Verhinderung von Phantomen in XML-Datenbanksystemen mit wertbasierten Achsensperren

Michael P. Haustein

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

Abstract: Die parallele und transaktionssichere Verarbeitung von operationalen Daten in XML-Datenbanksystemen erfordert ein XML-Datenmodell mit zugeschnittenen Speicherungsstrukturen und Synchronisationsalgorithmen, um einen hohen Transaktionsdurchsatz zu erzielen. Zur Gewährleistung der höchsten Isolationsstufe *serializable* reicht es nicht aus, die während der Anfrageverarbeitung gelesenen XML-Knoten und Dokumententeile vor Änderungen durch nebenläufige Transaktionen zu schützen. Zusätzlich gilt es zu verhindern, dass Daten modifiziert oder neu eingefügt werden können, wenn sie sich dadurch nachträglich für eine bereits ausgewertete Anfrage einer noch laufenden Transaktion qualifizieren (so genannte Phantome).
In diesem Beitrag stellen wir das Konzept und die Implementierung wertbasierter Achsensperren vor, die das Auftreten von Phantomen bei gleichzeitigem Zugriff auf ein XML-Dokument mit der SAX-, DOM- und XQuery-Schnittstelle verhindern und somit die Realisierung der Isolationsstufe *serializable* für XML-Datenbanksysteme ermöglichen.

1 Einleitung

Um native XML-Datenbanksysteme (XDBSe) als Alternative zu klassischen relationalen Datenbanksystemen für die Verarbeitung operationaler Daten zu etablieren, erfordert die Verwaltung von XML-Daten in XDBSen die Gewährleistung der bekannten ACID-Eigenschaften [HR99] während der Transaktionsverarbeitung. Neben der Konsistenz der Daten und deren atomaren dauerhaften Speicherung ist ein möglichst feingranularer Synchronisationsmechanismus zur Verfügung zu stellen, um bei der transaktionsorientierten Verarbeitung der XML-Daten einen möglichst hohen Grad an Parallelität zu erreichen. Dies wird dadurch erschwert, dass für XDBSe (im Gegensatz zur einheitlichen Anfragesprache SQL bei den relationalen Systemen) schon für den Grundmechanismus der Transaktionsisolation verschiedenartige Schnittstellen und Anfragesprachen, wie beispielsweise SAX [Br02], DOM [Ho04] oder XQuery [Bo05], berücksichtigt werden müssen.

1.1 Problembeschreibung

Mit dem taDOM-Synchronisationskonzept [HH03, HH04] haben wir die Isolation nebenläufiger Transaktionen bis zur Isolationsstufe *repeatable* gelöst. Anwendungen können an beliebigen Positionen in XML-Dokumente „einspringen“ und diese mittels Navigationsoperationen traversieren und modifizieren. Eine prototypische Implementierung in unserem nativen XDBS *XML Transaction Coordinator* [XTC] wurde in [Ha05] präsentiert. Eine detaillierte Beschreibung der Datenbankschnittstelle und die Optimierung der Sperrmechanismen zu den resultierenden Protokollen taDOM2, taDOM2+, taDOM3 und taDOM3+, sowie dem Beweis deren Korrektheit, wird in [HH05a] behandelt.

Für die höchste Isolationsstufe *serializable* zählt zusätzlich zu den Maßnahmen der Stufe *repeatable* die Phantomverhinderung [HR99]. In einem XDBS tritt ein Phantom beispielsweise auf, wenn nach der Anfrageauswertung für eine Transaktion T_1 eine weitere Transaktion T_2 einen für die Anfrage relevanten Knoten k einfügt, und T_1 bei wiederholter identischer Anfrage ein um k erweitertes Ergebnis erhält. Dies ist möglich, da bei der ersten Anfrageauswertung für T_1 nur die zu diesem Zeitpunkt relevanten Knoten gesperrt werden[1]. Selbst die DOM-Schnittstelle bietet bereits Operationen für einfache Suchanfragen (siehe Abschnitt 2.4), für die das Auftreten von Phantomen zu behandeln ist.

Ein wesentlich größeres Problem stellt die Verarbeitung von XQuery-Anfragen dar. Prinzipiell lassen sich diese Anfragen zur Auswertung auf DOM-basierte Operationen abbilden. Eine Lösung des Phantomproblems für die DOM-Schnittstelle würde damit auch das Auftreten von Phantomen an der XQuery-Schnittstelle verhindern [Ha05]. Allerdings führt die Auswertung von komplexen XQuery-Anfragen (bis zu zwölf nutzbare Pfadachsen [Bo05]) mit ausschließlich navigationsorientiertem Zugriff auf die Datenbasis zu einem katastrophalen Leistungsverhalten, da ohne Indexunterstützung oft große Dokumententeile traversiert werden müssen, um deren Relevanz für die Anfrage auszuschließen.. Daher sind zusätzliche Indexstrukturen erforderlich, die einerseits die Anfrageauswertung erheblich beschleunigen können, anderseits jedoch zusätzliche Maßnahmen zur Vermeidung von Phantomen erfordern. Das dazu bekannteste Verfahren ist die Key-range-locking-Methode [GR97], bei der die für die Anfrage relevanten Schlüsselbereiche eines Index lesend und bei Änderungen der Datenbasis die betroffenen Schlüsselbereiche aller Indexe exklusive gesperrt werden müssen. Dieses Verfahren ist für XML-Datenbanksysteme jedoch zu restriktiv und grobgranular, da für die Auswertung einer Anfrage im optimalen Fall nur Wertebereiche in ausgesuchten Teilbäumen und nicht im gesamten XML-Dokument zu sperren sind.

1.2 Lösungsansatz

Die Transaktionsisolation bis zur Isolationsstufe *repeatable* wird durch das Sperren einzelner XML-Knoten und virtueller Navigationskanten zwischen diesen Knoten sichergestellt [HH04, HH05a]. Lesesperren auf Kanten verhindern bereits Phantome für Bereiche, auf die mit reinen Navigationsoperationen zugegriffen wird [Ha05], da somit

[1] Das Phänomen der Phantome sollte nicht mit der Eigenschaft *repeatable read* verwechselt werden, bei der nur gefordert wird, dass sich bereits gelesene Daten nicht mehr ändern dürfen, aber keine Aussage über neu erzeugte Daten getroffen wird.

Einfügungen in bereits durchlaufenen Pfaden ausgeschlossen werden. Zur Phantomvermeidung bei Zugriffen über sekundäre Indexstrukturen stellen wir in diesem Beitrag den Ansatz der wertbasierten Achsensperren vor. Damit wird es möglich, Knoteneinfügungen oder –umbenennungen in einem Dokumentenbereich zu blockieren, der durch einen XML-Knoten, eine Pfadachse und einen Anfragewert spezifiziert wird.

Zur Erläuterung des Zusammenspiels von Speicherungsstrukturen, Anfrageverarbeitung und Sperranforderungen stellen wir in Kapitel 2 zunächst kurz das taDOM-Datenmodell und die von uns eingesetzten DeweyIDs zur Knotenadressierung vor. Darauf aufbauend erläutern wir die Strukturen zur Dokumentenspeicherung und –indexierung und erklären, wie damit die Anfrageauswertung beschleunigt wird. Kapitel 3 verdeutlicht die Problematik der Phantomerscheinungen beim Einsatz dieser Indexstrukturen mit den Schnittstellen SAX, DOM und XQuery und erklärt detailliert unseren Lösungsansatz der wertbasierten Achsensperren. Kapitel 4 fasst den Inhalt des Beitrags zusammen und geht auf künftige Forschungsarbeiten ein.

1.3 Verwandte Arbeiten

Auf die Phantomverhinderung in der Isolationsstufe *serializable* wird in Forschungsarbeiten noch seltener eingegangen als auf generelle XML-Transaktionsisolation selbst. Nach unseren Kenntnissen gibt es bisher nur drei Arbeiten, die sich mit dem Problem der Phantome beschäftigen. Allerdings unterstützt keiner dieser Ansätze sowohl navigationsorientierte als auch pfadbasierte Anfragen zugleich, noch werden bei den pfadbasierten Ansätzen alle XQuery-Pfadachsen unterstützt.

Grabs, Böhm und Schek präsentieren in [GBS02] einen XML-Transaktionsmanager, der auf dem Datenbanksystem DB2 von IBM mit dem XML-Extender aufsetzt und zur Synchronisation Pfadsperren mit einem hierarchischen Sperrprotokoll und einem XML-Dokumentenschema (*DataGuide*) kombiniert. Phantome werden in den somit gesperrten Pfaden verhindert. Durch die Sperren auf dem *DataGuide* werden jedoch nicht einzelne Bereiche eines Dokuments gesperrt, sondern alle Bereiche, die sich für den Pfad einer Sperre qualifizieren. Die Autoren betrachten nur die *Child*- und *Descendant*-Achse und unterstützen keine Zugriffe über eine navigationsorientierte Schnittstelle.

Dekeyser und Hidders führen mit ihrem Transaktionsmodell in [DH02] und [DHP03] ebenfalls Pfadsperren ein, die zwar auf separaten Knoten angefordert werden können, allerdings auch nur die *Child*- und *Descendant*-Pfadachse einer Anfrage berücksichtigen. Das Modell sieht nur eine sehr kleine Zahl von Basisoperationen vor; Navigationen oder Abhängigkeiten von weiteren Indexstrukturen werden nicht berücksichtigt.

Helmer, Kanne und Moerkotte beschreiben und evaluieren in [HKM03] und [HKM04] die navigationsbasierten Sperrverfahren *Node2PL*, *NO2PL* und *OO2PL*, die jedoch alle voraussetzen, dass eine Folge von Navigationsoperationen stets bei der Dokumentenwurzel beginnt. Direkte Adressierung von beliebigen (evtl. durch sekundäre Zugriffspfade indexierte) Knoten für die Auswertung von Pfadanfragen sind somit nicht möglich. Als Zielknoten für direkte „Einsprünge“ in XML-Dokumente sind nur Elemente mit ID-Attributen vorgesehen. Dazu wird ein spezieller Sperrmodus verwendet, der bzgl. dieser Elemente Phantome auch zuverlässig verhindert.

2 Speicherungsstrukturen für XML-Dokumente

Jedes XML-Dokument wird zur Speicherung in unserem XML-Datenbanksystem [XTC] zunächst in das taDOM-Datenmodell überführt, wobei alle Knoten mit einer so genannten *DeweyID* adressiert werden. Danach wird das Dokument vollständig in einem B*-Baum abgelegt und für alle Elementknoten und ID-Attribute jeweils eine weitere Indexstruktur zum schnelleren Zugriff angelegt. Die Vorteile dieser Speicherungsstrukturen für die Auswertung von DOM- und XQuery-Anfragen werden am Ende dieses Kapitels behandelt.

2.1 Das taDOM-Datenmodell und die DeweyID

Für unseren XDBS-Forschungsprototypen XTC betrachten wir XML-Dokumente, die nur aus Element-, Attribut- und Textknoten bestehen. Zur Speicherung eines Dokuments wird dessen textuelle Repräsentation (üblicherweise in Form einer Textdatei) zunächst in das taDOM-Datenmodell 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. Werte von Attribut- und Textknoten werden als so genannte *String-Knoten* gespeichert. Durch die neuen Knotentypen lassen sich einige Vorteile für eine erhöhte Parallelität von Transaktionsoperationen erzielen. So können beispielsweise mit einer einzigen Knotensperre auf der Attributwurzel alle Attribute eines Elements geschützt werden und die Existenzprüfung von Attribut- und Textknoten blockiert noch nicht das Auslesen und Ändern deren Werte in den separaten String-Knoten [HH03]. Ein Beispiel für die Überführung eines XML-Dokuments in dessen Darstellung im taDOM-Datenmodell ist in Abbildung 1 gegeben.

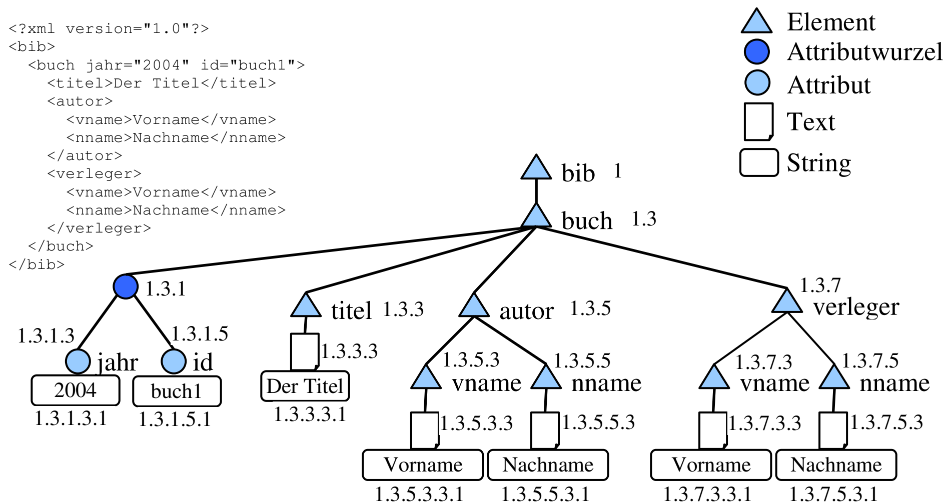

Abbildung 1: Überführung in das taDOM-Datenmodell

Jedem XML-Knoten wird zur eindeutigen Adressierung eine so genannte DeweyID zugewiesen. 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 der DeweyID adaptiert den in [Ne04] publizierten ORDPATH-Ansatz an das taDOM-Datenmodell.

Die DeweyID basiert auf der Dewey-Dezimalklassifikation und nummeriert die Knoten in jeder Ebene separat mit ungeraden aufsteigenden Nummern (so genannte *Divisions*) durch. Mit Ausnahme des Wurzelelements ist die Division *1* für Attributwurzel- und String-Knoten reserviert. Die Divisions einer Ebene werden durch Punkte getrennt; die ID des Elternknotens wird in die ID des Kindknotens aufgenommen. Durch die initiale Vergabe der ungeraden Divisions ist es möglich, an jeder Position beliebig viele neue Knoten einzufügen. Zwischen den DeweyIDs *1.3.5* und *1.3.7* in Abbildung 1 können so beispielsweise die IDs *1.3.6.3*, *1.3.6.5*, *1.3.6.7,* usw. eingefügt werden. Zwischen den IDs *1.3.6.3* und *1.3.6.5* finden wiederum die IDs *1.3.6.4.3*, *1.3.6.4.5*, *1.3.6.4.7*, usw. Platz (siehe dazu auch [HH05b]). Für eine gegebene DeweyID ist somit die Berechnung der Ebene im Dokument (Anzahl der ungeraden Divisions minus 1 bei Wurzelelementebene 0) und der IDs aller Vorfahren bis zur Dokumentenwurzel ohne Zugriff auf das gespeicherte XML-Dokument möglich.

2.2 Speicherung und Indexierung von XML-Elementen

Zur Speicherung von XML-Dokumenten lässt sich die DeweyID-Adressierung sehr effektiv nutzen, da sich die IDs neu einzufügender Knoten stets in die sequentielle Ordnung (Baumdurchlauf *left-most depth-first*) der bereits vorhandenen Nachbarknoten einfügen. Somit wird nur ein einziger B*-Baum zur Speicherung des gesamten Dokuments benötigt und die Modifikation von Knoten erfordert keine Reorganisation von benachbarten Fragmenten (Abbildung 2a). Ein Eintrag im B*-Baum wird aus der DeweyID als Schlüsselanteil und dem eigentlichen Knotenwert als Wertanteil gebildet. Da die Speicherung der einzelnen XML-Knoten in den Blättern des Baums auch deren sequentieller Reihenfolge im Dokument entspricht, und die DeweyID jedes Knotens die ID des Elternknotens als Präfix enthält, implementieren wir zur Zeit eine Präfix-Komprimierung [HR99] der B*-Baum-Schlüssel zur weiteren Optimierung. Der Speicherplatzverbrauch des Wertanteils für Element- und Attributknoten wird zusätzlich 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.

Zur Beschleunigung der Auswertung mengenorientierter XQuery-Anfragen wird für jedes XML-Dokument zusätzlich ein *Elementindex* (Abbildung 2b) angelegt. In dieser Indexstruktur wird jeder Elementname in einem B-Baum geführt. Für jeden Elementnamen wird ein weiterer B*-Baum angelegt, in dem die DeweyIDs aller Elementknoten gespeichert werden, die den entsprechenden Namen tragen. Da der B*-Baum die Nutzdaten in den Baumblättern ablegt, können die DeweyIDs aller XML-Knoten für einen gegebenen Elementnamen sehr effizient mit sequentiellen Leseoperationen für die Anfrageverarbeitung bestimmt werden.

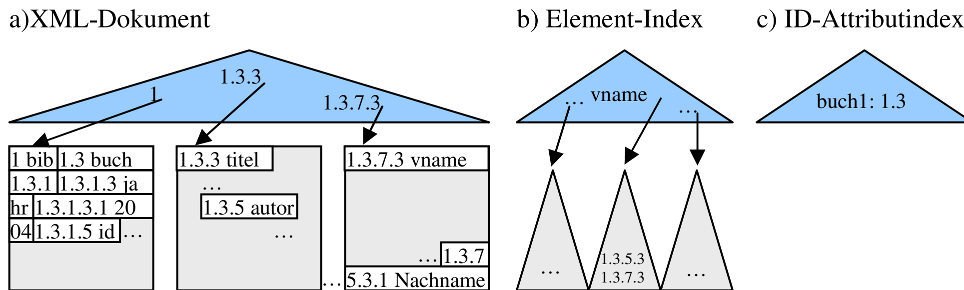

Abbildung 2: Speicherungsstrukturen

Zum schnelleren Zugriff auf Elemente, die mit ID-Attributen ausgezeichnet sind, wird ein weiterer B-Baum, der *ID-Attributindex* (Abbildung 2c), erzeugt. In diesem wird zu jedem ID-Attributwert die DeweyID des besitzenden Elements abgelegt. Ein Aufruf der DOM-Operation *getElementById(„buch1“)* greift so beispielsweise nach einer Suche im ID-Attributindex direkt auf das XML-Element *buch* mit der ID *1.3* zu, ohne das gesamte Dokument nach Elementen mit ID-Attributen durchsuchen zu müssen.

2.3 Auswertung von Pfadachsen mit Indexstrukturen

Aufgrund der Präfixeigenschaft der DeweyIDs (ein XML-Knoten enthält in seiner DeweyID die IDs aller Vorfahren bis zur Dokumentwurzel) kann der Elementindex zur effizienten Auswertung von XQuery-Pfadanfragen auf einem Kontextknoten genutzt werden. Sollen beispielsweise im XML-Dokument in Abbildung 1 alle Elemente *vname* bestimmt werden, die im Teilbaum des Elements *buch* mit der DeweyID *1.3* liegen, so wird nicht der gesamte Teilbaum des gespeicherten Dokuments geladen und durchsucht, sondern der Elementindex für den Namen *vname* geöffnet und alle DeweyIDs zurückgeliefert, die sich im Teilbaum (siehe dazu die unten erläuterte XQuery-Pfadachse *descendant*) mit der Wurzel-ID 1.3 befinden—also *1.3.5.3* und *1.3.7.3*.

Für die dazu benötigte Bestimmung der Lage zweier DeweyIDs bzgl. einer Pfadachse muss zunächst die sequentielle Ordnung auf DeweyIDs definiert werden: Für zwei gegebene DeweyIDs d_1 und d_2 wird zunächst die kleinere Anzahl *n* der jeweils vorhandenen Divisions ermittelt. Daraufhin werden nacheinander die Divisions an den Positionen *1* bis *n* verglichen. Die DeweyID d_1 ist *kleiner* als d_2, sobald die Division von d_1 an der aktuell untersuchten Position kleiner ist als die Division 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 Divisions gleich sein, so ist die DeweyID mit der geringe-

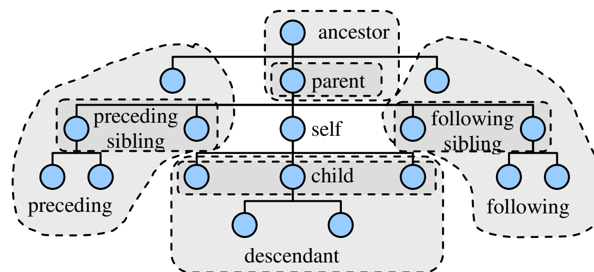

Abbildung 3: XQuery-Pfadachsen

ren Anzahl von Divisions die *kleinere*. Sind zusätzlich auch noch die Anzahl der Divisions gleich, so handelt es sich bei d_1 und d_2 um *dieselbe* DeweyID.

Sollen nun für einen Kontextknoten k mit der DeweyID d_k alle Elemente mit dem Namen *tagname* innerhalb einer Achse ermittelt werden, so können sieben der zwölf in XQuery [Bo05] definierten Pfadachsen (siehe dazu Abbildung 3) mit Hilfe des Elementindex ausgewertet werden. Dazu wird der Elementindex mit dem angegebenen Schlüssel *tagname* geöffnet und alle dort abgelegten DeweyIDs d_i (i=1,…,m) mit d_k bzgl. der angefragten Pfadachse untersucht:

1. *self::tagname*: Hat k den Elementnamen *tagname*, so qualifiziert er sich für die *Self*-Achse. Dazu wird der Elementindex nicht benötigt.
2. *attribute::tagname*: Attributnamen werden nicht im Elementindex verwaltet, daher können Attributanfrage nicht mit dem Index beantwortet werden.
3. *parent::tagname*: Die DeweyID des Elternknotens von k lässt sich direkt aus d_k ermitteln. Dazu wird zuerst die letzte Division von d_k und daraufhin alle geraden Division-Werte am Ende von d_k entfernt. Der Elternknoten mit der so ermittelten ID kann direkt geladen und ein Namensvergleich durchgeführt werden. Der Elementindex wird dazu nicht benötigt.
4. *ancestor::tagname*: Um alle Vorfahren von k mit dem angegebenen Namen bis zur Dokumentwurzel zu bestimmen, kann die Methode zur Bestimmung des Elternknotens bis zum Erreichen der Wurzel rekursiv angewendet werden. Ein Zugriff auf den Elementindex ist nicht erforderlich.
5. *ancestor-or-self::tagname*: Liegt d_i bzgl. d_k auf der *Ancestor*-Achse oder der *Self*-Achse, so liegt d_i auf der *Ancestor-or-self*-Achse von k. Da nur die *Ancestor*- und die *Self*-Achse berücksichtigt werden müssen, ist auch hier der Zugriff auf den Elementindex nicht erforderlich.
6. *preceding::tagname*: 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 k.
7. *preceding-sibling::tagname*: 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 k.
8. *descendant::tagname*: Liegt d_k auf der *Ancestor*-Achse von d_i, so gehört d_i zu einem Nachfahren (*Descendant*-Achse) von k.
9. *descendant-or-self::tagname*: 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 k.
10. *child::tagname:* Ist die DeweyID des Elternknotens von d_i gleich d_k (*Parent*-Achse), so adressiert d_i einen Kindknoten von k.
11. *following-sibling::tagname*: 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.
12. *following::tagname*: Wenn d_k kleiner als d_i ist und d_k nicht zu einem Vorfahren (*Ancestor*-Achse) von d_i gehört, so adressiert d_i einen *Following*-Knoten von k.

Die XML-Knoten, deren DeweyIDs aus dem Elementindex ermittelt wurden, können während der folgenden Anfrageverarbeitung bei Bedarf direkt ohne Navigationsoperationen adressiert und geladen werden. Wie mehrere solcher Ergebnisse in die Auswertung einer vollständigen XQuery-Anfrage einfließen, wird in [MH05] geschildert und ist zur Diskussion der Phantomproblematik im Rahmen dieses Beitrags nicht relevant.

Auch die DOM-Schnittstelle [Ho04] bietet zwei Suchanfragen, die mit Hilfe des Elementindex und des ID-Attributindex beantwortet werden können:

- *getElementById(IDvalue)*: Liefert das XML-Element, das ein ID-Attribut mit dem angegebenen Wert besitzt. Dazu wird der ID-Attributindex mit *IDvalue* als Schlüsselwert geöffnet, die DeweyID des besitzenden Elements ermittelt und der entsprechende Elementknoten geladen. Für diesen Indexzugriff ergänzen wir zur Transaktionsisolation die Achse *IDvalue-descendant-or-self*.
- *getElementsByTagName(tagname)*: Liefert alle Elemente mit dem angegebenen Namen tag*name* im Teilbaum des Kontextknotens und entspricht damit einer Anfrage auf der *Descendant-or-self*-Achse, die mit Hilfe des Elementindex ausgewertet werden kann.

2.4 Phantomerscheinungen bei SAX, DOM und XQuery

Die SAX-Schnittstelle [Br02] ist bzgl. der Phantomerscheinungen unproblematisch, da alle Knoten des zu verarbeitenden XML-Dokuments mit sequentiellen Leseoperationen bestimmt werden können. Dabei wird der traversierte Dokumentenbereich mit Knoten- und Kantensperren gegen Änderungs- und Einfügeoperationen parallel ablaufender Transaktionen vollständig isoliert [Ha05]. Phantome können somit nicht auftreten.

Die *getElementById()*-Methode der DOM-Schnittstelle (siehe Abschnitt 2.3) ist die problematischste Operation bzgl. der Phantomvermeidung. Zum einen ist zu verhindern, dass ein weiteres ID-Attribut mit einem bereits angefragten und nicht vorhandenen ID-Wert eingefügt wird. Zum anderen muss auch verhindert werden, dass ein beliebiges existierendes Attribut mit diesem Wert in ein ID-Attribut umbenannt wird. Ergänzend bietet DOM zwei weitere Operationen an, die in unserer Implementierung zwar nicht über einen Index ausgewertet werden, jedoch ebenfalls vor Phantomen zu schützen sind:

- *getAttribute(name)*: Diese Operation liefert einer Transaktion T_1 für ein Kontextelement den Wert des Attributs mit dem angegebenen Namen. Existiert ein solches Attribut nicht, so kann auch kein entsprechender Knoten gesperrt werden. Es ist jedoch zu verhindern, dass ein Attribut mit diesem Namen von einer weiteren Transaktion T_2 eingefügt wird, solange T_1 noch läuft.
- *hasAttribute(name)*: Ähnlich der Operation getAttribute() wird hier jedoch nur die Existenz eines Attributs mit dem angegebenen Namen überprüft. Wenn das Attribut nicht vorhanden ist, so muss analog zum obigen Fall das Einfügen eines solchen Attributs bis zum Transaktionsende blockiert werden.

Die Anteile einer XQuery-Anfrage, die nicht mit dem Elementindex beantwortet werden können, müssen durch einen Einsprung in das XML-Dokument und nachfolgende Navigationsoperationen ausgewertet werden. Für diese Operationenfolge ist durch Knoten-

und Kantensperren [Ha05] Phantomschutz gewährleistet. Für die drei Pfadachsen *Parent*, *Ancestor* und *Ancestor-or-self* sind ebenfalls keine Vorkehrungen zur Phantomvermeidung zu treffen, da. diese durch direktes Berechnen der DeweyIDs, Sperren und Laden der ermittelten Vorgängerknoten ausgewertet werden. Da jeder Knoten (bis auf die Dokumentwurzel) genau einen Elternknoten besitzt und keine neuen Elternknoten eingefügt werden können, entstehen im Pfad zur Dokumentenwurzel keine Phantome.

Die Auswertung der *Attribute*-Achse kann analog zu den oben beschriebenen Attribut-Methoden der DOM-Schnittstelle Phantome beinhalten. Somit sind in einem einheitlichen Isolationsmechanismus für Anfragen auf der *Attribute*-Achse, der *Self*-Achse, der verbleibenden sieben XQuery-Pfadachsen, die über den Elementindex ausgewertet werden können, sowie der beschriebenen ID-Attribut-Anfrage für die Schnittstellen SAX, DOM und XQuery Sperren anzufordern, die das Auftreten von Phantomen verhindern.

3 Phantome an der XDBS-Satzschnittstelle verhindern

Für die Anfrageverarbeitung müssen alle Operationen der externen XML-Schnittstellen SAX, DOM und XQuery auf die Satzschnittstelle unseres XDBS abgebildet werden. Wie im letzten Abschnitt beschrieben, können sieben der zwölf in XQuery definierten Pfadachsen und zwei Suchanfragen der DOM-Schnittstelle mit Hilfe des Elementindex bzw. des ID-Attributindex ausgewertet werden. Dazu stellt unsere Satzschnittstelle zwei Operationen zur Verfügung, die zusammen mit den 19 Basisoperationen zur Navigation und Modifikation eines XML-Dokuments im folgenden Abschnitt beschrieben werden. Abschnitt 3.2 führt zur Phantomvermeidung die *wertbasierten Achsensperren* ein und erläutert die erforderlichen Sperrmodi für die Basisoperationen. Der letzte Abschnitt 3.3 geht auf Implementierungsaspekte wertbasierter Achsensperren ein.

3.1 Die Satzschnittstelle des XML Transaction Coordinator

Zum Zugriff auf die indexierten DeweyIDs im Elementindex (siehe Abbildung 2b) dient die Operation *getElementDeweyIDs()*, die alle Elemente mit einem spezifizierten Namen liefert, die sich bzgl. eines bereits geladenen Kontextknotens auf einer der in Abschnitt 2.3 definierten Pfadachsen befinden. Die Operation *getElementByIDattribute()* lädt mit dem ID-Attributindex (Abbildung 2c) das XML-Element, welches ein ID-Attribut mit übergebenem Wert besitzt. Zusätzlich zu diesen beiden Operationen gibt es an der Satzschnittstelle 19 Basisoperationen:

Die *getNode()*, *getParentNode()*, *getPrevSibling()*, *getNextSibling()*, *getFirstChild()* und *getLastChild()* Operationen laden einen XML-Knoten und ermöglichen einfache Navigationsschritte zu dessen Eltern-, Geschwister- und erstem und letztem Kindknoten. Die Operationen *getChildNodes()* bzw. *getFragmentNodes()* liefern alle Kindknoten des Kontextknotens bzw. alle Knoten des Teilbaums, der durch den Kontextknoten als Wurzel adressiert wird.

Die Operationen *getValue()* und *setValue()* laden und schreiben die Werte einzelner XML-Knoten. Im Fall von Elementknoten ist dies der Elementname selbst, bei Attribut- und Textknoten ist der Wert in einem String-Knoten gespeichert (siehe Abbildung 1).

Auf einem Elementknoten kann die *getAttribute()*-Operation ausgeführt werden, die für einen spezifizierten Attributnamen den entsprechenden Attributknoten liefert (oder *null,* wenn ein Attribut mit diesem Namen nicht existiert). Eine Liste aller Attribute eines Elementknotens lässt sich mit *getAttributes()* erzeugen. Die Operation *setAttribute()* setzt für ein Attribut den angegebenen Wert oder legt ein Attribut-Wert-Paar an, falls ein solches Attribut noch nicht vorhanden ist. *renameAttribute()* benennt ein existierendes Attribut um, ohne auf dessen Wert zuzugreifen oder diesen zu ändern.

Neue Element- oder Textknoten können mit den Operationen *appendChild()*, *prepend-Child()*, *insertBefore()* oder *insertAfter()* eingefügt werden, die für den Kontextknoten einen neuen ersten oder letzten Kindknoten bzw. einen neuen davor oder dahinter liegenden Geschwisterknoten anlegen.

Die Operation *deleteNode()* löscht innerhalb eines XML-Dokuments ein vollständiges Fragment, das durch den Kontextknoten als Wurzel bestimmt wird.

3.2 Wertbasierten Achsensperren

Zur Vermeidung von Phantomen müssen bei der Ausführung der oben beschriebenen Operationen auf einem Kontextknoten zusätzlich zu den in [HH05a] beschriebenen Knoten- und Kantensperren so genannte *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 Pfadachsen *Self, Attribute, Child, Descendant, Preceding-Sibling, Preceding, Following-Sibling,* oder *Following* auswählt,
- *value* den Anfragewert[2] angibt, und
- *mode* einer der beiden Sperrmodi *shared* (R) oder *exclusive* (X) ist.

Sollen beispielsweise für eine XQuery-Anfrage die IDs aller *Following*-Elementknoten des Titel-Elements in Abbildung 1 mit dem Namen *vname* bestimmt werden, so ist mit dem Aufruf der Operation *getElementDeweyIDs* die wertbasierte Achsensperre *laxis(1.3.3, following, „vname“, R)* anzufordern. Danach können die indexierten IDs *1.3.5.3* und *1.3.7.3* aus dem Elementindex geladen werden. Die Suche des Elements mit ID-Attributwert *buch1* über die Operation getElementByIDattribute() hat vor dem Zugriff auf den ID-Attributindex die Achsensperre l_{axis}*(1, IDvalue-descendant-or-self, „buch1“, R)* zur Folge.

Umgekehrt müssen bei Modifikationen der XML-Dokumente zur Phantomvermeidung vor dem Einfügen neuer Knoten bzw. vor dem Schreiben neuer Werte auf den betroffenen Kontextknoten für die *Self*-Achse exklusive Achsensperren angefordert werden. Wird zum Beispiel in Abbildung 1 das Element *nname* des Autors in *nachname* umbenannt, so muss dafür zunächst die Sperre l_{axis}*(1.3.5.5, Self, „nachname“, X)* eingerichtet

---

[2] Für die im Folgenden aufgeführten Beispiele beschreibt der Parameter *value* zur Vereinfachung den exakten Anfragewert. Analog zum Key-range-Verfahren [GR97] kann jedoch auch ein Bereich mit Anfangs- und Endwert angegeben werden.

werden. Das Hinzufügen eines ID-Attributs mit dem Wert *verl1* zum Element *verleger* würde die Achsenperren l_{axis}*(1.3.7, Attribute, „id“, X)* und l_{axis}*(1.3.7, IDvalue-descendant-or-self, „verl1“, X)* erfordern. Mit der ersten Sperre wird die Operation verzögert, falls eine noch laufende Transaktion für die Operation *getAttribute(„id“)* auf dem Verleger-Element eine negative Antwort erhalten hat (Attribut existiert nicht); die zweite Sperre verzögert die Operation, falls eine noch laufende Transaktion für die Operation *getElementById(„verl1“)* eine negative Antwort erhalten hat (Element mit einem solchen ID-Attributwert existiert nicht). Wären die entsprechenden Knoten vorhanden, so hätte eine Operationsblockierung aufgrund von Lesesperren auf den Knoten selbst stattgefunden. Eine vollständige Übersicht der anzufordernden Achsensperren für die Operationen der Satzschnittstelle ist im Anhang A gegeben.

Um eine angeforderte Achsensperre auf dem Kontextknoten zu gewähren, muss diese kompatibel zu allen für denselben Wert bereits eingerichteten Achsensperren sein, deren Bereiche sich überlagern. Dazu muss zunächst die relative Lage der DeweyID jeder Achsensperre bzgl. der DeweyID des Kontextknotens ermittelt werden. Mit den in Abschnitt 2.3 beschriebenen Pfadeigenschaften der DeweyIDs bestimmt man sehr effizient eine der Lagen *Ancestor*, *Parent*, *Preceding*, *Preceding-Sibling*, *Self*, *Child*, *Descendant*, *Following-Sibling* oder *Following*. Für jede dieser neun Lagen wird nun eine *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 diese Bereiche, so müssen für die Kompatibilität der Achsensperren die eingetragenen Sperrmodi gemäß dem klassischen R/X-Protokoll (Abbildung 4) kompatibel sein.

	R	X
R	+	-
X	-	-

Abbildung 4: R/X-Sperrprotokoll

Exemplarisch für die *Preceding*-Lage wird eine Achsenüberlagerungstabelle in Abbildung 5 gezeigt. Die Kopfspalte gibt die angeforderte Pfadachse für den Kontextknoten an, die Kopfzeile die Pfadachse für eine bereits vorhandene Sperre auf einem Knoten in der *Preceding*-Lage. Ist beispielsweise bereits eine Achsensperre l_{axis}*(1.3.3, Following, „vname“, R)* für eine Transaktion T_1 eingerichtet und eine weitere Transakti-

	Preceding	Preceding Sibling	Self	Attribute	Child	Descendant	Following Sibling	Following	ID-value
Preceding	ja	ja	ja	nein	ja	ja	ja	ja	nein
Preceding Sibling	nein	nein	nein	nein	nein	nein	nein	ja	nein
Self	nein	nein	nein	nein	nein	nein	nein	ja	nein
Attribute	nein	nein	nein	nein	nein	nein	nein	nein	nein
Child	nein	nein	nein	nein	nein	nein	nein	ja	nein
Descendant	nein	nein	nein	nein	nein	nein	nein	ja	nein
Following Sibling	nein	nein	nein	nein	nein	nein	nein	ja	nein
Following	nein	nein	nein	nein	nein	nein	ja	ja	nein
IDvalue	nein	nein	nein	nein	nein	nein	nein	nein	nein

Abbildung 5: Achsenüberlagerungstabelle für die *Preceding*-Lage

on T_2 fordert die Sperre l_{axis}*(1.3.7.5, Preceding-Sibling, „vname“, R)* an, so kann Abbildung 5 angewendet werden, da der Wert *vname* übereinstimmt und sich *1.3.3* bzgl. *1.3.7.5* in der *Preceding*-Lage befindet. Die Achsenüberlagerungstabelle sagt nun aus, dass sich die *Preceding-Sibling*-Achse des Kontextknotens mit der Following-Achse eines Knotens in der *Preceding*-Lage überschneidet und somit die R/X-Kompatibilitätsmatix (Abbildung 4) anzuwenden ist. Da der bereits eingetragene Sperrmodus *R* mit dem angeforderten Sperrmodus *R* kompatibel ist, kann die angeforderte Achsensperre trotz der Achsenüberlagerung gewährt werden.

3.3 Implementierung wertbasierter Achsensperren

In unserem nativen XDBS-Prototyp werden Achsensperren zusammen mit den in [HH05a] vorgestellten Knoten- und Kantensperren in einem gemeinsamen Sperrpuffer (Abbildung 6) verwaltet. Um eine Fragmentierung des Puffers zu verhindern, wird jede Sperre mit einer einheitlichen Größe von 22 Byte angelegt. Darin sind die betroffene Achse, der Sperrmodus und alle erforderlichen Verweiszeiger der Hilfsstrukturen kodiert. Da die Werte einer Achsensperre variable Länge besitzen, werden diese in einem separaten Bereich (Phantomwerte) verwaltet und von der eigentlichen Sperre referenziert. Somit muss für mehrere Sperren auf demselben Wert dieser auch nur einmal gespeichert werden. Zur effizienteren Freispeicherverwaltung ist der Sperrpuffer in Blöcke gleicher Größe unterteilt, sodass beim Einrichten einer neuen Sperre nicht der gesamte Puffer durchsucht werden muss, sondern direkt auf einen Block mit ausreichendem Speicherplatz zugegriffen werden kann. Die DeweyIDs der XML-Knoten, auf denen Sperren eingerichtet sind, werden in einer Hash-Tabelle (DHT) abgelegt. Alle Sperren einer DeweyID werden innerhalb des Sperrpuffers in einer doppelt verketteten Liste verwaltet; das erste Element der Kette ist über einen Zeiger von der entsprechenden DeweyID aus erreichbar. Zusätzlich werden alle Sperren auch bzgl. der besitzenden Transaktion in der Transaktionsliste (TAL) doppelt verkettet [GR97], um ein schnelles Auffinden aller Sperren zur Freigabe beim Transaktionsende zu erreichen.

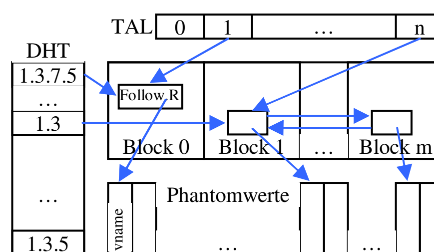

Abbildung 6: Sperrtabelle

4 Zusammenfassung und Ausblick

In diesem Beitrag haben wir zunächst einen Überblick über die Techniken der Speicherung und Indexierung von XML-Dokumenten und der Anfrageverarbeitung in unserem nativen XDBS-Forschungsprototypen XTC gegeben. Auf dieser Grundlage wurde das Konzept der wertbasierter Achsensperren vorgestellt und einige Implementierungsaspekte skizziert. Eine Achsensperre wird mit der Angabe einer Pfadachse, einem Wertebereich und einem entsprechenden Sperrmodus auf einem Kontextknoten angefordert und blockiert in einem somit festgelegten Dokumentenbereich Einfügungen von XML-

Knoten, die dadurch in bereits angefragte Wertebereiche fallen würden. Auf diese Weise wird für XML-Datenbanksysteme, die die feingranulare transaktionssichere Verarbeitung paralleler SAX-, DOM- und XQuery-Anfragen unterstützen, die Gewährleistung der Isolationsstufe *serializable* ermöglicht.

Unsere zukünftige Arbeit bzgl. der Transaktionsisolation wird sich der Kostenbeurteilung der wertbasierten Achsensperren widmen. Dazu gilt es besonders beim Übergang von der Isolationsstufe *repeatable* zur Stufe *serializable* zu untersuchen, welcher Laufzeitaufwand für die Achsensperren zusätzlich zu dem der Knoten- und Kantensperren entsteht und wie stark sich dieser Zusatzaufwand auf den gesamten Transaktionsdurchsatz auswirkt.

Anhang A: Achsensperren für die Operationen der Satzschnittstelle

getNode(deweyID)	Nur interne Anwendung zum Zugriff auf sicher vorhandene Knoten.
getParentNode(k)	Keine Phantome auf der *Parent*-Achse möglich (siehe Abschnitt 2.4)
getPrevSibling(k)	Phantomschutz durch Kantensperre zum Geschwisterknoten [HH04].
getNextSibling(k)	Phantomschutz durch Kantensperre zum Geschwisterknoten [HH04].
getFirstChild(k)	Phantomschutz durch Kantensperre zum ersten Kindknoten [HH04].
getLastChild(k)	Phantomschutz durch Kantensperre zum letzten Kindknoten [HH04].
getChildNodes(k)	Ebenensperre auf k verhindert das Einfügen neuer Kinder [HH04].
getFragmentNodes(k)	Teilbaumsperre auf k verhindert das Einfügen von Knoten [HH04].
getValue(k)	Knotenwert wird durch einzelne Knotenlesesperre geschützt [HH04].
setValue(k, newValue)	Für Elementknoten: l_{axis}(k.deweyID, Self, newValue, X) Für ID-Attributknoten: l_{axis}(element.deweyID,IDvalue,newValue,X)
getAttribute(k,name)	l_{axis}(k.deweyID, Attribute, name, R)
getAttributes(k)	Ebenensperre auf der Attributwurzel (siehe Abbildung 1) von k verhindert das Einfügen neuer Attribute [HH04].
setAttribute(k,name,value)	Wenn Attribut nicht existiert: l_{axis}(k.deweyID, Attribute, name, X) Für ID-Attribute zusätzlich: l_{axis}(k.deweyID, IDvalue, value, X)
renameAttribute(k,old,new)	l_{axis}(k.deweyID, Attribute, new, X)
prependChild(k,newChild)	l_{axis}(newChild.deweyID,Self,newChild.name,X)
appendChild(k,newChild)	l_{axis}(newChild.deweyID,Self,newChild.name,X)
insertBefore(k,newNode)	l_{axis}(newNode.deweyID,Self,newNode.name,X)
insertAfter(k,newNode)	l_{axis}(newNode.deweyID,Self,newNode.name,X)
deleteNode(k)	Teilbaumsperre im Schreibmodus auf k verhindert den Zugriff durch weitere Transaktionen vollständig.
getElementDeweyIDs (k,axis,name)	l_{axis}(k.deweyID, axis, name)
getElementByIDattribute (value)	l_{axis}(1, IDvalue, value)

Literaturverzeichnis

[Bo05] Boag, S., Chamberlin, D., Fernández, M., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML Query Language. W3C Working Draft (April 2005)

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

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

[DH02] Dekeyser S., Hidders, J.: Path Locks for XML Document Collaboration. 3rd International Conference on Web Information Systems Engineering (WISE), Singapur S. 105-114 (2002)

[DHP03] Dekeyser S., Hidders, J., Pareadens J.: A Transaction Model for XML Databases. World Wide Web Journal, Volume 7, Issue 2, S. 29-57 (Juni 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)

[GBS02] Grabs, T., Böhm, K., Schek H.-J.: XMLTM: Efficient Transaction Management for XML Documents. Conference on Information and Knowledge Management (CIKM), McLean, Virginia, USA (November 2002)

[GR97] Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers (1997)

[Ha05] Haustein, M.: Eine XML-Programmierschnittstelle zur transaktionsgeschützten Kombination von DOM, SAX und XQuery. 11. GI-Fachtagung Datenbanksysteme in Business, Technologie und Web (BTW), Karlsruhe, Deutschland, S. 265-284 (März 2005)

[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)

[HH05a] Haustein, M., Härder, T.: Optimizing Concurrent XML Processing. Eingereicht 2005.

[HH05b] Haustein, M., Härder, T.: DeweyIDs—The Key to Fine-Grained Management of XML Documents. Eingereicht 2005.

[HKM03] Helmer S., Kanne C.-C., Moerkotte G.: Lock-Based Protocols for Cooperation on XML Documents. 14th International Workshop on Database on Expert System Applications (DEXA Konferenz), S. 230-244, Prag, Tschechien, (2003)

[HKM04] Helmer S., Kanne C.-C., Moerkotte G.: Evaluating Lock-Based Protocols for Cooperation on XML Documents. SIGMOD Record 33, Nummer 1, S. 58-63 (März 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)

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

[MH05] Mathis, C., Härder T.: A Query Processing Approach for XML Database Systems. 17. Workshop über Grundlagen von Datenbanken, Wörlitz, Deutschland (Mai 2005)

[Ne04] O'Neil, P., O'Neil E., Pal S., Cseri I., Schaller G., Westbury N.: ORDPATHs: Insert-Friendly XML Node Labels. ACM SIGMOD Conference Industrial Track, Paris, Frankreich (Juni 2004)

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

[XTC] Projektseite des XML Transaction Coordinator: http://www.xtc-project.de