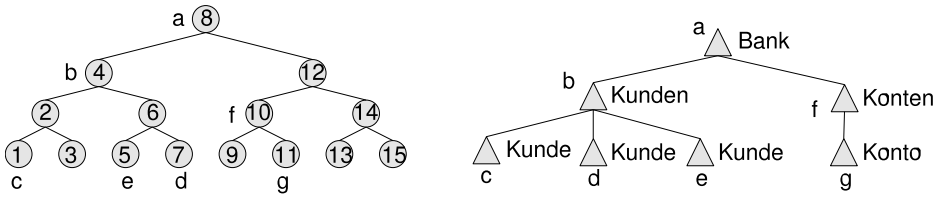


Abbildung 50: Nummerierung mit dem PBiTree

Die Stabilität des Nummerierungsschemas bei Einfügeoperationen hängt von der Abbildung des Dokuments auf den PBiTree ab und erfordert bei vollständiger Füllung eines Asts natürlich eine Neunummerierung. Löschoperationen haben keine Auswirkungen auf die bereits zugewiesenen Knoten-IDs. Eine Berechnung aller Vorfahren ist mit der oben angegebenen Formel möglich, die Speicherung der Knoten muss aufgrund der recht willkürlichen Zuordnung nicht gezwungenermaßen in Dokumentenordnung erfolgen. Da ein Knoten im PBiTree nur Vorfahre aller seiner Nachfolger im XML-Dokument sein muss, können die *Preceding*- und *Following*-Achse nicht über die Knotenadresse ausgewertet werden.

3.2.3 Primzahlbasierte Nummerierung

In [WLH04] wird eine primzahlbasierte Nummerierung für XML-Dokumente vorgeschlagen. Dabei werden zunächst zwei Strategien zur Vergabe der Knoten-IDs untersucht, die in Abbildung 51 gezeigt werden. Der *Bottom-up*-Ansatz weist jedem Blatt aufsteigend eine jeweils eindeutige Primzahl zu. Die Adressen der inneren Knoten und der Dokumentwurzel werden durch das Produkt der Kindknotenadressen berechnet. Man kann leicht erkennen, dass für XML-Elemente, die nur ein Kind besitzen, eine Sonderregelung vorzusehen ist, da das Elternelement sonst dieselbe Adresse wie das Kind erhält. Die Autoren gehen darauf nicht ein. Mit der Bottom-up-Nummerierung ist ein Knoten *x* ein Vorfahre von *y*, wenn *id(x) mod id(y) = 0* gilt. Da durch die Multiplikation die Zahlenwerte der Knotenadressen jedoch sehr schnell ansteigen, wird das zweite Verfahren der Top-down-Nummerierung favorisiert.

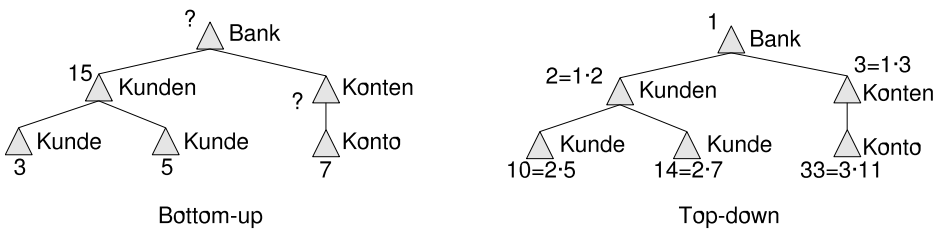


Abbildung 51: Bottom-up- und Top-down-Nummerierung mit Primzahlen

Beim *Top-down*-Ansatz erhält der Wurzelknoten die Adresse 1. Die ID jedes Nachfahren berechnet sich aus einer aufsteigend vergebenen Primzahl multipliziert mit der ID des Elternknotens (die Knoten in Abbildung 51 wurden in Ebenenordnung eingefügt). Die Zahlenwerte steigen nun nicht mehr ganz so schnell wie beim Bottom-up-Verfahren an, und die Eigenschaft, ob Knoten *x* ein Vorfahre von *y* ist, lässt sich mit der Formel *id(y) mod id(x) = 0* überprüfen. Zusätzlich können mit der Primfaktorzerlegung aus der Adresse eines Knotens alle seine Vorfahren bestimmt werden, da sich die Knoten-ID als Produkt aller IDs der Vorfahren darstellen lässt. Somit ist das primzahlbasierte Top-down-Verfahren auch zur Klasse der präfixbasierten Nummerierungsschemata zu zählen. Die Nutzung ist allerdings aufgrund der Komplexität der Primfaktorzerlegung eher zweifelhaft.

Das bisher beschriebene Top-down-Verfahren hat weiterhin das Problem, dass es nicht ordnungserhaltend ist, d. h., für zwei gegebene Knoten kann mit Hilfe ihrer Adressen nicht deren Reihenfolge bestimmt werden. Dazu schlagen die Autoren die Anwendung des Chinesischen Restsatzes vor, der eine Abbildung der Adresse eines Knotens auf seine globale Ordnungsnummer liefert. Die neu gewählte Primzahl für einen Knoten, die in die Berechnung seiner Adresse ergänzend zur Elternadresse einfließt, wird als *self label* bezeichnet. Für jeden Knoten des XML-Dokuments wird nun eine Kongruenzgleichung aufgestellt, die dem *self label* die Ordnungsnummer zuweist (Abbildung 52). Nachdem für dieses Gleichungssystem die Lösung *x*, die der Chinesischen Restsatz für die teilerfremden Primzahlen garantiert, bestimmt ist, kann mit *x mod y* die Ordnungsnummer jeder Knotenadresse *y* berechnet werden.

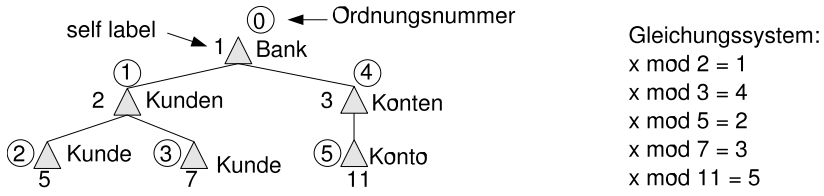


Abbildung 52: Der Chinesische Restsatz bei der Primzahlnummerierung

Dass die Anwendung des Chinesischen Restsatzes zur Gewährleistung der Dokumentenordnung bei einem XML-Dokument mit mehreren Millionen Knoten auch ein System mit ebenso vielen Gleichungen produziert und dieses System bei beliebigen Knoteneinfügungen, die die Ordnungsnummern verschieben, neu gelöst werden muss, stellt wohl den Albtraum jedes Datenbankentwicklers dar.

3.2.4 Globale und lokale Nummerierung

Die Autoren in [TBS+02] untersuchen zur Adressierung von Knoten in XML-Dokumenten die so genannte *globale* und *lokale* Nummerierung, deren Kombination als *Dewey-Nummerierung* im nächsten Abschnitt 3.2.5 vorgestellt wird.

Das globale Nummerierungsschema weist jedem Knoten seine Ordnungsnummer in Dokumentenordnung zu (Abbildung 53a). Diese Nummer lässt allerdings keinerlei Rückschlüsse auf Knotenbeziehungen bzgl. einer Pfadachse zu. Einfüge- und Löschoperationen erfordern stets die Neunummerierung aller nachfolgenden Knoten, was durch die Berücksichtigung von Lücken bei der Vergabe der Ordnungsnummern etwas entschärft werden kann.

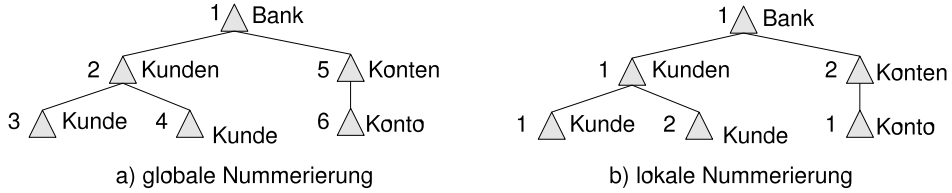


Abbildung 53: Globale und lokale Nummerierung von XML-Dokumenten

Eine leichte Verbesserung der Stabilität der Knotenadressen verspricht die lokale Nummerierung. Hierbei werden alle Knoten innerhalb jeder Ebene eines Teilbaum bei 1 beginnend durchnummeriert. Das hat zur Folge, dass beim Einfügen oder Löschen eines Knotens nur die nachfolgenden Geschwisterknoten neu nummeriert werden müssen. Allerdings ist dieses Verfahren für eine konkrete Implementierung völlig ungeeignet, da offensichtlich keine eindeutigen Knotenadressen vergeben werden.

3.2.5 Dewey-Nummerierung

Die Dewey-Nummerierung für XML-Dokumente [TBS+02] basiert auf der Dewey-Dezimalklassifikation [DEW] und ergibt sich aus der Kombination der globalen und lokalen Nummerierung (Abschnitt 3.2.4). Dazu werden, wie bei der lokalen Nummerierung, alle Knoten in einer Teilbaumebene aufsteigend nummeriert. Diese lokale Nummer bezeichnen wir im Folgenden als *Division*. Um die globale Information zu erhalten, wird die Adresse des Kindknotens mit der Adresse des Elternknotens durch Anhängen eines Punkts und des *Division*-Werts innerhalb der Ebene gebildet. Abbildung 54 verdeutlicht dies an unserem Beispielfragment.

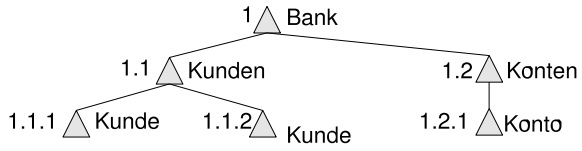


Abbildung 54: Nummerierung von Knoten mit der Dewey-Dezimalklassifikation

Das Einfügen oder Löschen eines Knotens erfordert die Neunummerierung aller nachfolgenden Geschwister und, zusätzlich zur lokalen Nummerierung, auch die deren Nachfolger in der *Descendant*-Achse. Wenn die *Division*-Werte der Knoten innerhalb einer Ebene über einen größeren Wertebereich äquidistant verteilt werden (und somit Lücken bei der Nummerierung erlaubt sind), erfordert das Löschen von Knoten keine Reorganisation und die Neunummerierung bei Einfügeoperationen kann etwas hinausgezögert werden.

Die Berechnung aller Vorfahren ist bei diesem Nummerierungsschema sehr einfach durch das Abtrennen der jeweils letzten *Division* möglich (Elternknoten von *1.1.2* ist *1.1*, dessen Elternknoten wiederum ist *1*). Da mit der Dewey-Dezimalklassifikation das Dokument (bspw. mit einer einfachen lexikographischen Sortierung) ordnungserhaltend verwaltet wird, können auch für zwei gegebene Adressen die Achsen *Preceding/Following* bzw. *Preceding-Sibling/Following-Sibling* ausgewertet werden. Dies wird in Abschnitt 4.3.3 detaillierter diskutiert.

3.2.6 ORDPATHs

ORDPATHs [NNP+04] benutzen zur Knotenadressierung die Dewey-Nummerierung, verbessern allerdings entscheidend das Reorganisationsverhalten. So ist bei beliebigen Einfüge- oder Löschoperationen eine Neunummerierung von existierenden Knoten nie erforderlich. Dieses Verhalten wird durch eine geschickte Vergabe der Knotenadressen erreicht, die bei der initialen Nummerierung nur ungerade *Division*-Werte vergibt (Abbildung 55a).

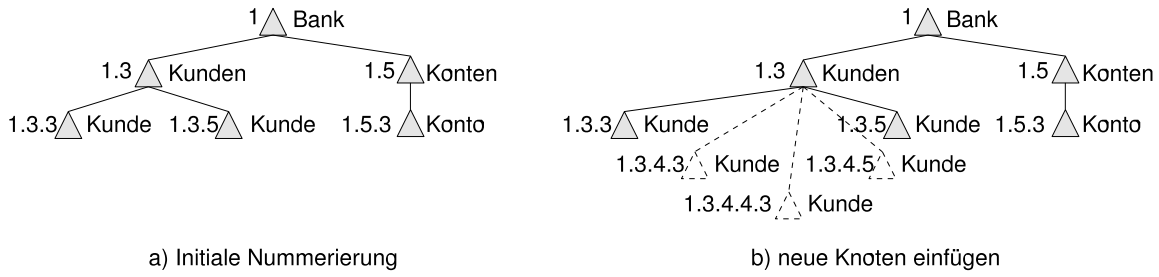


Abbildung 55: Knotenadressierung mit ORDPATHs

Zum Einfügen neuer Knoten werden die nicht-vergebenen geraden *Division*-Werte benutzt und mit wiederum ungeraden Werten ergänzt. So kann zwischen die IDs *1.3.3* und *1.3.5* die ID *1.3.4.3* und direkt dahinter die *1.3.4.5* eingefügt werden (Abbildung 55b). Der *Division*-Wert *1* ist für die Dokumentwurzel und für Attribute (hier nicht behandelt) reserviert. Für die Semantik der vergebenen Werte ist es wichtig, dass nur *ungerade* Division-Werte eine neue Ebene im Dokument einleiten und gerade *Division*-Werte zum Erzeugen neuer „Lücken“ für Knoteneinfügungen dienen. Auf dieser Grundlage kann die Ebene eines Knotens leicht aus der Anzahl der ungeraden *Division*-Werte in seiner Adresse bestimmt werden. Auch der Elternknoten lässt sich leicht berechnen: Dazu werden die letzte *Division* (immer ungerade) und danach zusätzlich alle evtl. am Ende befindlichen geraden *Division*-Werte entfernt. Auf dieser Weise kann rekursiv der gesamte Pfad von einem Kontextknoten zur Dokumentwurzel ermittelt werden. Analog zum Dewey-Nummerierungsschema lassen sich auch die übrigen XQuery-Pfadachsen effizient auswerten.

Zur physischen Speicherung der ORDPATHs-Adressen erläutern die Autoren in [NNP+04] eine kompakte Repräsentation auf Bit-Ebene. Wir haben diese Speicherungsstruktur für unsere auf ORDPATHs basierenden DeweyIDs weiter optimiert und erläutern dies genauer in Abschnitt 6.1.4.

3.2.7 DLN

Einen ähnlichen Ansatz wie ORDPATHs verfolgt das Konzept des *Dynamic Level Numbering* (*DLN*) [BR04]. Die initiale Nummerierung der Knoten erfolgt äquivalent zur Dewey-Dezimalklassifikation (dargestellt in Abbildung 56a). Zum Einfügen neuer IDs zwischen zwei existierenden Knotenadressen im Dokument wird – da alle Zahlenwerte durch die klassische Dewey-Nummerierung bereits zugeordnet sind – die ID des kleineren Knotens bzgl. der Dokumentenordnung mit einem Schrägstrich und einer darauf folgenden Nummer erweitert. Dies ist am Beispiel der Adresse *1.1.1/1* zwischen *1.1.1* und *1.1.2* in Abbildung 56b zu sehen.

Zur physischen Speicherung der Adresse wird pro Ebene des Dokuments eine zuvor festgelegte Bit-Anzahl verwendet; der Punkt wird mit *0* und der Schrägstrich mit *1* kodiert. Wenn nun für alle Ebenen des Dokuments jeweils 2 Bits vorgesehen sind, wird die Adresse *1.1.1/1* binär als *01 0 01 0 01 1 01* dargestellt. Diese Darstellung kann nur in die logische Repräsentation zurück

überführt werden, wenn die verwendete Bit-Länge pro Ebene bekannt ist. Wie die zu verwendende Bit-Länge gewählt wird, welche Auswirkungen dies auf ein Dokument mit bspw. jeweils 1.000.000 und 100 Kindknoten auf gleicher Ebene in verschiedenen Teilbäumen hat oder wie auf die pro Dokument und Ebene individuellen Bit-Längen effizient bei potenziell über 100.000 XML-Dokumenten zugegriffen wird, sprechen die Autoren nicht an. Ist die logische Darstellung der DLN-Adressen aber erst einmal rekonstruiert, bietet sie dieselben Vorteile wie die der ORDPATHs.

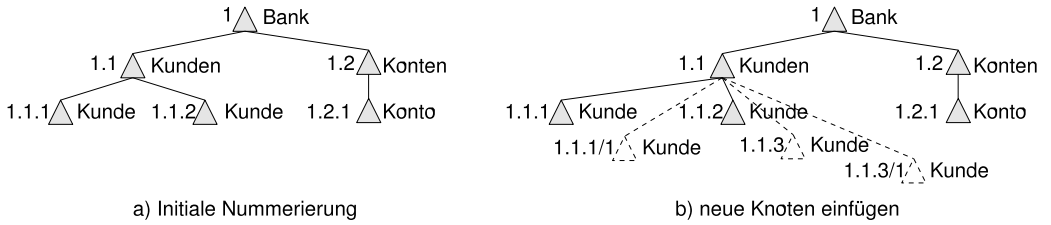


Abbildung 56: Knotenadressierung mit Dynamic Level Numbering

Durch die Festlegung auf eine vorgegebene Bit-Länge pro Ebene wird ein weiterer Unterschied zur ORDPATHs- oder Dewey-Nummerierung deutlich. Das Anhängen neuer Knoten erzwingt beim Erreichen des maximalen physischen Wertebereichs (in Abhängigkeit der Bit-Länge) die Erweiterung der logischen DLN-Adresse mit einem Schrägstrich. DLN bildet nach dem oben beschriebenen Prinzip alle Werte der Nummerierung direkt auf ihre binäre Darstellung ab [BR04]. Das hat zur Folge, dass in unserem Beispiel mit 2 Bits pro Ebene effektiv nur die Werte *1*, *2* und *3* dargestellt werden können. Wird nun (wie in Abbildung 56b zu sehen) hinter dem Knoten *1.1.3* ein neuer Knoten angehängt, muss dieser die DLN-Adresse *1.1.3/1* erhalten, weil die Adresse 1.1.4 nicht mit 2 Bits pro Ebene dargestellt werden kann.

3.2.8 Nummerierungsschema in Sedna

Eine weitere Variante der präfixbasierten Nummerierung kommt im XML-Datenbanksystem Sedna (siehe Abschnitt 3.1.2) zum Einsatz. Hierbei werden die Knotenadressen als Zeichenfolgen durch das Aneinanderhängen von Buchstaben gebildet [PN04]. Eine Knotenadresse *n* besteht aus einem *Präfix p* und einem *Delimiter d* und wird als $n=(p_n,d_n)$ dargestellt. Da somit ein Bereich definiert wird, kann dieses Nummerierungsschema auch zu den bereichsbasierten gezählt werden. Wird die lexikographische Ordnung verwendet, können alle Adressen so vergeben werden, dass das Intervall $[p_n,p_nd_n]$ alle Nachfahren im Teilbaum von p_n enthält und somit die *Ancestor/Descendant*-Achse ausgewertet werden kann. Ähnlich wie bei den zuvor vorgestellten Verfahren beruht die Idee zur Gewährleistung der Stabilität der bereits vergebenen Adressen darin, dass zwischen zwei Zeichenfolgen stets eine weitere durch das Anhängen einer Buchstabenfolge konstruiert werden kann. Abbildung 57 zeigt an unserem Beispielfragment wie Knotenadressen in Sedna vergeben werden können.

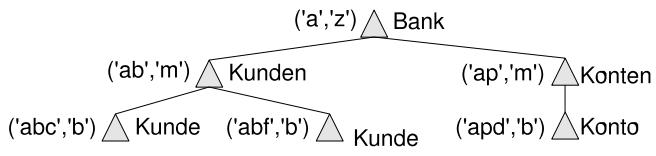


Abbildung 57: Knotenadressierung in Sedna

Da man durch das beliebige Anhängen von Buchstaben zur Konstruktion neuer Adressen bei Einfügeoperationen aus einer Adresse nicht die Ebene des entsprechenden Knotens berechnen kann, ist nur die Auswertung der *Preceding*- bzw. *Following*-Achse, aber nicht der *Preceding-Sibling*- bzw. *Following-Sibling*-Achse möglich. Betrachtet man die Publikationen [PN04] und [GFK04] scheint der Algorithmus zur Vergabe der Knoten-IDs sehr willkürlich zu sein. Die wirklich interessante Frage, wie eine Buchstabenfolge zur Speicherung auf ein internes numerisches Format abzubilden ist, bleibt mit einer Bemerkung als *nicht signifikant* im Dunkeln.

3.2.9 µPID

µPID stellt ein sehr einfaches präfixbasiertes Nummerierungsschema auf Bit-Ebene dar. Für die Wurzel wird keine Adresse kodiert, für alle anderen Knoten des XML-Dokuments wird zunächst ein DataGuide [GW97] aufgebaut. Somit kann jedem existierenden Pfad eine eindeutige *Pfadnummer* zugewiesen werden. Für jeden Pfad des Dokuments wird daraufhin die maximale Anzahl *n* der Kinder bestimmt, sodass $log_2(n)$ die benötigte Bit-Länge zur Nummerierung der Kindknoten liefert. Die *Positionsnummer* jedes Knotens wird aus der Nummer des Elternknotens durch Anhängen der eigenen Position gebildet. Die eindeutige Adresse jedes Knotens besteht aus der Pfadnummer und der Positionsnummer. Abbildung 58 zeigt die Nummerierung unseres Beispielfragments mit dem µPID-Schema.

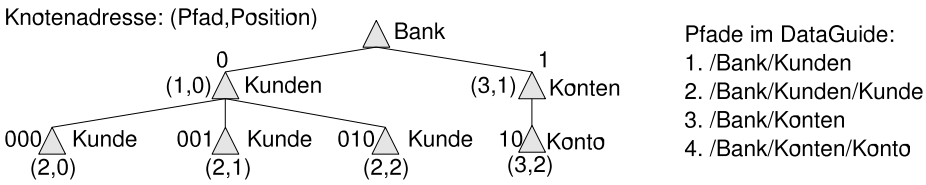


Abbildung 58: Knotenadressierung mit µPIDs

Stabilität der Knotenadressen bei Einfüge- oder Löschoperationen ist aufgrund der festen Bit-Längen bzgl. der Anzahl der Kindknoten nicht gegeben. Die Reorganisation könnte durch eine gleichmäßige Verteilung der IDs auf einen größeren Wertebereich nur etwas verzögert werden, allerdings kann durch die so entstehenden „Lücken“ auch das i-te Kind eines gegebenen Kontextknotens nicht mehr direkt berechnet werden, da Größe und Füllgrad der Lücken unbekannt sind. Eine Speicherung der Knoten in Dokumentenordnung und Auswertung der XQuery-Pfadachsen ist (nur) mit Hilfe des DataGuide möglich, da dieser Auskunft über die Reihenfolge der Pfade und die Pfadnummern von Elternknoten gibt. Hier stellt sich allerdings wiederum die Frage, mit welchem Aufwand diese Auswertung geschieht, da in einer konkreten Systemimplementierung die DataGuides für mehrere 100.000 Dokumente nicht vollständig im Hauptspeicher verwaltet werden können.

3.2.10 Nummerierungsschema nach Dietz

Ursprünglich gehen alle bereichsbasierten Nummerierungsschemata auf Paul F. Dietz zurück [Di82], der jedem Datenelement in einer Baumstruktur eine Positionsnummer während des *Preorder*- und *Postorder*-Durchlaufs zuweist. Beim *Preorder*-Durchlauf wird klassischerweise die Dokumentenordnung durchlaufen, indem rekursiv von der Wurzel zum ersten Kind navigiert wird und, wenn ein Blatt erreicht ist, der nächsten Geschwisterknoten adressiert wird. Der *Postorder*-Durchlauf beginnt beim „linken äußeren“ Knoten, der von der Dokumentenwurzel aus durch iteratives Bestimmen des ersten Kindes ermittelt werden kann. Danach wird zur Nummerierung jeweils zum nächsten Geschwisterknoten navigiert. Falls es sich um den letzten

Knoten innerhalb der Teilbaumebene handelt, wird die Nummerierung wiederum beim „linken äußeren“ Knoten des nächsten Geschwisterknotens des Elternknotens fortgeführt. Abbildung 59 verdeutlicht dieses Vorgehen.

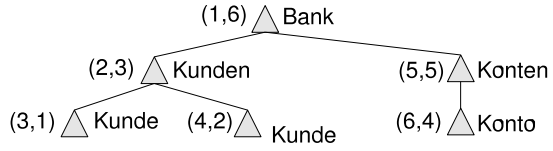


Abbildung 59: Nummerierungsschema nach Dietz

Das Nummerierungsschema nach Dietz erlaubt die Speicherung von XML-Knoten in Dokumentenordnung, da diese genau dem *Preorder*-Durchlauf entspricht. Die direkte Berechnung der Vorfahren ist nicht möglich, allerdings kann die *Ancestor/Descendant*-Beziehung ermittelt werden: Ein Knoten *x* ist Vorfahre eines Knoten *y*, wenn die *Preorder*-Nummer von *x* kleiner der *Preorder*-Nummer von *y* und die *Postorder*-Nummer von *x* größer der *Postorder*-Nummer von *y* ist. Die übrigen Pfadachsen können nicht ausgewertet werden. Aufgrund der sequentiellen Knotennummerierung in *Pre-* und *Postorder* können auch keine Knoten eingefügt oder gelöscht werden, ohne eine Reorganisation anzustoßen.

3.2.11 Nummerierungsschema nach Li und Moon

Li und Moon [LM01] verbessern das Verfahren von Dietz, indem die Bereichsgrenzen nicht durch *Pre-* und *Postorder* bestimmt werden, sondern durch eine *Ordnungsnummer* und eine *Bereichsgröße* (dieses Schema wird in [CTZ+01] als *Durable Node Numbers* bezeichnet und mit einer Versionierung von Teilbäumen ergänzt). Die Bereichsgröße muss mindestens so groß wie die Anzahl aller Nachfolger im Teilbaum sein, die Ordnungsnummer ergibt sich aus der Ordnungsnummer des vorherigen Geschwisterknotens erhöht um dessen Bereichsgröße + 1. Beim ersten Kind kann zwischen der eigenen Ordnungsnummer und der des Elternknotens eine „Lücke“ für zukünftige Einfügungen gelassen werden. Abbildung 60 zeigt eine Nummerierung nach Li und Moon mit (bis auf die Berücksichtigung der Anzahl der Nachfolger) willkürlich gewählten Bereichsgrößen.

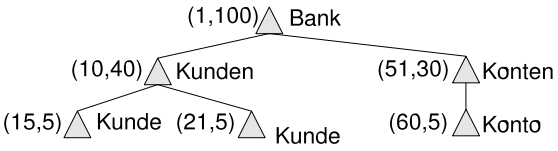


Abbildung 60: Nummerierung nach Moon und Li

Eine Speicherung der XML-Knoten in Dokumentenordnung ist anhand der Ordnungsnummern nach wie vor möglich. Zur Auswertung der *Ancestor/Descendant*-Beziehung ist die Formel von Dietz zu modifizieren: Ein Knoten *x* ist Vorgänger eines Knotens *y*, wenn die Ordnungsnummer von *y* zwischen der Ordnungsnummer von *x* und deren Summe mit der Bereichsgröße von *x* liegt. Die direkte Berechnung aller Vorfahren eines Knotens ist nicht möglich, allerdings können Einfügeoperationen bis zur Belegung aller freien Ordnungsnummern ohne Neunummerierung durchgeführt werden.

Li und Moon sagen nichts über die initiale Wahl der Ordnungsnummern und Bereichsgrößen beim Speichern eines Dokuments aus. Dafür schlägt das folgende BIRD-Nummerierungsschema einen Algorithmus vor.

3.2.12 BIRD

Das *Balanced Index-based numbering scheme for Reconstruction and Decision* (BIRD) annotiert die Knoten eines XML-Dokuments ähnlich zum Ansatz von Moon und Li. Die Bereichsgrößen werden bei BIRD allerdings als *Gewichte* bezeichnet und algorithmisch bestimmt.

Für ein XML-Dokument wird zunächst ein DataGuide aufgestellt, dessen Blätter jeweils die Gewichte *1* erhalten. Danach werden alle übrigen Gewichte von den Blättern ausgehend bis zur Wurzel bestimmt. Dabei wird für jeden Knoten der Pfad *p* von der Wurzel zu dessen Elternknoten betrachtet und die maximale Anzahl *n* der Kinder bestimmt, die ein Knoten im XML-Dokument mit Pfad *p* besitzt. Die so ermittelte Anzahl *n* wird dem Elternknoten als *n+1* multipliziert mit dem Gewicht des ersten Kindes als *Vorgewicht* zugewiesen. Das Gewicht eines Knotens ist schließlich durch das maximale Vorgewicht aller Geschwisterknoten bestimmt. Abbildung 61 illustriert dieses Vorgehen (Vorgewichte werden in Klammern dargestellt). Der Pfad */Bank/Kunden* hat im XML-Dokument höchstens 2 Kinder (die beiden *Kunde*-Elemente) und erhält daher das Vorgewicht *(2+1)*1=3*. Der Pfad */Bank/Konten* hat maximal 1 Kind und erhält das Vorgewicht *(1+1)*1=2*. Das Maximum beider Vorgewichte *3* wird nun *Kunden* und *Konten* im DataGuide als Gewicht zugewiesen. Zuletzt wird das Gewicht des Pfads */Bank* ermittelt, der im Dokument 2 Kinder hat und somit das Vorgewicht *(2+1)*3=9* erhält. Da es zu */Bank* im DataGuide keine Geschwisterknoten gibt, sind Gewicht und Vorgewicht identisch.

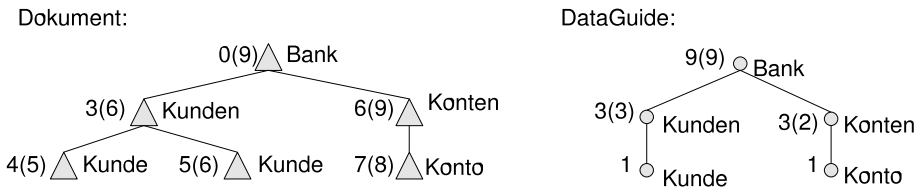


Abbildung 61: Knotenadressierung mit BIRD

Nachdem alle Gewichte der DataGuide-Knoten berechnet sind, werden die Positionsnummern der Knoten im XML-Dokument einfach über die durch die Gewichte bestimmten Bereiche äquidistant verteilt. Da die Gewichte zur Bestimmung der Knotenadressen im DataGuide vermerkt sind, kann mit dieser Information (im Gegensatz zu den bisher betrachteten bereichsbasierten Nummerierungsschemata) für einen beliebigen Knoten mit dem Gewicht des Elternpfads direkt die Adresse des Elternknotens aus *elternid(x) = id(x) - (id(x) mod elternpfadgewicht(x))* berechnet werden.

Da die Positionsnummern aufsteigend vergeben werden, erlaubt dies in Kombination mit der Berechnung der Elternadressen die Auswertung aller XQuery-Pfadachsen und die Speicherung aller Knoten in Dokumentenordnung. Einfügeoperationen können allerdings nur die vom Nummerierungsschema produzierten „Lücken“ ausnutzen, ohne eine Reorganisation der Positionsnummern zu erzwingen. Da die Gewichte im DataGuide jeweils aus den höchsten Gewichten aller Kinder bestimmt werden, müssen Dokumente, die in vielen Zweigen eine ähnliche Anzahl von Kindknoten aufweisen, häufig neu nummeriert werden. In unserem Beispielfragment in Abbildung 61 ist dies nach der initialen Nummerierung schon beim Einfügen eines weiteren *Kunde*-Elements erforderlich.

3.2.13 BOXes

Der *B-Tree for Ordering XML* (*BOX*) [SHY+05] beschreibt ein Nummerierungsschema, das wir schon in ähnlicher Weise in den ersten Versionen unseren XTC-Prototypen (siehe Kapitel 6) implementiert, jedoch aufgrund der geringen Leistungseigenschaften bei der Navigation und der Verwaltung von Sperren wieder verworfen haben.

Das zu nummerierende XML-Dokument wird dabei vollständig mit öffnenden und schließenden Tags sequentiell in den Blättern eines B*-Baums gespeichert. Die Einträge im B*-Baum sind über so genannte *regular labels* aufzufinden. Jedes Element der Baumrepräsentation des XML-Dokuments erhält als Adresse einen Bereich, dessen Anfangs- bzw. Endwert die Seitennummer des öffnenden Tags bzw. schließenden Tags darstellt. Damit dieser Bereich gegen alle Modifikationen am Dokument stabil bleibt, wird zwischen den Bereichswerten und den Seitennummern eine weitere Indirektion über die *Label-Identifier*-Tabelle eingeführt. Abbildung 62 zeigt diese Zusammenhänge.

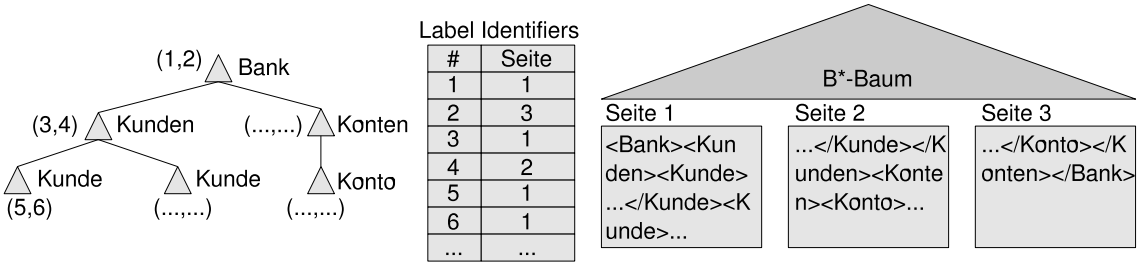


Abbildung 62: Knotenadressierung mit BOXes

Werden nun neue Knoten in das Dokument eingefügt oder Knoten aus dem Dokument gelöscht, so können einige bereits vorhandene Knoten durch Reorganisationsmaßnahmen im B*-Baum neue *regular labels* erhalten, weil sie in andere Seiten verschoben werden müssen. Durch eine Aktualisierung der Verweise in der *Label-Identifier*-Tabelle bleiben die Bereichsadressen der Knoten im XML-Dokument jedoch unverändert.

Das Problem der *label identifiers* besteht darin, dass für neu vergebenen *labels* der einzufügenden Knoten keinerlei Voraussetzungen zu erfüllen sind [SHY+05], d. h. sie in der Regel sequentiell vergeben werden. Daher können für zwei gegebene Knotenadressen bzgl. einer Achsenbeziehung aus den Adressen selbst keinerlei Rückschlüsse gezogen werden. Die Autoren äußern sich zumindest bzgl. der Auswertung der *Ancestor/Descendant*-Achse. Dazu werden für zwei Adressen zunächst über deren vier *label identifiers* die Datenseiten bestimmt, in denen die betroffenen Start- und End-Tags gespeichert sind. Somit werden die *label identifiers* in *regular labels* aufgelöst. Danach erfolgt für jedes *regular label* ein erneuter Abstieg von der B*-Baum-Wurzel zum gespeicherten Tag. Da BOXes in den inneren Seiten des B*-Baums zusätzlich zu den Seitenverweisen auch die Anzahl der dort verwalteten Knoten vermerkt, kann während der Navigation zum Blatt die absolute Ordinalzahl des gespeicherten Tags innerhalb des Dokuments aufsummiert werden. Mit den so berechneten Ordinalzahlen der Bereichsadressen ist ein Knoten x Vorfahre eines Knotens y, wenn die Bereichsadresse von x die von y umschließt. Der Aufwand mit vier initialen Seitenzugriffen und vier darauf folgenden Zugriffen über die Höhe des Baums für eine einzige Achsenauswertung lässt dieses Verfahren für die XML-Anfrageverarbeitung mit typischerweise mehreren 10.000 Knoten jedoch eher als ungeeignet erscheinen.

3.2.14 L-Tree

Der *L-Tree* [CMB+94] weist jedem Knoten zur Adressierung einen Bereich zu, der durch die Ordinalzahlen der Start- und End-Tags in der textuellen Darstellung des XML-Dokuments bestimmt wird. Dies entspräche einer Bereichsdefinition aufgrund der globalen Nummerierung (siehe Abschnitt 3.2.4). Um die Reorganisation des Dokuments bei Einfügeoperationen etwas hinauszuzögern, wird das globale Nummerierungsschema mit „Lücken“ versehen, die dadurch gebildet werden, dass über dem Dokument ein virtueller B*-Baum aufgebaut wird, der als *L-Tree* bezeichnet wird. Der Fan-out des Baums bestimmt die Zuordnung der XML-Knoten zu den Blättern und somit deren Ordinalzahlen (bei voller Belegung der Blätter). Eine Nummerierung des Beispielfragments mit einem Fan-out von *5* zeigt Abbildung 63.

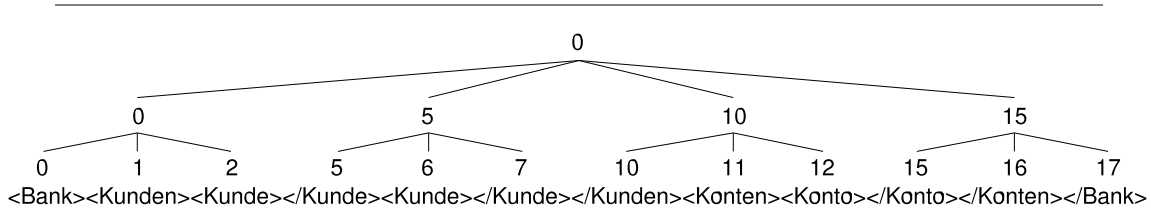


Abbildung 63: Nummerierung der Tags im L-Tree

Die Adresse eines Knotens wird als Bereich von der Ordinalzahl des Start-Tag bis zur Ordinalzahl des End-Tag dargestellt. Somit können (wie bei allen bereichsbasierten Verfahren) durch die Enthaltenseinseigenschaft und Ordnung von Intervallen die *Ancestor/Descendant-* und *Preceding/Following*-Pfadachsen für zwei gegebene Adressen ausgewertet werden, allerdings nicht die *Preceding-Sibling-* oder *Following-Sibling*-Achse, da die Adresse des Elternknotens nicht berechnet werden kann. Die sequentiell vergebenen Ordinalzahlen ermöglichen die Speicherung der XML-Knoten in Dokumentenordnung. Wenn alle „Lücken“ der Nummerierung ausgenutzt sind, muss auch bei diesem Schema eine Reorganisation durchgeführt werden. Diese kann aber, da die Adressvergabe auf einer Baumstruktur basiert, enorme Auswirkungen haben. Wird in den B*-Baum in Abbildung 63 (Fan-out 5) ein neuer erster Kunde eingefügt (jeweils ein Eintrag für Start- und End-Tag), so muss eine Teilung des ersten B*-Baum-Blatts erfolgen, was die Neunummerierung des gesamten Dokuments bewirkt.

3.3 Transaktionsisolation

Im Gegensatz zur Vielfalt der publizierten Nummerierungsschemata ist die Isolation nebenläufiger Transaktionen in XML-Datenbanksystemen ein noch kaum untersuchtes Themengebiet. Dies liegt womöglich daran, dass zum einen zu Beginn der XML-Euphorie eine Vielzahl von Verfahren zur Abbildung von XML-Dokumenten auf relationale Datenstrukturen untersucht wurde, die bereits eine Isolation für die modellinhärenten Objekte bereitstellen. Ähnlich verhält es sich bei XML-Datenbanksystemen, die keinen XML-spezifischen Kern besitzen und „lediglich“ XML-Schnittstellen anbieten. Zum anderen beinhalten die Standardisierungsprozesse der Anfragesprachen zur Zeit noch keine Modifikationsoperatoren.

Die im Folgenden vorgestellten Isolationsprotokolle lassen sich zunächst in pessimistische und optimistische Verfahren unterscheiden, wobei nur eine Arbeit den optimistischen Ansatz wählt. Die pessimistischen Protokolle lassen sich weiterhin bzgl. ihrer Sperren auf Knoten-, Kanten- oder Pfadbasis unterscheiden. Abbildung 64 zeigt eine Übersicht dieser Klassifikation.

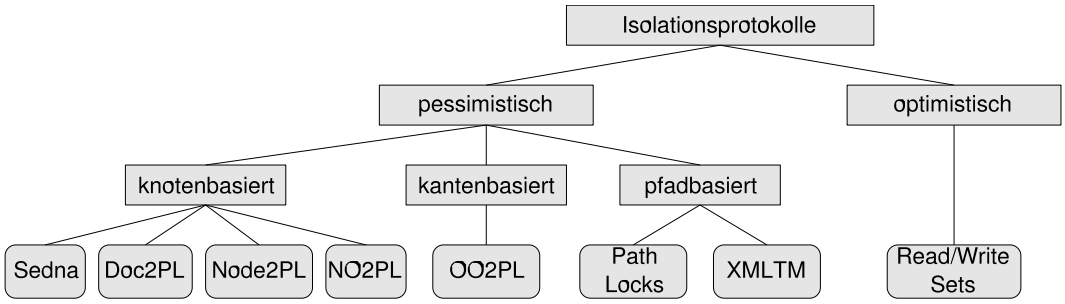


Abbildung 64: Klassifikation der Isolationsprotokolle

3.3.1 Ein einfaches RX-Sperrprotokoll

Das native XML-Datenbanksystem Sedna (siehe Abschnitt 3.1.2) implementiert zur Transaktionsisolation ein einfaches RX-Sperrprotokoll, das neben einer Lese- und Schreibsperre noch die bekannte Update-Sperre [HR99] zur Vermeidung von Deadlocks einsetzt.

Die drei möglichen Sperrmodi werden auf den einzelnen Knoten des XML-Dokuments angefordert. Da das Nummerierungsschema in Sedna (siehe Abschnitt 3.2.8) jeden Knoten mit dem Bereich seines darunter liegenden Teilbaums adressiert (Bereichsgrenzen werden präfixbasiert dargestellt), betrifft jede Sperranforderung auch immer den gesamten Teilbaum. Die Autoren stellen dies als besondere Eigenschaft zur Vermeidung von Phantomen dar [PN04]. Für eine navigationsorientierte Schnittstelle (Sedna bietet zur Zeit allerdings nur XQuery an) und die Auswertung deklarativer Anfragen ohne Indexunterstützung hat dies allerdings katastrophale Folgen, da ein Einstieg auf der Dokumentwurzel stets eine Sperre auf dem gesamten Dokument bewirkt, was für eine operationale Datenbasis ein indiskutabler Zustand ist.

Zudem bietet das RX-Verfahren keine Anwartschaftssperren wie bei hierarchischen Protokollen an. Somit kann bei der Sperranforderung auf einem Kontextknoten die Überprüfung der Kompatibilität nicht mit den für den Kontextknoten verwalteten Sperren durchgeführt werden, sondern es muss entweder auf jedem Knoten des Teilbaums ein Kompatibilitätstest erfolgen, oder es müssen stets alle verwalteten Sperren des Systems getestet werden.

3.3.2 Doc2PL, Node2PL, NO2PL und OO2PL

Helmer, Kanne und Moerkotte präsentieren in [HKM01] vier Sperrprotokolle, die das *Document Object Model* (siehe Abschnitt 2.3.2) zugrunde legen. Die Auswertungen der Protokolle in [HKM03] und [HKM04] wurden allerdings in einer Simulationsumgebung mit zufällig gewählten Zugriffsmustern durchgeführt, weswegen wir ihre Leistungsfähigkeit hier nicht diskutieren. Der Meta-Locking-Ansatz unseres XTC-Prototyps (siehe Kapitel 6) erlaubt uns die Integration der Protokolle in eine echte Mehrbenutzerumgebung. Daher wird deren Einfluss auf die Transaktionsverwaltung detailliert bei den Leistungsmessungen in Kapitel 7 untersucht.

Doc2PL

Das Doc2PL-Protokoll stellt das einfachste Verfahren dar und bedarf keiner umfangreichen Erläuterungen, da es auch nicht auf einem Datenmodell für XML-Dokumente basiert.

Alle Operationen auf einem Dokument werden auf Dokumentebene synchronisiert, indem stets Lese- oder exklusive Schreibsperren für das gesamte Dokument vor dem Zugriff beantragt werden. Das Protokoll simuliert somit bspw. das Verhalten mehrerer Transaktionen, wenn ein XML-Dokument als *character large object* (*CLOB*) in einem relationalen Datenbanksystem als Attributwert eines einzigen Tupels gespeichert wird.

Node2PL

Node2PL sperrt beim Zugriff auf einen Kontextknoten stets dessen Elternknoten und hindert somit andere Transaktionen in Bereiche mit nicht freigegebenen Änderungen vorzudringen. Die Sperrmodi, die eine Transaktion anfordern kann, werden in *Strukturspерren*, *Inhaltsperren* und *Einsprungsperren* unterschieden.

Während der reinen Navigation werden Strukturspерren im Lesemodus (*T - traverse*) auf den Elternknoten der besuchten Kontextknoten angefordert. Soll die Struktur des Dokuments von einer Transaktion durch Einfügen oder Löschen von Knoten verändert werden, so muss dafür eine Exklusivsperre (*M - modify*) angefordert werden. Beim Lesen oder Ändern von Knotenwerten müssen dagegen Inhaltsperren erworben werden. Die entsprechenden Sperrmodi werden mit *S* (*shared*) und *X* (*exclusive*) bezeichnet. Zusätzlich synchronisiert das Node2PL-Protokoll auch Aufrufe der DOM-Operation *getElementById()*, die zu dem gegebenen Wert eines ID-Attributs das besitzende Element liefert. Dazu muss vor dem Einsprung in das XML-Dokument eine IDR-Sperre auf dem betroffenen Element angefordert werden. Dies ist erforderlich, da das Node2PL-Protokoll keine Anwartschaftssperren bereitstellt, und nur so gewährleistet werden kann, dass sich der Einsprungspunkt nicht in einem Teilbaum befindet, der von einer anderen Transaktion gerade modifiziert wird. Umgekehrt erfordert das Löschen eines XML-Knotens und dem darunter liegenden Teilbaum mit Node2PL, dass zuerst der gesamte Teilbaum durchsucht und für jedes Element, das ein ID-Attribut besitzt, eine IDX-Sperre angefordert werden muss.

Abbildung 65a zeigt die Kompatibilitätsmatrizen der Struktur-, Inhalt- und Einsprungsperren gemäß der Schreibweise in [HR99]. Die Matrixzeilenköpfe beinhalten die bereits vorhandenen Sperrmodi einer Transaktion, während die Matrixspaltenköpfe die angeforderten Modi einer weiteren Transaktion zeigen; + bzw. – kodieren die Kompatibilität bzw. Inkompatibilität. Diese Schreibweise werden wir im Folgenden immer verwenden.

Strukturspерren

	-	T	M
T	+	+	-
M	+	-	-

Inhaltsperren

	-	S	X
S	+	+	-
X	+	-	-

Einsprungsperren

	-	IDR	IDX
IDR	+	+	-
IDX	+	-	-

a) Kompatibilitätsmatrizen

Strukturspерren

	-	T	M
T	T	T	M
M	M	M	M

Inhaltsperren

	-	S	X
S	S	S	X
X	X	X	X

Einsprungsperren

	-	IDR	IDX
IDR	IDR	IDR	IDX
IDX	IDX	IDX	IDX

b) Konversionsmatrizen

Abbildung 65: Kompatibilitäts- und Konversionsmatrizen von Node2PL

Hat eine Transaktion bereits eine Sperre auf einem Knoten erhalten und benötigt für eine Folgeoperation eine weitere Sperre, so muss, wenn pro Knoten und Transaktion nur eine Sperre verwaltet wird, eine neue Sperre eingerichtet werden. Diese muss sowohl dem bereits eingerichteten Sperrmodus als auch dem neu angeforderten gerecht werden. Diesen Vorgang bezeichnet man als *Sperrkonversion*. [HKM01] trifft keine Aussage über die Konversion von Sperren und die damit verbundenen Probleme. In einer konkreten Implementierung treten Sperrkonversionen jedoch sehr häufig auf, sodass diese Funktionalität von uns in [Lu05] hinzugefügt werden musste. Die Sperrkonversionen werden durch Konversionsmatrizen beschrieben und sind für das Node2PL-Protokoll in Abbildung 65b zu sehen. Die Matrixzeilenköpfe beinhalten den bereits gehaltenen Sperrmodus einer Transaktion auf einem XML-Knoten. Die Matrixspaltenköpfe zeigen den derselben Transaktion angeforderten Sperrmodus auf demselben Knoten. Der mit einer Spalte und Zeile adressierte Eintrag der Matrix liefert den resultierenden Sperrmodus, der für die Transaktion auf dem Knoten einzutragen ist.

NO2PL

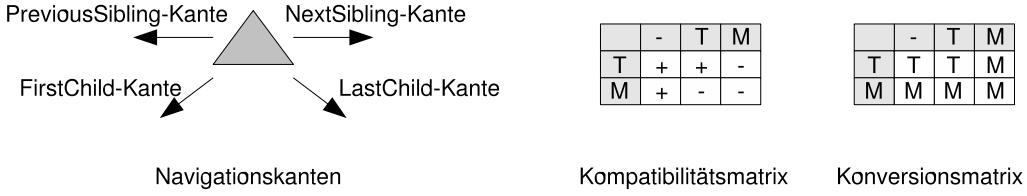
Das Node2PL-Protokoll des vorherigen Abschnitts verhält sich durch die Anforderung einer Sperre auf dem Elternknoten sehr restriktiv, da mit diesem Verfahren eine Modifikation stets die gesamte Ebene des Kontextknotens blockiert. NO2PL sperrt ebenfalls alle Knoten, über die der Kontextknoten erreichbar ist, allerdings sind davon durch eine feingranularere Strategie nur die tatsächlichen Nachbarknoten betroffen.

NO2PL benutzt dieselben Sperrmodi wie Node2PL, daher gelten auch für dieses Protokoll die Kompatibilitäts- und Konversionsmatrizen aus Abbildung 65. Während der Navigation muss eine Transaktion für jeden besuchten Kontextknoten eine T-Sperre anfordern. Zur strukturellen Modifikation, d. h. für das Einfügen oder Löschen von Knoten, werden alle Nachbarknoten, von denen der Kontextknoten aus erreichbar ist, mit M-Sperren versehen. Die Nachbarknoten sind gewöhnlich der vorherige und nächste Geschwisterknoten. Handelt es sich beim Kontextknoten um das erste bzw. letzte Kind seines Elternknotens, so ist der Elternknoten ebenfalls zu sperren, da ein Aufruf der *getFirstChild()*- bzw. *getLastChild()*-Methode auf dem Elternknoten den Kontextknoten liefert. Die Kinder des Kontextknotens, die den Kontextknoten selbst über die Bestimmung ihres Elternknotens erreichen könnten, müssen nicht gesperrt werden, da alle Protokolle aus [HKM01] davon ausgehen, dass sämtliche Navigationsoperationen an der Dokumentwurzel beginnen und somit die Ermittlung eines Elternknotens dessen vorherigen Besuch voraussetzt.

Das Auslesen oder Ändern von Knotenwerten sowie der Einsprung auf Elemente mit ID-Attributen wird in NO2PL genau wie in Node2PL mit Hilfe der S-, X-, IDR- und IDX-Sperre synchronisiert. Daher ist beim Löschen eines Knotens mit NO2PL ebenfalls der gesamte Teilbaum unterhalb des Knotens zu durchsuchen und für jedes Element mit einem ID-Attribut eine IDX-Sperre anzufordern.

OO2PL

OO2PL versucht die Parallelität der Verarbeitung gegenüber NO2PL weiter zu erhöhen, indem eine Transaktion nicht die benachbarten Knoten, sondern nur virtuelle Navigationskanten zwischen den benachbarten Knoten und dem Kontextknoten sperrt. Somit können andere Transaktionen auf die Nachbarknoten zugreifen, lediglich der Versuch, über die Navigationskante zum Kontextknoten zu gelangen, wird blockiert.



	-	T	M
T	+	+	-
M	+	-	-

	-	T	M
T	T	T	M
M	M	M	M

Abbildung 66: Kompatibilitäts- und Konversionsmatrix von OO2PL

Abbildung 66 zeigt die Navigationskanten eines XML-Knotens und die Kompatibilitäts- und Konversionsmatrix des OO2PL-Protokolls. Jeder Knoten erlaubt die Navigation zu dessen vorherigem und nächstem Geschwisterknoten und zu dessen erstem und letztem Kind. Auf diesen Kanten kann eine T- oder M-Sperre zum Traversieren oder Modifizieren angefordert werden. Wie auch die Protokolle Node2PL und NO2PL setzt OO2PL voraus, dass eine Transaktion die Navigation (bis auf ID-Attribut-Einsprünge) stets von der Dokumentenwurzel beginnt, was jedoch für eine konkrete Implementierung nicht besonders realistisch ist. Daher müssen in diesen

Protokollen für die rückwärtige Navigation zum Elternknoten keine Sperren angefordert werden.

Lesen und Ändern von Knotenwerten und direkte Einsprünge in das Dokument über ID-Attributwerte werden wie bei den beiden Vorgängerprotokollen mit S-, X-, IDR- und IDX-Sperren koordiniert und daher nicht erneut beschrieben.

Da Transaktionen mit allen drei Protokollen Node2PL, NO2PL und OO2PL rein navigationsbasiert synchronisiert werden und nur eine Sonderbehandlung für direkte Einsprünge auf ID-Attributwerte erfolgt, lösen diese Verfahren nicht vollständig die Phantomproblematik. Bereiche eines XML-Dokuments, die mit Navigationsoperationen erschlossen wurden, sind vor dem Einfügen weiterer Knoten geschützt. Führt aber bspw. eine Transaktion die *getElementById()*-Methode für einen gegebenen ID-Attributwert aus und erhält als Resultat, dass ein solcher Wert nicht existiert, verhindern die Protokolle nicht, dass während der Transaktionslaufzeit eine weitere Transaktion nachträglich ein Element mit diesem ID-Attributwert einfügt. Die erste Transaktion bekäme somit bei einer erneuten Anfrage ein anderes Ergebnis präsentiert.

3.3.3 Pfadbasierte Sperren

Dekeyser und Hidders stellen in [DH02] einen pfadbasierten Ansatz zur Isolation von Operationen auf XML-Dokumenten vor, der in [DHP04] zu einem vollständigen Transaktionsmodell erweitert wird.

Als Basis des Transaktionsmodells wird ein XML-Dokument mit einem *Xp-Baum* repräsentiert, der als ein Baum definiert ist, dessen Knoten mit ungerichteten Kanten verbunden sind. Ergänzend wird eine Abbildung eingeführt, mit der jedem Knoten ein Name zugewiesen wird. Auf dem Xp-Baum wird nur eine einfache Pfadanfragesprache definiert, die lediglich die *Descendant*- (//), *Child*- (/) und *Self*-Achse (*) unterstützt. Attribute oder die Formulierung von Prädikaten werden nicht erwähnt. Zur Modifikation eines als Xp-Baum dargestellten XML-Dokuments wird eine Operation zum Hinzufügen und eine Operation zum Löschen von Knoten und den entsprechenden Kanten angeboten. Die Aktualisierung eines Knotenwerts muss mit dem Entfernen und Erzeugen eines neuen Textknotens erfolgen.

Alle Transaktionen des Modells werden durch eine Folge von Pfadanfragen, Hinzufüge- und Löschoperationen und ein abschließendes Commit beschrieben. Dabei werden zur Isolation der Transaktionen Lese- und Schreibsperren verwaltet, die jeweils auf einem Kontextknoten für einen Pfadausdruck anzufordern sind. Eine Lesesperre zum Zugriff auf alle *Kunde*-Elemente des Beispieldokuments hätte bspw. die Form *readLock(T_1,Bank-Knoten,/Kunden/Kunde)*. Für die Sperranforderung selbst und die damit verbundene Überprüfung der Kompatibilität einer Anforderung mit den bereits gewährten Sperren, werden zwei Mechanismen vorgeschlagen, die im Folgenden kurz beschrieben werden.

Path Lock Propagation Scheme

Beim *Path Lock Propagation Scheme* wird eine Sperranforderung auf alle relevanten Kinder des Kontextknotens mit Anpassung des Sperrpfads propagiert. Abbildung 67 zeigt das Resultat der oben exemplarisch beschriebenen Sperranforderung auf dem Beispieldokument.

readLock(T_1,Bank-Knoten,/Kunden/Kunde)
readLock$_1$(T_1,Kunden-Knoten,/Kunde)
readLock$_2$(T_1,Kunde-Knoten1,)*
readLock$_3$(T_1,Kunde-Knoten2,)*

Abbildung 67: Propagierung von Pfadsperren

Die Propagierung von Pfadsperren erlaubt die Kompatibilitätsprüfung direkt auf den Sperren, die für einen Kontextknoten verwaltet werden. Allerdings wird dies damit erkauft, dass Sperren u. U. auf mehrere 100.000 nachfolgende Knoten propagiert werden müssen. Zusätzlich ist eine Überprüfung und evtl. nachträgliche Ergänzung von Sperren bei Modifikationen des Dokuments erforderlich, sodass dieses Protokoll eher theoretischer Natur ist.

Path Lock Satisfiability Scheme

Das *Path Lock Satisfiability Scheme* verwaltet im Gegensatz zum *Propagation Scheme* Sperren nur auf den Kontextknoten, auf denen sie angefordert wurden. Dies hat zur Folge, dass wesentlich weniger Sperren verwaltet werden. Die Kompatibilitätsprüfung für die Sperranforderung auf einem Kontextknoten muss auf jedem Vorfahren bis zur Dokumentwurzel durchgeführt werden. Da jedoch nur ein sehr geringer Sprachumfang zur Anfrage und Modifikation von Xp-Bäumen bzw. XML-Dokumenten berücksichtigt wurde, eignet sich auch dieses Protokoll nicht zur Synchronisation der De-facto-XML-Schnittstellen.

3.3.4 XMLTM

XMLTM [GBS02] beschreibt einen Transaktionsmanager für die Verarbeitung von XML-Dokumenten im Datenbanksystem DB2 der IBM. Die XML-Daten werden dabei mit Hilfe des *XML Extender* [DB2X] verwaltet, der auch die Auswertung proprietärer Pfadanfragen (ähnlich XPath) übernimmt. Um die Parallelität der Verarbeitung durch unnötige Blockierungen zu erhöhen, laufen die Transaktionen im Datenbanksystem in einer niedrigen Isolationsstufe und die Transaktionsisolation zur Gewährleistung der Stufe *serializable* sowie die Koordination der Anwendungen wird von XMLTM übernommen.

Zur Synchronisation wird für jedes XML-Dokument ein DataGuide aufgebaut, auf dem die Sperranforderung gemäß einem klassischen hierarchischen Protokoll [GR93] mit den Sperrmodi IS, IX, S, SIX und X erfolgt. Abbildung 68 zeigt das Beispieldokument aus Abbildung 1 und den DataGuide mit der resultierenden Sperrsituation nach eine Anfrage für den Pfad */Bank/Kunden/Kunde*.

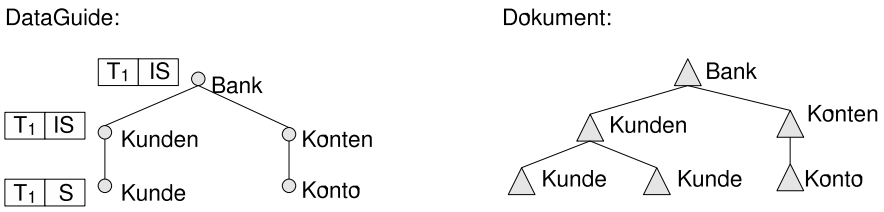


Abbildung 68: Sperrverwaltung mit einem DataGuide in XMLTM

Die Autoren beschreiben, dass die Sperren zur Verfeinerung der Granulate noch mit zusätzlichen Prädikaten annotiert werden können, sodass eine Sperre in diesem Fall erst gewährt wird, wenn die Sperrmodi kompatibel sind und keine Überschneidung der Prädikate vorliegt. Die dazu in [GBS02] angeführte Beschreibung ignoriert leider völlig die Problematik, dass die Überlappung von Prädikaten im Allgemeinen nur auf Instanzebene überprüft werden kann. Beispielsweise müssten in unserem Dokument die Sperren *S* auf *//Kunde* mit Prädikat */Name="Haustein"* und *X* auf *//Kunde* mit Prädikat */@id="kd1606"* inkompatibel sein, da es sich um dasselbe Fragment im XML-Dokument handelt. Dies kann jedoch auf Ebene der Prädikatdefinitionen nicht entschieden werden, womit diese vorgeschlagene Erweiterung vollkommen unbrauchbar ist.

3.3.5 Synchronisation mit Read- und Write-Sets

Das recht triviale Synchronisationsprotokoll in [BT01] wird in einer Web-Anwendung des E-Learning-Umfelds eingesetzt und basiert auf einem Check-in/Check-out-Mechanismus. Mehrere Anwendungen greifen auf zentral gespeicherte Daten im XML-Format zu. Die Autoren nennen Beispiele wie Schulungsunterlagen, Kursanmeldungen oder Online-Prüfungen. Dass die eigentlichen Daten in einem relationalen Datenbanksystem gespeichert sind und für den Datenaustausch während des Check-out-Vorgangs in einen XML-Dialekt abgebildet werden, ist für das Protokoll unerheblich.

Die Operationen, die auf Anwendungsseite auf den XML-Daten durchgeführt werden können, setzen sich aus *Browse* (Lesen ohne das Protokollieren der Leseoperation), *Lesen*, *Ändern*, *Einfügen* und *Löschen* von Knoten zusammen. Für jede Transaktion wird in der Anwendung ein *Read-Set* bzw. *Write-Set* geführt, in dem die ID des Knotens und der gelesene bzw. neu geschriebene Wert protokolliert werden. Wie die Knoten-ID aufgebaut ist, wird nicht erwähnt.

Beim Check-in-Prozess werden die XML-Daten zurück in ihre relationale Repräsentation transformiert und dabei die *Read-Sets* überprüft. Stimmen alle während der Check-out-Phase gelesenen Daten mit den Werten der *Read-Sets* überein, so werden alle Änderungen der *Write-Sets* in die Datenbank propagiert. Wurde ein Wert, den die Anwendung gelesen hat (und der sich daher im *Read-Set* befindet) zwischenzeitlich geändert, so wird der Check-in-Vorgang abgebrochen und das weitere Vorgehen an die Anwendung delegiert. Diese hat nun die Möglichkeit, ihr *Read-Set* zu aktualisieren und ein erneutes Check-in zu starten oder die gesamte Transaktion abzubrechen.

3.4 Zusammenfassung

In diesem Kapitel wurden verwandte Arbeiten der XML-Datenverarbeitung bzgl. der Themenschwerpunkte dieser Arbeit diskutiert. Abschnitt 3.1 untersuchte zunächst die Funktionalität, die Speicherungsstrukturen und die eingesetzten Techniken zur Transaktionsverwaltung existierender nativer XML-Datenbanksysteme. Abbildung 69 stellt die Ansätze nochmals tabellarisch gegenüber.

	Speicherungsstrategie	Anfragesprachen und Schnittstellen	Transaktionsisolation	ACID
eXist	elementbasiert	SAX,DOM über XML:DB-API XQuery + Lehti-Erweiterungen SOAP, WebDAV	-	nein
Sedna	elementbasiert	XQuery + Lehti-Erweiterungen	RX-Protokoll auf Knotenebene	ja
OrientX	element- und teilbaumbasiert	SAX, DOM und XQuery über proprietäre Schnittstelle	-	nein
SystemRX	teilbaumbasiert	SQL/XML und XQuery	Versionierung	ja
Natix	teilbaumbasiert	SAX, DOM, XQuery, WebDAV, Dateisystemtreiber	hierarchisch + TOUCH-Sperre	ja
Xindice	dokumentenbasiert	SAX, DOM und XPath über XML:DB-API, XML-RPC	-	nein
Tamino	dokumentenbasiert	SAX, DOM und XPath über XML:DB-API, HTTP, XQuery + proprietäre Update-Erweiterung	RX-Protokoll auf Dokumentenebene	ja

Abbildung 69: Vergleich nativer XML-Datenbanksysteme

Abschnitt 3.2 gibt einen Überblick über Nummerierungsschemata zur Adressierung von Knoten innerhalb eines XML-Dokuments. Dabei wurden bei den jeweiligen Ansätzen Anforderungen für die Implementierung in einem nativen XML-Datenbanksystem untersucht. Dies sind im Besonderen die Erhaltung der Dokumentenordnung durch die zugewiesenen Knoten-IDs, die Stabilität gegen Änderungsoperationen, die Berechnung aller Vorfahren für ein hierarchisches Sperrprotokoll und die Unterstützung der Auswertung von Pfadachsen zur Anfrageverarbeitung. Abbildung 70 vergleicht die vorgestellten Ansätze miteinander. Wie zu erkennen ist, unterstützen nur die Nummerierungsschemata ORDPATHs und DLN alle Anforderungen.

	k-Wege Baum	PBiTree	Prim Bott.-up	Prim Top-dn	Global	Lokal	Dewey	ORD PATHs	DLN	Sedna	µPID	Dietz, Moon	BIRD	BOX	L-Tree
Stabil gegen bel. Einfügen	nein	nein	nein	ja*2	nein	nein	nein	ja	ja	ja	nein	nein	nein	ja	nein
Stabil gegen bel. Löschen	ja*1	ja	ja	ja	ja	ja	ja	ja	ja	ja	ja	ja	ja	ja	ja
Berechnung aller Vorfahren	ja	nein	nein	ja*3	nein	nein	ja	ja	ja	nein	ja	nein	ja*4	nein	nein
Speicherung in Dok.ordnung	nein	nein	nein	nein	ja	nein	ja	ja	ja	ja	ja*4	ja	ja	ja	nein
Auswertung der Anc./Desc.-Achsen	ja	ja	nein	ja	nein	nein	ja	ja	ja	ja	ja*4	ja	ja	nein	ja
Auswertung der Prec./Foll.-Achsen	ja	nein	nein	ja	nein	nein	ja	ja	ja	nein	ja*4	nein	ja	nein	nein
	*1 beim Zulassen von Lücken, *2 Kongruenzgleichungen lösen, *3 mit Primfaktorzerlegung, *4 mit DataGuide														

Abbildung 70: Vergleich der Nummerierungsschemata

Der letzte Abschnitt in diesem Kapitel behandelte bereits publizierte Verfahren zur Isolation nebenläufiger Transaktionen auf XML-Dokumenten. Abbildung 71 zeigt eine Gegenüberstellung der Ansätze bzgl. der Unterstützung verschiedener Operationen. Keines der Verfahren ist in der Lage, Transaktionen vollständig gegeneinander zu isolieren, die in einem einzigen Verarbeitungskontext alle typischen XML-Schnittstellen gemeinsam nutzen.

	Sedna RX	Doc2PL	Node2PL	NO2PL	OO2PL	Pfadsperren	XMLTM	Read-/Write-Sets
Navigation	nein	ja*2	ja	ja	ja	nein	nein	ja*4
direkte Einsprünge auf ID-Attribute	ja	ja*2	ja	ja		nein	nein	ja*4
beliebige direkte Einsprünge	ja	ja*2	nein	nein	nein	ja*3	nein	ja*4
Pfadanfragen	nein	ja*2	nein	nein	nein	ja	ja	nein
Phantom-verhinderung	ja*1	ja*2	nein	nein	nein	ja	ja	nein
	*1 durch gröberes Sperrgranulat, *2 Dokumentensperre, *3 alle Knoten mit gleichem Pfad wie der Kontextknoten werden gesperrt, *4 asynchron im Anwendungspuffer							

Abbildung 71: Vergleich der Ansätze zur Transaktionsisolation

KAPITEL 4 taDOM-Transaktionsmodell

Jede theoretische Erklärung ist
eine Reduzierung der Intuition.
(Peter Hoeg)

Wie wir in Kapitel 3 gesehen haben, gibt es zur Zeit kein Konzept, das in einem nativen XML-Datenbanksystem Änderungen an XML-Dokumenten auf Knotenebene ermöglicht und dabei nebenläufige Transaktionen auf diesem Granulat gegeneinander isoliert, wenn sie alle typischen XML-Schnittstellen gemeinsam in einem Verarbeitungskontext nutzen. Zur Realisierung eines solchen Systems gemäß dieser Anforderungen benötigt man zunächst ein klar definiertes Grundgerüst, auf dem die gesamte Datenhaltung und Anfrageverarbeitung basiert. Zusätzlich muss für eine standardisierte Anbindung von Anwendungen eine Abbildung der XML-Schnittstellen auf das Grundgerüst möglich sein. Dazu beginnen wir in diesem Kapitel mit der Definition des *taDOM-Transaktionsmodells*. Abschnitt 4.1 führt zunächst das *taDOM-Datenmodell* ein, mit dem ein XML-Dokument in einem Datenbanksystem als so genannter *taDOM-Baum* repräsentiert wird. Auf diesem Datenmodell definiert Abschnitt 4.2 eine Reihe von Basisoperationen, über die auf einen taDOM-Baum zugegriffen und dessen Inhalt und Struktur modifiziert werden kann. Abschnitt 4.3 beschreibt schließlich mit dem Konzept der *DeweyIDs* eine Knotenadressierung für taDOM-Bäume, die stabil gegen jede Art von Modifikationsoperationen ist. Dieses Nummerierungsschema wird benötigt, um auszudrücken, auf welchem Knoten eines taDOM-Baums eine Operation ausgeführt wird und welche Knoten in der Ergebnismenge einer Operation liegen.

Mit dem so festgelegten Datenmodell, den darauf definierten Operationen und einer Adressierung kann das Transaktionsmodell mit weiteren Techniken angereichert werden, um alle ACID-Eigenschaften [HR99] für die Datenverarbeitung zu garantieren. Wie wir in Kapitel 6 noch sehen werden, können alle Operationen des taDOM-Datenmodells auf Operationen in B*-Bäumen, die in Datenseiten fester Länge verwaltet werden, abgebildet werden. Diese B*-Baum-Operationen gehören zur Grundfunktionalität jedes relationalen Datenbanksystems [Co79, HR83]. Daher können Modifikationen an einem taDOM-Baum mit den von dort bekannten und zuverlässigen Techniken atomar und dauerhaft in eine Datenbank eingebracht werden. Die Konsistenz eines taDOM-Baums, bzw. die Gültigkeit des durch ihn repräsentierten XML-Dokuments bzgl. eines Schemas, ist nicht Thema dieser Arbeit und wird im Ausblick in Kapitel 8 nochmals aufgegriffen. Zur gegenseitigen Isolation von Transaktionen, die parallel auf taDOM-Bäumen operieren, werden im folgenden Kapitel 5 verschiedene Sperrprotokolle vorgestellt, die die Serialisierbarkeit der Transaktionen gewährleisten. Zusammenfassend wird somit ein Transaktionsmodell definiert, das – unter Voraussetzung einer Lösung zur Konsistenzsicherung – die ACID-Eigenschaften bei der nativen XML-Datenverarbeitung garantiert. Abbildung 72 zeigt nochmals eine Übersicht dieser Zusammenhänge.

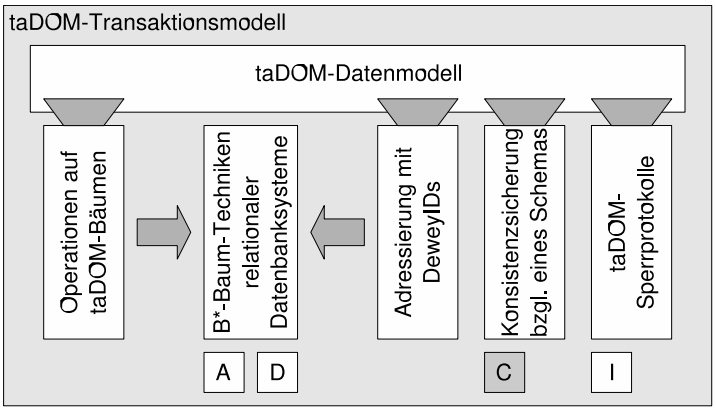


Abbildung 72: Das taDOM-Transaktionsmodell

4.1 taDOM-Datenmodell

Das *taDOM-Datenmodell* [HH03] erweitert das Document Object Model um zwei neue Knotentypen zur besseren Unterstützung der Transaktionsisolation. Für die Beschreibung des Modells beschränken wir uns auf XML-Dokumente, die ausschließlich aus Element-, Attribut- und Textknoten bestehen. Eine Einschränkung auf diese Typen ist durchaus zulässig, da alle zu lösenden Probleme bzgl. der Dokumentenverwaltung und Nebenläufigkeitskontrolle damit diskutiert werden können. Die übrigen Knotentypen der DOM-Spezifikation können problemlos in das taDOM-Datenmodell aufgenommen werden, da sie lediglich Spezialisierungen der bereits berücksichtigten Typen sind bzw. eine sehr einfache Struktur und keine weiteren Kindknoten besitzen. Die ID/IDREF(S)-Beziehungen werden in der Literatur aufgrund einer logisch entstehenden Graphstruktur (siehe Abbildung 7 auf Seite 10) oft als problematisch dargestellt und daher in Abschnitt 4.1.2 gesondert behandelt.

4.1.1 taDOM-Bäume

taDOM-Bäume haben ihren Namen vom Document Object Model erhalten, das sie mit zwei neuen Knotentypen (der *Attributwurzel* und dem *String-Knoten*) und *virtuellen Navigationskanten* für die feingranulare Transaktionsisolation erweitern: *transactional DOM – taDOM*. Eine Repräsentation des Beispieldokuments aus Abbildung 1 als taDOM-Baum ist in Abbildung 73 dargestellt.

Elemente des XML-Dokuments werden auf Elementknoten des taDOM-Baums abgebildet. Jeder Elementknoten erhält zusätzlich eine virtuelle *prevSibling*-, *nextSibling*-, *firstChild*- und *lastChild*-Navigationskante zu dessen Geschwister- und erstem und letztem Kindknoten. Diese Kanten werden als *virtuell* bezeichnet, da sie nicht als persistente Objekte verwaltet werden und keine Operation sie als Ergebnis zurückliefern kann. Sie spielen allerdings bei der Transaktionsisolation (Kapitel 5) eine wichtige Rolle. Im taDOM-Baum werden alle Knoten über *einfache Kanten* mit ihren Elternknoten verbunden. Einfache Kanten haben bzgl. der taDOM-Operationen keine Bedeutung und dienen nur der Zuordnung von Eltern- zu Kindknoten.

Attribute werden im Gegensatz zu DOM nicht direkt mit dem besitzenden Element verbunden, sondern als Kinder einer *Attributwurzel* verwaltet. Erst die Attributwurzel besitzt eine Kante zum Elementknoten. Dies hat den Vorteil, dass unter Einsatz eines hierarchischen Sperrprotokolls nach wie vor jedes Attribut separat gesperrt werden kann, aber beim Zugriff auf alle Attribute eines Elements nur eine einzige Sperre auf der Attributwurzel für die Transaktionsisolation benötigt wird. Da Attribute keine Kindknoten besitzen können und für Attribute keine Reihenfolge definiert ist, erhalten sie keine virtuellen Navigationskanten.

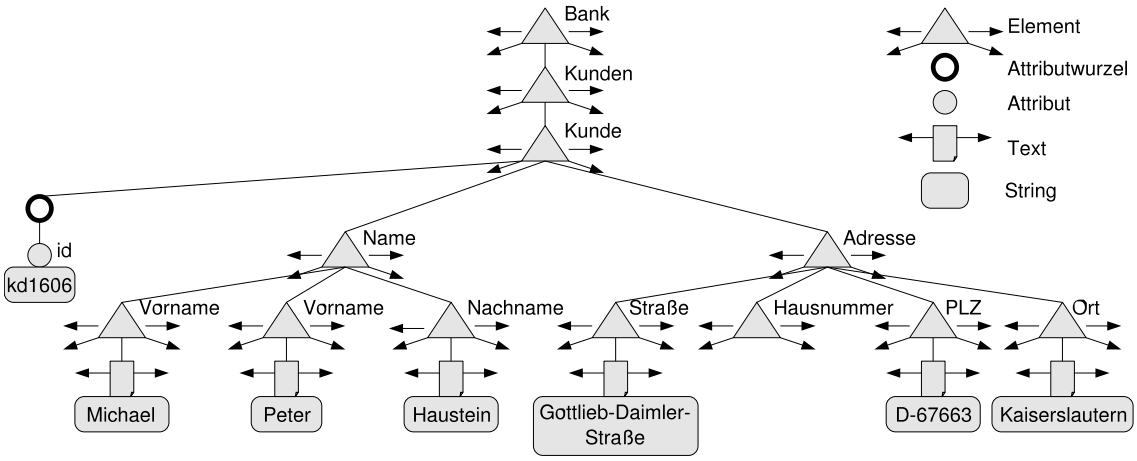


Abbildung 73: taDOM-Repräsentation des Beispieldokuments

Textknoten enthalten nicht direkt die von ihnen repräsentierten Werte, sondern übernehmen nur eine Stellvertreterrolle. Da sie nur Geschwister- und keine Kindknoten besitzen können, werden Textknoten nur mit einer virtuellen *prevSibling*- und *nextSibling*-Kante ausgestattet. Der Vorteil einer Trennung des Textknotens von seinem eigentlichen Wert wird bei der Transaktionsisolation bei Elementen mit gemischtem Inhaltsmodell deutlich. Navigiert eine Transaktion über alle Kinder eines solchen Elements auf der Suche nach einem speziellen Kindelementknoten, so werden bei dieser Operation nur die taDOM-Textknoten vom Sperrprotokoll erfasst. Die für die Transaktion uninteressanten Werte der Textknoten können somit von anderen parallel laufenden Transaktionen sogar modifiziert werden.

String-Knoten nehmen im taDOM-Baum die Werte von Attribut- und Textknoten auf. Sie werden direkt ohne eine Kante an die betroffenen Knoten geheftet. Da Attribut- und Textknoten keine Kinder besitzen und deren Werte nicht auf mehrere String-Knoten aufgeteilt werden können, erhalten String-Knoten keine virtuellen Navigationskanten.

Die virtuellen *prevSibling*- und *nextSibling*-Kanten verbinden Geschwisterknoten miteinander. Wenn eine Transaktion von einem Knoten k_1 zu dessen nächstem Geschwisterknoten k_2 navigiert, so folgt sie dabei der *nextSibling*-Kante von k_1. Greift sie schließlich auf den Knoten k_2 zu, so hat sie damit auch Kenntnis über dessen *prevSibling*-Kante, die auf k_1 verweist. Wie wir in Kapitel 5 noch sehen werden, erfordert die Navigation in Verbindung mit einem Sperrprotokoll, dass zunächst eine Lesesperre entlang einer Navigationskante angefordert werden muss. Verweist eine *prevSibling*- bzw. *nextSibling*-Kante auf einen *null*-Wert, so impliziert dies, dass es sich um das erste bzw. letzte Kind unter den Geschwistern handelt. Analog weisen eine *null*-wertige *firstChild*- oder *lastChild*-Kante bzw. eine gleichwertige *firstChild*- oder *lastChild*-Kante darauf hin, dass ein Knoten keine Kinder bzw. nur ein Kind besitzt.

4.1.2 ID/IDREF(S)-Beziehungen

Mit der ID/IDREF(S)-Beziehung (Abschnitt 2.2.1) können Verweise innerhalb eines XML-Dokuments realisiert werden, deren Integrität durch die Konsistenzsicherungskomponente eines Datenbanksystems sicherzustellen ist. Durch diese Verweise entstehen (analog zur KEY/KEYREF-Beziehung in Abschnitt 2.2.2) Graphstrukturen wie in Abbildung 7 auf Seite 10. Diese Graphen sind allerdings rein logischer Natur, d. h., die Kante von der Verweisquelle (IDREF) zum Verweisziel (ID) ist nur durch die Gleichheit der Attributwerte definiert. Die textuelle Darstellung des XML-Dokuments repräsentiert nach wie vor einen Baum.

Für die ID/IDREF(S)-Beziehung muss keine Sonderbehandlung für die Speicherung, die Verfolgung der Verweise auf taDOM-Knoten oder die Transaktionsisolation erfolgen. Das Dokument selbst wird nach wie vor in seiner Baumrepräsentation gespeichert, die Konsistenzsicherung des XML-Datenbanksystems muss nur vor der persistenten Sicherung die referentielle Integrität garantieren. Zur Verfolgung eines Verweises erfolgt ein direkter Einsprung auf das Element mit dem entsprechenden ID-Attribut, was in einer Systemimplementierung gewöhnlich über einen sekundären Index abgewickelt wird.

Das Sperrprotokoll muss hierzu lediglich einen direkten Einsprung in das XML-Dokument unterstützen. Dass ein ID-Attribut, welches Ziel einer Referenz ist, nicht gelöscht, umbenannt oder im Wert verändert wird, ist nicht in den Basisoperationen des Datenmodells oder vom Sperrprotokoll zu überwachen. Dies ist Aufgabe der Konsistenzsicherung, die auch evtl. durch Einstellungen des Anwenders erst am Transaktionsende erfolgt (entspricht dem *set constraints <constraints_name_list> immediate/deferred* in SQL). Die Zusammenhänge zwischen Datenmodell, Transaktionsisolation und Konsistenzsicherung werden noch deutlicher, wenn man sie auf das Relationenmodell überträgt. Der Verweis in einem XML-Dokument mit einer ID/IDREF-Beziehung entspricht der Definition eines *Primary-key*- oder *Unique*-Attributs und dem Verweis darauf mit einer *Foreign-key*-Referenz. Abbildung 74 zeigt dies mit je einer Tabelle für *Kunden* und *Konten*, wobei das Attribut *besitzer* der Tabelle *Konten* auf den Primärschlüssel der Tabelle *Kunden* verweist.

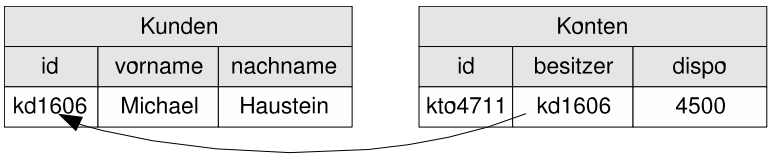


Abbildung 74: Referentielle Integrität im Relationenmodell

Verschiebt man die Überprüfung der referentiellen Integrität ans Transaktionsende, kann man in der Tabelle *Kunden* das Tupel *(kd1606,Michael,Haustein)* ohne Fehlermeldung löschen oder den Primärschlüssel ändern. Diese Operation führt nur zu Sperren auf der Tabelle *Kunden* und den evtl. eingerichteten Indexen. Auf der Tabelle *Konten* kann in diesem Fall nach wie vor gelesen werden. Nur der Zugriff auf die Tabelle *Kunden*, bspw. für eine Verbundoperation, wird blockiert, da der Verweis von *Konten.besitzer* zu *Kunden.id* allein durch die Wertegleichheit der Attribute definiert ist.

4.2 Operationen auf taDOM-Bäumen

Für die Ermittlung der Knotenbeziehungen in einem taDOM-Baum und zur Modifikation der Baumstruktur und Knotenwerte werden für das taDOM-Datenmodell 19 Basisoperationen definiert. Für diese Operationen erfolgt die Adressierung der übergebenen und zurückgelieferten XML-Knoten mit DeweyIDs, die im nächsten Abschnitt vorgestellt werden.

Die Basisoperationen lassen sich in drei Klassen zur *Navigation*, zum *Auslesen und Ändern von Knotenwerten* und zur *strukturellen Modifikation* eines taDOM-Baums unterteilen. Abbildung 75 zeigt die Klassifizierung der Basisoperationen; danach wird deren Semantik detailliert beschrieben.

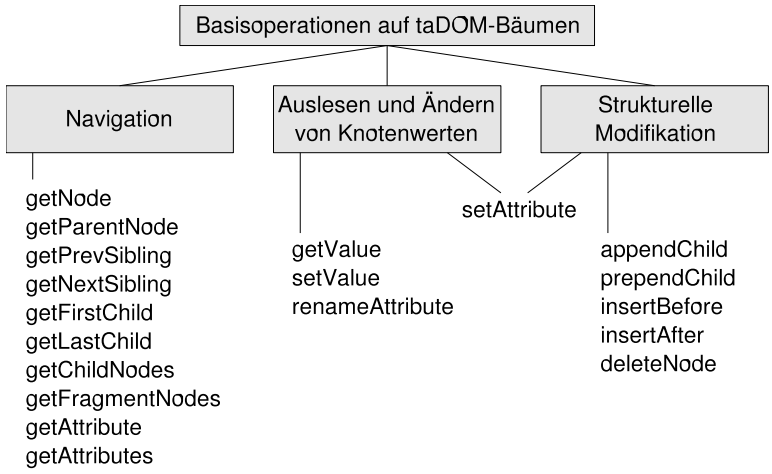


Abbildung 75: Klassifizierung der taDOM-Basisoperationen

Basisoperationen zur Navigation

- *getNode (deweyID)*
  Liefert mit einem direkten Einsprung in das Dokument den Knoten mit der angegebenen DeweyID oder *null*, wenn ein Knoten mit dieser Adresse nicht existiert. Diese Operation ist für jeden Knotentyp definiert.
- *getParentNode (node)*
  Liefert den Elternknoten des übergebenen Kontextknotens oder *null*, wenn der Kontextknoten die Wurzel eines XML-Dokuments repräsentiert. Diese Operation ist nur für Element- und Textknoten definiert.
- *getPrevSibling (node)*
  Liefert den vorherigen Geschwisterknoten des übergebenen Kontextknotens oder *null*, wenn der Kontextknoten das erste Kind seines Elternknotens ist. Diese Operation ist nur für Element- und Textknoten definiert.
- *getNextSibling (node)*
  Liefert den nächsten Geschwisterknoten des übergebenen Kontextknotens oder *null*, wenn der Kontextknoten das letzte Kind seines Elternknotens ist. Diese Operation ist nur für Element- und Textknoten definiert.
- *getFirstChild (node)*
  Liefert den ersten Kindknoten des übergebenen Kontextknotens oder *null*, wenn der Kontextknoten keine Kinder besitzt. Im letzteren Fall liefert auch die *getLastChild()*-Methode einen *null*-Wert. Diese Operation ist nur für Elementknoten definiert.
- *getLastChild (node)*
  Liefert den letzten Kindknoten des übergebenen Kontextknotens oder *null*, wenn der Kontextknoten keine Kinder besitzt. Diese Operation ist nur für Elementknoten definiert.
- *getChildNodes (node)*
  Liefert eine Liste aller Kinder des übergebenen Kontextknotens. Besitzt der Kontextknoten keine Kinder, so wird eine leere Liste zurückgeliefert. Diese Operation ist nur für Elementknoten definiert.

- *getFragmentNodes (node)*
  Liefert in einer Liste den Kontextknoten und alle seine Nachfahren in Dokumentenordnung. Es wird eine Liste erstellt, die den Kontextknoten und alle Knoten des Teilbaums, der durch den Kontextknoten als Wurzel definiert ist, enthält. Besitzt der Kontextknoten keine Nachfahren, so liefert diese Methode eine Liste, die nur den Kontextknoten selbst enthält. Diese Operation ist für alle Knotentypen definiert.
- *getAttribute (node,attributeName)*
  Liefert für den übergebenen Kontextknoten den Knoten des Attributs, mit dem angegebenen Namen oder *null*, wenn der Kontextknoten kein Attribut mit diesem Namen besitzt. Diese Operation ist nur Elementknoten definiert.
- *getAttributes (node)*
  Liefert für den übergebenen Kontextknoten eine Liste aller seiner Attributknoten. Falls der Kontextknoten keine Attribute besitzt, wird eine leere Liste zurückgeliefert. Diese Operation ist nur für Elementknoten definiert.

Basisoperationen zum Auslesen und Ändern von Knotenwerten

- *setAttribute (node,attributeName,attributeValue)*
  Setzt bei dem übergebenen Kontextknoten für das Attribut mit dem angegebenen Namen den spezifizierten Wert. Existiert ein solches Attribut noch nicht, so wird ein Attributknoten mit einem angehefteten String-Knoten für den entsprechenden Wert neu angelegt. Daher bewirkt die Operation *setAttribute* u. U. auch eine Veränderung der Baumstruktur. Diese Operation ist nur für Elementknoten definiert.
- *getValue (node)* bzw. *setValue (node,value)*
  Liefert bzw. setzt den Wert des übergebenen Kontextknotens. Für Element- bzw. String-Knoten betrifft dies den Namen des Elements bzw. den Wert des String-Knotens, für Attribut- und Textknoten den Wert des angehefteten String-Knotens. Für Attributwurzelknoten ist diese Operation nicht definiert.
- *renameAttribute (node,attributeName,newAttributeName)*
  Benennt für den übergebenen Kontextknoten den Namen des angegebenen Attributs in den spezifizierten neuen Attributnamen um, ohne auf den Wert des Attributs zuzugreifen. Falls der Kontextknoten kein Attribut mit dem angegebenen Namen besitzt, so wird eine Fehlermeldung erzeugt. Diese Operation ist nur für Elementknoten definiert.

Basisoperationen zur strukturellen Modifikation

- *appendChild (node,newNode)* bzw. *prependChild (node,newNode)*
  Fügt den spezifizierten neuen Knoten als neuen letzten bzw. neuen ersten Kindknoten des übergebenen Kontextknotens ein. Diese Operation ist nur für Elementknoten definiert.
- *insertBefore (node,newNode) bzw. insertAfter (node,newNode)*
  Fügt den spezifizierten neuen Knoten vor bzw. hinter dem übergebenen Kontextknoten als neuen vorherigen bzw. nächsten Geschwisterknoten ein. Diese Operation ist nur für Element- und Textknoten definiert.
- *deleteNode (node)*
  Löscht den übergebenen Kontextknoten und alle seine Nachfahren. Damit wird der Kontextknoten und alle Knoten, die im durch ihn als Wurzel definierten Teilbaum liegen, aus dem Dokument entfernt. Diese Operation ist für alle Knotentypen definiert.

Zur Realisierung einer standardkonformen XML-Schnittstelle für DOM Level 3 [DOM3] mit Hilfe dieser Basisoperationen ist es erforderlich, dass die *setValue()*-Methode auch für Elementknoten zu deren Umbenennung definiert ist, da DOM Level 3 u. a. als Erweiterung zu Level 2 die Methode *renameNode()* für Element- und Attributknoten einführt.

4.3 Adressierung in taDOM-Bäumen mit der DeweyID

Die Adressierung aller Knoten eines taDOM-Baums und der darauf definierten Operationen im vorherigen Abschnitt basieren auf dem Konzept der *DeweyIDs* [HHM+05]. DeweyIDs adaptieren und erweitern das ORDPATHs-Nummerierungsschema (siehe Abschnitt 3.2.6) für das taDOM-Datenmodell.

Eine DeweyID beginnt zunächst mit einer eindeutigen Nummer, die das XML-Dokument der zu nummerierenden Knoten identifiziert. Danach trennt ein Doppelpunkt die Dokumentennummer von der Adressierung der eigentlichen Knoten. Die Knotenadressierung setzt sich aus einer Folge von *Divisions* zusammen, die durch einfache Punkte voneinander getrennt werden. Der Wert einer *Division* (oder kurz eine *Division*) ist eine natürliche Zahl, die die Position eines Knotens innerhalb einer Teilbaumebene beschreibt. Die Adresse eines Knotens wird aus der Adresse seines Elternknotens als Präfix und weiteren *Divisions* zur Positionierung zwischen seinen Geschwistern bestimmt. Der *Division*-Wert *1* ist dabei für die Dokumentwurzel, Attributwurzel- und String-Knoten reserviert. Analog zum ORDPATHs-Konzept werden Knoten beim initialen Speichern eines Dokuments nur mit ungeraden *Division*-Werten versehen. Abbildung 76 zeigt die initiale Nummerierung des taDOM-Baums aus Abbildung 73 mit DeweyIDs.

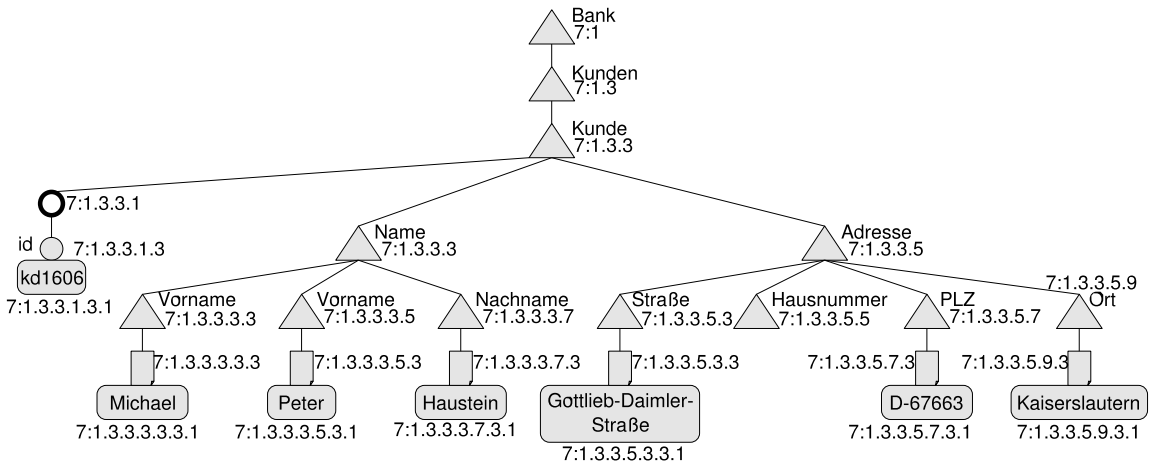


Abbildung 76: Initiale Nummerierung mit DeweyIDs

Wie bei den ORDPATHs können mit Hilfe gerader *Divisions* bei Modifikationen des Dokuments zwischen zwei DeweyIDs beliebig viele weitere IDs eingefügt werden. So haben zwischen der ID *7:1.3.3.3* und *7:1.3.3.5* die IDs *7:1.3.3.4.3* und *7:1.3.3.4.5* Platz. Dazwischen können wiederum die DeweyIDs *7:1.3.3.4.4.3* und *7:1.3.3.4.4.5* eingefügt werden. Vor der ID *7:1.3.3.3* des initial ersten Kindes muss die ID *7:1.3.3.2.3* eingefügt werden, da die *7:1.3.3.1* für den Attributwurzelknoten des Elternelements reserviert ist. Abbildung 77 zeigt einen Ausschnitt aus dem taDOM-Baum nach der Ausführung dieser Operationen.

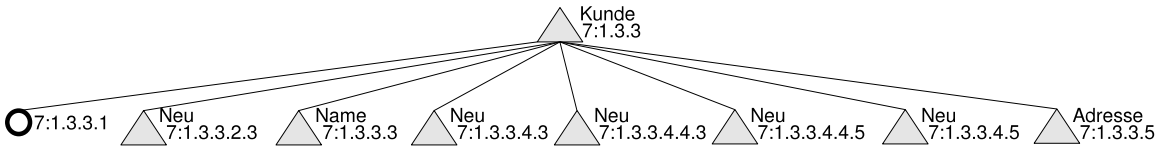


Abbildung 77: Einfügen von DeweyIDs

Durch die gezielte Vergabe der geraden und ungeraden *Division*-Werte kann auch bei nachträglichen Einfügeoperationen für jeden Kontextknoten die DeweyID des Elternknotens ohne Zugriff auf den taDOM-Baum berechnet werden: Die letzte (immer ungerade) *Division* des Kontextknotens wird abgetrennt, danach werden alle dann evtl. am Ende stehenden geraden *Division*-Werte entfernt. Die so ermittelte DeweyID adressiert den Elternknoten. Besteht die DeweyID nur aus der Dokumentnummer, einem Doppelpunkt und dem einzigen *Division*-Wert *1*, so handelt es sich um die Dokumentwurzel und ein Elternknoten existiert nicht.

Da die Länge der DeweyIDs sehr schnell zunehmen kann, führen wir im folgenden Abschnitt den *Distance*-Parameter ein, der dafür sorgt, dass bei der initialen Vergabe von IDs zusätzlich größere „Lücken“ zwischen zwei Adressen reserviert werden. Diese „Lücken“ können bei nachfolgenden Einfügeoperationen anfänglich für neue Knoten ausgenutzt werden, ohne längere *Division*-Folgen erzeugen zu müssen. Sind schließlich auch diese „Lücken“ belegt, so werden neue Knoten wie zuvor mit geraden *Division*-Werten adressiert.

4.3.1 Parametrisierung mit dem Distance-Wert

Der Parameter *Distance* bestimmt durch eine gerade natürliche Zahl eine Schrittweite, die die Vergabe neuer DeweyIDs beeinflusst [Wa05a], indem für neue Knoten nicht jeder ungerade *Division*-Wert belegt wird (Schrittweite *2*), sondern der durch *Distance* definierte Wert als Abstand eingehalten wird. Je größer der *Division*-Wert gewählt wird, desto mehr Adressen können zwischen zwei bereits vorhandenen Knoten vergeben werden, ohne dass die Anzahl der *Divisions* steigt. Wir werden jedoch bei den Implementierungsdetails in Kapitel 6 noch sehen, dass die durch einen größeren *Distance*-Wert schneller ansteigenden *Division*-Werte mehr Platz bei der Speicherung benötigen. Dieser Speicherplatzbedarf ist u. U. allerdings geringer als der für weitere *Divisions*. Daher werden in Kapitel 6 genauere Untersuchungen durchgeführt, wie der *Distance*-Wert gegen die erwartete Änderungshäufigkeit des Dokuments und den steigenden Speicherplatzbedarf einzelner *Division*-Werte bzw. dem zusätzlicher *Divisions* abzuwägen ist.

Mit dem Einsatz des *Distance*-Parameters ergeben sich somit zur initialen Adressierung eines taDOM-Baums die folgenden sechs Regeln:

- Der Wurzelknoten des Dokuments erhält als DeweyID die Dokumentnummer gefolgt von einem Doppelpunkt und dem einzigen *Division*-Wert *1*.
- Der erste Element- oder Textknoten einer Teilbaumebene bekommt die DeweyID des Elternelements erweitert um eine weitere *Division* mit dem Wert *Distance+1*.
- Ein neuer Knoten hinter dem letzten Knoten einer Teilbaumebene erhält die DeweyID dieses letzten Knotens, wobei sich der Wert der letzten *Division* um *Distance* erhöht.
- Ein Elementknoten, der mindestens ein Attribut besitzt, erhält eine Attributwurzel. Die Attributwurzel wird mit der DeweyID des Elementknotens adressiert, ergänzt um eine weitere *Division* mit dem Wert *1*.

- Attributknoten erhalten die DeweyID ihrer Attributwurzel erweitert um eine weitere *Division*. Da für Attribute keine Ordnung definiert ist, spielt der *Distance*-Wert hier keine Rolle. Beim ersten Attributknoten wird der Wert der letzten *Division* auf *3* gesetzt, jedes weitere Attribut erhält einen um *2* erhöhten *Division*-Wert.
- Im taDOM-Baum wird der Wert eines Attribut- oder Textknotens in angehefteten String-Knoten verwaltet. String-Knoten besitzen keine Geschwister- oder Kindknoten und erhalten die DeweyID des entsprechenden Attribut- bzw. Textknotens ergänzt um eine weitere *Division* mit dem Wert *1*.

4.3.2 Einfügen neuer Knoten

Das Einfügen oder Löschen von Knoten in einem taDOM-Baum erfordert keine Reorganisation der bereits vergebenen DeweyIDs. Beim Löschen können die betroffenen Knoten mitsamt ihren Nachfahren einfach aus dem Baum entfernt werden. Beim Einfügen neuer Knoten unterscheiden wir bei der Wahl ihrer DeweyID zwischen drei Fällen: Vergabe einer neuen DeweyID hinter dem letzten Knoten, vor dem ersten Knoten und zwischen zwei benachbarten Knoten einer Teilbaumebene.

Neue DeweyID hinter dem letzten Knoten einer Teilbaumebene

Die Vergabe einer DeweyID für einen neuen Kontextknoten hinter dem letzten Knoten einer Teilbaumebene erfolgt ähnlich der Adressvergabe bei der anfänglichen Nummerierung eines taDOM-Baums. Befinden sich vor dem letzten *Division*-Wert, der immer ungerade ist, keine geraden *Divisions*, so wird der letzte *Division*-Wert wie bei der initialen Vergabe der DeweyIDs um den Wert *Distance* erhöht. Der Nachfolger von *7:1.3.15* ist somit bei einem *Distance*-Wert von *4* bspw. *7:1.3.19*.

Wenn sich vor dem letzten *Division*-Wert gerade *Division*-Werte befinden, so werden alle *Divisions* bis auf die letzte *Division* mit geradem Wert abgeschnitten, die danach um *Distance-1* erhöht wird. Somit erhält der neue Knoten an letzter Position einen ungeraden *Division*-Wert, der wieder der initialen Vergabe entspricht. Bei einem angenommenen *Distance*-Wert von *4* erhält der nachfolgende Geschwisterknoten von *7:1.3.14.6.5* die DeweyID *7:1.3.17*.

Neue DeweyID vor dem ersten Knoten einer Teilbaumebene

Befindet sich beim ersten Knoten der Ebene vor dem letzten *Division*-Wert ebenfalls ein ungerader *Division*-Wert, so wird der letzte *Division*-Wert für die weitere Berechnung benutzt, ansonsten wird der erste der geraden *Division*-Werte vor der letzten *Division* bestimmt. Der so ermittelte *Division*-Wert wird halbiert und bei Bedarf aufgerundet oder um *1* erhöht, um wieder einen ungeraden Wert für die letzte *Division* zu erhalten. Durch das Halbieren können vor und hinter dem neuen Knoten wiederum gleich viele Knoten eingefügt werden. Beispielsweise wird der Geschwisterknoten vor dem ersten Kindknoten mit der ID *7:1.5.9* mit *7:1.5.5* adressiert und der vor *7:1.3.8.4.3* mit *7:1.3.5*.

Ist der für die Halbierung bestimmte *Division*-Wert gleich 2, so muss dieser für die neue DeweyID übernommen werden, da aufgrund der Attributwurzeladressierung mit *1* kleinere Werte als *2* nicht erlaubt sind. Daher wird der erste *Division*-Wert ungleich *2* für die Halbierung herangezogen. Als Beispiel erhält der Geschwisterknoten vor *7:1.5.2.2.8.9* die ID *7:1.5.2.2.5*.

Ist der letzte *Division*-Wert des ersten Knotens der Teilbaumebene gleich *3*, so muss dieser entfernt werden, und die neue DeweyID erhält ergänzend die *Divisions 2* und *Distance+1*. Der Geschwisterknoten vor *7:1.5.3* wäre mit einem Distance-Wert von *4* auf diese Weise *7:1.5.2.5*.

Neue DeweyID zwischen zwei benachbarten Knoten

Wird zwischen zwei bestehende Knoten mit den DeweyIDs d_1 und d_3 ein neuer Geschwisterknoten eingefügt, so muss eine DeweyID d_2 gefunden werden, die zwischen d_1 und d_3 liegt (eine genaue Definition der Ordnung auf DeweyIDs folgt im nächsten Abschnitt 4.3.3) und möglichst viele weitere Einfügungen zwischen d_1 und d_2 bzw. d_2 und d_3 ermöglicht. Der neue Geschwisterknoten mit der ID d_2 hat dasselbe Elternelement wie die beiden bereits vorhandenen Knoten, d. h., d_1 und d_2 unterscheiden sich nur im letzten *Division*-Wert, der immer ungerade ist, und evtl. in den direkt davor stehenden geraden *Division*-Werten. Alle wiederum davor liegenden *Divisions* werden vom Elternelement bestimmt und sind für die Ermittlung der neuen DeweyID d_2 uninteressant, da sie als Präfix übernommen werden müssen.

Wenn zwischen die ersten beiden *Division*-Werte, in denen sich d_1 und d_3 unterscheiden, eine ungerade Zahl passt, so wird die in der Mitte dieser beiden Werte liegende ungerade Zahl als *Division*-Wert für die neue DeweyID d_2 gewählt. Für die IDs *7:1.5.6.7.5* und *7:1.5.6.7.16.5* ist dies bspw. die *7:1.5.6.7.11*. Wenn zwischen die beiden *Division*-Werte nur noch eine gerade Zahl passt, so wird diese gerade Zahl benutzt und die DeweyID um eine weitere *Division* mit dem Wert *Distance+1* erweitert. Zwischen der ID *7:1.5.6.7.5* und *7:1.5.6.7.7* findet so mit einem *Distance*-Wert von *4* die ID *7:1.5.6.7.6.5* Platz.

Passt zwischen die beiden sich unterscheidenden *Division*-Werte kein weiterer Wert mehr, weil sie direkt aufeinander folgen, so ist einer der beiden Werte gerade und der andere ungerade. Da DeweyIDs immer mit ungeraden *Division*-Werten enden, bedeutet dies, dass hinter der ID mit dem geraden *Division*-Wert noch weitere *Divisions* folgen müssen.

Besitzt die ID d_1 weitere *Divisions*, so wird die neue DeweyID d_2 aus den gemeinsamen *Divisions* von d_1 und d_3, der ersten unterschiedlichen *Division* von d_1 und dem darauf folgenden *Division*-Wert von d_1 erhöht um *Distance* gebildet. Sollte dieser neu ermittelte *Division*-Wert gerade sein, wird er um *1* erniedrigt, um wieder eine ungerade Division an der letzten Position zu erhalten. Diese Subtraktion liefert immer eine gültige DeweyID, da zuvor ein *Distance*-Wert von mindestens *2* addiert wurde. Die neue DeweyID zwischen bspw. *7:1.5.4.5* und *7:1.5.5* ist bei einem *Distance*-Wert von *4* somit *7:1.5.4.9*.

Besitzt im umgekehrten Fall die ID d_3 noch weitere *Divisions*, so erhält die neue DeweyID d_2 alle übereinstimmenden *Divisions* von d_1 und d_3, die erste unterschiedliche *Division* von d_3, alle evtl. darauf folgenden *Divisions* mit dem Wert *2* und die erste dann folgende *Division* geteilt durch *2*. Sollte dabei ein gerader Wert entstehen, muss dieser für die abschließende ungerade *Division* um *1* erhöht werden. Hat die zu teilende *Division* den Wert *3*, so wird nicht dividiert, sondern die *Division*-Werte *2* und *Distance+1* angehängt. Im Beispiel wird so zwischen die DeweyIDs *7:1.5.6.7.5* und *7:1.5.6.7.6.2.2.13* die ID *7:1.5.6.7.6.2.2.7* eingefügt.

4.3.3 Auswertung von Pfadachsen

Zur Unterstützung der Anfrageverarbeitung setzen Datenbanksysteme u. a. Indexstrukturen ein, in denen auf relevante Knoten bzgl. eines festgelegten Kriteriums verwiesen wird. Der Verweis auf einen Knoten findet über dessen Adresse, in unserem Fall also eine DeweyID statt. DeweyIDs eignen sich in XML-Datenbanksystemen in idealer Weise für die Anwendung in Indexstrukturen und der Anfrageverarbeitung. Zum einen sind (wie wir im obigen Abschnitt gesehen haben) DeweyIDs stabil gegen jegliche Modifikationen am taDOM-Baum, d. h., es muss bei Änderungsoperationen keine Reorganisation der Verweise in einem Index durchgeführt werden. Zum anderen können DeweyIDs dazu eingesetzt werden, um einen Teil der Anfrageverarbeitung direkt auf den Indexdaten durchzuführen, da die Lage zweier DeweyIDs bzgl. aller zwölf XQuery-Pfadachsen (siehe Abschnitt 2.3.3) direkt aus den DeweyIDs selbst ohne Zugriff auf das eigentliche XML-Dokument berechnet werden kann.

Für die Bestimmung der Lage zweier DeweyIDs bzgl. einer Pfadachse muss zunächst die *sequentielle Ordnung* auf DeweyIDs definiert werden, die der Dokumentenordnung der durch sie adressierten Knoten entspricht: Für zwei gegebene DeweyIDs d_1 und d_2 wird zunächst die kleinere Anzahl *n* der jeweils vorhandenen *Divisions* ermittelt. Daraufhin werden nacheinander alle *Division*-Werte an den Positionen *1* bis *n* verglichen. Die DeweyID d_1 ist *kleiner* als d_2, sobald der *Division*-Wert von d_1 an der aktuell untersuchten Position kleiner ist als der *Division*-Wert von d_2. Ist im umgekehrten Fall der *Division*-Wert von d_2 geringer als der von d_1, so ist die DeweyID d_2 *kleiner* als d_1. Sollten alle verglichenen *Division*-Werte gleich sein, so ist die DeweyID mit der geringeren Anzahl von *Divisions* die *kleinere*. Ist zusätzlich auch noch die Anzahl der *Divisions* gleich, so handelt es sich bei d_1 und d_2 um dieselbe DeweyID.

Für eine beliebige DeweyID d_i, die bspw. während der Iteration über einen Index ermittelt wurde, kann nun die Lage bzgl. einer XQuery-Pfadachse zur DeweyID d_k des Kontextknotens *k* aus d_i und d_k berechnet werden:

- *Self*-Achse: Handelt es sich bei d_i und d_k um dieselbe DeweyID, so qualifiziert sich d_i bzgl. d_k für die *Self*-Achse.
- *Attribute*-Achse: Besteht d_i aus der DeweyID von d_k, erweitert um den *Division*-Wert *1* für die Attributwurzel und einer beliebigen weiteren Division, so adressiert d_i ein Attribut des Kontextknotens.
- *Parent*-Achse: Die Adresse des Elternknotens von *k* lässt sich aus d_k durch das Abschneiden der letzten *Division* und das anschließende Entfernen aller dann am Ende stehenden geraden *Division*-Werte ermitteln. Handelt es sich bei d_i und der Adresse des Elternknotens von *k* um dieselbe DeweyID, so liegt d_i auf der *Parent*-Achse des Kontextknotens.
- *Ancestor*-Achse: Ist d_i ein Präfix von d_k und ist zusätzlich d_i ungleich d_k (d_k besteht also aus mehr *Divisions* als d_i), so liegt d_i bzgl. des Kontextknotens auf der *Ancestor*-Achse.
- *Ancestor-or-self-Achse*: Liegt d_i auf der *Self*-Achse oder der *Ancestor*-Achse des Kontextknotens, so liegt d_i auf der *Ancestor-or-self*-Achse des Kontextknotens.
- *Preceding*-Achse: Ist d_i bzgl. der sequentiellen Ordnung kleiner als d_k und liegt d_i nicht auf der *Ancestor*-Achse von d_k, so adressiert d_i einen *Preceding*-Knoten von d_k.
- *Preceding-sibling*-Achse: Ist d_i kleiner als d_k und haben d_i und d_k dieselbe Eltern-DeweyID, so adressiert d_i einen *Preceding-sibling*-Knoten von d_k.
- *Descendant*-Achse: Liegt d_k auf der *Ancestor*-Achse von d_i, so gehört d_i zu einem Nachfahren auf der *Descendant*-Achse von *k*.
- *Descendant-or-self*-Achse: Liegt d_i bzgl. d_k auf der *Descendant*-Achse oder der *Self*-Achse, so liegt d_i auf der *Descendant-or-self*-Achse von d_k.
- *Child*-Achse: Adressiert d_k bzgl. d_i einen Knoten auf der *Parent*-Achse, so liegt d_i bzgl. d_k auf der *Child*-Achse.
- *Following-sibling*-Achse: Ist d_k kleiner als d_i und haben d_k und d_i dieselbe Eltern-DeweyID, so adressiert d_i einen *Following-sibling*-Knoten von *k*.
- *Following*-Achse: Wenn d_k kleiner als d_i ist und d_k nicht zu einem Vorfahren auf der *Ancestor*-Achse von d_i gehört, so adressiert d_i einen *Following*-Knoten von *k*.

4.4 Zusammenfassung

Dieses Kapitel hat das taDOM-Transaktionsmodell eingeführt. Dazu wurde zunächst mit dem taDOM-Baum das zugrunde liegende Datenmodell beschrieben und ausführlich erklärt, wie man Referenzen (bspw. durch die ID/IDREF(S)-Beziehung) auf das Datenmodell abbildet. Auf taDOM-Bäumen sind 19 Basisoperationen definiert, mit denen die Bäume traversiert und modifiziert werden können. Die Adressierung der Knoten erfolgt mit DeweyIDs, die trotz beliebiger Änderungsoperationen auf einem taDOM-Baum nicht reorganisiert werden müssen. Die Erzeugung solcher stabilen DeweyIDs für das Einfügen neuer Knoten wurde für alle auftretenden Sonderfälle detailliert behandelt. DeweyIDs eignen sich somit nicht nur für die reorganisationsfreie Knotenreferenzierung in Indexstrukturen, sondern können auch zur Unterstützung der Anfrageverarbeitung in XML-Datenbanksystemen eingesetzt werden. Dazu wurde gezeigt, wie die Berechnung der Lage zweier DeweyIDs bzgl. aller XQuery-Pfadachsen erfolgt.

KAPITEL 5 Kontrolle der Nebenläufigkeit

Wissenschaftliche Forschung läuft immer darauf hinaus, dass es plötzlich mehrere Probleme gibt, wo es früher ein einziges gab.
(Norman Mailer)

Zur gegenseitigen Isolation von Transaktionen, die nebenläufig auf taDOM-Bäumen operieren, führt dieses Kapitel die vier aufeinander aufbauenden pessimistischen hierarchischen Sperrprotokolle *taDOM2*, *taDOM2+*, *taDOM3* und *taDOM3+* ein. Da die Komplexität dieser Verfahren stark zunimmt, beschreibt Abschnitt 5.6 einen Beweis zur Gewährleistung der Vollständigkeit und Korrektheit der Protokolle bzgl. der in Abschnitt 4.2 definierten Basisoperationen auf taDOM-Bäumen. Die Protokolle basieren auf der Annahme rein navigierender Transaktionen und sind daher nicht in der Lage, alle potenziell auftretenden Phantome [HR99] beim Zugriff auf taDOM-Bäume über sekundäre Indexstrukturen zu verhindern. Zu deren Vermeidung wird in Abschnitt 5.8 der Ansatz der *wertbasierten Achsensperren* vorgestellt.

Der Einsatz von *Prädikat-Sperren* (logische Sperren) verbietet sich aus den bekannten Gründen [HR99] der Unentscheidbarkeit, ob zwei beliebige Prädikate disjunkte Mengen beschreiben. Auch das weiterführende Konzept der Präzisionssperren [JBB81] zur Verringerung der Implementierungsprobleme stellt für einen navigationsbasierten Datenzugriff keine Lösung dar, weil für jeden einzelnen Navigationsschritt ein neues Prädikat erstellt oder bereits vorhandene Prädikate erweitert werden müssten. Dies hätte im ersten Fall die Folge eines gewaltigen Suchraums, im letzteren Fall müsste eine komplexe Logik entwickelt werden, die die vorhandenen Prädikate analysiert und entsprechend der durchgeführten Navigationsschritte und Zugriffsoperatoren neu definiert. Aus diesen Gründen favorisieren wir die auch in relationalen Datenbanksystemen eingesetzten hierarchischen Sperrprotokolle in Kombination mit Bereichssperren auf Indexstrukturen.

Die Namensgebung der taDOM-Sperrprotokolle basiert auf der Unterstützung der geforderten Funktionalität der DOM Level [DOM2, DOM3]. taDOM2 stellt nur Sperrmodi bereit, die eine genau abgestimmte Isolation der Funktionalität des DOM Level 2 ermöglichen. Dies bedeutet, dass zu einer feingranularen Synchronisation der mächtigeren Basisoperationen auf dem taDOM-Datenmodell das zusätzliche Konzept der *virtuellen Namensknoten* in Abschnitt 5.1.3 eingeführt werden muss. Das Protokoll taDOM3 umgeht dies mit speziell dafür definierten Sperrmodi. Beide Protokolle taDOM2 und taDOM3 erfordern bei der Konversion diverser Sperrmodi den Zugriff auf den zugrunde liegenden taDOM-Baum, sodass in diesem Fall der Sperrverwalter einer konkreten Systemimplementierung nicht jede beliebige Sperranforderung unabhängig von der Speicherungskomponente bearbeiten kann. Durch eine erneute Hinzunahme weiterer Sperrmodi beheben die Protokolle taDOM2+ und taDOM3+ auch diesen Nachteil.

Die Notwendigkeit dieser kontinuierlichen Erweiterungen der Protokolle mit neuen Sperrmodi wird durch entsprechende Leistungsmessungen in Kapitel 7 mit einem ansteigenden Transaktionsdurchsatz nachgewiesen.

5.1 taDOM2

taDOM2 [HH04a] ist ein hierarchisches Sperrprotokoll zur Transaktionsisolation auf taDOM-Bäumen. Vor dem Zugriff auf den Knoten eines taDOM-Baums muss eine Transaktion für diesen Knoten eine Sperre anfordern. Die Sperranforderung beinhaltet neben dem Identifikator der Transaktion die Adresse des zu sperrenden Kontextknotens und den benötigten Sperrmodus. Die Knotenadresse wird mit der DeweyID (Abschnitt 4.3) des Knotens angegeben. Da hierarchische Sperrprotokolle auf Baumstrukturen Anwartschaftssperren auf allen Knoten von der Wurzel abwärts zum Kontextknoten anfordern müssen [GR93], eignen sich DeweyIDs besonders für die Knotenadressierung, da sich der vollständige Pfad aller dieser Knoten allein aus der Adresse des Kontextknotens ohne Zugriff auf einen taDOM-Baum berechnen lässt.

5.1.1 Knotensperren

Zum Sperren von Knoten eines taDOM-Baums bietet das taDOM2-Protokoll acht verschiedene Sperrmodi an, die im Wesentlichen dem verfeinerten RIX-Protokoll [HR99] mit einer Erweiterung zum Sperren einer vollständigen Teilbaumebene entsprechen. Diese Sperrmodi werden nun im Folgenden detailliert beschrieben.

- Die IR-Sperre (*intention read*) weist darauf hin, dass in dem darunter liegenden Teilbaum eine Lesesperre mit den Sperrmodi NR, LR oder SR vorliegt. Eine IR-Sperre erfordert auf dem Elternknoten ebenfalls eine IR-Sperre.
- Die NR-Sperre (*node read*) wird auf einem Kontextknoten für den Lesezugriff angefordert. Dabei ist zu beachten, dass die NR-Sperre nicht der R-Sperre klassischer Verfahren entspricht, da sie nur den Kontextknoten selbst für eine Leseoperation sperrt und keinerlei Einfluss auf den darunter liegenden Teilbaum hat. Dieser Sperrmodus wird typischerweise bei der Anfrageverarbeitung angefordert, wenn gezielt der Wert eines einzelnen Knotens ermittelt werden muss. Die NR-Sperre erfordert auf dem Elternknoten eine IR-Sperre.
- Die LR-Sperre (*level read*) sperrt den Kontextknoten selbst und alle seine direkten Kindknoten für den Lesezugriff einer Transaktion. Die Knoten, die sich im darunter liegenden Teilbaum unterhalb der Kindknoten befinden, sind von dieser Sperre nicht betroffen. Der Sperrmodus ermöglicht das Ermitteln aller Kinder eines Knotens mit nur einer Sperranforderung. Die LR-Sperre erfordert auf dem Elternknoten eine IR-Sperre.
- Die SR-Sperre (*subtree read*) sperrt den Kontextknoten und alle Nachfahren im darunter liegenden Teilbaum für den Lesezugriff und entspricht damit der bekannten R-Sperre. Dieser Sperrmodus wird typischerweise für die Rekonstruktion eines gesamten XML-Fragments angefordert. Die SR-Sperre benötigt auf dem Elternknoten eine IR-Sperre.
- Die IX-Sperre (*intention exclusive*) weist darauf hin, dass im darunter liegenden Teilbaum eine Schreiboperation mit dem Sperrmodus SX durchgeführt wird. Im Gegensatz zu den klassischen hierarchischen Sperrprotokollen darf diese Sperre nur auf einem Knoten angefordert werden, wenn die betroffene Schreiboperation auf einem Nachfahren erfolgt, der kein Kindknoten ist. Findet die Schreiboperation auf einem Kindknoten statt, so muss die CX-Sperre angefordert werden. Die IX-Sperre erfordert auf dem Elternknoten ebenfalls eine IX-Sperre.
- Die CX-Sperre (*child exclusive*) zeigt an, dass auf einem Kindknoten eine Schreiboperation mit dem Sperrmodus SX stattfindet. Die CX-Sperre ist zusätzlich zur IX-Sperre für das Protokoll erforderlich, damit direkt auf dem Kontextknoten entschieden werden kann, ob eine LR-Sperre gewährt werden darf. Die IX-Sperre indiziert eine Schreiboperation auf einem

Nachfahren, der definitiv kein Kind ist, und ist somit zur LR-Sperre kompatibel. Die CX-Sperre dagegen deutet die Schreiboperation auf einem direkten Kindknoten an und ist daher nicht kompatibel zur LR-Sperre. Eine CX-Sperre erfordert auf dem Elternknoten eine IX-Sperre.

- Die SU-Sperre (*subtree update*) sperrt den Kontextknoten und den darunter liegenden Teilbaum für den Lesezugriff mit der Option, diesen Sperrmodus zu einem späteren Zeitpunkt in eine SR-Sperre (*Downgrade*) oder eine SX-Sperre (*Upgrade*) zu konvertieren. Daher ist der SU-Sperrmodus zu keinem anderen Modus kompatibel, sodass trotz der anfänglichen Leseabsicht weitere lesende Transaktionen blockiert werden, um später ein mögliches *Upgrade* zügig ohne Wartezeiten durchführen zu können. Werden bei der Änderungsabsicht von Transaktionsoperationen konsequent direkt SU-Sperren angefordert, ohne die betroffenen Teilbäume zuvor mit SR-Sperren zu schützen, so vermeidet dieses Vorgehen in solchen Situationen jegliches Auftreten von Deadlocks, da diese Situationen in Wartezustände serialisiert werden [GR93]. Die SU-Sperre erfordert auf dem Elternknoten eine IR-Sperre, da zunächst nur eine Leseabsicht vorliegt. Eine IX-Sperre darf nicht angefordert werden, weil diese im Falle eines *Downgrade* in eine IR-Sperre konvertiert werden müsste. Da auf allen Vorfahren aber ohne Zugriff auf die Sperren aller ihrer Kindknoten nicht entschieden werden kann, ob eine Konversion von IX nach IR zulässig ist (möglicherweise ist die IX-Sperre auch durch Änderungsoperationen in anderen Teilbäumen bedingt), wird diese Sperrkonversion aus Effizienzgründen verboten. Somit erfolgt beim *Downgrade* auf SR die Erhaltung der IR-Sperren bis zur Wurzel und beim *Upgrade* auf SX (s. u.) eine Konversion in CX- bzw. IX-Sperren.
- Die SX-Sperre (*subtree exclusive*) sperrt den Kontextknoten und alle Nachfahren in dem darunter liegenden Teilbaum für den exklusiven Schreibzugriff. Diese Zugriffe beinhalten die Modifikation von Werten und das Hinzufügen und Löschen von Knoten. Die Semantik dieses Sperrmodus entspricht somit der X-Sperre eines konventionellen hierarchischen Protokolls. Im Gegensatz dazu erfordert die SX-Sperre jedoch auf dem Elternknoten eine CX-Sperre, die wiederum auf allen Vorfahren eine IX-Sperre benötigt.

	-	IR	NR	LR	SR	IX	CX	SU	SX
IR	+	+	+	+	+	+	+	-	-
NR	+	+	+	+	+	+	+	-	-
LR	+	+	+	+	+	+	-	-	-
SR	+	+	+	+	+	-	-	-	-
IX	+	+	+	+	-	+	+	-	-
CX	+	+	+	-	-	+	+	-	-
SU	+	+	+	+	+	-	-	-	-
SX	+	-	-	-	-	-	-	-	-

a) Kompatibilitätsmatrix

	-	IR	NR	LR	SR	IX	CX	SU	SX
IR	IR	IR	NR	LR	SR	IX	CX	SU	SX
NR	NR	NR	NR	LR	SR	IX	CX	SU	SX
LR	LR	LR	LR	LR	SR	IX_{NR}	CX_{NR}	SU	SX
SR	SR	SR	SR	SR	SR	IX_{SR}	CX_{SR}	SR	SX
IX	IX	IX	IX	IX_{NR}	IX_{SR}	IX	CX	SX	SX
CX	CX	CX	CX	CX_{NR}	CX_{SR}	CX	CX	SX	SX
SU	SU	SU	SU	SU	SU	SX	SX	SU	SX
SX	SX	SX	SX	SX	SX	SX	SX	SX	SX

b) Konversionsmatrix

Abbildung 78: Das Sperrprotokoll taDOM2

Die Kompatibilitätsmatrix des taDOM2-Protokolls ist in Abbildung 78a dargestellt. Deren Anwendung und die Semantik der Sperrmodi werden nun mit einem Beispiel, welches das erweiterte XML-Dokument aus Kapitel 2 zugrunde legt, nochmals verdeutlicht. Die aus den folgenden Operationen resultierende Sperrsituation zeigt Abbildung 79. Transaktion T_1 liest das *Name*-Element des ersten Kunden. Dazu muss zunächst auf den Knoten *Bank*, *Kunden* und *Kunde* eine IR-Sperre angefordert werden, bevor auf *Name* eine NR-Sperre eingetragen wird.

Transaktion T_2 bestimmt alle Kinder des *Kunden*-Knotens und fordert dazu eine IR-Sperre auf der Dokumentwurzel und eine LR-Sperre auf dem Kontextknoten *Kunden* an. Hier entsteht keine Blockierung, da IR-Sperren zueinander und mit LR-Sperren kompatibel sind. Eine weitere Transaktion T_3 möchte vollständig die Daten eines Kunden rekonstruieren und fordert dazu auf *Bank* und *Kunden* jeweils eine IR-Sperre und auf der Teilbaumwurzel *Kunde* eine SR-Sperre an. Die Transaktion T_4 erstellt ein neues Konto und benötigt dazu eine IX-Sperre auf *Bank*, eine CX-Sperre auf dem Elternknoten des neuen Kontos und für das neue *Konto*-Element selbst eine SX-Sperre. Auch diese Operation kann sofort durchgeführt werden, da IR- und IX-Sperren kompatibel sind. Weil CX-Sperren ebenfalls miteinander verträglich sind, kann auch die Transaktion T_5 das bereits vorhandene erste Konto löschen; hierzu ist neben der SX-Sperre auf dem *Konto*-Element eine IX-Sperre auf der Dokumentwurzel und eine CX-Sperre auf *Konten* erforderlich. Die Transaktion T_6, die eine Liste alle *Konto*-Element erstellen möchte, wird dagegen bereits auf dem *Konten*-Element blockiert, da die dort erforderliche LR-Sperre zur Bestimmung aller Kinder mit den vorhandenen CX-Sperren von T_4 und T_5 inkompatibel ist (CX sagt aus, dass ein Kindknoten modifiziert wird, folglich kann bereits auf dem Elternknoten entschieden werden, dass nicht alle Kinder gelesen werden dürfen). Im Gegensatz zur CX-Sperre ist die IX-Sperre allerdings mit LR kompatibel, da IX auf die exklusive Nutzung eines Nachfahren, der kein Kind ist, hinweist. Somit kann Transaktion T_7 die Adresse des ersten Kunden löschen, da die erforderlichen CX- und IX-Sperren auf den Vorfahren mit den bereits eingetragenen IR-, IX- und LR-Sperren kompatibel sind. Eine IX-Sperre ist jedoch unverträglich mit einer SR-Sperre, somit wird Transaktion T_8 bei der Rekonstruktion aller Kunden blockiert.

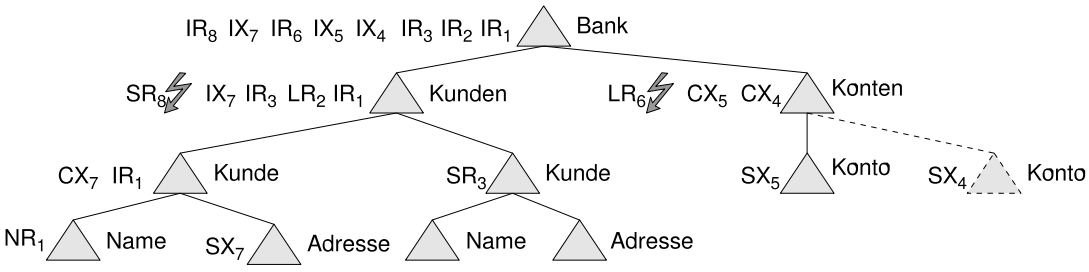


Abbildung 79: Knotensperren in taDOM2

Die taDOM-Sperrprotokolle verwalten pro Transaktion und Knoten nur höchstens eine Sperre. Das bedeutet, wenn für eine Transaktion schon eine Sperre auf einem Knoten eingetragen ist und dieselbe Transaktion fordert auf diesem Knoten für eine Folgeoperation eine weitere Sperre an, so muss ein Sperrmodus bestimmt werden, der die Transaktion bzgl. der bereits eingetragenen und der neu angeforderten Sperre ausreichend gegen andere Transaktionen isoliert. Dazu wird die in Abschnitt 3.3.2 eingeführte Technik der Konversionsmatrizen benutzt.

Abbildung 78b zeigt für das taDOM2-Sperrprotokoll die Konversionsmatrix, deren Anwendung im folgenden Beispiel verdeutlicht wird. Transaktion T_1 greift mit einer NR-Sperre auf den *Konto*-Knoten zu und fordert dafür zunächst auf dem *Bank*- und *Konten*-Element jeweils eine IR-Sperre an. Transaktion T_2 ermittelt alle Kinder des *Kunden*-Knotens und liest danach das *Adresse*-Element. Dazu benötigt T_2 eine IR-Sperre auf der Dokumentwurzel *Bank*, eine LR-Sperre auf *Kunden*, die den *Kunden*- und alle *Kunde*-Knoten schützt, und eine NR-Sperre auf *Adresse* selbst. Die Transaktion T_3 rekonstruiert vollständig das letzte *Kunde*-Fragment, fordert dazu auf den Vorfahren IR-Sperren und auf *Kunde* eine SR-Sperre an. Diese Ausgangssituation ist in Abbildung 80a dargestellt. Navigiert T_1 nun zum Elternelement, so ist dafür auf

dem betroffenen *Konten*-Knoten eine NR-Sperre für den Lesezugriff erforderlich. Da jedoch bereits eine IR-Sperre auf *Konten* vorliegt, muss eine Sperrkonversion durchgeführt werden. Die Konversionsmatrix liefert für eine existierende IR-Sperre (Matrixkopfzeile) und eine angeforderte NR-Sperre (Matrixkopfspalte) die resultierende NR-Sperre, die auf dem *Konten*-Knoten eingetragen wird. Die Konversion zur NR-Sperre bewirkt auf dem Wurzelknoten die Anforderung einer IR-Sperre; hier ist jedoch nichts mehr zu tun, da eine IR-Sperre für T_1 bereits vorliegt. Transaktion T_2 möchte den *Adresse*-Knoten modifizieren und benötigt dazu IX-Sperren auf *Bank* und *Kunden*, eine CX-Sperre auf dem ersten *Kunde*-Element und eine SX-Sperre auf dem betroffenen *Adresse*-Knoten. Gemäß der Konversionsmatrix werden die vorhandenen IR-Sperren bei Anforderung der IX-Sperrmodi durch IX-Sperren ersetzt. Für das *Kunden*-Element, auf dem zunächst eine LR-Sperre eingetragen ist, kommt eine besondere Regel zum Einsatz: Die Konversionsmatrix liefert für die Anforderung einer IX-Sperre bei existierender LR-Sperre den resultierenden Modus IX_{NR}. Dieser besagt, dass die LR-Sperre auf dem Kontextknoten durch eine IX-Sperre zu ersetzen ist und auf allen Kindknoten (in diesem Fall die beiden *Kunde*-Elemente) eine NR-Sperre angefordert werden muss. Somit sorgt die resultierende Sperrsituation für eine äquivalente Transaktionsisolation, da IX sowohl der neuen als auch der alten Sperranforderung auf dem Kontextknoten genügt und die neu eingetragenen NR-Sperren den impliziten Schutz der Kinder durch die frühere LR-Sperre bewahren. Nach dieser Aktion wird auf dem ersten *Kunde*-Element eine CX-Sperre angefordert, die auch als Ergebnis einer Konversion mit der nun existierenden NR-Sperre der vorherigen Konversion eingetragen wird. Schließlich kann die SX-Sperre auf *Adresse* gewährt werden. Auf ähnliche Weise erfolgt die Sperrkonversion für Transaktion T_3, die den *Adresse*-Knoten des zweiten *Kunde*-Elements ändert. Für die bereits vorhandene SR-Sperre muss der CX_{SR}-Modus beantragt werden, der eine CX-Sperre auf *Kunde* und zunächst SR-Sperren auf allen Kindern (*Name* und *Adresse*) anfordert und danach die SR-Sperre auf *Adresse* in SX konvertiert. Die Sperrsituation nach den beschriebenen Konversionen zeigt Abbildung 80b.

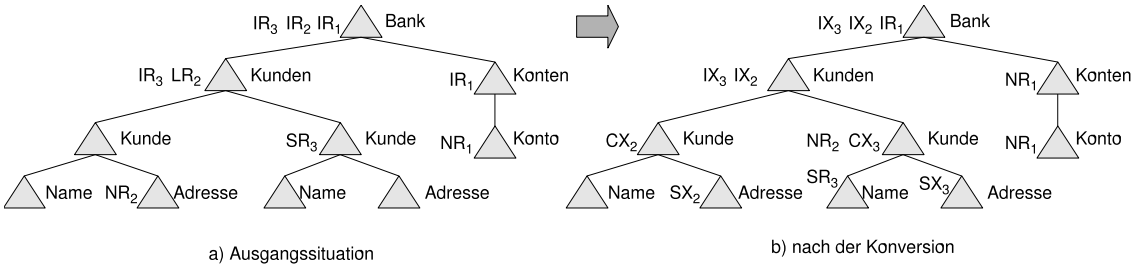


Abbildung 80: Sperrkonversion in taDOM2

Up- und Downgrade der Update-Sperre

Beabsichtigen zwei Transaktionen T_1 und T_2, parallel einen Teilbaum zu ändern, so fordern sie zunächst SR-Sperren zum Lesen der Ausgangsdaten an, die aufgrund der Kompatibilität für beide Transaktionen gewährt werden können. Beantragen beide Transaktionen kurz danach SX-Sperren für die eigentliche Änderung, so entsteht eine Deadlock-Situation, da beide Transaktionen blockiert werden und auf die Sperrfreigabe der jeweils anderen Lesesperre warten müssen. Dieser Deadlock kann nur mit dem Zurücksetzen einer der beiden Transaktionen aufgelöst werden. Mit der SU-Sperre entsteht aus diesem Szenario nur eine Wartesituation, sodass beide Transaktionen sequentiell zur Ausführung kommen [GR93]. Nachdem T_1 eine SU-Sperre beantragt hat, muss T_2 auf ihre SU-Sperre warten, da SU-Sperren zueinander inkompatibel

sind. T_1 kann somit von SU nach SX konvertieren (*Upgrade*), die Änderung vornehmen und die Sperre am Transaktionsende entfernen. Danach erhält T_2 die angeforderte SU-Sperre, liest den Teilbaum, konvertiert nach SX und führt ihre Änderung durch.

Da die SU-Sperre allerdings hauptsächlich zum Einsatz kommt, wenn zunächst nachgeprüft werden muss, ob der Teilbaum überhaupt für eine Änderungsoperation in Frage kommt, kann der SU-Modus auch zurück in eine SR-Sperre konvertiert werden (*Downgrade*). Dies bedeutet allerdings, dass unter einer SU-Sperre im Teilbaum ausschließlich Leseoperationen durchgeführt werden dürfen. Erfolgt die Anforderung einer SX-Sperre innerhalb des Teilbaums und nicht auf der Teilbaumwurzel, für die die SU-Sperre eingetragen ist, so bewirkt dies je nach Position der SX-Sperre die Anforderung einer IX- bzw. CX-Sperre auf der Teilbaumwurzel. In diesem Fall wird jedoch nicht eine Konversionsregel wie IX_{SU} oder CX_{SU} verwendet, da eine konkrete Systemimplementierung den *Up*- bzw. *Downgrade* direkt auf dem Knoten durchführen sollte, auf dem auch eine Update-Sperre angefordert wurde. Sollte doch eine Exklusivsperre unterhalb der Teilbaumwurzel angefordert werden, so müsste zum späteren *Downgrade* der nicht modifizierten Teile ein aufwändiger Algorithmus zur Lokalisierung von im Rahmen der Konversionen angeforderten Update-Sperren erfolgen, was einen deutlich erhöhten Aufwand für die Transaktionsverwaltung bedeutet. Daher wird bei Anforderungen von IX- oder CX-Sperren bei einer existierenden SU-Sperre (die wie eben beschrieben allerdings nicht vorkommen sollten) eine Konversion in den SX-Sperrmodus vorgenommen. Der explizite *Up*- bzw. *Downgrade* selbst erfolgt bei bereits vorhandener SU-Sperre mit Anforderung einer SR- bzw. SX-Sperre, womit gemäß der Konversionsmatrix eine Transformation in eine SR- bzw. SX-Sperre erfolgt.

Sperrtiefe

Da für das beschriebene Sperrprotokoll bei vielen nebenläufigen Transaktionen und großen taDOM-Bäumen u. U. eine hohe Anzahl von Sperren zu verwalten ist, muss bei einer konkreten Implementierung mit beschränkten Ressourcen eine Möglichkeit vorgesehen werden, zur Laufzeit die Anzahl der Sperren zu reduzieren bzw. unter einer vorgegebenen Obergrenze zu halten.

Dazu werden für taDOM-Bäume zunächst *Ebenen* eingeführt. Eine Ebene enthält alle Knoten, die dieselbe Anzahl von Vorfahren besitzen; die Anzahl der Vorfahren bestimmt die *Nummer* der Ebene. Knoten, die in einer Ebene enthalten sind, *liegen* in dieser Ebene. Die Dokumentwurzel liegt somit in *Ebene 0*, *Ebene 1* enthält alle Kinder der Wurzel und so weiter. Eine Ebene *j liegt* oder *ist tiefer* oder *unterhalb* einer Ebene *i*, wenn $i < j$ gilt. Für einen Kontextknoten *k*, der in einer Ebene *j* liegt, die tiefer als eine Ebene *i* ist, sagen wir, *k liegt* oder *ist tiefer* oder *unterhalb* der Ebene *i* oder der Sperrtiefe *i*.

Während der Anwendung des Sperrprotokolls kann nun zur Verringerung der verwalteten Sperren ein zusätzlicher Parameter, die so genannte *Sperrtiefe*, eingestellt werden. Die Sperrtiefe bewirkt, dass auf taDOM-Bäumen nur Sperren bis einschließlich der durch die Sperrtiefe bestimmten Ebene eingetragen werden dürfen. Die Anforderung einer Lesesperre (NR-, LR- und SR-Sperre) auf einem Kontextknoten *k*, der unterhalb der Sperrtiefe *t* liegt, wird in eine SR-Sperranforderung auf dem Vorfahren *a*, der in der Ebene *t* liegt, transformiert. Analog erfolgt eine Abbildung der Anforderung einer SU- bzw. SX-Sperre auf *k* in eine Anforderung der SU- bzw. SX-Sperre auf *a*. Die Sperrtiefe bewirkt somit, dass eine Sperre für einen Knoten unterhalb dieser Sperrtiefe nicht mehr auf dem Knoten selbst, sondern durch eine der Teilbaumsperren SR, SU oder SX auf einem Vorfahren in der Sperrtiefe realisiert wird. Wir sprechen dabei von einer *Vergröberung des Sperrgranulats*. Eine Sperrtiefe von *0* erlaubt in einfacher Weise die Simulation eines Protokolls mit Sperren auf Dokumentebene.

Abbildung 81 zeigt exemplarisch eine Vergröberung der Sperrsituation aus Abbildung 80b mit einer Sperrtiefe von *2*. Die Anforderung der NR-Sperre von T_1 auf dem *Konto*-Knoten in Ebene *2* resultiert in einer SR-Sperre. Ebenso verhält es sich mit der NR-Sperre von T_2 auf dem zweiten *Kunde*-Element. Die SX-Sperre der Transaktion T_2 auf dem *Adresse*-Knoten wird durch eine SX-Sperre auf dem Elternelement *Kunde* ersetzt, das sich in der Sperrtiefenebene befindet. Damit ist nun trotz der ursprünglichen Anforderung einer Exklusivsperre auf *Adresse* das gesamte Fragment des ersten *Kunde*-Elements gesperrt. Für das zweite *Kunde*-Element ergibt sich sogar die Situation, dass T_3 blockiert werden muss, da die aufgrund der Sperrtiefe erforderliche SX-Sperre auf *Kunde* nicht zur vorhandenen SR-Sperre von T_2 kompatibel ist. Auf diese Weise ist einfach zu erkennen, dass eine Verringerung der verwalteten Sperren durch eine geringere Sperrtiefe schnell zu einer Einschränkung der Parallelität bei der Transaktionsverarbeitung führen kann. Kapitel 7 belegt dies mit ausführlichen Leistungsmessungen.

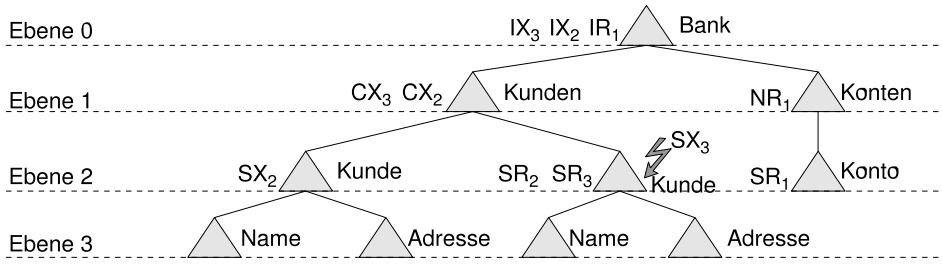


Abbildung 81: Anwendung der Sperrtiefe

Äquivalenz der IR- und NR-Sperre

Wie in Abbildung 78a zu erkennen ist, sind die Sperrmodi IR und NR sowohl in der Kompatibilitäts- als auch der Konversionsmatrix äquivalent zueinander. Dies ist auch beim Sperrprotokoll taDOM2+ der Fall, weil in diesen Protokollen der Sperrzustand eines Knotens nicht zwischen dem expliziten Lesen (NR) und der Anwartschaft (IR) unterscheidet. In einer konkreten Implementierung können daher beide Sperrmodi durch einen gemeinsamen Sperrmodus realisiert werden. Für die Argumentation in dieser Arbeit werden die Sperrmodi zur Vollständigkeit jedoch explizit aufgeführt, da sie in den folgenden Protokollen taDOM3 und taDOM3+ beide benötigt werden und sich dort voneinander unterscheiden.

5.1.2 Kantensperren

Die Knotensperren sorgen für ausreichende Isolation beim direkten Zugriff auf die Knoten des taDOM-Baums. Wenn eine Transaktion einen Knoten liest, sind somit der gelesene Knoten selbst und alle seine Vorfahren bis zur Dokumentwurzel durch die beschriebenen Lese- und Anwartschaftssperren vor Änderungen einer weiteren parallel laufenden Transaktion geschützt. Die Struktur des taDOM-Baums allerdings, insbesondere die Nachbarschaftsbeziehungen zwischen Geschwisterknoten und die Eigenschaft eines Knotens, das erste oder letzte Kind seines Elternelements zu sein, unterliegt nicht diesem Schutz.

Navigiert eine Transaktion bspw. von einem Kontextknoten k_1 zu dessen nächstem Geschwisterknoten k_2, so muss bis zum Ende der Transaktion sichergestellt werden, dass k_2 der nächste Geschwisterknoten von k_1 bleibt, und kein neuer Knoten dazwischen eingefügt wird (das Ändern oder Löschen von k_2 selbst verhindert die Knotenlesesperre, die von der Transaktion angefordert werden muss). Analog gilt dies für den vorherigen Geschwisterknoten und das erste und letzte Kind des Kontextknotens, welche direkt durch die in Abschnitt 4.2 eingeführten Navigationsoperationen auf taDOM-Bäumen erreichbar sind.

Ein weiteres Problem stellt das Löschen von Knoten aus dem taDOM-Baum dar. Löscht eine Transaktion T_1 bspw. den ersten Kindknoten eines Elements, so wird dadurch der bisher zweite Kindknoten das neue erste Kind. Eine weitere Transaktion T_2, die nun den ersten Kindknoten des Elements bestimmen möchte, muss bereits vor der Ermittlung der Knotenadresse des ersten Kindes blockiert werden, da der tatsächlich für T_2 zurückgelieferte erste Kindknoten vom Ausgang der Transaktion T_1 abhängt. Kommt T_1 erfolgreich zum Ende (*commit*), so wird das ursprünglich zweite Kind als erstes Kind für T_2 ermittelt. Wird T_1 allerdings abgebrochen (*roll-back*), so muss das ursprünglich erste Kind im Zuge der Kompensation von T_1 wieder in den Baum eingefügt und T_2 als erstes Kind zurückgeliefert werden.

Um dies zu gewährleisten, werden Kantensperren auf den virtuellen Navigationskanten eines taDOM-Baums benötigt, die gemäß dem klassischen RUX-Sperrverfahren [HR99] eine Lese-, Update- und Exklusivsperre zur Verfügung stellen. Damit die Sperrmodi im jeweiligen Kontext von denen der Knotensperren oder verwandten Ansätzen zu unterscheiden sind, werden sie mit *edge read*, *edge update* und *edge exclusive* bezeichnet.

- Eine ER-Sperre (*edge read*) wird für eine virtuelle Navigationskante angefordert, wenn eine Transaktion entlang dieser Kante zu dem durch sie bestimmten Knoten navigieren möchte. Erst wenn die Sperre gewährt ist, darf die Adresse des gewünschten Zielknotens bestimmt und eine entsprechende Knotensperre angefordert werden.
- Eine EU-Sperre (*edge update*) wird analog zur SU-Knotensperre angefordert, um das Ziel einer Kante zunächst für eine mögliche Änderung zu untersuchen.
- Eine EX-Sperre (*edge exclusive*) wird angefordert, wenn ein neuer Knoten eingefügt oder ein existierender Knoten entfernt werden soll. Die Sperre muss auf allen Navigationskanten eingetragen werden, die von der Strukturveränderung des Dokuments betroffen sind.

Die Kompatibilitäts- und Konversionsmatrix der Kantensperren für virtuelle Navigationskanten zeigt Abbildung 82. Die Konversion bei einer bereits eingetragenen EU-Sperre nach ER für eine angeforderte ER-Sperre bzw. nach EX für eine angeforderte EX-Sperre entspricht dem bereits bei den Kantensperren beschriebenen *Down*- bzw. *Upgrade*.

	-	ER	EU	EX
ER	+	+	-	-
EU	+	-	-	-
EX	+	-	-	-

Kompatibilitätsmatrix

	-	ER	EU	EX
ER	ER	ER	ER	EX
EU	EU	EU	EU	EX
EX	EX	EX	EX	EX

Konversionsmatrix

Abbildung 82: Kompatibilitäts- und Konversionsmatrix für Kantensperren

Eine gemeinsame Anwendung der Knoten- und Kantensperren zeigt Abbildung 83. Transaktion T_1 greift zunächst mit einer NR-Sperre auf den Wurzelknoten des Baums zu. Danach navigiert T_1 zum *Kunden*-Knoten, dem ersten Kind der Wurzel. Dazu wird zunächst auf der *firstChild*-Kante eine ER-Sperre angefordert und, wenn diese gewährt wurde, der *Kunden*-Knoten bestimmt und darauf eine NR-Sperre angefordert. Die dafür erforderliche IR-Sperre auf dem *Bank*-Knoten wird durch eine Konversion in die bereits vorhandene NR-Sperre überführt. Da T_1 nun der *Kunden*-Knoten als erstes Kind des *Bank*-Knotens bekannt ist, wird zusätzlich auf der *prevSibling*-Kante des *Kunden*-Knotens eine ER-Sperre eingetragen, da T_1 das Ergebnis dieses Navigationsschritts bereits kennt (*Kunden* kann keinen vorherigen Geschwisterknoten besitzen, weil *Kunden* erstes Kind von *Bank* ist). Analog verläuft die weitere Navigation zum ersten *Kunde*-Knoten. Von dort aus traversiert T_1 zum nächsten Geschwisterknoten, was eine

ER-Sperre auf der *nextSibling*-Kante des ersten *Kunde*-Knotens und eine NR-Sperre auf dem zweiten *Kunde*-Knoten zur Folge hat. Da T_1 nun auch Kenntnis über den vorherigen Geschwisterknoten des zweiten *Kunde*-Knotens besitzt (dies ist der *Kunde*-Knoten, von dem der Traversionsschritt ausging), wird auf der *prevSibling*-Kante des zweiten *Kunde*-Knotens ebenfalls eine ER-Sperre eingetragen. Transaktion T_2 navigiert vom Wurzelknoten zum *Konten*-Element, dem letzten Kind der Wurzel. Auf ähnliche Weise, wie oben beschrieben, bewirkt dies jeweils eine NR-Sperre auf *Bank* und *Konten* und eine ER-Sperre auf der *lastChild*-Kante von *Bank* und der *nextSibling*-Kante von *Konten*. Bewegt sich T_2 zum ersten Kind, so wird auf der *firstChild*-Kante von *Konten* eine ER-Sperre, danach auf dem *Konto*-Knoten eine NR-Sperre und schließlich auf der *prevSibling*-Kante von *Konto* eine ER-Sperre angefordert. Erfolgt nun ein Navigationsschritt von T_2 zum nächsten Geschwisterknoten, wird zuerst eine ER-Sperre auf der *nextSibling*-Kante des *Konto*-Knotens eingetragen. Danach wird versucht, auf den nächsten Geschwisterknoten zuzugreifen und dabei festgestellt, dass *Konto* bereits der letzte Kindknoten von *Konten* ist. Bevor jedoch dieses an T_2 gemeldet wird, erfolgt die Anforderung einer ER-Sperre auf der *lastChild*-Kante von *Konten*, da mit diesem Ergebnis T_2 nun auch das mit dem ersten identische letzte Kind des *Konten*-Knotens bekannt ist.

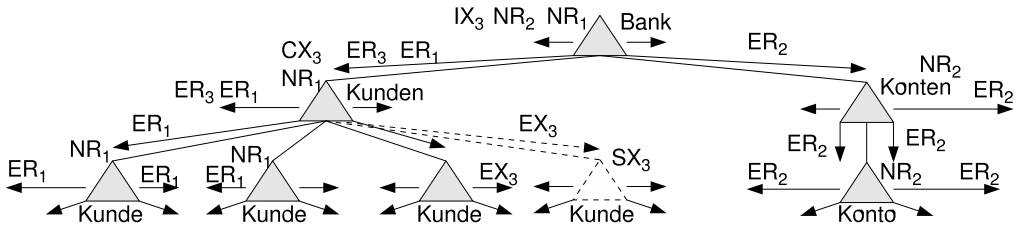


Abbildung 83: Knoten- und Kantensperren in taDOM2

Navigiert eine weitere Transaktion T_3 zum *Kunden*-Knoten und hängt ein weiteres *Kunde*-Element als letztes Kind an, so werden zunächst die IX- und CX-Sperre auf den Vorfahren angefordert, um mögliche Blockierungen mit LR- oder SR-Sperren auszulösen. Danach werden die zu modifizierenden Kanten *lastChild* des *Kunden*-Knotens und *nextSibling* des bisher letzten *Kunde*-Elements und schließlich der neue Knoten selbst exklusiv gesperrt.

Zur Aktualisierung eines Knotenwerts ist keine exklusive Kantensperre erforderlich, weil dadurch keine Änderung an der Dokumentstruktur vorgenommen wird. Wurde der zu ändernde Knoten bereits von einer anderen Transaktion gelesen, so ist dessen Wert durch eine entsprechende Knotensperre geschützt, die zur Blockierung der Änderungstransaktion führt. Benutzt im Beispiel in Abbildung 83 die Transaktion T_1 auf dem vor der Einfügeoperation letzten *Kunde*-Knoten nicht die *getNextSibling()*-Operation (inkompatibel zur vorhandenen EX-Sperre auf der *nextSibling*-Kante), so könnte T_1 auch vor dem Ende von T_3 ohne Blockierung den Knotenwert ändern.

5.1.3 Virtuelle Namensknoten

Die mit dem taDOM2-Sperrprotokoll zur Verfügung stehenden Sperrmodi eignen sich zur Isolation von Transaktionen, die die Funktionalität des DOM Level 2 [DOM2] ausnutzen. Soll jedoch der gesamte Umfang von DOM Level 3 [DOM3] und der Basisoperationen des taDOM-Datenmodells unterstützt werden, so verhält sich das Protokoll zu grobgranular: Sowohl DOM Level 3 (mit der Operation *renameNode()*) als auch die *setValue()*-Operation für taDOM-Bäume erlauben die Umbenennung eines Elements oder Attributs, bei denen es sich um innere

Knoten des Baums handelt. Da die SX-Sperre für eine Modifikation eines Knotens allerdings stets den gesamten darunter liegenden Teilbaum exklusiv sperrt, ist die Sperre für diese Situation zu restriktiv.

Um auch DOM Level 3 und alle Operationen auf dem taDOM-Datenmodell mit einer feingranularen Isolationsstrategie zu unterstützen, werden für Element- und Attributknoten eines taDOM-Baums zusätzliche *virtuelle Namensknoten* eingeführt. Virtuelle Namensknoten werden direkt an alle Element- bzw. Attributknoten angeheftet und mit einer DeweyID adressiert, die aus der ID des entsprechenden Element- bzw. Attributknotens erweitert um eine *Division* mit dem Wert *0* gebildet wird. Abbildung 84 zeigt einen taDOM-Baum mit virtuellen Namensknoten.

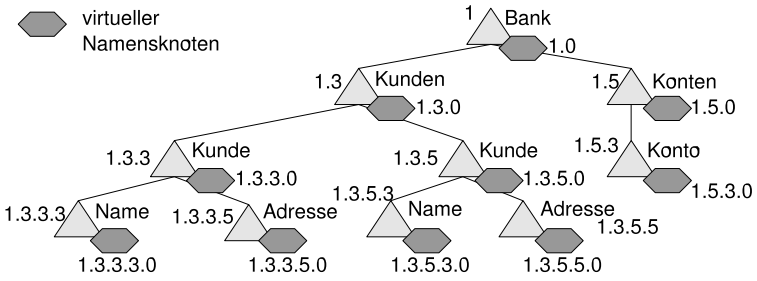
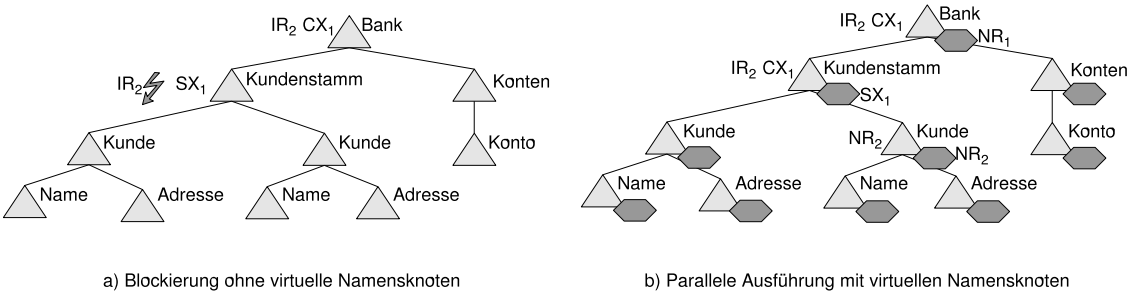


Abbildung 84: taDOM-Baum mit virtuellen Namensknoten

Zur Transaktionsisolation verlangt die Erweiterung des taDOM-Baums mit virtuellen Namensknoten zwei Sperren für jeden Element- bzw. Attributknoten: eine auf dem Element bzw. Attribut selbst und eine auf dem entsprechenden Namensknoten. Damit wird zur Isolation eine Trennung zwischen Struktur und Inhalt des taDOM-Baums erreicht. Zum Lesen bzw. Modifizieren von Knotenwerten sind NR- bzw. SX-Sperren auf den virtuellen Namensknoten erforderlich, wogegen zur Navigation auf der Baumstruktur bzw. Modifikation durch Einfügen und Löschen von Knoten NR- bzw. SX-Sperren auf den betroffenen Knoten selbst angefordert werden müssen. Die damit wesentlich größere Anzahl der zu verwaltenden Sperren lässt sich aber mit einem messbar höheren Transaktionsdurchsatz rechtfertigen (siehe dazu Kapitel 7). Ein Beispiel für den möglichen Parallelitätsgewinn ist in Abbildung 85 dargestellt.



a) Blockierung ohne virtuelle Namensknoten

b) Parallele Ausführung mit virtuellen Namensknoten

Abbildung 85: Parallelitätsgewinn durch virtuelle Namensknoten

Transaktion T_1 navigiert in Abbildung 85a vom Wurzelknoten zum ersten Kind und benennt dieses von *Kunden* in *Kundenstamm* um. Dazu werden zunächst NR-Sperren auf den Knoten *Bank* und *Kunden* angefordert und zur Umbenennung eine Konversion in CX bzw. SX durch-

geführt. Transaktion T_2, die über einen Sekundärindex auf das zweite *Kunde*-Element zugreifen möchte, wird bei der Anforderung der benötigten IR-Sperren bis zum Ende von T_1 blockiert, da IR nicht mit SX kompatibel ist. Diese Blockierung findet bei Anwendung der virtuellen Namensknoten in Abbildung 85b nicht mehr statt. T_1 fordert zur Navigation NR-Sperren auf dem *Bank*- und dem *Kunden*-Element und den entsprechenden virtuellen Namensknoten an. Für die Umbenennung des *Kunden*-Elements ist lediglich eine SX-Sperre auf dessen virtuellem Namensknoten nötig, was zu einer Konversion in CX auf *Bank* und *Kunden* führt. Die Anforderung einer SX-Sperre auf einem virtuellen Namensknoten erfordert, dass zusätzlich auf dem Elternknoten des besitzenden Elements eine CX-Sperre angefordert werden muss, um dort den Modus LR zu blockieren (es dürfen nicht alle Kinder bestimmt werden). T_2 kann nun direkt auf das *Kunde*-Element zugreifen, da die erforderlichen IR-Sperren auf *Bank* und *Kundenstamm* mit CX kompatibel sind. Ein inkonsistenter Zugriff auf *Kundenstamm* wird nach wie vor durch die Inkompatibilität der benötigten NR-Sperre mit der vorhandenen SX-Sperre auf dem virtuellen Namensknoten verhindert.

5.2 taDOM2+

Die in Abschnitt 5.1.1 eingeführten Regeln IX_{NR}, IX_{SR}, CX_{NR} und CX_{SR} sorgen bei einer Konversion durch Sperranforderungen auf den Kindern eines Kontextknotens für ausreichende Transaktionsisolation. In einer konkreten Systemimplementierung ist die Anwendung dieser Regeln jedoch sehr teuer, da der Sperrverwalter eine Sperranforderung nicht mehr unabhängig vom Gesamtsystem mit seinen eigenen Datenstrukturen verarbeiten kann. Die DeweyID zur Adressierung eines Kontextknotens erlaubt zwar die Berechnung aller Vorfahren bis zur Dokumentwurzel, allerdings sind keine Rückschlüsse auf die IDs der Kindknoten möglich. Daher müssen bei Anwendung einer der oben genannten Konversionsregeln alle Kinder eines Kontextknotens durch Navigationsoperationen auf dem betroffenen XML-Dokument bestimmt und schließlich Sperren für die so ermittelten Knotenadressen angefordert werden. Für eine einzige Sperranforderung können somit Konversionen mit den jeweiligen Navigationsoperationen auf mehreren Ebenen des Pfads von der Dokumentwurzel zum Kontextknoten erforderlich sein.

Zur Vermeidung dieses Leistungsengpasses wird für das Sperrprotokoll taDOM2+ für jede der oben genannten Konversionsregeln ein zusätzlicher Sperrmodus eingeführt, um den Sperrverwalter von den Speicherungsstrukturen der verwalteten XML-Dokumente zu entkoppeln:

- Die LRIX-Sperre (*level read intention exclusive*) sperrt den Kontextknoten und alle seine Kinder für den Lesezugriff und zeigt die Änderungsabsicht für einen Nachfahren, der kein Kindknoten ist, im Teilbaum des Kontextknotens an. Die LRIX-Sperre kombiniert die LR- und IX-Sperre und wird bei einer Konversion angefordert, wenn eine LR- oder IX-Sperre bereits vorliegt und eine Transaktion den jeweils anderen Sperrmodus anfordert. Somit entfällt die Regel IX_{NR}. Die LRIX-Sperre erfordert auf dem Elternknoten eine IX-Sperre.
- Die SRIX-Sperre (*subtree read intention exclusive*) sperrt den Kontextknoten und den gesamten Teilbaum aller seiner Nachfahren für den Lesezugriff und indiziert eine Exklusivsperre auf einem Nachfahren, der kein Kindknoten ist. Analog zur LRIX-Sperre ist die SRIX-Sperre eine Kombination der SR- und der IX-Sperre und entspricht der aus der Literatur bekannten RIX-Sperre [HR99]. Damit entfällt die Regel IX_{SR}. Die SRIX-Sperre erfordert auf dem Elternknoten eine IX-Sperre.
- Die LRCX-Sperre (*level read child exclusive*) wird für den Lesezugriff eines Kontextknotens und aller seiner Kinder angefordert, wobei mindestens eines der Kinder exklusiv gesperrt werden soll oder bereits ist. Als Kombination der LR- und CX-Sperre vermeidet die

LRCX-Sperre auf diese Weise die Konversionsregel CX_{NR}. Die LRCX-Sperre erfordert auf dem Elternknoten eine IX-Sperre.

- Die SRCX-Sperre (*subtree read child exclusive*) sperrt den Kontextknoten und alle seine Nachfahren für den Lesezugriff und weist auf mindestens eine Exklusivsperre auf einem Kindknoten hin. Somit entfällt durch die Kombination der SR- und CX-Sperre die Konversionsregel CX_{SR}. Die SRCX-Sperre erfordert auf dem Elternknoten eine IX-Sperre.

Die Kompatibilitäts- und Konversionsmatrix für das Sperrprotokoll taDOM2+ sind in Abbildung 86 aufgeführt. Durch die neuen Sperrmodi entfallen die problematischen Situationen, in denen alle Kinder eines Kontextknotens ermittelt werden müssen, um ausreichende Transaktionsisolation zu gewährleisten. Wie in der Kompatibilitätsmatrix zu erkennen ist, kann für jeden angeforderten Sperrmodus allein aus der vorhandenen Sperre auf dem Kontextknoten der resultierende Sperrmodus bestimmt werden.

	-	IR	NR	LR	SR	IX	LRIX	SRIX	CX	LRCX	SRCX	SU	SX
IR	+	+	+	+	+	+	+	+	+	+	+	-	-
NR	+	+	+	+	+	+	+	+	+	+	+	-	-
LR	+	+	+	+	+	+	+	+	-	-	-	-	-
SR	+	+	+	+	+	-	-	-	-	-	-	-	-
IX	+	+	+	+	-	+	+	-	+	+	-	-	-
LRIX	+	+	+	+	-	+	+	-	-	-	-	-	-
SRIX	+	+	+	+	-	-	-	-	-	-	-	-	-
CX	+	+	+	-	-	+	-	-	+	-	-	-	-
LRCX	+	+	+	-	-	+	-	-	-	-	-	-	-
SRCX	+	+	+	-	-	-	-	-	-	-	-	-	-
SU	+	+	+	+	+	-	-	-	-	-	-	-	-
SX	+	-	-	-	-	-	-	-	-	-	-	-	-

a) Kompatibilitätsmatrix

	-	IR	NR	LR	SR	IX	LRIX	SRIX	CX	LRCX	SRCX	SU	SX
IR	IR	IR	NR	LR	SR	IX	LRIX	SRIX	CX	LRCX	SRCX	SU	SX
NR	NR	NR	NR	LR	SR	IX	LRIX	SRIX	CX	LRCX	SRCX	SU	SX
LR	LR	LR	LR	LR	SR	LRIX	LRIX	SRIX	LRCX	LRCX	SRCX	SU	SX
SR	SR	SR	SR	SR	SR	SRIX	SRIX	SRIX	SRCX	SRCX	SRCX	SR	SX
IX	IX	IX	IX	LRIX	SRIX	IX	LRIX	SRIX	CX	LRCX	SRCX	SX	SX
LRIX	LRIX	LRIX	LRIX	LRIX	SRIX	LRIX	LRIX	SRIX	LRCX	LRCX	SRCX	SX	SX
SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRCX	SRCX	SRCX	SX	SX
CX	CX	CX	CX	LRCX	SRCX	CX	LRCX	SRCX	CX	LRCX	SRCX	SX	SX
LRCX	LRCX	LRCX	LRCX	LRCX	SRCX	LRCX	LRCX	SRCX	LRCX	LRCX	SRCX	SX	SX
SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SX	SX
SU	SU	SU	SU	SU	SU	SX	SX	SX	SX	SX	SX	SU	SX
SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX

b) Konversionsmatrix

Abbildung 86: Sperrprotokoll taDOM2+

Analog zu taDOM2 sind auch beim Sperrprotokoll taDOM2+ für jeden Element- und Attributknoten eines taDOM-Baums zur Unterstützung des DOM Level 3 und aller taDOM-Operationen zusätzlich virtuelle Namensknoten zu verwalten, auf denen NR- bzw. SX-Sperren zum Lesen bzw. zur Modifikation der Knotenwerte anzufordern sind.

5.3 taDOM3

Die Sperrprotokolle taDOM2 und taDOM2+ ermöglichen die feingranulare Transaktionsisolation für die Modifikation innerer Baumknoten durch das Konzept der virtuellen Namensknoten. Dies hat allerdings zur Folge, dass für Element- und Attributknoten beim Zugriff jeweils zwei Sperren (eine auf dem Knoten selbst und eine auf dem virtuellen Namensknoten)

verwaltet werden müssen. Das taDOM3-Protokoll erweitert taDOM2 mit einem Sperrmodus zur Modifikation eines einzelnen Kontextknotens, ohne dass die Nachfolger des Kontextknotens im darunter liegenden Teilbaum von der Sperre für diese Modifikation betroffen sind. Diese Erweiterung bedingt die Ergänzung drei zusätzlicher Modi, sodass gegenüber taDOM2 für das Protokoll taDOM3 vier neue Sperrmodi eingeführt werden:

- Die NU-Sperre (*node update*) sperrt den Kontextknoten für einen Lesezugriff mit der Option, die Sperre nach der Inspektion des Knotens in eine NR-Sperre (*Downgrade*) oder eine NX-Sperre (*Upgrade*) für den anschließenden Schreibzugriff zu konvertieren. Die NU-Sperre erfordert analog zur SU-Sperre eine IR-Sperre auf dem Elternknoten.
- Die NX-Sperre (*node exclusive*) sperrt den Kontextknoten für den exklusiven Schreibzugriff. Im Gegensatz zur SX-Sperre ist der unter dem Kontextknoten liegende Teilbaum von diesem Sperrmodus nicht betroffen. Die NX-Sperre erfordert auf dem Elternknoten eine CX-Sperre.
- Die NRIX-Sperre (*node read intention exclusive*) sperrt einen Kontextknoten für den Lesezugriff und weist auf eine exklusive Sperranforderung auf einem Nachfahren hin, der kein direktes Kind des Kontextknotens ist. Dieser Sperrmodus ist für taDOM3 erforderlich, da das Protokoll taDOM2 eine existierende NR-Sperre bei Anforderung einer IX-Sperre in eine resultierende IX-Sperre konvertiert. Da die NX-Sperre kompatibel mit der IX-Sperre, aber inkompatibel zur NR-Sperre ist, muss stets bekannt sein, ob eine Transaktion einen Knoten bereits explizit gelesen hat oder aufgrund eines direkten Einsprungs in das Dokument eine reine Anwartschaftssperre auf dem Knoten hält. Diese Information geht bei taDOM2 mit der oben beschriebenen Konversion von NR nach IX verloren und muss daher in taDOM3 mit einem speziellen Sperrmodus behandelt werden. Die NRIX-Sperre repräsentiert das Ergebnis einer solchen Konversion. Somit sind NR und NRIX inkompatibel und IX kompatibel zu NX. Die NRIX-Sperre erfordert auf dem Elternknoten eine IX-Sperre.
- Die NRCX-Sperre (*node read child exclusive*) sperrt analog zu NRIX einen Kontextknoten für den Lesezugriff und indiziert die Schreibabsicht auf einem direkten Kindknoten. Auf dem Elternknoten wird für die NRCX-Sperre eine IX-Sperre benötigt.

Die Abbildungen 87 und 88 zeigen die Kompatibilitäts- und Konversionsmatrix mit den 12 Sperrmodi für das taDOM3-Sperrprotokoll. Im Gegensatz zu den Protokollen taDOM2 und taDOM2+ verhalten sich die Sperrmodi IR und NR aufgrund der verschiedenen Kompatibilitäten auch unterschiedlich und sind in einer konkreten Implementierung somit beide erforderlich.

	-	IR	NR	LR	SR	IX	NRIX	CX	NRCX	NU	NX	SU	SX
IR	+	+	+	+	+	+	+	+	+	+	+	-	-
NR	+	+	+	+	+	+	+	+	+	-	-	-	-
LR	+	+	+	+	+	+	+	-	-	-	-	-	-
SR	+	+	+	+	+	-	-	-	-	-	-	-	-
IX	+	+	+	+	-	+	+	+	+	+	+	-	-
NRIX	+	+	+	+	-	+	+	+	+	-	-	-	-
CX	+	+	+	-	-	+	+	+	+	+	+	-	-
NRCX	+	+	+	-	-	+	+	+	+	-	-	-	-
NU	+	+	+	+	+	+	+	+	+	-	-	-	-
NX	+	+	-	-	-	+	-	+	-	-	-	-	-
SU	+	+	+	+	+	-	-	-	-	-	-	-	-
SX	+	-	-	-	-	-	-	-	-	-	-	-	-

Abbildung 87: Kompatibilitätsmatrix des Sperrprotokolls taDOM3

	-	IR	NR	LR	SR	IX	NRIX	CX	NRCX	NU	NX	SU	SX
IR	IR	IR	NR	LR	SR	IX	NRIX	CX	NRCX	NU	NX	SU	SX
NR	NR	NR	NR	LR	SR	NRIX	NRIX	NRCX	NRCX	NR	NX	SU	SX
LR	LR	LR	LR	LR	SR	$NRIX_{NR}$	$NRIX_{NR}$	$NRCX_{NR}$	$NRCX_{NR}$	NU_{NR}	NX_{NR}	SU	SX
SR	SR	SR	SR	SR	SR	$NRIX_{SR}$	$NRIX_{SR}$	$NRCX_{SR}$	$NRCX_{SR}$	NU_{SR}	NX_{SR}	SR	SX
IX	IX	IX	NRIX	$NRIX_{NR}$	$NRIX_{SR}$	IX	NRIX	CX	NRCX	NX	NX	SX	SX
NRIX	NRIX	NRIX	NRIX	$NRIX_{NR}$	$NRIX_{SR}$	NRIX	NRIX	NRCX	NRCX	NX	NX	SX	SX
CX	CX	CX	NRCX	$NRCX_{NR}$	$NRCX_{SR}$	CX	NRCX	CX	NRCX	NX	NX	SX	SX
NRCX	NRCX	NRCX	NRCX	$NRCX_{NR}$	$NRCX_{SR}$	NRCX	NRCX	NRCX	NRCX	NX	NX	SX	SX
NU	NU	NU	NU	NU_{NR}	NU_{SR}	NX	NX	NX	NX	NU	NX	SU	SX
NX	NX	NX	NX	NX_{NR}	NX_{SR}	NX	NX	NX	NX	NX	NX	SX	SX
SU	SU	SU	SU	SU	SU	SX	SX	SX	SX	SU	SX	SU	SX
SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX

Abbildung 88: Konversionsmatrix des Sperrprotokolls taDOM3

5.4 taDOM3+

In ähnlicher Weise wie taDOM2 enthält auch das taDOM3-Protokoll die indizierten Konversionsregeln $NRIX_{NR}$, $NRCX_{NR}$, $NRIX_{SR}$, $NRCX_{SR}$, NU_{NR}, NU_{SR}, NX_{NR} und NX_{SR}, die den Zugriff auf einen taDOM-Baum zur Ermittlung aller Kinder eines Kontextknotens erfordern. Analog zur Optimierung in taDOM2+ führt das Protokoll taDOM3+ weitere Sperrmodi ein, um den Zugriff auf die gespeicherte Repräsentation des taDOM-Baums während der Sperrkonversion zu vermeiden.

- Die LRIX-Sperre (*level read intention exclusive*) sperrt den Kontextknoten und alle seiner Kinder für den Lesezugriff und weist auf die Änderungsabsicht mindestens eines Nachfahren hin, der kein Kind des Kontextknotens ist. Die LRIX-Sperre erfordert auf dem Elternknoten eine IX-Sperre.
- Die SRIX-Sperre (*subtree read intention exclusive*) sperrt den Kontextknoten und alle seine Nachfahren für den Lesezugriff und mindestens einen Nachfahren, der kein Kind des Kontextknotens ist, für den exklusiven Schreibzugriff. Die SRIX-Sperre benötigt auf dem Elternknoten eine IX-Sperre.
- Die LRCX-Sperre (*level read child exclusive*) sperrt den Kontextknoten und alle Kindknoten für den Lesezugriff und ergänzend mindestens eines der Kinder für den exklusiven Schreibzugriff mit einer NX- oder SX-Sperre. Die LRCX-Sperre erfordert auf dem Elternknoten eine IX-Sperre.
- Die SRCX-Sperre (*subtree read child exclusive*) sperrt den Kontextknoten und alle Nachfahren im darunter liegenden Teilbaum für den Lesezugriff und mindestens ein direktes Kind des Kontextknotens für den Schreibzugriff. Die SRCX-Sperre erfordert auf dem Elternknoten eine IX-Sperre.
- Die LRNU-Sperre (*level read node update*) sperrt alle Kinder des Kontextknotens für den Lesezugriff und den Kontextknoten selbst für den Lesezugriff mit Änderungsabsicht. Nach der Inspektion des Kontextknotens kann die LRNU-Sperre in eine LR-Sperre (*Downgrade*) oder eine LRNX-Sperre (*Upgrade*) konvertiert werden. Die LRNU-Sperre erfordert auf dem Elternknoten eine IR-Sperre.
- Die SRNU-Sperre (*subtree read node update*) sperrt alle Nachfahren des Kontextknotens im darunter liegenden Teilbaum für den Lesezugriff und den Kontextknoten selbst für den Lesezugriff mit Änderungsabsicht. Nach der Inspektion des Kontextknotens kann die SRNU-

Sperre in eine SR-Sperre (*Downgrade*) oder eine SRNX-Sperre (*Upgrade*) konvertiert werden. Die SRNU-Sperre erfordert auf dem Elternknoten eine IR-Sperre.

- Die LRNX-Sperre (*level read node exclusive*) sperrt den Kontextknoten für den exklusiven Schreibzugriff und alle seine Kindknoten für den Lesezugriff. Die LRNX-Sperre benötigt auf dem Elternknoten eine CX-Sperre.
- Die SRNX-Sperre (*subtree read node exclusive*) sperrt den Kontextknoten für den exklusiven Schreibzugriff und alle seine Nachfahren im darunter liegenden Teilbaum für den Lesezugriff. Die SRNX-Sperre benötigt auf dem Elternknoten eine CX-Sperre.

Die Kompatibilitäts- und Konversionsmatrix für das Sperrprotokoll taDOM3+ zeigt Abbildung 89. Dieses Protokoll unterstützt mit einer feingranularen Transaktionsisolation die gesamte Funktionalität des DOM Level 3 und alle Operationen, die in Abschnitt 4.2 für taDOM-Bäume definiert wurden.

	-	IR	NR	LR	SR	IX	NRIX	LRIX	SRIX	CX	NRCX	LRCX	SRCX	NU	LRNU	SRNU	NX	LRNX	SRNX	SU	SX
IR	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-
NR	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-
LR	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-
SR	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
IX	+	+	+	+	-	+	+	+	-	+	+	+	-	+	+	-	+	+	-	-	-
NRIX	+	+	+	+	-	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	-
LRIX	+	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-
SRIX	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CX	+	+	+	-	-	+	+	-	-	+	+	-	-	+	-	-	+	-	-	-	-
NRCX	+	+	+	-	-	+	+	-	-	+	+	-	-	-	-	-	-	-	-	-	-
LRCX	+	+	+	-	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SRCX	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NU	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-
LRNU	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-
SRNU	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NX	+	+	-	-	-	+	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-
LRNX	+	+	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SRNX	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SU	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SX	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

a) Kompatibilitätsmatrix

	-	IR	NR	LR	SR	IX	NRIX	LRIX	SRIX	CX	NRCX	LRCX	SRCX	NU	LRNU	SRNU	NX	LRNX	SRNX	SU	SX
IR	IR	IR	NR	LR	SR	IX	NRIX	LRIX	SRIX	CX	NRCX	LRCX	SRCX	NU	LRNU	SRNU	NX	LRNX	SRNX	SU	SX
NR	NR	NR	NR	LR	SR	NRIX	NRIX	LRIX	SRIX	NRCX	NRCX	LRCX	SRCX	NR	LR	SR	NX	LRNX	SRNX	SU	SX
LR	LR	LR	LR	LR	SR	LRIX	LRIX	LRIX	SRIX	LRCX	LRCX	LRCX	SRCX	LRNU	LRNU	SRNU	LRNX	LRNX	SRNX	SU	SX
SR	SR	SR	SR	SR	SR	SRIX	SRIX	SRIX	SRIX	SRCX	SRCX	SRCX	SRCX	SRNU	SRNU	SRNU	SRNX	SRNX	SRNX	SR	SX
IX	IX	IX	NRIX	LRIX	SRIX	IX	NRIX	LRIX	SRIX	CX	NRCX	LRCX	SRCX	NX	LRNX	SRNX	NX	LRNX	SRNX	SX	SX
NRIX	NRIX	NRIX	NRIX	LRIX	SRIX	NRIX	NRIX	LRIX	SRIX	NRCX	NRCX	LRCX	SRCX	NX	LRNX	SRNX	NX	LRNX	SRNX	SX	SX
LRIX	LRIX	LRIX	LRIX	LRIX	SRIX	LRIX	LRIX	LRIX	SRIX	LRCX	LRCX	LRCX	SRCX	LRNX	LRNX	SRNX	LRNX	LRNX	SRNX	SX	SX
SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRCX	SRCX	SRCX	SRCX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SX	SX
CX	CX	CX	NRCX	LRCX	SRCX	CX	NRCX	LRCX	SRCX	CX	NRCX	LRCX	SRCX	NX	LRNX	SRNX	NX	LRNX	SRNX	SX	SX
NRCX	NRCX	NRCX	NRCX	LRCX	SRCX	NRCX	NRCX	LRCX	SRCX	NRCX	NRCX	LRCX	SRCX	NX	LRNX	SRNX	NX	LRNX	SRNX	SX	SX
LRCX	LRCX	LRCX	LRCX	LRCX	SRCX	LRCX	LRCX	LRCX	SRCX	LRCX	LRCX	LRCX	SRCX	LRNX	LRNX	SRNX	LRNX	LRNX	SRNX	SX	SX
SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SX	SX
NU	NU	NU	NU	LRNU	SRNU	NX	NX	LRNX	SRNX	NX	NX	LRNX	SRNX	NU	LRNU	SRNU	NX	LRNX	SRNX	SU	SX
LRNU	LRNU	LRNU	LRNU	LRNU	SRNU	LRNX	LRNX	LRNX	SRNX	LRNX	LRNX	LRNX	SRNX	LRNU	LRNU	SRNU	LRNX	LRNX	SRNX	SU	SX
SRNU	SRNU	SRNU	SRNU	SRNU	SRNU	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNU	SRNU	SRNU	SRNX	SRNX	SRNX	SU	SX
NX	NX	NX	NX	LRNX	SRNX	NX	NX	LRNX	SRNX	NX	NX	LRNX	SRNX	NX	LRNX	SRNX	NX	LRNX	SRNX	SX	SX
LRNX	LRNX	LRNX	LRNX	LRNX	SRNX	LRNX	LRNX	LRNX	SRNX	LRNX	LRNX	LRNX	SRNX	LRNX	LRNX	SRNX	LRNX	LRNX	SRNX	SX	SX
SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SX	SX
SU	SU	SU	SU	SU	SU	SX	SX	SX	SX	SX	SX	SX	SX	SU	SU	SU	SX	SX	SX	SU	SX
SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX

b) Konversionsmatrix

Abbildung 89: Sperrprotokoll taDOM3+

5.5 Zusammenfassung der taDOM-Sperrmodi

Die Tabelle in der folgenden Abbildung 90 fasst alle Sperrmodi der taDOM-Sperrprotokolle in einer Übersicht zusammen. Für eine Sperre auf dem Kontextknoten *k* sind der anzufordernde Sperrmodus auf dem Elternknoten und die durch die Sperre geschützten Knoten für den Lese- bzw. Schreibzugriff dargestellt.

Sperrmodus auf *k*	Sperrmodus auf dem Elternknoten	Für den Lesezugriff gesperrt	Für den Schreibzugriff gesperrt
IR	IR	mindestens ein Nachfahre von *k*	
NR	IR	Kontextknoten *k*	
LR	IR	Kontextknoten *k* und alle Kindknoten	
SR	IR	Kontextknoten *k* und alle Nachfahren	
IX	IX		Nachfahre von *k*, der kein Kind von *k* ist
NRIX	IX	Kontextknoten *k*	Nachfahre von *k*, der kein Kind von *k* ist
LRIX	IX	Kontextknoten *k* und alle Kindknoten	Nachfahre von *k*, der kein Kind von *k* ist
SRIX	IX	Kontextknoten *k* und alle Nachfahren	Nachfahre von *k*, der kein Kind von *k* ist
CX	IX		ein Kindknoten von *k*
NRCX	IX	Kontextknoten *k*	ein Kindknoten von *k*
LRCX	IX	Kontextknoten *k* und alle Kindknoten	ein Kindknoten von *k*
SRCX	IX	Kontextknoten *k* und alle Nachfahren	ein Kindknoten von *k*
NU	IR	*k* mit Option zum *Upgrade*(NX)/*Downgrade*(NR)	
LRNU	IR	*k* mit Option zum *Upgrade*(NX)/*Downgrade*(NR), alle Kindknoten von *k*	
SRNU	IR	*k* mit Option zum *Upgrade*(NX)/*Downgrade*(NR), alle Nachfahren von *k*	
NX	CX		Kontextknoten *k*
LRNX	CX	alle Kindknoten von *k*	Kontextknoten *k*
SRNX	CX	alle Nachfahren von *k*	Kontextknoten *k*
SU	IR	Kontextknoten *k* und alle Nachfahren mit Option zum *Upgrade*(SX)/*Downgrade*(SR)	
SX	CX		Kontextknoten *k* und alle Nachfahren

Abbildung 90: Zusammenfassung der taDOM-Sperrmodi

5.6 Vollständigkeit und Korrektheit der Sperrprotokolle

Für die zunehmende Komplexität der Kompatibilitäts- und Konversionsmatrizen der Sperrprotokolle und das Zusammenspiel der Knoten- und Kantensperren gilt es, die Vollständigkeit und Korrektheit der taDOM-Protokolle bzgl. des taDOM-Datenmodells und der dafür definierten Operationen zu beweisen.

Vollständigkeit

Die Vollständigkeit der Protokolle ist einfach nachzuweisen, da hierfür jedes der Protokolle taDOM2, taDOM2+, taDOM3 und taDOM3+ für jede taDOM-Operation ausreichende Sperrmodi zu deren Isolation gegen die Operationen parallel ablaufender Transaktionen zur Verfügung stellen muss. Aus der im nächsten Abschnitt 5.6.1 referenzierten Operationsübersicht im Anhang B kann entnommen werden, dass dies für jede Operation der Fall ist. Somit ist die Vollständigkeit der Sperrprotokolle bzgl. der Operationen des taDOM-Datenmodells gegeben.

Korrektheit

Das *Korrektheitskriterium der Serialisierbarkeit* [EGL+97] verlangt, dass die parallele Ausführung von Operationen verschiedener Transaktionen äquivalent zu der seriellen Ausführung der Transaktionen ist. Wird das Korrektheitskriterium erfüllt, so heißt die Operationenfolge *serialisierbar*.

Für den Beweis der Korrektheit der Sperrprotokolle gehen wir von einer serialisierbaren Operationsfolge der Transaktionen T_i (i=1,...,n) aus. Für jede weitere in einer Transaktion T_j ausgeführte Operation o_j,, die einen Schreib-/Lese-, Lese-/Schreib- oder Schreib-/Schreib-Konflikt zu einer bereits ausgeführten Operation o_i einer der laufenden Transaktionen T_i auslöst, muss zur Erhaltung der Serialisierbarkeit gelten:

- Das Sperrprotokoll muss die Transaktion T_j aufgrund ihrer Sperranforderungen bis zum Ende von T_i blockieren. Dies betrifft die Korrektheit der Kompatibilitätsmatrix.
- Sperren, die für o_i während der Verarbeitung angefordert wurden, dürfen bei nachträglichen Sperrkonversionen nur durch restriktivere Sperren ersetzt werden, sodass T_j auch nach der Konversion weiterhin bis zum Ende von T_i blockiert wird. Dies betrifft die Korrektheit der Konversionsmatrix.

Werden diese beiden Regeln eingehalten, so wird stets eine Ausführungsreihenfolge der Operationen gewährleistet, die den gleichen DB-Zustand und die gleichen Ausgabewerte wie die serielle Ausführung der Transaktionen liefert.

5.6.1 Korrektheit der Kompatibilitätsmatrizen

Zum Nachweis der Korrektheit der Kompatibilitätsmatrizen untersuchen wir zunächst den allgemeinen Fall der Sperranforderung auf einem Kontextknoten. Für das Eintragen einer Sperre sind zunächst von der Wurzel des taDOM-Baums bis zum Elternknoten des Kontextknotens gemäß den hierarchischen taDOM-Protokollen die entsprechenden Sperren anzufordern. Betrachtet man die Diversität der Sperrmodi, so ist der „schlimmste“ Fall eine NX- bzw. SX-Sperre auf dem Kontextknoten und eine CX-Sperre auf dessen Elternknoten. Auf allen weiteren Vorfahren bis zur Baumwurzel ist eine IX-Sperre anzufordern. Aus diesem Grund ist Abbildung 91 zur Darstellung einer Sperrsituation auf dem Kontextknoten und allen relevanten ihn umgebenden Knoten ausreichend. Wir bezeichnen diese Darstellung als die *Kontextumgebung*.

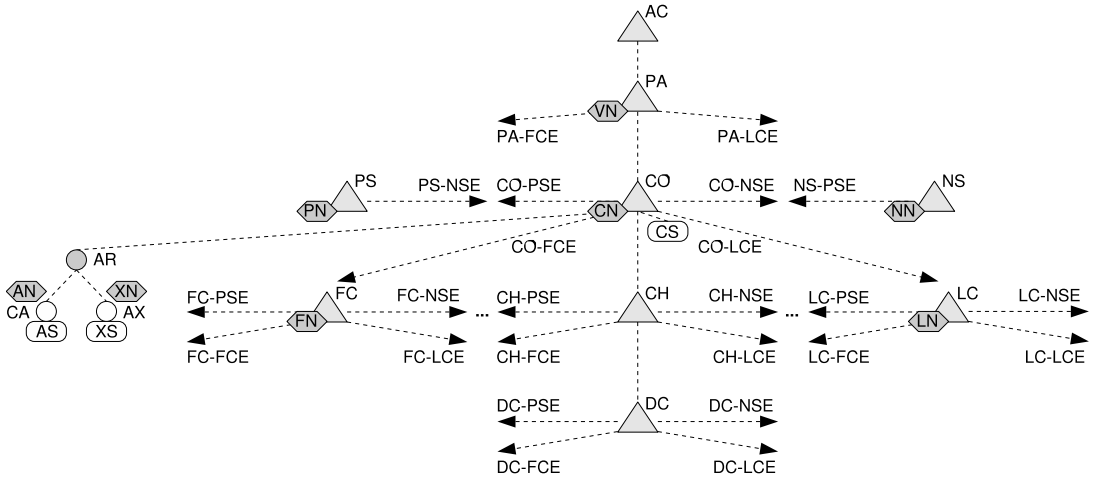


Abbildung 91: Kontextknoten und umgebende taDOM-Knoten

Eine Sperranforderung auf dem Kontextknoten CO kann höchstens eine CX-Sperre auf dem Elternknoten PA und IX-Sperren auf allen Vorfahren AC, die keine Elternknoten sind, nach sich ziehen. Sperranforderungen auf PA bzw. einem Vorfahrenknoten AC haben auf wiederum

deren Vorfahren nur die Sperrmodi IR, IX oder CX zur Folge. Da jedoch die durch PA bzw. AC bedingten IR-, IX- und CX-Sperren mit allen durch CO bedingten IX- oder IR-Sperren kompatibel sind, müssen die Sperren auf den Vorfahren von AC nicht weiter betrachtet werden, da sie an der Kompatibilität bzw. Inkompatibilität einer Sperranforderung nichts ändern: Ist eine Sperranforderung innerhalb der in Abbildung 91 gezeigten Umgebung kompatibel, so bleibt sie kompatibel, ist die Sperranforderung inkompatibel, so ändern die kompatiblen IR-, IX- und CX-Sperren ebenfalls nichts an dieser Inkompatibilität.

Für die Umgebung des Kontextknotens werden weiterhin der vorherige und nächste Geschwisterknoten PS und NS, das erste, letzte und ein beliebiges Kind FC, LC und CH und ein beliebiger Nachfahre DC, der kein Kindknoten ist, definiert. Weiterhin ist die Attributwurzel AR, das Attribut CA im Kontext einer attributbezogenen taDOM-Operation und ein weiteres beliebiges Attribut AX relevant, wenn es sich beim Kontextknoten um ein Element handelt. AS und XS stellen die String-Knoten der entsprechenden Attribute dar. Handelt es sich beim Kontextknoten um einen Textknoten, so ist dessen Wert in dem String-Knoten CS gespeichert. Die Knoten VN, CN, PN, NN, FN, LN, AN und XN stellen die virtuellen Namensknoten dar, die für die Protokolle taDOM2 und taDOM2+ von Bedeutung sind.

Für die Betrachtung der Sperren auf den virtuellen Navigationskanten werden die *prevSibling*-, *nextSibling*-, *firstChild*- und *lastChild*-Kante PSE, NSE, FCE und LCE eingeführt, die entsprechend des besitzenden Knotens benannt werden. So heißt beispielsweise die *prevSibling*-Kante des Kontextknotens CO-PSE, die *lastChild*-Kante des ersten Kindknotens FC-LCE usw.

Read- und Write-Sets durch die Semantik der taDOM-Operationen

Für jede der 19 in Abschnitt 4.2 definierten taDOM-Operationen wird mit Hilfe von *Basisoperationen* die Semantik der Operation in so genannten *Use Cases* beschrieben. Dafür werden insgesamt 57 Basisoperationen benötigt, die beispielsweise das Auslesen oder Ändern von Knotenwerten oder das Traversieren oder Umbiegen von Navigationskanten beschreiben. Eine Liste mit der Beschreibung aller Basisoperationen ist in Abschnitt B.1 im Anhang B zu finden.

Ein Use Case für *getFirstChild()* ist zum Beispiel die Ausführung dieser Operation, die als Resultat das erste Kind des Kontextknotens liefert. Dazu wird zunächst der Kontextknoten selbst gelesen (Basisoperation *readCO*), danach über die *firstChild*-Kante zum ersten Kindknoten navigiert und dieser gelesen (Basisoperationen *useCO-FCE* und *readFC*). Da wie in Abschnitt 5.1.2 erläutert, damit auch bekannt ist, dass die *prevSibling*-Kante des ersten Kindknotens keinen weiteren Knoten mehr adressieren kann, muss diese Kenntnis mit der Basisoperation *useFC-PSE* beschrieben werden. Ein weiterer Use Case für die Operation *getFirstChild()* ist die Ausführung mit einem „leeren“ Ergebnis, d. h., der Kontextknoten besitzt kein erstes Kind. In diesem Fall erhält die ausführende Transaktion auch das Wissen, dass der Kontextknoten überhaupt keine Kinder besitzt und somit auch kein letztes Kind besitzen kann. Dieser Use Case ist mit den Basisoperationen *readCO*, *useCO-FCE* und *useCO-LCE* zu beschreiben.

Für jeden Use Case werden zusätzlich alle Knoten der Kontextumgebung angegeben, auf denen die taDOM-Operation des Use Case ausgeführt werden kann, und welche Knoten in der Kontextumgebung für diesen Use Case vorhanden sind. Die Operation *getFirstChild()* kann z. B. nur auf Elementknoten, nicht aber auf einer Attributwurzel, einem Attribut oder virtuellen Namensknoten ausgeführt werden. Der erste, letzte, ein beliebiger Kindknoten oder Nachfolger FC, LC, CH oder DC des Kontextknotens sind nur vorhanden, wenn die Operation einen Kindknoten und kein „leeres“ Ergebnis zurückliefert.

Mit Hilfe dieser semantischen Beschreibung durch die Basisoperationen kann nun für jeden Use Case, der auf einem Knoten der Kontextumgebung ausgeführt wird, das Read- bzw. Write-

Set ermittelt werden, in denen alle gelesenen bzw. modifizierten Knoten und Kanten vermerkt sind. Wird der Use Case der *getFirstChild()*-Operation auf dem Kontextknoten CO ausgeführt, so enthält sein Read-Set CO, CO-FCE, FC und FC-PSE. Erfolgt die Ausführung des Use Case dagegen auf dem letzten Kindknoten LC des Kontextknotens, so müssen der Kontext der Basisoperationen für eine Ausführung auf dem letzten Kind angepasst werden, und als Read-Set ergibt sich LC, LC-FCE, DC und DC-PSE, da der erste Kindknoten des letzten Kindes LC in der Kontextumgebung einen Nachfahren DC des Kontextknotens darstellt.

Prüfung der Korrektheit

Neben der semantischen Beschreibung mit den Basisoperationen enthält ein Use Case zusätzlich für jedes der vier Sperrprotokolle taDOM2, taDOM2+, taDOM3 und taDOM3+ ein *Szenario*. Das Szenario beschreibt die Sperren, die für die Ausführung der taDOM-Operation des Use Case bei Anwendung des entsprechenden Sperrprotokolls anfordert werden. In einem konkreten System müssen diese Sperranforderungen mit der Implementierung der taDOM-Operationen übereinstimmen. Für das taDOM2-Protokoll bspw. sind für den Use Case der Operation *getFirstChild()* (mit einem existierenden ersten Kindknoten) auf dem Kontextknoten, dem ersten Kindknoten und deren virtuellen Namensknoten jeweils eine NR-Sperre und auf der *firstChild*-Kante des Kontextknotens und der *prevSibling*-Kante des ersten Kindknotens eine ER-Sperre anzufordern. Analog zu den Basisoperationen können die Sperranforderungen bei Ausführung des Use Case auf einem von CO abweichenden Knoten in der Kontextumgebung umgerechnet werden. Die folgende Abbildung 92 zeigt den vollständigen Use Case mit den Szenarien für taDOM2, taDOM2+, taDOM3 und taDOM3+ für die Operation *getFirstChild()*, die das existierende erste Kind des Kontextknotens zurückliefert. Eine Auflistung aller 36 Use Cases für die Spezifikation der taDOM-Operationen ist im Abschnitt B.2 im Anhang B nachzulesen.

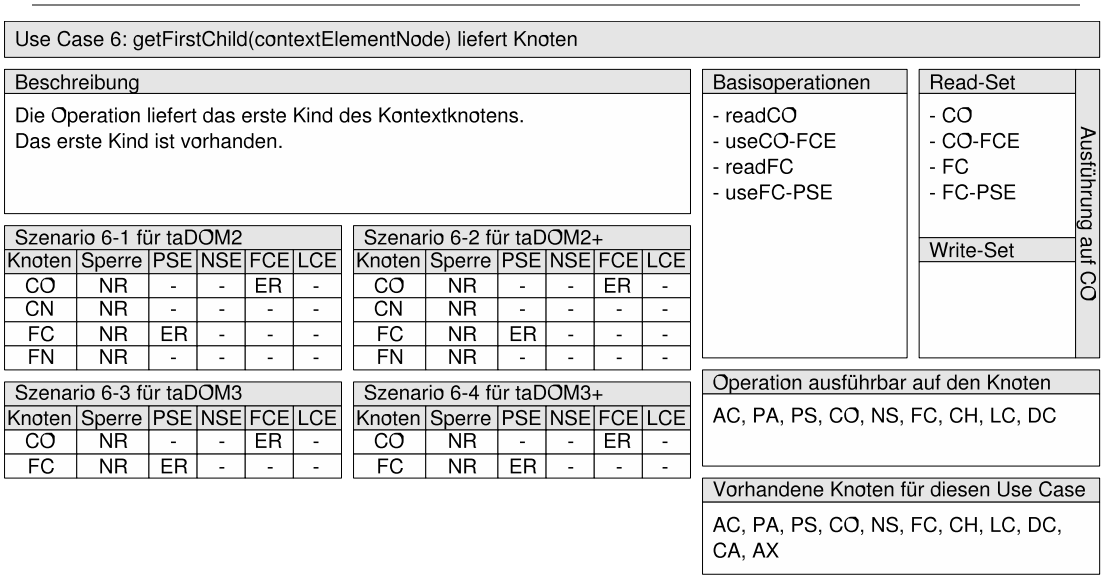


Abbildung 92: Use Case für die *getFirstChild()*-Operation

Für die Korrektheit der Kompatibilitätsmatrizen wird nun für jede mögliche Ausführungssituation der taDOM-Operationen innerhalb der Kontextumgebung überprüft, ob das jeweilige Sperrprotokoll die Serialisierbarkeit für die mit den Basisoperationen beschriebene Semantik garantiert. Dazu betrachten wir als *erste Operation* die taDOM-Operation jedes Use Case auf

dem Kontextknoten und führen als *zweite Operation* jede Operation der 36 Use Cases nacheinander auf jedem der für ihn möglichen Knoten in der Kontextumgebung aus. Solch eine Ausführung von erster und zweiter Operation bezeichnen wir als *Konstellation*. Für jede Konstellation werden nun mit den semantischen Beschreibungen die Read- und Write-Sets der beiden Operationen für die Knoten, auf denen sie ausgeführt werden, berechnet. Liegt zwischen den Operationen ein Lese-/Schreib-, Schreib-/Lese- oder Schreib-/Schreib-Konflikt vor, so muss zur Erhaltung der Serialisierbarkeit die zweite Operation bei der Anforderung einer Sperre blockiert werden. Dazu muss mindestens für eine der Sperren, welche die Operation in der Kontextumgebung einträgt, ein Konflikt zu einer bereits vorhandenen Sperre der ersten Operation auftreten. Liegt kein Konflikt zwischen den Read- und Write-Sets vor, so darf für eine maximale Parallelität der Transaktionsverarbeitung keine Sperrinkompatibilität auftreten, d. h., alle in der Kontextumgebung eingetragenen Sperren der Operationen müssen miteinander kompatibel sein.

Die Ausführung aller Use Cases der 19 taDOM-Operationen auf allen Knoten der Kontextumgebung für jedes der vier Sperrprotokolle führt zu über 38.000 Konstellationen und insgesamt über 250.000 einzeln zu überprüfenden Sperrkompatibilitäten, sodass dieses Ergebnis unmöglich in den Anhang dieser Arbeit aufgenommen werden kann. Ein generierter HTML-Bericht, der für den Beweis der Korrektheit der Kompatibilitätsmatrizen alle überprüften Konstellationen und Sperrsituationen enthält, kann auf der XTC-Projektseite der Arbeitsgruppe[5] eingesehen werden.

5.6.2 Korrektheit der Konversionsmatrizen

Zum Beweis der Korrektheit der Konversionsmatrizen definieren wir zunächst die *Stärke-Beziehung* für zwei Sperrmodi.

Ein Sperrmodus l_1 ist *stärker* als ein Sperrmodus l_2 ($l_1 > l_2$), wenn jeder weitere Sperrmodus l_i, der nicht mit l_2 kompatibel ist, auch nicht mit l_1 kompatibel ist. Das bedeutet, dass die stärkere Sperre l_1 zu weniger Sperrmodi kompatibel ist und alle Sperranforderungen blockiert, die auch l_2 blockiert. Wenn l_1 nicht stärker als l_2 ist, dann ist l_1 nur *schwächer* als l_2 ($l_1 < l_2$), wenn l_2 stärker als l_1 ist. Das heißt, es gibt auch Sperrmodi, die weder stärker noch schwächer als andere Sperrmodi sind (z. B. SR und CX, da IX inkompatibel zu SR, aber nicht zu CX ist und LR inkompatibel zu CX, aber nicht zu SR ist).

Die Stärke-Beziehung der Sperrmodi kann für die Garantie der Serialisierbarkeit einer verzahnten Transaktionsausführung herangezogen werden. Wird eine Operation o_2 in Transaktion T_2 wegen einer Sperre l_1 von Operation o_1 in Transaktion T_1 blockiert und somit erst nach dem Ende von T_1 ausgeführt, so wird o_2 auch durch das Ersetzen von l_1 durch eine stärkere Sperre l_2 während einer Konversion erst nach dem Ende von T_1 ausgeführt.

Für die Korrektheit einer Konversionsmatrix gilt, wenn ein vorhandener Sperrmodus l_1 bei Anforderung eines weiteren Sperrmodus l_2 durch den Modus l_3 ersetzt wird, dann bleibt die Serialisierbarkeit der Transaktionen erhalten, wenn $l_3 > l_1$ und $l_3 > l_2$ gilt. Damit blockiert der resultierende Modus l_3 mindestens alle Sperranforderungen, die auch vom ursprünglichen Modus l_1 und dem neu angeforderten Modus l_2 blockiert werden. Als Beispiel betrachten wir dazu die Kompatibilitäts- und Konversionsmatrix des taDOM2-Sperrprotokolls in Abbildung 78. Der Sperrmodus CX ist stärker als IX, da beide inkompatibel zu SR, SU und SX sind, und CX zusätzlich nicht mit LR verträglich ist. Somit ist die Konversion einer IX- in eine CX-Sperre oder die Erhaltung einer CX-Sperre bei Anforderung von IX erlaubt.

5 http://wwwdvs.informatik.uni-kl.de/agdbis/projects/xtc/

Mit dem Kriterium der Stärke-Beziehung können die Konversionsmatrizen jedoch nicht vollständig als korrekt nachgewiesen werden, da die Konversion der Update-Sperren beim *Downgrade* in einen schwächeren Modus und die indizierten Konversionsregeln (z. B. IX_{NR}) bei den Protokollen taDOM2 und taDOM3 das Kriterium verletzten. Daher sind für diese beiden Ausnahmen zusätzliche Bedingungen für die Korrektheit zu überprüfen.

Korrektheit eines Downgrade

Wird ein vorhandener Sperrmodus l_1 bei Anforderung eines weiteren Sperrmodus l_2 in einen Update-Sperrmodus l_u (NU, LRNU, SRNU oder SU) konvertiert, so muss für alle Sperrmodi l_i, in welche der Update-Modus l_u konvertiert werden kann, $l_i > l_1$ und $l_i > l_2$ gelten. Somit ist es zwar erlaubt, dass im Zuge eines *Downgrade* eine Update-Sperre wieder in eine schwächere Sperre „zurückkonvertiert“ werden kann, allerdings wird garantiert, dass dieser schwächere Sperrmodus stets stärker als die beiden Sperrmodi ist, aus denen die Update-Sperre ursprünglich entstanden ist.

Betrachten wir dazu wieder die Matrizen in Abbildung 78. Der Update-Modus SU entsteht aus der Anforderung einer IR-, NR-, LR-, SU- oder SR-Sperre bei einer bereits vorhandenen SU-Sperre oder der Anforderung des SU-Modus bei einer vorhandenen IR-, NR-, LR- oder SR-Sperre. SU ist stärker als alle diese Sperrmodi. Die Konversion einer SU-Sperre in eine nichtstärkere SR-Sperre im Zuge eines *Downgrade* ist ebenfalls erlaubt, da SR stärker ist als alle Sperrmodi IR, NR, LR und SR, aus denen SU hervorgegangen ist.

Korrektheit einer indizierten Konversion

Der Korrektheitsbeweis für eine indizierte Konversionsregel ist sehr kompliziert und wird daher schon zu Beginn der Argumentation mit dem begleitenden Beispiel der Regel CX_{NR} des Protokolls taDOM2 für die Anforderung einer CX-Sperre bei einer vorhandenen LR-Sperre erläutert.

Die Anwendung einer indizierten Konversionsregel a_b ersetzt die vorhandene Sperre auf dem Kontextknoten durch den Modus a und fordert auf allen Kindern des Kontextknotens den Sperrmodus b an. Für einen vorhandenen Sperrmodus l_1 auf dem Kontextknoten und einen weiteren angeforderten Modus l_2 werden zunächst analog zur Stärke-Beziehung die Kompatibilitäten zu allen möglichen Sperrmodi l_i (i=1,...,n) des zu untersuchenden Protokolls überprüft. Für das Protokoll taDOM2 sind dies die Modi IR, NR, LR, SR, IX, CX, SU und SX.

Ist die vorhandene Sperre l_1 und die angeforderte Sperre l_2 kompatibel zu l_i, so muss auch die resultierende Sperre a der Konversionsregel auf dem Kontextknoten zu l_i kompatibel sein, um eine maximale Parallelität der Transaktionsverarbeitung zu erreichen. In unserem Beispiel gilt dies für die Sperrmodi l_i aus {IR, NR, IX} des taDOM2-Protokolls, die alle zur vorhandenen Sperre LR und zur angeforderten Sperre CX kompatibel sind. Nach der Konversion liegt eine CX-Sperre auf dem Kontextknoten vor, die ebenfalls kompatibel zu IR, NR und IX ist.

Ist dagegen die vorhandene Sperre l_1 oder die angeforderte Sperre l_2 zu l_i inkompatibel, so muss auch die resultierende Sperre a auf dem Kontextknoten zu l_i inkompatibel sein, um dieselbe Restriktivität wie der vorhandene und der angeforderte Sperrmodus zu garantieren. Im Beispiel für das Protokoll taDOM2 gilt dies für die Sperren l_i aus {LR, SU}, die beide zur vorhandenen Sperre LR, aber nicht zum angeforderten CX kompatibel sind. Die durch die Konversion entstehende CX-Sperre auf dem Kontextknoten erhält diese Inkompatibilität. Die SX-Sperre ist sowohl zum vorhandenen bzw. angeforderten Modus LR bzw. CX als auch zur resultierenden CX-Sperre inkompatibel, womit auch diese Konversion zulässig ist. Für die letzte noch nicht betrachtete Sperre CX des taDOM2-Sperrprotokolls gilt die zu überprüfende Bedingung jedoch nicht: l_i = CX ist zu einer vorhandenen LR-Sperre inkompatibel und zu einer angeforderten CX-Sperre kompatibel. Zu der resultierenden CX-Sperre der Konversionsregel

ist l_i allerdings auch kompatibel, was jedoch aufgrund der Inkompatibilität zu LR nicht sein dürfte. Diese Konversion ist jedoch trotzdem zulässig, wenn für solche Fälle die folgenden Überprüfungen erfolgreich sind.

Wenn eine vorhandene Sperre l_1 für eine angeforderte Sperre l_2 auf dem Kontextknoten während einer Konversion durch eine Sperre a ersetzt wird, die nicht stärker als l_1 oder als l_2 ist, dann ist dies zulässig, wenn die Konversion weitere Sperranforderungen auf den Kindern des Kontextknotens anstößt und die folgenden Zusammenhänge nachgewiesen werden können. Zunächst werden aus der Konversionsmatrix des zu untersuchenden Sperrprotokolls alle Kombinationen von Sperren (a_1, a_2) bestimmt, bei deren Existenz bzw. Anforderung eine Konversion in den resultierenden Modus a erfolgt. Diese Sperrkombinationen stellen somit alle Möglichkeiten dar, durch die der Modus a auf dem Kontextknoten „entstehen“ kann. Für jede Kombination wird nun jeweils für die Sperren a_1 und a_2 untersucht, welche Sperranforderungen auf einem Kind des Kontextknotens diese Sperren bedingen. Für mindestens einer der beiden Sperren a_1 bzw. a_2 muss nun gelten, dass auf einem Kindknoten alle Sperranforderungen, die a_1 bzw. a_2 auf dem Kontextknoten erfordern, inkompatibel zum Sperrmodus b sind, der für die Anwendung der Konversionsregel auf jedem Kind des Kontextknotens eingetragen wird. Ist dies der Fall, so blockieren diese Inkompatibilitäten auf der Ebene der Kindknoten die vollständige Ausführung jeder möglichen Sperranforderung, die auf dem Kontextknoten zum Sperrmodus a führt. Das bedeutet, dass der nicht-stärkere Modus a auf dem Kontextknoten erlaubt ist, da die Sperranforderung im nächsten Schritt des hierarchischen Sperrprotokolls auf jeden Fall auf einem Kindknoten blockiert wird. Damit ist die Konversion korrekt.

Für unser Beispiel untersuchen wir zunächst, durch welche Konversionen eine CX-Sperre auf dem Kontextknoten eingetragen wird. Dies erfolgt erstens bei vorhandener IR-, NR-, LR-, SR-, IX- oder CX-Sperre für die Anforderung einer CX-Sperre und zweitens bei einer vorhandenen CX-Sperre für die Anforderung einer IR-, NR-, LR-, SR- oder IX-Sperre. Im ersten Fall muss gemäß dem taDOM2-Protokoll die angeforderte CX-Sperre auf dem Kontextknoten die Anforderung einer SX-Sperre auf einem Kindknoten zur Folge haben, die jedoch durch die NR-Sperren der Konversion auf jedem Kind blockiert wird. Die für den zweiten Fall auf dem Kontextknoten bereits eingetragene CX-Sperre kann aus demselben Grund gar nicht vorhanden sein. Somit werden alle möglichen Sperranforderungen, die eine CX-Sperre auf dem Kontextknoten auslösen können, auf dem Kontextknoten selbst oder einem seiner Kinder blockiert.

Die Konversion beim Sperrprotokoll taDOM2 für eine vorhandene LR-Sperre und eine anforderte CX-Sperre in den resultierenden Modus CX auf dem Kontextknoten und NR auf allen Kindknoten ist damit korrekt.

Zum Nachweis der Korrektheit der Konversionsmatrizen aller taDOM-Sperrprotokolle wurden die oben beschriebenen Stärke-Beziehungen, das Verhalten der Protokolle beim *Downgrade* und die Anwendung der indizierten Konversionsregeln in über 32.000 Fällen überprüft. Wie auch im Abschnitt zuvor sind diese Ergebnisse aufgrund des Umfangs nicht im Anhang enthalten und können auf der XTC-Projektseite [5] eingesehen werden.

5.7 Konsistenzstufen

Die taDOM-Sperrprotokolle unterstützen die Verarbeitung von Transaktionen in den nach [GLP+76] bekannten Konsistenzstufen, die sich durch die Dauer der gehaltenen Lese- und Schreibsperren unterscheiden und somit verschiedene Formen von Änderungsanomalien [GR93] verhindern (siehe Abbildung 93). Eine *lange* Sperre wird stets bis zum Transaktionsende gehalten, eine *kurze* Sperre bereits nach der Operation, für die sie auf einem Kontextknoten angefordert wurde, wieder freigegeben. Dies bezieht sich auch auf alle Sperren, die

beim Einsatz eines hierarchischen Protokolls von der Baumwurzel bis zum Kontextknoten angefordert werden müssen. Die Sperranforderung erfolgt durch das Eintragen der Sperren von der Wurzel abwärts bis zum Kontextknoten, die Freigabe durch das Entfernen der Sperren vom Kontextknoten aufwärts bis zur Wurzel. Zur Erläuterung des Verhaltens der taDOM-Protokolle in den jeweiligen Konsistenzstufen werden zunächst alle Sperrmodi in Lese- und Schreibsperren unterteilt.

- Lesesperren: IR, NR, LR, SR, NU, LRNU, SRNU, SU, ER, EU
- Schreibsperren: IX, NRIX, LRIX, SRIX, CX, NRCX, LRCX, SRCX, NX, LRNX, SRNX, SX, EX

Konsistenzstufe	Anomalie		
	Dirty Read	Non-Repeatable Read	Phantome
Uncommitted Read - lange taDOM-Schreibsperren - keine Lesesperren	möglich	möglich	möglich
Committed Read - lange taDOM-Schreibsperren - kurze taDOM-Lesesperren	verhindert	möglich	möglich
Repeatable Read - lange taDOM-Schreibsperren - lange taDOM-Lesesperren	verhindert	verhindert	möglich
Serializable - lange taDOM-Schreibsperren - lange taDOM-Lesesperren - wertbasierte Achsensperren	verhindert	verhindert	verhindert

Abbildung 93: Konsistenzstufen

In Konsistenzstufe 1 (*Uncommitted Read*) halten Transaktionen lange Schreibsperren, fordern jedoch keine Lesesperren an. Dies verhindert das gegenseitige Überschreiben von Knotenwerten (*Lost Update*), ermöglicht jedoch das Lesen schmutziger Daten (*Dirty Read*).

Transaktionen in Konsistenzstufe 2 (*Committed Read*) halten lange Schreibsperren und fordern kurze Lesesperren an, die nach Ausführung der entsprechenden Operation sofort wieder freigegeben werden. Eine Transaktion liest nur bereits freigegebene Daten, die jedoch bis zu einem erneuten Lesevorgang von einer parallel erfolgreich zum Ende gekommenen Transaktion modifiziert werden können. Die Anomalie des *Non-repeatable Read* kann somit auch in dieser Konsistenzstufe noch auftreten.

Konsistenzstufe 3 (*Repeatable Read*) verhindert diese Anomalie durch das Halten langer Lese- und Schreibsperren. Die von einer Transaktion gelesenen und geänderten Daten werden bis zu ihrem Ende durch Sperren vor Modifikationen geschützt.

Lange Lese- und Schreibsperren, die nur auf den von den Operationen betroffenen Knoten angefordert werden, können allerdings nicht die *Phantomanomalie* verhindern. Dies ist mit einem rein hierarchischen Protokoll nur durch eine Vergröberung der Sperrgranularität zu erreichen [HR99], wodurch jedoch die Parallelität der Transaktionsausführung unnötig stark eingeschränkt wird. Zur Verhinderung von Phantomen (Konsistenzstufe *Serializable*) führen wir daher zusätzlich zu den taDOM Knoten- und Kantensperren im nächsten Abschnitt das Konzept der wertbasierten Achsensperren ein.

5.8 Verhinderung von Phantomen

Die in Abschnitt 4.2 definierten 19 Basisoperationen unterstützen ausschließlich den navigierenden Zugriff auf taDOM-Bäume. Der Zugriff auf einen beliebigen Knoten erfolgt über dessen DeweyID mit der Operation *getNode()*. Danach sind Navigationsschritte zu dessen Vorfahren, Geschwistern, Kindern, Nachfahren und Attributen sowie Modifikationen von Knotenwerten und der Dokumentstruktur möglich. Während der Ausführung aller dieser Operationen auf einem taDOM-Baum werden für die taDOM-Protokolle die zuvor in den Abschnitten 5.1 bis 5.4 beschriebenen Sperren auf den Knoten und Kanten angefordert. In der Konsistenzstufe *Repeatable Read* bedeutet dies, dass für eine Transaktion die Kanten zwischen allen traversieren Knoten und die Knoten selbst bzw. alle vollständig gelesenen Ebenen und Teilbäume mit Lesesperren bis zum Transaktionsende belegt sind. In diese Bereiche des taDOM-Baums können keine weiteren Knoten durch parallel laufende Transaktionen eingefügt oder vorhandene Knoten modifiziert werden, da für die Änderungen exklusive Schreibsperren auf den entsprechenden Knoten und virtuellen Navigationskanten anzufordern sind, die inkompatibel zu den vorhandenen Lesesperren sind. Somit können in den traversierten Bereichen keine Phantome auftreten.

Die Auswertung deklarativer Anfragen mit dem ausschließlichen Einstieg auf der Baumwurzel und anschließenden Navigationsoperationen [Ha05a] führt bei großen taDOM-Bäumen zu merklichen Leistungseinbrüchen. Die Anfrageauswertung ist daher mit zusätzlichen Indexstrukturen zu unterstützen. Für die XML-Anfragesprache XQuery unterstützt solch eine Indexstruktur typischerweise direkt die Auswertung der XQuery-Pfadachsen, sodass ein Index für taDOM-Bäume eine Liste aller DeweyIDs der Knoten zurückliefert, die sich für einen gegebenen Kontextknoten und eine auszuwertende Pfadachse qualifizieren. Die so ermittelten Knoten können zur Konstruktion des Anfrageergebnisses mit der taDOM-Operation *getNode()* geladen werden.

Weil die Knoten nun jedoch direkt adressiert und geladen und bis auf die Knotensperren keine weiteren Sperren auf den virtuellen Navigationskanten angefordert werden, können beim Zugriff auf einen taDOM-Baum über eine Indexstruktur Phantome auftreten. Für die XQuery-Anfrage *.//following::Kunde* müssen bspw. für den Kontextknoten alle Nachfolger-Elemente ermittelt werden, die den Namen *Kunde* tragen (vgl. Abschnitt 2.3.3: Nachfolger sind alle Knoten, die in Dokumentenordnung hinter dem Kontextknoten stehen und keine Nachfahren sind). Zur Vermeidung von Phantomen muss nun verhindert werden, dass in den taDOM-Baum ein Element *Kunde* als Nachfolger des Kontextknotens eingefügt wird, weil dieses Element bei einer wiederholten Anfrage als Phantom erscheinen würde. Aus Gründen der parallelen Transaktionsverarbeitung scheidet eine exklusive Schreibsperre mit gröberem Granulat aus, da diese wegen der Pfadachse *Nachfolger* auf der Dokumentwurzel angefordert werden müsste und somit jegliche weitere Modifikationen des taDOM-Baums unterbunden wären. Auch das klassische Verfahren des *Key-Range Locking* [GR93] auf der Indexstruktur ist zu restriktiv, weil dadurch das Einfügen von *Kunde*-Elementen generell verhindert wird. Je nach Implementierung des Index (vgl. dazu die Indexstruktur von Xindice auf vollständigen Kollektionen in Abschnitt 3.1.6) kann dies sogar die gesamte Dokumentensammlung betreffen.

Wertbasierte Achsensperren

Auf der XQuery Pfadachse *Self* können für einen Kontextknoten keine Phantome auftreten, da der Kontextknoten beim Zugriff durch eine Lesesperre gegen jegliche Modifikationen geschützt wird. Aus diesem Grund müssen zur Phantomvermeidung auch nicht die Achsen *Ancestor-or-self* bzw. *Descendant-or-self* betrachtet werden, da Phantome nur auf dem *Ancestor*- bzw. *Descendant*-Anteil der Achse möglich sind und dort verhindert werden.

Zur Unterstützung einer weiteren XML-typischen Funktionalität führen wir die Achse *IDvalue* ein, die für einen Kontextknoten das Nachfahren-Element liefert, welches ein ID-Attribut mit einem angegebenen Wert besitzt. Soll beispielsweise im Teilbaum unterhalb des Kontextknotens das Element mit dem ID-Attributwert *kd1606* ermittelt werden, so geschieht dies mit dem Pfadachsenausdruck *IDvalue::kd1606*.

Zur Phantomvermeidung sind nun die 12 XQuery-Pfadachsen abzüglich der *Ancestor-or-self*- und *Descendant-or-self*-Achse sowie die *IDvalue*-Achse zu berücksichtigen. Dazu können für einen beliebigen Kontextknoten eines taDOM-Baums bzw. eines XML-Dokuments für jede der 11 soeben aufgeführten Pfadachsen *wertbasierte Achsensperren* angefordert werden. Eine wertbasierte Achsensperre ist ein Tupel l_{axis} *(deweyID, axis, value, mode)*, wobei

- *deweyID* die DeweyID des Kontextknotens darstellt,
- *axis* eine der 11 Pfadachsen *Ancestor*, *Parent*, *Preceding*, *Preceding-Sibling*, *Self*, *Attribute*, *Child*, *Descendant*, *Following-Sibling*, *Following* oder *IDvalue* auswählt,
- *value* den Anfragewert angibt und
- *mode* einer der beiden Sperrmodi *shared* (R) oder *exclusive* (X) ist.

Sollen beispielsweise für das *Kunde*-Element des Beispiel-taDOM-Baums in Abbildung 76 auf Seite 71 alle *Vorname*-Elemente im darunter liegenden Teilbaum über einen Index ohne explizites Traversieren des gesamten Teilbaums ermittelt werden, so ist dafür die wertbasierte Achsensperre l_{axis} *(7:1.3.3, Descendant, „Vorname“, R)* anzufordern. Die Suche des Elements mit dem ID-Attributwert *kd1606* im gesamten Baum erfordert auf der Wurzel die Achsensperre l_{axis} *(7:1, IDvalue, „kd1606“, R)*. Allgemein ist vor jedem Indexzugriff auf dem Kontextknoten für die durch den Index auszuwertende Achse eine wertbasierte Achsenlesesperre anzufordern.

Umgekehrt müssen bei Modifikationen vor dem Einfügen neuer Knoten bzw. dem Schreiben neuer Werte auf den betroffenen Kontextknoten für die *Self*-Achse exklusive Achsensperren angefordert werden. Wird zum Beispiel (wiederum in Abbildung 76) das Element *Hausnummer* in *Nr* umbenannt, so muss dafür zunächst die Sperre l_{axis} *(7:1.3.3.5.5, Self, „Nr“, X)* eingerichtet werden. Die Modifikation von Attributen kann mehrere Sperren erfordern. Hätte das Element *Kunde* im Beispiel noch kein ID-Attribut, so wären für das Hinzufügen die Sperren l_{axis} *(7:1.3.3, Attribute, „id“, X)* und l_{axis} *(7:1.3.3, IDvalue, „kd1606“, X)* erforderlich. Mit der ersten Sperre wird das Hinzufügen blockiert, falls eine noch laufende Transaktion für eine Anfrage mit der taDOM-Operation *getAttribute (..., „id“)* eine negative Antwort erhalten hat (dieses Attribut existiert nicht); die zweite Sperre verzögert die Operation, falls eine weitere Transaktion für die *IDvalue*-Achse mit dem Wert *kd1606* eine negative Antwort erhalten hat (ein Element mit diesem ID-Attributwert existiert nicht). Eine vollständige Übersicht aller anzufordernden Achsensperren für die taDOM-Operationen ist in Abbildung 94 gegeben.

Damit eine angeforderte Achsensperre auf einem Kontextknoten gewährt werden kann, muss diese kompatibel zu allen für denselben Wert bereits eingerichteten Achsensperren sein, deren adressierte Bereiche des taDOM-Baums sich überlagern. Um eine potenzielle Überlagerung zu bestimmen, muss zunächst die relative Lage der DeweyID jeder wertgleichen Achsensperre bzgl. der DeweyID des Kontextknotens ermittelt werden. Mit den in Abschnitt 4.3.3 beschriebenen Eigenschaften der DeweyIDs zur Pfadauswertung bestimmt man sehr effizient eine der Lagen *Ancestor*, *Parent*, *Preceding*, *Preceding-Sibling*, *Self*, *Child*, *Descendant*, *Following-Sibling* oder *Following* zwischen zwei Element-DeweyIDs, für die Sperren angefordert werden können. Für jede dieser 9 Lagen wird nun eine so genannte *Achsenüberlagerungstabelle* definiert, in der abzulesen ist, welche Achse der jeweiligen Lage, sich mit einer entsprechenden Achse des Kontextknotens schneidet. Überlagern sich die Bereiche der Achsensperren

nicht, so sind die Sperren miteinander kompatibel. Schneiden sich dagegen die Bereiche, so müssen für die Kompatibilität der Achsensperren die eingetragenen Sperrmodi gemäß dem klassischen R/X-Sperrprotokoll [HR99] kompatibel sein (nur der Modus R ist mit sich selbst verträglich).

taDOM-Operation	Achsensperre bzw. Bemerkung
getNode (deweyID)	Nur Zugriff auf sicher vorhandene Knoten.
getParentNode (element)	Zugriff ohne Namensangabe; keine Phantome durch Anwartschaftssperre.
getPrevSibling (element)	Phantomschutz durch Kantensperre zum vorherigen Geschwisterknoten.
getNextSibling (element)	Phantomschutz durch Kantensperre zum nächsten Geschwisterknoten.
getFirstChild (element)	Phantomschutz durch Kantensperre zum ersten Kindknoten.
getLastChild (element)	Phantomschutz durch Kantensperre zum letzten Kindknoten.
getChildNodes (element)	Phantomschutz durch Ebenensperre LR.
getFragmentNodes (element)	Phantomschutz durch Teilbaumsperre SR.
getValue (node)	Direkte Lesesperre auf dem Knotenwert; keine Phantome möglich.
setValue (node, value)	Für Elementknoten: l_{axis} (node.deweyID, Self, value, X) Für ID-Attributknoten: l_{axis} (owningElement.deweyID, IDvalue, value, X)
getAttribute (element, name)	l_{axis} (element.deweyID, Attribute, name, R)
getAttributes (element)	Ebenensperre LR auf Attributwurzel verhindert Einfügen neuer Attribute.
setAttribute (element, name, value)	Wenn Attribut nicht existiert: l_{axis} (element.deweyID, Attribute, name, X) Für ID-Attribute zusätzlich: l_{axis} (element.deweyID, IDvalue, value, X)
renameAttribute (element, old, new)	l_{axis} (element.deweyID, Attribute, new, X)
prependChild (element, newChild)	l_{axis} (newChild.deweyID, Self, newChild.name, X)
appendChild (element, newChild)	l_{axis} (newChild.deweyID, Self, newChild.name, X)
insertBefore (element, newNode)	Für Elemente: l_{axis} (newNode.deweyID, Self, newNode.name, X)
insertAfter (element, newNode)	Für Elemente: l_{axis} (newNode.deweyID, Self, newNode.name, X)
deleteNode (node)	Teilbaumsperre SX erlaubt nur das Löschen von nicht-gelesenen Knoten und verhindert den Zugriff auf den Teilbaum selbst vollständig.

Abbildung 94: Achsensperren für die taDOM-Operationen

Exemplarisch für die *Preceding*-Lage wird eine Achsenüberlagerungstabelle in Abbildung 95 gezeigt. Die Kopfspalte gibt die angeforderte Pfadachse für den Kontextknoten an, die Kopfzeile die Pfadachse einer bereits vorhandenen Sperre auf einem Knoten in *Preceding*-Lage bzgl. des Kontextknotens. Ist beispielsweise bereits eine Achsensperre l_{axis} *(7:1.3.3.3.3, Following, „Vorname“, R)* für eine Transaktion eingerichtet und eine weitere Transaktion fordert die Sperre l_{axis} *(7:1.3.3.3.7.1, Preceding-Sibling, „Vorname“, R)* an, so kann Abbildung 95 angewendet werden, da der Wert *Vorname* übereinstimmt und sich *7:1.3.3.3.3* bzgl. *7:1.3.3.3.7.1* in der *Preceding*-Lage befindet. Die Achsenüberlagerungstabelle sagt nun aus, dass sich die *Preceding-Sibling*-Achse des Kontextknotens (angeforderte Sperre) mit der *Following*-Achse eines Knotens in der *Preceding*-Lage (vorhandene Sperre) überschneidet und somit das R/X-Sperrverfahren anzuwenden ist. Der eingetragene Sperrmodus R ist mit dem angeforderten Sperrmodus R kompatibel, sodass die angeforderte Sperre trotz der Achsenüberlagerung gewährt werden kann.

Eine vollständige Auflistung aller Achsenüberlagerungstabellen für die 9 Lagen zweier Elementknoten zueinander ist in Anhang C zu finden.

Vorhandene Sperre auf einem Knoten der *Preceding*-Achse und nicht der *Preceding-Sibling*-Achse des Kontextknotens											
	Ancestor	Parent	Preceding	Preceding Sibling	Self	Attribute	Child	Descendant	Following Sibling	Following	IDvalue
Ancestor	ja	ja	nein	nein	nein	nein	nein	nein	ja	ja	nein
Parent	ja	nein	nein	nein	nein	nein	nein	nein	ja	ja	nein
Preceding	ja	ja	ja	ja	ja	nein	ja	ja	ja	ja	nein
Preceding Sibling	ja	ja	nein	nein	nein	nein	nein	nein	nein	ja	nein
Self	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja	nein
Attribute	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein
Child	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja	nein
Descendant	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja	nein
Following Sibling	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja	nein
Following	nein	nein	nein	nein	nein	nein	nein	nein	ja	ja	nein
IDvalue	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein

Abbildung 95: Achsenüberlagerungstabelle für die *Preceding*-Lage

5.9 Alternative Sperrprotokolle

Alternativ zu den taDOM-Sperrprotokollen untersuchen wir in Kapitel 7 detailliert die navigationsbasierten Protokolle Node2PL, NO2PL und OO2PL, die im Zusammenhang mit Natix veröffentlicht wurden (Abschnitt 3.3.2) und einige klassische hierarchische Ansätze für relationale Datenbanksysteme [HR99, GR93]. Da die letztgenannten Verfahren in ihrer ursprünglichen Variante keine Navigation unterstützen (das relationale Datenmodell definiert keine Ordnung auf den Tupeln einer Relation), ergänzen wir bei der Implementierung (Kapitel 6) für alle im Folgenden vorgestellten Protokolle die Kantensperren des taDOM-Transaktionsmodells. Für den lesenden Zugriff auf einen einzelnen Kontextknoten benutzen wir eine Anwartschaftssperre, weil diese eine Modifikation des Kontextknotens ausreichend blockiert und nicht den gesamten darunter liegenden Teilbaum sperrt.

5.9.1 IRX / IRX+

Das IRX-Protokoll ist ein einfaches hierarchisches Sperrprotokoll mit allgemeinen Anwartschaftssperren. Als Sperrmodi stehen die R-Sperre (*read*) und X-Sperre (*exclusive*) zum Lesen und Modifizieren eines Teilbaums und die Anwartschaftssperre I (*intention*) zur Verfügung. Da die allgemeine Anwartschaftssperre nicht zwischen einer Lese- und Schreibanwartschaft unterscheidet, ist sie weder mit dem Modus R noch mit X kompatibel. Die Kompatibilitäts- und Konversionsmatrix für das IRX-Protokoll ist in Abbildung 96 dargestellt.

	-	I	R	X
I	+	+	-	-
R	+	-	+	-
X	+	-	-	-

a) Kompatibilitätsmatrix

	-	I	R	X
I	I	I	X	X
R	R	X	R	X
X	X	X	X	X

b) Konversionsmatrix

Abbildung 96: Sperrprotokoll IRX

Bei einer vorhandenen I- oder R-Sperre und der Anforderung des jeweils anderen Sperrmodus fällt auf, dass eine Konversion in den X-Modus erfolgt. Dies ist erforderlich, weil aus der allgemeinen Anwartschaftssperre nicht ersehen werden kann, ob sie aufgrund eines lesenden oder schreibenden Zugriffs eingetragen wurde. Daher muss für eine ausreichende Transaktionsiso-

lation immer der „schlimmste“ Fall des schreibenden Zugriffs angenommen werden. Weil jedoch die zu sperrenden XML-Knoten in unserem Transaktionsmodell mit DeweyIDs adressiert werden, kann diese Konversion optimiert werden. Dazu müssen mit Hilfe der DeweyIDs aus der Menge aller gesperrten XML-Knoten diejenigen Knoten ermittelt werden, die im Teilbaum unterhalb des Kontextknotens liegen (*Descendant*), für den eine Konversion durchgeführt wird. Existieren im Teilbaum nur Lesesperren, so kann eine Konversion in den R-Modus erfolgen, andernfalls muss eine X-Sperre eingetragen werden. Wir bezeichnen dieses optimierte Verfahren als IRX+. Die Matrizen für die Kompatibilität und Konversion zeigt Abbildung 97.

	-	I	R	X
I	+	+	-	-
R	+	-	+	-
X	+	-	-	-

a) Kompatibilitätsmatrix

	-	I	R	X
I	I	I	R/X	X
R	R	R/X	R	X
X	X	X	X	X

b) Konversionsmatrix

Abbildung 97: Sperrprotokoll IRX+

5.9.2 IRIX / IRIX+

Eine Verbesserung des IRX-Protokolls ohne die Analyse weiterer verwalteter Sperren erreicht IRIX durch die Einführung der speziellen Anwartschaftssperren IR und IX, die zwischen einer Lese- und Schreibanwartschaft unterscheiden. Im Gegensatz zu I und R des IRX-Protokolls ist nun IR mit R kompatibel, sodass eine Steigerung der parallelen Transaktionsverarbeitung zu erwarten ist. Die Kompatibilitäts- und Konversionsmatrix für das IRIX-Protokoll sind in Abbildung 98 dargestellt.

	-	IR	IX	R	X
IR	+	+	+	+	-
IX	+	+	+	-	-
R	+	+	-	+	-
X	+	-	-	-	-

a) Kompatibilitätsmatrix

	-	IR	IX	R	X
IR	IR	IR	IX	R	X
IX	IX	IX	IX	X	X
R	R	R	X	R	X
X	X	X	X	X	X

b) Konversionsmatrix

Abbildung 98: Sperrprotokoll IRIX

Auch beim IRIX-Protokoll ist in der Konversionsmatrix ein restriktives Verhalten zu erkennen. Bei einer vorhandenen R- bzw. IX- Sperre und Anforderung einer IX- bzw. R-Sperre muss auf dem Kontextknoten eine X-Sperre eingetragen werden, weil nur mit diesem Sperrmodus ein Teilbaum, der einzelne modifizierte Knoten enthält, vollständig gelesen werden kann. Wir optimieren dieses Verhalten durch indizierte Sperrkonversionsregeln, die schon bei den taDOM-Protokollen zum Einsatz kamen, und nennen das resultierende Protokoll IRIX+ (Abbildung 99). Bei Anforderung einer IX-Sperre für einen Kontextknoten, auf dem die ausführende Transaktion bereits eine R-Sperre besitzt (Entsprechendes gilt für den umgekehrten Fall), wird auf dem Kontextknoten eine IX-Sperre und auf allen direkten Kindern eine R-Sperre eingetragen. Fordert die Transaktion danach auf einem Kind eine weitere IX-Sperre an (Anwartschaftssperren müssen von der Wurzel abwärts zum Kontextknoten eingetragen werden), so wiederholt sich diese Ersetzung im betroffenen Teilbaum, bis schließlich eine X-Sperre gesetzt wird.

	-	IR	IX	R	X
IR	+	+	+	+	-
IX	+	+	+	-	-
R	+	+	-	+	-
X	+	-	-	-	-

a) Kompatibilitätsmatrix

	-	IR	IX	R	X
IR	IR	IR	IX	R	X
IX	IX	IX	IX	IX_R	X
R	R	R	IX_R	R	X
X	X	X	X	X	X

b) Konversionsmatrix

Abbildung 99: Sperrprotokoll IRIX+

5.9.3 URIX

Das URIX-Protokoll umgeht das restriktive Konversionsverhalten und die Anwendung indizierter Sperrregeln bei IRIX durch den ergänzten Sperrmodus RIX. Dieser Modus kombiniert die Modi R und IX und erlaubt damit das Sperren eines vollständigen Teilbaums für den Lesezugriff und die Modifikation einzelner Knoten innerhalb dieses Baums. Zusätzlich führt dieses Protokoll eine Update-Sperre zur Vermeidung von Deadlocks ein. Abbildung 100 zeigt die Kompatibilitäts- und Konversionsmatrix für das URIX-Sperrprotokoll.

	-	IR	IX	R	RIX	U	X
IR	+	+	+	+	+	-	-
IX	+	+	+	-	-	-	-
R	+	+	-	+	-	-	-
RIX	+	+	-	-	-	-	-
U	+	+	-	+	-	-	-
X	+	-	-	-	-	-	-

a) Kompatibilitätsmatrix

	-	IR	IX	R	RIX	U	X
IR	IR	IR	IX	R	RIX	U	X
IX	IX	IX	IX	RIX	RIX	X	X
R	R	R	RIX	R	RIX	R	X
RIX	RIX	RIX	RIX	RIX	RIX	X	X
U	U	U	X	U	X	U	X
X	X	X	X	X	X	X	X

b) Konversionsmatrix

Abbildung 100: Sperrprotokoll URIX

5.10 Zusammenfassung

Dieses Kapitel beschreibt einen Mechanismus zur Synchronisation des nebenläufigen Zugriffs auf durch taDOM-Bäume repräsentierte XML-Dokumente über eine Schnittstelle mit 19 Basisoperationen und sekundäre Indexstrukturen. Dabei werden die aus der Literatur bekannten Konsistenzstufen *Uncommitted Read*, *Committed Read*, *Repeatable Read* und *Serializable* unterstützt.

Zunächst wurden die vier aufeinander aufbauenden hierarchischen Sperrprotokolle taDOM2, taDOM2+, taDOM3 und taDOM3+ für den Zugriff auf das taDOM-Datenmodell diskutiert. Das Protokoll taDOM2 besteht aus einigen elementaren Sperrmodi, die auf den Knoten und Kanten eines taDOM-Baums angefordert werden. Wir sind bei der Diskussion besonders auf die Konversion von Sperrmodi, den Vorgang des *Up*- und *Downgrade* und die Definition der Sperrtiefe eingegangen. Weil das recht einfache Sperrprotokoll taDOM2 nicht alle taDOM-Operationen feingranular synchronisieren kann, wurde das Konzept der virtuellen Namensknoten ergänzt. Einen weiteren Nachteil des Protokolls stellt die Ermittlung aller Kinder eines Kontextknotens während der Sperrkonversion dar. Dieses Problem behebt taDOM2+ durch die Einführung spezieller auf die Konversion zugeschnittener Sperrmodi. Virtuelle Namensknoten erlauben zwar eine sehr feingranulare Transaktionsisolation für alle taDOM-Operationen, benötigen dazu allerdings für jedes Element und Attribut die Verwaltung

von zwei separaten Sperren. Weitere Sperrmodi im Protokoll taDOM3 machen auch dies überflüssig, sodass für alle Transaktionen jeweils nur höchstens eine Sperre pro Knoten erforderlich ist. Analog zum Protokoll taDOM2 erfordert auch taDOM3 u. U. die Ermittlung aller Kinder eines Kontextknotens bei der Sperrkonversion, eine weitere Optimierung liefert das Protokoll taDOM3+.

Für die Korrektheit der Sperrprotokolle bzgl. des taDOM-Datenmodells und der darauf definierten taDOM-Basisoperationen wurde anschließend ein Verfahren vorgestellt, welches das korrekte Verhalten der Basisoperationen beim nebenläufigen Zugriff algorithmisch durch mehrere hunderttausend Einzelprüfungen belegt. Dieses Vorgehen entspricht dem *Model Checking* [CGP99], bei dem die Verifikation einer Systembeschreibung (Use Cases der taDOM-Operationen) gegen eine Spezifikation (Kompatibilitäts- und Konversionsmatrizen) automatisiert durch die Untersuchung eines sehr großen Zustandsraums durchgeführt wird.

Die Phantomproblematik beim Zugriff auf das taDOM-Datenmodell über sekundäre Indexstrukturen wird mit dem Konzept der wertbasierten Achsensperren gelöst. Auf diese Weise wird das Auftreten von Phantomen bei der Auswertung aller XQuery-Pfadachsen verhindert.

Der letzte Teil des Kapitels widmet sich alternativen Sperrprotokollen, die aus der Literatur relationaler Datenbanksysteme entnommen sind und am Ende dieser Arbeit mit Hilfe konkreter Messergebnisse für den Einsatz in XML-Datenbanksystemen beurteilt werden.

KAPITEL 6 Implementierungsaspekte

Zwei Dinge sind zu unserer Arbeit nötig: Unermüdliche Ausdauer und die Bereitschaft, etwas, in das man viel Zeit und Arbeit investiert hat, wieder wegzuwerfen.
(Albert Einstein)

Neben dem taDOM-Transaktionsmodell, das ein Datenmodell, Zugriffsoperationen und Konzepte zur Transaktionsisolation bereitstellt, enthält das XTC-Projekt den Prototyp eines nativen XML-Datenbanksystems (XML Database Management System – XDBMS) zur praktischen Evaluierung der entwickelten Konzepte. Dieses Kapitel stellt zunächst die Systemarchitektur des XDBMS vor und geht dabei detailliert auf die Techniken zur nativen XML-Datenverwaltung ein. Abschließend wird ein Treiberpaket vorgestellt, das auf Anwendungsseite den transaktionsgeschützten Zugriff auf das Datenbanksystem mit den typischen XML-Schnittstellen SAX, DOM und XQuery ermöglicht.

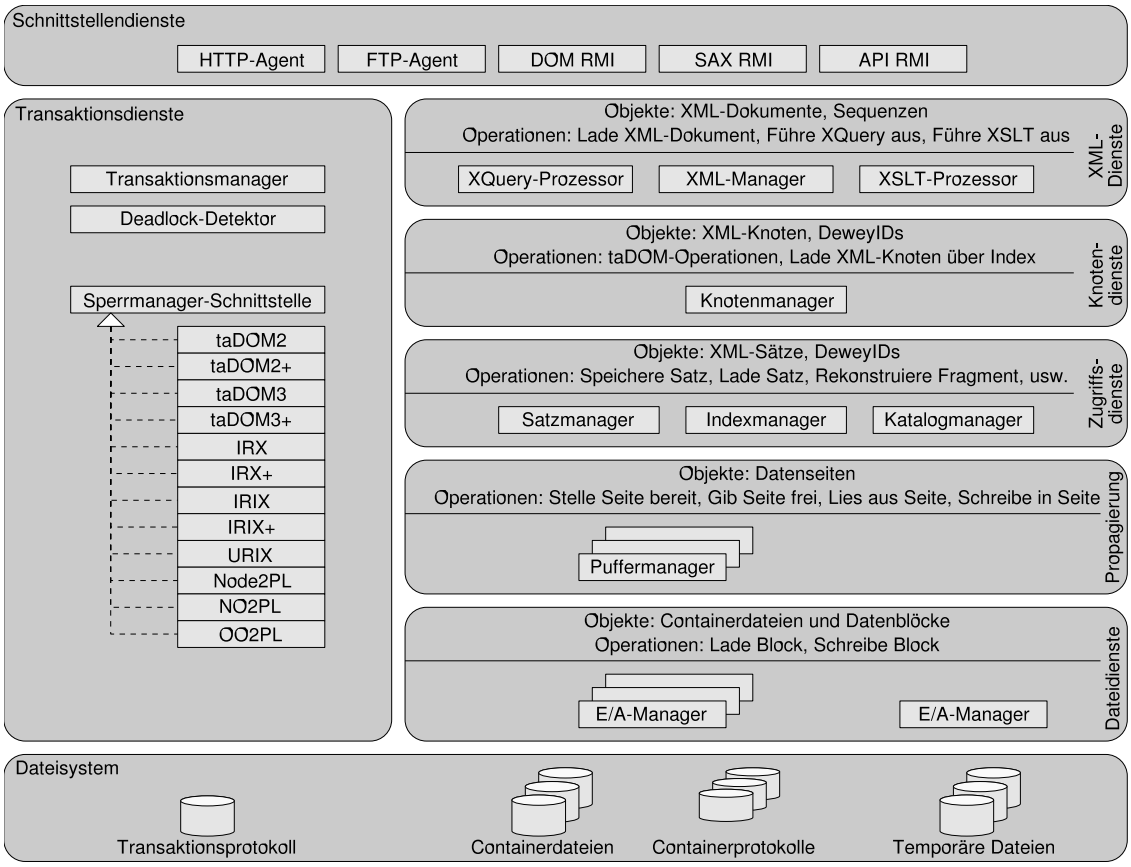


Abbildung 101: Systemarchitektur

6.1 Systemarchitektur

Der XTC-Server ist ein natives XML-Datenbanksystem, das nach dem klassischen Schichtenmodell für Datenbanksysteme [HR83] aus fünf aufeinander aufbauenden Schichten in *Java* implementiert ist. Die Architektur des Systems ist in Abbildung 101 dargestellt.

6.1.1 Dateidienste und Propagierung

Bis auf das Transaktionsprotokoll und einige temporär benötigte Dateien werden alle Daten des XTC-Servers in *Containerdateien* verwaltet. Eine Containerdatei besteht aus *Blöcken* gleicher Größe, die sequentiell in der Containerdatei abgelegt sind. Der Zugriff erfolgt für jede Containerdatei über einen dedizierten *E/A-Manager* in der Schicht der *Dateidienste*. Abbildung 102 zeigt den Aufbau einer Containerdatei.

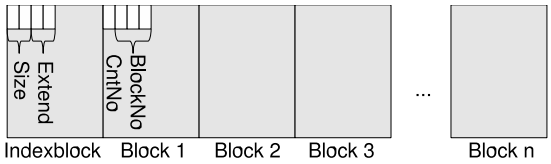


Abbildung 102: Aufbau einer Containerdatei

Bei der Systeminstallation können mehrere Containerdateien unterschiedlicher Größe und mit jeweils verschiedenen Blockgrößen erstellt werden. Der erste Block in einer Containerdatei ist der so genannte *Indexblock*, danach folgen die *Datenblöcke*. Der Indexblock enthält in jeweils zwei Bytes die Parameter *Size* und *Extend*. *Size* gibt die Blockgröße an, mit welcher der E/A-Manager beim Öffnen der Datei die Blockgrenzen ermitteln kann. Jeder Datenblock enthält zunächst eine Nummer, die die Containerdatei identifiziert, und danach seine Blocknummer, gefolgt von den *Nutzdaten* des Blocks. Ist die Blocknummer ungleich 0, so wird der Block als *belegt* interpretiert, ist die Blocknummer gleich 0, so gilt der Block als *frei* und kann bei der Anforderung eines neuen Blocks zugewiesen werden. Wenn eine Containerdatei vollständig belegt ist, so gibt der Parameter *Extend* des Indexblocks die Anzahl der Datenblöcke an, um welche die Datei erweitert wird.

Die *Puffermanager* implementieren die Systemschicht der *Propagierung*. Jedem E/A-Manager wird ein Puffermanager zugewiesen, der die Blöcke der entsprechenden Containerdatei der nächsthöheren Schicht als *Datenseiten* zur Verfügung stellt. Dabei werden die ersten vier Bytes eines Blocks (Containeridentifikator und Blocknummer) als *Seitennummer* interpretiert, sodass direkt aus der Nummer einer Datenseite der verantwortliche Puffermanager bestimmt werden kann. Ein Puffermanager reserviert im Hauptspeicher einen *Puffer* mit einer einstellbaren Anzahl von *Rahmen*, deren Größe der Blockgröße der Containerdatei seines zugewiesenen E/A-Managers entspricht. Fordert eine Komponente der nächsthöheren Schicht eine Datenseite an, so wird diese, falls sie nicht bereits im Puffer vorhandenen ist, als Block über den E/A-Manager geladen und in einem freien Rahmen abgelegt. Ist kein freier Rahmen verfügbar, so muss eine Datenseite aus dem Puffer verdrängt und, wenn sie modifiziert wurde, zurück in die Containerdatei geschrieben werden. Da das XDBMS für jede Containerdatei einen eigenen Puffermanager instanziert, können gegebenenfalls auch individuelle Verdrängungsstrategien eingesetzt werden. Zur Zeit stehen die Verfahren LRD-V2 und LRU zur Verfügung [EH84].

Die Funktionalität der beiden unteren Schichten entspricht der konventioneller Datenbanksysteme und kann mit den dort erprobten Techniken [HR83] realisiert werden.

6.1.2 Zugriffsdienste

Die Systemschicht der *Zugriffsdienste* implementiert die nativen Speicherungsstrukturen für XML-Dokumente. Dazu müssen die taDOM-Bäume des taDOM-Datenmodells (siehe Kapitel 4) auf Datenseiten abgebildet werden, welche die Propagierungsschicht zur Verfügung stellt. Da das Datenmodell auf einer Baumstruktur basiert und die Isolationskonzepte des Transaktionsmodells für den parallelen Zugriff auf einzelne Baumknoten optimiert sind, liegt es nahe, die Knoten eines taDOM-Baums (die Repräsentanten von XML-Elementen, -Attributen und -Textknoten) direkt auf die Knoten einer auf Datenseiten basierenden Baumimplementierung abzubilden. Dafür werden die für Datenbanksysteme etablierten B*-Bäume verwendet [HR99], die als Schlüssel DeweyIDs und als Werte die entsprechenden taDOM-Knoten verwalten.

Satzformate

Die Implementierung von B*-Bäumen für Datenseiten fester Größe realisiert der *Indexmanager*. Die dazu bekannten Algorithmen werden in dieser Arbeit nicht wiederholt und können z. B. in [Co79, Wa73] nachgelesen werden. Für die einzelnen Baumknoten widmet sich dieser Abschnitt stattdessen nur deren physischen Ausprägungen, die für das Verständnis der Auswertungen in Kapitel 7 relevant sind. Das Format eines Baumknotens in der Blattseite eines B*-Baums ist in Abbildung 103 dargestellt.

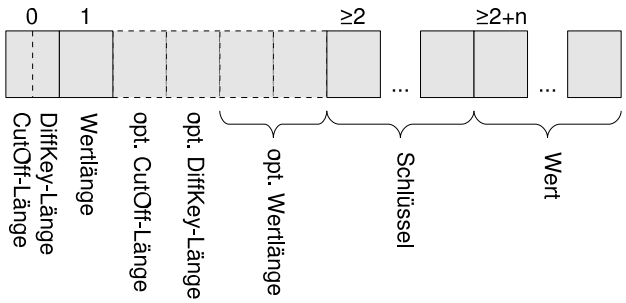


Abbildung 103: Format eines B*-Baum-Knotens

Zur Unterstützung einer Präfix-Komprimierung speichert der Parameter *CutOff* die Anzahl der Bytes, die zur Bildung eines Knotenschlüssels in einer Datenseite vom Schlüssel des davor gespeicherten Knotens abgeschnitten werden müssen. *DiffKey* gibt die Länge des Schlüsselanteils an, der im aktuellen Knoten gespeichert ist und zur Vervollständigung des Schlüssels an die übernommenen Bytes des Vorgängerschlüssels angehängt werden muss. Durch die auf DeweyIDs definierte Ordnung (siehe Abschnitt 4.3.3) werden die Knoten des taDOM-Baums in Dokumentenordnung in den Blattseiten des B*-Baums abgelegt. Dies hat zum einen den Vorteil, dass Teilbäume durch sequentielle Seiten-Scans rekonstruiert werden können, zum anderen profitiert die Präfix-Komprimierung davon: Da die DeweyID eines Knotens in der Regel viele *Divisions* von dem davor gespeicherten Knoten übernimmt und nur wenige neue Divisions ergänzt, ist es sinnvoller, die kleinere Anzahl abzuschneidender Bytes mit dem Parameter *CutOff* zu verwalten statt die größere Anzahl der vom Vorgängerknoten zu übernehmenden Bytes. Daher werden *CutOff* und *DiffKey* für einen zunächst geringen Wertebereich von 0-14 mit jeweils vier Bits in einem einzigen Byte verwaltet. Müssen mehr als 14 Bytes vom Vorgängerschlüssel übernommen bzw. neu angehängt werden, so wird *CutOff* bzw. *DiffKey* auf den Wert 15 gesetzt, welcher ausdrückt, dass ein weiteres Byte mit dem tatsächlichen Wert zwischen 15 und 270 (kodiert durch die Werte 0 bis 255) folgt.

Nach den Parametern *CutOff* und *DiffKey* folgt in einem Byte die Angabe der Länge des Knotenwerts (0 bis 254 Bytes). Auch hier werden für längere Knotenwerte zwei optionale Bytes zur Kodierung größerer Längenangaben (255 bis 65.536) angehängt. Danach ist schließlich mit der zuvor definierten Anzahl von Bytes der Schlüsselanteil des Knotens und sein Wert gespeichert. Eine detaillierte Auswertung dieser Speicherungsstruktur für typische XML-Dokumente wird in Abschnitt 7.1 diskutiert, die Erzeugung einer Byte-Repräsentation für DeweyIDs zur Bildung eines B*-Baum-Schlüssels gesondert im Abschnitt 6.1.4. Für den Zugriff auf B*-Bäume stellt der Indexmanager Operationen zum Einfügen, Ändern, Löschen und Lesen einzelner Knoten zur Verfügung.

Der *Satzmanager* implementiert die taDOM-Operationen des Transaktionsmodells mit Hilfe der Funktionalität, die der Indexmanager anbietet. Die dazu benötigten Meta-Daten stellt der *Katalogmanager* bereit. Die Werte der XML-Knoten werden vom Satzmanager in einem internen byte-basierten Format an den Indexmanager als Werte der B*-Baum-Knoten übergeben. Auch an die nächsthöhere Schicht der *Knotendienste* wird ein Element-, Attribut- oder Textwert als Byte-Folge geliefert, sodass die Zugriffsdienste mit dem Satzmanager die klassische Aufgabe der *internen Satzschnittstelle* [HR99] übernehmen. Das Satzformat, das vom Satzmanager an den Indexmanager übergeben wird, ist in Abbildung 104 dargestellt.

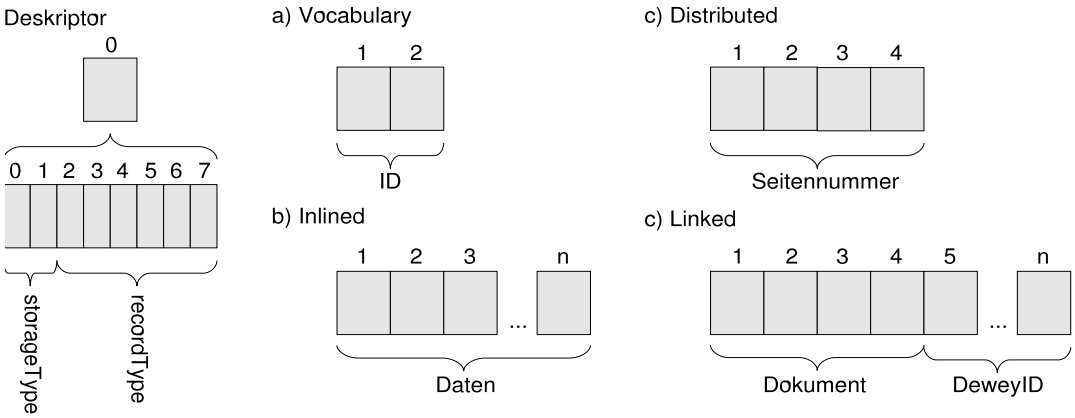


Abbildung 104: Satzformat

Das erste Byte eines Satzes bildet den so genannten *Deskriptor*, der Angaben über die nachfolgende *Speicherungsstruktur* (Bits 0 und 1) und den *Satztyp* (Bits 2 bis 6) kodiert. Für den Satztyp sind zur Zeit *Element*, *Attribut* und *Text* möglich. Die reservierten sechs Bits erlauben bspw. auch die zukünftige Unterstützung von XML-Schema-Datentypen. Für die Speicherungsstruktur sind die Varianten *Vocabulary*, *Inlined*, *Distributed* und *Linked* möglich.

Vocabulary wird für die Speicherung von Elementen verwendet. Der Satzmanager verwaltet ein Vokabular mit den Namen aller Elemente, sodass es eine Zuordnung einer eindeutigen Nummer zu jedem existierenden Elementnamen der gespeicherten Dokumente gibt. Diese Nummer wird für ein Element in den beiden auf den Deskriptor folgenden Bytes gespeichert, womit für den Wert jedes Elementknotens drei Bytes benötigt werden.

Inlined speichert den Knotenwert direkt hinter dem Satzdeskriptor. Eine Längenangabe ist nicht erforderlich, da der gesamte Satz als B*-Baum-Knotenwert an den Indexmanager übergeben und die Länge des Werts dort verwaltet wird. Die *Inlined*-Variante wird bei der Speicherung von Text- und Attributknoten angewendet. Der Wert eines Textknotens wird direkt an den Deskriptor angehängt; bei Attributknoten wird analog zum Elementvokabular der Attribut-

name in den ersten beiden Bytes kodiert, danach folgt der variabel lange Attributwert. Die Trennung im taDOM-Datenmodell zwischen Attribut- bzw. Textknoten und deren Werte in den angehefteten String-Knoten findet auf der Ebene der Speicherungsstrukturen nicht statt. Der Satzmanager ist dafür verantwortlich, dass ein Attribut- bzw. Textsatz als zwei separate taDOM-Knoten mit individuellen DeweyIDs an die höheren Schichten weitergeleitet wird.

Die Implementierung eines B*-Baums erfordert die Festlegung einer maximal möglichen Länge der Schlüssel und Werte. Da textuelle Werte in einem XML-Dokument diese Längen und durchaus auch die Größe einer Datenseite überschreiten können, wurde die Variante *Distributed* eingeführt. Hierbei wird der Knotenwert sequentiell über mehrere, miteinander verkettete Datenseiten verteilt. Auf den Satz-Deskriptor folgen in diesem Fall vier Bytes, die die erste Datenseite der Kette adressieren.

Die letzte Variante wird für die XQuery-Anfrageverarbeitung benötigt und genauer in Abschnitt 6.2.3 motiviert. *Linked* erlaubt die Erzeugung eines Knotens als transparenter Verweis auf einen bereits gespeicherten taDOM-Knoten. Der Zugriff auf einen als *Linked* gespeicherten Knoten wird stets auf das Verweisziel weitergeleitet. Um dieses Ziel zu identifizieren, wird hinter dem Deskriptor das entsprechende Dokument und die Byte-Repräsentation der Ziel-DeweyID gespeichert.

Dokumentspeicherung und -indexierung

Die Datenstrukturen zur Organisation von Bäumen und Listen in Datenseiten implementiert der Indexmanager wie oben beschrieben. Die Speicherung, Indexierung und den Zugriff auf taDOM-Bäume übernimmt der Satzmanager, indem er dazu die entsprechenden Schnittstellenoperationen des Indexmanagers aufruft.

Ein taDOM-Baum (bzw. das durch ihn repräsentierte XML-Dokument) wird mit der in Abbildung 105 dargestellten Speicherungsstruktur verwaltet. Der Satzmanager legt zunächst eine Datenseite für den so genannten *Dokumentkatalog* an, der durch seine Seitennummer einen taDOM-Baum systemweit identifiziert und Verweise zum *Dokumentindex*, *Elementindex* und *ID-Attribut-Index* enthält.

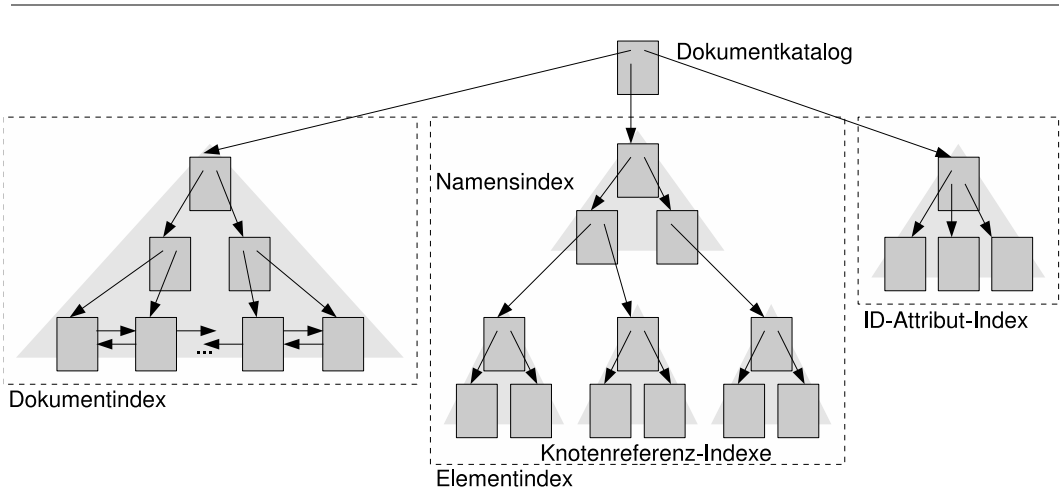


Abbildung 105: Speicherungsstruktur für taDOM-Bäume bzw. XML-Dokumente

Der Dokumentindex speichert, wie zu Beginn dieses Abschnitts beschrieben, die Knoten des taDOM-Baums in Dokumentenordnung in einem B*-Baum. Dazu wird die DeweyID eines taDOM-Knotens auf den Schlüssel und der taDOM-Knoten selbst auf den Wert eines B*-Baum-Knotens abgebildet und für die Schlüssel eine Präfix-Komprimierung angewendet.

Zur Unterstützung der Auswertung von Pfadanfragen erfolgt eine Indexierung aller Elementknoten nach deren Namen im Elementindex, der aus dem *Namensindex* und den *Knotenreferenz-Indexen* besteht. Der Namensindex ist ein B-Baum, der alle im taDOM-Baum auftretenden Elementnamen enthält. Für jeden Elementnamen wird im Namensindex ein Verweis zu einem Knotenreferenz-Index gespeichert. Der Knotenreferenz-Index selbst ist ein B*-Baum, der als Schlüssel die DeweyIDs aller Elementknoten beinhaltet, die den von ihm indexierten Elementnamen tragen. Die Werte der B*-Baum-Einträge werden zur Zeit nicht benutzt. Der Namensindex ist als B-Baum realisiert, weil er in der Regel als Verzeichnis aller möglichen Elementnamen sehr wenige Einträge enthält und selten modifiziert wird. Die Knotenreferenz-Indexe müssen dagegen bei allen Operationen, die das Einfügen, Löschen oder Umbenennen von Elementen betreffen, aktualisiert werden. Zudem ist bei der XML-Anfrageverarbeitung das Ermitteln einer Liste von Knotenreferenzen eine häufig benötigte Operation, die durch die Speicherung der Referenzen in den B*-Baum-Blättern mit sequentiellen Seiten-Scans effizient durchgeführt werden kann.

Um möglichst schnell auf Elementknoten, die mit einem ID-Attribut ausgezeichnet sind, zugreifen zu können, erfolgt eine weitere Indexierung mit dem ID-Attribut-Index. Dieser Index ist als B-Baum realisiert, enthält als Schlüssel die im XML-Dokument auftretenden ID-Attribut-Werte und als Wert jeweils eine DeweyID als Referenz auf das das ID-Attribut besitzende XML-Element.

6.1.3 Knoten-, XML- und Schnittstellendienste

Der *Knotenmanager* in der Schicht der *Knotendienste* bildet die *externe Satzschnittstelle* des Datenbanksystems. Dazu wird für jede taDOM-Operation (Abschnitt 4.2) die entsprechende Implementierung des Satzmanagers aufgerufen und eine Konvertierung der Daten vom internen Format in eine lesbare Zeichenkette vorgenommen. Zusätzlich übernimmt der Knotenmanager vor dem Aufruf der Methoden des Satzmanagers die Anforderung aller benötigten Sperren beim Sperrmanager. Dies entspricht den beschriebenen Sperranforderungen der taDOM-Operationen bei der Definitionen der *Use Cases* im Anhang B, die auch für den Beweis der Korrektheit der Sperrprotokolle benötigt werden (Abschnitt 5.6).

Die *XML-Dienste* realisieren mit der *mengenorientierten Schnittstelle* die höchste Schicht des XML-Datenbankkernsystems. Einzelne XML-Knoten der darunter liegenden Schicht werden zu XML-Dokumenten oder Sequenzen für die XQuery-Verarbeitung zusammengefasst. Der *XML-Manager* stößt die Speicherung und Rekonstruktion von XML-Dokumenten an und stellt Methoden für die Verwaltung von Verzeichnisstrukturen zur Verfügung, in die XML-Dokumente abgelegt werden können. Bevor ein XML-Dokument ausgeliefert wird, kann es optional einer XSL-Transformation des XSLT-Prozessors unterzogen werden. Die Verzeichnisstrukturen sowie die benötigten Meta-Daten des gesamten Systems (bspw. die Dateipfade der Containerdateien und die Parameter der Puffermanager) werden in einem XML-Dokument, dem so genannten *Masterdokument* abgelegt. Der *XQuery-Prozessor* sorgt für die Abarbeitung der XQuery-Anfragen. Dieser Vorgang wird zusammen mit den oben erwähnten Verweisknoten der Speicherungsstrukturen in Abschnitt 6.2.3 beginnend mit dem Abschicken einer Anfrage aus einer Anwendung beschrieben.

Für Anwendungsprogramme wird die gesamte Funktionalität des XTC-Servers über die *Schnittstellendienste* zugänglich gemacht. HTTP- bzw. FTP-Agenten erlauben den Zugriff auf die Dokumente in den Verzeichnisstrukturen des XML-Managers mit einem standardkonformen Web-Browser bzw. FTP-Programm. Für den Zugriff beliebiger Anwendungen ist ein Treiberpaket vorhanden, dessen Funktionsweise in Abschnitt 6.2 beschrieben wird. Das Treiberpaket kommuniziert über den Mechanismus der *Remote Method Invocation* mit den

Komponenten *DOM RMI*, *SAX RMI* und *API RMI*. Diese Komponenten leiten die Funktionsaufrufe zur Bearbeitung an die verantwortlichen Instanzen innerhalb der XML- und Knotendienste weiter.

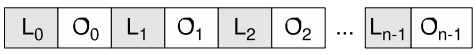
6.1.4 DeweyIDs

Zur Verwaltung der DeweyIDs in einer B*-Baum-Implementierung muss man eine bijektive Abbildung von den logischen Objekten der DeweyIDs auf eine physische Byte-Folge für die Speicherung als Baumschlüssel in einer Datenseite definieren. Dazu werden wir uns zunächst an dem in [NNP+04] beschriebenen Ansatz orientieren, diesen danach schrittweise verbessern und Algorithmen für eine konkrete Umsetzung beschreiben.

Eine DeweyID besteht aus einer Dokumentnummer, gefolgt von einem Doppelpunkt und einer Folge von *Division*-Werten, die durch Punkte voneinander getrennt sind (Abschnitt 4.3). Da die Implementierung der Speicherungsstrukturen vorsieht, für jedes Dokument einen separaten B*-Baum zur Verwaltung der XML-Knoten anzulegen, muss die Dokumentnummer in der Byte-Repräsentation einer DeweyID nicht explizit berücksichtigt werden, weil das entsprechende Dokument durch den B*-Baum implizit identifiziert wird. Somit werden nur die einzelnen *Division*-Werte in einer Datenseite kodiert. Jede DeweyID beginnt mit dem *Division*-Wert *1* (als Präfix von der Dokumentwurzel übernommen). Dieser Wert muss nicht in der physischen Repräsentation dargestellt werden, da er bei jeder Rekonstruktion wieder ergänzt werden kann. Für die exemplarische DeweyID *7:1.12.3* sind somit nur die *Division*-Werte *12* und *3* zu betrachten. Weiterhin ist zu berücksichtigen, dass XML-Elemente in vielen Fällen nur wenige Kinder und Attribute besitzen und dadurch kleine *Division*-Werte sehr viel häufiger als große Werte auftreten. Für Elemente, die dagegen sehr viele Kinder mit dementsprechend hohen *Division*-Werten besitzen, darf die Repräsentation die effiziente Speicherung kleinerer Werte nicht negativ beeinflussen.

Kodierung einer DeweyID

Die Kodierung einer DeweyID erfolgt auf Bit-Ebene durch eine Folge von L_i/O_i-Paaren (Abbildung 106a), wobei O_i den Wert einer *Division* und L_i die Länge (Anzahl der Bits) zur Speicherung von O_i angibt. Beginnend beim ersten Bit einer physischen DeweyID-Repräsentation kann aus L_i stets die Position von L_{i+1} ermittelt werden. L_i selbst ist *präfixfrei*, d. h., kein L_i hat ein anderes L_i als Präfix. Abbildung 106b zeigt in Anlehnung an [NNP+04] eine Kodierungstabelle für den Wertebereich von *Integer*-Zahlen (maximaler Wert 2.147.483.647).



a) DeweyID-Repräsentation

L_i-Kodierung	L_i	O_i-Wertebereich
0	3	1-7
100	4	8-23
101	6	23-87
1100	8	88-343
1101	12	344-4439
11100	16	4440-69975
11101	20	69976-1118551
11110	24	1118552-17895767
11111	31	17895768-2147483647

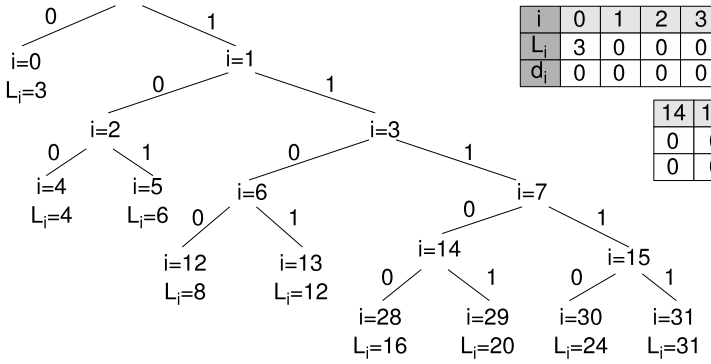
b) L_i-Kodierungstabelle

Abbildung 106: Byte-Repräsentation einer DeweyID

Zur Kodierung eines *Division*-Werts wird zunächst der O_i-Wertebereich gewählt, der den *Division*-Wert beinhaltet. Für den Wert *12* ist dies bspw. der zweite Bereich in der Kodierungstabelle, wodurch L_i mit der Bit-Folge *100* die Kodierung von O_i=*12* mit einer Länge von *4* Bits einleitet. *4* Bits erlauben die Darstellung von *16* Werten, sodass *0000* auf den Wert *8*, *0001* auf den Wert *9* usw. abgebildet wird. Damit ergibt sich eine Repräsentation von O_i=*12* durch *0100*, da es sich bei 12 um den fünften Wert des Wertebereichs *8-23* handelt (Zählen der Werte bei *0* beginnend). Der *Division*-Wert *3* liegt im ersten Wertebereich der Kodierungstabelle, sodass L_i=*3* mit dem Bit *0* dargestellt wird. Im Gegensatz zu den übrigen Wertebereichen beginnt das Zählen der Werte im ersten Bereich aus folgendem Grund nicht bei *0*, sondern bei *1*: Die Bit-Folge einer DeweyID muss zur Speicherung auf die Bytes einer Datenseite abgebildet werden und wird dazu evtl. mit *0*-Bits aufgefüllt, damit sie vollständige Bytes darstellt. Daher wird zwischen der Kodierung des ersten Wertebereichs mit L_i=*0* und erstem Wert *001* und den Füll-Bits als Folge von *0*-Werten unterschieden. Der Division-Wert *3* erhält somit die O_i-Kodierung *011*, für die DeweyID *7:1.12.3* ergibt sich die Bit-Folge *100 0100 0.011 00000* (der Punkt stellt die Byte-Grenze dar), die mit zwei Bytes gespeichert wird.

Dekodierung einer DeweyID

Für die Dekodierung einer DeweyID wird die Byte-Darstellung Bit für Bit analysiert, um die L_i/O_i-Paare zu rekonstruieren. Während der sequentiellen Verarbeitung der Bit-Folge muss jedes L_i effizient erkannt und der nachfolgende O_i-Wert schnell berechnet werden können. Dazu stellen wir in Abbildung 107 die möglichen L_i-Werte mit einem Binärbaum dar und führen als Hilfsstruktur eine *Dekodierungstabelle* ein.



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
L_i	3	0	0	0	4	6	0	0	0	0	0	0	8	12
d_i	0	0	0	0	8	23	0	0	0	0	0	0	88	344

14	15	16	17	18	19	20	21	22	23	24	25
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

26	27	28	29	30	31
0	0	16	20	24	31
0	0	4440	69976	1118552	17895768

Abbildung 107: Dekodierungstabelle für DeweyIDs

Da die möglichen L_i-Werte präfixfrei vergeben werden, sind alle gültigen L_i-Werte in den Blättern des Binärbaums zu finden. Jede Bit-Folge eines Pfads von der Wurzel des Binärbaums zu einem Blatt stellt somit einen L_i-Wert dar, der in der Dekodierungstabelle an der Position *i* des Dezimalwerts der L_i-Bit-Folge abgelegt wird. Zusätzlich wird in der Dekodierungstabelle ein Differenzwert d_i zur Berechnung eines O_i-Werts aus dem durch L_i definierten Wertebereich gespeichert.

Für die Analyse einer Bit-Folge wird nun ausgenutzt, dass in der Dekodierungstabelle für den Positionswert *i* ein L_i-Wert gleich *0* auf einen inneren Knoten des Binärbaums hinweist und ein L_i-Wert ungleich *0* auf ein Blatt, wodurch das Ende einer L_i-Kodierung angezeigt wird. Für die sequentielle Verarbeitung einer Bit-Folge wird der Positionswert *i* zunächst mit *0* initialisiert. Um den durch L_i kodierten Dezimalwert zu berechnen, wird für ein *0*-Bit an der Verarbeitungsposition der Positionswert *i* um *i* erhöht, für ein *1*-Bit um *i+1*. Für den so errechneten Posi-

tionswert wird der L_i-Eintrag in der Dekodierungstabelle überprüft: Ist der Wert gleich 0, so wird durch die analysierte Bit-Folge im Binärbaum ein Pfad zu einem inneren Knoten angegeben und die Verarbeitung muss fortgesetzt werden. Ist L_i in der Dekodierungstabelle dagegen ungleich *0*, so adressiert die Bit-Folge ein Blatt im Binärbaum und ein präfixfreies L_i wurde vollständig erkannt. Somit kodieren die nachfolgenden L_i Bits einen O_i-Wert, der direkt eingelesen werden kann. Um schließlich den O_i-Wert auf den eigentlichen Wert der entsprechenden *Division* der DeweyID abzubilden, muss zu diesem noch der in der Dekodierungstabelle gespeicherte Differenzwert d_i addiert werden. Damit ist eine *Division* dekodiert und die Rekonstruktion der DeweyID kann mit der Erkennung der nächsten L_i-Bit-Folge fortgesetzt werden.

Wir erläutern diesen Algorithmus zur Verdeutlichung am Beispiel einer Rekonstruktion der oben kodierten DeweyID *7:1.12.3* in der Bit-Folge *100 0100 0.011 00000*. Die Dokumentnummer ist implizit durch den B*-Baum definiert, in dem die Bit-Folge als Schlüsselwert gespeichert ist; die erste Division mit dem Wert *1* wird generell ergänzt. Der Positionswert wird mit *0* initialisiert. Bei der Analyse der Bit-Folge wird zunächst ein *1*-Bit gelesen und daraus der Positionswert *0+(0+1) = 1* berechnet. An dieser Position ist in der Dekodierungstabelle der L_i-Wert *0* abgelegt, was bedeutet, dass L_i noch nicht vollständig gelesen wurde. Die beiden folgenden *0*-Bits führen schließlich zu den Positionswerten *1+(1) = 2* und *2+(2) = 4*. An der Position *4* ist der L_i-Wert *4* vermerkt, sodass die Erkennung von L_i abgeschlossen ist und O_i aus den nachfolgenden vier Bits *0100* als Wert *4* gelesen wird. Danach wird O_i=*4* um den Differenzwert d_i=*8* erhöht, sodass der *Division*-Wert *12* der DeweyID berechnet ist. Es folgt die Analyse des nächsten L_i-Werts mit dem Einlesen eines *0*-Bits und der Berechnung des Positionswerts *0+(0) = 0*. Da an der Position *0* der L_i-Wert *3* vorliegt, ist die Analyse für dieses L_i/O_i-Paar bereits beendet. Die nachfolgenden drei Bits liefern den O_i-Wert *3*, der mit dem Differenzwert d_i=*0* den Division-Wert *3* liefert. Zur Bestimmung des nächsten L_i-Werts wird wiederum ein *0*-Bit gelesen, das die Kodierung eines O_i-Werts in den nächsten drei Bits anzeigt. Da in diesen Bits mit *000* jedoch ein ungültiger Wert vorliegt (der kleinstmögliche Wert mit drei Bits ist *001*), muss es sich um Füll-Bits am Ende des analysierten Byte handeln, und die Dekodierung ergibt somit die DeweyID *7:1.12.3*.

Optimierte Kodierungstabellen

Die aus [NNP+04] übernommene Kodierungstabelle bietet eine sehr feine Unterteilung der *Division*-Wertebereiche für eine Kodierung in L_i/O_i-Paaren an. Am oben diskutierten Beispiel für die Kodierung der DeweyID *7:1.12.3* zeigt sich jedoch ein gravierender Nachteil dieser Kodierungstabelle, da zur Abbildung der Bit-Folge auf eine konkrete Speicherungsstruktur (wie z. B. einen B*-Baum) Füll-Bits ergänzt werden müssen.

Legt man bei der Kodierung nun nicht eine möglichst feine Unterteilung der Wertebereiche zugrunde, sondern optimiert die Kodierungstabelle hinsichtlich einer möglichst effizienten Ausnutzung der Bytes, die den Schlüssel eines B*-Baum-Knotens bilden, so benötigt man dadurch weniger Füll-Bits und u. U. weniger Speicherplatz für das gesamte XML-Dokument. Abbildung 108 zeigt einen ersten Ansatz für fünf weitere Kodierungstabellen, die maximal vier Bits für die Kodierung der Länge L_i eines Division-Werts O_i verwenden und somit eine Unterteilung von O_i in fünf Wertebereiche ermöglichen. Entsprechend der Länge einer L_i-Kodierung werden die Bits zur Kodierung des O_i-Wertebereichs so gewählt, dass jeweils alle Bits eines Byte belegt werden. Für die L_i-Kodierung *10* der Länge *2* werden so bspw. *6* bzw. *14* Bits für O_i verwendet, um das L_i/O_i-Paar in einem bzw. zwei Bytes vollständig ohne Füll-Bits abzulegen. Für Attribute, die durch die Attributwurzel des taDOM-Datenmodells (Abschnitt 4.1) die DeweyID des Elternelements erhalten und um *Division*-Werte wie z. B. *1.3* oder *1.5* erweitert werden, führen wir auch die Kodierung besonders kleiner Werte mit vier Bits ein (ein Bit für L_i und drei Bits für O_i), sodass die gesamte Erweiterung in einem Byte Platz findet.

Kodierungstabelle K1		
L_i-Kodierung	L_i	O_i-Wertebereich
0	7	1-127
10	14	128-16.511
110	21	16.512-2.113.663
1110	28	2.113.664-270.549.119
1111	36	270.549.120-68.990.025.855

Kodierungstabelle K2		
L_i-Kodierung	L_i	O_i-Wertebereich
0	3	1-7
10	6	8-71
110	13	72-8.263
1110	20	8.264-1.056.839
1111	36	1.056.840-68.720.533.575

Kodierungstabelle K3		
L_i-Kodierung	L_i	O_i-Wertebereich
0	3	1-7
10	6	8-71
110	13	72-8.263
1110	28	8.264-268.443.719
1111	36	268.443.720-68.987.920.455

Kodierungstabelle K4		
L_i-Kodierung	L_i	O_i-Wertebereich
0	3	1-7
10	6	8-71
110	21	72-2.097.223
1110	28	2.097.224-270.532.679
1111	36	270.532.680-68.990.009.415

Kodierungstabelle K5		
L_i-Kodierung	L_i	O_i-Wertebereich
0	3	1-7
10	14	8-16.391
110	21	16.392-2.113.543
1110	28	2.113.544-270.548.999
1111	36	270.549.000-68.990.025.735

Abbildung 108: 4-Bit-Kodierungstabellen

Abbildung 109 führt acht weitere mögliche Kodierungstabellen ein, wobei die Anzahl der Bits für die Kodierung der präfixfreien L_i-Werte auf höchstens *3* beschränkt wird und somit vier Wertebereiche für O_i möglich sind.

Die Auswirkungen dieser optimierten Kodierungstabellen erörtern wir mit detaillierten Untersuchungen im Abschnitt 7.1 des nächsten Kapitels. Abbildung 110 zeigt in den Teilen a) bzw. b) die Tabellen des Dekodierungsalgorithmus für DeweyIDs mit der 4-Bit- bzw. 3-Bit-Kodierung aus Abbildung 108 bzw. 109.

Kodierungstabelle K6		
L_i-Kodierung	L_i	O_i-Wertebereich
0	7	1-127
10	14	128-16.511
110	21	16.512-2.113.663
111	37	2.113.664-137.441.067.135

Kodierungstabelle K7		
L_i-Kodierung	L_i	O_i-Wertebereich
0	7	1-127
10	14	128-16.511
110	29	16.512-536.887.423
111	37	536.887.424-137.975.840.895

Kodierungstabelle K8		
L_i-Kodierung	L_i	O_i-Wertebereich
0	7	1-127
10	22	128-4.194.431
110	29	4.194.432-541.065.343
111	37	541.065.344-137.980.018.815

Kodierungstabelle K9		
L_i-Kodierung	L_i	O_i-Wertebereich
0	3	1-7
10	6	8-71
110	13	72-8.263
111	37	8.264-137.438.961.735

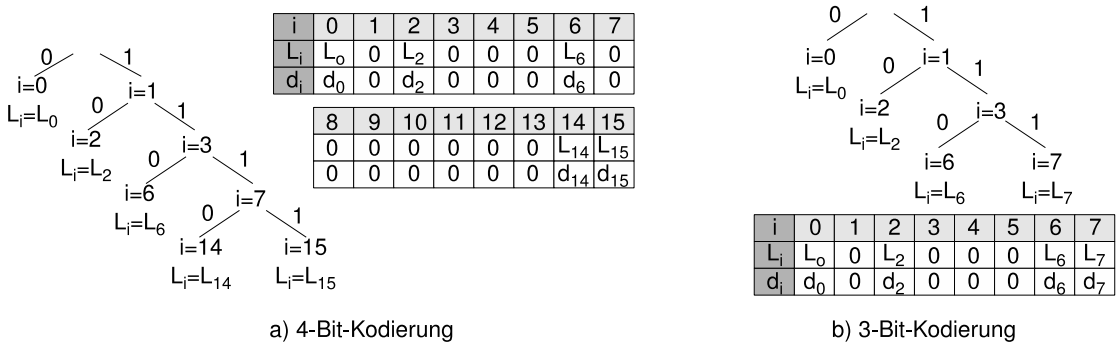
Kodierungstabelle K10		
L_i-Kodierung	L_i	O_i-Wertebereich
0	3	1-7
10	6	8-71
110	21	72-2.097.223
111	37	2.097.224-137.441.050.695

Kodierungstabelle K11		
L_i-Kodierung	L_i	O_i-Wertebereich
0	3	1-7
10	14	8-16.391
110	21	16.392-2.113.543
111	37	2.113.544-137.441.067.015

Kodierungstabelle K12		
L_i-Kodierung	L_i	O_i-Wertebereich
0	3	1-7
10	14	8-16.391
110	29	16.392-536.887.303
111	37	536.887.304-137.975.840.775

Kodierungstabelle K13		
L_i-Kodierung	L_i	O_i-Wertebereich
0	3	1-7
10	22	8-4.194.311
110	29	4.194.312-541.065.223
111	37	541.065.224-137.980.018.695

Abbildung 109: 3-Bit-Kodierungstabellen



a) 4-Bit-Kodierung b) 3-Bit-Kodierung

Abbildung 110: Optimierte Dekodierungstabellen

6.1.5 Sperrverwaltung

Alle Knoten- und Achsensperren (Kapitel 5), die der Knotenmanager bei der Ausführung von taDOM-Operationen beim Sperrmanager anfordert, werden in einer *Sperrtabelle* im Hauptspeicher verwaltet [HH04b]. Damit im folgenden Kapitel 7 verschiedene Sperrprotokolle miteinander verglichen werden können, führen wir das Konzept des *Meta-Locking* (s. u.) ein.

Sperrtabelle

Alle Sperren werden innerhalb der Sperrtabelle im *Sperrpuffer* verwaltet. Für einen schnellen Zugriff auf alle Sperren einer Transaktion (bspw. zur Sperrfreigabe am Transaktionsende) sind diese in einer doppelt verketteten Liste organisiert. Eine analoge Verkettung wird auch für alle Sperren einer DeweyID (bzw. des durch sie adressierten Knotens) geführt, um schnell die Kompatibilität bei Sperranforderungen überprüfen zu können. Dabei werden separate Ketten für Knoten- und wertbasierte Achsensperren (WBA) verwaltet, weil diese nur jeweils untereinander verglichen werden müssen und somit die Anzahl der Kompatibilitätsprüfungen möglichst gering gehalten wird.

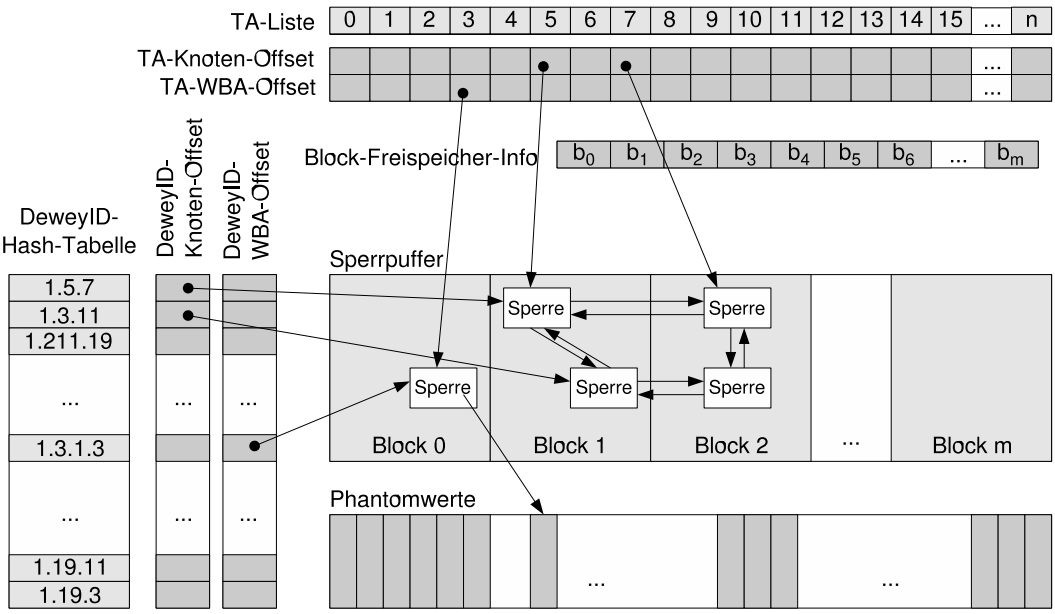


Abbildung 111: Sperrtabelle

Beim Eintragen einer neuen Sperre muss zunächst ein freier Bereich im Sperrpuffer gefunden werden. Weil der Sperrpuffer mehrere Megabytes umfassen kann, wird er in *Blöcke* unterteilt und für jeden Block zusätzlich eine *Freispeicher-Information* in einer kompakten Tabelle verwaltet. Auf diese Weise kann sehr schnell ein Block ermittelt werden, der noch Kapazität für eine weitere Sperre besitzt.

Die Phantomwerte, die für eine wertbasierte Achsensperre gespeichert werden müssen, sind in einer separaten Hash-Tabelle abgelegt. Eine Sperre im Sperrpuffer verweist lediglich auf die Position eines Werts innerhalb der Hash-Tabelle. Somit wird ein Phantomwert, für den mehrere Achsensperren angefordert werden, nur einmal gespeichert. Zusätzlich ermöglicht der Verweis, der als Zahlenwert mit fester Byte-Länge implementiert werden kann, eine Speicherung von Knoten- und Achsensperren mit gleicher Größe, sodass eine Fragmentierung innerhalb des Sperrpuffers vermieden wird.

Meta-Locking

Um mehrere Sperrprotokolle in einem implementierten System miteinander vergleichen zu können, sollte bei den Messungen in einer Testumgebung mit Ausnahme der Sperrverwaltung der Kode des Datenbanksystems unverändert bleiben. Wir führen dazu das Konzept des *Meta-Locking* ein, durch das der Knotenmanager nicht direkt beim Sperrmanager den Sperrmodus eines konkreten Protokolls anfordert, sondern zunächst mit entsprechenden Methodenaufrufen die Eigenschaften des eingesetzten Protokolls erfragt und danach einen Sperrmodus durch dessen Beschreibung anfordert. Die Umsetzung der Beschreibung auf den eigentlich anzufordernden Modus übernimmt der Sperrmanager unter Berücksichtigung des aktuell gewählten Protokolls.

Zur Bestimmung der Eigenschaften eines Sperrprotokolls stellt der Sperrmanager die folgenden drei Methoden zur Verfügung:

- *supportsSharedLevelLocking*: Wenn das Protokoll das Sperren aller Knoten einer Ebene unterstützt, dann wird für die Bestimmung aller Kinder oder aller Attribute eines Kontextknotens eine Ebenensperre auf dem Kontextknoten bzw. seiner Attributwurzel angefordert. Werden keine Ebenensperren unterstützt, so müssen alle Knoten und die betroffenen Navigationskanten einzeln gesperrt werden.
- *supportsSharedTreeLocking*: Analog zu den Ebenensperren gibt diese Eigenschaft an, ob das Protokoll das Sperren vollständiger Teilbäume im Lesemodus unterstützt. Ist dies der Fall, so kann eine Teilbaumsperre bei der Rekonstruktion eines Fragments angefordert werden, ansonsten müssen wiederum jeder Knoten und die Navigationskanten einzeln gesperrt werden.
- *supportExclusiveTreeLocking*: Diese Eigenschaft gibt an, ob das Sperrprotokoll einen Teilbaum vollständig durch eine einzelne Sperre auf der Teilbaumwurzel für den exklusiven Schreibzugriff sperren kann. Besitzt das Protokoll diese Eigenschaft nicht, so müssen beim Löschen eines Fragments zunächst in einem ersten Durchlauf alle enthaltenen Knoten ermittelt und gesperrt werden. Die eigentliche Löschoperation darf erst in einem weiteren Durchlauf erfolgen, nachdem alle betroffenen Knoten gesperrt sind und somit Zugriffe paralleler Transaktionen ausgeschlossen werden können.

Für die Anforderung einer Sperre auf einem Kontextknoten kann einer der Modi *Read*, *Update* oder *Exclusive* für den Kontextknoten selbst, den Kontextknoten und die Ebene aller seiner Kinder oder den Kontextknoten und den darunter liegenden Teilbaum gewählt werden. Zusätzlich kann zur Navigation einer der Sperrmodi auf der *prevSibling*-, *nextSibling*-, *firstChild*- oder *lastChild*-Kante eingetragen werden.

Wie oben bereits erwähnt, erfolgt die Umsetzung der Sperranforderung auf einen konkreten Sperrmodus durch den Sperrmanager in Abhängigkeit des gewählten Protokolls. So wird bspw. die Anforderung des *Read*-Modus auf einem einzelnen Kontextknoten für die taDOM-Protokolle auf eine NR-Sperre abgebildet, wogegen die Anforderung bei den Protokollen RIX, RIX+, IRIX, IRIX+ und URIX in einer I- bzw. IR-Sperre, bei Node2PL und NO2PL in S-Sperren auf den entsprechenden Umgebungsknoten und bei OO2PL in T-Sperren auf den betroffenen Navigationskanten resultieren.

6.2 Anwendungsprogrammierschnittstellen

Für den Zugriff auf den XTC-Server aus einer Java-Anwendung ist ein Treiberpaket vorhanden, das die Verbindungsverwaltung übernimmt und die typischen XML-Schnittstellen zur Auswertung von SAX-, DOM- und XQuery-Anfragen implementiert. In ähnlicher Weise wie die XML:DB API (siehe Abschnitt 2.3.4), jedoch wesentlich feingranularer, ermöglicht das Treiberpaket die kombinierte Nutzung von SAX, DOM und XQuery auf XML-Daten innerhalb eines gemeinsamen Transaktionskontexts [Ha05a].

Der Verbindungsaufbau erfolgt durch einen entsprechenden Methodenaufruf mit Übergabe der Parameter für Benutzername, Passwort, Rechnername und Port des Datenbankservers. Wurde eine Verbindung erfolgreich aufgebaut, so kann darauf über ein *Verbindungsobjekt* zugegriffen werden. Das Verbindungsobjekt erlaubt das Starten, Beenden und Zurücksetzen von Transaktionen (*begin*, *commit*, *rollback*) und das Einstellen einer der Isolationsstufen *uncommitted*, *committed*, *repeatable* oder *serializable* (siehe Abschnitt 5.7). Die Isolationsstufe gilt jeweils für die nächste gestartete Transaktion. Wird ein Transaktionsstart nicht explizit eingeleitet und eine Verarbeitungsanweisung an das Datenbanksystem geschickt, so wird für die Anweisung implizit eine Transaktion gestartet und nach der Verarbeitung der Anweisung wieder beendet (*single statement commit*).

Nach erfolgreichem Verbindungsaufbau bietet der XTC-Server einer Anwendung die Sicht auf die vom XML-Manager verwaltete Verzeichnisstruktur, aus der die Anwendung XML-Dokumente für die Verarbeitung auswählen kann. Die zentrale Idee zur Kombination der drei Schnittstellen SAX, DOM und XQuery innerhalb des Kontexts der gestarteten Transaktion besteht darin, dass der Zugriff auf ein XML-Dokument ein Referenzobjekt liefert, das die Schnittstelle *org.w3c.dom.Document* der DOM-Spezifikation des W3C [DOM3] implementiert. Dieses Referenzobjekt bezeichnen wir im Folgenden als *W3C-Dokumentreferenz*. Alle Methoden des Verbindungsobjekts und der XML-Schnittstellen sind auf diesen Referenztyp zugeschnitten.

6.2.1 SAX

Mit der Methode *saxParse* des Verbindungsobjekts wird das ereignisbasierte Parsen eines XML-Dokuments durchgeführt. Dazu erfolgt zunächst die Instanzierung einer *Content-Handler*-Klasse, die mit der anwendungsspezifischen Implementierung der entsprechenden *Callback*-Methoden auf die eintretenden Ereignisse während der Dokumentanalyse reagiert (siehe Abschnitt 2.3.1). Der Methode *saxParse* werden beim Aufruf als Parameter die W3C-Dokumentreferenz des XML-Dokuments und die Instanz der *Content-Handler*-Klasse übergeben. Das Verbindungsobjekt fordert daraufhin beim Datenbanksystem die Knoten des Dokuments für die SAX-Verarbeitung an (diese werden aus Effizienzgründen zusammengefasst in Paketen übertragen) und ruft entsprechend der ermittelten Knotentypen die jeweiligen *Callback*-Methoden des *Content Handler* auf.

Abbildung 112 zeigt ein Beispiel mit einem Java-Programmfragment für den SAX-Zugriff. In den ersten beiden Zeilen wird zunächst eine Verbindung aufgebaut und eine Transaktion gestartet. In den Zeilen 3 und 4 wird für das XML-Dokument *bank.xml* eine W3C-Dokumentreferenz erzeugt und ein *Content Handler* instanziert. Die SAX-Verarbeitung wird in Zeile 5 mit dem Aufruf der Methode *saxParse* gestartet. Schließlich wird die Transaktion beendet und die Verbindung geschlossen.

```
1 XTCconnection connection = XTCdriver.getConnection(host,port,user,password);
2 connection.beginWork();
3 org.w3c.dom.Document document = connection.getDocument("bank.xml");
4 org.xml.sax.ContentHandler myContentHandler = new MyContentHandler();
5 connection.saxParse(document,myContentHandler);
6 connection.commitWork();
7 connection.close();
```

Abbildung 112: SAX-Zugriff mit dem Treiberpaket

6.2.2 DOM

Der Zugriff auf ein im Datenbanksystem gespeichertes Dokument liefert wie oben beschrieben stets eine W3C-Dokumentreferenz. Somit kann nach der Instanzierung eines solches Referenzobjekts in der Anwendung direkt mittels der DOM-Implementierung des Treiberpakets auf dem Dokument mit der Navigation und Modifikation begonnen werden. Während der Navigation werden die angeforderten Knoten vom XTC-Server zum Treiber übertragen und dort als DOM-Knoten für die Anwendung bereitgestellt. Die mit der DOM-API durchgeführten Modifikationen werden sofort auf die Datenstrukturen im Server unter Anforderung der benötigten Sperren propagiert.

Ein Java-Programmfragment für einen beispielhaften DOM-Zugriff zeigt Abbildung 113. Zunächst wird analog zum SAX-Beispiel eine Verbindung aufgebaut und eine W3C-Dokumentreferenz erzeugt. Danach wird in Zeile 4 mit der Methode *getDocumentElement* das Wurzelelement des XML-Dokuments bestimmt und in Zeile 5 die Methode *getChildNodes* zur Ermittlung aller Kindknoten aufgerufen.

```
1 XTCconnection connection = XTCdriver.getConnection(host,port,user,password);
2 connection.beginWork();
3 org.w3c.dom.Document document = connection.getDocument("bank.xml");
4 org.w3c.dom.Node bankNode = document.getDocumentElement();
5 org.w3c.dom.NodeList bankChildNodes = bankNode.getChildNodes();
6 connection.commitWork();
7 connection.close();
```

Abbildung 113: DOM-Zugriff mit dem Treiberpaket

6.2.3 XQuery

Für die Ausführung von XQuery-Anfragen in Anwendungsprogrammen stellt das Verbindungsobjekt die Methode *executeXQuery* zur Verfügung. Diese Methode erhält als Parameter eine Zeichenkette mit der eigentlichen Anfrage und einen booleschen Wert, mit dem das Ergebnis der Anfrage als *aktualisierbar* (s. u.) deklariert werden kann. Handelt es sich beim Ergebnis der XQuery-Anfrage um ein XML-Dokument (es könnte auch eine Sequenz von Werten und Fragmenten sein), so wird dieses Ergebnisdokument der Anwendung als W3C-Dokumentreferenz

übergeben, damit auf das XQuery-Ergebnis wiederum direkt mit der SAX- oder DOM-Schnittstelle zugegriffen werden kann. Ein Ergebnis, das kein XML-Dokument repräsentiert, wird als einfache Zeichenkette weiterverarbeitet. Abbildung 114 zeigt das Ausführen einer XQuery-Anfrage auf dem Dokument *bank.xml* (Kapitel 2) mit einer anschließenden Verarbeitung des Ergebnisses über die DOM-Schnittstelle. Nach dem Verbindungsaufbau wird in Zeile 3 eine XQuery-Anfrage ausgeführt, die den *Kunden*-Knoten auswählt und in ein *Ergebnis*-Element schachtelt. Für das so erzeugte Dokument wird aus dem XQuery-Ergebnis eine W3C-Dokumentreferenz ermittelt und auf das Wurzelelement zugegriffen (Zeilen 4 und 5). Für das Wurzelelement können anschließend direkt die bekannten DOM-Operationen aufgerufen werden.

```
1 XTCconnection connection = XTCdriver.getConnection(host,port,user,password);
2 connection.beginWork();
3 XTCxqueryResult result
            = connection.executeXQuery("<Ergebnis>
                                          {doc('bank.xml')/Bank/Kunden}
                                        </Ergebnis>",false);
4 org.w3c.dom.Document resultDocument = result.getDocumentResult();
5 org.w3c.dom.Node resultNode = resultDocument.getDocumentElement();
6 org.w3c.dom.Node kundenNode = resultNode.getFirstChild();
7 connection.commitWork();
8 connection.close();
```

Abbildung 114: XQuery-Zugriff mit dem Treiberpaket

Um auf dem XQuery-Ergebnisdokument erneut eine XQuery-Anfrage zu stellen, bietet das Verbindungsobjekt die *executeXQuery*-Methode mit einer zweiten Signatur an. Dabei wird als weiterer Parameter eine W3C-Dokumentreferenz übergeben, auf die in der XQuery-Anfrage mit dem Funktionsaufruf *doc('.')* zugegriffen werden kann. Abbildung 115 zeigt die Ausführung der XQuery-Anfrage aus Abbildung 114 und eine weitere Anfrage auf dem Ergebnisdokument.

```
...
XTCxqueryResult result
            = connection.executeXQuery("<Ergebnis>
                                          {doc('bank.xml')/Bank/Kunden}
                                        </Ergebnis>",false);
...
XTCxqueryResult result2
            = connection.executeXQuery(result,
                                       "<Ergebnis2>
                                          {doc('.')//Kunde}
                                       "</Ergebnis2>",false);
...
```

Abbildung 115: XQuery-Anfrage auf dem Ergebnis einer XQuery-Anfrage

Mit den beiden Varianten der *executeXQuery*-Methode und der Implementierung aller Operationen des Treiberpakets mit W3C-Dokumentreferenzen ist es nun möglich, auf ein XML-Dokument innerhalb eines Transaktionskontexts mit der SAX- und DOM-Schnittstelle oder einer XQuery-Anfrage zuzugreifen. Das XQuery-Ergebnis kann im Fall eines wohlgeformten XML-Dokuments wiederum mit der SAX- und DOM-Schnittstelle verarbeitet werden oder Basis weiterer XQuery-Anfragen sein.

Aktualisierbare XQuery-Ergebnisdokumente

Ist das Ergebnis einer XQuery-Anfrage ein XML-Dokument, so wird es vollständig in einer temporären Containerdatei des Datenbanksystems materialisiert und eine Referenz auf dieses Dokument an die Anwendung geliefert. Mit dem booleschen Parameter der *executeXQuery*-Methode kann das Ergebnis als *aktualisierbar* gekennzeichnet werden. In diesem Fall wird nicht das gesamte Ergebnis materialisiert, sondern nur Knoten, die durch die XQuery-Anfrage neu deklariert werden. Alle übrigen Knoten, die die Anfrage aus bereits gespeicherten Dokumenten (sog. *Basisdokumente*) selektiert, werden über *Verweisknoten* in das Ergebnis aufgenommen. Abbildung 116 zeigt diese Technik mit einem aktualisierbaren (a) und einem nicht-aktualisierbaren (b) Ergebnis.

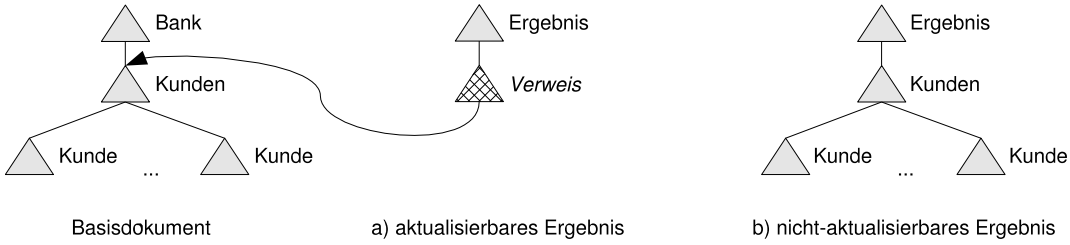


Abbildung 116: Materialisierte XQuery-Ergebnisdokumente

Ein *nicht-aktualisierbares* Ergebnis wird vollständig als XML-Dokument materialisiert. Dieses Ergebnis kann in Abhängigkeit des Basisdokuments u. U. sehr groß werden. Bei einem *aktualisierbaren* Ergebnis werden die durch die Anfrage selektierten Knoten als Verweise in das Ergebnis aufgenommen, sodass nur neu erzeugte Knoten und die Verweisknoten selbst gespeichert werden müssen. Führt eine Anwendung auf dem Ergebnis einer Anfrage Navigationsoperationen durch, so führen diese durch die Verfolgung der Verweise zurück auf das Basisdokument.

In Abbildung 116a liefert das Datenbanksystem der Anwendung als erstes Kind des *Ergebnis*-Knotens den *Kunden*-Knoten aus dem Basisdokument. Für die Anwendung ist dieser Verweis jedoch transparent: Die Navigation erfolgt aus Anwendungssicht auf dem Ergebnisdokument. *Aktualisierbar* bedeutet in diesem Zusammenhang, dass die Modifikation eines Ergebnisknotens mit der DOM-Schnittstelle unmittelbar auf das Basisdokument propagiert wird, falls der Knoten durch Selektion aus einem Basisdokument in das Ergebnis aufgenommen wurde und nicht in der XQuery-Anfrage selbst (bspw. als einschachtelndes Element oder beschreibender Text) erzeugt wurde.

Propagierung von Modifikationen

Bei der Materialisierung aktualisierbarer Ergebnisse können in Abhängigkeit der XQuery-Anfrage zwischen dem Ergebnisdokument mit seinen Verweisknoten und dem Basisdokument der Anfrage vier verschiedene Beziehungstypen auftreten, die in Abbildung 117 dargestellt sind. Die Semantik dieser Beziehungen und die Auswirkungen bei der Modifikation des Ergebnisdokuments sollte schon bei der Formulierung der XQuery-Anfrage berücksichtigt werden.

Im ersten Fall handelt es sich um eine 1:1-Beziehung, die bereits in den vorangegangenen Beispielen betrachtet wurde. Verweisknoten im Anfrageergebnis zeigen auf Knoten im Basisdokument; dabei wird jeder Knoten im Basisdokument von nur einem Verweisknoten referenziert. Die Änderung eines Knotenwerts im Ergebnisdokument bewirkt die Änderung genau eines Knotens im Basisdokument.

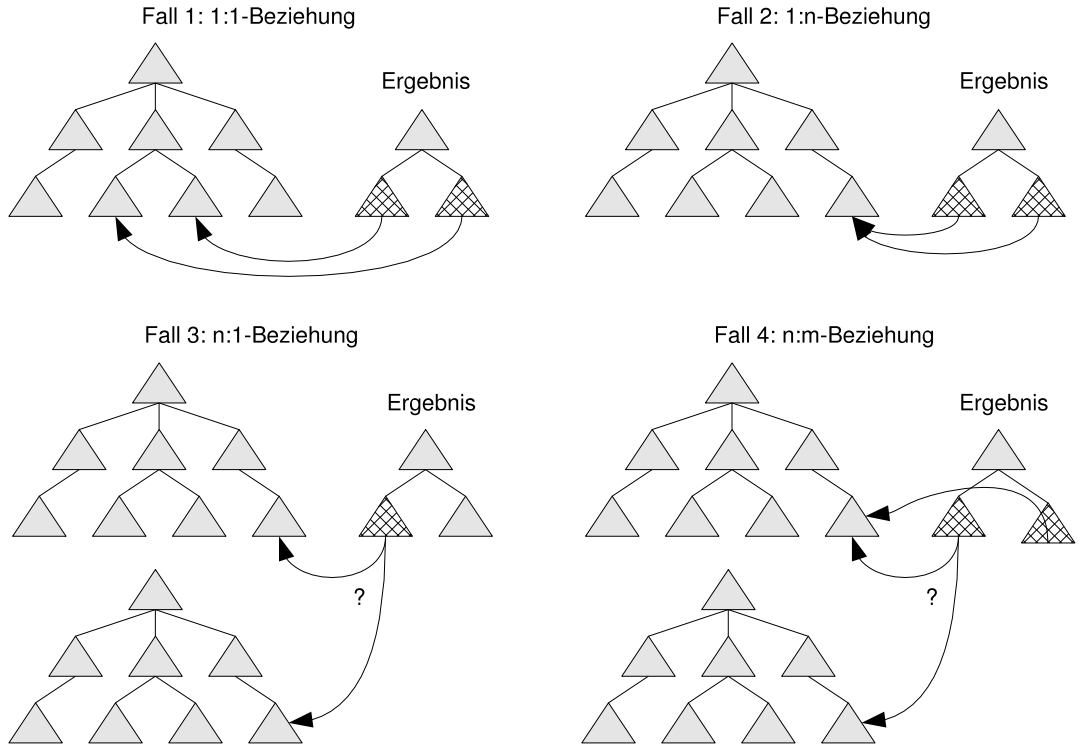


Abbildung 117: Beziehungstypen bei der XQuery-Ergebnismaterialisierung

Im Fall einer 1:n-Beziehung werden bei der Ergebnismaterialisierung zwei oder mehrere Verweisknoten erzeugt, die jeweils denselben Knoten im Basisdokument referenzieren. Das bedeutet, dass die Änderung eines Knotens im Ergebnisdokument durch die Propagierung auf das Basisdokument mit sofortiger Wirkung die Änderung zwei oder mehrerer Knoten im Ergebnis bewirkt, da der von allen referenzierte Basisknoten geändert wird. Diese Semantik muss der Anwender stets beachten.

Fall 3 entsteht durch eine Verbundoperation, bei der der Verweisknoten aufgrund der Wertegleichheit zweier Basisknoten erzeugt wird. Hier muss bei der Formulierung der Anfrage sorgfältig ausgewählt werden, welcher der beiden Knoten (*Verbundpartner*) trotz gleichen Werts für das Ergebnis selektiert wird, da bei einer Aktualisierung des Ergebnisdokuments die Änderung nur auf diesen propagiert wird. Hierbei wird auch besonders deutlich, dass das materialisierte Anfrageergebnis nicht als stets aktuelle Sicht auf das Basisdokument betrachtet werden kann, da nach Änderung eines Knotenwerts keine Anpassung des Ergebnisdokuments bzgl. der gestellten Anfrage erfolgt. Das bedeutet, dass ein Verweisknoten, der aufgrund der Qualifikation bzgl. einer Anfragebedingung erzeugt wurde, nach einer Änderung nicht aus dem Ergebnis entfernt wird, falls er dadurch nicht mehr die Anfragebedingung erfüllt.

Der vierte Fall einer n:m-Beziehung entsteht aus der Kombination einer 1:n- und einer n:1-Beziehung. Die beiden Konstruktionen können getrennt voneinander betrachtet werden, sodass sich für die Änderungssemantik jeweils dieselben Überlegungen wie für die Fälle 2 und 3 ergeben.

6.3 Zusammenfassung

Dieses Kapitel beschreibt die grundlegenden Komponenten der XTC-Systemarchitektur. Der XTC-Server ist ein natives XML-Datenbanksystem, das nach dem bewährten 5-Schichten-Modell aufgebaut ist, wobei besonders in den unteren Schichten viel erprobte Technologie von relationalen Systemen übernommen werden konnte. Die Schichten zur Implementierung der internen und externen Satzschnittstelle und die Sperrverwaltung realisieren die zentrale Funktionalität für die native XML-Datenverarbeitung. Hier wurde die Speicherung von XML-Knoten in Datenseiten, die Kodierung und Dekodierung von DeweyIDs zur Knotenadressierung und die Sperrverwaltung mit dem Konzept des Meta-Locking detailliert betrachtet.

Zur Anwendungsentwicklung stellt das XTC-Projekt die SAX-, DOM- und XQuery-Schnittstelle über ein Treiberpaket zur Verfügung, das die kombinierte Nutzung der drei Schnittstellen in einem gemeinsamen Transaktionskontext erlaubt.

KAPITEL 7 Messungen

Wenn man zuviel weiß, wird es immer schwieriger, einfache Entscheidungen zu treffen.
(Frank Herbert)

Zur Bewertung der bisher beschriebenen Konzepte für die native XML-Datenverarbeitung werden in diesem Kapitel detaillierte Messungen mit dem implementierten XTC-Prototyp durchgeführt.

Wir untersuchen zunächst in Abschnitt 7.1 den Platzbedarf für die Verwaltung von DeweyIDs als Schlüsselwerte in B*-Bäumen und analysieren den Einfluss der Präfix-Komprimierung auf diese Speicherungsstruktur. Mit der Auswertung verschiedener Kodierungsmöglichkeiten zur Abbildung von DeweyIDs auf Bit-Folgen bestimmen wir eine für viele Fälle sehr effiziente Kodierung.

Die Abschnitte 7.2 und 7.3 widmen sich XML-Benchmarks und der Transaktionsisolation und vergleichen die in dieser Arbeit beschriebenen Synchronisationsmechanismen mit zahlreichen Leistungsmessungen. Um diese Messungen möglichst realistisch zu gestalten, werden alle Experimente automatisiert mit einem eigens entwickelten Framework in einer verteilten Testumgebung durchgeführt.

7.1 Speicherungsstrukturen

Um die Eigenschaften verschiedener DeweyID-Kodierungen für unterschiedlich strukturierte XML-Dokumente zu untersuchen [Wa05b], greifen wir auf die XML-Dokumentensammlung der Universität von Washington zurück [WXDR]. Aus dieser Sammlung wählen wir elf *Testdokumente* zwischen 34 KB und 683 MB aus, die in Abbildung 118 nach der Dokumenttiefe und dem Fan-out der Knoten klassifiziert sind.

Das kleinste Dokument *ebay.xml* enthält mit 34 KB die Daten für eine Internet-Auktion. Die Dokumente *customer.xml*, *lineitem.xml* und *orders.xml* sind Abbildungen der entsprechenden Relationen des TPC-H-Benchmark [TPCH] auf XML-Dokumente und speichern Kunden, Rechnungsartikel und Bestellungen. Mit 503 KB, 5,1 MB und 30 MB zählen diese bei unseren Untersuchungen zu den Dokumenten mittlerer Größe. Die Daten des DBLP-Publikationsindex der Universität Trier (*dblp.xml*) haben wir in einer größeren Version mit 271 MB direkt von [DXR] bezogen. Weitere sehr große Dokumente stellen *psd.xml* und *swissprot.xml,* die Proteinsequenzen enthalten, mit jeweils 683 MB und 109 MB dar. Die Datei *mondial.xml* ist eine geographische Datenbank, *nasa.xml* enthält astronomische Daten. Die mit Abstand größte Dokumenttiefe von 37 besitzt das Dokument *treebank.xml,* das Auszüge aus dem Wall Street Journal enthält. Mit dem letzten Dokument *uwm.xml* der Sammlung wird das Kursangebot auf der Webseite der Universität Washington verwaltet.

Dateiname	Beschreibung	Größe [Byte]	max. Tiefe	Ø Tiefe	max. Fan-out	Ø Fan-out
customer.xml	Teil des TPC-H Benchmark	515.660	4	3,41	1501	1,89
dblp.xml	DBLP-Publikationsdatenbank	284.994.162	7	3,39	649.080	2,11
ebay.xml	Auktionsdaten	35.562	6	4,26	12	1,9
lineitem.xml	Teil des TPC-H Benchmark	32.295.475	4	3,45	60.176	1,94
mondial.xml	Geographische Daten	1.784.825	6	4,15	955	3,45
nasa.xml	Astronomische Daten	476.646	9	6,08	2.435	1,76
orders.xml	Teil des TPC-H Benchmark	5.378.845	4	3.42	15.001	1.9
psd.xml	Proteinsequenzen	716.853.016	8	5,68	262.529	1,81
swissprot.xml	Proteinsequenzen	114.820.211	6	4,07	50.000	2,41
treebank.xml	Wall Street Journal Auszüge	86.082.517	37	8,44	56.385	1,58
uwm.xml	Kurse an einer Universität	2.337.522	6	4,37	2.112	1,91

Abbildung 118: XML-Testdokumente

7.1.1 Präfix-Komprimierung

Wir beginnen die Untersuchungen mit dem Einfluss der Präfix-Komprimierung auf den Speicherplatzbedarf der DeweyIDs eines XML-Dokuments. Dazu betrachten wir in separaten Analysen jeweils die Längen der DeweyIDs, die im Dokumentindex und im Elementindex (siehe Abschnitt 6.1.2) gespeichert werden, und variieren dabei den *Distance*-Parameter. Zur Abbildung der DeweyIDs auf Bit-Folgen verwenden wir zunächst die in Anlehnung an [NNP+04] in Abschnitt 6.1.4 initial beschriebene Kodierungstabelle.

Eine Bewertung der optimierten Kodierungstabellen, die bei der Erzeugung der Bit-Folgen auch die Byte-Grenzen der B*-Baum-Schlüssel berücksichtigen, erfolgt im nächsten Abschnitt 7.1.2.

Dokumentindex

Abbildung 119 zeigt den Speicherplatzbedarf aller zu speichernder DeweyIDs für die Testdokumente im Dokumentindex mit *Distance*-Werten von 2 bis 256. Ohne Präfix-Komprimierung werden für Dateien mit besonders großen Dokumenttiefen (bspw. *treebank.xml*, *nasa.xml* oder *psd.xml*) durchschnittlich zwischen etwa 7 und 16 Bytes pro DeweyID zur Speicherung benötigt. Für die übrigen Dokumente bewegt sich der Speicherplatzbedarf je nach *Distance*-Wert zwischen 3 und 9 Bytes.

Es ist zu erkennen, dass mit größerer durchschnittlicher Dokumenttiefe eine Erhöhung des *Distance*-Werts auch zu einem stärkeren Anstieg des Speicherplatzbedarfs der DeweyIDs führt, da für die Knoten aller Ebenen größere *Division*-Werte kodiert werden müssen und ein Knoten stets alle *Divisions* seines Elternknotens übernimmt. Dies bewirkt eine *Streuung* der Messpunkte. Für Dokumente mit geringer Tiefe nimmt der Speicherplatzbedarf bei steigendem *Distance*-Wert weniger stark zu, was in einer *Stauchung* der Messpunkte resultiert.

Der Fan-out beeinflusst die generelle Streuung oder Stauchung der Messpunkte weniger, bewirkt mit seiner Erhöhung allerdings eine Vergrößerung der DeweyID-Längen aller Knoten, die einen Knoten mit hohem Fan-out als Vorfahre besitzen. Dies führt zu einer Verschiebung aller Messpunkte parallel zur y-Achse. Dabei ist eine Zunahme des Speicherplatzbedarfs umso stärker ausgeprägt, je mehr Nachfahren für ein Element mit hohem Fan-out im Dokument gespeichert sind.

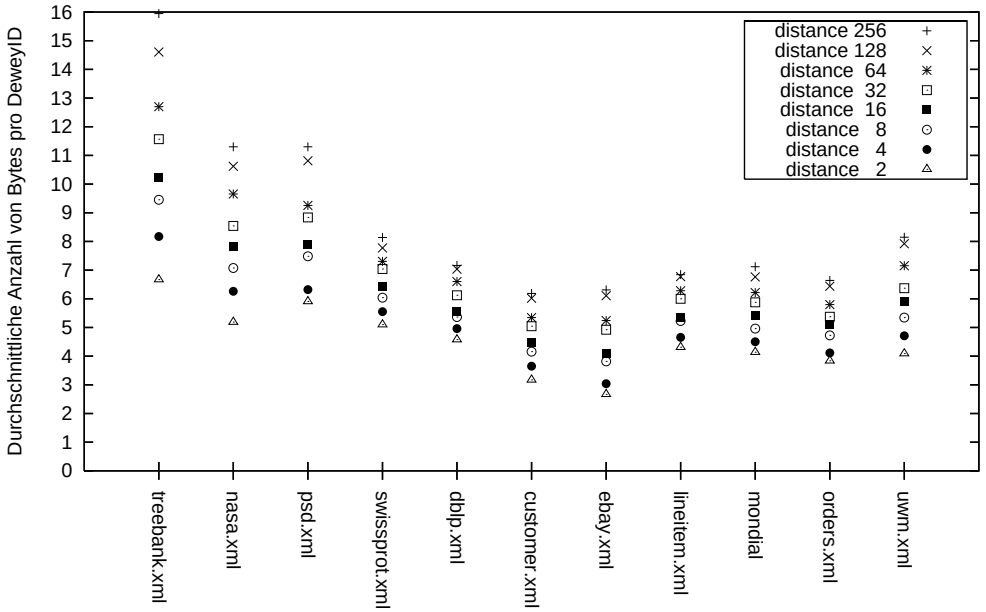


Abbildung 119: DeweyIDs ohne Präfix-Komprimierung im Dokumentindex

Die Knoten eines XML-Dokuments werden in den Blatt-Seiten des Dokumentindex in Dokumentenordnung abgelegt (siehe Abschnitt 6.1.2). Dies bedeutet, dass sich bei der initialen Speicherung eines Dokuments die DeweyID eines beliebigen Knotens von der ID ihres Vorgängers hinter dem Ende des gemeinsamen Präfix nur durch einen weiteren *Division*-Wert unterscheidet. Eine Ausnahme bildet der erste Attributknoten, dessen ID sich von der ID des zuvor gespeicherten Elementknotens aufgrund der Attributwurzel um zwei Divisions unterscheidet.

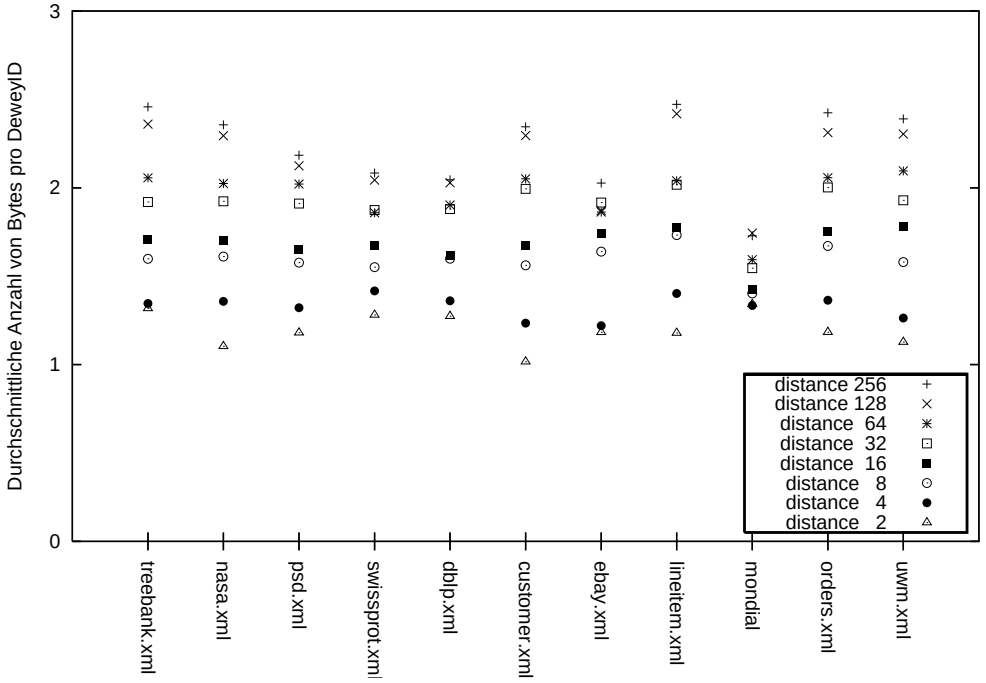


Abbildung 120: DeweyIDs mit Präfix-Komprimierung im Dokumentindex

Die Implementierung einer Präfix-Komprimierung für die B*-Baum-Schlüssel profitiert von dieser Eigenschaft enorm, da für jeden Knoten meist nur eine *Division* als Differenz zum vorherigen Eintrag gespeichert werden muss. Je nach Größe des *Division*-Werts kann dieser in Abhängigkeit der Kodierung oft mit einem oder zwei Bytes dargestellt werden. Abbildung 120 zeigt die durchschnittliche Länge einer DeweyID im Dokumentindex bei Anwendung der Präfix-Komprimierung. Der Speicherplatzbedarf der DeweyIDs sinkt von den ursprünglichen 3 bis 16 Bytes auf 1 bis 2,5 Bytes ab, sodass nur noch etwa 25% des Speicherplatzes für die Adressierung der Knoten benötigt wird.

Elementindex

Im Elementindex wird für die Speicherung der DeweyIDs als Knotenreferenzen etwas weniger Platz benötigt als im Dokumentindex. Der Speicherplatzbedarf ist in Abbildung 121 dargestellt. Der Grund für den geringeren Platzbedarf liegt in der ausschließlichen Speicherung der DeweyIDs von Elementknoten, für die zwei Divisions weniger als für die Attributknoten und eine Division weniger als für die Textknoten im Dokumentindex verwaltet werden müssen. Somit sinkt die durchschnittliche Länge der Bit-Folgen aller DeweyIDs.

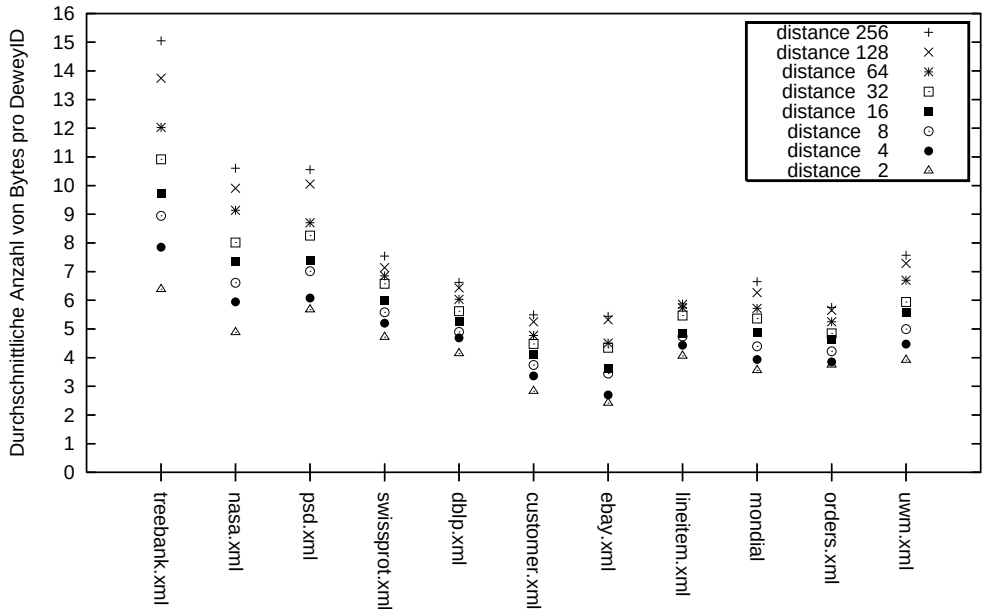


Abbildung 121: DeweyIDs ohne Präfix-Komprimierung im Elementindex

Abbildung 122 zeigt das Ergebnis für die Anwendung der Präfix-Komprimierung im Elementindex. Die durchschnittliche Länge für die Speicherung der DeweyIDs sinkt von zwischen 3 und 15 Bytes (ohne Kompression) auf Werte zwischen 2 und 8 Bytes ab. Die Einsparung ist mit etwa 50% immer noch enorm, dennoch erreicht sie nicht das Potenzial des Dokumentindex. Das liegt daran, dass alle DeweyIDs innerhalb eines Knotenreferenz-Index (siehe Abschnitt 6.1.2) auf Elemente mit demselben Namen verweisen. Da diese referenzierten Elemente in der sequentiellen Ordnung aller Knoten des XML-Dokuments in der Regel nicht direkt aufeinander folgen, besitzen zwei hintereinander gespeicherte Referenzen meist ein kürzeres gemeinsames Präfix, sodass durchschnittlich mehr *Division*-Werte für die Rekonstruktion einer DeweyID aus ihrem Vorgänger gespeichert werden müssen.

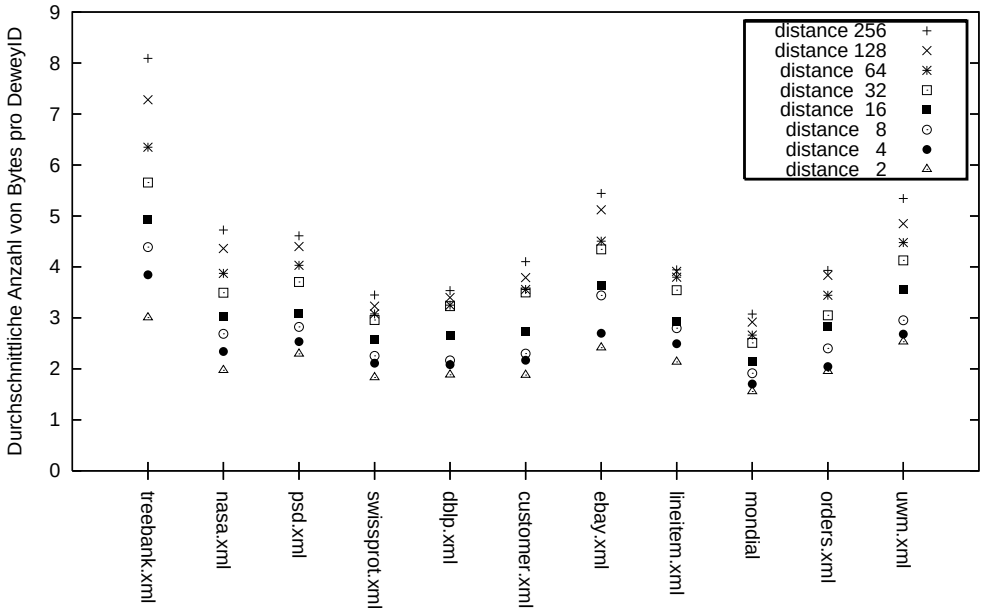


Abbildung 122: DeweyIDs mit Präfix-Komprimierung im Elementindex

CutOff- und DiffKey-Längenfeld

Die Parameter *CutOff* und *DiffKey* verwalten bei der Präfix-Komprimierung für einen Datensatz im B*-Baum die Anzahl der Bytes, die vom Vorgängerschlüssel abgeschnitten werden bzw. im aktuellen Schlüssel zur Ergänzung für eine Rekonstruktion gespeichert sind. Da diese Werte oft sehr klein sind, werden sie mit jeweils vier Bits in einem gemeinsamen Byte verwaltet und können somit die Werte 0 bis 14 kodieren. Für größere Werte wird der entsprechende Parameter auf 15 gesetzt und damit ausgedrückt, dass ein weiteres zusätzliches Byte folgt, das einen der Werte 15 bis 270 kodiert (siehe Abschnitt 6.1.2).

Elementknoten werden üblicherweise in einem Datensatz mit etwa *7* Bytes gespeichert: 1 Byte für die Parameter *CutOff* und *DiffKey* (je 4 Bits), 1 Byte für die Längenangabe des B*-Baum-Knotenwerts, 1-2 Bytes für die zu ergänzenden *Division*-Werte, 1 Byte für den Satzdeskriptor und 2 Bytes für die Vokabularreferenz zur Auflösung des Elementnamens. Daher macht es Sinn, für die Testdokumente zu untersuchen, wie oft jeweils weitere Bytes für *CutOff* oder *DiffKey* zur Kodierung von Werten größer 14 zu speichern sind, weil dadurch die Satzlänge eines B*-Baum-Eintrag um bis zu 30% zunehmen kann.

Die möglichen Werte des *CutOff*-Parameters hängen entscheidend von der Dokumenttiefe ab. Wenn ein Knoten auf einer sehr tiefen Ebene liegt und sein Nachfolger in Dokumentenordnung auf einer hohen Ebene, so müssen dementsprechend viele Bytes vom Vorgängerschlüssel abgeschnitten werden. Eine genauere Untersuchung der Testdokumente ergibt jedoch, dass nur beim Dokument *treebank.xml* (maximale Tiefe 37) für die Rekonstruktion von DeweyIDs mehr als 14 Bytes von Vorgängerschlüsseln abgeschnitten werden. In diesen Fällen wird ein zusätzliches Byte für den *CutOff*-Parameter gespeichert. Abbildung 123 zeigt in Abhängigkeit des *Distance*-Werts, wie oft dies im Dokumentindex eintritt. Erst ab einer gewählten *Distance* von über 128 betrifft die Erweiterung mit einem zusätzlichen Byte mehr als ein Prozent aller Datensätze. Das bedeutet, die optionale Erweiterungsmöglichkeit ist, da sie definitiv auftritt, für eine konkrete Implementierung unerlässlich, die generelle Speicherung des *CutOff*-Werts mit 4 Bits ist für eine kompakte Darstellung der DeweyIDs im Dokumentindex jedoch ausreichend.

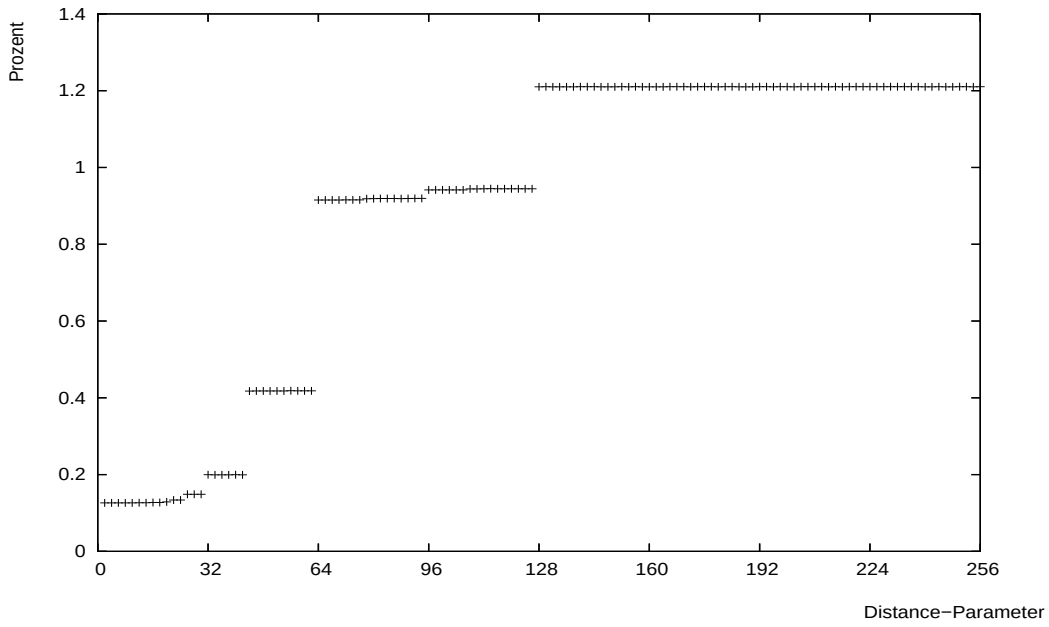


Abbildung 123: Nutzung des optionalen *CutOff*-Felds im Dokumentindex

Im Elementindex wird das optionale *CutOff*-Byte öfter benötigt, da zwei aufeinander folgende Elementreferenzen in der Regel im Dokumentindex nicht benachbart gespeichert sind und sich somit durch mehrere *Divisions* unterscheiden. Dies betrifft allerdings nur zwei Dokumente. Für *nasa.xml* wird ein zusätzliches Byte ab Distance 88 überhaupt erst benötigt, bei einem *Distance*-Wert von 256 tritt dies sogar nur 21 Mal auf. Bei *treebank.xml* steigt die Nutzung aufgrund der exotischen Tiefe des Dokuments bei hohen *Distance*-Werten auf etwa 20 Prozent an.

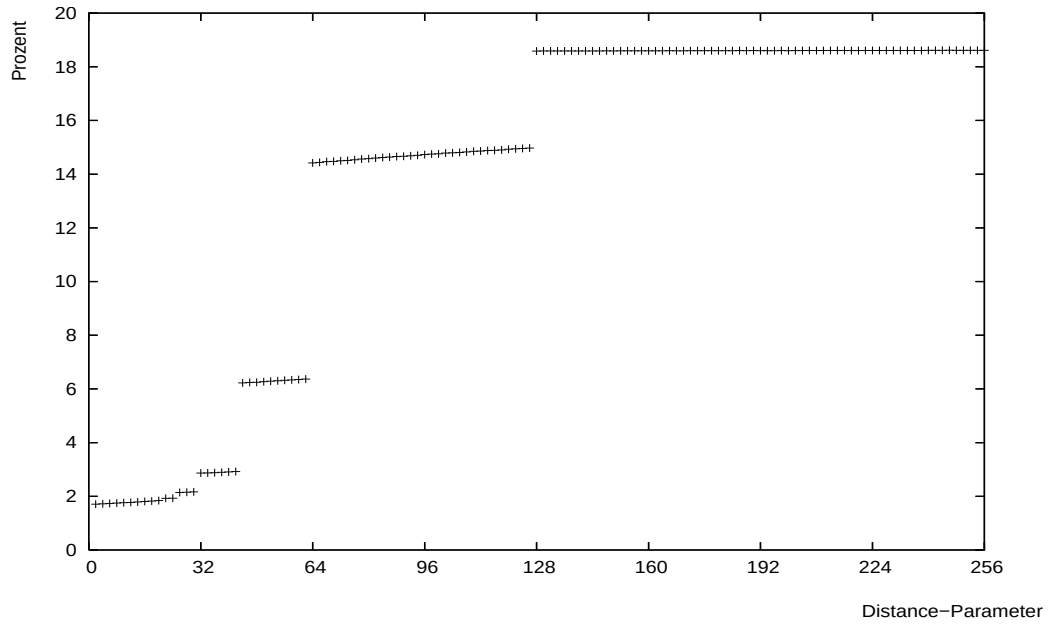


Abbildung 124: Nutzung des optionalen *CutOff*-Felds im Elementindex

Das optionale Byte für den *DiffKey*-Parameter kommt im Dokumentindex bei der initialen Speicherung eines Dokuments nie zur Anwendung. Bei einem gemeinsam genutzten Präfix unterscheidet sich eine DeweyID von ihrem Vorgänger im Index durch höchstens zwei *Divisions*: Für den ersten Attributknoten eines Elements muss der *Division*-Wert *1* für die Attributwurzel und *3* für das erste Attribut gespeichert werden. Alle weiteren Attribut-, Element- und Textknoten können mit einer zusätzlich gespeicherten *Division* rekonstruiert werden. Da mit allen in Abschnitt 6.1.4 vorgestellten Kodierungstabellen diese *Division*-Werte mit maximal 40 Bits darstellbar sind (3 bzw. 4 Bits für L_i und 37 bzw. 36 Bits für O_i), kann sich ein Schlüsselwert durch höchstens 5 Bytes (dies entspricht dem *DiffKey*-Wert) von seinem Vorgänger unterscheiden.

Im Elementindex dagegen wird das optionale Byte für *DiffKey* benötigt, allerdings wieder nur für die beiden Dokumente *nasa.xml* und *treebank.xml*. Bei *nasa.xml* tritt ein zusätzliches *DiffKey*-Byte erst ab einem *Distance*-Wert von 128 auf, bei *Distance* 256 sind es sogar nur 18 Fälle. Für *treebank.xml* zeigt das Auftreten eines zusätzlichen Byte für *DiffKey* im Elementindex ein analoges Verhalten wie für *CutOff*, die Erweiterung ist jedoch nur für etwa 12% aller Knoten erforderlich.

7.1.2 Optimierte Kodierungstabellen

Die Präfix-Komprimierung des Indexmanagers basiert auf der Berechnung eines Schlüsselwerts durch die Übernahme gemeinsamer Bytes aus dem Vorgängerschlüssel. Es bietet sich daher an, die Kodierung einer DeweyID nicht nur auf eine möglichst geringe Anzahl von Bits pro *Division* auszurichten, sondern zusätzlich die Byte-Grenzen der entstehenden Bit-Folge zu berücksichtigen.

Wir haben dazu in Abschnitt 6.1.4 die alternativen Kodierungstabellen K1 bis K13 eingeführt, die im Folgenden genauer untersucht werden. Wir betrachten zunächst den Einfluss von K1 auf die Präfix-Komprimierung im Dokument- und Elementindex und diskutieren danach die Unterschiede zu den Tabellen K2 bis K13, um eine für möglichst viele Dokumenttypen geeignete DeweyID-Kodierung zu bestimmen.

DeweyID-Kodierung mit Berücksichtigung von Byte-Grenzen

Der Einsatz der Kodierungstabelle K1, die in ihrem ersten Wertebereich alle *Division*-Werte zwischen 1 und 127 mit einem vollständig belegten Byte kodiert, führt bei der Speicherung eines XML-Dokuments ohne Präfix-Komprimierung sowohl im Dokument- als auch im Elementindex erwartungsgemäß zu mehr Platzbedarf. Diese Zunahme ist umso stärker ausgeprägt, je mehr Knoten ein Dokument mit vielen keinen *Division*-Werten besitzt (tiefe schmale Bäume), da für diese Werte bei K1 stets ein Byte benötigt wird.

Bei Anwendung der Präfix-Komprimierung zahlt sich die Anpassung der Kodierung an die Byte-Grenzen jedoch aus. Abbildung 125 zeigt die Länge der DeweyIDs für die Speicherung der Testdokumente im Dokumentindex mit der Kodierungstabelle K1. Gegenüber der zunächst nach [NNP+04] gewählten Kodierung sinkt die Anzahl der Bytes pro DeweyID je nach *Distance*-Wert auf durchschnittlich unter 2 Bytes ab.

Auch im Elementindex (hier nicht dargestellt, siehe dazu [Wa05b]) ist mit einem geringeren Speicherplatzbedarf eine Verbesserung durch die Kodierungstabelle K1 zu erkennen. Dies macht sich vor allem dadurch bemerkbar, dass bei der schrittweisen Erhöhung der *Distance*-Werte von 2 bis 256 die Längen der kodierten DeweyIDs weniger stark ansteigen.

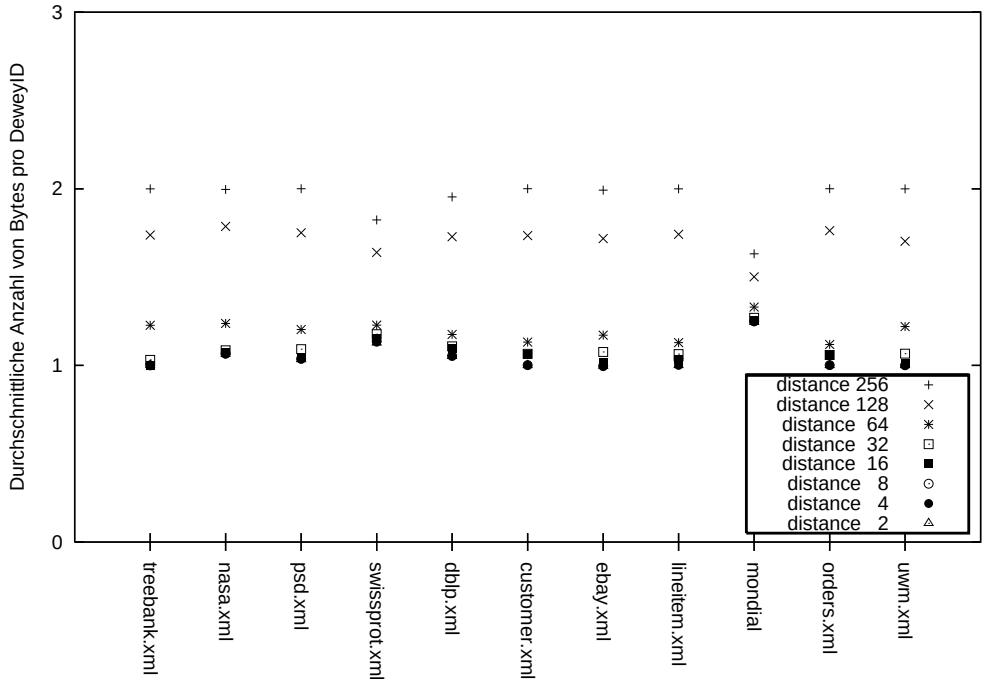


Abbildung 125: DeweyIDs mit Kodierungstabelle K1 im Dokumentindex

Vergleich alternativer Kodierungstabellen

Wir vergleichen zunächst K1 mit den Kodierungstabellen K6, K7 und K8, bei denen jeweils der erste Wertebereich die *Division*-Werte 1 bis 127 enthält. K1 und K6 kodieren die ersten drei Wertebereiche gleich, der vierte Bereich von K6 ist bei K1 in zwei Bereiche aufgeteilt. Das bedeutet, dass die Werte der ersten drei Bereiche identisch kodiert werden und Werte, die in den vierten Bereich von K1 fallen, dort effizienter dargestellt werden. K1 und K7 kodieren die *Division*-Werte bis 16.511 identisch. Werte, die in den dritten Bereich von K1 fallen, werden in K1 effizienter dargestellt; Werte des dritten Bereichs von K7, die nicht im vierten Bereich von K1 liegen, werden in K7 effizienter dargestellt. Für die Präfix-Komprimierung lässt sich zwischen K1 und K7 kein Unterschied feststellen, K1 schneidet jedoch ohne Präfix-Komprimierung bei den Testdokumenten besser ab [Wa05b]. K8 verwendet zur Speicherung des zweiten Wertebereichs bereits drei Bytes, sodass *Division*-Werte über 128 mit K1 deutlich kleiner kodiert werden können. Somit eignet sich K1 im Allgemeinen besser als K6, K7 oder K8 für die Kodierung.

Beim Vergleich der übrigen 3-Bit-Kodierungstabellen geht K3 als Sieger hervor. *Division*-Werte, die bei K2 nicht mehr in den vierten Wertebereich passen, werden mit K3 platzsparender gespeichert. K4 benötigt bereits für Werte ab 72 drei Bytes, wogegen Werte bis 8.263 bei K3 noch mit zwei Bytes darstellbar sind. Auch K5 schneidet schlechter ab, da für die sehr häufig auftretenden Werte 8 bis 71 bereits zwei Bytes benötigt werden, die K3 noch mit einem Byte kodieren kann.

Bei Betrachtung der 4-Bit-Kodierungstabellen K9 bis K13 schneidet K3 ebenfalls besser ab, da in den kleineren Wertebereichen eine feinere Unterteilung für eine effizientere Darstellung sorgt (detaillierte Grafiken zu den Testdokumenten und allen Kodierungstabellen können in [Wa05b] nachgeschlagen werden). Zwar kodieren die 4-Bit-Tabellen große Werte oft durch kürzere Bit-Folgen, allerdings geschieht dies auf Kosten einer schlechteren Darstellung der kleinen *Division*-Werte, die wesentlich häufiger auftreten.

Maximale Kodierungslänge

Aus der bisher geführten Diskussion gehen K1 und K3 als die geeignetsten Tabellen für die Kodierung von DeweyIDs hervor. Um aus diesen beiden Kodierungstabellen einen Favoriten auszuwählen, betrachten wir für das besonders tiefe Dokumente *treebank.xml* als weiteres Kriterium die maximale Länge der kodierten DeweyIDs. Trotz der Präfix-Komprimierung wird am Anfang jeder Datenseite eine vollständige DeweyID gespeichert, die als Basis für die Rekonstruktion der in der Seite nachfolgenden komprimierten Schlüssel dient.

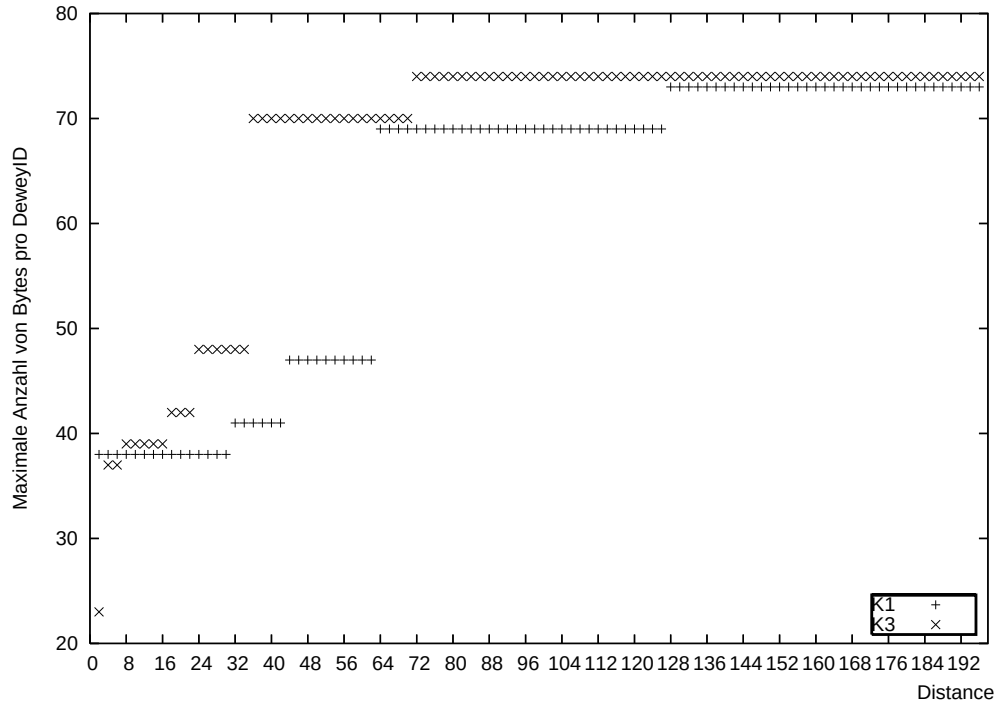


Abbildung 126: Maximale DeweyID-Länge bei *treebank.xml*

Abbildung 126 zeigt die maximale Anzahl von Bytes für die Speicherung einer DeweyID in Abhängigkeit des *Distance*-Parameters mit den Kodierungstabellen K1 und K3. Ab *Distance*-Wert 128 benötigen beide Tabellen annähernd die gleiche Anzahl von Bytes zur Kodierung der längsten DeweyID. Im unteren Bereich für *Distance*-Werte bis 128 dominiert jedoch K1 mit deutlich weniger Speicherplatzbedarf und wird daher als Standardkodierung gewählt. Die geringe DeweyID-Länge von K3 beim *Distance*-Wert 2 bildet eine Ausnahme. Bei der Berücksichtigung aller Testdokumente [Wa05b] liegt K1 auch für kleine *Distance*-Werte im Vorteil.

7.1.3 Reorganisation

Betrachtet man DeweyIDs als rein logisches Konzept, so kann zum Einfügen eines Knotens zwischen zwei existierende Knoten stets eine DeweyID zur Adressierung des neuen Knotens konstruiert werden, indem ein noch „freier“ *Division*-Wert vergeben oder eine weitere *Division* an die DeweyID eines bereits vorhandenen Knotens angehängt wird. Bei der Abbildung von DeweyIDs auf eine Speicherungsstruktur stößt man dabei jedoch an Grenzen, da eine DeweyID in einer konkreten Implementierung (bspw. einem B*-Baum) eine vorgegebene maximale Anzahl von Bytes nicht überschreiten darf. Ist diese Höchstgrenze erreicht, so müssen beim Einfügen eines neuen Knotens alle Geschwisterknoten und die darunter liegenden Fragmente neu nummeriert werden.

Da bei der Implementierung des B*-Baums im XTC-Server eine Präfix-Komprimierung eingesetzt wird und somit die Speicherung einer vollständigen DeweyID nur am Anfang jeder Datenseite erfolgt, ist die Häufigkeit von Reorganisationen schwer abzuschätzen. Zudem hängt die Länge einer DeweyID noch von der Ebene im Dokument und der gewählten Kodierungstabelle ab. Unter Umständen muss eine Neunummerierung auch gar nicht erfolgen, wenn die eigentliche Größe einer DeweyID zwar die maximale Länge überschreitet, aber innerhalb einer Datenseite nur die sehr kleine Differenz zur Vorgänger-ID gespeichert wird.

Um trotzdem ein Gefühl für die Größenordnung der möglichen Knoteneinfügungen bis zu einer Reorganisation zu bekommen, haben wir in [Wa05a] eine Worst-case-Analyse für die Kodierungstabelle nach [NNP+04] durchgeführt. Für diese Analyse wurden die DeweyIDs d_1=*1.2147483647* mit maximalem Division-Wert (über zwei Milliarden Geschwisterknoten) und d_2= *1.2147483647-distance* erzeugt. Zwischen diese beiden IDs werden nun weitere DeweyIDs eingefügt. Bei jeder Einfügung wird die Länge der neu erzeugten DeweyID betrachtet. Ist diese größer als d_2, so wird die neue ID als d_2 benutzt, ansonsten bildet die neue DeweyID d_1. Auf diese Weise ermitteln wir für jede Einfügeoperation den jeweils ungünstigsten Fall für die Konstruktion einer DeweyID.

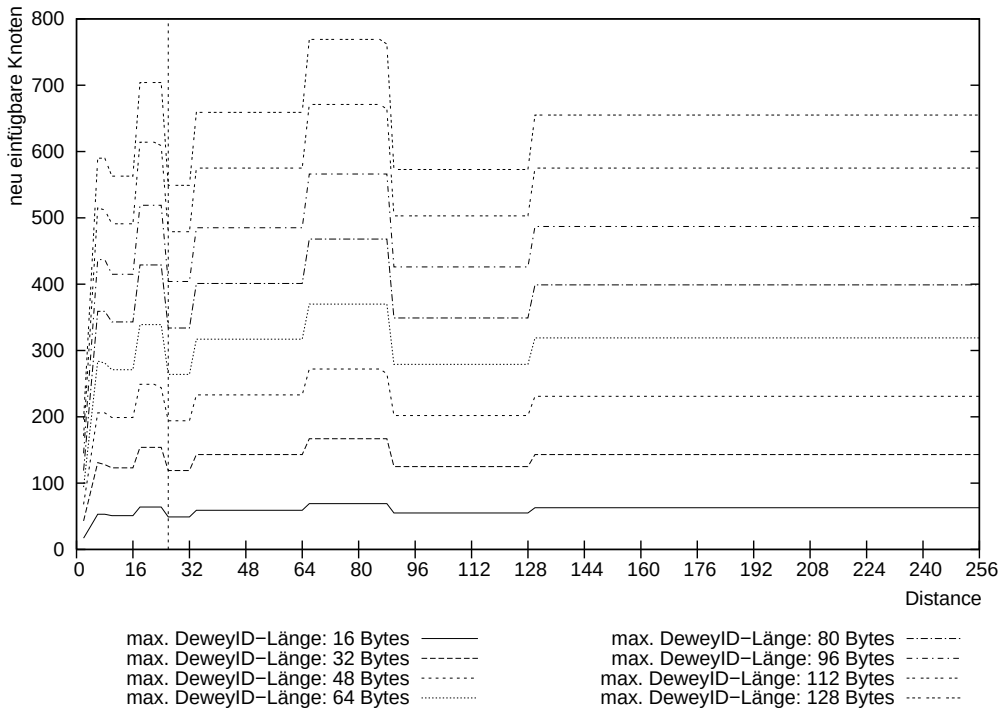


Abbildung 127: Worst-case-Analyse neu einfügbarer Knoten

Abbildung 127 zeigt die Anzahl der einfügbaren Knoten bis zum Erreichen einer maximalen DeweyID-Länge in Abhängigkeit des *Distance*-Parameters. Höhere *Distance*-Werte erlauben das Einfügen einer größeren Anzahl von Knoten, ohne zu den DeweyIDs eine weitere *Division* ergänzen zu müssen. Dass ein größerer *Distance*-Wert zu mehr potenziellen Einfügungen ohne Reorganisation führt, gilt nur für sequentielle Einfügevorgänge [Wa05a] aber nicht für das oben beschriebene Worst-case-Szenario. Hier gilt vielmehr, dass größere *Distance*-Werte die Kodierungslänge der neu vergebenen *Division*-Werte negativ beeinflussen und somit eine Erhöhung des *Distance*-Werts eine frühere Reorganisation erzwingen kann.

7.2 Benchmarks für XML-Datenbanksysteme

Dieser Abschnitt beschreibt zunächst einige allgemeine Eigenschaften, die ein Benchmark für Datenbanksysteme besitzen sollte. Danach werden die für XML-Systeme vorhandenen Benchmark-Ansätze vorgestellt. Da sich diese jedoch nicht für eine detaillierte Analyse der eingesetzten Sperrprotokolle in einem Datenbanksystem eignen, stellen wir schließlich ein eigenes Framework für die Durchführung von Benchmarks in verteilten Testumgebungen vor.

7.2.1 Vorhandene Benchmarks

Nach [Gr91] sollten Benchmarks für Datenbanksysteme im Allgemeinen die folgenden Eigenschaften besitzen, die wir auch für unser Framework in Abschnitt 7.2.2 berücksichtigen.

- Auslegung auf eine Anwendungsdomäne
  Der Anwendungsbereich des Benchmark sollte auf eine Domäne (wie bspw. ein Warenwirtschaftssystem) zugeschnitten sein.
- Relevanz
  Die einzelnen Transaktionen des Benchmark sollten typische Operationen bzgl. des Anwendungsbereichs durchführen, um aussagekräftige Resultate zu erzielen.
- Portabilität
  Der Benchmark sollte einfach auf allen zur Zeit gängigen Systemplattformen zu implementieren sein.
- Skalierbarkeit
  Es muss möglich sein, Ergebnisse für unterschiedlich große Konfigurationen zu erzeugen. Der Benchmark sollte auf einem Einzelplatzsystem genauso einsetzbar sein wie auf einem Großrechner.
- Einfachheit
  Der Benchmark sollte möglichst einfach gehalten werden, dass die Ergebnisse interpretierbar bleiben und nicht durch eine Vielzahl von Einflüssen verfälscht werden.

Zur Beschreibung der vorhandenen Ansätze für XDBMS-Leistungsmessungen verwenden wir zusätzlich die Klassifikation aus [BR02], worin XML-Benchmarks anhand folgender Eigenschaften gegeneinander abgegrenzt werden.

- Skopus
  Der *Skopus* eines Benchmark beschreibt, welche Teile des Datenbanksystems einer Leistungsmessung unterzogen werden. Meist ist die Gesamtleistung des Systems zu beurteilen, es kann jedoch auch die Beurteilung einzelner Komponenten (wie bspw. der Anfrageprozessor) aussagekräftige Ergebnisse liefern.
- Ein- und Mehrbenutzerbetrieb
  Benchmarks sollten auf die gleichzeitige Nutzung des Systems mit mehreren Benutzern zugeschnitten sein. Tests einzelner Komponenten können aber durchaus auch im Einbenutzerbetrieb erfolgen.
- Domäne
  Die *Domäne* für XML-Benchmarks nach [BR02] unterscheidet die Art der verwendeten XML-Dokumente nach Dokumentkollektionen, strukturierten Daten und Daten mit überwiegend gemischtem Inhaltsmodell.

XMach-1

XMach-1 [BR01] zählt zur Anwendungsdomäne der Web-Anwendungen und definiert zwei Dokumenttypen. Es existieren Dokumente, die Bücher beschreiben (dokumentenorientierter Inhalt), und Dokumente mit strukturierten Meta-Daten (datenorientierter Inhalt). Die zu verarbeitende Datenmenge für den Benchmark ist von 2 bis etwa 100 MB skalierbar.

Der Skopus von XMach-1 liegt auf dem gesamten Datenbanksystem. Dazu werden acht Anfragen und drei Änderungsoperationen definiert, die sowohl in einer textuellen Beschreibung als auch in XQuery angegeben sind. Die Änderungsoperationen beschränken sich allerdings nur auf das Einfügen und Löschen ganzer XML-Dokumente und die Modifikation der Dokument-Meta-Daten (URL und letzter Änderungszeitpunkt); Änderungen am Inhalt der Dokumente ist nicht vorgesehen.

XMark

XMark [SWK+02] ist am *National Research Institute for Mathematics and Computer Science* der Niederlande entstanden und wählt als Anwendungsdomäne ein XML-Dokument zur Verwaltung von Internet-Auktionsdaten. Das Benchmark-Dokument lässt sich über ein entsprechendes Generator-Werkzeug von 10 MB bis 10 GB skalieren. Die Domäne der XML-Daten ist sehr stark datenorientiert, zusätzliche beschreibende Texte im Dokument berücksichtigen jedoch auch eine zum Teil dokumentenorientierte Datenbasis.

Der XMark-Benchmark legt den Skopus auf den Anfrageprozessor, wofür 20 Anfragen definiert werden, die in [SWK+01] detailliert beschrieben und auch in einer XQuery-Variante angegeben sind. Die Anfragen sind jeweils im Einbenutzerbetrieb einzeln nacheinander auszuführen, sodass verschiedene Datenbanksysteme aufgrund der benötigten Ausführungszeiten miteinander verglichen werden können.

XOO7

Der XML-Benchmark XOO7 [BDL+01] basiert auf dem bereits für objekt-orientierte Datenbanksysteme spezifizierten Benchmark OO7 [CDN93]. Zur Erzeugung der Datenbasis wurde die in OO7 definierte Hierarchie von Bauteilen auf XML-Strukturen abgebildet. Dadurch ist ein einzelnes Benchmark-Dokument mit datenorientiertem Charakter entstanden. Der Skopus von XOO7 liegt wie bei XMark auf dem Anfrageprozessor des zu testenden Systems. Zu den Anfragen des OO7-Benchmark wurde typische XML-Funktionalität (wie z. B. Navigation) ergänzt, sodass insgesamt 13 Anfragetypen mit ausschließlichem Lesezugriff entstanden sind. Die Anfragen werden analog zu XMark einzeln nacheinander im Einbenutzerbetrieb durchgeführt und die Ausführungszeiten gemessen.

7.2.2 TaMix Benchmark-Framework

Da die vorgestellten Benchmarks für XML-Datenbanksysteme den Skopus fast ausschließlich auf den Anfrageprozessor legen und keine Änderungsoperationen innerhalb eines Dokuments definieren, haben wir mit *TaMix* [Lu05] ein eigenes Framework zur Leistungsmessung des XTC-Servers entwickelt. TaMix erlaubt die Implementierung beliebiger Transaktionstypen, die in einer verteilten Testumgebung automatisiert ausgeführt und visualisiert werden.

Abbildung 128 zeigt die TaMix-Infrastruktur. Der XTC-Server und die Clients, die die Transaktionslast erzeugen, laufen jeweils auf verschiedenen Rechnern, die über ein Netzwerk miteinander verbunden sind. Auf dem Server-Rechner läuft zusätzlich ein *Koordinator*, der die Durchführung der gesamten Messung steuert.

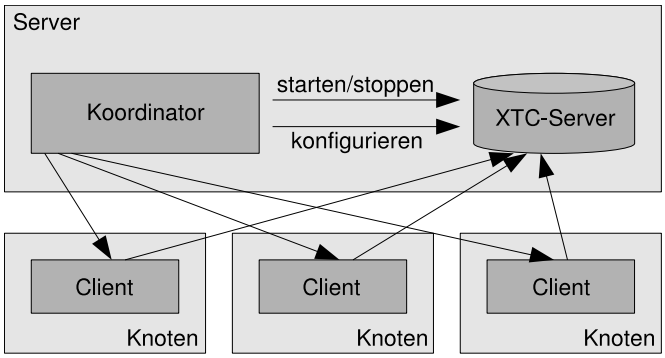


Abbildung 128: TaMix-Infrastruktur

Die Einrichtung eines Benchmark beginnt zunächst mit dem Erzeugen eines Benchmark-Dokuments, das im XTC-Server gespeichert wird. Danach werden für dieses Dokument verschiedene Transaktionstypen implementiert. Eine Transaktion verbindet sich in der Regel zum Datenbankserver, greift auf das Benchmark-Dokument zu und führt eine Folge von lesenden und schreibenden DOM-Operationen aus, die typische XML-Zugriffsmuster repräsentieren. Die Transaktionstypen werden anschließend zusammen mit dem XTC-Treiberpaket (siehe Abschnitt 6.2) auf die Client-Rechner kopiert.

Die Parametrisierung des Benchmark erfolgt beim Koordinator mit einer *Koordinator-Konfigurationsdatei*. Hier werden die auszuführenden Transaktionstypen, die Dauer und Anzahl der Durchläufe (sog. *Runs*) und die zu untersuchenden Isolationsstufen und Sperrtiefen angegeben. Weil das TaMix-Framework hauptsächlich entwickelt wurde, um verschiedene Ansätze der Transaktionsisolation miteinander zu vergleichen, verwaltet der Koordinator zusätzlich für den XTC-Server eine Menge von *Server-Konfigurationsdateien,* die sich nur in der Angabe des zu verwendenden Sperrprotokolls unterscheiden. Die Umsetzung von Sperranforderungen auf konkrete Sperrmodi eines Sperrverfahrens erfolgt im XTC-Server durch das in Abschnitt 6.1.5 beschriebene Konzept des Meta-Locking.

Zu Beginn eines Benchmark erstellt der Koordinator zunächst eine Sicherung der XTC-Server Containerdateien, damit jeder Durchlauf mit gleichem Datenzustand gestartet werden kann. Danach leitet der Koordinator die eigentliche Messung ein. Dazu werden mit entsprechenden Schleifendurchläufen die einzelnen Runs für jedes Sperrprotokoll, jede Isolationsstufe und jede Sperrtiefe erzeugt. Für jeden Run spielt der Koordinator die Sicherung des XTC-Servers wieder ein, kopiert die Server-Konfigurationsdatei für das zu untersuchende Sperrverfahren zum Server und startet schließlich die XTC-Server-Instanz. Danach werden den Clients die auszuführenden Transaktionstypen, die Isolationsstufe und Sperrtiefe für den aktuellen Durchlauf mitgeteilt. Jeder Client erzeugt daraufhin für jeden Transaktionstyp einen *Slot.* Während des Benchmark-Laufs wird für jeden Slot eine Transaktion des entsprechenden Typs gestartet und abgearbeitet. Ist die Transaktion beendet (erfolgreich oder aufgrund eines Deadlock vom XDBMS zurückgesetzt), so wird für den Slot sofort eine neue Transaktion gestartet. Somit produziert jeder Client während eines Laufs eine konstante Last, da für jeden Slot immer eine Transaktion des spezifizierten Typs läuft.

Ist die eingestellte Zeit für einen Durchlauf abgelaufen, so teilt jeder Client dem Koordinator die Anzahl der erfolgreich abgeschlossenen und zurückgesetzten Transaktionen mit, und der Koordinator startet mit dem Einspielen der XTC-Server-Sicherung den nächsten Run. Nach der Abarbeitung aller Durchläufe generiert der Koordinator Diagramme zur graphischen Darstellung der Messergebnisse.

7.3 Transaktionsisolation

Im Folgenden werden detailliert die in Kapitel 5 beschriebenen Protokolle zur Isolation nebenläufiger Transaktionen mit dem oben beschriebenen TaMix Benchmark-Framework untersucht. Dazu vergleicht Abschnitt 7.3.1 zunächst den Transaktionsdurchsatz (sowohl erfolgreiche als auch abgebrochene Transaktionen) bei Anwendung der einzelnen Sperrprotokolle für verschiedene Zugriffsmuster. Abschnitt 7.3.2 geht näher auf den Einfluss der gewählten Isolationsstufe ein und beurteilt somit auch den Aufwand für die Sperrverwaltung.

Als Datenbasis, auf der die Benchmarks durchgeführt werden, erweitern wir das in Kapitel 2 eingeführte Beispieldokument *bank.xml* durch weitere Elemente für *Konto*. Wir ergänzen den *Kontostand*, umschlossen vom Element *Daueraufträge* eine Liste von *Dauerauftrag*-Elementen und analog dazu eine Liste von *Protokoll*-Informationen (z. B. Vermerk der Erstellung eines Kontoauszugs) und durchgeführte *Buchungen*. Das resultierende Benchmark-Dokument, dessen DTD in Abbildung 129 dargestellt ist, wird für 1.000 Kunden und 2.500 Konten erstellt und umfasst eine Größe von etwa 8 MB und 580.000 einzelnen taDOM-Knoten.

```
<!ELEMENT Bank (Kunden,Konten)>

<!ELEMENT Kunden (Kunde*)>
<!ELEMENT Kunde (Name,Adresse)>
<!ATTLIST Kunde id ID #REQUIRED>
<!ELEMENT Name (Vorname+,Nachname)>
<!ELEMENT Vorname (#PCDATA)>
<!ELEMENT Nachname (#PCDATA)>
<!ELEMENT Adresse (Straße,Hausnummer?,PLZ,Ort)>
<!ELEMENT Straße (#PCDATA)>
<!ELEMENT Hausnummer (#PCDATA)>
<!ELEMENT PLZ (#PCDATA)>
<!ELEMENT Ort (#PCDATA)>

<!ELEMENT Konten (Konto*)>
<!ELEMENT Konto (Kontostand,Dispo,Daueraufträge,Protokolle,Buchungen)>
<!ATTLIST Konto id       ID     #REQUIRED
                Besitzer IDREFS #REQUIRED>
<!ELEMENT Kontostand (#PCDATA)>
<!ELEMENT Dispo (#PCDATA)>

<!ELEMENT Daueraufträge (Dauerauftrag*)>
<!ELEMENT Dauerauftrag (Tag,Empfänger,Kontonummer,
                        BLZ,Betrag,Verwendungszweck)>
<!ELEMENT Tag (#PCDATA)>
<!ELEMENT Empfänger (#PCDATA)>
<!ELEMENT Kontonummer (#PCDATA)>
<!ELEMENT BLZ (#PCDATA)>
<!ELEMENT Betrag (#PCDATA)>
<!ELEMENT Verwendungszweck (#PCDATA)>

<!ELEMENT Protokolle (Protokoll*)>
<!ELEMENT Protokoll (#PCDATA)>

<!ELEMENT Buchungen (Buchung*)>
<!ELEMENT Buchung (#PCDATA)>
```

Abbildung 129: DTD des Benchmark-Dokuments

Alle Messungen werden mit dem TaMix-Framework in einem Linux Cluster mit vier Rechnern (ein Rechner für das XDBMS und den Koordinator, drei Rechner mit je einem TaMix-Client) durchgeführt, die in einem eigenen Sub-Netz über ein 100 MBit-Netzwerk miteinander verbunden sind. Die Rechner sind jeweils mit 256 MB Arbeitsspeicher und einem *Intel Celeron* Prozessor mit 1,7 GHz ausgestattet. Jeder Messpunkt in den Ergebnisgrafiken wird mit drei Durchläufen mit jeweils fünf Minuten Laufzeit ermittelt.

7.3.1 Analyse der Sperrprotokolle

Für die Analyse der Sperrprotokolle beschreiben wir verschiedene Transaktionstypen, die für die XML-Verarbeitung typische Zugriffsmuster enthalten. Wir diskutieren die Messergebnisse jeweils für einen Mix parallel laufender Transaktionen, die das jeweilige Zugriffsmuster implementieren. Abschließend wird der Transaktionsdurchsatz für die gleichzeitige Verarbeitung aller beschriebenen Muster untersucht.

Navigation

Der erste Transaktionstyp der *Überweisung* besteht aus einer *Folge von Navigationsschritten* zur Ermittlung von Daten aus dem Benchmark-Dokument, die anschließend als Basis für einen geringen Anteil von Änderungsoperationen dienen. Dazu wird zunächst zufällig ein *Konto*-Element ausgewählt und alle dessen Kinder werden bestimmt. Danach wird der Kontostand und der Dispo-Wert gelesen. Überschreitet ein zufällig gewählter Überweisungsbetrag nicht das Guthaben (mit Berücksichtigung des Dispokredits), so erfolgt eine Überweisung. In diesem Fall wird ein neuer Kontostand geschrieben und eine Buchung für die Überweisung angelegt. Ist dagegen der Überweisungsbetrag zu hoch, so wird lediglich ein Protokolleintrag über die zurückgewiesene Überweisung erzeugt. In beiden Fällen gilt die Transaktion aus Sicht des XDBMS als erfolgreich abgeschlossen. Jeder TaMix-Client führt fünf Slots des Transaktionstyps aus, sodass eine konstante Last von 15 parallel laufenden Transaktionen gehalten wird.

Abbildung 130 stellt die Anzahl der erfolgreich beendeten Transaktionen für die untersuchten Sperrprotokolle in Abhängigkeit der Sperrtiefe dar. Für die Tiefen 0 und 1 zeigen die hierarchischen Protokolle einen sehr geringen Durchsatz, was durch die hohe Rücksetzrate aufgrund der Sperrkonflikte zu erklären ist (siehe Anzahl der abgebrochenen Transaktionen in Abbildung 131). Dabei erzielen Protokolle mit einer Update-Sperre (taDOM und URIX) einen höheren Durchsatz und eine geringere Rücksetzrate. Ab der Sperrtiefe 2 kommt es zu fast keiner Rücksetzung mehr, da eine Blockierung nur noch dann auftritt, wenn zwei oder mehr Transaktionen aus den 2.500 Konten zufällig dasselbe für eine Überweisung auswählen.

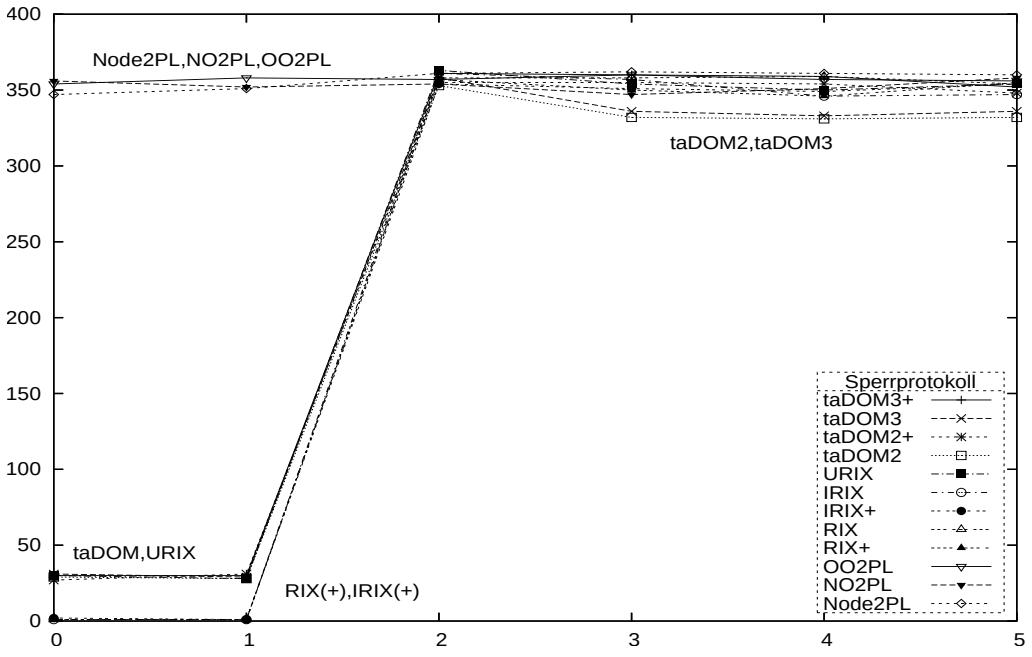


Abbildung 130: Erfolgreich beendete Überweisungen

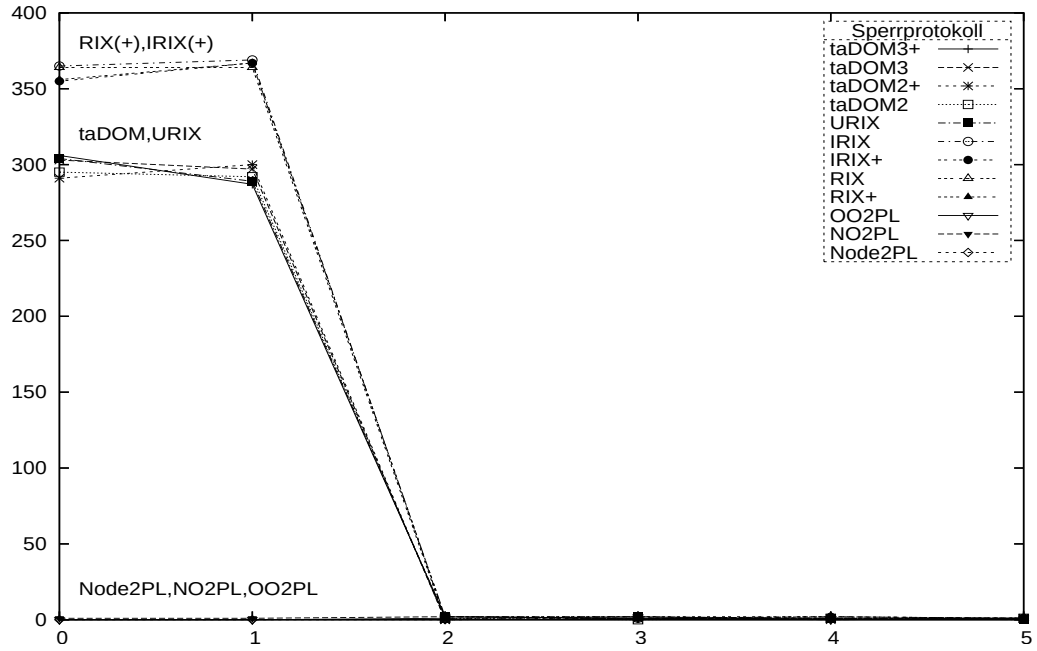


Abbildung 131: Abgebrochene Überweisungen

Die rein knotenbasierten Protokolle Node2PL, NO2PL und OO2PL unterstützen aufgrund der fehlenden Sperrhierarchien keine einstellbare Sperrtiefe und liegen bei den hier betrachteten Navigationsmustern mit geringem Änderungsanteil durchweg im oberen Leistungsbereich.

Der Transaktionsdurchsatz fällt für die beiden Protokolle taDOM2 und taDOM3 ab einer Sperrtiefe von 3 leicht ab, weil für die Bestimmung der Kindknoten des *Konto*-Elements eine Ebenensperre angefordert wird, für die ab Sperrtiefe 3 bei Änderungsoperationen im darunter liegenden Teilbaum zusätzliche Sperren auf den Kindern von *Konto* während der Sperrkonversion angefordert werden müssen (siehe Abschnitt 5.1). Dieser Effekt ist beim folgenden Transaktionstyp noch deutlicher zu erkennen.

Ermitteln aller Kindknoten (Auswertung der *Child*-Achse)

Dieser Transaktionstyp wählt zunächst wieder zufällig ein Konto aus, navigiert zum *Daueraufträge*-Element und *bestimmt alle Kindknoten* (*Dauerauftrag*-Elemente). Für die taDOM-Protokolle erfordert diese Operation eine Ebenensperre auf dem *Daueraufträge*-Element; für die übrigen Protokolle wird für jedes Kind eine separate Knotenlesesperre angefordert. Danach wird zufällig eines der ermittelten *Dauerauftrag*-Elemente ausgewählt und der Wert des Überweisungstags geändert. Auch für diesen Transaktionstyp wird eine Gesamtlast von 15 Transaktionen (fünf Transaktionen pro TaMix-Client) erzeugt.

Abbildung 132 zeigt die Anzahl der erfolgreich modifizierten Daueraufträge in Abhängigkeit der Sperrtiefe. Bei Sperranforderungen auf Ebene 0 und 1 zeigt sich analog zum vorherigen Transaktionstyp ein geringer Durchsatz, wobei die Protokolle mit einer Update-Sperre wieder leicht im Vorteil liegen. Ab Sperrtiefe 2 kollidieren die auszuführenden Transaktionen nicht mehr beim Zugriff auf die *Konto*-Elemente und der Durchsatz steigt für die hierarchischen Protokolle auf das Maximum an. Während die beiden Protokolle taDOM3+ und taDOM2+ den Transaktionsdurchsatz bis zur Sperrtiefe 5 auf diesem Niveau halten, nimmt der Durchsatz bei taDOM3 und taDOM2 wegen der Modifikationsoperation und des dadurch bedingten nachträglichen Zugriffs auf alle Kindknoten für die Konversion der Ebenensperre auf dem *Daueraufträge*-Element stark ab.

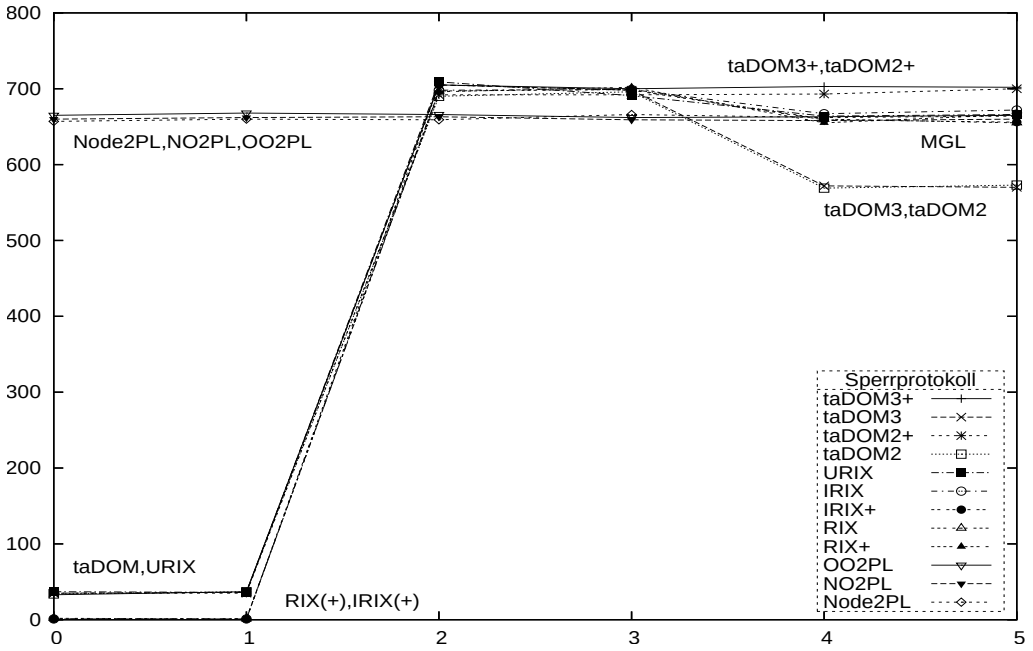


Abbildung 132: Erfolgreich modifizierte Daueraufträge

Die übrigen hierarchischen Protokolle (*multi-granularity locking – MGL*) zeigen ebenfalls einen Rückgang des Transaktionsdurchsatzes, der jedoch nicht so stark ausfällt. Das liegt daran, dass diese Protokolle das Sperren einer Teilbaumebene nicht unterstützen und somit ab Sperrtiefe 3 für jeden *Dauerauftrag*-Knoten ein separater Methodenaufruf beim Sperrmanager zur Anforderung einer Sperre erforderlich ist.

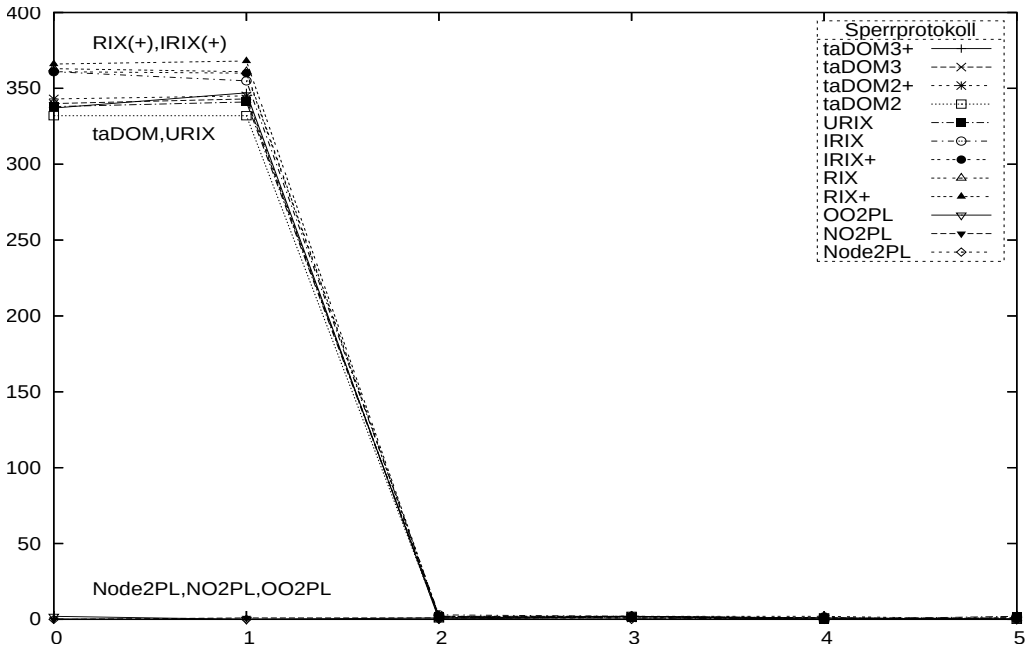


Abbildung 133: Abgebrochene Änderungen von Daueraufträgen

Den für die Sperrtiefen 4 und 5 erreichten Transaktionsdurchsatz der MGL-Protokolle erzielen auch Node2PL, NO2PL und OO2PL. Dies gilt analog zum ersten Transaktionstyp wiederum unabhängig der Sperrtiefe, weil die Protokolle dieses Konzept nicht unterstützen.

Die Anzahl der abgebrochenen Transaktionen durch Deadlocks zeigt Abbildung 133. Der Einsatz der Protokolle Node2PL, NO2PL und OO2PL führt zu fast keinen Rücksetzungen (mit der seltenen Ausnahme zweier zufällig identisch gewählter Konten). Ab einer Sperrtiefe von 2 weisen auch die hierarchischen Protokolle dieses Verhalten auf.

Umbenennen von Elementen

Mit diesem Transaktionsmix werden die Auswirkungen von *Elementumbenennungen* untersucht. Dazu wird ein erster Transaktionstyp definiert, der das erste Kind des *Bank*-Elements lädt, und dieses abhängig von seinem Wert umbenennt: Hat das Element den Namen *Kunden*, so wird es in *Kundenstamm* umbenannt; hat es den Namen *Kundenstamm*, so erfolgt eine Umbenennung zurück nach *Kunden*. Ein zweiter Transaktionstyp greift parallel auf ein zufällig ausgewähltes *Kunde*-Element zu und rekonstruiert vollständig das so bestimmte Fragment. Für die Durchführung dieses Benchmark-Laufs werden pro TaMix-Client ein Slot mit dem ersten und vier Slots mit dem zweiten Transaktionstyp belegt.

Abbildung 134 zeigt die Gesamtanzahl aller erfolgreich durchgeführten Transaktionen für diesen Mix. Die hierarchischen Protokolle erreichen bei den Sperrtiefen 0 und 1 einen jeweils ähnlichen Durchsatz, der durch die Update-Sperre bei den taDOM-Protokollen und URIX wieder etwas höher liegt. Da die MGL-Protokolle keine Trennung zwischen der Struktur des Dokuments und dem Wert einzelner Knoten vornehmen (und somit auch keine Sperre zum Ändern eines einzelnen Knotens ohne Auswirkung auf den darunter liegenden Teilbaum anbieten), resultiert eine Elementumbenennung immer in einer exklusiven Teilbaumsperre. Damit führt eine Erhöhung der Sperrtiefe nicht zu einer Steigerung des Transaktionsdurchsatzes, weil die Sperrkonflikte zwischen den Änderungs- und Lesetransaktionen unverändert auf dem *Bank*- und ab Sperrtiefe 1 stets auf dem *Kunden*- bzw. *Kundenstamm*-Element stattfinden.

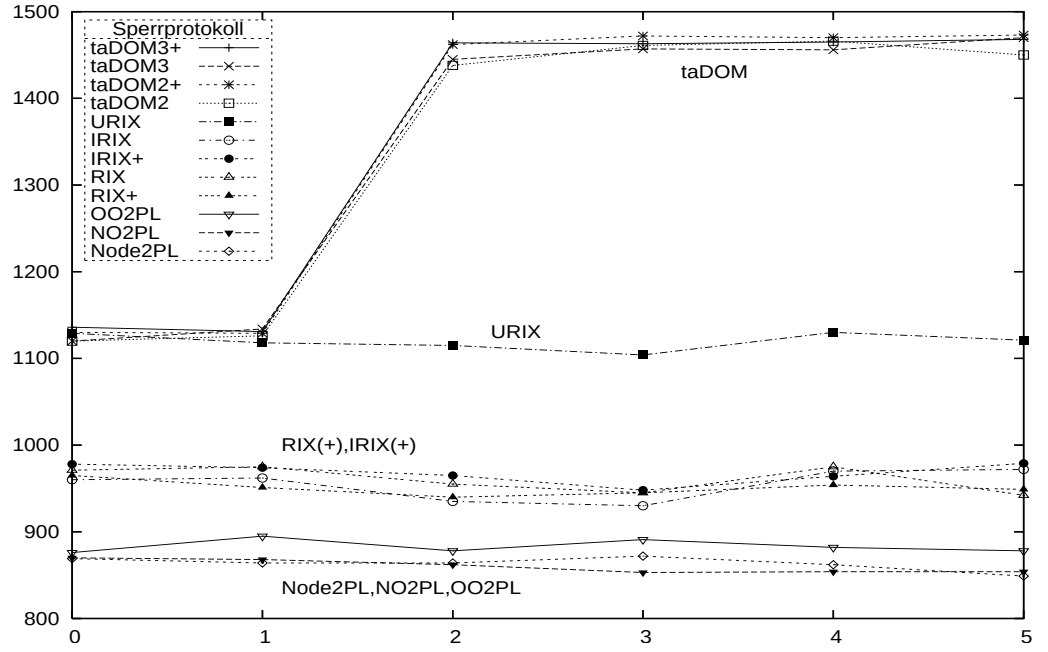


Abbildung 134: Erfolgreiche Transaktionen während der Umbenennung

Die taDOM-Protokolle steigern den Transaktionsdurchsatz ab Sperrtiefe 2 dagegen sehr stark, weil durch das Konzept der virtuellen Namensknoten bei taDOM2(+) bzw. eine explizite Exklusivsperre für einzelne Knoten bei taDOM3(+) ein umzubenennender Knoten ohne Auswirkungen auf das darunter liegende Fragment gesperrt werden kann (siehe Kapitel 5). Somit beeinflusst die Umbenennung des *Kunden*-Elements nicht den direkten Einsprung auf *Kunde*-Elemente und alle Transaktionen können ohne Blockierungen ausgeführt werden.

Prinzipiell ist diese blockierungsfreie Umbenennung auch mit den Protokollen Node2PL, NO2PL und OO2PL möglich, allerdings zeigen diese Protokolle enorme Nachteile bei den parallel laufenden Rekonstruktionsoperationen. Weil die Protokolle ausschließlich auf Knotenbasis operieren, unterstützen sie keine Teilbaumsperren. Somit kann ein Fragment nicht mit Anforderung einer einzelnen Sperre und einem sequentiellen Seiten-Scan rekonstruiert werden, sondern es ist eine schrittweise Navigation mit einer Sperranforderung für jeden einzelnen Knoten des Fragments erforderlich. Dieses Phänomen wird beim nächsten Transaktionstyp, der sich ausschließlich mit der Fragmentrekonstruktion beschäftigt, noch deutlicher sichtbar.

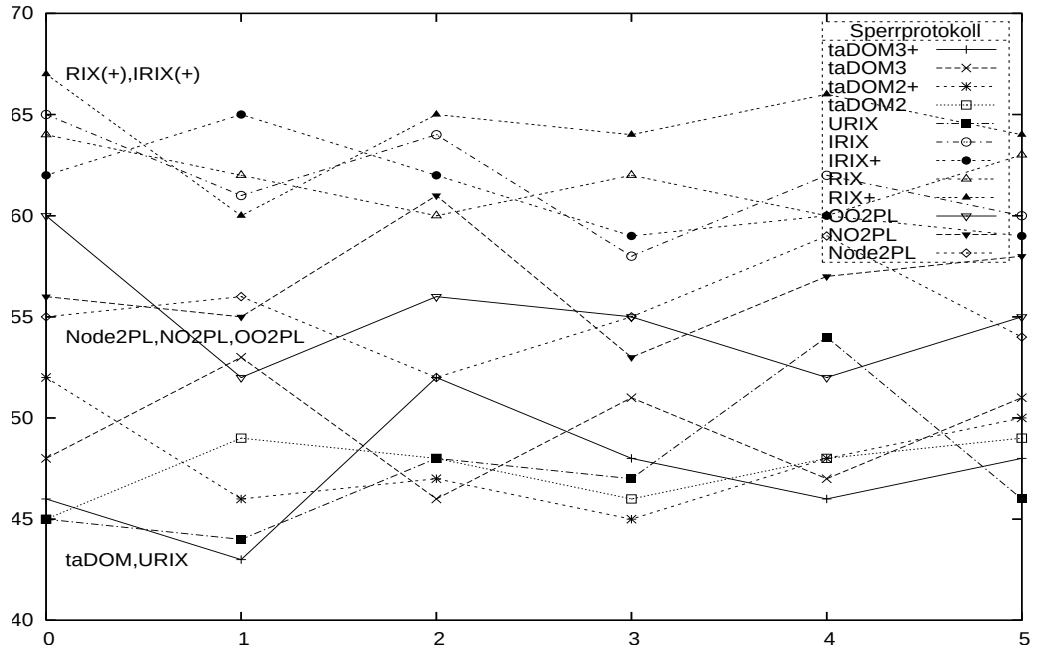


Abbildung 135: Abgebrochene Transaktionen während der Umbenennung

Abbildung 135 zeigt in Abhängigkeit der Sperrtiefe die Anzahl aller zurückgesetzten Transaktionen während der Umbenennung des *Kunden*-Elements. Aufgrund des geringen Wertebereichs der abgebrochenen Transaktionen wirkt das Messergebnis auf den ersten Blick nicht besonders aussagekräftig. Bei genauerer Betrachtung kristallisieren sich jedoch drei Bereiche für die Protokolle RIX(+)/IRIX(+), Node2PL/NO2PL/OO2PL und taDOM/URIX heraus. Die Rücksetzraten verändern sich bzgl. der Sperrtiefe jedoch kaum, weil die Abbrüche hauptsächlich die Änderungstransaktionen betreffen, die mit Ausnahme der gewählten Sperrtiefe von 0 immer auf Ebene 1 stattfinden. Der Großteil der Transaktionslast wird von den Rekonstruktionen verursacht, die durch die ausgelösten Wartebeziehungen nur eine längere Ausführungszeit benötigten, aber keine Deadlocks verursachen.

Rekonstruktion von Fragmenten

Ein weiteres häufig durchzuführendes Zugriffsmuster während der XML-Verarbeitung ist das *Rekonstruieren von Fragmenten*. Wir definieren dazu einen Transaktionstyp, der einen Kontoauszug erstellt. Dafür wird zunächst ein zufällig ausgewähltes *Konto*-Fragment vollständig rekonstruiert und danach ein Protokolleintrag erzeugt, in dem die Erstellung des Kontoauszugs mit dem aktuellen Datum vermerkt wird. Jeder TaMix-Client führt fünf dieser Transaktionstypen parallel aus.

Abbildung 136 zeigt in Abhängigkeit der Sperrtiefe die Anzahl der erfolgreich durchgeführten Transaktionen. Aufgrund des zunächst lesenden und anschließend schreibenden Zugriffs kommt beim Einsatz eines hierarchischen Protokolls bei den Sperrtiefen 0 und 1 fast keine Transaktion erfolgreich zum Ende. Ab Sperrtiefe 2 steigt der Durchsatz enorm an, da fast keine Blockierung mehr auftritt (u. U. wählen zwei Transaktionen dasselbe Konto und blockieren sich gegenseitig bei der Erzeugung des Protokolleintrags). Ab Sperrtiefe 3 sinkt der Transaktionsdurchsatz bei taDOM2, taDOM3 und IRIX+ leicht ab, weil bei diesen Protokollen die Konversion der Teilbaumlesesperre beim Schreiben des Protokolleintrags durch separate Sperren auf den Kindknoten des *Konto*-Elements ersetzt werden muss.

Die Protokolle Node2PL, NO2PL und OO2PL zeigen bei der Rekonstruktion größerer Fragmente durch die schon beim vorherigen Zugriffsmuster *Umbenennung* beschriebene Problematik einen noch stärkeren Einbruch des Transaktionsdurchsatzes. Da die Protokolle keine Teilbaumsperren bereitstellen, muss jeder Knoten eines Fragments einzeln gesperrt werden, wenn parallele Einsprünge auf beliebig indexierte Knoten möglich sein sollen (die in Abschnitt 3.3.2 beschriebenen IDR- und IDX-Sperren ermöglichen lediglich den Einsprung auf Elemente mit ID-Attributen). Dies hat zur Folge, dass ein Fragment nicht mit Hilfe einer einzigen Teilbaumsperre und einem anschließenden Seiten-Scan rekonstruiert werden kann, sondern das gesamte Fragment mit einzelnen Navigationsschritten und Sperranforderungen traversiert werden muss.

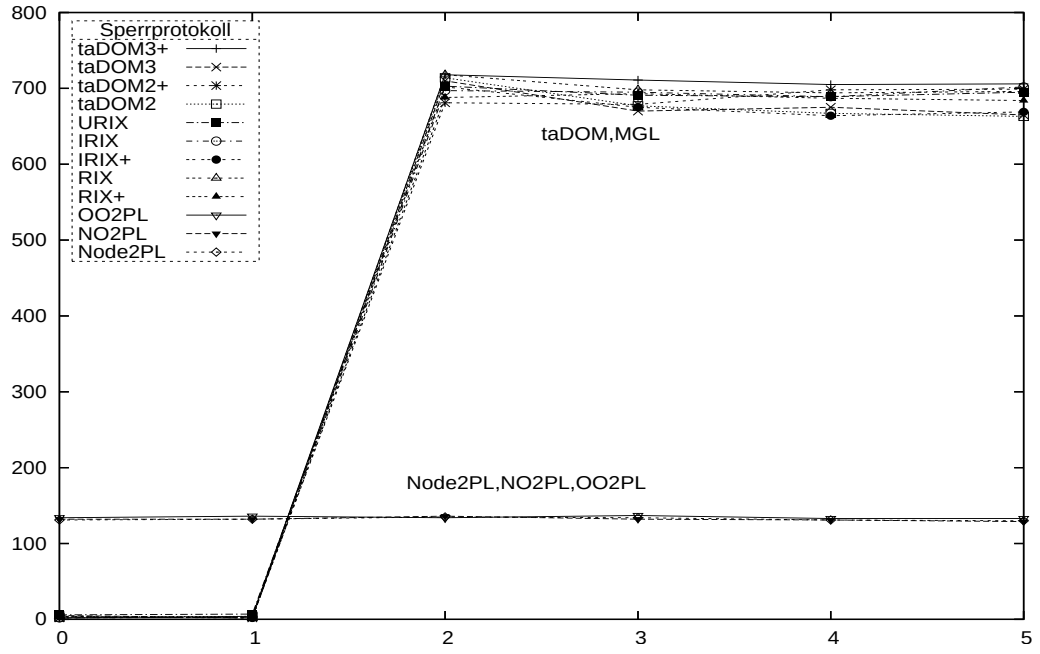


Abbildung 136: Erfolgreiche Transaktionen während der Rekonstruktion

Die Anzahl der abgebrochenen Transaktionen in Abbildung 137 entspricht dem erwarteten Verhalten. Mit den hierarchischen Protokollen wird bei einer Sperrtiefe von 0 oder 1 annähernd jede Transaktion abgebrochen, ab einer Sperrtiefe von 2 treten fast keine Rücksetzungen mehr auf (wie immer mit der Ausnahme, dass zwei Transaktionen zufällig dasselbe Konto wählen). Dies gilt auch für Protokolle Node2PL, NO2PL und OO2PL.

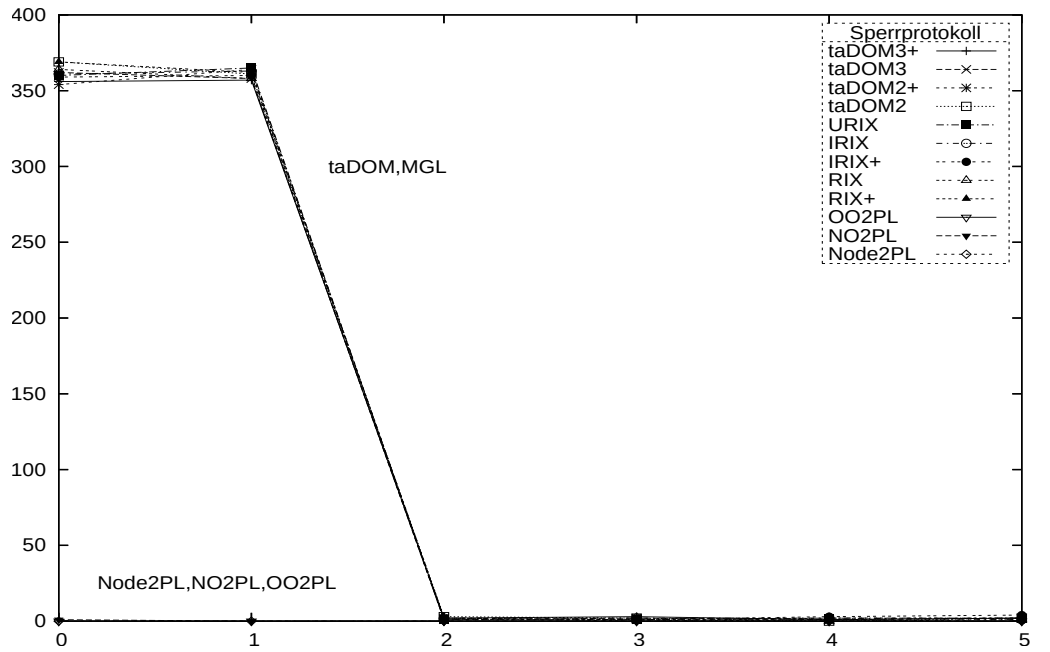


Abbildung 137: Abgebrochene Transaktionen während der Rekonstruktion

Löschen von Fragmenten

Als letztes Zugriffsmuster untersuchen wir das Löschen von Fragmenten in einem XML-Dokument. Dazu wird ein Transaktionstyp definiert, der einen zufällig ausgewählten *Kunde*-Knoten liest, und danach das gesamte Fragment aus dem Dokument entfernt. Weil das Löschen von Fragmenten durch die Reorganisation der B*-Bäume eine höhere Transaktionslast zur Folge hat, werden pro TaMix-Client nur zwei Slots für diesen Transaktionstyp erzeugt, sodass insgesamt sechs Transaktionen kontinuierlich Fragmente entfernen.

Abbildung 138 zeigt die Anzahl der erfolgreichen Transaktionen. Ab einer Sperrtiefe von 2 werden die exklusiven Teilbaumsperren auf den *Kunde*-Elementen angefordert, sodass in diesem Bereich der höchste Transaktionsdurchsatz erzielt wird. Für die Sperrtiefen 0 und 1 profitieren die taDOM-Protokolle und das URIX-Protokoll wieder von der Update-Sperre. Den Sperrprotokollen NO2PL und OO2PL fehlt zum Erreichen des maximalen Durchsatzes wieder die Möglichkeit einer Teilbaumsperre, sodass ein Fragment nicht durch eine einzelne Sperre für den exklusiven Zugriff reserviert werden kann, sondern vor dem eigentlichen Löschvorgang eine Traversierung des Fragments zum exklusiven Sperren jedes Knotens erfolgen muss. Da das Löschen eines Knotens jedoch deutlich aufwändiger als eine Navigationsoperation ist, fällt der Unterschied des Transaktionsdurchsatzes zu den hierarchischen Protokollen nicht so stark wie beim Zugriffsmuster der reinen Fragmentrekonstruktion aus.

Beim Einsatz des Protokolls Node2PL wird fast jede Transaktion zurückgesetzt, da die Sperranforderungen für alle *Kunde*-Elemente auf dem Elternknoten *Kunden* kollidieren.

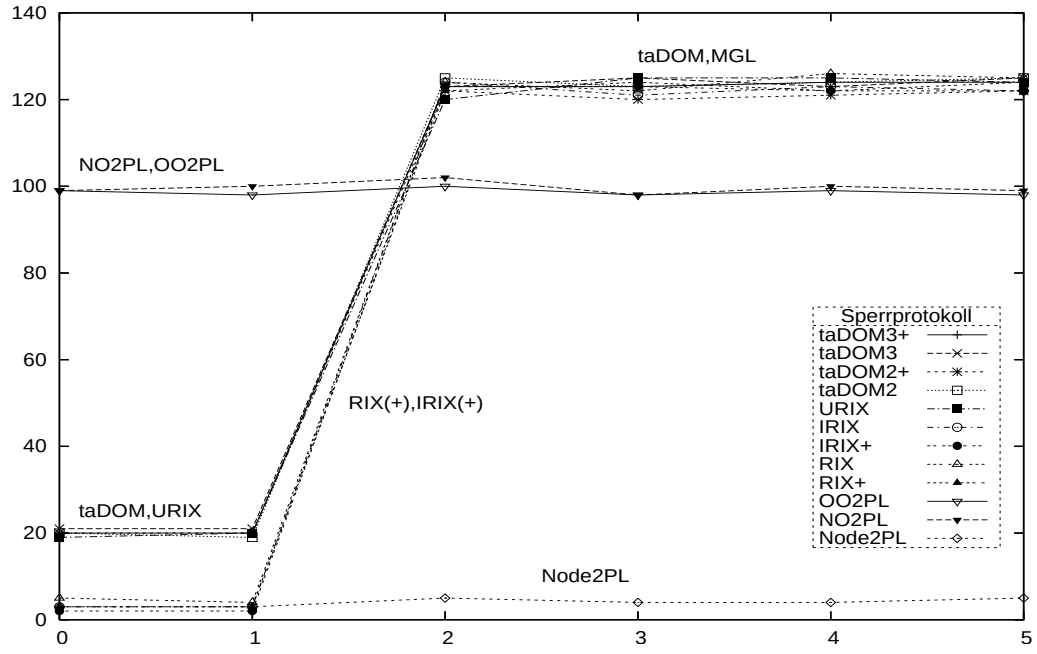


Abbildung 138: Erfolgreich gelöschte Fragmente

Entsprechend umgekehrt zur Anzahl der erfolgreich abgeschlossenen Transaktionen verhält sich die Rücksetzrate in Abbildung 139. Bei NO2PL, OO2PL und ab Sperrtiefe 2 bei den hierarchischen Protokollen kommt es zu keinem Abbruch mehr, wogegen bei Node2PL durchweg fast alle Transaktionen abgebrochen werden.

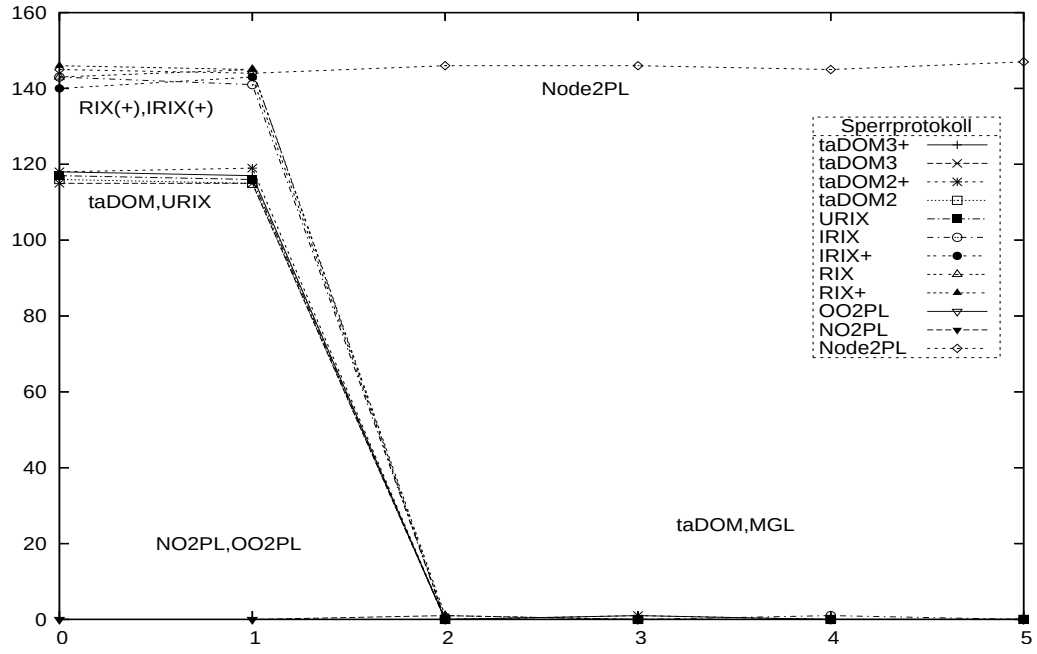


Abbildung 139: Abgebrochene Transaktionen bei Löschen von Fragmenten

Transaktionsmix

In einer letzten Vergleichsmessung werden in einem gemeinsamen Transaktionsmix alle beschriebenen Transaktionstypen zusammen gestartet (insgesamt 22 Slots pro TaMix-Client). Um die Varianz der Messergebnisse bei dieser höheren Last möglichst gering zu halten, wird für jeden Messpunkt (d. h. jedes Protokoll und jede Sperrtiefe) der Durchschnittswert aus fünf Benchmark-Läufen ermittelt. Abbildung 140 stellt die Anzahl aller erfolgreich abgeschlossenen Transaktionen für den gemeinsamen Mix in Abhängigkeit der Sperrtiefe dar.

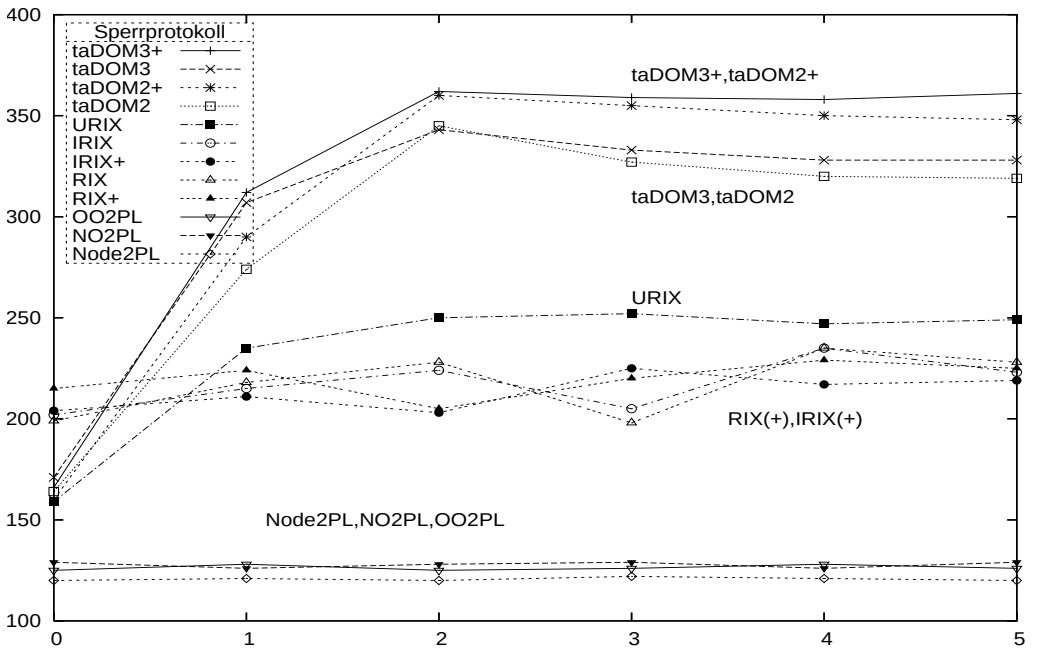


Abbildung 140: Erfolgreich beendete Transaktionen im Mix

Mit den taDOM-Protokollen und dem URIX-Protokoll nehmen die erfolgreich beendeten Transaktionen bis zur Sperrtiefe 2 stetig zu; ab dieser Tiefe treten fast keine Blockierungen mehr auf. Dabei erzielen die taDOM-Protokolle mit den speziell auf die XML-typischen Zugriffsmuster auslegten Sperrmodi einen deutlich höheren Durchsatz als das URIX-Protokoll. Beim Vergleich der taDOM-Protokolle schneiden die „Plus-Versionen“ jeweils besser ab, weil während der Sperrkonversionen keine nachträglichen Zugriffe auf das gespeicherte Dokument erforderlich sind, sondern spezielle Sperrmodi bereit gestellt werden. Die beiden Protokolle taDOM3+ und taDOM3 erreichen gegenüber den Versionen taDOM2+ und taDOM2 eine zusätzliche Steigerung, weil durch weitere Sperrmodi und den Verzicht auf virtuelle Namensknoten während des gesamten Benchmark weniger Sperranforderungen durchgeführt werden müssen. Eine Sperranforderung benötigt jedoch weniger Rechenzeit als ein Dokumentzugriff, sodass das Protokoll taDOM2+ besser abschneidet als taDOM3.

Die hierarchischen Protokolle RIX(+) und IRIX(+) zeigen ein interessantes Verhalten, da scheinbar unabhängig von der gewählten Sperrtiefe ein gleichmäßiger Transaktionsdurchsatz erzielt wird. Bei der genauen Analyse der jeweils durchgeführten Transaktionstypen zeigt sich jedoch, dass bei den Sperrtiefen 0 und 1 fast ausschließlich die Rekonstruktion von *Kunde*-Fragmenten aus dem Zugriffsmuster *Elementumbenennung* erfolgt, während nahezu alle Änderungstransaktionen durch eine fehlende Update-Sperre zurückgesetzt werden. Ab der Sperrtiefe 2 gleichen sich Lese- und Schreibtransaktionen wieder aus; der Gesamtdurchsatz

bleibt jedoch gleich und beträgt etwa 60% des Protokolls taDOM3+. Die Optimierung von RIX und IRIX zu RIX+ und IRIX+ lassen im Mix aller Transaktionstypen keinen messbaren Unterschied erkennen.

Node2PL, NO2PL und OO2PL erreichen eine gleichmäßige Verteilung von Lese- und Schreibtransaktionen, allerdings erzielen die Protokolle durch die für die jeweiligen Transaktionstypen zuvor diskutierten Nachteile nur etwa 40% des Durchsatzes der taDOM-Protokolle.

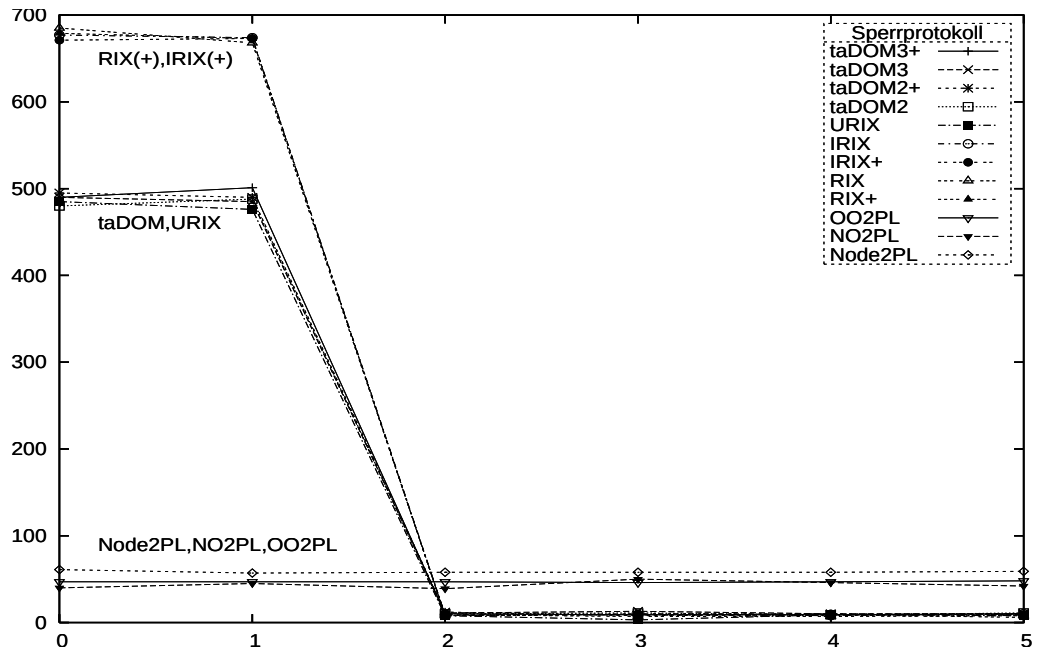


Abbildung 141: Abgebrochene Transaktionen im Mix

Die Anzahl der abgebrochenen Transaktionen im Gesamtmix ist in Abbildung 141 dargestellt. Die Protokolle Node2PL, NO2PL und OO2PL zeigen eine eher geringe Rücksetzrate, trotzdem verhindert die fehlende Sperrfunktionalität einen hohen Transaktionsdurchsatz. Der Einsatz eines hierarchischen Protokolls führt ab Sperrtiefe 2 zu einer geringen Anzahl von Rücksetzungen; bei den Sperrtiefen 0 und 1 mit höheren Kollisionsraten sind Verfahren mit einer Update-Sperre im Vorteil.

7.3.2 Analyse der Isolationsstufen

Abschließend untersuchen wir den Einfluss der gewählten Isolationsstufe auf den Transaktionsdurchsatz. Da unser prototypisches XML-Datenbanksystem momentan noch keine XQuery-Auswertung mit dem Zugriff auf Indexstrukturen unterstützt, sind mit den oben beschriebenen Zugriffsmustern alle zur Zeit typischen Operationsfolgen abgedeckt. Da bei der reinen Navigation keine Phantome auftreten können (siehe Abschnitt 5.8), stellt *repeatable* für unsere Messungen die höchste Isolationsstufe dar.

Für eine Analyse des Einflusses der Konsistenzstufen auf den Transaktionsdurchsatz führen wir den oben beschriebenen Transaktionsmix aller Transaktionstypen mit dem Sperrprotokoll taDOM3+ aus und variieren die Isolationsstufe für die zu startenden Transaktionen bei den jeweiligen Benchmark-Läufen. Da sich mit der Isolationsstufe *none* jegliche Sperranforderungen

abschalten lassen[6], kann auf diese Weise der Aufwand innerhalb des Systems zur Sicherstellung einer konsistenten Datenbasis abgeschätzt werden. Abbildung 142 zeigt das Ergebnis dieser Messung.

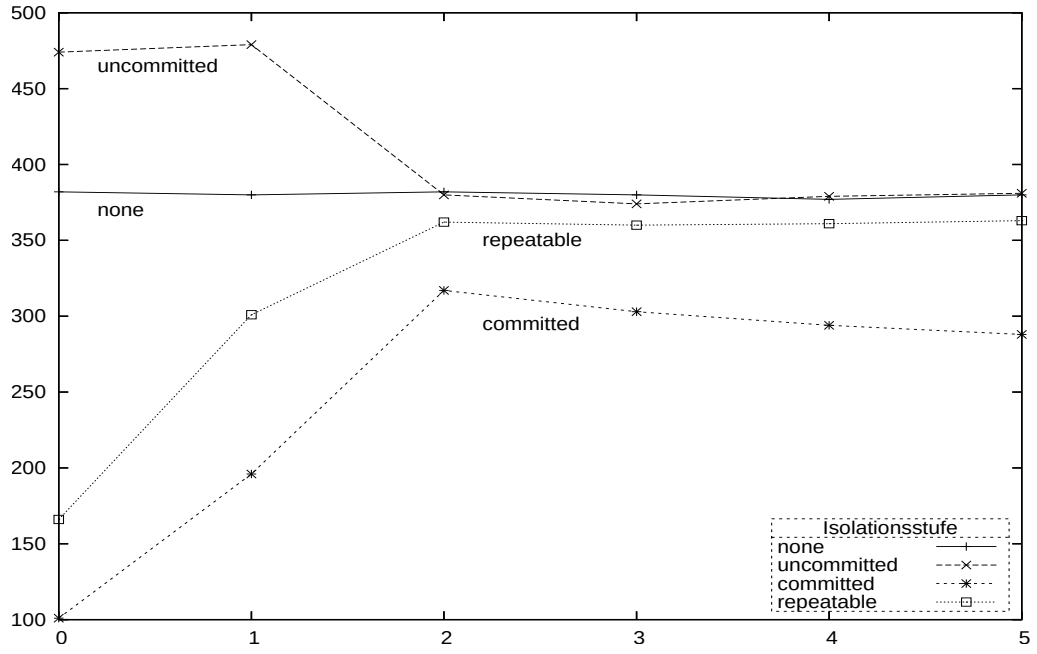


Abbildung 142: Vergleich der Isolationsstufen erfolgreicher Transaktionen

Der maximale Transaktionsdurchsatz bei der parallelen Ausführung aller Transaktionstypen in der Isolationsstufe *none* liegt bei etwa 380 Transaktionen. Diese Messwerte sind natürlich unabhängig von der Sperrtiefe, da weder Sperranforderungen für den Schreib- noch für den Lesezugriff stattfinden.

In der Isolationsstufe *uncommitted* werden nur Schreibsperren angefordert, die bis zum Ende der betroffenen Transaktion gehalten werden. Das hat zur Folge, dass für die Sperrtiefen 0 und 1 eine hohe Zahl von Blockierungen der Schreibtransaktionen stattfindet und fast ausschließlich Lesetransaktionen abgearbeitet werden, die keinerlei Sperranforderungen an den Sperrmanager stellen. Da das XML-Datenbanksystem für Leseoperationen einer Transaktion keine Log-Daten schreibt, werden diese schneller als schreibende Operationen verarbeitet. Somit liegt der Durchsatz der fast ausschließlichen Lesetransaktionen in der Isolationsstufe *uncommitted* für die Sperrtiefen 0 und 1 über denen der Isolationsstufe *none*, in der alle definierten Transaktionstypen gleichmäßig ausgeführt werden. Ab der Sperrtiefe 2 erreichen *uncommitted* und *none* etwa den gleichen Transaktionsdurchsatz, da für *uncommitted* nur relativ wenige Schreibsperren angefordert werden müssen und fast keine Blockierungen mehr auftreten.

Die Isolationsstufe *committed*, in der für jede lesende Transaktionsoperation eine Sperre angefordert wird, die sofort nach der Operation wieder freigegeben werden muss, zeigt im Vergleich zu den Konsistenzebenen in relationalen Datenbanksystemen [HR99] ein überraschendes Verhalten. In diesem Fall führt die Wahl einer geringeren Isolationsstufe (geringer als die Stufe *repeatable*) nicht zu einem höheren Transaktionsdurchsatz, da im baumorientierten Datenmodell des XML-Datenbanksystems kontinuierlich für jede Operation Sperren auf den Knoten

6 Der mit der Isolationsstufe *none* zwangsläufig entstehende inkonsistente Systemzustand dient nur der Messung des maximal möglichen Transaktionsdurchsatzes bzgl. der zu verarbeitenden Last.

mehrerer Ebenen des XML-Dokuments eingetragen und wieder entfernt werden müssen. Dieser Aufwand nimmt mit größerer Sperrtiefe zu. Für die Sperrtiefen 0 bis 2 ist zwar die Anzahl der anzufordernden Sperren deutlich geringer, allerdings muss nach wie vor jede einzelne Operation erneut um die Reservierung der für die Transaktion benötigten Ressourcen konkurrieren, sodass der Durchsatz in diesem Bereich zwischen etwa 100 und 300 Transaktionen liegt.

Der Transaktionsdurchsatz in der Isolationsstufe *repeatable* zeigt das erwartete Verhalten aus dem Transaktionsmix in Abschnitt 7.3.1. Hier werden Lese- und Schreibsperren bis zum Transaktionsende gehalten und eine Erhöhung der Sperrtiefe führt zu weniger Blockierungen und damit zu einem höheren Durchsatz.

Abbildung 143 vergleicht die Anzahl der abgebrochenen Transaktionen in Abhängigkeit der gewählten Isolationsstufe und Sperrtiefe. Interessant ist hierbei, dass nur in der Isolationsstufe *repeatable*, in der lange Lese- und Schreibsperren bis zum Transaktionsende gehalten werden, die Blockierungen in den Sperrtiefen 0 und 1 zu Deadlocks und einer hohen Rücksetzrate der Transaktionen führen. In den übrigen Isolationsstufen finden fast keine Transaktionsrücksetzungen statt (teilweise auch durch Deadlocks bei *Fix-/Unfix*-Operationen auf Datenseitenebene ausgelöst), womit die Messwerte der erfolgreichen Transaktionsabschlüsse in Abbildung 142 hauptsächlich durch Wartezyklen und den Aufwand der Sperrverwaltung beeinflusst werden.

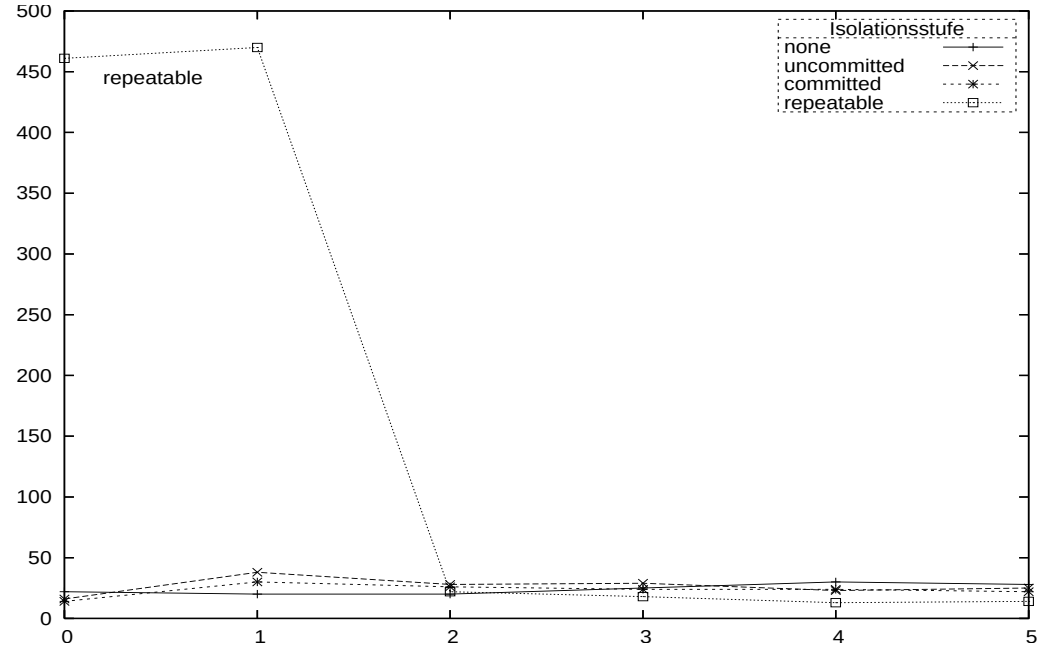


Abbildung 143: Vergleich der Isolationsstufen abgebrochener Transaktionen

Für eine letzte Messung werden „langlebige“ Transaktionen betrachtet. Dazu wartet jeder Transaktionstyp des Gesamtmix nach der Ausführung aller Operationen weitere zehn Sekunden, bevor mit einem *Commit* alle Ressourcen freigegeben werden. Mit dieser künstlich erzeugten Wartephase wird der Einfluss der Sperrverwaltung reduziert (typisch für längere Transaktionslaufzeiten bspw. aufgrund menschlicher Interaktion), und die Auswirkungen von Blockierungen treten deutlicher hervor. Das Ergebnis dieser Messung für die erfolgreich abgeschlossenen Transaktionen ist in Abbildung 144 dargestellt.

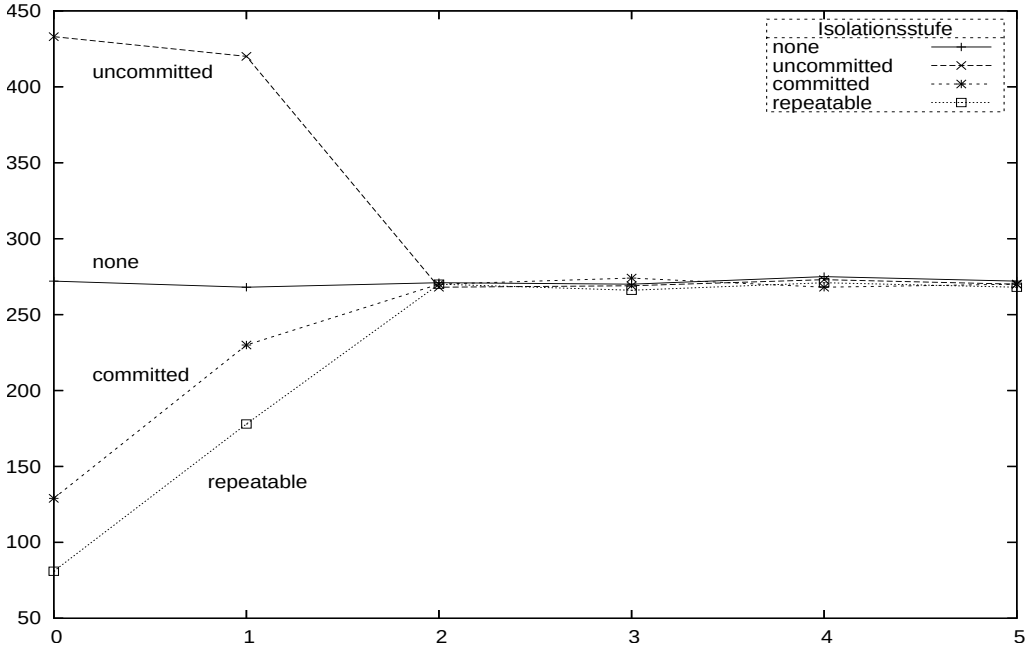


Abbildung 144: Transaktionsdurchsatz für „langlebige“ Transaktionen

Aufgrund der künstlich erzeugten Wartephase am Ende jeder Transaktion sinkt der Transaktionsdurchsatz in der Isolationsstufe *none* auf etwa 270 ab. Der Durchsatz für die Isolationsstufe *uncommitted* zeigt dasselbe Verhalten wie in der zuvor durchgeführten Messung. Schreibende Transaktionen blockieren sich bei den Sperrtiefen 0 und 1 und lesende Transaktionen, für die in dieser Isolationsstufe keine Sperranforderungen nötig sind, werden in hoher Anzahl verarbeitet. Ab einer Sperrtiefe von 2 treten fast keine Blockierungen mehr auf, sodass sich die Anzahl der erfolgreich beendeten Transaktionen für alle Isolationsstufen bei etwa 270 einpendelt. Für die Sperrtiefen 0 und 1 mit hoher Kollisionsrate gilt nun durch die Entkopplung des Aufwands für die Sperrverwaltung der Grundsatz, dass die Wahl einer niedrigeren Isolationsstufe einen höheren Transaktionsdurchsatz ermöglicht.

7.4 Zusammenfassung

Dieses Kapitel wertet die in dieser Arbeit vorgestellten Konzepte zur nativen XML-Datenverarbeitung mit detaillierten Leistungsmessungen aus und beurteilt die jeweiligen Ansätze.

Zunächst betrachtet Abschnitt 7.1 die Speicherungsstrukturen und untersucht den Speicherplatzbedarf für die in Datenseiten kodierten DeweyIDs und die damit verbundenen Vorteile einer Präfix-Komprimierung. Da die Speicherung von DeweyIDs in einzelnen Bytes erfolgt, wurden zusätzlich Kodierungen analysiert, die bei der Konstruktion der Bit-Folgen für die *Division*-Werte die Byte-Grenzen der Speicherungsstruktur berücksichtigen.

Abschnitt 7.2 gibt einen kurzen Überblick über bereits existierende XML-Benchmarks und erläutert das entwickelte TaMix-Framework zur automatisierten Leistungsmessung in einer verteilten Testumgebung. Abschnitt 7.3 wertet mit zahlreichen Messergebnissen die Leistungsfähigkeit der besprochenen Sperrprotokolle im XML-Datenbanksystem XTC aus. Dazu wurden mehrere Transaktionstypen definiert, die typische XML-Zugriffsmuster implementieren, und parallel auf einem Benchmark-Dokument zur Ausführung kamen. Abschließend wurde der Einfluss der gewählten Isolationsstufe auf den Transaktionsdurchsatz untersucht.

Die hier ermittelten Messwerte sollten jedoch nicht als generelle Kennzahlen interpretiert werden, da sie stark von den eingesetzten Hardware-Komponenten abhängen. Der Durchsatz eines Transaktionsmix, der aus vielen schreibenden Transaktionsoperationen zusammengesetzt ist, hängt in erster Linie von der Leistung der eingesetzten Sekundärspeichermedien ab, wogegen viele lesende Transaktionen von einem großen Datenbankpuffer profitieren. Die Kosten, die für die Sperrverwaltung entstehen, werden hauptsächlich von der Geschwindigkeit des Systemprozessors bestimmt. Die ermittelten Messkurven werden für verschiedene Systemkonfigurationen nicht ihre Charakteristik verlieren, allerdings wird das Verhältnis der Messwerte durch das Leistungspotential einzelner Systemkomponenten entscheidend beeinflusst.

KAPITEL 8 Zusammenfassung und Ausblick

Es ist sinnlos zu sagen: Wir tun unser Bestes.
Es muss dir gelingen, das zu tun, was erforderlich ist.
(Winston Churchill)

8.1 Zusammenfassung

Die transaktionsgeschützte Verarbeitung von XML-Dokumenten, auf die verschiedene Anwendungen parallel mit den typischen XML-Schnittstellen auf der Ebene einzelner Knoten sowohl lesend als auch schreibend zugreifen, erfordert die Implementierung neuer Konzepte beim Entwurf von Datenbanksystemen. Diese Arbeit beschreibt solche Konzepte und zeigt mit einer prototypischen Realisierung deren Umsetzbarkeit in einem konkreten System.

Nach der Einführung der XML-Grundlagen und der Diskussion der verwandten Arbeiten auf dem Gebiet der nativen XML-Datenverarbeitung führt das taDOM-Transaktionsmodell die konzeptionelle Infrastruktur für die Verarbeitung von XML-Dokumenten in einer Mehrbenutzerumgebung ein. Dafür wird zunächst das taDOM-Datenmodell spezifiziert, welches XML-Dokumente mit taDOM-Bäumen feingranular durch einzelne Knoten repräsentiert. Die taDOM-Bäume erweitern das Document Object Model um die beiden Knotentypen Attributwurzel und String zur Steigerung der Transaktionsparallelität. Auf dem taDOM-Datenmodell werden 19 Basisoperationen definiert, die die Traversierung und die inhaltliche sowie strukturelle Modifikation von taDOM-Bäumen unterstützen. Zur Identifikation der Knoten eines taDOM-Baums werden DeweyIDs eingesetzt, ein Nummerierungsschema, das auf der Dewey-Dezimalklassifikation basiert und beliebige Dokumentmodifikationen ohne Reorganisationen bereits vergebener IDs ermöglicht. Zusätzlich unterstützen DeweyIDs die Sperrverwaltung und Anfrageverarbeitung, da die Lage zweier XML-Knoten bzgl. aller XQuery-Pfadachsen aus ihren IDs ohne Zugriff auf das gespeicherte Dokument berechnet werden kann.

Mit der Transaktionsisolation durch die vier aufeinander aufbauenden hierarchischen Sperrprotokolle taDOM2, taDOM2+, taDOM3 und taDOM3+ wird die Beschreibung des taDOM-Transaktionsmodells abgeschlossen. Das Basisprotokoll taDOM2 erweitert die aus der Datenbankliteratur bekannten hierarchischen Sperrverfahren um die Fähigkeit, eine gesamte Ebene von Knoten zu sperren, ohne dass dieser Sperrmodus auf den darunter liegende Teilbaum Auswirkungen hat. Bei der erforderlichen Sperrkonversion zeigt das taDOM2-Protokoll jedoch den Nachteil, dass u. U. auf das Speichersystem zugegriffen werden muss, um alle Kinder eines gesperrten Knotens zu ermitteln. Dieses Problem wird durch weitere Sperrmodi im Protokoll taDOM2+ behoben. Analog zum Sperren einer Teilbaumebene muss es auch möglich sein, einen inneren Elementknoten in einem Dokument umzubenennen, ohne dass das darunter liegende Fragment dadurch für Änderungen blockiert wird. Für die Protokolle taDOM2 und taDOM2+ werden dafür virtuelle Namensknoten einführt, welche die gewünschte Funktionalität realisieren, allerdings die Verwaltung von zwei Sperren pro XML-Knoten erfordern. Eine Erweiterung der Sperrprotokolle um zusätzliche Sperrmodi zum Verfahren taDOM3 macht dieses Konzept überflüssig. Jedoch auch hier zeigt die Sperrkonversion die Erfordernis, teil-

weise auf das Speichersystem zugreifen zu müssen. Eine letzte Erweiterung auf insgesamt 20 Sperrmodi zum Protokoll taDOM3+ definiert einen Sperrmechanismus, der die Sperranforderungen und -konversionen für alle Basisoperationen des Transaktionsmodells ohne Zugriff auf das gespeichert XML-Dokument unterstützt.

Zum Beweis der Vollständigkeit und Korrektheit der taDOM-Sperrprotokolle wurden für einen beliebigen Kontextknoten eines taDOM-Baums dessen umgebende Knoten beschrieben und für diese sog. Kontextumgebung mit Use Cases das korrekte Ausführungsverhalten der Basisoperationen spezifiziert. Mit dem Prinzip des Model Checking wurde automatisiert in mehreren hunderttausend Einzelprüfungen die korrekte Transaktionsisolation in den entsprechenden Ausführungszuständen der Basisoperationen nachgewiesen.

Die taDOM-Protokolle garantieren bei reiner Navigation auf XML-Dokumenten die Konsistenzstufe *Repeatable Read*. Erfolgen jedoch Einsprünge in das Dokument auf Knoten, deren Adresse als Referenz in Indexstrukturen abgelegt ist, können bei der erneuten Auswertung der Indexstrukturen Phantome auftreten. Diese werden mit dem Konzept der wertbasierten Achsensperren, einer Erweiterung des Key-range Locking, verhindert, sodass damit für Transaktionen die Konsistenzstufe *Serializable* bei der Auswertung aller XQuery-Pfadachsen realisiert wird.

Der Entwurfsprozess für die taDOM-Protokolle und die Erweiterung mit wertbasierten Achsensperren orientieren sich somit an einer möglichst optimalen Unterstützung sowohl der taDOM-Basisoperationen als auch der XQuery-Pfadachsen, die u. U. über sekundäre Indexstrukturen ausgewertet werden. Eine Ebenensperre synchronisiert die explizite Ermittlung aller Kindknoten und die Auswertung der *Child*-Achse. Eine Teilbaumsperre dient zur Rekonstruktion vollständiger Fragmente und wird für die Auswertung der *Descendant*-, *Preceding*- und *Following*-Achse mit einer geringen Anzahl von Sperranforderungen eingesetzt. Die Möglichkeit, Sperren auf einzelnen Knoten anzufordern, ermöglicht das Auslesen und Modifizieren von Knoten ohne die darunter liegenden Teilbäume zu blockieren und unterstützt somit die knotenbasierte Navigation und Operationen auf der *Self*-Achse.

Zur Auswertung der Umsetzung aller vorgestellten Konzepte in einem realen System haben wir die prototypische Implementierung des XML Transaction Coordinator (XTC) entwickelt. In diesem System werden taDOM-Bäume zur persistenten Speicherung auf B*-Bäume abgebildet, die Präfix-komprimierte DeweyIDs zur Adressierung der Datensätze verwenden. Dazu müssen DeweyIDs in Byte-Folgen kodiert werden, wofür es einige, detailliert untersuchte Varianten gibt.

Abschließend wurden mit dem Prototyp zahlreiche Messungen in einer verteilten Testumgebung durchgeführt, um den Einfluss verschiedener Sperrprotokolle auf den Transaktionsdurchsatz zu ermitteln. Dabei zeigt sich, dass sich die klassischen hierarchischen Sperrprotokolle in einigen Situationen durchaus für die Transaktionsisolation in XML-Datenbanksystemen einsetzen lassen, in einem Transaktionsmix, der viele verschiedene XML-typische Zugriffsmuster enthält, die präsentierten taDOM-Protokolle jedoch einen wesentlich höheren Transaktionsdurchsatz erzielen.

8.2 Ausblick

Der XTC-Server ist ein Datenbanksystem zur zentralen Verwaltung von XML-Dokumenten, die aufgrund der vorgestellten Isolationsmechanismen parallel verarbeitet werden können. Für die DOM-Schnittstelle, die Methoden auf einzelnen XML-Knoten ausführt, bedeutet dies, dass entweder vollständige Fragmente gesperrt und zur Anwendung transportiert werden müssen oder jeder Methodenaufruf separat zum Datenbanksystem propagiert wird. Der Aufwand für die dafür erforderliche Kommunikation könnte reduziert werden, indem für oft durchzuführende Aufgaben *DOM-Module* erstellt werden, die im Datenbanksystem abgelegt und zu gegebener Zeit aufgerufen werden. Damit wird nur das relevante Ergebnis einer Operationsfolge zur Anwendung transportiert. Zudem könnte für die DOM-Module mit Hilfe von Dokumentenschemata bereits bei deren initialen Übertragung eine Optimierung erfolgen: Die Aufruffolge *getFirstChild()*, *getNextSibling()*, *getNextSibling()* kann bspw. durch *getLastChild()* ersetzt werden, wenn aus dem Schema hervorgeht, dass der adressierte Kontextknoten immer drei Kindknoten besitzt.

Langfristig wird jedoch die navigationsorientierte DOM-Schnittstelle vermutlich nur von Spezialanwendungen benutzt und ein Großteil der Anfragen mit deklarativen Sprachen formuliert werden.

Für die Verarbeitung solch deklarativer Anfragen (bspw. XQuery) eröffnet sich das gesamte Gebiet der Anfrageoptimierung. Hier spielen vor allem die Nutzung von Indexstrukturen, die Implementierung von Planoperatoren zur alternativen Auswertung spezieller Sprachkonstrukte, das Erstellen von Kostenmodellen und die Konstruktion und Bewertung von Zugriffsplänen eine wichtige Rolle. Diese Aspekte sind zur Zeit ebenfalls Thema der Untersuchungen im Rahmen des XTC-Projekts [MH05].

Atomarität und Dauerhaftigkeit werden für Transaktionen durch die Abbildung der Speicherungsstrukturen auf Datenseiten und deren Verwaltung in Containerdateien mit den entsprechenden Transaktionsprotokollen gewährleistet. Die Isolation ist durch die taDOM-Sperrprotokolle und den *Fix-unfix*-Mechanismus für Datenseiten im Datenbankpuffer sichergestellt. Ein weiterer großer Themenkomplex, den es für die Garantie aller ACID-Eigenschaften der Transaktionsverarbeitung zu untersuchen gilt, ist die Konsistenzsicherung von XML-Dokumenten bzgl. einer Schemabeschreibung. Solch ein Schema kann zudem für die Anfrageauswertung herangezogen werden, da es Informationen über die Dokumentstruktur enthält. Es muss jedoch nach wie vor möglich sein, Dokumente auch ohne ein spezifiziertes Schema speichern zu können. Das bedeutet, dass Dokumente mit einer Schemabeschreibung bei der Speicherung gegen diese Beschreibung validiert werden müssen und für Dokumente ohne Schemabeschreibung eine solche aus dem Dokument selbst erzeugt werden muss. Für die Dokumentmodifikation während des Systembetriebs ergeben sich im Besonderen zwei laufzeitkritische Anforderungen: Wurde bei der Speicherung für ein Dokument kein Schema angegeben, so kann eine Änderung des Dokumentinhalts zu einer Evolution des generierten Schemas führen; wurde ein Schema spezifiziert, so müssen bei einer Änderung des Dokuments möglichst schnell die betroffenen Teile des Schemas lokalisiert werden, die zur Validierung der modifizierten Stellen benötigt werden. Da für die Schemabeschreibung eines XML-Dokuments der gesamte XML-Schema-Standard berücksichtigt werden sollte, bietet auch dieses Thema großes Potenzial für weitere Arbeiten.

ANHANG A Das Beispieldokument

Anhang A zeigt das Beispieldokument mit Schemabeschreibungen in der Document Type Definition und XML Schema. Da die beiden Schemabeschreibungen verschiedene Dokumentinstanzen zur Realisierung der referenziellen Integrität erzwingen, wird das Beispieldokument zunächst für die DTD und danach für XML Schema angegeben.

A.1 Umsetzung mit der Document Type Definition

Das XML-Dokument

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Bank SYSTEM "bank.dtd">

<Bank>

    <Kunden>
        <Kunde id="kd1606">
            <Name>
                <Vorname>Michael</Vorname>
                <Vorname>Peter</Vorname>
                <Nachname>Haustein</Nachname>
            </Name>
            <Adresse>
                <Straße>Gottlieb-Daimler-Straße</Straße>
                <Hausnummer/>
                <PLZ>D-67663</PLZ>
                <Ort>Kaiserslautern</Ort>
            </Adresse>
        </Kunde>
        <Kunde id="kd1407">
            <Name>
                <Vorname>Sandra</Vorname>
                <Nachname>Haustein</Nachname>
            </Name>
            <Adresse>
                <Straße>...</Straße>
                <Hausnummer/>
                <PLZ>...</PLZ>
                <Ort>...</Ort>
            </Adresse>
        </Kunde>
    </Kunden>

    <Konten>
        <Konto id="kto4711" Besitzer="kd1606 kd1407">
            <Dispo>4500.00</Dispo>
        </Konto>
    </Konten>

</Bank>
```

Die Schemabeschreibung

```
<!ELEMENT Bank (Kunden,Konten)>

<!ELEMENT Kunden (Kunde*)>
<!ELEMENT Kunde (Name,Adresse)>
<!ATTLIST Kunde id ID #REQUIRED>
<!ELEMENT Name (Vorname+,Nachname)>
<!ELEMENT Vorname (#PCDATA)>
<!ELEMENT Nachname (#PCDATA)>
<!ELEMENT Adresse (Straße,Hausnummer?,PLZ,Ort)>
<!ELEMENT Straße (#PCDATA)>
<!ELEMENT Hausnummer (#PCDATA)>
<!ELEMENT PLZ (#PCDATA)>
<!ELEMENT Ort (#PCDATA)>

<!ELEMENT Konten (Konto*)>
<!ELEMENT Konto (Dispo)>
<!ELEMENT Dispo (#PCDATA)>
<!ATTLIST Konto id      ID      #REQUIRED
           Besitzer    IDREFS  #REQUIRED>
```

A.2 Umsetzung mit XML Schema

Das XML-Dokument

```
<?xml version="1.0" encoding="UTF-8"?>
<Bank   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="bank.xsd">

    <Kunden>
        <Kunde id="kd1606">
            <Name>
                <Vorname>Michael</Vorname>
                <Vorname>Peter</Vorname>
                <Nachname>Haustein</Nachname>
            </Name>
            <Adresse>
                <Straße>Gottlieb-Daimler-Straße</Straße>
                <Hausnummer/>
                <PLZ>D-67663</PLZ>
                <Ort>Kaiserslautern</Ort>
            </Adresse>
        </Kunde>
        <Kunde id="kd1407">
            <Name>
                <Vorname>Sandra</Vorname>
                <Nachname>Haustein</Nachname>
            </Name>
            <Adresse>
                <Straße>...</Straße>
                <Hausnummer>...</Hausnummer>
                <PLZ>D-67663</PLZ>
                <Ort>...</Ort>
            </Adresse>
        </Kunde>
    </Kunden>
    <Konten>
        <Konto id="kto4711">
            <Besitzer>kd1606</Besitzer>
            <Besitzer>kd1407</Besitzer>
            <Dispo>4500.00</Dispo>
        </Konto>
    </Konten>

</Bank>
```

Die Schemabeschreibung

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="Bank">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element maxOccurs="1" minOccurs="1" name="Kunden">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="Kunde"
                                    maxOccurs="unbounded" minOccurs="0">
                                <xsd:complexType>
                                    <xsd:sequence>
                                        <xsd:element name="Name">
                                            <xsd:complexType>
                                                <xsd:sequence>
                                                    <xsd:element
                                                        maxOccurs="3"
                                                        minOccurs="1"
                                                        name="Vorname"
                                                        type="xsd:string"/>
                                                    <xsd:element
                                                        name="Nachname"
                                                        type="xsd:string"/>
                                                </xsd:sequence>
                                            </xsd:complexType>
                                        </xsd:element>
                                        <xsd:element name="Adresse">
                                            <xsd:complexType>
                                                <xsd:sequence>
                                                    <xsd:element
                                                        name="Straße"
                                                        type="xsd:string"/>
                                                    <xsd:element
                                                        name="Hausnummer"
                                                        type="xsd:string"/>
                                                    <xsd:element name="PLZ">
                                                        <xsd:simpleType>
                                                            <xsd:restriction
                                                            base="xsd:string">
                                                                <xsd:pattern
                                        value="D-[0-9][0-9][0-9][0-9][0-9]"/>
                                                            </xsd:restriction>
                                                        </xsd:simpleType>
                                                    </xsd:element>
                                                    <xsd:element
                                                        name="Ort"
                                                        type="xsd:string"/>
                                                </xsd:sequence>
                                            </xsd:complexType>
                                        </xsd:element>
                                    </xsd:sequence>
                                    <xsd:attribute name="id" type="xsd:string"/>
                                </xsd:complexType>
                            </xsd:element>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
```

```
                <xsd:element name="Konten" minOccurs="0" maxOccurs="1">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="Konto" maxOccurs="unbounded">
                                <xsd:complexType>
                                    <xsd:sequence>
                                        <xsd:element
                                            name="Besitzer"
                                            type="xsd:string"
                                            minOccurs="1"
                                            maxOccurs="5"/>
                                        <xsd:element name="Dispo">
                                            <xsd:simpleType>
                                                <xsd:restriction
                                                    base="xsd:decimal">
                                                    <xsd:fractionDigits
                                                        value="2"/>
                                                    <xsd:minInclusive
                                                        value="0.00"/>
                                                    <xsd:maxInclusive
                                                        value="10000.00"/>
                                                </xsd:restriction>
                                            </xsd:simpleType>
                                        </xsd:element>
                                    </xsd:sequence>
                                    <xsd:attribute name="id" type="xsd:string"/>
                                </xsd:complexType>
                            </xsd:element>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>

        <xsd:key name="Kundennummer">
            <xsd:selector xpath="Kunden/Kunde"/>
            <xsd:field xpath="@id"/>
        </xsd:key>
        <xsd:key name="Kontonummer">
            <xsd:selector xpath="Konten/Konto"/>
            <xsd:field xpath="@id"/>
        </xsd:key>
        <xsd:keyref name="Kontobesitzer" refer="Kundennummer">
            <xsd:selector xpath="Konten/Konto/Besitzer"/>
            <xsd:field xpath="."/>
        </xsd:keyref>
    </xsd:element>
</xsd:schema>
```

ANHANG B Use Cases für die taDOM-Operationen

B.1 Basisoperationen

- readCO — Liest den Wert des Kontextknotens.
- writeCO — Schreibt einen neuen Wert für den Kontextknoten.
- useCO-PSE — Navigiert entlang der *prevSibling*-Kante des Kontextknotens.
- useCO-NSE — Navigiert entlang der *nextSibling*-Kante des Kontextknotens.
- useCO-FCE — Navigiert entlang der *firstChild*-Kante des Kontextknotens.
- useCO-LCE — Navigiert entlang der *lastChild*-Kante des Kontextknotens.
- modifyCO-PSE — Modifiziert das Verweisziel der *prevSibling*-Kante des Kontextknotens.
- modifyCO-NSE — Modifiziert das Verweisziel der *nextSibling*-Kante des Kontextknotens.
- modifyCO-FCE — Modifiziert das Verweisziel der *firstChild*-Kante des Kontextknotens.
- modifyCO-LCE — Modifiziert das Verweisziel der *lastChild*-Kante des Kontextknotens.
- readCS — Liest den Wert des angehefteten String-Knotens des Kontextknotens.
- writeCS — Schreibt einen neuen Wert für den String-Knoten des Kontextknotens.
- readPA — Liest den Elternknoten des Kontextknotens.
- usePA-FCE — Navigiert entlang der *firstChild*-Kante des Elternknotens.
- usePA-LCE — Navigiert entlang der *lastChild*-Kante des Elternknotens.
- modifyPA-FCE — Modifiziert das Verweisziel der *firstChild*-Kante des Elternknotens.
- modifyPA-LCE — Modifiziert das Verweisziel der *lastChild*-Kante des Elternknotens.
- readPS — Liest den Wert des vorherigen Geschwisterknotens.
- usePS-NSE — Navigiert entlang der *nextSibling*-Kante des vorherigen Geschwisterknotens.
- modifyPS-NSE — Modifiziert das Verweisziel der nextSibling-Kante des vorherigen Geschwisterknotens.

- readNS — Liest den Wert des nächsten Geschwisterknotens.
- useNS-PSE — Navigiert entlang der *prevSibling*-Kante des nächsten Geschwisterknotens.
- modifyNS-PSE — Modifiziert das Verweisziel der *prevSibling*-Kante des nächsten Geschwisterknotens.
- readFC — Liest den Wert des ersten Kindknotens.
- writeFC — Schreibt einen neuen Wert für den ersten Kindknoten.
- useFC-PSE — Navigiert entlang der *prevSibling*-Kante des ersten Kindknotens.
- useFC-NSE — Navigiert entlang der *nextSibling*-Kante des ersten Kindknotens.
- useFC-FCE — Navigiert entlang der *firstChild*-Kante des ersten Kindknotens.
- useFC-LCE — Navigiert entlang der *lastChild*-Kante des ersten Kindknotens.
- modifyFC-PSE — Modifiziert das Verweisziel der *prevSibling*-Kante des ersten Kindknotens.
- readCH — Liest den Wert eines beliebigen Kindknotens.
- writeCH — Schreibt einen neuen Wert für einen beliebigen Kindknoten.
- useCH-PSE — Navigiert entlang der *prevSibling*-Kante eines beliebigen Kindknotens.
- useCH-NSE — Navigiert entlang der *nextSibling*-Kante eines beliebigen Kindknotens.
- useCH-FCE — Navigiert entlang der *firstChild*-Kante eines beliebigen Kindknotens.
- useCH-LCE — Navigiert entlang der *lastChild*-Kante eines beliebigen Kindknotens.
- readLC — Liest den Wert des letzten Kindknotens.
- writeLC — Schreibt einen neuen Wert für den letzten Kindknoten.
- useLC-PSE — Navigiert entlang der *prevSibling*-Kante des letzten Kindknotens.
- useLC-NSE — Navigiert entlang der *nextSibling*-Kante des letzten Kindknotens.
- useLC-FCE — Navigiert entlang der *firstChild*-Kante des letzten Kindknotens.
- useLC-LCE — Navigiert entlang der *lastChild*-Kante des letzten Kindknotens.
- modifyLC-NSE — Modifiziert das Verweisziel der *nextSibling*-Kante des letzten Kindknotens.

- readDC — Liest den Wert eines beliebigen Nachfahren.
- writeDC — Schreibt einen neuen Wert für einen beliebigen Nachfahren.
- useDC-PSE — Navigiert entlang der *prevSibling*-Kante eines beliebigen Nachfahren.
- useDC-NSE — Navigiert entlang der *nextSibling*-Kante eines beliebigen Nachfahren.
- useDC-FCE — Navigiert entlang der *firstChild*-Kante eines beliebigen Nachfahren.
- useDC-LCE — Navigiert entlang der *lastChild*-Kante eines beliebigen Nachfahren.
- readCA — Liest das Kontextattribut.
- writeCA — Schreibt einen neuen Namen für das Kontextattribut.
- readAS — Liest den Wert des Kontextattributs im angehefteten String-Knoten.
- writeAS — Schreibt einen neuen Wert für das Kontextattribut in den angehefteten String-Knoten.
- readAX — Liest ein beliebiges Attribut.
- writeAX — Schreibt einen neuen Namen für ein beliebiges Attribut.
- readXS — Liest den Wert eines beliebigen Attributs aus dem angehefteten String-Knoten.
- writeXS — Schreibt einen neuen Wert für ein beliebiges Attribut in den angehefteten String-Knoten.

B.2 Use Cases

Use Case 0: getNode(DeweyID) liefert Knoten

Beschreibung

Die Operation liefert den durch die DeweyID adressierten Kontextknoten.

Basisoperationen

- readCO

Read-Set

- CO

Write-Set

Ausführung auf CO

Szenario 0-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-

Szenario 0-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-

Szenario 0-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-

Szenario 0-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 1: getParentNode(contextNode) liefert Knoten

Beschreibung

Die Operation liefert den Elternknoten des Kontextknotens.

Basisoperationen

- readCO
- readPA

Read-Set

- CO
- PA

Write-Set

Ausführung auf CO

Szenario 1-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-
PA	NR	-	-	-	-
VN	NR	-	-	-	-

Szenario 1-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-
PA	NR	-	-	-	-
VN	NR	-	-	-	-

Szenario 1-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
PA	NR	-	-	-	-

Szenario 1-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
PA	NR	-	-	-	-

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 2: getPrevSibling(contextNode) liefert Knoten

Beschreibung

Die Operation liefert den vorherigen Geschwisterknoten des Kontextknotens. Der Geschwisterknoten ist vorhanden.

Szenario 2-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	ER	-	-	-
CN	NR	-	-	-	-
PS	NR	-	ER	-	-
PN	NR	-	-	-	-

Szenario 2-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	ER	-	-	-
CN	NR	-	-	-	-
PS	NR	-	ER	-	-
PN	NR	-	-	-	-

Szenario 2-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	ER	-	-	-
PS	NR	-	ER	-	-

Szenario 2-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	ER	-	-	-
PS	NR	-	ER	-	-

Basisoperationen

- readCO
- useCO-PSE
- readPS
- usePS-NSE

Ausführung auf CO

Read-Set

- CO
- CO-PSE
- PS
- PS-NSE

Write-Set

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 3: getPrevSibling(contextNode) liefert *null*

Beschreibung

Die Operation liefert *null*, da der Kontextknoten keinen vorherigen Geschwisterknoten besitzt. Damit wird bekannt, dass der Kontextknoten das erste Kind seines Elternknotens ist.

Szenario 3-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	ER	-	-	-
CN	NR	-	-	-	-
PA	-	-	-	ER	-

Szenario 3-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	ER	-	-	-
CN	NR	-	-	-	-
PA	-	-	-	ER	-

Szenario 3-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	ER	-	-	-
PA	-	-	-	ER	-

Szenario 3-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	ER	-	-	-
PA	-	-	-	ER	-

Basisoperationen

- readCO
- useCO-PSE
- usePA-FCE

Ausführung auf CO

Read-Set

- CO
- CO-PSE
- PA-FCE

Write-Set

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 4: getNextSibling(contextNode) liefert Knoten

Beschreibung

Die Operation liefert den nächsten Geschwisterknoten des Kontextknotens. Der Geschwisterknoten ist vorhanden.

Szenario 4-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	ER	-	-
CN	NR	-	-	-	-
NS	NR	ER	-	-	-
NN	NR	-	-	-	-

Szenario 4-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	ER	-	-
CN	NR	-	-	-	-
NS	NR	ER	-	-	-
NN	NR	-	-	-	-

Szenario 4-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	ER	-	-
NS	NR	ER	-	-	-

Szenario 4-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	ER	-	-
NS	NR	ER	-	-	-

Basisoperationen

- readCO
- useCO-NSE
- readNS
- useNS-PSE

Ausführung auf CO

Read-Set

- CO
- CO-NSE
- NS
- NS-PSE

Write-Set

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 5: getNextSibling(contextNode) liefert *null*

Beschreibung

Die Operation liefert *null*, da der Kontextknoten keinen nächsten Geschwisterknoten besitzt. Damit wird bekannt, dass der Kontextknoten das letzte Kind seines Elternknotens ist.

Szenario 5-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	ER	-	-
CN	NR	-	-	-	-
PA	-	-	-	-	ER

Szenario 5-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	ER	-	-
CN	NR	-	-	-	-
PA	-	-	-	-	ER

Szenario 5-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	ER	-	-
PA	-	-	-	-	ER

Szenario 5-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	ER	-	-
PA	-	-	-	-	ER

Basisoperationen

- readCO
- useCO-NSE
- usePA-LCE

Read-Set

- CO
- CO-NSE
- PA-LCE

Write-Set

Ausführung auf CO

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, FC, CH, LC, DC, CA, AX

Use Case 6: getFirstChild(contextElementNode) liefert Knoten

Beschreibung

Die Operation liefert das erste Kind des Kontextknotens. Das erste Kind ist vorhanden.

Szenario 6-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	ER	-
CN	NR	-	-	-	-
FC	NR	ER	-	-	-
FN	NR	-	-	-	-

Szenario 6-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	ER	-
CN	NR	-	-	-	-
FC	NR	ER	-	-	-
FN	NR	-	-	-	-

Szenario 6-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	ER	-
FC	NR	ER	-	-	-

Szenario 6-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	ER	-
FC	NR	ER	-	-	-

Basisoperationen

- readCO
- useCO-FCE
- readFC
- useFC-PSE

Read-Set

- CO
- CO-FCE
- FC
- FC-PSE

Write-Set

Ausführung auf CO

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 7: getFirstChild(contextElementNode) liefert *null*

Beschreibung

Die Operation liefert null, weil der Kontextknoten kein erstes Kind besitzt. Damit wird bekannt, dass der Kontextknoten überhaupt keine Kinder besitzt.

Szenario 7-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	ER	ER
CN	NR	-	-	-	-

Szenario 7-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	ER	ER
CN	NR	-	-	-	-

Szenario 7-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	ER	ER

Szenario 7-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	ER	ER

Basisoperationen

- readCO
- useCO-FCE
- useCO-LCE

Read-Set

- CO
- CO-FCE
- CO-LCE

Write-Set

Ausführung auf CO

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, CA, AX

Use Case 8: getLastChild(contextElementNode) liefert Knoten

Beschreibung

Die Operation liefert das letzte Kind des Kontextknotens.
Das letzte Kind ist vorhanden.

Szenario 8-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	ER
CN	NR	-	-	-	-
LC	NR	-	ER	-	-
LN	NR	-	-	-	-

Szenario 8-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	ER
CN	NR	-	-	-	-
LC	NR	-	ER	-	-
LN	NR	-	-	-	-

Szenario 8-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	ER
LC	NR	-	ER	-	-

Szenario 8-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	ER
LC	NR	-	ER	-	-

Basisoperationen

- readCO
- useCO-LCE
- readLC
- useLC-NSE

Ausführung auf CO

Read-Set

- CO
- CO-LCE
- LC
- LC-NSE

Write-Set

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 9: getLastChild(contextElementNode) liefert *null*

Beschreibung

Die Operation liefert null, weil der Kontextknoten kein letzes Kind besitzt. Damit wird bekannt, dass der Kontextknoten überhaupt keine Kinder besitzt.

Szenario 9-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	ER	ER
CN	NR	-	-	-	-

Szenario 9-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	ER	ER
CN	NR	-	-	-	-

Szenario 9-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	ER	ER

Szenario 9-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	ER	ER

Basisoperationen

- readCO
- useCO-LCE
- useCO-FCE

Ausführung auf CO

Read-Set

- CO
- CO-LCE
- CO-FCE

Write-Set

Operation ausführbar auf den Knoten

PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, CA, AX

Use Case 10: getChildNodes(contextElementNode) liefert Knotenliste

Beschreibung

Die Operation liefert eine Liste aller Kinder des Kontextknotens. Besitzt der Kontextknoten keine Kinder, so wird eine leere Liste zurückgeliefert.

Szenario 10-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	LR	-	-	-	-
CN	NR	-	-	-	-

Szenario 10-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	LR	-	-	-	-
CN	NR	-	-	-	-

Szenario 10-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	LR	-	-	-	-

Szenario 10-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	LR	-	-	-	-

Basisoperationen

- readCO
- useCO-FCE
- useCO-LCE
- readFC
- useFC-PSE
- useFC-NSE
- readCH
- useCH-PSE
- useCH-NSE
- readLC
- useLC-PSE
- useLC-NSE

Read-Set

- CO
- CO-FCE
- CO-LCE
- FC
- FC-PSE
- FC-NSE
- CH
- CH-PSE
- CH-NSE
- LC
- LC-PSE
- LC-NSE

Write-Set

Ausführung auf CO

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 11: getFragmentNodes(contextElementNode) liefert Knotenliste

Beschreibung

Die Operation liefert eine Liste mit dem Kontextknoten und aller seiner Nachfahren. Besitzt der Kontextknoten keine Nachfahren, so wird eine Liste zurückgeliefert, die nur den Kontextknoten enthält.

Szenario 11-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SR	-	-	-	-
CN	NR	-	-	-	-

Szenario 11-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SR	-	-	-	-
CN	NR	-	-	-	-

Szenario 11-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SR	-	-	-	-

Szenario 11-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SR	-	-	-	-

Basisoperationen

- readCO
- readCS
- useCO-FCE
- useCO-LCE
- readFC
- useFC-PSE
- useFC-NSE
- useFC-FCE
- useFC-LCE
- readCH
- useCH-PSE
- useCH-NSE
- useCH-FCE
- useCH-LCE
- readLC
- useLC-PSE
- useLC-NSE
- useLC-FCE
- useLC-LCE
- readDC
- useDC-PSE
- useDC-NSE
- useDC-FCE
- useDC-LCE
- readCA
- readAS
- readAX
- readXS

Read-Set

- CO
- CS
- CO-FCE
- CO-LCE
- FC
- FC-PSE
- FC-NSE
- FC-FCE
- FC-LCE
- CH
- CH-PSE
- CH-NSE
- CH-FCE
- CH-LCE
- LC
- LC-PSE
- LC-NSE
- LC-FCE
- LC-LCE
- DC
- DC-PSE
- DC-NSE
- DC-FCE
- DC-LCE
- CA
- AS
- AX
- XS

Write-Set

Ausführung auf CO

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 12: getValue(contextElementNode) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und liefert den Kontextknoten selbst als Wert des Elements zurück.

Szenario 12-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-

Szenario 12-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-

Szenario 12-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-

Szenario 12-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-

Basisoperationen

- readCO

Ausführung auf CO

Read-Set

- CO

Write-Set

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 13: getValue(contextTextNode) liefert Knoten

Beschreibung

Die Operation wird auf einem Textknoten aufgerufen und liefert den angehefteten String-Knoten als Wert des Knotens zurück.

Szenario 13-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CS	NR	-	-	-	-

Szenario 13-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CS	NR	-	-	-	-

Szenario 13-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CS	NR	-	-	-	-

Szenario 13-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CS	NR	-	-	-	-

Basisoperationen

- readCO
- readCS

Ausführung auf CO

Read-Set

- CO
- CS

Write-Set

Operation ausführbar auf den Knoten

PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 14: getValue(contextAttributeNode) liefert Knoten

Beschreibung

Die Operation wird auf einem Attributknoten aufgerufen und liefert den angehefteten String-Knoten als Wert des Knotens zurück.

Basisoperationen

- readCA
- readAS

Read-Set

- CA
- AS

Write-Set

Ausführung auf CO

Szenario 14-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CA	NR	-	-	-	-
AN	NR	-	-	-	-
AS	NR	-	-	-	-

Szenario 14-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CA	NR	-	-	-	-
AN	NR	-	-	-	-
AS	NR	-	-	-	-

Szenario 14-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CA	NR	-	-	-	-
AS	NR	-	-	-	-

Szenario 14-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CA	NR	-	-	-	-
AS	NR	-	-	-	-

Operation ausführbar auf den Knoten

CA, AX

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 15: getAttribute(contextElementNode,attributeName) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und liefert den Knoten des Attributs mit dem angegebenen Namen.

Basisoperationen

- readCO
- readCA

Read-Set

- CO
- CA

Write-Set

Ausführung auf CO

Szenario 15-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-
CA	NR	-	-	-	-
AN	NR	-	-	-	-

Szenario 15-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-
CA	NR	-	-	-	-
AN	NR	-	-	-	-

Szenario 15-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CA	NR	-	-	-	-

Szenario 15-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CA	NR	-	-	-	-

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 16: getAttributes(contextElementNode) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und liefert eine Liste aller Attribute des Elements. Besitzt das Element keine Attribute, so wird eine leere Liste zurückgeliefert.

Basisoperationen

- readCO
- readCA
- readAX

Read-Set

- CO
- CA
- AX

Write-Set

Ausführung auf CO

Szenario 16-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-
AR	LR	-	-	-	-

Szenario 16-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-
AR	LR	-	-	-	-

Szenario 16-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
AR	LR	-	-	-	-

Szenario 16-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
AR	LR	-	-	-	-

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 17: setValue(contextElementNode,value) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und setzt für das Element einen neuen Wert. Der modifizierte Elementknoten wird zurückgeliefert. Für taDOM2(+) muss auf dem Elternknoten zusätzlich eine CX-Sperre angefordert werden, um *getChildNodes()* zu verhindern.

Szenario 17-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	SX	-	-	-	-
PA	CX	-	-	-	-

Szenario 17-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	SX	-	-	-	-
PA	CX	-	-	-	-

Szenario 17-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NX	-	-	-	-

Szenario 17-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NX	-	-	-	-

Basisoperationen

- writeCO

Ausführung auf CO

Read-Set

Write-Set

- CO

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 18: setValue(contextTextNode,value) liefert Knoten

Beschreibung

Die Operation wird auf einem Textknoten aufgerufen und setzt für den Text einen neuen Wert. Der modifizierte String-Knoten wird zurückgeliefert.

Szenario 18-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CS	SX	-	-	-	-

Szenario 18-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CS	SX	-	-	-	-

Szenario 18-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CS	NX	-	-	-	-

Szenario 18-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CS	NX	-	-	-	-

Basisoperationen

- readCO
- writeCS

Ausführung auf CO

Read-Set

- CO

Write-Set

- CS

Operation ausführbar auf den Knoten

PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 19: setValue(contextAttributeNode,value) liefert Knoten

Beschreibung

Die Operation wird auf einem Attributknoten aufgerufen und setzt für das Attribut einen neuen Wert. Der modifizierte String-Knoten wird zurückgeliefert.

Szenario 19-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CA	NR	-	-	-	-
AN	NR	-	-	-	-
AS	SX	-	-	-	-

Szenario 19-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CA	NR	-	-	-	-
AN	NR	-	-	-	-
AS	SX	-	-	-	-

Szenario 19-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CA	NR	-	-	-	-
AS	NX	-	-	-	-

Szenario 19-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CA	NR	-	-	-	-
AS	NX	-	-	-	-

Basisoperationen

- readCA
- writeAS

Ausführung auf CO

Read-Set

- CA

Write-Set

- AS

Operation ausführbar auf den Knoten

CA, AX

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 20: setAttribute(contextElementNode,attributeName,attributeValue) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und setzt für das Attribut mit dem angegebenen Namen einen neuen Wert. Das Attribut existiert bereits und der modifizierte String-Knoten des Attributs wird zurückgeliefert.

Szenario 20-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-
CA	NR	-	-	-	-
AN	NR	-	-	-	-
AS	SX	-	-	-	-

Szenario 20-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-
CA	NR	-	-	-	-
AN	NR	-	-	-	-
AS	SX	-	-	-	-

Szenario 20-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CA	NR	-	-	-	-
AS	NX	-	-	-	-

Szenario 20-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CA	NR	-	-	-	-
AS	NX	-	-	-	-

Basisoperationen

- readCO
- readCA
- writeAS

Ausführung auf CO

Read-Set

- CO
- CA

Write-Set

- AS

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 21: setAttribute(contextElementNode,attributeName,attributeValue) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und setzt für das Attribut mit dem angegebenen Namen einen neuen Wert. Das Attribut existiert noch nicht und muss angelegt werden. Der modifizierte String-Knoten des Attributs wird zurückgeliefert.

Szenario 21-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-
CA	SX	-	-	-	-

Szenario 21-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-
CA	SX	-	-	-	-

Szenario 21-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CA	SX	-	-	-	-

Szenario 21-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CA	SX	-	-	-	-

Basisoperationen

- readCO
- writeCA
- writeAS

Ausführung auf CO

Read-Set

- CO

Write-Set

- CA
- AS

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 22: renameAttribute(contextElementNode,oldAttributeName,newAttributeName) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und benennt das Attribut mit dem angegebenen Namen in den angegebenen neuen Namen um. Der umbenannte Attributknoten wird zurückgeliefert. In taDOM2(+) muss zusätzlich auf der Attributwurzel eine CX-Sperre angefordert werden, um die *getAttributes()*-Operation zu blockieren.

Szenario 22-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-
CA	NR	-	-	-	-
AN	SX	-	-	-	-
AR	CX	-	-	-	-

Szenario 22-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CN	NR	-	-	-	-
CA	NR	-	-	-	-
AN	SX	-	-	-	-
AR	CX	-	-	-	-

Szenario 22-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CA	NX	-	-	-	-

Szenario 22-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	-	-	-
CA	NX	-	-	-	-

Basisoperationen

- readCO
- writeCA

Ausführung auf CO

Read-Set

- CO

Write-Set

- CA

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 23: appendChild(contextElementNode,childType,childValue) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und fügt einen neuen Knoten als nächsten Geschwisterknoten hinter das bereits vorhandene letzte Kind des Kontextknotens ein. Der neu eingefügte Knoten wird zurückgeliefert.

Szenario 23-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	CX	-	-	-	EX
CN	NR	-	-	-	-
LC	-	-	EX	-	-

Szenario 23-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	CX	-	-	-	EX
CN	NR	-	-	-	-
LC	-	-	EX	-	-

Szenario 23-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NRCX	-	-	-	EX
LC	-	-	EX	-	-

Szenario 23-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NRCX	-	-	-	EX
LC	-	-	EX	-	-

Basisoperationen

- readCO
- modifyCO-LCE
- modifyLC-NSE

Ausführung auf CO

Read-Set

- CO

Write-Set

- CO-LCE
- LC-NSE

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 24: appendChild(contextElementNode,childType,childValue) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und fügt einen neuen Knoten als letztes Kind ein. Der Kontextknoten besitzt zu diesem Zeitpunkt noch keine Kindknoten. Der neu eingefügte Knoten wird zurückgeliefert.

Szenario 24-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	CX	-	-	EX	EX
CN	NR	-	-	-	-

Szenario 24-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	CX	-	-	EX	EX
CN	NR	-	-	-	-

Szenario 24-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NRCX	-	-	EX	EX

Szenario 24-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NRCX	-	-	EX	EX

Basisoperationen

- readCO
- modifyCO-FCE
- modifyCO-LCE

Ausführung auf CO

Read-Set

- CO

Write-Set

- CO-FCE
- CO-LCE

Operation ausführbar auf den Knoten

PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, CA, AX

Use Case 25: prependChild(contextElementNode,childType,childValue) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und fügt einen neuen Knoten als vorherigen Geschwisterknoten vor das bereits vorhandene erste Kind des Kontextknotens ein. Der neu eingefügte Knoten wird zurückgeliefert.

Szenario 25-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	CX	-	-	EX	-
CN	NR	-	-	-	-
FC	-	EX	-	-	-

Szenario 25-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	CX	-	-	EX	-
CN	NR	-	-	-	-
FC	-	EX	-	-	-

Szenario 25-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NRCX	-	-	EX	-
FC	-	EX	-	-	-

Szenario 25-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NRCX	-	-	EX	-
FC	-	EX	-	-	-

Basisoperationen

- readCO
- modifyCO-FCE
- modifyFC-PSE

Ausführung auf CO

Read-Set

- CO

Write-Set

- CO-FCE
- FC-PSE

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 26: prependChild(contextElementNode,childType,childValue) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und fügt einen neuen Knoten als erstes Kind ein. Der Kontextknoten besitzt zu diesem Zeitpunkt noch keine Kindknoten. Der neu eingefügte Knoten wird zurückgeliefert.

Szenario 26-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	CX	-	-	EX	EX
CN	NR	-	-	-	-

Szenario 26-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	CX	-	-	EX	EX
CN	NR	-	-	-	-

Szenario 26-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NRCX	-	-	EX	EX

Szenario 26-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NRCX	-	-	EX	EX

Basisoperationen

- readCO
- modifyCO-FCE
- modifyCO-LCE

Ausführung auf CO

Read-Set

- CO

Write-Set

- CO-FCE
- CO-LCE

Operation ausführbar auf den Knoten

PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, CA, AX

Use Case 27: insertBefore(contextElementNode,siblingType,siblingValue) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und fügt einen neuen Knoten als vorherigen Geschwisterknoten ein. Der Kontextknoten besitzt bereits einen vorherigen Geschwisterknoten. Der neu eingefügte Knoten wird zurückgeliefert.

Szenario 27-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	EX	-	-	-
CN	NR	-	-	-	-
PS	-	-	EX	-	-
PA	CX	-	-	-	-

Szenario 27-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	EX	-	-	-
CN	NR	-	-	-	-
PS	-	-	EX	-	-
PA	CX	-	-	-	-

Szenario 27-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	EX	-	-	-
PS	-	-	EX	-	-
PA	CX	-	-	-	-

Szenario 27-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	EX	-	-	-
PS	-	-	EX	-	-
PA	CX	-	-	-	-

Basisoperationen

- readCO
- modifyCO-PSE
- modifyPS-NSE

Ausführung auf CO

Read-Set

- CO

Write-Set

- CO-PSE
- PS-NSE

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 28: insertBefore(contextElementNode,siblingType,siblingValue) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und fügt einen neuen Knoten als vorherigen Geschwisterknoten ein. Der Kontextknoten besitzt keinen vorherigen Geschwisterknoten, d. h. er ist das erste Kind seines Elternknotens. Der neu eingefügte Knoten wird zurückgeliefert.

Szenario 28-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	EX	-	-	-
CN	NR	-	-	-	-
PA	CX	-	-	EX	-

Szenario 28-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	EX	-	-	-
CN	NR	-	-	-	-
PA	CX	-	-	EX	-

Szenario 28-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	EX	-	-	-
PA	CX	-	-	EX	-

Szenario 28-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	EX	-	-	-
PA	CX	-	-	EX	-

Basisoperationen

- readCO
- modifyCO-PSE
- modifyPA-FCE

Ausführung auf CO

Read-Set

- CO

Write-Set

- CO-PSE
- PA-FCE

Operation ausführbar auf den Knoten

AC, PA, PS, CO, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 29: insertAfter(contextElementNode,siblingType,siblingValue) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und fügt einen neuen Knoten als nächsten Geschwisterknoten ein. Der Kontextknoten besitzt bereits einen nächsten Geschwisterknoten. Der neu eingefügte Knoten wird zurückgeliefert.

Szenario 29-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	EX	-	-
CN	NR	-	-	-	-
NS	-	EX	-	-	-
PA	CX	-	-	-	-

Szenario 29-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	EX	-	-
CN	NR	-	-	-	-
NS	-	EX	-	-	-
PA	CX	-	-	-	-

Szenario 29-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	EX	-	-
NS	-	EX	-	-	-
PA	CX	-	-	-	-

Szenario 29-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	EX	-	-
NS	-	EX	-	-	-
PA	CX	-	-	-	-

Basisoperationen

- readCO
- modifyCO-NSE
- modifyNS-PSE

Ausführung auf CO

Read-Set

- CO

Write-Set

- CO-NSE
- NS-PSE

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 30: insertAfter(contextElementNode,siblingType,siblingValue) liefert Knoten

Beschreibung

Die Operation wird auf einem Elementknoten aufgerufen und fügt einen neuen Knoten als nächsten Geschwisterknoten ein. Der Kontextknoten besitzt keinen nächsten Geschwisterknoten, d. h. er ist das letzte Kind seines Elternknotens. Der neu eingefügte Knoten wird zurückgeliefert.

Szenario 30-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	EX	-	-
CN	NR	-	-	-	-
PA	CX	-	-	-	EX

Szenario 30-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	EX	-	-
CN	NR	-	-	-	-
PA	CX	-	-	-	EX

Szenario 30-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	EX	-	-
PA	CX	-	-	-	EX

Szenario 30-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	NR	-	EX	-	-
PA	CX	-	-	-	EX

Basisoperationen

- readCO
- modifyCO-NSE
- modifyPA-LCE

Ausführung auf CO

Read-Set

- CO

Write-Set

- CO-NSE
- PA-LCE

Operation ausführbar auf den Knoten

AC, PA, CO, NS, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, FC, CH, LC, DC, CA, AX

Use Case 31: deleteNode(contextNode)

Beschreibung

Die Operation löscht den angegebenen Kontextknoten. Der Kontextknoten besitzt einen vorherigen und nächsten Geschwisterknoten.

Szenario 31-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
PS	-	-	EX	-	-
NS	-	EX	-	-	-

Szenario 31-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
PS	-	-	EX	-	-
NS	-	EX	-	-	-

Szenario 31-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
PS	-	-	EX	-	-
NS	-	EX	-	-	-

Szenario 31-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
PS	-	-	EX	-	-
NS	-	EX	-	-	-

Basisoperationen

- modifyPS-NSE
- modifyNS-PSE
- writeCO
- writeFC
- writeCH
- writeLC
- writeDC

Read-Set

Write-Set

- PS-NSE
- NS-PSE
- CO
- FC
- CH
- LC
- DC
- CA
- AS
- AX
- XS

Ausführung auf CO

Operation ausführbar auf den Knoten

AC, PA, PS, CO, NS, CH, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 32: deleteNode(contextNode)

Beschreibung

Die Operation löscht den angegebenen Kontextknoten. Der Kontextknoten besitzt nur einen nächsten und keinen vorherigen Geschwisterknoten, d. h. der Kontextknoten ist das erste Kind seines Elternknotens.

Szenario 32-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
NS	-	EX	-	-	-
PA	-	-	-	EX	-

Szenario 32-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
NS	-	EX	-	-	-
PA	-	-	-	EX	-

Szenario 32-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
NS	-	EX	-	-	-
PA	-	-	-	EX	-

Szenario 32-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
NS	-	EX	-	-	-
PA	-	-	-	EX	-

Basisoperationen

- modifyPA-FCE
- modifyNS-PSE
- writeCO
- writeFC
- writeCH
- writeLC
- writeDC

Read-Set

Write-Set

- PA-FCE
- NS-PSE
- CO
- FC
- CH
- LC
- DC
- CA
- AS
- AX
- XS

Ausführung auf CO

Operation ausführbar auf den Knoten

AC, PA, PS, CO, FC, CH, DC

Vorhandene Knoten für diesen Use Case

AC, PA, CO, NS, FC, CH, LC, DC, CA, AX

Use Case 33: deleteNode(contextNode)

Beschreibung

Die Operation löscht den angegebenen Kontextknoten. Der Kontextknoten besitzt nur einen vorherigen und keinen nächsten Geschwisterknoten, d. h. der Kontextknoten ist das letzte Kind seines Elternknotens.

Basisoperationen

- modifyPA-LCE
- modifyPS-NSE
- writeCO
- writeFC
- writeCH
- writeLC
- writeDC

Read-Set

Write-Set

- PA-LCE
- PS-NSE
- CO
- FC
- CH
- LC
- DC
- CA
- AS
- AX
- XS

Ausführung auf CO

Szenario 33-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
PS	-	-	EX	-	-
PA	-	-	-	-	EX

Szenario 33-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
PS	-	-	EX	-	-
PA	-	-	-	-	EX

Szenario 33-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
PS	-	-	EX	-	-
PA	-	-	-	-	EX

Szenario 33-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
PS	-	-	EX	-	-
PA	-	-	-	-	EX

Operation ausführbar auf den Knoten

AC, PA, CO, NS, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, FC, CH, LC, DC, CA, AX

Use Case 34: deleteNode(contextNode)

Beschreibung

Die Operation löscht den angegebenen Kontextknoten. Der Kontextknoten besitzt keinen vorherigen und keinen nächsten Geschwisterknoten, d. h. der Kontextknoten ist das einzige Kind seines Elternknotens.

Basisoperationen

- modifyPA-FCE
- modifyPA-LCE
- writeCO
- writeFC
- writeCH
- writeLC
- writeDC

Read-Set

Write-Set

- PA-FCE
- PA-LCE
- CO
- FC
- CH
- LC
- DC
- CA
- AS
- AX
- XS

Ausführung auf CO

Szenario 34-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
PA	-	-	-	EX	EX

Szenario 34-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
PA	-	-	-	EX	EX

Szenario 34-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
PA	-	-	-	EX	EX

Szenario 34-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CO	SX	-	-	-	-
PA	-	-	-	EX	EX

Operation ausführbar auf den Knoten

AC, PA, CO, FC, CH, LC, DC

Vorhandene Knoten für diesen Use Case

AC, PA, CO, FC, CH, LC, DC, CA, AX

Use Case 35: deleteNode(contextAttributeNode)

Beschreibung

Die Operation löscht das angegebenen Attribut und den angehefteten String-Knoten.

Basisoperationen

- writeCA
- writeAS

Read-Set

Write-Set

- CA
- AS

Ausführung auf CO

Szenario 35-1 für taDOM2

Knoten	Sperre	PSE	NSE	FCE	LCE
CA	SX	-	-	-	-

Szenario 35-2 für taDOM2+

Knoten	Sperre	PSE	NSE	FCE	LCE
CA	SX	-	-	-	-

Szenario 35-3 für taDOM3

Knoten	Sperre	PSE	NSE	FCE	LCE
CA	SX	-	-	-	-

Szenario 35-4 für taDOM3+

Knoten	Sperre	PSE	NSE	FCE	LCE
CA	SX	-	-	-	-

Operation ausführbar auf den Knoten

CA, AX

Vorhandene Knoten für diesen Use Case

AC, PA, PS, CO, NS, FC, CH, LC, DC, CA, AX

ANHANG C Achsenüberlagerungstabellen

Vorhandene Sperre auf einem Knoten der *Ancestor*-Achse und nicht der *Parent*-Achse des Kontextknotens											
	Ancestor	Parent	Preceding	Preceding Sibling	Self	Attribute	Child	Descendant	Following Sibling	Following	IDvalue
Ancestor	ja	ja	nein	nein	ja	nein	ja	ja	nein	nein	nein
Parent	nein	nein	nein	nein	nein	nein	ja	ja	nein	nein	nein
Preceding	nein	nein	ja	ja	nein	nein	ja	ja	nein	nein	nein
Preceding Sibling	nein	nein	nein	nein	nein	nein	nein	ja	nein	nein	nein
Self	nein	nein	nein	nein	nein	nein	nein	ja	nein	nein	nein
Attribute	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein
Child	nein	nein	nein	nein	nein	nein	nein	ja	nein	nein	nein
Descendant	nein	nein	nein	nein	nein	nein	nein	ja	nein	nein	nein
Following Sibling	nein	nein	nein	nein	nein	nein	nein	ja	nein	nein	nein
Following	nein	nein	nein	nein	nein	nein	ja	ja	ja	ja	nein
IDvalue	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja

Vorhandene Sperre auf einem Knoten der *Parent*-Achse des Kontextknotens											
	Ancestor	Parent	Preceding	Preceding Sibling	Self	Attribute	Child	Descendant	Following Sibling	Following	IDvalue
Ancestor	ja	ja	nein	nein	ja	nein	nein	nein	nein	nein	nein
Parent	nein	nein	nein	nein	ja	nein	nein	nein	nein	nein	nein
Preceding	nein	nein	ja	ja	nein	nein	ja	ja	nein	nein	nein
Preceding Sibling	nein	nein	nein	nein	nein	nein	ja	ja	nein	nein	nein
Self	nein	nein	nein	nein	nein	nein	ja	ja	nein	nein	nein
Attribute	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein
Child	nein	nein	nein	nein	nein	nein	nein	ja	nein	nein	nein
Descendant	nein	nein	nein	nein	nein	nein	nein	ja	nein	nein	nein
Following Sibling	nein	nein	nein	nein	nein	nein	ja	ja	nein	nein	nein
Following	nein	nein	nein	nein	nein	nein	ja	ja	ja	ja	nein
IDvalue	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja

Vorhandene Sperre auf einem Knoten der *Preceding*-Achse und nicht der *Preceding-Sibling*-Achse des Kontextknotens											
	Ancestor	Parent	Preceding	Preceding Sibling	Self	Attribute	Child	Descendant	Following Sibling	Following	IDvalue
Ancestor	ja	ja	nein	nein	nein	nein	nein	nein	ja	ja	nein
Parent	ja	nein	nein	nein	nein	nein	nein	nein	ja	ja	nein
Preceding	ja	ja	ja	ja	ja	nein	ja	ja	ja	ja	nein
Preceding Sibling	ja	ja	nein	nein	nein	nein	nein	nein	nein	ja	nein
Self	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja	nein
Attribute	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein
Child	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja	nein
Descendant	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja	nein
Following Sibling	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja	nein
Following	nein	nein	nein	nein	nein	nein	nein	nein	ja	ja	nein
IDvalue	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein

Vorhandene Sperre auf einem Knoten der *Preceding-Sibling*-Achse des Kontextknotens											
	Ancestor	Parent	Preceding	Preceding Sibling	Self	Attribute	Child	Descendant	Following Sibling	Following	IDvalue
Ancestor	ja	ja	nein	nein	nein	nein	nein	nein	nein	nein	nein
Parent	ja	ja	nein	nein	nein	nein	nein	nein	nein	nein	nein
Preceding	nein	nein	ja	ja	ja	nein	ja	ja	ja	ja	nein
Preceding Sibling	nein	nein	ja	ja	ja	nein	nein	nein	ja	ja	nein
Self	nein	nein	nein	nein	nein	nein	nein	nein	ja	ja	nein
Attribute	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein
Child	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja	nein
Descendant	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja	nein
Following Sibling	nein	nein	nein	nein	nein	nein	nein	nein	ja	ja	nein
Following	nein	nein	nein	nein	nein	nein	nein	nein	ja	ja	nein
IDvalue	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein

Vorhandene Sperre auf einem Knoten der *Self*-Achse des Kontextknotens											
	Ancestor	Parent	Preceding	Preceding Sibling	Self	Attribute	Child	Descendant	Following Sibling	Following	IDvalue
Ancestor	ja	ja	nein	nein	nein	nein	nein	nein	nein	nein	nein
Parent	ja	ja	nein	nein	nein	nein	nein	nein	nein	nein	nein
Preceding	nein	nein	ja	ja	nein	nein	nein	nein	nein	nein	nein
Preceding Sibling	nein	nein	ja	ja	nein	nein	nein	nein	nein	nein	nein
Self	nein	nein	nein	nein	ja	nein	nein	nein	nein	nein	nein
Attribute	nein	nein	nein	nein	nein	ja	nein	nein	nein	nein	nein
Child	nein	nein	nein	nein	nein	nein	ja	ja	nein	nein	nein
Descendant	nein	nein	nein	nein	nein	nein	ja	ja	nein	nein	nein
Following Sibling	nein	nein	nein	nein	nein	nein	nein	nein	ja	ja	nein
Following	nein	nein	nein	nein	nein	nein	nein	nein	ja	ja	nein
IDvalue	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja

Vorhandene Sperre auf einem Knoten der *Child*-Achse des Kontextknotens											
	Ancestor	Parent	Preceding	Preceding Sibling	Self	Attribute	Child	Descendant	Following Sibling	Following	IDvalue
Ancestor	ja	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein
Parent	ja	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein
Preceding	nein	nein	ja	nein	nein	nein	nein	nein	nein	nein	nein
Preceding Sibling	nein	nein	ja	nein	nein	nein	nein	nein	nein	nein	nein
Self	ja	ja	nein	nein	nein	nein	nein	nein	nein	nein	nein
Attribute	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein
Child	nein	nein	ja	ja	ja	nein	nein	nein	ja	ja	nein
Descendant	nein	nein	ja	ja	ja	nein	ja	ja	ja	ja	nein
Following Sibling	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja	nein
Following	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja	nein
IDvalue	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja

Vorhandene Sperre auf einem Knoten der *Descendant*-Achse des Kontextknotens											
	Ancestor	Parent	Preceding	Preceding Sibling	Self	Attribute	Child	Descendant	Following Sibling	Following	IDvalue
Ancestor	ja	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein
Parent	ja	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein
Preceding	nein	nein	ja	nein	nein	nein	nein	nein	nein	nein	nein
Preceding Sibling	nein	nein	ja	nein	nein	nein	nein	nein	nein	nein	nein
Self	ja	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein
Attribute	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein
Child	ja	ja	ja	nein	nein	nein	nein	nein	nein	ja	nein
Descendant	ja	ja	ja	ja	ja	nein	ja	ja	ja	ja	nein
Following Sibling	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja	nein
Following	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja	nein
IDvalue	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	ja

Vorhandene Sperre auf einem Knoten der *Following-Sibling*-Achse des Kontextknotens											
	Ancestor	Parent	Preceding	Preceding Sibling	Self	Attribute	Child	Descendant	Following Sibling	Following	IDvalue
Ancestor	ja	ja	nein	nein	nein	nein	nein	nein	nein	nein	nein
Parent	ja	ja	nein	nein	nein	nein	nein	nein	nein	nein	nein
Preceding	nein	nein	ja	ja	nein	nein	nein	nein	nein	nein	nein
Preceding Sibling	nein	nein	ja	ja	nein	nein	nein	nein	nein	nein	nein
Self	nein	nein	ja	ja	nein	nein	nein	nein	nein	nein	nein
Attribute	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein
Child	nein	nein	ja	nein	nein	nein	nein	nein	nein	nein	nein
Descendant	nein	nein	ja	nein	nein	nein	nein	nein	nein	nein	nein
Following Sibling	nein	nein	ja	ja	ja	nein	nein	nein	ja	ja	nein
Following	nein	nein	ja	ja	ja	nein	ja	ja	ja	ja	nein
IDvalue	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein

Vorhandene Sperre auf einem Knoten der *Following*-Achse und nicht der *Following-Sibling*-Achse des Kontextknotens											
	Ancestor	Parent	Preceding	Preceding Sibling	Self	Attribute	Child	Descendant	Following Sibling	Following	IDvalue
Ancestor	ja	ja	ja	ja	nein	nein	nein	nein	nein	nein	nein
Parent	ja	nein	ja	ja	nein	nein	nein	nein	nein	nein	nein
Preceding	nein	nein	ja	ja	nein	nein	nein	nein	nein	nein	nein
Preceding Sibling	nein	nein	ja	nein	nein	nein	nein	nein	nein	nein	nein
Self	nein	nein	ja	nein	nein	nein	nein	nein	nein	nein	nein
Attribute	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein
Child	nein	nein	ja	nein	nein	nein	nein	nein	nein	nein	nein
Descendant	nein	nein	ja	nein	nein	nein	nein	nein	nein	nein	nein
Following Sibling	ja	ja	ja	nein	nein	nein	nein	nein	nein	nein	nein
Following	ja	ja	ja	ja	ja	nein	ja	ja	ja	ja	nein
IDvalue	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein	nein

Literaturverzeichnis

BCJ+05 Beyer, K., Cochrane, R. J., Josifovski, V., Kleewein, J., Lapis, G., Lohman, G., Lyle, B., Özcan, F., Pirahesh, H., Seemann, N., Truong, T., Van der Linden, B., Vickery, B., Zhang, C.: *System RX: One Part Relational, One Part XML*. SIGMOD-Konferenz 2005, Baltimore, Maryland, USA, Juni 2005, 374-358

BDG+01 Birbeck, M., Duckett, J., Gudmundsson, O. G., Kobak, P., Lenz, E., Livingstone, S., Marcus, D., Mohr, S., Pinnock, J., Visco, K., Watt, A., Williams, K., Zaev, Z., Ozu, N.: *Professional XML 2nd Edition.* Wrox Press, UK, 2001

Be03 Berscheid, G.: *taDOM – Synchronisation für XML-Dokumente, Implementierung und Evaluierung*. Diplomarbeit, AG DBIS, TU Kaiserslautern, September 2003

BDL+01 Bressan, S., Dobbie, G., Lacroix, Z., Lee, M. L., Li, Y. G., Wadhwa, B., Nambiar, U.: *XOO7: Applying OO7 Benchmark to XML Query Processing Tools*. 10. CIKM-Konferenz, Atlanta, Georgia, USA, November 2001, 167-174

BHL99 Bray, T., Hollander, D., Layman, A.: *Namespaces in XML*. W3C Recommendation, Januar 1999

Br02 Brownell, D.: *SAX2 – Processing XML Efficiently with Java*. O'Reilly 2002

BPS98 Bray, T., Paoli, J., Sperberg-McQueen, C. M.: *Extensible Markup Language (XML) 1.0.* W3C Recommendation, Februar 1998

BPS+04 Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F.: *Extensible Markup Language (XML) 1.0 (Third Edition)*. W3C Recommendation, Februar 2004

BR01 Böhme, T., Rahm, E.: *XMach-1: A Benchmark for XML Data Management.* 9. GI-Fachtagung für Datenbanksysteme in Büro, Technik und Wissenschaft, Oldenburg, Deutschland, März 2001, 264-273

BR02 Böhme, T., Rahm, E.: *XML-Datenbanksysteme – Architekturen und Benchmarks.* Fraunhofer Innovationsforum, Stuttgart, Deutschland, Juni 2002, 1-14

BR04 Böhme, T., Rahm, E.: *Supporting Efficient Streaming and Insertion of XML Data in RDBMS.* 3. DIWeb-Workshop, Riga, Lettland, Juni 2004, 70-81

BT01 Böttcher, S., Türling, A.: *Transaction Synchronization for XML Data in Client-Server Web Applications*. GI-Workshop „Web-Datenbanken“, Wien, Österreich, 2001, 388-395

CDF+94 Carey, M. J., DeWitt, D. J., Franklin, M. J., Hall, N. E., McAuliffe, M. L., Naughton, J. F., Schuh, D. T., Solomon, M. H., Tan, C. K., Tsatalos, O. G., White, S. J., Zwilling, M. J.: *Shoring Up Persistent Applications*. SIGMOD-Konferenz, Minneapolis, Minnesote, USA, Mai 1994, 383-394

CDN93 Carey, M. J., DeWitt, D. J., Naughton, J. F.: *The OO7 Benchmark*. SIGMOD-Konferenz, Washington, D. C., USA, Mai 1993, 12-21

CGP99 Clarke, E. M., Grumberg, O., Peled, D. A.: *Model Checking*. MIT Press, 1999

CM01 Clark, J., Makoto, M.: *RELAX NG Specification.* Oasis Committee Specification, Dezember 2001

CMB+94 Chen, Y., Mihaila, G. A., Bordawekar, R., Padmanabhan, S.: *L-Tree: a Dynamic Labeling Structure for Ordered XML Data*. EDBT PhD Workshop, Heraklion, Kreta, Griechenland, März 2004, 209-218

Co79 Comer, D.: *The Ubiquitous B-Tree*. ACM Computing Surveys, Vol. 11, No 2, Juni 1979, 121-137

CRZ03 Chaudhri, A. B., Rashid, A., Zicari, R.: *XML Data Management – Native XML and XML-Enabled Database Systems*. Addison-Wesley, 2003

CTZ+01 Chien, S.-Y., Tsotras, V. J., Zaniolo, C., Zhang, D.: *Storing and Querying Multiversion XML Documents using Durable Node Numbers*. 2. WISE-Konferenz, Kyoto, Japan, Dezember 2001, 232-252

DB2X IBM DB2 Universal Database: *XML Extender Administration and Programming*. IBM Corporation, 1999-2004

DEW Online Computer Library Center: *The Dewey Decimal Classification (DDC)*, http://www.oclc.org/dewey

DH02 Dekeyser, S., Hidders, J.: *Path Locks for XML Document Collaboration*. 3. WISE-Konferenz, Singapur, Dezember 2002, 105-114

DHP04 Dekeyser, S., Hidders, J., Paredaens, J.: *A Transaction Model for XML Databases*. World Wide Web Journal, Volume 7, Issue 2, Juni 2004, 29-57

Di82 Dietz, P. F.: *Maintaining Order in a Linked List.* 14. ACM Symposium of Theory of Computing, San Fransisco, Kalifornien, USA, Mai 1982, 122-127

DOM1 Wood, L., Le Hors, A., Apparao, V., Byrne, S., Champion, M., Isaacs, S., Jacobs, I., Nicol, G., Robie, J., Sutor, R., Wilson, C.: *Document Object Model (DOM) Level 1 Specification (Second Edition)*. W3C Working Draft, September 2000

DOM2 Le Hors, A., Le Hégaret, P., Wood, L., Nicol, G., Robie, J., Champion, M., Byrne, S.: *Document Object Model (DOM) Level 2 Core Specification, Version 1.0*. W3C Recommendation, November 2000

DOM3 Le Hors, A., Le Hégaret, P., Wood, L., Nicol, G., Robie, J., Champion, M., Byrne, S.: *Document Object Model (DOM) Level 3 Core Specification, Version 1.0.* W3C Recommendation, April 2004

DXR Computer Science Bibliography, Universität Trier: *DBLP XML Records*. http://www.informatik.uni-trier.de/~ley/db/

EGL+97 Eswaran, K. P., Gray, J. N., Lorie, R. A., Traiger, I. L.: *The Notions of Consistency and Predicate Locks in a Database System*. Communications of the ACM, Volume 19, Issue 11, November 1976, 624-633

EH84 Effelsberg, W., Härder, T.: *Principles of Database Buffer Management*. ACM Transactions on Database Systems, Vol. 9, No. 4, Dezember 1984, 560-595

FHK+02a Fiebig, T., Helmer, S., Kanne, C.-C., Mildenberger, J., Moerkotte, G., Schiele, R., Westmann, T.: *Anatomy of a Native XML Base Management System*. Interner Bericht, Reihe Informatik, Band 1, Fakultät für Mathematik und Informatik, Universität Mannheim, 2002

FHK+02b Fiebig, T., Helmer, S., Kanne, C.-C., Moerkotte, G., Neumann, J. Schiele, R., Westmann, T.: *Anatomy of a native XML base management system*. VLDB Journal, Volume 11, Issue 4, Dezember 2002, 292-314

FHK+03 Fiebig, T., Helmer, S., Kanne, C.-C., Moerkotte, G., Neumann, J., Schiele R., Westmann, T.: *Natix: A Technology Overview*. Web Databases and Web Services, LNCS 2593, Springer-Verlag, 2003, 12-33

FKM01 Fiebig, T., Kanne, C.-C., Moerkotte, G.: *Natix – ein natives XML-DBMS*. Datenbank-Spektrum, Nummer 1, 2002, 5-13

Fl05 Flehmig, M.: *Datenintegration über das Web mit SHARX*. Dissertation, Arbeitsgruppe Datenbanken und Informationssysteme, Technische Universität Kaiserslautern, 2005

FM00 Fiebig, T., Moerkotte, G.: *Evaluating Queries on Structure with eXtended Access Support Relations*. 3. WebDB-Workshop 2000, Dallas, Texas, USA, Mai 2000, 41-46

GB03 Gertz, M., Bremer, J.-M.: *Distributed XML Repositories: Top-down Design and Transparent Query Processing*. Technischer Report CSE-2003-20, University of California, DBIS group, 2003

GBS02 Grabs, T., Böhm, K., Schek, H.-J.: *XMLTM: Efficient Transaction Management for XML Documents*. 11. CIKM-Konferenz, McLean, Virginia, USA, November 2002, 142-152

GFK04 Grinev, M., Formichev, A., Kuznetsov, S.: *Sedna: A Native XML DBMS*. Interner Bericht, Institute for System Programming of the Russian Academy of Science, 2004

GLP+76 Gray, J. N., Lorie, R. A., Putzolu, G. R., Traiger, I. L.: *Granularity of Locks and Degrees of Consistency in a Shared Data Base*. IFIP Working Conference on Modelling in Data Base Management Systems. North Holland 1976, 365-394

Gr91 Gray, J.: *The Benchmark Handbook for Database and Transaction Processing Systems.* Morgan Kaufmann Publishers, 1991

GR93 Gray, J. N., Reuter, A.: *Transaction Processing – Concepts and Techniques*. Morgan Kaufmann Publishers, 1993

GW97 Goldman, R., Widom, J.: *DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases*. 29. VLDB-Konferenz, Athen, Griechenland, August 1997, 436-445

Ha05a Haustein, M. P.: *Eine XML-Programmierschnittstelle zur transaktionsgeschützten Kombination von DOM, SAX und XQuery*. 11. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW), Karlsruhe, Deutschland, März 2005, 265-284

Ha05b Haustein, M. P.: *Verhinderung von Phantomen in XML-Datenbanksystemen mit wertbasierten Achsensperren*. Berliner XML Tage, Berlin, Deutschland, September 2005, 79-92

HH03 Haustein, M., Härder, T.: *taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API*. 7. ABDIS-Konferenz, Dresden, Deutschland, September 2003, 88-102

HH04a Haustein, M., Härder, T.: *Adjustable Transaction Isolation in XML Database Management Systems*. 2. Internationales XML Datenbank Symposium, Toronto, Kanada, August 2004, 173-188

HH04b Haustein, M., Härder, T.: *A Lock-Manager for Collaborative Processing of Natively Stored XML Documents*. 19. SBBD-Konferenz, Brasilia, Brasilien, Oktober 2004, 230-244

HHM+05 Haustein, M., Härder, T., Mathis, C., Wagner, M.: *DeweyIDs – The Key to Fine-Grained Management of XML Documents*. 20. SBBD-Konferenz, Uberlandia, Brasilien, Oktober 2005

HKM01 Helmer, S., Kanne, C.-C., Moerkotte, G.: *Isolation in XML Bases*. Interner Bericht, Reihe Informatik, Band 15, Fakultät für Mathematik und Informatik, Universität Mannheim, 2001

HKM03 Helmer, S., Kanne, C.-C., Moerkotte, G.: *Lock-based Protocols for Cooperation on XML Documents*. 14th International Workshop on Database and Expert System Applications, Prag, Tschechien, September 2003, 230-246

HKM04 Helmer, S., Kanne, C.-C., Moerkotte, G.: *Evaluating Lock-based Protocols for Cooperation on XML Documents*. SIGMOD Record, Volume 33, No. 1, März 2004, 58-63

HR83 Härder, T., Reuter A.: *Concepts for Implementing a Centralized Database Management System*. International Computing Symposium on Application Systems Development, Nürnberg, Deutschland, März 1983, 28-60

HR99 Härder, T., Rahm, E.: *Datenbanksysteme – Konzepte und Techniken der Implementierung*. Springer-Verlag, 1999

JBB81 Jordan, J. R., Banerjee, J., Batman, R. B.: *Precision Locks*. SIGMOD-Konferenz, Ann Arbor, Michigan, USA, April 1981, 143-147

JKC+02 Jagadish, H. V., Al Khalifa, S., Chapman, A., Lakshmanan, L. V. S., Nierman, A., Paparizos, S., Patel, J. M., Srivastava, D., Wiwatwattana, N., Wu, Y., Yu, C.: *TIMBER: A native XML database*. VLDB Journal, Volume 11, Issue 4, Dezember 2002, 274-291

Le01 Lehti, P.: *Design and Implementation of a Data Manipulation Processor for an XML Query Language*. Diplomarbeit, Technische Universität Darmstadt, August 2001

LYY96 Lee, Y. K., Yoo, S.-J., Yoon, K.: *Index Structures for Structured Documents*. 1. ACM-Konferenz über Digitale Bibliotheken, Bethesda, Maryland, USA, März 1996, 91-99

LM00 Laux, A., Martin, L.: *XUpdate – XML Update Language*. Working Draft, http://xmldb-org.sourceforge.net/xupdate, September 2000

LM01 Li, Q., Moon, B.: *Indexing and Querying XML Data for Regular Path Expressions*. 27. VLDB-Konferenz, Rom, Italien, September 2001, 361-370

LS04 Lehner, W., Schöning, H.: *XQuery – Grundlagen und fortgeschrittene Methoden*. Dpunkt-Verlag, 2004

Lu05 Luttenberger, K.: *Sperrprotokolle in XML-Datenbanksystemen*. Diplomarbeit, AG DBIS, TU Kaiserslautern, April 2005

Ma04 Mathis, C.: *Anwendungsprogrammierschnittstellen für XML-Datenbanksysteme*. Diplomarbeit, AG DBIS, TU Kaiserslautern, September 2004

MH05 Mathis, C., Härder, T.: A *Query Processing Approach for XML Database Systems*. 17. Workshop über Grundlagen von Datenbanksystemen, Wörlitz, Deutschland, Mai 2005, 89-93

Me02 Meier, W.: *eXist: An Open Source Native XML Database*. NODe-Konferenz, Web- and Database-Related Workshops, LNCS 2593, Springer-Verlag, Erfurt, Deutschland, Oktober 2002, 169-183

MLL+03 Meng, X., Luo, D., Lee, M. L., An, J.: *OrientStore: A Schema Based Native XML Storage System*. 29. VLDB-Konferenz, Berlin, Deutschland, September 2003, 1057-1060

MWL+04 Meng, X., Wang, Y., Luo, D., Lu, S., An J., Chen, Y., Ou, J., Jiang, Y.: *OrientX: A Schema-based Native XML Database System*. Interner Bericht, Information School, Renmin University of China, März 2004

NNP+04 O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N.: *ORDPATHS: Insert-Friendly XML Node Labels*. SIGMOD-Konferenz, Paris, Frankreich, Juni 2004, 903-908

Ob02a Obasanjo, D.: *SiXDML – Simple XML Data Manipulation Language*. Working Draft, http://xmldb-org.sourceforge.net/sixdml/sixdml-lang.html, April 2002

Ob02b Obasanjo, D.: *SiXDML-API Working Draft* http://xmldb-org.sourceforge.net/sixdml/sixdml-api.html, Juni 2002

Ph04 Pharo, H.: *Synchronisationsprobleme bei der Verarbeitung von XML-Dokumenten in relationalen Datenbanksystemen*. Diplomarbeit, AG DBIS, TU Kaiserslautern, August 2004

PN04 Pleshachkov, P., Novak, L.: *Transaction Isolation In the Sedna Native XML DBMS*. Spring Young Researcher's Colloquium On Database and Information Systems (SYRCoDIS), St.-Petersburg, Russland, Mai 2004

SAX Simple API for XML: http://www.saxproject.org

Sc01 Schöning, H.: *Tamino – a DBMS Designed for XML*. 17. ICDE-Konferenz, Heidelberg, Deutschland, April 2001, 149-154

ScR01 Schiele, R.: *NatiXync – Synchronisation für XML-Datenbanksysteme*. Diplomarbeit, Universität Mannheim, Lehrstuhl für Praktische Informatik III, September 2001

Sc03 Schöning, H.: *XML und Datenbanken*. Carl Hanser Verlag, 2003

Sch04 *ISO Schematron Specification*, ISO/IEC FDIS 19757-3, Oktober 2004

SED05 *Sedna Programmer's Guide*. http://www.modis.ispras.ru/Development/sedna.htm, April 2004

SGML86 *Standard Generalized Markup Language (SGML)*, ISO 8879, International Organization for Standardization, Information Processing, Text and Office Systems, 1986

SHY+05 Silberstein, A., He, H., Yi, K., Yang, J.: *BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data*. 21. ICDE-Konferenz, Tokyo, Japan, April 2005, 285-296

SOAP03a Mitra, N.: *SOAP Version 1.2 Part 0: Primer*. W3C Recommendation, Juni 2003

SOAP03b Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Nielson, H. F.: *SOAP Version 1.2 Part 1: Messaging Framework.* W3C Recommendation, Juni 2003

SOAP03c Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Nielson, H. F.: *SOAP Version 1.2 Part 2: Adjuncts*. W3C Recommendation, Juni 2003

St01 Staken, K.: *XML:DB Database API*. Working Draft, http://xmldb-org.sourceforge.net/xapi, September 2001

SW00 Schöning, H., Wäsch, J.: *Tamino – An Internet Database System*. 7. EDBT-Konferenz, Konstanz, Deutschland, März 2000, 383-387

SWK+01 Schmidt, A. R., Waas, F., Kersten, M. L., Florescu, D., Manulesco, I., Carey, M.J., Busse, R.: *The XML Benchmark Project.* Bericht INS-R0103, National Research Institute for Mathematics and Computer Science, Amsterdam, Holland, April 2001

SWK+02 Schmidt, A., Waas, F., Kersten, M., Carey, M. J., Manulesco, I., Busse, R.: *XMark: A Benchmark for XML Data Management*. 28. VLDB-Konferenz, Hong Kong, China, August 2002, 974-985

TBS+02 Tatarinov, I., Beyer, K., Shanmugasundaram, J., Viglas, S., Shekita, E., Zhang, C.: *Storing and Querying Ordered XML Using a Relational Database System*. SIGMOD-Konferenz, Madison, Wisconsin, USA, Juni 2002, 204-215

TFW02 Tesch, T., Fankhauser, P., Weitzel, T.: *Skalierbare Verarbeitung von XML mit Infonyte-DB*. Wirtschaftsinformatik, Ausgabe 44, Nummer 5, 2002, 469-475

TPCH Transaction Processing Council: *TPC-H Decision Support Benchmark.* http://www.tpc.org/tpch

Wa73 Wagner, R. E.: *Indexing design considerations*. IBM Systems Journal, No. 4, 1973, 351-367

Wa05a Wagner, M.: *Die Dewey-Dezimalklassifikation in XML-Datenbanksystemen*. Projektarbeit, Arbeitsgruppe Datenbanken und Informationssysteme, Technische Universität Kaiserslautern, März 2005

Wa05b Wagner, M.: *Speicherungsstrukturen für native XML-Datenbanksysteme*. Diplomarbeit, Arbeitsgruppe Datenbanken und Informationssysteme, Technische Universität Kaiserslautern, September 2005

Wal02 Walmsley, P.: *Definitive XML Schema.* Prentice Hall PTR, New Jersey, 2002

WDAV99 Goland, Y., Whitehead, E., Faizi, A., Carter, S. R., Jensen, D.: *HTTP Extensions for Distrubuted Authoring – WEBDAV*. RfC2518, Februar 1999

WDAV02 Clemm, G., Amsden, J., Ellison, T., Kaker, C., Whitehead, J.: *Versioning Extensions to WebDAV*. RfC3253, März 2002

WDAV03 Whitehead, J., Reschke, J. F.: *WebDAV Ordered Collections Protocol*. RfC3648, Dezember 2003

WDAV04 Clemm, G., Reschke, J. F., Sedlar, E., Whitehead, J.: *WebDAV Access Control Protocol*. RfC3744, Mai 2004

Wi03 Winer, D.: *XML-RPC Specification, 3rd Update*. XML-RPC Homepage, http://www.xml-rpc.com, Juni 2003

WJL+03 Wang, W., Jiang, H., Lu, H., Yu, J. X.: *PBiTree Coding and Efficient Processing of Containment Joins*. 19. ICDE-Konferenz, Bangalore, Indien, März 2003, 391-402

WLH04 Wu, X., Lee, M., Hsu, W.: *A Prime Number Labeling Scheme for Dynamic Ordered XML Trees*. 20. ICDE-Konferenz, Boston, USA, April 2004, 66-78

WSM05 Weigel, F., Schulz, K. U., Meuss, H.: *The BIRD Numbering Scheme for XML and Tree Databases – Deciding and Reconstructing Tree Relations using Efficient Arithmetic Operations*. 3. XML Symposium (XSYM), Trondheim, Norwegen, August 2005, 49-67

WXDR University of Washington, Dept. of Computer Science & Engineering: *XML Data Repository.* http://www.cs.washington.edu/research/xmldatasets

XIN05a Staken, K.: *Xindice 1.1 User Guide*. http://xml.apache.org/xindice/guide-user.html, 2005

XIN05b Staken, K., Viner, D.: *Xindice 1.1 Developer Guide.* http://xml.apache.org/xindice/guide-developer.html, 2005

XIN05c *Xindice 1.1 Internals Guide.* http://xml.apache.org/xindice/dev/guide-internals.html, 2005

XIN05d Staken, K., Rabellino, G.: *Xindice 1.1 Administration Guide*. http://xml.apache.org/xindice/guide-administrator.html, 2005

XMLDB XML:DB-Initiative: http://xmldb-org.sourceforge.net

XQL XQuery Specifications, W3C Consortium, http://www.w3c.org/XML/Query

XQL05a Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., Siméon, J.: *XQuery 1.0: An XML Query Language*. W3C Working Draft, April 2005

XQL05b Draper, D., Fankhauser, P., Fernández, M., Malhotra, A., Rose, K., Rys, M., Siméon, J., Wadler, P.: *XQuery 1.0 and XPath 2.0 Formal Semantics*. W3C Working Draft, Juni 2005

XQL05c Fernández, M., Malhotra, A., Marsh, J., Nagy, M., Walsh, N.: *XQuery 1.0 and XPath 2.0 Data Model*. W3C Working Draft, April 2005

XQL05d Malhotra, A., Melton, J., Walsh, N.: *XQuery 1.0 and XPath 2.0 Functions and Operators*. W3C Working Draft, April 2005

XQL05e Melton, J., Muralidhar, S.: *XML Syntax for XQuery (XQueryX)*. W3C Working Draft, April 2005

XQL05f Berglund, A., Boag, S., Chamberlin, D., Fernández, M. F., Kay, M., Robie, J., Siméon, J.: *XML Path Language (XPath) 2.0*. W3C Working Draft, April 2005

XSD04a Fallside, D. C., Walmsley, P.: *XML Schema Part 0: Primer Second Edition.* W3C Recommendation, Oktober 2004

XSD04b Thompson, H. S., Beech, D., Maloney, M., Mendelsohn, N.: *XML Schema Part 1: Structures Second Edition.* W3C Recommendation, Oktober 2004

XSD04c Biron, P. V., Malhotra, A.: *XML Schema Part 2: Datatypes Second Edition.* W3C Recommendation, Oktober 2004

Lebenslauf

Persönliche Daten

Name:	Michael Peter Haustein
Geburtsdatum:	16. Juni 1975
Geburtsort:	Landau/Pfalz
Familienstand:	verheiratet, 1 Kind

Schulausbildung

1981-1985	Grundschule Gerolsheim
1985-1994	Albert-Einstein-Gymnasium Frankenthal/Pfalz
6/1994	Allgemeine Hochschulreife

Wehrdienst

7/1994-10/1994	Militärische Grundausbildung 4./Jägerbataillon 292 Donaueschingen

Zivildienst

10/1994-9/1995	Disposition der Einsatzfahrzeuge im Behindertenfahrdienst, Blutkonserven- und Organtransport Johanniter-Unfall-Hilfe Ludwigshafen/Rhein

Studium

10/1995	Immatrikulation an der Universität Kaiserslautern Studiengang Informatik (Diplom)
8/1999-9/1999	Praktikum bei der SI-Software Innovation in Neustadt/Weinstraße (Entwurf und Implementierung einer Workflow-Management-Engine)
5/2002	Abschluss der Diplomarbeit mit dem Titel „Ähnlichkeitssuche in objekt-relationalen Datenbanksystemen“
5/2002	Erlangen des akademischen Grades „Dipl.-Inform.“

Berufstätigkeit

seit 7/2002	Wissenschaftlicher Mitarbeiter Arbeitsgruppe Datenbanken und Informationssysteme Technische Universität Kaiserslautern