XML Seminar SS 2003
Universität des Saarlandes

Thema:
Updating XML Data
Transaction Support for XML

Stefan Bender, Christian Fuchs, Michael Schmidt

Betreuung:
Dr.-Ing. Ralf Schenkel

03. Juni 2003

Inhaltsverzeichnis

1 Updating XML

1.1 Einführung

Innerhalb der letzten Jahre hat der Austausch von Daten über das Internet immer mehr an Bedeutung gewonnen. Diese Entwicklung wurde vor allem durch das schnelle Heranwachsen des e-Commerce und des weltweiten Netzwerks hervorgerufen. Eine immer größer werdende Datenmenge muss über das Internet übertragen werden. Am Anfang stand hierzu nur die HTML-Sprache zur Verfügung. Es zeigte sich jedoch schnell, dass HTML zur Übertragung großer Datenmengen weniger gut geeignet war. Aus diesem Grund veröffentlichte das World-Wide-Web-Konsortium 1998 den Standard für eine neue Sprache Namens XML. Bereits in sehr kurzer Zeit hat sich diese Sprache zum de-facto-Standard für den Austausch von Daten entwickelt.

Mit Hilfe von XML ist es möglich, auf einfache Art und Weise Daten in strukturierter Form zwischen heterogenen Computersystemen auszutauschen. Ein typisches Szenario zeigt Abbildung 1. Im e-Commerce tauschen Firmen häufig ihre Produktkataloge im XML-Format aus. Der Produktkatalog enthält dabei alle Informationen zu den Produkten, die vom Supplier angeboten werden. Der Supplier sendet zunächst diesen Katalog an den Customer. Der Customer verwendet diesen Katalog und sendet seine Bestellung zurück an den Supplier. Für dieses Szenario ist die Verwendung von XML bestens geeignet.

Schwierig wird es jedoch, wenn sich die Daten häufig ändern. Soll zum Beispiel ein neues Produkt zum Katalog hinzugefügt werden, so muss der gesamte Katalog neu übertragen werden. Dies bedeutet natürlich, dass viele Daten unnötigerweise neu übertragen werden. Im Anschluss an die Übertragung der Daten müssen diese noch in den Datenbestand des Customers eingefügt werden. Hier gibt es prinzipiell zwei Möglichkeiten. Die erste Lösungmöglichkeit entfernt den alten Katalog und fügt anschließend den neuen Katalog ein. Dies erfordert eine Vielzahl an Änderungen am Datenbestand, welcher für diese Zeit blockiert ist. Die zweite Möglichkeit besteht darin, die beiden Kataloge zu vergleichen und anschließend, den alten Katalog durch Updates in den neuen Katalog zu überführen. Dies er-

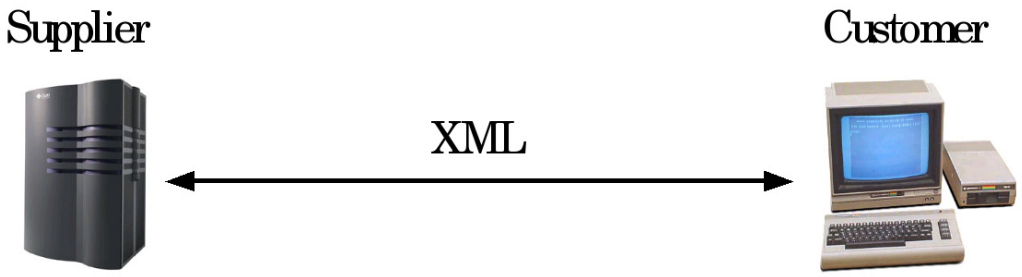

Abbildung 1: Beispielszenario

fordert deutlich weniger Änderungen am Datenbestand, benötigt jedoch einiges an Rechenleistung, um die Unterschiede zwischen den Katalogversionen zu erkennen. Diese Arten der Datenmanipulation mit Hilfe von XML sind nicht sehr komfortabel. Zum einen müssen große Datenmengen auch für kleine Änderungen übertragen werden zum anderen bedeutet das Updaten des alten Kataloges einen beträchtlichen Aufwand.

Um diesem Problem zu begegnen, versucht man, XML um eine Update Funktionalität zu erweitern. Durch diese Erweiterung soll es möglich sein, nur die Änderungen eines Dokumentes zu übertragen. Die übertragenen Änderungen können dann dazu verwendet werde, den alten Katalog in den neuen zu überführen.

1.2 XQuery extensions

1.2.1 XQuery

Ausgangspunkt der Überlegungen ist, Update-Funktionalitäten auch im Umfeld von XML zur Verfügung zu stellen. Hierzu stellt sich zunächst die Frage, ob es bereits Lösungen für dieses Problem gibt. Bei einer ersten Betrachtung fällt auf, dass das w3c bereits eine Abfragesprache (query language) veröffentlicht hat. Diese wird als XQuery bezeichnet. Mit XQuery können Daten abgefragt werden, die im XML-Format vorhanden bzw. gespeichert sind. Eine solche XQuery-Abfrage könnte folgendermaßen aussehen.

```
FOR = $cust IN document("cutomer.xml")/customer,
    $name IN $cust,
    $city IN $cust/address,
WHERE $name = "Michael Schmidt"
RETURN $city
```

Diese Abfrage dient dazu, die Stadt des Kunden mit dem Namen „Michael Schmidt“ auszugeben. Ohne ein tiefes Verständnis für XQuery zu haben, kann man durchaus bereits hier wesentliche Vorteile dieser Abfragesprache erkennen. Zum einen ist die Syntax recht einfach zu verstehen. Dies wird durch eine Anlehnung an SQL erreicht. Ein Benutzer, der mit SQL vertraut ist, sollte keine Schwierigkeiten haben, die obige Abfrage zu verstehen.

Der zweite wesentliche Vorteil von XQuery liegt darin, dass die Abfragen von der physischen Speicherung der Daten unabhängig sind. Das bedeutet, die obige Abfrage kann unabhängig davon verwendet werden, ob die XML-Daten in einem File, einer relationalen Datenbank oder in einer native Datenbank gespeichert sind. XQuery setzt hier lediglich das Wissen über den hierarchischen Aufbau der XML-Daten vorraus.

Dennoch besitzt XQuery auch einige Nachteile. Der wichtigste Nachteil ist wohl, dass es keine Möglichkeiten gibt, XML-Daten zu manipulieren. Mit XQuery können nur Abfragen an XML-Daten gestellt werden. Dies steht im Gegen-

satz zum üblichen Verständnis einer Abfragesprachen, wie z.B. SQL. Bei dieser ist es durchaus möglich, Daten durch Insert-, Delete- oder Update-Abfragen zu verändern. Um XQuery ebenfalls zu einer „vollständigen“ Abfragesprache zu erweitern, bedarf es also einer Erweiterung(extention).

1.2.2 Erweiterung

Im obigen Abschnitt wurde das Problem des beschränkten Funktionsumfanges von XQuery erläutert. Es wird versucht, diesem Problem zu begegnen, indem Erweiterungen für XQuery einfügt werden. Mit diesen Erweiterungen sollte es dann möglich sein, XML-Daten zu verändern. Weiterhin sollten sie abwärtskompatibel sein, d.h. eine normale XQuery, die auf Basis des Standards (ohne extentions) erstellt wurde, sollte weiterhin ausführbar sein. Mit anderen Worten, die Erweiterungen sollen den Standard nicht beeinflussen. Des weiteren sollten sie ebenfalls von der physischen Speicherung der Daten unabhängig sein. Es würde keinen Sinn ergeben, als Erweiterung z.B. einen SQL-Befehl an die XQuery-Abfrage anzuhängen, da dies nur in Zusammenhang mir Datenbanken funktionieren würde.

Nachdem nun die generellen Anforderungen an eine Erweiterung von XQuery vorgestellt wurden, stellt sich die Frage, welche Funktionen die Erweiterung enthalten sollte. Der Artikel [3] führt dabei folgende Operationen ein.

- Delete
- Rename / Replace
- Insert / InsertBefore / InsertAfter
- Sub-Update

Zur Funktionsweise von Delete, Rename, Replace und Insert bedarf es keiner weiteren Erklärung, da die Namen selbsterklärend sind. Die Funktionen InsertBefore und InsertAfter sind dann von Bedeutung, wenn geordnete XML-Dokumente verändert werden sollen. Ein XML-Dokument wird als geordnet bezeichnet, wenn die Reihenfolge zweier Elemente von Bedeutung ist. Als Beispiel sei hier die XML-Darstellung eines Buchs genannt. Bei dieser ist es durchaus relevant, ob das Kapitel 1 vor oder nach dem Kapitel 2 erscheint. Um in ein solches Dokument Daten einzufügen, werden die InsertBefore- und InsertAfter-Funktionen verwendet. Die letzte Funktion die von den Erweiterungen bereitgestellt werden soll, ist das Sub-Update. Darunter versteht man eine geschachtelte Update-Anweisung. Sollen z.B. alle Kunden einer englischen Firma in den Kundenstamm einer deutschen Firma aufgenommen werden, so kann man dies mit der Insert-Operation durchführen. Um aber gleichzeitig die englische Länderbezeichnung (Germany) durch die deutsche Bezeichnung (Deutschland) zu ersetzen, verwendet man innerhalb der Insert Operation eine Sub-Update-Operation.

1.2.3 Syntax

Wie sieht nun eine solche XQuery-extention aus. Der Artikel [3] schlägt hier folgenden generelle Syntax vor.

```
FOR $binding IN XPath-expression,...

LET $binding := XPath-expression,...

WHERE predicate,...

UPDATE $binding { subOp { ,subOp }* }
```

Zu Beginn eine Update-Anweisung steht zunächst ein FOR-LET-WHERE Statement, wie es bereits aus XQuery bekannt ist. Bei einer Standard XQuery-Abfrage stünde anschließend ein RETURN-Statement. Dieses RETURN-Statement wird hier durch ein UPDATE-Statement ersetzt. Dieses Statement stellt die eigentliche Erweiterung dar. Dem Schlüsselwort UPDATE schließt sich eine Variable an, welche in der obigen Definition $binding bezeichnet wird. Die Variable entspricht dabei dem Teil des XML-Dokuments, der verändert werden soll. Im Rumpf der Erweiterung stehen dann die einzelnen Update-Operationen. Die Syntax der einzelnen Operationen sieht dabei wie folgt aus:

```
DELETE $child,
RENAME $child TO name,
INSERT $content [BEFORE|AFTER] $child,
REPLACE $child WITH $content,
FOR $binding2 IN XPath-expression,...
        WHERE predicate,...
        UPDATE...
```

In der obigen Syntax steht die Variable $child jeweils für das Element, dass verändert werden soll. Ein Element kann in diesem Zusammenhang ein Attribut, eine IDREF oder ein PCDATA Element sein. Der neue Inhalt einer Variablen ist in $content enthalten. Bei der Definition des Funktionsumfanges für eine Erweiterung wurde auch die Möglichkeit eines Sub-Updates gefordert. Dieses Sub-Update wird durch das obige FOR-WHERE-UPDATE-Statement realisiert.

1.2.4 Beispiel

Um ein besseres Verständnis der Funktionsweise der Erweiterungen zu bekommen, soll die Erweiterung an einem Beispiel dargestellt werden. Hierzu wird zunächst das folgende XML-Dokument vorgestellt.

```
<Customers>
    <Customer>
        <Name>Schmidt</Name>
        <Address>
            <City>Trier</City>
            <State>RLP</State>
        </Address>
        <Order>
            <Date>12.05.2003</Date>
            <OrderLine> [...]
        </Order>
    </Customer>
    <Customer>
        <Name>Müller</Name>
        <Address>
            <City>Saarbrücken</City>
            <State>Saarland</State>
        </Address>
        <Order>
            <Date>14.05.2003</Date>
            <OrderLine> [...]
        </Order>
    </Customer>
</Customers>
```

Das XML-Dokument zeigt einen Ausschnitt aus einer Kundendatenbank. Wie man erkennen kann, besitzt jeder Kunde ein Element `Name`, ein Element `Address` und ein Element `Order`. Das Element `Name` enthält lediglich PCDATA, es ist somit ein einfaches Datenelement. Das Element `Address` besteht aus einem Subbaum, der die beiden Elemente `City` und `State` enthält. Diese beiden Elemente enthalten wiederum PCDATA-Einträge. Das dritte Element ist wiederum ein Subbaum, der die Bestellungen eines Kunden enthält und mit `Order` bezeichnet wird.

Nachdem das Beispieldokument eingeführt wurde, soll nun an zwei Beispielen die Funktionsweise der Updates erläutert werden. Beim ersten Beispiel sollen alle `State`-Elemente aus dem XML-Dokument entfernt werden. Die Syntax der Update-Anweisung sieht dabei wie folgt aus.

```
FOR
    $doc   IN document ("Customer.xml")
    $cust  IN $doc/customer,
    $addr  IN $cust/address,
    $state IN $addr/state
```

```
UPDATE $addr {

    DELETE $state

}
```

Mit Hilfe der `XPath-Expression` im `FOR`-Teil der Anweisung werden zunächst alle Address-Elemente in der Variablen `$addr` selektiert. Auf dieser Variablen wird dann die Update-Anweisung `DELETE $state` ausgeführt. Im obigen Fall entspricht das dem Löschen der `State`-Elemente. In einem zweiten Beispiel sollen nun alle City-Elemente mit dem Inhalt Karl-Max-Stadt in Chemnitz umbenannt werden. Hierzu könnte man folgende Anweisung verwenden

```
FOR
    $doc  IN document ("Customer.xml")
    $cust IN $doc/customer,
    $addr IN $cust/address,
    $city IN $addr/city
WHERE $city = "Karl-Max-Stadt"

UPDATE $addr {

    REPLACE $city
    WITH
        <appellation>Chemnitz</appellation>
}
```

Zunächst werden, analog zum ersten Beispiel, alle Address-Elemente ausgewählt. Anschließend werden dann mithilfe der WHERE-Klausel die Elemente selektiert, deren City-Element den Wert `Karl-Marx-Stadt` enthält. Dannach wird bei diesen Elementen der Inhalt des City-Elements durch Chemnitz ersetzt. Das `appellation`-Tag enthält dabei den neuen Inhalt.

1.2.5 Probleme mit dieser Erweiterung

Wie man durch die beiden Beispiele erkennen konnte, ist es sehr einfach mit Hilfe der erweiterten XQuery Veränderungen am Datenbestand durchzuführen. Besonders angenehm fällt auf, dass kein Wissen über die Art der Speicherung eines Dokuments benötigt wird. Die Auswahl eines bestimmten Elements mit Hilfe der FOR-LET-WHERE-Statements ist dabei sehr komfortabel. Auch eine recht komplizierte Operation wie das Löschen eines Teilbaumes, kann durch ein einfaches Update-Statement realisiert werden. Leider bietet diese Erweiterung nicht nur Vorteile. Die Probleme entstehen, wenn die XQuery extentions in die Praxis umgesetzt werden sollen. Wie schon mehrfach erwähnt, sind die Extentions

von der physischen Speicherung unabhängig. Dennoch müssen diese Anweisungen irgendwie auf die physische Ebenen der Datenspeicherung konvertiert werden. An dieser Stelle entstehen dann die Probleme. Um dies besser zu verstehen, muss man zunächst betrachten, wie Daten in XML gespeichert werden. Prinzipiell gibt es drei Möglichkeiten:

1. Einfache Datei
2. Relationale Datenbank
3. Native Datenbank

Am häufigsten wird wohl die relationale Datenbank verwendet. Aus diesem Grund soll nur noch diese Möglichkeit betrachtet werden. Bei der relationalen Datenbank müssen die XML-Daten in Relationen (Tabellen) überführt werden. Der hierzu verwendete Ansatz wird als Inline-Sharing bezeichnet. Bei diesem Ansatz werden alle Kind-Elemente, die in einer 1:1-Beziehung zum Vater-Element stehen, in der Vater-Relation zusammengefasst. Besteht hingegen eine 1:n-Bezeichung zwischen Vater-Element und Kind, so muss das Kind in einer eigenen Relation gespeichert werden. Für das Beispieldokument müssten 4 Relationen (Customers, Customer, Order, OrderLine) angelegt werden. Die Relation `Customer` enthält dabei die Felder `ID, ParentID, Name, City, State`. Wie man erkennen kann, wurden die Elemente City und State mit in die Customer-Relation übernommen. Dies ist möglich, da zwischen `Customer` und `Address`, sowie zwischen `Address` und `City` bzw. `State`, eine 1:1 Beziehung besteht. Die Felder `ID` und `ParentID` werden benötigt, um die Baumstruktur eines XML-Dokuments in der Datenbank abzubilden.

Ein Problem, das im Zusammenhang mit Updates von XML-Dokumenten auftaucht, sind die Referenzen innerhalb eines Dokuments. Dabei muss man zwei Fälle unterscheiden. Beim Löschen von Elementen kann es passieren, das ein Element gelöscht wird, welches von einem zweiten Element referenziert wird. In diesem Fall ist es nach Angabe des Artikels [3] durchaus zulässig, dass Referenzen frei „herumhängen“. Der zweite Fall tritt auf, wenn neue Daten in ein bestehendes Dokument eingefügt werden sollen. Bei dieser Einfügung muß es unbedingt vermieden werden, dass IDREF-Werte mehrfach vorkommen. Die Definition von XML sieht hier vor, dass IDREF's innerhalb eines Dokuments eindeutig sind. Fügt man nun vorhandene Daten aus einem zweiten XML-Dokument ein, kann dies dazu führen, dass die IDREF's nicht mehr eindeutig sind. Bevor also neue Daten eingefügt werden, müssen daher zunächst die IDREF's kontrolliert und eventuell angepasst werden.

Das zweite, beim Updaten entstehende, Problem ist, dass eine Änderung eines Elements unter Umständen eine ganzen Subbaum beeinflusst. Will man beispielsweise einen Customer aus dem Beispieldokument löschen, so muß man auch die Subelemente (z.B. Address,City,State,Order...) löschen. Dies wird insbesondere

dann schwierig, wenn man das XML-Dokument in einer relationalen Datenbank gespeichert hat. In diesem Fall muß die obige Änderung an der Relation `Customer` auch an die übrigen Relationen durchpropagiert werden. Hierzu gibt es wiederum drei Möglichkeiten. Zum einen könnte beim Anlegen der Relationen die Datenbankfunktion Cascading Delete verwenden werden. Dieser Lösung sehr ähnlich ist die Verwendung von Triggern. Mit diesen beiden Funktion kann die Änderung wie gewünscht durchpropagiert werden. Dennoch ist dieses Vorgehen sehr statisch und muss bei Änderung in der Struktur des XML-Dokuments immer neu angepasst werden. Etwas flexibler ist die Verwendung von Access Support Relations (ASR). Diese ASR enthalten alle Beziehungen zwischen Vätern und Kindern. Soll nun ein Vater-Element gelöscht werden, können mit Hilfe der ASR alle Kinder dieses Elements festgestellt und anschließend gelöscht werden. Für diese Möglichkeit müssen mehrere SQL-Anweisungen generiert werden. Wie diese jedoch generiert werden sollen, ist noch offen.

Ein drittes Problem, das beim Updaten entsteht, ist ebenfalls noch offen. Viele XML-Dokumente besitzen eine DTD, mit deren Hilfe man feststellen kann, ob ein vorliegendes Dokument gültig ist. Führt man nun ein Update auf ein XML-Dokument aus, so kann es passieren, dass ein zuvor gültiges Dokument nicht mehr gültig ist. Im ersten Beispiel für Updates wurden aus dem XML-Dokument alle States entfernt. Die DTD für das Beispiel sieht nun aber vor, dass jedes Address-Element ein State-Element besitzt. Bevor das Update ausgeführt wurde, war dieser Zustand gegeben. Im Anschluss an das Update hingegen ist der Zustand nicht mehr gegeben, dass Dokument ist ungültig.

1.3 Fazit

Wie man im Besonderen im letzten Abschnitt gesehen hat, gibt es noch viele Probleme bei der Verwendung von Updates für XML-Dokumente. Dennoch erscheint es durchaus sinnvoll, eine Erweiterung für XQuery zu schaffen, die das Updaten ermöglicht. Besonders die Abstraktion der Anweisung von der physischen Speicherung macht die Verwendung von Updates so interessant. Zusätzlich ist es durch die Erweiterung von XQuery möglich, diese Update-Anweisungen einfach und verständlich zu halten.

2 Isolation in XML-Datenbanken

Isolation ist ein wichtiger Bestandteil von Datenbanksystemen. Jede Anwendung, die auf einem Datenbanksystem arbeitet, soll so ablaufen können, als ob sie die einzige Anwendung auf der Datenbank ist. Hauptsächlich geht es darum, parallel arbeitende Applikationen korrekt voneinander zu trennen. Es werden im Folgenden verschiedene Protokolle aus [5] vorgestellt, die genau dies ermöglichen sollen. Zunächst jedoch werden die grundlegenden Konzepte von Transaktionen

und Zweiphasen-Sperren erläutert.

2.1 Transaktionen

Eine Transaktion ist eine Folge von Operationen auf einer oder mehreren Datenbanken, für die folgende Eigenschaften vom Datenbanksystem garantiert werden:

Atomarität: Die Operationen einer Transaktion werden entweder vollständig oder überhaupt nicht ausgeführt.

Konsistenz: Jede Transaktion überführt die Datenbank von einem gültigen Zustand in einen anderen gültigen Zustand.

Isolation: Transaktionen arbeiten auf der Datenbank so, als ob es keine parallel laufenden Transaktionen gäbe. Die Datenbank muss gleichzeitig laufende Transaktionen voreinander verbergen.

Persistenz: Auch im Fall eines Fehlers der Hard- oder Software sollen die Änderungen einer abgeschlossenen Transaktion nicht verloren gehen.

2.2 Operationen auf XML-Dokumenten

Die Operationen einer typischen API zum Bearbeiten von XML-Dokumenten (z.B. DOM) können in 4 Kategorien eingeteilt werden. Dabei wird unterschieden, ob eine Operation liest oder schreibt, und ob sie dies auf dem Inhalt oder der Struktur des Dokuments tut.

In [5] wird davon ausgegangen, dass Inhaltsänderungen von vorhandenen Synchronisationsprotokollen abgehandelt werden können. Daher werden im Folgenden nur Strukturveränderungen betrachtet. Ein XML-Dokument wird als Baum betrachtet, in dem jeder Knoten je einen Zeiger auf seinen linken und rechten Nachbarn und je einen Zeiger auf sein erstes und sein letztes Kind hat. Um nun Strukturänderungen durchführen zu können, benötigt man eine repräsentative Menge von Befehlen:

sd: springe zur Wurzel des Dokuments.

nthF: gehe zum n-ten Kind des aktuellen Knotens.

nthB: gehe zum n-tem Kind des aktuellen Knotens (rückwärts gezählt ab dem letzten Kind)

del: lösche den aktuellen Knoten.

ins: füge einen neuen Knoten vor dem aktuellen Knoten ein.

2.3 Protokolle zur Synchronisation von Lese- und Schreiboperationen

2.3.1 Zweiphasen-Sperrprotokolle

Sperrprotokolle verlangen von Transaktionen, dass vor einem Zugriff auf ein Objekt eine entsprechende Sperre für das Objekt angelegt wird. Diese Sperren werden von dem Datenbanksystem selbst implementiert. Normalerweise gibt es mindestens zwei verschiedene Arten von Sperren.

Zum einen wären da die *exclusive* Sperren, sie werden benötigt, wenn schreibend auf die Daten zugegriffen werden soll. Sollen nur Lesezugriffe auf die Objekte stattfinden, werden *shared* Sperren angefordert. Wenn auf einem Objekt eine *exclusive* Sperre angelegt wurde, ist es nicht mehr möglich, eine weitere Sperre für dieses Objekt anzufordern. In diesem Fall muss die spätere Transaktion so lange warten, bis die Sperre wieder freigegeben wurde.

Beim Zweiphasen-Sperren läuft das Anlegen und Freigeben von Sperren, wie der Name schon sagt, in zwei Phasen ab. In Phase 1, der Wachstumsphase, dürfen Sperren zwar angefordert, jedoch nicht wieder freigegeben werden. Bei Phase 2, auch Schrumpfungsphase genannt, werden Sperren freigegeben, es dürfen allerdings keine Sperren angefordert werden. Zweiphasen-Sperrprotokolle garantieren, dass alle entstehenden Schedules serialisierbar sind.

Doc2PL

Das erste und einfachste Protokoll *Doc2PL* erwirbt Sperren für gesamte Dokumente. Diese Vorgehensweise ist sinnvoll, wenn Transaktionen überwiegend auf unterschiedlichen Dokumenten arbeiten, also z.B. verschiedene Autoren ihr jeweils eigenes Dokument bearbeiten. *Doc2PL* ist einfach zu implementieren und erzeugt kaum Protokolloverhead.

Node2PL und NO2PL

Diese beiden Protokolle erwerben Sperren für Knoten. *Node2PL* sperrt Elternknoten. Wenn zum Beispiel zu einem Kind eines Knotens P abgestiegen werden soll, so wird eine *shared* Sperre für den Knoten P erworben. Wird ein Kind von Knoten P verändert, muss P *exclusive* gesperrt werden.

NO2PL Protokoll dagegen sperrt jeweils diejenigen Knoten, deren Zeiger passiert oder verändert werden. Wenn zum Beispiel ein neues Kind C0 vor Kind C1 eingefügt werden soll, müssen zwei *exclusive* Sperren erworben werden:

- eine am Elternknoten P, da sein Zeiger auf das erste Kind verändert werden muss
- eine am Kind C1, weil der Zeiger auf den linken Nachbarn nun auf C0 zeigen sollte

Bemerkenswert ist, dass C0 nicht gesperrt werden muss, da keine andere Transaktion in der Lage sein wird, diesen Knoten zu erreichen. Alle Wege zu C0 sind bereits durch die vorhandenen Sperren blockiert.

Sowohl *Node2PL* als auch *NO2PL* benötigen höchstens eine Sperre pro Knoten pro Transaktion. Der Unterschied ist, dass auf Blattebene (wo sich die meisten Knoten befinden), *Node2PL* keine Sperren erwerben muss. *NO2PL* dagegen benötigt diese Sperren.

OO2PL

Von den hier vorgestellten Zweiphasen-Sperrprotokollen erreicht *OO2PL* den feinsten Grad der Granularität, indem es Sperren auf Zeiger erwirbt. Dadurch benötigt *OO2PL* höchstens vier Sperren pro Transaktion pro Knoten und erzeugt damit folglich im schlechtesten Fall den vierfachen Overhead im Vergleich zu *Node2PL* beziehungsweise *NO2PL*.

Beispiel

Betrachten wir nun abschließend ein Beispiel für die Funktionsweise der Zweiphasen-Sperrprotokolle. In Tabelle 1 werden die beiden Transaktionen T_1 und T_2 vorgestellt. T_1 löscht einen Knoten, T_2 dagegen enthält keine Strukturverändernden Operationen.

In Abbildung 2 ist dargestellt, welche *exclusive* Sperren T_1 je nach Sperrprotokoll erwerben muss, um den Knoten n_3 löschen zu können. Bei diesem Beispiel sind nur die Sperren die für *OO2PL* angelegt werden müssen, so fein, dass T_2 gleichzeitig mit T_1 ausgeführt werden kann. Bei *Node2PL* darf T_2 nicht auf die Wurzel n_1 zugreifen. *NP2PL* ermöglicht zwar noch den Zugang zu n_1, eine Überquerung von n_4 ist jedoch nicht mehr möglich.

2.3.2 Timestamp-based protocol

Das *timestamp-based protocol* beruht auf der Idee, mehrere Versionen eines Dokuments im Speicher zu halten. Somit ist es möglich, konkurrierenden Transaktionen die jeweils richtige Version als Arbeitsgrundlage zur Verfügung zu stellen.

T_1		T_2	
`sd`	$\Longrightarrow n_1$		
`nthF(2)`	$\Longrightarrow n_2, n_3$		
`del`	delete n_3		
		`sd`	$\Longrightarrow n_1$
		`nthB(1)`	$\Longrightarrow n_4$
		`nthF(1)`	$\Longrightarrow n_7$

Tabelle 1: Beispiel Transaktionen

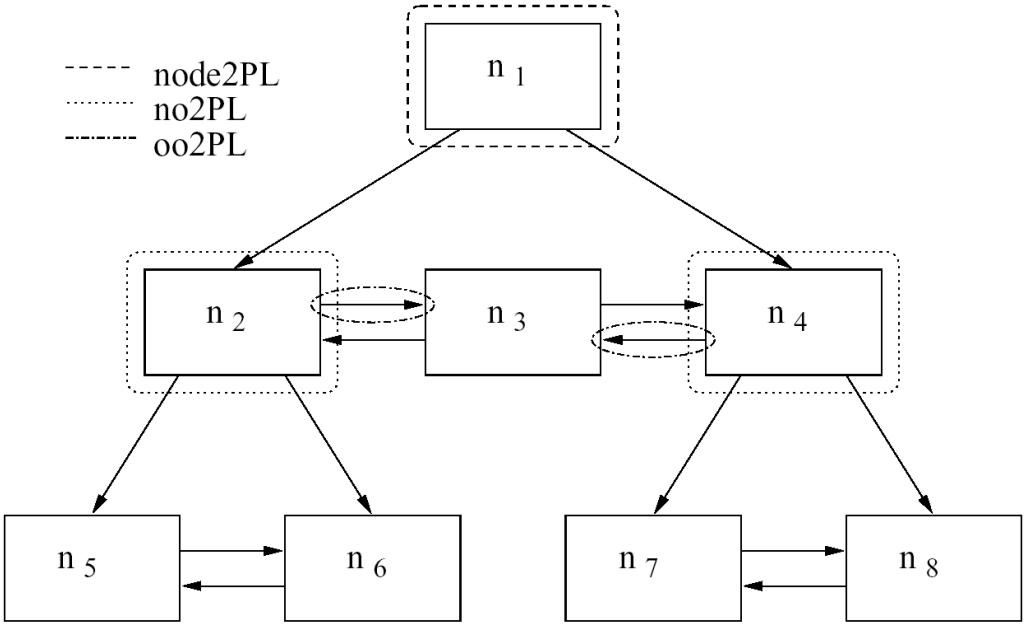

Abbildung 2: *exclusive* Sperren einer Transaktion bei verschiedenen 2PL Protokollen (n_3 wird gelöscht)

Basis dieser Vorgehensweise sind Zeitstempel. Jeder Transaktion wird vom Transaction Manager ein eindeutiger „timestamp“ zugeteilt. Sämtliche Operationen einer Tranaktion erben diesen Zeitstempel. Damit ist gewährleistet, dass der Scheduler die einzelnen Operationen den entsprechenden Dokumentversionen zuordnen kann.

Der eigentliche Trick besteht nun darin, Strukturänderungen wie Einfügen und Löschen erst dann für andere Transaktionen sichtbar zu machen, wenn die ändernde Transaktion abgeschlossen ist. Deshalb werden Knoten nicht sofort gelöscht, sondern nur als *deleted* markiert. Analog dazu werden eingefügte Knoten zwar in das Dokument eingesetzt, werden aber als *inserted* markiert. Erst nach Abschluss der Transaktion werden die als gelöscht markierten Knoten aus dem Dokument entfernt und alle *inserted* Markierungen aufgehoben.

Wie nun die einzelnen Operationen auf den Daten arbeiten dürfen, ist in Tabelle 2 dargestellt. op_F stellt hierbei eine bereits durchgeführte Operation auf dem Knoten N dar. op_S hingegen ist eine Operation auf dem gleichen Knoten N, die der Scheduler als nächstes durchführen will. Vergleicht man nun die beiden Zeitstempel der Operationen, dann ergeben sich zwei verschiedene Fälle. Einmal ist die bereits durchgeführte Operation von ihrem Zeitstempel her gesehen vor der Anstehenden ($op_F <_{TS} op_S$), oder aber nicht ($op_F >_{TS} op_S$). Zum Verständnis der Tabelle 2 fehlt noch, dass `T` für Überqueren, `D` für Löschen und `I` für Einfügen steht.

Wie man sieht, gibt es keine Konflikte, wenn beide Operationen den Knoten

op_F	op_S	$op_F <_{TS} op_S$	$op_F >_{TS} op_S$
T	T	kein Konflikt	kein Konflikt
D	T	ignorieren (cascading abort)	überqueren
I	T	überqueren	ignorieren
T	D	als gelöscht markieren	T_S abbrechen
D	D	nicht möglich	T_S abbrechen
I	D	T_S blockieren, dann als gelöscht markieren	nicht möglich
T	I	nicht möglich	nicht möglich
D	I	nicht möglich	nicht möglich
I	I	nicht möglich	nicht möglich

Tabelle 2: Aktionstabelle für XTO

nur überqueren möchten (`TT`). Ebenfalls einfach wird es, wenn der Knoten N erst von op_S eingefügt werden soll. Denn op_F kann in diesem Fall den Knoten N nicht gesehen haben. Aus diesem Umstand resultieren die drei unteren Zeilen in Tabelle 2.

Der Fall `DT` ist interessanter. Falls op_S einen späteren Zeitstempel hat, als die Operation, die den Knoten N gelöscht hat, so muss op_S diesen Knoten ignorieren. Falls der Elternknoten von N keine nicht gelöschten Kinder mehr hat, führt dies zu einem Abbruch der Transaktion, zu der op_S gehört. Wurde N jedoch von einer Transaktion gelöscht, die von ihrem Zeitstempel her nach der Operation kommt, die N jetzt überqueren möchte, so wird das passieren des Knotens erlaubt.

Im `TD` Fall möchte die anstehende Operation einen Knoten löschen, welcher zuvor schon von einer anderen Operation überquert wurde. Falls die löschende Operation op_S älter ist als op_F, entsteht kein Konflikt. Der Knoten wird einfach als gelöscht markiert. Im umgekehrten Fall muss die zu op_S entsprechende Transaktion abgebrochen werden. Theoretisch wäre es allerdings auch möglich, die andere Transaktion abzubrechen. Da aber nur ein maximaler Zeitstempel pro Knoten gespeichert wird, können Zugriffe jüngerer Transaktionen nicht mehr zweifelsfrei zurückverfolgt werden.

2.4 Fazit

Es wurden verschiedene Protokolle zur Synchronisation von Lese- und Schreiboperationen vorgestellt. Sie sollen die Eigenschaft der Isolation innerhalb eines auf XML basierenden Datenbanksystems gewährleisten.

Leider fehlen in [5] jegliche Untersuchungen über die Effizienz der vorgestellten Protokolle. Es lässt sich zwar vermuten, dass die Anzahl der parallel arbeitenden Transaktionen von der Granularität des verwendeten Sperrprotokolls abhängt. Wie stark der durch feines Sperren erzeugte Overhead Einfluss auf die Performanz hat, ist jedoch ohne Untersuchung nicht abzusehen.

3 Erkennen von Änderungen in XML-Dokumenten

3.1 Einführung

Das Erkennen von Änderungen in XML-Dokumenten ist eine wichtige Anwendung im Bereich von Internet-Suchmaschinen, die XML-Dokumente indizieren sollen. Es ist zu erwarten, dass sich der Inhalt dieser Dokumente oft ändert. Die Suchmaschine sollte erstens diese Änderungen überhaupt bemerken und zweitens die Änderungen anschliessend in ihren Index einpflegen können, anstatt den ganzen Index für das Dokument neu aufzubauen. Ausserdem ist diese Anwendung wichtig für sogenannte *Continuous Query Systems*, auch *Trigger Systems* genannt. Das sind Systeme, die auf die Änderung eines Dokuments reagieren und eine Aktion auslösen sollen. Schliesslich kann man das Finden der Unterschiede zwischen zwei Dokumenten auch dazu benutzen, um eine Versionierung der Daten durchzuführen. Im Folgenden stellen wir zwei verschiede Algorithmen vor, die diese Aufgaben bewerkstelligen. Beide Algorithmen sind auf XML-Dokumente spezialisiert und liefern nach Eingabe zweier Dokumente eine Anleitung, wie das erste Dokument in das zweite überführt werden kann.

3.2 XY-Diff

XY-Diff [2] wurde als Teil des Xyleme-Projektes [6] entwickelt. Seine Aufgabe ist es, von einem Web-Crawler gefundene Dokumente auf Änderungen zu untersuchen, um die gefundenen Dokumente schneller verarbeiten zu können. Da XY-Diff seine Eingabedokumente nicht als reinen Datenspeicher ansieht, sondern eher als Beschreibungssprache für Dokumente, behandelt XY-Diff die XML-Dokument-Bäume als geordnete Bäume.[1] Der Algorithmus wurde auf Geschwindigkeit optimiert. Eine wesentliche Anforderung bei der Entwicklung war, dass XY-Diff im Mittel in linearer Zeit laufen sollte (linear in Bezug auf die Grösse der Eingabedokumente). Dies hat zur Folge, dass das sog. *Diff* (die Anleitung, wie das erste Dokument in das zweite überführt werden kann) länger sind, als unbedingt notwendig.

3.2.1 Arbeitsweise von XY-Diff

Um die Änderungen am Dokument zu finden betrachtet XY-Diff die Bäume des alten und neuen Dokuments. Zunächst wird jeder Knoten darin mit einem Gewicht, das die Grösse des Baumes unter dem Knoten darstellt, und einem Hash, der den jeweiligen Knoten einschliesslich des kompletten Sub-Baumes unter diesem Knoten repräsentiert, versehen. Dann legt sich der Algorithmus für das neue Dokument eine Priority Queue an und legt den Wurzelknoten mitsamt seinem Gewicht hinein.

[1] die Links-Rechts-Beziehung von Geschwistern im Baum spielt eine Rolle

Anschliessend wird immer der Knoten mit dem grössten Gewicht aus der Priority Queue genommen, danach sein Hash mit dem entsprechenden[2] Knoten des alten Dokuments verglichen und, falls unterschiedlich, alle Kinder des Knotens mit ihren jeweiligen Gewichten in die Priority Queue abgelegt. Falls die Hashes gleich sind, kann der ganze Teilbaum keine Änderungen mehr enthalten und ist damit komplett abgehandelt. Der Algorithmus wählt also in jedem Schritt immer die schwersten Teilbäume zur weiteren Verarbeitung aus, in der Hoffnung, große Teilbäume, an denen sich nichts geändert hat, zuerst zu finden.

Wenn alle Änderungen gefunden wurden, muss XY-Diff noch das *Diff* erstellen. Auch hierzu bedient sich der Algorithmus ganz massiv verschiedener Heuristiken, die zwar einerseits die Verarbeitungsgeschwindigkeit steigern, aber andererseits die Genauigkeit des *Diffs* herabsetzen.

3.3 X-Diff

X-Diff [1] wurde entwickelt, um in einer XML-basierten Datenbank Änderungen zu finden. Dementsprechend betrachtet der Algorithmus die Bäume der zu verarbeitenden XML-Dokumente als ungeordnet, d.h. die Reihenfolge von Knoten auf der gleichen Ebene spielt keinerlei Rolle. Dies ist auch leicht einzusehen, wenn man sich z.B. eine Kundentabelle in einem XML-Dokument kodiert vorstellt. Ob ein Kunde oben oder unten im Dokument steht, ist dabei nicht relevant.

Der Algorithmus wurde so gestaltet, dass er immer optimale *Diffs* liefert. Allerdings ist das Problem, Änderungen in ungeordneten Bäumen zu finden, NP-Vollständig. X-Diff nutzt dabei aber noch Kentnisse über XML-Strukturen aus, um das Problem dennoch in polynomieller Zeit zu lösen. Trotzdem ist X-Diff damit immer noch bedeutent langsamer als XY-Diff.

In [1] wurde noch eine Variante von X-Diff namens X-Diff+ vorgestellt, die sich einiger Heuristiken bei der Zuordnung von Knoten im neuen Dokument zu Knoten im alten Dokument bedient. Dadurch steigt die Verarbeitungsgeschwindigkeit von X-Diff erkennbar an, aber die *diffs* sind manchmal nicht ganz optimal.

3.4 Vergleich von XY-Diff und X-Diff

3.4.1 Geschwindigkeit

Um die Geschwindigkeit der verschiedenen Diff-Algorithmen zu beurteilen, wurden Laufzeitmessungen auf den Daten von Schauspielern aus [7] durchgeführt. In Abbildung 3 sind die Messergebnisse graphisch dargestellt. Auf der X-Achse ist die Grösse des Eingabedokuments, auf der Y-Achse die Laufzeit abgetragen. Beide Achsen sind logarithmisch skaliert. Man kann sehen, dass die Laufzeit von XY-Diff ungefähr linear im Verhältnis zur Grösse der Eingabedaten ist, während

[2]Um zu entscheiden, welche Knoten entsprechende Knoten sind, bedient sich der Algorithmus einiger Heuristiken, deren Erklärung hier den Rahmen sprengen würde.

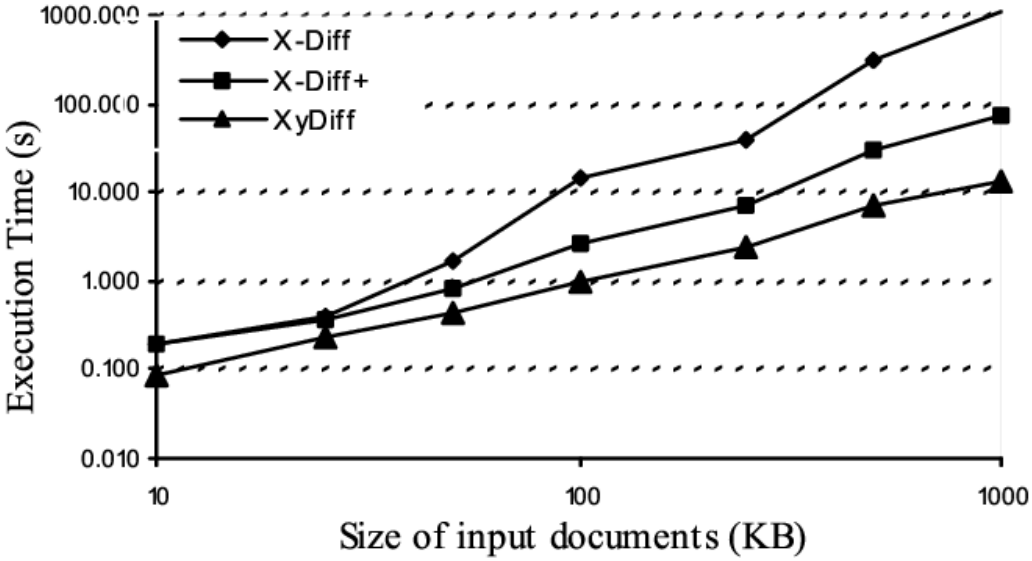

Abbildung 3: Performance-Vergleich von X-Diff und XY-Diff auf logarithmischen Skalen.

X-Diff quadratische Laufzeit hat. Man sieht auch, dass X-Diff+ bei kleineren Dokumenten keine Vorteile gegenüber X-Diff bietet, bei grösseren Dokumenten aber 5-10 mal schneller läuft.

3.4.2 Qualität

In Abbildung 4 ist auf der Y-Achse abgetragen, wie lang die resultierenden *diffs* der verschiedenen Algorithmen im Verhältnis zu einem optimalen *diff* sind. Auf der X-Achse ist die jeweilige Änderungsrate des Dokuments abgetragen. Von X-Diff wurde in [1] gezeigt, dass er immer optimale Ergebnisse liefert (im Graphen also eine Gerade bei $y = 1$ hätte). Die Variante mit Heuristik liefert fast immer optimale Ergebnisse. Nur bei relativ hohen Änderungsraten wird das *diff* etwas länger als unbedingt nötig. XY-Diff dagegen liefert schon bei moderaten Änderungsraten schlechtere Ergebnisse, bei hohen Raten sogar Ergebnisse die doppelt so lang sind, wie die von X-Diff.

3.5 Fazit

Wir haben zwei verschiedene Algorithmen kennen gelernt, die in völlig verschiedenen Kontexten mit verschiedenen Anforderungen entwickelt worden sind. Dementsprechend unterschiedlich sind sie auch im Bezug auf Laufzeit und Qualität des Ergebnisses. Beide verwenden die Struktur des zugrundeliegenden XML-Dokuments um die Laufzeit zu verkürzen. XY-Diff und X-Diff+ verwenden zusätzlich noch Heuristiken um nochmals massiv Zeit zu sparen, allerdings auf Kosten

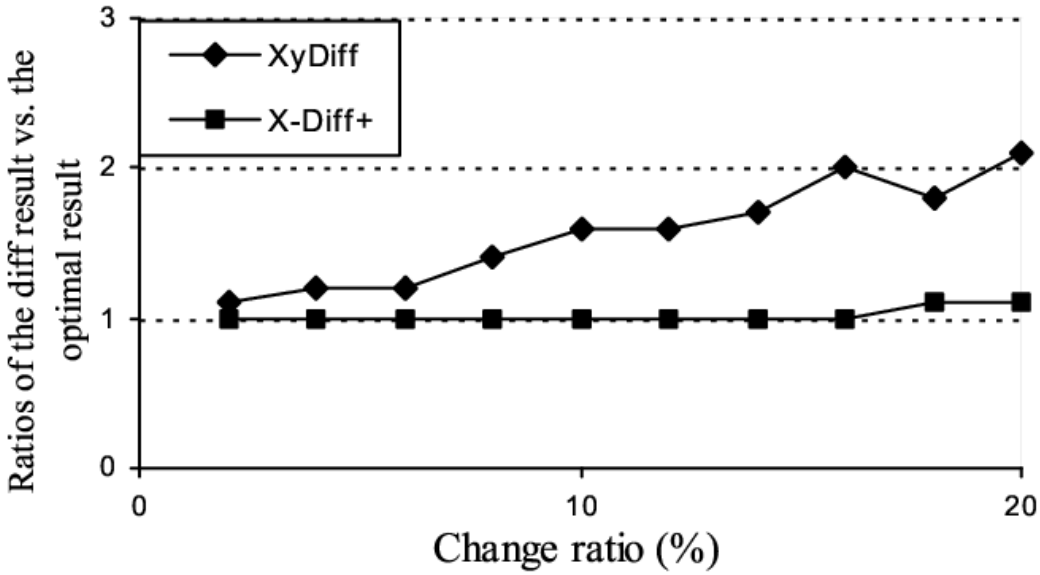

Abbildung 4: Länge der resultierenden *diffs*.

der Qualität. Herkömmliche, für Textdateien entworfene *diff*-Algorithmen, eignen sich nur bedingt oder gar nicht für XML-Dokumente, da sie meist zeilenbasiert arbeiten, das ganze XML-Dokument aber in einer einzigen Zeile stehen kann.

Literatur

[1] Y. Wang, D. DeWitt, J-Y Cai: *X-Diff: An Efficient Change-Detection Algorithm for XML Documents.* In: Proceedings of the 19th International Conference on Data Engineering (ICDE), Bangalore, India, 2003.

[2] G. Cobena, S. Abiteboul, A. Marian: *Detecting Changes in XML documents.* In: Proceedings of the 18th International Conference on Data Engineering (ICDE), San Jose, CA, 2002.

[3] Igor Tatarinov et al.: *Updating XML*.In: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA, 2001.

[4] Torsten Grabs, Klemens Böhm, Hans-Jörg Schek: *XMLTM: Efficient Transaction Management for XML Documents.* In: Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 4-9, 2002.

[5] Sven Helmer, Carl-Christian Kanne, Guido Moerkotte: *Isolation in XML Bases*. Technical Report, University of Mannheim, Germany, 2001.

[6] Xyleme. `http://www.xyleme.com`

[7] `http://www.cs.wisc.edu/niagara/data.html`