Eine XML-Programmierschnittstelle zur transaktionsgeschützten Kombination von DOM, SAX und XQuery

Michael P. Haustein

Arbeitsgruppe Datenbanken und Informationssysteme
Fachbereich Informatik
Technische Universität Kaiserslautern
Postfach 3049
67653 Kaiserslautern
haustein@informatik.uni-kl.de

Abstract: Ereignisbasiertes Parsen, navigationsorientiertes Modifizieren oder deskriptives Anfragen sind typische Operationen der Datenverarbeitung auf XML-Dokumenten. Für diese Aufgaben werden die De-facto-Standards SAX, DOM und XQuery genutzt. Obwohl diese Ansätze schon länger eingesetzt werden, können sie nicht von mehreren Anwendungen gleichzeitig auf einem gemeinsamen XML-Dokument in einer transaktionsgeschützten Umgebung benutzt werden. In diesem Beitrag beschreiben wir unser natives XML-Datenbanksystem XTC und erläutern detailliert die Anwendungsprogrammierschnittstelle, die die Isolation nebenläufiger Transaktionen gewährleistet. Dabei können Anwendungen mittels eines speziellen Treiberpakets gleichzeitig mit SAX-, DOM- und XQuery-Zugriffen auf einem gemeinsamen XML-Dokument Anfrage- und Modifikationsoperationen durchführen.

1 Einleitung

Zur Verarbeitung von XML-Daten innerhalb eines Anwendungsprogramms werden meist De-facto-Standards wie die *Simple API for XML* (SAX) [Br02], die *Document-Object-Model*-Schnittstelle (DOM) [Ho04] oder die *XML Query Language* (XQuery) [Bo04] eingesetzt. Diese Lösungen bieten konzeptionell verschiedenartige Zugriffsmöglichkeiten auf XML-Dokumente, die je nach Problemstellung der Anwendung ausgewählt werden sollten und daher durchaus auch parallel zum Einsatz kommen können.

Ein SAX Parser bietet dem Anwender einen ereignisbasierten Zugriff auf ein XML-Dokument. Dazu wird das gesamte Dokument sequentiell durchlaufen und für jede erkannte Komponente des Dokuments (Element, Attribut, Text, Kommentar, usw.) eine so genannte *Callback-Methode* in einer speziellen dem Parser übergebenen Klasse (bezeichnet als *Content Handler*) aufgerufen. Auf diese Weise wird das XML-Dokument der Anwendung als Folge von auftretenden Ereignissen dargestellt. Durch die konkrete Implementierung der Content-Handler-Klasse hat der Anwender die Möglichkeit, auf die durch den Parser erzeugten Ereignisse zu reagieren und die gewünschte Semantik in die

Verarbeitung zu integrieren. Ein Vorteil dieses Verfahrens ist die hohe Verarbeitungsgeschwindigkeit und der geringe Hauptspeicherverbrauch, da der Parser das XML-Dokument rein sequentiell durchläuft, die dabei erkannten Komponenten durch die jeweiligen Funktionsaufrufe meldet, aber darüber hinaus keine interne Repräsentation des Dokuments aufbaut. Als sehr aufwendig erweist es sich aus Sicht des Programmierers, auf bereits zurückliegende Ereignisse (d.h. vom Parser bereits verarbeitete Dokumententeile) zuzugreifen, da dies mit eigenen Datenstrukturen zusätzlich implementiert werden muss. Darüber hinaus ist eine Änderung des XML-Dokuments nicht möglich.

Um solche Funktionalität zu unterstützen, repräsentiert die DOM-Schnittstelle ein XML-Dokument als Baumstruktur und bietet Methoden zur Navigation und Modifikation sowie zur einfachen Suche an. Anwendungen können mit entsprechenden Methoden zur gewünschten Position innerhalb eines Dokuments navigieren, direkt zu einem Element im Dokument „springen“ und ein so adressiertes Objekt auslesen, ändern oder löschen. Im Gegensatz zu SAX bietet DOM damit den Vorteil, dass ein Anwender jederzeit Zugriff auf das gesamte Dokument hat und bereits verarbeitete Dokumententeile erneut bearbeiten und auch beliebig verändern kann. Bei den meisten Implementierungen wird jedoch das gesamte XML-Dokument vor dem eigentlichen Zugriff vom DOM-Prozessor vollständig im Hauptspeicherbereich der Anwendung materialisiert und benötigt somit (vor allem bei größeren Dokumenten) enorm viele Ressourcen. Dieses Problem wird durch Lösungen entschärft, die im Hauptspeicher speziell komprimierte Repräsentationen eines Dokuments verwalten oder nur benötigte Dokumententeile laden [TFW03].

Die Anfragesprache XQuery bietet ergänzend zu den beiden bereits vorgestellten Ansätzen einen deskriptiven Zugriff auf XML-Daten. Dazu wird typischerweise nur spezifiziert, welche Teile eines Dokuments von Interesse sind und wie diese als Ergebnis dargestellt werden sollen. Die Wahl der geeigneten Zugriffsverfahren, die Auswertung der Anfrage und die Konstruktion des Ergebnisses werden dem XQuery-Prozessor überlassen. Änderungsoperationen können zurzeit noch nicht durchgeführt werden, es existieren jedoch einige Vorschläge. Da sich XQuery mit großer Wahrscheinlichkeit zur Standardanfragesprache für XML-Datenbanksysteme entwickeln wird, werden sicherlich auch in absehbarer Zeit Sprachkonstrukte zur Dokumentenmodifikation aufgenommen. Als etwas problematischer erweist sich die Verarbeitung eines Anfrageergebnisses auf Anwendungsseite, da das Ergebnis (in vielen Fällen in Form eines XML-Dokuments) zur Weiterverarbeitung in einer Wirtssprache meist wiederum SAX oder DOM unterzogen wird. Das Propagieren von Änderungen im Ergebnisdokument auf das Quelldokument der Anfrage (*updatable view*) ist damit nicht mehr möglich, da das erzeugte Ergebnisdokument keinerlei Rückschlüsse auf die Herkunft der enthaltenen Daten erlaubt.

Aufgrund der konzeptionell verschiedenen Zugriffsarten der beschriebenen Schnittstellentypen und der damit verbundenen Vorteile im jeweiligen Anwendungsgebiet sollte ein XML-Datenbanksystem einer Anwendung alle drei Schnittstellen anbieten. Da durch den Einsatz eines Datenbanksystems XML-Dokumente nicht mehr im Dateisystem vorliegen, sondern den Anwendungen über Datenbankverbindungen zur Verfügung gestellt werden, muss es darüber hinaus möglich sein, dass einerseits mehrere Anwendungen die Datenoperationen der drei Schnittstellen gleichzeitig auf einem Dokument durchführen können, und andererseits eine einzelne Anwendung die drei Schnittstellen in einem gemeinsamen Verarbeitungskontext anwenden kann. Zur Realisierung solch einer Programmierschnittstelle haben wir unser natives XML-Datenbanksystem *XML Transaction*

Coordinator (XTC) mit einem Treiberpaket versehen, das einem Anwendungsentwickler erlaubt, SAX, DOM und XQuery gemeinsam in einer transaktionsgeschützten Umgebung zu verwenden. Dabei können mehrere Anwendungsprogramme in jeweils eigenen Transaktionskontexten mit wählbarer Isolationsstufe gleichzeitig auf ein im Datenbanksystem zentral verwaltetes XML-Dokument zugreifen. Für Änderungen im Ergebnisdokument einer XQuery-Anfrage wird zusätzlich die Möglichkeit geboten, diese Änderungen sofort auf das Quelldokument der Anfrage zu propagieren.

Zur detaillierten Erläuterung unserer Programmierschnittstelle und ihrer prototypischen Realisierung ist dieser Beitrag wie folgt gegliedert. Zunächst gibt Kapitel 2 einen Überblick über die Systemarchitektur des Forschungsprojekts XTC, die darin implementierten Techniken zur Speicherung von XML-Dokumenten und Lösungen zur Transaktionsverwaltung. Die Programmierschnittstelle auf Anwendungsseite und die im Datenbanksystem durchzuführenden Operationen zur Anfrageverarbeitung und Transaktionsisolation werden in Kapitel 3 beschrieben. Kapitel 4 fasst den Inhalt des Beitrags zusammen und geht auf die künftigen Forschungsarbeiten ein.

2 Systemarchitektur

Der Entwurf des nativen XML-Datenbanksystems XTC orientiert sich am Modell der Fünf-Schichten-Architektur für Datenbanksysteme [HR99]. Eine kurze Beschreibung der einzelnen Schichten und der jeweils implementierten Funktionalität wird in Abschnitt 2.1 gegeben. Abschnitt 2.2 widmet sich dem internen Speicherungsmodell für XML-Dokumente und beschreibt unsere Erfahrungen mit zwei alternativen Implementierungen. Schließlich werden in Abschnitt 2.3 und 2.4 die Techniken für Logging und Recovery und die für diesen Beitrag relevanten Sperrmechanismen zur Isolation erläutert.

2.1 Architektur des Datenbanksystems

XTC-Server ist ein vollständig in Java implementiertes natives XML-Datenbanksystem (XDBS), das aus fünf aufeinander aufbauenden Schichten besteht (Abbildung 1). Die zusätzliche Schicht der Schnittstellendienste bietet den Datenzugriff für Standardanwendungen über das Http- oder Ftp-Protokoll an und sorgt für die Kommunikation mit dem bei der Anwendungsentwicklung eingesetzten Treiberpaket. Diese Treiber ermöglichen einer Anwendung den Zugriff über die SAX- und DOM-Schnittstelle und das Ausführen und Bearbeiten von XQuery-Anfragen über eine Java-RMI-Verbindung.

Die Dateidienste bilden die unterste Schicht und speichern mit Hilfe der *E/A-Manager* Datenseiten als einzelne Blöcke in den so genannten *Containerdateien* ab. Jedem E/A-Manager wird eine Containerdatei zugewiesen, die bei der Installation des XDBS mit einer Initialgröße erzeugt und bei vollständiger Belegung automatisch um eine zuvor festgelegte Größe erweitert wird.

In der Propagierungsschicht wird jedem E/A-Manager ein *Puffermanager* zugewiesen, der mittels der LRD-V2-Verdrängungsstrategie [HR99] versucht, möglichst viele aktuell benötigte Datenseiten im Hauptspeicher zu halten, um so den teuren E/A-Zugriff auf die Containerdateien zu minimieren.

Die Zugriffsdienste bilden die komplexeste Schicht, da hier die Abbildung der logischen Datenstrukturen (XML-Dokumente bestehend aus Elementen, Attributen und Texten) auf deren physische Repräsentation in den Datenseiten erfolgt. Dazu realisiert der *Indexmanager* die Verwaltung von Listen, B- und B*-Bäumen, und der *Satzmanager* steuert die Speicherung, Rekonstruktion und Modifikation der XML-Dokumente auf diesen Strukturen. Im folgenden Abschnitt 2.2 diskutieren wir zwei alternative Implementierungen für die Speicherung von Dokumenten in dieser Schicht. Zur Verwaltung der benötigten Metadaten unseres Datenbanksystems wird der *Katalogmanager* eingesetzt.

Die satzorientierten Knotendienste werden durch den *Knotenmanager* zur Verfügung gestellt, der die Umwandlung der internen Speicherungsstruktur der Datenobjekte auf deren externe Repräsentation durchführt und XML-typische Operationen (ähnlich der DOM-Schnittstelle) ermöglicht. Zur Gewährleistung der Transaktionsisolation fordert der Knotenmanager bei der Ausführung von Datenbankoperationen entsprechende Sperren beim *Sperrmanager* an (siehe dazu Kapitel 2.4). Die proprietären Objekte und Methoden des Knotenmanagers werden durch die Schnittstellendienste auf die DOM- bzw. SAX-Schnittstelle abgebildet und mit dem Treiberpaket in die entsprechenden Objekte der Anwendungswirtssprache überführt.

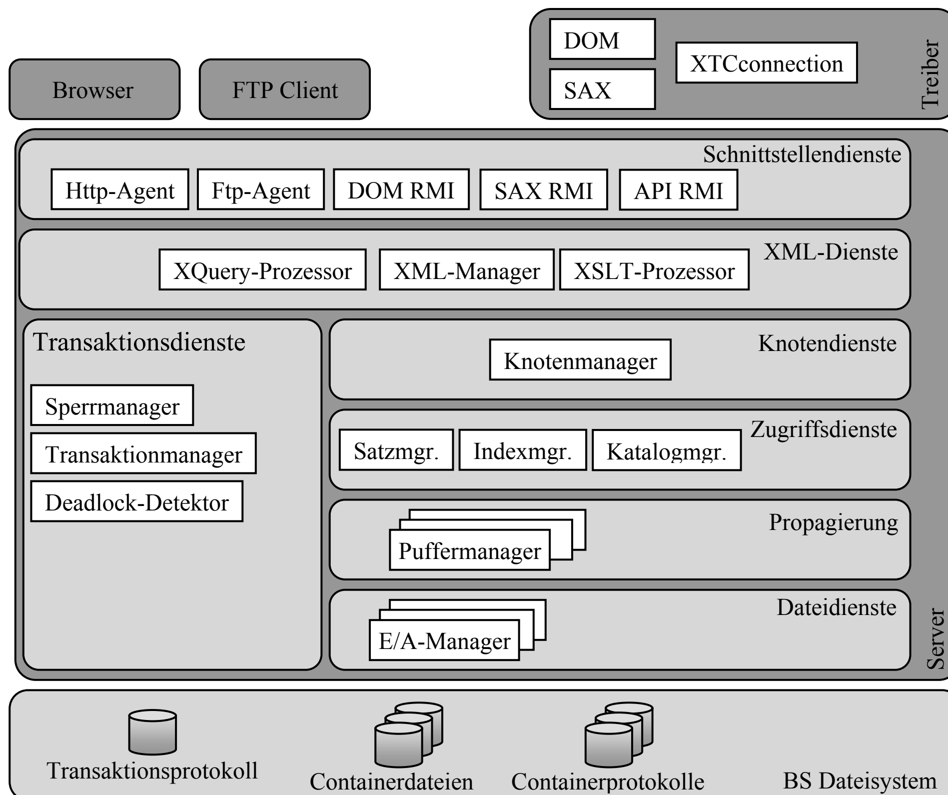


Abbildung 1: XTC-Systemarchitektur

Die mengenorientierten XML-Dienste erlauben Operationen mit der Standardanfragesprache XQuery und die Verarbeitung von XML-Dokumenten mit einem XSLT-Prozessor. Der *XML-Manager* stellt weiterhin ein virtuelles Dateisystem mit Operationen zum Anlegen, Löschen oder Umbenennen von Verzeichnissen, XML-Dokumenten und beliebigen binären Dateien (BLOBs) zur Verfügung. Mit den Schnittstellendiensten kann der Zugriff auf diese Strukturen auch über das Http- oder Ftp-Protokoll erfolgen.

2.2 Speicherungsmodell

Für unseren XDBS-Forschungsprototypen XTC betrachten wir zunächst nur XML-Dokumente, die aus Element-, Attribut- und Textknoten bestehen. Zur Speicherung eines Dokuments wird dessen textuelle Repräsentation (üblicherweise in Form einer Textdatei) zunächst in das taDOM-Datenmodell [HH03] überführt. Dabei werden Element-, Attribut- und Textknoten des XML-Dokuments auf Element-, Attribut- und Textknoten des taDOM-Datenmodells abgebildet. Zusätzlich betrachten wir alle Attributknoten als Kindknoten einer neu eingeführten *Attributwurzel*, die wiederum als Kindknoten an den die Attribute besitzenden Elementknoten angehängt wird. Die Werte von Attribut- und Textknoten werden als so genannte *String-Knoten* gespeichert. Ein Beispiel für die Überführung eines XML-Dokuments, das im Folgenden mit *sample.xml* bezeichnet wird, in dessen Darstellung im taDOM-Datenmodell ist in Abbildung 2 gegeben.

Bei der Speicherung eines XML-Dokuments wird nach einem Analysevorgang die physische Repräsentation der identifizierten XML-Knoten (Folgen einzelner Bytes) in die vom Puffermanager zur Verfügung gestellten Datenseiten geschrieben. Um beim späteren Zugriff die wechselseitige Isolation der Transaktionen auf Knotenebene zu erreichen, muss es möglich sein, jeden einzelnen Knoten direkt adressieren und sperren zu können. Dazu wird bei der Speicherung jedem Knoten eine eindeutige ID zugewiesen. Die Knotendienste in Schicht vier unseres Prototypen garantieren, dass ein Knoten seine ID (bis

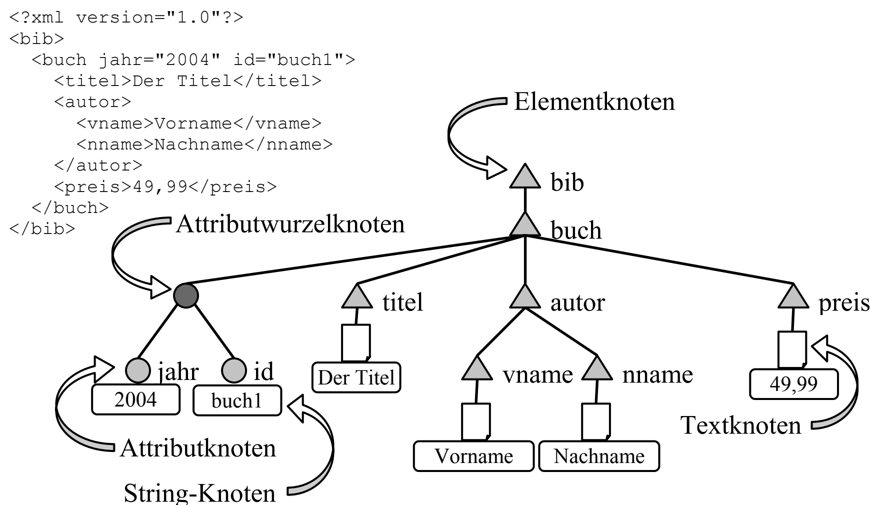


Abbildung 2: Überführung von *sample.xml* in das taDOM-Datenmodell

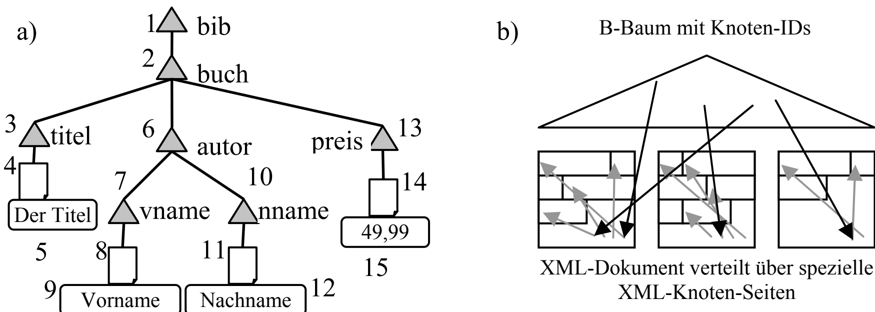


Abbildung 3: Nummerierung und Speicherung mit sequentieller Knoten-ID

auf sehr selten auftretende Neunummerierungen von Teilbäumen) während seines gesamten Lebenszyklus unabhängig von Modifikationen im Dokument behält.

Zur Vergabe einer solchen eindeutigen ID haben wir zunächst für die einzelnen Knoten sequentiell aufsteigende ganzzahlige Nummern vergeben (Abbildung 3a). Zur Speicherung werden ähnlich zu Implementierungen relationaler Datenbanksysteme die Knoten (Datensätze) des XML-Dokuments sequentiell über Datenseiten verteilt und die ID-Nummern durch eine zusätzliche Indirektion mittels eines B-Baums den aktuellen physischen Speicheradressen der Knoten zugeordnet (Abbildung 4b). Dieses Verfahren entspricht dem Database-Key-Konzept [HR99]. Da jedoch (wie in Kapitel 2.4 noch zu sehen ist) für ein Sperrverfahren auf einer baumartigen Datenstruktur stets das zu sperrende Objekt und alle Vorfahren bis zur Baumwurzel mit Sperren belegt werden müssen, hat dieser Ansatz einen entscheidenden Nachteil. Die aufsteigend vergebenen ID-Nummern lassen für eine gegebene Knoten-ID keinerlei Rückschlüsse auf den jeweiligen Elternknoten zu, da bei Einfügeoperationen an beliebigen Stellen im XML-Dokument die nächste zu vergebende Knoten-ID einfach erhöht wird. Als direkte Folge dieses Vorgehens muss nun der Sperrmanager beim Sperren eines Knotenpfads für jeden Knoten bis zur Dokumentwurzel das Speichersystem zur Ermittlung des Elternknotens aufrufen. Dies erzeugt vor allem in der Isolationsstufe *committed* (für jeden Lesezugriff auf einen Knoten wird der gesamte Pfad bis zur Wurzel gesperrt und direkt nach dem Zugriff wieder freigegeben) eine erhebliche zusätzliche Last auf dem gesamten Datenbanksystem. Auch Ansätze, die bei der systematischen Nummerierung der Knoten entsprechende Leerräume zwischen den Knoten-IDs reservieren [Me02, Ta02], sind nur bedingt geeignet, da solche Leerräume nicht alle Änderungsmuster beliebiger Transaktionstypen berücksichtigen können und somit ebenfalls teure Neunummerierungen der XML-Knoten verbunden mit entsprechenden Verwaltungsoperationen der Indexstrukturen ausgelöst werden.

Um das Sperren eines Knotenpfads effizient zu unterstützen, haben wir daher eine weitere Alternative implementiert, die jedem Knoten eine so genannte *DeweyID* zuweist. Dieser Ansatz wird auch von den kommerziellen Datenbankherstellern IBM [Ta02] und Microsoft [Ne04] zur Knotenadressierung bei der XML-Datenverarbeitung favorisiert und in [BR04] in ähnlicher Form als *Dynamic Level Numbering DLN* vorgestellt.

Unsere Implementierung adaptiert den in [Ne04] publizierten ORDPATH-Ansatz an das taDOM-Datenmodell (Abbildung 4a). Die DeweyID basiert auf der Dewey-

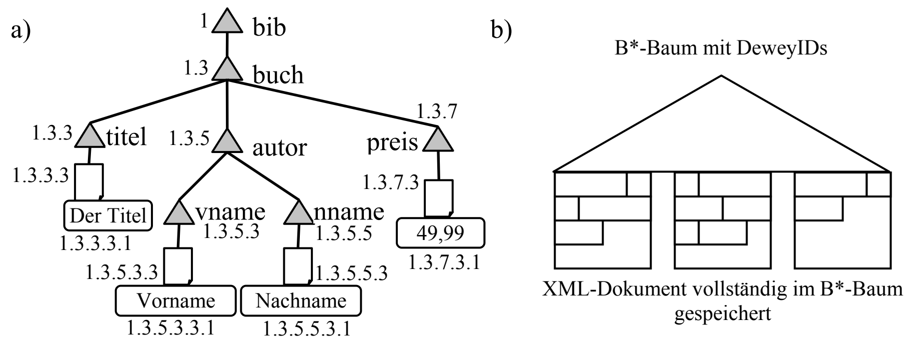


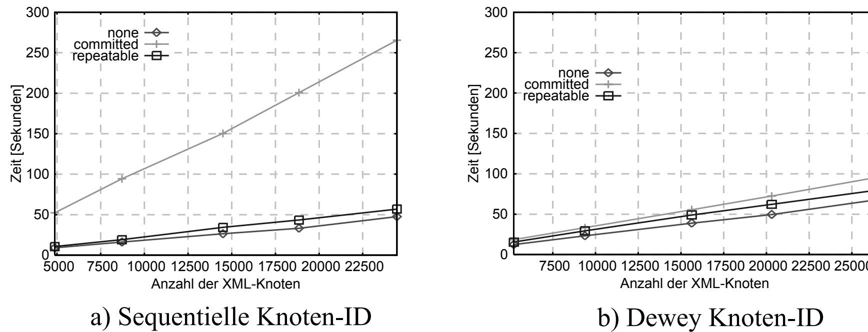
Abbildung 4: Nummerierung und Speicherung mit DeweyID

Dezimalklassifikation und nummeriert die Knoten in jeder Ebene separat mit ungeraden aufsteigenden Nummern durch. Mit Ausnahme des Wurzelelements ist die Nummer *1* für Attributwurzel- und String-Knoten reserviert. Die Nummern einer Ebene werden durch Punkte getrennt; die ID des Elternknotens wird in die ID des Kindknotens aufgenommen. Durch die Vergabe der ungeraden Nummern ist es möglich, an jeder Position beliebig viele neue Knoten einzufügen. Zwischen den DeweyIDs *1.3* und *1.5* ist dies beispielsweise mit den IDs *1.4.3*, *1.4.5*, *1.4.7,* usw. möglich. Zwischen die IDs *1.4.5* und *1.4.7* können wiederum die IDs *1.4.6.3*, *1.4.6.5*, *1.4.6.7*, usw. eingefügt werden. Für einen gegebenen Knoten erlaubt die DeweyID somit aufgrund der systematischen Kombination von ungeraden und geraden Nummern die Berechnung der Knotenebene und der IDs aller Vorfahren bis zur Dokumentwurzel ohne Zugriff auf das Speichersystem.

Bei der Speicherung wird für ein XML-Dokument eine Katalogseite mit Metadaten angelegt, deren Seitennummer für eine eindeutige Identifikation des Dokuments benutzt wird. Diese Seitennummer wird bei der Adressierung eines XML-Knotens seiner DeweyID mit einem Doppelpunkt getrennt vorangestellt. Gehört das Fragment in Abbildung 4 beispielsweise zum Dokument *4711*, so wird das Element *titel* mit der DeweyID *4711:1.3.3* systemweit eindeutig identifiziert.

Ein weiterer Vorteil der DeweyID-Adressierung ergibt sich für die Speicherung eines XML-Dokuments, da sich die ID für einen neu einzufügenden Knoten stets in die sequentielle Ordnung der bereits vorhandenen Nachbarknoten einfügt. Somit wird nur ein einziger B*-Baum zur Speicherung des gesamten XML-Dokuments benötigt (Abbildung 4b). Ein Eintrag im B*-Baum wird aus der Byte-Repräsentation der DeweyID als Schlüsselanteil und der Byte-Repräsentation des eigentlichen Knotenwerts als Wertanteil gebildet. Da die Speicherung der einzelnen XML-Knoten in den Datenseiten auch deren sequentieller Reihenfolge im XML-Dokument entspricht, und die DeweyID jedes Knotens auch die ID des Elternknotens als Präfix enthält, lässt eine Präfix-Suffix-Komprimierung [HR99] der B*-Baum-Schlüssel weiteres Optimierungspotential erwarten. Dies ist zurzeit jedoch noch nicht implementiert. Der Speicherplatzverbrauch des Wertanteils wird für Element- und Attributknoten durch die Verwendung eines Vokabulars verringert. Für diese Knoten werden nicht deren in XML-Dokumenten sehr häufig auftretende Namen gespeichert, sondern Schlüsselwerte, die die eigentlichen Namenswerte in einem Vokabular adressieren.

Abbildung 5: Rekonstruktion eines XML-Dokuments



a) Sequentielle Knoten-ID b) Dewey Knoten-ID

Um die Vorteile der DeweyID gegenüber der sequentiellen Knoten-ID in einer konkreten Systemumgebung messen zu können, haben wir verschiedene XML-Dokumente mit bis zu 25.000 einzelnen Knoten mit dem XMark-Benchmarktool *xmlgen* erzeugt [SWK02] und mit den jeweiligen Implementierungen der Zugriffsdienste in unserem XDBS gespeichert. Anschließend werden die Dokumente mit verschiedenen Isolationsstufen jeweils zweimal innerhalb einer einzigen Transaktion über einzelne DOM-Schnittstellenaufrufe rekonstruiert.

Durch die zweifache Rekonstruktion wird der Unterschied zwischen den Isolationsstufen *committed* (Sperranforderung und -freigabe vor und nach jedem einzelnen Knotenzugriff) und *repeatable* (Sperranforderung vor jedem Zugriff und Sperrfreigabe erst am Transaktionsende) in Abbildung 5 deutlich hervorgehoben. Eine Rekonstruktion mit der Isolationsstufe *repeatable* benötigt typischerweise mehr Zeit als ohne Isolation, bei der keinerlei Sperranforderungen durchgeführt werden. Da in der Isolationsstufe *repeatable* die Sperren bis zum Ende der Transaktion gehalten werden, ist beim zweiten Durchlauf der Dokumentrekonstruktion keine Sperre mehr anzufordern, da für jeden Knoten bereits beim ersten Durchlauf eine Sperre eingerichtet wurde. Somit entsteht weniger Rechenlast auf dem System als in der Isolationsstufe *committed*. Gemäß dem Sperrprotokoll muss in dieser Stufe für jeden Knotenzugriff eine Sperre angefordert und sofort nach dem Zugriff wieder freigegeben werden. Bei der Implementierung der sequentiellen Knoten-ID (Abbildung 5a) erfordert die Ermittlung und Sperrung eines Knotenpfads (wie oben beschrieben) weitere Aufrufe des Speichersystems, da für einen gegebenen Knoten die ID des Elternknotens nicht ohne Zugriff auf das gespeicherte Dokument berechnet werden kann. Dies treibt die Zeit zur Rekonstruktion in nicht vertretbarem Maße in die Höhe (über 500% der Zeit ohne Transaktionsisolation). Im Gegensatz dazu benötigt die DeweyID-Implementierung (Abbildung 5b) ausschließlich Rechenoperationen mit DeweyIDs und Hauptspeicherzugriffe auf die Datenstrukturen des Sperrmanagers, sodass der Zusatzaufwand für die Sperrverwaltung zur Transaktionsisolation bei einer Rekonstruktion von 25.000 Knoten selbst mit der Isolationsstufe *committed* nur etwa 40% beträgt[1].

[1] Die etwas höhere Rekonstruktionszeit bei Verwendung der DeweyID gegenüber der sequentiellen Knoten-ID ohne Sperrverwaltung ist durch die B*-Baum-Implementierung begründet, die im Fall der DeweyID variabel lange Schlüssel- und Wertefelder unterstützen muss.

2.3 Logging und Recovery

Zu einer Transaktionsumgebung gehört neben der Isolation auch ein Logging- und Recovery-Mechanismus, der die Wiederholbarkeit und das Zurücksetzen von Transaktionsoperationen sowohl im Fehlerfall als auch im Normalbetrieb gewährleistet.

Auf unterster Ebene ist in jeder Containerdatei ein Block für ein so genanntes *Before-Image* reserviert. Muss der E/A-Manager einen Block aus dem Hauptspeicher zurück in die Containerdatei schreiben, so wird zunächst ein Abbild des zu überschreibenden Blocks in den Before-Image-Block kopiert, bevor der eigentliche Schreibvorgang durchgeführt wird. Kommt es nun während dem Erstellen des Before-Image zu einem Systemabsturz, so ist die Containerdatei nach wie vor konsistent, da der zu überschreibende Block noch nicht modifiziert wurde. Stürzt das System beim Schreiben des Blocks ab, so wird beim Wiederanlauf der ursprüngliche Block aus dem Before-Image geladen. Der E/A-Manager kann die Containerdatei also stets in einen blockkonsistenten Zustand überführen.

Auf der Ebene der Propagierungsschicht werden einzelne DML-Transaktionsoperationen in so genannte *Elementaroperationen* zerlegt (Einfügen, Ersetzen oder Löschen von Byte-Folgen innerhalb einer Datenseite), die mit Hilfe des Konzepts der *Log Sequence Numbers* [HR99] von jedem Puffermanager in einer Log-Datei protokolliert werden. Da nach einem Systemabsturz wie oben beschrieben der E/A-Manager eine blockkonsistente Containerdatei erstellt, kann der Puffermanager darauf basierend aufgrund der Log Sequence Number für jede der protokollierten Elementaroperationen entscheiden, ob diese zu wiederholen ist. Somit rekonstruiert der Puffermanager beim Wiederanlauf einen DML-operationskonsistenten Datenbankzustand.

Nach einem Systemabsturz müssen die Änderungen aller nicht beendeten Transaktionen (so genannte *Verlierer*) zurückgesetzt werden. Der Transaktionsmanager protokolliert in einer weiteren Log-Datei Transaktionsanfang und -ende und die Umkehroperationen für alle DML-Transaktionsoperationen. Beim Einfügen eines Datensatzes wird sein Löschen protokolliert, beim Aktualisieren auf einen neuen Wert sein Aktualisieren auf den alten Wert, beim Löschen eines Datensatzes das Einfügen des ursprünglichen Datensatzes usw.. Da der Puffermanager nach dem Wiederanlauf einen DML-operationskonsistenten Datenbankzustand garantiert, kann der Transaktionsmanager während der Recovery-Phase die protokollierten Umkehroperationen der Verlierertransaktionen direkt ausführen. Somit wird die gesamte Datenbank nach einem Systemabsturz wieder in einen transaktionskonsistenten Zustand überführt.

Das Zurücksetzen von Transaktionen während des Datenbankbetriebs (unabhängig davon, ob eine Transaktion explizit oder aufgrund eines Deadlock zurückgesetzt werden muss) erfolgt nach dem gleichen Prinzip. Alle Sperranforderungen zur Modifikation von Datenobjekten eines XML-Dokuments werden an den Sperrmanager gestellt. Erst wenn dieser die entsprechenden Sperren gewährt hat, werden die benötigten Datenseiten exklusiv angefordert, die Änderungen durchgeführt und die Datenseiten sofort nach Operationsende wieder freigegeben. Eine weitere Sperranforderung findet in dieser Phase nicht statt. Deadlocks treten daher nur bei Sperranforderungen an den Sperrmanager auf, bevor die gewünschte Modifikation durchgeführt wird. Bei Anforderungen nach Datenseiten kommt es nur zu Wartebeziehungen. Da alle Sperren auf den modifizierten Datenob-

jekten stets bis zum Transaktionsende gehalten werden (siehe Abschnitte 2.4 und 3.1) und somit keine weiteren Sperren für die Umkehroperationen angefordert werden müssen, können sämtliche Änderungen einer Transaktion durch Anwendung der protokollierten Umkehroperationen rückgängig gemacht werden.

2.4 Transaktionsisolation

Zur Isolation nebenläufiger Transaktionen setzen wir ein pessimistisches Protokoll ein, das Sperren bis auf die Ebene einzelner XML-Knoten vergibt. Um die bei der Verwendung navigationsorientierter Schnittstellen traversierten Pfade bei der Synchronisation zu berücksichtigen, werden zusätzlich Sperren auf so genannten *virtuellen Navigationskanten* verwaltet. Da somit ebenfalls die „Knotenzwischenräume" gesperrt werden, können Phantome nur beim Einsprung in ein Dokument über Sekundärindexe auftreten. Eine detaillierte Beschreibung mit Beispielen für Sperranforderungen und -ersetzungen bei variierenden Sperrtiefen kann in [HH04] nachgelesen werden. Durch die feingranulare Synchronisation wird ein sehr hoher Grad an Nebenläufigkeit für Transaktionen erreicht. Zum besseren Verständnis der Sperranforderungen auf XML-Knoten verschiedener Dokumente während der Anfrageverarbeitung (Kapitel 3) werden die in diesem Beitrag relevanten Sperren NR, IX, LR, CX und SX im Folgenden kurz erläutert. Die *Kompatibilitätsmatrix* für das Sperrprotokoll ist in Tabelle 1 gegeben. Um die später diskutierten Sperrsituationen übersichtlicher zu gestalten, verzichten wir in diesem Zusammenhang auf die Darstellung und Erläuterung der Navigationssperren.

	-	NR	IX	LR	CX	SX
NR	+	+	+	+	+	-
IX	+	+	+	+	+	-
LR	+	+	+	+	-	-
CX	+	+	+	-	+	-
SX	+	-	-	-	-	-

Tabelle 1: Kompatibilitätsmatrix

Für den Lesezugriff auf ein XML-Dokument werden die Sperren NR (*node read*) und LR (*level read*) zur Verfügung gestellt. Beim Zugriff auf einen beliebigen Knoten im Dokument (*Kontextknoten*) wird dieser Knoten mit einer NR-Sperre versehen und zusätzlich alle Knoten auf dem Pfad bis zur Dokumentwurzel ebenfalls mit einer NR-Sperre belegt. Um nicht alle Knoten in einer Ebene separat mit NR sperren zu müssen, kann auf einem Kontextknoten eine LR-Sperre angefordert werden, die den Kontextknoten selbst und alle seine Kinder lesend sperrt. Analog zur NR-Sperre werden bei einer LR-Sperre alle Vorgängerknoten des Kontextknotens bis zur Dokumentwurzel mit einer NR-Sperre versehen. Ein Beispiel zur Anwendung der Lesesperren ist in Abbildung 6 gegeben, bei der eine Transaktion T_1 den Buchtitel liest (NR-Sperre auf *titel*) und eine weitere Transaktion T_2 alle Kinder des Buchknotens ermittelt (LR-Sperre auf *buch*).

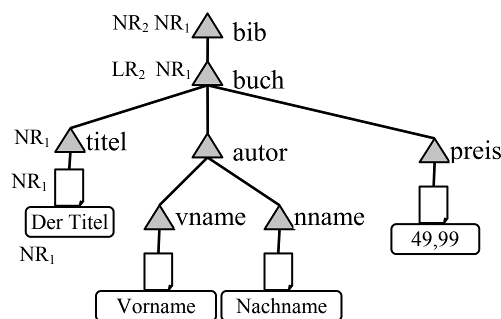


Abbildung 6: Lesesperren

Zur Modifikation eines Knotens (und des gesamten XML-Fragments, das durch den Knoten als Wurzel bestimmt wird) werden die Schreibsperren IX (*intention exclusive*), CX (*child exclusive*) und SX (*subtree exclusive*) benutzt. Dazu wird auf dem zu ändernden Knoten eine SX-Sperre, auf dessen Elternknoten eine CX-Sperre und auf allen weiteren Vorgängerknoten bis zur Dokumentwurzel eine IX-Sperre angefordert. Da die CX-Sperre nicht mit der LR-Sperre kompatibel ist, wird somit verhindert, dass ein Knoten in einer Ebene verändert werden kann, die bereits vollständig durch eine LR-Sperre auf dem Elternknoten zum Lesen gesperrt ist. Die Anwendung der Schreibsperren wird beispielhaft in Abbildung 7 gezeigt. Während Transaktion T_1 den Buchtitel liest (NR-Sperre auf dem String-Knoten *Der Titel*), kann Transaktion T_2 den Autor löschen (SX-Sperre auf *autor*). Dadurch wird Transaktion T_3, die alle Kinder des Buchknotens bestimmen möchte (LR-Sperre auf *buch*), in einen Wartezustand gesetzt, da die angeforderte LR-Sperre auf dem Buchknoten mit der vorhandenen CX-Sperre inkompatibel ist.

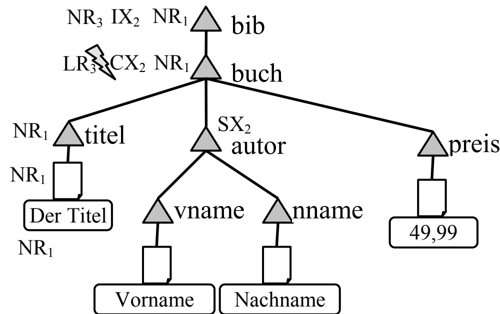


Abbildung 7: Schreibsperren

Unser Sperrprotokoll sieht vor, dass eine Transaktion auf einem XML-Knoten nur maximal eine Sperre hält. Da für verschiedene innerhalb einer Transaktion aufeinander folgende Operationen jeweils eine geeignete Sperre anzufordern ist, kommt es sehr häufig vor, dass eine Sperranforderung für einen Knoten vorliegt, auf dem die betreffende Transaktion bereits eine Sperre hält. In diesem Fall wird aus der vorliegenden und der angeforderten Sperre eine resultierende Sperre ermittelt, mit der die Transaktion bezüglich ihrer durchgeführten Operationen ausreichend gegen parallel laufende Transaktionen isoliert ist. Dazu wird die in Tabelle 2 gezeigte *Sperrkonversionsmatrix* benutzt. Diese gibt für jede bereits gehaltene Sperre (Kopfzeile) und eine beliebige angeforderte Sperre (Kopfspalte) die resultierende Sperre an. Die Bedeutung der indizierten Sperren in der Matrix erläutert das folgende Beispiel. Angenommen, eine Transaktion hält auf einem Kontextknoten eine IX-Sperre, weil ein Nachfahre mit SX für eine Modifikation gesperrt ist. Möchte diese Transaktion nun in einem weiteren Schritt alle Kinder des Kontextknotens bestimmen, so muss sie auf dem Knoten eine LR-Sperre anfordern. Die Sperrkonversionsmatrix liefert für diesen Fall die resultierende Sperre IX_{NR}, die besagt, dass die IX-Sperre auf dem Kontextknoten durch eine IX-Sperre ersetzt (also beibehalten) und auf allen Kindknoten des Kontextknotens eine NR-Sperre anzufordern ist. Somit gewährleistet die entstandene Sperrsituation ausreichende Isolation für beide Operationen der Transaktion.

	-	**NR**	**IX**	**LR**	**CX**	**SX**
NR	NR	NR	IX	LR	CX	SX
IX	IX	IX	IX	IX_{NR}	CX	SX
LR	LR	LR	IX_{NR}	LR	CX_{NR}	SX
CX	CX	CX	CX	CX_{NR}	CX	SX
SX	SX	SX	SX	SX	SX	SX

Tabelle 2: Sperrkonversionsmatrix

3 Programmierschnittstelle und Anfrageverarbeitung

Zum Zugriff auf XML-Daten haben sich drei Schnittstellentypen zur ereignisbasierten, navigationsorientierten und deskriptiven Bearbeitung etabliert. Mit der Programmierschnittstelle unseres nativen XML-Datenbanksystems ermöglichen wir Anwendungen, diese drei Schnittstellen gleichzeitig in einer transaktionsgeschützten Umgebung auf einem gemeinsamen XML-Dokument zu nutzen. In diesem Kapitel erläutern wir in Abschnitt 3.1 zunächst kurz die grundlegenden Funktionen des Treiberpakets zum Verbindungsaufbau und zur Datenorganisation. Die Abschnitte 3.2 bis 3.4 erläutern detailliert den Datenzugriff mittels der SAX-, DOM- und XQuery-Schnittstelle und deren kombinierte Anwendung. Abschnitt 3.5 geht schließlich noch auf einige Besonderheiten im Zusammenhang mit aktualisierbaren XQuery-Anfrageergebnissen ein. Während der gesamten Datenverarbeitung verhindert das in Abschnitt 2.4 vorgestellte Sperrprotokoll das Auftreten von Änderungsanomalien, sofern man entsprechende Isolationsstufen für die laufenden Transaktionen gewählt hat.

3.1 Verbindungsaufbau und Datenorganisation

Der Verbindungsaufbau zum XTC-Server erfolgt mit Hilfe des XTC-Treibers (siehe Abbildung 1), der dem Anwendungsprogramm nach einem entsprechenden Methodenaufruf mit Benutzername, Passwort, Rechnername und Port des Datenbankservers ein *Verbindungsobjekt* zurückliefert, über das die gesamte Datenverarbeitung und Transaktionssteuerung abgewickelt wird.

Das Verbindungsobjekt erlaubt das explizite Starten, Beenden und Zurücksetzen von Transaktionen mit den Methoden *beginWork*, *commitWork* und *rollbackWork*. Wird eine Transaktion nicht explizit gestartet und eine Verarbeitungsanweisung an das Datenbanksystem geschickt, so wird für die Anweisung implizit eine Transaktion gestartet und nach der Verarbeitung der Anweisung beendet (*single statement commit*). Mit der Methode *setIsolation* des Verbindungsobjekts wird eine der Isolationsstufen *uncommitted*, *committed*, *repeatable* oder *serializable* gewählt. Diese eingestellte Stufe gilt für jede nachfolgend gestartete Transaktion und bestimmt gemäß der klassischen Datenbankliteratur beim Anfordern einer Lese- oder Schreibsperre die Dauer, mit der diese Sperre nach der durchgeführten Operation gehalten wird (Tabelle 3). Eine lange Sperre wird bis zum Ende der laufenden Transaktion gehalten, eine kurze Sperre wird sofort nach dem Ende der Operation freigegeben, für die sie angefordert wurde. Die Isolationsstufe *serializable* verhindert gegenüber *repeatable* zusätzlich das Auftreten von Phantomen.

Isolationsstufe	**Schreib-sperre**	**Lese-sperre**
Uncommitted	lang	keine
Committed	lang	kurz
Repeatable	lang	lang
Serializable	lang	lang

Tabelle 3: Isolationsstufen

Nach erfolgreichem Verbindungsaufbau eröffnet der XTC-Server der Anwendung über das Verbindungsobjekt die Sicht auf ein virtuelles Dateisystem, welches das Anlegen, Löschen und Umbenennen von XML-Dokumenten, binären Objekten (z. B. Bilddateien) und Verzeichnisstrukturen erlaubt. Der Zugriff auf ein im XDBS gespeichertes XML-

Dokument erfolgt über das Verbindungsobjekt mit der Methode *getDocument,* für die zunächst eine neue Transaktion gestartet werden muss. Diese Methode erhält als Parameter den Namen, unter dem das XML-Dokument im virtuellen Dateisystem des XDBS abgelegt ist (eventuell mit entsprechender Pfadangabe). Die zentrale Idee zur Kombination der drei Schnittstellen SAX, DOM und XQuery innerhalb des Kontexts der gestarteten Transaktion besteht darin, dass die Methode *getDocument* eine Referenz auf das Dokument als standardkonformes Objekt einer Klasse, die die Schnittstelle *org.w3c.dom.Document* implementiert (im Folgenden als *W3C-Dokumentreferenz* bezeichnet) gemäß der DOM-Spezifikation des W3C [Ho04] zurückliefert. Alle weiteren Methoden des Verbindungsobjekts sind zugeschnitten auf diese Klasse implementiert. Die Methode *saxParse* (Abschnitt 3.2), eine DOM-Implementierung (Abschnitt 3.3) und die Methode *executeXQuery* (Abschnitt 3.4) können somit gleichzeitig auf der W3C-Dokumentreferenz ausgeführt werden.

Ist das Ergebnis einer XQuery-Anfrage (Methode *executeXQuery*) ein wohlgeformtes XML-Dokument, so wird dieses Ergebnisdokument dem Anwendungsprogramm ebenfalls als W3C-Dokumentreferenz zurückgeliefert, sodass dieses Dokument wiederum einem SAX Parser übergeben, mittels der DOM-Schnittstelle modifiziert oder einer erneuten XQuery-Anfrage unterzogen werden kann. Handelt es sich dagegen beim Ergebnis der Anfrage nicht um ein XML-Dokument, so kann das Ergebnis im Anwendungsprogramm als einfache Zeichenkette weiterverarbeitet werden.

3.2 SAX-Zugriff

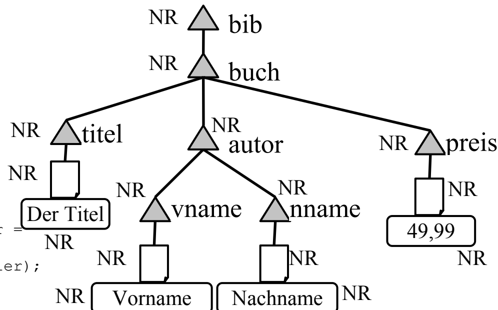
Mit der Methode *saxParse* des Verbindungsobjekts wird das ereignisbasierte Parsen eines XML-Dokuments durchgeführt. Dazu erfolgt zunächst die Instanzierung einer Content-Handler-Klasse, die mit der anwendungsspezifischen Implementierung der entsprechenden Callback-Methoden auf die eintretenden Ereignisse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

Bei der Zusammenstellung der Knotenpakete im Datenbanksystem durch den Knotenmanager werden dabei die einzelnen Knoten des XML-Dokuments mit einer NR-Sperre für den Lesezugriff geschützt. Dies hat den Vorteil, dass bei anwendungsseitigem Abbruch des Parse-Vorgangs (beispielsweise wurde die benötigte Information schon vor Erreichen des Dokumentenendes verarbeitet) nicht zwangsläufig das gesamte Dokument gesperrt werden muss.

Abbildung 8 zeigt ein Beispiel mit einem Java Programmfragment für den SAX-Zugriff und der daraus resultierenden Sperrsituation für das XML-Dokument. Zunächst wird in den Zeilen 1 und 2 eine Verbindung zum Datenbanksystem aufgebaut und eine Transaktion gestartet. Mit dem Verbindungsobjekt *connection* wird in Zeile 3 das gespeicherte XML-Dokument *sample.xml* geöffnet und die W3C-Dokumentreferenz auf dieses Dokument in der Variable *document* vermerkt. Danach wird in Zeile 4 und 5 die Instanz

Abbildung 8: SAX-Zugriff

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



einer selbst implementierten Content-Handler-Klasse erzeugt und der SAX-Zugriff mit der Methode *saxParse* gestartet. In Zeile 6 und 7 wird schließlich die Transaktion beendet und die Datenbankverbindung geschlossen. Vor dem Ende der Transaktion liegt im XDBS die in Abbildung 8 rechts dargestellte Sperrsituation vor. Alle Knoten des Beispieldokuments sind aufgrund des Lesezugriffs mit einer NR-Sperre geschützt. Diese besonders für größere Dokumente hohe Anzahl einzelner Knotensperren kann mit dem Konzept der Sperreskalation [HH03] reduziert werden.

3.3 DOM-Zugriff

Der Zugriff auf ein im XDBS gespeichertes Dokument mit der Methode *getDocument* liefert wie beschrieben stets eine W3C-Dokumentreferenz. Somit kann nach der Instanzierung eines solchen Referenzobjekts in der Anwendung direkt mittels der DOM-Implementierung des Treibers auf dem Dokument mit der Navigation und Modifikation begonnen werden. Während der Navigation werden die angeforderten Knoten vom XDBS zum Treiber übertragen und dort als DOM-Knoten für die Anwendung instanziert. Mit der DOM-API durchgeführte Modifikationen werden sofort auf die Datenstrukturen im XDBS unter Anforderung der benötigten Knotensperren propagiert.

Abbildung 9 zeigt mit einem Java-Programmfragment ein Beispiel für einen DOM-Zugriff und die dazu verwalteten Sperren auf dem XML-Dokument. In den Zeilen 1 und 2 wird eine Verbindung zum XDBS aufgebaut und eine Transaktion gestartet. Nach der Erzeugung der W3C-Dokumentreferenz auf *sample.xml* in Zeile 3 ermittelt die Anwendung zunächst in Zeile 4 den Wurzelknoten *bib* und navigiert daraufhin in den Zeilen 5 bis 7 über die Knoten *buch* und *preis* zum Textknoten, der den Preis *49,99* enthält. Der Wert des Textknotens wird in Zeile 8 ausgelesen. In den Zeilen 9 und 10 wird die Transaktion beendet und die Datenbankverbindung abgebaut.

```
 1 XTCconnection connection = XTCdriver.getConnection (host,port,username,password);
 2 connection.beginWork();
 3 org.w3c.dom.Document document =
     connection.getDocument("sample.xml");
 4 org.w3c.dom.Node bibNode = document.getDocumentElement();
 5 org.w3c.dom.Node buchNode = bibNode.getFirstChild();
 6 org.w3c.dom.Node preisNode = buchNode.getLastChild();
 7 org.w3c.dom.Node preisText = preisNode.getFirstChild();
 8 String preisTextValue = preisText.getNodeValue();
 9 connection.commitWork();
10 connection.close();
```

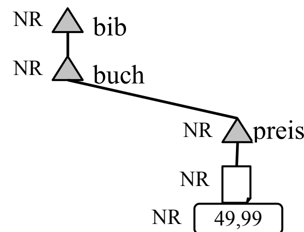


Abbildung 9: DOM-Zugriff

3.4 XQuery-Anfrageverarbeitung

Zur Ausführung einer XQuery-Anfrage im Anwendungsprogramm stellt das Verbindungsobjekt die Methode *executeXQuery* in zwei Varianten mit unterschiedlichen Signaturen zur Verfügung. Beide Varianten erwarten als Parameter eine Zeichenkette mit der eigentlichen XQuery-Anfrage und einen booleschen Wert, mit dem das Ergebnis der Anfrage als aktualisierbar spezifiziert werden kann. Für die zweite Variante wird zusätzlich eine W3C-Dokumentreferenz angegeben, auf der die Anfrage ausgeführt wird. Beide Varianten werden im Folgenden anhand von Beispielanfragen detailliert erläutert.

Abbildung 10 zeigt ein Java-Programmfragment, mit dem eine XQuery-Anfrage auf dem XML-Dokument *sample.xml* ausgeführt wird (im Folgenden als *Basisdokument* bezeichnet), mit der zunächst der Autor bestimmt wird (Zeile 3) und deren Ergebnis als aktualisierbar in ein *result*-Tag geschachtelt wird. Mit den darauf folgenden DOM-Operationen in den Zeilen 5 bis 9 wird zunächst der Nachname des zurückgelieferten Autors ermittelt und anschließend verändert. In Zeile 10 wird die Transaktion beendet und somit alle belegten Ressourcen freigegeben. Erst zu diesem Zeitpunkt ist die Bearbeitung der XQuery-Anfrage abgeschlossen. Das Ergebnis wird der Anwendung als *XTCxqueryResult*-Objekt übergeben. Falls es sich beim Ergebnis der XQuery-Anfrage um ein wohlgeformtes XML-Dokument handelt, so kann eine W3C-Dokumentreferenz auf dieses Dokument gebildet werden (Zeile 4), ansonsten kann über das *XTCxqueryResult*-Objekt das Anfrageergebnis als einfache Zeichenkette ausgelesen werden.

Die Verarbeitung einer XQuery-Anfrage im XTC-Server [Ma04] erfolgt in mehreren Stufen gemäß der *XQuery Formal Semantics* [Dr04]. Zunächst wird die erhaltene Anfrage im Server einer *syntaktischen Analyse* unterzogen, bei der die Anfrage in eine Interndarstellung umgewandelt wird. Die in diesem ersten Schritt erkannten Syntaxfehler führen sofort zu einem Abbruch der weiteren Verarbeitung. Da eine XQuery-Anfrage eine Vielzahl syntaktischer Konstrukte enthalten kann, erfolgt im nächsten Schritt die *Normalisierung* der Anfrage auf die so genannte *XQuery Core Language* [Fe04], die nur eine Teilmenge der ursprünglichen Ausdrücke und Operatoren enthält, aber dadurch mit geringerem Implementierungsaufwand bedeutungsgleich ausgewertet werden kann. Der nächste Schritt ist die *statische Analyse*, bei der die Typen verwendeter Ausdrücke und Operatoren auf Konsistenz überprüft werden. Wurde kein Typfehler entdeckt, so kann mit der eigentlichen Auswertung des Ausdrucks, der *dynamischen Evaluierung*, begonnen werden.

Da die Beispielanfrage in Abbildung 10 mit aktualisierbarem Ergebnis ausgeführt wurde (Parameter *true* in Zeile 3), wird die Änderung des Nachnamens (Zeile 9) direkt auf dem

```
 1 XTCconnection connection = XTCdriver.getConnection (host,port,username,password);
 2 connection.beginWork();
 3 XTCxqueryResult result = connection.executeXQuery("<result>
                                                      {doc('sample.xml')/bib/buch/autor}
                                                    </result>",true);
 4 org.w3c.dom.Document resultDocument = result.getDocumentResult();
 5 org.w3c.dom.Node resultNode = resultDocument.getDocumentElement();
 6 org.w3c.dom.Node autorNode = resultNode.getFirstChild();
 7 org.w3c.dom.Node nnameNode = autorNode.getLastChild();
 8 org.w3c.dom.Node nnameText = nnameNode.getFirstChild();
 9 nnameText.setNodeValue("Neuer Nachname");
10 connection.commitWork();
```

Abbildung 10: XQuery-Zugriff

Basisdokument durchgeführt. Um dies zu erreichen, werden bei der dynamischen Evaluierung die Ergebnisdokumente von XQuery-Anfragen im XDBS in einer temporären Containerdatei materialisiert. Bei einem aktualisierbaren Ergebnis werden die XML-Knoten, die nicht explizit in der XQuery-Anfrage vorgegeben werden (wie z. B. *result* in Abbildung 10), sondern während der Anfrageauswertung aus dem Basisdokument ermittelt werden, über spezielle *Verweisknoten* gespeichert, die die entsprechenden XML-Knoten im Basisdokument referenzieren. Abbildung 11 zeigt das materialisierte Anfrageergebnis und die vorliegende Sperrsituation vor der Änderung des Nachnamens. Da unser XDBS zurzeit noch keinen Indexzugriff nutzt, beginnt die Evaluierung der XQuery-Anfrage */bib/buch/autor* systematisch an der Dokumentwurzel, wo zunächst */bib* mit dem Wurzelelement verglichen wird. Aufgrund der Übereinstimmung werden alle Kinder der Wurzel ermittelt (LR-Sperre auf *bib*) und mit dem nächsten Schritt */buch* der Anfrage verglichen. Es qualifiziert sich der Buchknoten, für den wiederum die Auswertung mit */autor* mit Bestimmung aller Kindknoten und einer entsprechenden LR-Sperre durchgeführt wird. Somit ist der Autorknoten als Ergebnis der Anfrage ermittelt und es wird, da ein aktualisierbares Ergebnis gewünscht ist, ein Verweisknoten als Kind des neu angelegten *result*-Knotens in das Ergebnisdokument aufgenommen. Für ein nicht-aktualisierbares Ergebnis wird hier eine Kopie des gesamten *autor*-Fragments im Ergebnisdokument materialisiert; eine Änderung des Nachnamens wird in diesem Fall nur im Ergebnis vorgenommen und nicht auf das Basisdokument propagiert. Eine SX-Sperre auf der Wurzel des erzeugten Ergebnisdokuments sperrt alle Knoten exklusiv. Der referenzierte Autorknoten im Basisdokument ist vor Änderungen des Nachnamens implizit durch die LR-Sperre auf dem Buchknoten gesperrt, da auf ihn zur Anfrageauswertung nur lesend zugegriffen wurde.

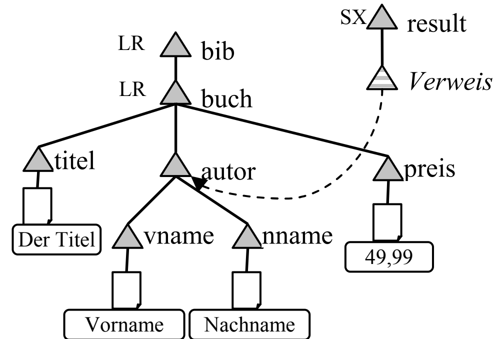


Abbildung 11: Materialisiertes XQuery-Anfrageergebnis vor der Modifikation

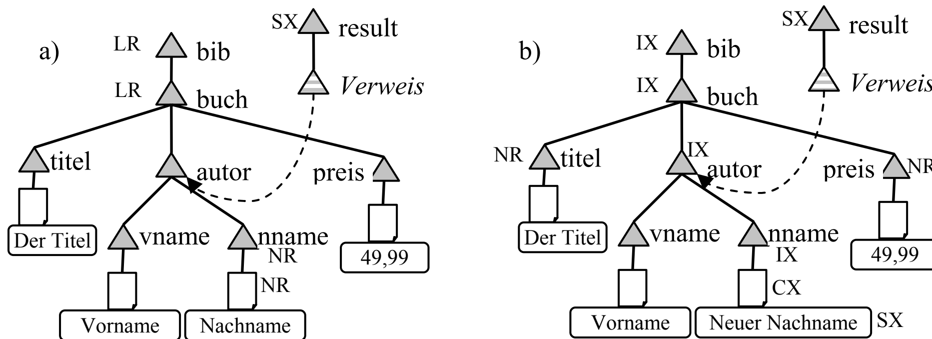


Abbildung 12: Materialisiertes XQuery-Anfrageergebnis während der Modifikation

```
...
XTCxqueryResult result = connection.executeXQuery("<result>
                                                    {doc('sample.xml')/bib/buch/autor}
                                                  </result>",true);
...
XTCxqueryResult result2 = connection.executeXQuery(result,
                                                  "<result2>
                                                     {doc('.')/result/autor/nname/text()}
                                                   </result2>", true);
...
```

Abbildung 13: Erneute XQuery-Anfrage auf dem Ergebnis einer XQuery-Anfrage

Zur Modifikation des Nachnamens findet gemäß des Programmfragments in Abbildung 10 zunächst eine Navigation vom *result*-Knoten des Ergebnisdokuments zu dessen erstem Kindknoten (dem Verweisknoten) statt. Die nachfolgende Ermittlung des letzten Kindknotens des Verweisknotens hat nun zur Folge, dass zuerst der Referenz zum Autorknoten im Basisdokument gefolgt wird und schließlich dessen letztes Kind *nname* als letztes Kind des Verweisknotens zurückgeliefert wird. Dazu muss auch die Anforderung einer NR-Sperre auf dem *nname*-Knoten erfolgen. Das Bestimmen des Textknotens, der den eigentlichen Nachnamen enthält, und die Belegung mit der erforderlichen NR-Sperre für den Lesezugriff erfolgen direkt auf dem Basisdokument (siehe Abbildung 12a). Zur Aktualisierung des Nachnamens wird zunächst eine SX-Sperre auf dem zu modifizierenden String-Knoten angefordert. Dies hat zur Folge, dass gemäß des Sperrprotokolls auf dem Textknoten eine CX-Sperre und auf allen Vorgängerknoten bis zur Wurzel eine IX-Sperre angefordert wird (Abbildung 12b). Da jedoch auf dem Textknoten und dem Knoten *nname* bereits eine NR-Sperre und auf den Knoten *bib* und *buch* LR-Sperren vorliegen, muss auf diesen Knoten eine Sperrersetzung gemäß der Sperrkonversionsmatrix in Abschnitt 2.3 vorgenommen werden, die zusätzliche NR-Sperren auf *titel* und *preis* einrichtet.

Wie im Beispiel der Abbildung 10 deutlich wird, ermöglicht das Ergebnis einer XQuery-Anfrage als W3C-Dokumentreferenz, dass das Ergebnisdokument innerhalb einer laufenden Transaktion sofort mit einem SAX Parser analysiert, mit der DOM-Schnittstelle modifiziert oder mit einer weiteren XQuery-Anfrage durchsucht werden kann. Da ein Ergebnisdokument jedoch keinen expliziten Namen erhält, sondern der Anwendung innerhalb einer Transaktion nur als Referenz bekannt ist, wird im Treiberpaket die Methode *executeXQuery* in einer zweiten Variante angeboten, bei der als weiterer Parameter eine W3C-Dokumentreferenz (z. B. das Ergebnis einer zuvor ausgeführten XQuery-Anfrage) übergeben wird. Auf das so übergebene Dokument kann in einer XQuery-Anfrage mit dem Konstrukt *doc(".")* zugegriffen werden. Möchte man beispielsweise im Anwendungsprogramm aus Abbildung 10 vor dem Transaktionsende mit einer weiteren XQuery-Anfrage direkt auf den geänderten Nachnamen des Autors zugreifen, so kann dies mit einem Methodenaufruf wie in Abbildung 13 durchgeführt werden. Da für dieses Anfrageergebnis wiederum eine W3C-Dokumentreferenz ermittelt werden kann, stehen auch hier wieder alle drei Schnittstellen für die weitere Verarbeitung zur Verfügung.

3.5 Propagierung von Änderungen

Wird nach der Ausführung einer XQuery-Anfrage das Ergebnisdokument mit Methoden der DOM-Schnittstelle bearbeitet, können neben Navigationsschritten auch beliebige

Änderungen an diesem Ergebnis vorgenommen werden. Bei der Materialisierung eines aktualisierbaren Anfrageergebnisses mit Verweisknoten können dabei vier verschiedene Beziehungstypen zwischen dem Ergebnis- und dem Basisdokument entstehen, die in Abbildung 14 dargestellt sind. Die Semantik dieser Beziehungen und deren Auswirkungen bei der Modifikation des Ergebnisdokuments auf das Basisdokument sollte der Benutzer schon bei der Formulierung seiner XQuery-Anfrage beachten.

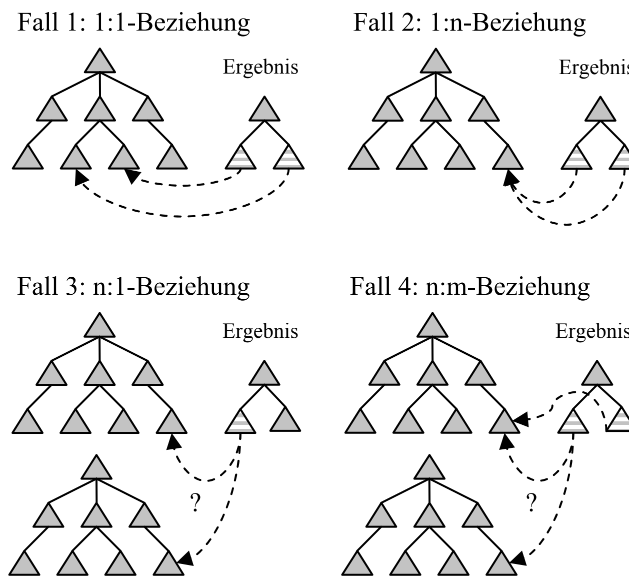


Abbildung 14: Beziehungstypen bei der Materialisierung

Im ersten Fall handelt es sich um eine 1:1-Beziehung, die bereits im vorangegangenen Beispiel in Abschnitt 3.4 betrachtet wurde. Verweisknoten im Anfrageergebnis zeigen auf Knoten im Basisdokument, dabei wird jeder Knoten im Basisdokument nur von einem Verweisknoten referenziert. Eine Änderung im Ergebnisdokument bewirkt die Änderung genau eines Knotens im Basisdokument und umgekehrt.

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

Fall 3 entsteht bei einer Verbundoperation, bei der der Verweisknoten aufgrund der Wertgleichheit zweier Basisknoten erzeugt wird. Hier muss bei der Formulierung der Anfrage sorgfältig ausgewählt werden, welcher der beiden Knoten (trotz gleichen Werts) referenziert 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 betrachtet werden kann, da nach Änderung eines Knotenwerts keine Anpassung des Ergebnisdokuments bezüglich der gestellten Anfrage erfolgt. Das bedeutet, dass ein Verweisknoten, der aufgrund der Qualifikation bezüglich einer Anfragebedingung erzeugt wurde, nach einer Änderungsoperation nicht aus dem Ergebnis entfernt wird, falls er nun nicht mehr die Anfragebedingung erfüllen würde.

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

Die Behandlung von Textknoten stellt ein weiteres Problem bei der Materialisierung eines Anfrageergebnisses dar. Zur Gewährleistung der XML-Konformität fordert XQuery, dass zwei nebeneinander liegende Textknoten zu einem einzigen Knoten verschmolzen werden müssen. Diese Forderung weichen wir bei der Nutzung von Verweisknoten auf, da bei einer Verschmelzung nicht mehr bestimmt werden kann, welcher Teil des entstandenen Textknotens auf welchen Basisknoten beruht. Daher führen wir die geforderte Verschmelzung nur aus, wenn beide benachbarten Textknoten „echte" XML-Knoten und keine Verweisknoten sind. Ist dagegen mindestens einer der beiden benachbarten Textknoten ein Verweisknoten, so verzichten wir auf die Verschmelzung. Ein nachgeschalteter Zugriff auf das Ergebnisdokuments mit der DOM-Schnittstelle liefert in diesem Fall zwei direkt benachbarte Textknoten, die getrennt voneinander mit Propagierung auf deren jeweilige Basisdokumente aktualisiert werden können.

4 Zusammenfassung und Ausblick

In diesem Beitrag wurde ein natives XML-Datenbanksystem und dessen anwendungsseitige XML-Programmierschnittstelle vorgestellt, mit der es möglich ist, drei verschiedenartige Schnittstellentypen zum ereignisbasierten SAX-Zugriff, navigationsorientierten DOM-Zugriff und deskriptiven XQuery-Zugriff auf XML-Daten kombiniert in einer transaktionsgeschützten Umgebung einzusetzen. Die Transaktionsisolation zur Vermeidung von Mehrbenutzeranomalien wird mit einem auf Knotenebene realisierten pessimistischen Sperrverfahren erreicht. Da eine effiziente Unterstützung der Sperranforderung und -freigabe stark vom zugrunde liegenden Adressierungs- und Zugriffsverfahren auf die einzelnen XML-Knoten abhängt, diskutierten wir zwei alternative Implementierungen der Speicherungsstrukturen. Das Ergebnis einer XQuery-Anfrage wird im Datenbanksystem mit so genannten Verweisknoten materialisiert und ermöglicht somit die Propagierung von Ergebnisaktualisierungen auf das Basisdokument und die erneute Verarbeitung des Ergebnisses mit SAX, DOM oder XQuery innerhalb desselben Transaktionskontexts. Dabei können mehrere Anwendungen im logischen Einbenutzerbetrieb gleichzeitig auf einem gemeinsamen XML-Dokument arbeiten.

In weiteren Schritten unserer Arbeit werden wir prototypische Anwendungen entwerfen, die mit den hier vorgestellten Schnittstellen auf zentral verwaltete XML-Dokumente im Mehrbenutzerbetrieb zugreifen. Dabei gilt es zu untersuchen, inwieweit die Einbettung von klassischen XML-Schnittstellen in eine transaktionsgeschützte Umgebung die Anwendungsentwicklung unterstützen kann und ob die bekannten Methoden *beginWork*, *commitWork* und *abortWork* zur Transaktionssteuerung im XML-Umfeld ausreichend sind. Hier wäre z. B. eine Methode zur Freigabe von Änderungen unter Beibehaltung der gehaltenen Lesensperren denkbar.

Ein weiteres großes Forschungsgebiet bildet für uns nach wie vor die Synchronisation nebenläufiger Transaktionen in XML-Datenbanksystemen, wobei wir zunächst noch

eine Lösung zur Vermeidung von Phantomen in der Isolationsstufe *serializable* finden müssen, die das Auftreten von Phantomen auch zuverlässig bei gleichzeitiger Anwendung aller drei Schnittstellen beim Einsatz von Sekundärindexen verhindert. Des Weiteren ist über ein persistentes Sperrkonzept nachzudenken, das für „lange" Transaktionen einen Check-In/Out-Mechanismus für XML-Fragmente erlaubt und das sich in die vorhandene Transaktionssynchronisation integrieren lässt. Da wir mit dem XTC-Server über ein lauffähiges Datenbanksystem verfügen, wird es uns damit möglich sein, durch Austausch der Sperrmanager-Implementierung verschiedene Sperrprotokolle in jeweils exakt derselben Laufzeitumgebung miteinander zu vergleichen.

Literaturverzeichnis

[Bo04] Boag, S., Chamberlin, D., Fernández, M., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML Query Language. W3C Working Draft (2004)

[BR04] Böhme, T., Rahm, E.: Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In Proc. 3rd International Workshop on Data Integration over the Web (DIWeb), Riga, Lettland (2004)

[Br02] Brownell, D.: SAX2. O'Reilly-Verlag (2002)

[Dr04] 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 (2004)

[Fe04] Fernández, M., Malhotra, A., Marsh, J., Nagy, M., Walsh, N.: XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft (2004)

[Ho04] 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. W3C Recommendation (2004)

[HH03] Haustein, M., Härder, T.: taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API. In Proc. 7th ADBIS Conference, Dresden, Deutschland, S. 88-102 (2003)

[HH04] Haustein, M., Härder, T.: Adjustable Transaction Isolation in XML Database Management Systems. In Proc. 2nd Int. XML Database Symposium, Toronto, Kanada, S. 46-53 (2004)

[HR99] Härder, T., Rahm E.: Datenbanksysteme – Konzepte und Techniken der Implementierung. Springer-Verlag (1999)

[Ma04] Mathis, C.: Anwendungsprogrammierschnittstellen für XML-Datenbanksysteme. Diplomarbeit, Technische Universität Kaiserslautern, AG DBIS (2004)

[Me02] Meier, W.: eXist: An Open Source Native XML Database. In Proc. NODe 2002 Web- and Database-Related Workshops, Erfurt, Deutschland (2002)

[Ne04] O'Neil, P. et. al.: ORDPATHs: Insert-Friendly XML Node Labels. In Proc. ACM SIGMOD Conference Industrial Track, Paris, Frankreich (2004)

[SWK02] Schmidt, A., Waas, F., Kersten M.: XMark: A Benchmark for XML Data Management. In Proc. 28th VLDB Conference, Hong Kong, China (2002)

[Ta02] Tatarinov, I., Viglas, S. D., Beyer, K., Shanmugasundaram, J., Shekita, E., Zhang, C.: Storing and Querying Ordered XML Using a Relational Database System. In Proc. ACM Sigmod Conference, Madison, Wisconsin, USA (2002)

[TFW03] Tesch, T., Fankhauser, P., Weitzel, T.: Skalierbare Verarbeitung von XML mit Infonyte-DB. In Proc. Datenbanksysteme für Business, Technologie und Web (BTW), Leipzig, Deutschland S. 578-590 (2003)