XML-Algebren

Martin Winkler

Universität Kaiserslautern,
FB Informatik,
Postfach 3049,
D-67653 Kaiserslautern,
Deutschland

Zusammenfassung Im vorliegenden Beitrag werden drei Algebren vorgestellt, die zur Beschreibung von Anfragen in einer XML-Anfragesprache entworfen wurden. Dazu werden erst das jeweils zu Grunde liegende Datenmodell, dann die jeweils angebotenen Operatoren, ihre Eingabeparameter und ihre Fuktionsweise vorgestellt. Im Anschluss an die Vorstellung einer jeden Algebra wird anhand einer konkreten XQuery-Anfrage die Abbildung auf Operatoren der jeweiligen Algebra aufgezeigt. Dabei werden tupelbasierte Algebren von baumbasierten Algebren abgegrenzt. Einige Vorteile und Nachteile, die sich aus den Charakteristika des jeweiligen Konzeptes ergeben, werden im abschließenden Abschnitt diskutiert und Ansätze zur Kompensation genannt.

1 Einführung

Datenbanksysteme sind aus heutigen Softwaresystemen nicht mehr wegzudenken. Einem fortlaufenden Weiterentwicklungsprozess unterliegend, wurde die Geschwindigkeit, in der Anfragen auch auf mehreren Gigabyte großen Datenbeständen bearbeitet und beantwortet werden können, immer weiter gesteigert. Das zu Grunde liegende Relationenmodell und die darauf arbeitenden Zugriffs- und Manipulationsoperatoren sind ebenso zu einem unverzichtbaren Teil dieser Systeme geworden, denn sie bieten die Möglichkeit deklarative Anfragen formulieren zu können. Trotzdem musste dieses Konzept sich erst gegenüber konkurrierenden Konzepten beweisen und konnte seinen wahren Wert erst mit der Zeit zeigen.

Die zunehmende Popularität von XML zur Beschreibung von Daten führt dazu, dass zukünftige Datenbanken ebenfalls in XML repräsentierte Daten aufnehmen und mit hoher Geschwindigkeit Operationen darauf durchführen können müssen. Dies führt zur Entwicklung neuer Konzepte. Dabei ist es von hohem Interesse, altbekannte und bewährte – für das Relationenmodell und die relationale Algebra entwickelte – Optimierungsmöglichkeiten auf die neuen Konzepte übertragen zu können. Algebraische Optimierungen sind allerdings nur möglich, wenn Anfragen in einer Anfragesprache wie beispielsweise XQuery in einer Algebra repräsentierbar sind. Im nachfolgenden Abschnitt werden einige Algebren vorgestellt, die zu diesem Zwecke entwickelt wurden. Darauf folgend werden einige Vor- und Nachteile der jeweiligen Ansätze gegenübergestellt.

2 Algebren zur XML-Verarbeitung

Eine Algebra für die Verarbeitung von XML-Dokumenten oder allgemein einer Datenbasis, die in XML definiert ist, zu entwickeln, ist schon von mehreren Autoren in Angriff genommen worden. Daher ist es auch nicht überraschend, dass sich diese Algebren in ihrer Konzeption, dass heißt die Sichtweise auf die Daten und, daraus resultierend, die Manipulation dieser Daten mittels algebraischer Operatoren, mitunter stark unterscheiden. Erfolgreich entwickelt und bewährt hat sich die relationale Algebra mit ihrem Datenmodell der Tupel und Relationen und den Möglichkeiten der Manipulation durch Operatoren wie *Selektion*, *Projektion*, *Join* und *Aggregation*. An deren Konzept anknüpfend sind tupelbasierte Algebren denkbar, die die Knoten eines XML-Baumes wie Tupel behandeln und deren Kindknoten wie Attributwerte dieser Tupel. Eine konkrete Algebra, der dieses Konzept zu Grunde liegt, ist *NAL*.

Ein zweiter Ansatz sieht sämtliche XML-Elemente als Knoten von Bäumen an. Elemente innerhalb anderer Elemente eines XML-Dokuments liegen im Baum als Kindknoten vor, um die im Dokument herrschende Hierarchie und Reihenfolge beizubehalten. Die gesamte Datenbasis besteht in der Regel aus einem ganzen Wald solcher Bäume. *TAX* ist eine solche *baumbasierte Algebra*. Beide Ansätze sind in ihrer Ausdrucksmächtigkeit vergleichbar und bieten eine Reihe von Operatoren zur Extraktion von Informationen aus der Datenbasis sowie zu ihrer Manipulation. Die Arbeitsweise dieser Algebra-Operatoren ist allerdings, entsprechend der zu Grunde liegenden Sichtweise, sehr unterschiedlich.

Ein Ansatz, um die Vorteile beider Sichtweisen zu vereinen, ist *TLC*. Diese „hybride“ Algebra arbeitet ebenfalls auf Bäumen, ein einzelner Knoten eines solchen Baumes steht hier aber nicht zwangsläufig für ein einziges XML-Element, sondern beispielsweise für alle Elemente mit bestimmten Eigenschaftsausprägungen, was an die Tupelsequenzen von *NAL* erinnert.

In Abschnitt 2.1 wird *NAL* näher dargestellt. Abschnitt 2.2 befasst sich mit *TAX*, Abschnitt 2.3 mit *TLC*. Alle drei Abschnitte gliedern sich jeweils in die Vorstellung des Datenmodells, der Operatoren und abschließend der Abbildung von XQuery-Anfragen auf Operatoren der Algebra. Dazu wird ein konkretes Beispiel in die jeweilige Algebra übersetzt.

2.1 Tupelbasierte XML-Algebren

In diesem Abschnitt wird ein Einblick in XML-Algebren gegeben, deren Operatoren auf Tupeln arbeiten.

Ein an relationaler Algebra angelehnter Ansatz, eine Algebra zur Verarbeitung von XQuery-Ausdrücken zu entwickeln, stammt von Brantner et al. [1]. Die Natix-Algebra (*NAL*) arbeitet auf Tupelsequenzen und soll hier daher als Beispiel für tupelbasierte Algebren vorgestellt werden. Sie wurde von Brantner et al. an der Universität Mannheim für das XML Datenbankmanagementsystem *Natix* entwickelt und eingesetzt.

Datenmodell. In *NAL* werden Knoten eines XML-Baumes als Tupel betrachtet. Tupel, deren Attributwerte im einfachsten Fall vom Typ *String*, *number* oder *boolean* sind, aber genausogut auch selbst wieder Tupelsequenzen sein können. Die Operatoren der Algebra arbeiten auf geordneten Tupelsequenzen. Die Tupel, die zu einer Sequenz gehören, haben immer die gleichen Attribute und unterscheiden sich nur in deren Belegungen. Nachfolgend werden die Operatoren vorgestellt.

Selektion. Der Selektionsoperator σ prüft, beginnend beim ersten Tupel der Tupelsequenz, die er als Eingabe erhält, ob das betrachtete Tupel das Selektionsprädikat p erfüllt. Ist dies der Fall, so findet sich dieses Tupel in der Ergebnissequenz wieder, andernfalls nicht. Er fährt anschließend mit der Prüfung des nächsten Tupels der Sequenz fort. Die Reihenfolge der Tupel der Ergebnissequenz entspricht der in der Eingabesequenz.

Formale Notation der Selektion: $\sigma_p(e)$

Projektion. Beginnend beim ersten Tupel der Eingabesequenz werden nacheinander sämtliche Tupel in die Ausgabesequenz übertragen, wobei jedes übertragene Tupel nur noch über die im Eingabeparameter A, welcher aus einer Reihe von Bezeichnern besteht, gelisteten Attribute verfügt. Der Operator wird mit Π_A bezeichnet.

Formale Notation der Projektion: $\Pi_A(e)$

Duplikateliminierung. Als spezielle Form des Projektionsoperators existiert in *NAL* eine Duplikateliminierung Π^D. Angewandt auf eine Tupelsequenz ist die resultierende Ausgabesequenz frei von Duplikaten.

Formale Notation der Duplikateliminierung: $\Pi^D_A(e)$oder $\Pi^D(e)$

Umbenennung. Der Umbenennungsoperator $\Pi_{a':a}$ erhält als Eingabe den neuen Namen eines Attributes, den alten Namen eines Attributes und eine Tupelsequenz. Die Tupel der Eingabe werden beginnend beim ersten Tupel in die Ausgabesequenz übertragen. Wobei das spezifizierte Attribut in sämtlichen Tupeln der Ausgabe den neuen Namen trägt.

Formale Notation der Umbenennung: $\Pi_{a':a}(e)$

Hinzufügen eines Attributes. Der *Map*-Operator χ erhält den Namen eines neu hinzuzufügenden Attributes, einen Algebra-Ausdruck e_2 und die Tupelsequenz e_1. Beginnend beim ersten Tupel der Eingabesequenz wird jedem Eingabetupel das neue Attribut hinzugefügt. Es erhält als Wert das Resultat des ausgewerteten Ausdruckes e_2. In den Ausdruck e_2 wird das aktuell betrachtete

Tupel eingesetzt, somit entsprechen die Belegungen freier Variablen im Ausdruck e_2 den Attributwerten dieses Tupels.

Wird dem *Map*-Operator als neuer Attributname ein in den Tupeln der Eingabesequenz bereits vorhandener Attributname übergeben, so wird dieses Attribut einfach mit den neuen Werten überschrieben.

Formale Notation des Map-Operators: $\chi_{a:e_2}(e_1)$

Kreuzprodukt. Der Kreuzproduktoperator $e_1 \times e_2$ ist ein binärer Operator, der zwei Tupelsequenzen e_1 und e_2 in eine Ausgabetupelsequenz überführt. Dabei werden je zwei Tupel konkateniert und bilden ein neues Tupel der Ausgabesequenz mit den Attributen des ersten Tupels gefolgt von denen des zweiten. Die zu konkatenierenden Tupelpaare entstehen dabei wie folgt: Jedem Tupel der linken Sequenz, beginnend beim ersten, wird ein Tupel der rechten Seite zugewiesen, bis alle möglichen Paarungen gebildet wurden. Ist die linke Tupelsequenz die leere Sequenz ε, so ist das Ergebnis ebenfalls die leere Sequenz.

Formale Notation des Kreuzproduktes: $e_1 \times e_2$

Produkt. Das Produkt $t_1 \bar{\times} e_2$ ist ein binärer Operator, der aus einem Tupel t_1 und einer Tupelsequenz e_2 eine resultierende Tupelsequenz bildet. Die Tupel der Ausgabesequenz ergeben sich aus der Konkatenation des Eingabetupels mit je einem Tupel der Eingabesequenz. Die Attributmenge der resultierenden Sequenz entspricht somit der Vereinigung der Attributmengen des Eingabetupels und der Eingabesequenz.

Formale Notation des Produktes: $t_1 \bar{\times} e_2$

Dependency Join. Der *Dependency Join* $\overset{\bowtie}{\rightarrow}$, kurz *D-Join*, erzeugt aus zwei Tupelsequenzen eine Ausgabesequenz und ähnelt daher stark dem Kreuzproduktoperator. Der *D-Join* bildet das Produkt aus dem ersten Tupel der linken Eingabesequenz e_1 und der rechten Eingabesequenz e_2, wobei er e_1 in e_2 einsetzt. Das heißt, freie Variablen in e_2 werden mit Werten aus e_1 belegt: e_2 ist von e_1 abhängig, daher die Bezeichnung *Dependency Join*. Anschließend fährt er mit dem nächsten Tupel von e_1 fort und bildet wiederum das Produkt aus diesem Tupel und der Sequenz e_2, in die jetzt e_1 ohne das erste Tupel eingesetzt wird.

Formale Notation des Dependency Join: $e_1 \overset{\bowtie}{\rightarrow} e_2$ oder $e_1 < e_2 >$

Semi-Join. Der *Semi-Join*-Operator $\ltimes$ erhält als Eingabe zwei Tupelsequenzen e_1, e_2 und ein logisches Prädikat p. Ein Tupel aus e_1 wird genau dann in die Ausgabetupelsequenz übertragen, wenn in e_2 ein Tupel existiert, so dass die Konkatenation beider Tupel das Prädikat erfüllt. Ist e_1 die leere Sequenz ε, so ist auch das Ergebnis die leere Sequenz.

Formale Notation des Semi-Join: $e_1 \ltimes_p e_2$

Anti-Join. Der *Anti-Join*-Operator $\triangleright$ ähnelt dem *Semi-Join*. Er erhält als Eingabe zwei Tupelsequenzen e_1, e_2 und ein logisches Prädikat P. Ein Tupel aus e_1 wird genau dann in die Ausgabesequenz übertragen, wenn für alle Elemente aus e_2 die Konkatenation beider Tupel das Prädikat nicht erfüllt.

Formale Notation des Anti-Join: $e_1 \triangleright_p e_2$

Unnesting. Der *Unnesting*-Operator μ wird benötigt um Attribute „*auszupacken*“. Dazu wird das auszupackende Attribut g für ein Tupel der Sequenz ausgeblendet und anschließend als seperates Tupel mit diesem konkateniert.

Formale Notation des Unnesting-Operators: $\mu_g(e)$

Unnest-Map. Der *Unnest-Map*-Operator Υ verlangt als Eingabeparameter eine Sequenz von *non-Tupel*-Werten e_2, eine Attributbezeichnung a und die Tupelsequenz e_1. Die Sequenz von *non-Tupel*-Werten wird in eine Sequenz von Tupeln überführt, die alle nur das eine spezifizierte Attribut a besitzen mit jeweils einem Wert der *non-Tupel*-Wertsequenz. Jedem Tupel t_i der Sequenz e_1 wird nun durch eine *Map*-Operation ein neues Attribut hinzugefügt, dass den Wert $e_2(t_i)$ erhält. Durch eine anschließende *Unnest*-Operation wird eben dieses neu angelegte Attribut ausgepackt.

Formale Notation des Unnest-Map-Operators: $\Upsilon_{a:e_2}(e_1)$

Gruppieren. Der Gruppierungsoperator Γ wird auf eine Eingabesequenz e_1 angewandt. Die übrigen Eingabeparameter sind ein Attributbezeichner g, das Selektionskriterium $A_1 \theta A_2$, eine Funktion f und der Ausdruck e_2. Er arbeitet dabei die Sequenz beginnend beim ersten Tupel ab und fügt jedem dieser Tupel ein Attribut g hinzu. Der Wert von g wird von der Funktion G – angewandt auf das betrachtete Tupel – bestimmt. $G(x)$ ist dabei definiert als das Resultat der Funktion f angewandt auf die Tupelsequenz all jener Tupel von e_2, die das Prädikat $x|_{A_1} \theta A_2$ erfüllen.

Formale Notation der Gruppierung: $e_1 \Gamma_{g;A_1 \theta A_2;f} e_2$

Aggregation. Der Aggregationsoperator $\mathfrak{A}$ wird auf eine Tupelsequenz e angewandt und erzeugt eine einelementige Ausgabesequenz, deren Tupel nur das Attribut a mit dem Wert $f(e)$ enthält. Attributbezeichner a und Aggregationsfunktion f sind Eingabeparameter.

Formale Notation der Aggregation: $\mathfrak{A}_{a;f}(e)$

Sortieren. Um eine Sequenz e anhand eines Attributes a aufsteigend zu sortieren, wird für jedes betrachtete Tupel, beginnend beim ersten, die restliche Sequenz durch Selektion aller Tupel mit kleinerem Attributwert für a bzw. mit gleichem oder größerem Attributwert für a in zwei Teilsequenzen aufgespalten. Das aktuell betrachtete Tupel steht in der Ausgabesequenz dann zwischen beiden Teilsequenzen. Die Teilsequenzen werden jeweils auf die gleiche Weise wieder anhand eines ihrer Tupel in weitere Teilsequenzen unterteilt bis diese vollständig in einelementige Sequenzen unterteilt sind. Die Tupel liegen dann in der Ausgabe in aufsteigender Reihenfolge vor. Der Operator wird mit *Sort* bezeichnet.

Formale Notation der Sortierung: $Sort_a(e)$

Singleton Scan. Der *Singleton-Scan* $\Box$ gibt eine einelementige Sequenz mit dem leeren Tupel zurück. Er dient der Erzeugung eines *neuen* Tupels, das dann mittels *Map*-Operator mit Attributen und Werten belegt werden kann.

Formale Notation des Singleton Scan: $\Box$

Von XQuery nach NAL. In diesem Abschnitt wird erläutert, wie XQuery-Ausdrücke auf Operatoren der Algebra NAL abgebildet werden.

Als Beispiel für alle drei vorgestellten Algebren dient die Anfrage zur Ermittlung aller im Bibliotheksbestand verzeichneten, mindestens 100 Seiten umfassenden Bücher mit Titel, Autor und dessen Lebenslauf, alphabetisch sortiert. Denn hierbei werden sowohl der Join, die Aggregation, die Selektion und die Projektion als auch die Sortierung benötigt.

Da aber die von May, Helmer und Moerkotte [5] gegebene Übersetzungsfunktion T keine *Oder-By*-Statements übersetzt, wird das Beispiel um diesen Ausdruck reduziert.

```
FOR $autor IN document("autoren.xml")//author
FOR $buch IN document("bestand.xml")//book
LET $cv := $autor/curriculum_vitae
WHERE count($buch//page) >= 100
    AND $autor/authored/title/text() = $buch/title/text()

RETURN <book title={$buch/title/text()}>
           <author name={$autor/name/text()}>
               <curriculum_vitae>{$cv}</curriculum_vitae>
           </author>
       </book>
```

Zunächst wird der FLWR-Ausdruck in eine kanonische Form gebracht, dies erfordert die besagte Übersetzungsfunktion T. Der *Return*-Teil wird dabei in mehrere *Let*-Statements umgeschrieben, so dass die Rückgabe letztendlich durch eine einzige Variable repräsentiert wird.

```
FOR $autor IN document("autoren.xml")//author
FOR $buch IN document("bestand.xml")//book
LET $cv := $autor/curriculum_vitae
LET $titel := $buch/title/text()                                  # neu
LET $name := $author/name/text()                                  # neu
LET $cv_elem := <curriculum_vitae>{$cv}</curriculum_vitae>  # neu
LET $autor_elem := <author name=$name>{$cv_elem}</author>   # neu
LET $buch_elem := <book title=$titel>{$autor_elem}</book>   # neu
WHERE count($buch//page) >= 100
    AND $autor/authored/title/text() = $buch/title/text()

RETURN $buch_elem                                                 # neu
```

Die Übersetzungsfunktion beginnt mit dem ersten Statement und der leeren Sequenz $\Box$. Für alle weiteren Übersetzungsschritte wird immer das nächste Query-Statement übersetzt und der bisher erzeugte Algebra-Ausdruck als Eingabe für den neu erzeugten Operator verwendet. Der besseren Lesbarbeit halber werden die Algebra-Ausdrücke mit e_i bezeichnet, für den i-ten Übersetzungsschritt, der sie erzeugt.

Das erste For-Statement wird in einen Unnest-Map-Operator überführt:

$$e_1 := \Upsilon_{autor:document(\text{“}autoren.xml\text{''})//author}(\Box)$$

Ebenso das zweite For-Statement, jedoch jetzt mit dem Ausdruck e_1 als Eingabe:

$$e_2 := \Upsilon_{buch:document(\text{“}bestand.xml\text{''})//book}(e_1)$$

Let-Statements werden auf Map-Operatoren abgebildet.

$$e_3 := \chi_{cv:autor/curriculum\_vitae}(e_2)$$

$$e_4 := \chi_{titel:buch/title/text()}(e_3)$$

$$e_5 := \chi_{name:autor/name/text()}(e_4)$$

Die Variablen, die XML-Elemente repräsentieren, erfordern die Konstruktion eines solchen Elements, hier mit $C(elem, tag[, attribut], inhalt)$ bezeichnet. Der optionale Parameter *attribut* ist eine für das Beispiel eingeführte Erweiterung der in [5] verwendeten Funktion C.

$$e_6 := \chi_{cv\_elem:C(elem,curriculum\_vitae,cv)}(e_5)$$

$$e_7 := \chi_{autor\_elem:C(elem,author,name,cv\_elem)}(e_6)$$

$$e_8 := \chi_{buch\_elem:C(elem,book,titel,autor\_elem)}(e_7)$$

Eine Selektion stellt sicher, dass das Prädikat der *Where*-Klausel von allen resultierenden Tupeln erfüllt wird.

$$e_9 := \sigma_{count(buch//page)\geq 100 AND autor/authored/title/text()=buch/title/text()}(e_8)$$

Der aufgrund der Query-Umformung vereinfachte *Return*-Teil kann jetzt durch eine einfache Projektion abgebildet werden:

$$e_{10} := \Pi_{buch\_elem}(e_9)$$

Die vollständige Übersetzung der ursprünglichen Anfrage ist also der Ausdruck e_{10}. Ein so umgeformter algebraischer Ausdruck kann jetzt mit Verfahren algebraischer Optimierung weiter umgeformt werden.

2.2 Baumbasierte XML-Algebren

Als Beispiel zur Erläuterung der Eigenschaften und Funktionsweise von baumbasierten XML-Algebren wird hier TAX verwendet. TAX wurde von Jagadish et al. [3] als Algebra vorgestellt, deren Anwendung zwar stark an relationale Algebra erinnert, da diese Pate stand – das gegebene Arsenal an Operatoren mit *Selektion*, *Projektion*, *Join* und *Aggregation* daher altbekannt ist – die jedoch auf einem völlig anderen Datenmodell arbeitet: auf Bäumen. Zur Änderung der Baumstruktur wurden zusätzliche Operatoren eingeführt. Die Operatoren von *TAX* werden im Anschluss an das Datenmodell vorgestellt.

Datenmodell. Der Algebra liegt eine Menge von Bäumen oder *Data Trees* zu Grunde, keine Relationen gefüllt mit Tupeln. Ein Beispiel ist in Abbildung 1 gezeigt. Ein solcher Baum fungiert als Gegenstück zum relationalen Tupel. Aufgrund der Heterogenität von XML-Elementen eignen sich die Knoten eines solchen Baumes nicht als Gegenstück zum Tupel. Jeder Knoten repräsentiert ein XML-Element, XML-Attribute dieses Elements sind seine Kindknoten und XML-Subelemente hängen als (Teil-)Baum unter diesem Knoten mit der Wurzel dieses Teilbaumes als direktes Kind des ursprünglich betrachteten Knotens. Jeder Knoten besitzt darüberhinaus ein Attribut `tag`, das den Typ des Elements angibt und ein Attribut `pedigree` (engl. für Stammbaum) das die Knotenherkunft angibt. Die `pedigrees` entstammen einer total geordneten Menge, beispielsweise den Natürlichen Zahlen; somit ist ein Sortieren anhand dieses Wertes möglich. Die Knotenreihenfolge ist relevant, da diese der Reihenfolge von Elementen in einem XML-Dokument entspricht. Schließlich ist die Reihenfolge der Absätze eines Artikels gewollt und nicht beliebig. Allerdings wahrt nicht jeder Operator die gegebene Reihenfolge.

Pattern Tree. Eines der wichtigsten Konzepte in baumbasierten Algebren sind *Pattern Trees*. Hiermit wird ein Baum bezeichnet, der als Muster dient. Diese Musterbäume dienen als Eingabe für die Operatoren der Algebra und erlauben den Zugriff auf einzelne Knoten, da ihre Knoten, wie in Abbildung 2 zu sehen ist, mit eindeutigen Bezeichnern versehen sind. Aus der zu Grunde liegenden Datenbasis werden all jene Teilbäume ausgewählt, die dem Musterbaum entsprechen. Dabei muss die Übereinstimmung sowohl in der Struktur

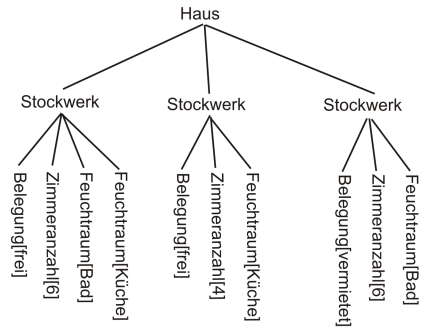

Abbildung 1. Beispiel eines *Data Tree*.

als auch in den spezifizierten Knoteneigenschaften gegeben sein. Ersteres bedeutet, alle im Musterbaum spezifizierten Parent-Child-Beziehungen bzw. Ancestor-Descendant-Beziehungen müssen auch im gefundenen Teilbaum herrschen. Jede Kante eines *Pattern Trees* trägt daher die Markierung *pc* wenn sie für eine Vater-Kind-Beziehung zwischen den beiden verbundenen Knoten steht oder die Markierung *ad* bei einer Ancestor-Descendant-Beziehung. Die spezifizierten Knoteneigenschaften sind durch eine Prädikatliste, die zu einem jeden *Pattern Tree* gehört, festgelegt. Die Prädikatliste kann dabei auch die leere Menge sein, dann wird lediglich anhand der Struktur eine Auswahl aus der Datenbasis getroffen.

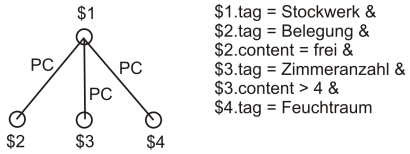

Abbildung 2. Beispiel eines *Pattern Tree*.

Witness Tree. Als *Witness Tree* werden diejenigen (Teil-)Bäume bezeichnet, die durch ein „Matching“ mithilfe eines *Pattern Tree* aus der Datenbasis ermittelt werden. Die *Witness Trees* sind daher konkrete Instanzen des *Pattern Tree*. Sie können sich überlappen und müssen nicht einmal die gleiche Struktur haben, da die AD-Beziehung Knoten beliebiger Tiefe zulässt. Wobei auch mehrere *Pattern-Tree*-Knoten auf ein und denselben konkreten Knoten eines *Witness-Tree* abgebildet werden können. Allen gemein ist, dass zu jedem *Pattern-Tree*-Knoten ein Pendant in jedem *Witness-Tree* existiert und dieses Pendant durch

den Musterbaum einen eindeutigen Bezeichner für die laufende Operation (in deren Kontext das Matching ausgeführt wurde) erhalten hat. Die Knotenreihenfolge im Musterbaum wird bei der Abbildung auf einen *Witness Tree* ignoriert. Ist die Knotenreihenfolge ein Auswahlkriterium, so muss ein entsprechendes Prädikat in der Prädikatliste enthalten sein, z.B. $2 `BEFORE` $3. Ein Beispiel zweier *Witness Trees* ist in Abbildung 3 zu sehen.

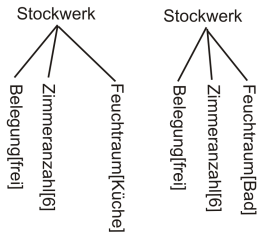

Abbildung 3. Beispiel zweier *Witness Trees*, die durch eine Selektion mit dem *Pattern Tree* aus der Datenbasis gewonnen wurden.

Data Tree. Wie in der relationalen Algebra, deren Operatoren auf Relationen arbeiten und deren Ergebnis jeweils wieder eine Relation ist, arbeiten die *TAX*-Operatoren auf *Data-Tree*-Kollektionen. Ein *Data Tree* bezeichnet einen konkreten (Teil-)Baum des XML-Dokumentes. Bei einer Abbildung eines *Pattern Tree* auf einen *Data Tree* kann dieser konkrete Baum in mehreren verschiedenen *Witness Trees* resultieren, da der Musterbaum zwar auf sich überlappende aber trozdem verschiedene Teilbäume passt.

Source Tree. *Data Trees* der Datenbasis, deren Knoten sich zumindest zum Teil in einem *Witness Tree* wiederfinden, werden als *Source Trees* dieser *Witness Trees* bezeichnet.

Tree-Value-Funktion. Tree-Value-Funktionen bilden *Data Trees* auf eine totalgeordnete Menge ab, z.B. auf die natürlichen Zahlen. Sie ermöglichen damit die Einführung von Gruppierungsfunktionen oder Sortierfunktionen.

Selektion. Mit Hilfe der Selektion σ lassen sich all jene Teilbäume aus der Datenbasis auswählen, die eine gewünschte Struktur und gewünschte Attributwerte aufweisen. Der weiter oben vorgestellte *Pattern Tree* stellt hierbei das Auswahlkriterium dar. Das Ergebnis der Operation enthält also konkrete Bäume und konkrete Attributwerte. Damit ist hier ein Bruch in der bisher beobachteten Analogie zur relationalen Algebra zu erkennen, da diese das gesamte qualifizierte Tupel auswählt, während hier nicht der *Source Tree* (der ja das Äquivalent

zum Tupel ist) sondern nur ein Teilbaum ausgewählt wird und den Rückgabewert darstellt. Diese Entscheidung wird von den Autoren dadurch erklärt, dass sich in einem *Data Tree* mehrere Elemente mit gleichem `tag` befinden können, von denen einige, aber nicht alle, die Selektionskriterien erfüllen und somit der Rückgabebaum keinen Rückschluss darauf zulässt, welches der Elemente jetzt das Gesuchte ist. Der Selektionsoperator hat als Eingabe eine Kollektion von *Data Trees*, einen *Pattern Tree* und eine Liste SL. Die Ausgabe ist wiederum eine Kollektion von Bäumen. Jeder Teilbaum, auf den der *Pattern Tree* passt, wird als *Witness Tree* in die Ausgabe mit aufgenommen. Ist die Liste SL nicht leer, sondern enthält Knotenbezeichner des *Pattern Tree*, dann finden sich auch alle Descendant-Knoten des spezifizierten Knotens in der Ausgabe. Die Knotenreihenfolge bleibt erhalten.

Grundsätzlich können also aus *einem Data Tree* der Eingabe n Bäume der Ausgabe resultieren. Möchte man dies verhindern, muss man einen Wurzelknoten \$1 in den *Pattern Tree* einfügen, der eine AD-Beziehung zur vorherigen Wurzel des *Pattern Tree* besitzt, erweitert die Prädikatliste um *isRoot(\$1)* und fügt der Liste SL den Knoten \$1 hinzu.

Projektion. Eine Projektion π blendet alle außer den im *Pattern Tree* spezifizierten Knoten aus, dabei werden Verwandschaftsbeziehungen erhalten. Fällt also ein Vaterknoten weg, so ist im Ergebnis der Projektion an dieser Stelle eine Kante, die den Kindknoten mit dem Vaterknoten des weggefallenen Knotens verbindet. Als Eingabe erhält der Operator eine Kollektion von *Data Trees*, einen *Pattern Tree* und eine Liste PL. Diese Liste kann Knotenbezeichner enthalten, die im *Pattern Tree* auftauchen. Diese können in der Liste mit einem $*$ markiert sein. Das Ergebnis beinhaltet diejenigen Teilbäume, die durch den *Pattern Tree* identifiziert wurden und bestehen nur aus den Knoten, die in der Projektionsliste PL enthalten sind. Darüber hinaus werden alle Descendant-Knoten in das Ergebnis übernommen, die von einem $*$-markierten Knoten abstammen. Die Knotenreihenfolge bleibt erhalten. In wievielen *Witness Trees* ein *Data Tree* der Eingabe resultiert, hängt davon ab, auf wieviele seiner Teilbäume der *Pattern Tree* passt und welche der Knoten in der Projektionsliste enthalten sind. Besteht nämlich zwischen diesen Knoten keine AD-Beziehung, dann werden die Teilbäume im Ergebnis getrennt aufgeführt. Will man sicherstellen, dass jeder Baum der Eingabe höchstens einen Baum im Ergebnis erzeugt, so muss man den Wurzelknoten des *Pattern Tree* in die Projektionsliste aufnehmen und ein Prädikat, welches fordert, dass dieser Knoten ein Wurzelknoten im *Data Tree* ist, der Prädikatliste hinzufügen. Die Projektion erzeugt nicht für jeden Treffer auf einen Teilbaum einen neuen *Witness Tree*, sondern erhält lediglich die Knoten eines Baumes, die auf einen *Pattern-Tree*-Knoten passen und durch die Projektionsliste erfasst sind.

Produkt. Das Kreuzprodukt $C \times D$ bildet aus zwei Kollektionen C und D von *Data Trees* eine Kollektion von Bäumen dergestalt, dass deren Wurzelknoten „künstliche“ Knoten mit dem `tag`-Namen `tax_product_root` sind. Diese künst-

lichen Knoten haben einem *null*-Wert für den `pedigree` und keine weiteren Attribute.

Das linke Kind des Baumes ist der Wurzelknoten eines Baumes der ersten *Data-Tree*-Kollektion C, das rechte Kind die Wurzel eines Baumes der zweiten *Data-Tree*-Kollektion D. Der linke und rechte Teilbaum der künstlichen Wurzel ist dabei jeweils der vollständige ursprüngliche *Data Tree* aus C bzw. D. Das Ergebnis enthält alle möglichen Paare, die sich aus jeweils einem Element aus C und einem Element aus D bilden lassen. Die Reihenfolge der Eingabe spielt demnach eine Rolle: $C \times D$ ist nicht das gleiche wie $D \times C$.

Join. Zwei Kollektionen von Bäumen lassen sich mit dem *Join*-Operator $\bowtie$ zu einer neuen Kollektion zusammenfassen. Zunächst wird mit jeder der beiden Kollektionen eine Selektion oder eine Projektion durchgeführt. Die durchgeführte Operation muss dabei nicht die gleiche für die beiden Eingabe-Kollektionen sein. Das bedeutet, als nötige Eingabe für den *Join*-Operator muss ein *Pattern Tree* P_1, eine Kollektion von *Data Trees* C_1 sowie eine Liste L_1, entweder Projektionsliste oder Selektionsliste, angegeben werden, um eine Kollektion von resultierenden *Witness Trees* der linken Seite zu erhalten. Um auch eine Kollektion von *Witness Trees* der rechten Seite zu erhalten, muss dementsprechend die Eingabe zusätzlich einen *Pattern Tree* p_2, eine *Data-Tree*-Kollektion C_2 und eine Liste L_2 – entweder Projektions- oder Selektionsliste – enthalten. Im Anschluss an die durchgeführte Operation o_1 und o_2 erhält man die linke Kollektion von *Witness Trees* W_1 und das Resultat der rechten Seite W_2. Es wird für jeden *Witness Tree* aus W_1 und für jeden *Witness Tree* aus W_2, die die *Join*-Bedingung $\$i.attr1 = \$j.attr2$ erfüllen, ein Baum der Ausgabe hinzugefügt, dessen Wurzel den `tag` „newTag“ trägt, dessen linker Kindknoten der Wurzelknoten des *Witness Trees* aus W_1 ist und dessen rechter Kindknoten der Wurzelknoten des *Witness Trees* aus W_2 ist.

Outer Join. Stellvertretend für die *Outer-Joins* wird hier nur der *Left-Outer-Join* vorgestellt. Beim *Left-Outer-Join* wird ebenfalls für beide Seiten die Selektion bzw. Projektion durchgeführt. Die Ausgabe enthält auch all die Bäume, die durch das oben beschriebene *Join*-Verfahren gebildet werden können, darüber hinaus enthält sie aber auch für jeden Baum aus W_1, zu dem kein *Join*-Partner aus W_2 gefunden wurde, einen Baum mit *newTag*-Wurzel deren linker und einziger Kindknoten die Wurzel des betroffenen Baumes aus W_1 ist.

Mengenoperationen. Zur Definition der Mengenoperationen ist zunächst eine Definition der Gleichheit zweier Bäume nötig.

Definition 1. *Zwei Bäume T_1 und T_2 heißen gleich, wenn zu jedem Knoten aus T_1 ein Knoten in T_2 existiert, der den gleichen `tag`, den gleichen `pedigree` und exakt den gleichen Knoteninhalt besitzt, diese Beziehung auch in der Gegenrichtung gilt, also auch zu jedem Knoten aus T_2 ein Knoten in T_1 existiert, so dass diese Eigenschaft erfüllt ist, die Reihenfolge dieser Knoten die gleiche ist und*

die Verwandschaftsbeziehungen zwischen den Knoten in beiden Bäumen gleich sind.

Mit Hilfe dieser Gleichheitsdefinition lassen sich Durchschnitt, Vereinigung und Differenz definieren:

Durchschnitt. Der Durchschnitt zweier Kollektionen von Bäumen C_1 und C_2 sind all jene Bäume aus C_1 für die ein identischer Baum in C_2 existiert.

Vereinigung. Die Vereinigung zweier Kollektionen von Bäumen C_1 und C_2 ist eine Kollektion, die alle Bäume aus C_1 und alle Bäume aus C_2 enthält.

Differenz. Die Differenz zweier Kollektionen von Bäumen C_1 und C_2 ist eine Kollektion von Bäumen aus C_1 für die kein identischer Baum in C_2 existiert.

Gruppierung. Bei der Gruppierung wird eine Eingabemenge von Bäumen nach einem Gruppierungskriterium aufgeteilt. Diese Aufteilung muss nicht disjunkt sein. Der Gruppierungoperator γ nimmt als Eingabe die Kollektion der Eingabebäume, einen *Pattern Tree*, eine Gruppierungsfunktion und eine Ordnungsfunktion entgegen. Die Gruppierung erfolgt dann dergestalt, dass zuerst die Menge der *Witness Trees* W durch eine Abbildung des *Pattern Tree* auf die Menge der Eingabebäume gebildet wird. Die Gruppierungsfunktion unterteilt die Menge W in Partitionen. Diese Funktion kann aus einer Liste von Knotenbezeichnern des *Patter Tree* zusammen mit Bedingungen für deren Attributbelegungen bestehen, die je nach Erfüllung dieser Bedingungen den *Witness Tree* der einen oder der anderen Partition zuweist. Standardmäßig wird hierbei nur überprüft, ob der in der Liste angegebene Knoten des *Pattern Tree* mit dem konkreten Knoten eines *Data Tree* übereinstimmt, nicht aber die Übereinstimmung seiner Kindknoten. Letzteres wird nur dann durchgeführt, wenn der Knoten in der Liste mit einem $*$ markiert ist.

Für jede der so erzielten Partitionen gibt es einen Baum in der Ausgabe dessen Wurzel ein „künstlicher“ Knoten ist (*null*-Wert als `pedigree`), der den Tag `tax_group_root` trägt, sowie zwei Kindknoten besitzt: Der linke Kindknoten ist ebenfalls ein „künstlicher“ Knoten mit Tag `tax_grouping_basis` (einem *null*-`pedigree`) und bildet die Wurzel für einen Baum der die *grouping basis* ist. Der rechte Kindknoten besitzt den Tag `tax_group_subroot` (ebenfalls mit *null*-`pedigree`) und als Kindknoten die Wurzelknoten aller *Source Trees* die einen *Witness Tree* in der betreffenden Partition haben. Aufsteigend sortiert durch die Ordnungsfunktion, die jedem *Witness Tree* einen Wert zuweist, der die Stelle angibt, an der der zugehörige *Source Tree* steht. Ein *Source Tree* kann also in verschiedenen Gruppen der Ausgabe vertreten sein. Um die Ausgabe aussagekräftiger zu gestalten, wird unter dem Knoten `tax_grouping_basis` ein Baum gehängt, der durch eine Projektion der *Source Trees* entstand. Die Projektion nimmt den *Pattern Tree* als Eingabe. Die Gruppierungsfunktion in Listenform dient dabei als Projektionsliste. Per Definition müssen innerhalb einer Gruppe all diese Projektionen bis auf ihr `pedigree` identisch sein [3].

Verwendung im aus der Projektion resultierenden Baum findet der kleinste `pedigree`-Wert für einen jeden Knoten. Wenn mehrere Bäume aus der Projektion hervorgehen, dann hat der `tax_grouping_basis`-Knoten mehrere Kindknoten, deren Reihenfolge der Reihenfolge im *Pattern Tree* entspricht.

Duplikat-Eliminierung anhand von Attributwerten. Als Duplikate werden Knoten identischen Inhalts betrachtet, d. h. sie müssen auch den gleichen `pedigree` besitzen. *TAX* bietet darüber hinaus noch eine Funktion, die eine Duplikat-Eliminierung durchführt ohne sich von abweichenden `pedigree`-Werten stören zu lassen. Die Autoren begründen dies damit, dass beispielsweise der *distinct*-Operator aus XQuery eine solche Funktion benötige.

Sortieren. Die Gruppierungsfunktion sortiert *Source Trees* innerhalb einer Gruppe. Stellt man durch eine leere *grouping basis* sicher, dass alle *Source Trees* der Eingabe in die gleiche Gruppe übertragen werden, erhält man eine Sortierung der Eingabe. Durch Projektion lassen sich die Knoten `tax_group_root` und `tax_grouping_basis` aus dem Ergebnis entfernen.

Aggregation. Die aus der relationalen Algebra bekannten Aggregationsfunktionen `min`, `max`, `count`, `sum`, `avg` etc. sind auch in *TAX* verfügbar. Hier ist jedoch wichtig, dass der Knoten des Baumes, der den aggregierten Wert zu tragen hat, spezifiziert wird. Als Eingabe für den Operator **A** wird eine Kollektion von Bäumen, ein *Pattern Tree*, eine Aggregationsfunktion f sowie eine Update-Spezifikation benötigt. Letztere gibt die Position an, an der in sämtlichen Bäumen der Ausgabe ein neuer Knoten (mit *null*-`pedigree`) enthalten ist, welcher den Tag *tax_aggNode* und ein Attribut, dessen Name Teil der Eingabe des Aggregationsoperators ist, besitzt, und dessen Wert durch die Aggregationsfunktion f ermittelt wird. In der Ausgabe findet sich für jeden Baum der Eingabe ein, um den das Aggregations-Ergebnis tragenden Knoten, erweiterter Baum mit sonst identischer Struktur und Knoteninhalten. Möchte man nun alle Bäume erhalten, deren Knoten tatsächlich den mit der Aggregationsfunktion ermittelten Wert besitzen, beispielsweise das Extremum über alle vorhandenen Werte, so können diese durch eine einfache Selektion ermittelt werden.

Umbenennung. Bei der Umbenennung von Knotentags oder -Attributen wird folgendermaßen vorgegangen: Der Operator ρ erhält eine Kollektion von Bäumen, einen *Pattern Tree* und eine Umbenennungsspezifikation RS, deren Einträge aus einem im *Pattern Tree* vorkommenden Knotenbezeichner, gefolgt vom alten Tagname bzw. Attributnamen und neuem Tagname bzw. Attributnamen bestehen. Die Umbenennung geschützter Attributnamen wie `tag` oder `pedigree` ist nicht möglich. Bei der Umbenennung werden die Bäume der Eingabe zunächst auf mit dem *Pattern Tree* übereinstimmende Teilbäume durchsucht, Knoten, die dabei einem *Pattern-Tree*-Knoten entsprechen, dessen Knotenbezeichner in der Umbenennungsspezifikation vorkommen, werden beim ersten Durchgang mit

dem Index des entsprechenden Knotens markiert; beispielsweise 1, falls der Knoten dem Knoten *$1* des *Pattern Trees* entspricht. Im Anschluss daran werden die mit einer solchen Markierung versehenen Knoten dahingehend überprüft, ob die in der Umbenennungsspezifikation aufgelistete Umbenennungsregel angewandt werden kann; trägt also der mit i markierte Knoten ein Attribut oder einen Tag mit dem namen *oldName* und die Regel $\$i : oldName \leftarrow newName$ ist in der Liste, dann wird die Umbenennung durchgeführt. Damit wird verhindert, dass es zu einem kaskadieren der Umbenennung kommt. Lassen sich mehrere *Pattern-Tree*-Knoten auf einen konkreten Knoten abbilden, so trägt dieser Knoten mehrere Markierungen. Dies führt solange zu keinem nicht eindeutigen Zustand bis zwei Regeln für zwei dieser *Pattern-Tree*-Knoten den gleichen Tagname bzw. den gleichen Attributnamen umbenennen. In diesem Fall bleibt es der *TAX*-Implementierung überlassen, welche der Regeln zum Einsatz kommt.

Umsortieren der Knotenreihenfolge. Der *Reorder*-Operator ϱ erhält als Eingabe eine Kollektion von Bäumen, einen *Pattern Tree*, sowie eine Tree-Value-Funktion und eine Reorderlist RL.

Die Kindknoten und untergeordneten Teilbäume eines Knotens der Reorderliste werden durch die Tree-Value-Funktion, die jedem dieser Knoten einen Wert zuweist, in eine aufsteigende Reihenfolge gebracht. Dabei ist zu beobachten, dass für den Fall, dass die Kindknoten mehrerer Knoten umsortiert werden sollen, zuerst die Knoten, die am tiefsten im Baum liegen, umsortiert werden. Die Autoren geben ein Beispiel, das zeigt, dass bei Nichtbeachtung der letztgenannten Regel mehrmaliges Anwenden des *Reorder*-Operators mehrfaches Umsortieren bewirkt, was (bei einer idempotenten Funktion) nicht sein darf.

Copy and Paste. Der Operator κ prüft jeden Baum der Eingabemenge C auf Übereinstimmung mit dem *Pattern Tree* P, erzeugt für jeden so gefundenen *Witness Tree* eine Menge aller möglichen Paare: bestehend aus Knotenbezeichner der Copylist und einem Knoten der Update-Spezifikation US. Diese Paarmenge wird in Partitionen mit gleichen US-Knoten unterteilt. Jede dieser Partitionen gibt damit alle Einfügungen an, die an einem bestimmten Knoten erfolgen sollen. Die Copylist-Knoten werden nach `pedigree` geordnet, dann wird ein neuer Knoten an der spezifizierten Stelle mit dem Tag *tax_cnp_group* eingefügt. Unter diesen neuen Knoten wird nun jeweils eine Kopie der Knoten, die auf die Knoten der Copylist mappen, gehängt. Ist einer dieser Copylist-Knoten mit $*$ markiert wird der gesamte Teilbaum dieses Knotens mitkopiert.

Value Update. Um die Attributwerte einzelner Knoten zu ändern, ist ein *Pattern Tree* nötig, der die betreffenden Knoten identifiziert. Ferner die eigentliche Update-Spezifikation US, deren Listeneinträge aus Knotenbezeichner, Attributname und neuem Attributwert bestehen. Der in einem Listeneintrag aufgeführte neue Attributwert kann dabei auch in Form eines arithmetischen Ausdrucks bestehen, der Variablen enthält. Dies ist von Nöten, wenn sich dieser z. B. aus den

Attributwerten anderer Knoten ergibt. Als letztes Element der Eingabe ist die Menge der zu aktualisierenden *Source Trees* an den Operator υ zu übergeben.

Der Vorgang ist wie bei der Umbenennung zweischrittig, um auch hier kaskadierende Änderungen zu verhindern. Die *Data-Tree*-Knoten werden mit dem Index der auf sie zutreffenden *Pattern-Tree*-Knoten markiert, anschließend werden den markierten Knoten die entsprechenden Konstanten bzw. das Ergebnis des ausgewerteten arithmetischen Ausdrucks der Update-Spezifikation als neuer Wert zugewiesen. Auch hier kann der Fall auftreten, dass ein Knotenattribut eines Knotens zwei verschiedene Änderungen erhalten soll, in diesem Fall ist es wieder der *TAX*-Implementierung überlassen, welche Regel zur Anwendung kommt.

Löschen von Knoten und Attributen. Die Identifikation der zu löschenden Knoten innerhalb der *Data Trees* wird über einen *Pattern Tree* erreicht. Eine Liste von Knotenbezeichnern, die Delete-Spezifikation DS, enthält die zu löschenden Knoten. Sind diese mit $*$ markiert, wird der gesamte Teilbaum dessen Wurzel sie sind, ebenfalls gelöscht. Der Operator wird mit δ bezeichnet. Die Durchführung ist abermals zweischrittig: Nach der Markierung der *Source-Tree*-Knoten mit den Indices passender *Patter-Tree*-Knoten werden die so markierten Knoten daraufhin untersucht, welcher DS-Listeneintrag anzuwenden ist. Wird nicht der gesamte Teilbaum gelöscht und der gelöschte Knoten hatte Kindknoten, so sind diese Kindknoten nun direkte Kindknoten des „Großvaters“, die Reihenfolge untereinander bleibt erhalten, ihre Position unter den anderen Kindknoten des Großvaters entspricht der des Gelöschten. Soll nur ein Knotenattribut gelöscht werden, nicht aber der gesamte Knoten, so muss der DS-Eintrag hinter dem Knotenbezeichner den betreffenden Attributnamen führen: $i.attrName

Einfügen von Knoten und Attributen. Die Position des oder der einzufügenden Knoten(s) wird über einen *Patter Tree* angegeben. Ferner ist nur das Einfügen von Blattknoten möglich, nachfolgende Operationen können dem neuen Knoten dann Kinder zuweisen. Die Einträge der Insert-Spezifikation IS unterscheiden sich in ihrer Struktur, je nachdem ob ein Attribut oder ein ganzer Knoten eingefügt wird. Erstere bestehen aus Knotenbezeichner gefolgt von Attributnamen und initialem Attributwert. Dabei ist das Einfügen bereits vorhandener Attribute nicht möglich. Einträge für das Einfügen eines ganzen Knotens werden von einem Schlüsselwort angeführt, das die Position relativ zu einem *Pattern-Tree*-Knoten angibt; beispielsweise *AfterLastChild* oder *NextSibling*. Der Knotenbezeichner folgt auf dieses Schlüsselwort. Der Ausdruck wird mit einem n-Tupel der Form $(attrName_1 = val_1, attrName_2 = val_2, ..., attrName_n = val_n)$ vervollständigt. Die Auswertung des Einfügeoperators ι ist aus den zuvor genannten Gründen zweischrittig.

Von XQuery nach TAX. In diesem Abschnitt wird erläutert, wie XQuery-Ausdrücke auf Operatoren der Algebra *TAX* abgebildet werden. Die Überset-

zung von XQuery-FLWOR-Ausdrücken in die *TAX*-Algebra folgt den von Jagadish, Lakshmanan, Srivastava und Thompson [3] angegebenen Schritten. Diese sind:

1. Identifizieren der Tree Patterns
2. TreePatterns um Prädikate erweitern
3. Eliminieren von Duplikaten
4. Auflösen der Aggregationen in Let-Statements
5. Bilden der Joins
6. verbleibende Prädikate
7. Auflösen der Aggregationen in Return-Statements
8. Sortieren
9. Das Ergebnis in Form bringen

Die Beispielanfrage zur Ermittlung aller Bücher mit mindetens 100 Seiten aus Abschnitt 2.1 soll auch hier die Übersetzung veranschaulichen.

Die Aggregation wird zunächst aus dem *Where*-Statement herausgezogen. Die XQuery-Anfrage lautet dann wie folgt:

```
FOR $autor IN document("autoren.xml")//author
FOR $buch IN document("bestand.xml")//book
LET $cv := $autor/curriculum_vitae
LET $page_count := count($buch//page)
WHERE $page_count >= 100
    AND $autor/authored/title/text() = $buch/title/text()
ORDER BY $buch/title Ascending

RETURN <book title={$buch/title/text()}>
           <author name={$autor/name/text()}>
               <curriculum_vitae>{$cv}</curriculum_vitae>
           </author>
       </book>
```

Im ersten Schritt werden Pattern Trees zu jedem For-Statement gebildet. Alle benötigten Knoten werden in die Patterns übernommen. Dabei werden Knoten für Variablen gebildet, sowie Zwischenknoten, die sich aus den Pfadausdrücken ergeben. Daraus ergibt sich hier je ein Pattern Tree pro *For*-Klausel. Die Patterns P_1 und P_2 in den Abbildungen 4 und 5 weisen aber schon Knoten auf, die erst in nachfolgenden Schritten ermittelt werden. Beide Bäume werden als Eingabe für eine Selektion $\sigma_{P_i,\{\$1\}}(C_i)$ auf den Kollektionen C_1 und C_2 verwendet. Erstere ist eine Kollektion von Bäumen aus *autoren.xml* und letztere eine Kollektion von Bäumen aus *bestand.xml*.

Im zweiten Schritt werden Prädikate aus der *Where*-Klausel in die Pattern Trees gezogen. Da sich die Aggregation im *Where*-Statement nur auf einen der Bäume, nämlich P_2, bezieht, kann die Auswerung dieses Prädikates in den PatternTree P_2 verlegt werden. Die Aggregation wird der betreffenden Selektion vorangestellt. Es dient also zunächst einmal die Kollektion C_2 (die Datenbasis

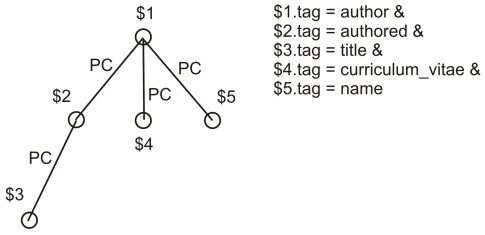

Abbildung 4. Pattern Tree P_1 mit Prädikatliste.

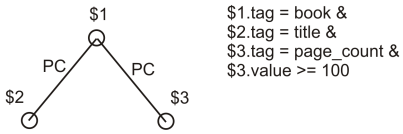

Abbildung 5. Pattern Tree P_2 mit Prädikatliste.

des Bücherbestandes) als Eingabe für die vorgezogene Aggregation, um im Anschluss daran die Selektion anhand des aggregierten Wertes zu ermöglichen. Ein zusätzlicher Pattern Tree ist nötig, um die Knoten, über die aggregiert werden soll, zu benennen.

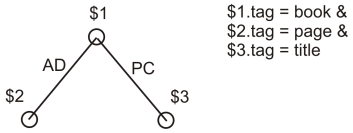

Abbildung 6. Pattern Tree P_3 mit Prädikatliste.

Pattern Tree P_3 weist die benötigten Knoten aus, zu beachten ist die *Ancestor-Descendant*-Kante im Baum, um dem Pfadausdruck zu entsprechen, da auch dort eine beliebige Schachtelungstiefe erlaubt ist. Das Ergebnis dieser Aggregation sei C_3.

$$C_3 := \mathbf{A}_{P_3, page\_count = count(\$2), afterLastChild(\$1)}(C_2)$$

Damit ist nicht länger C_2 Eingabe für die Selektion mit P_2, sondern C_3.

Der Dritte Schritt, die Eliminierung von Duplikaten, wird im Beispiel nicht benötigt. Der vierte Schritt, das Auswerten der Aggregation in *Let*-Klauseln, wurde bereits vorgezogen. Im fünften Schritt werden die Joins gebildet. Dazu werden Prädikate der *Where*-Klausel immer dann als *Join*-Bedingung eingesetzt, wenn sie sich auf Variablen zweier Pattern Trees aus den vorhergehenden Schritten beziehen. In unserem Beispiel genügt ein *Join*. Das einzig verbliebene Prädikat des *Where*-Statements prüft auf Gleichheit der Buchtitel und wird angewandt, sobald das Ergebnis der Selektion auf der linken und der rechten Seite

berechnet wurde. Als Wurzel wird der neue Knoten *join_root* verwendet. Die in der *Join*-Bedingung stehenden Variablen $1 beziehen sich auf unterschiedliche Knoten: die linke Seite bezieht sich auf Pattern Tree P_1, die rechte Seite auf Pattern Tree P_2. Das Ergebnis dieses *Joins* sei C_4.

$$C_4 := \sigma_{P_1,\{\$1\}}(C_1) \bowtie_{join\_root;\$3.text()=\$2.text()} \sigma_{P_2,\{\$1\}}(C_3)$$

Da keine weiteren verbleibenden Prädikate des *Where*-Statements vorliegen, führt der sechste Schritt zu keinen weiteren Operatoren. Ebenso Schritt sieben: Aggregationen innerhalb der *Return*-Klausel kommen im Beispiel nicht vor.

In Schritt acht werden *Order-By*-Statements ausgewertet. Dazu werden Gruppierungen eingeführt, die auf den im Return-Statement gelisteten Elementen operieren und deren *Tree-Value*-Funktionen die Reihenfolge festlegen. Daraus ergibt sich für das Beispiel ein Pattern P_4 (siehe Abbildung 7) für die Gruppierung mit leerer *grouping*-Liste und einer *Tree-Value*-Funktion, die die Elemente einer Gruppe lexikographisch sortiert. Das Ergebnis sei C_5.

$$C_5 := \gamma_{P_4,\{\},order\_lex(\$8)}(C_4)$$

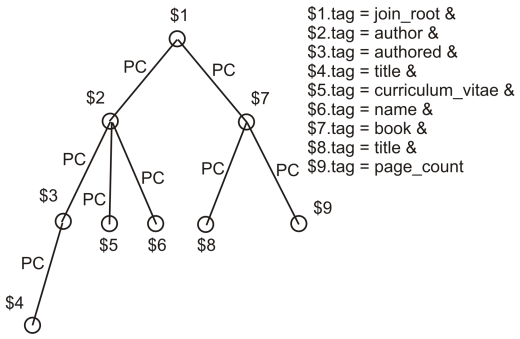

Abbildung 7. Pattern Tree P_4 mit Prädikatliste.

Schritt neun ist die Zusammenstellung des Ergebnisses. Dazu kann es nötig sein, dass Knoten per *Copy-and-Paste* umstrukturiert werden. Im Beispiel genügt eine Projektion, dabei bilden die im *Return*-Statement gelisteten Elemente die Projektionsliste. Die Projektion erfordert einen weiteren Pattern Tree P_5 wie in Abbildung 8 gezeigt.

Das Resultat der Projektion und damit der gesamten Anfrage ist dann C_6 mit:

$$C_6 := \pi_{P_5\{\$5*,\$8*\}}(C_5)$$

2.3 Hybride XML-Algebren

Jagadish et al. [7] zeigen mit ihrem Konzept der *Tree Logical Classes* (*TLC*) einen *Hybrid*: Die Operatoren arbeiten zwar auf Bäumen, die Knoten dieser

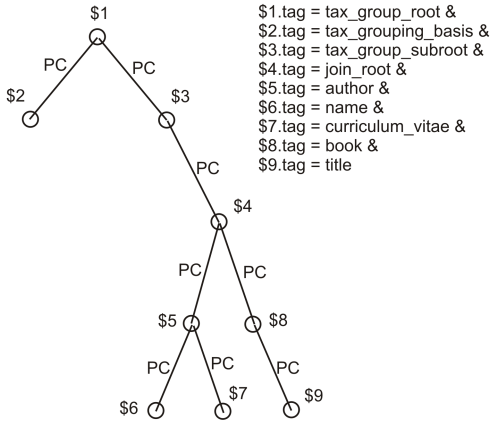

Abbildung 8. Pattern Tree P_5 mit Prädikatliste.

Bäume sind allerdings Mengen von XML-Elementen, und erinnern mehr an eine Sammlung von Tupeln, die sequenziell abzuarbeiten ist. Von den Autoren wird die Tatsache, dass XML-Elemente trotz gleichen Typs oftmals keine einheitliche Struktur besitzen, als Argument angeführt, um zu untermauern, dass bisherige XML-Algebren den an sie gestellten Anforderungen nicht oder nur ungenügend gerecht werden. Diese verlangten *Structural Clustering*, *Redundant Access* und *Redundant Tree Matching*. Andere hybride Ansätze beschreiben beispielsweise Michiels et al. [6].

Datenmodell: Annotated Pattern Tree. In der TLC Algebra werden *Annotated Pattern Trees* (APT) eingesetzt, um die gewünschten Elemente aus der Datenbasis zu ermitteln, diese weisen aber einige wichtige Erweiterungen zu herkömmlichen *Pattern Trees* auf: Zum einen die annotierten Kanten, zum anderen logische Knotenklassen, die es erlauben, heterogene Mengen von Bäumen wie homogene zu behandeln.

– **Kantenannotationen** Die Kanten des annotierten Pattern Trees tragen jeweils eine von vier möglichen Annotationen:
  - **„−“:** Für einen gefundenen Datenknoten, der auf den Vaterknoten der Kante passt, muss genau ein passender Kindknoten im betreffenden Datenbaum existieren, damit eine Übereinstimmung zwischen *Pattern Tree* und *Data Tree* vorliegt. Diese Annotation wird standardmäßig für alle nicht annotierten Kanten angenommen.
  - **„?“:** Eine Übereinstimmung zwischen *Pattern-* und *Data Tree* liegt genau dann vor, wenn für einen gefundenen Datenknoten, der auf den zur Kante gehörenden Vaterknoten abgebildet wird, kein oder ein Kindknoten im *Data Tree* existiert, der auf den zur Kante gehörenden Kindknoten passt.

  - **„+“:** Eine Übereinstimmung zwischen *Pattern-* und *Data Tree* liegt bei einem oder mehreren gefundenen Kindknoten des *Data Tree* für einen gefundenen Vaterknoten des *Data Tree* vor.
  - **„∗“:** Eine Übereinstimmung zwischen *Pattern-* und *Data Tree* liegt für jeden gefundenen Knoten vor, der auf den Vaterknoten des *Pattern Trees* passt. Die Anzahl der passenden Kindknoten kann beliebig sein.
- **Logische Klassen** Da durch die Annotationen die *Witness Trees* der Ergebnismenge eines *Pattern Tree Matching* sehr unterschiedlich sein können, stellt sich die Frage, wie Knoten in den *Witness Trees* adressiert werden können, um weitere Operationen auf oder mit ihnen durchführen zu können. Zu diesem Zweck wurden logische Klassen eingeführt. Diese entsprechen jeweils einer Menge von Knoten eines *Witness Tree*. Eine logische Klasse $LC(v)$ ist definiert als die Menge aller Knoten eines *Witness Tree*, auf die der *Pattern-Tree*-Knoten v abgebildet werden kann

Datenmodell: Annotated Construct Pattern Tree. Eine spezielle Form des APT ist der *Annotated Construct Pattern Tree*. Mit ihm ist es möglich beliebige Bäume zusammenzusetzen. Mit *Logical Class Label* versehene Knoten geben dabei an, wo im Resultat die in vorherigen Operationen ermittelten Knoten der *Data Trees* platziert oder wo neue Knoten eingefügt werden, mit welchem Knotentag und mit welchen Attributen. Die Umbenennung von Knoten ist hiermit auch möglich.

Datenmodell: Logical Class Reduction. Ein Baum der gleichen Kantenstruktur wie ein *Annotated Pattern Tree*, dessen Knoten logische Klassen sind, welche jeweils einem Knoten des *Pattern Trees* entsprechen, ist eine Logical Class Reduction oder LCR. Einer solchen logischen Klasse ist ein eindeutiges Klassenlabel zugeordnet. Jeder Knoten (eines Zwischenergebnisses) ist mindestens einer LC zugeordnet. Die *Logical Class Reductions* von *Witness Trees*, die aus dem gleichen *Pattern Tree Match* stammen, sind gleich.

Wiederverwendung von Pattern Tree Matches. Jeder Datenknoten, der nicht unmittelbar aus der Datenbasis gewonnen wird, sondern Ergebnis eines *Pattern-Tree*-Vergleichs ist, ist mindestens einer logischen Klasse zugeordnet. Diese Zugehörigkeit kann innerhalb logischer Ausdrücke (Prädikate) verwendet werden und erlaubt somit die Wiederverwendung vorheriger *PatternTree*-Vergleichsoperationen.

Filter. Der Filter-Operator erhält *Logical Class Label*, Prädikat P und Modus als Eingabeparameter und wird – so parametrisiert – auf eine Menge von Bäumen S angewandt. Die Ausgabe enthält all jene Bäume aus S, deren Knoten, die der spezifizierten *Logical Class* angehören, P erfüllen. Der Modus bestimmt dabei, ob alle Knoten der Klasse (allquantifiziert) oder mindestens einer (existenzquantifiziert) P erfüllen muss. Ein Modus, für den genau ein Knoten der

Klasse das Prädikat P erfüllen muss, ist auch möglich. Weitere Modi, die bestimmen, welche der Knoten der Klasse das Prädikat P erfüllen müssen damit der betreffende Baum in die Ausgabe übernommen wird, sind möglich.

Join. Angewandt auf zwei Mengen von Bäumen, mit einem APT und einem Prädikat als Parameter, erzeugt der *Join*-Operator eine Menge von zusammengesetzten Bäumen. Der APT gibt dabei die Struktur der Bäume der Ergebnismenge an. Daher müssen die Wurzelknoten der *Logical Class Reductions* der beteiligten Bäume im APT enthalten sein. Die Struktur besteht aus einem künstlichen Knoten und den Wurzeln der LCRs der Eingabebäume. Die Kante zur linken LCR-Wurzel ist mit einem „$-$“ annotiert, die rechte Seite ist beliebig annotiert. Das heißt, ein Baum der linken Eingabemenge kann mit mehreren Bäumen der rechten Eingabemenge zusammengelegt werden. Das Prädikat ist die *Join*-Bedingung (und verwendet *Logical Classes* um sich auf Knoten der Eingabebäume zu beziehen). Die dabei verwendeten LCs dürfen nicht mehrwertig sein.

Selektion. Als Eingabe der Selektion dient ein *Annotated Pattern Tree* und eine Menge von Bäumen. Der *Pattern Tree* wird auf jeden Eingabebaum abgebildet. Jeder so erzeugte *Witness Tree* wird in die Ausgabemenge übertragen.

Projektion. Projektion, angewandt auf eine Menge von Bäumen, erzeugt eine Ausgabemenge, deren Bäume nur all jene Knoten enthalten, die in der Liste nl – ein Parameter des Operators – aufgelistet sind. Diese Liste beinhaltet LCLs, daher können auch leere Klassen in dieser Liste sein. Ein Spezialfall ist eine Projektion, die auf Bäume der Datenbasis angewandt wird: in diesem Fall sind alle *Logical Classes* leer. Sollte die Projektion die Aufspaltung eines Eingabebaumes bewirken, wird auch der Wurzelknoten als verbindendes Element in die Ausgabe mitübernommen.

Duplikateliminierung. Als Eingabe dienen dem Operator eine Liste mit *Logical Class Labels*, ein Parameter ci, der bestimmt, ob zu eliminierende Duplikate anhand des Knoteninhalts oder seiner id ausgewählt werden und eine Menge S von Bäumen. Die verwendeten LCs dürfen nur einelementige Mengen sein.

Aggregation. Dem Aggregationsoperator sind als Parameter eine Aggregatfunktion, ein *Logical Class Label* sowie ein neues *Logical Class Label* zu übergeben. Angewandt auf eine Menge von Bäumen zieht der Operator all jene Knoten zur Berechnung des Ergebnisses heran, die durch das *Logical Class Label* identifiziert werden. Auf diese wird die Aggregatfunktion angewandt und das Ergebnis im Knoten *newLabel* festgehalten. Der Operator wiederholt dies für jeden Baum der Eingabemenge und erzeugt somit ebensoviele Bäume in der Ausgabemenge wie in der Eingabe enthalten sind. Diese tragen einen neuen *Logical-Class*-Knoten mit dem neuen *Label* auf der gleichen Hierarchiestufe und unter dem gleichen Vaterknoten wie die Knoten der LC_a.

Konstruktion. Die Eingabemenge S wird so vom Konstruktionsoperator in eine Ausgabemenge transformiert, dass deren Bäume eine neue Struktur, wie im ACPT c festgelegt, besitzt. Der Eingabeparameter c ist ein *Annotated Construct Pattern Tree*, kurz ACPT. Da sich der ACPT auf *Logical Class Labels* bezieht, ist ein der Konstruktion vorangehender Operator dafür verantwortlich, die Knoten der *Data Trees* solchen *Logical Classes* zuzuordnen, die andernfalls leer sind.

Von XQuery nach TLC. In diesem Abschnitt wird erläutert, wie XQuery-Ausdrücke auf Operatoren der Algebra TLC abgebildet werden. Die graphische Notation entspricht der von Jagadish et al. [7] vorgegebenen. Einfache Kanten zwischen zwei Knoten entsprechen einer *Parent-Child*-Beziehung und doppelte Kanten einer *Ancestor-Descendant*-Beziehung. *Default*-Kantenannotationen werden wie in der zu Grunde liegenden Arbeit der Übersichtlichkeithalber weggelassen.

Die zuvor verwendete XQuery-Anfrage dient auch hier zur Veranschaulichung der Übersetzungsregeln zur Erzeugung entsprechender Algebra-Ausdrücke.

```
FOR $autor IN document("autoren.xml")//author
FOR $buch IN document("bestand.xml")//book
LET $cv := $autor/curriculum_vitae
WHERE count($buch//page) >= 100
    AND $autor/authored/title/text() = $buch/title/text()
ORDER BY $buch/title Ascending

RETURN <book title={$buch/title/text()}>
           <author name={$autor/name/text()}>
               <curriculum_vitae>{$cv}</curriculum_vitae>
           </author>
       </book>
```

Von Jagadish et al. [7] wurde ein Algorithmus zur Erzeugung der Algebra-Pläne aus XQuery gegeben. Nach diesem Algorithmus werden zunächst für beide FOR-Statements APT erzeugt, die aus den im jeweiligen Pfadausdruck vorkommenden Knoten bestehen und Default-Kanten (d. h. mit „$-$“ annotiert sind) besitzen. Sie dienen als Parameter für jeweils einen Selektionsoperator. Für aufeinanderfolgende For-Statements wird ein Join-Operator erzeugt, der die Wurzelknoten der APTs in einen Baum fasst. Abbildung 9 zeigt die initialen APTs des Beispiels.

Für ein LET-Statement wird ebenfalls ein APT erzeugt, dessen Kanten mit einem „$*$“ annotiert sind. Da die Wurzel des APT keine Dokumentwurzel ist, wird er an den initialen APT, der diese Variablen enthält, angehängt. Abbildung 10 zeigt diese Erweiterung.

Für ein WHERE-Statement der im Beispiel verwendeten Form wird erst die linke Seite der Konjunktion, anschließend die rechte Seite übersetzt und schließlich zusammengefasst. Der Pfadausdruck im ersten Prädikat, in Verbindung mit

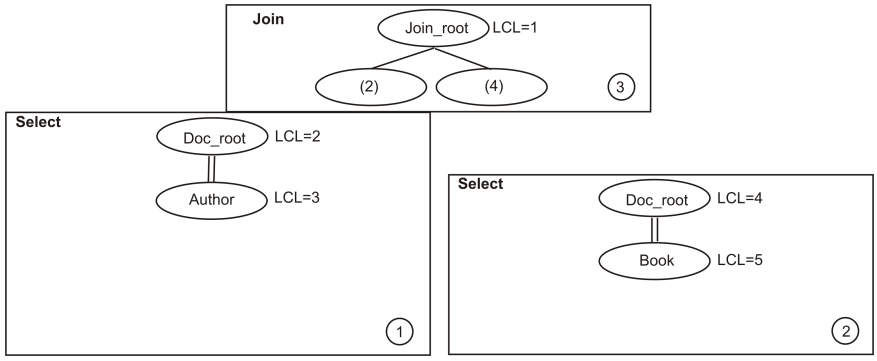

Abbildung 9. Initiale Selektion und Join.

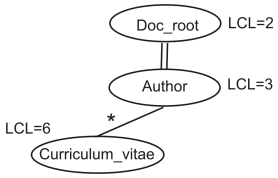

Abbildung 10. Um LET-Variable erweiterter APT.

der Aggregationsfunktion `count`, wird zu einem APT (mit „$*$“ annotiert) umgeformt und dem initialen Selektions-APT, der die Variable enthält, angehängt. Abbildung 11 zeigt diese Erweiterung.

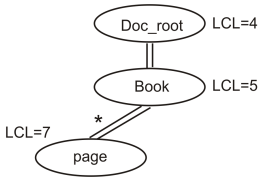

Abbildung 11. Um Aggregations-Variable erweiterter APT.

Zwei weitere Operatoren sind nötig, um das Ergebnis der Aggregation zu erzeugen. Zum einen wird ein Aggregationsoperator `count` angelegt. Zum anderen ein Filter-Operator.

Das zweite Prädikat erfordert einen Wertbasierten Verbund (Value Join): Für beide Seiten wird jeweils ein APT mit Default-Kanten angelegt und an die initialen Selektions-APTs angehängt. Der zugehörige Join-Operator kann jetzt um das Join-Prädikat `(9) = (10)` erweitert werden.

Der nachfolgende OrderBy-Ausdruck kann im Allgemeinen mehrere Pfadausdrücke beinhalten. Für jeden wird der daraus zu bildende APT mit Default-

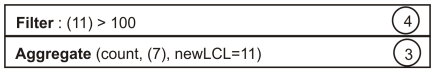

Abbildung 12. Aggregations- und Filteroperator.

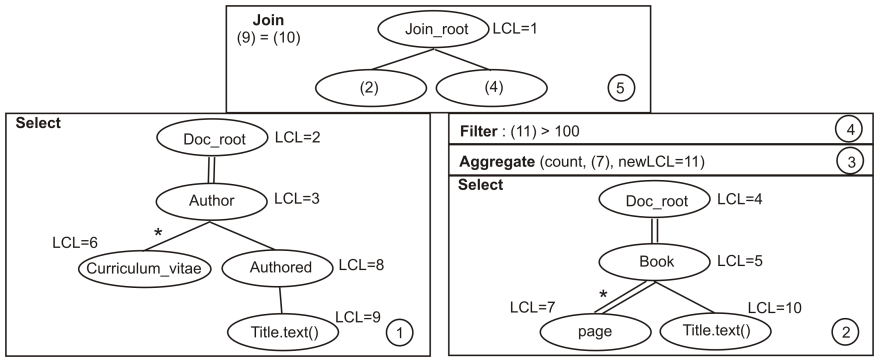

Abbildung 13. Vollständig parametrisierte Selektions- und Join-Operatoren.

Kanten erstellt. Dieser dient einem Selektionsoperator als Parameter. Für alle diese APTs ist eine einzige abschließende Sortieroperation nötig.

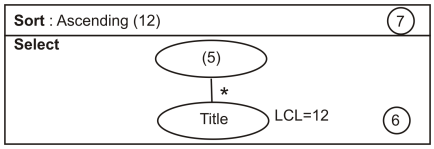

Abbildung 14. Selektion und Sortierung.

Das RETURN-Statement wird auf eine Reihe von Operatoren abgebildet, beginnend bei einer Projektion. Dieses projiziert auf die gebundenen Variablen und die Join-Wurzel. Gefolgt wird diese Projektion von einer Duplikateliminierung, die ausschließlich mit LogicalClasses durchgeführt wird, die aus dem FOR-Statement stammen.

Darauf folgt jeweils eine Selektion mit einem APT für jeden im RETURN-Statement vorkommenden Pfad.

Abschließend wird ein Construct-Operator angelegt, der das in der RETURN-Klausel angegebene XML-Fragment erzeugt.

Der resultierende Ablaufplan in der Algebra ist von unten nach oben abzuarbeiten. Abbildung 18 zeigt den vollständigen Ablaufplan der Anfrage.

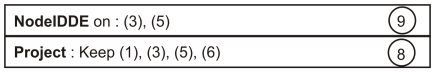

Abbildung 15. Projektion und Duplikateliminierung.

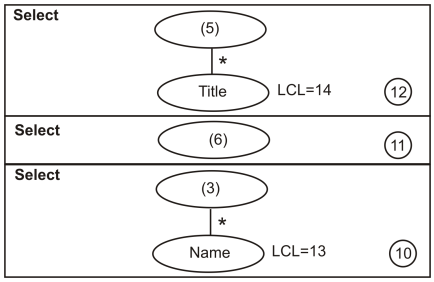

Abbildung 16. Selektion der Rückgabewerte.

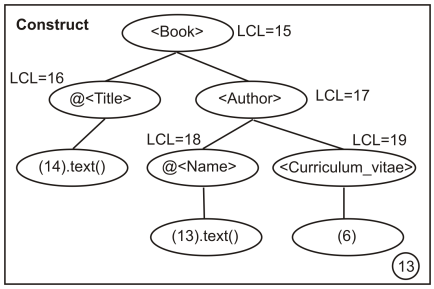

Abbildung 17. Konstruktion des XML-Fragments der Rückgabe.

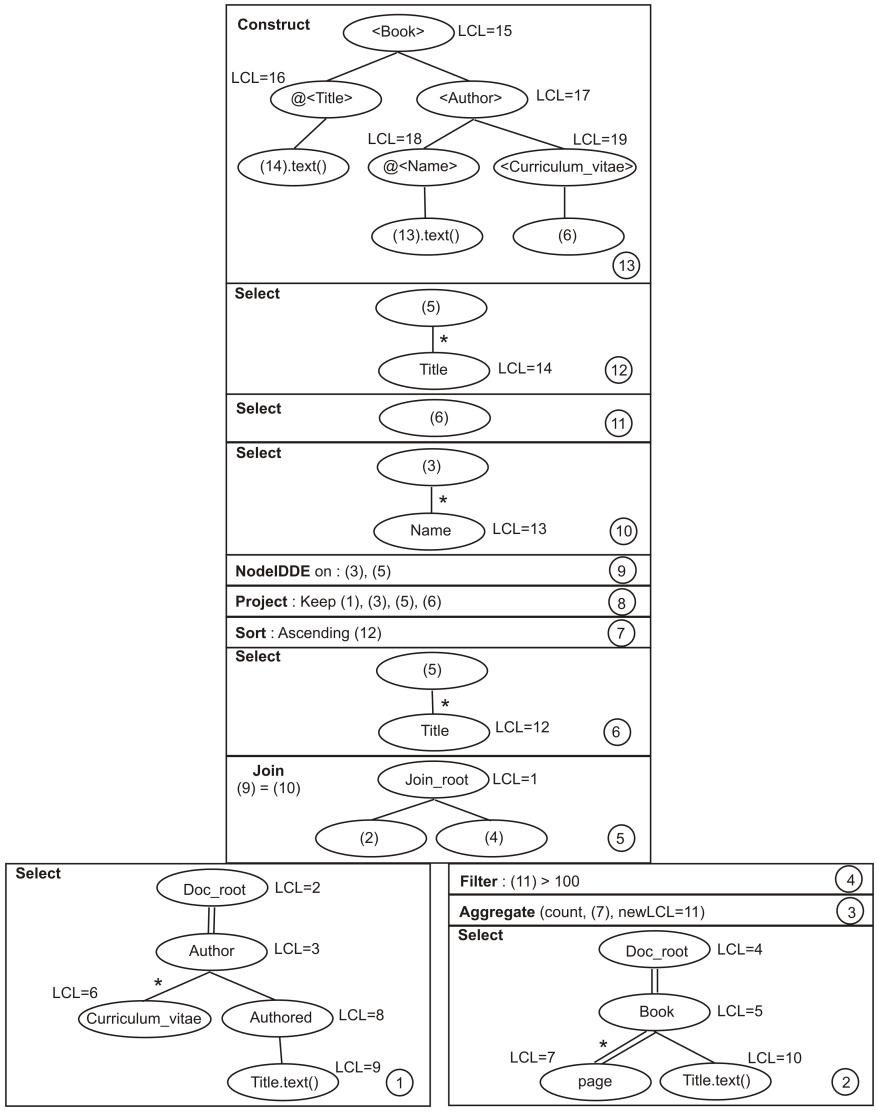

Abbildung 18. Vollständiger Operatorenplan. Ausführung von unten nach oben.

3 Gegenüberstellung der Algebren

Vergleicht man die vorgestellten Algebren wird klar, dass jedes Konzept seine eigenen Stärken und Schwächen aufweist. Mit diesen Schwächen – speziell bei NAL – hat sich Mathis [4] befasst und die nachfolgend aufgeführten Argumente vorgebracht.

Wie bei Mathis erwähnt, ist eine wichtige Aufgabe bei der Auswertung von XQuery das Auflösen von Pfadausdrücken, was aber gleichzeitig eine teure Operation ist, da physische Zugriffe auf die Datenbasis nötig sind. Hier liegt einer der Nachteile von *NAL*. So werden für diese Aufgabe der besseren Pfadauswertung hervoragend geeignete Konzepte, wie *Structural Join*, *Holistic Twig Join* und Pfadindexkonzepte, in *NAL* nicht beachtet. Der Autor stellt eine Erweiterung der Algebra vor, um den Einsatz dieser Konzepte zu ermöglichen. Als Vorteil führt er an, dass das zugrundeliegende tupelbasierte Datenmodell allgemeiner anwendbar wäre als das der baumbasierten Algebren. Umständlich scheint jedoch das „Auspacken“ oder „*tupel-flattening*“ mithilfe von *Unnest-Map*-Operatoren.

Mathis führt weiter aus, dass in *NAL* Selektions-Ausdrücken die wiederholte Auswertung von Pfadausdrücken im Selektions-Prädikat ineffizient gestaltet ist: Über eine Selektion in einem gegebenen Beispiel schreibt er: „It is evaluated for each context node provided by the unnest map operator [...]. This implies *node-at-a-time* calculation of the path step [...]. However, many publications [...] have pointed out that *set-at-a-time* processing of path steps provides better performance in most cases.“[4]

TAX liegt dagegen ein Datenmodell zu Grunde, dass angesicht des baumartigen Aufbaus von XML-Dokumenten und *DOM*-Trees passender und eingängiger erscheint. Ohne Nachteile ist diese Algebra jedoch nicht: „However, its expressive power is definitely too limited for the evaluation of XPath queries: only the descendant and child axis are supported for the definition of a pattern tree.“[4] Womit beispielsweise die Möglichkeiten von XPath gemeint sind, „rückwärtsgerichtete“ Achsen auszuwerten. TAX muss sich darüber hinaus vorwerfen lassen, *position based predicates* umständlich zu handhaben. In NAL können *position based predicates* effizient ausgewertet werden.

TAX hat ebenfalls den Nachteil, künstliche *parent*-Knoten z.B. `joinRoot` und `tax_group_root` einzufügen, die in den nachfolgenden Operatoren beachtet werden müssen. Sei es, dass sie in eine Selektion mitaufgenommen werden müssen oder durch Projektion entfernt werden.

TLC bietet durch die Wiederverwendbarkeit von vorher berechneten Ergebnissen einen großen Vorteil. Die Autoren Jagadish, Lakshmanan, Paparizos, Wu [7] geben mit den Operatoren *Shadow* und *Illuminate* ein Mittel zur Steigerung der Wiederverwendbarkeit: Lassen sich mit *Shadow* Teile von Zwischenergebnissen zeitweise ausblenden, um nicht in die weitere Verarbeitung miteinbezogen zu werden, so können eben diese Teile in späteren Schritten durch *Illuminate* wieder für Algebra-Operatoren verwertbar gemacht werden. Ein weiterer Vorteil ist die Verwendung logischer Klassen als Knoten der Bäume, um aus heterogenen Daten homogene *Logical-Class*-Bäume zu gewinnen.

4 Fazit

Eine Algebra anzubieten, die es erlaubt, die Semantik von XQuery-Anfragen in einer Abfolge entsprechend parametrisierter Algebraoperatoren wiederzugeben, gestattet weitere – auf algebraischen Verfahren basierende – Optimierungsmaßnahmen. Die hier vorgestellten Algebren sind daher sicherlich ein Schritt in die richtige Richtung, auch wenn jede von ihnen noch mit spezifischen Vor- und Nachteilen verbunden ist.

Mathis argumentiert, dass bisherige bekannte Datenbankmanagementsysteme auf Relationen basieren; sie sind also tupelbasiert. Er vermutet, dass die Hersteller tupelbasierten XML-Algebren den Vorzug geben werden, um nicht gezwungen zu sein, zusätzlich zu Tupeln Bäume verarbeiten zu müssen [4]. Doch durch die weitere Arbeit auf diesem Gebiet könnten bisherige Nachteile ausgeglichen werden. Ansätze zur Optimierung von NAL sind zum Beispiel das Ersetzen der Konstruktionsoperation am Ende eines FLWOR-Ausdrucks – wie von Fiebig und Moerkotte [2] vorgestellt – durch Konstruktionspläne, die für die Generierung der XML-Tags in der richtigen Reihenfolge und Schachtelungstiefe sorgen.

Die Erkenntnis, dass *Tree Patterns* in manchen Situationen ein intuitiveres und geeigneteres Mittel sind, führt zum Versuch, Bäume in bisher rein tupelbasierte Algebren einzuführen. Eine solche Algebra könnte mit baumbasierten und tupelbasierten Operatoren ausgestattet sein. Wie eine solche Erweiterung aussehen kann wird von Michiels et al. [6] gezeigt.

Eine Bewertung anhand der Vor- und Nachteile der vorgestellten Algebren allein ist daher nicht möglich, denn Erweiterungen dieser Algebren, um genannte Nachteile auszugleichen, verändern die Beurteilung. Welche der hier vorgestellten Algebren sich durchsetzen wird, bleibt abzuwarten.

Literatur

1. Brantner, M.; Helmer, S.; Kanne, C. Moerkotte, G. Full-fledged Algebraic XPath Processing in Natix ICDE, 2005, 705-716
2. Fiebig, T. Moerkotte, G. Algebraic XML Construction and its Optimization in Natix World Wide Web, 2001, 4, 167-187
3. Jagadish, H. V.; Lakshmanan, L. V. S.; Srivastava, D. Thompson, K. TAX: A Tree Algebra for XML DBPL, 2001, 149-164
4. Mathis, C. Extending a tuple-based XPath algebra to enhance evaluation flexibility Inform., Forsch. Entwickl., 2007, 21, 147-164
5. May, N.; Helmer, S. Moerkotte, G. Strategies for query unnesting in XML databases ACM Trans. Database Syst., 2006, 31, 968-1013
6. Michiels, P.; Mihaila, G. A. Siméon, J. Put a Tree Pattern in Your Algebra ICDE, 2007, 246-255
7. Paparizos, S.; Wu, Y.; Lakshmanan, L. V. S. Jagadish, H. V. Tree Logical Classes for Efficient Evaluation of XQuery SIGMOD Conference, 2004, 71-82