Speicherung von XML-Dateien in Objekt-relationalen Datenbanken

Diplomarbeit

Universität Rostock, Fachbereich Informatik

vorgelegt von:	Timm, Jens
Matrikelnummer:	092209012
Betreuer:	Prof. Dr. Andreas Heuer Dr. Meike Klettke
Bearbeitungszeitraum:	01.07.1999 - 31.12.1999

Zusammenfassung

Semistrukturierte Daten haben in vielen Anwendungsgebieten eine zunehmende Bedeutung. XML ist eine typische Repräsentationsform semistrukturierter Daten. XML ist eine vom W3C standardisierte Sprache für den Austausch von Dokumenten. Es gibt zahlreiche Anwendungen, die XML einsetzen, jedoch bisher nur sehr wenige Werkzeuge, die die Verarbeitung von XML unterstützen. Viele Anwendungen verwenden nur einen Teil des Umfangs von XML. Für diese wäre es vorteilhaft, Daten in XML in relationalen oder Objekt-relationalen Datenbanken zu speichern. Damit ließen sich diese Daten innerhalb eines Datenbanksystems verwalten und anfragen.
In dieser Diplomarbeit wurde eine Übersetzung von semistrukturierten Daten in Form von XML-Dokumenten in Objekt-relationale Datenbanken konzipiert. Das Datenbankschema wird auf der Grundlage der DTD für eine spezifische Dokumentenklasse generiert.
Im Rahmen des GETESS-Projekts wurde eine Speicherung der in XML dargestellten Abstracts vorgenommen. Die prototypische Implementierung erfolgte für DB2.

Abstract

The Importance of semistructured data is increasing in a variety of applications. XML is a typical representation of semistructured data. XML is a standard language developed by the W3C for the exchange of documents. A lot of applications use XML. There are but a few tools to support the processing of XML. Many applications employ only parts of the XML standard. The storage of such data in relational or object-relational databases would be an advantage to those applications. Thus the data could be managed and queried within a database system.
This Master thesis introduces a new approach to the mapping of semistructured data represented in XML to object-relational databases. The database schema is generated based on the DTD of a specific class of documents.
The storage of XML documents (abstracts) has been developed for the GETESS-project as a prototyp implementation for DB2.

CR-Klassifikation:

H.2 Database Management
H.2.1 Logical Design - Schema und Subschema
I.7.2 Markup Languages

Stichworte:

semistrukturierte Daten, XML, Mapping, Objekt-relationale Datenbanken, GETESS, DB2

Inhaltsverzeichnis

1 Einleitung **5**

2 Semistrukturierte Daten **6**
2.1 Beispiele semistrukturierter Daten 6
2.1.1 Das World Wide Web 6
2.1.2 Das ACeDB-DBMS 6
2.1.3 Datenintegration 7
2.1.4 Browsen in Daten 7
2.2 Merkmale semistrukturierter Daten 8

3 XML **11**
3.1 Die Wurzeln von XML 11
3.2 Anwendungsgebiete vom XML 12
3.3 Aufbau eines XML-Dokuments 12
3.3.1 Die Struktur eines Dokuments in XML 12
3.3.2 DTD - Eine Grammatik für XML-Dokumente 12
3.4 Ein Beispiel in XML 13

4 Objekt-relationale Datenbanken **14**
4.1 Erzeugen neuer Datentypen 14
4.2 DBMS-Mechanismen 15
4.2.1 Trigger 15
4.2.2 Stored Procedures 15
4.2.3 Nutzerdefinierte Funktionen 15
4.3 Unterstützung flexibler Tabellenstrukturen 16
4.3.1 Zeilen mit mehrwertigen Spalten 16
4.3.2 Komplexe Datentypen 17

5 XML-Speicherung in Datenbanken **19**
5.1 Speicherung nach Florescu/Kossmann 19
5.1.1 Mapping von XML-Daten in relationale Tabellen 19
5.1.2 Abbildung der Kanten 20
5.1.3 Abbildung von Werten 22

5.1.4 Vor- und Nachteile der vorgestellten Methode 23
5.2 Speicherung in relationalen Datenbanken 24
5.2.1 Vereinfachen der DTD 24
5.2.2 Basic Inlining 26
5.2.3 Shared Inlining 27
5.2.4 Hybrid Inlining 27
5.2.5 Vor- und Nachteile dieser Verfahren 28

6 Abbildung von XML in OR-Datenbanken 30
6.1 Abbildung von einfachen Elementen 30
6.2 Abbildung von Element-Blöcken 31
6.2.1 Gruppierung von Elementen und Alternativen 32
6.2.2 Vor- und Nachteile dieses Vorgehens 33
6.2.3 Speicherung in separaten Relationen als Alternative 33
6.2.4 Quantifizierung von Elementen 34
6.2.5 Komplexe Elemente 36
6.2.6 Vor- und Nachteile dieser Methode 37
6.3 Behandlung von Rekursionen 38
6.3.1 Rekursion zwischen zwei Elementen 40
6.3.2 Rekursion über mehrere Stufen 41
6.3.3 Vorteile und Nachteile 42
6.4 Abbildung von Attributen 42
6.4.1 Abbildung auf ein Relationenattribut 43
6.4.2 Abbildung der verschiedenen Attributtypen 43
6.4.3 Abbildung von `ID, IDREF` 43
6.4.4 Unterscheidung von Elementen und Attributen 44
6.5 Abbildung von Entities 45
6.6 Elemente als Typ XML speichern 46
6.6.1 Mixed content 46
6.6.2 Berücksichtigen der Häufigkeit von Elementen 46
6.6.3 Unstrukturierter Text 47
6.7 Schematischer Algorithmus 47
6.7.1 Bildung eines DTD-Graphen 48
6.7.2 Bildung des Relationen-Schemas 48
6.8 Zusammenfassende Übersicht 50

7 Realisierung für das GETESS-Projekt 52
7.1 Entstehung des DB-Schemas aus der Ontologie 52
7.1.1 Informationen aus der Ontologie 52
7.1.2 Algorithmus zur Umsetzung 53
7.1.3 Metainformationen 54
7.2 Implementierung der Speicherung von Abstracts 56

8 Zusammenfassung und Ausblick **58**
8.1 Zusammenfassung 58
8.2 Ausblick 59

A Beispiel zu Abschnitt 5.2 **60**
A.1 DTD für das Beispiel 60
A.2 DTD- und Element-Graph 61
A.3 Generierte Relationenschemata 61

Kapitel 1

Einleitung

Das World Wide Web ist zu einer unerschöpflichen Informationsquelle herangewachsen. Allerdings steigt damit auch die Unübersichtlichkeit und es wird schwieriger, gezielt nach Informationen zu suchen. Die Leistungsfähigkeit der vorhandenen Werkzeuge und Dienste zur Recherche ist jedoch stark eingeschränkt. Sie verwenden für ihre Suche sogenannte syntaktische Attribute, wie z. B. TITLE, beachten dabei aber nicht die eigentliche Bedeutung der Information.
Das GETESS-Projekt [GETESS] dient der Entwicklung eines intelligenten Werkzeuges zur Informationsrecherche im Internet. Der Benutzer soll in die Lage versetzt werden, seine Suchanforderungen in natürlicher Sprache zu formulieren, da die meisten im Internet vorliegenden Informationen ebenfalls in natürlicher Sprache, die durch Multimediainformationen ergänzt wurden, gehalten sind. Den Suchdiensten in GETESS soll ein intelligentes System zur Seite gestellt werden, das aus umfangreichen und komplexen Informationsangeboten sprachunabhängige und konzentrierte Zusammenfassungen, sogenannte *Abstracts*, erzeugt.
Um diese Abstracts effizient zu speichern und einen schnellen Zugriff zu gewährleisten, sollen sie in Datenbanken gespeichert werden. Die Abstracts werden in Form von XML-Dokumenten (E*X*tensible *M*arkup *L*anguage) bereitgestellt. Es gibt wenige Werkzeuge, die XML verarbeiten können. Anwendungen verwenden meist nur eine Teilmenge des Umfangs von XML. Es wäre für sie von Vorteil, wenn sie die XML-Daten in relationalen oder objekt-relationalen Datenbanken speichern. Die Daten ließen sich dann innerhalb eines Datenbanksystems verwalten und anfragen.
Diese Arbeit beschäftigt sich mit der Konzeption einer Übersetzung von semistrukturierten Daten, den Abstracts in XML, in objekt-relationale Datenbanken und der Realisierung für das GETESS-Projekt.

Kapitel 2

Semistrukturierte Daten

Daten treten in verschiedenster Form auf. Das reicht von unstrukturierten Daten, z. B. in Dateisystemen, bis zu stark strukturierten Daten wie in relationalen DBMS. Auf diese Daten kann man mit verschiedenen Mitteln zugreifen, z. B. mit einem Web-Browser, Datenbankanfragesprachen, mit applikationsspezifischen Interfaces oder mittels Datenaustauschformaten. Daten können unstrukturiert sein, wie z. B. Bitmap-Bilder. Andere Daten haben eine implizite Struktur, die keinen Beschränkungen, wie sie in Datenbanksystemen auftreten, unterliegen. In einigen Fällen kann die Struktur aus den Daten extrahiert werden. Auch wenn eine Struktur existiert, ist es, z. B. für das *Browsen* in Daten, von Vorteil, diese zu ignorieren. Als semistrukturierte Daten werden Daten bezeichnet, die weder ganz unstrukturiert noch streng getypt sind.

2.1 Beispiele semistrukturierter Daten

2.1.1 Das World Wide Web

Das augenscheinlichste Beispiel für Daten, die nicht durch ein Schema beschrieben werden können, ist das *World Wide Web*. Es besteht aus Dateien im HTML-Format mit Strukturelementen, sogenannten 'tags'. Anfragen im Web liefern mit Hilfe von Information Retrieval Techniken individuelle Seiten aufgrund ihres Inhalts. Die Struktur des Webs kann bei der Formulierung von Anfragen kaum genutzt werden, da das Web keinem Standarddatenmodell entspricht.

2.1.2 Das ACeDB-DBMS

Ein weiteres Beispiel ist das *Datenbankmanagementsystem ACeDB*, das von Biologen verwendet wird [T-MD92]. Oberflächlich sieht es aus wie ein objekt-orientiertes DBMS, da es eine DDL enthält, die der eines objekt-orientierten DBMS ähnelt. Das damit erzeugte Schema legt den Daten allerdings nur lose Zwänge auf. Die Beziehungen zwischen Daten und Schema sind auch nicht einfach durch objekt-orientierte Ausdrücke beschrieben. Es gibt hier Strukturen, die natürlich ausgedrückt sind, wie zum Beispiel Bäume willkürlicher Tiefe. Diese können nicht mit herkömmlichen Techniken bearbeitet werden.

2.1.3 Datenintegration

Semistrukturierte Daten treten auch auf, wenn man verschiedene, auch strukturierte, Datenquellen zusammenfaßt. So können beispielsweise Adressen als String, aber auch als Tupel in den Quellen auftreten. Es ist eine Heterogenität in der Organisation der Daten zu erwarten. Semistrukturierte Daten treten in verschiedener Form in vielen Applikationen auf, so bei wissenschaftlichen Datenbanken, Programmbibliotheken, Online-Dokumentationen, dem E-Commerce.
Im Tsimmis-Projekt [GPQ+95, PGW95] in Stanford geht man davon aus, daß kein existierendes Datenmodell allumfassend ist. Die Entwicklung von Konvertierungssoftware wird dementsprechend schwierig eingeschätzt. Das *Objekt Exchange Model* (OEM) bietet eine hochflexible Datenstruktur, mit der man die meisten Arten von Daten darstellen kann. OEM ist eine interne Datenstruktur zum Austausch von Daten zwischen Datenbankmanagementsystemen, die dazu einlädt, auf den Daten im OEM-Format Anfragen zu stellen.

2.1.4 Browsen in Daten

Für das Browsen in Daten ist ein Konzept wie das semistrukturierter Daten wünschenswert. Ein Nutzer kann keine Anfrage an eine Datenbank stellen, ohne das Schema zu kennen. Ein solches Schema ist nicht immer sofort zugänglich und verständlich für den Nutzer. Hier könnte es hilfreich sein, Anfragen stellen zu können, ohne volles Wissen um das Schema. Zum Beispiel Anfragen wie:

- Welche Attributnamen der Datenbank beginnen mit 'Act'?

- Wo kann ich den String 'Romeo' in der Datenbank finden?

können nicht beantwortet werden mit den Standard-Relationalen oder objektorientierten Anfragesprachen.

2.2 Merkmale semistrukturierter Daten

Die hier aufgeführten Merkmale semistrukturierter Daten gehen auf Serge Abiteboul [Abi97] zurück.
Die folgenden Merkmale charakterisieren semistrukturierte Daten:

1. Die Struktur ist irregulär:
Die Datenmengen aus unterschiedlichen Quellen bestehen aus heterogenen Elementen. Einige sind unvollständig, andere können zusätzliche Informationen enthalten (Kommentare). Es können auch verschiedene Typen für die gleiche Art Information benutzt werden.

2. Die Struktur ist implizit:
In vielen Applikationen ist die Struktur implizit gegeben, z. B. bestehen Dokumente oft aus Text und einer Grammatik (z. B. DTD in SGML). Durch Parsen der Daten können Informationsteile extrahiert und Beziehungen zwischen ihnen erkannt werden. Die Interpretation der Beziehungen liegt außerhalb der Möglichkeiten von Standard-Datenbankmodellen und werden von speziellen Applikationen vorgenommen. Die Struktur wird als implizit angesehen, da:

- eine Verarbeitung der Daten (Parsen) nötig ist, um sie zu erhalten
- nicht immer ein direkter Zusammenhang zwischen parse-tree und logischer Repräsentation der Daten existiert.

3. Die Struktur ist partiell:
Daten vollständig zu strukturieren kann nahezu unmöglich sein. Einige Teile weisen keine Struktur auf (Bitmaps), andere haben nur eine sehr grobe Struktur (unstrukturierter Text). Information Retrieval Tools liefern eine begrenzte Form der Datenstruktur, z. B. durch Bestimmen des Auftretens bestimmter Wörter oder Wortgruppen und durch die Klassifizierung von Daten aufgrund ihres Inhalts.
Große Mengen von Daten können von einer Applikation auch außerhalb einer Datenbank gespeichert werden. Aus DB-Sicht sind diese Daten unstrukturiert. Das Laden der externen Daten, ihre Analyse und Integration in die Datenbank müssen effizient geschehen.

4. Indikative Struktur gegen 'Constraint'-Struktur:
In Standard-Datenbankapplikationen wird eine strenge Typisierung zum Schutz der Daten eingeführt. Solche strengen Praktiken werden bei einigen Applikationen als zu einschränkend angesehen. Jemand, der beispielsweise einen persönlichen Website entwickelt, würde eine strenge Typenrestriktion nur widerstrebend akzeptieren.

5. A-priori Schema gegen a-posteriori Data Guide:
Traditionelle Datenbanksysteme basieren auf der Hypothese, daß ein festes Schema definiert wird, bevor die Daten eingeführt werden. Bei semistrukturierten Daten ist das nicht der Fall. Hier wird man ein Schema aus den bereits vorhandenen Daten erstellen.

6. Das Schema ist sehr groß:
Bedingt durch die Heterogenität ist das Schema vergleichsweise sehr groß. Im Vergleich zur Größe des entstehenden Schemas fällt die darin gespeicherte Datenmenge verhältnismäßig klein aus.
Als Konsequenz daraus muß der Nutzer nicht mehr das ganze Schema kennen. Damit werden Anfragen zum Schema genauso wichtig wie Anfragen zu den Daten.

7. Das Schema wird ignoriert:
Für Anfragen, deren Funktion darin besteht, Informationen zu 'entdecken', ist es sinnvoll, das Schema zu ignorieren. Solche Anfragen können einfaches Browsen durch die Daten sein oder das Suchen nach Strings oder Mustern ohne genaue Angaben, wo sie auftreten können. Mit SQL-artigen Sprachen ist das so nicht möglich. Es erfordert eine Erweiterung der Anfragesprachen und die Integration neuer Optimierungstechniken, wie Volltextindizierung [ACC+96] oder die Ermittlung von verallgemeinerten Pfadausdrücken [CCM96].

8. Das Schema entwickelt sich rapide:
In Standard-Datenbanksystemen gilt das Schema als unveränderlich, Schema-Updates sind selten und gelten als sehr teuer.
Das gilt nicht im Falle von Genomen Daten (*genome data*). Es wird erwartet, daß sich hier das Schema ändert, wenn experimentelle Techniken verbessert oder neuere Techniken eingeführt werden. Formate wie ACeDB werden dafür relationalen oder objekt-orientierten Datenbanken vorgezogen.
In Zusammenhang mit semistrukturierten Daten kann man davon ausgehen, daß das Schema sehr flexibel ist und sich so schnell wie die Daten selbst ändern kann.

9. Der Typ der Datenelemente ist variabel:
Ein anderer Aspekt ist, daß die Struktur von Datenelementen von der Betrachtungsweise oder von einer Phase im Verarbeitungsprozeß der Daten abhängt. Der Typ muß variabler sein als in Standard-Datenbanksystemen, in denen die Struktur eines Rekords oder Objekts sehr präzise ist. Z. B. kann ein Objekt eine Datei sein. Nach der Klassifizierung durch ein entsprechendes Tool ist es eine BibTeX-Datei. Es kann Felder wie *owner*, *creation-date* und weitere nach einer Informationsextraktionsphase enthalten. Und schließlich wird es eine Menge von Referenzobjekten mit komplexer Struktur, nachdem es von einem Parser verarbeitet wurde.

10. Die Unterscheidung zwischen Schema und Daten ist undeutlich: Standard-Datenbanksysteme unterscheiden zwischen Schema und Daten. Diese Unterscheidung verschwindet bei semistrukturierten Daten. Es kommt zu häufigen Schema-Updates, das Schema kann sehr groß werden, die selben Anfragen/Updates betreffen sowohl die Daten als auch das Schema. Auch macht die Unterscheidung zwischen Schema und Daten bei semistrukturierten Daten logisch gesehen wenig Sinn, da die gleiche Art Information in einer Quelle als Daten, in einer anderen als Typ zum Schema gehörend, enthalten sein kann.

Kapitel 3

XML

Die Abkürzung XML steht für E*X*tensible *M*arkup *L*anguage. Die Sprache wurde vom World Wide Web Consortium (W3C) entwickelt [BPS+98, Bra98].

3.1 Die Wurzeln von XML

XML basiert auf den Prinzipien und Konventionen von HTML und SGML. Obwohl primär auf SGML aufbauend, sind in XML einige Charakteristiken von HTML und weitere Features für die Nutzung im Internet integriert.
Mit SGML [Gol90] wurde 1986 ein Standard für ein allgemeines Auszeichnen (*generalized markup*) eingeführt. Aufgrund der Mächtigkeit von SGML ist es allerdings sehr aufwendig, Software zu entwickeln, die auch nur eine typische Untermenge von SGML unterstützt. Vergleichbare Lösungen sind schneller, weniger fehleranfällig und kostengünstiger. Es wurde daher als notwendig erachtet, eine vereinfachte Version von SGML zu entwickeln, um das allgemeine Auszeichnen attraktiver zu machen.
HTML erlaubt die Verknüpfung von Dokumenten im Internet durch Links und bietet eine einfache Textformatierung an. Die Syntax von SGML wurde in HTML größtenteils übernommen, die Prinzipien blieben zunächst außen vor. Sie wurden im späteren Versionen integriert. Entwickler von Web-Browsern brachten in HTML eigene Erweiterungen ein, um einen kommerziellen Vorteil zu erzielen. Diese sind allerdings inkompatibel zum bestehenden Standard. Davon abgesehen existiert keine Möglichkeit für den Autor eines Dokuments, die Sprache durch eigene HTML-tags zu erweitern.
Im Jahre 1996 definierte das W3C 10 Designziele für XML und veröffentlichte am 10. Februar 1998 XML in der Version 1.0.

3.2 Anwendungsgebiete vom XML

XML ist ein ideales Datenformat zur Speicherung von strukturiertem und semistrukturiertem Text, der publiziert werden soll. Ein XML-Dokument enthält spezielle Elemente, sogenannte *tags*, die identifizierbare Teile des Dokuments einschließen. Folgendes Beispiel verdeutlicht dies:

```
<beispiel>
<text> Dies ist ein Text. </text>
<zahl> 12345 </zahl>
</beispiel>
```

Der identifizierbare Teil ist in diesem Beispiel durch von eckigen Klammern umgebenen tags eingeschlossen.

3.3 Aufbau eines XML-Dokuments

3.3.1 Die Struktur eines Dokuments in XML

Ein XML-Dokument hat eine logische und eine physische Struktur. Das Dokument ist logisch in *Elemente* und *Subelemente* unterteilt. Ein Element kann Subelemente enthalten. Diese Subelemente können ebenfalls aus Subelementen bestehen usw. Physisch können Komponenten des Dokuments separat bezeichnet und gespeichert werden. Diese Komponenten heißen *Entities*. Diese Entities können in eigenen Dateien gespeichert werden. Das sind auch Daten, die nicht im XML-Format vorliegen, wie z. B. Bilder, die dann referenziert werden.
Ein XML-Prozessormodul wird für das Entitiy-Management benutzt. Es faßt die Entities in einen einzelnen Datenstrom für die Validierung durch einen *Parser* oder die Verwendung in einer Applikation zusammen.

3.3.2 DTD - Eine Grammatik für XML-Dokumente

XML ist eine Metasprache. Mit ihr lassen sich andere Sprachen beschreiben. Es gibt keine vordefinierte Liste von Elementen. Der Nutzer kann Elemente nach seinen Wünschen bezeichnen und benutzen. Dennoch existiert ein optionaler Mechanismus, der es erlaubt, Elemente zu spezifizieren, die in einer speziellen Klasse von Dokumenten erlaubt sind. Eine solche *DTD (Document Type Definition)*, die einen speziellen Dokumententyp definiert, gewährleistet ein hohes Maß an Kontrolle über die logische Struktur eines Dokuments. Diese DTD kann für die Validierung eines Dokuments durch einen *validierenden Parser* herangezogen werden.

3.4 Ein Beispiel in XML

Im GETESS-Projekt werden sogenannte *Abstracts* durch XML-Dokumente dargestellt. Dieses Beispiel zeigt das Aussehen eines Abstracts:

XML-Struktur für Unterkunft:

```
<Unterkunft>
<klassifizierung> ... </klassifizierung>
<saison> ... </saison>
<verkehrsanbindung> ... </verkehrsanbindung>
<verpflegung> ... </verpflegung>
...
</Unterkunft>
```

Dieses Beispiel verdeutlicht die Struktur eines XML-Dokuments. Es hat ein Element `Unterkunft` und mehrere Subelemente, wie `saison` etc. Die dazugehörige DTD hat folgendes Aussehen:

DTD für Unterkunft:

```
<!ELEMENT Unterkunft (klassifizierung, saison,
                      verkehrsanbindung, verpflegung, ...)>
<!ELEMENT klassifizierung (#PCDATA)>
<!ELEMENT saison (#PCDATA)>
<!ELEMENT verkehrsanbindung (#PCDATA)>
<!ELEMENT verpflegung (#PCDATA)>
...
```

Die DTD für Unterkunft spezifiziert die Elemente aus dem Beispiel darüber. Es legt das Element `Unterkunft` als eine Liste seiner Subelemente fest. Die Zeichen `?` und `*` schränken das Auftreten der Subelemente ein. Das `?` ist ein Indikator für optionales Auftreten. Das Zeichen `*` gibt zusätzlich an, daß das Element auch mehrfach auftreten kann. Der Typ der Subelemente ist mit `PCDATA` spezifiziert. Dabei handelt es sich um einen Stringtyp.

Kapitel 4

Mächtigkeit von Objekt-relationalen Datenbanken

Der Wunsch nach einer effektiven Verwaltung vieler gleicher Daten führte zur Entwicklung von Datenbanksystemen. Zunächst wurden mit ihnen Daten wie z. B. Finanzdaten, Kundendaten, Angestelltendaten usw. verwaltet. Diese konnten leicht durch Charakterstrings und Zahlentypen dargestellt werden. Bilder, Volltextdokumente, geographische Daten (z. B. Landkarten) und andere Typen von Daten spielen in vielen Bereichen inzwischen eine wichtige Rolle. Solche Daten lassen sich in traditionellen relationalen DBMS mit den vorhandenen einfachen Datentypen kaum oder gar nicht darstellen. Aus diesem und weiteren Gründen entwickeln die Hersteller von relationalen DBMS ihre Produkte zu 'universellen' DBMS [Sar98].
Allgemein versucht ein universelles DBMS einen größeren Umfang von Datentypen zu unterstützen. Dazu gehört auch die Unterstützung von Analysefunktionen für diese Datentypen. Um dies zu erreichen, werden in den meisten Fällen objekt-orientierte Konzepte integriert. Diese Erweiterungen können substantiell sein und viele Komponenten des DBMS betreffen. Die Mischung objekt-orientierter Konzepte mit relationalen Möglichkeiten findet sich in den Objekt-relationalen Datenbanksystemen [SB99], wie sie viele Hersteller bereits anbieten. Dazu gehören zum Beispiel DB2 von IBM [Cha98] und INFORMIX Universal Server von Informix Software Inc. [Pet98]

4.1 Erzeugen neuer Datentypen

Objekt-relationale Datenbanksysteme bieten die Möglichkeit, ein flexibles Datentyp-System zu erstellen. Damit können Anwender neue Typen einführen, wenn sie benötigt werden. Zu den bekannten einfachen Typen, z. B. `CHARACTER, FLOAT, TIME`, können eigene Typen wie `ACCOUNT` oder `EMPLOYEE_ID` hinzugefügt werden.

In objekt-orientierten Sprachen geschieht das durch die Definition neuer Typen oder Klassen.
Dieses Konzept ist in ORDBMS in den neuen SQL-Statements für die DDL (Data Definition Language) zu finden. Neue Typen können aus den vorhandenen Datentypen gebildet werden und das sowohl aus den im DB-System enthaltenen Datentypen, als auch aus selbstdefinierten Typen.

4.2 DBMS-Mechanismen

Das ORDBMS stellt dem Nutzer eine Reihe von Mechanismen zur Verfügung, um mit den neuen Datentypen arbeiten zu können. Diese Mechanismen sind in ihrer Funktion den Methoden in OO-Programmiersprachen sehr ähnlich.

4.2.1 Trigger

Trigger sind von Anwendern geschriebene Programme, die immer dann ausgeführt werden, wenn in der Datenbank ein Ereignis eintritt, z. B. das Eintragen von Werten in eine bestimmte Spalte einer Tabelle oder das Löschen von Zeilen. Trigger machen DBMS mehr 'aktiv', lassen es automatisch reagieren. Einige Möglichkeiten von Methoden, wie sie bei der OO-Programmierung zu finden sind, lassen sich mit Triggern nachbilden.

4.2.2 Stored Procedures

Im Gegensatz zu Triggern werden Stored Procedures, ebenfalls nutzerspezifischer Programmcode, nur durch einen spezifischen Aufruf ausgeführt. Sie können mehrere Eingabeparameter erwarten und Ergebnisse (eine Menge von Zeilen) an den Nutzer oder die aufrufende Applikation zurückgeben. In einigen Systemen können sie in OO-Programmiersprachen wie `C++` oder `Java` geschrieben werden.

4.2.3 Nutzerdefinierte Funktionen

Zu den in relationalen DBMS vorhandenen Basisdatentypen gibt es eine Anzahl spezieller Funktionen, die in SQL-Anfragen aufgerufen werden können. Dazu gehören Funktionen wie `sum` oder `average` für verschiedene numerische Datentypen.
Zur effektiven Analyse selbst-definierter Datentypen ist es möglich, solche Funktionen in Objekt-relationalen Datenbankmanagementsystemen zu definieren. Die-

se können dann auch in SQL-Anfragen verwendet werden.
Dieses Konzept ähnelt stark den Methoden in der OO-Programmierung. Beide unterstützen die Wiederverwendung von Programmcode. Sie erlauben beide die Definition von Operationen über spezifischen Daten.

4.3 Unterstützung flexibler Tabellenstrukturen

Mit der Möglichkeit neue Typen und Funktionen erzeugen zu können, werden objekt-orientierte Konzepte in ORDBMS integriert. Objekt-orientierte Programmiersprachen erlauben mit der Definition von neuen Klassen von Objekten eine Vielfalt an internen Strukturen.
Die Suche nach Varianten in Datenbanken zu traditionellen Tabellenstrukturen resultierte aus Gründen der Performance und der Datenmodellierung. Beispielsweise sollen so Join-Operationen zwischen mehreren Tabellen minimiert bzw. eliminiert werden, da Joins in der Ausführung teure relationale Operationen sind.

4.3.1 Zeilen mit mehrwertigen Spalten

In ORDBMS können Tabellen mehrwertige Spalten und komplexe Werte enthalten. In einer einzelnen Spalte einer Zeile können mehrere Datenelemente enthalten sein. Eine solche Spalte beinhaltet eine Menge von Datenelementen statt eines einzelnen Datenelements. Es stehen dafür verschiedene Arten solcher 'Sammlungen' zur Verfügung:

- Set : Sets sind ungeordnet und enthalten keine Duplikate.
- Bag : Bags sind ungeordnet und dürfen Duplikate enthalten.
- List : Listen sind geordnet und können Duplikate enthalten.
- Array : Arrays sind geordnet, können Duplikate enthalten. Jedes Element ist direkt zugreifbar. Die Größe ist vorgegeben.

Das folgende Beispiel zeigt eine Tabelle mit einer Menge in der Spalte `SKILLS`:

Tabelle `EMPLOYEE`

ID	NAME	JOBTITLE	SKILLS
123	C.Ward	Analyst	{UNIX, NT, TCP/IP}
480	B. Perez	Engineer	{MVS, C, COBOL}
...	...	...	...

Eine solche Tabelle läßt sich mit der folgenden SQL-Anweisung erzeugen:

```
CREATE TABLE EMPLOYEE (

    ID EMPLOYEE_ID,
    NAME VARCHAR(50),
    JOBTITLE VARCHAR(50),
    SKILLS SET(VARCHAR(20))

)
```

Zur Erzeugung von Mengen sind entprechende Konstruktoren, in diesem Beispiel `SET`, vorhanden. Die Anwendung ist im obigen Beispiel zu sehen.

4.3.2 Komplexe Datentypen

Zeilen mit komplexen Datentypen haben Spalten, die eine interne Struktur besitzen. Statt eines einzelnen einfachen Datenwerts bestehen diese Spalten aus Datenwerten, die mehrere Attribute (= Felder) mit verschiedenen Datentypen besitzen.

Die Möglichkeit neue Datentypen zu definieren, erlaubt es zum Beispiel einen Typ `ADDRESS` zu erstellen. Dieser Datentyp kann dann in beliebigen Tabellen-Definitionen für eine Spalte verwendet werden.

```
CREATE TYPE ADDRESS AS (

    (STREET VARCHAR(30),
    QUALIFIER VARCHAR(20),
    CITY VARCHAR(20,),
    STATE CHAR(2),
    ZIP INTEGER

) INSTANTIABLE
NOT FINAL
```

Diese Definition beschreibt die Attribute des Typs `ADDRESS`.
Die beiden letzten Zeilen stehen in Zusammenhang mit der Typ-Vererbung. In diesem Fall bedeutet `INSTANTIABLE`, daß Instanzen des Typs gebildet werden können. Das bedeutet schlicht, daß in Tabellen mit Attributen, die mit diesem Typ definiert sind, auch Werte gespeichert werden können. Mit `NOT FINAL` wird spezifiziert, daß man Untertypen von `ADDRESS` kreieren kann.
Mit diesem Typ können sowohl Attribute als auch ganze Relationen definiert werden.
Im folgenden Beispiel wird eine Relation angelegt, die ein Attribut `HOME_ADDRESS` enthält, das mit dem Typ `ADDRESS` definiert wird.

```
CREATE TABLE EMPLOYEE (

    (ID EMPLOYEE_ID,
    NAME VARCHAR(30),
    HOME_ADDRESS ADDRESS,
    JOBCODE INTEGER

)
```

Damit erhält die Tabelle folgendes Aussehen:

Tabelle `EMPLOYEE`

ID	NAME	HOME_ADDRESS					JOBCODE
		STREET	QUALIFIER	CITY	STATE	ZIP	
...	...	...	...	...	...	...	...

In den existierenden Systemen sind diese Techniken in unterschiedlichem Maße integriert. `INFORMIX` erlaubt die Definition solcher geschachtelten Attribute. In `DB2` von IBM ist dies nicht möglich.

Komplexe Typen können auch für die Definition der Tabellenstruktur und nicht nur einer Spalte verwendet werden. In diesem Fall enthält der Typ alle Attribute der Tabelle. Vorausgesetzt, es wurde ein Typ `PROJECT_TYPE` definiert, kann eine Tabelle `PROJECT` definiert werden durch:

```
CREATE TABLE PROJECT OF PROJECT_TYPE
```

Weiterhin lassen sich Untertypen ableiten und mit ihnen ganze Tabellenhierarchien erstellen. Dieses Konzept der Vererbung ist aus der OO-Programmierung hinlänglich bekannt. Eine Tabellenhierarchie wird auch bei der Ausführung von Anfragen berücksichtigt. D. h., wenn ein `Select`-Statement an eine Relation gestellt wird, zu der Relationen mit einem erweiterten Untertyp existieren, werden diese für die Bildung der Ergebnisse zusammengefaßt. Ein Einfügen von Werten wird dagegen nur in der Relation ausgeführt, die in dem `Insert`-Statement genannt wird. Relationen mit einem abgeleiteten Typ sind davon nicht betroffen.

Kapitel 5

Ansätze zur Speicherung von XML in Datenbanken

In diesem Kapitel werden verschiedene Ansätze zur Speicherung von XML-Dokumenten in Datenbanken vorgestellt. Anhand dieser Beispiele werden Vor- und Nachteile dieser Ansätze diskutiert.

5.1 Speicherung in einem RDBMS nach Florescu/Kossmann

Als ein Ansatz zur Speicherung von XML-Daten in relationalen DBMS wird hier das von Daniela Florescu und Donald Kossmann zu Testzwecken entwickelte Verfahren vorgestellt [FK99]. Sie untersuchten die Performance des, wie sie es nannten, einfachsten und offensichtlichsten Vorgehens bei der Verwaltung von XML-Dokumenten in Hinsicht auf das Abspeichern selbst und das Anfragen der gespeicherten Daten. Für die Erzeugung der Datenbank-Schemata wurden keine Eingaben des Anwenders, keine DTD und keine Analyse der XML-Daten benötigt.

5.1.1 Mapping von XML-Daten in relationale Tabellen

Am Anfang des Abbildungsprozesses steht eine Menge von XML-Dokumenten. Diese Dokumente werden einzeln von einem Parser bearbeitet und die gewonnenen Informationen in relationalen Tabellen gespeichert. Zur Vereinfachung betrachten die Autoren ein XML-Dokument als einen geordneten gerichteten Graphen. Jedes XML-Element wird durch einen Knoten im Graphen dargestellt, wobei jedem Knoten der `id`-Wert des `ID`-Attributs des XML-Objekts als Label zugeordnet wird. Besitzt ein Objekt kein `ID`-Attribut, wird eine entsprechende

ID vom System generiert. Element-Subelement-Beziehungen werden durch die Kanten des Graphen dargestellt. Der Name des Subelements wird das Label der Kante. Die Ordnung der Subelemente wird in die Anordnung der von einem Knoten ausgehenden Kanten aufgenommen. Die Werte eines XML-Elements bilden die Blätter des Graphen.
Die Autoren Florescu und Kossmann geben 3 Alternativen zur Speicherung der Kanten und 2 zur Speicherung der Werte an, die sich kombinieren lassen, so daß 6 Möglichkeiten zur Abbildung von XML-Daten angeboten werden.
Um zu zeigen, wie die dabei entstehenden Schemata aussehen, folgt zunächst ein Beispiel eines XML-Dokuments:

```
<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 Fruitdale Ave. </address>
       </person>
    </child>
</person>

<person> <id='2' age='38' child='4'>
   <name> Mary </name>
   <address> 4711 Fruitdale Ave. </address>
   <hobby> painting </hobby>
</person>
```

5.1.2 Abbildung der Kanten

Der *'Kanten'*-Ansatz

Die einfachste Methode der Speicherung ist, die Kanten des Graphen in einer einzelnen Tabelle zu speichern, der sogenannten Kantentabelle (*edge table*). In der Tabelle werden die `OID` der Quellobjekte und des Zielobjektes jeder Kante

des Graphen, sein Label, ein Flag zur Unterscheidung, ob die Kante eine Inter-Objekt-Referenz darstellt oder auf einen Datenwert (Blatt) verweist, sowie eine Ordnungsnummer gespeichert.
Die Kantentabelle für das XML-Beispiel hat folgendes Aussehen:

Tabelle `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	...	...	...
...	...	...	...	...

Tabelle `VALUE_INT`

vid	value
v1	55
v4	38
v8	22
v13	7
...	...

Tabelle `VALUE_STRING`

vid	value
v2	Peter
v3	4711 Fruitdale Ave.
v5	Mary
...	...

Dieses Beispiel zeigt eine Möglichkeit zur Speicherung von Werten in separaten Tabellen. Dazu später mehr. Die Nummern in der `target`-Spalte sind die `OID's` der Zielobjekte. Die anderen Werte der Spalte verweisen auf die Darstellung der Werte in den separaten Tabellen.

Der *Binary*-Ansatz

Ein zweiter Ansatz ist, alle Kanten mit gleichem Namen in einer Tabelle zu speichern. Konzeptuell ist dieses Vorgehen ein horizontales Aufspalten der `EDGE`-

Tabelle, wobei z. B. `name` als Attribut des Aufspaltens benutzt wird:

BINARY_name (source, ordinal, flag, target)

Es werden so viele Binary-Tabellen erzeugt, wie es verschiedene Subelement- und Attributnamen im XML-Dokument gibt.

Eine Universelle Tabelle

Beim dritten Ansatz werden alle Kanten in einer Tabelle gespeichert. Diese Tabelle entspräche dem vollen Outer-Join aller Binary-Tabellen. Die Struktur ist die folgende, wobei n1, ..., nk die Labelnamen sind:

Universal(source, ordinal_n1, flag_n1, target_n1, ..., ordinal_nk, flag_nk, target_nk)

Für das XML-Beispiel hat die Tabelle dann dieses Aussehen:

Tabelle `Universal`

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	...	...	...	...	...	...	...	...

5.1.3 Abbildung von Werten

Es erfolgt nun eine kurze Beschreibung der 2 Alternativen zur Speicherung von Werten, wie es Florescu/Kossmann vorsehen. Die Werte eines XML-Objekts können a) in separaten Tabellen und b) zusammen mit den Kanten gespeichert werden. Diese Varianten in Verbindung mit den Möglichkeiten zur Kantenspeicherung ergeben 6 verschiedene Mapping-Schemata.

Speichern in separaten Tabellen

Für jeden Datentyp wird eine eigene Wertetabelle gebildet. Z. B. gibt es Tabellen für alle Integer-, Date- und String-Werte. Die Struktur hat dieses Aussehen:

VALUE_TYPE (vid, value)

Der Typ von `value` entspricht dem `type` der Wertetabelle. Die `vid` wird als Teil der Implementierung des Mapping generiert. Dieser Ansatz ist im Beispiel zu der `EDGE`-Tabelle gezeigt. Die `flag`-Spalte aus der `EDGE`-Tabelle gibt an, in welcher Wertetabelle der Wert gespeichert ist (z. B. `int` für `VALUE_INT`).

Speichern mit *Inlining*

Die andere Alternative ist die gemeinsame Speicherung von Kanten und Werten in einer Tabelle. Man kann sie z. B. aus der `EDGE`-Tabelle und den Wertetabellen durch einen Outer Join erzeugen. Dieses Verfahren wird als *Inlining* bezeichnet. Folgende Tabellen sind ein Beispiel für Inlining in Binary-Tabellen:

Tabelle `BINARY_HOBBY`

source	ord	value_int	val_string	target
2	5	NULL	painting	NULL
3	4	NULL	swimming	NULL
3	5	NULL	cycling	NULL

Tabelle `BINARY_CHILD`

source	ord	value_int	val_string	target
1	4	NULL	NULL	3
1	5	NULL	NULL	4
2	4	NULL	NULL	4

5.1.4 Vor- und Nachteile der vorgestellten Methode

Vorteile

Ein Vorteil dieser Abbildungsmethode ist offensichtlich. Jedes beliebige XML-Dokument kann auf diese Weise in relationalen Tabellen gespeichert werden. Das Schema der Tabellen ist unabhängig von einer DTD oder weiteren Eingaben eines Anwenders. Die Hierarchie eines XML-Dokuments läßt sich so einfach und direkt abbilden.

Nachteile

Da keine Berücksichtigung der DTD oder des Inhalts der XML-Dokumente vorgenommen wird, entstehen zwar im Falle der Verwendung von `EDGE`-Tabellen und Wertetabellen sehr wenige Tabellen, im Falle von `Universal` und `Inlining` gar

nur eine. Diese sind dafür aber sehr umfangreich.
Im Falle von Binary wird das Dokument in viele kleine Tabellen, die jeweils nur ein Element enthalten, aufgespalten. Durch die Aufnahme der Werte zu ihren Elementen, die ebenfalls nur als Attribute behandelt werden, entstehen viele `NULL`-Werte in Spalten für die Datentypen, für die ein Element keine Werte besitzt. Viele Informationen sind redundant gespeichert, siehe Beispiel für `Universal`-Tabelle in Abschnitt 4.3.3. (z. B. 'Peter').
Die fehlende Unterscheidung zwischen Elementen und Attributen läßt sich durch Protokollieren in entprechenden Meta-Tabellen integrieren. Ohne sie ist eine Restaurierung des XML-Dokuments nicht möglich.
Die Schemata der Tabellen enthalten Informationen, die weder ein Äquivalent als Element noch als Attribut im XML-Dokument besitzen. Die Spalte `target` in den Beispielen ist ein solcher Fall.
Die in diesen Ansätzen gewählten Schemata der Tabellen erschweren die Formulierung von Anfragen an die gespeicherten Informationen, da zum Beispiel zur Bestimmung eines Werts eines Elements erst das Element, das in diesem Schema selbst einen Wert darstellt, gesucht werden muß. Die gewünschte Information kann erst anschließend durch einen Join gefunden werden.

5.2 Speicherung in relationalen Datenbanken

Ein anderer Ansatz zur Speicherung von XML-Dokumenten in Relationalen Datenbanken wird von mehreren Autoren in [STH+99] vorgestellt. Die Existenz einer DTD ist in diesem Ansatz für die Generierung von relationalen Schemata zwingende Voraussetzung. Schwierigkeiten entstehen dabei einmal mit der Komplexität der Element-Spezifikation in der DTD, dem Konflikt zwischen der 2-Ebenen-Natur relationaler Schemata (Tabelle - Attribut) und mit mehrwertigen Attributen und Rekursionen.
Die angebotenen Lösungen werden im folgenden dargestellt.

5.2.1 Vereinfachen der DTD

Relationale Schemata, die aus der DTD generiert werden, können sehr umfangreich werden, um die Komplexität der DTD wiederzuspiegeln. Es wird daher eine Möglichkeit angeboten, die Details einer DTD zu vereinfachen und dennoch Dokumente speichern und anfragen zu können, die der DTD entprechen. Wichtig ist, daß jedes Dokument zu der entprechenden DTD im relationalen Schema gespeichert und jede semi-strukturierte Anfrage über ein XML-Dokument, das konform zur DTD ist, über der relationalen Datenbankinstanz evaluiert werden kann.
Ein Großteil der Komplexität der DTD steckt in der komplexen Spezifikation des

Typs eines Elements. Quantifizierung und Gruppierung sind die Ursache. Aus Sicht der Anfrageebene ist einzig von Bedeutung, an welcher Stelle ein Element im XML-Dokument steht. Daher wird eine Menge von Transformationen zur 'Vereinfachung' von DTDs vorgeschlagen. Diese Transformationen beeinflussen nicht die Effizienz von Anfragen an Dokumente zu dieser DTD.
Die verwendeten Transformationen sind eine Obermenge ähnlicher Transformationen, wie sie in [DFS99] vorgestellt wurden. Folgende Transformationen fanden Verwendung:

(e1, e2)* − > e1*, e2*

(e1, e2)? − > e1?, e2?

(e1|e2) − > e1?, e2?

e1** − > e1*

e1*? − > e1*

e1?* − > e1*

e1?? − > e1?

..., a* , ..., a* , ... − > a* , ...

..., a* , ..., a? , ... − > a* , ...

..., a? , ..., a* , ... − > a* , ...

..., a? , ..., a? , ... − > a* , ...

..., a , ..., a , ... − > a* , ...

Es gibt 3 Arten von Transformationen. Erstens solche, die die geschachtelte Definition auflösen. Z. B. sollen das Zeichen ',' und die Operation '|' nicht innerhalb anderer Operationen auftreten. Zweitens gibt es Transformationen, die viele unäre Operationen in einen unären Operator überführen. Und drittens Transformationen, die Elemente und Subelemente mit gleichem Namen zu einem Element zusammenfassen. Zusätzlich werden alle '+'-Operatoren in '*'-Operatoren transformiert.
Diese Transformationen erhalten die Semantik von 'einem oder vielen' und Null oder nicht Null.
Es wird von den Autoren angemerkt, daß die Information über die *relative Ordnung der Elemente* bei dieser Transformation verlorengeht. Diese Information kann festgehalten werden, wenn ein spezifisches XML-Dokument in das relationale Schema geladen wird.
Ausgehend von einer transformierten DTD werden verschiedene Techniken zur Konvertierung vorgestellt. Ein Beispiel einer DTD, die verwendeten Graphen und die generierten Relationen sind aufgrund ihres Umfangs im Anhang A dargestellt worden.

5.2.2 Basic Inlining

Bei dieser Technik wird versucht, soviele Nachfolger eines Elements wie möglich in derselben Relation zu speichern, um eine zu große Fragmentierung zu vermeiden. Allerdings entsteht bei dieser Methode für jedes Element eine eigene Relation.
Es werden 2 Fälle als problematisch angesehen: *mengenwertige Attribute* und *Rekursionen*. Bei mengenwertigen Attributen wird die Standardtechnik zur Speicherung von Mengen in RDBMS verfolgt. Das entprechende Attribut bildet eine eigene Tabelle und wird mit der Relation der DTD mittels Fremdschlüssel verbunden.
Um den Grad des Nesting in der Rekursion zu begrenzen, werden rekursive Beziehungen durch die Verwendung von relationalen Schlüsseln ausgedrückt. Die Beziehung wird mit relationalen rekursiven Prozessen bestimmt. Grundlage dafür sind Graphen, die die Struktur der DTD wiedergeben (s. Anhang A). Die Knoten sind dabei die Elemente, Attribute und Operatoren. Jedes Element tritt exakt einmal im Graphen auf. Die Attribute und Operatoren erscheinen so oft, wie sie in der DTD auftauchen. Zyklen im Graph repräsentieren Rekursionen.

Das Schema besteht aus der Vereinigung der Relationen, die für jedes Element gebildet wurden. Zur Bestimmung der Menge der für ein Element zu generierenden Relationen wird ein sogenannter Element-Graph erzeugt. Dabei wird folgendermaßen vorgegangen:
Auf ein Element, für das Relationen erzeugt werden sollen, wird ein *deep first*-Algorithmus angewandt. Jeder Knoten wird markiert, wenn er das erste Mal durchlaufen wird. Die Markierung wird zurückgenommen, wenn alle seine Nachfolgeknoten besucht wurden.
Wird ein unmarkierter Knoten im DTD-Graph erreicht, so wird ein Knoten mit gleichem Namen im Element-Graph erzeugt. Zusätzlich wird eine reguläre Kante zu dem neuen Knoten von dem Knoten im Element-Graph erzeugt, der unter gleichem Namen im DFS-Pedant (*D*eep *F*irst *S*trategie) des DTD-Graphen eine Kante zum zuletzt eingefügten namensgleichen Knoten unterhält.
Wird ein markierter Knoten ein zweites Mal durchlaufen, fügt man eine *Backpointer*-Kante zwischen dem letzten eingefügten Knoten und dem markierten Knoten in den Element-Graphen ein.
Eine Relation wird für das Wurzelelement des Element-Graphen generiert. Alle darunter liegende Elemente bzw. Knoten werden als Attribute in diese Relation aufgenommen. Davon ausgenommen sind Knoten, die auf einen '*'-Knoten folgen. Diese werden in einer eigenen Relation gespeichert. Auch Knoten, auf die ein Backpointer verweist, werden in separate Relationen ausgelagert, um die dahinter stehende Rekursion zu handhaben.
Attribute in den Relationen sind nach dem Pfad vom Wurzelelement aus in einer Punktnotation bezeichnet. Jede Relation hat ein `ID`-Feld als Schlüssel. Alle zu Element-Knoten gehörenden Relationen haben außerdem ein `parentID`-Feld als

Fremdschlüssel.
Elemente vom Typ `ANY` werden als nicht interpretierter String gespeichert. Die Struktur solcher Elemente kann komplexe Formen annehmen, die dem DB-System ohne eine zusätzliche Unterstützung von XML verborgen bleibt.

5.2.3 Shared Inlining

Mit dieser Technik sollen die Nachteile des Basic Inlining vermieden werden. Prinzipiell wird mit diesem Vorgehen versucht, Elemente die beim Basic Inlining in mehreren Relationen vorkommen, in separaten Relationen zu speichern und diese mit allen Relationen zu teilen.
Dazu werden zunächst die zu erzeugenden Relationen bestimmt. Es wird für jeden Knoten des DTD-Graphen eine Relation erzeugt, deren Eingangsgrad größer ist als Eins (= mehr als eine eingehende Kante besitzt). Elemente zu Knoten ohne eingehende Kanten bilden ebenfalls eigene Relationen, da sie von keinem anderen Knoten erreicht werden können. Weiterhin werden Relationen für Knoten generiert, die auf einen '*'-Knoten folgen analog zum Vorgehen des Basic Inlining. Alle anderen Elemente mit einem Eingangsgrad von Eins bilden die Attribute der Relationen.
Von den Elementen, die rekursiv zueinander in Beziehung stehen und nur eine eingehende Kante haben, wird eines in eine separate Relation gespeichert. Diese können gefunden werden, wenn nach streng verbundenen Elementen im DTD-Graph gesucht wird.
Die Anzahl der so zu generierenden Relationen ist im Vergleich zum Basic Inlining wesentlich geringer. Problematisch ist allerdings, daß ein Subelement eines Elements in die entprechende Relation aufgenommen wird, es aber in einem gegebenen XML-Dokument durchaus die Wurzel bilden könnte. Zur Kennzeichnung wird daher den Elementen in den Relationen ein `isRoot`-Feld vom Typ Boolean zugeordnet.
Vorteilhaft ist, daß bei Anfragen weniger Relationen bearbeitet werden müssen, als beim Basic Inlining. Die Wahrscheinlichkeit von Join-Operationen ist geringer, abgesehen von einem speziellen Fall. Das Basic Inlining reduziert die Anzahl der Joins, wenn die Anfrage bei einem Teilelementknoten gestartet wird. Aus diesem Grund stellen die Autoren einem dritten Ansatz vor, der die Vorteile des Basic und Shared Inlining kombiniert.

5.2.4 Hybrid Inlining

Das Vorgehen entspricht bei dieser Technik dem Shared Inlining mit der Ausnahme, daß auch Elemente in Relationen aufgenommen werden, die beim Shared Inlining in separate Relationen gespeichert wurden. Dazu gehören Elemente, die

einen Eingangsgrad größer als Eins besitzen, aber weder rekursiv sind, noch auf einen '*'-Knoten folgen. Mengenwertige und rekursive Elemente werden analog dem Vorgehen im Shared Inlining behandelt.
Die Anzahl der generierten Relationen kann so weiter reduziert werden.
Für das Datenmodell wurde angenommen, daß es unsortiert ist. Eine Ordnung läßt sich leicht durch die Speicherung der Position eines Elements in einem Dokument in einem entsprechenden Feld darstellen.

5.2.5 Vor- und Nachteile dieser Verfahren

Vorteile

Durch die Berücksichtigung der DTD von XML-Dokumenten spiegelt sich ihre Struktur in den generierten Relationen wider. Anfragen an einzelne Elemente können über mit Basic Inlining erzeugten Relationen ohne größeren Join-Aufwand ausgeführt werden. Die Techniken des Shared und Hybrid Inlining verringern die Anzahl der zu generierenden Relationen. Das hat zur Folge, daß bei Anfragen an diese Relationen weniger Join-Operationen notwendig sind und somit eine deutliche Verbesserung der Performance vorliegt.
Der Aufwand zur Erzeugung von Relationenschemata wird durch die verwendeten Transformationen gesenkt. Dabei ist es möglich, Dokumente, die entsprechend den Konventionen einer nichttransformierten DTD gebildet wurden, in die Schemata zu speichern, die aus der zugehörigen transformierten DTD generiert wurden.
Durch die Verwendung von Graphen können bewährte Algorithmen zur Zyklenerkennung bei der Schemaerstellung Anwendung finden, so daß Rekursionen gut bearbeitet werden können.

Nachteile

Gerade das Basic Inlining sticht durch seine Vorgehensweise der Generierung von Relationen zu jedem Element des DTD-Graphen negativ hervor. Das Schema wird so unübersichtlich groß. Anfragen ziehen die Vereinigung mehrerer Relationen zur Auswertung nach sich.
Die Schemata, die bei der Generierung entstehen, enthalten viele Informationen, die eher implizit in der DTD gegeben sind. Z. B. das Feld `isRoot` zur Kennung eines Elements als Wurzel eines XML-Dokuments ist in keiner DTD definiert. Diese Information und weitere, z. B. die Einführung eines Feldes zur Erkennung der Hierarchie von Elementen, sollten besser als Meta-Informationen in beschreibenden Relationen gespeichert werden.
Die Transformationen zur Vereinfachung der Element-Definition in der DTD birgt

in sich den Nachteil, daß eine Rückführung der gespeicherten Daten in ein XML-Dokument im Originalzustand nicht möglich ist. Es kann nur eine der transformierten DTD entprechende Form gebildet werden.
Der größte Nachteil besteht darin, daß sich dieser Ansatz auf das reine Relationenmodell beschränkt.

Kapitel 6

Abbildung von XML in OR-Datenbanken

In den folgenden Abschnitten wird ein Entwurf zur Speicherung von XML-Dokumenten in Objekt-relationalen Datenbanken vorgestellt. Voraussetzung dafür ist das Vorhandensein einer DTD zu einem XML-Dokument. Die in der DTD definierte Struktur eines XML-Dokuments wird für die Schemabildung herangezogen. Auf diese Weise ist eine natürliche Abbildung eines XML-Dokuments in eine Datenbank gewährleistet. Die Elemente eines Dokuments können so auf die Attribute der Relation abgebildet und die Inhalte der Elemente, ihre Werte, als Attributwerte zugeordnet werden.
Neben den Angaben über die Struktur der XML-Dokumente müssen aber zusätzliche Informationen über diese Datenquelle angegeben werden, wie die Häufigkeit des tatsächlichen Auftretens von Elementen in den Dokumenten oder bevorzugte Elemente bei Anfragen.
Das allgemeine Vorgehen besteht darin, soviele Elemente und Attribute eines XML-Dokuments wie möglich in einer Relation als Attribute abzubilden. In einigen Fällen müssen diese Bestandteile in separate Relationen ausgelagert werden. Dazu müssen dann Beziehungen modelliert werden, um den Zusammenhang zwischen diesen Relationen herzustellen, so daß es möglich wird, ein XML-Objekt wieder herzustellen.

6.1 Abbildung von einfachen Elementen

In einer DTD werden das Hauptelement und seine Subelemente definiert. Besitzen die Subelemente keine weitere Struktur, so werden sie im folgenden als einfache Elemente bezeichnet.
Ein einfaches Element wird durch einen Bezeichner und einen Typ, z. B. `CDATA`, `PCDATA` oder auch `EMPTY`, definiert. Solche Elemente werden unter dem Namen

des Elements als Attribute vom Typ `String` in einer Relation gespeichert.
Das folgende Beispiel zeigt die Abbildung einer DTD mit einfachen XML-Elementen:

DTD:

```
<!ELEMENT hotel (name, address, gastronomy)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT gastronomy (#PCDATA)>
```

Ein Dokument kann dann diesen Inhalt haben:

XML:

```
<hotel>
<name> Hotel Sonnenschein </name>
<address> Rostock-Warnemuende< /address>
<gastronomy> Restaurant </gastronomy>
</hotel>
```

Dieses Beispiel sieht in einer Datenbank so aus:

Relation `hotel`

OID	name	address	gastronomy
1	Hotel Sonnenschein	Rostock-Warnemuende	Restaurant
2	...	...	...

Das umschließende Element `hotel` wird zum Relationennamen. Alle anderen Elemente bilden die Attribute der Relation. Zusätzlich wird ein künstlicher Schlüssel `OID` eingeführt.

6.2 Abbildung von Element-Blöcken

Elemente in XML-Dokumenten können in verschiedenen ‘Anordnungen‘ auftreten. Sie können als Folgen in Erscheinung treten. Das entspricht dem Beispiel aus dem letzten Abschnitt. Die Folge ist dort `hotel (name, adress, gastronomy)`. Es sind weiter Gruppierungen, Quantifizierungen und Alternativen möglich. Alle diese Varianten können auch gemischt vorkommen. Die einfachen Möglichkeiten von Blöcken werden im folgenden betrachtet.

6.2.1 Gruppierung von Elementen und Alternativen

Durch Klammersetzung können Elemente gruppiert werden. Damit wird eine Reihenfolge der Elemente festgelegt. Elemente können auch alternativ auftreten. Es ist also möglich, daß einem Element innerhalb eines Dokuments eine Reihe verschiedener Elemente folgen können.
Dazu ein Beispiel:

DTD:

```
<!ELEMENT hotel (name, (address | (city, street)),
                gastronomy)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT gastronomy (#PCDATA)>
```

Die Elemente `city` und `street` sind durch ein Klammerpaar zusammengefaßt, sie treten immer zusammen auf.
Außerdem kann entweder das Element `address` oder das Elementpaar `(city, street)` in einem Dokument auftreten. Dies wird im Beispiel-Dokument dargestellt:

XML:

```
<hotel>
<name> Hotel Sonnenschein </name>
<address> Rostock-Warnemuende </address>
<gastronomy> Restaurant </gastronomy>
</hotel>
<hotel>
<name> Hotel Neptun </name>
<city> Warnemuende </city>
<street> Seestrasse 12 </street>
<gastronomy> Bar </gastronomy>
</hotel>
```

Nach der Methode im ersten Abschnitt werden alle Elemente in einer Relation abgebildet. Sie hätte das folgende Aussehen:

Relation `hotel`

OID	name	address	city	...	street	gastronomy
1	Hotel Sonnenschein	Rostock-Warnemuende	NULL	...	NULL	Restaurant
2	Hotel Neptun	NULL	Warnemuende	...	Seestrasse 12	Bar

6.2.2 Vor- und Nachteile dieses Vorgehens

Die Abbildung in einer Relation führt zur Entstehung von `NULL`-Werten. Treten sehr viele Alternativen auf, wird der Anteil an `NULL`-Werten steigen. Vorteilhaft ist allerdings, das alle Attribute in einer Relation vorhanden sind. Dies vereinfacht Anfragen an die Daten.

6.2.3 Speicherung in separaten Relationen als Alternative

Es kann auch eine andere Art der Abbildung gewählt werden, indem Attribute, die `NULL`-Werte enthalten, in eigene Relationen ausgelagert werden, zu sehen im nächsten Beispiel:

Relation `hotel`

OID	name	gastronomy
1	Hotel Sonnenschein	Restaurant
2	Hotel Neptun	Bar

Relation `hotel.address`

F_OID	address
1	Rostock-Warnemuende

Relation `hotel.city.street`

F_OID	city	street
2	Warnemuende	Seestrasse 12

Die `OID` aus der Relation `hotel` ist ein `primary key`, auf den sich die Attribute `F_OID` aus `hotel.address` und `hotel.city.street` als `foreign key` beziehen.

Das XML-Dokument ist auf 3 Relationen abgebildet worden, die nun keine `NULL`-Werte mehr enthalten.
Eine Anfrage nach dem Attribut `address` wird nun nicht mehr ganz einfach, da es nicht mehr in der Relation `hotel` enthalten ist. Hier muß der 'Umweg' über die System-Tabellen des DBMS gegangen werden. Sie geben Auskunft über die Relationen- und Attributnamen der Datenbank, sowie ihre Beziehungen zueinander. Erst nachdem die Relationen bestimmt wurden, die einen Fremdschlüssel enthalten, der auf `hotel` verweist, ist eine Anfrage nach allen Attributen möglich.
Als weitere Alternative dazu könnte man in die Hauptrelation eine Spalte mit einem Verweis auf die Alternative und die entprechende Relation aufnehmen. Die Relation `hotel` sieht dann so aus:

Relation `hotel`

OID	name	in_relation	gastronomy
1	Hotel Sonnenschein	hotel.address	Restaurant
2	Hotel Neptun	hotel.city.street	Bar

Die neue Spalte enthält den Namen der Relation, in der die Alternative des Objekts gespeichert ist. Die Verbindung zwischen einem Objekt und der zugehörigen Alternative kann durch einen Join, z. B. mit der Bedingung `hotel.OID` = `hotel.address.F_OID`, wiederhergestellt werden.
Nur im ungünstigen Fall, daß die Alternativen so definiert sind, daß keine von ihnen auftreten muß, kann die Spalte `in_relation` `NULL`-Werte enthalten.

6.2.4 Quantifizierung von Elementen

Elemente können in XML-Dokumenten 0 oder mehrmals hintereinander auftreten. Dies wird durch Quantifikatoren angegeben. Die verschiedenen Möglichkeiten sind in der folgenden Tabelle zusammengefaßt:

Quantifizierer	Bedeutung
?	0 oder einmal
$*$	0 oder mehrfach
+	mindestens einmal oder mehrfach

Das mehrfache Auftreten bei Quantifizierung mit $*$ und `+` läßt sich in Objektrelationalen Datenbanken durch die Typkonstruktoren `set of` oder besser `list of` darstellen. Eine Liste garantiert hierbei die Reihenfolge der Werte, wie man sie im Dokument vorfindet.
Davon abgesehen, ist das Vorgehen identisch mit der Abbildung von Gruppierungen bzw. Alternativen. Bei einer Abbildung treten auch ähnliche Probleme, wie

bei Gruppierung/Alternativen auf. Bei Quantifizierung mit `?` und $*$ können hier zusätzlich `NULL`-Werte bzw. die leere Menge {} in Relationen auftreten, wenn sie alle Elemente enthält. Das bekannte Beispiel einer DTD für Hotels wird zunächst um einen Quantor `+` für das Element `gastronomy` erweitert:

DTD:

```
<!ELEMENT hotel (name, (address | (city, street)),
                 gastronomy+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT gastronomy (#PCDATA)>
```

Das Element `gastronomy` kann nun mehrmals innerhalb des Elements `hotel` auftreten:

XML:

```
<hotel>
<name> Hotel Sonnenschein </name>
<address> Rostock-Warnemuende </address>
<gastronomy> Restaurant </gastronomy>
<gastronomy> Bar </gastronomy>
</hotel>
<hotel>
<name>Hotel Neptun </name>
<city> Warnemuende </city>
<street> Seestrasse 12 </street>
<gastronomy> Bar </gastronomy>
</hotel>
```

Das Element `gastronomy` kann nun entweder mit dem Konstruktor `set-of` oder `list-of` abgebildet werden. Für unser Beispiel nehmen wir hier einmal eine Menge. Die folgende Relation `hotel` enthält alle Elemente des Beispiels und damit auch `NULL`-Werte:

Relation `hotel`

OID	name	address	city	street	gastronomy
1	Hotel Sonnenschein	Rostock-Warnemuende	NULL	NULL	{Restaurant, Bar}
2	Hotel Neptun	NULL	Warnemuende	Seestrasse 12	Bar

Will man das vermeiden, speichert man diese Elemente separat in Relationen. Die Relationen zu dem Beispiel sehen dann wie folgt aus:

Relation `hotel`

OID	name	in_relation	gastronomy
1	Hotel Sonnenschein	hotel.address	{Restaurant, Bar}
2	Hotel Neptun	hotel.city.street	Bar

Relation `hotel.address`

F_OID	address
1	Rostock-Warnemuende

Relation `hotel.city.street`

F_OID	city	street
2	Warnemuende	Seestrasse 12

Das Prinzip der Schlüsselbildung entspricht dabei dem im vorherigen Abschnitt zur separaten Speicherung von Alternativen beschriebenen Vorgehen.

6.2.5 Komplexe Elemente

Einige Elemente können eine komplexe Struktur aufweisen. Objekt-relationale Datenbanken bieten mit Hilfe des Konstruktors `type-of` die Möglichkeit, diese Struktur zu definieren und als ein Attribut (= dessen Typ) darzustellen.
Das Beispiel `hotel` kann einen solchen komplexen Typ, z. B. für das Element `address`, enthalten. Die DTD für `hotel` ändert sich dadurch wie folgt:

```
<!ELEMENT hotel (name, address, gastronomy+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (ZIP, street, nbr, city)>
<!ELEMENT ZIP (#PCDATA)>
<!ELEMENT street (#PCDATA)>
```

```
<!ELEMENT nbr (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT gastronomy (#PCDATA)>
```

Das XML-Dokument kann nun folgende Struktur aufweisen:

```
<hotel>
   <name>Hotel Neptun</name>
   <address>
      <ZIP> 18052 </ZIP>
      <street> Seestrasse </street>
      <nbr> 12 </nbr>
      <city>Warnemuende</city>
   </address>
   <gastronomy>Bar</gastronomy>
</hotel>
```

Für das Element `address` ist nun ein Typ `address` bestehend aus den Komponenten `ZIP, street, nbr, city` in der Datenbank zu definieren. Dieser kann zur Definition des Relationenschemas herangezogen werden. Die Relation hat nun die fogende Form:

Relation `hotel`

OID	name	address				...
		ZIP	street	nbr	city	
1	Hotel Sonnenschein	18052	Seestrasse	12	Warnemuende	...

...	gastraonomy
...	{Restaurant, Bar}

6.2.6 Vor- und Nachteile dieser Methode

Der wesentliche Vorteil bei der Erzeugung mehrerer Tabellen mit Schlüssel und Fremdschlüssel liegt in der Vermeidung von `NULL`-Werten. Diese würden entstehen, wenn alternative, quantifizierte oder gruppierte Elemente in eine Tabelle aufgenommen würden. Außerdem führt eine Quantifizierung dazu, daß viele Informationen so mehrfach gespeichert werden müßten. Anfragen wären aber leichter zu formulieren, da alle Elemente in einer Tabelle vorhanden sind und so direkt angefragt werden können. Durch die Möglichkeit der Verwendung von Mengen- und Typkonstruktoren, die in ORDBMS enthalten sind, sind weniger Relationen zur Abbildung der Dokumentenstruktur in ein Schema nötig, als etwa mit relationalen DBMS. Diese verlangen für Mengen und komplexe Typen die Speicherung in separaten Relationen und die Definition einer Schlüsselbeziehung.

Negativ wirkt sich aus, daß bei der Aufteilung in mehrere Tabellen ein sehr großes Datenbankschema entstehen kann. Im Extremfall besteht jede Tabelle nur aus einem Attribut und einem Schlüssel, wenn alle Elemente alternativ auftreten. Anfragen an die Daten werden sehr aufwendig, da zusätzlich die Metadaten des DBMS herangezogen werden müssen.
Welcher Ansatz, mit oder ohne `NULL`-Werte, gewählt wird, hängt von der Zusammensetzung der Domäne der XML-Dokumente und dem Verwendungszweck der gespeicherten Daten ab.

6.3 Behandlung von Rekursionen

Der weitaus komplizierteste Fall bei der Abbildung stellt das Auftreten von Rekursionen dar. Ein Element kann so definiert sein, daß es sich selbst beliebig oft enthalten kann. Auch ist eine Rekursionskette möglich, bei der ein Element Subelemente mit verschiedener Struktur enthält und eines oder auch mehrere dieser Subelemente können das umgebende Element enthalten.
Vorschläge zur Abbildung von Rekursionen werden in folgenden erläutert.

Rekursion innerhalb eines Elements

Die DTD im folgenden Beispiel enthält eine Definition, bei der ein Element sich selbst beliebig oft enthalten kann:

```
<!ELEMENT person (name, alter, kind*)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT alter (#PCDATA)>
<!ELEMENT kind (person)>
```

Ein Beispiel-Dokument könnte dieses Aussehen haben:

```
<person>
  <name> Rolf </name>
  <alter> 44 </alter>
    <kind>
      <name> Peter </name>
      <alter> 22 </alter>
        <kind>
          <name> Paul </name>
          <alter> 2 </alter>
        </kind>
    </kind>
    <kind>
```

```
        <name> Petra </name>
        <alter> 8 </alter>
      </kind>
</person>
```

Nach der bis zu diesem Punkt vorgestellten Vorgehensweise würde das Element `kind` nun entweder in der Relation `person` oder separat gespeichert werden, da es durch den Quantor $*$ entweder mehrfach oder überhaupt nicht auftreten kann. Da das Element `kind` zudem komplex ist (`kind` ist eine `person`, die aus 3 Subelementen besteht), wird hier die Bildung eines Typs `personen` versucht, mit dem das Attribut `kind` definiert werden soll. Der Typ für `personen` enthält wiederum ein Element `kind`. Diese Schleife wird nun endlos. An dieser Stelle verhindert die Rekursion das bisherige Vorgehen.
Für diesen einfachen Fall der Rekursion sieht die Abbildung stattdessen so aus:

Relation `person`

OID	name	alter	kind
1	Rolf	44	{2, 4}
2	Peter	22	3
3	Paul	2	NULL
4	Petra	8	NULL

Wie das Beispiel zeigt, wurde das Element `kind` entprechend der Vorgehensweise bei quantifizierten Elementen mit in die Relation aufgenommen. Statt hier aber die Struktur des Elements, weitere Personen, zu speichern, enthält die Spalte Verweise auf das tatsächliche Personen-Objekt. Diese sind in der selben Relation unter den in der Menge angegeben Nummern (= `OID`) zu finden.
Alternativ kann das Element `kind` auch in einer separaten Relation gespeichert werden. Diese besitzt ein zur Relation `personen` identisches Schema. Die Spalten `kind` beider Relationen verweisen nun wechselseitg auf Elemente in der jeweils anderen Relation.
Das Schema besteht nun aus zwei Relationen:

Relation `person`

OID	name	alter	kind
1	Rolf	44	{1, 2}
2	Paul	2	NULL

Relation `person.kind`

OID	name	alter	kind
1	Peter	22	2
2	Petra	8	NULL

Die Werte `1, 2` aus der Spalte `kind` der Relation `personen` verweisen auf die `OID` der Relation `personen.kind`. Die in dieser Relation enthaltene Spalte `kind` enthält einen Fremdschlüsselverweis auf die `OID` in `personen`.

6.3.1 Rekursion zwischen zwei Elementen

Rekursion innerhalb eines XML-Dokuments kann auch zwischen Subelementen auftreten. Die bisher beschriebene Vorgehensweise führt in diesem Fall zu keinem Ergebnis. Anhand des folgenden Beispiels wird erläutert, welche Schritte zur Erzeugung eines Schemas führen, die Dokumente mit dieser Form der Rekursion abbilden können:

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

Die Elemente sowie deren Subelemente, die an der Rekursion beteiligt sind, sollten in separate Relationen gespeichert werden. Beide Relationen enthalten anstelle des rekursiven Elements einen Verweis auf das Element in der jeweils anderen Relation:

Relation `Elem.Elem2` für Element `Elem2`

Elem2.ID	Elem2	
	Elem3	Elem4
1	...	{Elem4.ID}
2	...	...

Relation `Elem.Elem4` für Element `Elem4`

Elem4.ID	Elem4	
	Elem5	Elem2
1	...	Elem2.ID
2	...	...

Die Attribute `Elem2.ID` und `Elem4.ID` sind Primärschlüssel in den jeweiligen Relationen.
In der Relation `Elem.Elem2` enthält in der Spalte `Elem4` einen Verweis auf den Primärschlüssel der Relation `Elem.Elem4`. Das Element `Elem4` kann innerhalb des Elements `Elem2` mehrfach auftreten, s. o. DTD. Die entprechende Spalte in der Relation `Elem.Elem2` wird daher mit dem `set-of`-Konstruktor definiert.
In der Relation `Elem.Elem4` ist das Attribut `Elem2` ein Verweis auf die Relation `Elem.Elem2`. Sie enthält also die `Elem2.ID` eines enthaltenen Elements.
Auf die rekursiven Elemente wird innerhalb der Hauptrelation verwiesen. Sie hat dieses Schema:

Relation `Elem`

OID	Elem1	Elem2	Elem4
1	...	Elem2.ID	Elem4.ID
...	...	...	...

6.3.2 Rekursion über mehrere Stufen

Eine Rekursion kann auch weniger direkt auftreten. Es ist durchaus möglich, daß ein Zyklus erst durch ein drittes oder viertes Element gebildet wird. Das folgende Beispiel einer DTD illustriert diesen Fall:

```
<!ELEMENT Elem (Elem1, Elem2, Elem4)>
<!ELEMENT Elem1 (#PCDATA)>
<!ELEMENT Elem2 (Elem3, Elem4*)>
<!ELEMENT Elem3 (#PCDATA)>
<!ELEMENT Elem4 (Elem5, Elem6?)>
<!ELEMENT Elem5 (#PCDATA)>
<!ELEMENT Elem6 (Elem7, Elem2?)>
<!ELEMENT Elem7 (#PCDATA)>
```

Das Vorgehen in diesem Fall unterscheidet sich nicht wesentlich von dem im letzten Abschnitt beschriebenen. Auch hier werden die für die Rekursion verantwortlichen Elemente in separate Relationen abgebildet. Für diese werden ebenfalls Primärschlüssel und Verweise auf diese definiert. Die Schemata der Relationen für die rekursiven Elemente sind:

Relation `Elem.Elem2` für Element `Elem2`

Elem2.ID	Elem2		
	Elem3	Elem4	
		Elem5	Elem6
1	...	...	Elem6.ID
2	...	...	...

Relation `Elem.Elem6` für Element `Elem6`

Elem6.ID	Elem6	
	Elem7	Elem2
1	...	Elem2.ID
2	...	...

Das Element `Elem4` kann in die Relation `Elem.Elem2` als Attribut aufgenommen werden. Wie im letzten Abschnitt beschrieben werden an den Stellen, an denen die Rekursion auftritt, Verweise auf den Primärschlüssel eines Elements eingefügt, das an dieser Stelle in einem XML-Dokument auftritt.

6.3.3 Vorteile und Nachteile

Vorteile

Anhand von separaten Relationen und der Modellierung von Schlüssel-Beziehungen können Rekursionen leicht in DB-Schemata modelliert werden. Das entstehende Schema eines Elements, das aufgrund von Rekursion ausgelagert wurde, entsteht auf die in den Abschnitten 6.1 bis 6.2.5 beschriebene Weise zur Abbildung von Elementen und den Sonderfällen Gruppierung, Alternativen und Quantifikation.

Nachteile

Durch Rekursion entsteht ein größeres DB-Schema, da auch mehr Relationen gebildet werden. Die Attribute, die auf Elemente der anderen Relationen verweisen, können `NULL`-Werte bzw. die leere Menge {} enthalten. Das ist der Fall, wenn die Rekursion endet oder auch gar nicht auftritt.

6.4 Abbildung von Attributen

Einem Element können Attribute zugeordnet werden. Diese treten innerhalb des `XML-tags` zusammmen mit ihren Werten auf. Attribute können wahlweise auftreten, gekennzeichnet durch `IMPLIED`, oder müssen vorhanden sein, gekennzeichnet durch `REQUIRED`. Attribute können mit verschiedenen Typen definiert werden. Die Möglichkeiten zur Abbildung von Attributen in Relationen stehen im Mittelpunkt dieses Abschnitts.

6.4.1 Abbildung auf ein Relationenattribut

Auch XML-Attribute können als Datenbank-Attribute dargestellt werden. Sie sollten zusammen mit dem Element, zu dem sie gehören, als ein Typ zusammengefaßt werden. So wird ihre Zusammengehörigkeit deutlich modelliert.
Es treten hier einige spezielle Fälle auf. Ist das XML-Attribut als `REQUIRED` definiert, muß es in der Datenbank als `NOT NULL` definiert sein.
Die Attribute eines Elements sind in einem XML-Dokument Bestandteil dieses Elements. Daher sollten sie in einer Relation direkt auf das Attribut, das dieses Element darstellt, folgen.

6.4.2 Abbildung der verschiedenen Attributtypen

Es stehen verschiedene Typen zur Definition von Attributen in einer DTD zur Verfügung. Die folgende Übersicht listet die Typen auf und gibt an, auf welchen Typ er in der Datenbank abgebildet werden soll:

XML-Attribut-Typ	**Typ in der Datenbank**
CDATA	STRING
NMTOKEN	STRING
NMTOKENS	STRING oder `set-of` STRING
ENTITY	STRING
ID	spezial (s. Abschnitt 6.4.3)
IDREFS	spezial
IDREFS	spezial
notation	STRING
name groups	STRING; evtl. SQL-`CHECK`

Attribute, die *name groups* mit einer bestimmten Wertemenge bilden, können durch die SQL-Anweisung `CHECK` dargestellt werden.
Die Abbildung von Referenzen in XML-Dokumenten, durch `ID-, IDREF`-Attribute modelliert, bedarf einer gesonderten Betrachtung. Diese folgt im nächsten Abschnitt.

6.4.3 Abbildung von `ID, IDREF`

Ein Attribut vom Typ `ID` identifiziert ein Objekt bzw. Element eindeutig innerhalb eines XML-Dokuments. Im Gegensatz zu einem Primärschlüssel einer Relation, der jedes Tupel eindeutig identifiziert, muß das `ID`-Attribut nicht für jedes Element, für das es definiert wurde, vorhanden sein. Es ist möglich, das `ID`-Attribut mit `IMPLIED` zu definieren, so daß es in einem Dokument nicht auftauchen muß. Zusätzlich wird die Abbildung von Attributen vom Typ `ID, IDREF` dadurch erschwert, daß sie nicht typisiert sind. Eine `IDREF`-Referenz eines Elements kann auf `ID`-Attribute verschiedener Elemente verweisen. Dies ist aber aus der Definition in der DTD nicht ersichtlich, sondern wird erst in einem konkreten Dokument ersichtlich.

Abbildung als `STRING`

Die einfachste, aber unerfreulichste Methode, mit solchen Referenzen umzugehen, ist es, keine besondere Behandlung für Referenzen einzuführen. In diesem Fall werden entprechende Attribute auch als Typ `STRING` gespeichert. Dadurch gehen dann aber leider auch die Informationen über die Referenzen verloren.

Nachbildung der Referenz

Die Referenz kann nicht durch eine Schlüssel-Fremdschlüssel-Beziehung dargestellt werden, da das `IDREF`-Attribut eines Elements auf verschiedene Elemente verweisen kann. Aus Datenbanksicht handelt es sich dabei um verschiedene Relationen-Attribute. Es lässt sich in einem DBMS kein Fremdschlüssel definieren, der auf verschiedene Primärschlüssel verweist.

Man kann auf andere Art versuchen, die Referenz nachzubilden. Ein Attribut `ID` wird mit Typ `STRING` gespeichert. Ein Attribut vom Typ `IDREF` wird als ein Attribut mit einer Referenz gespeichert. Sie enthält den Wert des `ID`-Attributs, das das Element refenrenziert und die Relation, in der das referenzierte Objekt gespeichert ist. Im Falle einer Menge von Referenzen, durch ein `IDREFS`-Attribut dargestellt, wird statt eines einzelnen Paares von `ID`-Wert und Relationenname eine Menge solcher Paare zu einem Element gespeichert.
Da nicht jedes Element über ein `ID`- oder `IDREF`-Attribut verfügen, sollten diese Attribute zusammen mit dem Element, zu dem sie gehören, zu einem Typ zusammengefaßt werden.
Diese Methode erfährt keine Unterstützung durch ein DBMS bzgl. der Funktionalität von Schlüsseln.
Zur Zeit der Bildung des Schemas liegen keine Informationen darüber vor, zu welchen Attributen in welchen Relationen Referenzen bestehen werden. Daher muß bei der Speicherung von XML-Dokumenten innerhalb des dafür zuständigen Programms Vorsorge getroffen werden, daß diese Informationen korrekt eingetragen werden.

6.4.4 Unterscheidung von Elementen und Attributen

Durch die beschriebene Abbildung werden Elemente und Attribute der XML-Dokumente als Datenbank-Attribute abgebildet. Aus der Datenbank ist nicht mehr ersichtlich, ob ein Datenbank-Attribut ein XML-Element oder XML-Attribut war. Die Unterscheidung erfordert das Anlegen eigener 'Metadaten'. In diesen Metadaten sind die Informationen über die Abbildung der DTD in die Datenbank zu speichern. Dazu gehören die Namen der Relationen und ihre XML-Elementnamen, die Attribute der Relationen und der entprechende Name des XML-Elements bzw. XML-Attributs, der Typ des DB-Attributs etc. Die Tabelle könnte folgendes Aussehen haben:

Relation `DTD_TO_DB`

DTD_TYP	XML_NAME	DB_NAME	DB_TYP
ELEMENT	hotel	hotel	RELATION
ELEMENT	name	name	ATTRIBUT
ATTRIBUT	para	cdata	ATTRIBUT
...	...	...	...

Zusätzlich empfiehlt es sich auch den Typ von XML-Attributen in diese Informationen aufzunehmen. Dies erleichtert die Rekonstruktion eines Dokuments aus den Relationen.

6.5 Abbildung von Entities

Es gibt verschiedene Arten von Entities. Built-in Entities, die es erlauben, im Text Sonderzeichen zu verwenden, die in XML reserviert sind, sollten vor dem Abspeichern aufgelöst werden.
Dazu gehören:

- < für <
- > für >
- & für &
- ' für ’
- " für “

Interne Text-Entities sind in den Text eines Elementes eingebettet. Beispielsweise wird ein oft verwendeter Text abgekürzt:

```
<!ENTITY XML "eXtensible Markup Language">
```

In einem XML-Dokument wird es so verwendet:

```
...
<text>Die &XML enthaelt Entities.</text>
...
```

Der Text sieht folgendermaßen aus:

```
Die eXtensible Markup Language enthaelt Entities.
```

Auf die gleiche Weise werden built-in Entities verwendet. Alle Arten von Entities sollten vor der Speicherung des Elements, das ein Entity enthält, aufgelöst werden.
Dieser String wird in die Datenbank im Attribut `text` gespeichert. Es wird dabei in Kauf genommen, daß die Information, daß es sich um ein Entity handelte, verlorengeht.
Will man diesen Informationsverlust vermeiden, bleibt nur, dieses Element in einen Typ XML zu speichern. Im folgenden Abschnitt werden Fälle vorgestellt, die im Originalzustand in einen zu konstruierenden Typ XML zu speichern sind.

6.6 Elemente als Typ XML speichern

Es können Elemente auftreten, die aufgrund ihres Inhalts schwierig oder gar nicht auf einen Datenbank-Typ abgebildet werden können. Das ist nicht aus der DTD des Dokuments ersichtlich. Solche Spezialfälle sollten in einer Datenbank als nicht intepretierter String gespeichert werden. Dieser String entspricht der XML-Syntax, wie sie in einem XML-Dokument auftritt. Dazu ist zunächst ein neuer Attribut-Typ `XML` mit einem Typkonstruktor zu erstellen. Es können weiterhin neue Operationen definiert werden, die in SQL-Anfragen verwendet werden können.
Im folgenden werden diese Fälle näher vorgestellt.

6.6.1 Mixed content

Schwierig ist die Abbildung von Elementen mit `mixed content`. Diese Elemente können mitten im Text eingebettete Elemente enthalten:

```
<text> Dieser Text kann ein <embElem> eingebettetes
       Element </embElem> enthalten </text>
```

Solche Konstrukte speichert man besser als XML-Typ.

6.6.2 Berücksichtigen der Häufigkeit von Elementen

Weitere wichtige Informationen zur Entscheidung der Abbildung direkt als XML-Typ oder in Datenbanktypen müssen vom Autor der XML-Quellen und dem Datenbankdesigner zusammengetragen werden. Dazu gehören:

- das absolute Auftreten des Datenelements
- das relative Auftreten des Datenelements
- die absoluten Referenzierungen in Transaktionen
- die relativen Referenzierungen in Transaktionen

Selten auftretende Elemente sollten im Original gespeichert werden, da eine Erzeugung einer Tabelle für sie keinen Vorteil bringt. Elemente, nach denen häufig angefragt wird, sollten in Tabellenattribute abgebildet werden, um die Mächtigkeit von DB-Anfragesprachen nutzen zu können. Mit diesen Daten können Grenzwerte bestimmt werden, ab welchem Level z. B. ein Element häufig genug auftritt, um nicht als XML-Typ gespeichert zu werden.

6.6.3 Unstrukturierter Text

Der folgende Text enthält mehrere Informationen in einem Text:

XML:

```
...
<service>Fitness-Bereich auf 1500 Quadratmeter</service>
...
```

In der Datenbank ist dieser Teil dann so enthalten:

Tabelle:

...	service	...
...	<service> Fitness-Bereich auf 1500 Quadratmeter </service>	...

Weiterhin empfielt sich die Speicherung von Steueranweisungen und Elementen mit Formatierungsangaben als XML-String, für den Fall, daß eine Rekonstruktion des Original-Dokumentes geplant ist.

6.7 Schematischer Algorithmus

In den vorangegangenen Abschnitten wurden die Abbildungsmöglichkeiten für die in einer DTD auftretenden Elemente und Attribute sowie besonderer Fälle diskutiert. Der folgende Abschnitt beschreibt das Vorgehen der Erzeugung des DB-Schemas aus einer DTD. Hierbei wird im Besonderen darauf eingegangen, an welchen Stellen zu entscheiden ist, ob für ein Element oder eine Elementgruppe

eine separate Relation gebildet werden soll bzw. wie Rekursionen erkannt werden können.
Das folgende abstrakte Beispiel einer DTD wird im weiteren Verlauf für die Erläuterung der Grundzüge des Algorithmus dienen:

DTD-Beispiel:

```
<!ELEMENT E (E1, E2*, E3)>
<!ELEMENT E1 (#PCDATA)>
<!ELEMENT E2 (E3, E4)>
<!ELEMENT E3 (E2? | E5)>
<!ELEMENT E4 (#PCDATA)>
<!ATTLIST E4 A1 #CDATA
             A2 NMTOKEN>
<!ELEMENT E5 (#PCDATA)>
```

6.7.1 Bildung eines DTD-Graphen

Zunächst ist aus der DTD ein Graph zu bilden. Die Knoten des Graphen stellen die Elemente bzw. Attribute der DTD dar. Weiterhin werden alle Symbole für Quantoren und Alternativen als Knoten in den Graph aufgenommen. Diese stehen vor den betroffenen Elementen. Die Kanten bilden den Übergang zwischen den Knoten. Dies verdeutlicht, welche Elemente bzw. Attribute einem Element in einem zu der DTD konformen XML-Dokument folgen können. So verweisen z. B. alle Kanten eines '|'-Knoten (Alternativen) auf die Elemente, die an dieser Stelle alternativ auftreten können. In den Blättern des Graphen ist der Typ des Elements bzw. Attributs gespeichert. Der DTD-Graph zu dem oben gezeigten Beispiel ist in Abbildung 6.1 dargestellt.
Rekursionen sind innerhalb des Graphen durch `Backpointer`, zurücklaufende Kanten, gekennzeichnet. Zur Unterscheidung von mixed content, Subelementen und Attributen zu einem Element wird ein Knoten `ATTLIST` eingeführt, dem alle Attribute des Elements und deren Typ folgen.

6.7.2 Bildung des Relationen-Schemas

Der entstandene Graph wird mit der Deep First Strategie durchlaufen. Dabei wird die Anzahl der zu bildenden Relationen und die Struktur bzw. Typen der Relationenattribute festgelegt. Das Wurzelelement des Graphen bildet den Relationennamen der Hauptrelation.

Bestimmung der Anzahl der Relationen

Das Ziel bei der Bildung des Relationen-Schemas ist, die Anzahl der entstehenden Relationen möglichst gering zu halten. Dazu sollen soviele Elemente/Attribute

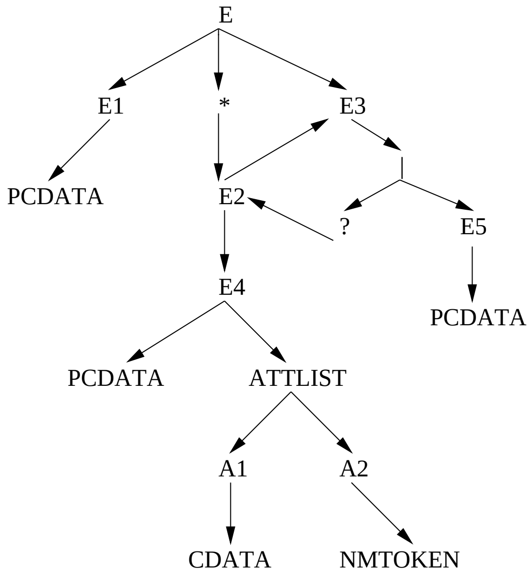

Abbildung 6.1: Element-Graph für Element `editor`

aus der DTD wie möglich in einer Relation aufgenommen werden. Der Algorithmus muß demnach die Elemente bestimmen, die in separate Relationen gespeichert werden sollen.
In den vorangegangenen Abschnitten wurde erläutert, welche Fälle zur Bildung separater Relationen führen. An dieser Stelle werden diese Fälle daher nur noch einmal aufgezählt. Separate Relationen sollen gebildet werden, wenn Alternativen, mengenwertige Elemente oder Rekursion zwischen Elementen auftreten.
Dies betrifft die auf '|'-, '*'-, und '?'-Knoten folgenden Teilgraphen, wenn eine Aufspaltung gewünscht wird, um `NULL`-Werte bzw. leere Mengen zu vermeiden. Nimmt man diese in Kauf, kann auf eine Auslagerung an diesen Stellen verzichtet werden.
Die Rekursion kann ähnlich dem in Abschnitt 5.2.2 beschriebenen Vorgehen erkannt werden. Ein `Backpointer` weist auf Rekursion hin. Die entsprechenden Elemente und ihre Teilgraphen bilden jeweils separate Relationen. Für jede Relation wird ein Primärschlüssel definiert. An der Stelle, an der das rekursive Element auftritt, wird ein Schlüsselattribut eingefügt. Führt der Backpointer zurück zum Wurzelknoten des Graphen, ist eine Aufspaltung nicht nötig. Das Schlüsselattribut kann hier innerhalb der Relation als Fremdschlüssel auf den Primärschlüssel derselben Relation definiert werden.

In unserem Beispiel werden für die Elemente `E2` und `E3` separate Relationen gebildet, da sie sich rekursiv verhalten.

Bestimmung der Definition der Attribute

Bei der Definition des Relationen-Schemas sollen auch die von ORDBMS gebotenen Möglichkeiten berücksichtigt werden. Dazu gehören Typ- und Mengenkonstruktoren.
Sind die Mengenkonstruktoren `set-of` bzw. `list-of` in dem ORDBMS, für das die Abbildung vorgenommen wird, vorhanden, werden diese auf die Teilgraphen angewendet, die auf '*'- oder '+'-Knoten folgen.
Der Typkonstruktor `type-of` findet Verwendung, wenn ein Element eine komplexe Struktur besitzt. D.h., wenn ein Element mehrere nachfolgende Element-Knoten hat, werden diese in einem Typ zusammengefaßt. Auch Attribute eines Elements werden mit diesem zusammen durch eine Typdefinition dargestellt.
Ein Element-Knoten, dessen Nachfolger ein Blattknoten mit dem Typ dieses Elements ist, wird als Attribut mit dem Typ String abgebildet.

Die folgenden Relationen wurden aus dem DTD-Beispiel vom Anfang dieses Abschnitts gebildet:

```
E (OID : Integer, E1 : String, E.E2.OID : set-of(Integer),
   E.E3.OID : Integer)

E.E2 (OID : Integer, E.E3.OID : Integer,
      E4 : type-of (E4 : String,
                    A1 : String,
                    A2 : set-of(String))
     )

E.E3 (OID : Integer, E.E2.OID : Integer, E5 : String)
```

Da die Alternativen, aus denen Element `E3` bestehen kann in diesem Beispiel nicht in separate Relationen gespeichert wurden, entstandenen nur 3 Relationen. Die Speicherung der Alternativen in separaten Relationen hätte die Gesamtanzahl auf 5 Relationen erhöht.

6.8 Zusammenfassende Übersicht

Für eine bessere Übersicht soll an dieser Stelle die Abbildung von der DTD in ein DB-Schema anhand einer Tabelle dargestellt werden. In ihr sind die Elemente aus der DTD und ihre Gegenstücke in einer Datenbank inklusive aller Alternativen aufgelistet:

tabellarische Übersicht der Abbildung

Typ in der DTD	**Abbildung in ORDB**	**Bemerkungen**
einfaches Element	Attribut vom Typ String	keine
komplexe Elemente	neuer Typ mit `type-of`	
Gruppierung	`type-of` bei Elementgruppen	
Alternativen	alle in einer Relation	viele `NULL`-Werte
	jede in separater Tabelle	Verknüpfung selbst definieren
mengenwertige Elemente: +	Attribut mit `set-of`	
$*$	Attribut mit `set-of`	leere Menge möglich
alternativ für $*$	separate Relation	vermeidet leere Mengen
mögliches Auftreten ?	als Spalte aufnehmen	`NULL`-Werte möglich
Rekursion:		
Element enhält sich selbst	in eine Relation	Verweis auf `OID` des enthaltenen Elements
2-stufige Rekursion	rekursive Elemente in separate Relationen	kann `NULL`-Werte enthalten
mehrstufige Rekursion	wie bei 2-stufiger Rekursion	siehe 2-stufige Rekursion
Attribut	Attribut vom Typ String	
ID, IDREF	als Attibut speichern	Referenzinfo verloren
oder	in IDREF-Attribut Tupel mit Tupel {ID-Wert, Relation}	in INSERT-Programm lösen
Entities	Referenzen auflösen	Verlust bei Rekonstruktion
mixed content	in Typ XML speichern	
seltene Elemente bzgl. Auftreten	in Typ XML speichern	
seltene Elemente bzgl. Anfragen	in Typ XML speichern	
unstrukturierter Text	in Typ XML speichern	
Steueranweisungen	in Typ XML speichern	
Formatierungen	in Typ XML speichern	

Kapitel 7

Realisierung für das GETESS-Projekt

Im Rahmen der Umsetzung der Speicherung von XML-Dokumenten im GETESS-Projekt sind einige Besonderheiten zu beachten. Anfragen, die von der Suchmaschine in GETESS bearbeitet werden, produzieren sogenannte *Abstracts* in Form von XML-Dokumenten. Diese Abstracts werden in einer Objekt-relationalen Datenbank gespeichert, um für künftige Anfragen zur Verfügung zu stehen.
Die Datenbank-Schemata, die diese Abstracts aufnehmen, und eine DTD zur Beschreibung der Abstracts werden aus einer Ontologie gewonnen. Die Ontologie bildet ein semantisches Repräsentationsmodell eines Anwendungsgebietes. Diese ermöglicht eine inhaltliche Einordnung der Suchanfragen für die Suchmaschine und gibt Kontextinformationen.
Als Datenbanksysteme finden die Objekt-relationalen Systeme DB2 und Informix Verwendung.
Im folgenden werden die Bildung der Datenbank-Schemata und die durch ein Programm zur Speicherung der Abstracts zu lösenden Aufgaben vorgestellt.

7.1 Entstehung des DB-Schemas aus der Ontologie

Der in Kapitel 6 vorgestellte Entwurf zur Abbildung von XML-Dokumenten in Datenbanken basiert auf der Schema-Bildung auf Grundlage einer DTD. Im Gegensatz dazu wird das Datenbank-Schema in GETESS aus der Ontologie gebildet.

7.1.1 Informationen aus der Ontologie

Aus der Ontologie sind verschiedene Informationen für die Erstellung solcher Abstractdatenbanken ablesbar.

Dazu gehören:

- Relationennamen
- Attributnamen
- Typen
- Bereiche
- Aufzählungstypen
- Fremdschlüsselbeziehungen
- Is-a-Hierarchien

Weitere benötigte Informationen für den Datenbankentwurf sind:

- `NOT NULL`
- `PRIMARY KEY`
- `FOREIGN KEY`
- `UNIQUE CONSTRAINT`

Diese sollen in spätere Versionen der Ontologie integriert werden.

7.1.2 Algorithmus zur Umsetzung

Der Algorithmus zur Bildung eines Datenbankschemas aus der Ontologie umfaßt die folgenden Schritte:

- Parsen der Ontologie
- Bildung eines Graphen aus Ontologieinformationen
- Manipulation des Graphen durch spezielle Umwandlungsschritte
- Bildung von `CREATE TABLE`-Anweisungen
- Ausführen der `CREATE TABLE`-Anweisungen
- Speichern der Metainformationen

Die Ontologie soll alle Begriffe, die in einer Domäne auftreten, ordnen und systematisieren. Sie stellt selbst noch keinen Datenbankentwurf dar. In der Ontologie sind weitere Informationen enthalten, die den Entwurf eines Schemas unterstützen.
Die Ontologie enthält Begriffe, die so allgemein sind, daß für sie keine konkreten Daten anfallen. Die Unterscheidung geschieht durch die Angabe von `role concrete` und `role abstract`. Ist `role abstract` gesetzt, braucht die entsprechende Klasse nicht in die Datenbank übersetzt zu werden.
Einige Teile der Ontologie dienen nur der Einordnung, die modellierten Informationen liegen außerhalb der Domäne. Für diese Teile entworfene Datenbanken würden leer bleiben. Diese Teile werden vom Datenbankentwurf ausgeklammert.
Andere Teile der Ontologie sind wiederum so speziell, daß es nicht sinvoll wäre diese strukturiert in der Datenbank zu speichern. Dabei handelt es sich um selten vorkommende bzw. angefragte Teile. Diese werden als Attribut mit dem Typ XML zusammengefaßt. Solche Teile lassen sich erst aus vorhandenen Daten erkennen. Diese konkreten Daten liegen zum Zeitpunkt der Bildung der Ontologie noch nicht vor. Daher erfolgt eine Kennzeichnung diese Teile. Dies geschieht in der Ontologie durch den Term:

```
def_h_border name
```

Alle auf `name` folgende Knoten des Ontologiegraphen werden zu solch einem Typ zusammengefaßt.

7.1.3 Metainformationen

Metainformationen werden erstellt, wenn die Datenbank aus den Ontologieinformationen gebildet wird. Sie sind erforderlich, da die Ontologie nicht 1:1 abgebildet werden kann, wie im letzten Abschnitt erklärt wurde. Sie umfassen die folgenden Informationen:

- die Ontologieinformationen
- die Informationen über die erzeugten Datenbanken
- die Abbildung zwischen diesen Formaten
- die Häufigkeit des Auftretens von Tupeln und Attributen

Die Häufigkeiten werden zyklisch in den Metainformationen ergänzt. Dies geschieht, wenn neue Tupel bzw. Attribute in eine Relation eingefügt oder aus ihr entfernt werden.

Verwendung der Metainformationen

Die Metainformationen erlauben es, Anfragen über die Ontologie zu stellen und die Abbildung der Ontologieinformationen auf die Datenbanken nachzuvollziehen. Solche Anfragen betreffen z. B. die Speicherung der Abstracts. Die dabei auftretenden Anfragen betreffen beispielsweise die Art der Abbildung eines Attributs (direkt oder innerhalb einer XML-Struktur), die Ermittlung der Relation zu einem Attribut oder auch den Zusammenhang zwischen zwei Attributen.

Schema der Metainformationen

Die Metainformationen umfassen zwei Relationen. In einer werden alle Informationen zu den gebildeten Relationen und in der anderen die zugehörigen Attributinformationen gespeichert. Diese Relationen haben den folgenden Aufbau:

Metainfo_Relationen:

Class_name	Print_name	Vaterknoten	Art_Abb	Relation_name	Anzahl_Tupel

Metainfo_Attribute:

Slot_name	Slot/Multislot	Print_name	Attribut_name	Relation

Referenz_auf	Typ	Länge	Anzahl_Tupel

Beispiel zur Umsetzung

Das folgende Beispiel zeigt einen Ausschnitt aus der Klasse Unterkunft der Ontologie über Tourismusinformationen:

```
(defclass Unterkunft 'Wo kann ich am Reiseziel unterkommen?'
(is-a Dienstleistungsbetrieb)
(role concrete)
(multislot hat_Saisonabhaengigkeit
(allowed-classes Qualitatives_Zeitkonzept)
(type INSTANCE)
(create-accessor read-write))
(slot bietet_aktivitaet
(allowed-classes Aktion)
(type INSTANCE)
(create-accessor read-write))
.....
```

```
(slot Anzahl_Betten
(type INTEGER)
(create-accessor read-write))
.....)
```

In den Metainformationen wird die Abbildung in die Datenbank für diese Informationen in dieser Form festgehalten:

Metainfo_Relationen:

Class_name	Print_name	Vaterknoten	Art_Abb
Unterkunft	Unterkunft	Dienstleistungsbetrieb	D

Relation_name	Anzahl_Tupel
Unterkunft	0

Metainfo_Attribute:

Slot_name	Slot/Multislot	Print_name	Attribut_name
hat_Saisonabhaengigkeit	M	Saisonabhaengigkeit	hat_Saisonabhaengigkeit
bietet_aktivitaet	S	aktivitaet	bietet_aktivitaet
Anzahl_Betten	S	Anzahl_Betten	Anzahl_Betten

Relation	Referenz_auf	Typ	Länge	Anzahl_Tupel
Unterkunft	Qualitatives_Zeitkonzept	INTEGER		0
Unterkunft	Aktion	INTEGER		0
Unterkunft		INTEGER		0

Die Schemabildung ist auf diese Weise in das GETESS-Projekt integriert. Für eine vollständige Speicherung von XML-Dokumenten ist somit ein Programm zu entwickeln, daß die Abstracts in die aus der Ontologie erzeugten Datenbank speichert.

7.2 Implementierung der Speicherung von Abstracts

Die Implementierung erfolgt mit dem Ziel, die Abstracts konform zur Ontologie in der Datenbank zu speichern. Das Programm muß folgende Anforderungen erfüllen:

- Parsen des Abstracts
- Suche der gefundenen Tags in den Metainformationen

- `INSERT`-Anweisungen aus den bestimmten Relationen/Attributen erstellen
- Verbindung zur Datenbank herstellen
- Ausführung der `INSERT`-Anweisungen
- Fehlerprotokollierung

Zum Parsen der Abstracts kann ein XML-Parser verwendet werden. Die gefundenen Elemente bzw. Attribute in den Abstracts sind dann in den Metainformationen zu suchen. Mit dieser Suche werden die zur Generierung von `INSERT`-Anweisungen notwendigen Informationen gewonnen. Dazu gehören z. B. der Relationenname, in den die Elemente zu speichern sind, der Name der Attribute in der Relation sowie ihr Typ. Letzterer ist notwendig, um Typkonvertierungen vorzunehmen, da die Daten in XML-Dokumenten einen String darstellen. Bei dieser Suche und auch bei der Ausführung der `INSERT`-Anweisungen können Fehler auftreten. Diese werden protokolliert und in Dateien zur weiteren Auswertung abgelegt.

Die Realisierung erfolgt mit JAVA/JDBC für DB2 von IBM. Die Beschreibung der dabei verwendeten Klassen und Methoden ist im Anhang enthalten.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Semistrukturierte Daten gewinnen in Applikationen immer mehr an Bedeutung. XML findet dabei Anwendung auf verschiedenen Gebieten, insbesondere beim Datenaustausch. Der Speicherung von XML-Dokumenten in Datenbanken kommt somit eine besondere Bedeutung zu. Es stellte sich heraus, daß relationale Datenbanken nur eine unbefriedigende Möglichkeit zur Darstellung von XML-Dokumenten bieten.
Objekt-relationale Datenbanken erlauben eine bessere Modellierung durch die einem Entwickler gebotenen spezielleren Möglichkeiten der Schemagenerierung. Diese lassen sich in Verbindung mit der Auswertung der DTD zu einer bestimmten Dokumentenmenge verwenden, um eine natürlichere Abbildung von XML-Dokumenten vornehmen zu können, als es mit relationalen Datenbanken möglich ist.
Die Abbildung von Elementen und Attributen in Datenbanken kann direkt auf Datenbankattribute erfolgen. Dennoch ist es nicht möglich alle Feinheiten von XML in Datenbanken abzubilden. Schwierig ist die Überführung von Referenzen. Sie lassen sich nur unzureichend abbilden.
Relationenschemata können klein gehalten werden, beinhalten dann allerdings viele `NULL`-Werte und sind somit schlecht gefüllt. Eine weitgehende Vermeidung von `NULL`-Werten führt zu einem großen Schema mit vielen Relationen. Dies hat eine Fragmentierung der Daten zur Folge. Für einen möglichst ausgeglichenen Datenbankentwurf empfielt es sich, einen hybriden Ansatz zur Speicherung von XML-Dokumenten zu wählen. Teile von Dokumenten, die zu einer schwachen Füllung von Relationen führen oder die selten angefragt werden, sind uninterpretiert als XML-String in einen Typ XML in der Datenbank zu speichern.

8.2 Ausblick

Im Rahmen des GETESS-Projekts erfolgte eine andere Schemabildung als in dem in Kapitel 6 vorgestellten Entwurf zur Speicherung von XML-Dokumenten. Daher ist eine Umsetzung und Evaluierung dieses Entwurfs hinsichtlich des Aufwands und der Performance zu empfehlen.
Die Entwicklung von XML steht erst am Anfang. Erweiterungen der Spezifikation von XML erfordern eine Anpassung des Entwurfs an die neuen Bedingungen.
Datenbanksysteme werden in Zukunft eine immer bessere Unterstützung von XML-Daten bieten. Diese Entwicklung ist zu verfolgen und die dann zur Verügung stehenden neuen Möglichkeiten können für die Weiterentwicklung des vorgestellten Entwurfs verwendet werden.

Anhang A

Beispiel zur Speicherung von XML-Dokumenten aus Abschnitt 5.2

Ausgehend von einer DTD werden der DTD-Graph, ein Element-Graph sowie die von den Techniken *Basic Inlining*, *Shared Inlining* und *Hybrid Inlining* gebildeten Relationenschemata gezeigt. Diese Techniken wurden in Kapitel 5 im Abschnitt 5.2 besprochen.

A.1 DTD für das Beispiel

Die folgende DTD beschreibt Buch-Dokumente:

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

Aus dieser DTD wird nach Anwendung der Reduzierungsschritte aus Abschnitt 5.2.1 ein DTD-Graph gebildet.

A.2 DTD- und Element-Graph

Der DTD-Graph ist in Abbildung A.1 dargestellt. Für das *Basic Inlining* wird

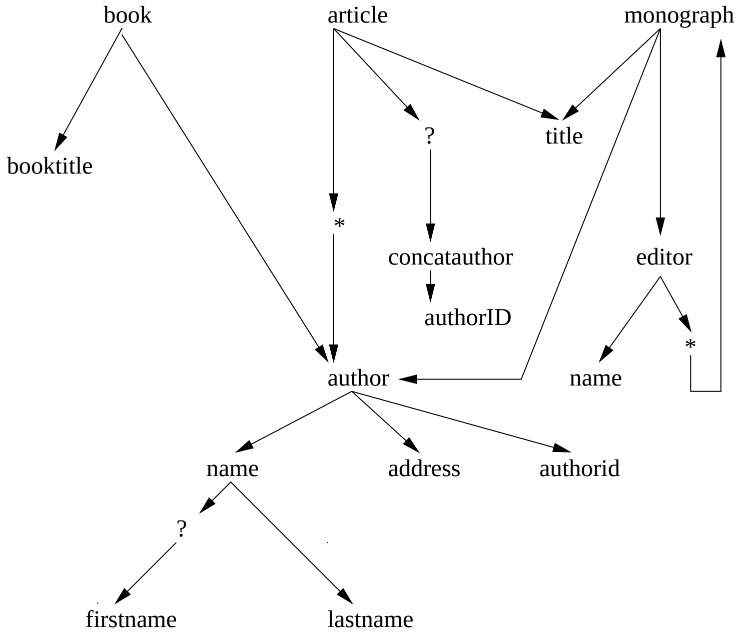

Abbildung A.1: DTD-Graph

aus dem DTD-Graphen für jedes in der DTD auftretende Element ein entsprechender Element-Graph gebildet. Der Element-Graph für das Element `editor` ist in Abbildung A.2 abgebildet.

A.3 Generierte Relationenschemata

Aus der Beispiel-DTD generiert das *Basic Inlining* das folgende Relationenschemata:

```
book (bookID : integer, book.booktitle : string,
      book.author.name.firstname : string,
      book.author.lastname : string,
      book.author.address : string,
      author.authorid:string)

booktitle (booktitleID : integer, booktitle : string)

article (articleID : integer,
         article.contactauthor.authorid : string,
         article.title : string)
```

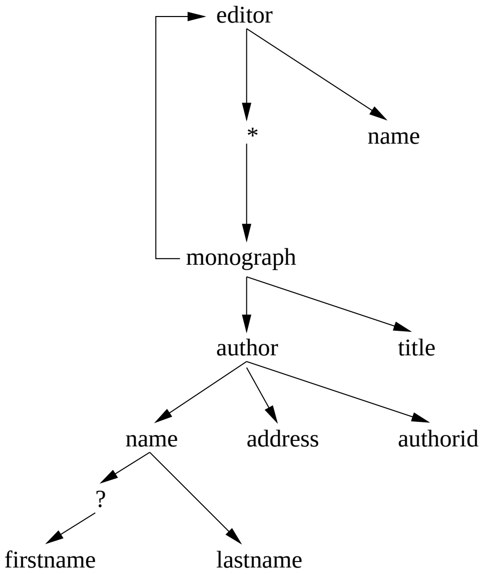

Abbildung A.2: Element-Graph für Element `editor`

```
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 : integer)

contactauthor (contactauthorID : integer,
               contactauthor.authorid : srting)

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.firstname : 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)
```

Es sind insgesamt 14 Relationen entstanden.
Beim *Shared Inlining* entstehen nur noch 5 Relationen:

```
book (bookID : integer, book.booktitle.isroot : boolean,
      book.booktitle : string)

article (articleID : integer,
         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 : string,
        author.authorid : string)
```

Das *Hybrid Inlining* reduziert die Relationenanzahl weiter, in diesem Fall entstanden 4 Relationen:

```
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 : integer,
         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.name.address.isroot : boolean,
        author.address : string,
        author.authorid : string)
```

Literaturverzeichnis

[GETESS] Die GETESS-Homepage
http://www.getess.de

[T-MD92] Jean Thierry-Mieg, Richard Durbin. ACeDB - A C. elegans Database: Syntactic definitions for the ACeDB data base manager, 1992

[GPQ+95] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Segev, J. Ullman, J. Widom. The tsimmis aproach to mediation: Data models and languages. *Proceedings of Second International Workshop on Next Generation Information Technologies and Systems*, pages 185-193, June 1995

[PGW95] Y. Papakonstantinou, H. Garcia-Molina, J. Widom. Objekt exchange across heterogenous information sources. *Proceedings of IEEE International Conference on Data Engeneering*, pages 251-260, March 1995.

[Abi97] S. Abiteboul. Querying Semi-Structured Data. *Proceedings of the International Conference on Database Theory*, 1997

[ACC+96] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, J. Simeon. Querying documents in object databases. Technical Report, INRIA, 1996.

[CCM96] V. Chritophides, S. Cluet, G. Moerkotte. Evaluating queries with generalized path expressions. In *SIGMOD'94*. ACM, 1994.

[Gol90] C. F. Goldfarg. *The SGML Handbook.* Clarendon Press, Oxford, 1990.

[BPS+98] Editors: T. Bray, J. Paoli, C. Sperberg-McQueen. Extensible markup language (XML) 1.0, February 1998. W3C Recommendation available at http://www.w3c.org/TR/1998/REC-xml-19980210.

[Bra98] N. Bradley. *The XML Companion.* Addison Wesley, Harlow, England, 1. Auflage, 1998.

[Sar98] C. M. Saracco. *Universal Database Management.* Morgan Kaufmann Publishers, Inc., San Francisco, 1998.

[SB99] M. Stonebraker, P. Brown, with D. Moore, *Objekt-relational DBMS Tracking the next great wave.* Morgan Kaufmann Publishers, Inc., San Francisco, 2. Auflage, 1999.

[Cha98] D. Chamberlin. *A complete guide to DB2 Universal Database.* Morgan Kaufmann Publishers, San Francisco, 1998.

[Pet98] D. Petkovic. *INFORMIX Universal Server.* Addison-Wesley, Bonn, 1998.

[FK99] D. Florescu, D. Kossmann. Storing and Querying XML Data using an RDBMS. *Data Engineering*, September 1999, Vol.22, No. 3.

[STH+99] J. Shanmugasundaram, Kristin Tufte, Gang He, Chung Zhang, David DeWitt, Jeffrey Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. *Proceedings of the 25th VLDB Conference*, Edingburgh, Scotland, 1999.

[DFS99] Deutsch, M. Fernandez, D. Sucio. Storing Semi-structured Data with STORED. *Proceedings of the ACM SIGMOD Conference*, Philadelphia, Pensylvania, May 1999.

Selbständigkeitserklärung

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

Rostock, 31.12.1999 Jens Timm

Thesen

1. Applikationen werden zunehmend mit semistrukturierten Daten umgehen müssen.
2. XML wird sich als Format für den Datenaustausch etablieren.
3. Objekt-relationale Datenbankmanagementsysteme sind dafür besser geeignet als relationale Datenbankmanagementsysteme.
4. Die Restriktionen von semistrukturierten Daten und von DBMS lassen eine Abbildung nur eingeschränkt zu.
5. Durch die Natur semistrukturierter Daten entstehen zwangsläufig `NULL`-Werte in einer Datenbank.
6. Die Nachbildung von Referenzen mit `ID/IDREF`-Attributen ist in Datenbanken mit den bisher zur Verfügung stehenden Mitteln nicht zufriedenstellend lösbar.
7. Zur Abbildung semistrukturierter Daten ist das Schlüssel-Fremdschlüssel-Konzept von Datenbanken nicht ausreichend.
8. Einige Bestandteile semistrukturierter Daten lassen sich nur schwierig oder gar nicht auf Relationenattribute abbilden.
9. Ein kleines Datenbankschema mit der geringsten möglichen Anzahl Relationen führt zu schwach gefüllten Relationen.
10. Die Vermeidung schwach gefüllter Relationen bedeutet ein großes Datenbankschema mit vielen Relationen und damit eine Streuung der Daten.
11. Ein hybrider Ansatz zur Speicherung von semistrukturierten Daten stellt den besten Kompromiß dar.