Universität Kaiserslautern
Fachbereich Informatik
AG Datenbanken und Informationssysteme
Prof. Dr.-Ing. Dr. h.c. Theo Härder

taDOM - Synchronisierung für XML-Dokumente

Implementierung und Evaluierung

Diplomarbeit

von
Georges Berscheid

Betreuer:

Dipl.-Inform. Michael P. Haustein

September 2003

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur angefertigt habe.

Kaiserslautern, den 4. September 2003

Georges Berscheid

Inhaltsverzeichnis

Kapitel 1 Einführung

Die fortwährend wachsenden Anwendungsgebiete der erweiterbaren Auszeichnungssprache XML zum elekronischen Datenaustausch, sowohl im industriellen als auch im wissenschaftlichen Bereich, führt zu einer großen Anzahl an einzelnen Dokumenten, in denen die Daten abgelegt werden. Es ist heute schon häufig notwendig, dass mehrere Benutzer gleichzeitig auf den Datenbestand in solchen XML-Dokumenten zugreifen müssen und es ist ein deutlicher Trend in Richtung XML-Datenbankmanagementsysteme (DBMS) abzusehen.

Der aktuelle Stand der Forschung beruht hauptsächlich darauf, strukturierte Daten in relationalen, beziehungsweise objektrelationalen Datenbanken abzulegen. In den letzten dreißig Jahren haben sich die Verfahren zur Datenverwaltung in Relationenmodellen perfektioniert und alle führenden Datenbanksysteme auf dem Markt sichern die so genannten ACID-Eigenschaften zu [12], die die Qualität der Datenhaltung durch Konsistenz, Zuverlässigkeit und Robustheit garantieren. Diese Forderungen möchte man nun auch für die Verwaltung von XML-Daten geltend machen.

Es existieren bereits verschiedene Ansätze zur Abspeicherung von XML-Daten in relationalen Datenbankenmanagementsystemen (RDBMS), um die Techniken und Verfahren für die Datenhaltung in RDBMS auch für XML-Daten auszunutzen. Dadurch könnte man die Synchronisationsmechanismen für Transaktionen auch für den Zugriff auf XML-Daten ausnutzen. Diese Methode mag auf den ersten Blick zwar einfach und effizient sein, verbirgt aber eine Reihe von gravierenden Nachteilen. Legt man beispielsweise ein komplettes XML-Dokument in einem BLOB oder CLOB Datenfeld einer relationalen Datenbank ab, können Sperren nur auf der Dokumentenebene angelegt werden, das heißt, es hat zu jedem Zeitpunkt immer nur eine einzige Transaktion die Zugriffsrechte für ein Dokument. Bricht man auf der anderen Seite das XML-Dokument auf ein Relationenschema herunter und verteilt es auf mehrere Tabellen, so hat das Einfügen eines XML-Elements Auswirkungen auf einen großen Teil des Dokuments oder sogar andere Dokumente. Das Einfügen eines XML-Elements muss oftmals auf mehrere relationale Einfügeoperationen abgebildet werden. Viele Datenbanksysteme sperren die betroffenen Tabellen in diesem Fall exklusiv, um so genannte Phantome zu vermeiden. Durch dieses Vorgehen werden also Teile eines Dokuments durch den Synchronisationsalgorithmus gesperrt, obwohl diese eigentlich überhaupt nicht durch die Einfügeoperation beeinflusst werden. Verschiedene Ansätze zum Ablegen von XML-Daten in relationalen Datenbanken werden im Detail in [5], [16] und [19] vorgestellt.

Obwohl es zwar technisch möglich ist, XML-Daten in ein relationales Modell umzuwandeln, stellt dieser Ansatz aber eher eine schlechte Lösung bezüglich Parallelität der Transaktionen dar.

Die sehr unterschiedliche Struktur der Daten erfordert ein maßgeschneidertes Konzept zur Synchronisierung nebenläufiger Zugriffe.

Wir werden uns in dieser Arbeit hauptsächlich damit beschäftigen, ein solches Synchronisationsmodell zu beschreiben und dessen Eigenschaften und Leistungsverhalten anhand einer Realisierung zu untersuchen. Dazu werden wir unser Konzept speziell auf die Zugriffsmechanismen des DOM-API zum Lesen, Traversieren und Aktualisieren von XML-Dokumenten auslegen. Da die Kenntnis der Strukturen und Mechanismen des DOM-API grundlegend für diese Arbeit sind, werden wir diese in Kapitel 2 kurz vorstellen.

In Kapitel 3 gehen wir etwas detaillierter auf verschiedene Ansätze anderer Arbeitsgruppen ein, und stellen deren Vor- und Nachteile heraus. Wir mussten jedoch mit Bedauern feststellen, dass alle diese Ansätze unseren Anforderungen an feingranulare Synchronisierungsmechanismen nur unzureichend genügen. Aus diesem Grund werden wir in Kapitel 4 den so genannten taDOM-Baum vorstellen [10]. Es handelt sich dabei um ein Datenmodell, das den wohlbekannten DOM-Baum um einige Eigenschaften erweitert und das wir zur Verwaltung der Synchronisationssperren heranziehen werden. Da DOM-Methoden die Möglichkeit bieten, sowohl Knoten um Knoten durch den Baum zu navigieren, als auch einfache Anfragen zu stellen, werden wir zusätzlich zu den Knotensperren auch Navigationssperren für die Kanten zwischen den Knoten und logische Sperren zum Vermeiden des Phantomproblems einführen. Darüber hinaus bietet unser Konzept die Möglichkeit die Sperrgranularität für jedes einzelne Dokument über Parameter einzustellen, oder diese durch einen Sperreskalationsmechanismus zur Laufzeit anzupassen.

In Kapitel 5 werden wir dann eine Systemarchitektur vorschlagen, um das taDOM-Sperrkonzept zu implementieren und dessen Verhalten in einer Mehrbenutzerumgebung zu erforschen. Wir verfolgen dabei zwei verschiedene Ansätze. Zunächst implementieren wir einen grobgranular synchronisierenden Anfrageserver, der die einkommenden Anfragen erst in einer Warteschlange serialisiert und dann in der Reihenfolge ihrer Ankunft nacheinander abarbeitet. Da durch diesen Ansatz ein Großteil der Parallelität der Transaktionen verlorengeht, haben wir in einem zweiten Schritt die Systemarchitektur insofern verändert, dass parallele Zugriffe auf die Daten und die Sperrtabellen auch innerhalb einer XML-Anfrage durch parallel laufenden Code möglich werden.

Die Auswirkungen der verschiedenen Ansätze sowie der Einstellungsparameter auf die Leistungsfähigkeit des Systems werden wir in Kapitel 6 untersuchen. Wir führen dazu eine Reihe von Experimenten durch und stellen die Ergebnisse sowohl in tabellarischer als auch in grafischer Form dar, bevor wir in Kapitel 7 Schlussfolgerungen ziehen und einen Ausblick auf zukünftige Arbeiten geben.

Kapitel 2

XML und das Document Object Model

Sowohl die Extensible Markup Language (XML) [18] Spezifikation als auch das Document Object Model (DOM) kommen vom World Wide Web Consortium [17]. DOM bietet eine einfach zu benutzende Schnittstelle zum Zugriff auf XML-Daten. Wir werden im weiteren Verlauf der Arbeit häufig auf DOM zurückkommen, da die gesamte Schnittstelle des Prototypen (siehe Kapitel 5) darauf aufsetzt und somit DOM zu einem grundlegenden Baustein dieser Arbeit macht.

In diesem Kapitel werden wir zunächst auf XML eingehen, um daraufhin den DOM-Baum als Darstellung eines XML-Dokuments zu untersuchen. Dann beschreiben wir die DOM-Schnittstelle zur Verarbeitung des DOM-Baums und die Eigenschaften des Modells sowie dessen Vor- und Nachteile.

2.1 XML

XML steht für *Extensible Markup Language* [18] und ist eine Meta-Sprache, mit deren Hilfe andere Sprachen definiert werden können. XML definiert weder die zur Verfügung stehende Tag-Menge, noch die Grammatik der Zielsprache. Diese werden erst durch die Applikation, die die Zielsprache liest spezifiziert. Nach [6] definiert sich ein XML-Tag durch einen von spitzen Klammern umschlossener Elementname, der zur Auszeichnung eines Dokuments verwendet wird. In XML ist dabei die Wahl der Elementnamen prinzipiell frei, durch *Document Type Definitions* (DTDs, siehe Abschnitt 2.2) kann aber die Zahl der gültigen Tags eingeschränkt werden.

So gibt es zum Beispiel in HTML, einer Markup-Sprache für die Darstellung von Inhalten in einem Web-Browser, eine festgelegte Anzahl von Tags, die ein Browser versteht. Beispielsweise darf ein <*table*> Tag benutzt werden, <*chair*> dagegen ist ungültig. Die Grammatik der Zielsprache dient der korrekten Verwendung der Tags. Sie macht zum Beispiel Aussagen darüber, welche Attribute ein Tag haben darf oder wie die Tags verschachtelt sein dürfen.

Ein XML-Dokument weist verschiedene Eigenschaften auf, wie wir am Beispiel in Abbildung 1 sehen. Jedes Dokument besitzt genau ein Wurzelelement, in unserem Beispiel ist das <*Rezepte*>.

Zu jedem Tag gibt es ein ensprechendes Ende-Tag. Dies ist durch ein '/' vor dem Tagnamen gekennzeichnet. So wird zum Beispiel der <*Titel*> Tag durch </*Titel*> beendet. Start- und Ende-Tag lassen sich auch zu einem <*Tag* /> Tag zusammenfassen, falls dieses außer Attributen keine weiteren verschachtelten Tags enthält. Im Beispiel erkennt man dies am <*Zutat* /> Tag.

2.1.1 XML-Komponenten

Ein XML-Dokument besteht aus verschiedenen Komponenten. Diese sollen hier kurz erläutert werden. Für weitergehende Informationen verweise ich auf die Spezifikation des W3C [18].

Die wohl wichtigste Komponente eines Dokuments ist das *Element*. Es dient zur Strukturierung des Dokuments und trägt ansonsten keine Information. Die XML-Spezifikation gibt keine Einschränkungen bezüglich der Verschachtelungstiefe vor. Dadurch sind beliebig komplexe Baumstrukturen realisierbar. Die Reihenfolge der Nachfolgerelemente eines Knotens muss bei der Verarbeitung beachtet werden, da sich durch Vertauschen von Elementen die Struktur des Baumes im Allgemeinen verändert.

XML erlaubt es, Eigenschaften für Elemente zu definieren. Dies wird durch *Attribute* ermöglicht. Ein Attribut besteht aus einem Namen und einem Wert, wie man in Abbildung 1 anhand des ersten <*Zutat*> Elements erkennen kann. *Anzahl* ist dabei der Name des Attributs, *1* sein Wert. Die Reihenfolge der Attribute spielt keine Rolle.

Die eigentlichen Daten eines XML-Dokumentes sind in *Text*-Abschnitten enthalten. Diese sind einem Element eindeutig zugeordnet, können somit auf einen Namen verzichten und bestehen ausschließlich aus ihrem Wert. Abbildung 1 zeigt das anhand der <*Schritt*> Elemente. Ein Element kann auch mehrere direkt aufeinanderfolgende Text-Abschnitte enthalten. DOM spezifiziert aber einen Mechanismus, das so genannte Normalisieren, der solche Text-Abschnitte zusammenfasst, und somit die Strukur des Dokuments vereinfacht.

Es kann aber vorkommen, dass ein Text-Abschnitt XML-Daten enthalten soll, die als uninterpretierter Text verarbeitet werden sollen. Dies ist mit der Syntax für Text-Abschnitte nicht möglich, da alle XML-Tags innerhalb dieses Text-Abschnitts als Bestandteil des aktuellen Dokuments interpretiert werden. Deswegen wird die Syntax um die so genannten *CDATA-Section* Tags erweitert. Innerhalb eines solchen Tags kann ein beliebiger Text stehen. Das CDATA-Section Tag beginnt mit der Zeichenfolge <*![CDATA[* und endet mit *]]*>. Da der gesamte Inhalt als Rohdaten interpretiert wird, ist eine Verschachtelung von CDATA Sections nicht möglich. Abbildung 1 enthält dazu ein Beispiel.

Des Weiteren ist es möglich, *Kommentare* in ein Dokument einzufügen, die nicht Bestandteil der eigentlichen Information sind, sondern im Allgemeinen nur zusätzliche Auskünfte geben und in den meisten Fällen maschinell nicht ausgewertet werden. Kommentare befinden sich zwischen <*!*-- und -->. Ein Beispiel-Kommentar ist in Abbildung 1 zu finden.

Die *Processing Instruction* ermöglicht es, spezielle Eigenschaften des Dokuments zu spezifizieren. Dies ist vor allem bei maschineller Verarbeitung nützlich. Die Processing Instruction wird durch die Zeichenfolge <*?* eingeleitet und endet mit *?*>. Eine Processing Instruction besteht aus einem so genannten *Target*, einem Identifier für die zu spezifizierende Eigenschaft, und *Data*, dem Wert dieser Eigenschaft. Mit einer Processing Instruction lassen sich zum Beispiel die Versionsnummer oder das Encoding des XML-Dokumentes definieren. In Zeile 1 des Beispiels aus

Abbildung 1 definiert `XML` das Target und `ENCODING="UTF-8" VERSION="1.0"` die entsprechenden Daten.

Abbildung 1 Beispiel für ein XML-Fragment

```
<?XML ENCODING="UTF-8" VERSION="1.0"?>
<!-- Beispiel: XML-Fragment zum Erläutern der verschiedenen XML-Tags -->
<Rezepte>
    <!-- Der Data-Tag enthält zusätzliche Daten zu diesem Dokument -->
    <Data>
      <![CDATA[
      Grobe Struktur: <Rezept><Titel /><Zutat /><Zubereitung><Schritt />
      </Zubereitung></Rezept>
      ]]>
    </Data>
    <Rezept>
        <Titel>Milchkakao</Titel>
        <Zutat Name="Milch" Anzahl="1" Einheit="Tasse" />
        <Zutat Name="Kakao" Anzahl="3" Einheit="Teelöffel" />
        <Zubereitung>
            <Schritt>Kakao in die Milch geben</Schritt>
            <Schritt>Umrühren</Schritt>
        </Zubereitung>
    </Rezept>
    <Rezept>
        <Titel>Butterbrot</Titel>
        <Zutat Name="Brot" Anzahl="1" Einheit="Scheibe" />
        <Zutat Name="Butter" Anzahl="3" Einheit="Messerspitze" />
        <Zubereitung>
            <Schritt>Butter auf das Brot streichen</Schritt>
            <Schritt>Brot in 2 Teile schneiden</Schritt>
        </Zubereitung>
    </Rezept>
</Rezepte>
```

2.1.2 Eigenschaften von XML-Dokumenten

Zwei besonders wichtige Eigenschaften, auf die man immer wieder im Zusammenhang mit XML trifft, sind die *Wohlgeformtheit* und die *Gültigkeit.*

Ein XML-Dokument ist genau dann wohlgeformt, wenn es folgende Kriterien erfüllt:

- Die Deklaration des verwendeten XML-Standards mit Hilfe der entsprechenden Processing Instruction am Anfang des Dokuments muss vorhanden sein.

- Es gibt im XML-Dokument genau ein Wurzelelement. Dieses umschließt alle untergeordneten Elemente.
- XML berücksichtigt Groß- und Kleinschreibung (*case-sensitive*). Deshalb ist darauf zu achten, dass das schließende Element genau so geschrieben ist, wie das öffnende.

  Folgende Schreibweise ist somit ungültig:

  <Name>Hans Mustermann</name>
- Attributwerte müssen zwischen Anführungszeichen stehen.
- Leere Elemente müssen ebenfalls abgeschlossen sein. Dabei sind folgende Schreibweisen äquivalent:

  <LeeresElement></LeeresElement>

  <LeeresElement />
- Elementnamen dürfen nicht mit Ziffern oder der Zeichenkette 'xml' beginnen, und dürfen keine Leerzeichen enthalten.
- Elemente dürfen sich nicht überlappen, sondern müssen komplett ineinander verschachtelt sein. Folgendes Konstrukt ist ungültig:

  <ErstesElement>Text<ZweitesElement>Text</ErstesElement>Text</ZweitesElement>

Auch wenn alle diese Kriterien für ein XML-Dokument erfüllt sind, so ist es zwar wohlgeformt, aber nicht notwendigerweise gültig. In einem gültigen Dokument dürfen nur Elementnamen verwendet werden, die vorher definiert worden sind. Außerdem muss die Struktur des Dokuments der mit Hilfe der so genannten Document Type Definition (DTD) spezifizierten Struktur entsprechen. Wie eine solche Definition aussieht, wird im folgenden Abschnitt beschrieben.

2.2 DTD - Document Type Definition

Wie bereits erwähnt, können in XML prinzipiell beliebige Tagnamen für Elemente in beliebiger Verschachtelung verwendet werden. Da für das maschinelle Verarbeiten von XML-Daten oftmals eine spezielle Strukturierung dieser Daten vorausgesetzt werden muss, gibt es die Möglichkeit, den Aufbau eines XML-Dokuments mit Hilfe von so genannten *Document Type Definitions* (DTD) zu spezifizieren. Somit lassen sich XML-Parser (siehe Abschnitt 2.3) konstruieren, die vor der Verarbeitung eines XML-Dokuments dieses mit der entsprechenden DTD abgleichen und überprüfen, ob die Struktur des Dokuments der Spezifikation entspricht.

Da sich mit DTDs auch sehr komplexe Dokumentstrukuren definieren lassen und die Spezifikation deswegen auch entsprechend umfangreich wird, wollen wir an dieser Stelle nur die wichtigsten und für den weiteren Verlauf dieser Arbeit relevanten Eigenschaften von DTDs betrachten.

2.2.1 Element-Deklaration

Jedes Tag in einem gültigen XML-Dokument muss mit einer Element-Deklaration in der DTD definiert werden. Eine Element-Deklaration spezifiziert den Namen und den möglichen Inhalt eines Elements. Falls als Inhalt eines Elements wieder andere Elemente erlaubt werden sollen, so wird deren Multiplizität mit Hilfe der von regulären Ausdrücken bekannten Interpunktionszeichen ausgedrückt. * steht dabei für ‘beliebig viele’, + für ‘mindestens eines’ und ? für ‘keines

oder eines'. Wichtig ist auch, dass DTDs konservativ sind, das heisst, alles was nicht explizit erlaubt ist, ist verboten.

Als erstes muss man das Root Element eines Dokuments im <!DOCTYPE [ ]> Tag festlegen. Dies ist zugleich auch Container für alle weiteren Tag-Definitionen die zwischen die eckigen Klammern gehören. Für das Dokument aus Abbildung 1 würde dies folgendermaßen aussehen:

```
<!DOCTYPE Rezepte []>
```

Dadurch ist aber erst das Root Element festgelegt, nicht aber der mögliche Inhalt des Rezepte-Tags. Die Definiton selbst erfolgt mittels der <!ELEMENT> Deklaration. Da der Rezepte-Tag einen einzigen Data-Tag und beliebig viele Rezept-Tags enthalten kann, sieht die Element-Deklaration folgendermaßen aus:

```
<!ELEMENT Rezepte (Data, Rezept*)>
```

Alle in der Definition eines Elements verwendeten Elementnamen müssen natürlich auch definiert werden. Wir wollen dies nun für Data und Rezept tun:

```
<!ELEMENT Data (#PCDATA)>
<!ELEMENT Rezept (Titel, Zutat*, Zubereitung)>
```

Das Schlüsselwort #PCDATA steht für *parsed character data* und bedeutet, dass an dieser Stelle nur flacher Text im XML-Dokument vorkommen darf. Das Data-Tag darf demnach keine weiteren Elemente enthalten, da sonst der die Spezifikation verletzt würde.

Außer weitere Elementnamen oder das #PCDATA Schlüsselwort, kann die Inhaltsangabe einer Elementdefinion auch den Wert *ANY* oder *EMPTY* haben. *ANY* definiert das Element, ohne Einschränkungen an den Inhalt des Elements vorzugeben. Es darf dann sowohl beliebige andere Elemente als auch Textdaten enthalten. Wurde *EMPTY* als Inhalt angegeben, so darf das Element weder Text noch andere Elemente enthalten. Dies wird zum Beispiel dann verwendet, wenn Elemente gebraucht werden, deren ganze Information von den Attributen getragen wird.

2.2.2 Attribut-Deklaration

Analog zur Deklaration der Elementhierarchie kann man auch die zu einem Element gehörigen Attribute definieren. Dies ist mit der <!ATTLIST> Deklaration möglich. Die Definition der Attribute eines Elements hat folgenden Gestalt:

```
<!ATTLIST Elementname Attributname Datentyp Standardwert>
```

Elementname ist das zuvor definierte Element, dem das Attribut zugeordnet ist, *Attributname* der Name des Attributs, *Datentyp* der Typ des Attributwertes und *Standardwert* der Wert des Attributs, falls kein anderer explizit angegeben wurde. Es existieren insgesamt zehn Datentypen die man als Attributtypen verwenden kann. Wir beschänken uns allerdings auf zwei, nämlich *CDATA* und *ID*, weil die restlichen absolut keinen Einfluss auf die Synchronisierungsmechanismen haben und somit für diese Arbeit nicht relevant sind.

CDATA ist der wohl am meisten verwendete Attributtyp. CDATA steht für *character data* und sagt aus, dass der Wert des Attributs beliebiger Text sein kann. Dieser Text darf allerdings keine Kleiner-Zeichen (<) oder Anführungszeichen (") enthalten, da das Kleiner-Zeichen den Anfang eines neuen Tags darstellt und das Anführungszeichen das Ende des Textes markiert.

Attributwerte können ebenfalls den Typ ID besitzen. Dies bedeutet, dass der Wert dieses Attributs einen eindeutigen Schlüssel für das entsprechende Element innerhalb des Dokuments darstellt. Für ein Element darf nur ein einziges Attribut als ID deklariert werden. Falls im XML-Dokument zwei verschiedene Elemente gleichen Typs ein ID-Attribut haben das den gleichen Wert hat, so wird der XML-Parser einen Fehler zurückliefern.

Mit einer verkürzenden Syntax lassen sich auch mehrere Attribute eines Elements gleichzeitig definieren. Am Beispiel des Zutat-Tags aus Abbildung 1 könnte das wie folgt aussehen:

```
<!ATTLIST Zutat Name CDATA #REQUIRED
                Anzahl CDATA "1"
                Einheit CDATA #IMPLIED>
```

In diesem Beispiel fällt auf, dass der Standardwert in zwei Fällen durch Schlüsselwörter ersetzt worden ist. Dies tut man dann, wenn kein geeigneter Standardwert für das Attribut vorliegt. #REQUIRED drückt aus, dass das Attribut auf jeden Fall bei allen Elementen dieses Typs vorhanden sein muss. Ist dies nicht der Fall, so liefert der XML-Parser einen Fehler zurück. Ist die Attributdefinition mit dem #IMPLIED Schlüsselwort versehen, so ist es dem Autor des Dokuments überlassen, ob er dieses Attribut angeben möchte oder nicht. Das Dokument ist dann in beiden Fällen gültig.

Abschließend zeigt Abbildung 2 die komplette Dokument-Definition des XML-Fragments aus Abbildung 1.

Abbildung 2 Document Type Definition für das XML-Fragment aus Abbildung 1

```
<!DOCTYPE Rezepte [
    <!ELEMENT Rezepte (Data, Rezept*)>
    <!ELEMENT Data (#PCDATA)>
    <!ELEMENT Rezept (Titel, Zutat*, Zubereitung)>
    <!ELEMENT Titel (#PCDATA)>
    <!ELEMENT Zutat EMPTY>
    <!ELEMENT Zubereitung (Schritt*)>
    <!ATTLIST Zutat Name CDATA #REQUIRED
                    Anzahl CDATA "1"
                    Einheit CDATA #IMPLIED>
]>
```

2.3 XML-Parser

XML ist dazu entwickelt worden, Informationen darzustellen, so dass sie von Mensch und Maschine gleichwohl verarbeitet werden können. Wo der Mensch die Struktur eines Dokuments schon beim ersten Blick erkennen kann, ist dies bei einer Maschine etwas aufwendiger.

Das Dokument muss erst von einem so genannten Parser analysiert werden. Dieser Parser hat die Aufgabe, das flache Textdokument in eine Datenstruktur zu überführen, die von einem Rechner leichter verarbeitet werden kann. Es gibt mehrere Verfahren auf ein XML-Dokument zuzugrei-

fen, woraus verschiedene Parser entstanden sind. Die beiden wichtigsten Vertreter, SAX und DOM, werden in den beiden folgenden Abschnitten vorgestellt.

2.3.1 SAX - Simple API for XML

Den wohl einfachsten Parser seiner Art stellt der *SAX Parser* [14] dar. SAX steht für *Simple API for XML* und definiert Schnittstellen für das Parsen von XML-Dokumenten. Die Verarbeitung erfolgt seriell, das heißt, die Elemente werden in der Reihenfolge ihres Auftretens im XML-Dokument behandelt.

Die Anwendung, die das Dokument bearbeiten möchte, stellt dem Parser so genannte Handler zur Verfügung, die speziell vorgegebene Methoden implementieren. Der Parser wird nun das Dokument sequentiell lesen und je nach Element, das er liest, bestimmte Callback-Methoden im jeweiligen Handler aufrufen. So lässt sich die Verarbeitung des XML-Dokuments steuern.

Der wichtigste von SAX definierte Handler ist der *DocumentHandler*. Er definiert die Methoden *startDocument* und *endDocument*, die jeweils zu Beginn und am Ende eines Dokuments aufgerufen werden. Des Weiteren muss ein DocumentHandler die *startElement*()- und *endElement()*-Methoden implementieren, die beim Auftreten eines einleitenden beziehungsweise eines abschließenden Elementtags aufgerufen werden. *startElement* erhält als Parameter die Attribute, die dem Element zugeordnet sind. Beim Auftreten von Text außerhalb von Tags wird die *characters()*-Methode mit dem gelesenen Text als Parameter aufgerufen. Darüber hinaus existieren die Methoden *processingInstruction()* zum Verarbeiten einer Processing Instruction (siehe Abschnitt 2.1.1) und *ignorableWhiteSpace()*, die aufgerufen wird, falls so genannter Whitespace, also Leerzeichen zur Einrückung oder Zeilenumbrüche zur besseren Lesbarkeit des Dokuments aufgetreten sind.

Neben dem DocumentHandler definiert SAX auch den *ErrorHandler* zum Behandeln von Fehlern, die während des Parsens auftreten können, den *DTDHandler* zur Behandlung von *Unparsed Entities* in DTDs und den *EntityResolver* zum Auflösen von Referenzen auf externe Dateien. Da wir uns in dieser Arbeit nicht mit SAX beschäftigen werden, möchte ich an dieser Stelle nicht auf diese Details eingehen. Die genaue Spezifikation und Beispiele zur Anwendung von SAX finden sich in der Literatur [14].

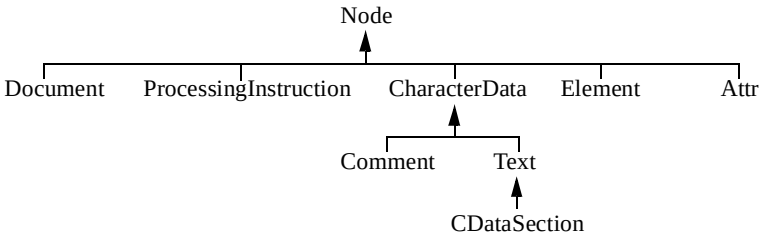
2.3.2 DOM - Document Object Model

DOM ist, genau wie XML selbst vom W3C spezifiziert worden. Wir betrachten hier den so genannten DOM Level 2 [3].

Nach der Analyse des XML-Dokuments durch einen DOM-Parser erhält man den so genannten *DOM-Baum*. Er besteht aus einer Hierarchie von Knotenobjekten, die man mit Hilfe von speziellen Methoden durchlaufen und modifizieren kann. In Abbildung 4 ist der aus dem XML-Fragment aus Abbildung 1 abgeleitete DOM-Baum abgebildet. Die Konstruktion dieses Baums und der Zugriff darauf wird in den nun folgenden Abschnitten beschrieben.

Zum Aufbau des DOM-Baums stellt DOM eine Reihe von Objektypen (Klassen) zur Verfügung, für die es in jeder Programmiersprache eine eigene Implementierung gibt. Abbildung 3 zeigt den für uns relevanten Ausschnitt aus dem von der W3C vorgegebenen Klassenmodell.

Abbildung 3 Klassenmodell der zentralen Klassen von DOM Level 2



Node ist die Hauptklasse, von der alle anderen erben, und implementiert eine Reihe von Methoden, die für die meisten, jedoch nicht alle Unterklassen definiert sind. Dies vereinfacht den Umgang mit dem Klassenmodell erheblich und macht das ständige Umwandeln von *Node* in eine seiner Unterklassen oftmals überflüssig.

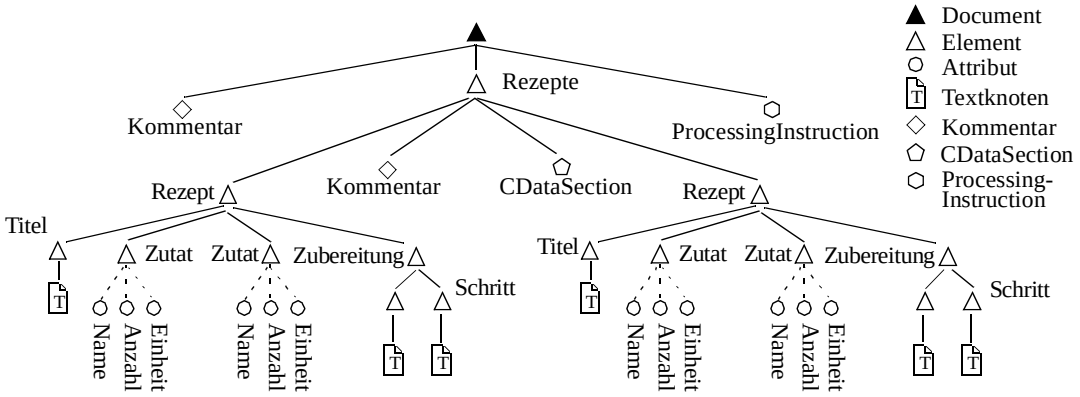
Document repräsentiert das XML-Dokument selbst. Der DOM-Parser liefert als Ergebnis ein solches Objekt als Referenz auf den DOM-Baum zurück. *Document* implementiert ebenfalls Methoden zur Erstellung neuer Knoten, die in den Baum hinzugefügt werden können (siehe Abschnitt 2.4.1).

Element, *Attr* und *ProcessingInstruction* vertreten ihre jeweiligen Tags aus dem XML-Dokument. Dabei ist Attr eine Abkürzung für Attribut.

CharacterData ist die Oberklasse für alle Text-Objekte. *Comment*, *Text* und *CDataSection* sind wiederum Repräsentanten der entsprechenden XML-Tags.

Darüber hinaus existieren *NodeList* und *NamedNodeMap*. Erstere dient zum Zusammenfassen von Elementen, letztere zum Zusammenfassen von Attributen zu einer Liste. Der Einsatz dieser Klassen wird bei der näheren Betrachtung der DOM-Methoden (siehe Abschnitt 2.4) verdeutlicht.

Abbildung 4 DOM-Baum-Darstellung des XML-Fragments aus Abbildung 1



2.3.2.1 Konstruktion des DOM-Baums

Im folgenden Abschnitt werden wir den Vorgang der Konstruktion eines DOM-Baums betrachten. Abbildung 5 zeigt ein Java Code-Fragment zur Erstellung eines DOM-Baums aus einem XML-Dokument.

Abbildung 5 Java Code-Fragment zur Erstellung eines DOM-Baums aus einem XML-Dokument

```
1 | DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
2 | dbf.setValidating(true);
3 | dbf.setIgnoringComments(true);
4 | dbf.setIgnoringElementContentWhitespace(true);
5 | DocumentBuilder db = dbf.newDocumentBuilder();
6 | Document doc = db.parse(new File("xmldocument.xml"));
```

Als erstes wird eine neue DocumentBuilderFactory erzeugt (1). Diese erlaubt es wiederum den eigentlichen DocumentBuilder (oder Parser) zu erstellen (5). Die Eigenschaften des erzeugten Parsers werden über die Factory selbst eingestellt. So ist der in Abbildung 5 erzeugte Parser validierend (2), das heißt, er überprüft während des Aufbaus des DOM-Baums, ob der zugrundeliegende Dokument der entsprechenden Document Type Definiton entspricht. Zusätzlich werden alle Kommentare ignoriert (3), das heißt diese tauchen später nicht mehr als Comment-Knoten im DOM-Baum auf. Dies dient vor allem dazu, wenn man die reinen Daten eines Dokuments verarbeiten möchte und die Kommentare für das menschliche Auge nicht beachtet werden sollen. Darüber hinaus wird der Parser auch überflüssige Leerzeichen und Zeilenumbrüche zwischen zwei Elementen entfernen (4). Diese werden vor allem zur besseren Lesbarkeit für den Menschen hinzugefügt, tragen aber im Allgemeinen nicht zum Inhalt des Dokuments bei.

Nach dem Konfigurieren des Parsers kann das entsprechende Dokument geparst werden. Dabei wird dem Parser ein das XML-Dokument repräsentierendes File-Objekt übergeben. Der *parse()*-Aufruf liefert eine Referenz auf den erzeugten DOM-Baum zurück. Dieser wird durch ein *Document*-Objekt dargestellt. Dieses Document-Objekt enthält einen Zeiger auf das Root-Element des Dokuments. Jedes Element im DOM-Baum enthält fünf Zeiger auf weitere Knoten im Baum: *firstChild* und *lastChild* zeigen jeweils auf den ersten und letzten Kindknoten, *previousSibling* und *nextSibling* auf den linken und rechten Nachbarn und *attributes* auf die Attribute die dem Element zugeordnet sind. Diese Zeiger werden zur Navigation durch den DOM-Baum herangezogen. Wie der Zugriff auf die Knoten im DOM-Baum genau funktionniert wird in Abschnitt 2.4 im Detail erläutert.

2.3.2.2 Speicherverbrauch und Leistungverhalten

Eine sehr wichtige Eigenschaft von DOM ist, dass das komplette Dokument als Datenstruktur in den Hauptspeicher geladen wird. Das hat Vor- und Nachteile. Auf der einen Seite erlaubt es einen sehr schnellen und unverzögerten Zugriff auf die Daten, auf der anderen Seite verhält sich der Speicherverbrauch proportional zu Größe und Komplexität des XML-Dokuments, was sehr schnell zur Erschöpfung des Hauptspeichers führen kann. Ist der Hauptspeicher allerdings erst

einmal voll ausgelastet, so fangen die meisten Betriebssysteme an, Teile davon auf die Festplatten auszulagern. Damit sind wieder alle Leistungsvorteile zunichte gemacht.

Man sollte sich daher darüber im Klaren sein, dass sich DOM besonders gut für kleinere Dokumente eignet, für sehr große Dokumente allerdings wegen des hohen Speicherplatzbedarfs eher unbrauchbar ist.

Manche XML-Parser (zum Beispiel Xerxes von Apache [4]) setzen auf das so genannte *Deferred DOM* Konzept. Dabei wird beim Parsen des Dokuments nicht der komplette DOM-Baum im Speicher aufgebaut, sondern nur ein kleiner Teil davon. Dies beschleunigt erstens das initiale Parsen des Dokuments und verbraucht erheblich weniger Speicher. Möchte ein Benutzer jedoch auf einen Teil des Dokuments zugreifen, der beim ersten Parsen nicht in den DOM-Baum aufgenommen wurde, so muss dieser Teil aus der Datei nachgeladen werden. Dabei kann es zu größeren Verzögerungen kommen, wenn die Datei nicht komplett im Datei-Cache des Betriebssystems vorhanden war.

Da aber im Allgemeinen, und dies ist besonders für größere Dokumente der Fall, ein XML-Dokument selten komplett durchlaufen wird, sondern nur bestimmte Teile der Daten benötigt werden, kann mit Deferred DOM die Leistungsfähigkeit des Systems gesteigert werden. Untersuchungen von Apache haben ergeben, dass sich das verzögerte Nachladen von Teilbäumen nur bei großen Dokumenten lohnt. Bei kleinen Dokumenten (0-10Kb) geht gegenüber dem normalen Parsen sogar Teil verloren.

2.4 DOM-Methoden

Nachdem der DOM-Baum vom Parser aufgebaut worden ist, hat der Benutzer natürlich die Möglichkeit, diesen zu traversieren und zu bearbeiten. Zu diesem Zweck stellt DOM ein API zur Verfügung, dessen wichtigsten Methoden wir nun kurz vorstellen wollen.

2.4.1 Das Document-Interface

Das Document-Interface repräsentiert ein komplettes XML-Dokument. Es stellt die Wurzel des Baumes dar und erlaubt den Zugriff auf die Daten in diesem Baum. Da Elemente, Textknoten, Attribute, Processing Instructions usw. nicht außerhalb eines Dokuments existieren können, enthält das Document-Interface auch die Methoden zur Erstellung dieser Objekte. Folgende Methoden werden durch das Document-Interface definiert:

- `Element getDocumentElement()`

  Liefert ausgehend vom Document-Objekt, welches vom DOM-Parser zurückgegeben worden ist, das Wurzelelement des Dokuments.

- `Element getElementByID(String id)`

  Liefert das Element, das das durch den String Parameter gegebene ID-Attribut besitzt. Das Verhalten dieser Methode ist nicht definiert, falls mehrere Elemente mit derselben ID existieren. Die DOM-Implementierung muss vor dem Aufruf dieser Methode wissen, welches Attribut als eindeutige ID interpretiert werden kann. Auch das Attribut mit dem Namen ‘ID’ ist nur dann ID wenn dies vorher in der entsprechenden DTD (siehe Abschnitt 2.2) so definiert worden ist. Es ist auch möglich für jedes Element ein anderes ID-Attribut zu definieren.

- `Element createElement(String tagname)`

  Erstellt ein neues Element, das dem Dokument später (zum Beispiel mit *insertBefore()*) hinzugefügt werden kann. Der Tagname des zu erstellenden Elements wird als Parameter *tagname* übergeben.

- `Text createTextNode(String data)`

  Erstellt einen neuen Textknoten. Der Wert wird als Parameter *data* übergeben.

- `Attr createAttribute(String name)`

  Erstellt ein neues Attribut mit dem als Parameter *name* übergebenen Tagnamen.

- `Comment createComment(String data)`

  Erstellt einen neuen Kommentar-Knoten. Der Wert wird als *data* Parameter übergeben.

- `ProcessingInstruction createProcessingInstruction(String target, String data)`

  Erstellt eine neue Processing Instruction. Das Ziel und der Wert werden als Parameter übergeben.

2.4.2 Das Node-Interface

Das Node-Interface ist das Hauptinterface im Document Object Model. Ein Node repräsentiert einen Knoten im DOM-Baum. Alle anderen Knotenrepräsentanten leiten von Node ab. Das Node-Interface stellt Methoden zum Zugriff auf Kindknoten bereit, obwohl nicht alle Knotentypen, die von Node ableiten, Kindknoten haben können. Auch der Zugriff auf Werte und Attribute wird vom Node Interface definiert, obwohl er nicht für alle Knotentypen sinnvoll ist. *getAttributes()* ist zum Beispiel nur für Elemente und *getValue()* nur für Attribute und Textknoten definiert. Dieser Mechanismus dient dazu, nicht jedes Mal, wenn ein Node-Objekt vorliegt, dieses in den speziellen Typen umzuwandeln, um die entsprechende Methode aufrufen zu können (*narrowing cast*). Greift eine Methode auf eine Eigenschaft oder einen Unterknoten zu, der nicht existiert, so liefert diese auf jeden Fall *null* zurück. Folgenden Methoden werden durch das Node-Interface definiert:

- `Element getFirstChild()`

  Liefert den ersten Kindknoten des jeweiligen Elements.

- `Element getLastChild()`

  Liefert den letzten Kindknoten des jeweiligen Elements.

- `Element getPreviousSibling()`

  Liefert den nächsten (rechten) Nachbarknoten.

- `Element getNextSibling()`

  Liefert den vorherigen (linken) Nachbarknoten.

- `Element getParent()`

  Liefert den Vorgängerknoten.

- `NodeList getElementsByTagname(String tagname)`

  Liefert eine Liste von allen Elementen im Unterbaum des aktuellen Elements, deren Tagname mit dem String Parameter *tagname* übereinstimmt.

- `String getValue()`

  Liefert den Wert eines Textknotens (auch Kommentar oder CDataSection), eines Attributs oder einer ProcessingInstruction. Für alle anderen Knoten ist die Methode nicht definiert und liefert *null*.

- `NodeList getChildNodes()`

  Liefert eine Liste aller direkten Kindknoten. Falls es keine Kindknoten gibt, ist die Liste leer.

- `NamedNodeMap getAttributes()`

  Liefert eine Liste aller Attribute des jeweiligen Knotens (als *Attr*-Objekte) oder eine leere Liste, falls keine Attribute existieren.

- `Element insertBefore(Node newChild, Node refChild)`

  Fügt den *newChild* Knoten vor den bereits existierenden Kindknoten *refChild* ein. Falls *refChild null* ist, so wird *newChild* am Ende der Nachfolgerknotenliste eingefügt. Befindet sich *newChild* bereits im DOM-Baum, so wird er erst entfernt.
  Die Methode liefert den eingefügten Knoten zurück.

- `Element removeChild(Node oldChild)`

  Löscht den übergebenen Knoten aus der Liste der Kindknoten und liefert ihn zurück.

- `Element replaceChild(Node newChild, node oldChild)`

  Ersetzt *oldChild* durch *newChild* in der Liste der Kindknoten. Ist *newChild* bereits im DOM-Baum enthalten, so wird er erst entfernt.

2.4.3 Das Element-Interface

Da Elemente die einzigen Knoten sind, die Attribute besitzen können, bietet das Element-Interface weitere Methoden zum Bearbeiten der Attribute.

- `String getTagName()`

  Liefert den Tagnamen des Elements.

- `Boolean hasAttribute(String name)`

  Liefert *true*, falls das Element ein Attribut besitzt, dessen Name mit dem übergebenen String übereinstimmt, andernfalls *false*.

- `void setAttribute(String name, String value)`

  Fügt dem Element ein Attribut mit gegebenem Namen und Wert hinzu. Ist bereits ein Attribut mit gleichem Namen vorhanden, so wird der Wert durch *value* ersetzt.

- `void removeAttribute(String name)`

  Entfernt das Attribut mit dem angegebenen Namen. Wurde dieses Attribut zuvor mit einem Standardwert definiert, so wird sofort ein neues Attribut mit diesem Standardwert angelegt. Die Standardwerte von Attributen lassen sich in der DTD definieren (siehe Abschnitt 2.2).

2.4.4 Das Attr-Interface

Das Attr-Interface repräsentiert ein Attribut eines Elements. Seine Methoden erlauben es, auf die Eigenschaften Name und Wert des Attributs zuzugreifen.

- `String getName()`

  Liefert den Namen des Attributs.

- `Element getOwnerElement()`

  Gibt eine Referenz auf das Element zurück, dem dieses Attribut zugeordnet ist.

- `String getValue()`

  Liefert den Wert des Attributs.

2.4.5 Das ProcessingInstruction-Interface

Das ProcessingInstruction-Interface repräsentiert eine Processing Instruction im DOM-Baum. Sie besteht aus einem so genannten Target, einem Identifier für die zu spezifizierende Eigenschaft des Dokuments (zum Beispiel die XML-Versionsnummer, zu der das Dokument konform ist), und den dazugehörigen Wert. Die Methoden des ProcessingInstruction-Interfaces erlauben den Zugriff auf diese Eigenschaften.

- `String getData()`

  Liefert den Wert der Processing Instruction.

- `String getTarget()`

  Liefert das Target der Processing Instruction.

- `void setData(String data)`

  Setzt den Wert der Processing Instruction.

2.4.6 Das CharacterData-Interface

Das CharacterData-Interface erweitert das Node-Interface um einige Zeichenketten-Funktionen. Diese dienen der Verarbeitung von Text in einem XML-Dokument. Das CharacterData-Interface entspricht nicht direkt einem Knotentypen, sondern bildet nur eine Oberklasse für *Comment, CDATASection* und *Text*. Die Methoden von CharacterData müssen somit nicht für jede dieser Klassen einzeln definiert werden.

Die Methoden dienen im Wesentlichen dem Einfügen, Löschen, Ersetzen und Anhängen von Zeichenketten, und verdienen keine besondere Betrachtung in diesem Kontext. Eine genaue Beschreibung des Interfaces befindet sich in der DOM-Definition des W3C [3].

2.4.7 Das Text-Interface

Das Text-Interface identifiziert nur einen Textknoten im DOM-Baum. Gegenüber dem CharacterData-Interface besitzt es nur eine weitere Methode zur Zeichenkettenverarbeitung:

- `Text splitText(int offset)`

  Schneidet den im Textknoten enthaltenen Text an der Stelle *offset* ab, und liefert einen neuen Textknoten mit dem verbleibenden Inhalt zurück. Wenn der ursprüngliche Knoten einen Vorgänger hatte, so wird der neue Knoten als nächster (rechter) Nachbar eingefügt. Ist *offset* gleich der Länge des Textes des ursprünglichen Textknotens, so ist der neue Knoten leer.

2.4.8 Das CDATASection-Interface

Das CDATASection-Interface ist vom Text-Interface abgeleitet und definiert keine weiteren Methoden. Es repräsentiert einen CData-Knoten im DOM-Baum und existiert als neuer Typ, um

die CData-Knoten bei der Verarbeitung des XML-Dokuments von herkömmlichen Textknoten unterscheiden zu können.

2.4.9 Das Comment-Interface

Genau wie das CDATASection-Interface dient auch das Comment-Interface nur zur Unterscheidung gegenüber anderer Knoten, die Text enthalten. Es definiert keine weiteren Methoden.

2.4.10 Das NodeList-Interface

Das NodeList-Interface repräsentiert eine geordnete Menge von Node Objekten. Es wird zum Beispiel beim Aufruf der *getChildNodes()*- oder *getElementsByTagname()*-Methoden verwendet und bietet Methoden zum Zugriff auf diese Knotenmenge.

- `int getLength()`

  Liefert die Anzahl der enthaltenen Knoten zurück.

- `Node item(int index)`

  Liefert eine Referenz auf den Knoten an Position *index* in der Liste.

2.4.11 Das NamedNodeMap-Interface

Das NamedNodeMap-Interface repräsentiert, genau wie das NodeList-Interface eine Menge von Knoten. Der Unterschied liegt aber im Zugriff auf die Elemente dieser Menge. Die Knoten in einer NamedNodeMap werden über ihren Namen referenziert, da eine Reihenfolge der Knoten nicht festgelegt ist. Dies ist zum Beispiel beim *getAttributes()*-Aufruf sinnvoll, da die Reihenfolge der Attribute eines Elements nicht definiert ist. Das NamedNodeMap-Interface definiert eine Reihe von Methoden zum Zugriff über den Namen der Elemente.

- `int getLength()`

  Liefert die Anzahl der in der Menge enthaltenen Knoten zurück.

- `Node getNamedItem(String name)`

  Liefert eine Referenz auf den Knoten mit dem Namen *name*.

- `Node item(int index)`

  Liefert den *index*-ten Knoten in der Menge.

- `Node removeNamedItem(String name)`

  Entfernt den Knoten mit dem Namen *name* aus der Menge.

- `Node setNamedItem(Node arg)`

  Fügt den Knoten *arg* in die Menge hinzu.

Kapitel 3 Vorhandene Ansätze

XML hat in den letzten Jahren einen regelrechten Boom erlebt und sich in sehr vielen verschiedenen Anwendungsgebieten ausgebreitet. Dadurch sind in der jüngsten Vergangenheit Werkzeuge zum Verarbeiten von XML-Daten auf den Markt gekommen. Auch andere Arbeitsgruppen haben sich schon mit dem Thema der effizienten Speicherung von XML-Dokumenten sowie deren Verarbeitung in einer Mehrbenutzerumgebung beschäftigt. Wir wollen uns in diesem Kapitel mit bereits vorhandenen Ansätzen auseinandersetzen und die Arbeiten der verschiedenen Autoren im Einzelnen betrachten.

3.1 Isolation in XML Bases

In dem Artikel von Helmer, Kanne und Moerkotte [11] werden sechs verschiedene Synchronisationsprotokolle beschrieben. Bei den ersten vier betrachteten handelt es sich um pessimistische Sperrverfahren. Später wird auch ein Zeitstempelprotokoll und das so genannte *Dynamic Commit Ordering* Protokoll vorgestellt.

3.1.1 Zwei-Phasen-Sperrprotokolle

Die vier pessimistischen Sperrverfahren basieren alle auf dem wohlbekannten Zwei-Phasen-Sperrprotokoll (2PL / 2-Phase-Locking) und unterscheiden sich lediglich durch die erreichte Sperrgranularität. Es wird nur zwischen zwei Sperrmodi unterschieden: Lese- und Schreibsperren. Die Verwendung der Sperren wird im Folgenden anhand von kleinen Beispielen erklärt.

- **Doc2PL** ist das einfachste Protokoll und sperrt auf Dokumentenebene. Für Applikationen, bei denen jeweils nur eine Transaktion auf einem XML-Dokument arbeitet, ist dieses Protokoll am effizientesten, weil der Overhead für die Sperrverwaltung sehr gering und zugleich sehr einfach zu implementieren ist.
- **Node2PL** führt Knotensperren ein. Dabei werden immer die Vorgänger der betrachteten Knoten mit Sperren versehen. Möchte man zum Beispiel den n-ten Kindknoten eines Knotens *P* traversieren, so wird eine Lesesperre auf *P* angelegt. Möchte man einen neuen Kindknoten einfügen, so wird eine Schreibsperre auf *P* angelegt.
- **NO2PL** erweitert das Node2PL Protokoll, indem es Sperren auf jedem Knoten anlegt der durchlaufen oder verändert wird. Möchte man beispielsweise einen Knoten C_0 vor dem bereits vorhandenen ersten Kindknoten C_1 zu einem Knoten *P* hinzufügen, so sind zwei Schreibsperren erforderlich. Zum ersten muss eine Sperre auf dem Knoten *P* selbst angelegt

werden, da sich das *FirstChild* geändert hat und zum zweiten wird der vor dem Einfügen erste Kindknoten C_1 schreibend gesperrt, da sich dessen *PreviousSibling* geändert hat. Der eingefügte Knoten selbst braucht nicht gesperrt zu werden, da keine andere Transaktion bis zu diesem Knoten navigieren kann. Alle möglichen Zugangspfade sind gesperrt.

- **OO2PL** sperrt, anders als Node2PL oder NO2PL nicht die Knoten selbst, sondern die Kanten die die Knoten verbinden. Jeder Knoten im XML-Dokument (außer der Wurzel) hat vier Kanten zu anderen Knoten, die jedoch nicht alle belegt sein müssen: *FirstChild, LastChild, PreviousSibling* und *NextSibling*. Beim Durchlaufen oder Verändern der Struktur des Baumes werden jeweils die betroffenen Kanten mit Sperren versehen. Soll beispielsweise der erst-Kindknoten *C* von *P* gelöscht werden, so müssen Schreibsperren für die *FirstChild*-Kante von *P* und für die *PreviousSibling*-Kante des rechten Nachbarn angelegt werden.

Wir möchten nun die vorgestellen Protokolle bezüglich der Anzahl der notwendigen Sperren und des erreichten Nebenläufigkeitsgrads vergleichen.

Bei Doc2PL gibt es höchstens eine Sperre pro Transaktion pro Dokument. Damit ist der Overhead für die Sperrverwaltung sehr gering, es kann aber jeweils nur eine Transaktion schreibend auf ein Dokument zugreifen, auch wenn eine zweite Transaktion einen ganz anderen Teilbaum bearbeiten möchte als die erste.

Node2PL und NO2PL legen höchstens eine Sperre pro Knoten pro Transaktion an. Der Unterschied liegt auf der untersten Ebene, der Ebene der Blattknoten des Baumes, wo sich die meisten Knoten befinden. Node2PL legt im Gegensatz zu NO2PL keine Sperren für Blattknoten an. NO2PL benötigt im Extremfall also erheblich mehr Sperren als Node2PL, jedoch ist der Nebenläufigkeitsgrad bei Node2PL geringer weil hier der parallele Zugriff zweier Transaktionen auf unterschiedliche Kindknoten eines Knotens K nicht möglich ist. Eigene Sperren auf den traversierten Knoten beim NO2PL Verfahren würden ein solches Szenario erlauben. Für weitere Beispiele sei auf [11] verwiesen.

3.1.2 Zeitstempelverfahren

Ich möchte an dieser Stelle nur sehr kurz auf das in [11] vorgestellte Zeitstempelverfahren eingehen, da es auf den weiteren Verlauf der Arbeit keinen Einfluss mehr nehmen wird, und ihm aufgrund erheblicher Mängel und Probleme keine große Bedeutung zukommen wird.

Beim Zeitstempelverfahren wird jede Transaktion mit einem Zeitstempel versehen. Dieser Zeitstempel wird an alle Operationen innerhalb dieser Transaktion vererbt. Das Verfahren verwendet implizit mehrere Versionen der XML-Dokumente. So werden von einer Transaktion gelöschte Knoten nur als solche markiert und erst am Ende der Transaktion endgültig entfernt. Gleiches gilt für Einfügeoperationen.

Besondere Beachtung kommt bei diesem Verfahren wieder den Konfliktoperationen zu. Sie treten zum Beispiel dann auf, wenn zwei parallele Transaktionen lesend beziehungsweise schreibend auf denselben Knoten zugreifen möchten. Konfliktoperationen von verschiedenen Transaktionen müssen in der Reihenfolge derer Zeitstempel durchgeführt werden. Kann dies nicht gewährleistet werden, wenn zum Beispiel jüngere Transaktionen auf noch benötigte Knoten einer anderen Transaktion zugreifen, so muss diese Transaktion zurückgesetzt werden.

Durch diese Eigenschaft des Zeitstempelverfahrens werden lange Transaktionen sehr benachteiligt. Es kann sogar passieren, dass eine lange Transaktion immer wieder durch jüngere Transaktionen zurückgesetzt werden muss und unter Umständen nie zum Commit kommt.

Ein weiterer Nachteil besteht darin, dass die Transaktionen in der Reihenfolge ihrer Zeitstempel Commit ausführen müssen. Dies ist notwendig, damit eine jüngere Transaktion nicht Knoten endgültig entfernen kann, die eventuell noch von einer anderen, älteren Transaktion benötigt werden. Da die Knoten durch das Verfahren nicht explizit gesperrt werden, ist es zulässig, dass Transaktionen schmutzige Werte lesen dürfen. Schmutzige Werte sind Veränderungen am Baum, die wegen einer noch nicht abgeschlossenen Transaktion nur vorläufig sind. Daraus folgt, dass keine Transaktion Commit ausführen darf, die schmutzige Werte gelesen hat, und dass solche Transaktionen durch Abbruch der Transaktion, die den schmutzigen Wert geschrieben hat, zurückgesetzt werden können. Damit können lange Ketten von kaskadierenden Rücksetzungen entstehen. Ein solches Verhalten ist nicht wünschenswert und aus diesem Grund werden wir Zeitstempelverfahren im weiteren Verlauf dieser Arbeit nicht weiter berücksichtigen. Das ebenfalls in [11] eingeführte *Dynamic Commit Order* Konzept hebt zwar die Notwendigkeit auf, dass alle Transaktionen strikt in der Reihenfolge der Zeitstempel Commit ausführen müssen, die anderen schwerwiegenden Nachteile bleiben jedoch erhalten.

3.2 Transaction Synchronization for XML Data in Client-Server Web Applications

In dem Artikel von Stefan Böttcher und Adelhard Türling von der Universität Paderborn [2] wird die transaktionale Synchronisation von XML-Daten für Client-Server Web-Applikationen vorgestellt. Der Ansatz basiert auf dem Mapping der XML-Daten auf ein relationales Datenbanksystem. Dabei werden die Daten von den Clients vor dem Verarbeiten durch ein Check-Out vom Server geladen und danach mit einem Check-In wieder gespeichert. Der Server überprüft beim Commit-Vorgang einer Transaktion, ob unauflösbare Konflikte aufgetreten sind, und setzt die Transaktion gegebenenfalls zurück.

3.2.1 Speicherung der XML-Daten

Die Speicherung der XML-Daten findet in diesem Fall in einem relationalen Datenbanksystem statt. Um die XML-Dokumente aber auf ein Relationenschema abbilden zu können, ist eine Abbildungsvorschrift notwendig. Diese muss für jedes Dokument einzeln definiert werden. Die Anfragen an den XML-Server werden dann mit Hilfe dieser Vorschrift in SQL-Anfragen umgewandelt, um auf die Daten im Relationenschema zugreifen zu können.

Möchte man zusätzlich zum lesenden Zugriff auch den schreibenden Zugriff auf die XML-Daten zulassen, so muss die Abbildung auch umkehrbar sein. Die Daten müssen also nicht nur aus dem Relationenschema in XML-Daten überführt werden können, sondern auch umgekehrt.

Ich möchte in diesem Zusammenhang die genauen Algorithmen zur Abbildung nicht vertiefen und verweise hierzu auf [2]. Es sei aber noch angemerkt, dass diese Verfahren nicht unbedingt alle Daten im Relationenschema auf ein XML-Dokument abbilden, und dass mehrere Benutzer durchaus auch verschiedene Abbildungen verwenden können.

3.2.2 Synchronisierung

Der betrachtete XML-Server unterstützt folgende Operationen: *read committed, browse, insert, delete this, delete any, update this* und *update any.*

Beim Browsen durch den XML-Baum wird der Zugriff nicht mit anderen Transaktionen synchronisiert. Das bringt mit sich, dass die gelesenen Daten gegebenenfalls inkonsistent sind, aber in manchen Anwendungsfällen ist die Konsistenz der Daten auch nicht dringend notwendig. Dies ist der Fall, wenn man sich nur einen groben Überblick über die Daten verschaffen möchte (ein so genannter Snapshot), um beispielsweise eine grobe Abschätzung der Anzahl der Datensätze im Dokument zu erhalten. Möchte man aber nicht auf die Konsistenz verzichten, so kann man *read committed* verwenden. In diesem Fall werden alle durchlaufenen Knoten in ein Read-Set aufgenommen.

Insert fügt einen Knoten in das XML-Dokument ein. Dabei wird der eingefügte Teilbaum in das Insert-Set aufgenommen. Das Read-Set wird dabei nicht verändert. Es kann allerdings vorkommen, dass die Einfügeoperationen vom Read-Set anderer Transaktionen abhängen. Außerdem muss gewährleistet sein, dass die eingefügten Knoten die durch die DTD definierte Struktur nicht verletzen.

Delete this entfernt den Wert eines Knotens, der zuvor gelesen wurde. Vor dem Löschen ist also der Wert des zu löschenden Knotens bekannt. Aus diesem Grund muss der Knoten nach dem Ausführen der *delete this* Operation ins Read-Set aufgenommen worden sein. Diese Tatsache unterscheidet *delete this* von *delete any.* In letzterem Fall wird der Wert ohne Kenntnis des vorhandenen Wertes gelöscht. In beiden Fällen wird der Knoten in das Delete-Set der Transaktion aufgenommen.

Update this und *update any* unterscheiden sich in gleicher Weise voneinander wie *delete this* und *delete any.* In beiden Fällen muss der aktualisierte Knoten in das Update-Set der Transaktion aufgenommen werden.

Beim Commit der Transaktion werden Read-Set, Insert-Set, Delete-Set und Update-Set an den Server geschickt. Dieser überprüft die Mengen dann auf Konflikte. Dabei wird zuerst sichergestellt, dass alle im Read-Set befindlichen Knoten denselben Wert haben wie die Knoten in der Datenbank. Ist dies nicht der Fall, so muss die Transaktion abgebrochen werden, weil eine andere Transaktion diese Werte zuvor geändert hat. Wurde dabei kein Konflikt festgestellt, so können die Auswirkungen der Schreiboperationen über die entsprechende Abbildung an die Datenbank propagiert werden. Beim Schreiben der Daten in die relationale Datenbank dürfen keine Constraints verletzt werden. Tritt ein solcher Fall trotzdem auf, wird die relationale Transaktion abgebrochen und es muss dafür gesorgt werden, dass auch die XML-Transaktion abbricht.

Im Falle eines Transaktionsabbruchs kann sich der Client dafür entscheiden, alle Operationen abzubrechen. Dies hat dann keine weiteren Auswirkungen, und die Transaktion gilt endgültig als gescheitert. Er kann aber auch versuchen, das Read-Set zu verkleinern, indem er eine gewisse Knotenmenge nur noch mit der Browse Operation liest anstatt mit Committed Read. Ist dies aus Konsistenzgründen nicht möglich, so kann der Client sein Read-Set durch neue Daten vom Server aktualisieren und somit die Wahrscheinlichkeit einer Abweisung der Commit-Anfrage minimieren.

Dieses optimistische Sperrverfahren hat den Vorteil, dass es verklemmungsfrei ist und eine potentiell hohe Nebenläufigkeit bietet. Auf der anderen Seite kann es aber auch zu massiven

Rücksetzungen von Transaktionen kommen, da dies die einzige Methode ist, Konflikte aufzulösen. Diese Gefahr besteht vor allem für lange Transaktionen, die beispielsweise die Werte einer großen Anzahl von Knoten relativ zum vorherigen Wert ändern müssen. Dazu müssen erst alle Werte gelesen werden. Somit wird das Read-Set der Transaktion schnell sehr groß, und es kann schnell zu Rücksetzungen kommen. Dasselbe Problem stellt sich für häufig geänderte Knoten. Auch hier entstehen schnell Konflikte zwischen zwei (potentiell langen) Transaktionen, die dann aufgrund nur weniger Konflikte, die man leicht mit Warten hätte umgehen können, komplett zurückgesetzt werden müssen.

Diese Nachteile haben uns davon überzeugt, eine solche Vorgehensweise nicht weiter zu untersuchen. Davon abgesehen ist für uns das Aufbrechen von XML-Dokumenten auf ein Relationenschema absolut inakzeptabel.

3.3 XMLTM: Efficient Transaction Management for XML Documents

In dem Artikel von Torsten Grabs, Klemens Böhm und Hans-Jörg Schek [7] wird ein Synchronisierungsverfahren für XML-Dokumente vorgestellt, das auf dem XML-Extender für das DB2 Datenbanksystem von IBM aufbaut.

Dabei werden die kompletten XML-Dokumente in CLOB-Feldern der DB2 abgelegt. Der Zugriff erfolgt über die von XML-Extender vorgesehene Schnittstelle. Damit nicht bei jeder Anfrage das Dokument aus dem CLOB Feld der Datenbank ausgelesen werden muss, wird mit Hilfe von so genannten *Side-Tables* ein Index über die wichtigsten Teile des Dokuments angelegt. So ist es möglich, Anfragen, die nur auf diese Teile des Dokuments zugreifen, zu beantworten, ohne dabei das XML-Dokument selbst anzufassen.

Dieses Vorgehen hat aber wieder den signifikanten Nachteil, dass wegen der *repeatable read* Eigenschaft der Side-Tables, bei Einfüge- oder Löschoperationen Teile des XML-Baums gesperrt werden, die eigentlich nicht betroffen sind. Die von herkömmlichen relationalen Datenbanksystemen zur Verfügung gestellten Synchronisierungsmechanismen sind also nicht ausreichend, um nebenläufige Zugriffe auf XML-Dokumente effizient zu verwalten.

Aus diesem Grund wird ein so genannter *Transaction Manager* eingesetzt, der auf den vorhandenen Schnittstellen des XML-Extenders für IBM DB2 aufsetzt. Die Idee ist, die Verarbeitung von XML-Anfragen auf zwei logische Ebenen zu verteilen. Mehrere Anfragen können zu einer Transaktion auf der Anwendungsebene zusammengefasst werden. Diese globale Transaktion wird dann in mehrere unabhängige Subtransaktionen auf der Datenbankebene aufgebrochen. Diese Datenbanktransaktionen können gegebenenfalls früher Commit ausführen als die Anwendungstransaktion, um somit das Problem von Blockierungen ohne richtigen Konflikt zu vermeiden.

Das vorgestellte *DGLOCK* Sperrprotokoll verwendet so genannte *Data Guides* als zugrundeliegende Struktur zur Verwaltung der Sperren auf Anwendungsebene. Data Guides sind baumartige, nicht notwendigerweise komplette Darstellungen des dazugehörigen XML-Dokuments. Die Knoten dieser Datenstrukur enthalten die Sperren des entsprechenden Knotens im XML-Baum. Das Sperrprotokoll unterscheidet zwischen fünf Arten von Sperren: S, X, IS, IX und SIX. Soll lesend auf einen Knoten im DataGuide zugegriffen werden, so ist eine S-Sperre notwendig, für schreibenden Zugriff eine X-Sperre. IS- und IX-Sperren zeigen das Vorhaben an, eine S- oder X-

Sperre auf einem Knoten weiter unten in der Baumhierarchie anzulegen. SIX ist eine Kombination von S- und IX-Sperren.

Ist eine anzulegende Sperre S_1 mit einer bereits vorhandenen Sperre verträglich, so kann S_1 angelegt werden. Sind die Sperren nicht kompatibel, so heißt das nicht unbedingt, dass die Transaktion blockiert werden muss. Dies ist nur der Fall, wenn Prädikate im Zugriffspfad nicht verträglich sind. Möchte Transaktion T1 mit dem Pfadausdruck **/store/auction[price > 5]/description** alle Beschreibungen der *auction* Objekte abrufen, deren *price* größer 5 ist, und T2 mit dem Pfadausdruck **/store/auction[price < 0.5]/price** alle zutreffenden *price* Objekte auf den Wert 4 setzen, so würde der Transaction Manager beide Transaktionen gleichzeitig ausführen können, da die entsprechenden Prädikate miteinander verträglich sind.

Dieser Ansatz bietet sicherlich einen recht hohen Datendurchsatz und die bekannten Probleme des Parallelitätsverlusts durch das Aufbrechen der XML-Struktur auf ein relationales Datenmodell werden geschickt umgangen. Es hat jedoch den Nachteil, dass die Dokumente praktisch in doppelter oder sogar dreifacher Ausführung vorhanden sein müssen. Zum ersten wird das XML-Dokument als Text in einem CLOB-Feld der Datenbank abgelegt. Um nicht jegliche Leistungsfähigkeit zu verlieren, wird ein Index in Form so genannter Side-Tables angelegt. Zur Sperrverwaltung auf Anwendungsebene muss dann nach zusätzlich ein Data Guide erstellt werden, der die komplette Struktur des XML-Dokuments enthält. Außerdem baut der Ansatz auf dem XML-Extender von IBM DB2 auf, stellt also eine proprietäre Lösung dar.

3.4 NatiXync - Synchronisation für XML-Datenbanksysteme

In der Diplomarbeit von Robert Schiele [15] wird das Synchronisationsprotokoll für Natix vorgestellt. Natix ist ein natives Datenbankmanagementsystem für XML-Dokumente, das an der Universität Mannheim entwickelt wird. Wir möchten in diesem Zusammenhang nur die Synchronisationskomponente betrachten.

Da die XML-Daten in Natix nativ in physischen Sätzen ausgelagert werden, beschäftigt sich die Arbeit von Robert Schiele hauptsächlich mit den damit verbundenen Problemen. Diese entstehen vor allem dadurch, dass die Zuordnung von Sätzen zu Knoten nicht eindeutig ist. Mehrere Knoten eines Teilbaums können in einem physischen Satz abgelegt sein, und die Lokalität der Knoten in den Sätzen zur Laufzeit des Systems bleibt meist nicht konstant. Erreicht beispielsweise ein Knoten durch Änderungsoperationen eine Größe, die nicht mehr im aktuellen Satz untergebracht werden kann, so muss dieser Satz aufgespalten werden, um den Inhalt auf mehreren Sätze zu verteilen. Das Sperrprotokoll sieht aber vor, Sperren ausschließlich auf physischen Sätzen zu vergeben. Deswegen kann es passieren, dass Referenzen mancher Sperren nicht mehr gültig sind, wenn zwischendurch der Satz aufgespalten wurde.

NatiXync basiert auf dem klassischen Zwei-Phasen-Sperrprotokoll, das dem in Abschnitt 3.3 vorgestellten NO2PL ähnelt. Unterschieden werden, genau wie in Abschnitt 3.3 die folgenden Sperrmodi: S, X, IS, IX und SIX. Dazu kommt die Update-Sperre U die vermeiden soll, dass eine schreibende Transaktion durch immer neue lesende Transaktionen auf einem Objekt aushungert. Auf dieses Konzept kommen wir in Abschnitt 4.2 noch einmal zurück. Für die korrekte Abwicklung von direkten Zugriffen auf Knoten (zum Beispiel über Indizes) wird ein weiterer Sperrmodus TOUCH eingeführt. Dieser ist mit allen Sperren kompatibel, außer mit den jenen, die

Änderungen an der Struktur des Baumes erlauben. Die TOUCH-Sperre wird verwendet, um beim direkten Zugriff auf einen beliebigen Knoten im Baum den Weg bis zur Dokumentwurzel zu Sperren. Dadurch soll verhindert werden, dass andere Transaktionen den Teilbaum dieses Knotens strukturell verändern beziehungsweise löschen.

Da die Sperranforderungen bei Natix auf physischen Sätzen stattfinden, muss bei jedem Zugriff auf einen Knoten der umgebende Satz ermittelt werden und mit der der Zugriffsoperation entsprechenden Sperre versehen werden.

- Werden nur textuelle Informationen eines Knotens gelesen, so wird der physische Satz mit einer S-Sperre versehen, der den entprechenden Knoten enthält.
- Werden textuelle Informationen eines Knotens geschrieben, so muss eine X-Sperre auf dem entsprechenden physischen Satz angelegt werden.
- Wird der XML-Baum mit Lese-Operationen traversiert, so müssen alle physischen Sätze mit S-Sperren versehen werden, deren Kntoen einen direkten Zeiger auf den traversierten Knoten enthalten. Ist der betroffene Knoten von Knoten aus anderen physischen Sätzen nicht erreichbar, so muss nur der entsprechende Satz dieses Knotens gesperrt werden.
- Beim Ändern der Struktur des XML-Baumes muss darauf geachtet werden, dass alle Sätze gesperrt werden, die Knoten enthalten, die einen Zeiger auf den hinzugefügten oder gelöschten Knoten besitzen. Falls der eingefügte oder gelöschte Knoten der erste oder letzte Kindknoten ist, so muss der Satz, der den Vaterknoten enthält mit einer X-Sperre versehen werden. Die Nachbarknoten (Brüder) müssen nicht gesperrt werden, da diese durch die Sperre auf dem Vaterknoten sowieso nicht mehr für andere Transaktionen zu erreichen sind. Ist er weder der erste noch der letzte, so müssen der rechte und linke Nachbar gesperrt werden. Der Vaterknoten muss in diesem Fall nicht gesperrt werden, da er keinen direkten Zeiger auf den betroffenen Knoten hat.

Das NatiXync Sperrprotokoll sieht auch einen Sperreskalationsmechanismus vor, der versucht, viele Sperren zu einer übergeordneten Sperre zusammenzufassen, um so die Anzahl der gehaltenen Sperren zu verringern und die Leistungsfähigkeit des Servers zu steigern. Durch eine solche Maßnahme wird aber unweigerlich die Granularität der Sperren vergröbert und somit die mögliche Nebenläufigkeit eingeschränkt. Dies ist aber in Folge der höheren Leistungsfähigkeit des Servers in Kauf zu nehmen. Wir werden diesen Mechanismus in Abschnitt 4.2.4.2 noch genauer untersuchen.

3.5 Efficient Synchronization for Mobile XML Data

In dem Artikel von Franky Lam, Nicole Lam und Raymond Wong [13] geht es um die Synchronisierung von XML-Daten zwischen einem zentralen XML-Server und mehreren mobilen Clients wie zum Beispiel Handhelds oder Mobiltelefonen.

Als Schnittstelle betrachtet man die *XML Query Language* (XQL). Es werden die gewünschten Objekte durch ihre Pfade im XML-Baum adressiert. Dabei können beliebig viele Clients gleichzeitig solche Pfadausdrücke an den Server absetzen, um XML-Daten abzurufen. Diese werden dann beim Client in lokalen Datenbanken abgelegt, um sie von dort aus weiterzuverarbeiten. Dies ist so lange unproblematisch, wie alle Clients nur lesend auf die Daten zugreifen wollen.

Probleme treten aber dann auf, wenn ein oder mehrere Clients ihre lokalen Daten verändern möchten. Da die Clients nicht unmittelbar miteinander verbunden sind, müssen die Änderungen erst an den Server geschickt werden. Dieser überprüft daraufhin, für welche anderen Clients diese Änderungen relevant sind. Als relevant für einen Client C_1 werden die Daten gewertet, die dieser Client vor der Änderung vom Server abgerufen hat, also direkt nach der Änderung durch Client C_2 im Besitz einer nicht mehr aktuellen Kopie dieser Daten ist.

Um die Clients, für die ein bestimmter Datensatz relevant ist, zu identifizieren, muss festgestellt werden, ob der Pfadausdruck der Update-Operation von Client C_2 mit dem Pfadausdruck der Query-Operation von Client C1 überlappt oder einer im anderen komplett enthalten ist. Das Auffinden von Überlappungen zwischen mehreren Pfadausdrücken kann ebenfalls zu einer gesteigerten Leistungsfähigkeit des Servers führen. Wird eine Überlappung zwischen zwei Pfadausdrücken festgestellt, so können diese zu einem zusammengefasst werden und weniger Zugriffe auf die Datenbank benötigt. Auch der Kommunikationsaufwand kann verringert werden, indem gemeinsam verwendete Dokumentenfragmente via Multicast an die Clients verschickt werden.

Die wohl einfachste Möglichkeit, solche Überlappungen aufzusuchen ist, bei jedem neuen Eintreffen einer Anfrage diese mit allen bereits im System befindlichen Pfadausdrücken zu vergleichen. Dies kann aber unter Umständen sehr viel Zeit und Betriebsmittel in Anspruch nehmen, insbesondere wenn die Anzahl der zu vergleichenden Ausdrücke sehr hoch ist.

Um Zeit und Betriebsmittel zu sparen, wird eine ausgeklügelte Indexstruktur vorgestellt, die diesen Vorgang beschleunigen soll. Es handelt sich dabei um einen gerichteten, azyklischen Graphen. Jeder Knoten in diesem Graphen enthält eine Liste von so genannten *cid* (Child Identifier). Jeder Client ist eindeutig durch einen solchen cid bestimmt. Ein Knoten A enthält genau dann eine gerichtete Kante zu einem anderen Knoten B, falls die relevanten Daten aus dem XML-Baum aller durch die cids aus Knoten A identifizierten Clients eine Obermenge jedes durch die cids aus Knoten B identifizierten Clients sind. Stellt man sich den Graphen als Baum vor, in dem gerichtete Kanten nur von einem Knoten zu dessen Kindknoten verlaufen können, so bedeutet dies, dass die relevanten Daten für die cids aus einem Knoten auch relevant sind für alle cids aus dem Vorgängerknoten.

Bei jeder neuen Client-Anfrage durchläuft der Server den Baum von der Wurzel ausgehend nach unten und fügt den cid des entsprechenden Clients zu den Knoten hinzu, die die gleichen Teilbereiche aus dem XML-Baum abgerufen haben, oder fügt einen neuen Knoten als Kindknoten des Knoten hinzu, der diesen Teilbereich im geringsten überlappt, falls es einen solchen Knoten vorher noch nicht gab.

Die vorgestellte Indexstruktur wird dazu verwendet, die von einer Update-Operation betroffenen Clients zu finden. Dabei sucht der Server den Knoten C mit dem cid des Client, der die Update-Operation durchgeführt hat im Überlappungsbaum (dies passiert über eine weitere Index-Tabelle, die aber hier nicht weiter betrachtet werden soll) und sendet die Änderung an alle cids in allen Vorgängerknoten bis hin zur Wurzel des Baums. Für die Knoten im Unterbaum von C muss noch zusätzlich überprüft werden, ob die geänderten Daten für die darin enthaltenen cids relevant sind. Dies funktioniert aber nach demselben Prinzip wie das Einfügen einer neuen Client-Anfrage in den Baum.

Falls ein Client Updates aus mehreren verschiedenen Quellen empfangen soll, so wird der Server versuchen, diese zu einer Operation zusammenzufassen, um den Kommunikationsaufwand zu

minimieren. Dabei können allerdings Konflikte auftreten, falls zwei Clients gleichzeitig den gleichen Knoten oder Teilbaum modifizieren möchten.

Man unterscheidet hier zwischen zwei Arten von möglichen Konflikten. Ein direkter Konflikt impliziert, dass die Reihenfolge, in der die Operationen ausgeführt werden wichtig ist. Dieser Fall tritt beispielsweise dann auf, wenn ein Client einen Knoten aktualisieren möchte, während gleichzeitig ein anderer diesen Knoten löschen will. Die Anfragen werden in der Reihenfolge ihrer Ankunft beim Server serialisiert. Ist das Ausführen einer Operation nicht möglich, so wird eine der in Konflikt stehenden Operationen in folgender Reihenfolge zurückgesetzt: *delete, move, update.*

Die zweite Art der möglichen Konflikte ist der so genannte Syntax-Konflikt. Dieser tritt dann auf, wenn zwei Clients gleichzeitig denselben Knoten beziehungsweise denselben Pfad im XML-Baum aktualisieren möchten. In diesem Fall werden die Operationen in der Reihenfolge ihrer Ankunft im Server ausgeführt.

Das auf dem Check-Out / Check-In Prinzip basierende Verfahren, das hier vorgestellt wurde, mag sich auf den Spezialfall der mobilen Clientgeräte gut anwenden lassen, ist aber für XML-Server, die gleichzeitige Anfragen vieler Clients bearbeiten müssen nicht geeignet. Dies liegt nicht zuletzt daran, dass optimistische Synchronisationsverfahren schnell dazu führen können, dass schreibende Operationen nicht ausgeführt werden können, weil sie immer wieder Konflikte mit anderen, lesenden Operationen hervorrufen und dann zurückgesetzt werden müssen.

Darüber hinaus interessieren wir uns für Systeme, die bezüglich der Synchronisationskomponente das Isolations-Kriterium des Datenbankparadigmas ACID erfüllen. Dies ist bei dem vorgestellten Verfahren in keinster Weise gegeben.

Kapitel 4 taDOM-Sperrkonzept

In diesem Kapitel werden wir im Detail auf das Sperrkonzept für XML-Dokumente eingehen. Dabei definieren wir den gegenüber dem in Kapitel 2 vorgestellen DOM-Baum erweiterten taDOM-Baum, und beschreiben die verschiedenen Sperrtypen, sowie die dazugehörigen Sperrmodi.

4.1 Der taDOM-Baum

Die Synchronisierung von nebenläufigen Zugriffen auf ein XML-Dokument erfordert eine speziell dafür ausgelegte interne Darstellung des XML-Dokuments, um ein feingranulares Anlegen der Sperren zu gewährleisten.

Zu diesem Zweck führen wir zwei weitere Knotentypen ein: *AttributeRoot* und *String*. Diese werden ausschließlich für die interne Darstellung der Sperren benötigt und beeinflussen in keinster Weise die Handhabung des darunterliegenden XML-Dokumentes. Aus diesem Grund wird die Sperrverwaltung in der späteren Implementierung von einer unabhängigen Komponente übernommen (siehe Kapitel 5). Abbildung 6 zeigt den DOM-Baum aus Abbildung 4 auf Seite 10 als taDOM-Erweiterung. Dabei wurden die Comments, CDATASections und ProcessingInstructions zur besseren Übersichtlichkeit entfernt. Diese Knotentypen werden auch in der aktuellen Version des taDOM-Baums nicht unterstützt.

4.1.1 AttributeRoot

Anders als beim herkömmlichen XML-Baum sind die Attribute eines Elements nicht mehr direkt mit dem Element selbst verbunden. Stattdessen verbindet nun der neu eingeführte *AttributeRoot-Knoten* die Attribute mit dem Element. Dies dient dazu, bei einem *getAttributes()* Aufruf (siehe Kapitel 2) nicht alle Attributknoten einzeln sperren zu müssen, sondern alle diese Sperren zu einer Sperre auf dem AttributeRoot-Knoten zusammenfassen zu können. Dieses Vorgehen beeinflusst nicht die Parallelität von nebenläufigen Zugriffen, sondern komprimiert lediglich die Sperrtabellen und führt somit zu einer potentiell höheren Leistungsfähigkeit.

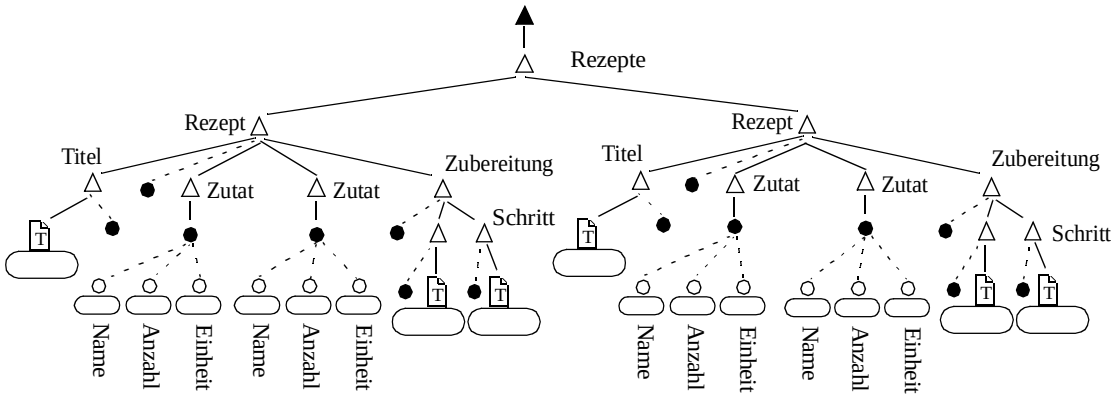
4.1.2 String

Ein weiterer wichtiger Unterschied zum bekannten XML-Baum stellt im taDOM-Baum der String-Knoten dar. Er ist an den zughörigen Text- oder Attributknoten angehängt und enthält

lediglich den Wert dieses Textknotens. Da für die Bestimmung des des Wertes eines Text- oder Attibutknotens ein expliziter *getValue()* Aufruf notwendig ist, braucht der String-Knoten nur dann gesperrt zu werden, wenn ein Benutzer diesen Aufruf auch getätigt hat, nicht aber, wenn dieser nur über die Existenz eines solchen Stringknotens informiert ist, dessen eigentlichen Wert aber noch nicht kennt.

Hierdurch wird gewährleistet, dass die Transaktion eines Benutzers nicht blockiert wird, wenn er nur über einen Textknoten hinwegnavigiert, obwohl eine andere Transaktion vorher den Wert dieses Knotens geändert hat, und somit eine exklusive Sperre auf dem Knoten hält.

Abbildung 6 Erweiterter taDOM-Baum (vergleiche Abbildung 4 auf Seite 10)



4.2 Synchronisierung von Zugriffen

Bisher haben wir die neu eingeführten Knotentypen im taDOM-Baum kennengelernt. Nun wollen wir etwas genauer auf die verschiedenen Sperrtypen und deren Modi eingehen. Zur besseren Übersicht teilen wir die Zugriffsmethoden in drei Kategorien auf: Anfrage-Zugriff, navigierender Zugriff und schreibender Zugriff.

In allen Fällen muss mindestens eine Sperre für den jeweiligen Knoten im taDOM-Baum angelegt werden. Darauf gehen wir in Abschnitt 4.2.1 genauer ein.

Möchte ein Benutzer mit den dafür vorgesehenen Methoden über ein XML-Dokument navigieren, so muss auch der Weg den er dabei zurückgelegt hat, synchronisiert werden. Damit soll vermieden werden, dass ein Benutzer Knoten aus einem einer anderen Transaktion bekannten Pfad herauslöscht oder diesem unbekannte Knoten hinzufügt. In diesem Fall wäre der logische Einbenutzerbetrieb innerhalb einer Transaktion nicht mehr gewährleistet. Um dies zu vermeiden, verwenden wir so genannte *Navigationssperren,* die wir in Abschnitt 4.2.2 behandeln werden.

Ebenso müssen Anfrage-Zugriffe synchronisiert werden um die bekannte *Repeatable Read* Eigenschaft zu erfüllen. Die *getElementsByTagname()*-Methode muss innerhalb einer Transaktion immer dieselbe Menge von Knoten zurückliefern. Deswegen muss verhindert werden dass andere Transaktionen zwischen zwei solchen Aufrufen Knoten einfügen, löschen oder verändern.

Zu diesem Zweck führen wir zusätzlich noch so genannte *logischen Sperren* ein. Diese werden in Abschnitt 4.2.3 erläutert.

Wie bereits erwähnt, streben wir eine dem Benutzer gegenüber transparente Sperrverwaltung an, weswegen wir diese auch später in eine eigene Komponente verlegen werden. Diese Komponente ist dann auch zuständig für die Verwaltung der einstellbaren Sperrgranularität und der Sperreskalation in Abschnitt 4.2.4.

4.2.1 Knotensperren

Beim Navigieren durch den XML-Baum und Ändern einzelner Knoten, müssen einer Transaktion vor dem Zugriff auf den Knoten alle dafür notwendigen Sperren gewährt worden sein. Den von einer Transaktion aktuell bearbeiteten Knoten bezeichnen wir im Folgenden mit *Arbeitsknoten*. Der Sperrmodus der anzufordernden Sperre hängt von der auszuführenden Operation auf dem Arbeitsknoten ab. Da es nicht unbedingt notwendig ist, bei jedem Zugriff auf einen Knoten den kompletten Weg von der Dokumentwurzel bis zum Arbeitsknoten zu durchlaufen, sondern auch (z.B. über die *getElementsByTagname()*-Methode) direkt auf eine beliebige Stelle im Baum zugegriffen werden kann, müssen die Sperren von der Dokumentwurzel bis zum Arbeitsknoten herab vergeben werden. Die Sperren auf den aufsteigenden Knoten dienen lediglich dazu, zu vermeiden dass andere, nebenläufige Transaktionen inkompatible Sperren auf einem der Vorgängerknoten erhalten.

4.2.1.1 Sperrmodi für Knotensperren

Um den Anforderungen einer effizienten Sperrverwaltung zu genügen, unterscheiden wir zwischen sieben verschiedenen Knotensperrmodi. Abbildung 7 zeigt die entsprechende Kompatibilitätsmatrix. Diese ist eine Erweiterung zu den so genannten DAG-Sperrmechanismen (*directed acyclic graph)* [9].

Abbildung 7 Sperrkompatibilitätsmatrix für Knotensperren.

Die bereits existierende Sperre auf einem Knoten ist in der ersten Zeile, die angeforderte Sperre in der ersten Spalte dargestellt. Ein '**+**' in der entsprechenden Zelle bedeutet dass die angeforderte Sperre mit der bereits existierenden kompatibel ist, und somit ohne Konflike angelegt werden darf. Steht ein '**-**' in der Zelle, so kann die angeforderte Sperre erst angelegt werden, wenn die existierende entfernt wurde.

	IX	NR	CX	LR	SR	U	X
IX	+	+	+	+	-	-	-
NR	+	+	+	+	+	-	-
CX	+	+	+	-	-	-	-
LR	+	+	-	+	+	-	-
SR	-	+	-	+	+	-	-
U	+	+	+	+	+	-	-
X	-	-	-	-	-	-	-

- Die **NR-Sperre** (Node Read) sperrt einen einzelnen Knoten im Lesemodus. Eine NR-Sperre auf dem Arbeitsknoten erfordert ebenfalls NR-Sperren auf jedem seiner Vorgängerknoten.
- Der **LR**-Modus (Level Read) dient dazu, den Arbeitsknoten, sowie alle dessen direkte Kindknoten für den Lesezugriff zu-sperren. Dies ist zum Beispiel bei einer *getChildNodes()* Anfrage vorteilhaft, da lediglich eine einzige Sperre auf dem Arbeitsknoten, anstatt einer NR-Sperre auf jedem einzelnen Kindknoten angefordert werden muss. In gleicher Weise ist eine LR-Sperre auf einem AttributeRoot-Knoten gleichwertig mit NR-Sperren auf allen Attributen.
- Die **SR-Sperre** (Subtree Read) erweitert die LR-Sperre auf den kompletten Unterbaum des Arbeitsknotens.
- Um einen Knoten verändern zu dürfen, muss vorher eine exklusive **X-Sperre** auf dem jeweiligen Knoten angefordert werden. Eine exklusive Sperre ist immer die einzige Sperre auf einem Knoten.
- Um eine X-Sperre auf dem Arbeitsknoten gewähren zu können, muss auf dem Vorgängerknoten eine **CX-Sperre** (Child Exclusive) vorhanden sein. Dies soll verhindern dass eine andere Transaktion eine LR-Sperre auf dem Vorgängerknoten hat, wodurch das anlegen der X-Sperre auf dem Arbeitsknoten unzulässig wäre.
- Um auszuschließen dass eine andere Transaktion mit einer SR-Sperre einen kompletten Unterbaum, zu dem auch der exklusiv zu-sperrende Arbeitsknoten gehört, gesperrt hat, müssen vor der Vergabe der X-Sperre ebenfalls **IX-Sperren** auf allen Knoten oberhalb des Vaterknotens bis zur Dokumentwurzel angelegt werden.
- Schließlich verwenden wir die **U-Sperre** um zu vermeiden, dass nach einer gescheiterten Anforderung einer X-Sperre von anderen Transaktionen weitere Lese-Sperren auf dem Knoten angefordert werden können. Dies soll verhindern, dass die schreibende Transaktion immer wieder durch Leseoperationen anderer blockiert wird und so ‘verhungern’ müsste.

4.2.1.2 Beispiel

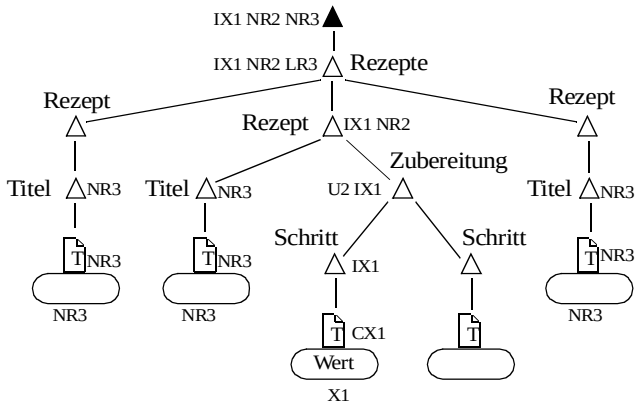
Das folgende Beispiel zeigt den Ablauf der Knotensperranforderungen für drei nebenläufige Transaktionen.

Transaktion T_1 beginnt damit, den Wert des Textknotens zu ändern. Dafür benötigt sie eine X-Sperre auf dem dazugehörigen String Knoten. Nun müssen auch CX- beziehungsweise IX-Sperren auf den Vorgängerknoten bis hin zur Dokumentwurzel angelegt werden.

Gleichzeitig versucht Transaktion T_2 den <*Zubereitung*> Knoten zu löschen (in dessen Unterbaum auch der von Transaktion T_1 geänderte String-Knoten liegt). Dafür braucht Transaktion T_2 aber eine X-Sperre auf dem ensprechenden <*Zubereitung*> Knoten. Diese Sperre kann allerdings nicht gewährt werden, da sich auf diesem Knoten schon eine IX-Sperre von Transaktion T_1 befindet, und diese inkompatibel mit der beantragten X-Sperre ist. Um Transaktion T_2 aber vor dem ‘aushungern’ zu schützen, wird anstatt der X-Sperre nun erst einmal eine U-Sperre angelegt, damit keine anderen Transaktionen diesen Knoten im Lesemodus sperren können.

Unterdessen erstellt eine dritte Transaktion T_3 eine Liste von allen Rezepten und deren Titeln und fordert eine LR-Sperre auf dem <*Rezepte*> Knoten an um Leserechte auf allen direkten

Abbildung 8 taDOM-Baum nach den Sperranforderungen von drei nebenläufigen Transaktionen.



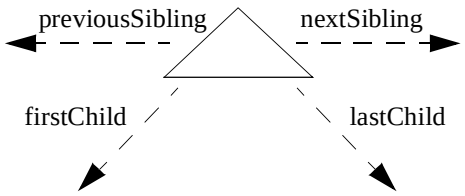
Kindknoten zu erhalten. Um die Titel der Rezepte lesen zu dürfen, sind Sperren von jedem <*Rezept*> Knoten bis hinunter zum entsprechenden String-Knoten erforderlich.

Abbildung 8 zeigt die Sperrsituation nach den Anforderungen der Transaktionen T_1, T_2 und T_3.

4.2.2 Navigationssperren

Bisher haben wir ausschließlich Knotensperren, die beim direkten Zugriff auf einen Knoten angefordert werden müssen, betrachtet. Darüber hinaus erlaubt es das DOM-API auch, durch ein XML-Dokument zu navigieren. Da ein zurückgelegter Pfad innerhalb einer Transaktion exakt reproduzierbar sein muss, ist es unablässig diesen Pfad angemessen zu-sperren um das Einfügen oder Löschen von Knoten in oder aus diesem Pfad zu verhindern. Dafür führen wir *Navigationskanten* (siehe Abbildung 9) und die dazugehörigen *Navigationssperren* ein. Diese sind allerdings nur für Elemente oder Textknoten zulässig. Für Attribute existiert ein solcher Navigationsmechanismus nicht, da für diese keine Reihenfolge spezifiziert ist.

Abbildung 9 Kanten eines Knotens die für Navigationssperren zu berücksichtigen sind



4.2.2.1 Sperrmodi für Navigationssperren

Beim Navigieren durch ein XML-Dokument muss eine Transaktion eine Kantensperre für jede durchlaufene Kante anfordern. Bei einer horizontalen Navigation (nextSibling, previousSibling) ist darauf zu achten, dass die Kante von beiden Seiten gesperrt wird. So muss beim Aufruf von

getNextSibling() auf dem Arbeitsknoten nicht nur dessen nextSibling-Kante, sondern auch die previousSibling-Kante des Nachfolgers (falls dieser existiert) gesperrt werden.

Abbildung 10 Sperrkompatibilitätsmatrix für Navigationssperren.

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

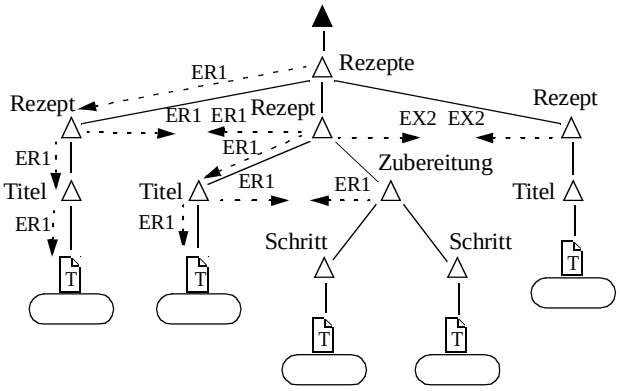
Anders als bei den Knotensperren ist für die Kantensperren das wohlbekannte RUX-Sperrverfahren ([9], [12]) ausreichend. Die hierbei verwendeten Sperren ER, EU und EX (siehe Abbildung 10) werden folgendermaßen eingesetzt:

- Eine **ER-Sperre** (Edge Read) muss angefordert werden, wenn von einem Knoten zu dessen Geschwisterknoten navigiert wird. Dabei sind alle betroffenen Kanten mit einer ER-Sperre zu belegen.
- Bevor eine Kante geändert werden darf, muss zuvor eine **EX-Sperre** angefordert worden sein. Eine Kante kann nur durch Einfügen oder Löschen von Knoten an der jeweiligen Stelle geändert werden. Nach dem Einfügen oder Löschen muss darauf geachtet werden dass die Kanten zwischen den Knoten entsprechend angepasst werden.
- Die **EU-Sperre** soll, wie schon bei den Knotensperren, das Verhungern schreibender Transaktionen verhindern.

4.2.2.2 Beispiel

Das folgende Beispiel zeigt den Ablauf von Kantensperranforderungen für zwei nebenläufige Transaktionen.

Abbildung 11 taDOM-Baum nach den Sperranforderungen von zwei nebenläufigen Transaktionen.



Transaktion T_1 ruft erst drei Mal hintereinander die *getFirstChild()*-Methode auf dem <*Rezepte*> Knoten auf (durchläuft also die Knoten in folgender Reihenfolge: <*Rezepte*>, <*Rezept*>,

<Titel>, *<Text>*) um den Wert des ersten Rezepttitels zu erhalten. Daraufhin ruft sie die *getNextSibling()*-Methode auf dem ersten *<Rezept>* und anschließend erneut zwei Mal die *getFirstChild()*-Methode auf, um auch den Wert des zweiten Rezepttitels zu erhalten. Nun sei das gesuchte Rezept gefunden, und durch einen *getNextSibling()* Aufruf auf *<Titel>* kann auf die entsprechende *<Zubereitung>* zugegriffen werden. Dieses Sperrkonzept erlaubt es nun einer zweiten, nebenläufigen Transaktion T_2 einen weiteren *<Rezept>* Knoten einzufügen, durch Anlegen von zwei EX-Sperren auf den beiden letzten *<Rezept>* Knoten.

Abbildung 11 zeigt die Sperrsituation nach den Anforderungen der Transaktionen T_1 und T_2.

4.2.3 Logische Sperren

Das DOM-API stellt weiterhin die Methoden *getElementsByTagname()* und *getElementById()*, so genannte Query-Methoden zur Verfügung, und liefert die Elemente nach ihrem Tagnamen beziehungsweise nach einem ID-Attribut zurück. Die *hasAttibute()*-Methode überprüft, ob ein Attribut mit dem angegebenen Namen existiert.

Wie es das Transaktionsparadigma verlangt, müssen diese Aufrufe innerhalb einer Transaktion mit gleichem Ergebnis wiederholbar sein (*Repeatable Read*). Das Löschen von Knoten aus der Ergebnismenge durch eine andere Transaktion wird durch NR-Sperren auf den Knoten selbst verhindert. Damit ist von vorne herein ausgeschlossen dass bei einem zweiten Aufruf der Methode, diese weniger Knoten zurückliefert als beim ersten Mal. Anders sieht es aber mit dem Einfügen von neuen Knoten aus. Bisher verbietet keine Sperre das Einfügen eines Knotens der das Prädikat der Query-Methode (zum Beispiel den Tagnamen bei einem *getElementsById()* Aufruf) erfüllt. Dies ist jedoch ebenfalls unzulässig, da ein zweiter Aufruf der Methode eine größere Knotenmenge zurückliefern würde als der erste Aufruf. Um aber nun diese so genannten Phantome zu vermeiden, führen wir logische Sperren ein.

Man hätte in diesem Fall auch ein gröberes Sperrgranulat verwenden können, beispielsweise den Knoten, auf dem die *getElementsByTagname()*-Methode aufgerufen wurde, mit einer SR-Sperre belegen. Dies würde aber zu einer erheblichen Einschränkung der Parallelität sowie erhöhter Deadlock-Gefahr führen. Besonders für die *getElementByID()*, die nur auf der Dokumentwurzel angewendet werden kann, würde dies das Sperren des gesamten Dokuments erzwingen, obwohl maximal nur ein einziger Knoten betroffen ist.

4.2.3.1 Logische Sperrtabellen

Um nun diese potentiellen Phantome effizient zu vermeiden, können wir nicht, wie bei den Knoten- und Kantensperren auf den taDOM-Baum zurückgreifen. Dies erfordert eine etwas komplexere Handhabung. Deswegen führen wir drei Sperrtabellen ein, die die logischen Sperren verwalten. Die Sperrkompatibilität innerhalb jeder solchen Tabelle ist durch das wohlbekannte RUX Verfahren gegeben. Im Folgenden seien die Sperrtabellen im einzelnen beschrieben:

- Die *LocksTagnameQuery* Tabelle enthält Sperren, die für alle Tagnamen-Anfragen angelegt werden müssen. Der Aufruf der *getElementsByTagname()*-Methode erfordert eine R-Sperre für die entsprechende Transaktion, Tagnamen und Knoten-ID des Knotens auf dem der Aufruf durchgeführt wurde. Das Einfügen eines neuen Knotens erfordert eine X-Sperre für den entsprechenden Tagnamen in der *LocksTagnameQuery* Tabelle, sowohl für den Knoten dem ein Kindknoten eingefügt wurde, als auch für alle dessen Vorgänger bis hin zur Dokumenten-

wurzel. Dadurch wird sichergestellt, dass die Ergebnismenge sich nicht ändert. Das Löschen von Knoten wird nach wie vor durch Knotensperren (siehe Abschnitt 4.2.1) verhindert.

Abbildung 12 Aufbau der LocksTagnameQuery Sperrtabelle

LocksTagnameQuery			
TID	Sperre	Knoten-ID	Tagname
...	R/U/X	...	...

- Aufrufe der *hasAttribute()*-Methode werden mit Hilfe der *LocksAttributeQuery* Tabelle synchronisiert. Auch hier darf sich das Ergebnis (*true* oder *false*) innerhalb einer Transaktion nicht ändern. Dies erfordert das Anlegen einer R-Sperre für das entsprechende Attribut bevor das Ergebnis zurückgeliefert werden kann. Das Einfügen von Attributen erfordert somit eine X-Sperre für den neuen Attributnamen. Zusätzlich muss vor jedem Löschen eines Attributs eine X-Sperre für den entsprechenden Attributnamen angefordert werden, da die *hasAttibute()-Methode* lediglich *true* oder *false* zurückliefert, jedoch nicht eine R-Sperre auf dem Attibut selbst anlegt.

Abbildung 13 Aufbau der LocksAttributeQuery Sperrtabelle

LocksAttributeQuery			
TID	Sperre	Knoten-ID	Attribut
...	R/U/X	...	...

- Die *LocksIDQuery* Tabelle enthält die Sperren von ID-Attribut Anfragen. Da diese Methode nur auf der Dokumentwurzel aufgerufen werden kann, ist es nicht notwendig, eine Knoten-ID in die Tabelle aufzunehmen. Suchen, einfügen und löschen erfordert das gleiche Vorgehen wie bei der *LocksTagnameQuery* Tabelle.

Abbildung 14 Aufbau der LocksIDQuery Sperrtabelle

LocksIDQuery		
TID	Sperre	ID
...	R/U/X	...

Weiterhin kann das Einfügen von kompletten Teilbäumen zu Problemen bei der Synchronisierung führen. Es muss darauf geachtet werden, dass für jeden Knoten in diesem Teilbaum eine X-Sperre in der *LocksTagnameQuery* Tabelle, und für jedes ID-Attribut eine X-Sperre in der *LocksIDQuery* Tabelle angelegt wird. Dies ist notwendig, da durch das Einfügen von neuen Knoten

oder Attributen die *Repeatable Read* Eigenschaft für Aufrufe der *getElementsByTagname()* und *getElementById()* verletzt werden kann.

4.2.3.2 Beispiel

Das folgende Beispiel zeigt die Auswirkungen von verschiedenen Methodenaufrufen auf die logischen Sperrtabellen.

Abbildung 15 Beispielhafte Werte für die LocksTagnameQuery und LocksIDQuery Sperrtabellen

LocksTagnameQuery			
TID	Sperre	Knoten-ID	Tagname
2	R	4711	Schritt

LocksIDQuery		
TID	Sperre	ID
1	R	2
1	R	4

Dabei sucht Transaktion T_1 erst nach dem Knoten mit der ID 2 und der ID 4. Transaktion T_2 sucht nach allen <*Schritt*> Knoten unterhalb des <*Zubereitung*> Knotens mit der Knoten-ID 4711. Abbildung 15 zeigt die logischen Sperrtabellen nach diesen Sperranforderungen.

Eine weitere Transaktion T_3 wird nun blockiert wenn sie versucht, einen Knoten mit der ID 4 in den Baum einzufügen, beziehungsweise einen weiteren <*Schritt*> zu der <*Zubereitung*> mit der Knoten-ID 4711 hinzuzufügen. Eine solche Operation würde eine X-Sperre auf den jeweiligen logischen Sperrtabellen erfordern, die jedoch nicht gewährt werden kann.

4.2.4 Einstellbare Sperrgranularität und Sperreskalation

In den vorhergehenden Abschnitten haben wir die Vergabe von Sperren im Detail betrachtet. Es fällt aber auf dass auch für wenige Operationen auf dem DOM-Baum potentiell sehr viele Sperren angefordert, und diese dann auch von anderen, nebenläufigen Transaktionen auf Kompatibilität überprüft werden müssen. Ein solches Vorgehen garantiert zwar einen maximalen Grad an Parallelität, hat aber im Allgemeinen einen großen Einfluss auf Leistungsfähigkeit und Durchsatz des Datenbanksystems. Oftmals nimmt man aber Einbußen im Hinblick auf Parallelität in Kauf, wenn man dadurch die Leistung des Systems signifikant steigern kann. Wir stellen im Folgenden nun zwei solcher Konzepte vor, die es erlauben, die Anzahl der gehaltenen Sperren pro Transaktion drastisch zu reduzieren: *Einstellbare Sperrgranularität* und *Sperreskalation.*

4.2.4.1 Einstellbare Sperrgranularität

Die Sperrgranularität stellt die Feinheit der Sperren dar, die beim Arbeiten auf einem XML-Baum angelegt werden müssen. Je feiner die Granularität ist, desto mehr Sperren müssen angefordert werden um einen gleichwertigen Sperrumfang gegenüber gröberen Sperren zu erreichen.

Das Granulat einer Sperre definieren wir als die Entfernung des Knotens, auf dem die Sperre angelegt wird, zu der Dokumentwurzel, das heißt deren Tiefe im XML-Baum. Jede größer diese Entfernung ist, desto feingranularer ist die Sperre.

Nun werden wir jeder Transaktion erlauben, für deren Sperren die maximale Granularität, also die maximale Sperrtiefe (die so genannte *lock depth*), zu definieren. Alle feineren Sperren werden auf dem letzten Knoten oberhalb dieser Sperrtiefe zusammengefasst. Wir verwenden dafür, im Falle von Lesesperren, eine SR, und im Falle von Schreibsperren, eine X-Sperre. Dies erlaubt es einer Transaktion, frei durch einen Teilbaum (der tiefer liegt als die maximale Sperrtiefe) zu navigieren, ohne dafür zusätzliche Sperren anfordern zu müssen.

4.2.4.2 Sperreskalation

Nach einem ähnlichen Prinzip funktioniert auch die Sperreskalation. Dabei kann der Benutzer zwei weitere Parameter einstellen, *Escalation Threshold* und *Escalation Depth.*

Der Lockmanager durchläuft den taDOM-Baum in festgelegten Intervallen, und zählt dabei die Anzahl der gehaltenen Sperren pro Transaktion sowie die Gesamtzahl der Knoten für jeden Teilbaum im taDOM-Baum dessen Wurzel weiter von der Dokumentwurzel entfernt ist, als vom Escalation Depth Parameter spezifiziert worden ist. Überschreitet dabei das Verhältnis von gehaltenen Sperren gegenüber den gesamten Knoten eines Teilbaums den durch Escalation Threshold spezifizierten Wert, so werden alle Sperren der entsprechenden Transaktion in diesem Teilbaum entfernt, und durch eine passende Sperre an der Wurzel ersetzt.

Lesesperren und Schreibsperren im Unterbaum werden jeweils durch eine SR und X-Sperre an der Unterbaumwurzel ersetzt.

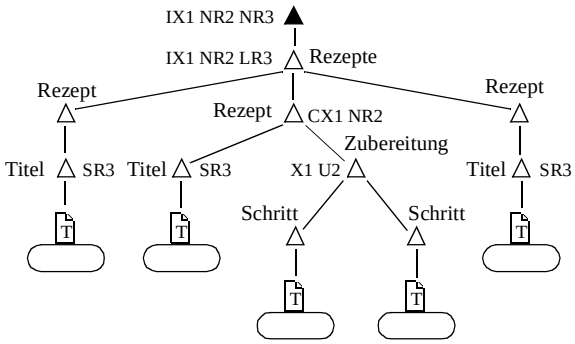
4.2.4.3 Beispiel

Im folgenden Beispiel soll die Arbeitsweise der Sperreskalation erläutert werden. Wir gehen dabei von der Situation aus Abbildung 8 aus.

Die maximale Sperrtiefe sei hier auf 3 festgelegt. Dadurch werden alle NR-Sperren unterhalb der <*Titel*> Knoten durch eine SR-Sperre auf demselben ersetzt, da die Entfernung zur Wurzel (also die Sperrtiefe) genau drei beträgt. Gleichermaßen werden die X-, CX- und IX-Sperren der Transaktion T1 unterhalb des <*Zubereitung*> Knotens durch eine X-Sperre auf demselben ersetzt. Dadurch müssen auch CX- beziehungsweise IX-Sperren auf den Vorgängerknoten angelegt werden. Dies wurde bereits in Abschnitt 4.2.1 im Detail erörtert.

Bemerkung: Die Auswirkungen der *Lock Depth* und *Escalation Depth* Parameter auf den taDOM-Baum sind genau äquivalent. Einziger Unterschied ist, dass sie beim *Lock Depth* Parameter synchron mit dem Zugriff sichtbar werden, beim *Escalation Depth* jedoch erst nachdem der Lockmanager den Baum durchlaufen hat.

Abbildung 16 taDOM-Baum aus Abbildung 8 nach dem Durchführen der Sperreskalation

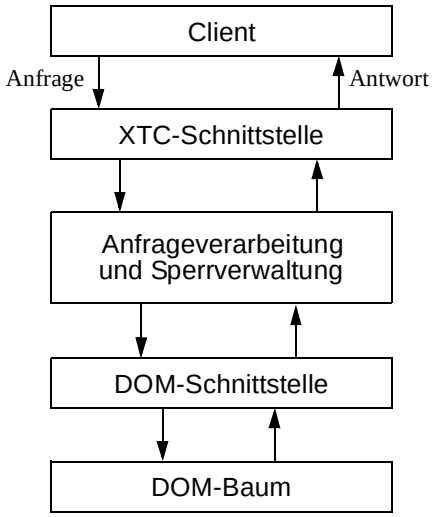


Kapitel 5 Implementierungsaspekte

Im letzten Kapitel haben wir das taDOM-Sperrkonzept im Detail erläutert. Nun wollen wir eine Prototyp-Implementierung dieses Konzepts durchführen, um neue Erkenntnisse über das Verhalten in unterschiedlichen Umgebungen zu gewinnen. Dabei verwaltet ein Server das XML-Dokument und synchronisiert die nebenläufigen Zugriffe mehrerer Clients. Als Programmiersprache haben wir uns für Java entschieden, da Leistungsuntersuchungen nur relativ zueinander unternommen werden und absolute Messwerte nur von zweitrangiger Bedeutung sind.

Wir werden die taDOM-Implementierung als Synchronisationsschicht über den DOM-Zugriffsstrukturen ansiedeln. Abbildung 17 zeigt den schematischen Ablauf einer Client-Anfrage. Der Client benutzt die so genannten XTC-Schnittstellen (*XML Transaction Coordinator*), um seine Anfrage an den Server abzusetzen. Die Sperrverwaltung überprüft, ob der Zugriff auf die gewünschten Knoten im DOM-Baum gewährt werden darf und legt die notwendigen Sperren an. Daraufhin wird über die DOM-Schnittstelle das gewünschte Objekt aus dem DOM-Baum geholt und über den gleichen Weg wieder zurückgeliefert.

Abbildung 17 Ablauf der Verarbeitung einer Clientanfrage



In den nächsten Abschnitten definieren wir die XTC DOM-Schnittstellen, die an das bekannte DOM-API angelehnt sind (Abschnitt 5.1). Dann präsentieren wir zunächst eine einfache, grob-

granular synchronisierte Systemarchitektur (Abschnitt 5.2). Anschließend werden wir diese Architektur etwas verändern, um eine erhöhte Parallelität der Transaktionen zu erreichen (Abschnitt 5.3). Am Ende dieses Kapitels stellen wir die Clients zum Testen des XTC-Servers (Abschnitt 5.4) und einen LockMonitor zum Visualisieren der Sperren (Abschnitt 5.5) vor.

5.1 XTC-Schnittstellen

Um die Transaktionsverwaltung für die Clients möglichst transparent zu gestalten, führen wir die XTC DOM-Schnittstellen ein. Sie sind den ursprünglichen Schnittstellen von DOM sehr ähnlich, jedoch etwas eingeschränkter, um den Implementierungsaufwand nicht explodieren zu lassen. Die XTC-Schnittstellen kapseln die Verbindung zum Server derart, dass der Client wie auf einem lokalen Dokument arbeiten kann.

Die Schnittstelle zwischen Client und Server werden wir mit Hilfe von Java-RMI realisieren. Dies erlaubt es dem Client, ohne jeglichen Zusatzaufwand Objektmethoden von entfernten (also in unserem Fall auf dem Server liegenden) Objekten aufzurufen und deren Ergebnisse lokal zu verarbeiten. Um das Kommunikationsprotokoll zwischen Client und Server besser zu verstehen, erklären wir an dieser Stelle kurz die Funkionsweise von RMI.

5.1.1 Java RMI

RMI steht für *Remote Method Invocation* und erlaubt es, verteilte Java Programme mit der gleichen Syntax und Semantik wie nicht-verteilte Programme zu entwickeln.

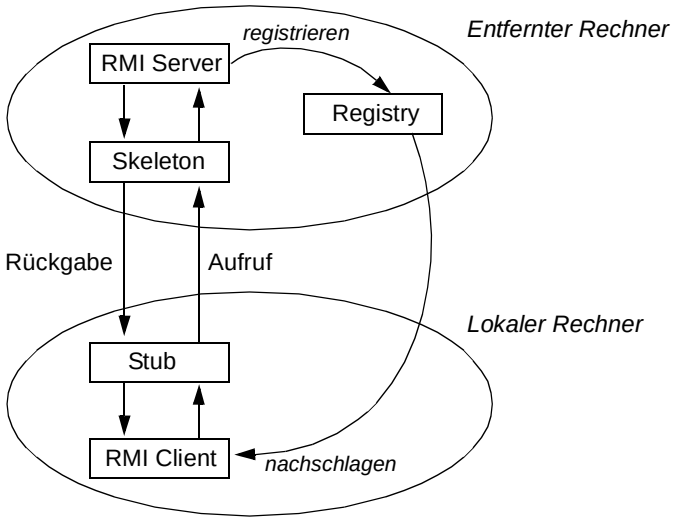
Die Architektur von RMI basiert auf dem so genannten Proxy Pattern [8] und ist in Abbildung 18 dargestellt. RMI benutzt einen wohlbekannten Mechanismus für die Kommunikation mit entfernten Objekten: *Stubs* und *Skeletons*. Ein Stub ist für den Client wie ein lokaler Repräsentant oder Proxy für das eigentliche Objekt. Die Methode wird auf dem Stub aufgerufen, der dafür verantwortlich ist, diesen Aufruf bis zum entfernten Objekt zu propagieren.

Der Stub stellt eine Verbindung zum Serverobjekt her, serialisiert die Parameter des Aufrufs und schickt diese zum Server. Er wartet bis der Server eine Antwort schickt, deserialisiert den Rückgabewert oder die Exception und gibt diese an den Client zurück.

Auf der Serverseite existiert im Allgemeinen ein Skeleton, das Gegenstück zum Stub, der seinerseits den Methodenaufruf entgegennimmt, die Parameter deserialisiert, die Methode auf dem echten Objekt aufruft und die Antwort wieder an den Stub zurückschickt. Seit Java 2 SDK ab Version 1.2 wurde ein erweitertes Stub-Protokoll entwickelt, das die Verwendung von Skeletons überflüssig macht.

Damit ein Client überhaupt Methodenaufrufe auf einem entfernten Objekt durchführen kann, muss er erst in den Besitz des passenden Stubs gelangen. Der Server legt diese mit einem Bezeichner bei einem dafür geeigneten Naming Service ab. Wir verwenden zu diesem Zweck die *RMI Registry* von SUN. Die Registry läuft als Dienst auf einem so genannten ‘well-known’ Port, das heißt man geht davon aus, dass der Client die Registry leicht finden kann, soweit der Host dieses Dienstes bekannt ist. Mit Hilfe eines Naming-Lookup mit dem entsprechenden Bezeichner kann sich der Client die für die Kommunikation mit dem Server notwendigen Stubs aus der Registry besorgen. Die Registry funktioniert also nur als Zwischenspeicher für die Stubs.

Abbildung 18 RMI Architektur



5.1.2 Das XTC Package

Das XTC Package stellt sämtliche Schnittstellen bereit, die ein Client für die Bearbeitung von XML-Dokumenten auf dem Server heranziehen kann. Es besteht aus Java Interfaces, die die Schnittstellen zu den entfernten Objekten des Servers beschreiben. Wir werden nun auf jedes dieser Interfaces kurz eingehen.

5.1.2.1 XTCDOMRMI

XTCDOMRMI ist das Hauptobjekt, das dem Client den Zugang auf den Server gestattet. Aus diesem Grund muss sich der Client den Stub aus der Registry laden. Es existiert pro Server genau ein XTCDOMRMI Objekt. Für jede Referenz, die sich ein Client aus der Registry besorgt, wird auf der Serverseite ein neuer Thread angelegt, der die Methodenaufrufe entgegennimmt und ausführt.

Es stellt dem Client lediglich eine einzige Methode, *beginTransaction()* zur Verfügung, die auf dem Server einen neuen Transaktionskontext anlegt und dem Client über die zurückgelieferte Referenz Zugriff auf diesen Kontext gibt.

Somit hat der Client die Möglichkeit, mit einer Instanz von XTCDOMRMI beliebig viele Transakionen zu verwalten.

5.1.2.2 XTCTransaction

Durch den Aufruf von *beginTransaction()* erhält der Client den Stub eines XTCTransaction Objekts. Es handelt sich also auch hier um ein Remote-Objekt.

Diese Transaktion stellt wiederum zwei Methoden zur Verfügung: *getDocument()* und *commit()*. Letztere dient dazu, den Kontext auf Serverseite wieder aufzulösen, was mit sich führt, dass alle dieser Transaktion zugeordnete Sperren entfernt werden. Danach sind keine weiteren Operatio-

nen innerhalb dieser Transaktion mehr möglich. Nur das Browsen auf dem lokal aufgebauten DOM-Baum (siehe Abschnitt 5.1.2.4) ist erlaubt. Jede Operation dessen Ergebnis nicht zwischengespeichert wurde, also an den Server propagiert werden müsste, führt zu einem Fehler. Durch Aufruf der *getDocument()-Methode* wird der Server veranlasst, das XML-Dokument mit angegebenem Namen zu öffnen, zu parsen, und den entsprechenden DOM-Baum im Speicher aufzubauen (falls nicht schon geschehen) und einen Handle in Form eines *XTCDocument*-Objekts auf dieses XML-Dokument zurückzugeben. Alle XML-Dokumente, die den Clients verfügbar gemacht werden sollen, befinden sich in einem Verzeichnis auf der Server-Seite. Der Name dieses Verzeichnisses wird über die XTC-Server Konfigurationsdatei (siehe Abschnitt 5.2.7) festgelegt.

Auf auf eine konkrete Implementierung einer *rollback()*-Methode wurde bewusst verzichtet, da die Realisierung der dazu notwendigen Logging- und Recovery-Methoden den Rahmen dieser Arbeit überschreiten würde. Im Folgenden sei *rollback()* als Synonym für *commit()* zu betrachten, da hierbei nur die gehaltenen Sperren einer Transaktion frei gegeben werden.

5.1.2.3 XTCDocument

Die XTCDocument Objekte liegen wiederum auf dem Server, und der Zugriff auf die Methoden erfolgt über RMI. An dieser Stelle findet nun auch der Schnitt zwischen Client und Server statt. Das XTCDocument Objekt bietet alle Methoden, die zur Verarbeitung eines XML-Dokuments notwendig sind. Alle Methoden, die sich auf einen bestimmten Knoten beziehen, werden mit der ID dieses Knotens als Parameter aufgerufen. Die ID eines Knotens ist dessen interne Identifikationsnummer, nicht zu verwechseln mit den ID-Attributen (siehe Abschnitt 5.2.4). Interne Identifikationsnummern werden nur für Element-, Attribut- und Textknoten vergeben.

Möchte ein Client beispielsweise die Kindknoten des Elements mit der internen Identifikationsnummer 4711 erhalten, so ruft er *getChildNodes(4711)* vom entsprechenden XTCDocument auf.

Dieser hierarchische Aufbau der Client-Schnittstelle ermöglicht es dem Benutzer, mehrere nebenläufige Transaktionen gleichzeitig zu verwalten, sowie innerhalb einer Transaktion mehrere XML-Dokumente zu bearbeiten.

Es ist allerdings zu beachten, dass die Anfragen aller Transaktionen die durch *beginTransaction()* von demselben XTCDOMRMI Objekt erhalten wurden, innerhalb eines Threads auf der Serverseite abgearbeitet werden. Ist also eine Transaktion blockiert, so können auch die anderen Transaktionen keine Anfragen mehr absetzen. Es ist also anzuraten, sich für jede Transaktion ein eigenes XTCDOMRMI Objekt aus der Registry zu holen, um sicherzustellen, dass diese komplett unabhängig voneinander arbeiten können.

5.1.2.4 XTCNode

Um die Client-Schnittstelle an die DOM-Schnittstelle anzugleichen, führen wir weitere Klassen als Repräsentanten der Knoten des Baums ein. Dies hat den Vorteil, dass man als Rückgabewert von Anfragemethoden nicht die IDs der Knoten erhält, sondern konkrete Objekte, die man zur Weiterverarbeitung heranziehen kann.

XTCNode ist das Super-Interface aller nachfolgenden Schnittstellen (vergleiche *Node* in Abschnitt 2.4.2). Jedem XTCNode ist das XTCDocument, zu dem es gehört als Attribut zugeordnet. Jede XTCNode-Methode muss den Aufruf nur an das entsprechende XTCDocument Objekt

delegieren. So würde zum Beispiel der *getChildNodes()* Aufruf auf einem XTCNode mit der ID 4711 über den *getChildNodes(4711)* Aufruf an das XTCDocument an den Server weitergegeben.

Somit ist es möglich, eine an das *org.w3c.dom* Package angelehnte Struktur nachzubilden. Die implementierten Methoden gehen aus der Definition des *org.w3c.dom.Node* hervor. Genau wie das Node Interface aus dem DOM-Package, werden auch Methoden zur Verarbeitung von Kindknoten zur Verfügung gestellt, obwohl diese nicht für alle Sub-Typen zulässig sind. Dies verhindert jedoch das oftmals lästige Umwandeln von XTCNode Objekten in die jeweiligen Sub-Typen.

Da aber mit der vorhandenen Client-Server-Architektur jeder Aufruf einer Methode auf einem lokalen XTCNode Objekt zu einer Anfrage auf dem Server führt, ist es sinnvoll, einen lokalen Cache für Anfrageergebnisse anzulegen. Damit wird vermieden, dass gleiche Anfragen mehrfach an den Server propagiert werden. Dazu enthält jedes XTCNode Objekt eine HashMap, die die bereits vom Server übertragenen Informationen speichert und im Falle einer erneuten Anfrage ohne Serverkontakt zurückgibt. Es sollen die Ergebnisse folgender Methodenaufrufe zwischengespeichert werden: *getFirstChild(), getLastChild(), getNextSibling(), getPreviousSibling(), getParent, getAttributes()* und *getChildNodes().* Als Schlüssel verwenden wir den Namen der Methode, deren Ergebnis als Value in der HashMap abgespeichert wird. Mit Hilfe der *get()* und *containsKey()-Methode*n lässt sich unterscheiden, ob der Wert bereits vom Server übertragen wurde, oder ob dieser den Wert *NULL* hat. Wurde der Wert noch nicht übertragen, so liefert *containsKey()* für die jeweilige Operation *false*. Nur in diesem Fall wird die Anfrage an den Server weitergeleitet. Nachdem das Ergebnis in der HashMap abgespeichert wurde, liefert *containsKey()* nur noch *true.*

Man beachte, dass diese Methode auch funktioniert, wenn das Anfrageergebnis des Servers gleich *null* ist. Würde man die Ergebnisse in lokalen Variabeln speichern, wäre es nicht möglich, die Zustände ‘Wert ist gleich *null*’ und ‘Wert wurde noch nicht beim Server angefordert’ zu unterscheiden.

Im folgenden Beispiel 1 gehen wir davon aus, dass ein Client mittelbar oder unmittelbar hintereinander zweimal die *getChildNodes()-Methode* aufruft. Der Ablauf der Anfrage sei hier stichpunktartig aufgelistet.

Beispiel 1 Verwendung der HashMap zur Zwischenspeicherung von Anfrageergebnissen

- 1. Aufruf von *getChildNodes()*
- HashMap.containsKey(„getChildNodes“) -> liefert *false*
- Document.getChildNodes(*eigene Knoten-ID*)
- HashMap.put(„getChildNodes“, *Anfrageergebnis vom Server)*
- Anfrageergebnis an den Client zurückliefern
- 2. Aufruf von *getChildNodes()*
- HashMap.containsKey(„getChildNodes“) -> liefert *true*
- HashMap.get(„getChildNodes“) an Client zurückliefern

5.1.2.5 XTCElement

Das *XTCElement* Objekt repräsentiert jeweils ein Element des XML-Dokuments. Es ist von *XTCNode* abgeleitet und stellt weitere Methoden zur Verarbeitung von Attributen bereit.

Des Weiteren werden die Methoden zur Handhabung von Kindknoten überladen. Dies ist wichtig, weil nicht nur die Anfragen an den Server weitergegeben werden müssen, sondern auch die lokale Kopie des XTCElements gewartet werden muss. So muss nach jedem *getFirstChild(), getLastChild(), getNextSibling()* oder *getPreviousSibling()* darauf geachtet werden, dass alle Referenzen auf alle Nachfolger- als auch Nachbarknoten korrekt sind. Die Struktur ist bekanntlicherweise ein Baum, für den alle Nachfolger eines Knotens durch eine doppelt verkettete Liste miteinander verbunden sind.

So muss zum Beispiel bei einem *getChildNodes()* Aufruf für alle Knoten aus dem Ergebnis, der Parent-Link auf den aktuellen Knoten gesetzt werden. Der FirstChild- und LastChild-Link des aktuellen Knotens müssen jeweils auf den ersten beziehungsweise letzen Knoten aus der Ergebnismenge zeigen. Darüber hinaus muss auch für jeden dieser Knoten der PreviousSibling- und NextSibling-Link auf den jeweilgen linken beziehungsweise rechten Nachbarn verweisen.

5.1.2.6 XTCTextnode

Der *XTCTextnode* repräsentiert einen Textknoten im XML-Dokument. Da dieser nur aus einem Wert besteht, bietet das Interface zusätzlich zu den XTCNode Methoden nur *getValue()* und *setValue()*. Dieser Wert wird auch in der HashMap des XTCNode abgespeichert, um mehrfache Anforderungen an den Server zu vermeiden.

5.1.2.7 XTCAttribute

Das *XTCAttribute* repräsentiert einen Attributknoten im XML-Dokument. Wie der XTCTextnode verfügt er ebenfalls über Methoden zum Verarbeiten seines Wertes. Zusätzlich kann der Attributname abgefragt werden. Dieser kann jedoch nicht geändert werden, weil der Attributname das Attribut identifiziert.

5.1.3 RMI Verbindungsverlust

Wir wollen uns in diesem Abschnitt noch kurz mit einem nicht unwesentlichen Problem der Kommunikation beschäftigen. Da der Client über RMI mit dem Server verbunden ist und alle Sperren erst dann freigegeben werden, wenn der Client die *commit()-Methode* aufruft, stellt sich die Frage: Was passiert wenn der Client kein *commit()* ausführt und sich ohne Vorwarnung beendet (zum Beispiel durch Terminierung der virtuellen Maschine des Clients oder durch einen Systemabsturz) ?

Der Client unterhält für jedes Remote-Object, auf das er eine Referenz besitzt eine TCP-Verbindung mit dem Server. Beendet sich der Client, so wird auch der Socket auf Client-Seite geschlossen. Dies kann der Server allerdings nicht ohne weiteres feststellen, was zur Folge hat, dass die von diesem Client gehaltenen Sperren nicht automatisch freigegeben werden können, obwohl der Client nicht mehr existiert.

Genau zu diesem Zweck unterstützt das RMI-Protokoll einen Ping-Pong-Mechanismus, der dazu dient, festzustellen, ob der Referenzhalter, also im Allgemeinen ein Client, oder die Netzwerk-

verbindung zu diesem Client noch existiert. Durch Setzen der Java-Systemeigenschaft *java.rmi.dgc.leaseValue* veranlasst man den Server dazu, in regelmäßigen Abständen *Ping-Nachrichten* an alle Clients zu schicken. Trifft nach der durch *leaseValue* spezifizierten Zeit keine Antwort beim Server ein, so wird die Referenz als ungültig markiert, und das Objekt gegebenenfalls beim nächsten Durchlauf des Garbage Collectors aus dem Adressraum der virtuellen Maschine entfernt.

Da die Sperren möglichst schnell nach Erkennen der verlorenen Verbindung zum Client freigegeben werden sollen, ist es äußerst unvorteilhaft, auf den Garbage Collector warten zu müssen, zumal dieser in unregelmäßigen Abständen und zu unvorhersehbaren Zeitpunkten von der virtuellen Maschine aufgerufen wird. Java RMI bietet aber die Möglichkeit, mit der Remote-Object Implementierung das *Unreferenced* Interface zu implementieren. Dies führt dazu, dass die virtuelle Maschine implizit die *unreferenced()*-Methode aufruft, falls keine Referenz mehr auf das Remote-Object verweist.

Dieser Mechanismus wurde für XTCTransaction benutzt, sodass, wenn die Verbindung zum Client, der eine Referenz auf dieses XTCTransaction Objekt hat, abbricht, implizit die *unreferenced()*-Methode aufgerufen wird. Diese ruft dann *rollback()* auf, wodurch alle Sperren dieser Transaktion sofort freigegeben werden können.

5.2 Datenbankserver

Der Datenbankserver bildet einen eigenen Thread auf der Serverseite und ist für die komplette Abarbeitung einer Anfrage zuständig. Dabei entnimmt er die Aufträge seriell aus einer Warteschlange, und legt die Antwort nach der Bearbeitung wieder in einer Warteschlange beim Client ab.

Die Bearbeitung erfolgt in zwei großen Schritten:

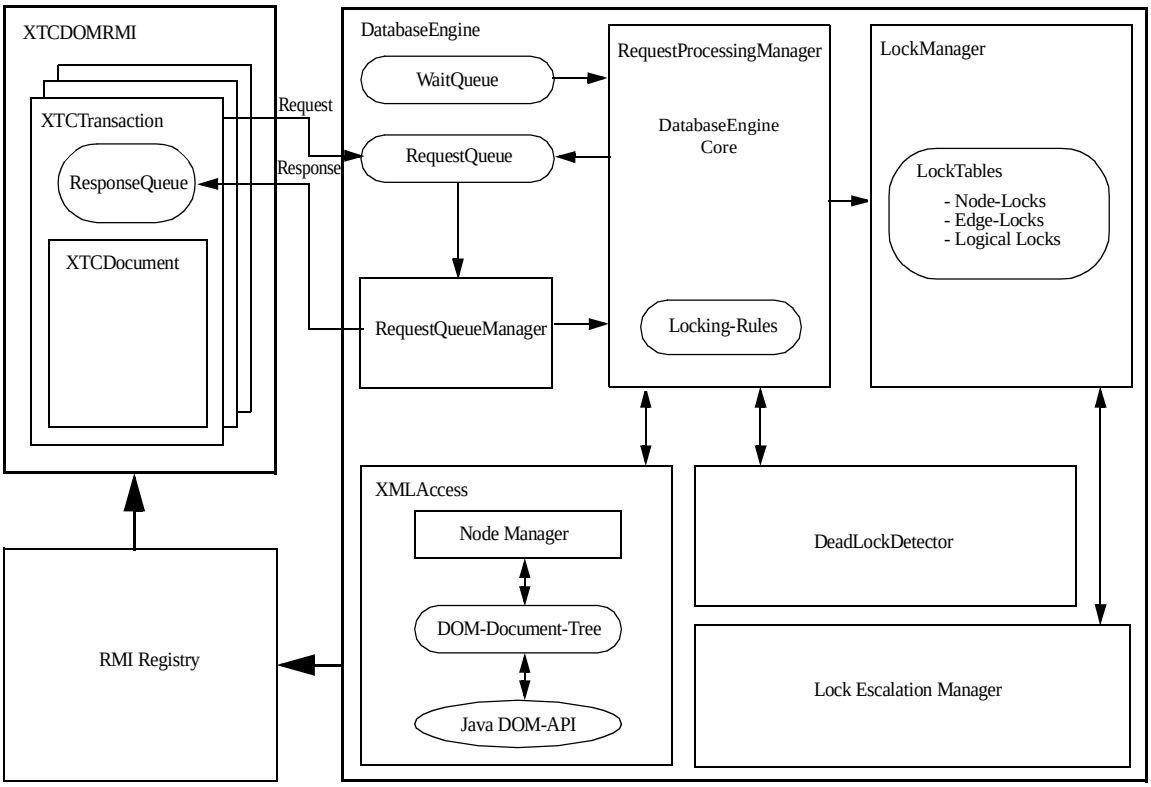
1. Bestimmen und Anlegen aller notwendigen Sperren.
2. Holen der angeforderten Knotenmenge aus dem DOM-Baum beziehungsweise angeforderte Änderungen in den DOM-Baum propagieren.

Um eine bessere Übersichtlichkeit und Wartbarkeit zu erreichen, werden diese Aufgaben verschiedenen Unterkomponenten innerhalb des Datenbankservers zugeteilt. Die Systemarchitektur ist in Abbildung 19 dargestellt. Die Komponenten werden wir nun einzeln und im Detail beschreiben.

5.2.1 RequestQueueManager

Der RequestQueueManager synchronisiert die Zugriffe auf die Warteschlange. Diese Synchronisierung ist notwendig, da mehrere Threads gleichzeitig auf die Warteschlange zugreifen werden. Zum einen sind es die Clients, die asynchron ihre Requests ablegen, zum anderen der Datenbankprozess, der die Requests in der Reihenfolge ihrer Ankunft wieder abholt und sequentiell verarbeitet. Wir werden zwei QueueManager mit verschiedenen Eigenschaften implementieren: den *LazyQueueManager* und den *BusyQueueManager.*

Abbildung 19 Übersicht über die Systemarchitektur



Der LazyQueueManager synchronisiert Clients und Server durch Thread-Operationen. Sind zu einem gegebenen Zeitpunkt alle Requests abgearbeitet und die Warteschlange ist leer, so wird durch einen *wait()*-Aufruf der Datenbankserver-Thread in den Wartezustand versetzt und nimmt erst dann die Arbeit wieder auf, wenn ein Client einen neuen Request in die Warteschlange legt. Dabei wird der Server-Thread durch einen *notifyAll()*-Aufruf wieder aufweckt. Dieses Verfahren ist sehr betriebsmittelsparend, da sich der Server-Thread im Falle einer leeren Warteschlange in einem passiven Wartezustand befindet und keine Betriebsmittel verbraucht. Es hat aber auch den Nachteil, dass eventuell zwischen jedem ankommenden Request der Server-Thread vom aktiven in den passiven Zustand und wieder zurück überführt werden muss. Dadurch kann wertvolle Zeit verloren gehen.

Genau dieses Problem verhindert der BusyQueueManager. Er synchronisiert die Zugriffe auf die Warteschlange mit einem *Busy-Wait*. Dabei läuft der Server in einer Schleife und überprüft die Warteschlange auf neue Requests. Ist ein solcher eingetroffen, so kann er sofort verarbeitet werden, ohne Zeit für das Aufwecken des Server-Threads zu verlieren. Das Busy-Wait hat aber den gravierenden Nachteil, dass der Server bei einer leeren Warteschlange, diese ununterbrochen auf neue Requests überprüft und er somit, im Quasi-Leerlauf, also ohne einen Request zu bearbeiten, die CPU-Leistung voll ausnutzt.

Um das Zeitverhalten des Servers mit beiden Implementierungen zu messen, wird das Umschalten zwischen den beiden QueueManagern über die Konfigurationsdatei (siehe Abschnitt 5.2.7)

ermöglicht. Demnach ist es nur durch Neustart des Servers möglich, den verwendeten QueueManager zu wechseln.

5.2.2 RequestProcessingManager

Der RequestProcessingManager nimmt die Anfragen in der Reihenfolge der Ankunft vom RequestQueueManager entgegen und kontrolliert deren Ablauf. Zuerst werden, entsprechend der Operation des Requests die Sperren beim LockManager angefordert. Dies läuft in zwei Phasen ab:

Zunächst bestimmt der RequestProcessingManager die für die jeweilige Operation notwendigen Sperren nach der nachfolgenden Tabelle 1. Die Sperren werden zu einer Liste zusammengefasst und zur Überprüfung auf Konflikte mit bereits existierenden Sperren anderer Transaktionen an den LockManager weitergegeben.

Erst wenn feststeht, dass bei keiner der zu überprüfenden Sperren ein Konflikt aufgetreten ist, wird der LockManager beauftragt, diese Sperren anzulegen. Dies ist notwendig, um zu verhindern, dass erst eine Teilmenge der notwendigen Sperren angelegt wird, die dann beim Auftreten eines späteren Konflikts wieder entfernt werden müssten. So wird sichergestellt, dass alle Sperren einer Operation atomar angelegt werden.

Daraufhin kann sich der RequestProcessingManager über die XMLAccess-Schnittstelle (siehe Abschnitt 5.2.4) den angeforderten Knoten holen beziehungsweise die entsprechende Änderungsoperation durchführen.

Eine Transaktion darf nur eine Methode auf einem Knoten ausführen, falls sie eine Knotensperre auf diesem Knoten besitzt. Um zu vermeiden, dass Transaktionen auf beliebige Knoten im DOM-Baum zugreifen (durch Angabe ihrer ID), überprüft der RequestProcessingManager, ob eine entsprechende Sperre für diese Transaktion vorhanden ist. Dies ist genau dann gegeben, falls die Transaktion eine beliebige Knotensperre auf diesem Knoten hat, eine LR-, U- oder X-Sperre auf dem direkten Vorgängerknoten oder eine SR-, U- oder X-Sperre auf einem beliebigen Vorgängerknoten auf dem Weg zur Wurzel des Dokuments.

Da jede Operation eine unterschiedliche Menge von Knoten bearbeitet, ist auch die Menge der anzufordernden Sperren jeweils unterschiedlich. Der genaue Ablauf für jede Operation ist in der folgenden Tabelle 1 aufgelistet. Im Falle eines Konfliks wird die Anfrageverarbeitung abgebrochen und der Request in eine Warteschlange aufgenommen. Dadurch ist der Client blockiert, weil der Rückgabewert seines Funktionsaufrufs ausbleibt. Der Request wird erst dann neu bearbeitet, wenn die blockierende Transaktion *commit()* aufruft und somit ihre Sperren wieder freigibt.

Hat beispielsweise T_1 mit *setValue()* den Wert eines Textknotens verändert, und kurz darauf (vor dem Commit von T_1) möchte T_2 diesen Wert lesen, so wird der *getValue()* Request von T_2 solange blockiert, bis T_1 die Transaktion mit Commit abschließt. Dadurch wird der veränderte Wert auch für andere Transaktionen sichtbar, der *getValue()* Request von T_2 kann ausgeführt werden.

Tabelle 1 Sperranforderungen für die jeweiligen Operationen. Als Arbeitsknoten bezeichnen wir den Knoten, auf dem eine Methode aufgerufen wird.

Operation	Sperre	Objekte der Sperranforderung
getRoot	NR	Dokumentwurzel
getFirstChild	R	firstChild-Kante des Arbeitsknotens
	NR	erster Kindknoten des Arbeitsknotens
	NR	alle Vorgängerknoten
getLastChild	R	lastChild-Kante des Arbeitsknotens
	NR	letzter Kindknoten des Arbeitsknotens
	NR	alle Vorgängerknoten
getNextSibling	R	nextSibling-Kante des Arbeitsknotens
	R	previousSibling-Kante des Nachbarn
	NR	alle Vorgängerknoten
getPreviousSibling	R	previousSibling-Kante des Arbeitsknotens
	R	nextSibling-Kante des Nachbarn
	NR	alle Vorgängerknoten
getParent	NR	alle Vorgängerknoten
getElementsByTagname	NR	alle zutreffenden Knoten
	NR	alle Vorgänger aller zutreffenden Knoten
	R	logische Sperre für den entsprechenden Tagnamen
getElementById	NR	zutreffenden Knoten
	NR	alle Vorgängerknoten
	R	logische Sperre für das entsprechende ID-Attribut
getValue	R	Stringknoten des entsprechenden Text- oder Attibutknotens
	NR	alle Vorgängerknoten
getChildNodes	LR	Arbeitsknoten
	NR	alle Vorgängerknoten
getAttribute	NR	AttributeRoot
	NR	Attributknoten
	NR	alle Vorgängerknoten
getAttributes	LR	AttributeRoot
	NR	alle Vorgängerknoten
insertBefore	X	logische Sperre für die Tagnamen aller Knoten des eingefügten Unterbaums
	X	logische Sperre für die ID-Attribute aller Knoten des eingefügten unterbaums
	X	previousSibling-Kante des Referenzknotens
	X	nextSibling-Kante des zukünftigen linken Nachbars
	X	eingefügter Knoten
	CX	Arbeitsknoten
	IX	alle Vorgängerknoten
appendChild		kann durch Übergabe von *null* als refChild Parameter durch insertBefore ersetzt werden

Operation	Sperre	Objekte der Sperranforderung
removeChild	X	logische Sperre für die Tagnamen aller Knoten des entfernten Unterbaums
	X	logische Sperre für die ID-Attribute aller Knoten des entfernten Unterbaums
	X	previousSibling-Kante des rechten Nachbarn
	X	nextSibling-Kante des linken Nachbarn
	X	zu entfernender Kindknoten
	CX	Arbeitsknoten
	IX	alle Vorgängerknoten des Arbeitsknotens
replaceChild	X	logische Sperre für die Tagnamen aller Knoten des ersetzten und des eingefügten Unterbaums
	X	logische Sperre für die ID-Attribute aller Knoten des ersetzten und des eingefügten Unterbaums
	X	previousSibling-Kante des rechten Nachbarn
	X	nextSibling-Kante des linken Nachbarn
	X	zu ersetzender Kindknoten
	CX	Arbeitsknoten
	IX	alle Vorgängerknoten des Arbeitsknotens
addAttribute	X	Attributknoten
	X	logische Attributsperre
	CX	AttributeRoot des Arbeitsknotens
	IX	alle Vorgängerknoten
removeAttibute	X	Attributknoten
	CX	AttributeRoot des Arbeitsknotens
	IX	alle Vorgängerknoten

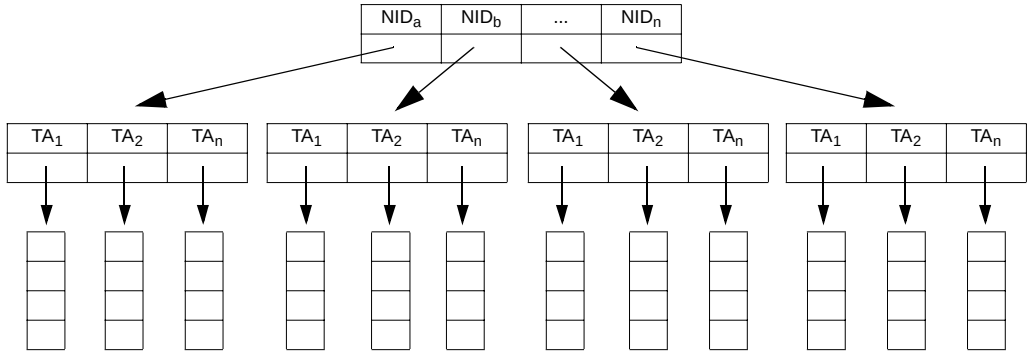
5.2.3 LockManager

Der LockManager ist für die Verwaltung der Sperren zuständig. Anders als es die Abstraktion des taDOM-Baums vermuten lassen würde, baut der LockManager keinen weiteren kompletten Baum auf, um die Sperren zu verwalten. Diese werden vielmehr in einer dreistufigen Tabellenstruktur abgelegt, was einen sehr schnellen Zugriff auf die existierenden Sperren zulässt.

Die Datenstrukur zum Speichern der Sperren ist in Abbildung 1 dargestellt. Die erste Ebene der Speicherungsstrukur bildet eine Hash-Tabelle, die mit den IDs der Knoten indiziert ist, denen die Sperren zugeordnet sind. Die Werte dieser Hash-Tabelle sind wiederum Hash-Tabellen, indiziert mit der Nummer der Transaktionen, die die Sperren halten. Die Werte dieser Hash-Tabellen sind Listen von Lock-Objekten. Ein Lock-Objekt ist dabei ein Repräsentant für eine Sperre. Es enthält Informationen über den Typ und den Modus der Sperre, sowie die ID des Knotens, dem sie zugeordnet ist. Diese Listen müssen bei der Suche nach einer bestimmten Sperre linear durchlaufen werden, was aber zu keinem einschlagenden Leistungsdefizit führt, da die Anzahl der Sperren einer Transaktion auf einem Knoten im Allgemeinen gering ist.

Diese Speicherungsstruktur ist dem abstrahierten taDOM-Baum (siehe Kapitel 4) insofern überlegen, dass die einem Knoten zugeordneten Sperren direkt über die ID dieses Knotens erreicht werden können, und nicht der gesamte übergeordnete Baum durchlaufen werden muss. Außerdem ist die Tabellenstruktur weniger speicherintensiv, da nur für diejenigen Knoten Sperrtabellen angelegt werden müssen, für die auch wirklich Sperren existieren.

Abbildung 1 Aufbau der Sperrtabellen



Die Verwendung der Sperrtabellen gegenüber einem Sperrbaum, hat aber noch weitere Konsequenzen, insbesondere für die logischen Sperren. Diese waren beim Entwurf (siehe Abschnitt 4.2.3) auch schon in Tabellen untergebracht, sodass wir diese drei logischen Sperrtabellen mit in die Implementierung einbeziehen können. Jede logische Sperre ist einem Knoten zugeordnet. Somit können wir die logischen Sperren als eigenen Sperrtypen für die jeweiligen Knoten und Transakionen in die Sperrtabelle einfügen.

Des Weiteren sieht der taDOM-Baum einen AttributeRoot-Knoten pro Element vor, auf dem ebenfalls Knotensperren angelegt werden können. Deswegen werden wir zu den Knoten-, Kanten- und logischen Sperren noch zusätzlich die *AttributeRoot-Sperren* als neuen Sperrtypen einführen. Diese können dann, wie alle anderen Sperrtypen, einem Elementknoten zugeordnet werden und in die Sperrtabelle eingefügt werden.

Ähnlich sieht es für die vom taDOM-Baum vorgesehenen String-Knoten aus. Dazu führen wir erneut einen neuen Sperrtypen ein: *String-Sperren*. Durch Zuordnung zu ihrem Textknoten, können auch sie problemlos in die Sperrtabellen eingefügt werden.

Dies führt zu sieben verschiedenen Sperrtypen:

- Knoten-Sperren
- Kanten-Sperren
- Logische Tagname-Sperren
- Logische Attribut-Sperren
- Logische ID-Sperren
- AttributeRoot-Sperren
- String-Sperren

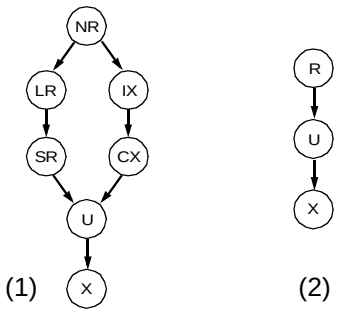
Darüber hinaus kennt der LockManager auch die Sperrkompatibilitätsmatrizen (siehe Abschnitt 4.2), die notwendig sind, um die Verträglichkeit von zwei Sperren festzustellen. Beim Anlegen einer Sperre muss der LockManager erst sicherstellen, dass diese neue Sperre keine Konflikte mit bereits auf dem entsprechenden Knoten vorhandenen Sperren hervorruft. Die meisten atomaren Operationen (siehe Tabelle 1) erfordern mehrere Sperren, von denen jedoch entweder alle

oder überhaupt keine angelegt werden darf. Aus diesem Grund erlaubt es der LockManager, erst zu überprüfen, ob Konflikte für die angeforderten Sperren entstehen. Ist dies nicht der Fall, werden die Sperren erst in einem zweiten Schritt in die Sperrtabelle eingetragen.

Beim Anlegen neuer Sperren muss auch darauf geachtet werden, das nicht eventuell bereits vorhandene Sperren durch die neue Sperre ersetzt werden. Die genauen Ersetzungsschemata finden sich in Abbildung 20. Dabei sind alle möglichen Sperrtypen als Knoten eines gerichteten Graphen dargestellt. Die Kanten stellen die zulässigen Übergänge dar. So darf eine Sperre S_1 nur durch eine Sperre S_2 ersetzt werden, falls ein Weg von S_1 zu S_2 existiert. Dieser Weg muss kein direkter Übergang sein, sondern kann auch Zwischenknoten enthalten. Die Anwendung der Sperrersetzung wird in Beispiel 2 verdeutlicht.

Es muss darauf geachtet werden, dass eine Sperre immer nur eine andere Sperre vom selben Typ ersetzen kann. Es ist also nicht möglich eine Knotensperre durch eine Kantensperre oder eine logische Sperre zu ersetzen.

Abbildung 20 Sperrersetzungsgraphen für Knoten-, AttributeRoot- und Stringsperren (1) sowie logische Sperren und Kantensperren (2)



Beispiel 2 Beispiele zur Sperrersetzung durch den LockManager

- Hat eine Transaktion eine R-Sperre zum Lesen eines Attributwerts angefordert und möchte diesen Wert daraufhin verändern, so kann die bereits existierende R-Sperre durch die nun erforderliche X-Sperre ersetzt werden.

- Eine Transaktion hat beim Navigieren über einen Knoten K_1 eine NR-Sperre auf K_1 angefordert und möchte nun einen Kindknoten löschen. Dafür ist eine CX-Sperre auf K_1 notwendig. Die bereits existierende NR-Sperre kann also durch die CX-Sperre ersetzt werden, da es einen Weg im Sperrersetzungsgraphen von NR nach CX gibt. Hat die Transaktion allerdings vor dem Löschen des Nachfolgerknotens die *getChildNodes()*-Methode aufgerufen und besitzt somit eine LR-Sperre auf K_1, so kann diese nicht durch die CX-Sperre ersetzt werden; es existiert kein Weg von LR nach CX. Dieses Phänomen lässt sich durch das Betrachten der Sperrkompatibilitätsmatrix für Knotensperren (Abbildung 7 auf Seite 29) leicht erklären. Die CX-Sperre ist mit CX-Sperren anderer Transaktionen verträglich, dies ist aber für LR-Sperren nicht der Fall. Durch

das Ersetzen der LR-Sperre durch die CX-Sperre würde diese Einschränkung verloren gehen. Dies ist auf jeden Fall zu vermeiden.

5.2.4 XMLAccess

Die XMLAccess-Schnittstelle stellt Methoden zum direkten Zugriff auf den DOM-Baum zur Verfügung. Da wir in der aktuellen Implementierung auf das DOM-API von Sun zurückgreifen, um die XML-Dokumente zu parsen, aber darüber hinaus nicht von den durch dieses API zur Verfügung gestellten Klassen mit deren Methoden abhängig sein wollen (um diese Schnittstelle eventuell durch eine native Speicherungskomponente zu ersetzen), kapseln wir diese in eigene Klassen und arbeiten außerhalb der XMLAccess-Komponente nur noch mit diesen neuen Objekten.

Da der Zugriff auf den DOM-Baum nicht mehr nur lokal erfolgt, sondern auch entfernte Clients lokale Kopien von Knoten besitzen können, ist es nicht mehr ausreichend, diese über Referenzen eindeutig bestimmen zu können, sondern es müssen zusätzlich eindeutige Identifikationsnummern an die Knoten vergeben werden.

Hierfür ist der NodeManager zuständig. Direkt nach dem Parsen eines XML-Dokuments ermittelt er für jeden Element-, Attribut- und Stringknoten eine systemweit eindeutige Identifikationsnummer und legt diese zusammen mit einer Referenz auf das Node-Objekt in zwei Hash-Tabellen ab. Dies dient dazu, mit gegebener ID schnell den dazugehörigen Knoten zu finden und für einen gegebenen Knoten möglichst schnell die dazugehörige ID zu bestimmen.

Die XMLAccess-Schnittstelle kaspelt nach außen hin komplett den DOM-Baum von Sun und erlaubt Knoten-Zugriffe ausschließlich über die jeweiligen Knoten-IDs. Die kompletten Methodensignaturen in XMLAccess sind in Anhang A aufgeführt.

5.2.5 LockEscalationManager

Der LockEscalationManager ist für die in Abschnitt 4.2.4 beschriebene Sperreskalation zuständig. Dabei untersucht ein eigener Thread in regelmäßigen Abständen die Sperrtabelle und versucht, mehrere Sperren zu einer übergeordneten Sperre zusammenzufassen.

Besondere Betrachtung kommt beim Ersetzen den logischen Sperren zu. Diese dienen dazu, das Einfügen von Knoten, die das entsprechende Prädikat erfüllen, zu unterbinden (siehe Abschnitt 4.2.3). Da das Einfügen von Knoten in einen Unterbaum CX- und IX-Sperren auf allen Vorgängerknoten bis hin zur Dokumentenwurzel erfordert, diese aber mit einer SR-Sperre unverträglich ist, verhindert eine SR-Sperre auf einem Knoten jegliches Einfügen von Knoten in dessen Unterbaum. Damit werden auch alle logischen Sperren in diesem Unterbaum überflüssig und können entfernt werden.

Wir werden im Folgenden zwei Varianten der Sperr-Eskalation betrachten.

Nach der ersten Variante bestimmt der LockEscalationManager erst das Verhältnis von gehaltenen Sperren und Gesamtzahl der Knoten im Unterbaum für jeden einzelnen Knoten im taDOM-Baum. Er steigt dann rekursiv in den Baum hinab, und beginnt die Berechnung bei den Blättern,

weil hier der Unterbaum am kleinsten (nämlich 0) ist. Für die darüberliegenden Knoten können die Ergebnisse der Unterbaumknoten weiterverwendet werden.

Im zweiten Schritt wird für jeden Knoten, der eine Baumtiefe größer als der vom Benutzer spezifizierte *Escalation Depth* Wert das im ersten Schritt berechnete Verhältnis mit dem wiederum vom Benutzer eingestellten *Escalation Threshold* Wert verglichen. Liegt das Verhältnis über diesem Grenzwert, werden im Unterbaum des entsprechenden Knotens alle Sperren, mit Ausnahme der logischen Sperren, entfernt, und durch eine passende Sperre auf der Unterbaumwurzel (wie in Abschnitt 4.2.4 beschrieben) ersetzt.

Dabei ist zu beachten, dass auch der zweite Schritt von den Blättern zur Wurzel des Baums durchgeführt wird. Dadurch kann es passieren dass auf einem kleinen, tieferliegenden Unterbaum die Sperren zusammengefasst werden, und somit der *Escalation Threshold* Grenzwert für den darüberliegenden Knoten nicht mehr erreicht wird, die Sperrsituation hier also nicht beeinflusst wird, obwohl ein Vorgehen von der Wurzel zu den Blättern ein Zusammenfassen der Sperren eine oder mehrere Stufen höher im Baum hervorgerufen hätte.

Nach der zweiten Variante wird das Verhältnis von gehaltenen Sperren und Gesamtzahl der Knoten im Unterbaum nur für diejenigen Knoten ausgerechnet, deren Tiefe im Baum genau dem *Escalation Depth* Parameter entsprechen. Das Ersetzen der Sperren findet ebenfalls nur auf genau dieser Baumhöhe statt.

Die zweite Variante hat den Vorteil, dass weniger Schritte ausgeführt werden müssen, um die Sperr-Eskalation einmal auf der Sperrtabelle durchzuführen, und dass im Allgemeinen mehr Sperren ersetzt werden können. Dies liegt daran, dass jeweils nur ein gesamter Unterbaum eines Knotens auf der durch *Escalation Depth* spezifizierten Baumhöhe für die Ersetzung betrachtet wird und keine Ersetzungen in Teilunterbäumen durch vorheriges rekursives Absteigen durchgeführt wurden. Diese können bewirken, dass auf der *Escalation Depth* der *Escalation Threshold* nicht mehr erreicht wird, und somit auf dieser Höhe keine Ersetzungen mehr stattfinden.

Durch das Anwenden der zweiten Variante entsteht, durch die geringere Anzahl an Sperren, eine gröbere Sperrgranularität. Dies schränkt wiederum die Parallelität von Transaktionen ein. Es entsteht also ein Trade-Off zwischen Anzahl der Sperren und Nebenläufigkeitsgrad von Transaktionen. Die zu verwendende Variante wird in der Konfigurationsdatei des Servers eingestellt (siehe Abschnitt 5.2.7).

5.2.6 DeadLockDetector

Der DeadLockDetector wird bei jedem Auftreten eines Konflikts zwischen zwei oder mehreren Transaktionen gestartet und überprüft das System auf Verklemmungen. Dadurch wird sichergestellt, dass eventuell auftretende Deadlocks sofort erkannt und aufgelöst werden können.

Dazu baut er zunächst aus den Wartebeziehungen zwischen Transaktionen einen so genannten Wait-For-Graphen auf. Es handelt sich dabei um einen gerichteten Graphen, der als Knoten die zum jeweiligen Zeitpunkt aktiven Transaktionen enthält. Ist eine Transaktion T1 durch eine Operation der Transaktion T2 blockiert, so gibt es im Wait-For-Graphen eine gerichtete Kante von T1 zu T2. Das System ist genau dann verklemmungsfrei, wenn in diesem Graphen keine Zyklen vorkommen.

Um nun mögliche Zyklen aufzusuchen, beginnt der DeadLockDetector bei einem beliebigen Knoten (also einer aktiven Transaktion) und durchläuft rekursiv jeden möglichen Weg durch den Wait-For-Graphen. Dabei merkt er sich den bereits zurückgelegten Weg vom Anfangsknoten in einer Liste. Stößt er beim Durchlaufen eines solchen Weges auf einen Knoten, der schon in der Liste der bereits passierten Knoten vorhanden ist, handelt es sich um einen Zyklus und somit um eine Verklemmung. Alle Transaktionen, die diesen Zyklus bilden, befinden sich also in einer zyklischen Wartebeziehung.

Zur Auflösung dieser Verklemmung ist es nun notwendig, genau eine Transaktion aus diesem Zyklus zurückzusetzen, um somit den Kreis der Wartebeziehungen aufzubrechen. Dadurch können die verbleibenden Transaktionen ungehindert die noch fehlenden Sperren anfordern und ihre Arbeit fortsetzen.

Die zurückzusetzende Transaktion wird dabei nicht nach dem Zufallsprinzip ausgesucht, sondern, um möglichst wenig Arbeit zu verlieren, diejenige, die in ihrem bisherigen Verlauf am wenigsten Sperren angefordert hat.

5.2.7 Server-Konfiguration

Die Parametrisierung des XTC-Servers erfolgt über eine XML-Konfigurationsdatei, die beim Start des Servers eingelesen wird. Eine Beispiel-Konfigurationsdatei befindet sich in Anhang B. In der Konfigurationsdatei lassen sich folgende Parameter für den XTC-Server festlegen:

- **DocumentPath**: Der XML-Dokumentenpfad ist der Pfad, in dem die XML-Dokumente liegen, die den Clients verfügbar gemacht werden sollen. Ist dieser Pfad falsch angegeben oder befinden sich in ihm keine gültigen XML-Dokumente, so ist der Datenbestand des Servers leer.

- **DisableLockMechanism**: Diese Konfigurationsanweisung dient dazu, den Sperrmechanismus ein- oder auszuschalten. Ist dieser Wert auf *true* gesetzt, findet keine Synchronisierung der Zugriffe durch den Server statt. Diese Einstellung dient dazu, das Leistungsverhalten des Servers zu messen. Die Standardeinstellung ist *false*.

- **UseAttributeRoot**: Diese Anweisung legt fest, ob der zusätzliche Knotentyp *AttributeRoot* im taDOM-Baum verwendet werden soll. Setzt man diesen Wert auf *false*, so wird bei einem *getAttributes()*-Aufruf keine LR-Sperre mehr auf AttributeRoot angelegt, sondern jedes Attribut einzeln mit einer NR-Sperre versehen. Mit Hilfe von Leistungsmessungen werden wir versuchen, die Existenz des AttributeRoot-Knotens zu rechtfertigen. Der Standardwert ist *true*.

- **QueueManagerType**: Diese Anweisung legt den zu verwendenden QueueManager-Typen fest. Zur Auswahl stehen *Lazy* und *Busy* (siehe Abschnitt 5.2.1). Der Standardwert ist *Lazy.*

- **RMI-Host**: Diese Anweisung legt den Hostnamen des Rechners fest, auf dem die Stubs der Remote-Objekte abgelegt werden. Standardeinstellung ist *localhost*.

- **RMI-Handle**: Die Anweisung legt den Bezeichner fest, der für die Identifizierung des XTC-DOMRMI Objekts (siehe Abschnitt 5.1.2.1) beim Naming-Service (im Allgemeinen rmiregistry) verwendet wird.

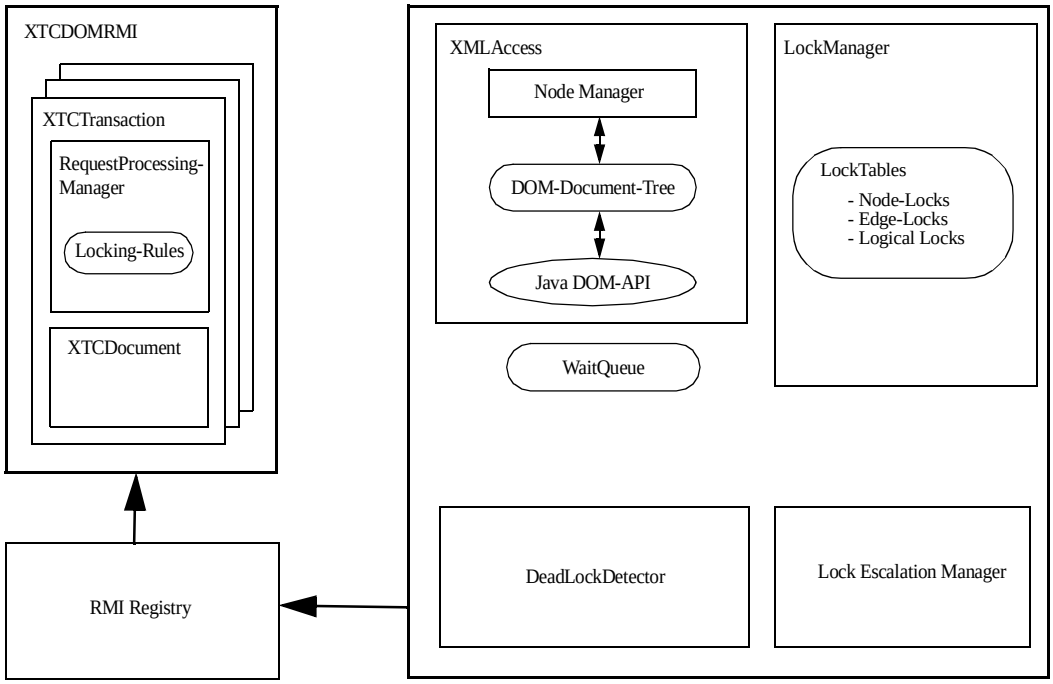
- **RMI-Timeout**: Die Anweisung legt die Anzahl der Sekunden fest, nach der eine nicht mehr erreichbare Referenz auf ein Remote-Objekt als verloren betrachtet wird. Der Verbindungsverlust in RMI ist in Abschnitt 5.1.3 im Detail beschrieben.

5.3 Veränderte Systemarchitektur

Nach der Implementierung der in Abschnitt 5.2 vorgestellten Serverarchitektur haben wir festgestellt, dass ein großer Teil der potentiellen Leistungsfähigkeit des Systems dadurch verloren geht, dass alle Requests von den Clients in einer Warteschlange aufgenommen werden und seriell abgearbeitet werden. Durch die grobgranulare Synchronisierung der Zugriffe und die Verarbeitung der Daten im logischen Einbenutzerbetrieb geht jegliche Nebenläufigkeit seitens des Servers verloren.

Aus diesem Grund sind wir in einem nächsten Schritt zu einer etwas veränderten Systemarchitektur übergegangen. Dabei werden wir auf die Serialisierung der Requests durch eine Warteschlange verzichten und stattdessen für jeden Client einen eigenen Thread auf der Serverseite anlegen der direkt auf den XML-Datenbestand zugreifen kann. Abbildung 21 zeigt die veränderte Systemarchitektur im Überblick.

Abbildung 21 Überblick über die veränderte Systemarchitektur



Die Veränderungen an der Architektur wirken sich vor allem auf folgende Komponenten aus: DatabaseEngineThread, XTCTransaction, RequestProcessingManager, LockManager, Dead-

LockDetector und XMLAccess. In den folgenden Abschnitten werden wir auf diese Komponenten im Einzelnen eingehen.

5.3.1 DatabaseEngineThread

Da sich nach der neuen Architektur der Server nicht mehr im Einbenutzerbetrieb befindet, sondern für jeden verbundenen Client ein eigener Thread gestartet wird, ist der DatabaseEngine-Thread nicht mehr notwendig und taucht in der aktuellen Systemarchitektur nicht mehr auf.

5.3.2 XTCTransaction

Wie bisher läuft auch jetzt die XTCTransaction in ihrem eigenen Thread auf dem Server und wird vom Client über RMI angesteuert. Dieser Thread fügt die Aufträge des Clients aber nicht mehr in eine Warteschlange ein und wartet bis eine Antwort vom Server in der ResponseQueue erscheint, sondern führt die Requests unmittelbar aus. Dazu gibt die XTCTransaction diese direkt an den RequestProcessingManager weiter. Jede XTCTransaction hat ihre eigene Instanz des RequestProcessingManagers. Diese Instanz führt die Operation gleichzeitig mit den Instanzen anderer Transaktionen aus, was zu nebenläufigen Zugriffen auf den LockManager und XMLAccess führt.

5.3.3 RequestProcessingManager

Der RequestProcessingManager ist nach wie vor für das Anfordern der Sperren sowie für das Abrufen der Daten zuständig. Der größte Unterschied zur bisherigen Implementierung liegt in der Behandlung von auftretenden Konflikten. Da wir nach der neuen Architektur sowohl auf die Auftragswarteschlange als auch auf einen speziellen Server-Thread zum Verarbeiten der Requests verzichtet haben, muss die Synchronisierung anderweitig erfolgen. Stellt der RequestProcessingManager also fest, dass der aktuelle Auftrag nicht verarbeitet werden kann, weil ein Konflikt beim Zugriff auf die Daten vorliegt, so kann er, durch Aufrufen der *wait()-Methode* auf dem XTCTransaction Objekt, das den Auftrag abgesetzt hat, den aktuellen Thread vorübergehend deaktivieren. Dadurch wird der Methodenaufruf des Clients solange blockiert, bis die Transaktion die die Blockierung verursacht hat, entweder Commit oder Rollback ausgeführt hat, oder vom DeadLockDetector zurückgesetzt worden ist. Die Warteabhängigkeiten werden nach wie vor durch die WaitQueue verwaltet. Diese sucht auch beim Commit oder Rollback einer Transaktion T_1 diejenigen anderen Transaktionen die durch T_1 blockiert wurden und ruft dann *notifyAll()* auf den entsprechenden XTCTransaction Objekten auf, um diese zu reaktivieren.

Ein weiterer wichtiger Unterschied zur vorherigen Implementierung liegt in der Art und Weise wie die Sperren anfordert werden. Da der logische Einbenutzerbetrieb innerhalb des Servers in der neuen Implementierung nicht mehr gegeben ist, können für eine Transaktion T_1 nicht erst alle Sperren überprüft und dann erst angelegt werden, wenn kein Konflikt aufgetreten ist. In der Zwischenzeit, also zwischen dem Überprüfen und dem Anlegen könnte eine andere Transaktion T_2 Sperren angelegt haben, die mit denjenigen von T_1 kollidieren. Dieses Problem wird nun dadurch verhindert, dass unmittelbar nach der Überprüfung einer Sperre diese auch angelegt wird. Die Atomizität dieser beiden Schritte wird durch einen *synchronized* Block sichergestellt. Tritt nun ein Konflikt bei diesem Vorgang auf, so müssen alle Sperren der entsprechenden Operation, die bis zu dem Zeitpunkt schon angelegt worden waren, wieder entfernt werden.

Mit der neuen Version des RequestProcessingManagers haben wir noch eine weitere Optimierung bei der Sperranforderung eingeführt. Bei fast allen Operationen auf einem Knoten K im taDOM-Baum muss sichergestellt werden, dass die entsprechenden Sperren auf allen Knoten bis zur Wurzel des Dokuments angelegt werden. Dies erfordert umso mehr Schritte, je tiefer sich der Knoten im taDOM-Baum befindet. Da aber in der großen Mehrheit der Fälle von einem benachbarten Knoten auf den Knoten K navigiert wurde, wurden die Sperren auf den Vorgängerknoten schon von einer vorhergehenden Operation angelegt und müssen nicht noch einmal überprüft werden. Somit bricht der Algorithmus beim Überprüfen der notwendigen Knotensperren vom Knoten K bis hin zur Wurzel (also von unten nach oben) genau dann ab, wenn die angeforderte Sperre auf dem Knoten schon vorhanden ist. Dies impliziert, dass auch auf den Vorgängerknoten die notwendigen Sperren schon durch eine vorherige Operation angelegt worden sind.

Das Überprüfen und Anlegen von Sperren von den Blättern zur Wurzel hat aber auch Nachteile. Da für jede Operation eine entsprechende Sperre auf der Wurzel des Dokuments angelegt werden muss, ist die Wahrscheinlichkeit am größten, dass auch an dieser Stelle ein Konflikt auftritt. Überprüft man die Sperren aber von unten nach oben, so werden unter Umständen sehr viele Sperren angelegt bevor es zum Konflikt an der Wurzel kommt. Dann müssen alle bereits angelegten Sperren wieder entfernt werden, was einen erheblichen Aufwand darstellt. Aus diesem Grund haben wir den Algorithmus so verändert, dass Sperren auch von oben (also von der Wurzel) nach unten vergeben werden können. Dadurch werden zwar Konflikte früher erkannt, treten aber keine Konflikte auf, so müssen alle Sperren von der Wurzel bis zum entsprechenden Knoten einzeln überprüft werden. Eine Optimierung wie im Fall der Sperranforderung von unten nach oben ist nun nicht mehr möglich. Die Reihenfolge der Sperranforderungen wird beim Serverstart durch die Konfigurationsdatei festgelegt. Die *lockingorder* Anweisung hat zwei mögliche Werte: *topdown* (von oben nach unten) oder *bottomup* (von unten nach oben).

5.3.4 LockManager

Die einzige Änderung der der LockManager unterzogen wurde, ist eine Synchronisierungsmaßnahme. Um zu verhindern, dass mehrere Transaktionen gleichzeitig direkt oder indirekt auf die Sperrtabellen zugreifen, wurden alle Methoden als *synchronized* deklariert. Dadurch wird bei jedem Methodenaufruf eine Sperre auf dem LockManager Objekt angelegt, die bei Beendigung der Methode wieder freigegeben wird. Eine andere Transaktion kann eine *synchronized* Methode nur dann aufrufen, wenn zu dem Zeitpunkt keine Sperre auf dem LockManager Objekt vorhanden ist. Andernfalls wird diese Transaktion blockiert, bis die Sperre wieder freigegeben ist.

5.3.5 DeadLockDetector

Für den DeadLockDetector gilt im Groben das gleiche wie für den RequestProcessingManager. Konnte ein Deadlock erkannt werden, so wird eine Transaktion ausgewählt die zurückgesetzt werden muss. Da der entsprechende Thread durch das Auftreten eines Konflikts zuvor vom RequestProcessingManager vorübergehend deaktiviert wurde, muss der DeadLockDetector die zurückgesetzte Transaktion erst als solche kennzeichnen und durch einen *notifyAll()* Aufruf wieder aktivieren. Die Transaktion erkennt nun sofort, dass sie aufgrund eines Deadlocks zurückgesetzt worden ist, bricht die Ausführung der laufenden Operation ab und liefert dem Client eine entsprechende Fehlermeldung zurück.

Durch das Zurücksetzen einer am Deadlock beteiligten Transaktion können im Allgemeinen die anderen Transaktionen ihre Arbeit wieder aufnehmen. Dazu müssen sie aber erst reaktiviert werden. Wie im Falle eines normalen Konflikts, werden die Abhängigkeiten durch die WaitQueue aufgesucht und die nun nicht mehr blockierten Transaktionen durch einen *notifyAll()*-Aufruf reaktiviert.

5.3.6 XMLAccess

Da nun auch beliebig viele Transaktionen (Threads) gleichzeitig auf XMLAccess zugreifen können, müssen auch hier Vorkehrungen getroffen werden, damit keine Isolationsvorschriften verletzt werden. Da aber durch den RequestProcessingManager und den LockManager gleichzeitige, in Konflikt stehende Operationen verhindert werden, müssen hier keine weiteren Änderungen vorgenommen werden. Die einzige Operation, die nicht durch Sperren synchronisiert wird, ist das Öffnen von XML-Dokumenten. Es muss auf jeden Fall verhindert werden, dass mehrere Transaktionen gleichzeitig versuchen, mit der gleichen Instanz des DOM-Parsers ein XML-Dokument zu öffnen, da dies unmittelbar zu einer Flut von Exceptions führt. Aus diesem Grund wurde die *getDocument()*-Methode von XMLAccess ebenfalls als *synchronized* deklariert.

5.3.7 Weitere Optimierungsansätze

Der oben beschriebene Ansatz hat zwar die Nebenläufigkeit im System gegenüber der ersten Version sehr stark erhöht, schränkt aber die Parallelität mehrerer Transaktionen durch die große Anzahl an *synchronized* Blöcken immer noch erheblich ein. Vor allem die komplett synchronisierten Methoden des LockManagers lassen nur einen gleichzeitigen Zugriff auf die Sperrtabellen zu, obwohl mehrere Zugriffe möglich wären ohne Inkonsistenzen hervorzurufen.

5.3.7.1 Verkleinerung der kritischen Abschnitte

In einer ersten Phase versuchen wir, die so genannten kritischen Abschnitte zu kurz wie möglich zu gestalten, um eine maximale Nebenläufigkeit zu erreichen. Kritische Abschnitte sind Programmteile, die durch einen umgebenden *synchronized* Block gekennzeichnet sind und maximal von einem Thread gleichzeitig ausgeführt werden dürfen.

Da es für Operationen auf der Sperrtabelle nicht unbedingt notwendig ist, diese komplett für gleichzeitige Zugriffe zu-sperren, werden wir in diesem Ansatz die Synchronisationssperren auf den Sperrtabellen für die einzelnen Knoten auf der zweiten Ebene (siehe Abbildung 1) anlegen. Dies ermöglicht das gleichzeitige Einfügen und Entfernen von Sperren für verschiedene Knoten des taDOM-Baums. Es treten allerdings dann Probleme auf, wenn auf dem Knoten, auf dem die Sperre angelegt werden soll, vorher noch keine Sperre vorhanden war. In diesem Fall muss die Sperrtabelle für den Knoten erst angelegt werden, bevor sie zum Objekt der Synchronisierung werden kann.

Nun kann es aber vorkommen, dass zwei Transaktionen T_1 und T_2 gleichzeitig die erste Sperre für einen Knoten anlegen wollen und auch gleichzeitig feststellen, dass erst die entsprechende Sperrtabelle eingefügt werden muss. T_1 fügt daraufhin die neue Sperrtabelle für diesen Knoten mitsamt der angeforderten Sperren in die Hauptsperrtabelle ein. Da die atomaren Zugriffe auf Hash-Tabellen synchronisiert sind, muss T_2 warten bis die Einfügeoperation von T_1 abgeschlos-

sen ist. Nun fügt auch T_2 eine neue Sperrtabelle für den Knoten ein und entfernt die soeben von T_1 eingefügten Sperren.

Da das Einfügen in eine Hash-Tabelle als atomare Operation den zuvor an der Stelle (für den gleichen Schlüssel) vorhandenen Wert zurückliefert, könnte man sich überlegen, die durch T_2 überschriebene Sperrtabelle (also die von T_1) durch T_2 wieder einzufügen und die Sperren von T_2 in die von T_1 angelegte Tabelle zu verlegen. Dies funktioniert solange, wie keine dritte Transaktion ebenfalls eine Sperrtabelle erstellen möchte. Diese hätte zwischen dem Einfügen und dem Rückersetzen von T_2 eine eigene Tabelle einbringen können. In diesem Fall geraten die zuvor einfachen Zusammenhänge durcheinander und es ist nicht mehr festzustellen, wer in einer Kette von n Transaktionen zuletzt eine Tabelle eingefügt hat, und wie die Sperren eingefügt werden müssen, ohne einen Verlust von Sperren mit sich zu führen.

Wir haben uns angesichts dieses Problems überlegt, ob man die Struktur der Sperrtabellen ändern sollte. Man könnte wohl durch andere Datenstrukturen die Nebenläufigkeit der Transaktionen erhöhen, müsste dann aber mit erheblichen Leistungseinbußen wegen der langsameren Zugriffe auf die Sperrtabellen rechnen.

Eine weitere Möglichkeit wäre, von den Sperrtabellen selbst als Synchronisierungsobjekte abzusehen und dafür die Knoten im taDOM-Baum heranzuziehen. Dann müsste allerdings bei jedem Sperrvorgang auf einem Knoten dieser zuerst über die XMLAccess-Schnittstelle geholt werden, obwohl dies nicht unbedingt notwendig wäre. Beim Aufruf von *setValue()* müsste dann vorher erst ein *getValue()* ausgeführt werden, um die entsprechenden Sperren für *setValue()* anlegen zu können. Dies würde aber soviel zusätzliche Zeit in Anspruch nehmen, dass sich der Aufwand überhaupt nicht lohnt.

Andererseits könnte man aber auch künstliche Synchronisierungsobjekte für jeden Knoten erstellen und in einer Liste, oder besser, einer Hash-Tabelle ablegen. Tests haben aber ergeben, dass auch eine solche Vorgehensweise zu keiner Leistungsverbesserung führt.

Daher werden wir doch die Einfügeoperationen von Knotensperrtabellen synchronisieren müssen.

5.3.7.2 Umordnung der Operationen

Bisher bestand eine Sperranforderung aus dem Überprüfen auf Konflikte und dem anschließenden Anlegen der Sperre, falls keine Konflikte aufgetreten sind. Dieser Vorgang darf durch keine andere Transaktion unterbrochen werden, da ansonsten eventuell Sperren angelegt werden, die nicht verträglich sind. Das würde wiederum zu Verletzungen des Transaktionsparadigmas führen und ist absolut inakzeptabel.

Um aber auf einen kritischen Abschnitt auf dieser Ebene zu verzichten und die Parallelität nicht unnötig einzuschränken, haben wir die Reihenfolge der Operationen vertauscht. Es wird nun zuerst die Sperre angelegt und dann erst überprüft, ob ein Konflikt einstanden ist. Ist dies der Fall, muss die Sperre natürlich nachträglich wieder entfernt werden. Da wir davon ausgehen, dass in der großen Mehrheit der Zugriffe keine Konflikte entstehen, steigt durch diesen Eingriff der Gesamtaufwand nur unerheblich. Außerdem ist das Überprüfen auf Konflikte nur dann notwendig, wenn die Sperre auch wirklich angelegt werden musste. Wurde aber beispielsweise eine NR-Sperre auf einem Knoten angefordert, auf dem für die entsprechende Transaktion bereits

eine SR-Sperre vorhanden war, so braucht die NR-Sperre nicht angelegt zu werden und auch das Überprüfen ob die NR-Sperre einen Konflikt hervorgerufen hat, entfällt.

Durch diese Maßnahme ist es möglich geworden, dass mehrere Transaktionen absolut gleichzeitig Lesesperren (oder im Allgemeinen, nicht in Konflikt stehende Sperren) anlegen können, ohne dass die gesamten Sperranforderungsvorgänge, also das Überprüfen und das Anlegen, voneinander getrennt ablaufen müssen.

Da nach diesem Verfahren mehrere Transaktionen ihre Sperren erst anlegen können, bevor diese auf Konflikte überprüft werden, kann es vorkommen dass Konflikte erkannt werden, die eigentlich nicht existieren. Angenommen zwei Transaktionen möchten derart auf einen Knoten zugreifen, dass ein Konflikt entsteht. Beide fügen gleichzeitig ihre jeweiligen Sperren in die Sperrtabelle ein und überprüfen dann erst, ob ein Konflikt entstanden ist. Beide Transaktionen werden deswegen einen Konflikt entdecken und ihre Sperren wieder entfernen, obwohl eigentlich nur eine der beiden Transaktionen blockiert werden sollte.

Nach dem bisherigen Verfahren wäre jetzt ein unnötiger Deadlock entstanden. Aus diesem Grund haben wir das Vorgehen insofern erweitert, dass nach dem Auftreten eines solchen Konflikts die blockierte Transaktion noch zwei weitere Versuche starten wird, bevor sie endgültig blockiert wird. Diese Versuche werden beide beteiligten Transaktionen wieder gleichzeitig durchführen wollen, was wiederum zu einem ähnlichen Konflikt führen könnte. Deswegen wird zwischen zwei Versuchen, eine Sperre anzulegen, eine zufällige Wartezeit zwischen 5 und 15 Millisekunden eingefügt.

5.4 XTC-Client

In Abschnitt 5.2 haben wir den Aufbau und die Funktionsweise des XTC-Servers betrachtet. Um die Funktionalitäten des Servers nutzen und testen zu können, benötigen wir einen Client, der Anfragen über die XTC-Schnittstellen an den Server absetzt. Dazu gibt es zwei verschiedene Prototyp-Implementierungen. Zum einen den Kommandozeilen-Client, mit dem vorher hartkodierte Zugriffsschemata ausgeführt werden können, zum anderen einen grafischen Client, der es erlaubt, mit Hilfe eines Baumes, Schritt für Schritt durch das XML-Dokument zu navigieren.

5.4.1 Kommandozeilen-Client

Der Kommandozeilen-Client dient hauptsächlich zum Testen des XTC-Servers, sowie zur Durchführung von Leistungsuntersuchungen. Der Client wird mit den folgenden Parametern von der Kommandozeile gestartet:

```
XTCClient <Module> <NrOfTransactions> <ModuleParams>
```

<*Module*> ist das auszuführende Modul. Diese Module sind hartkodiert und beschreiben das Zugriffsschema auf das XML-Dokument. Folgende Module wurden für diese Prototypimplementierung realisiert:

- **RecursiveRead**: Das Modul holt sich zunächst das Root-Element des Dokuments und baut dann rekursiv eine lokale Kopie des Baums mit Hilfe von *getChildNodes()* und *getAttributes()*

Aufrufen auf. Somit werden alle Knoten im DOM-Baum durchlaufen und die maximale Anzahl an Lesesperren angelegt.

- **RecursiveReadChangeValue**: Dieses Modul funktioniert ähnlich wie RecursiveRead, nur dass es zusätzlich noch alle Werte von Textknoten verändert. Dadurch müssen weitere Schreibsperren angelegt werden, was dazu führt, dass auf demselben Dokument weitaus mehr Sperranforderungen stattfinden können, als beim herkömmlichen RecursiveRead Modul.

- **RecursiveElementsByTagname**: Dieses Modul durchläuft das Dokument ebenfalls rekursiv, beschränkt sich dabei jedoch auf die Teilbäume, deren Root-Element den angegebenen Tagnamen besitzen. Dadurch lässt sich die betroffene Tag-Menge durch Angabe des Tagnamens vergrößern beziehungsweise verkleinern.

- **RecursiveReadSlow**: Dieses Modul durchläuft das Dokument, genau wie *RecursiveRead* von der Wurzel bis zu den Blättern rekursiv. Es wendet allerdings eine andere, langsamere Strategie an. So werden die Kindknoten eines Knotens nicht mehr mit *getChildNodes()* geholt, sondern mit der Abfolge *getFirstChild(), getNextSibling(), getNextSibling()...*

- **RecursiveReadRandom**: Dieses Modul liest das Dokument erneut rekursiv von der Wurzel zu den Blättern ein, wählt aber auf jeder Ebene eine zufällige Reihenfolge in der die Kindknoten verarbeitet werden. Dadurch wird sichergestellt, dass gleichzeitig gestartete Transaktionen nicht denselben Zugriffspfad wählen. So können parallele Zugriffe mehrerer Transaktionen auf die Sperrtabelle vermieden werden.

<*NrOfTransactions*> gibt die Anzahl der Transaktionen an, die gleichzeitig die Vorschriften aus dem Modul ausführen. So lässt sich das Verhalten des XTC-Servers unter der Belastung von mehreren nebenläufigen Transaktionen beobachten. Die Nebenläufigkeit wird dadurch erreicht, dass jede Instanz des auszuführenden Moduls in einem eigenen Thread gestartet wird.

<*ModuleParams*> sind die vom jeweiligen Modul abhängigen Zusatz-Parameter. Dies sind im Allgemeinen der Name des Dokuments und die maximale Sperrtiefe (siehe Abschnitt 4.2.4). Im Falle des RecursiveElementsByTagname Moduls muss zusätzlich noch der zu betrachtende Tagname mit angegeben werden.

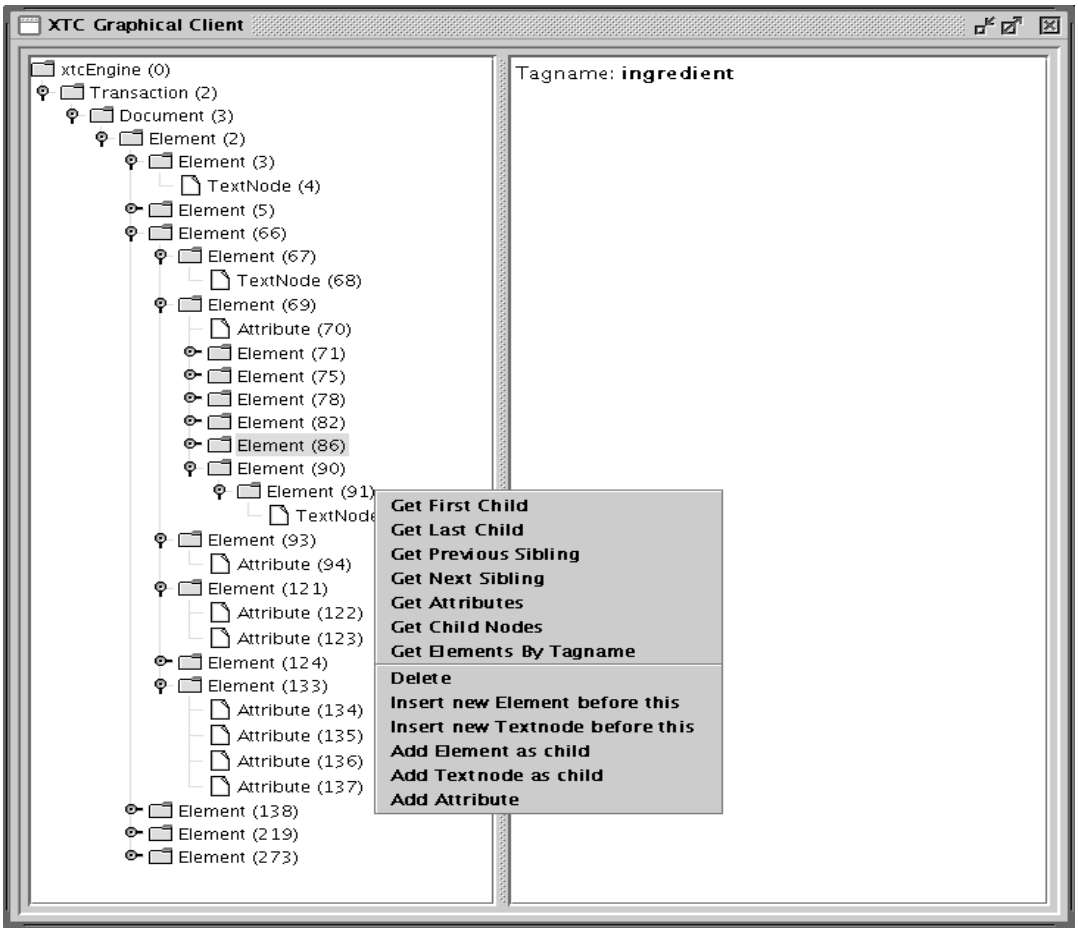
5.4.2 Grafischer Client

Der oben beschriebene Kommandozeilen-Client eignet sich am besten für das Untersuchen des Leistungverhaltens des XTC-Servers. Möchte man jedoch individuelle Szenarien durchführen, so ist es sehr aufwendig, dafür jedes Mal ein neues Modul zu programmieren und dieses über den Kommandozeilen-Client auszuführen.

Aus diesem Grund existiert ein grafischer Client der das Navigieren durch den DOM-Baum zulässt. Abbildung 22 zeigt die Oberfläche des Prototypen.

Das Hauptfenster ist vertikal in zwei Rahmen unterteilt. In der linken Hälfte befindet sich ein Baum, ein so genannter JTree, der das ausgewählte XML-Dokument darstellt. Beim Start des Clients befindet sich lediglich ein einziger Knoten in diesem Baum, nämlich ein Repräsentant des XTC-Servers. Der Client verbindet sich automatisch beim Start mit dem Server. Ist dieser nicht verfügbar, so liefert der Client eine entsprechende Fehlermeldung und terminiert.

Abbildung 22 Screenshot des grafischen Clients in einem Beispiel-Szenario



Mit einem Rechtsklick auf diesen Root-Knoten öffnet sich ein Kontext-Menü, der nur den Menüpunkt zum Erstellen einer neuen Transaktion enthält. Nach diesem Schritt ist die neue Transaktion als Kindknoten des XTC-Servers im JTree sichtbar.

Durch einen erneuten Rechtsklick auf den Transaktionsknoten öffnet sich wieder ein Kontextmenü. Zur Auswahl stehen zwei Menüpunkte. Durch die Aktivierung von *Commit Transaktion* werden alle Änderungen der entsprechenden Transaktion für alle anderen Transaktionen sichtbar und alle durch die Transaktion gehaltenen Sperren wieder freigegeben. Beim Auswählen des Menüpunktes *Open Document* erscheint ein Fenster zur Eingabe des Namens des zu öffnenden XML-Dokuments. Dabei ist der Pfad- beziehungsweise Dateiname relativ zu dem durch den XTC-Server festgelegten Dokumentenpfad (siehe Abschnitt 5.2.7). Wurde der Name eines gültigen XML-Dokuments angegeben und die Auswahl mit einem Klick auf den *OK* Knopf bestätigt, so erscheint ein Repräsentant dieses XML-Dokuments im JTree.

Mit einem weiteren Rechtsklick auf den Dokumentenknoten im JTree öffnet sich erneut ein Kontextmenü. Als Menüpunkte sind *Get Root, Get Element By ID, Set Maximum Lock Depth, Set Escalation Depth* und *Set Escalation Threshold* verfügbar. Mit *Get Root* holt man sich das Root Element des DOM-Baums vom Server. Dieser wird nach der Anfrage als weiterer Knoten im

JTree angezeigt. Ein Klick auf *Get Element By ID* führt die gleichnamige Anfrage durch und zeigt den resultierenden Knoten (falls vorhanden) im JTree an. Mit *Set Maximum Lock Depth, Set Escalation Depth* und *Set Escalation Threshold* lassen sich die entsprechenden Dokumentenparameter setzen (siehe Abschnitt 4.2.4)

Nachdem das Root-Element im JTree sichtbar geworden ist, lässt sich der Rest des Dokuments mit weiteren Kontextmenüaufrufen nach und nach aufbauen. Dazu stehen die bereits in Abschnitt 2.4 vorgestellten Funktionen zur Verfügung.

Beim einfachen Linksklicken auf einen Knoten im JTree wird dieser ausgewählt. Dabei werden alle verfügbaren Informationen über diesen Knoten in der rechten Hälfte des Hauptfensters angezeigt. Für eine Transaktion ist dies deren Remote-ID, also beim Instanziieren der Transaktion von RMI vergebenen internen Identifikationsnummer. Ist im JTree ein Element ausgewählt, so wird in der rechten Hälfte der Tagname angezeigt, bei einem Textknoten ist es dessen Inhalt und bei einem Attribut wird sowohl der Tagname als auch der Wert angezeigt. In den letzen beiden Fällen wird der Wert aber nur dann angezeigt, falls dieser vorher explizit mit dem *getValue()* Aufruf vom Server angefordert wurde. Ansonsten erscheint die Angabe: *Value not retrieved.*

5.5 XTC-LockMonitor

Der XTC-Server verwaltet die Sperrtabellen intern in einer zwar möglichst effizienten jedoch recht schwer zugänglichen Datenstruktur. Da es aber oftmals, vor allem beim Testen des Servers sinnvoll ist, die Anzahl und Natur der Sperren auf einem bestimmten Knoten zu überprüfen, gibt es den XTC-LockMonitor, der die vom Server verwalteten Sperren visualisiert. Abbildung 23 zeigt die Oberfläche des LockMonitors.

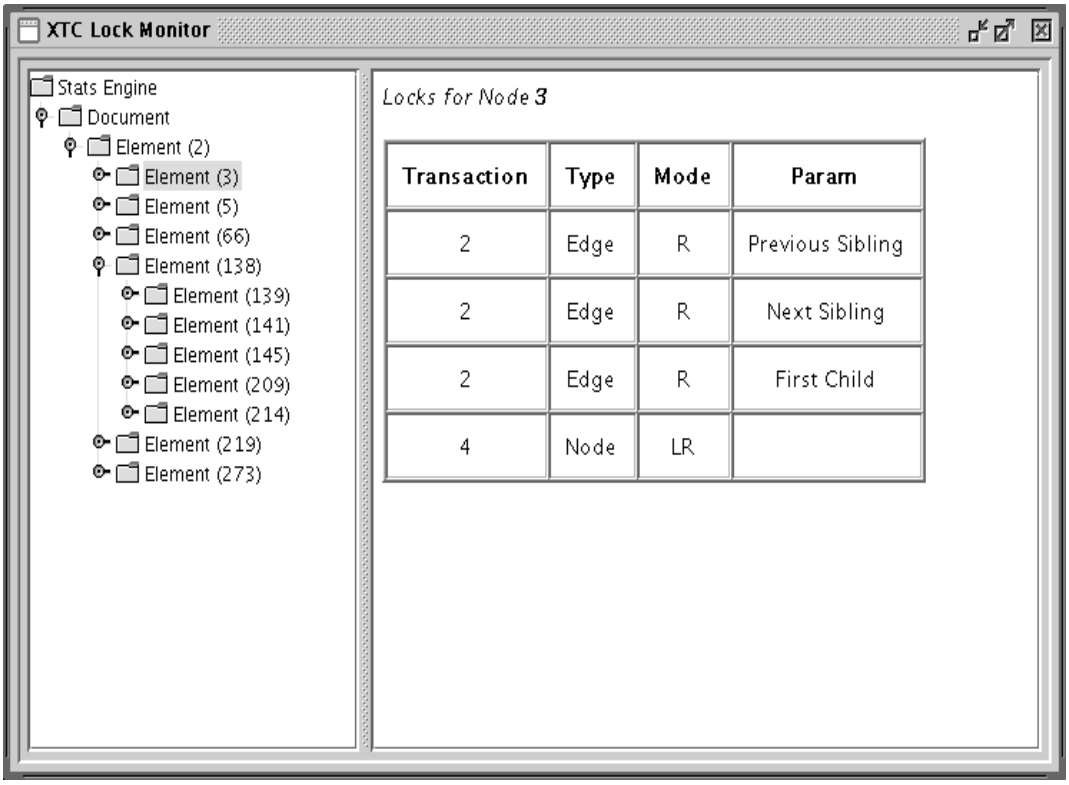
Der Aufbau des GUI (Graphical User Interface) ist dem des grafischen XTC-Clients sehr ähnlich. Das Hauptfenster ist ebenfalls vertikal in zwei Hälften unterteilt. In der linken Hälfte befindet sich ein JTree, der alle vom Server geöffneten XML-Dokumente als Teilbäume darstellt. Anders als beim grafischen Client wir der entsprechende Teilbaum beim Klick auf dessen Knoten aktualisiert. Zur Ansicht der Sperren muss dann nur noch der gewünschte Teilbaum mit Hilfe der dafür vorgesehenen Schaltern im JTree aufgeklappt werden und der Knoten mit einem Linksklick ausgewählt werden.

Die rechte Hälfte des Hauptfensters enthält für jeden Knoten eine Tabelle mit den auf dem entsprechenden Knoten im taDOM-Baum vorliegenden Sperren. Dabei wird für jede Sperre die dazugehörige Transaktion, der Typ (Knotensperre, Kantensperre u.s.w.), der Modus (R, U, X u.s.w.) und, falls vorhanden, die passenden Parameter (zum Beispiel zu welcher Kante die Kantensperre gehört) angezeigt.

Die Informationen über den Aufbau der vom XTC-Server geöffneten XML-Dokumente sowie über die Sperren auf den einzelnen Knoten werden dem LockMonitor von einem Remote Object, dem LockStatsManager über RMI zur Verfügung gestellt. Dieses Objekt hat direkten Zugriff auf die Sperrtabellen des Servers und kann somit die notwendigen Informationen über Remote-Methoden-Aufrufe den Clients zur Verfügung stellen.

Beim Auswählen eines Knotens im JTree holt sich der LockMonitor dessen Kindknoten vom LockStatsManager und speichert diese in einer Hash-Tabelle ab. So kann beim zweiten Auswäh-

Abbildung 23 Screenshot des LockMonitors in einem Beispiel-Szenario



len dieses Knotens die lokale Kopie der Kindknoten angezeigt werden. Es kann aber nun passieren, dass inzwischen ein Client Knoten entfernt hat, oder neue Knoten in den ausgewählten Teilbaum hinzugefügt hat. Aus diesem Grund prüft der LockMonitor bei der Auswahl eines Knotens im JTree die Anzahl der Kinder dieses Knotens auf dem Server. Unterscheidet sich diese Anzahl mit der Anzahl der Kinder auf der lokalen Kopie, so wird der Knoten automatisch aktualisiert. Da es aber mit diesem Verfahren nicht möglich ist festzustellen, ob ein Client nicht einen Knoten entfernt hat und einen anderen hinzugefügt hat, weil die Anzahl der Knoten sich nicht geändert hat, hat der Benutzer die Möglichkeit, die Aktualisierung eines Knotens zu erzwingen. Dies passiert durch einen einfachen Rechtsklick auf den entsprechenden Knoten.

Kapitel 6 Auswertung

Im vorhergehenden Kapitel haben wir die Architektur und die Realisierung des Prototypen im Detail betrachtet. Wir wollen uns nun damit beschäftigen, dessen Verhalten anhand einiger Leistungsuntersuchungen mit Beispieldokumenten zu analysieren. Dabei geht es in erster Linie darum, die Auswirkungen der verschiedenen einstellbaren Parameter auszuwerten.

6.1 Voraussetzungen und Vorbereitungen

Alle in diesem Kapitel aufgelisteten Testreihen wurden auf einem Rechner mit folgender Hard- und Softwarekonfiguration durchgeführt: Pentium 4 2.4GHz, 768 MB Hauptspeicher, Windows XP und Java Virtual Machine 1.4.1.

Um die Testergebnisse nicht durch externe Faktoren wie Kommunikationsoverhead zu verfälschen, werden wir sowohl den XTC-Server als auch den XTC-Client innerhalb derselben virtuellen Maschine laufen lassen. Dies hat den großen Vorteil, dass bei der Kommunikation zwischen Server und Client kein RMI verwendet werden muss und die Parameterübergabe über Referenzen erfolgen kann. Damit entfällt auch der Aufwand, die ausgetauschten Daten bei jedem Aufruf serialisieren und wieder deserialisieren zu müssen (siehe Abschnitt 5.1.1). Das Zusammenfassen beider Applikationen wurde dadurch möglich, dass der in Abschnitt 5.4.1 beschriebene Client in den Server mit integriert wurde und so die Clientfunktionalität direkt über eine Eingabeaufforderung im Server aufgerufen werden kann.

6.1.1 Testdokumente

Die XML-Dokumente, die wir für die folgenden Tests verwendet haben, wurden zu diesem Zweck generiert. Dazu bietet sich besonders der von IBM Alphaworks frei zur Verfügung gestellte XML-Generator [1] an. Leider ermöglicht es dieser Generator nicht, die Größe des resultierenden Dokuments oder die Anzahl der Knoten im Dokument einzustellen. Aus diesem Grund haben wir einen eigenen XML-Generator entwickelt, der genau diese Voraussetzung erfüllt. Dafür mussten wir aber auf komplexere Funktionalitäten, die uns der IBM Alphaworks Generator geboten hätte, verzichten. Diese sind aber für die folgenden Testreihen von weniger großer Bedeutung, sodass wir uns doch für den eigenen Generator entschieden haben. Wir werden die Funktionsweise beider Generatoren im folgenden Abschnitt kurz erläutern und beschreiben.

6.1.1.1 IBM Alphaworks XML-Generator

Der IBM Alphaworks XML-Generator elaubt es, ausgehend von einer DTD-Beschreibung ein wohlgeformtes XML-Dokument zu generieren, was zu der zugrundeliegenden DTD konform ist. Der XML-Generator erlaubt die Angabe folgender Parameter:

- *root*: Gibt den Namen des Root-Elements des Dokuments an.
- *numLevels*: Bestimmt die maximale Anzahl der Ebenen im XML-Dokument. Wird die Baumtiefe während der Generierung des Dokuments überschritten, so werden keine optionalen Elemente mehr hinzugefügt (durch * oder ? in der DTD gekennzeichnet). Obligatorische Element werden in ihrer minimalen Anzahl eingefügt, um die Expansion des Baumes so schnell wie möglich zu unterbrechen ohne dabei die Vorschriften der DTD zu verletzen.
- *numRepeats*: Bezeichnet die maximale Anzahl der Wiederholungen eines Elements, das in der DTD mit * oder ? gekennzeichnet ist. Der Generator wird beim Generieren der Kindknoten jedes einzelnen Knotens im Baum eine Zufallszahl zwischen 0 und *numRepeats* bestimmen und entsprechend viele Kindknoten hinzufügen. Dies gilt nur dann, wenn die aktuelle Baumtiefe die maximale Baumtiefe (siehe *numLevels*) noch nicht überschritten hat.
- *fixedOdds*: 1/*fixedOdds* ist die Wahrscheinlichkeit, dass ein so genanntes *fixed attribute* erscheint. Wenn *fixedOdds* gleich 0 ist, dann wird niemals ein fixed attribute erscheinen, ist es gleich 1, so erscheint es immer. *fixedOdds* muss ein Integerwert sein, ansonsten tritt ein Fehler auf.
- *impliedOdds*: 1/*impliedOdds* ist die Wahrscheinlichkeit, dass ein optionales Attribut nicht erscheint. Wenn *impliedOdds* gleich 1 ist, dann wird niemals ein optionales Attribut erscheinen, ist es gleich 0, so erscheint es immer. *impliedOdds* muss ein Integerwert sein, ansonsten tritt ein Fehler auf.
- *defaultOdds*: 1/*defaultOdds* ist die Wahrscheinlichkeit, dass ein Attribut mit einem Vorgabewert nicht erscheint. Wenn *defaultOdds* gleich 1 ist, dann wird niemals ein optionales Attribut erscheinen, ist es gleich 0, so erscheint es immer. *defaultOdds* muss ein Integerwert sein, ansonsten tritt ein Fehler auf.
- *randomSeed*: *randomSeed* ist ein Integerwert, der als Startwert für den Zufallsgenerator benutzt wird. Wird *randomSeed* nicht explizit angegeben, so wird die aktuelle Zeit in Millisekunden verwendet, wie sie von Date.getTime() berechnet wird.
- *maxIdRefs*: *maxIdRefs* ist die maximale Anzahl an IDs, die ein IDREFS Attribut referenzieren kann. Für jedes IDREFS Attribut wird eine Zufallszahl zwischen 1 und *maxIdRefs* erzeugt, und entsprechend viele IDs werden für den Attributwert zufällig ausgewählt, falls solche verfügbar sind. Andernfalls wird der Attributwert auf ‘noValue’ gesetzt.
- *maxNMTokens*: *maxNMTokens* ist die maximale Anzahl an NMTOKENs ein NMTOKENS Attribut enthalten kann. Für jedes NMTOKENS Attribut wird eine Zufallszahl zwischen 1 und *maxNMTokens* erzeugt, und entsprechend oft wird die Standard NMToken-Zeichenkette, durch Leerzeichen getrennt, konkateniert und als Wert für das Attribut gesetzt.
- *maxEntities*: *maxEntities* ist die maximale Anzahl an ENTITYs, die ein ENTITIES Attribut enthalten kann. Für jedes ENTITIES Attribut wird eine Zufallszahl zwischen 1 und *maxEntities* erzeugt, und entsprechend viele ungeparste Entities werden zufällig für den Wert des Attributs ausgewählt.

- *entConfFile*: *entConfFile* ist eine Datei, die alle Entities enthält, die in PCDATA Feldern eingesetzt werden können.
- *entOdds*: 1/*entOdds* ist die Wahrscheinlichkeit, dass ein Entity in einem PCDATA Feld erscheint. Wenn entOdds gleich 1 ist, dann wird in jedem PCDATA Feld ein Entity erscheinen. Falls keine Entities in den PCDATA Feldern erwünscht sind, dann sollte dieses Flag nicht angegeben werden.

Da der Wert des *numRepeats* Parameters für jeden Knoten zufällig bestimmt wird, kann es vorkommen, dass die maximale Tiefe des generierten Baums (durch *numLevels* bestimmt) überhaupt nicht erreicht wird. Das führt dazu, dass die Größe des resultierenden Dokuments stark vom Zufall abhängig ist und vom Benutzer nicht gesteuert werden kann. Es kann also notwendig werden, das Programm mehrere Male zu starten, bevor man ein Dokument erhält, das einigermaßen den Vorstellungen bezüglich der Anzahl der Knoten oder der Dateigröße entspricht.

6.1.1.2 XTC-Generator

Für die Leistungsuntersuchungen des XTC-Servers ist es allerdings sehr von Vorteil, die Anzahl der Knoten des zu generierenden Dokuments festlegen zu können. Dies ermöglicht es uns dann, die Anzahl der Gesamtknoten im Dokument mit der Anzahl der für dessen Verarbeitung notwendigen Sperren zu vergleichen.

Wegen dieses Defizits des XML-Generators vom IBM Alphaworks ist der XTC-Generator entwickelt worden. Er ist dem IBM Generator gegenüber insofern einfacher, dass er die Dokumente rein zufällig aufbaut, ohne dabei Rücksicht auf Einschränkungen durch DTD-Beschreibungen zu nehmen. Es gibt deswegen auch nur entsprechend wenig Einstellmöglichkeiten:

- docSize: Bestimmt die Anzahl der generierten Element-, Attribut- und Textknoten im Dokument. Die Erzeugung von anderen Knotentypen wie beispielsweise Processing Instruction oder Kommentaren wird vom XTC-Generator nicht unterstützt. Da das erzeugte Dokument keinerlei Einschränkungen durch DTD-Beschreibungen unterliegt, kann der Aufbau des Dokuments genau dann abgebrochen werden, wenn die maximale Anzahl der Knoten im Dokument erreicht ist.
  Bemerkung: Man beachte, dass es sich hierbei um XML-Knoten, nicht aber um taDOM-Knoten handelt. Die aus der taDOM-Abstraktion resultierenden zusätzlichen String- und AttributeRoot-Knoten (siehe Abschnitt 4.1.1 und 4.1.2) werden bei der Zählung nicht berücksichtigt.
- numLevels: Gibt die maximale Tiefe des XML-Baumes an. Wurde die maximale Tiefe des Baumes vor der maximalen Knotenanzahl erreicht, so bricht der Algorithmus trotzdem ab.
- maxAttributes, minAttributes: Begrenzt die Anzahl der Attribute, die pro Element angelegt werden jeweils nach oben und nach unten. Für jedes Element wird zufällig die Anzahl der Attribute bestimmt, die den Einschränkungen genügt. Ist minAttributes größer als maxAttributes, ist das Verhalten nicht definiert.
- maxChildren, minChildren: Legt die maximale und minimale Anzahl der Kindknoten pro Element fest.

Beim Anlegen eines neuen Kindknotens wird dieser mit einer Wahrscheinlichkeit von 50% zu einem weiteren Element oder zu einem Textknoten. Die Namen der Elemente, sowie die Namen der Attribute und Werte der Attribut- und Textknoten werden ebenfalls zufällig erzeugt.

Die Vorgehensweise bei der Erstellung eines solchen XML-Dokuments ist nicht weiter kompliziert. Mit Hilfe des DOM-API von Java wird ein DOM-Baum mit den entsprechenden Eigenschaften im Speicher aufgebaut und mit Hilfe eines Transformers aus dem *javax.xml.transform* Package zu einem regulären XML-Dokument serialisiert.

6.1.2 Verwendete Dokumente

Da sich die Größe der mit dem IBM Alphaworks XML-Generator erzeugten XML-Dokumente sehr schlecht steuern lässt, haben wir uns entschieden, sämltliche Dokumente für die folgenden Testreihen mit dem XTC-Generator zu erzeugen. Die Untersuchungen basieren auf acht Dokumenten verschiedener Größe. Die Eigenschaften sind in Tabelle 2 aufgelistet.

Tabelle 2 Auflistung der XML-Dokumente, die wir für die folgenden Untersuchungen erzeugt haben

Dokument	# Knoten	Größe	Baumtiefe	max Attr.	min Attr.	max Child	min Child
1000.xml	1000	10 Kb	5	8	2	10	2
5000.xml	5000	50 Kb	8	8	2	10	2
10000.xml	10000	100 Kb	10	8	2	10	2
25000.xml	25000	250 Kb	10	10	2	10	2
50000.xml	50000	500 Kb	15	10	2	10	2
75000.xml	75000	750 Kb	20	12	2	10	2
100000.xml	100000	1 MB	20	12	2	20	5
150000.xml	150000	1,5 MB	20	12	2	20	5

Die Spalten in der obigen Tabelle haben folgende Bedeutungen:

- Dokument: Der Name des XML-Dokuments
- # Knoten: Die Anzahl der Knoten (Elemente, Attribute, Textknoten) im Dokument
- Baumtiefe: Die maximale Distanz von der Baumwurzel zu den Blättern
- max Attr.: Die maximale Anzahl an Attributen pro Element
- min Attr.: Die minimale Anzahl an Attributen pro Element
- max Child: Die maximale Anzahl an Kindknoten pro Element
- min Child: Die minimale Anzahl an Kindknoten pro Element

6.2 Messungen

Im folgenden Abschnitt werden wir uns die Messergebnisse des XTC-Servers unter verschiedenen Voraussetzungen und Konfigurationen ansehen. Ein Zeitmesswert besteht dabei aus dem Mittelwert der gemessenen Werte aus vier aufeinanderfolgenden Messungen. Aufgrund von systeminternen Faktoren wie Cachezustand oder Threadscheduling können die Messwerte leicht voneinander abweichen. Falls nicht anders angegeben, verwenden wir für alle folgenden Test-

läufe dasselbe Zugriffsschema. Der Baum wird dabei rekursiv von der Wurzel bis zu den Blättern durchlaufen. Die rekursive Funktion *recursiveRead()* wird auf der Wurzel aufgerufen. Falls der zugrundeliegende Knoten ein Elementknoten ist, so wird für jeden Knoten aus der Ergebnismenge von *getAttributes()* und *getChildNodes() recursiveRead()* aufgerufen. Ist der zugrundeliegende Knoten ein Attribut- oder Textknoten, so wird lediglich *getValue()* aufgerufen. Der Algorithmus teminiert, wenn alle Knoten des Baumes einmal durchlaufen worden sind.

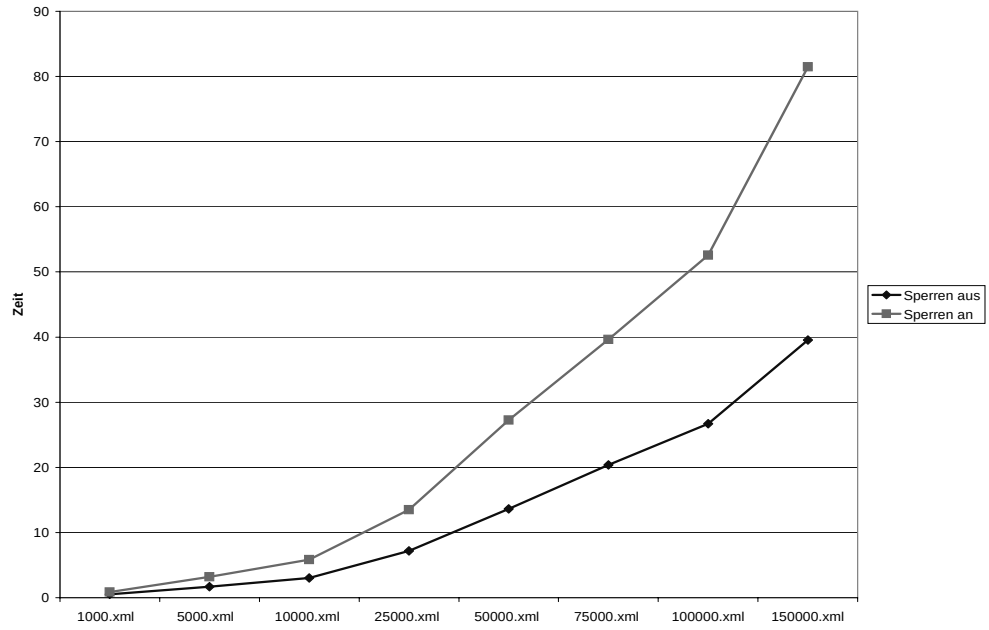
Wir werden in diesem Kapitel lediglich Grafiken als Auswertung der Testläufe betrachten. Eine Auflistung der gesamten Messwerte befindet sich in Anhang C.

6.2.1 Aufwand der Sperrverwaltung

In einer ersten Testreihe möchten wir den Aufwand der Sperrverwaltung bestimmen. Dazu benutzen wir die in Abschnitt 6.1.2 beschriebenen XML-Dokumente. Abbildung 26 zeigt den Zeitverlauf für das Verarbeiten der Dokumente bei ein- und ausgeschalteter Sperrverwaltung. Für den Test wurden 10 Transaktionen gleichzeitig gestartet, die das Dokument mit dem *recursiveRead()* Algorithmus komplett von der Wurzel bis zu den Blättern durchlaufen. Die Option zur Verwendung von AttributeRoot ist aktiviert und die maximale Sperrtiefe ist nicht begrenzt.

Nach dem in Kapitel 4 vorgestellten Sperrprotokoll sind für das Lesen des Wertes eines Attribut- oder Textknotens zwei Sperren notwendig. Dies erklärt auch, weshalb pro Transaktion mehr Sperren angelegt werden müssen, als im Dokument Knoten enthalten sind.

Abbildung 24 Zeitverlauf für das Verarbeiten der Dokumente bei ein- und ausgeschalteter Sperrverwaltung



Wie zu erwarten, ist der Zeitaufwand bei ausgeschalteter Sperrverwaltung geringer, als wenn diese eingeschaltet ist. Durch die Deaktivierung der Sperrverwaltungskomponente kann der Durchsatz des Servers sogar quasi verdoppelt werden. Der Zeitaufwand verhält sich weitgehend linear zur Anzahl der Knoten im bearbeiteten Dokument. Dies ist allerdings auf dem Bild nicht auf den ersten Blick zu erkennen, da die Anzahl der Knoten auf der X-Achse nicht linear wächst.

6.2.2 Einfluss des Zugriffsschemas

Im vorigen Abschnitt hat eine Transaktion auf jedem Elementknoten *getChildNodes()* aufgerufen, um rekursiv in den XML-Baum absteigen zu können. Hätte sie die Kindknoten eines Element mit der Abfolge *getFirstChild(), getNextSibling(), getNextSibling()...* durchlaufen, so wäre eine Vielzahl zusätzlicher Kantensperren angefallen, wie wir in der nächsten Testreihe sehen werden. Das so definierte Zugriffsschema bezeichnen wir im Folgenden als *recursiveReadSlow.*

Abbildung 25 Zeitverlauf (Linien) und Anzahl der benötigten Sperren (Balken) zum Durchlaufen der Dokumente mit verschiedenen Zugriffsmustern in Abhängigkeit der Dokumentgröße

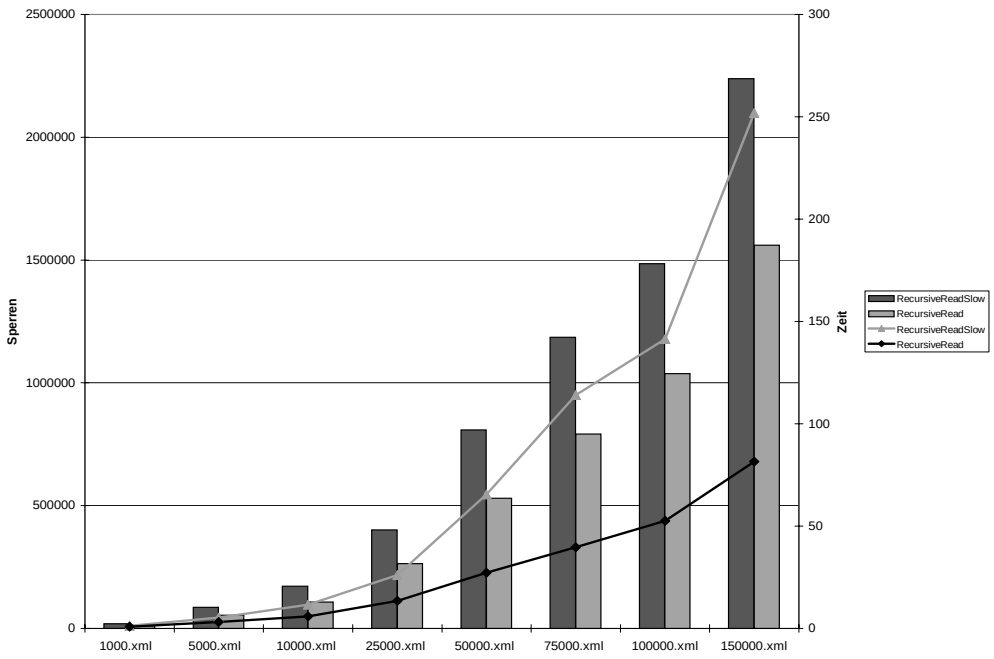


Abbildung 25 zeigt, wie wichtig es ist, das Zugriffsschema auf das Dokument so zu wählen, dass möglichst wenig Sperren angefordert werden müssen, da sich dies stark auf das Verhalten des gesamten Systems auswirkt.

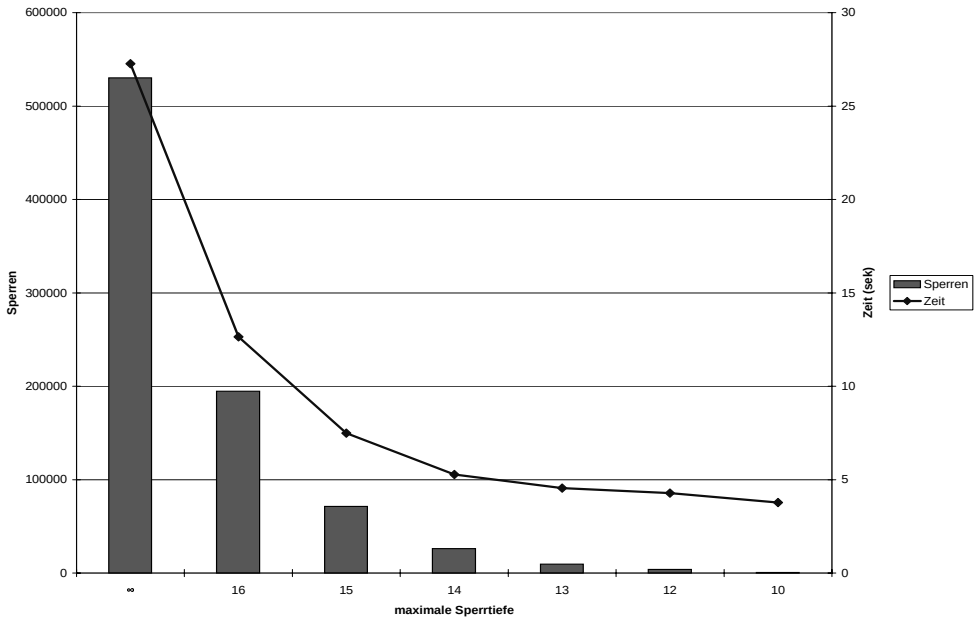
Die Balken in der Grafik stellen die Anzahl der jeweils benötigten Sperren für *recursiveRead* und *recursiveReadSlow* dar. Die Linien entsprechen der benötigten Zeit zum Durchlaufen der Dokumente nach dem jeweiligen Zugriffsschema.

6.2.3 Einfluss der maximalen Sperrtiefe

Einer der wohl weitreichendsten Einstellungen einer Transaktion ist die der maximalen Sperrtiefe. Vor allem bei Dokumenten, bei denen sich der größte Teil der Knoten in den Blättern befindet wirkt sich diese Einstellung sehr stark aus. Wir betrachten in dieser Messreihe erneut die Dokumente aus Abschnitt 6.2.1.

Abbildung 26 zeigt, wie sich die Veränderung der maximalen Sperrtiefe auf den Zeitbedarf beim Durchlaufen eines kompletten Dokumentes auswirkt. Die Angabe ‘unendlich’ als Sperrtiefe besagt lediglich, dass der XTC-Server in keinem Fall automatisch eine SR-Sperre auf einem Knoten anlegt und den entsprechenden Unterbaum komplett zurückliefert. Die Sperrgranularität ist also auf der feinsten Stufe. In den anderen Fällen ist zu beachten, dass nicht nur die Sperren unterhalb der maximalen Sperrtiefe nicht mehr angelegt werden, sondern dem Client auch direkt der komplette Unterbaum des betroffenen Knotens zurückgeliefert wird. So kann der Client beim Durchlaufen eines Teilbaums unterhalb der maximalen Sperrtiefe auf seine lokale Kopie zugreifen und belastet dadurch den Server nicht zusätzlich. Dadurch lassen sich auch die extrem kurzen Laufzeiten bei geringer Sperrtiefe erklären.

Abbildung 26 Zeitverlauf (Linie) und Anzahl der benötigten Sperren (Balken) zum Durchlaufen des 50000.xml-dokuments in Abhängigkeit der maximalen Sperrtiefe.



Die Grafik ist hier für das Dokument 50000.xml dargestellt. Die Messwerte für die restlichen Dokumente sind in Anhang C aufgeführt. Wir stellen fest, dass schon bei geringer Veränderung der maximalen Sperrtiefe neben der Anzahl der notwendigen Sperren auch der Zeitbedarf stark abfällt. Dies ist dadurch zu erklären, dass sich ein Großteil der Knoten in den Blättern des XML-Dokuments befinden.

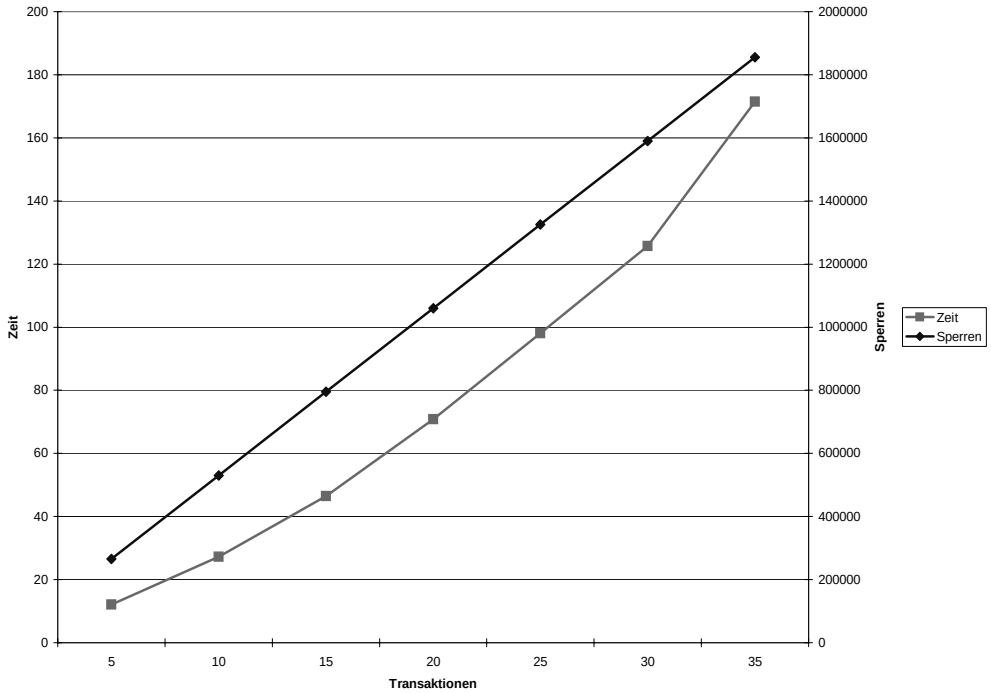
6.2.4 Nichtlineares Verhältnis zwischen Zeit und Anzahl der Transaktionen

Um speziell hervorzuheben, dass der Zeitaufwand zum Anlegen einer Sperre nicht konstant ist, sondern von der Anzahl der bereits vorhandenen Sperren abhängt, führen wir das Experiment aus Abschnitt 6.2.3 noch einmal durch. Diesmal steigern wir die Anzahl der simultanen Transaktionen schrittweise von fünf auf dreißig.

Dabei kann man leicht erkennen, dass sich die benötigte Zeit nicht proportional zu den angelegten Sperren verhält. Dies liegt daran, dass bei jedem Anlegen einer neuen Sperre, die Kompatibilität mit allen anderen auf dem entsprechenden Knoten bereits vorhandenen Sperren überprüft werden müssen. Je mehr Sperren bereits auf einem Knoten vorhanden sind, desto aufwendiger wird die Überprüfung der Kompatibilität.

Abbildung 27 zeigt den Zeitaufwand in Abhängigkeit von der Anzahl der gleichzeitigen Transaktionen. Man kann leicht erkennen, dass die Zeit polynomial gegenüber der Anzahl der Transaktionen wächst.

Abbildung 27 Zeitverlauf und Anzahl der Sperren für das 50000.xml-dokument in Abhängigkeit der Anza parallelen Transaktionen



6.2.5 Verwendung von AttributeRoot

Eine wichtige Eigenschaft des taDOM-Baums ist der zusätzliche AttributeRoot-Knoten, der das Element mit seinen jeweiligen Attributen verbindet. Man kann sich nun fragen, ob eine solche Modellierung überhaupt sinnvoll ist und sich der Aufwand lohnt, einen weiteren Knotentypen einzuführen. Aus diesem Grund werden wir im folgenden Experiment die Verwendung des Attri-

buteRoot-Knotens im XTC-Server ausschalten und bei jedem *getAttributes()* Aufruf alle Attributknoten des entsprechenden Elements einzeln mit einer NR-Sperre versehen.

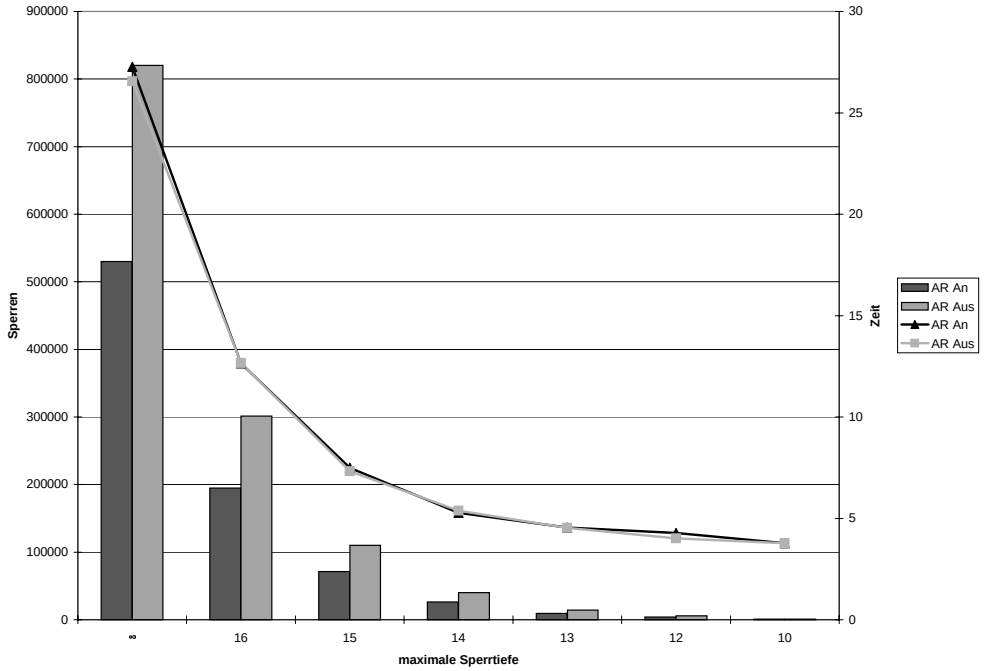
Abbildung 28 zeigt eindrucksvoll wie sich die Zahl der notwendigen Sperren gegenüber dem Experiment aus Abbildung 27 erhöht hat. Auch wenn der Zeitbedarf nicht proportional zu der Anzahl der Sperren ist, so trifft dies sehr wohl für den Speicherbedarf zu. Die Speicherbelastung konnte in unserem Fall durch die Verwendung des zusätzlichen AttributeRoot-Knotens um über 30% gesenkt werden. Dies mag auf den ersten Blick keine besonderen Vorteile bringen. Diese werden erst dann sichtbar, wenn der physikalische Speicher des Servers aufgebraucht ist und auf Auslagerungsstrategien zurückgegriffen werden muss. Wenn man berücksichtigt, dass die Zugriffszeiten auf Magnetfestplatten um Faktor 10^6 langsamer sind als auf physikalischen Speicherbaussteinen, dann kann man erahnen, welchen Leistungseinbruch der Server in einem solchen Fall erleben würde.

Der Grund für den geringen bis kaum vorhandenen Zeitvorteil durch die Verwendung des zusätzlichen AttributeRoot-Knotens lässt sich folgendermaßen erklären. Bei jedem Zugriff auf einen Knoten muss erst überprüft werden, ob die entsprechende Transaktion auch die Zugriffsrechte, sprich, die richtigen Sperren auf diesem Knoten besitzt. Dies ist genau dann gegeben, wenn die Transaktion eine beliebige Sperre auf dem aktuellen Knoten, eine LR-, U- oder X-Sperre auf dem direkten Vorgängerknoten oder eine SR-, U- oder X-Sperre auf einem der Vorgänger bis zur Dokumentwurzel besitzt.

Angenommen, wir möchten nun mit *getValue()* den Wert eines Attributs abrufen. Dann wird zunächst überprüft, ob die Transaktion eine beliebige Sperre auf dem aktuellen Knoten besitzt. Dies ist der Fall, wenn die Verwendung von AttributeRoot ausgeschaltet ist, da dann auf jedem Attributknoten eine eigene NR-Sperre angelegt wird. Wird AttributeRoot jedoch verwendet, so befindet sich keine NR-Sperre auf dem aktuellen Attributknoten. Deswegen muss im nächsten Schritt überprüft werden, ob sich eine LR-Sperre auf dem direkten Vorgänger befindet. Das wäre in diesem Fall AttributeRoot des Knotens, dem die Attribute zugeordnet sind. Dieser direkte Vorgänger ist aber nicht unmittelbar verfügbar und muss erst über die XMLAccess-Schnittstelle geholt werden. Die zusätzliche Operation alleine für das Überprüfen der Zugriffsrechte kompensiert die ganzen zeitlichen Vorteile die sich daraus ergeben, dass viel weniger Sperren angefordert und überprüft werden müssen.

Unter diesen Umständen ist es nicht besonders sinnvoll den zusätzlichen AttributeRoot-Knoten zu implementieren. Wir gehen davon aus, dass ein Datenbankmanagementsystem genügend Speicher zur Verfügung hat, und wir im Allgemeinen auf Auslagerungsstrategien verzichten können. Könnte man aber auf eine native Speicherungskomponente zurückgreifen, die nicht bei jedem Zugriff einen DOM-Knoten in einen XTC-Knoten umwandeln müsste (wenn die Daten zum Beispiel schon nativ als XTC-Knoten vorliegen), so würde der zusätzliche Zugriff auf den Vorgänger der Attributknoten zur Überprüfung der Zugriffsrechte erheblich weniger Zeit in Anspruch nehmen. In einem solchen Fall könnte die Verwendung der AttributeRoot-Knoten durchaus sinnvoll sein. Um dies aber genau festlegen zu können, müsste der Versuch noch einmal durchgeführt werden, wenn eine solche native Speicherungskomponente zur Verfügung steht.

Abbildung 28 Zeitverlauf (Linien) und Anzahl der benötigten Sperren (Balken) für das Durchlaufen des 50000.xml-dokuments jeweils mit ein- und ausgeschaltetem AttributeRoot



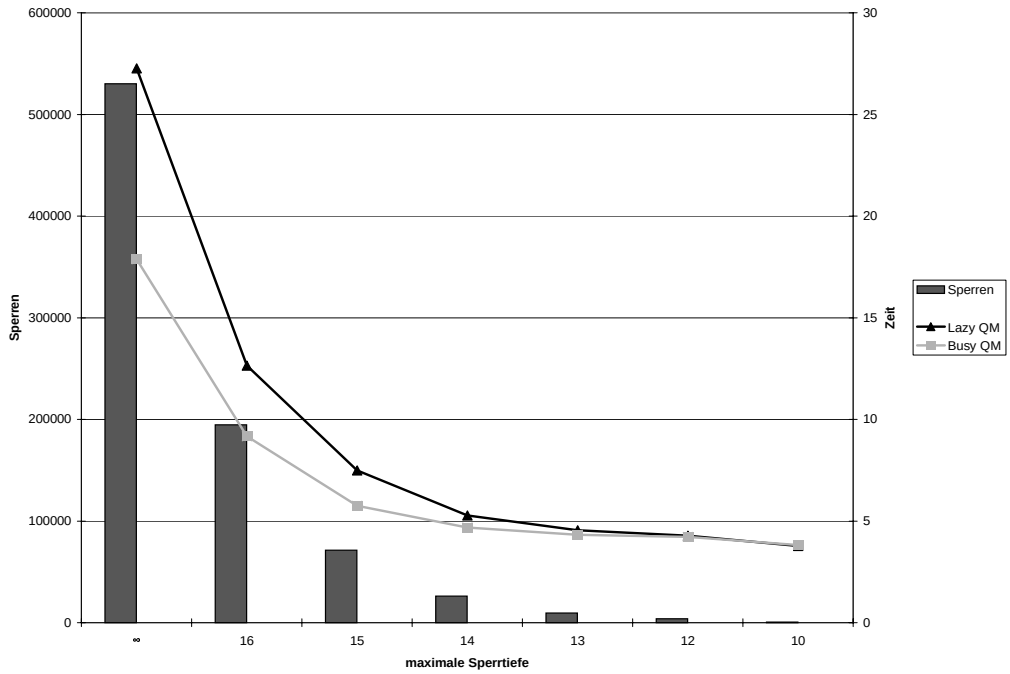
6.2.6 Einfluss des QueueManagers

Wie bereits in Abschnitt 5.2.1 erwähnt wurde, ist das Verwenden des LazyQueueManagers zwar betriebsmittelsparender, führt aber bei Volllast zu einer schlechteren Ausbeute, da die Zustandswechsel der Threads relativ viel Totzeit verursachen. Der BusyQueueManager, der die Warteschlangen laufend auf neue Einträge überprüft verbraucht zwar mehr Betriebsmittel, reagiert aber im Allgemeinen schneller auf Änderungen in der Warteschlange.

Aus diesem Grund hat der BusyQueueManager vor allem bei Volllast des Servers eine bessere Ausbeute der Prozessorleistung, was wir mit folgendem Experiment unterstreichen wollen. Wir führen das Experiment jeweils mit LazyQueueManager und BusyQueueManager durch. Dabei verwenden wir das Dokument aus Abschnitt 6.2.3 und starten zwanzig simultane Transaktionen, wobei die Option zur Verwendung von AttributeRoot-Knoten ausgeschaltet ist, um das Anlegen möglichst vieler Sperren zu verursachen. So kann man die Auswirkungen der QueueManagers besser erkennen. Abbildung 29 zeigt den Zeitverlauf für die beiden Verfahren im Vergleich.

Die Grafik zeigt, dass zwar die Anzahl der angeforderten Sperren in beiden Fällen gleich sind, der Datendurchsatz mit dem BusyQueueManager im Extremfall bis zu 60% höher ist als beim LazyQueueManager.

Abbildung 29 Zeitverlauf (Linien) und Anzahl der Sperren (Balken) zum Durchlaufen des 50000.xml-dokuments unter Verwendung des Busy- und des Lazy QueueManagers



6.2.7 Veränderte Systemarchitektur

Im folgenden Abschnitt wollen wir uns mit dem Testen der Implementierung der neuen Systemarchitektur (wie sie in Abschnitt 5.3 beschrieben ist) beschäftigen.

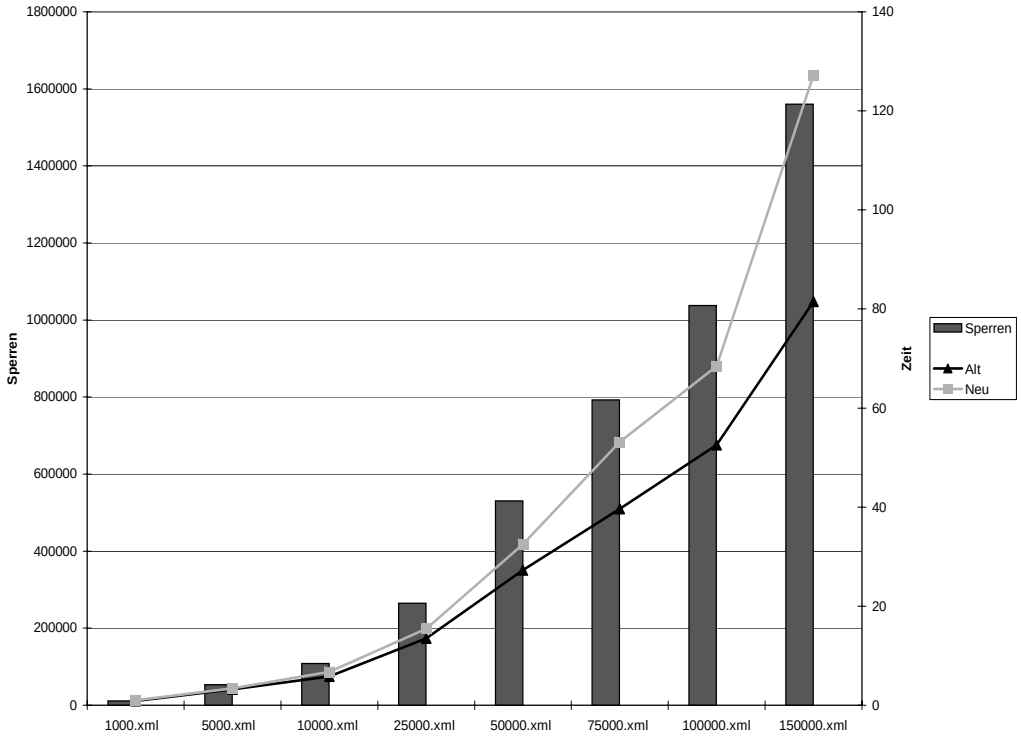
Zweck der neuen Architektur war es, die Nebenläufigkeit der verschiedenen Transaktionen zu erhöhen, sodass Anfragen parallel verarbeitet werden können und nicht wie bisher sequentiell aus einer Warteschlange abgearbeitet werden müssen. Wie bereits in Abschnitt 5.3 erwähnt, sind wir bei der Implementierung auf mehr Probleme gestoßen, als es auf den ersten Blick den Anschein hatte. Die gewünschte Parallelität wurde stark eingeschränkt durch die Notwendigkeit, die Zugriffe auf die Sperrtabellen auf unterster Ebene zu synchronisieren, um zu vermeiden, dass von mehreren Threads gleichzeitig lesend und schreibend auf ein Objekt zugegriffen wird. Das hat dann sogar soweit geführt, dass beim Hinzufügen der ersten Sperre auf einem Knoten die komplette Sperrtabelle exklusiv gesperrt werden musste.

Durch diese Umstände hat sich der geplante Vorteil der erhöhten Parallelität der Transaktionen eher als Nachteil herausgestellt. Die Anfragen können nun nicht mehr am Stück ausgeführt werden, sondern die Bearbeitung wird immer wieder durch Synchronisationssperren von anderen parallelen Threads unterbrochen. Dies führt dazu, dass die Threads häufig angehalten werden müssen, wodurch der Aufwand und so auch der Zeitverlust durch Threadscheduling stark anwächst. Und je kleiner die Synchronisationssperrgranularität, also je kleiner die kritischen Abschnitte sind, desto größer wird zwar die Parallelität der Threads, aber um so häufiger müssen auch die Threads synchronisiert werden.

Aus dem Experiment mit den verschiedenen QueueManagern aus Abschnitt 6.2.6 haben wir gesehen, wie stark die Leisungsfähigkeit des Systems durch häufige Threadwechsel beeinträchtigt werden kann. Mit dem BusyQueueManager konnten bis zu 60% mehr Anfragen bearbeitet werden als mit dem LazyQueueManager.

Der Verdacht, dass die gesteigerte Parallelität bezüglich der Leistungsfähigkeit des Systems die Nachteile nicht ausgleichen werden kann, hat sich dann auch bei den Versuchen bestätigt. Abbildung 30 zeigt den Zeitverlauf der neuen und der alten Systemarchitektur.

Abbildung 30 Zeitverlauf (Linien) und Anzahl der Sperren (Balken) zum Durchlaufen der Dokumente nach der alten und der neuen Systemarchitektur in Abhängigkeit der Größe der Dokumente



Die Grafik zeigt, dass bei gleichbleibender Anzahl angeforderter Sperren, die neue Systemarchitektur aufgrund häufiger Kontextwechsel bei der Threadsynchronisierung deutlich langsamer ist als die grob synchronisierte Architektur mit den Warteschlangen.

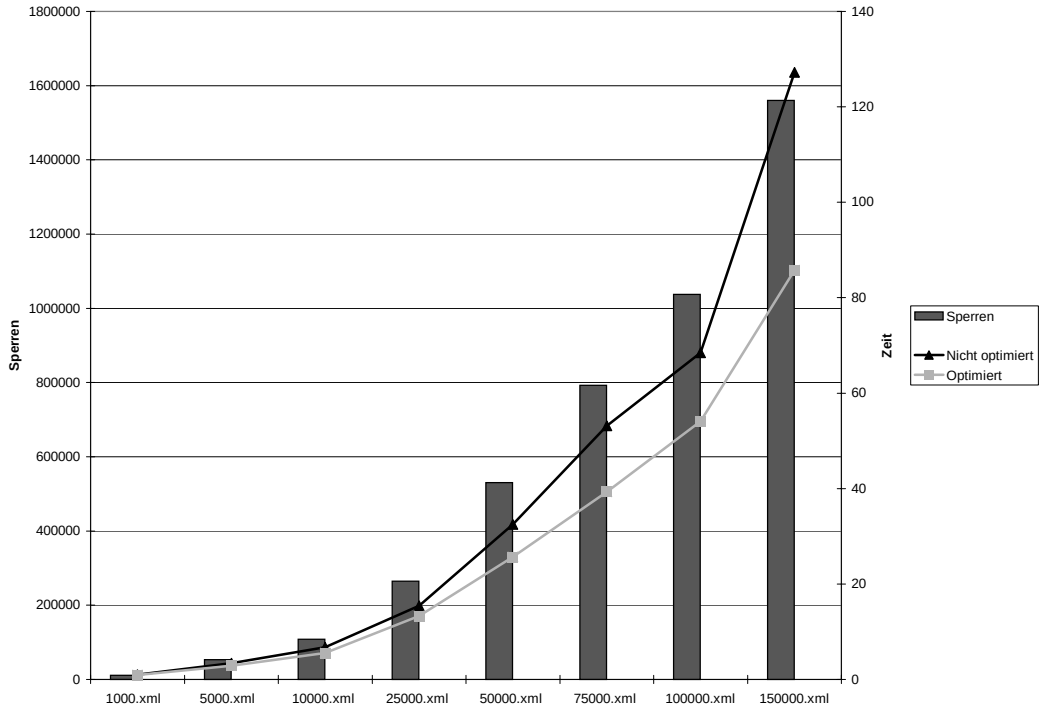
6.2.7.1 Bottom-Up Sperranforderungsverfahren

Wie bereits in Abschnitt 5.3.3 erwähnt, lässt sich das Sperranforderungsverfahren für die Knotensperren optimieren, die bei einer Anfrage vom betroffenen Knoten bis zur Dokumentwurzel angefordert werden müssen. Dabei muss der Weg von unten (also dem entsprechenden Knoten) bis oben (Wurzel) nur solange verfolgt werden, wie auch tatsächlich eine neue Sperre angelegt wird. Ist auf einem Knoten bereits eine gleichwertige oder höherwertigere Sperre (siehe Abbildung 20 auf Seite 51) vorhanden, so kann der Vorgang abgebrochen werden, da die darauf-

hin folgenden Knoten auf dem Weg zur Wurzel schon von vorherigen Operationen mit den notwendigen Sperren versorgt wurden.

Abbildung 31 zeigt die Anzahl der angeforderten Sperren sowie die benötigte Zeit zum Durchlaufen eines Dokuments jeweils mit und ohne die beschriebene Optimierung. Die Anzahl der tatsächlich angelegten Sperren ist in beiden Fällen gleich.

Abbildung 31 Zeitverlauf (Linien) und Anzahl der Sperren (Balken) zum Durchlaufen der Dokumente jeweils mit dem optimierten und dem nicht-optimierten Zugriffsverfahren in Abhängigkeit der Größe der Dokumente



Die Grafik zeigt, dass durch das Vermeiden unnötiger Schritte bei der Sperranforderung mehr als 30% Zeitersparnis erzielt werden kann. Dies lässt sich am Beispiel des 150000.xml-dokuments erkennen.

6.2.7.2 Top-Down Sperranforderungsverfahren

Das Bottom-Up Verfahren hat den Nachteil, dass eventuell auftretende Konflikte erst relativ spät erkannt werden. Da für jede Operation auf dem XML-Dokument entsprechende Sperren auf der Dokumentwurzel angelegt werden müssen, ist es am wahrscheinlichsten, dass auch da die Konflikte entstehen. Beim Anlegen der Sperren von unten nach oben trifft man aber erst ganz am Ende des Vorgangs auf die Dokumentwurzel und muss, im Falle eines Konflikts, alle vorher angelegten Sperren wieder entfernen.

Um dies zu vermeiden, haben wir für den folgenden Versuch die Reihenfolge der Sperranforderungen umgekehrt, so dass sie von oben nach unten angelegt werden. Allerdings entfällt dabei

der Vorteil beim Bottom-Up Verfahren unter Umständen nicht alle Sperren überprüfen zu müssen. Damit ist die Anzahl der angeforderten Sperren für das Top-Down Verfahren gleich der des nicht-optimierten Bottom-Up Verfahrens.

Beim folgenden Versuch werden wir die Anzahl der Sperranforderungen bis zum Auffinden eines Konflikts für das Bottom-Up und das Top-Down Verfahren vergleichen. Da für diesen Versuch etwas andere Voraussetzungen nötig sind, haben wir mit Hilfe des IBM Alphaworks XML-Generators (siehe Abschnitt 6.1.1.1) sechs Dokumente verschiedener maximaler Baumtiefen generiert. Die dazu notwendige DTD ist in Abbildung 32 dargestellt.

Abbildung 32 DTD für die Generierung der Testdokumente

```
<!ELEMENT collection (item)+>
<!ELEMENT item (item)*>
<!ATTLIST item id ID #REQUIRED>
```

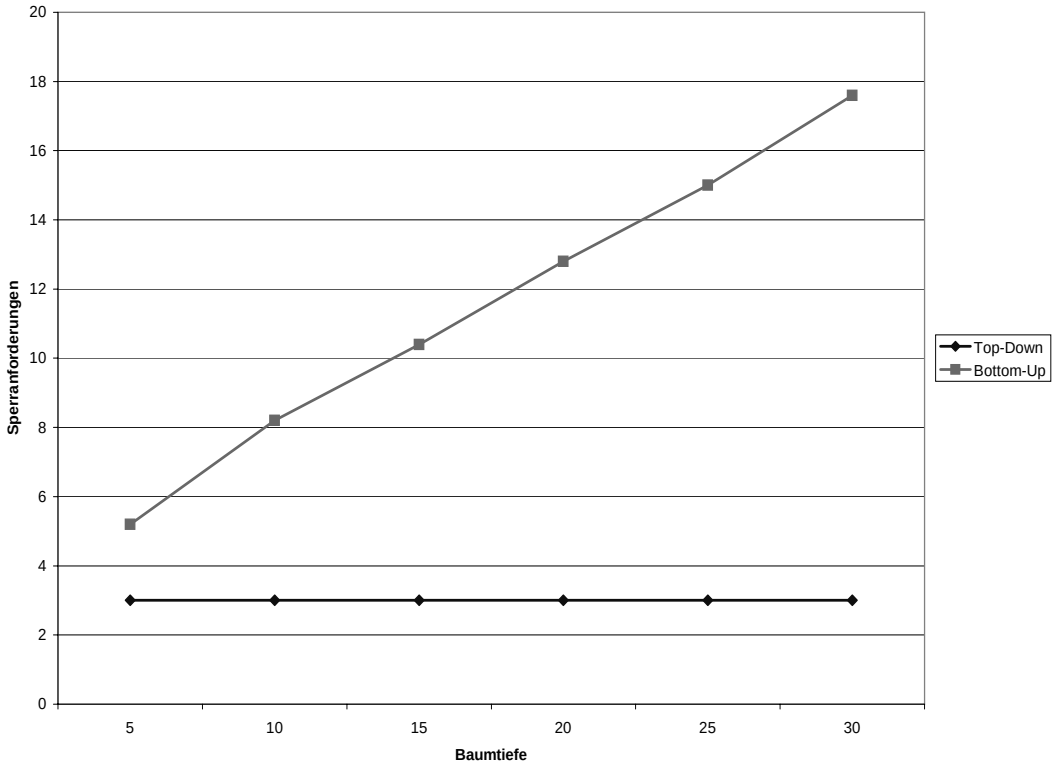
Wie aus der DTD erkennbar ist, besteht das Dokument aus beliebig verschachtelten *item*-Elementen, die alle ein nicht-optionales ID-Attribut enthalten. Der XML-Generator nummeriert die Werte der ID-Attribute folgendermaßen durch: a_0, a_1, a_2, ... a_n. Mit Hilfe dieser ID-Attribute können wir beim folgenden Versuch wahllos auf beliebige Knoten innerhalb des Dokuments zugreifen, indem wir zufällig einen ID-Attributwert generieren und den entsprechenden Knoten mit *getElementById()* abrufen.

Für die Testreihe gehen wir von folgendem Szenario aus: Eine Transaktion T_1 möchte das XML-Dokument komplett lesend durchlaufen und stellt die maximale Sperrtiefe auf einen geringen Wert (hier im Beispiel 1), um die Anzahl der notwendigen Operationen zu reduzieren. Dadurch wird direkt auf der Dokumentwurzel eine SR-Sperre angelegt. Nun kommt eine zweite Transaktion T_2 und holt sich mit der *getElementById()*-Methode einen zufällig ausgewählten Knoten aus dem Baum und löscht diesen. Dadurch wird auf dem Knoten selbst eine X-Sperre angelegt, auf dessen Vorgänger eine CX-Sperre und auf allen folgenden Vorgängerknoten bis zur Dokumentwurzel IX-Sperren. Diese IX-Sperre kollidiert dann mit der SR-Sperre der Transaktion T_1 auf der Dokumentwurzel.

Abbildung 33 zeigt die Anzahl der notwendigen Schritte bis zum Auffinden eines Konflikts jeweils für das Bottom-Up und das Top-Down Verfahren in Abhängigkeit der maximalen Baumtiefe der Dokumente. Die Testwerte bestehen jeweils aus dem Mittelwert von fünf aufeinanderfolgenden Messungen. Die Messungen können voneinander abweichen, weil der zu löschende Knoten zufällig aus dem Baum ausgewählt wird, und somit auch die Distanz bis zur Dokumentwurzel für die verschiedenen Knoten nicht konstant ist.

Wie zu erwarten war, wächst die Anzahl der notwendigen Schritte beim Bottom-Up Verfahren etwa linear mit der maximalen Baumtiefe. Beim Top-Down Verfahren ist die Anzahl der Schritte unabhängig von der maximalen Baumtiefe des Dokuments und beträgt konstant drei. Dies ist dadurch zu erklären, dass immer zuerst die Sperre auf der Dokumentwurzel angelegt wird, und der Konflikt direkt bei der ersten Sperranforderung auftritt. Die ersten beiden Sperren die angefordert werden sind in diesem Fall die logische Tagname Sperre für den Tagnamen und die logische ID-Sperre für die ID des Knotens der enfernt wurde.

Abbildung 33 Anzahl der notwendigen Schritte zum Auffinden eines Konflikts für das Bottom-Up und das Top-Down Sperranforderungsverfahren



Es sei aber darauf hingewiesen, dass es sich bei diesem Beispiel um ein extrem für das Top-Down Verfahren optimierte Szenario handelt und dass in der Realität die Unterschiede deutlich weniger beeindruckend ausfallen. Außerdem muss man beachten, dass die Optimierungen des Bottom-Up Verfahrens nicht mehr angewendet werden können, und deswegen das Top-Down Verfahren sehr viel schlechter abschneidet wenn keine Konflikte auftreten. Dann verhält sich Top-Down genau so wie das nicht-optimierte Bottom-Up Verfahren (siehe Abbildung 31).

Man muss berücksichtigen, dass beim Top-Down Verfahren auf jeden Fall erst alle Sperren bestimmt werden müssen, bevor diese von der Wurzel zum betroffenen Knoten angelegt werden können. Da dieser Vorgang aber nur von unten nach oben möglich ist, wäre es vorteilhafter, die Sperren auch direkt beim Bestimmen anzulegen, wobei wir wieder beim Bottom-Up Verfahren wären.

Es gibt also nur sehr wenige Ausnahmefälle (genau dann, wenn der Weg vom betroffenen Knoten lang ist, und ein Konflikt an der Dokumentwurzel entsteht) in denen das Top-Down Verfahren dem Bottom-Up Verfahren überlegen ist. Aus diesem Grund ist Bottom-Up im Allgemeinen erheblich leistungsfähiger.

Kapitel 7 Zusammenfassung und Ausblick

Die Synchronisation von nebenläufigen Zugriffen auf XML-Dokumente stellt bei der praktischen Verarbeitung von XML-Daten ein großes Problem dar. Bis zum heutigen Tag bieten nur sehr wenige kommerzielle Anbieter von relationalen beziehungsweise objektrelationalen Datenbanksystemen Erweiterungen an, um XML-Daten abzuspeichern. Diese erlauben aber nur sehr grobgranulare Sperrmechanismen oder haben andere schwerwiegende Nachteile. Zudem ist die Forschungsarbeit in diesem Gebiet noch nicht weit fortgeschritten und es existieren sehr wenige Beiträge anderer Arbeitsgruppen zu diesem Thema.

In dieser Arbeit haben wir zunächst den taDOM-Baum vorgestellt, ein erweitertes Datenmodell der DOM-Darstellung für XML-Dokumente. Der taDOM-Baum erlaubt die Synchronisation von Zugriffen auf einer sehr feingranularen Ebene. Basierend auf dem taDOM-Baum haben wir ein Sperrkonzept eingeführt, das mit den Mitteln von Knoten-, Navigations- und logischen Sperren nebenläufige Zugriffe mit den aus dem DOM-API bekannten Methoden synchronisiert. Knotensperren werden dabei auf den Knoten selbst angelegt, wobei die Navigationssperren die virtuellen Pfade zwischen den Knoten sperren. Die logischen Sperren dienen dazu, das Phantomproblem zu eliminieren. Das Sperrkonzept garantiert nicht nur die Isolationseigenschaft für Transaktionen aus dem ACID-Paradigma, sondern ermöglicht auch die Konfiguration verschiedener Parameter wie zum Beispiel die Einstellung der maximalen Sperrtiefe oder der Sperreskalation für gesteigerten Datendurchsatz.

In einem zweiten Schritt haben wir das vorgestellte taDOM-Sperrkonzept in einem Prototypen implementiert, um die allgemeine Leistungsfähigkeit und Auswirkungen der Konfigurationsparameter zu untersuchen. Dabei haben wir zwei verschiedene Systemarchitekturen realisiert und verglichen. Auf der einen Seite ist dies eine grobgranular synchronisierende Architektur, die alle Clientanfragen zunächst mit Hilfe einer Eingangswarteschlage serialisiert und dann in der Reihenfolge ihrer Ankunft nacheinander ausführt. Auf der anderen Seite gibt es die feingranularer synchronisierende Version, die es erlaubt, mehrere Anfragen verschiedener Transaktionen gleichzeitig auszuführen und die Zugriffe auf den zugrundeliegenden Datenstrukturen auf der Anweisungsebene synchronisiert. Beim Vergleich hat die zweite Architektur bezüglich Leistung deutlich schlechter abgeschnitten, weil durch die feingranulare Synchronisation der nebenläufigen Transaktionen sehr viele Kontextwechsel in kleinen Zeiträumen durchgeführt werden mussten, was den Datendurchsatz erheblich beeinflusst hat.

Wir haben in einem weiteren Experiment die Existenz des AttributeRoot-Knotentyps im taDOM-Baum in Frage gestellt. Wir haben dabei festgestellt, dass der Durchsatz sich dadurch nur unwesentlich verändert, die Anzahl der notwendigen Sperren beim Navigieren durch ein XML-Dokument erheblich vermindert werden kann. Dies hat vor allem große Auswirkungen auf den Speicherplatzverbrauch des Anfrageservers, was sich indirekt auch auf die Leistung auswirkt, nämlich vor allem dann, wenn die physikalischen Speichermöglichkeiten ausgeschöpft sind, und auf Auslagerungsstategien auf permanenten Speichermedien zurückgegriffen werden muss.

Darüber hinaus haben wir Untersuchungen durchgeführt inwiefern sich die Anzahl der parallelen Transaktionen auf den Zeitaufwand zum kompletten Durchlaufen eines Dokuments auswirkt. Wie zu erwarten war, wächst die Zeit polynomial schneller als die Anzahl der Transaktionen, weil sich auch die Anzahl der Sperrüberprüfungen pro Zugriff linear mit der Zahl der Transaktionen erhöht.

Zur Optimierung der Zugriffe auf die Sperrtabellen haben wir zwei weitere Strategien zum Anfordern der Sperren vorgestellt. Auf der einen Seite ist dies das Bottom-Up Verfahren, bei dem die entsprechenden Sperren von unten nach oben angelegt werden. Dies hat den Vorteil, dass man den Prozess schon dann abbrechen kann, wenn die angeforderte Sperre (oder eine höherwertigere) für die jeweilige Transaktion auf dem Knoten schon existiert. In einem solchen Fall darf man davon ausgehen, dass die verbleibenden Sperren bereits durch eine vorherige Operation derselben Transaktion angelegt worden sind. Auf der anderen Seite gibt es das Top-Down Verfahren, das die Sperren erst auf der Dokumentwurzel anlegt und dann den Weg nach unten bis zum entsprechenden Knoten verfolgt. Dies hat den Vorteil, dass eventuell auftretende Konflikte viel früher erkannt werden können, da diese mit erhöhter Wahrscheinlichkeit in den oberen horizontalen Schichten des DOM-Baums auftreten. Im Falle eines Konflikts ist das Top-Down Verfahren deutlich schneller, da weniger Schritte bis zum Entdecken des Konflikts ausgeführt werden müssen. Tritt jedoch ein solcher Konflikt nicht auf, was wohl in der großen Mehrheit der Zugriffe der Fall ist, so stellt sich doch das Bottom-Up als vorteilhafter heraus.

Der nächste Schritt wäre die Implementierung einer effizienten Speicherungskomponente für XML-Daten. Die aktuelle Realisierung setzt noch auf dem DOM-API von Java auf und implementiert einen Wrapper auf die vom XTC-System verwendeten Objekte. Dieses Verfahren kann durch eine native Speicherungskomponente erheblich beschleunigt werden.

Ob sich das vorgestellte Sperrkonzept auch für andere Zugriffsverfahren (wie zum Beispiel XQuery) eignet bleibt ebenfalls noch zu untersuchen. Man könnte sich ebenfalls vorstellen, dieses Sperrverfahren als Komponente in ein relationales beziehungsweise objektrelationales Datenbankmanagementsystem zu integrieren und so innerhalb einer Transaktion sowohl mit relationalen als auch mit XML-Daten zu arbeiten. Vor allem unter diesen Umständen wäre es sinnvoll das Sperrprotokoll so zu erweitern, dass es auch den Zugriff mit XML-Anfragesprachen wie XQuery unterstützt.

Kapitel 8 Literaturverzeichnis

[1] *IBM Alphaworks XML-Generator*
http://www.alphaworks.ibm.com/tech/xmlgenerator
September 1999

[2] *Transaction Synchronization for XML Data in Client-Server Web Applications*
Stefan Böttcher, Adelhard Türling
Fachbereich 17 (Mathematik / Informatik) / Universität Paderborn

[3] *The Apache XML Project. Xerxes2 Java Parser.*
http://xml.apache.org/xerxes2-j/index.html

[4] *Document Object Model (DOM) Level 2 Core Specification*
http://www.w3.org/TR/DOM-Level-2-Core/

[5] *A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database*
Daniela Florescu, Donald Koassmann
Rapport de Recherche No.3680, INRIA, Rocquencourt, France, 1999

[6] *Galileo Computer - Definition Tag*
http://www.galileocomputing.de/glossar/gp/anzeige-8350/FirstLetter-T

[7] *XMLTM: Efficient Transaction Management for XML Documents*
Torsten Grabs, Klements Böhm, Hans-Jörg Schek
Database Resarch Group / Inst. of Information Systems / ETH Zürich
November 2002

[8] *Design Patterns - Elements of Reusable Object-Oriented Software*
Erich Gamme, Richard Helm, Ralph Johnson, John Vlissides

[9] *Transaction Processing - Concepts and Techniques.*
Jim Gray, Andreas Reuter
Morgan Kaufmann Publishers Inc., 1993

[10] *taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API*
Michael P. Haustein, Theo Härder
Universität Kaiserslautern, Deutschland

[11] *Isolation in XML Bases*
S. Helmer, C.-C. Kanne, G. Moerkotte
Fakultät für Mathematik und Informatik / Universität Mannheim

[12] *Datenbanksysteme - Konzepte und Techniken der Implementierung*
Theo Härder, Erhard Rahm
Springer Verlag, 2001

[13] *Efficient Synchronization for Mobile XML Data*
Franky Lam, Nicole Lam, Raymond Wong
School of Computer Science and Engineering / University of New South Wales

[14] *Simple API for XML*
http://www.saxproject.org

[15] *NatiXync - Synchronisation für XML-Datenbanksysteme*
Robert Schiele - Universität Mannheim

[16] *Storing and Querying Ordered XML Using a Relational Database System.*
I. Tatarinov, S.D. Viglas, K. Beyer, J. Shanmugasundaram, E.Shekita, C. Zhang
Proc. ACM SIGMOD, Madison, Wisconsin, 2002

[17] *World Wide Web Consortium*
http://www.w3c.org

[18] *Extensible Markup Language*
http://www.w3c.org/XML

[19] *XRel - A Path-Based Approach to Starage and Retrieveal of XML Documents using Relationsal Databases*
M. Yoshikawa, T. Amagasa
ACM Transactions on Internet Technology, Aug. 2001

Anhang A Methodensignaturen für das XMLAccess-Interface

Im Folgenden seien die Methodensignaturen der Methoden aus dem XMLAccess-Interface aufgelistet. Jede Implementierung einer Datenhaltungsschnittstelle für das XTC-System muss alle folgenden Methoden implementieren.

- `public XTCDocument getDocument(String docName)`

  Öffnet das XML-Dokument mit dem angegebenen Dokumentnamen *docName* und liefert ein XTCDocument-Objekt zurück. Alle Zugriffsoperationen auf dieses Dokument tragen das zurückgelieferte XTCDocument als Parameter.

- `public Collection getDocuments()`

  Gibt eine *Collection* aller zum Zeitpunkt des Aufrufs von Transaktionen geöffneten Dokumente. Die Elemente der *Collection* sind die den geöffneten Dokumenten entsprechenden XTCDocument-Objekte.

- `public XTCElement getRoot(XTCDocument doc)`

  Liefert das Wurzelelement des angegebenen Dokuments *doc* in Form eines XTCElement-Objekts.

- `public XTCNode getNode(XTCDocument doc, int nodeID)`

  Liefert das XTCNode-Objekt in dem angegebenen Dokument *doc*, das mit der angegebenen Knoten-ID *nodeID* versehen ist. Das zurückgegebene Objekt kann vom Typ XTCElement, XTCAttribute oder XTCTextnode sein.

- `public XTCElement getParent(XTCDocument doc, int nodeID)`

  Liefert den Vorgängerknoten des durch die Knoten-ID *nodeID* identifizierten Knotens im Dokument *doc*. Diese Methode ist nicht nur für Elemente oder Textknoten gültig, sondern liefert auch für Attribute das Element, dem sie zugeordnet sind.

- `public Collection getChildNodes(XTCDocument doc, int nodeID)`

  Liefert eine Collection der Kindknoten des durch *nodeID* identifizierten Knotens im Dokument *doc*. Die Elemente der Collection sind ausschließlich XTCElement- oder XTCTextnode-Objekte (also allgemein vom Typ XTCNode).

- `public XTCNode getFirstChild(XTCDocument doc, int nodeID)`

  Liefert den ersten Kindknoten des durch *nodeID* identifizierten Knotens im Dokument *doc*. Dieser kann nur vom Typ XTCElement oder XTCTextnode sein. Existiert er nicht, so ist der Rückgabewert *null*.

- `public XTCNode getLastChild(XTCDocument doc, int nodeID)`

  Liefert den letzten Kindknoten des durch *nodeID* identifizierten Knotens im Dokument *doc*. Dieser kann nur vom Typ XTCElement oder XTCTextnode sein. Existiert er nicht, so ist der Rückgabewert *null*.

- `public XTCNode getPrevSibling(XTCDocument doc, int nodeID)`

  Liefert den vorherigen (linken) Nachbarn des durch *nodeID* identifizierten Knotens im Dokument *doc*. Dieser kann nur vom Typ XTCElement oder XTCTextnode sein. Existiert er nicht, so ist der Rückgabewert *null*.

- `public XTCNode getNextSibling(XTCDocument doc, int nodeID)`

  Liefert den nächsten (rechten) Nachbarn des durch *nodeID* identifizierten Knotens im Dokument *doc*. Dieser kann nur vom Typ XTCElement oder XTCTextnode sein. Existiert er nicht, so ist der Rückgabewert *null*.

- `public String getValue(XTCDocument doc, int nodeID)`

  Liefert den Wert des durch *nodeID* identifizierten Knotens im Dokument *doc*. Der Aufruf ist nur gültig, wenn *nodeID* auf einen XTCAttribute- oder XTCTextnode-Knoten verweist. Andernfalls liefert die Methode *null*.

- `public Collection getAttributes(XTCDocument doc, int nodeID)`

  Liefert eine *Collection* der Attribute des durch *nodeID* identifizierten Knotens im Dokument *doc* in Form von XTCAttribute-Objekten.

- `public boolean hasAttribute(XTCDocument doc, int nodeID, String tagname)`

  Liefert *true* falls der durch *nodeID* identifizierte Knoten im Dokument *doc* ein Attribut mit dem Namen *tagname* besitzt, andernfalls *false*.

- `public String getAttribute(XTCDocument doc, int nodeID, String tagname)`

  Liefert den Wert des Attributs des durch *nodeID* identifizierten Knotens im Dokument *doc* mit dem Namen *tagname*. Falls ein solches Attribut nicht existiert ist der Rückgabewert *null.*

- `public XTCAttribute getAttributeNode(XTCDocument doc, int nodeID, String tagname)`

  Liefert das Attribut des durch *nodeID* identifizierten Knotens im Dokument *doc* mit dem Namen *tagname* in Form eines XTCAttribute-Objekts. Falls ein solches Attribut nicht existiert, ist der Rückgabewert *null.*

- `public Collection getElementsByTagname(XTCDocument doc, int nodeID, String tagname)`

  Liefert eine *Collection* von allen XTCElement-Objekten mit dem Tagnamen *tagname,* die sich im Unterbaum des durch *nodeID* identifizierten Knotens im Dokument *doc* befinden. Trifft die Anfrage für keinen Knoten zu, so hat die zurückgelieferte *Collection* keine Elemente.

- `public XTCElement getElementById(XTCDocument doc, String elementID)`

  Liefert das XTCElement-Objekt dessen in der DTD spezifiziertes ID-Attribut den Wert *elementID* hat. Die XML-Spezifikation schreibt vor, dass dieser Wert im Dokument nur ein einziges Mal vorkommen darf. Deswegen ist die Anfrage auch eindeutig und das Anfrageergebnis ist ein Elementknoten.

- `public XTCNode insertBefore(XTCDocument doc, int nodeID, XTCNode node, int refID)`

  Fügt den Knoten *node* als Kindknoten des durch *nodeID* identifizierten Knotens im Dokument *doc* vor dem Knoten mit der ID *refID* ein. Die Methode liefert den eingefügten Knoten zurück. Ist *refID* gleich 0, so wird der Knoten an die erste Position der Kindknoten eingefügt.

- `public XTCNode replaceChild(XTCDocument doc, int nodeID, XTCNode newChild, int oldChildID)`

  Ersetzt den durch *oldChildID* identifizierten Knoten im Dokument *doc* durch *newChild*. Dabei ist *nodeID* die ID des Vaterknotens des zu ersetzenden Kindknotens. Die Methode liefert den eingefügten Knoten zurück.

- `public void removeChild(XTCDocument doc, int nodeID, int oldChildID)`

  Entfernt den Kindknoten mit der ID *oldChildID* des durch *nodeID* identifizierten Knotens im Dokument *doc*.

- `public void setValue(XTCDocument doc, int nodeID, String nodeValue)`

  Setzt den Wert des durch *nodeID* identifizierten Knotens im Dokument *doc* auf *nodeValue*. Der Aufruf ist nur für Attribut- oder Textknoten zulässig, ansonsten hat er keine Auswirkung.

- `public void removeAttribute(XTCDocument doc, int nodeID, String attrName)`

  Entfernt das Attribut mit dem Namen *attrName* für den durch *nodeID* identifizierten Knoten. Existiert ein solches Attribut nicht, bleibt der Aufruf ohne Auswirkung.

- `public XTCAttribute addAttribute(XTCDocument doc, int nodeID, String attrName, String attrValue)`

  Fügt dem durch *nodeID* identifizierten Element im Dokument *doc* ein Attribut mit dem Namen *attrName* und dem Wert *attrValue* hinzu. Ist der referenzierte Knoten kein Element, so hat der Methodenaufruf keine Auswirkung. Das hinzugefügte Attribut wird als XTCAttribute-Objekt zurückgeliefert.

- `public XTCElement createElement(XTCDocument doc, String tagName)`

  Erstellt ein neues XTCElement-Objekt und fügt es dem Dokument *doc* hinzu. Der neue Knoten wird allerdings noch nicht in den DOM-Baum eingefügt. Dazu ist ein weiterer Aufruf von *insertBefore()* oder *replaceChild()* notwendig. Das erzeugte XTCElement-Objekt wird zurückgeliefert.

- `public XTCTextnode createTextnode(XTCDocument doc, String data)`

  Erstellt ein neues XTCTextnode-Objekt und fügt es dem Dokument *doc* hinzu. Der neue Knoten wird allerdings noch nicht in den DOM-Baum eingefügt. Dazu ist ein weiterer Aufruf von *insertBefore()* oder *replaceChild()* notwendig. Das erzeugte XTCTextnode-Objekt wird zurückgeliefert.

Anhang B Konfigurationsdateien

Beispiel 1 und Beispiel 2 zeigen jeweils eine Bespiel-Konfigurationsdatei für den XTC-Server und den XTC-Client.

Beispiel 1 Konfigurationsdatei für den XTC-Server

```
<xtc>

  <server>

    <listen>4445</listen>
    <documentpath>/home/georges/xml/</documentpath>
    <useattributeroot>true</useattributeroot>
    <queuemanagertype>lazy</queuemanagertype>
    <disablelockmechanism>false</disablelockmechanism>
    <escalationmanger>recursive</escalationmanger>
    <lockingorder>bottomup</lockingorder>

    <rmi>
      <host>localhost</host>
      <handle>XTCEngine</handle>
      <timeout>10</timeout>
    </rmi>

  </server>

</xtc>
```

❑

Beispiel 2 Konfigurationsdatei für den XTC-Client

```
<xtc>

  <client>

    <rmi>
      <host>localhost</host>
      <handle>XTCEngine</handle>
      <timeout>10</timeout>
    </rmi>

  </client>

</xtc>
```

❑

Anhang C Testergebnisse

Im folgenden Anhang sind die kompletten Messwerte zu den Grafiken aus Kapitel 6 aufgeführt.

C.1 Aufwand der Sperrverwaltung

Im folgenden Abschnitt sind die Messwerte zur Bestimmung des relativen Aufwands der Sperrverwaltung aufgelistet. Das Experiment findet unter Verwendung der AttributeRoot-Knotens statt und die maximale Sperrtiefe ist unbegrenzt.

Tabelle 1 Zeitverlauf für das Durchlaufen aller Dokumente bei ein- und ausgeschalteter Sperrverwaltung

Dokument	Sperren an	Sperren aus
1000.xml	0,859	0,516
5000.xml	3,188	1,687
10000.xml	5,844	3,047
25000.xml	13,485	7,172
50000.xml	27,265	13,635
75000.xml	39,641	20,360
100000.xml	52,562	26,718
150000.xml	81,485	39,547

C.2 Einfluss der maximalen Sperrtiefe

Im folgenden Abschnitt sind die Messwerte und die entsprechenden Grafiken, für alle Dokumente für die die Testreihe durchgeführt worden ist, aufgelistet (siehe Abschnitt 6.2.3)

Tabelle 2 Messergebnisse für das 1000.xml-dokument mit 10 parallelen Transaktionen und unter Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	11100	0,859
6	3870	0,531
5	1070	0,313
4	200	0,234
3	10	0,188

Tabelle 3 Messergebnisse für das 5000.xml-dokument mit 10 parallelen Transaktionen und unter Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	53720	3,188
9	20100	1,532
8	7490	0,984
7	3010	0,672
6	1280	0,515
5	500	0,469

Tabelle 4 Messergebnisse für das 10000.xml-dokument mit 10 parallelen Transaktionen und unter Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	107880	5,844
11	38560	2,813
10	15400	1,862
9	6310	1,266
8	3010	1,171
7	1290	0,984
5	140	0,703

Tabelle 5 Messergebnisse für das 25000.xml-dokument mit 10 parallelen Transaktionen und unter Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	264380	13,485
11	95610	6,484
10	32160	3,672
9	12050	2,797
8	3990	2,266
7	1490	2,125
5	180	1,953

Tabelle 6 Messergebnisse für das 50000.xml-dokument mit 10 parallelen Transaktionen und unter Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	530130	27,265
16	194580	12,641
15	71400	7,484
14	26180	5,281
13	9480	4,547
12	3800	4,282
10	700	3,781

Tabelle 7 Messergebnisse für das 75000.xml-dokument mit 10 parallelen Transaktionen und unter Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	792140	39,641
21	197560	15,125
20	49340	8,313
19	12720	6,484
18	3520	6,141
17	890	5,953
15	350	5,828

Tabelle 8 Messergebnisse für das 100000.xml-dokument mit 10 parallelen Transaktionen und unter Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	1037200	52,562
21	175380	16,219
20	29200	9,265
19	5080	7,938
18	990	7,578
17	430	7,513
15	310	7,502

Tabelle 9 Messergebnisse für das 150000.xml-dokument mit 10 parallelen Transaktionen und unter Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	1560200	81,485
26	298310	25,422
25	57140	14,391
24	11190	12,172
23	2430	11,578
22	690	11,062
20	510	11,375

C.3 Nichlineares Verhältnis zwischen Zeit und Anzahl der Transaktionen

Tabelle 10 Messergebnisse für das 1000.xml-dokument unter Verwendung des AttributeRoot-Knotens mit variierender Anzahl paralleler Transaktionen

# Transaktionen	Sperren	Zeit (sek.)
5	5550	0,547
10	11100	0,844
15	16650	1,235
20	22200	1,751
25	27750	2,218
30	33300	2,813
35	38850	3,344

Tabelle 11 Messergebnisse für das 5000.xml-dokument unter Verwendung des AttributeRoot-Knotens mit variierender Anzahl paralleler Transaktionen

# Transaktionen	Sperren	Zeit (sek.)
5	26860	1,594
10	53720	3,172
15	80580	5,110
20	107440	7,281
25	134300	9,953
30	161160	12,782
35	188020	16,016

Tabelle 12 Messergebnisse für das 10000.xml-dokument unter Verwendung des AttributeRoot-Knotens mit variierender Anzahl paralleler Transaktionen

# Transaktionen	Sperren	Zeit (sek.)
5	53940	2,906
10	107880	6,016
15	161820	9,954
20	215760	14,937
25	269700	20,110
30	323640	26,734
35	377580	32,391

Tabelle 13 Messergebnisse für das 25000.xml-dokument unter Verwendung des AttributeRoot-Knotens mit variierender Anzahl paralleler Transaktionen

# Transaktionen	Sperren	Zeit (sek.)
5	132190	6,454
10	264380	14,265
15	396570	24,156
20	528760	37,344
25	660950	50,109
30	793140	65,969
35	925330	82,672

Tabelle 14 Messergebnisse für das 50000.xml-dokument unter Verwendung des AttributeRoot-Knotens mit variierender Anzahl paralleler Transaktionen

# Transaktionen	Sperren	Zeit (sek.)
5	265065	12,078
10	530130	27,265
15	795195	46,484
20	1060260	70,859
25	1325325	98,078
30	1590390	125,718
35	1855455	171,437

Tabelle 15 Messergebnisse für das 75000.xml-dokument unter Verwendung des AttributeRoot-Knotens mit variierender Anzahl paralleler Transaktionen. Wegen zu starker Speicherauslastung konnte die Testreihe nur bis maximal 25 parallele Transaktionen durchgeführt werden.

# Transaktionen	Sperren	Zeit (sek.)
5	396070	18,265
10	792140	42,078
15	1188210	72,719
20	1584280	115,109
25	1980350	178,844

Tabelle 16 Messergebnisse für das 100000.xml-dokument unter Verwendung des AttributeRoot-Knotens mit variierender Anzahl paralleler Transaktionen. Wegen zu starker Speicherauslastung konnte die Testreihe nur bis maximal 20 parallele Transaktionen durchgeführt werden.

# Transaktionen	Sperren	Zeit (sek.)
5	518600	24,484
10	1037200	54,453
15	1555800	101,875
20	2074400	172,418

Aufgrund zu starkem Speicherverbrauch bei den Versuchen mit dem 150000.xml-dokument haben wir an dieser Stelle auf die Ergebnisse verzichtet. Diese sind durch Auslagerungsoperationen des Betriebssystems zu stark verfälscht.

C.4 Verwendung von AttributeRoot

Im folgenden Abschnitt sind die Messwerte und die entsprechenden Grafiken, für alle Dokumente für die die Testreihe durchgeführt worden ist aufgelistet (siehe Abschnitt 6.2.5)

Tabelle 17 Messergebnisse für das 1000.xml-dokument mit 10 parallelen Transaktionen und ohne Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	15000	0,813
6	5190	0,501
5	1400	0,375
4	180	0,188
3	20	0,172

Tabelle 18 Messergebnisse für das 5000.xml-dokument mit 10 parallelen Transaktionen und ohne Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	79600	2,844
9	19680	1,504
8	10900	0,862
7	4370	0,641
6	1830	0,578
5	670	0,453

Tabelle 19 Messergebnisse für das 10000.xml-dokument mit 10 parallelen Transaktionen und ohne Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	160210	5,793
11	56830	2,851
10	22780	1,825
9	9170	1,235
8	4310	1,047
7	1770	1,009
5	140	0,688

Tabelle 20 Messergebnisse für das 25000.xml-dokument mit 10 parallelen Transaktionen und ohne Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	409550	13,516
11	148120	6,531
10	49020	3,641
9	18460	2,797
8	6080	2,219
7	2210	2,125
5	220	1,954

Tabelle 21 Messergebnisse für das 50000.xml-dokument mit 10 parallelen Transaktionen und ohne Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	820300	26,547
16	301210	12,656
15	110000	7,328
14	39930	5,375
13	14260	4,531
12	5580	4,016
10	920	3,781

Tabelle 22 Messergebnisse für das 75000.xml-dokument mit 10 parallelen Transaktionen und ohne Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	1227130	38,969
21	305890	14,953
20	75730	8,344
19	19050	6,313
18	5030	6,094
17	960	5,890
15	220	5,828

Tabelle 23 Messergebnisse für das 100000.xml-dokument mit 10 parallelen Transaktionen und ohne Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	1656930	52,392
21	179840	15,828
20	45960	9,266
19	7330	7,813
18	1200	7,421
17	330	7,500
15	180	7,504

Tabelle 24 Messergebnisse für das 150000.xml-dokument mit 10 parallelen Transaktionen und ohne Verwendung des AttributeRoot-Knotens

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	2492200	89,031
26	476760	25,485
25	90700	13,875
24	17360	11,969
23	3370	11,547
22	580	11,501
20	340	11,266

C.5 Einfluss des QueueManagers

Im folgenden Abschnitt sind die Messwerte und die entsprechenden Grafiken, für alle Dokumente für die die Testreihe durchgeführt worden ist aufgelistet (siehe Abschnitt 6.2.6)

Tabelle 25 Messergebnisse für das 1000.xml-dokument mit 10 parallelen Transaktionen unter Verwendung des AttributeRoot-Knotens mit dem BusyQueueManager

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	11100	0,687
6	3870	0,437
5	1070	0,297
4	200	0,234
3	10	0,203

Tabelle 26 Messergebnisse für das 5000.xml-dokument mit 10 parallelen Transaktionen unter Verwendung des AttributeRoot-Knotens mit dem BusyQueueManager

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	53720	2,204
9	20100	1,141
8	7490	0,875
7	3010	0,610
6	1280	0,500
5	500	0,422

Tabelle 27 Messergebnisse für das 10000.xml-dokument mit 10 parallelen Transaktionen unter Verwendung des AttributeRoot-Knotens mit dem BusyQueueManager

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	107880	3,906
11	38560	2,078
10	15400	1,532
9	6310	1,156
8	3010	1,078
7	1290	0,985
5	140	0,735

Tabelle 28 Messergebnisse für das 25000.xml-dokument mit 10 parallelen Transaktionen unter Verwendung des AttributeRoot-Knotens mit dem BusyQueueManager

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	264380	9,016
11	95610	4,671
10	32160	2,907
9	12050	2,532
8	3990	2,172
7	1490	2,125
5	180	1,922

Tabelle 29 Messergebnisse für das 50000.xml-dokument mit 10 parallelen Transaktionen unter Verwendung des AttributeRoot-Knotens mit dem BusyQueueManager

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	530130	17,875
16	194580	9,172
15	71400	5,750
14	26180	4,687
13	9480	4,328
12	3800	4,218
10	700	3,812

Tabelle 30 Messergebnisse für das 75000.xml-dokument mit 10 parallelen Transaktionen unter Verwendung des AttributeRoot-Knotens mit dem BusyQueueManager

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	792140	27,062
21	197560	11,125
20	49340	7,156
19	12720	6,391
18	3520	6,062
17	890	5,875
15	350	5,813

Tabelle 31 Messergebnisse für das 100000.xml-dokument mit 10 parallelen Transaktionen unter Verwendung des AttributeRoot-Knotens mit dem BusyQueueManager

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	1037200	34,468
21	175380	12,406
20	29200	8,625
19	5080	7,797
18	990	7,578
17	430	7,500
15	310	7,502

Tabelle 32 Messergebnisse für das 150000.xml-dokument mit 10 parallelen Transaktionen unter Verwendung des AttributeRoot-Knotens mit dem BusyQueueManager

max. Sperrtiefe	Sperren	Zeit (sek.)
unendlich	1560200	57,563
26	298310	19,125
25	57140	13,281
24	11190	11,797
23	2430	11,515
22	690	11,422
20	510	11,360

C.6 Veränderte Systemarchitektur

Im folgenden Abschnitt sind die Messergebnisse für das normalen und das optimierte Bottom-Up Verfahren, sowie für das Top-Down Verfahren aufgeführt.

Tabelle 33 Zeitverlauf für das Durchlaufen aller Dokumente für die alte und die neue Systemarchitektur

Dokument	alte Architektur	neue Architektur
1000.xml	0,859	1,140
5000.xml	3,188	4,172
10000.xml	5,844	6,781
25000.xml	13,485	15,422
50000.xml	27,265	49,406
75000.xml	39,641	53,140
100000.xml	52,562	68,429
150000.xml	81,485	127,234

Tabelle 34 Zeitverlauf für das Durchlaufen aller Dokumente für das optimierte und nicht-optimierte Bottom-Up Verfahren

Dokument	nicht-optimiert	optimiert
1000.xml	1,016	0,906
5000.xml	3,407	2,891
10000.xml	6,781	5,531
25000.xml	15,422	13,297
50000.xml	32,469	25,657
75000.xml	53,140	39,296
100000.xml	68,429	54,078
150000.xml	127,234	85,766