Anfragen und XML-Datenbanken

Diplomarbeit

Universität Rostock, Fachbereich Informatik

vorgelegt von Bietz, geb. Meier, Mathias
geboren am 23.06.1974 in Rostock

Erstgutachter : Prof. Andreas Heuer
Zweitgutachter : Prof. Clemens Cap
Betreuer : Dr.-Ing. Holger Meyer

Abgabedatum: 28. Oktober 2000

Zusammenfassung

XML ist eine moderne Repräsentationsform für semistrukturierte Daten und hat in vielen Anwendungsgebieten an Bedeutung gewonnen. Seit der Einführung von XML 1998 wurde viel Arbeit in Werkzeuge und Mechanismen gesteckt, die die Verarbeitung von XML ermöglichen bzw. vereinfachen sollten. Dazu zählen auch die Methoden, mit denen XML in Datenbanken gespeichert werden kann. Nicht nur als Folge dessen benötigt man eine Anfragesprache, die speziell auf XML zugeschnitten ist und die gewüschte Information aus einem XML-Dokument extrahiert.

In dieser Diplomarbeit werden daher XML-Anfragesprachen und Verfahren zur Speicherung von XML in Datenbanken untersucht. Dabei wird versucht, eine theoretisch fundierte Grundlage für eine XML-Anfragesprache zu finden, wozu auch die Wahl eines Datenmodells gehört. Dieses Datenmodell wird dabei hauptsächlich von einer gewählten Speicherungsform für XML beeinflußt. Die gefundene Lösung besteht aus einer Erweiterung einer objektrelationalen Speicherungsart und einer zugehörigen NF^2-Algebra.

Diese Arbeit wird auch durch die Tatsache gerechtfertigt, daß es noch keinen befriedigenden Standard im Bereich der XML-Anfragesprachen gibt.

Abstract

XML is a kind of modern representation for semistructured data. It has an increasing importance in many application areas. There's a lot of work done since 1998 developing different tools and mechanisms supporting XML and simplifying XML tasks. This takes into account especially the methods for storing XML data in databases. As a consequence a query language specially designed for XML is needed. This query language should allow extraction and transformation of any desired information of an XML document.

Hence this diploma thesis examines XML query languages and techniques for storing XML data in databases. Thereby an attempt is made for a theoretical foundation of a new XML query language including a data model. But, the data model is mostly influenced by a choosen storage method. The solution found in this thesis consists of an enhancement of the object relational storage method and the corresponding NF^2 algebra.

This work is motivated by the fact that there is no satisfying standard in the area of XML query languages up to now.

CR-Klassifikation

D.2.1 Requirements / Specifications
E.2 Data Storage Representations
H.2.1 Logical Design — Data models
H.2.3 Query languages
H.2.8 Database Applications

Key Words

XML, Anfragesprache, objektrelationale Datenbank, semistrukturierte Daten

Inhaltsverzeichnis

Kapitel 1

Einführung

"Data on the web" [ABS00] heißt nicht nur der Titel eines Buches zum Thema, sondern steht auch als ein Synonym für die Bestrebungen aller Wirtschaftszweige, möglichst viele Informationen "online" bereitzustellen. Damit wächst der Datenumfang stündlich an. Viele der bisher vorhandenen und kommerziell eingesetzten Mittel zur Aufbereitung und Darstellung der Daten besitzen ein Format, das nur wenig Unterscheidungsmöglichkeiten zwischen Formatierung, also der graphischen Gestaltung, und inhaltlicher Gliederung solcher Dokumente zuläßt. Dazu zählt zum Beispiel das populäre HTML. Es ist zwar für die private Homepage ausreichend, Firmen mit einem riesigen Web-Portal haben unter anderem jedoch noch folgende Probleme:

Erstens, die Änderung des äußeren Erscheinungsbildes betrifft auch die Inhalte und kann nur mit den Daten gemeinsam erfolgen. Zweitens bringen spezifische Darstellungen für einen bestimmten Personenkreis einen hohen Aufwand für die Wahrung der Synchronität der Daten mit sich und sind jeweils aus den aktuellen Daten zu erzeugen.

Mit dem Einsatz von Datenbanken kann ganz allgemein das Synchronitätsproblem im zweiten Fall behoben werden. Um das erste Problem zu lösen, ist eine Trennung von Inhalt und Formatierung sinnvoll. Dies kann durch den Einsatz von XML erreicht werden. Folglich ergibt sich eine Speicherung der Daten in Form von XML und in einer Datenbank als eine mögliche Lösung dieser Probleme.

Ein weiteres Problem im Web stellt auch heute noch die Suche nach Informationen dar. Suchmaschinen konnten sich erfolgreich etablieren, basieren in den meisten Fällen aber auf einer Wortsuche bezüglich des gesamten Dokumentes.[1] Anfragesprachen für Datenbanken setzen leider immer Schemainformationen voraus, also Metainformationen, welche die Daten beschreiben und die bei im Web publizierten Dokumenten nicht zugreifbar sind, denn es handelt sich selten um strukturierte, sondern viel häufiger um semi- oder unstrukturierte Daten.

Wir befinden uns hier in der widersprüchlichen Situation, eine komfortable Anfragesprache benutzen zu wollen, die aber auf Daten aufsetzen muß, für die es kein explizites Schema (semistrukturierte Daten) oder überhaupt kein Schema (unstrukturierte Daten) gibt.

[1] Ein einfacher Fehler ist zum Beispiel, daß alle Wörter eines Hypertext-Dokuments *einschließlich* der Tags indiziert werden.

In der vorliegenden Arbeit soll nun untersucht werden, wie semistrukturierte Daten (XML) in einem DBMS abgelegt werden können und wie eine Anfragesprache für dieses Szenario aussehen soll.

1.1 Aufgabenstellung

Basierend auf einem zu erarbeitenden Anforderungskatalog sind verschiedene Ansätze für XML-Anfragesprachen und ihre effiziente Umsetzung zu untersuchen. Die Kriterien sollten mindestens die folgende Aspekte berücksichtigen:

- Allgemeine Anforderungen an Anfragesprachen (Vollständigkeit, Abgeschlossenheit, Sicherheit,...)
- Verknüpfung semistrukturierter Anfragemöglichkeiten (unter Nutzung von Information–Retrieval–Techniken) und exakter Datenbankanfragen
- effiziente Speicherungsmodelle von XML-Daten im Hinblick auf Anfragebearbeitung, speziell
- Optimierbarkeit

Eigene praktische Untersuchungen und Implementierungsarbeiten sollen nach begründeter Auswahl mit je einem der folgenden Systeme durchgeführt werden:

- erweiterbare, objekt-relationale Datenbanksysteme:
  - IBM DB2 Universal Database
  - Informix Universal Server
- XML-Server:
  - Software AG Tamino
  - ODI eXcelon

1.2 Projekteinordnung

Am Lehrstuhl Datenbanken- und Informationssysteme des Fachbereichs Informatik der Universität Rostock existieren zwei Projekte, in denen XML-Datenbanken zur Anwendung kommen, BlueView und GETESS. Diese sollen im einzelnen kurz vorgestellt werden.

1.2.1 BlueView

1998 wurde ein vom BMBF[2] gefördertes Projekt, Global-Info, ins Leben gerufen, dessen Ziel es war, Wissenschaftlern den Zugang zu Volltext-, Literaturhinweis-, Fakten und Softwareinformationen von ihrem Arbeitsplatz aus zu

[2]BMBF = Bundesministerium für Bildung und Forschung

ermöglichen. Hieran war die Universität Rostock mit dem Teilprojekt BlueRose[3] beteiligt, wobei Techniken zum Aufbau eines globalen Informationssystems für elektronisch verfügbare Fachinformationen entwickelt werden sollten. Die Nutzung sollte zunächst durch Wissenschaftler, Bibliothekare und Studenten erfolgen. In BlueRose wurde dazu folgende Architektur einer digitalen Biblio-

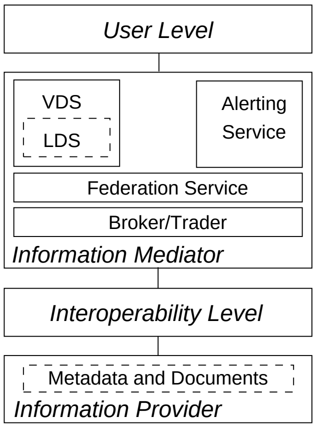


Abbildung 1.1: BlueRose Achitektur, Quelle: [HMP+00]

thek entworfen (siehe Abbildung 1.1). Dabei versteht man unter

- Informationsanbieter (*information providers*): Elektronisch vorhandene Dokumente werden klassifiziert und zusammen mit bibliographischen Metadaten bereitgestellt.
- Interoperabilitätsebene (*interoperability level*): Auf dieser Ebene erfolgt die Suche nach Dokumenten über das Internet.
- Ein Informationsvermittler (*information mediator*), wie zum Beispiel eine Universitätsbibliothek, sammelt Metadaten über wissenschaftliche Literatur. Die Metadaten und Dokumente werden den lokalen Benutzern verfügbar gemacht, wobei Volltext-Dokumente, für die eine Gebühr gefordert wird, bestellt werden.
- Benutzerebene (*user level*): An dieser Stelle erfolgen Dokument-Retrieval und Katalogansicht durch den Benutzer. Personalisierte Sichten auf den Dokumentenbestand sollen die Verwendung des Systems erleichtern.

Da BlueRose nicht weiter vom BMBF gefördert wurde, entstand daraus das Projekt BlueView [BLUurl], welches folgende Teilaspekte von BlueRose umfaßt (Abbildung 1.2):

- Ein lokaler Dokumenten-Server (LDS) speichert alle Dokumente und Metadaten eines Informationsanbieters.
- Der Informationsvermittler benutzt einen virtuellen Dokumenten-Server (VDS) und mehrere lokale Dokumenten-Server. Der VDS realisiert einen transparenten Zugriff auf alle Daten, insbesondere auch auf solche, die

[3] BlueRose = Building Libraries Unifying Enhanced Retrieval-Oriented user Services

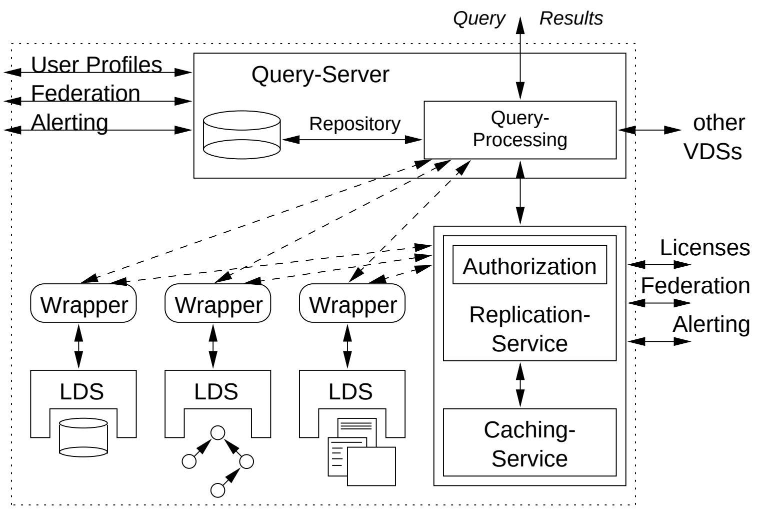


Abbildung 1.2: BlueView: Virtueller Dokumenten-Server, Quelle: [HMP+00]

nicht lokal vorhanden sind. Angeforderte bibliographische Daten und Dokumente werden vom VDS in einem zugehörigen LDS als Replikat gehalten.

- Entsprechend den Fähigkeiten des Benutzer-Arbeitsplatzes können die Dienste der virtuellen und lokalen Dokumenten-Server in Anspruch genommen werden. Ein LDS kann dabei auch durch ein einfaches Volltext-Retrieval-System (z.B. Harvest) realisiert werden.

In BlueView hat man also durch einen VDS die Möglichkeit, transparent auf bibliographische Daten und Dokumente zuzugreifen, die in verschiedenen LDS gespeichert und über das Internet verteilt sein können.

Bezug zur vorliegenden Arbeit

Als grundlegende Speicherungform für Dokumente im LDS wurde unter anderem XML in Erwägung gezogen. Dazu sind in [HMP+00] verschiedene Ansätze dieser Realisierung eines LDS betrachtet worden. Neben den Varianten objekt-relationales DBMS mit IR-Erweiterung, Volltextsystem und XML-Datentyp, wird in [KM99] auch ein hybrider Ansatz vorgestellt, wobei es sich um eine Abbildung von XML auf ein ORDBMS mit XML-Datentyp handelt, der die typischen Probleme[4], die bei der Abbildung von XML auf (O)RDBMS bisher entstanden, entschärfen kann.

Da für den Hybrid-Ansatz vorwiegend Überlegungen zur Art und Weise der Speicherung, nicht aber Anfragemöglichkeiten diskutiert wurden, sollen diese in der vorliegenden Arbeit erörtert werden. Im besonderen ist es für den Einsatz der Hybrid-XML-Datenbank als LDS erforderlich, die Anfragesprache auf diesen Einsatzzweck abzustimmen. Der bereits existierende Vorschlag für eine Anfragesprache für das VDS-System könnte dazu zusätzlich zu einem Vergleich herangezogen werden.

[4] Diese Probleme werden in Kapitel 3 noch genauer geschildert.

1.2.2 GETESS

Gegenstand des BMBF-Forschungsprojektes GETESS [GETurl] ist die Entwicklung einer Suchmaschine, die insbesondere deutschsprachige Informationen zielgerichtet im Internet findet und für den Benutzer aufbereitet. Der GETESS-Ansatz sieht dazu eine Kombination der Internet-Techniken mit den Techniken der Datenbanken, der Computerlinguistik und der Wissensrepräsentation vor.

Der Benutzer wird bei der Formulierung seiner natürlichsprachlichen Anfrage durch eine Ontologie unterstützt und erhält als Ergebnis sogenannte Abstracts, die die wesentlichen Inhalte beschreiben. Für die Speicherung dieser Abstracts sind eine Datenbanklösung und das Datenformat XML vorgesehen. Vorhanden ist bereits eine Variante, die auch die Anfragesprache IRQL [HP99] umfaßt. Eine XML-Speicherstruktur mit eigener Sprache (siehe BlueView) ist jedoch noch nicht realisiert worden.

1.3 Gliederung der Arbeit

Das Thema dieser Arbeit "Anfragen und XML-Datenbanken" ist recht weitläufig, und XML zudem eine sehr "junge" Sprache. Soweit bereits Arbeiten auf diesem Feld entstanden, sollen sie in dieser Arbeit auch Erwähnung finden. **Kapitel 2** und **3** beschäftigen sich daher umfassend mit XML-Anfragesprachen bzw. den Formen der Speicherung von XML-Daten in Datenbanken. Die Auswahl und der Umfang, in dem die Sprachen vorgestellt werden (Kapitel 2), richtet sich dennoch danach, welche Konzepte sich in eine neue Sprache integrieren lassen würden. Das Kapitel 3 (Speicherungsmodelle) setzt den Schwerpunkt auf die Präsentation relationaler Speicherungsmethoden und stellt insbesondere die objektrelationale Methode von H. Meyer und M. Klettke vor.

Auf den objektrelationalen Ansatz aufbauend werden in **Kapitel 4** einige Veränderungen an diesem Ansatz vorgeschlagen. Anforderungen an eine neue XML-Anfragesprache, Probleme und Realisierungsmöglichkeiten sollen dort ebenfalls diskutiert werden.

Im Gegensatz zu den Darstellungen des XML-Speicher- und Anfrage-Systems aus Kapitel 4 wird in **Kapitel 5** eine kurze Betrachtung bereits existierender XML-Server bzw. kommerzieller Lösungen diese Arbeit beenden.

Zunächst soll auf eine Einführung zu XML verzichtet und als bekannt vorausgesetzt werden. Dementsprechend ist die Definition des XML-Standards in [xml98] zu finden — als Einstiegsliteratur sind [BM98] und [GP99] zu empfehlen.

Kapitel 2

Anfragensprachen für XML

In diesem Kapitel werden Anfragemöglichkeiten bzw. Sprachen für XML-Daten erörtert. Die Betrachtungen stützen sich dabei hauptsächlich auf Sprachkonzepte für semistrukturierte Daten bzw. Dokumente.

2.1 Anfragen an semistrukturierte Daten

Aus Datenbank-Sichtweise kann man folgende Einteilung treffen: Auf der einen Seite gibt es strukturierte Daten, die durch die klassische relationale Technik gut (effizient, nicht-redundant, ...) in Datenbanken verwaltet werden können. Auf der anderen Seite gibt es die un- und semistrukturierten Daten, die aber durch ihre Unbeständigkeit und Unregelmäßigkeit in der Struktur bisher nicht so einfach mit Datenbank-Technik bewältigt werden konnten. Hervorzuheben sind hierbei die semistrukturierten Daten[1], die sich durch folgende Eigenschaften auszeichnen [Abi97]:

Ihre Struktur ist irregulär. Die Menge der Dokumente eines Anwendungsgebietes enthält viele unterschiedliche Strukturelemente. Dabei können einige Elemente völlig fehlen oder von unterschiedlichem Typ sein (z.B. Preis in DM oder Euro). Sie können auch zusätzliche Information enthalten (etwa den Umrechnungskurs).

Die Struktur ist implizit. Das Schema bzw. die Struktur ist mit den Daten verwoben und kann nur durch spezialisierte Software sichtbar gemacht werden (siehe HTML, SGML). Prinzipiell hat ein Datenbanksystem nicht die Möglichkeit, solche Schemainformationen zu erlangen.

Die Struktur ist partiell. Die erschöpfende Strukturierung aller Anwenderdaten ist ein kaum zu erreichendes Ziel. Insbesondere aus Datenbanksicht wird es immer einen unstrukturierten Anteil geben.

Die Struktur ist flexibel und unterliegt häufigen Änderungen. Ein starres Schema ist für einige Anwendungsgebiete nicht sinnvoll (Webseiten, ...). In Datenbanksystemen ist eine dementsprechend notwendi-

[1] Im Verlauf dieser Arbeit sollen die Begriffe *Dokumente* und *Daten* zunächst synonym verwendet werden.

ge Schemaevolution und anschließende Migration der Daten zudem unverhältnismäßig aufwendig. Anwendungsfelder, bei denen häufig Strukturveränderungen vorgenommen werden[2], verzichten daher meist auf den Einsatz von Datenbanken.

Das Schema ist relativ groß. Bedingt durch die Vielfältigkeit der Struktur sind die Schemainformationen entsprechend umfangreich. Neben Anfragen an die Daten selbst, müssen nun auch Anfragen an die Stuktur stärker beachtet werden, da etwa ein Nutzer die komplexen Schemainformationen nicht mehr vollständig kennen kann.

Die Unterscheidung zwischen Schema und Daten ist unscharf. Während für strukturierte Daten Schemainformationen und Daten eindeutig festgelegt werden können, ist diese Unterscheidung bei semistrukturierten Daten weniger wichtig. So ist beispielsweise die Umwandlung von Schemainformationen in Daten und umgekehrt durchaus möglich.

Mit dem Aufkommen von XML regte sich wieder vermehrt das Interesse an der Speicherung semistrukturierter Dokumente in Datenbanken. Doch welche Möglichkeiten gibt es, diese Dokumente formal darzustellen, d.h. welche *Datenmodelle* existieren bereits für semistrukturierte Daten?

Ein populärer Weg zur Modellierung semistrukturierter Daten ist der *schemalose Ansatz*. Hierbei erfolgt die Darstellung der Daten durch Bäume oder Graphen. Neue Daten können einfach durch Hinzufügen eines Knotens integriert werden, wodurch die nötige Flexibilität gegeben ist. Ein Vertreter schemaloser Datenmodelle ist das Object Exchange Model (OEM), auf das in einem späteren Abschnitt noch eingegangen wird.

Bei *schemagebundenen Modellen* ist hauptsächlich der Mangel an Flexibilität hervorzuheben. Dieser Nachteil ist nicht nur in der Verwendung eines starren (Datenbank-)Schemas zu sehen, sondern ergibt sich auch durch die damit verbundene Verwendung von Nullwerten, die generell Probleme bereiten, beispielsweise bei der Bildung von Verbunden (*dangling tuples*).

Vorerst soll die Frage nach der Wahl eines Datenmodells für XML-Daten unbeachtet bleiben und erst nach der Betrachtung existierender XML-Anfragesprachen im Kapitel 4 gründlicher untersucht werden.

Im Kontext semistrukturierter Daten muß nun eine Anfragesprache, genauso wie das Datenmodell, wesentlich flexibler sein als in einem konventionellen DBMS. [Abi97] hebt dazu fünf Punkte hervor, die eine Anfragesprache für semistrukturierte Daten erfüllen soll:

1. Anfrageprimitive im Datenbanken-Stil

2. Navigation im Sinne von Hypertext oder Web-Browsing

3. Suche nach Mustern nach Information-Retrieval-Art

4. temporale Anfragen (Anfrage-Versionierung z.B.)

[2]Man stelle sich eine Genome-Datenbank vor, die immer dann angepaßt werden muß, wenn neue Forschungsmethoden bessere Ergebnisse liefern.

5. Anfragen an Daten und Schema-Informationen können in einer Anfrageformulierung gemischt werden.

Da eine Anfragesprache für semistrukturierte Dokumente von einer anderen Datengrundlage ausgehen muß, scheint es aus Datenbank-Sicht zunächst sinnvoll, mit den genannten Punkten als erstes Datenbank-Funktionalität zu fordern, die dann mit Navigationmechanismen und Information-Retrieval-Techniken angereichert wird. Um die genannten fünf Punkte zu realisieren, hat Abiteboul folgende Vorschläge unterbreitet:

Objekt-basierter Ansatz. Der Begriff des Objekts erlaubt die Konzentration auf den Teil der Struktur, der für die Beantwortung der Anfrage relevant ist. Dies steht im Gegensatz zu einer Universalrelation, die sämtliche für die Beantwortung der Anfrage auch unwesentliche Informationen beinhaltet. Die Verwendung von Objekten bedeutet also einen besseren Zugriff auf Teilinformationen, z.B. durch Pfade.

Coercion. (Typzwang, Typkonvertierung): Es gibt zwei Gründe für den Einsatz der Typkonvertierung: Einerseits kann davon ausgegangen werden, daß der Anfragesteller die genaue Struktur und die verwendeten Datentypen nur ungenau kennt. Andererseits gibt es sehr wahrscheinlich Datenquellen, die für dieselbe Information unterschiedliche Typen verwenden, beispielsweise für eine Entfernungsangabe einmal eine Integer-Zahl (km-Angabe) und ein anderes Mal eine Zeichenkette (Meilen-Angabe). Zwei Mechanismen, die diese Probleme beseitigen können, sind eine automatische Typkonvertierung, die Entfernungen immer erst in eine Real-Zahl für Kilometer wandelt, und die Verwendung eines einheitlichen Datentyps. Die Bedeutung dieses Konzepts soll sich nicht nur auf atomare Typen beschränken.

Pfadausdrücke. Pfadausdrücke beschreiben den Weg zu einer Teilinformation in einem Objekt. So gibt *restaurant.address.zipcode* die Postleitzahlen der Restaurants in einem Dokument bzw. einer Dokumentkollektion an. Man unterscheidet zwischen einfachen und erweiterten Pfadausdrücken. Während einfache Ausdrücke einen einzigen Weg beschreiben, erreicht man bei erweiterten Pfadausdrücken mit dem Einsatz von Wildcards wesentlich mehr Ausdrucksstärke.

Sichten. Das Sichtkonzept hat sich beim Einsatz von Datenbanken bisher bewährt. Verschiedene Sichtweisen bieten sich beispielsweise bei der Integration heterogener Daten mit ihren unterschiedlichen Abstraktionsstufen an. Zu beachten sind in diesem Zusammenhang die beliebige Schachtelungstiefe und etwaige Zyklen in den Daten.

Prozeduraler Ansatz. Eine Anfragesprache soll per Definition deklarativ sein. Diese Forderung wäre im Zusammenhang mit der Restrukturierung komplexer Strukturen noch einmal überlegenswert. Ein Beispiel: Während ein Baum durchlaufen (traversiert) wird, könnte gleichzeitig eine Ersetzung aller Knoten A durch A' vorgenommen werden, mit der Bedingung, daß ein B an einer durch einen Pfad bezeichneten Stelle einen bestimmten Wert hat. Die Kontrolle über diesen Ablauf könnte etwaige entstehende Zyklen beispielsweise direkt behandeln.

Die Vorschläge sind bei dieser ersten Betrachtung durchaus plausibel gewählt. So läßt sich die Stuktur in einem semistrukturierten Dokument am besten in Form eines Graphen darstellen. Jeder Knoten enthält eine Teilinformation, die als Objekt betrachtet werden kann. Mechanismen, um diesen Graphen zu traversieren und damit in ihm zu navigieren, sind Pfadausdrücke, und der Typzwang erleichtert die Vergleichbarkeit und erhöht die Effizienz, da die (Vergleichs-)Operationen auf Datentypen zugeschnitten sind — ein Integer-Vergleich ist schneller als ein Vergleich auf String-Basis.[3] Eine prozedurale Formulierung widerspricht allerdings der Forderung nach Deskriptivität.

Auf Probleme im Zusammenhang mit dem vorletzten Vorschlag wird später, bei der Vorstellung der Sprachen, noch eingegangen. Eine grundlegende Betrachtung (unter anderem auch die Wahl eines Datenmodells betreffend) erfolgt aber erst in Kapitel 4, nachdem auch einige Speicherungsformen für XML diskutiert wurden.

Im Hinblick auf den Fokus dieser Arbeit, dessen Basis XML-Daten bilden, werden nun zuerst kurz die Merkmale und Besonderheiten von XML vorgestellt, bevor sich der Sprachüberblick anschließt.

2.1.1 XML-Konzepte

In der *eXtensible Markup Language*[xml98] ist der Grundbaustein das *Element*, welches Text (#PCDATA) und weitere Elemente beinhalten kann. Ein XML-Dokument kann mit Hilfe der folgenden Konzepte erzeugt werden:

- Sequenz: $E = (E1, E2, E3, \ldots)$
- Alternative: $E = (E1 \mid E2 \mid E3)$
- Quantifizierung: $E?, E+, E*$
- Gruppierung: $E = ((E1,\ E2) \mid E3)$

Jedes Element kann eine endliche Anzahl von *Attributen* besitzen. Diese können nur einen Wert aufnehmen und nicht weiter strukturiert werden. In der textuellen Repräsentation besteht ein Element aus einem Start- und einem End-Tag, welche die Subelemente einschließen. Ein Beispiel-Dokument ist etwa:

```
<person panr=0359>
   <name> Mathias Bietz </name>
   <alter> 26 </alter>
</person>
```

Das Element `person` besitzt das Attribut `panr` und zwei Subelemente `name` und `alter`. Ist ein XML-Dokument korrekt geschachtelt und sind die Attributnamen eindeutig, spricht man von einem wohlgeformten Dokument.

[3]Dabei sollte man natürlich weniger eine Typkonvertierung zur Laufzeit (der Anfrage) anstreben.

Mit den speziellen Attributen ID und IDREF/IDREFS kann in XML ein Referenzmechanismus realisiert werden. Desweiteren ist die Reihenfolge der Elemente relevant, während die Ordnung der Objekte bei semistrukturierten Daten in der Regel keine Rolle spielt.

Zu beachten ist noch, daß ein Element auch aus sogenanntem *Mixed Content* (Textanteile und Subelemente in beliebiger Anordung) bestehen kann.

Um die Struktur eines Dokumentes genau zu beschreiben, gibt es eine kontextfreie Grammatik in Form einer *Document Type Definition (DTD)*. Ein wohlgeformtes Dokument, das einer zugeordneten DTD entspricht, wird in diesem Zusammenhang dann als gültig (*valid*) bezeichnet.

DTD's können in gewisser Weise als (Datenbank-)Schemata für XML-Daten verstanden werden. Nachteilig ist dann unter anderem, daß es nur einen atomaren Datentyp #PCDATA gibt, daß alle Bezeichner global sind[4] und daß Referenzen unsicher sind, weil sie keinen Typ besitzen.[5,6]

Nun soll eine Vorstellung existierender Anfragesprachen für XML erfolgen.

2.2 Existierende Sprachvorschläge

Die ersten Sprachen für XML wurden aus dem Wunsch heraus entwickelt, überhaupt eine Möglichkeit zu haben, Informationen aus *einem* Dokument zu extrahieren. Andere Sprachen sind eigentlich nicht als Anfragesprache im Datenbank-Sinn entwickelt worden, und die dritte Art stellt eine Erweiterung vorhandener Sprachen dar.

Die vorgestellten Sprachen sind im einzelnen: XQL, XML-QL und Lorel. Daneben existieren eine ganze Reihe weiterer, z.B. XPath, XML-GL, Quilt, XSL/XSLT und YATL, oder auch UnQL.

2.2.1 XQL

XQL ist eine Anfragesprache für XML und wurde von Jonathan Robie (Texcel), Joe Lapp (webMethods) und David Schach (Microsoft) entwickelt [RLS00]. Robie entwarf XQL nicht aus Datenbanksichtweise, weshalb es sich bei dieser Sprache eher um einen Formalismus zum Suchen, Filtern und Addressieren von XML-Daten handelt. Die Designziele waren im einzelnen:

- Anfragen innerhalb eines Einzeldokuments (im Browser, Editor)
- Anfragen an große Dokumentmengen (sogenannte XML-Repositories)
- Adressierung von Dokumentbestandteilen (auch dokumentübergreifend: XLink, XPointer)
- Verwendung als XSL-Mustersprache

Dies sind sehr unterschiedliche Problemfelder und nicht alles Anfragen im herkömmlichen Sinn.

[4] dem jedoch mit der Einführung von Namesräumen bereits entgegengewirkt wurde

[5] Es kann beispielsweise nicht zugesichert werden, daß im Literaturverzeichnis eines Buches nur Referenzen auf andere Bücher oder Zeitschriften zulässig sind.

[6] Siehe [ABS00] Kapitel 3.3.

Datenmodell

XQL besitzt kein eigenes Datenmodell und benutzt daher XML und dessen implizite Baumdarstellung als solches ([Rob99a]). Hervorzuheben ist, daß jeder Knoten im XML-Dokument einen Typ hat (Attribut, Processing Instruction..) und entweder einen Wert oder einen Namen besitzt. Zwischen den Knoten gibt es Beziehungen, die als

- hierarchisch: Eltern-Kind-Beziehung, Vorfahre — Nachfahre
- positionsbezogen: absolute Position, relativ oder Bereich
- sequentiell: unmittelbar vorher/nachher

charakterisiert werden können. Die Beziehungen bilden die grundlegende logische Struktur von XQL.

Anfragefunktionalität

Eine Anfrage wird in XQL als ein String formuliert (der z.B. auch in URL's angegeben werden kann). Eine Dokumentselektion erfolgt durch Navigation anhand der Elemente, typischerweise mittels eines Pfades. Eine Anfrage kann folgendermaßen zusammengesetzt werden (EBNF-Darstellung):

```
Query ::= [ './' | '/' | '//' | './/' ] Element [ '[' Predicate
          ']' ] [ Path ]
Path  ::= [ '/' | '//' ] Element [ '[' Predicate ']' ] [ Path ]
```

`/` bzw. `//` drücken die hierachischen Beziehungen aus. Damit können Pfadausdrücke zur Selektion von Dokumentbestandteilen gebildet werden, wobei `/` einen absoluten Pfad erzwingt, während mit `//` ein relativer Pfad spezifiziert wird. Der Punkt `.` gibt den aktuellen Kontext für die Anfrage an. Alle Beziehungen (also auch positionale und sequentielle) können im Prädikat angegeben werden, das die Selektionsbedingungen aufnimmt und in den Pfadausdruck integriert wird. Eine einfache Anfrage ist beispielsweise

```
/personen[/personen/person/name='Mathias Bietz']
```

Es werden alle Dokumente mit der Wurzel `personen` selektiert, die von der Wurzel ausgehend den Pfad `personen/person/name` besitzen, wobei das `name`-Element den Text `'Mathias Bietz'` enthalten muß.

```
/personen/person/name
```

selektiert dagegen nicht das gesamte Dokument, sondern alle `name`-Elemente.

An den beiden Beispielen ist zu erkennen, daß mit der Anfrage bereits die logische Struktur des Ergebnisses festgelegt wird. Die Hierachie und die Ordnung der Elemente werden von XQL erhalten und die Rückgabe erfolgt als duplikatfreie Ergebnismenge (neues Dokument). Um das Ergebnis wohlgeformt zu gestalten, wird es in `<xql:result> ...</xql:result>` eingeschlossen.[7]

Zu beachten ist, daß in XQL keine Verbunde zwischen mehreren Dokumenten möglich sind — lediglich ein Semi-Verbund.

[7] Die physische Repräsentation des Ergebnisses (als ASCII-Text, DOM-Baum o.ä.) wurde allerdings offen gelassen.

```
book[author=//book[title='Moby Dick']/author]
```

sucht beispielsweise nach allen Büchern, deren Autor der gleiche ist, wie der des Buches mit dem Titel 'Moby Dick'.

In XQL gibt es auch eine Aggregatfunktion (`count()`), sowie die Mengenoperationen Vereinigung und Durchschnitt, den $\exists$- und $\forall$-Quantor, und die Negation.[8]

In einem ersten Design-Vorschlag von Robie ([Rob99a]) wird noch über die Einführung von Methoden und eine explizite Ergebniskonstruktion nachgedacht. Diese Konzepte sind im späteren XQL-Proposal [RLS00] nicht mehr zu finden.

Einschätzung

Vorteile. XQL baut direkt auf XML auf und ist damit genau auf XML nach den Bedürfnisse eines großen Benutzerkreises zugeschnitten worden. Die Sprache hat eine einfache und kompakte Syntax, die ideal für Webanwendungen ist (etwa in eine URL integrierbar). Dabei ist die Sprache durchweg orthogonal. Außerdem gibt es bereits zahlreiche Implementierungen, z.B. vom GMD-IPSI (siehe XQL-FAQ [Rob99b]).

Als erste Sprache nach der Standardisierung von XML sind Teile von XQL in XPath und XSL aufgenommen worden.

Nachteile. Da die Einfachheit der Sprache beim Entwurf im Vordergrund stand, sind damit einige Einschränkungen verbunden. So gibt es keine Unterstützung für Link-Mechanismen (etwa über ID/IDREF(S)) und auch keine "echten" Verbunde, da sich die Anfragen immer auf ein einzelnes Dokument beziehen. Es gibt auch keine Transformationsmöglichkeiten, die eine Neustrukturierung des Anfrageergebnisses zulassen würden. Auch die Ausdrucksfähigkeit für Pfadangaben oder bei Werten läßt zu wünschen übrig, da keine regulären Ausdrücke für diese Fälle erlaubt sind.

Offen geblieben ist bei der Untersuchung der Sprache auch, ob sie vollständig abgeschlossen ist.

2.2.2 XML-QL

XML-QL ist ein Projekt des AT&T Labs-Research von A. Deutsch, M. Fernandez, D. Florescu, A. Levy und D. Suciu ([DFF$^+$98]). Diese Sprache wurde für Zwecke

- der Datenextraktion aus großen XML-Dokumenten
- der Konvertierung zwischen XML-Daten und Daten aus relationalen oder objektorientierten Datenbanken
- der Transformation (von XML-Daten etwa zwischen verschiedenen DTD's), und
- der Datenintegration aus vielen (verschiedenen) Quellen

entworfen, welches grundlegend andere Ziele im Vergleich zu XQL sind.

[8]Eine weitaus ausführlichere Darstellung der Anfragefunktionalität entnehme man dem XQL-Proposal [RLS00].

Datenmodell

XML-QL bietet für XML zwei Datenmodelle an: ein ungeordnetes (*unordered*) und ein geordnetes (*ordered*) Modell.

unordered model: Die Reihenfolge der Elemente wird ignoriert. Das Dokument ist ein Graph mit einer Wurzel, wobei die Knoten einen *object identifier* (OID) besitzen und die Kanten mit den Elementnamen gekennzeichnet sind. Die XML-Attribute werden den Knoten zugeordnet und die Blätter haben einen String-Typ.

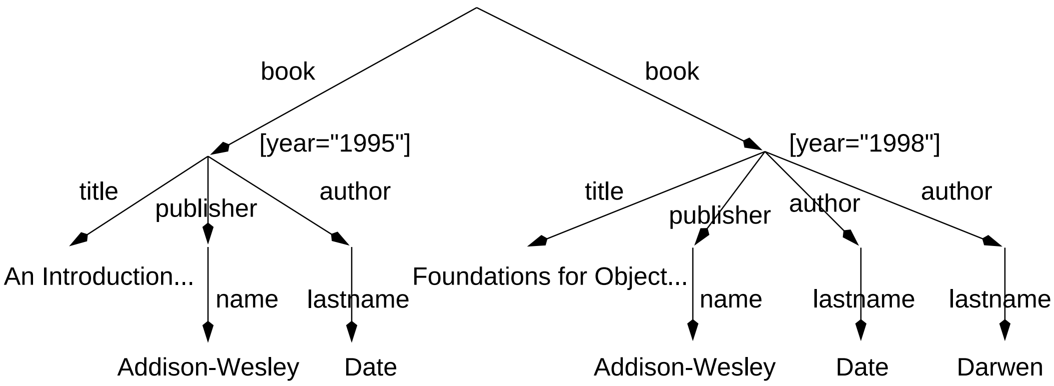


Abbildung 2.1: Beispiel-Graph im *unordered model*, Quelle: [RLS00]

Zu beachten ist die Einschränkung durch dieses Modell: Es kann keine ausgehenden Kanten mit der gleichen Bezeichnung und dem gleichen Wert geben. Dadurch ist nur eine Darstellung von Mengen möglich. Zum Beispiel ist der Graph für

```
<author>
   <lastname> Dates </lastname>
   <lastname> Dates </lastname>
</author>
```

äquivalent zu

```
<author>
   <lastname> Dates </lastname>
</author>.
```

ordered model: Dieses Modell ist ordnungserhaltend. Ein Dokument ist wie im *unordered model* ebenfalls ein Graph. Der Graph ist nun zusätzlich mit Indexzahlen dekoriert, welche die Ordnung unterstützen. Damit ist auch die zuvor genannte die Einschränkung des ungeordneten Modells beseitigt.

Ein Beispiel für das ungeordnete Modell ist in Abbildung 2.1 enthalten, wobei auf die Darstellung der *object identifier* verzichtet wurde. Das zugehörige XML-Fragment befindet sich in Abbildung 2.2.

```
<bib>
   <book year="1995">
      <title> An Introduction to Database Systems </title>
      <author> <lastname> Date </lastname> </author>
      <publisher> <name> Addison-Wesley </name >  </publisher>
   </book>
   ...
</bib>
```

Abbildung 2.2: Beispiel-Dokument zu Abbildung 2.1

Anfragefunktionalität

Eine Anfrage in XML-QL ist eine `WHERE...CONSTRUCT...`-Klausel, bei der im `WHERE`-Teil die Selektionsbedingungen und Dokumente angegeben werden und im `CONSTRUCT`-Teil eine explizite Ergebniskonstruktion erfolgen kann. Für das bereits angedeutete Beispiel-Dokument kann eine Anfrage folgendermaßen formuliert werden:

```
WHERE <book>
         <publisher><name>Addison-Wesley</name></publisher>
         <title> $t </title>
         <author> $a </author>
      </book> IN "www.a.b.c/bib.xml"
CONSTRUCT <result>
               <author> $a </author>
           </result>
```

Das Ergebnis dieser Anfrage ist die Menge der Autoren von Büchern, die ein `publisher`-, `title`- und `author`-Element haben und bei denen der Name des Verlages "Addison-Wesley" ist. `t` und `a` bezeichnen Variablen, die für jedes Buch die entsprechenden Werte annehmen. Mit Hilfe der Variablen `a` wird letztendlich das Ergebnis konstruiert, wobei `result` mehrere `author`-Elemente aufnimmt (Menge).

Weitere hervorzuhebende Merkmale von XML-QL sind:

– geschachtelte Anfragen
– Wahl zwischen ordnungserhaltend / nicht ordnungserhaltend
– beliebige Navigation in Graphen (`WHERE`-Teil)
– Verbunde über Variablen
– Pattern zur Navigation (reguläre Ausdrücke für Pfade, jedoch nicht für Werte)
– Funktionsdefinition (bietet vereinfachte Sichten)

An dieser Stelle ist zu erkennen, daß XML-QL eine wesentlich mächtigere Anfragesprache als XQL ist.[9]

[9] Auch für XML-QL entnehme man eine weitaus ausführlichere Schilderung der Anfragefunktionalität bitte dem Sprachvorschlag beim W3C [DFF+98].

Einschätzung

Vorteile. XML-QL wurde aus einer datenbankorientierten Sichtweise entworfen (im Gegensatz zur dokumentbasierten Sichtweise bei XQL). So ist XML-QL deklarativ und relational vollständig. Durch die graph-basierten Datenmodelle können auch XML-Eigenheiten wie der ID/IDREF(S)-Mechanismus von XML und die Ordnung der Elemente beachtet werden. Eine Transformation von XML-Dokumenten ist ebenfalls (durch die explizite Konstruktion) möglich.

Nachteile. In einigen Bereichen ist XML-QL noch unvollständig (fehlende Aggregatfunktionen, Quantoren). Ein Typkonzept ist auch nicht vorhanden und eine Update-Möglichkeit wurde ebenfalls nicht definiert.

Zwar war SQL ein Vorbild für XML-QL, eine theoretische Grundlage wie das Relationenmodell für SQL gibt es hier aber nicht. Darum sind Aussagen über die Optimierbarkeit oder relationale Vollständigkeit nur schwer möglich.

Ein offenes Problem ist z.B. die Behandlung von Alternativen oder Rekursionen in XML (siehe Abschnitt 2.1.1).

2.2.3 Lorel

Lore ist ein vollständiges DBMS, das speziell für die Speicherung semistrukturierter Daten entworfen wurde. Es wurde an der Stanford University unter der Leitung von Serge Abiteboul entwickelt und ist als Prototyp verfügbar. Lorel [AQM$^+$96] ist die Anfragesprache des Lore-Systems. Diese Sprache im SQL/OQL-Stil zeichnet sich besonders durch eine automatische Typkonvertierung und umfangreiche Spezifikationsmöglichkeiten für Pfade aus.

Lore und Lorel wurden in einer neueren Fassung um XML-Fähigkeiten erweitert [GMW99].

Datenmodell

Lore's ursprüngliches Datenmodell ist das OEM *(object exchange model)*, in dem alle Objekte durch einen gerichteten kantenbeschrifteten Graph dargestellt werden. Alle Realweltobjekte werden auf Objekte des OEM abgebildet, die entweder atomar oder komplex sein können und einen eindeutigen *object identifier* besitzen. Atomare Objekte enthalten den Wert eines atomaren Typs (integer, string, ...) — komplexe Objekte haben als Wert eine Menge von Paaren <label, subobject>, wobei label eine Beziehung zwischen Objekten textuell beschreibt.[10]

In einem *OEM-Graph* werden die Objekte als Knoten und die Beziehungen zwischen den Objekten als Kanten dargestellt. Kanten können nur von komplexen Objekten ausgehen — atomare Objekte besitzen nur ihren Wert. Treffen auf ein Objekt mehrere eingehende Kanten, spielt dieses Objekt mehrere Rollen.

Einem einzelnen Objekt kann ein Name gegeben werden, der als Einstiegspunkt in die Datenbank gilt. Mit der Anfragesprache (Lorel) können nur von diesen Einstiegspunkten ausgehend Anfragen formuliert werden.

[10] Hier handelt es sich zunächst nur um Objekt–Subobjekt–Beziehungen.

Die Veränderungen des OEM im Bezug auf XML begründen sich darauf, daß einem Objekt (Knoten) im OEM kein festes Label (eine Kante) zugeordnet ist und daß Namen nur sporadisch (als Einstiegspunkte) benutzt werden. Notwendig ist die Anpassung des OEM auch durch die Ordnung der Elemente bei XML.[11]

Ein XML-Element wird nun ein Paar <eid, value>, wobei eid ein eindeutiger *element identifier* ist, und value entweder ein atomarer Textstring oder ein komplexer Wert mit den folgenden vier Komponenten ist:

1. Ein String *tag* entsprechend dem XML-Tag für dieses Element.
2. Eine geordnete Liste von Paaren Attributname/atomarer Wert.
3. Eine geordnete Liste von Querverweisen der Form <label, eid> (entsprechend den korrespondierenden ID-/IDREF(S)-Attributen).
4. Eine geordnete Liste für normale Subelemnte (<label, eid>), die sich aus der Verschachtelung des XML-Dokuments ergeben.

Auch dieses veränderte Datenmodell läßt eine Graphdarstellung zu. Die Ele-

```
<DBGroup>
   <Member Name="Smith" Advisor="m1">
      <Age>28</Age>
   </Member>
   <Member ID="m1" Project="p1">
      <Name>Jones</Name>
      <Advisor>Ullman</Advisor>
   </Member>
   <Project ID="p1" Member="m1">
      <Title>Lore</Title>
   </Project>
</DBGroup>
```

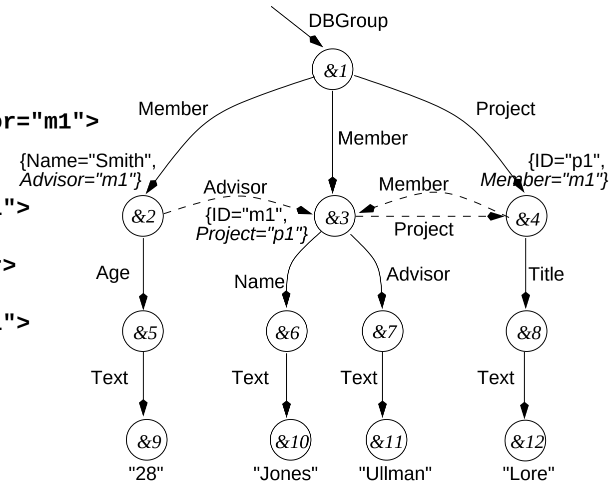


Abbildung 2.3: XML-Dokument und zugehörige Graph-Darstellung, Quelle: [GMW99]

mente bilden die Knoten und es gibt nun zwei Arten von Kanten (siehe Abbildung 2.3): Kanten, die zu Subelementen führen (einfache Pfeile) und Kanten für die Querverweise (ID/IDREF(S), gestrichelt).

Es bieten sich zwei Betrachtungsweisen an. Einerseits kann aus XML-orientierter Sicht auf die Querverweise verzichtet werden, wobei die ID/IDREF-Attribute wie Textstrings behandelt werden. Dies ist der *literal mode*. Die Datenbank ist demnach ein Baum. Andererseits, beachtet man nun aber die Kanten für die Querverweise, gibt es keinen Unterschied mehr zwischen den beiden Kantensorten und die Datenbank ist jetzt ein Graph. In diesem *semantic mode* erreicht man die gleiche Ausdrucksstärke wie im ursprünglichen OEM.

[11]Erfreulich ist, daß man mit dem OEM bereits eine Graph-Struktur definiert hat, die in XML über die speziellen Attribute vom Typ ID/IDREF(S) erzeugt wird.

Anfragefunktionalität

Grundlage der Anfragesprache Lorel ist OQL mit einigen Veränderungen für semistrukturierte Daten. Das sind vor allem das eingeführte Type-Casting (*type coercion*, siehe Seite 8) und die Pfadausdrücke. Im folgenden werden nur diese beiden Erweiterungen präsentiert, jeweils für das OEM und dann für das XML-OEM.

Type-Casting

Zur Vermeidung von Typfehlern wird in Lorel die Semantik für (Vergleichs-) Operatoren aufgeweicht. Beispielsweise lassen sich Anfragen wie

```
select X
from   DBGroup.Member X, X.Age Y
where  Y = 28
```

stellen, die in OQL bei dem Vergleich eines Objekt (Y) mit einem Integer (28) zu einem Laufzeitfehler geführt hätten. Man unterscheidet zwei Varianten, für die die Operatoren (der Prädikate, z.B. =, +,...) neu zu definieren sind: einmal im Zusammenhang mit atomaren Objekten (a), und (b) mit komplexen Objekten.

arg2 / arg1	string	real	int
string	–	string $\longrightarrow$ real	beide $\longrightarrow$ real
real		–	int $\longrightarrow$ real
int			–

Abbildung 2.4: Type-Casting für Vergleichsoperatoren.

Zu a): Nehmen atomare Objekte beispielsweise an einem Vergleich teil, wird das Objekt als erstes dereferenziert und man erhält einen Wert mit zugehörigem Datentyp. Im Beispiel wäre Y dann ein Integer-Wert und der Vergleich könnte durchgeführt werden. Sind die Typen dann noch nicht kompatibel, werden weitere Typumwandlungen vorgenommen (z.B. nach real).

Abbildung 2.4 gibt diese Typumwandlungen für String, Real und Integer an. Es gilt zu bedenken, daß die Operatoren ihre Transitivität verlieren. So ist "05" = 5 und 5 = "5", aber "05" $\neq$ "5".

In Lorel wird versucht, Typfehler weitestgehend zu vermeiden, auch wenn die Operanden beispielsweise nicht "logisch" zusammenpassen (etwa miteinander verglichen werden können). Der Vergleich des Strings "apple" mit einem Integer-Wert ergibt demnach *false*, jedoch keinen Fehler.

Zu b): Bei Operationen mit komplexen Objekten sind mehrere Fälle zu unterscheiden. Wird ein Objekt mit einem anderen Objekt verglichen, kann das einmal auf OID-Basis geschehen ("=="), oder auf Wert-Basis ("="). Ein Vergleich mittels der OIDs muß also explizit formuliert werden.

Desweiteren gibt es Vergleiche zwischen Mengen von Objekten mit Werten, atomaren Objekten usw. Im Zweifelsfall wird immer *false* zurückgegeben.[12]

[12] Genaueres hierzu ist im Artikel [AQM+96] zu finden.

Funktion	Beschreibung
Flatten(e)	Ignoriert alle Tags und serialisiert alle Text-Werte im gesamten Teilbaum unterhalb von e. Es werden nur die Subelementbeziehungen betrachtet.
Concatenate(e)	Konkateniert alle unmittelbaren Text-Elemente und ignoriert alle anderen Subelemente.
Tag(e)	Das XML-Tag des Elements e als String.
Eid(e)	Der *element identifier (eid)* als String.
XML(e)	Transformiert den Graph, beginnend beim Element e, in ein XML-Dokument.

Abbildung 2.5: *Built in*-Funktionen für Elemente

Änderungen für XML-Lorel. Die Besonderheit im Bezug auf XML besteht in der Behandlung der Attributwerte und der Textkomponenten, die ein Element besitzen kann. Attribute werden dabei immer als atomare Objekte angesehen — Elemente werden durch *built in*-Funktionen konvertiert. Diese Funktionen (siehe Abbildung 2.5.) werden automatisch angewandt, können aber auch explizit (z.B. in einer select-Klausel) angegeben werden. Sie erzeugen aus den Elementen eine Stringrepräsentation in folgender Weise:

1. Ein atomares (Text-)Element gibt seinen Wert selbst zurück.
2. Für Elemente ohne Attribute, aber mit mehreren (Text-)Elementen als Kinder, werden die Werte der Kindknoten zu einem Textstring verkettet (Spezialfall der Concatenate-Funktion in Abbildung 2.5)
3. Alle anderen Elemente geben ihre *eid* in Form eines String zurück (*Eid*-Funktion).

So ist im Beispiel (Abbildung 2.3) der Ausdruck DBGroup.Member.Advisor = "Ullman" nur korrekt, weil implizit DBGroup.Member.Advisor.*Text* = "Ullman" angenommen wird.[13,14]

Pfadausdrücke

In Lorel werden Pfadausdrücke in einer dot-Notation angegeben, was eine Vereinfachung gegenüber OQL bedeutet. Mittels Variablen lassen sich die vereinfachten Pfadausdrücke auf korrespondierende OQL-Ausdrücke abbilden. So ist beispielsweise

```
select Z
from   DBGroup.Member.Advisor Z
```

in

[13] Auch die weiter oben aufgeführte Anfrage `select ... X.Age Y where Y = 28` hätte eigentlich anders formuliert werden müssen, um auf das Alter zugreifen zu können — im OEM hätte es nur keinen weiteren Knoten &9 gegeben und &5 hätte den Integer-Wert 28 gehabt.

[14] Diese ausführliche Darstellung der Anfragefunktionalität macht im Fall von Lorel durchaus Sinn, da die Ideen in einem späteren Kapitel noch einmal aufgegriffen werden.

```
select Z
from   DBGroup.Member R, R.Advisor Z
```

zu übersetzen. Pfadausdrücke können im select-, from- und where-Teil einer Anfrage auftauchen. Durch reguläre Ausdrücke, Wildcards und Pfadvariable sind in Lorel umfangreiche Möglichkeiten für Pfadangaben vorhanden. Die regulären Ausdrücke

$$s_1 s_2 \quad s_1|s_2 \quad (s_1) \quad (s_1)? \quad (s_1)+ \quad (s_1)*$$

sind wie üblich definiert: ? bedeutet null- bis einmaliges Auftreten, + entspricht ein- oder mehrmaligem Auftreten, usw.

Die Wildcards

$$\% \quad \text{und} \quad \#$$

bedeuten folgendes: % steht für null bis mehrere Zeichen eines Labels (Kante, Pfadkomponente) und # kann einen beliebig langen Pfad ersetzen.

Pfadvariablen haben als Wert einen Pfad durch den Graph. Pfade lassen sich so auf Gleichheit prüfen. Mittels der Funktion *path-of* kann der Wert einer Pfadvariablen in einen String umgewandelt und ausgegeben werden.

Bedeutung	Symbol	Beispiel	E. (sem.)	E. (lit.)
nur Subelemente	>	DB.Member.>Name	&6	&6
		DB.Member.>Advisor	&3,&7	&7
nur Attribute	@	DB.Member.@Name	“Smith”	“Smith”
		DB.Member.@Advisor	*empty*	“m1”
keine Beschränk.	kein	DB.Member.Advisor	&3, &7	&7, “m1”

Abbildung 2.6: Bedeutung der Konstrukte für Pfadausdrücke

Für **XML-Lorel** sind **folgende Veränderungen** vorgenommen worden: Um zwischen Subelementen und Attributen in Pfadangaben unterscheiden zu können, wurden die Symbole > und @ eingeführt, siehe Abbildung 2.6. Die beiden letzten Spalten geben die Ergebnisse der Beispiele an, einmal für den *semantic mode* (E. (sem.)) und einmal für den *literal mode* (E. (lit.)). Pfadausdrücke können zusätzlich Bereichsangaben enthalten: x.Office[1-2] gibt dann die ersten zwei Office-Elemente unterhalb des x-Elementes zurück. Bereichsangaben können auch auf Variablen angewandt werden.

Sonstige XML-Erweiterungen von Lorel

Mittels einer with-Klausel, die an das select-from-where-Konstrukt angehängt werden kann, lassen sich Ergebnisse restrukturieren. Die anderen Erweiterungen betreffen die Unterstützung von *Skolem functions*, Updates und Schemabeschreibungen (DataGuides), auf deren Beschreibung hier verzichtet wird.

Einschätzung

Von OQL ausgehend besitzt Lorel eine standardisierte Grundlage. So gibt es auch bereits Quantoren, Aggregatfunktionen, Mengenvergleiche, Sortierung etc. Die Mächtigkeit der Pfadausdrücke und vor allem die automatische Typkonvertierung ermöglichen einen “bequemen Umgang” mit semistrukturierten Daten, also auch XML. Die XML-Erweiterungen des Datenmodells und der Anfragesprache entsprechen dem Charakter von XML voll und ganz (Adäquatheit). Die Sprache wäre somit ein idealer Kandidat für eine XML-Anfragesprache. Sie ist deklarativ und ausdruckstark.

2.2.4 Sprachvergleich

Ein Vergleich der drei vorgestellten Anfragsprachen zeigt die Vor- und Nachteile der einzelnen Sprachentwürfe (Abbildung 2.7). Lorel und XML-QL sind die mächtigsten Sprachen. Insbesondere Lorel (vergleichbar mit dem hochent-

	XQL	XML-QL	Lorel
Selektion	+	+	+
Extraktion	+	+	+
Restrukturierung	+	+	+
Dok.-coll.	-	+	+
ordnungserhaltend	+	(+)	+
Links	-	+	ID/IDREF
Join	-	+	+
Types	-	-	(+)

Abbildung 2.7: Vergleich von XML-Anfragesprachen

wickelten SQL-Standard) bietet die Anfragefunktionalität, die für alle Arten von Anfragen benötigt wird. XQL ist nur geeignet, um die Informationen eines Dokuments anzufragen. Anfragen über mehrere Dokumente lassen sich nicht stellen. Als eine der ersten XML-Anfragesprachen und aufgrund ihrer Einfachheit wurde XQL in vielen Systemen implementiert (siehe Kapitel 5).

Was fehlt?

Die meisten Sprachentwürfe lassen eine theoretisch fundierte Grundlage vermissen. Bei der Definition einer neuen Sprache für relationale Datenbanken kann hierfür immer vom Relationenmodell und der Relationenalgebra bzw. dem Bereichs- oder Tupel-Kalkül ausgegangen werden. Bei der Sprachfestlegung für XML-Anfragesprachen ging man oft nur von einer Syntax-Definition aus, bei der anhand von Beispielen die Bedeutung der Konstrukte erklärt wurde.

Einige Sprachkonzepte fehlen sogar bisher gänzlich. So gibt es selten Typen, d.h. man geht vom XML-eigenen und einzigen Datentyp #PCDATA aus. Das Sichtkonzept wurde in vielen Fällen auch vernachlässigt. Das größte Manko ist aber wohl das Fehlen einer Update-Sprache, um nicht nur Dokumente komplett in einer Datenbank abzulegen, sondern auch Teile von Dokumenten an bestimmten Stellen einzufügen (Insert), zu ändern (Update) oder zu löschen (Delete).

Wenig Forschung wurde bisher beim Aspekt der Anfrageoptimierung betrieben, um z.B. zu überprüfen, inwiefern sich Optimierungstechniken für XML-Anfragesprachen anwenden lassen. Da insbesondere Algebren für diese Sprachen fehlen, betrifft dies vor allem die algebraische Optimierung.

2.3 Zusammenfassung

Dieses Kapitel hat gezeigt, daß es bereits viele Bestrebungen im Bereich der XML-Anfragesprachen gibt. Die drei vorgestellten Sprachen stammten dabei aus unterschiedlichen Bereichen: XQL und die verwandten Sprachen XPath und XSLT besitzen eine eher dokumentbasierte Sicht. XML-QL lehnt sich an eine vorhandene Datenbanksprache, SQL, an, und Lorel (oder besser XML-Lorel) ist die Erweiterung einer Datenbanklösung für semistrukturierte Daten.

Diese Herangehensweisen führen zu einer unterschiedlichen Güte der Sprachen, die sich an den fünf eingangs erwähnten Punkten von Abiteboul (Seite 7) einschätzen läßt.

1. Anfrageprimitive im Stil bisheriger Datenbanken wurden nicht von allen Sprachen übernommen. Zählt man zu diesem Punkt auch die mit den Sprachkonstrukten verbundene Funktionalität, macht XQL bereits eine Ausnahme.[15]

2. Die Navigation über Dokumentgrenzen hinweg im Sinne von Hypertext-Browsing ist nicht selbstverständlich. Die Möglichkeit, die XML-Links zu benutzen, ist in den meisten Fällen aber gegeben. Nur die neueren Formen (XPointer, etc.) werden von keiner Sprache unterstützt.

3. Information–Retrieval–Methoden sind bei keiner der Sprachen zu finden. Die Verwendung von Wildcards für Inhalt und Struktur in einigen Sprachen mag zwar ein Anfang sein, Retrieval-Methoden ersetzen sie jedoch nicht.

4. Temporalen Anfragen, mit denen z.B. nur die Veränderungen der Daten abgefragt werden, finden kaum Beachtung. Da sich die meisten Sprachen erst im Anfangsstadium befinden, wurden Aspekte wie dieser vernachlässigt.

5. Anfragen an Schemainformationen wurden in den meisten Fällen in Betracht gezogen. So existieren beispielsweise dedizierte Funktionen, mit denen Namen von XML-Elementen auch Variablen zugewiesen werden können. Auch die Kombination von Anfragen an Daten und Schema zugleich ist gewährleistet worden.

Zum Schluß bleibt anzumerken, daß die Sprachen häufig unabhängig von der Art und Weise sind, wie die XML-Daten gespeichert werden. Eine Auseinandersetzung mit der Speicherung von XML in einer Datenbank im Zusammenhang

[15] da beispielsweise Verbunde umständlich bzw. gar nicht formuliert werden können.

mit einer Anfragesprache wurde bei der Sprachdiskussion leider immer vernachlässigt. Daß die Speicherungsform aber auch Einfluß auf das Datenmodell einer Sprache hat, wird sich in den nächsten Kapiteln zeigen.

Kapitel 3

Speicherungsmodelle für XML

Nachdem im letzten Kapitel einige XML-Anfragsprachen vorgestellt wurden, sollen jetzt die zugrundeliegenden Speicherungsformen dargestellt werden. Sie bestimmen maßgeblich den Aufwand, mit dem eine Anfragesprache realisiert wird.

3.1 Überblick

Die Möglichkeiten, wie XML-Daten dauerhaft gespeichert werden können, sollen für diese Arbeit grob in drei Bereiche eingeteilt werden: 1. Speicherung im Dateisystem, 2. in relationalen Datenbanken und 3. in objektorientierten Datenbanken.

Text-Dateien im Dateisystem

Diese Form der Speicherung ist sicher nur für kleine Mengen an XML-Daten vertretbar. Schwierigkeiten gibt es beipielsweise beim Update der Daten, da immer das gesamte Dokument für den Zugriff gesperrt werden muß. Die Dateistruktur bildet eine zusätzliche Granularität, die sich auch als nachteilig erweist, wenn das Update nur einen kleinen Teil einer großen Kollektion betrifft, die in einer einzigen Datei vorliegt.

Bei Anfragen verbessert die Verwendung von Indexen das Antwortzeit-Verhalten. Einen Index kann beispielsweise ein Volltext-Index darstellen, der mittels einer invertierten Liste realisiert wird. Generell kann man entscheiden, ob die XML-Struktur im Index beachtet werden soll, d.h. ob und wie die XML-Tags indiziert werden, oder ob die XML-Tags als normaler Text betrachtet werden. Die Verwendung solcher Indexe unterstützt hauptsächlich Information–Retrieval–Anfragen.

Relationale Datenbanken

Legt man XML-Daten in Datenbanken ab, so steht dahinter der Gedanke, die Eigenschaften eines DBMS auszunutzen: effiziente Speicherverwaltung, Transaktionsverwaltung, Anfragesprachen,... Jedoch ist die Abbildung semistrukturierter Daten auf eine Tabellenstruktur (relationaler DBS) nicht so einfach zu bewältigen. Beispielsweise gibt es in XML nur den "Typ" #PCDATA — die zum Teil reichhaltigen Typen von DBMS kommen damit kaum zum Einsatz. Die diversen Abbildungsmechanismen bilden den Schwerpunkt dieses Kapitels (siehe folgende Abschnitte).

Objektorientierte Datenbanken

OODBMS besitzen mächtigere Konzepte zur Datenmodellierung als relationale DBMS. Dies sollte sich auch vorteilhaft für die Repräsentation semistrukturierter Daten und damit XML ausnutzen lassen.

Aus einer *Document Type Definition* läßt sich quasi direkt eine Typhierachie aufbauen. Vereinfachend gesagt wird für jedes XML-Element ein Typ erzeugt. Der Nachteil einer solchen 1:1-Abbildung ist, daß die Daten stark fragmentiert werden infolge der Erzeugung vieler kleiner Objekte. Dies kann verbessert werden, soll in diesem Kapitel aber nicht weiter beachtet werden.

Die beiden zuletzt genannten Möglichkeiten (RDBMS und OODBMS) gehen zumeist von einer Abbildung der XML-Struktur (festgelegt durch eine DTD) auf ein Datenbankschema aus. Dies muß jedoch nicht der Fall sein, wenn man in das DBMS einen **komplexen XML-Datentyp** integriert. Die Speicherung erfolgt dann in einem CLOB/BLOB[1] und es gibt spezialisierte Funktionen (UDF's[2]), die den Zugriff realisieren. Allerdings können Anfragen nur schlecht unterstützt werden und es kommt kaum zur Ausnutzung vorhandener Datenbank-Techniken, wie z.B. der Zugriffsunterstützung für Relationen (B-Baum etc.).

Oberflächlich betrachtet scheint ein an die Menge der zu verarbeitenden XML-Dokumente angepaßtes Datenbankschema erfolgversprechend, da man das reichhaltige Repertoire von DB-spezifischen Konzepten (Datenmodellierung, Transaktionsverwaltung, usw.) nutzen kann. Zur Erzeugung eines Schemas müssen dabei nicht nur "statische" Informationen (DTD, ...) verwendet werden. Auch eine Analyse der Daten, die beispielsweise Aufschluß über Wertverteilungen gibt, kann für Zwecke der Performance-Steigerung herangezogen werden.

Als Vorgriff auf die folgenden Abschnitte sei hier die hybride Form der Speicherung erwähnt, d.h. daß strukturierte Anteile in ein DB-Schema umgesetzt werden und weniger strukturierte Anteile roh (ohne Typinformation, etwa *Mixed Content*) im XML-Datentyp abgelegt werden.

Der nächste Abschnitt stellt zunächst zwei relationale Speicherungsformen vor.

3.2 Relationale Speicherungsformen

Der einfachste Ansatz [ABS00] geht davon aus, daß die XML-Daten allgemein durch einen Graph dargestellt werden, bei dem jedem XML-Element ein Knoten des Graphen zugeordnet wird. Element-Subelement-Beziehungen werden

[1]CLOB = Character Large Object, BLOB = Binary Large Object
[2]UDF = User Defined Function, verwirklicht z.B. in DB2 von IBM

```
<person id='1' age='55'>
  <name> Peter </name>
  <address> 4711 Fruitdale Ave. </address>
  <child>
    <person id='3' age='22'>
      <name> John </name>
      <address> 5361 Columbia Ave. </address>
      <hobby> swimming </hobby>
      <hobby> cycling </hobby>
    </person>
  </child>
  <child>
    <person id='4' age'7'>
      <name> David </name>
      <address> 4711 Fruitdate Ave. </address>
    </person>
  </child>
</person>
<person id='2' age='38' child='4'>
  <name> Mary </name>
  <address> 4711 Fruitdale Ave. </address>
  <hobby> painting </hobby>
</person>
```

Abbildung 3.1: XML-Fragment (aus [FK99])

durch die Kanten dargestellt und mit dem Namen des Subelements bezeichnet. Die Werte der XML-Elemente bilden die Blätter.

Dieser Graph kann in zwei Tabellen `Ref`(src,label,dst) und `Val`(oid,value) untergebracht werden. Die erste Tabelle speichert dabei die Kanteninformationen und die zweite nimmt die Werte auf.

Als Beispiel sei das XML-Dokument aus Abbildung 3.1 gegeben. In Abbildung 3.2 sind dazu die Einträge in der `Ref`- und `Val`-Tabelle zu sehen. Die Werte der src- und dst-Spalten der `Ref`-Tabelle sowie die oid-Spalte der `Val`-Tabelle sind quasi "Objektidentifikatoren", mit denen der Graph rekonstruiert wird. Anfragen sind kaum noch deklarativ, da der Benutzer sein Ergebnis (mit unverhältnismäßig vielen Verbunden) selbst zusammensetzen muß.

Die Darstellung der Werte in einer Val-Tabelle besitzt noch eine kleine Unkorrektheit. Bei der Überführung der XML-Werte in Werte mit Typinformationen, z.B. auf der Basis einer der Schema-Erweiterungen für XML (DCD o.ä.), können diese nicht mehr in einer einzigen Spalte (value) gespeichert werden. Eine Lösung ist beispielsweise eine getrennte Speicherung in mehreren Tabellen, wie sie im folgenden von Florescu und Kossmann vorgeschlagen wird.

3.2.1 Relationale Speicherung — Florescu & Kossmann

Mit dem Ansatz, der von Daniela Florescu und Donald Kossmann zu ersten Testzwecken entwickelt wurde, sollte die zunächst einfachste und offensichtlichste Form der Speicherung und deren Performanceverhalten untersucht werden.

Für die Erzeugung der Datenbank-Schemata wird keine Zusatzinformation in Form einer DTD oder, durch eine Analyse der Daten gewonnene Information

`Ref`

src	label	dst
root	"person"	1
root	"person"	2
1	"age"	11
1	"name "	12
1	"address"	13
1	"child"	13
1	"child"	14
2	"age"	21
.	...	.
13	"age"	31
⋮	⋮	⋮

`Val`

oid	value
11	55
12	"Peter"
13	"4711 Fruitdale Ave."
21	38
.	.
31	22
⋮	⋮

Abbildung 3.2: Ref- und Val-Tabelle zum XML-Fragment aus Abbildung 3.1

benötigt. Die Grundlage wird auch hier durch eine Menge geordneter Graphen[3] gebildet. Wie bereits angedeutet wird jedes Element durch einen Knoten mit einem Objektidentifikator dargestellt. Ein künstlicher Objektidentifikator wird im Falle des Fehlens des ID-Attributes des jeweiligen XML-Elementes erzeugt.

Auch hier wird ebenfalls eine Unterscheidung in Kanten- und Wert-Tabelle vorgenommen, wobei die Autoren drei Möglichkeiten zur Speicherung der Kanten und zwei Formen für die Speicherung der Werte angeben, welche sich kombinieren lassen und somit sechs Möglichkeiten der Abbildung von XML-Daten darstellen.

Als anschauliches Beispiel dient im folgenden das bereits in Abbildung 3.1 angegebene XML-Fragment.

Edge

source	ordinal	name	flag	target
1	1	age	int	v1
1	2	name	string	v2
1	3	address	string	v3
1	4	child	ref	**3**
1	5	child	ref	**4**
2	1	age	int	v4
.	.	...	...	.

Abbildung 3.3: *Edge*-Tabelle

Abbildung der Kanten

Edge-Ansatz. Beim Edge-Ansatz wird die Speicherung der Kanten ähnlich wie in der Ref-Tabelle vorgenommen, die um einige Zusatzinformationen erweitert wird (Abbildung 3.3[4]). Neben den OIDs der Quell- und Ziel-Knoten und dem Label werden zusätzlich ein Flag zur Unterscheidung,

[3] entsprechend der Menge der XML-Dokumente

[4] Quelle aller Abbildungen dieses Abschnitts: [FK99]

V_{int}

vid	value
v1	55
v4	38
v8	22
v13	7

V_{string}

vid	value
v2	Peter
v3	4711 Fruitdale Ave.
v5	Mary
v6	4711 Fruitdale Ave.
v7	painting
.	...
v15	4711 Fruitdale Ave.

Abbildung 3.4: separate Werte-Tabellen

ob die Kante auf einen Datenwert(Blatt) verweist oder eine innere Kante darstellt, und die Ordnungsnummer der Kante in die Tabelle aufgenommen. Die Ordnungsnummer dient der Rekonstruktion der XML-Daten, falls die Ordnung der Elemente eine Rolle spielt.

Binary-Ansatz. Dieser Ansatz entspricht einer horizontalen Partitionierung der Kanten-Tabelle des Edge-Ansatzes. Dabei werden alle gleichnamigen Kanten (alle gleichnamigen Elemente also) in eigenen Tabellen untergebracht, die dem Schema

$$B_{kantenname}(source, ordinal, flag, target)$$

folgen.

B_{name}

source	ordinal	flag	target
1	2	string	v2
2	3	string	v5
.	.	...	.

Abbildung 3.5: B_{name}-Tabelle für alle XML-Elemente *name*

Für alle *"name"*-Kanten sieht die entsprechende Tabelle beispielsweise wie in Abbildung 3.5 aus.

Universal-Tabelle. Hier werden wieder alle Kanten in einer Tabelle abgelegt. Dies würde einem *Full Outer Join* der Edge-Tabelle (Abbildung 3.3) entsprechen. Die Universal-Tabelle weist folgendes Schema auf:

$$Universal(source, ordinal_{n_1}, flag_{n_1}, target_{n_1}, ordinal_{n_2}, flag_{n_2}, \ldots)$$

Für das XML-Beispiel hat die Tabelle die Ausprägung aus Abbildung 3.6. Auffallend sind hier bereits die vielen Null-Werte und die hohe Redundanz (etwa sind Peter und John zweimal in der Spalte $targ_{name}$ vorhanden).

source	...	ord_{name}	$targ_{name}$	...	ord_{child}	$targ_{child}$	ord_{hobby}	$targ_{hobby}$
1	...	2	Peter	...	4	**3**	null	null
1	...	2	Peter	...	5	**4**	null	null
2	...	2	Mary	...	4	**4**	5	painting
3	...	2	John	...	null	null	4	swimming
3	...	2	John	...	null	null	5	cycling
4	...	2	David	...	null	null	null	null

Abbildung 3.6: Beispiel-*Universal*-Tabelle

Abbildung der Werte

Separate Werte-Tabellen. Für jeden Datentyp wird eine eigene Tabelle angelegt. Dies entspricht der Aufteilung der `Val`-Tabelle aus Abbildung 3.2 unter Zuhilfenahme von Typinformationen.[5] Abbildung 3.4 zeigt zwei Werte-Tabellen für Strings und Integer-Werte. Allgemein entsprechen die separaten Werte-Tabellen dem Schema

$$V_{type}(vid, value).$$

Inlining. Eine offensichtliche Alternative zu den Werte-Tabellen stellt die Inlining-Technik dar. Dabei sind die Werte in die Kanten-Tabelle(n) integriert. Im Falle mehrerer Binary-Tabellen sähe unser Beispiel wie in

B_{hobby}

src	ord	val_{int}	val_{string}	target
2	5	null	painting	null
3	4	null	swimming	null
3	5	null	cycling	null

B_{child}

src	ord	val_{int}	val_{string}	target
1	4	null	null	**3**
1	5	null	null	**4**
2	4	null	null	**4**

Abbildung 3.7: *Binary*-Tabellen mit Inlining

Abbildung 3.7 abgebildet aus. Dem *flag*-Attribut kommt hierbei nun keine Bedeutung mehr zu und es kann eingespart werden.

Die Kombination der vorgestellten Abbildungsarten führt nun zu den sechs möglichen Speicherungsformen. Eine Form ist demnach der Edge-Ansatz zusammen mit separaten Werte-Tabellen (Abbildungen 3.3 und 3.4), was unserem Ausgangspunkt (`Ref`- und `Val`-Tabelle) nahekommt. Desweiteren stellt die Abbildung 3.6 eine Universal-Tabelle mit Inlining[6] dar.

Einschätzung

Zunächst zu den **Vorteilen**. Jedes denkbare (wohlgeformte) XML-Dokument kann mit den Methoden von Florescu und Kossmann gespeichert werden. Das Datenbankschema ist unabhängig von der DTD und den Entscheidungen des Benutzers. Plötzliche Änderungen der Struktur eines XML-Dokuments haben

[5] die wieder mittles DCD o.ä. beschafft werden müssen

[6] Das Element *hobby* wurde fürs Inlining hinzugenommen.

keine Auswirkungen auf das Schema. Diese Abbildungsform ist einfach und direkt.

Die **Nachteile**. Die Tabellen für den Edge- und Universal-Ansatz sind sehr umfangreich. Im Binary-Ansatz gibt es dagegen sehr viele, wenig gefüllte Tabellen. Desweiteren ist die Anzahl der Null-Werte im Universal- und Inlining-Ansatz sehr groß. Das Datenvolumen vergrößert sich zusätzlich, da Verwaltungsinformationen, wie sie in Form der target-Spalten entstehen, mit aufgenommen werden.

Es gibt auch keine Anfrageunterstützung. Anfragen und eine etwaige Restauration des Original-Dokuments müssen auf Anwenderebene vorgenommen werden. Ein Performance-Problem ergibt sich durch die Selbst-Verbunde über der Edge-Tabelle, da diese sehr viele Tupel enthält und diese Selbst-Verbunde womöglich mehrfach auftreten werden.

Als weiterer Nachteil sei die Redundanz in der Universal-Tabelle angeführt.

Fazit

Florescu und Kossmann kommen nach ihren Performance-Analysen zu dem Ergebnis, daß der Binary-Ansatz mit Inlining am besten abschneidet, wobei Inlining Verbunde in der Anfrageverarbeitung vermeidet und die restlichen Tabellen relativ klein im Gegensatz zu den anderen Formen bleiben. Der größte Nachteil bleibt jedoch, daß bei diesen Formen der Abbildung die Fähigkeiten eines DBMS infolge des starren Schemas nur wenig Nutzen bringen.

3.2.2 RDBMS - Schemaerzeugung anhand DTD

Im Gegensatz zum letzten Abschnitt wird in [STH+99] ein anderer Ansatz zur Speicherung von XML in relationalen Datenbanken vorgestellt. Voraussetzung ist hier nun die Existenz einer DTD für die Dokumente, da aus ihr jeweils ein spezifisches Datenbank-Schema erzeugt wird. Bei diesem Ansatz werden die zu lösenden Probleme vor allem durch die Komplexität der *Document Type Definition* bestimmt.

$$\begin{array}{c} (e_1,e_2)^* \rightarrow e_1^*,e_2^* \\ (e_1,e_2)? \rightarrow e_1?,e_2? \\ (e_1|e_2) \rightarrow e_1?,e_2? \end{array}$$

(a)

$$\begin{array}{c} e_1^{**} \rightarrow e_1^* \\ e_1^*? \rightarrow e_1^* \\ e_1?^* \rightarrow e_1^* \\ e_1?? \rightarrow e_1? \end{array}$$

(b)

$$\begin{array}{c} \ldots, a^*, \ldots, a^*, \ldots \rightarrow a^*, \ldots \\ \ldots, a^*, \ldots, a?, \ldots \rightarrow a^*, \ldots \\ \ldots, a?, \ldots, a^*, \ldots \rightarrow a^*, \ldots \\ \ldots, a?, \ldots, a?, \ldots \rightarrow a^*, \ldots \\ \ldots, a, \ldots, a, \ldots \rightarrow a^*, \ldots \end{array}$$

(c)

Abbildung 3.8: Transformationen der DTD-Vereinfachung, Quelle: [STH+99]

Vereinfachen der DTD

Um diese Komplexität zu verringern, werden von den Autoren DTD-Transformationen (siehe Abbildung 3.8) angegeben. Diese sind an die Bedingung gebunden, Anfragen entsprechend der Original-DTD und anhand des erzeugten

Datenbankschemas stets korrekt zu beantworten und alle DTD-konformen Dokumente abspeichern zu könnnen.

Ein Großteil der Komplexität entspringt der Element-Spezifikation in der DTD. Die Ursache hierfür sind Quantifizierung und Gruppierung. Den Vereinfachungen liegt zu Grunde, daß auf Anfrageebene lediglich die Position eines Elementes im XML-Dokument bezüglich seiner unmittelbaren Nachbarn (*siblings*) und seine Vater-Kind-Beziehung eine Rolle spielen.

Die Transformationen[7] lassen sich in drei Kategorien einteilen. Solche, die eine geschachtelte Definition auflösen (Abbildung 3.8a), verhindern z.B. daß die Konkatenation "**,**" und die Alternative "|" nicht innerhalb anderer Operationen auftauchen. Die Transformationen in Abbildung 3.8b reduzieren mehrere unäre Operationen zu einem unären Operator, und bei der dritten Art von Transformationen (Abbildung 3.8c) werden Elemente, die mehrfach vorkommen, zu einem Element zusammengefaßt. Desweiteren werden alle "+"-Operatoren in "*"-Operatoren umgewandelt.

Anzumerken bleibt, daß die relative Ordnung der Elemente verloren geht. Dies muß durch die explizite Speicherung eines Positionsfeldes selbst (siehe *ordinal*-Attribut bei Florescu & Kossmann) gelöst werden.

Schemaerzeugung

Nach der DTD-Vereinfachung ist nun zu entscheiden, wie das relationale Schema aussehen soll. Prinzipiell wird für jedes XML-Element eine Relation angelegt und dessen XML-Attribute als Attribute in die Relation aufgenommen. Diese einfache Umsetzung führt aber zu einer übermäßigen Fragmentation des

```
<!ELEMENT book (booktitle, author)>
<!ELEMENT article (title, author*, contactauthor)>
<!ELEMENT contactauthor EMPTY>
<!ATTLIST contactauthor authorID IDREF IMPLIED>
<!ELEMENT monograph (title, author, editor)>
<!ELEMENT editor (monograph*)>
<!ATTLIST editor name CDATA #REQUIRED>
<!ELEMENT author (name, address)
<!ATTLIST authorid ID #REQUIRED>
<!ELEMENT name (firstname?, lastname)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT address ANY>
```

Abbildung 3.9: Beispiel-DTD

Dokumentes. Im folgenden werden darum drei Techniken erläutert, die versuchen, die Effizienz der Speicherung zu erhöhen, indem sie z.B. die Anzahl der Relationen verringern.

Eine Beispiel-DTD ist für die folgenden Darstellungen in Abbildung 3.9 enthalten.

[7]Die in Abbildung 3.8 dargestellten Transformationen bilden aus Anschauungsgründen nur eine Obermenge, weitere wurden in [STH+99] referenziert.

Basic Inlining

Beim Basic Inlining wird versucht, soviele Nachfolger eines Elements wie möglich in derselben Relation zu speichern. Weiterhin wird aber für jedes Element eine eigene Relation angelegt.

Im Beispiel besitzt die author-Relation nun neben der *authorID* zusätzlich die Attribute *firstname*, *lastname*, *address*, wobei firstname, lastname und address auch noch als eigene Relationen existieren.

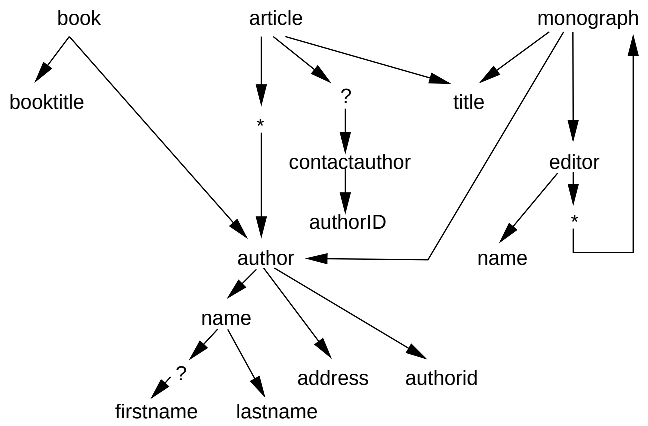


Abbildung 3.10: Graph zur DTD aus Abbildung 3.9, Quelle: [STH+99]

Zwei Fälle sind als problematisch anzusehen: eine mengenwertige Elementspezifikation und rekursive Definitionen. Da mengenwertige Attribute in einem rein relationalen DBMS nicht unterstützt werden, müssen solche Attribute in eine eigene Relation ausgelagert und über einen Fremdschlüssel eingebunden werden. Im Beispiel trifft dies auf *article* mit dem *author*-Element zu.[8]

Rekursionsbehandlung. Dazu wird die DTD in einem ersten Schritt als Graph dargestellt (Abbildung 3.10). Die Elemente, Attribute und Operatoren der DTD bilden hierfür die Knoten. Anhand der Zyklen erkennt man die Rekursionen nun besser.

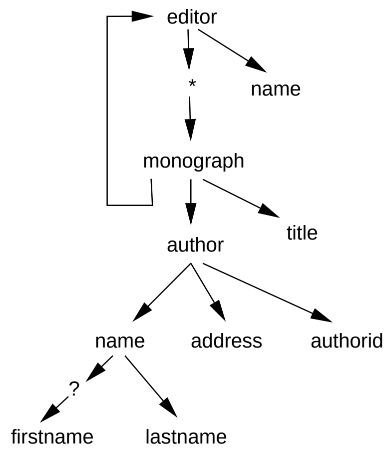


Abbildung 3.11: Element-Graph für *editor*, Quelle: [STH+99]

[8]Die vollständigen Schemata zur Beispiel-DTD sind im Anhang A zu finden.

Es folgt ein weiterer Schritt, bei dem aus dem DTD-Graphen für jeden möglichen Zyklus ein Element-Graph erstellt wird (Abbildung 3.11). Dies dient der konkreten Festlegung, für welche Elemente Relationen angelegt werden müssen. Als Startpunkt für den Element-Graphen wählt man ein beliebiges Element des DTD-Graphen als Wurzel, z.B. *editor*. Dann wird eine Tiefensuche (*depth first traversal*) im DTD-Graph vorgenommen, bei der jedes Element markiert und in den Element-Graph übertragen wird, wenn es das erste Mal durchlaufen wird. Die Markierung wird zurückgenommen, wenn alle seine Nachfolgeknoten besucht wurden.

Ein Knoten wird nur in den Element-Graph übertragen, wenn man auf einen unmarkierten Knoten trifft. Erreicht man einen bereits markierten Knoten, wird im Element-Graph vom zuletzt eingefügten Knoten eine sogenannte *Backpointer*-Kante zum bereits existierenden Zielknoten eingetragen. So entsteht ein Element-Baum mit Zyklen.

Die Relationen werden nun folgendermaßen erzeugt: Eine Relation wird für die Wurzel des Element-Graphen erzeugt. Alle darunterliegenden Knoten werden als Attribute in diese Relation aufgenommen. Davon ausgenommen sind Knoten, die auf einen "*"-Knoten folgen(mengenwertiges Attribut) und solche, auf die ein Backpointer verweist (Rekursion).

monograph (monographID: integer, monograph.parentID: integer, monograph.title: string, monograph.editor.name: string, monograph.author.name.firstname: string, monograph.author.name.lastname: string, monograph.author.address: string, monograph.author.authorid: string)
editor (editorID: integer, editor.parentID: integer, editor.name: string)
editor.monograph (editor.monographID: integer, editor.monograph.parentID: integer, editor.monograph.title: string, editor.monograph.author.name.firstname: string, editor.monograph.author.name.lastname: string, editor.monograph.author.address: string, editor.monograph.author.authorid: string)
author (authorID: integer, author.name.firstname: string, author.name.lastname: string, author.address: string, author.authorid: string)

Abbildung 3.12: Relationen zum Element-Graph für *editor*

Die entstehenden Relationen sind für *editor* (siehe Beispiel) in Abbildung 3.12 zu sehen. Die Bezeichnungen der Attribute bestimmen sich aus dem Pfad vom Wurzelelement der Relation ausgehend. Jede Relation besitzt einen künstlichen Primärschlüssel und einen Fremdschlüssel (parentID), der die Relationen miteinander verbindet.

Offensichtlich steht bei der (Basic-)Inlining-Technik wieder die Vermeidung von Verbunden bei Anfragen im Vordergrund.

Shared Inlining

Da die Anzahl der Relationen beim Basic Inlining immer noch sehr hoch ist (14 Relationen für 10 Element-Definitionen), versucht die Methode des Shared Inlining, diesen Nachteil zu verbessern. Dazu werden Elemente, die in mehreren Tabellen vorkommen, separat gespeichert und diese Relationen mit den anderen geteilt (*shared*).

Dazu wird, ausgehend vom DTD-Graph, die Anzahl eingehender Knoten bestimmt. Knoten, die nur eine eingehende Kante besitzen, werden in andere Relationen integriert (*inlined*). Knoten aber, die keine oder mehr als eine eingehende Kante besitzen, bilden jeweils eine eigene Relation.

Mengenwertige Attribute werden wieder wie beim Basic Inlining behandelt. Bei Elementen, die rekursiv zueinander in Beziehung stehen und nur eine eingehende Kante haben, wird ein Partner in einer separaten Relation gespeichert. Diese rekursiven Element-Beziehungen können gefunden werden, wenn nach streng verbundenen Elementen im DTD-Graphen gesucht wird.

monograph (monographID: integer, monograph.parentID: integer,
monograph.parentCODE: integer, monograph.editor.isroot: boolean,
monograph.editor.name: string)
author (authorID: integer, author.parentID: integer, author.parentCODE: integer,
author.name.isroot: boolean, author.name.firstname.isroot: boolean,
author.name.firstname: string, author.name.lastname.isroot: boolean,
author.name.lastname: string, author.address.isroot: boolean, author.address: string,
author.authorid: string)

Abbildung 3.13: Die Relationen *monograph* und *author* beim Shared Inlining

Im Beispiel entstehen jetzt nur fünf Relationen, wobei die *author*-Information jetzt gebündelt nur in der *author*-Tabelle steht, und nicht wie vorher in *book*, *article.author*, *monograph* usw. (Abbildung 3.13).[9]

Vorteilhaft an dieser Technik ist, daß bei Anfragen weniger Relationen bearbeitet werden müssen, was ebenfalls die Anzahl der Verbunde reduziert. Beim Basic Inlining zeigen Anfragen, die bei einem bestimmten Sub-Element starten, allerdings eine bessere Performance als beim Shared Inlining (deshalb die *isRoot*-Attribute in Abbildung 3.13, siehe auch [STH+99]).

Hybrid Inlining

Das Hybrid Inlining verringert die Anzahl der Relationen noch weiter. Die Vorgehensweise entspricht der des Shared Inlining mit der Ausnahme, daß auch Elemente, deren Anzahl eingehender Kanten größer als Eins ist, die aber weder rekursiv noch mengenwertig sind, integriert (inlined) werden.

Im Beispiel sind so nach dem Hybrid Inlining nur noch 4 statt 5 Relationen vorhanden, da *title* integriert wurde.

Einschätzung

Vorteile. Durch die Berücksichtigung der DTD spiegelt sich die Struktur der XML-Dokumente im Datenbankschema wider. Dadurch erreicht man eine wesentlich bessere Unterstützung für Anfragen, als dies bei Florescu & Kossmann der Fall war. Die Anzahl der Verbundoperationen verringert sich z.B. schon beim Basic Inlining. Die beiden weiterführenden Techniken Shared und Hybrid Inlining verkleinern zusätzlich das Datenbankschema.

[9] Auch zum Shared und Hybrid Inlining sind die vollständigen Schemata im Anhang A aufgeführt.

Um den Aufwand zur Erzeugung der Relationenschemata zu verringern, wurden DTD-Transformationen eingeführt. Trotz dieser Vereinfachungen lassen sich alle DTD-konformen Dokumente in den Relationen ablegen.

Der Problemfall der rekursiven Definitionen in der DTD wurde durch die Bildung von Graphen und einen darauf angewendeten Algorithmus zur Zyklenerkennung gut gelöst.

Nachteile. Auch diese Speicherungsform bietet keine direkte Unterstützung für Anfragen an XML-Dokumente an, wenn auch die Formulierung der Anfragen auf SQL-Ebene wesentlich leichter geworden ist.

Das erzeugte Datenbankschema ist gerade beim Basic Inlining unübersichtlich groß und es müssen auch hier Verwaltungsinformationen (isRoot, Attribute für die Ordnung der Elemente) aufgenommen werden. Beim Basic Inlining ist nicht klar, ob die Daten nicht-redundant gespeichert werden. Dies würde für den Fall zutreffen, daß die zu der Relation eines Elementes (z.B. *author*) gehörenden Attribute auch in der Relation eines anderen Elementes (z.B. *editor*) auftreten und die *author*-Daten eventuell zweimal abgelegt werden (siehe Abbildung 3.12).

Die Rekonstruktion der Original-XML-Dokumente ist aufgrund der DTD-Vereinfachungen nicht mehr möglich.[10]

Fazit

Mit dem Ansatz aus [STH+99] wurde eine adäquate Abbildung von XML auf Relationen gefunden. Allgemein hat sich die zugrundeliegende relationale Technik praktisch auch bewährt, doch bieten objektrelationale DBMS mit ihren Erweiterungen noch bessere Möglichkeiten zur Speicherung speziell von XML-Daten an. Diese Möglichkeit soll im Abschnitt 3.3.1 behandelt werden.

3.3 Objektorientierte Speicherungsformen

Wie am Anfang dieses Kapitels bereits erwähnt, bieten OODBMS reichhaltigere Konzepte zur Darstellung von XML als RDBMS an. Nachfolgend soll auf diese Möglichkeit nun jedoch nicht eingegangen werden - statt dessen wird ein objektrelationaler Ansatz vorgestellt, der von M. Klettke und H. Meyer am Lehrstuhl Datenbank- und Informationssysteme des Fachbereichs Informatik der Universität Rostock entwickelt wurde.

3.3.1 Objektrelationaler Ansatz

ORDBMS basieren zumeist auf einer Form des Modells der geschachtelten Relationen (NF^2). Allein die damit verbundenen Eigenschaften erlauben bereits eine wesentlich natürlichere Abbildung von XML auf Datenbankstrukturen. Voraussetzung ist auch hier das Vorhandensein einer DTD, aus der, ähnlich wie in Abschnitt 3.2.2, DTD-spezifische Relationenschemata abgeleitet werden. Vorteilhaft ist unter anderem, daß man jetzt mengenwertige Attribute benutzen kann, um die "*"- und "+"-Quantifikatoren direkt zu realisieren.

[10] Der Rekonstruktion soll in dieser Arbeit nur wenig Bedeutung zugemessen werden.

Die Autoren stellen zwei vereinfachende Bedingungen: Sie fordern neben den Standard-Datentypen string, numeric,... lediglich die zwei Typkonstruktoren *set-of* und *tuple-of*. Desweiteren müssen die erzeugten Relationen der *partitioned normal form* (PNF) als Spezialfall des NF^2-Modells genügen, die auf jeder Schachtelungsebene mindestens ein atomares Schlüsselattribut vorsieht. Letztendlich ist noch der komplexe Datentyp "XML" vorgesehen, der neben Anfragen mit Pfadausdrücken auch Volltext-Operationen erlauben soll. Damit handelt es sich hier um eine hybride Lösung, bei der die Daten teilweise in Relationen und zum anderen Teil im XML-Datentyp abgelegt werden.

Die Abbildung

Das allgemeine Vorgehen besteht darin, soviele XML-Elemente und -Attribute wie möglich auf die Attribute einer PNF-Relation abzubilden.

Einfache Elemente werden durch einen Bezeichner und einen Typ (z.B. #PCDATA oder auch EMPTY,...) definiert. Sie werden als Attribute vom Typ String in einer Relation gespeichert. Als Beispiel[11] diene das DTD-Fragment

```
<!ELEMENT author (first, second, e-mail?)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT second (#PCDATA)>
<!ELEMENT e-mail (#PCDATA)>,
```

das auf das Schema

author = <**first**, **second**, e-mail>

abgebildet wird. **first** und **second** sind der Schlüssel der Relation.

Optionale Elemente (mit '?' gekennzeichnet) entsprechen Attributen, die NULL-Werte besitzen dürfen (e-mail). *XML-Attribute* werden ebenfalls auf Attribute der Relation abgebildet. So ergibt sich aus

```
<!ELEMENT part (ptitle, chapter+)>
<!ATTLIST part id ID #REQUIRED>
```

das Schema

part = <**id**, ptitle, {chapter}>.

Hier sieht man auch schon, daß *quantifizierte Elemente* (mit "*" oder "+") direkt auf mengenwertige Attribute abgebildet werden, wobei für "+"-Elemente keine NULL-Werte erlaubt sein sollen.

Die *Kombination von Element-Definitionen* zeigt das folgende Beispiel. Das DTD-Fragment

```
<!ELEMENT chapter (ctitle, section+)>
<!ATTLIST chapter id ID #REQUIRED>
```

[11] Die in diesem Abschnitt verwendeten Beispiele stammen alle aus der im Anhang A.2 angegebenen DTD, die auch von den Autoren ([KM99]) verwendet wurde und im Anhang noch einmal zusammenhängend präsentiert sei.

```
<!ELEMENT ctitle (#PCDATA)>
```

ergibt das Schema

$$\text{chapter} = <\textbf{id}, \text{ctitle}, \{\text{section}\}>.$$

In ähnlicher Weise wird

```
<!ELEMENT section (stitle, parapgraph+)>
<!ATTLIST section id ID #REQUIRED>
<!ELEMENT stitle (#PCDATA)>
```

zu

$$\text{section} = <\textbf{id}, \text{stitle}, \{\text{paragraph}\}>.$$

Die beiden Schemata für *chapter* und *section* lassen sich nun zu folgendem Ergebnis kombinieren:

$$\text{chapter} = <\textbf{id}, \text{ctitle}, \{<\textbf{id}, \text{stitle}, \{\text{paragraph}\}> \}>.$$

Eine ähnliche Kombination ergibt sich mit den beiden Schemata

$$\text{conference} = <\textbf{conftitle}, \{\text{editor}\}, \text{city}, \text{year}>$$
$$\text{editor} = <\textbf{first}, \textbf{second}, \text{e-mail}>$$

zu

$$\text{conference} = <\text{conftitle}, \{<\textbf{first}, \textbf{second}, \text{e-mail}>\}, \text{city}, \text{year}>.$$

Eine Instanz für das letzte Beispielschema ist in Abbildung 3.14 zu sehen.

conference	**conftitle**	**editors**			**city**	**year**
		first	**second**	**e-mail**		
	VLDB '99	Malcom	Atkinson	...	...	'99
		Maria E.	Orlowska	...		
		Patrik	Valduriez	...		
		Stan	Zdonik	sbzcs.brown.edu		
		Michael	Brodie	...		
	ADL '98	...	...	...	...	'98
		...				

Abbildung 3.14: PNF-Relation zum Beispiel, Quelle: [KM99]

Alternativen

Für die Abbildung einer Alternative("|") in der DTD gibt es keine einfache Lösung. Man kann hier drei Fälle unterscheiden. Wenn es sich, **erstens**, nur um einfache Elemente handelt, die nur mit #PCDATA, EMPTY, ... definiert werden, so können diese Elemente alle in einer Relation zusammengefaßt werden. Für

```
<!ELEMENT address (ort, ((street, no) | po-box)>
<!ELEMENT ort (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT no (#PCDATA)>
<!ELEMENT po-box (#PCDATA)>
```

könnte die zugehörige Relation so aussehen:

`address`

ort	**street**	**no**	**po-box**
Rostock	A.-Einstein-Str.	21	NULL
Rostock	NULL	NULL	1220
London	NULL	NULL	4800

Der Nachteil dieser ersten Variante ist die Entstehung vieler NULL-Werte. Sind in der DTD für ein Element viele Alternativen angegeben, so steigt der Anteil an NULL-Werten sehr stark. Vorteilhaft ist aber, daß Anfragen vereinfacht werden, da bereits alle Attribute in *einer* Relation vorhanden sind.

Eine andere Art, solche einfachen Alternativen zu behandeln, ist die Einführung eines Attributs, dessen Wertebereich die Namen der (alternativen) Elemente sind, und eines zweiten Attributs, das die entsprechenden Werte (vom Typ #PCDATA) aufnimmt.

Das Problem der NULL-Werte ist nun beseitigt. Während man in der ersten Variante aber auch komplexere Elemente in der Relation unterbringen kann (relationenwertige Attribute), so müssen bei diesem '2-Attribute'-Ansatz nun alle Elemente den gleichen Typ besitzen.

Bei der **zweiten** Form der Abbildung der Alternativen wird für jede Alternative eine eigene Relation angelegt. Somit vermeidet man ebenfalls NULL-Werte und die Elemente können unterschiedlich strukturiert sein. Für das letzte DTD-Fragment würden beispielsweise die drei Relationen

`address`

OID	**ort**
o1	Rostock
o2	Rostock
o3	London

`address.po-box`

F_OID	**po-box**
o2	1220
o3	4800

`address.street_no`

F_OID	**street**	**no**
o1	A.-Einstein-Str.	21

entstehen. Es ist leicht zu sehen, daß hier bei Anfragen wieder erhöhter Aufwand durch Joins entsteht.

Die **dritte** Abbildungsform bildet die Alternativen komplett auf ein Attribut vom Typ XML ab. Alle Elemente der Alternative sowie deren Sub-Elemente sind hierin enthalten. Darum sind Anfragen nun nur noch mit den Operationen des XML-Datentyps realisierbar.

Um einen Kompromiß zwischen vollständiger Speicherung als Typ XML und der Aufspaltung in Relationen zu finden, bieten die Autoren in ihrem

Artikel ([KM99]) eine Möglichkeit an, mittels einer Gewichtsbestimmung solch eine Entscheidung zu treffen.[12,13]

Rekursion

Bei der Behandlung von Rekursionen wird ein ähnlicher Ansatz wie bei der rein relationalen Speicherungsform [STH$^+$99] gewählt. Man erstellt einen DTD-Graphen, startet bei der Wurzel und markiert hier aber genau die Elemente, die auf ein bereits besuchtes Element zeigen. Für jedes markierte Element wird eine eigene Relation angelegt. Für das (etwas abstraktere) Beispiel

```
<!ELEMENT Elem (Elem1, Elem2, Elem4)>
<!ELEMENT Elem1 (#PCDATA)>
<!ELEMENT Elem2 (Elem3, Elem4*)>
<!ELEMENT Elem3 (#PCDATA)>
<!ELEMENT Elem4 (Elem5, Elem2?)>
<!ELEMENT Elem5 (#PCDATA)>
```

entstehen so die drei Relationen

`Elem.Elem2`

Elem2.OID	Elem2	
	Elem3	Elem4.F_OID
o21	...	o42
o22		

`Elem.Elem4`

Elem4.OID	Elem4	
	Elem5	Elem2.F_OID
o41	...	o21
o42		

`Elem`

OID	Elem1	Elem2	Elem4
o1	...	o22	o41
o2			

.

Zusammenspiel

Zu einer gegebenen DTD werden die PNF-Schemata folgendermaßen (top-down) erzeugt. Von einem Startelement (publications, siehe DTD im Anhang) wird der Elementname zum Relationennamen. Dann werden, entsprechend den bereits erläuterten Abbildungsvorschriften, die (Sub-)Elemente als Attribute in das Relationenschema aufgenommen. Im Fall von publications trifft man zunächst auf eine Alternative, die in

$$\text{publications} \;=\; \text{book} \;\cup^{+}\; \text{article} \;\cup^{+}\; \text{conference}$$

[12] Um das Gewicht realistisch zu gestalten, fließen statistische Werte über die Daten und auch über die Anfragen ein (siehe [KM99]).

[13] Die Gewichtsbestimmung dient nicht nur dem Alternativenproblem, sondern erlaubt ganz allgemein eine Aufteilung des Dokuments in Bereiche, die als XML, und Teile, die in Relationen gespeichert werden sollen.

zerlegt wird.[14] Anschließend ergeben sich für book, article und conference die Schemata

$$\begin{aligned}\text{book} &= \{< \textbf{front}, \text{body}, \text{references} >\}\\ \text{article} &= \{< \textbf{meta}, \text{body}, \text{references} >\}\\ \text{conference} &= \{< \textbf{conftitle}, \{\text{editor}\}, \text{city}, \text{year} >\}.\end{aligned}$$

Trifft man während des "Abstiegs" in der DTD auf ein Konstrukt, das nicht nach den obigen Regeln integriert werden kann oder soll, wird an dieser Stelle ein Attribut vom Typ XML erzeugt. Alle folgenden Sub-Elemente (der restliche Teilbaum) sind dann ebenfalls in diesem Attribut enthalten. Solch ein Konstrukt stellt in XML zum Beispiel *Mixed Content* dar, zumal auch eine Alternative so realisiert werden kann.

An dieser Stelle sei erwähnt, daß die Autoren mit dieser Abbildung ein semiautomatisches Verfahren entwickelt haben, da für die Problemfälle (Alternativen, Rekursion, Mixed Content) mehrere Realisierungsarten vorgeschlagen wurden, und die Entscheidung, welche Variante eingesetzt wird, letztendlich immer noch anhand der konkreten DTD, dem konkreten Datenvolumen und dem Einsatzzweck getroffen werden muß. Dennoch läßt sich dieses Verfahren auch automatisieren.

Einschätzung

Durch den Einsatz von PNF-Relationen lassen sich die XML-Strukturen wesentlich natürlicher auf ein DBS abbilden, als dies beim reinen Relationenmodell möglich war. Die Ausnutzung der "moderneren" Konzepte erlaubt eine Ausreizung heutiger komerzieller DBMS wie Informix Universal Server und (eingeschränkt) auch DB2 von IBM. Für die Erzeugung der Datenbankschemata ist wieder die DTD nötig.

Die **Nachteile.** Durch die Art der Speicherung ist eine Rekonstruktion der Originaldokumente nur schwer bzw. gar nicht möglich. Dies ist jedoch nicht vordergründig zu bewerten. Schwerwiegender ist dagegen, daß die Anfragemöglichkeiten bisher nicht betrachtet wurden. Prinzipiell kann man vom SQL-Dialekt des zugrundeliegenden DBMS ausgehen, das sich seinerseits im günstigsten Fall bereits auf eine Anfragealgebra für geschachtelte Relationen stützt.

Abgekoppelt ist damit der XML-Datentyp. Für die im XML-Datentyp enthaltenen Informationen ließen sich Text-Retrieval-Methoden ausnutzen, die entweder manuell, in Form eines eigenständigen IR-Systems mit dem DBMS gekoppelt werden könnten ([VAB96]), oder als Extender im DBMS (siehe Text-Extender von DB2 und [Por99]) bereits integriert sein könnten. Da der XML-Datentyp hierbei immer noch nur als BLOB/CLOB betrachtet wird, wäre es sinnvoll, wenn er einen eigenen reichhaltigen Funktionsumfang besitzen würde.

[14] Das $\cup^+$ soll zum Ausdruck bringen, daß es sich hier nicht um die Vereinigung (der Anfragealgebra) handelt, die nur für typ-gleiche Schemata definiert ist. Die Tatsache, daß book, article und conference Publikationen sind, muß daher (noch) auf Anwenderebene, z.B. im Anwendungsprogramm, realisiert werden.

Da zur Anfrageproblematik zur Zeit noch keine Betrachtungen erfolgt sind, ist der Benutzer wieder wie bisher gezwungen, bei Anfragen die Realisierung der Abbildung zu beachten.

Problematisch ist auch der Einsatz des XML-Datentyps an sich zu sehen. Die Vorgehensweise bei der Schemaerzeugung ist top-down vom Wurzel-Element der DTD ausgehend. Eine DTD wird somit prinzipiell auf eine einzige Relation abgebildet, die zwar stark strukturiert ist, aber alle zu der DTD konformen Dokumente aufnehmen muß. Ein anderer Aspekt betrifft den Einsatz des XML-Typs überhaupt. Wird nämlich an einer Stelle in der DTD festgelegt, ein bestimmtes Element als XML zu speichern, müssen alle Sub-Elemente und Nachfahren ebenfalls in dieses XML-Attribut aufgenommen werden. Besser strukturierte Anteile, die sich nun in diesem Teilbaum befinden, können nicht mehr in einer Relation sichtbar gemacht werden und entziehen sich den Anfragemöglichkeiten des DBMS (bzw. der PNF-Algebra).[15]

3.4 Zusammenfassung

Um die Vorteile der Datenbank-Technik nutzen zu können, wurden verschiedene Ansätze präsentiert, wie XML in einem DBS gespeichert werden kann. Der Schwerpunkt lag dabei im Bereich relationaler Datenbanken, da sich diese Technik bereits bewährt hat (kommerziell erfolgreich ist) und daher am naheliegendsten war.

Das grundlegende Problem bei der Realisierung in einer Datenbank war die Abbildung der hierarchischen XML-Struktur auf eine (flache) Tabellenstruktur. Alle vorgestellten Ansätze konnten dieses Problem gröstenteils auf einfache Art und Weise bewältigen. Der Ansatz von Florescu & Kossmann erzeugte dazu ein von einer DTD unabhängiges Datenbank-Schema, während die anderen Ansätze mittels der DTD ein spezifisches Schema nur für DTD-konforme XML-Dokumente entwarfen. Hervorzuheben ist, daß sich die XML-Struktur besonders gut in den geschachtelten Relationen des objektrelationalen Ansatzes wiederspiegelt.

Alle Ansätze ließen eine Unterstützung für Anfragen vermissen. Nur mittels SQL konnten Anfragen formuliert werden. Dabei war der Aufwand, die Informationen des XML-Dokuments zu extrahieren, teilweise erhelblich (Selbst-Verbunde großer Tabellen).

[15]Zunächst sei davon auszugehen, daß Informationen im XML-Attribut nicht über die Operatoren einer Anfragealgebra erreichbar sind. Sie sind somit für die Anfrageformulierung auf Algebraebene verloren, obwohl sie in einigen Fällen gut strukturiert sind und angefragt werden könnten.

Kapitel 4

Eine XML-Anfragesprache

In den zurückliegenden Kapiteln wurde die Basis für diesen eigentlichen Hauptteil der Arbeit geschaffen. Kapitel 4 beschäftigt sich nun intensiver mit einer zukünftigen XML-Anfragesprache. Dies wird jedoch kein kompletter Sprachentwurf sein, vielmehr folgen zunächst Anforderungen an eine Anfragesprache für XML, bevor dann konkrete Vorschläge unterbreitet werden.

4.1 Anforderungen

Die Anforderungen an eine XML-Anfragesprache lassen sich grob in vier Bereiche einteilen: zum einen in allgemeine Anforderungen (Vollständigkeit, Abgeschlossenheit,...), zum anderen in einen Bereich, der die spezifischen Eigenheiten von XML betrifft, zum dritten in Funktionalität für das Information-

Kriterium	anwendbar bzgl. XML
Ad-Hoc-Formulierung	+
Deskriptivität	+
Mengenorientiertheit	o
Abgeschlossenheit	+
Adäquatheit	+
Orthogonalität	+
Optimierbarkeit	+
Effizienz	+
Sicherheit	–
Eingeschränktheit	–
Vollständigkeit	+

Abbildung 4.1: Allgemenine Anforderungen und ihr Bezug zu XML

Retrieval, und zuletzt in einen "physischen" Bereich, mit dem ein effizientes und flexibles Speichermodell gefordert wird.

In Abbildung 4.1 wurden einmal die allgemeinen Anforderungen an Datenbank-Anfragesprachen aufgelistet und ihre Gültigkeit im Fall von XML eingetragen. Ein + bedeutet dabei, daß dieses Kriterium anwendbar ist, o steht für eingeschränkt anwendbar, und – für vernachlässigbar.[1]

[1] Die Kriterien wurden [HS95] entnommen.

Die Mengenorientiertheit ist eingeschränkt worden, weil sie nicht fordert, ein XML-Dokument durchweg mengenorientiert zu gestalten, sondern nur auf der obersten Ebene des Ergebnisses, und, um die Ordnung der XML-Elemente auch bei der Berechnung von Zwischenergebnissen aufrecht zu erhalten. Aus anwendungstechnischen Gründen kann auf die Ordnung aber verzichtet werden (Zwischenergebnisse sind dann ebenfalls Mengen).

Abgeschlossenheit bedeutet die Konstruktion eines XML-Dokumentes als Ergebnis. Aus der Ergebnismenge ist wieder ein gültiges XML-Dokument zu erzeugen, z.B. durch Einführung eines künstlichen Wurzelelementes. Das Kriterium Abgeschlossenheit ist erfüllt, wenn Eingabe- und Ausgabe-Daten in XML-Form sind.

Die Sicherheit einer Anfragesprache wird gefordert, um eine Terminierung bei der Auswertung der Anfrage zu garantieren. Zwar gibt es in der DTD rekursive Definitionen, die Daten beinhalten jedoch selbst keine Rekursion.[2] Bei der Verfolgung der XML-Links müssen Endlosschleifen aber vermieden werden.

Weitere XML-spezifische Kriterien sind [Mai98]:

Einbettung. Anfragen sollen sich in XML einbetten lassen und umgekehrt könnten XML-Fragmente in einer Anfrage enthalten sein.

Ordnungserhaltend. Da XML-Elemente einer Ordnung folgen, ist diese möglichst zu beachten.

Links. Es sollen Hypertextstrukturen unterstützt werden (XML-Links über ID/IDREF(S) und weitere Formen).

Basisoperationen. Als Basisfunktionalität sollen die Selektion, Extraktion, Reduktion, Restrukturierung und Kombination angeboten werden. Die Basisoperationen bedeuten im einzelnen:

- **Selektion:** Selektion eines Dokumentes oder Dokumentelementes anhand von Inhalt oder Struktur
- **Extraktion:** gezieltes Herausziehen von Subelementen eines Dokumentes
- **Reduktion:** Löschen ausgewählter Subelemente eines Elementes
- **Restrukturierung:** Erzeugen einer neuen Elementmenge, die z.B. das Anfrageergebnis enthält
- **Kombination:** Zusammenfassen zweier oder mehrerer Elemente zu einem neuen Element

Das Kriterium *Basisoperationen* stellt eine Verfeinerung des Vollständigkeitskriteriums (siehe Abbildung 4.1) dar, da die Vollständigkeit eine ausreichende Ausdrucksstärke der Sprache garantieren soll. Auch die Kriterien *Ordnungserhaltend* und *Links* zählen zur allgemeinen Forderung nach Adäquatheit. Sie fanden jedoch in einigen bisher entwickelten Sprachen teilweise keine Beachtung.

[2] den ID/IDREF(S)-Mechanismus unbeachtet gelassen

IR-Funktionalität

Vor allem bei großen semistrukturierten Datenmengen steht Information–Retrieval als Anfrageform im Vordergrund. Eine Kopplung von Datenbank und externem Retrieval-Werkzeug ist jedoch inakzeptabel, da Anfragen immer noch getrennt gestellt werden müssen. Besonders für XML ist es wichtig, Retrieval-Funktionen in die Anfragesprache zu integrieren. Zur Information–Retrieval–Funktionalität gehören folgende Aspekte [Fuh97]:

Retrievalmodell: Aus Dokumentbeschreibungen und Fragebeschreibungen berechnet eine Retrievalfunktion das Retrievalgewicht. Das Retrievalmodell legt fest, wie die Dokumente beschrieben werden, z.B. durch einen Index. Die Retrievalfunktion wird ebenfalls festgelegt — beim Boolschen Retrieval liefert sie für ein Dokument z.B. nur die Gewichte 1 oder 0, je nachdem, ob ein Anfrageterm in der Dokumentbeschreibung vorkommt oder nicht.

Ranking: Beim Ranking wird jedem Dokument ein Meßwert zugeordnet, der die Güte zum Frageterm widerspiegelt. Beim Boolean Retrieval kann das Ranking über die Bestimmung der (Wort-)Häufigkeiten realisiert werden.

Relevance-Feedback: Bei Retrievalmodellen wie dem Fuzzy- oder Vektorraummodell kann die Retrieval-Qualität durch Relevance-Feedback verbessert werden. Eine Relevance-Beurteilung durch den Benutzer ergibt einen verbesserten Fragevektor, mit dem das Retrieval erneut durchgeführt wird.

Indexierung: Die Indexierung der Dokumente ist ein wichtiger Faktor, der die Performance positiv beeinflußt, jedoch einen erheblichen Mehraufwand für Speicher und Update-Kosten bedeutet.

Der vierte Bereich, der die Anfrageproblematik beeinflußt, ist letztendlich auch das zugrundegelegte Speicherungsmodell für die Daten. Die Algorithmen, die die Anfrageoperationen realisieren, hängen hauptsächlich von der jeweiligen Speicherungform ab. Einige dieser Formen wurden in Kapitel 3 behandelt. Der nächste Abschnitt wird diesen Aspekt noch einmal aufgreifen und weiterführen.

Eine Sprache, die all die genannten Kriterien vereint und erfüllt, stellt eine Herausforderung dar, der im Rahmen dieser Arbeit nicht in vollem Umfang Rechnung getragen werden kann. Die Herangehensweise wird jedoch, im Unterschied zu den anderen Sprachentwürfen (siehe Kapitel 2), insofern anders sein, als daß ein Weg eingeschlagen wird, der zuerst eine theoretische Grundlage schafft (eine Algebra), auf deren Basis Untersuchungen bezüglich der oben genannten Kriterien leichter vorgenommen werden können. Erst am Ende steht die Definition eines syntaktischen Layouts.

Aus Datenbank-Sicht bleibt zu den Kriterien noch ein Nachtrag. Die Unterstützung eines erweiterbaren Typkonzeptes birgt nicht nur den Vorteil einer unter Umständen besseren Speicherausnutzung, sondern gestattet auch die

Verwendung effizienterer Algorithmen bei der Anfrageauswertung.[3] Ein Punkt, über den es sich in diesem Zusammenhang auch nachzudenken lohnt, wäre der Aufbau einer Typhierachie für ein XML-Dokument.

Im folgenden Abschnitt wird noch einmal ein Blick auf das Speichermodell geworfen, da eine Entscheidung diesbezüglich Auswirkungen auf die Wahl bzw. den Entwurf einer Algebra haben wird.

4.2 Zugrundegelegte Speicherungsform

In Kapitel 3 wurde ein bezüglich XML einigermaßen adäquates Speichermodell bereits vorgestellt: der objektrelationale Ansatz von H. Meyer und M. Klettke (im folgenden mit Hybrid-Ansatz bezeichnet). Bei diesem Ansatz teilt sich die Anfragefunktionalität auf: in Anfragen über dem objektrelationalen Schema und in Anfragen über dem XML-Datentyp. Läßt man den XML-Datentyp außer Acht, bleiben "objektrelationale" Anfragen übrig, für die das Modell der geschachtelten Relationen (NF^2) gilt, mit der Einschänkung auf die *partitioned normal form* (PNF). Da sich die Baumstruktur eines XML-Dokumentes besonders gut in den geschachtelten Relationen widerspiegeln läßt, soll für die weiteren Betrachtungen der Hybrid-Ansatz als Speicherungsform zugrundeglegt werden.[4]

Für das NF^2-Modell gibt es diverse Algebren: z.B. die Algebra von Roth, Korth und Silberschatz [RKS88] oder die Colby-Algebra [Col89]. Bei den genannten Algebren handelt es sich um rekursive Algebren, d.h. daß die Bedeutung der Algebra-Operatoren in rekursiver Weise definiert wird — der Operator wird also auf jedes von der Operation betroffene Teilschema erneut angewandt. Die meisten dieser Algebren bauen auf der Relationenalgebra auf.

Die einfachste Variante einer NF^2-Algebra ist die Einführung eines Unnest- und eines Nest-Operators, die eine geschachtelte Relation in eine flache Form (1NF) überführen bzw. umgekehrt (in [HS95] skizziert). Die Operationen der Relationenalgebra können fast eins zu eins übernommen werden, wobei bei deren Anwendung oft eine Entnestung vor bzw. Nestung nach der Ausführung der anderen Operationen vorgenommen werden muß. Bei den rekursiven Algebren wird dieser Umweg über die flachen Relationen vermieden. Problematisch ist die Behandlung von Nullwerten. Außerdem ist die Nestung nur bei der PNF die inverse Operation zur Entnestung.

An dieser Stelle soll auf eine Vorstellung einer NF^2-Algebra verzichtet und stattdessen eine Erweiterung des Hybrid-Ansatzes präsentiert werden.

[3]Als einfachstes Beispiel stelle man sich an dieser Stelle den Vergleich zweier Werte(Zahlen) auf String- und auf Integer-Basis vor, wobei die Strings vor dem Vergleich vielleicht erst in einen "Zahlen"-Datentyp gewandelt werden müssen.

[4]Man geht somit einen eher minimalistischen Weg, der für ein XML-Dokument auf die Definition einer Typ- bzw. Klassenhierachie wie in einem OODBMS erst einmal verzichten will.

4.2.1 Erweiterter Hybrid-Ansatz

Einige Probleme des Hybrid-Ansatzes, wie zum Beipiel die Entscheidung zur Speicherung alternativer XML-Elemente oder die Behandlung der (Schema-) Rekursion, zwingen den Benutzer, die jeweilige Realisierung bei der Anfrageformulierung zu beachten, indem z.B. ein Verbund explizit angegeben werden muß. Ein Vorschlag für die Lösung des Alternativen-Problems soll im folgenden präsentiert werden. Da sich hieraus auch umfangreiche Änderungen für die Anfragealgebra ergeben, sollen die anderen Probleme zurückgestellt werden.

Behandlung der Alternativen

Im Relationenmodell (und damit auch im NF^2-Modell) werden Nullwerte eingeführt, um anzudeuten, daß ein Wert für ein Attribut nicht existiert bzw. dieser Wert unbekannt ist. Angenommen, alle XML-Elemente einer Alternative bilden die Attribute *einer* Relation. Bei einer aus zwei Elementen bestehenden Alternative ist dann jeweils ein Attribut NULL. Bei drei Elementen sind es bereits zwei NULL-Werte pro Eintrag, und die Anzahl der Null-Werte nimmt exponentiell zu, vergrößert sich die Anzahl der alternativen Elemente. Diesen Overhead möchte man aus zwei Gründen vermeiden. NULL-Werte müssen auch physisch repräsentiert werden und belegen damit unnötig Speicherplatz. Der wichtigere Grund ist aber, daß sich der Umgang mit Null-Werten umständ-

a) r_1

A	B
a	b
a	NULL

b) r_2

B	C
b	c
NULL	c

c)

A	B	C
a	b	c

d)

A	B	C
a	b	c
a	NULL	c

Abbildung 4.2: Verbund mit Null-Werten

licher gestaltet. Hier sei auf das Beispiel aus [HS95][5] verwiesen, bei dem ein Verbund der beiden Tabellen in Abbildung 4.2 a) und b) über dem Attribut B nicht zwangsläufig zum Ergebnis der Abbildung c) führt, sondern das Ergebnis in Abbildung d) auch liefern könnte, da man schließlich nicht weiß, ob die beiden Null-Werte in r_1 und r_2 nicht auch gleich einem b_1 sind.

Eine Methode, Null-Werte bei der Erzeugung des Schemas zu vermeiden, wäre eine Änderung des Modells. Die Einführung varianter Strukturen würde die direkte Abbildung der Alternativen auf variante Strukturen erlauben. Die Unterstützung varianter Strukturen in einem erweiterten relationalen Modell hat Ch. Kalus in [Kal95] bereits untersucht. Diese Idee wird nun skizzenhaft vorgestellt.

4.2.2 Variante Strukturen

Um die verschiedensten Formen der Varianten, also z.B. auch die Darstellung der Tatsache, daß von drei Attributen immer mindestens zwei beliebige präsent

[5] Seite 241

sein sollen, einheitlich repräsentieren zu können, gibt Kalus eine veränderte Schemadefinition an. Ein relationales Schema wird um zwei Zähler erweitert, die die Unter- und Obergrenze eines Intervalls darstellen, womit die erlaubte Anzahl der vorkommenden Attribute festgelegt wird. Ein solches erweitertes Schema ist ein Drei-Tupel

$$< k_{min}, k_{max}, \{A_1, \ldots, A_k\} >.$$

k_{min} und k_{max} bilden die Unter- bzw. Obergrenze und $\{A_1, \ldots, A_k\}$ ist die Menge der Attribute aus einem Universum $\mathcal{U}$. Die Bedeutung dieses Konstrukts besteht darin, daß alle Tupel zum Wertebereich eines solchen Schemas gehören, die n ($k_{min} \leq n \leq k_{max}$) der k spezifizierten Attribute besitzen. Folgende Ausprägungen dieser Definition sind möglich:

- Ein *relationales Schema* $\{A_1, \ldots, A_k\}$ wird auf $< k, k, \{A_1, \ldots, A_k\} >$ abgebildet. Die Wahl von $k_{min} = k_{max} = k$ legt fest, daß keine Auswahlmöglichkeit für die Attribute besteht.

- Eine *exklusive Disjunktion* der Attribute $A_1, \ldots, A_k$ wird durch das erweiterte Schema $< 1, 1, \{A_1, \ldots, A_k\} >$ dargestellt. Dieses Schema besagt, daß mindestens ein und höchstens ein, d.h. genau ein Attribut ausgewählt werden darf. Dies entspricht der exakten Modellierung einer Alternative, wie sie bei XML auftritt.

- Die *inklusive Disjunktion* korrespondiert zu $< 0, k, \{A_1, \ldots, A_k\} >$ bzw. $< 1, k, \{A_1, \ldots, A_k\} >$, wobei eine beliebige (nichtleere) Teilmenge der Attribute akzeptiert werden soll.

Andere Formen sind möglich, spielen im weiteren Verlauf jedoch keine Rolle. Als Beispiel sei noch ein einzelnes optionales Attribut angeführt, das dem erweiterten Schema $< 0, 1, \{A\} >$ entspricht.

Ein erweitertes Schema in der vorgestellten Form ist bisher einstufig. Für die Abbildung einer einzelnen Alternative ist das auch ausreichend, für ein ganzes XML-Dokument jedoch nicht. In diesem Sinn wird ein flexibles Schema nun rekursiv definiert.

Definition 4.1 *Die Menge $\mathcal{FS}$ der flexiblen Schemata ist die kleinste Menge, für die gilt*

1. $A \in \mathcal{FS}$, *wenn* $A \in \mathcal{U}$

2. $< k_{min}, k_{max}, \{FS_1, \ldots, FS_k\} > \in \mathcal{FS}$, *wenn* $k_{min}, k_{max}, k \in \mathbb{N}$,
$0 \leq k_{min} \leq k_{max} \leq k$,
$FS_1, \ldots, FS_k \in \mathcal{FS}$

Flexible Schemata sind jetzt also Baumstrukturen mit den als erweiterte Schemata bezeichneten Drei-Tupeln als innere Knoten und den Attributen als Blätter. Eine Baumstruktur ist im nächsten Beispiel zu sehen.

Mit den erweiterten Schemata für exklusive Disjunktion und relationale Schemata läßt sich nun die in Abschnitt 3.3.1 gefundene Abbildungvorschrift des Hybrid-Ansatzes anpassen. Vorerst wird dies nur an den folgenden Beispielen gezeigt.

Das DTD-Fragment

```
<!ELEMENT author (first, second, e-mail?)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT second (#PCDATA)>
<!ELEMENT e-mail (#PCDATA)>,
```

wurde in Abschnitt 3.3.1 auf das Schema

$$\text{author} = <\textbf{first}, \textbf{second}, \text{e-mail}>$$

abgebildet. Das flexible Schema ist entsprechend obigen Ausführungen nun

$$\text{author} = < 3, 3, \{\textbf{first}, \textbf{second}, \text{e-mail}\}>.$$

Einfach gesagt ist dies die relationale Form des erweiterten Schemas. Für die Abbildung einer einfachen XML-Alternative soll das DTD-Fragment

```
<!ELEMENT address1 (street | po-box)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT po-box (#PCDATA)>
```

dienen. Das flexible Schema

$$\text{address1} = < 1, 1, \{ \text{street}, \text{po-box}\}>$$

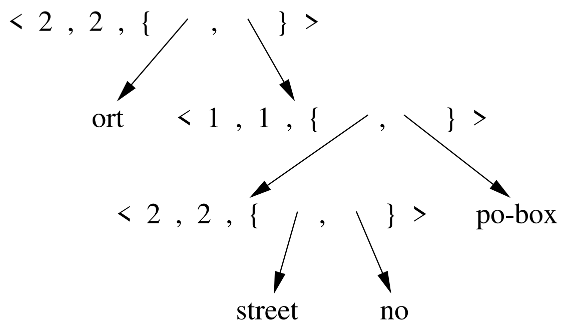
ist hier nur das erweiterte Schema für die exklusive Disjuktion. Als zweites Beispiel für eine XML-Alternative dient das DTD-Fragment

```
<!ELEMENT address (ort, ((street, no) | po-box)>
<!ELEMENT ort (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT no (#PCDATA)>
<!ELEMENT po-box (#PCDATA)>
```

Das flexible Schema ergibt jetzt sich als Konsequenz aus der Definition 4.1:

$$\begin{aligned}\text{address} = < 2, 2, \{ & \text{ort}, \\ & < 1, 1, \{ < 2, 2, \{ \text{street}, \text{no} \}>, \\ & \qquad\qquad \text{po-box} \\ & \qquad\quad \}> \\ \}> & \end{aligned}$$

Die Alternative `((street, no) | po-box)` erstreckt sich über die Zeilen 2–4: $< 1, 1, \{\text{FS}_1, \text{po-box}\} >$ mit $\text{FS}_1 = < 2, 2, \{$ `street`, `no` $\}>$. FS_1 ist das Pendant zum relationalen Schema und dient in diesem Fall nur der Gruppierung der Attribute `street` und `no`. Auf oberster Ebene ist *address* ebenfalls "relational". Das flexible Schema *address* sieht als Baumstruktur wie folgt aus:



Im Gegensatz zum Relationenmodell wird der Wertebereich flexibler Schemata (die Domäne *sdom*) definiert. Für diese und weitere Definitionen sei grundsätzlich auf [Kal95] verwiesen. Zum Wertebereich nur so viel: Das Gültigkeitsintervall eines flexiblen Schemas besagt, daß jede Kombination aus n Teiltupeln gebildet werden darf, solange $k_{min} \leq n \leq k_{max}$ gilt. Der Wertebereich *sdom(address)* des letzten Beispielschemas *address* ist die Menge aller Tupel zu allen Attributkombinationen des flexiblen Schemas und ergibt sich vereinfacht zu:

$$\begin{aligned} &sdom(<2,2,\{\text{ort},<1,1,\{<2,2,\{\text{street},\text{no}\}>,\text{po-box}\}>\}>) \\ &\quad = Tup(\text{ort},\text{street},\text{no}) \cup Tup(\text{ort},\text{po-box}) \end{aligned}$$

$Tup(X)$ bezeichnet dabei die Menge aller möglichen Tupel zu einer Attributmenge X. Eine Instanz zu diesem Schema, eine flexible Relation, ist in Ab-

`address`	**ort**	**street**	**no**	**po-box**
	Rostock	A.-Einstein-Str.	21	
	Rostock			1220
	London			4800

Abbildung 4.3: Beispielausprägung einer flexiblen Relation

bildung 4.3 zu sehen. Zu beachten ist, daß die Form der Darstellung nichts über eine mögliche physische Realisierung aussagt. Die leeren Felder besagen lediglich, daß es sich um ein nichtexistentes Attribut handelt.

Es ist vielleicht schon aufgefallen, daß die Erweiterung der Schemadefinition vom rein relationalen Modell ausgeht. Der Schritt zum Modell der geschachtelten Relationen, wie es im Hybrid-Ansatz verwendet wird, folgt aber erst im nächsten Abschnitt. Zuvor soll skizziert werden, welche Auswirkungen das erweiterte Datenmodell bisher auf die Anfragekomponente hat.

Anfragen

In [Kal95] wurde auch eine Algebra für das Flex-Modell entworfen, die Flex-Algebra. Großer Wert wurde auf die Beibehaltung der intuitiven Bedeutung der Operatoren bezüglich der Relationenalgebra gelegt. Insgesamt umfaßt die Flex-Algebra die relationalen Operatoren *Projektion*, *Selektion* und *kartesisches Produkt*, die Mengenoperationen *Vereinigung*, *Differenz* und *Schnitt*, den Erweiterungsoperator *Extension*, einen *Umbenennungsoperator* sowie den Typprüfungsoperator *Restriktion* als Basisoperatoren. Die Extension dient dazu, eine flexible Relation um ein Attribut zu erweitern. Die Restriktion spaltet sich auf in die *positive Restriktion*, die das Vorhandensein eines bestimmten Attributs gewährleistet, und die *negative Restriktion*, die die Abwesenheit des Attributs sicherstellt.

Als besondere Kriterien für die Flex-Algebra seien die Typpräzision und die Typprüfung hervorgehoben:

Typpräzision: Die Ergebnisschemata der Algebra-Operatoren sollen kleinstmögliche korrekte Schemata sein.

Typprüfung: Für die varianten Strukturen soll die Flex-Algebra neben den bekannten relationalen Operatoren auch geeignete Operatoren zur strukturellen Typüberprüfung flexibler Relationen zur Verfügung stellen.

Im folgenden soll nur auf den Projektionsoperator eingegangen werden. Die Projektion entspricht, im relationalen Sinn, dem Streichen von Spalten eines Relationenschemas. Das Ergebnis der Projektion $\pi_X(R)$ ist ein Relation, die nur auf den Attributen $X \cap attr(R)$ definiert ist. Das läßt sich formal darstellen als

$$\pi_X(r(R)) \quad = \quad \{\, t(X) \mid t \in r(R) \,\}$$

oder mit den beiden Funktionen $sch(R)$ und $inst(R)$, mit denen auf das Schema bzw. die Instanz zugegriffen wird:

$$\begin{aligned} sch(\pi_X(R)) &= X \cap sch(R) \\ inst(\pi_X(R) &= \{\, t(X \cap schR) \mid t \in inst(R) \,\} \end{aligned}$$

Der Projektionsoperator der Flex-Algebra unterscheidet sich in seiner Bedeutung nicht davon, er entfernt dementspechend die Spalten einer flexiblen Re-

ort	**street**	**no**	**po-box**
Rostock	A.-Ein....	21	
Rostock			1220
London			4800

$$\xrightarrow{\pi^f_{\{ort,no,po-box\}}(addr)}$$

ort	**no**	**po-box**
Rostock	21	
Rostock		1220
London		4800

Abbildung 4.4: Beispiel für den Projektionsoperator π^f der Flex-Algebra

lation. Dies sei beispielhaft in Abbildung 4.4 angedeutet. Hierbei muß darauf hingewiesen werden, daß die Projektion auf {ort, no, po-box} nicht bedeutet, daß die Resultattupel auf jedem dieser Attribute definiert sind. Die Instanzdefinition ergibt sich daher zu

$$inst(\pi^f_X(FR)) \quad = \quad \{\, t(X \cap attr(t)) \mid t \in inst(FR) \,\}$$

wobei $attr(t)$ die Funktion ist, die zu einem Tupel dessen Attribute bestimmt. Ein Vergleich mit dem relationalen Fall bestätigt diese Definition, da dort $attr(t) = sch(R)$ gilt.

Um die Schemadefinition bestimmen zu können, bedarf es einer flachen Darstellung eines flexiblen Schemas. Die flache Form entspricht einer Menge von Attributmengen und ist für das Beispielschema

$$FS =< 2, 2, \{\text{ort}, < 1, 1, \{< 2, 2, \{\text{street}, \text{no}\} >, \text{po-box}\} >\} >$$

die Menge, bezeichnet mit $dnf(FS)$:

$$dnf(FS) = \{ \{\text{ort}, \text{strasse}, \text{no}\}, \{\text{ort}, \text{po-box}\} \}$$

Die enthaltenen Mengen werden jeweils mit der Menge der Projektionsattribute $X = \{$ort, no, po-box$\}$ geschnitten. Es ergibt sich

$$dnf(\pi^f_X(FS)) = \{\{\text{ort}, \text{no}\}, \{\text{ort}, \text{po-box}\} \}$$

Das Schema kann aus der Rücküberführung der flachen Darstellung gewonnen werden und ist

$$\begin{aligned} sch(\pi^f_X(FS)) &= < 2, 2, \{\text{ort}, < 1, 1, \{< 1, 1, \{\text{no}\} >, \text{po-box}\} >\} > \\ &= < 2, 2, \{\text{ort}, < 1, 1, \{\text{no}, \text{po-box}\} >\} > \end{aligned}$$

Die letzten Ausführungen verdeutlichen, daß die haupsächliche Aufgabe für die Festlegung der Flex-Algebra in der Schemadefinition der einzelnen Operatoren besteht.

Bevor im nächten Abschnitt der Schritt vom Flex-Modell zum Flex-NF^2-Modell vollzogen wird, sei noch ein Wort zur Optimierbarkeit im Flex-Modell angemerkt. Die Verwandschaft zum Relationenmodell macht die für die relationale Algebra entworfenen Optimierungstechniken übertragbar. Es müßten daher insbesondere die Optimierung strukturbasierter Anfragen und Anfragen, die an entlang ihrer Struktur partitionierte Relationen gestellt werden, betrachtet werden.

4.2.3 Flexible NF^2-Relationen

Im objektrelationalen Ansatz in Kapitel 3 wurden die XML-Quantifikatoren "*" und "+" auf mengenwertige Attribute abgebildet. Im Flex-Modell gibt es jedoch keine mengen- bzw. relationenwertigen Attribute. Die Lösung hierfür würde eine Kombination von Flex- und NF^2-Modell bilden.[6] Im folgenden soll erläutert werden, was ein komplexes flexibles Schema im Flex-NF^2-Modell ist.

Zur Beschreibung von flexiblen NF^2-Schemata kann man versuchen, die Definition eines flexiblen Schemas (Definition 4.1) mit der Definition eines NF^2-Schemas zu kombinieren. Dazu sei zunächst die Definition für ein NF^2-Schema angegeben, wobei $\mathcal{NA}$ die Menge der geschachtelten Attribute *"nested attributes"* bezeichnet, die neben ihren Namen noch eine Beschreibung ihrer Komponenten enthalten.

Definition 4.2 *Die Menge $\mathcal{NS}$ der NF^2-Schemata enthält alle Mengen der Form $\{A_1(NS_1), \ldots, A_n(NS_n)\}$*

$$mit \begin{cases} A_i \in \mathcal{U} \quad \wedge NS_i = \emptyset \quad \textit{(atomare Attribute)} \\ oder \\ A_i \in \mathcal{NA} \wedge NS_i \in \mathcal{NS} \quad \textit{(komplexe Attribute)} \end{cases} \qquad (i = 1..n)$$

[6] In [Kal95] wurde auch an diese Möglichkeit gedacht, jedoch nur in Grundzügen skizziert.

Ein NF^2-Schema besteht aus einer Menge von Termen, die entweder atomare oder komplexe Attribute sein können, wobei ein komplexes Attribut wiederum ein NF^2-Schema als Teilschema besitzt.

Ein flexibles Schema (Definition 4.1) besteht entweder aus einem atomaren Attribut oder einem Drei-Tupel , in dem flexible Schemata als Teilschemata enthalten sind. Die Kombination der Definitionen 4.1 und 4.2 ergibt die Definition von $\mathcal{NFS}$, der Menge aller Flex-NF^2-Schemata:

Definition 4.3 *Die Menge $\mathcal{NFS}$ der komplexen flexiblen Schemata ist beschrieben durch*

1. $A \in \mathcal{NFS}$, *wenn* $A \in \mathcal{U}$
2. $A(NFS) \in \mathcal{NFS}$, *wenn* $A \in \mathcal{NA}$ *und* $NFS \in \mathcal{NFS}$
3. $< k_{min}, k_{max}, \{NFS_1, \ldots, NFS_k\} > \in \mathcal{NFS}$, *wenn*
$$k_{min}, k_{max}, k \in \mathbb{N},$$
$$0 \leq k_{min} \leq k_{max} \leq k,$$
$$NFS_1, \ldots, NFS_k \in \mathcal{NFS}$$

Zur Demonstration der nun erlangten Ausdrucksmächtigkeit soll das Beispiel aus Abschnitt 3.3.1 (Seite 38) aufgegriffen werden. Das Schema für das XML-Fragment

```
<!ELEMENT publications ( book | article | conference)*>
<!ELEMENT book (front, body, references)>
<!ELEMENT article (meta, body, references)>
<!ELEMENT conference (editor+, conftitle, city, year)>
```

ist salopp formuliert

$$\begin{aligned} \text{publications} &= \text{book } \cup^+ \text{ article } \cup^+ \text{ conference} \\ \text{book} &= \{\text{front}, \text{body}, \text{references}\} \\ \text{article} &= \{\text{meta}, \text{body}, \text{references}\} \\ \text{conference} &= \{\text{conftitle}, \{\text{editor}\}, \text{city}, \text{year}\}. \end{aligned}$$

Im Flex-NF^2-Modell hätte das Schema folgende Gestalt:

< 1, 1, { book (< 3, 3, { front, body, references }>),
article (< 3, 3, { meta, body, references }>),
conference (< 4, 4, { conftitle, editor(editor'), city, year }>
}>

Es ist erkennbar, daß die Alternative für *publications* direkt durch das mit dem Flex-Modell eingeführte Konstrukt (exklusive Disjunktion) realisiert werden kann. Die in *publications* enthaltenen Elemente sind alle drei mengenwertig, so daß für sie die im NF^2-Modell eingeführte Notation angewendet wird (book, article, conference $\in \mathcal{NA}$). Die weiteren Schemata sind schließlich die relationale Flex-Form (mit $k_{min} = k_{max}$), wobei editor wiederum mengenwertig ist und dafür ein künstliches Attribut editor' eingeführt wurde. Auf eine Instanz zu diesem Schema sei verzichtet.

Mit dem Flex-NF^2-Modell besitzt man nun ein mächtigeres Datenmodell, in dem sich (nicht nur) alternierende Daten direkt im Modell widerspiegeln lassen.

4.2.4 Zusammenfassung

Die Erweiterung des OR-Modells, wie es im Hybrid-Ansatz verwendet wurde, hat eine direktere Abbildung einer XML-DTD auf ein Datenbank-Schema ermöglicht. Durch die Verlagerung der Probleme bei der Abbildung im objektrelationalen Fall auf die Ebene des Datenbank-Modells ist der Gewinn verbunden, den Benutzer vom manuellen Zusammenfügen der entstandenen OR–Relationen entbinden zu können. Allerdings ist noch unklar, welche Anfrageoperationen in dem neuen Modell zur Verfügung stehen. Diese Frage soll im nächsten Abschnitt beantwortet werden. Ein anderer Aspekt, der noch geklärt werden muß, ist, wie die interne Ebene, d.h. die physische Realisierung, aussehen soll. Die Darstellung in Abbildung 4.3 ähnelt zwar einer Darstellung mit Null-Werten, dies muß aber vermieden werden.

4.3 Operationaler Teil

In diesem Teil des Kapitels 4 müßte eigentlich eine erschöpfende Darstellung der Funktionen der zukünfitgen XML-Anfragesprache erfolgen. Da dies sehr umfangreich wäre, die Vorstellung einzelner Konzepte (etwa aus den in Kapitel 2 vorgestellten Sprachen) jeweils auch eine Diskussion über Notwendigkeit und Nutzen mit sich bringt, soll in diesem Abschnitt nur der Anfang gemacht werden.

Dazu soll ein Entwurf für eine Flex-NF^2-Algebra diskutiert werden. Anschließend wird erläutert, wie die Basisoperaionen (siehe Seite 42) mit den Algebra-Operatoren korrespondieren. Der XML-Datentyp des Hybrid-Ansatzes soll auch hier wieder unbeachtet bleiben.

4.3.1 Operationen der Flex-NF^2-Algebra

Die Vorgehensweise zur Festlegung einer Flex-NF^2-Algebra kann in ähnlicher Art und Weise erfolgen, wie die Relationenalgebra zu einer NF^2-Algebra erweitert wurde: durch Einführung eines Nest- und Unnest-Operators bei weitgehender 1:1-Übernahme der anderen Operatoren.

Kalus machte in [Kal95] diesen Vorschlag ebenfalls und gab die beiden Definitionen für eine Nest- und einen Unnest-Operator an, ließ die Festlegung der anderen Operatoren aber offen. Der Nest-Operator soll im folgenden nur angedeutet werden.

Prinzipiell geht es bei der Nestung um die Extraktion von Tupeln mit gleichen Werten in den nicht zu nestenden Attributen. Im NF^2-Modell sind die Tupel auf allen Attributen definiert, nicht so im Flex-NF^2-Modell. Es wird daher eine *variante Projektion* $t\lfloor X \rfloor$ eingeführt, welche die Projektion auf die konkret vorhandenen Attribute des betreffenden Tupels beschreibt, d.h. $t\lfloor X \rfloor = t(X \cap attr(t))$. Damit läßt sich ein Nest-Operator ν^f definieren:

$$\begin{aligned} inst(\nu^f_{A_{neu}(A_1,\ldots A_m}(\mathrm{FNR})) = \{\, t \mid \exists t' \in inst(\mathrm{FNR}) \ : \\ t\lfloor A_{m+1},\ldots,A_k\rfloor = t'\lfloor A_{m+1},\ldots,A_k\rfloor \wedge \\ t(A_{neu}) = \{\, t''\lfloor A_1, ldots, A_m\rfloor \mid t'' \in inst(\mathrm{FNR}) \\ \wedge\, t'\lfloor A_{m+1},\ldots,A_k\rfloor = t''\lfloor A_{m+1},\ldots,A_k\rfloor \,\}\,\} \end{aligned}$$

Durch den Nest-Operator wird jedem Tupel t der Ergebnisrelation das Attribut A_{neu} zugeordnet. Ein Problem ergibt sich, wenn die zu schachtelnden Attribute aus nur einer Variante einer exklusiven Disjunktion stammen, die Definition zeigt jedoch das Prinzip. Eine gründliche Definition des Nest- und des Unnest-Operators kann in [Kal95] nachgeschlagen werden.

Im NF^2-Modell galt für PNF-Relationen die Eigenschaft, daß die Entnestung μ die Inverse der Nestung ν ist. Fordert man auch hier die PNF, d.h. atomare oder flexible (Schlüssel-)Attribute auf jeder Schachtelungsebene[7], gilt die Eigenschaft $\mu^{nf}_A(\nu^{nf}_{A(..)}(\mathrm{FNR})) = \mathrm{FNR}$ auch im Flex-NF^2-Modell.

Nun kann man, aufgrund dieser Eigenschaft und weil es einfacher ist, auf die vorhandenen Operatoren der Flex-Algebra zurückfallen und die neuen Operatoren in der Art $\rho^{nf}(\mathrm{FNR}) = \nu^{nf*}(\rho^f(\mu^{nf*}(\mathrm{FNR})))$ realisieren, wobei μ^{nf*} eine komplette Entnestung und ν^{nf*} eine komplette Nestung vornimmt. Diese Methode beeinträchtigt allerdings die Effizienz der Anfragesprache. Eine Möglichkeit dies zu umgehen ist die Anpassung einer rekursiven Algebra (z.B. von Roth, Korth und Silberschatz [RKS88]) an die hier verwendeten varianten Strukturen. Im folgenden soll daher solch ein Operator (die Vereinigung) stellvertretend definiert werden.

Vereinigung

Die Definition der Vereinigung im rekursiven Stil von [RKS88] setzt sich wie gehabt aus einer Schema- und einer Instanzdefinition zusammen. Die Schemadefinition ist leicht angebbar, da bekannt ist, daß sich das Schema für NF^2-Relationen nicht ändert und für unterschiedliche Schemata in der Flexalgebra als exklusive Disjunktion erklärt ist, somit ist lediglich eine Fallunterscheidung nötig.

Definition 4.4 *Seien FNR_1 und FNR_2 Flex-NF^2-Relationen. Sei X die Menge der atomaren oder flexiblen Attribute und Y die Menge der relationenwertigen Attribute. Dann gilt*

$$sch(FNR_1 \cup^{nf} FNR_2) \ = \ \begin{cases} sch(FNR_1) \ \ wenn\ sch(FNR_1) = sch(FNR_2) \\ \\ <1, 1, \{sch(FNR_1), sch(FNR_2)\}> \quad sonst \end{cases}$$

[7]Gemeint ist die Schachtelung, die nur die (komplexen) relationenwertigen Attribute betrifft, also alle flexiblen genauso wie atomare Attribute behandelt.

$$
\begin{aligned}
&inst(FNR_1 \cup^{nf} FNR_2) \;=\\
&\quad\begin{cases}
\{\, t \mid [\, \exists t_1 \in inst(FNR_1), \exists t_2 \in inst(FNR_2) \;: \\
\qquad\qquad (\, attr(t_1\lfloor X \rfloor) = attr(t_2\lfloor X \rfloor) : t(X) = t_1\lfloor X \rfloor = t_2\lfloor X \rfloor \,) \wedge \\
\qquad\qquad (\, attr(t_1\lfloor Y \rfloor) = attr(t_2\lfloor Y \rfloor) : t(Y) = (t_1\lfloor Y \rfloor \cup^{nf} t_2\lfloor Y \rfloor) \;\;] \\
\quad \vee[\, t \in FNR_1 \wedge (\, \forall t' \in FNR_2 : \\
\qquad\quad (\, X = A_1 \ldots A_m \in sch(FNR_1 \cup^{nf} FNR_2) \;: \\
\qquad\qquad (\, (\, attr(t'\lfloor X \rfloor) \neq attr(t\lfloor X \rfloor) \,) \vee \\
\qquad\qquad\quad (\, attr(t'\lfloor X \rfloor) = attr(t\lfloor X \rfloor) \;\wedge t'\lfloor X \rfloor \neq t\lfloor X \rfloor \,)\,)\,)\,))\,] \\
\quad \vee[\, t \in FNR_2 \wedge (\, \forall t' \in FNR_1 : \\
\qquad\quad (\, X = A_1 \ldots A_m \in sch(FNR_1 \cup^{nf} FNR_2) \;: \\
\qquad\qquad (\, (\, attr(t'\lfloor X \rfloor) \neq attr(t\lfloor X \rfloor) \,) \vee \\
\qquad\qquad\quad (\, attr(t'\lfloor X \rfloor) = attr(t\lfloor X \rfloor) \;\wedge t'\lfloor X \rfloor \neq t\lfloor X \rfloor \,)\,)\,)\,))\,] \\
\quad \textit{für}\, sch(FNR_1) = sch(FNR_2) \\
\\
inst(FNR_1) \cup inst(FNR_2) \qquad sonst
\end{cases}
\end{aligned}
$$

Die Instanzdefinition unterscheidet ebenfalls diese beiden Fälle, wobei nur bei gleichen Schemata eine Vereinigung auf tieferer Ebene fortgesetzt wird.[8] Zu beachten ist bei der Instanzdefinition, daß zunächst sichergestellt werden muß,

FNR_1

A	**X**		
	B	**C**	**D**
1	1	1	
	2		2
2	1	2	

FNR_2

A	**X**		
	B	**C**	**D**
1	1	1	
	1		3
	4	1	
3	3	3	

$FNR_1 \cup^{nf} FNR_2$

A	**X**		
	B	**C**	**D**
1	1	1	
	2		2
	1		3
	4	1	
2	1	2	
3	3	3	

$$sch(\mathrm{FNR}_1) \;=\; sch(\mathrm{FNR}_2) \;=\; sch(\mathrm{FNR}_1 \cup^{nf} \mathrm{FNR}_2) \;=$$

$$< 2, 2, \{\ \mathrm{A}, \mathrm{X}\ (\ \mathrm{B}, < 1, 1, \{\ \mathrm{C}, \mathrm{D}\ \}>\ )\ \}>$$

Abbildung 4.5: Beispiel zur Vereinigung in der Flex-NF^2-Algebra

daß die beiden zu vereinigenden Tupel auch für die gleichen Attribute definiert sind — das sind die Vergleiche der Form $attr(t_i\lfloor X \rfloor) = attr(t_j\lfloor X \rfloor)$.

Ein Beispiel für die Anwendung dieses Operators ist in Abbildung 4.5 enthalten. Es ist zu sehen, daß die Vereinigung für das erste Tupel der Relation

[8] Dies entspricht der Vereinigung, wie sie in [RKS88] definiert wurde.

FNR_1 und das erste Tupel von FNR_2 ($A = 1$) auf der Ebene des Attributs X fortgesetzt wird, die anderen Tupel dagegen einzeln übernommen werden. Bemerkenswert ist auch, daß die Vereinigung $\cup^+$ auf Seite 51 bzw. 38 genau der hier definierten Vereinigung entspricht.

Die anderen Operationen der Flex-NF^2-Algebra können ebenfalls der Flex-Algebra entnommen werden: Neben den Operationen Selektion, Projektion und Verbund die Mengenoperationen, die strukturbezogenen Operationen positive Restriktion und negative Restriktion, sowie die Umbenennungsoperation und die Extension. Für alle sollte eine rekursive Definition gefunden werden.

4.3.2 Die XML-Basisoperationen

Um den Bezug zwischen der FLex-NF^2-Algebra und den am Anfang des Kapitels genannten XML-Basisoperationen herzustellen, folgen hierzu nun einige Ausführungen. Diese wurden absichtlich so allgemein gehalten, da die Algebra nicht vollständig definiert wurde.

Selektion. Die Selektion eines Dokumentes ist gleichzusetzen mit der Selektion der Flex-NF^2-Algebra — durch eine Selektionsbedingung läßt sich eine inhaltliche Auswahl vornehmen. Möchte man aber anhand der Struktur ein Auswahl treffen, ist die Angabe eines Pfades nötig. Der hinter dieser Angabe verborgene strukturelle Existenzquantor kann mittels der (positiven) Restriktion realisiert werden. Die Restriktion könnte auch explizit angegeben werden.

Extraktion. Die Extraktion von Teilinformationen (z.B. Subelementen) kann durch eine Projektion auf diesen Teil des jeweiligen Datenbankschemas erfolgen.

Reduktion. Das Entfernen ausgewählter Subelemente kommt dem Update-Mechanismus einer DB-Spache gleich. Insert, Update und Delete wurden für die Flex-NF^2-Algebra aber noch nicht festgelegt. Betrifft das Entfernen die gesamte Dokumentkollektion sowie eine Veränderung der DTD, kommt dies einer Schemaevolution mit einer Datenmigration gleich, übertrifft aber sicherlich die Absicht der Reduktion.

Restrukturierung. Die Erzeugung einer neuen Elementmenge ist mit der Ergebnisrückgabe auf eine Anfrage vergleichbar. Das Ergebnis muß nur in XML-Form gebracht werden. Wenn es sich um die Umstrukturierung innerhalb eines Dokumentes handelt, gilt die Aussage des letzten Punktes: Eine Update-Möglichkeit läßt die Algebra noch offen.

Kombination. Die Kombination von Elementen ist im Flex-NF^2-Modell z.B. mit den Mengenoperationen realisierbar. Es gehören sicherlich auch arithmetische Funktionen und Aggregatfunktionen dazu, die aber nicht schon in der Algebra eingeführt werden sollten.[9] Das Einfügen eines neuen Ele-

[9]Man möchte die Algebra zunächst klein halten, um Kriterien wie Sicherheit und Abgeschlossenheit leicher beweisen zu können. Die Einführung der genannten Funktionen würde aber zu einer komplexeren, mehrsortigen Algebra führen — dies möchte man verhindern.

mentes betrifft die Insert-Funktion der Update-Sprache.

4.3.3 Zusammenfassung

Der operationale Teil dieses Kapitels war zunächt auf das Finden einer Algebra für das Flex-NF^2-Modell ausgerichtet. Die Lösung ergab sich durch die Kombination der Flex-Algebra [Kal95] mit einer NF^2-Algebra [RKS88].

Es ist jedoch nur eine Teillösung, da die Funktionalität des XML-Datentyps vorerst nicht betrachtet wurde. Neben der Festlegung dieser Funktionen ist hierbei die Frage zu klären, ob die Operatoren der Algebra auch auf diesem Datentyp realisierbar sind. Eine Baum-Algebra (z.B. von [Chi00]) könnte ein Ausgangspunkt sein. Ideal wäre in diesem Zusammenhang folgender Vorschlag: Auf Anfrageebene wird keine Unterscheidung gemacht — nur der Optimierer wählt die Algorithmen für die Relationen bzw. die Funtionen des XML-Datentyps aus. Eine einfachere Möglichkeit wäre die Verlagerung auf die Benutzerebene — der Benutzer wählt selbst die Funktionen, um auf die Daten des XML-Datentyps zuzugreifen. Dies setzt aber unbedingt Strukturkenntnisse beim Benutzer voraus und ist daher unhandlich, besitzt aber den Vorteil einer einfachen Integration dieses Datentyps in ein herkömmliches DBMS (z.B. mit UDF's).

Den letzten Vorschlag könnte man auch auf den Information–Retrieval–Anteil übertragen. Explizit durch den Benutzer formuliert, lassen sich die IR-Fähigkeiten sicherlich leichter integrieren.

4.4 Zusammenfassung

Das Ziel dieses Kapitels war nicht der Entwurf einer kompletten XML-Anfragealgebra oder etwa XML-Anfragesprache. Dafür wurden Möglichkeiten aufgezeigt, wie man dieses Ziel erreichen kann. Im Gegensatz zu den meisten existierenden Sprachen wurde nicht von einer Syntaxdefinition ausgegangen, vielmehr brachte die genauere Betrachtung der Speichermodells zu Beginn des Kapitels eine Verbesserung dieser Speicherart. Die Abbildung eines XML-Dokuments mit alternativen Elementen ist nun direkter möglich, d.h. daß es im Datenbank-Modell ein Konstrukt gibt, das der Formulierung einer Alternative (einem bisherigen Problemfall) eins zu eins entspricht.

Durch den in diesem Kapitel eingeschlagenen Weg wurde auch ein neuer Anfangspunkt für eine XML-Algebra geschaffen, die Flex-NF^2-Algebra, mit der die geforderte Basisfunktionalität für XML realisiert werden konnte. Die Flex-NF^2-Algebra ist als Konsequenz aus der Einführung eines veränderten Datenmodells (Flex-Modell) in Kombination mit der NF^2-Algebra des ursprüglichen Speichermodells entstanden. Prinzipiell hat man durch den Nest- und Unnest-Operator zusammen mit den schon entworfenen Flex-Operatoren bereits eine Flex-NF^2-Algebra, deren praktische Verwendung jedoch ineffizient ist. Abhilfe schafft eine rekursive Definition der Operatoren — am Beispiel der Vereinigung wurde solch ein Operator auf diese Weise definiert.

Die offenen Aspekte einer zukünftigen XML-Anfragesprache betreffen die Überprüfung der genannten Anforderungen an eine Anfragesprache für die

Flex-NF^2-Algebra, die Integration von Information–Retrieval–Funktionen, eine gründliche Untersuchung der internen Ebene für das neue Datenmodell und nicht zuletzt eine vervollständigte Flex-NF^2-Algebra, definiert in rekursiver Form.

Kapitel 5

Existierende XML-Datenbanken

Die Betrachtungen der bisherigen Kapitel stützten sich hauptsächlich auf den Forschungssektor im Bereich XML. In diesem Kapitel soll ein Blick auf das kommerzielle Umfeld geworfen werden — einerseits läßt dies Vergleiche zu, andererseits erhält man einen Überblick und kann abschätzen, wieviel Arbeit noch in die Realisierung solcher Systeme investiert werden muß.

5.1 Tamino

Eine kommerzielle Datenbank zum Verarbeiten und Speichern von XML ist Tamino von der Software AG [Mül00, Opp99]. Neben reinen XML-Daten können von diesem XML-Server auch Text-, Bild- und Audiodaten verwaltet werden. Tamino ist auf Windows NT und Sun Solaris realisiert worden.

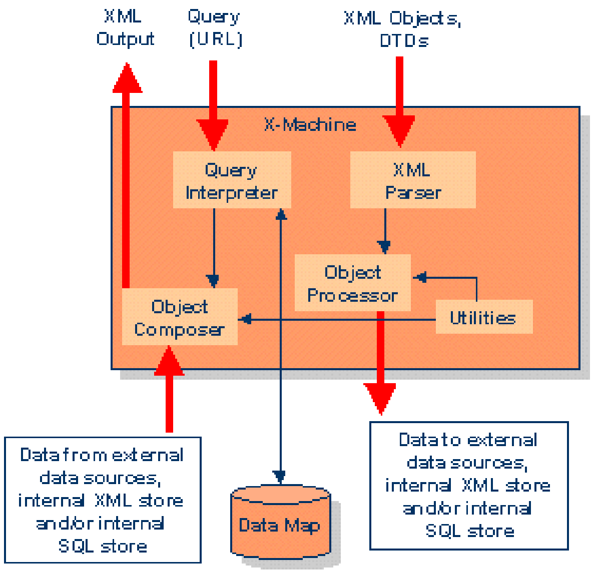


Abbildung 5.1: Tamino: Architektur, Quelle: [Mül00]

Architektur

Zentrale Verarbeitungseinheit ist die "X-Machine", die XML-Datenstrukturen ohne Änderung der Hierachie oder Sequenz speichert und verwaltet (Abbildung 5.1).

Im Kern verwendet Tamino Kompressions- und Pufferverfahren, um die Performance zu gewährleisten. Die XML-Objekte werden in ihrem natürlichen Format im *XML-Datenspeicher* gespeichert. Daneben gibt es einen *SQL-Datenspeicher* zur Unterstützung strukturierter Daten. Relationale Daten können so leicht in Tamino abgebildet werden und stehen über eine *SQL-Engine* direkt zur Verfügung.

Die *Data Map* enthält Systemdaten wie DTD's und relationale Schemata. Sie beschreibt, wie XML-Elemente auf den XML-Speicher, den SQL-Speicher oder über den X-Node auf externe Datenquellen abgebildet werden. Der *X-Node* ermöglicht den Zugriff auf mehrere heterogene Datenquellen (z.B. DB2, Adabas,...).

Speicherung und Anfragen

Ein XML-Dokument wird durch den *Object Processor* entspechend den im (zu definierenden) Schema angegebenen Vorgaben auf den XML- und SQL-Speicher verteilt. Das Schema wird zuvor vom Administrator festgelegt, der darüber entscheidet, welche Teile des XML-Dokuments "native" bzw. relational gespeichert werden. Mittels den in der Data-Map enthaltenen Informationen kann ein Dokument dann aus dem XML-Speicher, dem SQL-Speicher und externen Datenquellen zusammengesetzt werden.

Anfragen an Tamino werden im URL-Stil verfaßt. Dazu unterstützt Tamino eine Teilmenge der XQL-Anfragesprache, in einer neueren Version auch XPath. Über die SQL-Engine können Anfragen an die strukturierten Daten im SQL-Speicher auch via SQL gestellt werden. Eine Volltext-Retrieval-Möglichkeit über den gesamten Datenbestand ist ebenfalls enthalten.

Tamino's Funktionalität kann durch sogenannte *Server Extensions* erweitert werden. Diese sind in etwa vergleichbar mit Triggern und Stored Procedures.

5.2 eXcelon

Der eXcelon XML-Server [ODI00] ist ebenfalls ein System zur Speicherung und Anfrage von XML-Daten. Es stammt von der Firma eXcelon Corp. (Object Design Inc.) und läuft unter Windows NT und Solaris.

Neben XML lassen sich auch hier externe Datenquellen wie relationale Datenbanken und Multimedia-Objekte integrieren und unter einer einheitlichen Sicht präsentieren. Die XML-Daten können indexiert, mittels DOM manipuliert und mit einem XQL-Dialekt angefragt werden.

Architektur

eXcelon besteht im wesentlichen aus den folgenden Komponenten:

- XML Data Engine (XMLStore)
- Server
- Java Extensions
- Explorer
- Back-end Connectivity

Die *XML Data Engine* ist für die Abspeicherung beliebiger XML-Daten verantwortlich und basiert auf ObjectStore, einem früheren OODBMS der Firma Object Design. Die XML-Daten werden in einem internen XML-Datentyp verwaltet. Sie werden jedoch nicht zerlegt wie dies bei den relationalen Speicherungsformen (Kapitel 3) der Fall ist, so daß aufwendige Verbunde vermieden werden können.

Der eXcelon *Server* realisiert Caching-Mechanismen, in die die externen Datenquellen einbezogen werden und bildet die Verbindung zu den Anwendungs-Clients. Solch ein Client ist unter anderem der eXcelon *Explorer*, der eine graphische Schnittstelle zum XML-Server bietet und die Anfrageformulierung unterstützt.

Über die *Java-Extensions* läßt sich die Funktionalität des XML-Servers erweitern und über das *Back-end Connectivity* wird die Einbindung fremder Daten geregelt, die in der Regel in die Datenbank aufgenommen werden.

Anfragefunktionalität

Die Anfragefähigkeiten in eXcelon sind eine Mischung aus XQL, XPath, XSLT und eigenen Erweiterungen [eXc21b2]. Der Beschränkung von XQL auf Joins nur innerhalb eines Dokuments wurde mit der Einführung eines eigenen Link-Mechanismus entgegengenwirkt. Anfragen können an eine Menge von Dokumenten gestellt werden, die zuvor explizit in einem Binder-Dokument festgelegt wurde.[1]

Die Erweiterungen beinhalten z.B. Funktionen wie `contains-words()` oder `document-name()`, die eine Art Volltext-Erweiterung darstellen.

Der eXcelon Explorer unterstützt den Benutzer bei der Anfrageformulierung graphisch. Durch die Verwendung von XSL-Stylesheets können die Dokumente oder Teile der Dokumente sichtbar gemacht werden.

5.3 Einschätzung

Die beiden vorgestellten Systeme bieten jeweils eine Komplettlösung für den Umgang mit XML an. Sie erheben beide den Anspruch, eine universelle Datenintegration (für strukturierte und semistrukturierte Daten) zu ermöglichen.

[1] Ein Binder-Dokument bestehend aus anderen Binder-Dokumenten war leider (in Version 2.1 beta2) nicht möglich, was eine flexible Anfrageformulierung erheblich behinderte.

Die eingesetzten Methoden sind, soweit sie erkennbar waren, jedoch mit einigem Abstand zu bewerten. Die *Data Map* von Tamino verlangt z.B. Unterstützung bei der Speicherung der XML-Daten. Mittels einer Mapping-Sprache wird festgelegt, welche Anteile der XML-Daten strukturiert (im SQL-Speicher) und welche Anteile in den XML-Speicher übernommen werden, der wahrscheinlich als eine Art BLOB realisiert wurde.

Bei eXcelon wird das XML-Dokument auf Objekte der zugrundegelegten objektorientierten Datenbank abgebildet. Im einfachsten Fall kann man die Annahme treffen, daß für jedes XML-Element ein Objekt erzeugt wird. Dies würde auch die schlechte Performance erklären, die sich beim Test dieses Systems, besonders beim Einfügen von XML-Dokumenten, bemerkbar machte. Ein Test ergab, daß sich die Speicherauslastung des XML-Servers gegenüber einer Speicherung im Dateisystem verzehnfachte.

Die Anfragemöglichkeiten der Systeme bestehen leider nur aus einem XQL-Dialekt, der teilweise mit Funktionen angereichert wurde. Diese Funktionen bieten beispielsweise nur sehr eingeschränkte Retrieval-Fähigkeiten, die eher an das `grep`-Kommando unter UNIX erinnern.

Zusammenfassend läßt sich sagen, daß die beiden Systeme, Tamino und eXcelon, in ihrem jetzigen Stadium keine herausragenden Eigenschaften bezüglich der Verwaltung von XML besitzen, sie jedoch den Vorteil hatten, die ersten kommerziellen Produkte in diesem Bereich zu sein. Inzwischen haben auch die Datenbankhersteller ihre Produkte um XML-Fähigkeiten erweitert. Dazu gehören zum Beispiel DB2 (mit dem XML-Extender), Informix und Oracle8i.

Kapitel 6

Schlußbetrachtungen

XML hat seit seiner Einführung 1998 zunehmend an Akzeptanz gewonnen. Dies ist sicherlich auf die Tatsache zurückzuführen, daß XML eine einfache Trennung der inhaltlichen Struktur vom äußeren Erscheinungsbild eines Dokumentes erlaubt und das Anwendungsfeld sehr weit ist.

Datenbanksysteme bieten eine effiziente Speicherung, Wahrung der Integrität der Daten und haben viele weitere nützliche Eigenschaften, die man sich bei der Speicherung von XML-Daten zunutze machen möchte.

Die vorliegende Arbeit untersuchte daher die Beziehung zwischen XML und der Datenbank-Technik. Dazu wurden existierende XML-Anfragesprachen und bereits entwickelte Speicherungformen für XML untersucht, wobei der Schwerpunkt bei letzterem bei der relationalen Technik lag. Es war leider festzustellen, daß es bisher keine Verbindungen zwischen einer XML-Anfragesprache und den Techniken zur Speicherung in relationalen Datenbanken gibt. Viele Spachen wurden aufgrund einer Spachdefinition, die die physische Speicherung vernachlässigte, entwickelt. Als Konsequenz wurde in dieser Arbeit der Weg ausgehend von einer Speicherungsform hin zu einer Sprache für die Anfrage der gespeicherten XML-Daten gewählt. Dabei wurden neben Anforderungen, die eine XML-Anfragesprache erfüllen sollte, erste Vorschläge unterbreitet. Es wurde ein Ansatz für eine theoretische Grundlage erarbeitet, der auf einer objektrelationalen Speicherungsform aufbaut und diese erweitert.

Einige Aspekte mußten jedoch noch unbeachtet bleiben. Dazu gehört z.B. die Integration von Infomation–Retrieval–Fähigkeiten. Die interne Ebene des neuen Datenmodells konnte ebenfalls nicht untersucht werden, wobei von der Durchführbarkeit einer Realisierung mit einem kommerziellen DBMS, wie Informix Universal Server, jedoch ausgegangen werden kann. Dies und die Tatsache, daß man mit der entworfenen Algebra erst einen Grundstock an Funktionalität besitzt, sind die Gründe, warum auf einen Prototypen in dieser Arbeit verzichtet wurde.

Weitere theoretische Untersuchungen sind im Zusammenhang mit der Algebra notwendig. Dazu zählt auch die Untersuchung des Optimierungspotentials. Es wäre zu prüfen, wie die Optimierungstechniken aus dem relationalen Bereich übernommen werden könnten.

Anhang A

Beispiele

A.1 Relationenschemata zum Beispiel aus Abschnitt 3.2.2

book (bookID: integer, book.booktitle: string, book.author.name.firstname: string, book.author.name.lastname: string, book.author.address: string, author.authorid: string)
booktitle (booktitleID: integer, booktitle: string)
article (articleID: integer, article.contactauthor.authorid: string, article.title: string)
article.author (article.authorID: integer, article.author.parentID: integer, article.author.name.firstname: string, article.author.name.lastname: string, article.author.address: string, article.author.authorid: string)
contactauthor (contactauthorID: integer, contactauthor.authorid: string)
title (titleID: integer, title: string)
monograph (monographID: integer, monograph.parentID: integer, monograph.title: string, monograph.editor.name: string, monograph.author.name.firstname: string, monograph.author.name.lastname: string, monograph.author.address: string, monograph.author.authorid: string)
editor (editorID: integer, editor.parentID: integer, editor.name: string)
editor.monograph (editor.monographID: integer, editor.monograph.parentID: integer, editor.monograph.title: string, editor.monograph.author.name.firstname: string, editor.monograph.author.name.lastname: string, editor.monograph.author.address: string, editor.monograph.author.authorid: string)
author (authorID: integer, author.name.firstname: string, author.name.lastname: string, author.address: string, author.authorid: string)
name (nameID: integer, name.firstname: string, name.lastname: string)
firstname (firstnameID: integer, firstname: string)
lastname (lastnameID: integer, lastname: string)
address (addressID: integer, address: string)

Abbildung A.1: Datenbankschema beim Basic Inlining

book (bookID: integer, book.booktitle.isroot: boolean, book.booktitle: string)
article (articleID: iteger, article.contactauthor.isroot: boolean,
article.contactauthor.authorid: string)
monograph (monographID: integer, monograph.parentID: integer,
monograph.parentCODE: integer, monograph.editor.isroot: boolean,
monograph.editor.name: string)
title (titleID: integer, title.parentID: integer, title.parentCODE: integer, title: string)
author (authorID: integer, author.parentID: integer, author.parentCODE: integer,
author.name.isroot: boolean, author.name.firstname.isroot: boolean,
author.name.firstname: string, author.name.lastname.isroot: boolean,
author.name.lastname: string, author.address.isroot: boolean, author.address: string,
author.authorid: string)

Abbildung A.2: Datenbankschema beim Shared Inlining

book (bookID: integer, book.booktitle.isroot: boolean, book.booktitle: string,
author.name.firstname: string, author.name.lastname: string, author.address: string,
author.authorid: string)
article (articleID: iteger, article.contactauthor.isroot: boolean, article.contactauthor.authorid:
string, article.title.isroot: boolean, article.title: string)
monograph (monographID: integer, monograph.parentID: integer, monograph.parentCODE:
integer, monograph.title: string, monograph.editor.isroot: boolean,
monograph.editor.name: string, author.name.firstname: string,
author.name.lastname: string, author.address: string, author.authorid: string)
author (authorID: integer, author.parentID: integer, author.parentCODE: integer,
author.name.isroot: boolean, author.name.firstname.isroot: boolean,
author.name.firstname: string, author.name.lastname.isroot: boolean,
author.name.lastname: string, author.address.isroot: boolean, author.address: string,
author.authorid: string)

Abbildung A.3: Datenbankschema beim Hybrid Inlining

A.2 Objektrelationale Speicherungsform (Abschnitt 3.3.1)

```
<!ELEMENT publications ( book | article | conference)*>
<!ELEMENT book (front, body, references)>
<!ELEMENT front (title, author+, edition, publisher)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (first, second, e-mail?)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT second (#PCDATA)>
<!ELEMENT e-mail (#PCDATA)>
<!ELEMENT edition (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT body (part+ | chapter+)>
<!ELEMENT part (ptitle, chapter+)>
<!ATTLIST part id ID #REQUIRED>
<!ELEMENT ptitle (#PCDATA)>
<!ELEMENT chapter (ctitle, section+)>
<!ATTLIST chapter id ID #REQUIRED>
<!ELEMENT ctitle (#PCDATA)>
<!ELEMENT section (stitle, paragraph+)>
<!ATTLIST section id ID #REQUIRED>
<!ELEMENT stitle (#PCDATA)>
<!ELEMENT paragraph (#PCDATA)>
<!ELEMENT references (publications*)>
<!ATTLIST references reftype (book | article | conference |
   wwwaddress) "article">
<!ELEMENT article (meta, body, references)>
<!ELEMENT meta (author+, title, conference)>
<!ELEMENT conference (editor+, conftitle, city, year)>
<!ELEMENT editor (first, second, e-mail?)>
<!ELEMENT conftitle (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT year (#PCDATA)>
```

Abbildung A.4: Beispiel-DTD “Buch” zum objektrelationalen Ansatz

Anhang B

Zur Flex-NF^2-Algebra

B.1 Definition der Vereinigung

Die Vereinigung wurde in Kapitel 4 als Beispiel für einen neuen Operator der Flex-NF^2-Algebra vorgestellt. Zur Verdeutlichung der Definition sind hier nun ebenfalls die Definitionen der Vereinigung für das Relationenmodell (RM) [HS95], die Flex-Algebra (Flex) [Kal95] und das NF^2-Modell (NF^2) [RKS88] angegeben, durch die das Verständnis für die neue Definition verbessert werden soll. Die Darstellung wurde an die $sch(..)$– und $inst(..)$–Darstellung angepaßt.

Ein etwas umfangreicheres Beispiel als im Kapitel 4 schließt sich unmittelbar an.

RM:

$$\begin{aligned} inst(R_1 \cup R_2) &= \{\, t \mid t \in inst(R_1) \vee t \in inst(R_2) \,\} \\ sch(R_1 \cup R_2) &= sch(R_1) \quad (= sch(R_2)\,) \end{aligned}$$

Flex:

$$\begin{aligned} inst(FR_1 \cup^f FR_2) &= inst(FR_1) \cup inst(FR_2) \\ sch(FR_1 \cup^f FR_2) &= \,< 1, 1, \{sch(FR_1), sch(FR_2)\} > \end{aligned}$$

NF^2:

$$\begin{aligned} inst(\mathrm{NR}_1 \cup^n \mathrm{NR}_2) = \{\, t \mid &[\, \exists t_1 \in inst(\mathrm{NR}_1), \exists t_2 \in inst(\mathrm{NR}_2) : \\ &\quad (\,\forall X, Y \in sch(\mathrm{NR}_1) : t(X) = t_1(X) = t_2(X) \\ &\quad\ \wedge\ t(Y) = (\, t_1(Y) \cup^n t_2(Y))\,] \\ &\vee [\, t \in inst(\mathrm{NR}_1) \wedge (\forall t' \in inst(\mathrm{NR}_2) : \\ &\quad (\forall X \in sch(\mathrm{NR}_1) : t(X) \neq t'(X)))\,] \\ &\vee [\, t \in inst(\mathrm{NR}_2) \wedge (\forall t' \in inst(\mathrm{NR}_1) : \\ &\quad (\forall X \in sch(\mathrm{NR}_1) : t(X) \neq t'(X)))\,] \\ sch(\mathrm{NR}_1 \cup^n \mathrm{NR}_2) = \ & sch(\mathrm{NR}_1) \quad (= sch(\mathrm{NR}_2)\,) \end{aligned}$$

<u>Flex–NF²:</u>

$$sch(FNR_1 \cup^{nf} FNR_2) \;=\; \begin{cases} sch(FNR_1) \;\text{ wenn } sch(FNR_1) = sch(FNR_2) \\[2ex] <1, 1, \{sch(FNR_1), sch(FNR_2)\}> \quad \text{sonst} \end{cases}$$

$$inst(FNR_1 \cup^{nf} FNR_2) \;=$$

$$\begin{cases} \{\, t \mid [\, \exists t_1 \in inst(\mathrm{FNR}_1), \exists t_2 \in inst(\mathrm{FNR}_2) \;: \\ \qquad\qquad (\, attr(t_1\lfloor X \rfloor) = attr(t_2\lfloor X \rfloor) : t(X) = t_1\lfloor X \rfloor = t_2\lfloor X \rfloor \,) \wedge \\ \qquad\qquad (\, attr(t_1\lfloor Y \rfloor) = attr(t_2\lfloor Y \rfloor) : t(Y) = (t_1\lfloor Y \rfloor \cup^{nf} t_2\lfloor Y \rfloor) \;\;] \\ \quad \vee [\; t \in \mathrm{FNR}_1 \wedge (\, \forall t' \in \mathrm{FNR}_2 : \\ \qquad\quad (\, X = A_1 \ldots A_m \in sch(FNR_1 \cup^{nf} FNR_2) \;: \\ \qquad\qquad (\, (\, attr(t'\lfloor X \rfloor) \neq attr(t\lfloor X \rfloor) \,) \vee \\ \qquad\qquad\quad (\, attr(t'\lfloor X \rfloor) = attr(t\lfloor X \rfloor) \;\wedge t'\lfloor X \rfloor \neq t\lfloor X \rfloor \,)\,)\,))\,] \\ \quad \vee [\; t \in \mathrm{FNR}_2 \wedge (\, \forall t' \in \mathrm{FNR}_1 : \\ \qquad\quad (\, X = A_1 \ldots A_m \in sch(FNR_1 \cup^{nf} FNR_2) \;: \\ \qquad\qquad (\, (\, attr(t'\lfloor X \rfloor) \neq attr(t\lfloor X \rfloor) \,) \vee \\ \qquad\qquad\quad (\, attr(t'\lfloor X \rfloor) = attr(t\lfloor X \rfloor) \;\wedge t'\lfloor X \rfloor \neq t\lfloor X \rfloor \,)\,)\,))\,] \\ \quad \text{für } sch(FNR_1) = sch(FNR_2) \\[2ex] inst(\mathrm{FNR}_1) \cup inst(\mathrm{FNR}_2) \quad \text{sonst} \end{cases}$$

Das folgende Beispiel geht aus dem in Abbildung 4.5 auf Seite 54 hervor. Es wurde um ein alternatives mengenwertiges Attribut Y und um die beiden alternierenden Attribute E und F erweitert.

$sch(\mathrm{FNR}_3) \;=\; sch(\mathrm{FNR}_4) \;=\; sch(\mathrm{FNR}_3 \cup^{nf} \mathrm{FNR}_4) \;=$

$<3, 3, \{\, \mathrm{A}, <1, 1, \{\, \mathrm{X}\,(\, \mathrm{B}, <1, 1, \{\, \mathrm{C}, \mathrm{D} \,\}>), \mathrm{Y}\,(\, \mathrm{G} \,)\,\}>, <1, 1, \{\, \mathrm{E}, \mathrm{F} \,\}> \}>$

FNR_3

A	**X**			**Y**	**E**	**F**
	B	**C**	**D**	**G**		
1	1	1			1	
	2		2			
2	1	2				2
5				5	1	
5				5		2

FNR_4

A	**X**			**Y**	**E**	**F**
	B	**C**	**D**	**G**		
1	1	1			1	
	1		3			
	4	1				
5				6		2
				7		

$\mathrm{FNR}_3 \cup^{nf} \mathrm{FNR}_4$

<table>
<tr><th rowspan="2">A</th><th colspan="3">X</th><th>Y</th><th rowspan="2">E</th><th rowspan="2">F</th></tr>
<tr><th>B</th><th>C</th><th>D</th><th>G</th></tr>
<tr><td rowspan="4">1</td><td>1</td><td>1</td><td></td><td rowspan="4"></td><td rowspan="4">1</td><td rowspan="4"></td></tr>
<tr><td>2</td><td></td><td>2</td></tr>
<tr><td>1</td><td></td><td>3</td></tr>
<tr><td>4</td><td>1</td><td></td></tr>
<tr><td>2</td><td>1</td><td>2</td><td></td><td></td><td></td><td>2</td></tr>
<tr><td>5</td><td colspan="3"></td><td>5</td><td>1</td><td></td></tr>
<tr><td rowspan="3">5</td><td rowspan="3" colspan="3"></td><td>5</td><td rowspan="3"></td><td rowspan="3">2</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td></tr>
</table>

Literaturverzeichnis

[Abi97] Serge Abiteboul *Querying Semi-Structured Data.* International Conference on Database Theory, 1997

[ABS00] S. Abiteboul, P. Buneman, D. Suciu *Data on the web.* Morgan Kaufmann Publishers, San Francisco, California, 2000, ISBN 1-55860-622-X

[AQM+96] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, Janet L. Wiener *The Lorel Query Language for Semistructured Data.* Journal of Digital Libraries, Vol. 1, No. 1, 1996

[BLUurl] Projekt-Homepage am Lehrstuhl Datenbanken- und Informationsysteme unter *http://wwwdb.informatik.uni-rostock.de/blueview/*

[BM98] Henning Behme, Stefan Mintert *XML in der Praxis – Professionelles Web-Publishing mit der Extensible Markup Language.* Addison-Wesley, Bonn, 1998

[Chi00] Boris Chidlovskii *Using Regular Tree Automata as XML schemas.* Proceedings of the IEEE Forum on Research and Technology Advances in Digital Libraries (ADL), 2000

[Col89] Latha S. Colby *A Recursive Algebra and Query Optimization for Nested Relations.* Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD Record 18(2), June 1989, Seiten 273-283

[DFF+98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, Dan Suciu *XML-QL: A Query Language for XML.* Submission to the World Wide Web Consortium 19-August-1998, http://www.w3.org/TR/1998/NOTE-xml-ql-19980819.html

[eXc21b2] *Bookshelf for eXcelon Release 2.1 Beta 2.* Dokumentation zur Beta-Version von eXcelon für Solaris, Frühjahr 2000

[FK99] Daniela Florescu, Donald Kossmann *Storing and Querying XML Data using an RDBMS.* In: IEEE Bulletin of the Technical Committee on Data Engineering, Vol. 22 No. 3, September 1999, Seiten 27-34

[Fuh97] N. Fuhr *Information Retrieval — Skriptum zur Vorlesung.* 1997, http://www-ls6.informatik.uni-dortmund.de/ir/

[GETurl] Projekt-Homepage am Lehrstuhl Datenbanken- und Informationsysteme unter *http://wwwdb.informatik.uni-rostock.de/getess/*

[GMW99] Roy Goldman, Jason McHugh, Jennifer Widom *From Semistructured Data to XML: Migrating the Lore Data Model and Query Language.* ACM SIGMOD Workshop on The Web and Databases (WebDB'99), June 3-4, 1999, Philadelphia, Pennsylvania, USA

[GP99] Charles F. Goldfarb, Paul Prescod *XML Handbuch.* Prentice Hall, München, 1999, ISBN 3-8272-9575-0

[HMP+00] A. Heuer, H. Meyer, B. Porst, P. Titzler *BlueView: Virtual Document Servers for Digital Libraries.* Proceedings of the IEEE Forum on Research and Technology Advances in Digital Libraries (ADL), 2000

[HP99] Andreas Heuer, Denny Priebe *IRQL - Yet Another Language for Querying Semistructured Data?* Preprint CS-01-99, Lehrstuhl für Datenbank- und Informationssysteme, FAchbereich Informatik, Universität Rostock, 1999

[HS95] Andreas Heuer, Gunter Saake *Datenbanken — Konzepte und Sprachen.* International Thomson Publishing, Bonn, 1995, ISBN 3-929821-31-1

[Kal95] Christian Kalus *Unterstützung varianter Strukturen in einem erweiterten relationalen Datenmodell.* Dissertation, Fakultät für Informatik, Universität Ulm, 1995

[KM99] Meike Klettke, Holger Meyer *Managing XML documents in object-relational databases.* Rostocker Informatik-Fachberichte 24, Universität Rostock, 1999, auch erschienen als: Meike Klettke, Holger Meyer *XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics.* Proceedings of the Third International Workshop on the Web and Databases (WebDB), 2000

[Mai98] David Maier *Database Desiderata for an XML Query Language.* In Proceedings of the Query Languages workshop, Cambridge, Mass., 1998, http://www.w3.org/TandS/Ql/QL98/pp/maier.html

[MKH00] Holger Meyer, Meike Klettke, Andreas Heuer *Datenbanken im WWW — Von CGI bis JDBC und XML.* In: HMD — Praxis der Wirtschaftsinformatik, Nr. 214, ISBN 3-932588-79-7, dpunkt Verlag, Heidelberg, Seiten 5–22

[Mül00] Gabriele Müller-Steinhagen *Native XML-Datenbank.* In: UNIX^{open} 9/2000, AWi UNIX^{open} Verlagsgesellschaft mbH, Trostbach, Deutschland, ISSN 0943-8416, Seiten 34–37

[ODI00] *An XML Data Server For Building Enterprise Web Applications.* Whitepaper, Object Design Inc., (2000), http://www.exceloncorp.com/products/white_papers_index.htm

[Opp99] Karin Oppel *Tamino — Der Information Server für Electronic Business.* Whitepaper, Software AG, Oktober 1999, http://www.softwareag.com/tamino/technical/whitepaper.htm

[Por99] Beate Porst *Untersuchung zu Datentyperweiterungen für XML-Dokumente und ihre Anfragemethoden am Beispiel von DB2 und Informix.* Diplomarbeit, Fachbereich Informatik, Universität Rostock, April 1999

[RKS88] Mark A. Roth, Henry F. Korth, Abraham Silberschatz *Extended Algebra and calculus for Nested Relational Databases.* ACM Transactions on Database Systems, Vol. 13, No. 4, 1988, Seiten 389–417

[RLS00] Jonathan Robie, Joe Lapp, David Schach *XQL Proposal.* http://www.w3.org/TandS/QL/QL98/pp/xql.html bzw. http://metalab.unc.edu/xql/

[Rob99a] Jonathan Robie *XQL Design.* http://www.texcel.no/whitepapers/xql-design.html bzw. http://metalab.unc.edu/xql/

[Rob99b] Jonathan Robie *XQL FAQ.* http://metalab.unc.edu/xql/

[STH+99] Jayaval Shanmugasundaram, Kristin Tufte, Gang He, Chun Zhang, David DeWitt, Jeffrey Naughton *Relational Databases for Querying XML Documents: Limitations and Opportunities.* In: Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999, Seiten 302-314

[Tim99] Jens Timm *Speicherung von XML-Dokumenten in objekt-relationalen Datenbanken.* Diplomarbeit, Fachbereich Informatik, Universität Rostock, Dezember 1999

[VAB96] Marc Volz, Karl Aberer, Klemens Böhm *An OODBMS-IRS Coupling for Structured Documents.* IEEE Data Engineering Bulletin, Special Issue on Integrating Text Retrieval and Databases, Volume 19, 1996, Seiten 34-42

[xml98] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen (Herausgeber) *Extensible Markup Language (XML) 1.0.* W3C Homepage: http://www.w3.org/TR/1998/REC-xml-19980210

Erklärung

Ich erkläre, daß ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, 28.10.2000 Mathias Bietz

Thesen

1. XML ermöglicht die strikte Trennung der inhaltlichen Struktur vom äußeren Erscheinungsbild eines Dokumentes. Anwendungsprogramme werden daher zunehmend von der Verwendung von XML geprägt.

2. Datenbanksysteme bilden eine Grundlage für die effiziente Speicherung von XML. Ein Vorteil eines jeden DBMS ist seine deklarative Anfragesprache. Diese Anfragesprachen verlieren aber bei den bisher entwickelten Speicherungsmodellen für XML ihre Deklarativität.

3. Speicherungsformen, die XML in einem relationalen oder objektrelationalen Datenbankmanagmentsystem speichern, erlauben keine direkte Abbildung von XML auf die Datenbankstrukturen.

4. Spezielle XML-Anfragesprachen wurden entwickelt, schenken einer Realisierung in einem DBMS aber kaum Beachtung. Sie sind schlechte Kandidaten für eine XML-Anfragesprache mit (objekt-)relationaler Speicherungsgrundlage.

5. Weder die betrachteten Speicherungsformen noch die untersuchten XML-Anfragesprachen können effizient miteinander kombiniert werden.

6. Die objektrelationale Speicherungsform besitzt aufgrund ihrer Erweiterbarkeit die besten Voraussetzungen für eine Kombination einer XML-Anfragesprache mit dieser Speicherungsform.

7. Die Zusammenführung von Information–Retrieval–Techniken mit den strukturierten Anfragemöglichkeiten bisheriger DBMS trägt dem semistrukturierten Charakter der XML-Daten Rechnung.

8. Eine Erweiterung des OR-Modells und der inhärenten NF^2-Algebra bildet eine adäquate Abbildung von XML auf Datenbankstrukturen und eine angemessene Anfragemöglichkeit für die gespeicherten XML-Daten.