Synchronisationsverfahren für XML-Datenbanksysteme

Felix Kling

Technische Universität Kaiserslautern
Fachbereich Informatik
Lehrgebiet Informationssysteme
D-67653 Kaiserslautern, Deutschland
f_kling@informatik.uni-kl.de

1 Einleitung

XML ist ein de facto Standard zum Datenaustausch im Internet geworden. In den letzten Jahren konnte ein starker Wachstum in der Nutzung von in XML-Dokumenten beobachtet werden und aus diesem Grund ist davon auszugehen, dass die Anzahl noch weiter steigen wird. Um eine große Zahl an XML-Dokumenten effizienter verwalten zu können, implementieren immer mehr Datenbanken eine Unterstützung für XML-Dokumente. Dabei kommen alle Eigenschaften durch die sich RDBMS und ORDBMS auszeichnen auch XML-Dokumenten zu Gute. Doch obwohl diese Techniken effizient und erfolgreich für RDBMS eingesetzt werden, können sie nicht in allen Punkten den Anforderungen an XML-Dokumente gerecht werden. Dies ist besonders für das Transaktionsmanagement der Fall. Synchronisationsverfahren, die auf relationalen Datenbanken zufriedenstellenden arbeiten, sind für semi-strukturierte Daten wie XML nur bedingt geeignet. Oft verhalten sie sich zu restriktiv, so dass ein großer Teil der möglichen Parallelität und somit der Performance eingebüßt wird. Im Folgenden werden verschiedene Anforderungen an XML-Datenbanken betrachtet, einige Synchronisationsverfahren für XML-Datenbanken vorgestellt und hinsichtlich ihrer Eigenschaften untersucht.

2 Grundlagen und Bewertungskriterien

Um die verschiedenen Verfahren angemessen vergleichen zu können, muss eine gewisse Basis festgelegt und Anforderungen bestimmt werden. XML-Dokumente können auf unterschiedliche Weise verarbeitet und dargestellt werden. Dabei bringt jede Möglichkeit Vor- und Nachteile mit sich und stellt unterschiedliche Anforderungen an die Verfahren. Zudem müssen auch die klassischen Probleme, die im Mehrbenutzerbetrieb entstehen können, von den Verfahren gelöst werden können. Dieses Kapitel beschäftigt sich damit, die Punkte festzulegen, die ein Verfahren beachten und unterstützen sollte.

2.1 Datenmodell

Das Document Object Model (DOM)[1] ist ein von der W3C[2] vorgeschlagenes Model zur Repräsentation von XML-Dokumenten. Dabei wird das komplette Dokument eingelesen und eine Baum-ähnliche Struktur aus Knoten erstellt. Ein Knoten kann in unterschiedlichen Beziehungen zu anderen Knoten stehen. Es werden 12 verschiedene Knotentypen definiert. Abbildung 1 zeigt den DOM-Baum zu Codebeispiel 1.1.

Listing 1.1: XML-Fragment

```
<uni name='TU␣KL'>
    <angestellte>
        <hiwis>
            <person id='3523' fb='Informatik' >
                <name>Kling</name>
                <vorname>Felix</vorname>
            </person>
        </hiwis>
        <professoren>
            <person id='278' fb='Biologie' >
                <name>Professor</name>
                <vorname>Muster</vorname>
            </person>
        </professoren>
    </angestellte>
</uni>
```

2.2 Verarbeitung und Zugriffsschnittstellen

Mit SAX, DOM und XQuery/XPath haben sich drei unterschiedliche Verfahren entwickelt um XML-Dokumente zu verarbeiten. Im Idealfall bietet ein XML-Datenbanksystem alle diese Schnittstellen an und kann gewährleisten, dass das eingesetzte Synchronisationsverfahren auch gleichzeitige Zugriffe über unterschiedliche Schnittstellen korrekt verarbeiten kann. Im Folgenden werden die Methoden kurz erläutert und auf ihre Besonderheiten hingewiesen.

SAX. Simple API for XML (SAX) bietet eine ereignisgesteuerte Verarbeitung eines XML Dokuments. SAX liest ein Dokumente sequentiell ein und erzeugt beim Erkennen von festgelegten Elemente ein Ereignis, das weiter verarbeitet werden kann. Diese Art des Zugriffs ist besonders speichereffizient, da das Dokument nur Schritt für Schritt verarbeitet wird und somit für die Verarbeitung nicht komplett geladen werden muss. Dies macht es möglich, auch große Dokumente effizient zu verarbeiten. Es gibt jedoch keine Möglichkeit auf bereits verarbeitete Elemente zuzugreifen. SAX bietet eine ein lesende Zugriffsmethode und erfordert darum keine explizite Konfliktbehandlung.

[1] URL: http://www.w3.org/DOM/ [09.12.2007]
[2] World Wide Web Consortium

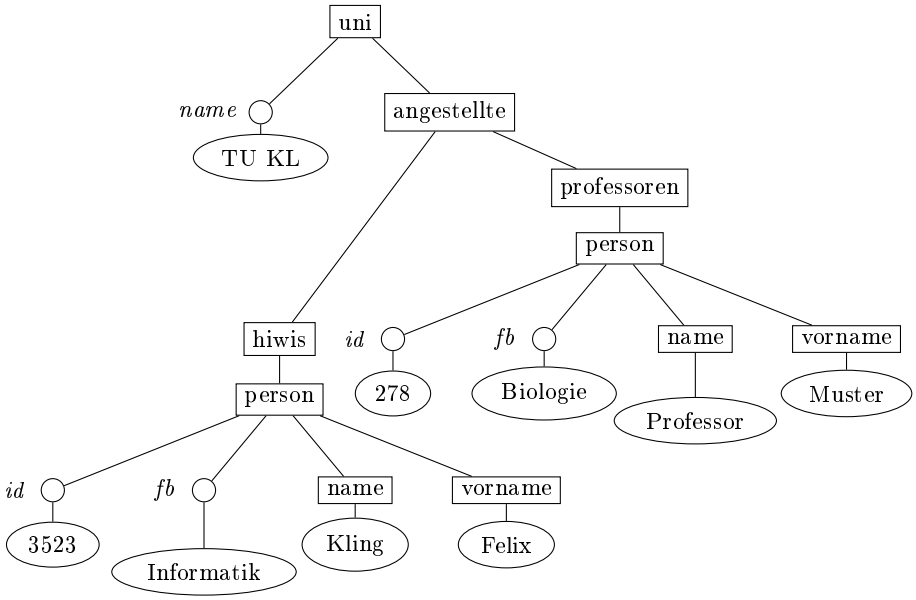


Abbildung 1: DOM-Baum zu XML-Fragment 1

DOM. Wir schon erwähnt, liest DOM zur Verarbeitung das komplette Dokumente ein und generiert daraus ein Objekt-Modell, das als Baum dargestellt wird. Das DOM-Modell definiert zusätzlich Operationen zum Navigieren über die Knoten, zum Auslesen und Modifizieren von Informationen aus den Knoten und Operationen zum Modifizieren der Baumstruktur. Da das gesamte Dokument als Modell im Speicher gehalten wird, kann, im Gegensatz zu SAX, auch auf schon verarbeitete Knoten zugegriffen werden. Allerdings verbraucht DOM dadurch mehr Speicher als SAX und kann dadurch große Dokumente eventuell nicht optimal verarbeiten.

XQuery/XPath. XQuery ist eine mächtige Verarbeitungssprache für XML. Sie bietet ähnliche Funktionen wie SQL. Dabei wird XPath von XQuery benutzt um Daten aus dem XML-Dokument deklerativ auszulesen. XPath liefert mit Hilfe so genannter *Pfadausdrücken* einzelne Elemente eines XML-Dokuments. Ein *Pfadausdruck* besteht dabei aus beliebig vielen *Lokalisierungs-Schritten*, von denen jeder die Form */Achse::Knotentest[Prädikattest]* hat. XPath definiert dabei 13 Achsen[3]: *child, descendant, attribute, self, descendent-or-self, following-sibling, following, namespace, parent, ancestor, preceding-sibling, preceding, ancestor-or-self.* Die Achsen beschreiben die relative Lage eines Knotens zu anderen Knoten und bieten eine mächtige Möglichkeit zur Anfragegestaltung. Dies muss von den Synchronisationsverfahren besonders berücksichtigt werden.

[3] URL: http://www.w3.org/TR/xpath20/ [17.01.2008]

2.3 Knotenadressierung

Um Zugriffe auf Knoten kontrollieren und synchronisieren zu können, muss jeder Knoten eindeutig identifizierbar sein. Dabei können verschiedene Anforderungen an ein Nummerierungsschema gestellt werden:

- Stabilität: Das Schema muss auch beim Einfügen oder Löschen von Knoten stabil bleiben, d.h. die Adresse eines bestehenden Knotens sollte sich dadurch nicht ändern.
- Ordnungserhaltend: Da XML-Dokumente von sich aus eine Ordnung vorgeben, muss die Nummerierung die Ordnung der Knoten widerspiegeln können, damit XML-Fragmente korrekt rekonstruiert werden können.
- Anfrageunterstüzend: Die Auswertung der XPath-Achsen sollte auf dem Index ohne Zugriff auf das Dokument möglich sein.
- Sperren: Beim Einsatz eines hierarchischen Sperrprotokolls ist es wichtig, alle Vorfahren aus der ID des betreffenden Knotens ermitteln zu können.

Ein Nummerierungsschema stellt auch die Grundlage für die Verwendung von Indexen dar. Dabei ist es von Vorteil, wenn das Schema stabil ist, da sonst jeder Index aktualisiert werden muss, wenn sich die Adresse eines darin enthaltenden Knotens ändert. Ein geeignetes Schema, das die Eigenschaften erfüllt, ist die Knotennummerierung per DeweyIDs.

DeweyIDs. bestehen aus einer eindeutigen natürlichen Zahl, die das Dokument identifiziert, gefolgt von einem Doppelpunkt und einer Folge von *Devisions*. Jede Devision (bis auf die erste) wird durch einen Punkt von der vorhergehenden Devision abgegrenzt (z.B. *3:1.3.9*). Die erste Division hinter dem Doppelpunkt adressiert somit die Dokumentwurzel und ist immer *1*. Die DeweyID jedes weiteren Knotens wird durch das Anhängen einer neuen Divison an die DeweyID seines Vaters gebildet (siehe dazu Abbildung 2). Die letzte Division ist immer eine ungerade natürliche Zahl und innerhalb einer Teilbaumebene bestimmt sie die Position des Knotens in dieser Ebene. Mit Hilfe ungerade Devisions können zwischen zwei DeweyIDs beliebig viele DeweyIDs eingefügt werden. Zum Beispiel kann zwischen *3:1.3.7* und *3:1.3.9* die ID *3:1.3.8.3* eingefügt werden.
Bei der initialen Vergabe der IDs, kann mit einem *Distance*-Parameter (*distance*) der Abstand zwischen den Adressen beeinflusst werden. Dabei ist *distance* eine gerade natürliche Zahl. Die Vergabe der IDs folgt dann wie folgt ab:

1. Die Dokumentwurzel bekommt als einzige Devision die *1*.
2. Die DeweyID des ersten Knotens einer Teilbaumebene bekommt die ID seines Vaters mit angehängtem *distance + 1*. Jeder weitere Knoten einer Teilbaumebene bekommt die ID seines Vorgängers auf dieser Ebene, wobei die letzte Devision um *distance* erhöht wird.
3. Attributknoten erhalten die ID ihres Vaters, wobei ein *1* angehängt wird. Daran werden ganz normal ungerade Devisions gehängt, also zum Beispiel *3:1.5.1.3* und *3:1.5.1.5*. Da die Reihenfolge der Attribute keine Rolle spielt, muss der *distance*-Parameter nicht berücksichtigt werden.

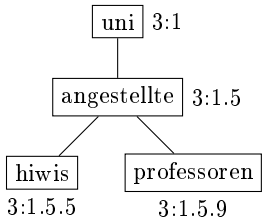


Abbildung 2: Initiale Vergabe der DeweyIDs. Die Dokument ID ist 7, *distance* = 4

Beim Einfügen von neuen Knoten müssen drei Fälle unterschieden werden (angenommen wird ein *distance*-Wert von 4):

Hinter dem letzten Knoten einer Teilbaumebene: Ist die Devision vor der letzten Devision des vorangehenden Kontens der Teilbaumebene ungerade, so wird die letzte Devision um *distance* erhöht (ID des vorangehenden Knotens: *3:1.5*, ID des neuen Knotens: *3:1.9*). Ist die Devision gerade, werden alle geraden Devisions bis auf die letzte gerade entfernt. Die letzte Devision wird dann um *distance - 1* erhöht (ID des vorangehenden Knotens: *3:1.10.4.3*, ID des neuen Knotens: *3:1.5.13*). In Abbildung 3a ist DOM-Baum nach dem Einfügen dargestellt.

Vor dem erstem Knoten einer Teilbaumebene: Ist die Devision vor der letzten Devision des ersten Knotens ungerade, so wird die letzte Devison zur Berechnung benutzt, ansonsten die erste gerade Devision vor der letzten Devision. Die ausgewählte Devision wird dann halbiert und entweder aufgerundet oder um *1* erhöht, damit sie wieder ungerade ist (ID des ersten Knotens: *3:1.4.2.3*, ID des neuen Knotens: *3:1.3*). In Abbildung 3a ist DOM-Baum nach dem Einfügen dargestellt. Da *1* für die DeweyIDs von Attributknoten reserviert ist, muss darauf geachtet werden, ob die letzte Devison der ID des erste Knoten in der Teilbaumebene *3* ist. Ist dies der Fall, wie die *3* durch *2.distance + 1* ersetzt. Lautet die ID des ersten Knotens 3:1.3 so ist die ID des neuen Knotens *3:1.2.5* (siehe Abbildung 3b).

Zwischen zwei Knoten: Soll zwischen zwei existierenden Knoten k_1 und k_3 ein dritter Knoten k_2 eingefügt werden, so wird der ungerade Mittelwert zwischen den beiden Devisions ermittelt, in denen sich k_1 und k_3 als erstes unterscheiden. Sei ID_{k_1} = 3:1.5.5.4.9 und ID_{k_3} = 3:1.5.5.9, dann beträgt die ID des neuen Knotens ID_{k_2} = 3:1.5.5.7. Existiert zwischen den beiden Devisions keine ungerade Zahl, so wird die neue ID mit ungeraden Devisions gebildet (zum Beispiel ID_{k_1} = *3:1.5.5*, ID_{k_3} = 3:1.5.7 $\Rightarrow$ ID_{k_2} = 3:1.5.6.$distance + 1$) (siehe Abbildung 3).

Die Nummerierung mit DeweyIDs erlaubt die Berechnung der Lage zweier Knoten bezüglich aller XQuery-Achsen. Eine ausführliche Erläuterung findet sich in [1]. [2] behandelt weiterführende Aspekte der DeweyIDs.

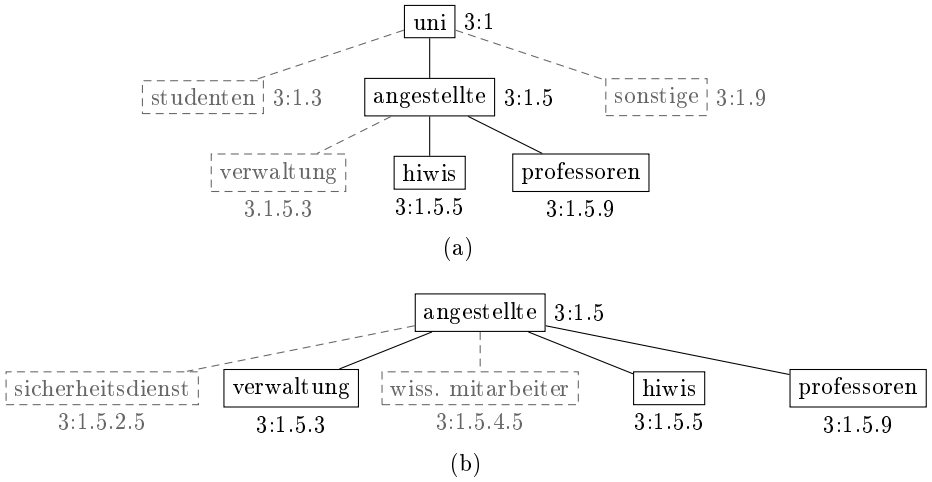


Abbildung 3: Einfügen von Knoten

2.4 Mehrbenutzerbetrieb

Anomalien. Arbeiten mehrere Transaktionen parallel auf denselben Daten, kann es ohne besonderer Kontrolle in den Transaktionen zu Fehlern, die Anomalien genannt werden, kommen. Ziel der Synchronisationverfahren ist es, diesen Anomalien vorzubeugen. [3] beschreibt die wichtigsten Anomalien, die auftreten können:

- Verlorengegangene Änderungen (lost update)
- Zugriff auf schmutzige Daten (dirty read, dirty write)
- Nicht-wiederholbares Lesen (non-repeatable read)
- Phantome

Serialisierbarkeit. Die Synchronisation von Transaktionen gilt als korrekt, wenn die Abfolge der Transaktionen zum gleichen Ergebnis kommt, wie eine serielle Abfolge der Transaktionen. Das Konzept der Serialisierbarkeit wird hier nicht nähere betrachtet. Weitere Informationen finden sich in [3,4].

Konsistenzebenen. Durch Vorkehrungen zur Vermeidung von Anomalien, können sich die Transaktionen gegenseitig so beeinflussen, dass die Performance eingeschränkt wird. Um eine höhere Performance unter Inkaufnahme von Anomalien zuzulassen, wurden im SQL92-Standard vier Konsistenzebenen festgelegt. Tabelle 1 zeigt, welche Anomalien in der jeweiligen Konsistenzebene erlaubt sind (vgl. auch [3]). Diese Konsistenzebenen lassen sich auch als Kriterium für Synchronisationsverfahren für XML-Dokumente heranziehen. Dabei sollte ein brauchbares Verfahren auch alle Ebenen unterstützen, um ausreichend flexibel auf eine Transaktion anwendbar zu sein. Allerdings wird an diesem Standard

Ebene	Anomalie		
	dirty read	non-repeatable read	phantomes
read uncommited	möglich	möglich	möglich
read commited	nicht möglich	möglich	möglich
repeatable read	nicht möglich	nicht möglich	möglich
serializable	nicht möglich	nicht möglich	nicht möglich

Tabelle 1: Konsistenzebenen in SQL92

auch kritisiert, dass nicht alle Anomalien erfasst werden. Es fehlen zum Beispiel *dirty write* und *write skrew*. In [5] wird *snapshot isolation* als weitere Konsistenzebene erwähnt, die in Mehr-Versionen-Verfahren ihre Verwendung findet. Auf dieser Ebene sind zwar Phantome möglich, jedoch nicht in der Weise, wie sie in SQL-92 definiert sind. Weiterführende Informationen dazu finden sich ebenfalls in [5].

3 Synchronisationsverfahren

In dieser Arbeit werden folgende Synchronisationsverfahren vorgestellt.

- *2PL (Abschnitt 4)
- taDOM (Abschnitt 8)
- XPath (Abschnitt 5)
- XGDL (Abschnitt 7)
- OptiX (Abschnitt 6)
- SnaX (Abschnitt 6)

4 *2PL

Die Verfahren Doc2PL, Node2PL, NO2PL und OO2PL werden von Sven Helmer, Carl-Christian Kanne und Guido Moerkotte in [6] vorgestellt. Alle diese Verfahren sind Sperrverfahren, die sich in den Objekten, auf denen Sperren angefordert werden, unterscheiden. Ihre größte Aufmerksamkeit liegt dabei in der Kontrolle von nebenläufigen Navigationsoperationen.

4.1 Datenmodell

Es wird davon ausgegangen, dass auf die verwalteten XML-Dokumente über DOM-ähnliche APIs[4] zugegriffen wird. Darum werden XML-Dokumente in einer ähnlichen Weise repräsentiert, wie es von DOM vorgeschlagen wird. Navigations- und Modifikationsoperationen werden übernommen. Von einer Unterscheidung der Knotentypen wird abgesehen, da die Typen keinen Einfluss auf Synchronisation oder Sperren haben. In [6] wird sich zur Darstellung der Funktionsweise der Sperrverfahren auf folgende Operationen beschränkt:

[4] engl. Application Programming Interface

sd: Dieser Operationen wählt das Dokument aus. Dabei wird der Wurzelknoten des XML-Dokuments zurückgeliefert.
nthP(n): Liefert das n-te Kind eines Kontextknotens[5].
nthM(n): Liefert das n-te Kind, ausgehend vom letzten Kind.
insA(n): Fügt einen neuen Knoten hinter dem Kontextknoten ein.
insB(n): Fügt vor dem Kontextknoten einen neuen Knoten ein.
del(n): Löscht den Kontextknoten und damit alle seine Nachfolger.

Zum Zugriff auf das XML-Dokument werden ausschließlich DOM-Operationen benutzt. Weiterhin wird in [6] vorausgesetzt, dass der Zugriff auf das Dokument stets über die Dokumentwurzel erfolgt, d.h. jede Transaktion muss zuerst den Wurzelknoten anfordern, um von diesem zu dem gewünschten Knoten zu navigieren. Diese Einschränkung wird im Zusammenhang mit ID/IDREF-Beziehungen teilweise aufgehoben (vgl. Abschnitt 4.5).

4.2 Sperrmodi

Wie beim Zwei-Phase-Sperr-Protokoll werden für Lese- und Schreiboperationen auf den Dokumentdaten S- und X-Sperren mit ihrer bekannten Sperrmatrix verwendet. Die Neuerung besteht in der Einführung von Sperrmodi, die das Navigieren über bestimmte Pfade und Modifikationen an der Dokumentstruktur kontrollieren. Dies ermöglicht eine Unterscheidung, ob eine Transaktionen einen Knoten verarbeitet, oder ob sie über diesen zu weiteren Knoten navigiert. Tabelle 2a zeigt die dazugehörige Sperrmatrix. Dabei ist T der *shared-lock*-Modus, der angefordert werden muss, um durch das Dokument zu navigieren, und M der *exclusive-lock*-Modus, der erforderlich ist, um Modifikationen an der Dokumentstruktur durchzuführen. Für welche Objekte die Sperren angefordert werden können, ist der Beschreibung des jeweiligen Verfahrens in Abschnitt 4.3 zu entnehmen.
Die Sperren werden nach den Regeln der Zweiphasigkeit angefordert. Kann eine Sperre nicht erteilt werden, weil das entsprechende Objekt schon gesperrt ist, wird die anfordernde Transaktion geblockt und in einen Wartegraph eingereiht. Wird der Graph zyklisch, wird die letzte Transaktion abgebrochen.

4.3 Sperrgranulate

Wie schon erwähnt, unterscheiden sich die *2PL-Sperrverfahren in den Objekten, auf denen die Sperren angefordert werden. Dies ist in Abbildung 4.3 nochmals verdeutlicht.

Doc2PL: Das einfachste Verfahren sperrt das gesamte Dokument. Da mit diesem Verfahren keine Nebenläufigkeit möglich ist, ist es für die weitere Betrachtung nicht interessant.

[5] Kontextknoten bezeichnet immer den Knoten, der gerade verarbeitet wird bzw. von dem eine Operation ausgeht

	T	M
T	+	-
M	-	-

(a)

	TL	TR	TA	TZ	ML	MR	MA	MZ
TL	+	+	+	+	-	+	+	+
TR	+	+	+	+	+	-	+	+
TA	+	+	+	+	+	+	-	+
TZ	+	+	+	+	+	+	+	-
ML	-	+	+	+	-	+	+	+
MR	+	-	+	+	+	-	+	+
MA	+	+	-	+	+	+	-	+
MZ	+	+	+	-	+	+	+	-

(b)

Tabelle 2: Sperrmatrizen von Node2PL, NO2PL und OO2PL

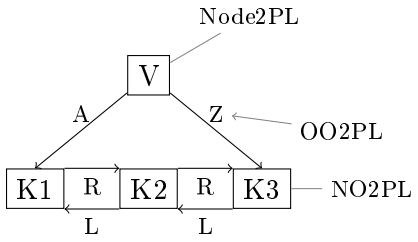


Abbildung 4: Sperrgranularität von Node2PL, NO2PL und OO2PL

Node2PL: Möchte eine Transaktion auf einen bestimmten Knoten zugreifen, so muss sie die entsprechende Sperre auf dem Vaterknoten anfordern. Soll zu einem beliebigen Kind eines Knotens n navigiert werden, so muss für n eine T-Sperre angefordert werden. Wird ein neues Kind unter n einfügt, muss eine M-Sperre auf n angefordert werden.

NO2PL: Dieses Verfahren sperrt die Knoten, deren Kanten traversiert bzw. modifiziert werden. Wird ein neuer Knoten eingefügt, so muss dieser nicht gesperrt werden, da alle Wege, die zu diesem Knoten führen (über den Eltern- und/oder Geschwisterknoten), gesperrt sind.

OO2PL: Im Unterschied zu Node2PL und NO2PL sperrt OO2PL nicht Knoten, sondern Kanten. Jeder Knoten hat maximal vier Kanten: *firstChild (A)*, *lastChild (B)*, *leftSibling (L)* und *rightSibling (R)*. Dementsprechend gibt es vier *shared (TA, TB, TL, TR)* und vier *exclusive locks (MA, MB, ML, MR)*. Die Sperrmatrix ist in Tabelle 2b zu sehen.

4.4 Inhaltssperren

Um die Verfahren zu vervollständigen muss auch das Auslesen und Modifizieren von Knoteninhalten gesperrt werden. Dafür werden S- und X-Sperren verwendet. Die Matrix für Node2PL und NO2PL ist in Tabelle 3 zu sehen. Für OO2PL kann solche eine Matrix nicht aufgestellt werden, da OO2PL auf Kanten sperrt.

Trotzdem sind die T^*-Sperren implizit kompatibel zu den S- und X-Sperren, M^*-Sperren dagegen nicht.

	S	X	T	M
S	+	-	+	-
X	-	-	+	-
T	+	+	+	-
M	-	-	-	-

Tabelle 3: Vollständige Sperrmatrix für Node2PL und NO2PL

4.5 Umgang mit ID/IDREF Beziehung

Wie anfangs erwähnt, wird bei den Verfahren davon ausgegangen, dass der Zugriff stets über die Dokumentwurzel erfolgt. Dies kann jedoch nicht mehr gefordert werden, wenn ID/IDREF-Beziehungen unterstützt werden sollen. Jedoch kann mit den bisher vorgestellten Sperren keine Serialisierbarkeit gewährleistet werden, wenn beliebige Sprünge im Dokument möglich sind. Aus diesem Grund werden zwei neue Sperrmodi eingeführt: *IDS* und *IDX*. Möchte eine Transaktion zu einem ID/IDREF-Link folgen, so muss sie zuerst eine *IDS*-Sperre anfordern. Soll ein beliebiger Knoten n gelöscht werden, wird für alle Nachfolger, die ein ID-Attribut besitzen, eine *IDX*-Sperre angefordert, damit keine andere Transaktion zu einem dieser Nachfolger springen kann.

4.6 Bessere Nebenläufigkeit mit Hilfe einer Schemadefinition

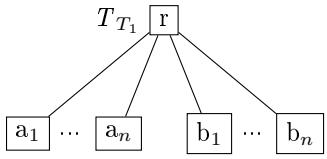
Existiert zu einem Dokument eine Schemdafinition (z.B DTD[6]), so kann diese zur Verbesserung der Nebenläufigkeit herangezogen werden.

Beispiel I. Für einen Knoten r definiert die DTD *a*b** als Kinder. *first(n)* und *last(n)* geben das erste bzw. letzte Kind vom Typ n eines Knotens zurück. Normalerweise würde folgende Operationsfolge geblockt werden (x ist von Typ b, y ist von Typ a):

T_1	T_2
$first(b)$	
	$last(a)$
$insB(x)$	
	$insA(y)$

Mit Hilfe der DTD ist es jedoch die Unterscheidung der Knotentypen möglich und somit die Ausführung der Operationen zu erlauben.

[6] engl. Document Type Definition



4.7 Bewertung

Die vier verschiedenen Verfahren garantieren Serialisierbarkeit, allerdings auf Kosten unterschiedlich hoher Blockierungsraten. Da Doc2PL keine Nebenläufigkeit zulässt, wird es nicht in der Bewertung berücksichtigt. Node2PL und NO2PL sperren beide auf Knoten, allerdings ist der Lock-Overhead bei NO2PL größer. Diese beiden Verfahren haben jedoch den Nachteil, dass Teilbäume bei Strukturveränderungen für den Zugriff von oben komplett gesperrt sind (bei NO2PL nur, wenn am Anfang oder Ende einer Teilbaumebene eingefügt wird). Möchte, wie in Abbildung 5 dargestellt, eine Transaktion T_1 unter dem Knoten *angestellte* ein neues Kind einfügen, so wird zuerst *uni* im T-Modus gesperrt und *angestellte* im M-Modus. Dies bedeutet, dass keine weitere Transaktion auf *hiwis* und *professoren* zugreifen kann. In komplexen Dokumenten mit hohen Bäumen bzw. bei Einfügungen in den obersten Ebenen werden somit große Teile des Dokumentes unnötigerweise gesperrt. Die Verfahren sperren den Baum zu restriktiv.

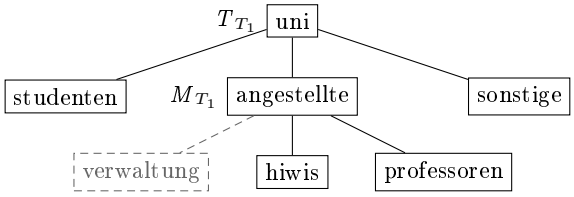
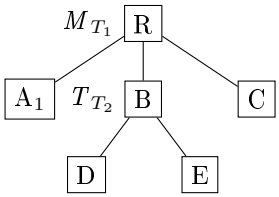


Abbildung 5: Sperrverteilung bei Einfügen von neuen Knoten mit Node2PL

Im Gegensatz dazu sperrt OO2PL auf Kanten. Bezogen auf das Beispiel bedeutet dies, dass nur die Kanten *firstSibling* des Knotens *angstellte* und *leftChild* des Knotens *hiwis* gesperrt werden. Die Knoten selbst bleiben zugreifbar und über die nicht gesperrten Kanten lässt sich tiefer in den Baum navigieren. Die höhere Parallelität wird allerdings durch einen komplexeren Sperrmechanismus erkauft. Trotzdem erfüllt dieses Verfahren am ehesten die Anforderungen eines praktikablen Sperrverfahren für XML-Dokumente.
Auf die Verwendung von Indexen zur Verbesserung der Zugriffszeit wir nicht eingegangen. Kontrolle von direkten Einsprüngen in ein Dokument werden, abgesehen von den ID/IDREF-Beziehungen, nicht gemacht. Eine Transaktion könnte über einen Index an eine beliebige Stelle eines Teilbaums springen (außer auf die

Wurzel des Teilbaums bzw. dessen Kinder), obwohl dieser Teilbaum von einer anderen Transaktion gelöscht wird (vgl. Beispiel II).

Beispiel II. Als Sperrprotokoll wird *Node2PL* angenommen. Das Problem besteht jedoch bei allen *2PL-Protokollen. T_1 navigiert zu Knoten B und möchte diesen Löschen. Dazu muss T_1 eine M-Sperre auf R anfordern. T_2 springt über einen Index auf D und fordert dazu eine T-Sperre auf B an, die gewährt wird.



Da sich die Autoren nur auf die DOM-Spezifikation berufen, kann keinerlei Aussage gemacht werden, wie sich das Sperrverfahren bei XQuery/XPath Anfragen verhält. Wenn davon ausgegangen wird, dass XPath-Pfadausdrücke über Indexe ausgewertet werden, so können die *2PL-Verfahren für solche Zugriffe keine Serialsierbarkeit gewährleisten.

Kriterium		Bemerkung
Datenmodell	√	
Schnittstellen	X	Keine Angaben zum Verhalten bei deklarativen Anfragen
Knotenadressierung	?	Die Art der Sperren gestattet keine Nutzung von Indexen
Serialisierbarkeit	√	

5 Pfadsperren - XPath

In diesem Abschnitt wird ein Sperrverfahren beschrieben, das sich vornehmlich auf die Unterstützung von XPath konzentriert. Es in [7] von Stjin Dekeyser und Jan Hidders vorgestellt wird. Ausgehend von der Annahme, dass auf XML-Dokumente meistens mit XPath-Ausdrücken zugegriffen wird, dienen vereinfachte Pfadausdrücke als Pfadsperren.

5.1 Datenmodell

Als Datenmodell wird eine vereinfachte Form des XPath-Datenmodells angenommen. Dabei wird die Darstellung eines XML-Dokuments allerdings nicht als

Baum beschrieben, sondern allgemeiner als Graph. Es wird nicht zwischen verschiedenen Knotentypen unterschieden und die Reihenfolge der Knoten einer Teilbaumebene wird ebenfalls nicht beachtet. Knoten können mit Transaktionsidentifikatoren markiert werden, um auszudrücken, dass diese Knoten von den Transaktionen gelöscht wurden. Jeder Knoten, dem keine Transaktionsidentifikation zugeordnet ist, gehört zur *aktuellen Instanz* des Graphen. Knoten erhalten eine eindeutige ID und den entsprechenden Elementnamen als Markierung, wie in Abbildung 6 dargestellt.

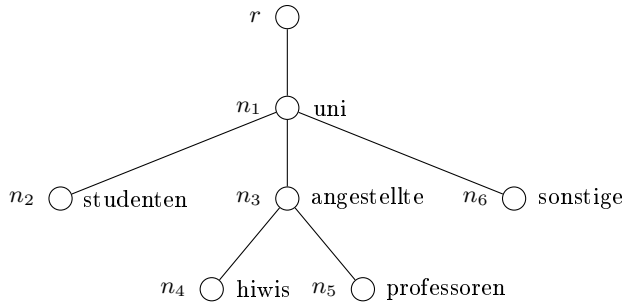


Abbildung 6: XPath-Graph

Die Anfragesprache ist ebenfalls eine vereinfachte Form der XPath-Pfadausdrücke und ist wie folgt definiert:

$$P ::= F|P/F|P//F$$
$$F ::= E|*$$

Dabei ist E die Menge der Knotennamen. Pfadausdrücke starten entweder in der Wurzel oder in Knoten die während der Transaktion angefordert wurden.
Da XPath keine Möglichkeit bietet, Modifikationen auf Dokumenten vorzunehmen, werden zusätzlich zwei Operationen zum Einfügen und löschen von Knoten eingeführt:

A(n, a): Fügt einen neuen Knoten a als Kind von n ein.
D(n): Markiert den Knoten n mit der entsprechenden Transkations-ID um auszudrücken, dass dieser gelöscht ist.
Q(n, p): Liefert alle Knoten zurück, die ausgehend von dem Knoten n den Pfadausdruck p erfüllen.

Die Änderungsoperationen schlagen fehl, wenn die neue Instanz des Graphen kein Baum mehr ist (vgl. Beispiel Beispiel III).

Beispiel III. Die Operation $D(n_3)$ markiert *angestellte* mit der entsprechenden Transaktions-ID. Die neue Instanz des Graphen bilden somit alle Knoten ohne *angestellte*. Da die Knoten *hiwis* und *professoren* (n_4,n_5) aber nicht mehr mit der Wurzel verbunden sind, schlägt diese Operation fehl.

5.2 Pfadsperren

Dekeyser und Hidders stellen zwei Verfahren für Pfadsperren vor: *path lock propagation* und *path lock satisfiability*. Beide Verfahren nutzten die gleichen Sperren, sie unterscheiden sich in den Auswirkungen, die die Anforderung einer Sperre nach sich zieht. Lese- und Schreibsperren sind wie folgt definiert:

Lesesperre: Eine Lesesperre besteht aus einem Tupel $rl(t, n, p)$, wobei r die anfordernde Transaktion identifiziert, n den Knoten im Instanz-Graphen, auf den die Sperre angewendet werden soll und der Ausgang des Pfadausdrucks $p \in P$ ist.

Schreibsperre: Das Tupel $wl(t, n, f)$ definiert eine Schreibsperre der Transaktion t auf dem Knoten n mit dem Ausdruck $f \in F$.

Path Lock Propagtion. Für eine Anfrageoperation $Q(n, p)$ wird zuerst ein *initial read lock* $rl(t, n, p)$ angefordert. Ausgehend von dieser Lesesperre werden abhängig von dem Pfadausdruck p weitere Lesesperren auf den Nachfolgerknoten von n angefordert. Beispiel IV zeigt ein Beispiel wie diese Sperren angefordert werden. Die genauen Regeln zur Sperrenverteilung findet sich in [7]. Für eine Änderungsoperation $A(n, p)$ wird eine Schreibsperre $wl(t, n, a)$ angefordert, wohingegen für eine Löschoperation $D(n)$ zwei Schreibsperren angefordert werden: $wl(t, n, *)$ und $wl(t, n', a)$. Dabei ist n' der Vater n und a ein Label von n. Folgende Sperren sind nicht zueinander kompatibel: $rl(t, n, a)$ bzw. $rl(t, n, *)$ zu $wl(t', n, a)$ und $wl(t', n, *)$, wenn $t \neq t'$.

Beispiel IV. Für die Aktionen $a_1(Q(r, uni/studenten), t_1)$ und $a_2(D(n_6), t_2)$ werden folgende Lese-Sperren angefordert:
$rl(t_1, r, uni/studenten)$
$rl(t_1, n_1, studenten)$
$wl(t_2, n_1, sonstige)$
$wl(t_2, n_6, *)$

Path Lock Satifiability. Im Gegensatz zum *path-lock-propagation*-Schema werden beim *path-lock-satifiability*-Schema weniger Sperren verwendet, allerdings ist damit das Prüfen auf Konflikte aufwendiger. Schreibsperren werden wie beim *path lock propagation scheme* vergeben, für Anfrageoperationen wird allerdings nur der *initial read lock* angefordert, d.h. es werden keine weiteren Sperren auf den Nachfolgern angefordert. Beispiel V zeigt die entsprechenden Sperranforderungen. Die Konfliktbedingungen sind wie folgt definiert:
Eine Lesesperre $rl(t, n, p)$ ist zu einer Schreibsperre $wl(t', n', f)$ inkompatibel, wenn gilt:

1. $t \neq t'$
2. n ist ein Vorgänger von n'
3. $\overline{\lambda}(n, n')/f \subseteq L(p)$, dabei ist $\overline{\lambda}(n, n')$ der Pfad von n nach n' und $L(p)$ die Menge aller Pfade die durch p bestimmt wird.

Beispiel V. Für die Aktionen $a_1(Q(r, uni/studenten), t_1)$ und $a_2(D(n_6), t_2)$ werden folgende Sperren angefordert:
$rl(t_1, r, uni/studenten)$
$wl(t_2, n_1, sonstige)$
$wl(t_2, n_6, *)$
Hier treten keine Konflikte auf, da
- $\overline{\lambda}(r, n_1)/sonstige \notin L(uni/studenten)$ und
- $\overline{\lambda}(r, n_6)/* \notin L(uni/studenten)$

5.3 Bewertung

Die Definition der Löschoperation lässt nur das Löschen von Blattknoten zu. Da immer nur der Knoten und nicht der gesamte Teilbaum gelöscht wird, wäre der resultierende Graph bei einer Löschung eines beliebigen inneren Knotens kein Baum mehr. Dies kann die Löschung eines Teilbaumes sehr aufwendig machen, da jeder Knoten explizit gelöscht werden muss.
Da zur Sperrverwaltung und Anfrage eine vereinfachte Form von XPath verwendet wird, kann XPath nicht in vollem Umfang unterstützt werden. Neben der fehlenden Prädikatunterstützung, kann nur die *descendant*-Achse zur Anfrageformulierung benutzt werden. Des Weiteren wird die Reihenfolge von Knoten einer Teilbaumebene nicht berücksichtigt. XML-Dokumente in denen die Reihenfolge der Knoten von Bedeutung ist, werden dadurch nicht optimal unterstützt. Dies kann jedoch mit ordnungserhaltenden Nummmerierung der Knoten und einer Verfeinerung der Einfügeoperation erreicht werden.
Wie und ob ein navigierender Zugriff auf ein Dokument unterstützt wird, wird nicht erläutert.

Kriterium		Bemerkung
Datenmodell	X	Nur Blattknoten können gelöscht werden.
Schnittstellen	X	Keine Angaben zum Verhalten bei navigierenden Anfragen
Knotenadressierung	?	Keine Vorgaben
Serialisierbarkeit	√	

6 OptiX & SnaX

OptiX und SnaX sind optimistische Verfahren und verwenden Versionen um Nebenläufigkeit zu gewährleisten. Beide Verfahren werden in [8] von Zeeshan Sardar und Bettina Kemme erläutert. Durch den Einsatz von Versionen kann auf ein komplexes Sperrsystem verzichtet werden und dadurch je nach Transaktion ein höherer Parallelitätsgrad erzielt werden. Schon bei relationalen Datenbanksystemen können Versionsverfahren bei häufigen Leseoperationen eine höhere Performance liefern als hierarchische Sperrverfahren.

6.1 Datenmodell

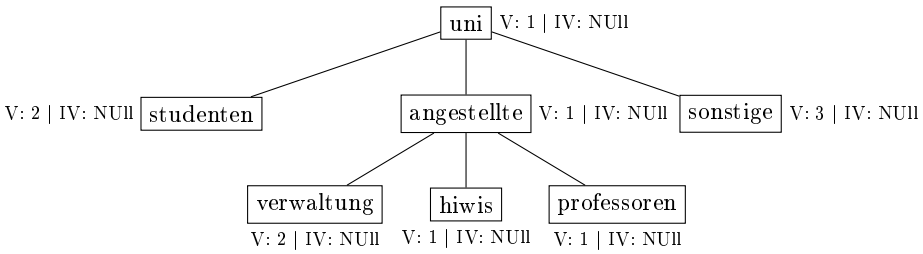
Diese Verfahren legen ebenfalls das XPath-Datenmodell zugrunde und unterscheiden nicht zwischen den einzelnen Knotentypen. XQuery/XPath wird als Anfragesprache angenommen.

6.2 Versionen

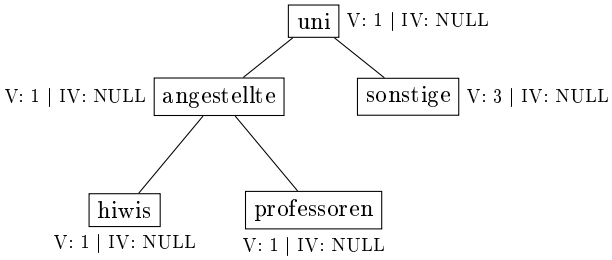
Bei beiden Verfahren ist es nicht notwendig Lesesperren zu setzen. Greift eine Transaktion lesend auf einen Knoten zu, so sieht die Transaktion die jüngste freigegebene Version dieses Knotens. Dies wird mit Hilfe von *Timestamps* gewährleistet. Jede Transaktion T_i bekommt bei ihrem Start einen Zeitstempel $ID(T_i)$. Zusätzlich wird eine Hilfsstruktur $EB(T_i)$ eingerichtet. $EB(T_i)$ ist die Menge aller $ID(T_j)$, die vor T_i beendet wurden. $ID(T_i)$ ist ebenfalls in dieser Menge enthalten. Damit kann kontrolliert werden, dass T_i nur auf Versionen von beendeten Transaktionen zugreift. Es wird nicht näher erläutert, wie diese Menge verwaltet wird. Jedem Knoten p werden zwei Zeitstempel zugeordnet: $V(p)$ und $IV(p)$. $V(p)$ ist der *valid timestamp* und identifiziert die Transaktion, die den Knoten erstellt hat. $IV(p)$ (*invalid timestamp*) bezeichnet dementsprechend die Transaktion, die den Knoten p als ungültig markiert hat. Ist p nicht ungültig, gilt $IV(p) = NULL$.
Damit eine Transaktion T_i auf einen Knoten p zugreifen darf, muss folgendes gelten: $V(p) \in EB(T_i) \wedge IV(p) \notin EB(T_i)$.

Beispiel VI. Die Transaktionen T_1 erstellt die Knoten, die in Abbildung 7a mit *V: 1* markiert sind. Eine Transaktion T_2 startet nachdem T_1 beendet ist und legt die Knoten *studenten* und *verwaltung* an. Währenddessen startet eine dritte Transaktion T_3 und fügt den Knoten *sonstige* ein. Da T_2 noch

läuft, sieht T_3 deren Änderungen nicht, sondern hat nur die Sicht auf das XML-Dokument, die in Abbildung 7b dargestellt ist.



(a) Vollständiges XML-Dokument



(b) XML-Dokument aus Sicht von T_3

Abbildung 7: Verschiedene Sichten auf das XML-Dokument

Wenn eine Transaktion T_i einen Teilbaum ändert bzw. einfügt, bieten sich mehrere Möglichkeiten an, die Zeitstempel zu verteilen.

1. Jedem neuen oder geänderten Knoten wird $V(p) = T_i$ bzw. $IV(p) = T_i$ zugewiesen.
2. Nur der Wurzel p des Teilbaums werden $V(p) = T_i$ bzw. $IV(p) = T_i$ zugeordnet und alle Kinder $V(p) = NULL$ bzw. $IV(p) = NULL$

Beide Möglichkeiten bieten Vor- und Nachteile: Wird jedem Knoten der entsprechende Zeitstempel zugeordnet, erhöht sich die Ausführungsdauer der Anfrage. Allerdings kann so für jeden Knoten sofort festgestellt werden, ob er gültig oder ungültig ist. Dies ist besonders von Vorteil, wenn über Indexe an eine beliebige Stelle in dem Dokumente gesprungen werden kann. Würden nur die Wurzelknoten der betroffenen Teilbäume mit Zeitstempeln markiert werden und zu einem Knoten q mit $V(q) = NULL$ oder $IV(q) = NULL$ gesprungen werden, so müssten zuerst alle Vorfahren des Knotens q untersucht werden, ob der entsprechende Zeitstempel gesetzt ist. Werden nun keine Einsprünge zugelassen, sondern das

Dokument immer von oben nach unten verarbeitet, so bietet die zweite Möglichkeit eine schnellere Bearbeitung der Anfrage, da weniger Knoten markiert werden müssen. Sardar und Kemme gehen in ihren Beispielen davon aus, dass dies der Fall ist.

6.3 OptiX: Optimistic Concurrency Control

Die Verarbeitung einer Transaktion wird in drei Phasen eingeteilt: *working phase*, *validation phase* und *update phase*. In der *working phase* werden die Versionen wie in Abschnitt 6.2 ausgewählt. Jede Transaktion T_i hält Informationen über die gelesenen bzw. geänderten Knoten in einem *Read-Set* $RS(T_i)$ bzw. einem *Write Set* $WS(T_i)$. Bevor die Transaktion T_i beendet werden kann, werden in der *validation phase* *Read-* und *Write-Set* gegen die *Sets* konkurrierender validierter Transaktionen T_j geprüft (d.h. T_i startete bevor T_j committed hat). T_i wird zurückgesetzt, wenn die Schnittmenge $WS(T_j) \cap RS(T_i)$ nicht leer ist. Ist die Validierung erfolgreich, werden die Änderungen festgeschrieben und neue Versionen der geänderten Knoten erstellt.

XML-Anpassungen. Um die Struktur von XML-Dokumenten besser zu Unterstützen, werden *Read-* und *Write-Sets* weiter verfeinert:

Read Sets

RR(T_i): Enthält die Wurzelknoten der zurückgelieferten Teilbäume.
ER(T_i): Enthält alle Knoten, die explizit gelesen werden. Dazu gehören alle Knoten, die einen Pfadausdruck oder eine Bedingung erfüllen.
$\mathbf{ER}_A$(T_i): In dieser Menge sind alle Knoten enthalten, hinter denen ein Knoten eingefügt wird. Diese Menge wird zusammen mit dem entsprechenden *Write-Set* ($\mathrm{I}_A(T_j)$) einer anderen Transaktion T_j verwendet, um die korrekte Ordnung von Knoten zu gewährleisten, wenn die Reihenfolge wichtig ist (z.B. bei nach dem Alphabet sortieren Knoten).

Write Sets

D(T_i): Enthält alle Knoten, die von T_i gelöscht oder ersetzt werden. Werden Teilbäume gelöscht oder ersetzt, enthält die Menge nur den Wurzelknoten des Baums.
$\mathbf{R}_n$(T_i): Enthält alle Knoten, die umbenannt wurden.
I(T_i): Diese Menge enthält die Elternknoten der Knoten, die von T_i eingefügt worden sind.
$\mathbf{I}_A$(T_i): In dieser Menge sind die gleichen Knoten wie in $\mathrm{ER}_A(T_i)$ enthalten.

Wenn eine Transaktion T_i nun gegen eine validierte Transaktion T_j geprüft wird und ein Knoten n sowohl in *RS(T_i)* als auch in *WS(T_j)* enthalten ist, kann mit Hilfe der verschiedenen *Write-* und *Read-Sets* eine genauere Überprüfung auf Konflikte vorgenommen werden. Vorgänger/Nachfolger-Beziehung müssen

dabei gesondert geprüft werden. Es ist beispielsweise bisher nicht möglich zu prüfen, ob T_i einen Knoten aus einem Teilbaumes liest, der von T_j gelöscht wurde. Aus diesem Grund muss ohne großen Aufwand ermittelbar sein, ob ein Knoten Vorgänger eines anderen ist, und umgekehrt. Dies ist z.B. mit einer entsprechenden Knotennummerierung wie den *DeweyIds* der Fall. Tabelle 4 zeigt, wann eine Validierung erfolgreich ($\surd$) ist. Dabei ist q ein Nachfolger von n.

T_i \ T_j		n ∈			q ∈
	D	R_n	I	I_A	WS
n ∈ RR	-	-	-	$\surd$	-
n ∈ ER	-	-	$\surd$	$\surd$	$\surd$
n ∈ ER_A	-	-	$\surd$	-	$\surd$
q ∈ RS	-	$\surd$	$\surd$	$\surd$	

Tabelle 4: Konfliktmatrix für OptiX

Beispiel VII. In diesem Beispiel wird gezeigt, welche Knoten in die *Read-* und *Write-Sites* aufgenommen werden. Dabei werden XQuery *FLWR-Ausdrücke* zur Anfrage und Modifikation verwendet.

- `FOR $a IN /uni/angestellte`
  `RETURN $a`
  - *RR(T_i)* enthält $a
  - *ER(T_i)* enthält $a und alle seine Vorgänger, da sie den Pfadausdruck */uni/angstellte* erfüllen
- `FOR $a IN /uni, $b IN $a/sonstige, $c IN $a/studenten`
  `UPDATE $a {`
  `DELETE $b,`
  `INSERT <alumni /> AFTER $c }`
  - *ER(T_i)* enthält $a, $b, $c und alle ihre Vorgänger
  - *D(T_i)* enthält $b
  - *I(T_i)* enthält $a
  - *I_A(T_i)* enthält $c

6.4 SnaX: Snapshot Isolation for XML

SnaX versucht den Kontrolloverhead für Lesetransaktionen bzw. -operationen noch weiter zu verringern, indem nur noch *write/write*-Konflikte überprüft werden. Die damit erreichte Stufe der Konsistenz wird *Snapshot Isolation* genannt,

wobei nicht alle Anomalien vermieden werden können. In [8] werden zwei Versionen von SnaX vorgestellt: *optimistic SnaX* und *locking based SnaX*. *Optimistic SnaX* arbeitet genauso wie *OptiX* allerdings werden nur Schreibkonflikte überprüft (siehe Konfliktmatrix Tabelle 5). *Locking based SnaX* versucht das Verfahren dahin zu optimieren, dass Konflikte schon während der Transaktion geprüft werden. Anstatt einen betroffenen Knoten in das entsprechende *Write-Set* aufzunehmen, fordert eine Transaktion T_i auf dem zu verändernden Knoten eine D-, R_n-, I- oder I_A-Sperre an. Dann wird geprüft, ob die letzte eingebrachte Version von einer konkurrierenden Transaktion stammt. Ist dies der Fall, wird T_i abgebrochen. Um Vorgänger/Nachfolger-Konflikte zu entdecken werden wie bei hierarchischen Sperrverfahren *intention locks* eingesetzt. Auf den Vorgängern jedes Knotens, auf dem eine Sperre gehalten wird, muss eine *intention-exclusive*-Sperre IX angefordert werden (es wird nur die IX-Sperre benötigt, da Knoten für Lesezugriffe nicht gesperrt werden). Eine IX-Sperre ist, bis auf die D-Sperre, zu allen Sperren kompatibel.
Die Sperren einer Transaktion T_i werden nicht sofort Aufgehoben wenn die Transaktionen erfolgreich beendet wurde, sondern erst, wenn alle konkurrierenden Transaktionen beendet sind.

T_i \ T_j		n ∈			q ∈
	D	R_n	I	I_A	WS
D	-	-	-	√	-
n ∈ R_n	-	-	√	√	√
n ∈ I	-	√	√	√	√
I_A	√	√	√	-	√
q ∈ RS	-	√	√	√	

Tabelle 5: Konfliktmatrix für SnaX

6.5 Bewertung

Obwohl die Verfahren in erster Linie davon ausgehen, dass XQuery ähnlichen Sprachen für den Zugriff auf Dokumente verwendet werden, lässt die Art der Konfliktkontrolle wahrscheinlich auch andere Zugriffsmöglichkeiten zu. Es muss nur sicher gestellt werden, dass *Read-* und *Write-Sets* korrekt geführt werden. Auch die Verwendung von Indexen stellt kein Problem dar, da zumindest der Lesezugriff generell erlaubt wird und erst am Ende der Transaktion auf Konflikte überprüft. Es ist nur wichtig, dass der Knoten in dem entsprechenden Set vermerkt ist.
Zu einem Problem bei *locking based SnaX* kann das späte Freigeben der Sperren werden. Dadurch kann es zu einer langen Blockierung eines Knotens durch eine

lange, konkurrierende Transaktion kommen. Zudem bietet *locking based SnaX* keine volle Serialisierbarkeit, sondern *snapshot isolation*.

Kriterium		Bemerkung
Datenmodell	√	
Schnittstellen	X	Keine Angaben zum Verhalten bei navigierenden Anfragen
Knotenadressierung	?	Keine Vorgaben
Serialisierbarkeit	√	*locking based SnaX* bietet *snapshot isolation*

7 XDGL

XDGL ist ein weiteres XPath-basiertes Sperrverfahren. Im Gegensatz zu den bisher vorgestellten Verfahren sperrt XDGL nicht auf dem Dokument, sondern auf einem *DataGuide*. Das Verfahren wird in [9] von Peter Pleshachkov, Petr Chardin und Sergei Kuznetsov vorgestellt.

7.1 Datenmodell

Wie schon erwähnt sperrt XDGL auf einem *DataGuide*. Ein *DataGuide* lässt sich als Strukturzusammenfassung eines Dokumentes beschreiben. Dabei hat jeder Pfad im Dokument genau einen Pfad im *DataGuide* und jeder Pfad im *DataGuide* ist im Dokument enthalten. In Abbildung 8 ist ein DataGuide dargestellt zum XML-Fragment 1.2. Als Anfragesprache wird eine vereinfachte Form von XPath verwendet, die Prädikate nicht berücksichtigt. Zusätzlich werden zwei Operationen definiert, mit denen die Struktur des Dokumentes verändert werden kann:

1. INSERT *constructor* (INTO | AFTER | BEFOR) $path - expr$
2. DELETE $path - expr$

constructor kann entweder ein *Element*- oder *Attribut*-Konstruktor sein und ist wie folgt definiert: $element\{name\}\{inhalt\}$ bzw. $attribut\{name\}\{text\}$. Der folgende Ausdruck fügt einen neuen Hiwi ein: *INSERT element {person} {element {name} {'Kling'} element {vorname} {'Felix'} } INTO /uni/angestellte/hiwis.*

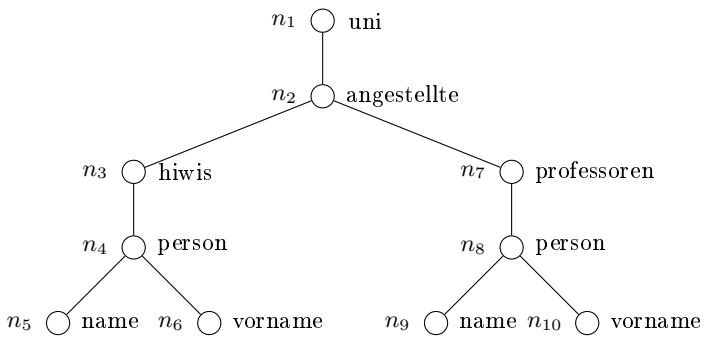


Abbildung 8: DataGuide

Listing 1.2: XML-Fragment

```
<uni>
    <angestellte>
        <hiwis>
            <person>
                <name>Kling</name>
                <vorname>Felix</vorname>
            </person>
        </hiwis>
        <professoren>
            <person>
                <name>Professor</name>
                <vorname>Muster</vorname>
            </person>
        </professoren>
    </angestellte>
</uni>
```

7.2 Sperrmodi

Das XDGL-Protokoll verwendet mehrere Sperrmodi, um einen kontrollierten Zugriff auf das XML-Dokument zu gewährleisten. Wie schon erwähnt werden dabei die Knoten des *DataGuide* gesperrt. Es existieren folgende Sperren (mit kurzer Erläuterung der Funktion):

***SI,SA,SB* (*shared insert*, *shared after*, *shared befor*):** Diese Sperrmodi kontrollieren die Einfügeoperationen. Dabei ist jede Sperre zu den anderen kompatibel, aber nicht zu sich selbst.

***X* (*exclusive lock*):** Die Sperre wird für einen neu erstellten Knoten angefordert.

***ST* (*shared tree*):** Anfragen fordern diese Sperre für die Ergebnisknoten des XPath-Pfadausdrucks an. Ist das Ergebnis ein Teilbaum des XML-Dokuments bzw. des DataGuides, so wird die Sperre nur für die Wurzel des Teilbaums angefordert.

***XT* (*exclusive tree*):** Diese Sperre wird von der Löschoperationen angefordert und sperrt die Wurzel eines Teilbaums exklusiv.

***IS*, *IX*:** Wie bei hierarischen Sperrverfahren werden diese Sperren auf den Vorfahren der jeweiligen Knoten, die mit einem *exclusive* oder *shared lock* belegt sind, angefordert.

Daraus ergibt sich die in Tabelle 6 dargestellte Sperrmatrix.

7.3 Phantomvermeidung

Die bisher eingeführten Sperrmodi können für Pfadausdrücke, die die *descendant*-Achse benutzen, keine Phantome verhindern. Aus diesem Grund werden *logische Sperren* eingeführt. Für jede Anfrage wird somit eine Sperre *(L, name)* auf den Knoten des vorletzten Schritts eines XPath-Pfadausdrucks angefordert. Ein Löschoperationen fordert ebenfalls eine Sperre *(L, name)* an, wobei *name* der Knotenname des zu löschenden Knotens im *DataGuide* ist. Eine Einfügeoperation fordert *(IN, name)* auf jeden Vorgänger von *name* an. *(L, name)* und *(IN, name')* sind nicht kompatibel, wenn $name = name'$.

	SI	SA	SB	X	ST	XT	IS	IX
SI	-	+	+	-	+	-	+	+
SA	+	-	+	-	+	-	+	+
SB	+	+	-	-	+	-	-	+
X	-	-	-	-	-	-	+	+
ST	+	+	+	-	+	-	+	+
XT	-	-	-	-	-	-	-	-
IS	+	+	+	+	+	-	+	+
IX	+	+	+	+	-	-	+	+

Tabelle 6: Sperrmatrix von XDGL

7.4 Bewertung

Da auf Basis eines *DataGuides* gesperrt wird, muss das Verfahren im Vergleich zu anderen Sperrverfahren, die auf dem Dokument direkt sperren, weniger Sperren verwalten. Es garantiert Serialisierbarkeit und löst das Phantomproblem. Allerdings bringt ein *DataGuide* auch Nachteile mit sich: Zum einen muss der *DataGuide* erst durch Analyse des Dokuments erstellt werden, was bei großen und komplexen Dokumenten aufwendig sein kann, zum anderen können für eine

semi-strukturierte Datenstruktur mehrere verschiedene *DataGuides* erstellt werden (siehe dazu [10]). Dabei hat die Ausprägung des *DataGuides* Auswirkung auf die Nebenläufigkeit der Transaktionen. So wird die Struktur des XML-Fragments korrekt durch den *DataGuide* in Abbildung 9 repräsentiert. Jedoch lässt sich erkennen, dass durch diese Darstellung die Konfliktwahrscheinlichkeit erhöht wird. Möchten z.B. T_1 einen Personenknoten unter *hiwis* einfügen und T_2 einen Personenknoten unter *professoren*, fordern beide Transaktionen eine X-Sperre auf *person* an und es kommt zum Konflikt.

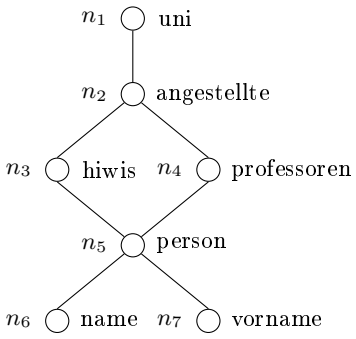


Abbildung 9: Ein weiterer *DataGuide* zur Repräsentation des XML-Fragments

Da in [9] nur die *parent*-, *descendant*- und *attribut*-Achse erwähnt werden, ist davon auszugehen, dass nur diese Achsen unterstützt werden. Da ein *DataGuide* die Struktur einer Datenstruktur repräsentiert, bleibt die Frage offen, wie individuelle Knoten gesperrt werden können, zumal keine Prädikate in den XPath-Ausdrücken unterstützt werden. Diese Unterstützten soll Teil der weiteren Arbeiten an diesem Verfahren werden.
Desweiteren wird in der Arbeit nicht auf den direkten Zugriff auf Knoten eingegangen, zum Beispiel über sekundäre Indexe oder ID/IDREF-Beziehungen. Auch wie sich das Verfahren bei Zugriffen über die DOM-Schnittstelle verhält und ob dies möglich ist, wird nicht erklärt.

Kriterium		Bemerkung
Datenmodell	X	Sperrgranulat zu grob
Schnittstellen	X	Keine Angaben zum Verhalten bei navigierenden Anfragen
Knotenadressierung	?	Keine Vorgaben
Serialisierbarkeit	√	

8 taDOM

taDOM ist ein Verfahren, das hierarchische Sperren zu Kontrolle der Nebenläufigkeit einsetzt. Es wird ausführlich in [1] behandelt.

8.1 Datenmodell

Dem taDOM[7]-Datenmodell liegt der taDOM-Baum zugrunde, der wiederum auf DOM basiert. Er erweitert DOM um zwei Knotentypen und "virtuelle Navigationskanten":

Attributwurzel: Alle Attribute eines Elementknotens sind Kinder einer Attributwurzel und diese wiederum ist Kind des Elementknotens. Somit reicht es beim Zugriff auf alle Attribute aus, die Attributwurzel zu sperren.

String-Knoten: Diese Knoten nehmen die Werte von Attribut- und Textknoten auf.

Virtuelle Navigationskanten: Knoten sind über die virtuelle Navigationskanten *prevSibling* und *nextSibling* mit ihren Geschwistern verbunden.

Des Weiteren werden 19 Basisoperationen definiert, die sich an die DOM-Operationen anlehnen.

Adressierung. Zur Adressierung der Knoten wird das in Kapitel 2.3 vorgestellte Konzept der DeweyIDs benutzt. Zusätzlich zu den schon angesprochenen Vorteilen der DeweyIDs, können die hierarchischen Sperrverfahren den Vorteil nutzen, dass die IDs der Vorfahren eines Knotens direkt aus der ID des Knotens ermittelt werden können und somit kein Zugriff auf das Dokument selbst notwendig ist.

8.2 Protokolle

taDOM versucht die unterschiedlichen DOM-Level zu unterstützen (DOM 2 und DOM 3) und bietet für jedes Level eigene Protokolle mit geeigneten Sperrmodi (taDOM2 und taDOM3). Um sich weiter von der Systemimplementierung zu lösen, werden beide Protokolle um zusätzliche Sperren erweitert (taDOM2+ und tDOM3+). Alle taDOM-Protokolle verwalten pro Transaktion und Knoten nur eine Sperre. Fordert eine Transaktion auf einem von ihr gesperrten Knoten einen neuen Sperrmodus an, so werden mit Hilfe von *Konversionsmatrizen* alle Sperren auf dem entsprechenden Pfad umgewandelt (falls nötig), um dieselbe Isolation der Transaktion gewährleisten zu können.

[7] transactional DOM

8.3 taDOM2

Die Sperranforderung für einen Knoten besteht aus der Transaktionsidentifikation, einer Adresse des zu sperrenden Knotens und dem Sperrmodi. Davon ausgehend werden auf allen Vorgängern des Knotens Anwartschaftssperren angefordert, die durch die Adresse des Knotens direkt ableitbar sind. Folgende Sperrmodi werden in taDOM2 bereitgestellt:

IR (*intention read*): Die Sperre deutet auf eine NR-, LR- oder SR-Sperre im darunter liegenden Teilbaum hin. Sie fordert eine IR-Sperre auf dem Elternknoten.

NR (*node read*): Die Sperre wird für den lesenden Zugriff auf einen Knoten angefordert. Sie erfordert eine IR-Sperre auf dem Elternknoten.

LR (*level read*): Diese Sperre sperrt den Knoten und seine direkten Kinder für den lesenden Zugriff. Sie erfordert eine IR-Sperre auf dem Elternknoten.

SR (*subtree read)*: Mit dieser Sperre wird der gesamte Teilbaum unterhalb des Kontextknotens für den lesenden Zugriff gesperrt. Auch sie fordert eine IR-Sperre auf dem Elternknoten.

IX (*intention exclusive*): Diese Sperre weist auf den Sperrmodi SX im darunter liegenden Teilbaum hin, der aber kein Kind diese Knotens ist (siehe dazu den nächsten Sperrmodus CX). Auf dem Elternknoten wird eine IX-Sperre gefordert.

CX (*child exklusive)*: Diese Sperre gibt an, dass auf einem Kind dieses Knotens eine Schreibsperre SX angefordert wurde. Durch diese zusätzliche Sperre, kann direkt auf einem Knoten geprüft werden, ob eine LR-Sperre gewährt werden kann (auf einem Knoten mit IX-Sperre kann somit direkt eine LR-Sperre gewährt werden, da kein Kind dieses Knotens mit einer Schreibsperre belegt ist). Diese Sperre erfordert auf dem Elternknoten eine IX-Sperre.

SX (*subtree exclusive*): Der gesamte Teilbaum wird für den Schreibzugriff gesperrt. Damit ist es möglich Werte zu verändern und Modifikationen an der Teilbaumstruktur vorzunehmen. Auf dem Elternknoten wird eine CX-Sperre gefordert.

SU (*subtree update*): Alle Nachfahren werden für den Lesezugriff gesperrt. Diese Sperre kann zu einer reinen Lesesperre (SR) oder Schreibsperre (SX) konvertiert werden. Auf dem Elternknoten wird eine IR-Sperre gefordert. Wird die Sperre zu einer SX-Sperre konvertiert, so werden die IR- zu IX- bzw. CX-Sperren konvertiert.

Abbildung 7a zeigt, welche Sperren zueinander kompatibel sind. Beispiel VIII soll den Ablauf der Sperranforderungen verdeutlichen. Die Sperrverteilung wird in Abbildung 10 dargestellt.

Beispiel VIII. Transaktion T_1 liest den Namen des Hiwis. Dazu muss eine IR-Sperre auf *uni*, *angestellte*, *hiwis* und *person* angefordert werden, um auf *name* eine NR-Sperre zu setzten. Die Transaktion T_2 möchte die gesamten Daten des Hiwis rekonstruieren und fordert dazu auf *uni*, *angestellte*, *hiwis*

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

(a) Sperrmatrix

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

(b) Konversionsmatrix

Tabelle 7: taDOM2

jeweils eine *IR*-Sperre und auf *person* eine *SR*-Sperre an. Um den Namen eines Professors zu bearbeiten, muss die Transaktion T_3 *IX*-Sperren auf *uni*, *angestellte* und *professoren* anfordern, auf *person* eine *CX*-Sperre und auf *name* eine *SX*-Sperre. Gleichzeitig möchte die Transaktion T_4 einen neuen Hiwi einfügen und muss dafür zuerst einen neuen *person*-Knoten einfügen. Dazu werden auf *uni* und *angestellte* *IX*-Sperren, auf *hiwis* eine *CX*-Sperre und auf den neuen *person*-Knoten eine *SX*-Sperre angefordert. Möchte die Transaktion T_5 alle Kinder von *hiwis* lesen, so muss auf diesem Knoten eine *LR*-Sperre angefordert werden. Diese ist allerdings nicht mit der *CX*-Sperre von Transaktion T_4 kompatibel und darum wird T_5 blockiert.

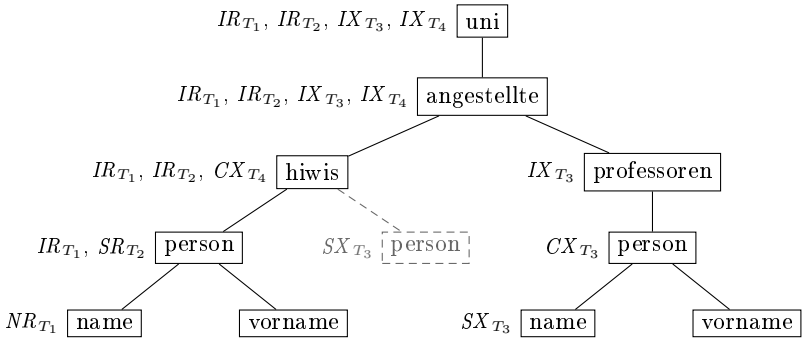


Abbildung 10: Knotensperren in taDOM2

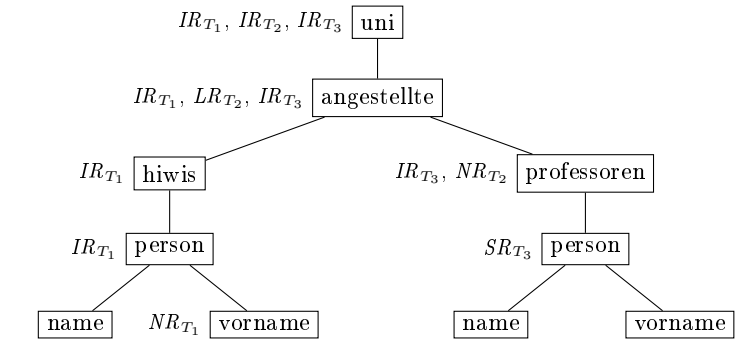
Sperrkonversion. Wie schon erwähnt verwaltet taDOM pro Knoten und Transaktion nur ein Sperre und nutzt Konversionsmatrizen zum umwandeln von

Sperren. Wie dies abläuft wird in Beispiel IX beschrieben. Dabei kommt die Konversionsmatrix in Tabelle 7b zum Einsatz. Dabei bedeutet IX_{NR}, dass auf dem Knoten eine IX-Sperre gesetzt wird und auf alle seine Kinder eine NR-Sperre. Dazu muss auf das Dokument zugegriffen werden, da die Kinder nicht über die DeweyID des Elternknotens bestimmt werden können.

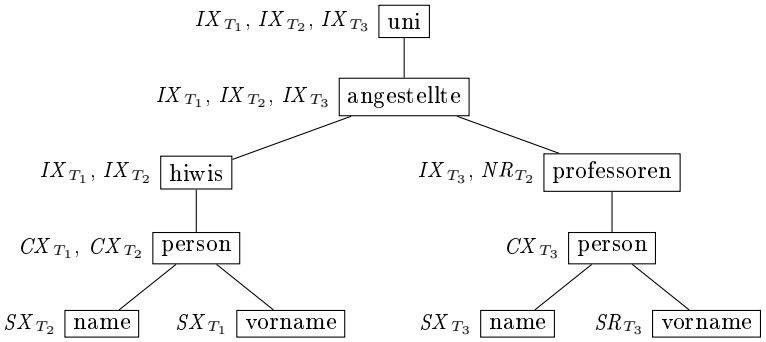
Beispiel IX. Ausgangssituation: T_1 liest den Vornamen des ersten Hiwis, T_2 rekonstruiert die Daten des Professors und T_3 liest die Kinder des Knotens *angestellte* (Abbildung 11a). T_1 möchte nun den Vornamen ändern. Dazu muss auf *vorname* eine SX-Sperre angefordert werden, die wiederum eine CX-Sperre auf dem Elternknoten *person* fordert. Zusätzlich muss auf allen Vorfahren von *person* eine IX-Sperre angefordert werden. In Tabelle 7b lässt sich nun erkennen, dass *vorname* mit einer SX-Sperre belegt werden muss. T_1 hält schon eine NR-Sperre und fordert eine SX-Sperre an. Genauso wird mit den anderen Sperren verfahren. Dabei sind die neuen Sperren alle zu den von anderen Transaktionen gehaltenen Sperren kompatibel. Verändert die Transaktion T_2 im weiteren Verlauf den Namen des Hiwis, so muss ebenfalls auf *name* eine SX-, auf dem Elternknoten eine CX- und auf allen weiteren Vorfahren eine IX-Sperre angefordert werden. T_2 hält schon ein LR-Sperre auf *angestellte*. Wird auf diesem Knoten von T_2 eine IX-Sperre angefordert, wird die LR-Sperre (laut Konversionsmatrix) in eine IX-Sperre überführt. Zusätzlich werden auf den Kindern NR-Sperren angefordert. Da auf dem Knoten *hiwis* eine IX-Sperre von T_2 verlangt wird, wird wieder eine Sperrkonversion durchgeführt. Auch hier sind alle neuen Sperren kompatibel zu den schon bestehenden. Zum Schluss möchte nun T_3 den Namen des Professors ändern. Dazu müssen SX-, CX- und IX-Sperren auf den entsprechenden Knoten angefordert werden. Die CX-Sperre wird auf *person* angefordert, auf dem T_3 schon eine SR-Sperre hält. Diese Sperre wird in eine CX-Sperre konvertiert und auf alle Kinder werden SR-Sperren angefordert. Die daraus entwickelte neue Sperrsituation ist in Abbildung 11b dargestellt.

Durch die Konversion der Sperren können Deadlocks entstehen, falls zwei Transaktionen auf demselben Knoten zu Sperren konvertieren wollen, die inkompatibel zu der durch die andere Transaktion gehaltenen Sperre. Die Blockierung kann dann nur durch die Rücksetzung einer Transaktion gelöst werden. Mit dem Einsatz der SU-Sperre, können solchen Situationen verhindert werden, da lediglich eine Transaktion warten muss.

Sperrtiefe. Bei großen und komplexen Dokumenten kann es zu einer großen Anzahl zu verwaltender Sperren kommen. Um diese Anzahl besser zu kontrollieren, kann eine so genannte (maximale)*Sperrtiefe* auf dem taDOM-Baum angegeben werden. Bis zu dieser Tiefe werden alle Sperren wie sie angefordert werden gewährt. Soll ein tieferer Knoten gesperrt werden, so wird diese Sperre durch eine



(a) Ausgangssituation



(b) Nach Sperrkonversion

Abbildung 11: Sperrkonversion in taDOM2

SR-, *SX*- oder *SU*-Sperre auf einem Vorgängerknoten innerhalb der Sperrtiefe ersetzt.

Kantensperren. Durch die bisher behandelten Sperren kann der direkte Zugriff auf Knoten ausreichend kontrolliert werden. Navigiert allerdings eine Transaktion über die im taDOM-Baum bestehenden Beziehungen zwischen den Knoten, so kann nicht gewährleistet werden, dass diese Beziehungen im Verlauf der Transaktion identisch bleiben. Liest T_1 zum Beispiel das zweite Kind eines Knotens und möchte über die *prevSibling*-Kante zum ersten Kind navigieren, kann von einer anderen Transaktion inzwischen ein neuer Knoten zwischen dem ersten und dem zweiten Kind eingefügt worden sein (Phantome). Um solchen Strukturänderungen vorzubeugen, werden so genannte *Kantensperren* eingesetzt:

ER (*edge read*): Wird für eine Kante angefordert, wenn eine Transaktion über diese Kante zu einem Knoten navigieren will. Ist die Sperre gewährt, können die Adresse des Zielknotens bestimmt und eine Knotensperre auf dem Zielknoten angefordert werden.

EU (*edge update*): Entspricht der Knotensperre *SU*. Diese Sperre gewährt zuerst einen lesenden Zugriff mit der Möglichkeit, einen Up- bzw. Downgrade der Sperre durchzuführen.

EX (*edge exclusive*): Bevor ein Knoten eingefügt oder gelöscht werden kann, muss diese Sperre auf allen Navigationskanten angefordert werden, die von der Strukturänderung betroffen sind.

Tabelle 8 zeigt dazugehörige die Sperr- und Konversionsmatrix.

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

(a) Sperrmatrix

	ER	EU	EX
ER	ER	ER	EX
EU	EU	EU	EX
EX	EX	EX	EX

(b) Konversionsmatrix

Tabelle 8: taDOM2 Kantensperren

Virtuelle Namensknoten. taDOM2 unterstützt den vollen Funktionsumfang von DOM Level 2. Um mit taDOM2 auch die DOM Level 3 Funktionalität zu unterstützten, muss das Sperrprotokoll erweitert. Besonders die zur Änderung eines Knotens anzufordernde *SX*-Sperre verhält sich sehr restriktiv, so dass auch bei Namens- oder Wertänderungen von Knoten der gesamte darunter liegende Teilbaum gesperrt wird und andere Transaktionen nicht mehr in diesem navigieren können. Um dieses Problem zu lösen werden so genannte *virtuelle Namensknoten* eingefügt. Sie werden direkt an den entsprechenden Element- oder Attributknoten gehängt und bekommen die DeweyID dieses Knoten mit einer zusätzlichen 0 als letzte Division. Damit eine Trennung zwischen Inhalts- und Strukturzugriffen und ein höhere Transaktionsdurchsatz geschaffen. Möchte eine Transaktion den Wert oder Namen eines Knotens ändern, so sperrt sie den entsprechenden Namensknoten. Soll ein Knoten eingefügt oder gelöscht werden, wird wie üblich auf dem entsprechenden Elementknoten gesperrt.

8.4 taDOM2+

Das bisher vorgestellte Verfahren zur Sperrkonversion erfordert für manche Sperren den Zugriff auf das Dokument, um die Kinder eines Knotens zu ermitteln. Dies kann bei komplexen Bäumen und vielen Transaktionen zu Leistungseinbußen führen. Aus diesem Grund und um den Sperrverwalter von den Speicherungsstrukturen zu entkoppeln, wird mit taDOM2+ für jeder dieser Sperren (IX_{NR}, IX_{SR}, CX_{NR}, CX_{SR}) ein neuer Sperrmodus eingeführt. Jeder Sperrmodus fordert auf dem Elternknoten eine *IX*-Sperre.

LRIX (*level read itention exclusive*): Diese Sperre sperrt einen Knoten und alle seine Kinder für den Lesezugriff und weist auf eine Änderungsabsicht eines Nachfahren, der kein Kind ist, hin. Sie ersetzt die IX_{NR}-Sperre.

SRIX (*subtree read intention exclusive*): Der Knoten und der gesamte darunter liegende Teilbaum wird für den Lesezugriff gesperrt. Zudem weist die Sperre ebenfalls auf eine Änderungsabsicht eines Nachfahren, der kein Kind ist, hin. Sie ersetzt die IX_{SR}-Sperre.

LRCX (*level read child exclusive*): Sperrt einen Knoten und seine Kinder für den Lesezugriff und weist auf eine exclusive Sperre auf einem seiner Kinder hin. Diese Sperre ersetzt die CX_{NR}-Sperre.

SRCX (*subtree read child exclusive*): Wie *SRIX* wird der gesamte Teilbaum für den Lesezugriff gesperrt und wie bei *LRCX* eine exklusive Sperre auf einem Kind des Kontextknotens angedeutet. Durch diese Sperre wird die CX_{SR}-Sperre ersetzt.

Die Sperr- und Konversionsmatrix sind in Tabelle 9 dargestellt.

8.5 taDOM3 / taDOM3+

taDOM3 unterstützt nun DOM Level 3 direkt, das heißt die virtuellen Namensknoten werden in taDOM3 direkt durch Sperren ersetzt. Dadurch muss pro Knoten nur noch eine Sperre verwaltet werden. taDOM3 erweitert taDOM2 damit mit vier weiteren Sperren. Dabei entstehen Konversionsregeln, die Zugriffe das Dokument erforderlich machen. Wie auch taDOM2+ löst taDOM3+ dieses Problem durch Einführungen weiterer Sperren. Folgende Sperren werden durch taDOM3 zu den taDOM2 hinzugefügt:

NU (*node update*): Diese Sperre verhält sich wie die *SU*-Sperre, sperrt allerdings nur den Knoten mit der Option die Sperre zu einer *NR*-Sperre (Downgrade) oder *NX*-Sperre zu konvertieren. Sie erfordert eine *IR*-Sperre auf dem Elternknoten.

NX (*node exclusive*): Sperrt den Knoten für den Schreibzugriff. Der darunter liegende Teilbaum ist nicht betroffen. Wie bei der *SX*-Sperre wird eine *CX*-Sperre auf dem Elternknoten gefordert.

NRIX (*node read intention exclusive*): Der Knoten wird für den Lesezugriff gesperrt. Dabei wird auf eine exklusive Sperre auf einem seiner Nachfahren, der kein Kind des Knotens ist, hingewiesen. Dieser Sperrmodus ist erforderlich , da die *NX*-Sperre inkompatibel zur *NR*-Sperre ist. Mussten bisher eine *NR*- und *IX*-Sperre konvertiert werden, so resultierte eine *IX*-Sperre und es war nicht ersichtlich, ob der Knoten von einer anderen Transaktion gelesen wurde (es darf keine *NX*-Sperre gewährt werden). Die *NRIX*-Sperre gibt nun an, dass der Knoten von einer Transaktion zuerst gelesen wurde und ist somit inkompatibel zur *NX*-Sperre. Die Sperre fordert eine *IX*-Sperre auf dem Elternknoten.

NRCX (*node read child exclusive*) Analog zu *NRIX*, weist aber auf eine exklusive Sperre auf einem der Kinder des Knotens hin.

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

(a) Sperrmatrix

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

(b) Konversionsmatrix

Tabelle 9: taDOM2+

taDOM3+ erweitert taDOM3 mit den gleichen Sperrmodi die von taDOM2+ hinzugefügt wurden und vier zusätzlichen Sperren:

LRNU (*level read node update*): Sperrt alle Kinder des Knotens für den Lesezugriff und den Knoten selbst für den Lesezugriff mit Änderungsabsicht. Die Sperre kann in eine *LR*-Sperre (Downgrade) oder eine *LRNX*-Sperre (Upgrade) konvertiert werden und erfordert eine *IR*-Sperre auf dem Elternknoten.

SRNU (*subtree read node update*): Äquivalent zur *LRNU*-Sperre, sperrt allerdings alle Nachfahren des Knotens. Die Sperre kann entweder zu einer *SR*-Sperre (Downgrade) oder *SRNX*-Sperre (Upgrade) konvertiert werden.

LRNX (*level read node exclusive*): Der Knoten wird für den Schreibzugriff und seine Kinder für den Lesezugriff gesperrt. Erfordert auf dem Elternknoten eine *CX*-Sperre.

SRNX (*subtree read node exclusive*): Der Knoten wird für den Schreibzugriff und alle Nachfahren für den Lesezugriff gesperrt. Diese Sperre erfordert ebenfalls eine *CX*-Sperre auf dem Elternknoten.

Die vollständigen Sperr- und Konversionsmatrizen von taDOM3 und taDOM3+ werden in [1] dargestellt.

8.6 Phantomvermeidung

Navigierende Zugriffe werden durch die verwendeten Knoten- und Kantensperren ausreichend vor Phantomen geschützt. Bei deklarativen Anfragen, die aus Performancegründen vor allem über Indexstrukturen ausgewertet werden, können allerdings immer noch Phantome auftreten. Um mit bisherigen Mitteln eine Anfrage wie */uni/angestellte/hiwis/following::person* vor Phantomen zu schützen, müsste auf der Wurzel, wegen der *following*-Achse, eine exklusive Sperre angefordert werden. Dadurch ist jedoch kein Zugriff auf das Dokument durch andere Transaktionen möglich.
Mit Hilfe von *wertbasierten Achsensperren* werden Zugriffe über Indexe und somit deklarative Anfragen besser unterstützt. Zu den von XPath vorgegeben Achsen definiert XPath noch eine weitere Achse: *IDvalue*. Sie liefert für einen Kontextknoten den Nachfahren mit der angegebenen ID (z. B. *IDvalue::13*). Nun kann für einen Kontextknoten eine Achsensperre $l_{axis}(deweyID, axis, value, node)$ angefordert werden, wobei

deweyID: die ID des Kontextknotens angibt
axis: eine der Achsen auswählt
value: den Anfragewert angibt und
mode: entweder ein *shared*- (S) oder *exclusive*-lock (X) sein kann

Generell werden bei jedem Zugriff über einen Index Achsensperren angefordert. Damit eine Achsensperre gewährt werden kann, muss sie zu allen anderen Achsensperren kompatibel sein, deren Bereiche sich überlagern. Eine Überlagerung wird mit Hilfe der Bestimmung der relativen Lage zwischen den DeweyIDs der

wertgleichen Achsensperren und der DeweyID des Kontextknotens ermittelt. Wie in Abschnitt 2.3 erwähnt ist dies aufgrund der Beschaffenheit der DeweyIDs relativ leicht. Für jede der 9 Lagen (*Ancestor*, *Parent*, *Preceding*, *Preceding-Sibling*, *Self*, *Child*, *Descendant*, *Following*, *Following-Sibling*) ist eine so genannte *Achsenüberlagerungstabelle* definiert, mit deren Hilfe eine Überlagerung festgestellt werden kann. Existiert kein Überlagerung, so wird die Sperre gewährt, ansonsten muss der Sperrmodus kompatibel sein. Beispiel X zeigt die Anwendung der Achsensperren. Um nun Anfragen, die über Indexe ausgwertet werden, auch vor Phantomen durch navigierende Zugriffe zu schützen, muss für einen Knoten, der eingefügt werden soll, eine Achsensperre auf der *self*-Achse angefordert werden.

Beispiel X. Eine Transaktion T_1 greift über */uni/angestellte/hiwis/following::person* auf alle Personen zu, die dem Knoten *hiwis* im Dokument folgen. T_1 fordert dazu die Sperre $l_{axis}(3{:}1.3.3, Following, "person", S)$ an, die auch gewährt wird. Die Adressen der in Frage kommenden Knoten werden über einen Index ermittelt und entsprechende Knotensperren gesetzt. Die Transaktion T_2 versucht nun einen neuen *person* Knoten unter *professoren* einzufügen. T_2 fordert die Sperre $l_{axis}(3{:}1.3.5, Child, "person", X)$ an. Um nun zu prüfen ob ein Konflikt vorliegt, werden alle wertgleichen Achsensperren gesucht. Es findet sich die Sperre von T_1. Anhand der DeweyIDs wird die Lage des Knotens *3:1.3.3* (*hiwis*) zu *3:1.3.5* (*professoren*) ermittelt (das Vorgehen wird in [2] beschrieben). Es ergibt sich, dass *hiwis* in der *preceding-sibling*-Lage zu *professoren* liegt. Mit Hilfe der *Achsenüberlagerungstabelle* 10 kann eine Achsenüberlagerung festgestellt werden. Da S- und R-Sperren nicht kompatibel sind, wird die Einfügeoperation von T_2 nicht ausgeführt.

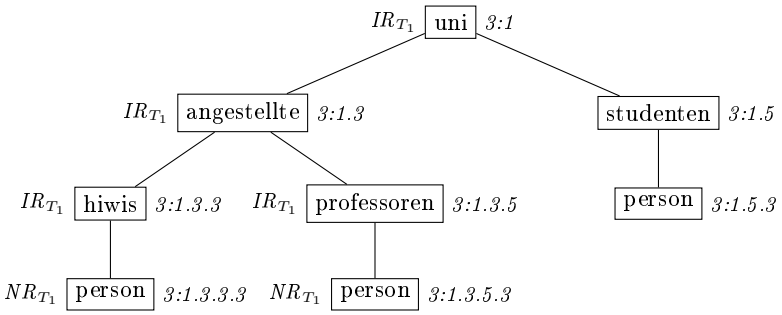


Abbildung 12: Grafik zu Beispiel X

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

Tabelle 10: *Achsenüberlagerungstabelle* der *preceding-sibling*-Lage

8.7 Bewertung

taDOM bemüht sich allen Anforderungen an die Synchronisation von Zugriffen auf XML-Dokumenten gerecht zu werden. Verglichen mit den anderen Verfahren, ist die Anzahl der Sperren die dafür notwendig sind, relativ hoch. Bei großen Bäumen und vielen Transaktionen müssen sehr viele Sperren verwaltet werden, was sich wiederum auf die Performance niederschlagen kann. Die Sperrtiefe lässt sich zwar einstellen, jedoch wird dann das Sperrgranulat mit jeder Ebene näher zur Wurzel größer und die Nebenläufigkeit weiter eingeschränkt. Zudem können die Sperren Prädikate in XPath-Pfadausdrücken nicht optimal unterstützen. Eine Achsensperre sperrt dich komplette Achse für einen Knotenwert, unabhängig davon, ob auf alle diese Knoten zugegriffen wird oder nur eine Teilmenge anhand der Prädikate ausgewählt wird.
Davon abgesehen ist taDOM das einzige der vorgestellten Verfahren, welches die Anforderungen erfüllt. Es werden explizit navigierende und deklerative Anfragesprachen unterstützt, die gegenseitig isoliert werden und somit parallel für den Zugriff auf ein Dokument benutzt werden können. Die DeweyIDs ermöglichen eine einfache Auswertung von Knotenbeziehungen und eine einfache Ermittlung der Vorgänger eines Knotens. Durch sie können XML-Fragment ordnungserhaltend wiederhergestellt werden.

Kriterium		Bemerkung
Datenmodell	√	
Schnittstellen	√	
Knotenadressierung	√	DeweyIDs
Serialisierbarkeit	√	

9 Zusammenfassung

Die in dieser Arbeit vorgestellten Synchronisationsverfahren basieren alle auf unterschiedlichen Grundlagen. Es wird von unterschiedlichen Datenmodellen ausgegangen (XML-Dokument, DataGuide), bestimmte Anfragensprachen vorausgesetzt (XQuery/XPath, DOM), unterschiedliche Sperrarten verwendet (Knotensperren, Kantensperren, Achsensperren, Pfadsperren) oder auf Versionen gesetzt. Die meisten Verfahren können nicht allen Anforderungen genügen. So wird von allen Verfahren, bis auf taDOM, nur eine Zugriffsmethode unterstützt (entweder DOM oder Sprachen, die auf XPath basieren), wobei vor allem XPath meist nur in vereinfachter Form verwendet werden kann (z.B. Pfadsperren - XPath, XDGL). Auch die mangelnde Unterstützung von Zugriffen über Indexe stellt ein Problem dar. Relationale Datenbanksysteme haben gezeigt, dass auf die Verwendung von Indexen praktisch nicht verzichtet werden kann, um eine gute Performance gewährleisten zu können. Einige Verfahren können die Serialisierbarkeit nicht mehr gewährleistet werden, wenn ein direkter Zugriff auf Knoten möglich ist (z.B. *2PL). Die Voraussetzungen, die für die Verfahren aufgestellt werden, wie z.B. der ausschließliche Zugriff über die Dokumentwurzel, können

im realen Einsatz nicht verlangt werden. Ein Verfahren muss darum ausreichend flexibel sein, um eine Vielzahl von Verwendungszwecken zu unterstützen.
taDOM und OptiX stellen gute Ausgangspunkte für die weitere Entwicklung dar. Wie schon erwähnt kann taDOM vielen Anforderungen genügen und ist hinsichtlich der Sperrverwaltung (z.B. Beschränkung der Sperrtiefe) und der Konsistenzebenen hinreichend flexibel. OptiX verfolgt als Mehr-Versionenverfahren eine andere Richtung, die sich jedoch auch in relationalen Datenbanksystemen etabliert hat.

Literatur

1. Haustein, M.P.: Feingranulare Transaktionsisolation in nativen XML-Datenbanksystemen. PhD thesis, TU Kaiserslautern (2005)
2. Härder, T., Haustein, M., Mathis, C., Wagner, M.: Node labeling schemes for dynamic XML documents reconsidered. Data & Knowledge Engineering **60**(1) (2007) 126–149
3. Härder, T., Rahm, E.: Datenbanksysteme: Konzepte und Techniken Der Implementierung. Springer (2001)
4. Kemper, A., Eickler, A.: Datenbanksysteme - Eine Einführung, 6. Auflage. Oldenbourg (2006)
5. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. In: SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM (1995) 1–10
6. Helmer, S., Kanne, C., Moerkotte, G.: Evaluating Lock-based Protocols for Cooperation on XML Documents. SIGMOD Record **33**(1) (2004) 58–63
7. Dekeyser, S., Hidders, J.: Conflict scheduling of transactions on xml documents. In: ADC '04: Proceedings of the 15th Australasian database conference, Darlinghurst, Australia, Australia, Australian Computer Society, Inc. (2004) 93–101
8. Sardar, Z., Kemme, B.: Don't be a Pessimist: Use Snapshot based Concurrency Control for XML. Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)-Volume 00 (2006)
9. Pleshachkov, P., Chardin, P., Kuznetsov, S.: XDGL: XPath-Based Concurrency Control Protocol for XML Data. Database: 22nd British National Conference on Databases, Bncod 22, Sunderland, UK, July 5-7, 2005, Proceedings (2005)
10. Goldman, R., Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. Proceedings of the 23rd International Conference on Very Large Data Bases (1997) 436–445