Diplomarbeit

zum Thema

Zur Implementierung von XQuery auf einem Objekt-relationalem XML–Speicher

Fakultät für Informatik der Universität Rostock

vorgelegt von:	Guido Rost
Matrikel–Nr.:	095201246
Diplomstudiengang:	Informatik
Bearbeitungszeit:	6 Monate

Gutachter:	Prof. Andreas Heuer
Zweitgutachter:	Prof. Dr. Clemens H. Cap
Betreuer:	Dr. Holger Meyer
Lehrstuhl:	Lehrstuhl für Datenbank- und Informationssysteme

Rostock, 22. Mai 2003

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

XML bietet die Möglichkeit, Daten zu strukturieren und diese in Form von XML–Dokumenten abzuspeichern. Die Struktur der Daten wird dabei mit Hilfe von XML–*Tags* beschrieben.
XML wird als Daten- bzw. Austauschformat im World Wide Web (*www*) immer populärer. Die Dokumente können entweder im Dateisystem selbst oder in Datenbanken gespeichert werden. Kommerzielle Datenbanksysteme bieten bereits Möglichkeiten zur Speicherung von XML–Dokumenten. So erweitert zum Beispiel IBM das Datenbank Management System (DBMS) *DB2 UDB* um den *XML–Extender* zur Verwaltung von XML–Daten.
Auf die XML–Daten in den Datenbanken oder Dateisystemen könnten dann zukünftige Web–Server über XML–Prozessoren zugreifen und diese z.B. mit Hilfe von HTML zur Präsentation grafisch aufbereitet werden.
Von Electronic Data Interchange (EDI) erhofft man sich "eine" wichtige *Business Application* für XML. Firmen könnten mit Hilfe von EDI Daten über ihre Produkte und Dienstleistungen im *WWW* veröffentlichen. Potentielle Kunden könnten diese Informationen automatisch vergleichen und auswerten lassen. Darüber hinaus könnten geschäftliche Partner interne, vertrauliche Daten über sichere Kommunikationskanäle zwischen ihren Systemen austauschen oder sogenannte *Search Robots* könnten XML–Daten

verschiedenster Quellen automatisch in eigene Anwendungen integrieren. Vorstellbare Szenarien wären in diesem Zusammenhang z.B. die Selektion bestimmter Aktienkurse von diversen Finanzseiten oder Sportdaten von verschiedenen Nachrichtenanbietern. Außerdem werden sich weitere Möglichkeiten ergeben, XML–Daten auf verschiedenste Art und Weise zu integrieren, zu transformieren oder zu aggregieren.
Hat sich XML erst einmal als Datenformat dauerhaft durchgesetzt, ist es ohne weiteres vorstellbar, daß viele Informationsquellen ihre Daten in XML–Dokumenten strukturieren und im Internet anbieten werden. Dann können Anwendungen wie EDI Realität werden.
Um Informationen aus einer Vielzahl von XML–Dokumenten nach bestimmten Kriterien zu durchsuchen und zu extrahieren, ist eine Anfragesprache für XML–Daten ein wichtiges Hilfsmittel. Diese soll strukturierte und inhaltsbezogene Anfragen ermöglichen, die als Ergebnis exakte Informationen liefern und keine 1000 Treffer, wie es der Internetnutzer derzeit von weniger brauchbaren Suchmaschinen gewohnt ist und die nur selten die gewünschten Informationen liefern. So entstanden gleich mehrere XML–Anfragesprachen wie z.B. Lorel , XML-QL, XQL, YATL, XML-GL oder XSL, die sich zum Teil stark unterscheiden. Den aktuellensten Stand der Forschung auf diesem Gebiet stellt *XQuery* dar. In der vorliegenden Diplomarbeit soll diese XML–Anfragesprache untersucht werden und eine prototypische Implementierung entwickelt werden.

Nach einer Motivation für diese Diplomarbeit wird im zweiten Kapitel eine vergleichende Übersicht über bereits existierenden XML–Anfragesprachen. Im folgenden Kapitel wird die Sprache *XQuery* in allen Details vorgestellt und mit den im zweiten Kapitel aufgeführten Anfragesprachen verglichen. Im vierten Kapitel werden die Konzepte für die Umsetzung der Sprache diskutiert. Deren Implementierung sowie Vorschläge zur Optimierung werden im fünften Kapitel erläutert. Die Arbeit schließt mit einer Zusammenfassung zur Implementierung und Vorschlägen zur Weiterführung dieser Arbeit.

Kapitel 2

Vergleich von XML–Anfragesprachen

XML–Anfragesprachen sind schon seit längerem Gegenstand der Forschung. So entstanden gleich mehrere Sprachen, die zum Teil verschiedene Konzepte verfolgten. Erste Vergleiche von XML–Anfragesprachen lieferten [1], [2], [3] und [4]. In diesem Kapitel sollen daraus drei Sprachen ausgewählt und verglichen werden.

2.1 Einführung zu Lorel, XML-QL und XQL

2.1.1 Lorel

Die **L**ightweight **O**bject **RE**pository **L**anguage (LOREL) ist eine datenbankorientierte Anfragesprache. Sie wurde an der Stanford University von S. Abiteoul, D. Quass, J. McHugh, J. Widom und J. Wiener entwickelt und implementiert. Der Prototyp ist unter *http://www-db.stanford.edu/lore* zu finden. Lorel ist eine benutzerfreundliche Sprache im SQL/OQL–Stil. Entwickelt wurde sie für den Umgang mit großen Datenbeständen. Sie soll Daten aus heterogenen Informationsquellen integrieren sowie in allgemeine Datenaustauschformate transformieren können.

2.1.2 XML-QL

Auch XML-QL ist eine datenbankorientierte Anfragesprache, die demzufolge auch die selben Aufgaben hat wie *Lorel*. XML-QL wurde in den AT&T Labs von Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy und Dan Suciu als Teil des Strudel–Projektes entwickelt. Der Prototyp ist unter *http://www.research.att.com/sw/tools/xmlql* zu finden. XML-QL besitzt eine explizite *CONSTRUCT*–Klausel, die zur Erstellung eines neuen Dokumentes aus dem Ergebnis der Anfrage dient.

2.1.3 XQL

XML **Q**uery **L**anguage ist eine dokumentorientierte Anfragesprache. Sie wurde von Jonathan Robie (Texcel Inc.), Joe Lapp (webMethods Inc.) und David Schach (Microsoft) entwickelt. Die Hauptaufgabe der Sprache ist die Suche innerhalb von sehr großen Dokumenten (Volltextsuche) und demzufolge muß XQL auch einen sehr effizienten Zugriffsmechanismus auf die Quellen besitzen. Da wir auf semistruktierte Daten zugreifen, soll XQL Volltext- und strukturierte Anfragen integrieren können. Mit Hilfe von XML–Behältern kann XQL auch Anfragen über mehrere Dokumente stellen, indem diese zu einem Dokument zusammengefügt werden. XQL ist keine vollständige Anfragesprache und bietet nur reduzierte Ausdrucksmöglichkeiten, kann aber in einer Vielzahl von Fällen eingesetzt werden und ist leicht zu erlernen. XQL ist eine natürliche Erweiterung zur *XSL Pattern Language*. Sie wurde erweitert um eine Boolesche Logik, Filter sowie eine Indizierung von Knotensammlungen. Genutzt wird XQL hauptsächlich zur Selektion und Filterung von Elementen und Text aus XML–Dokumenten. Sie ist eine sehr einfache und kompakt gehaltene Sprache und kann so als Teil einer URL eingesetzt werden.

2.2 Vergleich des syntaktischen Aufbaus

Der syntaktische Aufbau der Sprache soll hier vereinfacht in einer BNF-ähnlichen Notation dargestellt werden. Dabei sind Terminale, die die Schlüsselwörter der Sprache darstellen, in Hochkommas eingeschlossen. Geschweifte Klammern bedeuten eine Liste von 0 oder mehreren Elementen, getrennt durch Kommas. Eckige Klammern bedeuten das optionale Auftreten, und Alternativen werden durch ein “|“ getrennt.
Eine Query einer Anfragesprache kann grundsätzlich in drei Teile gegliedert werden:

1. **pattern-clause**
   Die *Muster–Klausel* überprüft Übereinstimmungen von verschachtelten Elementen und bindet Variablen an Elemente.

2. **filter-clause**
   Die *Filter–Klausel* testet die gebundenen Variablen.

3. **constructor-clause**
   Die *Erbauer–Klausel* spezifiziert das Ergebnis, als Terme von gebundenen Variablen.

Im folgenden soll die Syntax der verschiedenen Sprachen untersucht werden und die eben aufgezählten Klauseln identifiziert werden.

2.2.1 Lorel

```
Query := 'select' {select_expr}
         ['from'  {from_expr} ]
         ['where' {where_expr}]
```

Lorel–Ausdrücke werden in SQL–typischen *Select, From, Where (SFW)*–Blöcken geschrieben. Die "select"–Klausel ist die *Constructor*–Klausel, in ihr wird das Ergebnis spezifiziert. Die "from"–Klausel kann als *Muster*–Klausel

identifiziert werden. Und in der "where"–Klausel sind *Pattern* sowie *Filter* zu finden.
Die Sprache ist orthogonal, so kann jeder Ausdruck einer Klausel wieder eine Anfrage enthalten.

2.2.2 XML-QL

```
Query:= 'where' {Predicate}
        'construct' {'{'Query'}'}
```

In XML-QL befinden sich *Patterns* und *Filters* in der "where"–Klausel, die "contruct"–Klausel ist selbsterklärend die *Constructor*–Klausel.
Das Resultat der Query wird bei XML-QL in der "construct"–Klausel spezifiziert. Dort können neue XML–Elemente definiert werden, deren Inhalt durch eine geschachtelte Query konstruiert wird.

2.2.3 XQL

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

XQL unterstützt nur *Patterns* und *Filter*, keine *Constructors*. Das ist, wie sich in den folgenden Beispielen noch zeigen wird, ein großer Nachteil von XQL, da viele Queries allein dadurch nicht ausdrückbar sind .
Wie an der Syntax schon zu sehen ist, wird mit XQL durch einen XML–Baum navigiert, wobei Prädikate (*filter*) auf Elemente und Text entlang eines Pfades angewandt werden. Das Resultat einer XQL–Anfrage ist wieder ein XML Dokument.

2.3 Vergleich anhand von Beispielen

Im folgendem sollen die Sprachen anhand von drei wichtigen Operationen verglichen werden:

1. Selektion

   Die Selektion gehört zu den wichtigsten Operationen einer Anfragesprache, sie ermöglicht die Extrahierung bestimmter Daten aus XML–Dokumenten.

2. Join

   Diese Operation ist als wichtiger Bestandteil von *SQL* bekannt. Eine Anfragesprache sollte diese unterstützen, um komplexe Anfragen über mehrere Dokumente zu ermöglichen.

3. Umstrukturierung

   Oft repräsentiert ein Ergebnis ein neues Dokument, in dem die selektierten Daten in einem anderen Kontext stehen. Die Möglichkeit der Umstrukturierung ist daher ein wichtiges Feature einer Anfragesprache.

Die folgenden Beispiele beziehen sich auf zwei Arten von Dokumenten. Zum einen haben wir Dokumente von Herstellern (*manufacturer*), welche den Herstellernamen, Jahr, Modelle mit ihren Namen und verschiedenen Bewertungen wie *front rating* und *side rating* beinhalten. Zum anderen existieren Dokuemente vom Typ *vehicle*, die die Element *vendor, make, modle, year, color* und *price* enthalten. Die XML–Dokumente sehen folgendermaßen aus:

```
<manufacturer>
        <mn_name>Mercury</mn_name>
        <year>1999</year>
        <model> <mo_name>Sable LT</mo_name>
                <front_rating>3.84</fron_rating>
                <side_rating>2.14</side_rating>
                <rank>9</rank>
        ...
        </model>
```

```
</manufacturer>

<vehicle>
        <vendor>Scott Thomason</vendor>
        <make>Mercury</make>
        <model>Sable LT</model>
        <year>1999</year>
        <color>metallic blue</color>
        <price>27800</price>
        ...
</vehicle>
```

2.3.1 Selektion auf Dokumenten

Die Selektion ist das Ergebnis einer Anfrage auf ein Dokument. Dabei beinhaltet das Resultat die Elemente des Dokuments, die in der *Select*–Klausel spezifiziert wurden und bestimmten Bedingungen genügen.
Die Selektion wird natürlich von allen Sprachen unterstützt.

	Lorel	XML-QL	XQL
Selektion	X	X	X

2.3.1.1 Selektion: Lorel

```
select M
from nhsc.manufacturer M
where M.model.rank <=10
```

Einem Betrachter mit SQL–Erfahrung macht dieses Beispiel sicher keine Schwierigkeiten. Der *Constructor* befindet sich in der *select*-Klausel. In der *from*-Klausel steht das *Pattern* und in der *where*-Klausel befinden sich *Patterns* und *Filter*.

Das Resultat dieser Query ist eine Menge Objekt-IDs, die auf Elemente in dem abgefragten Dokument zeigen.

2.3.1.2 Selektion: XML-QL

```
WHERE <manufacturer>
        <model>
          <rank>$r</rank>
        </model>
      </manufacturer> ELEMENT_AS $m
                IN "www.nhsc/manufacturers.xml",$r<=10
CONSTRUCT $m
```

Die Query bezieht sich auf das Dokument *www.nhsc/manufacturers.xml*. Der Inhalt des Elementes *rank* wird an die Variable *r* gebunden, welche dann in der *Filter*–Klausel verglichen wird.
In der *Constructor*–Klausel werden die Elemente spezifiziert, welche Ergebnis der Anfrage sein sollen. Das Resultat ist stets eine duplikatfreie Ergebnismenge, wobei die Struktur des Ergebnisses nicht genau definiert ist.

2.3.1.3 Selektion: XQL

```
document("www.nhsc/manufacturer.xml")/manufacturer[model
   /rank<=10]
```

Wie schon in der Einführung erwähnt und in diesem Beispiel zu sehen ist, sind XQL–Ausdrücke sehr kompakt.
Das *Pattern* ist `/manufacturer` und der Filter `[model/rank<=10]`. Der Filter ist per Existenz quantifiziert. Das Resultat wird innerhalb eines Standardelementes erzeugt.

```
<xql:result>
    .
```

```
.
.
</xql:result>
```

2.3.2 Joins über Dokumente

Bei einer *join*-Operation ("inner join") können verschiedene Dokumente miteinander zu einem einzigen Ergebnisdokument kombiniert werden. Dabei werden Daten eines oder mehrerer Dokumente miteinander verglichen.
In Lorel werden *joins* voll unterstützt. Sie können über dieselben sowie über mehrere Dokumente definiert werden und sind, wie in Lorel üblich, im SQL–Stil geschrieben.
In XML-QL werden *joins* implizit durch die Gleichheit über Variablenbindungen ausgedrückt. Auch hier kann der *join* im selben Dokument sowie über mehrere Dokumente gebildet werden.
In XQL gab es ursprünglich keine joins, nur sogenannte *semi-joins*. Dabei können Daten nur entlang eines Pfades innerhalb desselben Dokumentes definiert werden. Echte *joins* gibt es als Erweiterung von Peter Fankhauser (GMD-IPSI), Harald Schöning (Software AG) und Gerald Huck (GMD-IPSI).
Im folgenden Beispiel sollen Paare aus *<manufacturer>* und *<vehicle>* Elementen gebildet werden. Dabei soll der *join* über die Elemente *<mn_name>*=*<make>*, *<mo_name>*=*<model>*, *<year>*=*<year>* gebildet werden.

	Lorel	XML-QL	XQL
Join	X	X	als Erweiterung

2.3.2.1 Join: Lorel

```
tmp:=select (M,V) as pair
      from nhsc.manufacturer M, nhsc.vehicle V
```

```
where M.mn_name = V.make
  and M.model.mo_name = V.model
  and M.year = V.year
```

Das Resultat der Anfrage in Lorel sind Paare von Dokument–OIDs. Der Knoten *tmp* wird als neuer Zugangsknoten definiert, der auf die OIDs der Ergebnisdokumente zeigt.

2.3.2.2 Join: XML-QL

```
WHERE <manufacturer>
        <mn_name>$mn</mn_name>
        <year>$y</year>
        <model> <mo_name> $mon </mo_name> </model>
CONTENT_AS $mo
      </manufacturer>
CONTENT_AS $m IN www.nhsc\manufacturers.xml
      <vehicle>
        <model>$mon</mo_name>
        <year>$y</year>
        <make>$mn</make>
      </vehicle> CONTENT_AS $v IN www.nhsc\vehicles.xml
CONSTRUCT <manufacturer>
           <mn_name>$mn</mn_name>
           <year>$y</year>
           <vehiclemodel> $mo,$v </vehiclemodel>
          </manufacturer>
```

In der *CONSTRUCT*-Klausel wird ein neues Element <vehiclemodel> erzeugt. Es beinhaltet die an die Variablen $mo und $v gebundenen Elemente <model> und <vehicle>.

2.3.2.3 Join: XQL

Die Operation *join* ist nicht im eigentlichen XQL–Sprachvorschlag enthalten. Da sie nur als Erweiterung dazu existiert, soll an dieser Stelle nur an einem vereinfachten Beispiel gezeigt werden, wie eine *join*–Operation mit XQL aussehen könnte. Es sollen jetzt zwei Dokumente über nur ein Element (<mn_name> und <make>) verbunden werden.

```
document("www.nhsc\manufacturer.xml")/manufacturer[$a:=mn_name]
        {
        $a | year
        document("www.nhsc\vehicle.xml")/vehicle[make=$a] {
                model | color | price
        }
}
```

Der *join* wird hier ebenfalls über eine Variablenbindung ($a) ausgeführt. Das Ergebnis der Anfrage wird durch die *Pattern*–Klausel bestimmt. Es enthält die Elemente <mn_name>,<year> von <manufacturer> und die Elemente <model>,<color>,<price> von <vehicle>.

2.3.3 Umstrukturierung

Um ein XML–Dokument umzustrukturieren, muß die Anfragesprache einen *Construction*–Mechanismus besitzen, der es erlaubt, neue Elemente zu erzeugen. Wie schon in den vorhergehenden Abschnitten erwähnt, ist XQL die einzige der hier betrachteten Sprachen, die diesen Mechanismus nicht besitzt. Lorel stellt zur Elementerzeugung die Funktion *xml(...)* zur Verfügung. In XML-QL wird die neue Struktur des Dokuments in der *CONSTRUCT*–Klausel definiert. Diese kann neue XML–*Tags*, Konstanten und Variablen enthalten.

	Lorel	XML-QL	XQL
Umstrukturierung	X	X	-

Im folgenden Beispiel sollen <car>-Elemente erzeugt werden, die die Elemente <make>,<model>,<vendor>,<rank> und <price> enthalten.

2.3.3.1 Umstrukturierung: Lorel

```
select xml(car: (select X.vehicle.make, X.vehicle.model,
                X.vehicle.vendor, X.manufacturer.rank,
                X.vehicle.price
                from tmp.pair X))
```

Die Elemente werden in der selben Reihenfolge aus den Dokumenten extrahiert, wie sie in der der *select*-Klausel der Query erscheinen. Die Funktion *xml(car: querystring)* erzeugt ein neues XML Element <*car*>, welches die in der *select*-Klausel spezifizierten Elemente beinhaltet.

2.3.3.2 Umstrukturierung: XML-QL

```
WHERE <manufacturer>
          <mn_name>$mn</mn_name>
          <vehiclemodel>
            <model> <mo_name>$mon</mo_name>
    <rank>$r</rank>
    </model>
          <vehicle>
            <price>$p</price>
            <vendor>$v</vendor>
          </vehicle>
        </vehiclemodel>
      </manufacturer> IN www.nhsc\queryresult3..xml
```

```
CONSTRUCT <car> <make>$mn</make>
                <mo_name> $mon </mo_name>
                <vendor>$v</vendor>
                <rank>$r</rank>
                <price>$p</price>
          </car>
```

Die neue Struktur oder Sortierung des Dokuments entsteht in der *CONSTRUCT*–Klausel. In ihr können neue XML–Elemente definiert werden. In XML-QL ist das explizite Aufrufen einer Funktion zur Elementerzeugung nicht notwendig. Die Benutzung ist sehr intuitiv, das Muster der *CONSTRUCT*–Klausel spiegelt das neue XML–Dokument wieder, es müssen nur die gebundenen Variablen ausgewertet werden.

2.4 Zusammenfassung

Tabelle 2.1 gibt noch einmal einen Überblick der betrachteten Sprachen und führt außerdem weitere Features auf, die von einer XML–Anfragesprache erwartet werden.
Wie aus der Tabelle zu erkennen ist, erweisen sich Lorel und XML-QL als die mächtigsten der drei Sprachen. Ihre Ausdrucksmöglichkeiten könnte man mit hochentwickelten SQL–Standards wie SQL92 vergleichen. Sie wurden unabhängig voneinander entwickelt und weisen doch viele Gemeinsamkeiten auf. Das mag sicher daran liegen, daß beide Sprachen aus der Datenbankgemeinschaft kommen und von vornherein versucht wurde, ihnen Datenbankfähigkeiten zukommen zu lassen (datenbankorientiert). Deshalb besitzt die Syntax der Sprachen auch die gleiche Struktur. Der Tabelle 2.1 ist aber auch zu entnehmen, daß Lorel die mächtigere der beiden Sprachen ist.
XQL ist dagegen doch stark in ihren Ausdrucksmöglichkeiten eingeschränkt. Der Grund hierfür liegt in den Ursprüngen der Sprache. Mit XQL wollten die

Entwickler eine Möglichkeit schaffen, Elemente aus XML–Dokumenten zu selektieren und auch eine Volltextsuche auf sehr großen Dokumenten durchzuführen und das möglichst effektiv. Deshalb nennt man XQL auch eine dokumentorientierte Sprache. Sie ist keine vollständige Anfragesprache, das soll sie auch nicht sein. Sie ist vielmehr der nützlichste Teil einer Anfragesprache. Deshalb sollte XQL aber nicht unterbewertet werden. Es gibt auch Anfragen, die mit XQL aber nicht mit Lorel und XML-QL ausgedrückt werden können. XQL ist eine einfach zu erlernende, allgemein einsetzbare und sehr kompakte Sprache, dadurch könnte eine XQL–Query als Teil einer URL mit übergeben werden.

Mittlerweile gibt es einige Erweiterungen, um XQL ausdrucksstärker zu machen und die großen Defizite der Sprache zu beseitigen. Inwieweit diese Neuerungen aber in dem Sprachvorschlag berücksichtigt werden, war an dieser Stelle nicht festzustellen.

	Lorel	XML-QL	XQL
Gruppierung	X	-	-
Quantoren	X	nur Existenz	X
Aggregatfkt.	X	- (geplant)	count()
Schachtelung	X	X	-
Binäre Queries	X	keine Differenz	X[1]
Sortierung	X	X	-
RDF/Schemas	-	-	-
XLink/XPointer	-	-	-
Insert,Delete,Update	X	-	-

Tabelle 2.1: Vergleich von Lorel, XML-QL und XQL

Kapitel 3

XQuery 1.0

Die XML–Anfragesprache *XQuery 1.0* liegt beim *W3C*[1] als "Working Draft" vor. Das zeigt, daß XQuery sich erst im Anfangsstadium der Entwicklung befindet und die Sprache noch vier weitere Schritte bei dem *W3C* bis zur *W3C Recommendation* zu durchlaufen hat.
Dieses Kapitel soll einen ausführlichen Überblick über die Anfragesprache XQuery geben, es bezieht sich dabei auf das "Working Draft" [7].

XQuery ist eine einfache, präzise und gut lesbare Anfragesprache. Mit ihr lassen sich Anfragen auf verschiedenste Art formulieren. XQuery ist von der älteren XML–Anfragesprache Quilt abgeleitet, welche wiederum Features von vielen anderen Sprachen in sich vereint. So benutzt Quilt und demzufolge ebenso XQuery Pfadausdrücke von XPath und XQL, welche zur Navigation durch hierarchischen Dokumenten dienen. Von XML-QL wird der Begriff der Variablenbindung genutzt, die es ermöglicht, Variablen an Pfadausdrücke zu binden und diese weiter zur Erzeugung von neuen Resultatstrukturen nutzt. Von SQL wurde die *SELECT-FROM-WHERE*–Klausel verwendet, welche sich im Datenbankbereich als

[1]World Wide Web Consortium; *http://www.w3.org*

ein geeignetes Muster zur Datenrestrukturierung bewährt hat. Außerdem wurde von OQL die Idee von funktionalen Sprachen benutzt, bei denen eine Anfrage aus verschiedenen Ausdrücken zusammengesetzt wird und uneingeschränkt geschachtelt werden kann. Quilt wurde auch von anderen XML–Anfragesprachen wie Lorel und YATL beeinflußt.

XQuery ist eine funktionale Sprache, das heißt, daß jede Anfrage durch einen Ausdruck repräsentiert wird. Die Sprache unterstützt verschiedene Ausdrücke, wodurch diverse Anfragen sich in Struktur und Aussehen stark unterscheiden können, abhängig von der Art des benutzten Ausdrucks. Die Ausdrücke sind orthogonal aufeinander anwendbar.
Ein- und Ausgabe einer XQuery–Anfrage sind Instanzen eines Datenmodells, welches von XQuery 1.0 und XPath 2.0 benutzt wird. Dieses Datenmodell ist eine Weiterentwicklung des Datenmodells von XPath 1.0, bei dem ein Dokument als ein Baum von Knoten modelliert wird. Eine Instanz des Datemmodells ist jetzt eine Sequenz von Knoten, von der jeder Knoten wieder eine geschachtelte Liste von Knoten enthalten kann. Das hat den Vorteil, daß jetzt mit geordneteen Knotenlisten operiert wird, anstatt auf Mengen von Knoten. Es soll hier explizit erwähnt werden, daß in diesem Datenmodell in einer Sequenz Duplikate von Knoten erlaubt sind.

Im folgenden sollen die verschiedenen Arten von Ausdrücken erläutert werden sowie die Syntax anhand von Beispielen gezeigt werden.
Die grundlegenden XQuery–Ausdrücke sind:

- Pfadausdrücke
- Element Konstruktoren
- FLWR–Ausdrücke
- Ausdrücke mit Operatoren und Funktionen

- Bedingte Ausdrücke
- Quantoren
- Ausdrücke, die Datentypen testen oder modifizieren

In den nächsten Abschnitten sollen die verschiedenen Ausdrücke erklärt werden und jeweils der entsprechende Teil, der zu den Ausdrücken gehörigen Syntax, gezeigt werden. Eine formale Beschreibung der Semantik ist in [11] zu finden.

3.1 Pfadausdrücke

Wesentlicher Bestandteil von XQuery sind Pfadausdrücke. Deren Syntax basiert auf XPath 1.0 [6]. Eine Erweiterung von XPath ist XPath 2.0 (basierend auf "XPath 2.0 Requirements" [12]), von dem unter anderem verlangt wird, daß es zu XPath 1.0 rückwärts kompatibel sein soll. Pfadausdrücke stellen einen wesentlichen Bestandteil und die Voraussetzung für XQuery dar. Sie ermöglichen die Navigation durch ein Dokument entlang eines bestimmten Pfades.

Syntax:

```
PathExpr         ::=  RelativePathExpr
                      | ("/" RelativePathExpr?)
                      | ("//" RelativePathExpr?)
RelativePathExpr ::=   StepExpr ( ("/" | "//") StepExpr)*
StepExpr         ::=   AxisStepExpr | OtherStepExpr
AxisStepExpr     ::=   Axis NodeTest StepQualifiers
OtherStepExpr    ::=   PrimaryExpr StepQualifiers
StepQualifiers   ::=   ( ("[" Expr "]") | ("=>" NameTest) )*
Axis             ::=   (NCName "::") | "@"
PrimaryExpr      ::=   "."
```

```
                     | ".."
                     | NodeTest
                     | Variable
                     | Literal
                     | FunctionCall
                     | ParenthesizedExpr
                     | CastExpr
                     | ElementConstructor
Literal         ::=  NumericLiteral | StringLiteral
NodeTest        ::=  NameTest | KindTest
NameTest        ::=  QName | Wildcard
KindTest        ::=  PITest | CommentTest | TextTest
                     | AnyKindTest
PITest          ::=  "processing-instruction"
                     "(" StringLiteral? ")"
CommentTest     ::=  "comment" "(" ")"
TextTest        ::=  "text" "(" ")"
AnyKindTest     ::=  "node" "(" ")"
```

Ein Pfadausdruck besteht aus mehreren Schritten (*steps*). Jeder Schritt bedeutet eine Bewegung durch das XML–Dokument in eine bestimmte Richtung. Die Richtung wird dabei durch die Achse bestimmt. Es können nur Knoten entlang dieses Pfades selektiert werden. Welche Knoten entlang der Achse selektiert werden, bestimmt der Name des *Name Test* sowie die optionalen Prädikate des *Location Step*. Das Ergebnis eines Schrittes ist eine Knotenliste, welche die Startknoten für die Auswertung des nächsten Schrittes enthält.

Die in XPath verwendeten Symbole haben folgende Bedeutung: Ein Ausdruck beginnt oft mit der Funktion "document(String)", sie iden-

Symbol	Sematik
"."	Bedeutet den aktuellen Knoten.
".."	Bedeutet den *parent* Knoten des aktuellen Knotens.
"/"	Steht für den *root*–Knoten oder wird zur Trennung von *Location Steps* verwendet und beschreibt in diesem Zusammenhang die *Child*–Achse.
"//"	Beschreibt alle "Nachkommen" des aktuellen Knotens (alle Knoten, die der aktuelle Knoten umschließt).
"@"	Steht für die Attribute des aktuellen Knotens.
"*"	Steht für alle Knoten eines *Location Steps*.
"[ ]"	In eckigen Klammern wird ein Boolscher Ausdruck definiert. Dieser dient als Prädikat eines *Location Steps*.
"n"	Ist Element der natürlichen Zahlen und dient zur Selektion des bezüglich der Ordnung n-ten Knotens.

Tabelle 3.1: In XPath verwendete Symbole

tifiziert genau einen Knoten, den *root*–Knoten des Dokumentes, dessen Name als Parameter der Funktion übergeben wurde. Außerdem kann ein Pfadausdruck mit einer gebundenen Variablen oder den Symbolen "/" und "//" beginnen. Diese implizieren den *root*–Knoten bezüglich des Kontextes, in dem sich der Pfadausdruck befindet. Das Ergebnis eines Pfadausdrucks ist eine Sequenz von Knoten oder einfachen Werten (Datentypen).

Das folgende Beispiel zeigt einen Pfadausdruck, der aus drei Schritten besteht. Im ersten Schritt wird der *root*–Knoten des Dokumentes "familien.xml" ermittelt. Der zweite Schritt wählt das ordnungsmäßig erste *child*– Element (ausgehend vom *root*–Knoten entlang der *child*–Achse) mit dem *Qualified Name* "familie". Im dritten und letzten Schritt werden alle "person" Elemente ausgewählt, die auf der *descendant*–Achse der "familie"–Elemente liegen (also dessen "Nachkommen" sind). Außerdem müssen sie die Bedingung erfüllen, daß sie *child*–Knoten mit dem *QName* "alter" besitzen, deren Inhalte größer als 10 sind.

```
document("familien.xml")/familie[1]//person[ alter > 10 ]
```

In XQuery wird ein neues Prädikat, das *RANGE*–Prädikat, eingeführt. Mit diesem ist es möglich, bezüglich der Ordnung mehrere Knoten eines definierten Bereiches auszuwählen.
Bsp.: Finde alle Figuren aus dem Dokument "buch.xml" aus dem zweiten bis fünften Kapitel.

```
document("buch.xml")/kapitel[RANGE 2 TO 5]//figuren
```

Zusätzlich zu den Operatoren von XPath wird in XQuery ein neuer Operator eingeführt, der Dereferenzierungsoperator (engl.: dereference operator). Der Aufruf des Operators erfolgt durch das Symbol "=>". Ein Dereferenzierungsoperator kann nach einem Attribut vom Typ *IDREF* oder *IDREFS* eingesetzt werden und liefert die Elemente, die von diesem Attribut referenziert werden. Auf der rechten Seite eines Dereferenzierungsopertors steht ein *Name Test*, welcher die Zielelemente spezifiziert.
Bsp.: Gebe alle Namen von weiblichen Vorgesetzten aus dem Dokument "firma.xml" aus.

```
document("firma.xml")/person/@vorgesetzter=>
                              person[geschlecht = "weiblich"]/name
```

In Pfadausdrücken können auch arithmetische Operatoren benutzt werden. Diese sind auf einfachen Datentypen definiert. Wird ein arithmetischer Operator in Verbindung mit einen Knoten als Operand angewendet, erfolgt ein impliziter Aufruf der Funktion "data()", welche den numerischen Wert des Knoteninhalts extrahiert.
In einem Beispiel soll der zwölffache Wert des Gehaltes von Personen, das Jahresgehalt, ausgeben werden:

```
document("firma.xml")/person/gehalt * 12
```

3.2 Elementkonstruktoren

Mit Hilfe von Elementkonstruktoren ist es möglich, neue XML–Elemente zu generieren.

Syntax

```
ElementConstructor    ::= "<" NameSpec AttributeList ("/>"
                          | (">" ElementContent*
                          "</" (QName S?)? ">") )
NameSpec              ::=  QName | ( "{" Expr "}" )
AttributeList         ::= (S (NameSpec S? "=" S?
                             (AttributeValue
                              | EnclosedExpr)
                             AttributeList)? )?
AttributeValue        ::= ( ["] AttributeValueContent* ["] )
                          | ( ['] AttributeValueContent* ['] )
ElementContent        ::=  Char
                          | ElementConstructor
                          | EnclosedExpr
                          | CdataSection
                          | CharRef
                          | PredefinedEntityRef
AttributeValueContent ::=  Char
                           | CharRef
                           | EnclosedExpr
                           | PredefinedEntityRef
CdataSection          ::=  "<![CDATA[" Char* "]]>"
EnclosedExpr          ::=  "{" ExprSequence "}"
```

Zu beachten ist in dem Grammatikausschnitt, daß hier *Whitespaces* berücksichtigt werden. Diese sind wichtig, wenn der Elementinhalt einen Text als

Konstante enthält. In diesem sollten auch *Whitespaces* beachtet werden. Das ist eine Erweiterung zu der Grammatik in [8].

Elementkonstruktoren beginnen mit einem *Start Tag* und enden mit einem *End Tag*. Dazwischen befindet sich der Inhalt des Elements (*Element Content*). Im *Start Tag* können außerdem Attribute und dessen Werte spezifiziert werden. Elementkonstruktoren können in andere XQuery–Anfragen eingebettet werden. Im Ergebnis der Query wird der Elementkonstruktor einfach durch sich selbst repräsentiert. Ein einfaches Beispiel für einen Elementkonstruktor kann wie folgt aussehen:

```
<person alter=51>
        <vorname>Hans Joachim</vorname>
        <nachname>Rost</nachname>
</person>
```

Teile von Elementkonstruktoren, wie der Elementinhalt oder der Attributwert, können auch durch einen geschachtelten *XQuery*–Ausdruck erzeugt werden. Diese Ausdrücke stehen anstelle des Attributwertes oder Elementinhaltes und müssen, da es keine XML–Literale sind, von geschweiften Klammern ("{...}") eingeschlossen werden. Diese stehen als Indikator für einen XQuery–Ausdruck, welcher weiter ausgewertet wird, im Gegensatz zu normalem Text, der einfach in das Ergebnis übernommen wird. Im Beispiel soll der Sachverhalt verdeutlicht werden:

```
<person alter= {$alter}>
        <vorname> {$vorname} </vorname>
        <nachname> {$nachname} </nachname>
</person>
```

Bis hierhin wurden die Elementnamen und Attributnamen als Konstanten spezifiert. Diese sollen jetzt ebenfalls durch XQuery–Ausdrücke erzeugt werden können. Dazu werden die XPath–Funktionen *name(Element)*, die den

Tag–Namen des Elementes liefern, benötigt. Wenn innerhalb eines Elementkonstruktors ein Attributelement evaluiert wird, so wird das Ergebnis automatisch zum Attribut des konstruierten Elementes. Im nächsten Beispiel wird ein Element erzeugt, das den Elementnamen von dem an $a gebundenen Element erhält.

```
<{name($a)}>
        { $a/@* }
        { 2 * number($a) }
</>
```

3.3 FLWR–Ausdrücke

FLWR (gesprochen engl.: *flower*) ist eine Abkürzung und steht für *FOR LET WHERE RETURN*. Diese Konstrukte müssen in der Reihenfolge *FOR|LET - WHERE - RETURN* verwendet werden.

In einem FLWR–Ausdruck lassen sich in der *FOR*- und *LET*–Klausel Knoten an Variablen binden. Diese können in der *WHERE*–Klausel weiter gefiltert werden, um die resultierende Knotenmenge unter Verwendung von Vergleichsoperatoren einzuschränken. Die *RETURN*–Klausel dient zur Generierung des Anfrageergebnisses. In ihr können die gebundenen Variablen in Verbindung mit Elementkonstruktoren zur Erzeugung eines neuen XML–Dokumentes, einer Instanz des verwendeteten Datenmodells, benutzt werden.

Syntax:

```
FLWRExpr    ::= (ForClause | LetClause)+ WhereClause?
                "return" Expr
ForClause   ::= "for" Variable "in" Expr
                ("," Variable "in" Expr)*
LetClause   ::= "let" Variable ":=" Expr
```

```
                  ("," Variable ":=" Expr)*
WhereClause ::= "where" Expr
```

3.3.1 Die FOR–Klausel

In der *FOR*–Klausel werden eine oder mehrere Variablen an einen Pfadausdruck gebunden, dessen Auswertung eine Liste (Sequenz) von Knoten ergibt. In der Regel sind es Pfadausdrücke (oder besser deren Ergebnis), die an Variablen gebunden werden, es können aber auch beliebige Ausdrücke sein.
Das Resultat einer *FOR*–Klausel ist eine Liste von Tupeln. Jedes dieser Tupel enthält eine Variablenbindung für jede in der *FOR*–Klausel definierte Variable. Die Variablen sind an jeden Knoten einzeln gebunden, die von den zugehörigen Ausdrücken erzeugt wurden. Mathematisch betrachtet repräsentiert somit die Menge der Tupel das Kreuzprodukt aller Variablenbindungen. Dieses Konzept der Variablenbindung stellt eine Iteration der Variablen über alle von dem entsprechenden Ausdruck gelieferten Knoten dar.

3.3.2 Die LET–Klausel

In der *LET*–Klausel können ebenfalls ein oder mehrere Knoten an eine oder mehrere Variablen gebunden werden. Im Unterschied zur *FOR*–Klausel, wird hier jede Variable an den zurückgelieferten Wert des zugehörigen Pfadausdrucks gebunden. Eine Iteration über die Ergebnisknotenmenge findet bei der *LET*–Klausel nicht statt, sodaß es nur eine einzige Variablenbindung gibt. Daher werden in der *LET*–Klausel gebundene Variablen oft als Parameter für setorientierte Funktionen wie zum Beispiel die "build in"–Funktionen *count*, *min*, *max*, *sum* oder *avg* benutzt.
Bsp. für eine *FOR*– Klausel: Es werden soviele Variablenbindungen erzeugt, wie es Bücher in der Bibliothek gibt. Bei jeder Variablenbindung wird ein

Buch an die Varibale \$a gebunden.

```
FOR $a IN document("bib.xml")/buecher
```

Im Gegensatz dazu wird im nächsten Beispiel nur eine Variablenbindung durchgeführt, nämlich die Liste aller Bücher einer Bibliothek an die Variable \$b. Es sei nochmals erwähnt, daß über diese Liste keine Iteration stattfindet.

```
LET $b := document("bib.xml")/buecher
```

Jetzt wird deutlich, daß die Anzahl der gebundenen Tupel nicht von der *LET*–Klausel abhängig ist. Die Größe der Tupelliste wird durch die in der *FOR*–Klausel gebundenen Variablen bestimmt.

3.3.3 Die WHERE–Klausel

Die *WHERE*–Klausel dient zur Spezifizierung weiterer Selektionsbedingungen. Nach dem Schlüsselwort *WHERE* muß ein *boolscher* Ausdruck, welcher die Selektionsbedingung repräsentiert und dem die Tupel genügen sollen, angegeben werden. Ergibt die Auswertung des Ausdrucks für das jeweilige Tupel *TRUE*, verbleibt es in der Liste der gebundenen Tupel zur späteren Auswertung durch die *RETURN*–Klausel. In der *WHERE*–Klausel können mehrere Ausdrücke spezifiziert werden und mit den logischen Operatoren *AND*, *OR* und *NOT* verknüpft werden. Die Ausdrücke referenzieren oft Variablen, die in der *FOR*- und *LET*–Klausel gebunden wurden. Wurde eine Variable in der *FOR*–Klausel gebunden, repräsentiert diese einen einzelnen Knoten und kann somit in skalaren Prädikaten, z.B. `$a/jahr > 2000`, verwendet werden. Variablen, die in der *LET*–Klausel gebunden wurden, werden hingegen häufig in "list orientierten" Prädikaten wie `avg(\$b/person/alter) > 10` verwendet.

3.3.4 Die RETURN–Klausel

Die *RETURN*–Klausel dient zur Erzeugung des Ergebnisses eines *FLWR*–Ausdrucks. Das Ergebnis kann eine beliebige Folge von Knoten oder einfache Datentypen beinhalten. Die *RETURN*–Klausel wird für jedes Tupel, das die Selektionbedingung in der *WHERE*–Klausel erüllt, einmal ausgeführt. Der in der *RETURN*–Klausel verwendete Ausdruck benutzt oft Referenzen zu in der *FOR* bzw. *LET* gebundenen Variablen, Elementkonstruktoren oder auch andere Ausdrücke.

3.4 Operatoren

XQuery erlaubt die Konstruktion von Ausdrücken unter Verwendung von *infix* und *prefix* Operatoren. In einer Anfrage können Ausdrücke, als Operanden bestimmter Operatoren eingesetzt werden. XQuery unterstützt die gewöhnlichen arithmetischen und logischen Operatoren sowie Operatoren auf Sequenzen.

3.4.1 Arithmetische Operatoren

Syntax

```
AdditiveExpr       ::=   Expr ("+" | "-") Expr
MultiplicativeExpr ::=   Expr ("*" | "div" | "mod") Expr
UnaryExpr          ::=   ("-" | "+") Expr
```

XQuery unterstützt Operatoren für die allgemein bekannten arithmetischen Operationen Addition (+), Subtraktion (-), Mulitplikation (*), Division (DIV) und Modulo (%) in den bekannten unären und binären Anwendungen.

Sind beide Operanden numerische Typen, ist das Ergebnis eindeutig. Wenn ein oder mehrere Operanden Knoten sind, wird der Inhalt des Knotens durch

den Aufruf der Funktion *data()* extrahiert und in einen numerischen Wert konvertiert. Ist diese Konvertierung nicht möglich, weil der Knoten keinen numerischen Wert beinhaltet, wird die Operation mit einem Fehler abgebrochen.
Ist hingegen ein Operand ein numerischer Typ und der andere eine Knotenliste, wird die Operation mit dem numerischen Typ und jedem einelnen Knoten der Liste durchgeführt. Das Ergebnis dieser Operation ist eine Liste von numerischen Werten.
Wie das Ergebnis aussieht, wenn die Operatoren auf zwei Listen angewendet werden, ist zum gegenwärtigen Zeitpunkt noch Gegenstand von Diskussionen, welche wahrscheinlich im nächsten "Working Draft" geklärt sind.

3.4.2 Vergleichsoperatoren

Syntax:

```
EqualityExpr   ::=   Expr ("=" | "!=" | "==" | "!==") Expr
RelationalExpr ::=   Expr ("<" | "<=" | ">" | ">=") Expr
```

Es werden in XQuery mehrere Vergleichsoperatoren unterstützt. Diese sind alle binäre Operatoren und geben einen boolschen Wert zurück.
Wird ein einzelner Wert mit einer Liste verglichen, gibt der Operator den Wert *TRUE* zurück, wenn mindestens ein Wert aus der Liste existiert, für den der Vergleich mit dem Einzelwert mit *TRUE* beantwortet werden kann. Werden zwei Listen miteinander verglichen, ist das Ergebnis *WAHR*, wenn mindestens ein Wert der einen Liste und ein Wert der anderen Liste existiert, auf die die Anwendung des Operators *TRUE* ergibt.

Es stehen in XQuery die Vergleichsoperatoren "=", "!=", "<", ">", "<=" und ">=" zur Verfügung. Sind die Operanden einfache Datentypen vom selben Typ, ist das Ergebnis klar. Wenn die Operanden einfache, zueinander kompatible, Datentypen sind (z.B. *integer* und *float*), wird

der weniger "umfassende" (die Obermenge) Datentyp zum mehr "umfassenden" Datentyp konvertiert. Zum Beispiel würde beim Vergleich eines *integer*–Wertes mit einem *float*–Wert der *integer*–Wert zu einem *float*–Wert konvertiert werden.
Wenn ein einfacher Datentyp mit einem Knoten verglichen wird, wird der Inhalt des Knotens implizit durch den Aufruf der Funktion *data()* extrahiert und anschließend der Vergleich durchgeführt. Sind beide Operanden Knoten, werden die *string*–Werte der beiden Knoten verglichen.

Die Operatoren "==" und "!==" sind für Vergleiche der Knotenidentität über Knoten und Knotenlisten definiert. Sind die Operanden des Operators "==" beide Knoten, wird als Ergebnis *TRUE* zurückgegeben, wenn die Identität beider Knoten gleich ist. Ist einer der Operanden eine Knotenliste, wird die oben genannte Regel ausgeführt. Der Operator "!==" liefert als Ergebnis *TRUE*, wenn der Operator == *FALSE* liefert.

3.4.3 Logische Operatoren

Syntax:

```
OrExpr  ::=  Expr "or" Expr
AndExpr ::=  Expr "and" Expr
```

In XQuery werden die logischen Operatoren *AND* und *OR* unterstützt. Als Operanden werden zwei boolsche Ausdrücke erwartet und das Ergebnis ist ebenfalls ein boolscher Wert. XQuery unterstützt nicht den logischen Operator *NOT*. Anscheinend war der Operator Gegenstand von Diskussionen, da dieser im "Working Draft vom Februar 2001" noch Bestandteil von XQuery sein sollte. Dafür wird eine Funktion *not()* bereitgestellt, die einen boolschen Wert als Argument entgegennimmt und die Negation des Argumentes zurückgibt.

3.4.4 Operatoren über Sequenzen

Syntax:

```
ParenthesizedExpr   ::=   "(" ExprSequence? ")"
ExprSequence        ::=   Expr ("," Expr)*
RangeExpr           ::=   Expr "to" Expr
UnionExpr           ::=   Expr ("union" | "|") Expr
IntersectExceptExpr ::=   Expr ("intersect" | "except") Expr
BeforeAfterExpr     ::=   Expr ("before" | "after") Expr
```

XQuery basiert mit *XPath 2.0* auf dem selben Datenmodell [9], in dem geordnetete Listen/Sequenzen[2] von Werten unterstützt werden. Dabei macht es keinen Unterschied, ob es sich um einen einzelen Wert oder eine Liste von Werten handelt. Eine Liste hat immer eine Tiefe von eins. Das bedeutet, daß eine Liste nie Element einer anderen Liste sein kann. Alle listenerzeugenden Operatoren konvertieren zuvor ihre Operanden zu Listen der Tiefe eins. Die Ergebnisliste ist dann eine neue Liste der Tiefe eins, die als Elemente, die der Quellisten enthält.

Der Basisoperator zur Erzeugung von Listen ist das Komma ",". Wird dieses auf zwei Ausdrücke angewendet, erzeugt er aus den Ergebnissen der Ausdrücke eine neue Liste. Dabei wird die Liste, die sich aus dem rechten Operanden ergibt an die Liste, die sich aus dem linken Operanden ergibt, angehängt. Die Größe der neuen Liste ergibt sich aus der Größe der Resultate der Operanden. Ein skalarer Wert wird als Liste der Länge eins behandelt. Eine leere Liste wird durch leere Klammern "()" repräsentiert. Besondere Beachtung muß dem Erzeugen von Listen innherhalb von

[2]Der Begriff Liste soll in diesem Dokument gleichbedeutend mit Sequenz sein. Dabei soll aber keine bestimmte Liste gemeint sein. Die Operationen auf dieser Liste werden in diesem Abschnitt vorgestellt. Der Begriff Liste soll die Möglichkeit von Duplikaten implizieren.

Funktionen geschenkt werden, da dort das Komma als Seperator der Funktionsargumente dient. Ein Funktionsaufruf mit einem skalaren Argument und einer Liste der Länge zwei sieht folgendermaßen aus: *f(1,(1,2))*.
Einen anderen Weg zur Erzeugung von Listen bietet der *TO*–Operator. Der binäre Operator konvertiert seine Operanden zu *integer* Werten (falls das nicht möglich ist, wird mit einem Fehler abgebrochen) und erzeugt eine Liste, die alle *integer* Werte, beginnend mit dem linken Wert bis zum rechten (einschließlich) Wert enthält. Ist der linke Wert größer als der rechte, wird eine Liste mit abnehmenden Werten erzeugt. Als Beispiel soll eine Liste mit fünf Werten, beginnend bei 10, abnehmend erzeugt werden:

```
10 TO 6
```

Weitere Operatoren auf Sequenzen sind *UNION*, *INTERSECT* und *EXCEPT*. Mit ihnen lassen sich Listen als Operatoren zu neuen Ergebnislisten kombinieren.
Der *UNION*–Operator vereinigt zwei Listen zu einer Ergebnisliste, indem er die Elemente beider Listen in das Resultat aufnimmt, dabei aber Duplikate eleminiert. Der *INTERSECT*–Opertor erzeugt eine Liste, die nur Elemente enthält, die in beiden Squenzen der Operanden enthalten sind. Der Operator *EXCEPT* generiert eine Ergebnisliste mit Elementen, die ausschließlich im linken Operator und nicht im rechten Operator erscheinen. Alle drei Operatoren sind duplikateleminierend bezüglich der Knotenidentität.

XQuery unterstützt die *infix* Operatoren *BEFORE* und *AFTER*, die es von XQL geerbt hat. Diese sind wichtig, wenn Informationen über die Position von Elementen innerhalb eines Dokumentes gesucht werden. Der Operator *BEFORE* arbeitet über zwei Listen und gibt diejenigen Elemente der ersten Liste zurück, die vor mindestens einem Element der zweiten Liste erscheinen. Voraussetzung dafür ist, daß beide Listen Untermenge der selben

Instanz des Datenmodells sind. Der Operator *AFTER* ist symmetrisch zu *BEFORE* definiert. Da die beiden Operatoren auf der globalen Ordnung eines Dokumentes basieren, können auch Elemente verglichen werden, deren unmittelbare Eltern nicht die selben sind. Direkt daraus folgt auch, daß die zu vergleichenden Listen sogar ungeordnet sein können.

3.5 Sortierung

Syntax

```
SortExpr     ::= Expr "sortby" "(" SortSpecList ")"
SortSpecList ::= Expr ("ascending" | "descending")?
                 ("," SortSpecList)?
```

Unter Anwendung der *SORTBY*–Klausel können die Ergebnisse von Ausdrücken geordnet werden. Dabei muß in der *SORTBY*–Klausel ein oder mehrere Sortierungsausdrücke angegeben werden. Jeder dieser Sortierungausdrücke wird für jeden Wert der Squenz ausgewertet. Die Auswertung jedes Knotens der Sequenz muß einen Wert ergeben, auf den der Operator ">" definiert ist. Ist das nicht der Fall, wird mit einem Fehler abgebrochen. Die Elemente werden bezüglich der erhaltenen Werte der Sortierungsausdrücke sortiert. Wenn mehr als ein Sortierungsausdruck angegeben wurde, bestimmt der links stehendende Ausdruck die Primäre Sortierung, gefolgt von den restlich Ausdrücken von links nach rechts. Außerdem können die Schlüsselwerte *ascending* und *descending* angegeben werden, um die Elemte der Sequenz auf- oder absteigend zu sortieren.

Desweiteren kann auf Sequenzen die Funktionen *UNORDERER* angewendet werden. Bei der *UNORDERED*–Funktion ist die Reihenfolge der Knoten der Sequenz nicht interessant. Die Funktion kann dann zur Optimierung genutzt werden, indem sie die Sequenz so ordnet, wie es für die Auswertung der

Knotenliste am günstigsten ist.

3.6 Bedingte Ausdrücke (IF, THEN, ELSE)

Syntax:

```
IfExpr ::=   "if" "(" Expr ")" "then" Expr "else" Expr
```

Bedingte Ausdrücke können benutzt werden, um die Struktur des erzeugten Ergebnisses von bestimmten Bedingungen abhängig zu machen. In der *IF*–Klausel wird der zu testende Ausdruck ausgewertet. In der anschließenden *THEN ... ELSE ...* werden die Ausdrücke zur Ergebnisbildung spezifiziert und abhänging vom zu testenden Ausdruck ausgeführt.
Bsp.: Erzeuge eine Liste "bestand", in der alle "autos" in "oldtimer" und "gebraucht" kategorisiert werden.

```
<bestand>
FOR $a IN document("autos.xml")/auto
RETURN { IF  $a/baujahr  < 1950
          THEN <oldtimer> { $a } </oldtimer>
          ELSE <gebraucht> { $a } </gebraucht>
        }
</bestand>
```

3.7 Quantifizierte Ausdrücke

Quantoren bieten eine Möglichkeit, einer Sequenz gewisse Existenzbedingungen von Elemneten zu unterziehen. Das geschieht mit Hilfe des Existenzquantors und des Allquantors oder Universal Quantors. Die zugehörigen XQuery– Operatoren sind *SOME* für den Existenzquantor und *EVERY* für den Allquantor.

Der Existenzquantor liefert den boolschen Wert *TRUE*, wenn in der Sequenz mindestens ein Element existiert, das einem bestimmten Testaudruck genügt. Der Allquantor liefert *TRUE* genau dann, wenn alle Elemente einer Sequenz einer Bedingung genügen.
Syntax:

```
SomeExpr := "SOME" Var "IN" Expr "SATISFIES" Expr
EveryExpr := "EVERY" Var "IN" Expr "SATISFIES" Expr
```

In einem quantifizierenden Ausdruck wird wie bei der *FOR*–Klausel eines *FLWR*–Ausdrucks für jedes Element, das von dem Ausdruck in der *IN*–Klausel zurückgegeben wird, eine Variablenbindung generiert. Für jede dieser Variablenbindungen wird die *SATISFIES*–Klausel einmal ausgeführt.
Die Operatoren *SOME* und *EVERY* liefern immer einen Wert, entweder *TRUE* oder *FALSE*. Wenn nach der Ausführung der *SATISFIES* für alle Variablenbindungen mindestens eine Auswertung *TRUE* ergibt, liefert der *SOME*–Operator den Wert *TRUE*, sonst *FALSE*. Ergibt die Auswertung für alle Variablenbindungen den Wert *TRUE*, liefert der *EVERY*–Operator den Wert *TRUE*, sonst *FALSE*.
Für den Fall, daß der Ausdruck der *IN*–Klausel eine leere Liste als Ergebnis hat, geben beide Operatoren den Wert *FALSE* zurück.
Bsp.: Gib alle Namen derjenigen Studenten aus, die alle Fächer mit einer Zensur besser als zwei abgeschlossen haben.

```
FOR $a IN document("semester2000.xml")//student
WHERE EVERY $b IN $a/faecher SATISFIES $b/pruefung < 2
RETURN $a/name
```

3.8 Datentypen

Das Typsystem von XQuery basiert auf *XML Schema*. So können alle *XML Schema* Datentypen in den XQuery–Anfragen benutzt werden.

Die Namen von Datentypen erscheinen nur in Funktionsdeklarationen (Rückgabewert, Funktionsparameter), *CAST*- und *TREAT*–Ausdrücken sowie als Operand eines *INSTANCEOF*–Operators. Bestimmte literale Datentypen von *XML Schema* werden von XQuery erkannt. Das betrifft die Typen *xsd:string* (Bsp.-Literal: "Test"), *xsd:integer* (Bsp.-Literal: 10), *xsd:decimal* (Bsp.-Literal: 3.5), *xsd:float* (Bsp.-Literal: 1.75E-2). Literale anderer *XML Schema* Datentypen können mit einer Art Konstruktorfunktion erzeugt werden, z.B. *true()—false()* für boolsche Werte oder *date("19.10.1971")* für ein Datum. Eine komplette Liste der Konstruktorfunktionen ist in [10] zu finden.

3.9 Funktionen

Syntax:

```
FunctionDefn ::= "define" "function" QName "(" ParamList? ")"
                      ("returns" Datatype)? EnclosedExpr
ParamList    ::=   Param ("," Param)*
Param        ::=   Datatype? Variable
FunctionCall ::=   QName "(" (Expr ("," Expr)*)? ")"
```

XQuery bietet eine Standardbibliothek für *build-in* Funktionen. Eine Funktion wurde bereits in allen benutzten Beispielen verwendet, die Funktion *document(string)*. Die Tabelle 3.2 gibt eine Übersicht über die wichtigsten *build-in* Funktionen: Eine vollständige Liste der XQuery Funktionsbibliothek ist in [10] zu finden.

Zusätzlich zu den *build-in* Funktionen hat der Anwender die Möglichkeit, seine eigenen Funktionen zu definieren. Eine Funktions Definition enthält den Namen der Funktion, die Namen und die Datentypen der Funktionsparameter sowie den Datentyp des Rückgabewertes. Außerdem

Funktion	Semantik
document(String)	liefert das *root*–Element eines Dokumentes
avg(Sequenz)	ermittelt den Durchschnitt von Werten
sum(Sequenz)	ermittelt die Summe von Werten
count(Sequenz)	ermittelt die Anzahl von Werten
min(Sequenz)	ermittelt das Minimum von Werten
min(Sequenz)	ermittelt das Maximum von Werten
empty(Sequenz)	liefert *TRUE*, wenn die Sequenz leer ist

Tabelle 3.2: Funktionen der *XQuery Standardbibliothek*

muß der *Function Body* definiert werden. In diesem wird definiert, wie das Ergebnis der Funktion aus den Parametern berechnet wird. Bei einem Funktionsaufruf müssen die übergebenen Parameter gültige Instanzen der definierten Datentypen sein. Gleiches gilt für den von der Funktion zurückgegebenen Wert der Funktion.
Wenn ein Funktionsparameter durch einen Namen aber ohne Typ deklariert wurde, wird der *default* Wert (irgendein Knoten) angenommen. Wird in einer Funktionsdefinition die *RETURNS*–Klausel nicht deklariert, wird erwartet, daß der Rückgabewert eine Sequenz von Knoten ist.

Eine Funktionsdefinition kann auch rekursiv sein. In diesem Fall referenziert die Funktion ihre eigene Definition. Ebenfalls erlaubt sind gegenseitig rekursive Funktionen, in deren *bodies* sich die Funktionen gegenseitig referenzieren. Das nächste Beispiel enthält eine rekursive Funktion, die die Tiefe einer Elementhierachie berechnet:

```
NAMESPACE xsd = "http://www.w3.org/2001/XMLSchema"

DEFINE FUNCTION depth($e) RETURNS xsd:integer
{
```

```
        IF (empty($e/*)) THEN 1
        ELSE max(depth($e/*))+1
}
depth(document("bib.xml"))
```

In diesem Beispiel wurde für den Parameter der Funktion kein Typ definiert. So wird von der Funktion jeder Knoten als Parameter akzeptiert. Die Funktion ermittelt dann rekursiv über alle Kindsknoten die Tiefe des Elementes und wählt anschließend den größten *integer* Wert der Funktionsrückgabewerte aus.

3.10 Benutzerdefinierte Datentypen

Benutzerdefinierte Datentypen sind fester Bestandteil von *XML Schema*. In XQuery kann jeder in *XML Schema* definierte Datentyp als XQuery Datentyp, zusätzlich zu den einfachen und abgeleiteten *XML Schema* Datentypen, referenziert werden. Ebenso können in *XML Schema* definierte *Qualified Names* für Datentypen oder Elemente in XQuery benutzt werden.
Werden in einer XML–Anfrage Element- oder Datentypnamen referenziert, müssen diese in *XML Schemata* definiert sein. Diese müssen von Dokumenten, die in einer Query benutzt werden, referenziert sein oder explizit in einer *NAMESPACE*–Definition und in einer *SCHEMA*–Deklaration referenziert werden.
Ein Beispiel dazu könnte folgendermaßen aussehen:
Zuerst wird der später zu verwendende *XML Schema* Typ "studenten" im *namespace* "http://www.abc.de/Studenten" definiert.

```
<?xml versioin="1.0">
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.abc.de/Studenten">
  <complexType name="studenten">
```

```
    <sequence>
      <element name="student" minOccurs="0"
              maxOccurs="unbounded">
        <complexType>
          <element name="name" type="string"/>
                .
                .
                .
        </complexType>
      <element name="semester" minOccurs="0"
       maxOccurs="unbounded">
        <complexType>
          <element name="jahr" type="integer"/>
          <element name="studentenzahl" type="integer"/>
      </element>
    </sequenz>
  </complexType>
</schema>
```

Anschließend wird eine Funktion definiert, die den eben definierten Datentyp benutzen soll. Die Funktion "kategorisieren(studenten)" soll eine Übersicht der verschiedenen Semester aus einer Menge von Studenten erstellen.

```
DEFAULT NAMESPACE = "http://www.abc.de/Studenten"
SCHEMA "http://www.abc.de/Studenten"
        "http://www.abc.de/schemas/names.xsd"

DEFINE FUNCTION kategorisieren(studenten $studs)
{
 FOR $a IN  distinct($studenten/jahr)
 LET $b := $studenten[ jahr = $a]
```

```
 RETURN
        <semester>
          {$a},
          <studentenzahl> count($b) </studentenzahl>
        </semester>
}
```

Danach kann der Aufruf der Funktion erfolgen:

```
kategorisieren(document("fachbereich_informatik.xml")/studenten)
```

3.11 Operationen auf Datentypen

Syntax:

```
InstanceofExpr ::= Expr "instanceof" "only"? Datatype
TypeSwitchExpr ::= "typeswitch" "(" Expr ")" ("as" Variable)?
                     CaseClause+ "default" "return" Expr
CaseClause     ::= "case" Datatype "return" Expr
CastExpr       ::= (("cast" "as") | ("treat" "as")) Datatype
                    "(" Expr ")"
Datatype       ::= QName
```

Mit dem booleschen Operator *INSTANCEOF* läßt sich überprüfen, ob der linke Operator dem im rechten Operator genannten Namen entspricht. Bei Übereinstimmung wird *TRUE* zurückgegeben, anderenfalls *FALSE*. Das Beispiel `$a INSTANCEOF uni:student` überprüft, ob der an `$a` gebundene Wert dem Typ `uni:person` entspricht oder ein Untertyp von `uni:person` ist. Wenn zusätzlich das Schlüsselwort *ONLY* angegeben wurde, liefert *INSTANCEOF* nur *TRUE*, wenn der erste Operand mit dem spezifizierten Datentyp exakt übereinstimmt.

Mit einem *TYPESWITCH*–Ausdruck kann in XQuery auf dynamische Datentypen reagiert werden. Als statischer Typ wird der Datentyp bezeichnet, der von einem Ausdruck bei einer statischen Analyse einer Query erwartet wird. Bei der Ausführung der Query kann der aktuelle Wert eines Ausdrucks aber auch ein Untertyp des statischen Typs sein. Dieser wird dann dynamischer Typ genannt.
In der *TYPESWITCH*–Klausel wird der Operand spezifiziert, der den statischen Typ darstellt. In den darauf folgenden *CASE*–Klauseln kann definiert werden, wie auf die entsprechenden dynamischen Typen reagiert werden soll.
Unter Verwendung des *TYPESWITCH*–Ausdrucks ist es möglich, eine Art Polymorphismus für Untertypen zu definieren.

Ein weiterer Operator über Datentypen ist der *CAST*–Operator. Damit lassen sich bestimmte Kombinationen von einfachen und abgeleiteten Datentypen ineinander konvertieren. In [10] sind alle unterstützten Konvertierungen angegeben.

Als Erweiterung zum *CAST*–Operator gibt es den *TREAT*–Operator. Anstatt einen Ausdruck von einem Datentyp zum anderen zu konvertieren, wird jetzt der Ausdruck so ausgewertet, als wäre er ein Untertyp des statischen Typs. So wird im folgenden Beispiel der Query Prozessor veranlaßt, die Variable $person, die vom Typ *person* sein soll, als einen Ausdruck des Typs *student* zu behandeln, obwohl dieser eigentlich ein dynamischer Datentyp vom statischen Typ *person* ist.:

```
TREAT AS student($person)
```

Ist jetzt die Variable $person zur Ausführungszeit nicht vom Typ "student", hat dies einen Laufzeitfehler zur Folge.

3.12 Struktur eines Query–Moduls

Syntax:

```
QueryModuleList ::=  QueryModule ( ";" QueryModule)*
QueryModule     ::=  ContextDecl* FunctionDefn* ExprSequence?
ContextDecl     ::=  ("namespace" NCName "=" StringLiteral)
                     | ("default" "namespace" "="
                       StringLiteral)
                     | ("schema" StringLiteral StringLiteral)
Expr            ::=  SortExpr
                     | OrExpr
                     | AndExpr
                     | BeforeAfterExpr
                     | FLWRExpr
                     | IfExpr
                     | SomeExpr
                     | EveryExpr
                     | TypeSwitchExpr
                     | EqualityExpr
                     | RelationalExpr
                     | InstanceofExpr
                     | RangeExpr
                     | AdditiveExpr
                     | MultiplicativeExpr
                     | UnaryExpr
                     | UnionExpr
                     | IntersectExceptExpr
                     | PathExpr
```

Die XML–Anfragesprache XQuery besteht aus ”units”, die Query–Module genannt werden. Ein Query–Modul ist eine unabhängige ”unit”. Es können aber mehrere ”units” zusammen geparst werden, wenn sie durch ein Semikolon voneinander getrennt sind.

3.13 Vergleich mit früheren XML–Anfragesprachen

In diesen Abschnitt wird die Sprache XQuery in den Vergleich von XML–Anfragesprachen aus Abschnitt 2.1 nachträglich einbezogen. Die Beispiele für das Selektieren von Elementen aus einem Dokument, einem Join (*inner join*) über zwei Dokumente sowie die Konstruktion eines bestimmten Ergebnisses sollen auch für XQuery anhand derselben in Abschnitt 2.3 (auf Seite 7) verwendeten Beispieldokumente gezeigt werden.

3.13.1 Syntaxtischer Aufbau

Wie bereits gezeigt, besteht die Sprache XQuery aus einer Vielzahl verschiedener Ausdrücke, welcher jeder für sich schon eine XQuery–Anfrage darstellen. Im Allgemeinen ist aber bei einer sinnvollen Anfrage an Dokumente eine Iteration über Mengen von Elementen (im Fall von XQuery über Sequenzen von Elementen) notwendig. In diesem Sinne bildet der *FLWR*–Ausdruck von XQuery den Kern dieser Anfragesprache. Deshalb soll jetzt die Syntax nur für den *FLWR*–Ausdruck gezeigt werden.

```
FLWRAusdr        := (ForKlausel | LetKlausel) + WhereKlausel?
                    "RETURN" Ausdr
ForKlausel       := "FOR" Variable "IN" Ausdr
                    ("," Variable "IN" Ausdr)*
```

```
LetKlausel      := "LET" Variable ":=" Ausdr
                   ("," Variable ":=" Ausdr)*
WhereKlausel    := "WHERE" Ausdr
```

In der gezeigten Syntaxdefinition steht der Term "Ausdr" für einen beliebigen XQuery–Ausdruck, dieser kann beispielsweise wieder ein *FLWR*–Ausdruck sein.

3.13.2 Selektion

Die Selektion wird in XQuery unter Verwendung von Pfadausdrücken [6] durchgeführt. Am Anfang eines Pfadausdrucks wird immer der Knoten identifiziert. Bei der Auswahl eines Dokumentes aus der Datenbank wird mit Hilfe der Funktion "document(String)" der Dokumentknoten selektiert. Unter Verwendung von einem oder mehreren *Location Steps* kann, beginnend mit dem *root*–Knoten, innerhalb eines Dokuments navigiert werden.

```
document("www.nhsc/manufacturer.xml")/manufacturer[model/rank<=10]
```

Wie zu sehen ist, ist dieser Pfadausdruck mit dem von *XQL* identisch. Wie bereits erwähnt, bedient sich *XQuery* vieler nützlicher Features anderer XML–Anfragesprachen. Hier ist ein erstes Beispiel dafür zu sehen, XQuery hat die Pfadausdrücke zur Selektion von Elementen aus XML–Dokumenten von *XQL* übernommen.

3.13.3 Join

Joins werden in XQuery unter Verwendung von Variablenbindungen durchgeführt. Zuerst werden in der *FOR*–Klausel die gewünschten Elemente aus den verschiedenen Dokumenten an Variablen gebunden. In der *WHERE*–Klausel wird das Ergebnis gefiltert und in der abschließenden *RETURN*–Klausel konstruiert.

```
FOR $a IN document("manufacturers.xml")/manufacturer,
    $b IN document("vehicles.xml")/vehicle
WHERE $a/mn_name = $b/make AND
      $a/model/mo_name = $b/model AND
      $a/year = $b/year
RETURN  <manufacturer>
                $a/mn_name,
                $a/year,
           <vehiclemodel>
                $a/model,
                $b
           </vehiclemodel>
        </manufacturer>
```

3.13.4 Umstrukturierung

Wie schon gezeigt, besitzt XQuery einen Ausdruck zur Generierung neuer Elemente, den *Elementkonstruktor*. Es ist somit kein Problem, die geforderte Aufgabenstellung zu erfüllen. In der *FOR*–Klausel wird der *root*–Knoten des Dokuments "result.xml" an die Variable $a gebunden. In der *RETURN*–Klausel werden die gewünschten Elemente mittels *Elementkonstruktoren* erzeugt und durch Iteration über die Variable $a die Informationen aus dem Dokument selektiert.

```
FOR $a IN document("result.xml")
RETURN  <car>
          <make>$a/mn_name/TEXT()</make>,
          <mo_name> $a/vehiclemodel/model/mo_name/TEXT()
  </mo_name>,
          <vendor> $a/vehiclemodel/vehicle/vendor/TEXT()
  </vendor>,
```

```
        <rank> $a/vehiclemodel/model/rank/TEXT() </rank>,
        <price> $a/vehiclemodel/vehicle/price/TEXT()
</price>
      </car>
```

Kapitel 4

Konzeption für die Umsetzung von XQuery

Beginnen soll das Kapitel mit einer detaillierten Darstellung der Gesamtaufgabe. Anschließend sollen die Konzepte vorgestellt werden, auf denen die Implementierung von XQuery basiert.

4.1 Der Prozess der Anfragebearbeitung

Der Prozess der Anfragebearbeitung stellt eine typische *2–Tear*–Architektur dar. Der Benutzer sitzt an einem *Client* und übergibt diesem seine XQuery–Anfrage. Der *Client* stellt als Gesamtheit die *Query–Engine*, die den Auswertungsprozeß steuert, dar. Diese kontaktiert im Verlauf der Anfrageauswertung einen Datenbankserver, in dem die XML–Dokumente gespeichert sind, um die gewünschten Daten anzufordern. Wurde die XQuery–Anfrage erfolgreich ausgewertet, wird von der *Query–Engine* ein Ergebnis erzeugt und am *Client* ausgegeben. Die folgende Abbildung 4.1 soll den Sachverhalt verdeutlichen.

Eine XQuery–Anfrage ist im eigentlichem Sinne nur eine Folge von Zeichen.

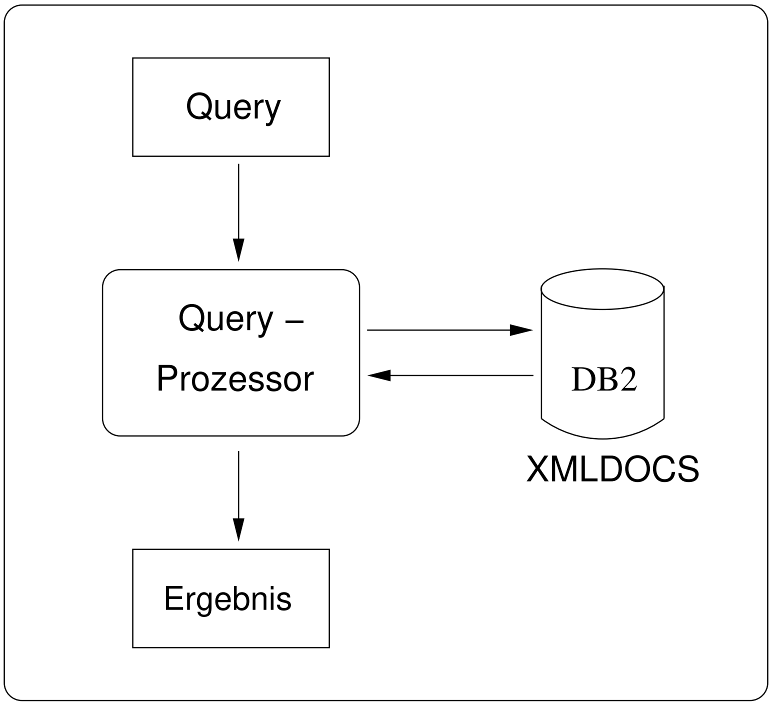

Abbildung 4.1: Konzeptionelle Übersicht des Datenflusses bei einer Query–Auswertung

Die Auswertung dieser Zeichenfolge läßt sich in verschiedene Teilaufgaben gliedern:

1. Aufspaltung der Zeichenfolge in *Token* in der *Lexikalischen Analyse*
2. Überprüfung, ob die Query Element der Sprache XQuery ist in der *Syntaktischen Analyse*
3. Die Auswertung der Query in der *Semantischen Analyse*
4. Ergebniserzeugung

Diese verschiedenen Teilaufgaben werden von unterschiedlichen Modulen erledigt. In der Abbildung 4.2 ist diese Aufgliederung grafisch dargestellt. Zur vollständigen Übersicht der benutzten Module wird auch der in der Vorbereitung abgewickelte Arbeitsschritt der Speicherung von XML–Dokumenten mit aufgeführt.

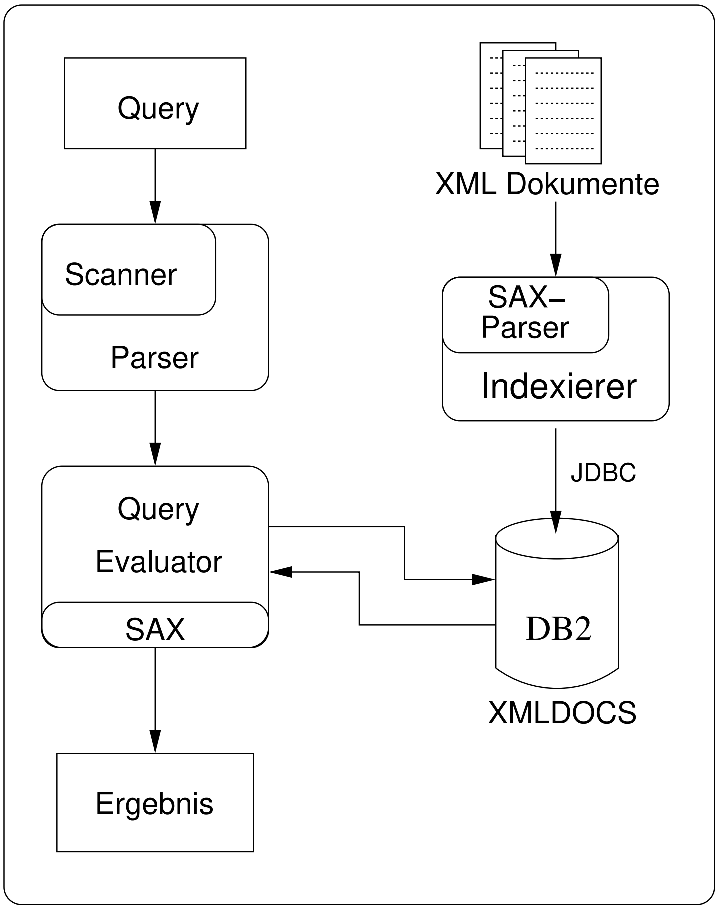

Abbildung 4.2: Module der Anfragebearbeitung

Im folgenden soll die Bearbeitung einer Anfrage näher betrachtet werden. Dabei kann nicht strikt zwischen den einzelnen Schritten getrennt werden, wie bei der obigen Aufzählung, da der Prozeß der Anfragebearbeitung eine ineinander verzahnte Ausführung der einzelnen Schritte ist.

Zu Beginn der Auswertung der Query muß die Zeichenfolge, die die XQuery–Anfrage repräsentiert nach Schlüsselwörtern, genannt *Token*, durchsucht werden, die in der Zeichenfolge enthalten sind. Dieser Arbeitsschritt wird *Lexikalische Analyse* genannt und das ausführende Werkzeug, in dem die Token definiert werden, ist der *Scanner*.

Bei der lexikalischen Analyse wird eine XQuery–Anfrage in einzelne *Token* zerlegt. Diese werden an den Parser zur syntaktischen Analyse übergeben. Im Parser wird die zu der Anfragesprache XQuery gehörende Grammatik in Form von Regeln definiert. Der Parser überprüft, ob die angegebene Query der XQuery–Grammatik genügt. Dabei erzeugt er einen Syntaxbaum unter Verwendung von semantischen Aktionen, die innerhalb der Grammatikregeln des Parsers definiert wurden. Wenn keine Regel existiert, nach der die Query aufgelöst werden kann, bricht der Parser mit einer Fehlermeldung ab. Durchläuft eine Query den Parser erfolgreich, gibt dieser einen Syntaxbaum zurück, der die gesamte XML–Anfrage repräsentiert.

Der Syntaxbaum wird mit Hilfe von Objekten generiert, die ganze XQuery–Ausdrücke oder Teile von Ausdrücken darstellen. Der vom Parser generierte Syntaxbaum ist demzufolge auch ein einzelnes Objekt (die Wurzel des Syntaxbaumes), der genau einen Ausdruck repräsentiert. Zur Evaluation der Query wird das Objekt, welches die Query darstellt, ausgewertet. Dazu wird in allen Objekten eine Funktion "eval()" implementiert.

In der Query–Auswertung gibt es genau zwei Möglichkeiten, Knoten im Ergebnis einer Query aufzunehmen. Eine Alternative ist die Generierung neuer XML–Knoten durch den Ausdruck des *Elementkonstruktors*. Eine andere ist die Aufnahme vorhandener XML–Knoten aus XML–Dokumenten durch Selektion unter Verwendung von Pfadausdrücken.

Einer der wichtigsten Ausdrücke von XQuery ist der *FLWR*–Ausdruck. Die Implementierung dieses Ausdrucks soll deshalb näher erläutert werden.

Zentraler Bestandteil des *FLWR*–Ausdrucks ist ein Baum, der sämtliche in einer Query gebundenen Tupel enthält, er soll hier "Tupelbaum" genannt werden. Dieser Tupelbaum wird in der *FOR* | *LET*–Klausel eines *FLWR*–Ausdrucks generiert, indem alle dort vorkommenden Ausdrücke ausgewertet werden und die resultierenden Knoten an definierten Variablen gebunden werden. Der Tupelbaum repräsentiert dann eine auf dem *FLWR*–Ausdruck bezogene Umgebung, die zur weiteren Modifizierung bzw. Filterung der *WHERE*– und *RETURN*–Klausel übergeben wird. In der *WHERE*–Klausel werden die Tupel des Tupelbaumes nach bestimmten Bedingungen gefiltert. In der anschließenden *RETURN*–Klausel wird dann unter Verwendung des Tupelbaumes und weiterer XQuery–Ausdrücke das Ergebnis des *FLWR*–Ausdrucks generiert.

4.2 Ein Speichermodell für XML–Dokumente

Ziel der Arbeit soll es sein, die Anfragesprache XQuery für einen möglichst universellen Einsatz zu implementieren. Die Umsetzung der Anfragesprache erfordert ein Speichermodell für XML–Dokumente, das es gestattet, möglichst jedes XML–Dokument in die Datenbank aufzunehmen, zumindest jedes "wohlgeformte". Das bedeutet, daß eine Menge von Dokumenten, die unter Umständen alle paarweise verschiedenen DTDs genügen, in ein und demselben Datenbankschema gespeichert werden. Diese Art von Datenbankschemata sollen hier als "generische" Datenbankschemata bezeichnet werden. In ihnen können alle Typen bezüglich einer DTD von XML–Dokumenten gespeichert werden. Dieser Vorteil wird jedoch mit dem Nachteil erkauft, daß die vorliegenden Strukturinformationen in Form von DTDs nicht gespeichert werden. Das bedeutet, daß diese Informationen verloren gehen und somit für die spätere Anfrageauswertung nicht zur Verfügung stehen.
Um auf die in einer Datenbank gespeicherten Dokumente eine XML–Anfrage auszuführen, muß es möglich sein, innerhalb des Dokumentes zu navigieren und einzelne Knoten des Dokumentes zu selektieren. Das Datenbankschema soll diese Kriterien direkt unterstützen.

Zum Thema der Speicherung von XML–Dokumenten gibt es mehrere wissenschaftliche Arbeiten, die verschiedene Ansätze verfolgen. So wird in [16] aus einem XML–Dokument ein indexierter Baum generiert. Die Kanten des Baumes repräsentieren die Elementknoten. Die Baumknoten werden aufsteigend numeriert und im Falle eines Blattes durch einen Textknoten dargestellt. Aus diesem Baum wird dann das Datenbankschema erzeugt, und zwar für jeden Elementknoten eine Relation. In diesen Relationen werden "Eltern–Kind–Beziehungen" der Kanten sowie die Dokumentordnung der Knoten gespeichert.
Ebenso gibt es bereits kommerzielle Produkte wie den *DB2 XML Extender* von IBM, die eine Speicherung von XML–Dokumenten ermöglichen.
Eine Methode der Speicherung, die den geforderten Kriterien entspricht, ist die von Shimura [5]. Bei diesem Verfahren wird der Inhalt eines XML–Dokumentes indexiert und anschließend in ein fest definiertes Datenbankschema eingefügt. Das Datenbankschema und die Indexierung des Dokumentes sollen in den folgenden Abschnitten erklärt werden.

4.2.1 Das DB–Schema

Die Methode von Shimura [5] zur Indexierung von XML–Dokumenten wurde bereits in der Implementierung der XML–Anfragesprache *Quilt* (einem Vorgänger von XQuery) erfolgreich eingesetzt.
Das Datenmodell eines indexierten Dokumentes soll zunächst mit Hilfe eines *Entity–Relationship(ER)*–Diagramms in Abbildung 4.3 modelliert werden. Die Abbildung des ER–Diagramms auf das relationale Datenmodell resultiert in folgendem Datenbankschema:

```
documents(docID,docName)
path(docID,pathID,pathexpression)
element(docID,pathID,wi,ei,lwi,lei)
text(docID,pathID,value,wi,lwi)
```

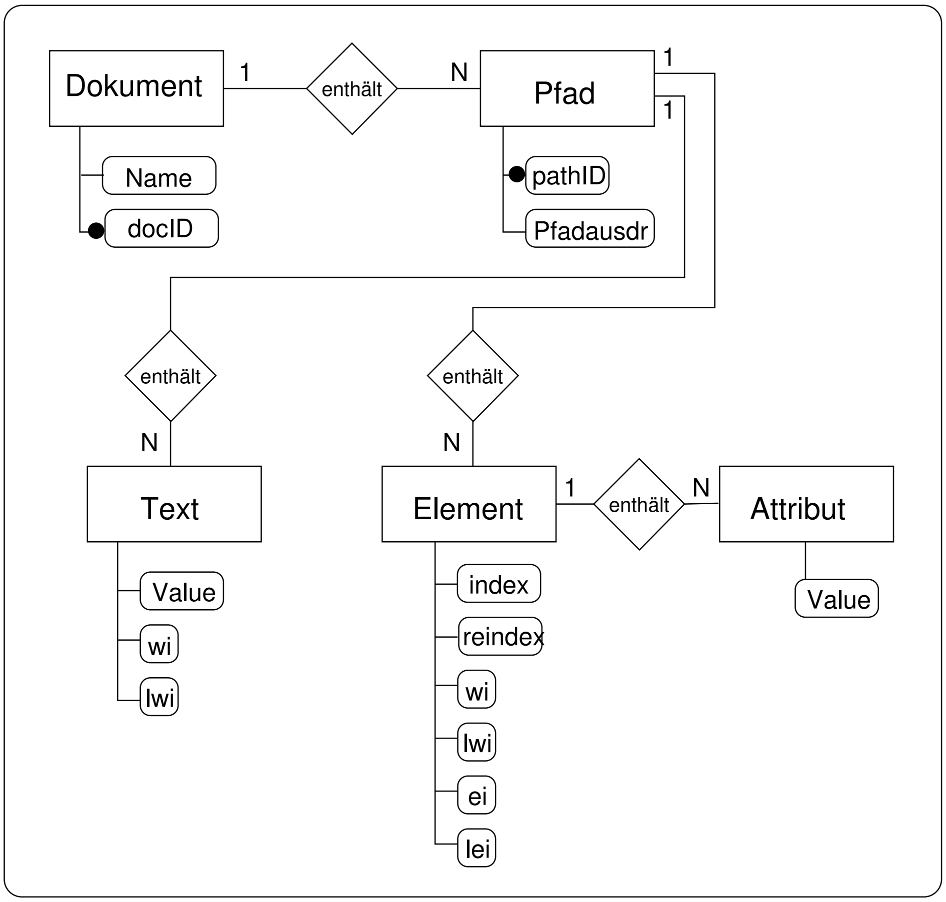

Abbildung 4.3: Modellierung der Daten als ER–Diagram

```
attribute(docID,pathID,attvalue,wi,ei,lwi,lei)
```

In der Relation "documents" werden die Namen der XML–Dokumente gespeichert. Den Namen wird ein Dokument–Identifikator zugeordnet, welcher als Fremdschlüssel in allen anderen Relationen dient.
Die Relation "path" speichert alle in einem Dokument vorkommenden Pfadausdrücke. Auch diese werden durch eine eindeutige *integer*–Zahl indentifiziert und dienen in den folgenden Relationen als Fremdschlüssel.
Die Tabelle "Element" dient zur Speicherung der Elementknoten. Die Zugehörigkeit der Elemente zu Pfadausdrücken und XML–Dokumenten geschieht über die Attribute "docID" und "pathID". Die restlichen Attribute

bestimmen die Position des Elementes innherhalb des Dokumentes. Wie diese bestimmt wird, wird im Abschnitt 4.2.2 gezeigt.
Die Relationen "text" und "attribute" speichern die Textknoten und Attributknoten von XML–Dokumenten. In ihnen werden die Indizes zur eindeutigen Dokumentzugehörigkeit, Indizes zur Bestimmung der Position innherhalb von Dokumenten sowie ihr Wert selbst gespeichert.

4.2.2 Die Kodierung des Dokumentes

Die Kodierung der XML–Dokumente soll jetzt an einem ausgesuchten Beispiel erfolgen. Das Beispiel ist ein Teil des XML–Dokuments *The Merchant of Venice* von William Shakespeare.

```
<PLAY author="Shakespeare">
<TITLE> The Merchant of Venice </TITLE>
<PERSONAE>
<TITLE> Dramatis Personae </TITLE>
<PERSONA> The DUKE OF VENICE </PERSONA>
```

Jedes Wort des XML–Dokumentens wird auf einen ganzzahligen Wert abgebildet, beginnend mit der Zahl "1". Als Worte sind dabei nur die Worte gemeint, die zu Textknoten gehören. Worte, die die Attributenwerte bilden, werden nicht indiziert.
Nach der Anwendung dieses Kodierungsschrittes ist das Dokument indiziert und hat folgendes Aussehen:

<PLAY author="Shakespeare">

<TITLE>$The^1\ Merchant^2\ of^3\ Venice^4$</TITLE>

<PERSONAE>

<TITLE>$Dramatis^5\ Personae^6$</TITLE>

<PERSONA>$The^7\ DUKE^8\ OF^9\ VENICE^{10}$</PERSONA>

Nach dieser Indizierung kann jeder Knoten des Dokumentes durch eine Referenz auf das Dokument und einen gültigen Pfad, beginnend von der Wurzel, eindeutig indentifiziert werden. Dabei ist jeder Knoten als ein Paar von *integer*–Werten *(start, ende)*, welche Indizes des Dokumentes sind, definiert. Bei Textknoten beziehen sich *start* und *ende* auf den Index des ersten und des letzten Wortes. Bsp.:

<TITLE>$The^1\ Merchant^2\ of^3\ Venice^4$</TITLE>

Hier ist der Textknoten durch das Paar (1,4) kodiert.

Elementknoten werden anders kodiert. Dabei ist *start* als Paar definiert. Der erste Wert des Paares ist der Index des ersten Textknotens (oder 0, wenn noch kein Textknoten vor dem Elementknoten existiert), der vor dem öffnenden Knoten erscheint. Dieser Wert wird als *Word Index* bezeichnet. Der zweite Wert des *start*–Paares bestimmt die Ordnung des Erscheinens des öffnenden Knotens. Dieser Wert wird als *Element Index* bezeichnet. Zur Veranschaulichung soll jetzt das zweite *<TITLE>* Element aus unserem Beispiel nach diesem Verfahren indiziert werden:

$...Venice^4$</TITLE><PERSONAE><TITLE>

Der *Word Index* für das öffnende *<TITLE>* Element ist 4. Das Element ist das dritte in der Reihe der aufeinanderfolgenden Elementknoten (*Element Index*). Daraus ergibt sich als *start*–Wert für das <TITLE> Element (4,3). Der *end*–Wert der Elementkodierung ist symmetrisch zum *start*–Wert definiert.

Attributknoten werden wieder anders kodiert, da ihr Anfang und Ende nicht verschieden sind. Deshalb erhalten der *start*- und *end*–Wert dieselbe Kodierung. Die beiden Paare erhalten den selben Wert, den *start*–Wert des dazugehörigen Element–Knotens.

4.2.3 DB2 UDT im DB–Schema

Das eben vorgestellte Datenbankschema soll jetzt um Typen erweitert werden. Die Erweiterung soll sich auf Spaltentypen beschränken, so daß die Textknoten sowie die Attributknoten eines XML–Dokuments nach ihrem Typ unterschieden werden können.
Als Folge können binäre Operationen zwischen Pfadausdrücken und einer Konstanten direkt auf *SQL* abgebildet werden. Abhängig vom Typ der Konstanten wird die Operationen auf verschiedene *SQL*–Konstrukte abgebildet. Das folgende Beispiel zeigt zwei binäre Operationen, in denen die konstanten Operanden verschiedenen Typs auf unterschiedliche *SQL*–Anfragen abgebildet werden. Wie die Abbildung aussieht, wird im Abschnitt 4.3 erklärt:

```
$a/alter > 30
```

und

```
$a/vorname = 'Gudrun'
```

Die von der Datenbank erhaltenen Ergebnisse der *SQL*–Anfragen sind somit gleich dem Ergebnis der binären Operation. Vorteil dieser Abbildung ist, daß die Filterung der Ergebnismenge auf der Datenbankseite stattfindet und nicht sämtliche (auch überflüssige) Daten auf den *Client* übertragen werden, die dann auf dem *Client* verglichen werden müßten.
In dieser Implementierung soll dabei nur zwischen Zeichenketten vom Typ *string* und ganzzahligen Werten vom Typ *float* unterschieden werden. Der Typ der reellen Zahlen soll hier ein Untertyp des Typs Zeichenkette sein.
Für diese neuen komplexen Typen müssen sogenannte *DB2 Transform Functions* definiert werden. Sie sollen die üblichen *Selektions-* und *Änderungsoperationen* auf den *UDTs* ermöglichen. Die Implementierung der *DB2 Transform Functions* wird im Abschnitt 5.1 erläutert.

4.3 Die Abbildung auf SQL

Bei der Anfragebearbeitung werden *SQL*–Queries ausschließlich bei der Auswertung von Pfadausdrücken erzeugt. Die Evaluation eines Pfadausdrucks wird schrittweise durchgeführt. Beginnend mit dem ersten Schritt des Pfadausdruckes wird jeder *Location Step* einzeln ausgewertet. Die Ergebnisknoten eines *Location Steps* werden in der *Context List* gespeichert.
Bei der Auswertung eines einzelnen Schrittes wird für jeden Knoten aus der *Context List* eine *SQL*–Query an die Datenbank geschickt. Die Ergebnisse dieser Datenbankanfrage bilden die Knoten der neuer *Context List* und dienen als Startknoten zur Auswertung des nächsten *Location Steps* des Pfadausdrucks. Nach jedem Schritt wird die aktuelle Knotenmenge in der *Context List* gespeichert. Die jeweiligen *SQL*–Abbildungen ergeben sich aus den im *Location Step* benutzten Achsenidentifkatoren und dem als *Name Test* spezifizierten Namen (*QName*).
Während der Auswertung von Pfadausdrücken werden verschiedene *SQL*–Abbildungen erzeugt. Sie lassen sich kategorisieren in:

1. *SQL*–Abbildungen zur Selektion des Dokumentknotens
2. *SQL*–Abbildungen der Achsenidentifikatoren und *Name Test*
3. *SQL*–Abbildungen von *Name Test* mit Prädikat
4. *SQL*–Abbildungen, die *UDTs* berücksichtigen

Im den folgenden Abschnitten sollen die verschiedenen *SQL*–Abbildungen erläutert werden. Die Abbildungen der Achsenidentifikatoren beschränken sich dabei auf die *Child*- und *Desendant*–Achse. Zum besseren Verständnis werden die Abbildungen jeweils durch ein entsprechendes "Template" dargestellt und danach der *SQL* angegeben.

4.3.1 Die Abbildung zur Selektion des Dokumentknotens

Die Selektion des *root*–Knotens eines Dokuments (Dokumentknoten) repräsentiert den Einstiegsknoten in ein Dokument. Einziger Parameter für die Abbildung ist der Name des Dokumentes. Dieser kann in der Datenbank genau einmal vorkommen.
Der Dokumentknoten wird nur durch eine einzige *integer*–Zahl repräsentiert (siehe Abschnitt 5.2.5), diese wird durch die folgende *SQL*–Query selektiert:

```
SELECT docID FROM document where name = DocName
```

4.3.2 Die Abbildung der *Child*–Achse

Aufgrund der Kodierung eines XML–Dokuments lassen sich die *Child*–Knoten eines aktuellen Knotens exakt bestimmen. Die *Child*–Knoten sind genau die Knoten, die eine Hierachie unter dem aktuellen Knoten liegen und vom Kontextknoten (der aktuelle Knoten) eingeschlossen sind. Bei der Selektierung von *Child*–Knoten werden nur Elementknoten selektiert, da nur diese für die weitere Auswertung interessant sind.

Startknoten *Document Node***:** Es wird eine spezielle Abbildungen benötigt, wenn der Startknoten für einen *Location Step* ein *Document Node* (dieser ist kein Elementknoten) ist. Seine Kindsknoten haben alle den *Wordindex*=0, da dieser nur Elementknoten und keine Textknoten als *Child* haben kann. Und sie haben einen *Elementindex*¡=2, da die Kindsknoten nicht innerhalb eines anderen Elementes liegen können und so maximal der schließende Knoten des vorhergehenden Knoten erscheinen kann.

```
-- Template --:
'select' * 'from' elements elem, path pfad
'where' pfad = "/" + NameTest.QNAME
```

```
        'and'  elem.istKindVon("DokumentKnoten")

-- SQL-Code --:
SELECT * FROM element e, path p
WHERE e.pathID=p.pathID AND e.docID= node.getDOCID()
        AND p.docID=e.docID AND e.wi = 0 AND e.ei <= 2
        AND p.pathexpr LIKE '/' + NodeName
        ORDER BY e.wi, e.ei
```

Für den Fall, daß der *Name Test* der *Wildcard*–Identifikator "*" ist, wird die Abbildung folgendermaßen verkürzt. Jetzt werden alle *Child*–Knoten selektiert, ohne Beachtung des *Name Test*:

```
-- Template --:
'select' * 'from' elements elem, path pfad
'where' elem.istKindVon("StartKnoten")

-- SQL-Code --:
SELECT * FROM element e, path p
WHERE e.pathID=p.pathID AND e.docID= node.getDOCID()
        AND p.docID=e.docID AND e.wi = 0 AND e.ei <= 2
        ORDER BY e.wi, e.ei
```

Startknoten *Element Node***:** Ist der auszuwertende Knoten ein Elementknoten, müssen seine *Child Nodes* innerhalb seiner Parameter – die Attribute *Wordindex, Lastwordindex* sowie sein Pfadausdruck – liegen.

```
-- Template --:
'select' * 'from' elements elem, path pfad
'where' pfad = "StartKnoten.Pfad" + NameTest.QNAME
        'and'  elem.istKindVon("StartKnoten")
```

```
-- SQL-Code --:
SELECT * FROM element e, path p
WHERE   e.pathID=p.pathID AND e.docID=node.getDOCID()
        AND p.docID=e.docID
        AND p.pathexpr LIKE  node.getPATHEXP() + '/%'
        AND p.pathexpr NOT LIKE node.getPATHEXP() + '/%/%'
        AND p.pathexpr LIKE '%/' + NodeName
        AND e.wi >= node.getWI() AND e.lwi<= node.getLWI()
        ORDER BY e.wi, e.ei
```

4.3.3 Die Abbildung der *Descendant*–Achse

Im Unterschied zur *Child*–Achse werden auf der *Descendant*–Achse alle Knoten unterhalb (alle Hierachien) des aktuellen Knotens selektiert, die der Bedingung des *Name Test* genügen.

Startknoten *Document Node***:** Beim Dokumentknoten müssen die Nachkommen einen *Wordindex* größer gleich 0 haben und der *Elementindex* kann jetzt auch beliebige Werte größer gleich 1 haben. Sie müssen lediglich dem *Name Test* genügen und als Elementnamen "NodeName" besitzen.

```
-- Template --:
'select' * 'from' elements elem, path pfad
'where' pfad = "/" + NameTest.QNAME
        'and'  elem.istNachfolgerVon("Dokument")


-- SQL-Code --:
SELECT * FROM element e, path p WHERE e.pathID=p.pathID AND
        e.docID= node.getDOCID() AND p.docID=e.docID AND
        e.wi >= 0 AND e.ei >= 1 AND
        p.pathexpr LIKE '/' + NodeName
```

```
        ORDER BY e.wi, e.ei
```

Startknoten *Element Node***:** Im Gegensatz zum Dokumentknoten müssen im Fall des Elementknotens die Nachkommen des aktuellen Knotens innerhalb der Grenzen von *Wordindex* und *Lastwordindex* liegen.

```
-- Template --:
'select' * 'from' elements elem, path pfad
'where' pfad = "StartKnoten.Pfad"+NameTest.QNAME
        'and'  elem.istNachfolgerVon("StartKnoten")


-- SQL-Code --:
SELECT * FROM element e, path p
WHERE   e.pathID = p.pathID AND
        e.docID = node.getDOCID() AND e.docID=p.docID
        AND p.pathexpr LIKE node.getPATHEXP() + '/%' AND
        e.wi >=  node.getWI() AND e.lwi <= node.getLWI()
        AND p.pathexpr LIKE '%/' + NodeName
        ORDER BY e.wi, e.ei
```

4.3.4 Die Abbildung von Prädikaten

In den oben gezeigten *SQL*–Abbildungen werden alle Elementknoten mit dem entsprechenden *Name Test* selektiert.
Da soviel Arbeit wie möglich auf der Datenbankseite abgewickelt werden soll, werden (soweit möglich) auch Prädikate von *Location Steps* direkt als Selektionbedingungen auf *SQL* abgebildet. Diese Abbildungen sind nur bei *Location Step*–Prädikaten der Form

```
 [n] , n = natürliche Zahl.
```

möglich. Das Datenbankschema unterstützt diese Art der Abbildung, da die Ordnungszahl eines Elementes direkt im Schema gespeichert ist.

Das folgende Beispiel zeigt eine *Child*–Query. Die Prädikatabbildung bezieht sich auf die letzte Zeile.

```
-- Template --:
'select' * 'from' elements elem, path pfad
'where' pfad = "StartKnoten.Pfad"+NameTest.QNAME
        'and'  elem.istKindVon("StartKnoten")
        'and'  elem.hatOrdnung(n)


-- SQL-Code --:
SELECT * FROM element e, path p
WHERE   e.pathID=p.pathID AND e.docID=node.getDOCID()
        AND p.docID=e.docID
        AND p.pathexpr LIKE  node.getPATHEXP() + '/%'
        AND p.pathexpr NOT LIKE node.getPATHEXP() + '/%/%'
        AND p.pathexpr LIKE '%/' + NodeName
        AND e.wi >= node.getWI() AND e.lwi<= node.getLWI()
        AND e.index= n
```

4.3.5 Berücksichtigung von DB2 UDTs bei der Abbildung

Die im Datenbankschema berücksichtigten *DB2 User Defined Types* (UDTs) können bei Abbildung von binären Operationen, in denen ein Operand ein Pfadausdruck und der zweite Operand eine Konstante ist, unterstützt werden. Die Selektionsbedingugn, die durch den Vergleich ausgedrückt wird, kann unter Verwendung der UDTs direkt in der *SQL*–Abbildung des *Locations Steps* berücksichtigt werden.
Bei den Konstanten kann zwischen den einfachen Datentypen *float* und *string* unterschieden werden. Diese Form von Vergleichen tritt oft in der *WHERE*–Klausel eines *FLWR*–Ausdrucks oder als Prädikat innheralb von Pfadausdrücken auf.

Das folgende Beispiel zeigt eine *Child*–Query, welche alle *Child*–Elemente liefert, die zusätzlich zum "NodeName" einem Vergleich mit einer Konstanten vom Typ *float* genügen. Der Platzhalter "Operator" steht dabei für die Operatoren "=", "! =", "<" oder ">".

```
-- Template --:
'select' * 'from' elements elem, path pfad,text text
'where' pfad = "StartKnoten.Pfad"+NameTest.QNAME
        'and'  elem.istKindVon("StartKnoten")
        'and'  text.floatVal.Operator(FloatWert)


-- SQL-Code --:
SELECT * FROM element e, path p,text t
WHERE   e.pathID=p.pathID AND e.pathID=t.pathID
        AND e.docID = node.getDOCID()
        AND e.docID=t.docID AND e.docID=p.docID
        AND p.pathexpr LIKE node.getPATHEXP()+ '/%'
        AND p.pathexpr NOT LIKE node.getPATHEXP()+'/%/%'
        AND p.pathexpr LIKE '%/' + NodeName
        AND e.wi>= node.getWI() AND e.lwi<= node.getLWI()
        AND t.wi> node.getWI() AND t.lwi<= node.getLWI()
        AND t.value IS OF (ONLY intvalue)
        AND floatvaluetofunc(TREAT(t.value AS floatvalue))
    Operator FloatWert
        ORDER BY e.wi, e.ei
```

(`Operator ist Element {<,>,=,!=}`)

Bsp. Pfadausdruck:

document("personen.xml")/person/vorname[alter > 1]

Bsp. *WHERE*–Klausel:

`WHERE $a/vorname/alter > 1`

Ein Vergleich mit einer Konstanten vom Typ *string* wird hingegen folgendermaßen abgebildet:

```
-- Template --:
'select' * 'from' elements elem, path pfad,text text
'where' pfad = "StartKnoten.Pfad"+NameTest.QNAME
        'and'  elem.istKindVon("StartKnoten")
        'and'  text.stringVal.Operator(StringWert)


-- SQL-Code --:
SELECT * FROM element e, path p,text t
WHERE   e.pathID=p.pathID AND e.pathID=t.pathID
        AND e.docID = node.getDOCID()
        AND e.docID=t.docID AND e.docID=p.docID
        AND p.pathexpr LIKE node.getPATHEXP()+ '/%'
        AND p.pathexpr NOT LIKE node.getPATHEXP()+'/%/%'
        AND p.pathexpr LIKE '%/' + NodeName
        AND e.wi>= node.getWI() AND e.lwi<= node.getLWI()
        AND t.wi> node.getWI() AND t.lwi<= node.getLWI()
        AND t.value IS OF (ONLY stringvalue)
        AND stringvaluetofunc(TREAT(t.value AS stringvalue))
        Operator 'Hans Joachim'
        ORDER BY e.wi, e.ei
```

(`Operator ist Element {=,!=}`)
Bsp. Pfadausdruck:

```
document("personen.xml")/person[vorname = "Hans Joachim"]
```

Bsp. *WHERE*–Klausel:

```
WHERE $a/vorname = "Hans Joachim"
```

Kapitel 5

Die Implementierung

Einen erheblichen Teil dieser Arbeit stellen die Implementierung der Anfragesprache selbst sowie andere vorbereitende Aufgaben wie die Kodierung der XML–Dokumente und die Konfiguration der Datenbank dar. In diesem Abschnitt sollen daher die Implementierung der konzeptionellen Überlegungen auf der Datenbank selbst gezeigt werden sowie eine Übersicht über die Mächtigkeit dieser *XQuery*–Implementierung gegeben werden.

5.1 Speicherung in *DB2*

Die Speicherung der XML–Dokumente stellt eine grundlegende Voraussetzung für die Arbeit dar. Dazu müssen einige vorbereitende Aufgaben ausgeführt werden. Am Anfang steht die Definition der in Abschnitt 4.2.3 auf Seite 56 diskutierten *User Defined Types*(UDT) und der damit verbundenen *DB2 Transform Funktions*. Dieser komplexe Prozeß soll jetzt detailliert erklärt werden.

Die Typendefinitionen sind sehr kurz, deshalb sollen hier die nötigen *SQL*–Anweisungen gezeigt werden:

```
create type stringvalue as (stringval varchar(260))
```

```
        MODE db2sql;
create type floatvalue under stringvalue
        as (floatval real) MODE db2sql;
```

Der Typ "stringvalue" ist dabei der Repräsentant von Zeichenketten als XML–Elementinhalt. Der Untertyp *floatvalue* repäsentiert den reellen Datentyp als XML–Elementinhalt.
Im nächsten Schritt werden die Relationen des Datenbankschemas erzeugt. In der Tabellendefinition wird als Spaltentyp für den Attributwert "value" der gemeinsame Obertyp aller Untertypen angegeben. In diesem Fall ist das der Typ *stringvalue*. Die Typen werden als Spaltentyp in den Tabellen *Text* und *Attribut* verwendet.

Zur Anwendung von Anfrage- und Änderungsoperationen auf die UDTs werden anschließend die *DB2 Transform Funtions* definiert.
Zuerst soll die *FROM SQL Transform Function* beschrieben werden. Sie wird bei Anfrageoperationen benötigt und soll den komplexen Datentyp in einer gewünschten Form darstellen. Die Definition erfordert mehrere Arbeitsschritte. Zuerst werden für jeden UDT zwei *DB2 USER DEFINED FUNCTIONS*(UDF) definiert, die unter Verwendung der *DB2 Oberserver Functions* die Attribute des UDT an eine andere UDF übergeben soll. Die Funktionen werden "stringvaluetofunc(stringvalue)" und "floatvlauetofunc(floatvalue)" genannt. Sie übergeben die Attribute der UDTs an die UDFs "val_to_client(stringvalue)" und "val_to_client(floatvalue)". Diese beiden Funktion rufen wiederum eine in *Java* geschriebene UDF auf, die den *string*–Wert der Attribute zurückgibt, welche den Wert des komplexen Types repräsentieren. Eine weitere UDF, hier "val_stream", ist nötig, die mit Hilfe der beiden Funktionen "val_to_client(stringvalue)" und "val_to_client(floatvalue)" eine virtuelle Tabelle erzeugt, in der die *string*–Werte aller UDTs gespeichert werden. Aus dieser virtuellen Tabelle

wird der gewünschte Textknoten selektiert.
Die Implementierung der *TO SQL Transform Functions*, welche zum Einfügen von UDTs dienen, folgt in gleicher Vorgehensweise. Zuerst werden die UDFs "functostringvalue(varchar(260))" und "functofloatvalue(real)" definiert, die unter Verwendung der *DB2 Mutator Functions* neue Instanzen des jeweiligen Typs generieren und die Werte für die entsprechenden Attribute setzen. Diese Funktionen erhalten ihre Parameter von den als nächstes zu definierenden Funktionen "c_strval(string)" und "c_floatval(string)". Diese erzeugen aus einem *string*, der die Attributwerte repräsentiert, die Attributwerte in ihrem entsprechenden Datentyp. Anschließend wird eine UDF "stream_val(varchar(260))" definiert, die aus den zuletzt beschriebenen Funktionen dynamisch die richtige aufruft.
Alle definierten Funktionen müssen noch in *DB2 TRANSFORM GROUPs* gegliedert werden. Diese ordnen den UDTs die entsprechenden *TRANSFORM FUNCTIONs* zu, je nachdem, ob es sich um eine *TO SQL*- oder *FROM SQL*–Operation handelt. Der *SQL*–Code der beschriebenen Funktionen ist im Anhang A.1 zu finden.

Nach diesen Arbeiten kann die Datenbank für die Speicherung von XML–Dokumenten und die Anfragebearbeitung benutzt werden. Die gesamte Implementierung wurde in *Java* durchgeführt. Demzufolge wurde für den Datenbankzugriff die *JDBC*–Schnittstelle von *DB2 UDB* benutzt. Im Fall von *SQL*–Queries, die nicht dynamisch erzeugt werden konnten, wurde *DB2 SQLJ* verwendet.

5.2 Realisierter *XQuery*–Sprachumfang

5.2.1 XPath

Grundlage und wichtigster Ausdruck von *XQuery* ist *XPath*. Darauf basiert die gesamte Anfragebearbeitung der Sprache, da nur sie die Navigation in einem XML–Dokument sowie die Selektion von XML–Elementen erlaubt. Die nächsten Abschnitte sollen einen Überblick über Besonderheiten in der Implementierung von *XPath* geben.

5.2.1.1 Die Achsenbezeichner

Beim gegenwärtigen Entwicklungsstand von *XQuery* ist noch nicht klar, wie die Achsenbezeichner innerhalb von Pfadausdrücken zu verwenden sind. Diskutiert wird dabei eine ausführliche Variante, in der die Achsenbezeichner durch ihren geschriebendenen Namen repräsentiert werden oder die verkürzte Variante, bei der die Achsenbezeichner durch Symbole dargestellt werden. Eine Übersicht über die in Pfadausdrücken verwendeten Symbole geben die in Tabelle 3.1 auf Seite 21 aufgeführten Symbolen.
In dieser Arbeit soll die Kurzschreibweise für Achsenidentifikatoren benutzt werden. Da, wie erwähnt, zum gegenwärtigen Zeitpunkt nur die Symbole für die Achsen *self*, *parent*, *child* und *descendant* feststehen, werden auch nur diese implementiert. Das soll aber keine erhebliche Einschränkung der Sprache sein, da es sich bei der *child*- und *descendant*–Achse, um die meistverwendeten Achsen in *XPath* handelt. Außerdem hat diese Tatsache keine Auswirkung auf die Implementierung, da die Achsenbezeichner direkt auf *SQL*–Konstrukte abgebildet werden. Der Mechanismus der Achsenauswertung wurde so implementiert, daß verschiedene Achsen direkt auf unterschiedlichen *SQL*–Konstrukten abgebildet werden. Das heißt, daß die nicht verwandten Achsenidentifikatoren leicht durch die entsprechenden *SQL*–Abbildungen in die Implementierung aufgenommen werden könnten.

Diese könnten auch sofort implementiert werden, sodaß nur die Achsenbezeichner ergänzt werden müßten. Zum Beispiel sieht die *SQL*–Abbildung für die *Ancestor*–Achse wie folgt aus:

```
SELECT * FROM element e, path p
WHERE   e.pathID=p.pathID AND e.docID=node.getDOCID()
        AND e.wi <= node.getWI() AND e.lwi>= node.getLWI()
        AND ( e.wi != node.getWI() OR e.ei != node.getEI() )
        AND p.pathexpr LIKE '%/' + NodeTest
        ORDER BY e.wi DESC, e.ei DESC
```

Desweiteren wurde die Wildcard "*" für Pfadausdrücke implementiert, sie kann als *Name Test* eines *Location Steps* eingesetzt werden und selektiert alle Knoten der entsprechenden Achse.

5.2.2 Der Dereferenzierungsoperator

Eine weitere Einschränkung ist die Verwendung des *Dereferenzierungsoperators "=>"*. Dieser konnte aufgrund der generischen Speicherungsstruktur nicht implementiert werden. Beim Speichern des Dokumentes gehen sämtliche strukturellen Informationen aus einer *Document Type Definition (DTD)*, einem *XML–Schema* oder anderen Typdefinitionen verloren. Zum Beispiel kann ein Pfadausdruck der Form

```
$a/person/@wohnort=>
```

(wobei "wohnort" vom Typ *IDREF* ist) nicht ausgewertet werden, da nicht bekannt ist, welche Elemente Attribute vom Typ *ID* beinhalten, die mit dem Wert von "@wohnort" verglichen werden könnten.

5.2.3 Der Elementkonstruktor

Die Möglichkeit der Erzeugung von neuen Elementen ist eine wichtige Eigenschaft von *XQuery*, deshalb wurde dieser Ausdruck vollständig implementiert. Beachten sollte der Benutzer bei der Verwendung, daß diese Implementierung auf der Grammatik aus [8] basiert. Dort werden die geschachtelten Ausdrücke innherhalb von Elementkonstruktoren, also keine XML–Literale, nicht von geschweiften Klammern "`{}`" umschlossen.

5.2.4 Der FLWR–Ausdruck

Der *FLWR*–Ausdruck wurde vollständig implementiert.

5.2.5 Funktionen

XQuery bietet eine Vielzahl von Funktionen, eine Übersicht gibt [10]. Im Rahmen dieser Thematik ist es ausreichend, nur die "build-in" Funktionen *document(String), count(Sequenz), max(Sequenz), min(Sequenz), sum(Sequenz)* und *avg(Sequenz)* zu implementieren.
Die Funktion *document(string)* wurde in dieser Implementierung gemäß [9] implementiert. Diese liefert den Dokumentknoten (*Document Node)* zurück, der nur durch eine *integer*–Zahl dargestellt wird. Diese Zahl indiziert die in der Datenbank gespeicherten Dokumente. So ist es möglich, auch Knotentypen, die parallel zum äußersten Elementknoten definiert wurden, wie z.B. *Processing Instructions, Comments* oder *Name Spaces*, zu selektieren.
Der Dokumentknoten enthält also nur eine Information über ein Dokument, er stellt lediglich einen Dokumentidentifikator dar und beinhaltet keinen *qualifizierten Namen (QName).*

5.2.6 Operationen auf Sequenzen

Es wurden die Operationen *INTERSECT, UNION, AFTER* und *BEFORE* implementiert. Nicht implementiert wurde die Operationen *UNORDERED.*

5.2.7 Bedingte Ausdrücke

Bedingte Ausdrücke wurde aus Zeitgründen nicht implementiert. Die Umsetzung ist aber unkompliziert und könnte später nachgeholt werden.

5.2.8 *Name Spaces, Comments* und *Proccesing Instructions*

Die Knoten *Name Spaces, Comments* und *Processing Instructions* wurden in dem hier verwendeten Datenbankschema nicht gespeichert. Dementsprechend konnten diese Knoten auch nicht implementiert werden.

5.2.9 Quantoren

Die Quantoren *SOME* und *EVERY* wurden implementiert. In einem quantifizierten Ausdruck werden Variablenbindungen wie in einem *FLWR*–Ausdruck generiert. In der *SATISFIES*–Klausel wird die Bedingung definiert, der die gebundenen Knoten entweder alle (im Fall von *EVERY*) oder mindestens ein Knoten (im Fall von *SOME*) genügen muß.

5.3 Vorschläge zur Optimierung

In dieser Implementierung erfolgt die *XQuery*–Anfragebearbeitung durch die schrittweise Bearbeitung von Pfadausdrücken. Dabei wird für jeden *Location Step* die *Context List* ausgewertet. Das hat den Nachteil, daß unter Umständen sehr viele *SQL*–Anfragen erzeugt und an den Server geschickt werden. Diese Art der Bearbeitung ist zwar konzeptionell eine logische Vorgehensweise, kann in der Praxis aber ein enormes Performanceproblem darstellen.
Eine Verbesserung würde die Abbildung der Pfadausdrücke auf eine einzige oder nur wenige *SQL*–Anfragen darstellen. Dazu müssen geschachtelte *SQL*–Anfragen erzeugt werden. Ein Pfadausdruck der Form

```
document(semester.xml)/semester/studenten
```

würde dann folgendermaßen übersetzt werden.

Die *SQL*–Abbildung des ersten Schrittes, nämlich die Selektion des *Root*–Knotens des Dokumentes (Dokumentknoten), würde folgendermaßen aussehen:

```
SELECT docID FROM  document WHERE name ='semester.xml'
```

Diese Abbildung kann im nächsten Schritt geschachtelt werden. Die folgende *SQL*–Abbildung selektiert alle *Child*–Knoten des *Document*–Knotens mit dem Namen "semester" aus dem Dokument "semester.xml":

```
SELECT e.docid,e.pathid,p.pathexpr,e.wi,e.lwi,e.ei,e.lei
        FROM element e, path p,
                (SELECT docID FROM  document
                 WHERE name ='test.xml') AS d
        WHERE   e.pathID=p.pathID AND e.docID = d.docID
                AND p.docID=e.docID
                AND e.wi = 0 AND e.ei=1 AND
                p.pathexpr LIKE '/semester'
```

Die folgende *SQL*–Abbildung soll die eben gezeigte Query wieder als geschachtelte Query nutzen. Die geschachtelte stellt eine Relation dar, die der *Context List* entspricht und dient zur Abarbeitung des nächsten *Location Steps*.

```
SELECT * FROM element e, path p ,
( SELECT e.docid,e.pathid,p.pathexpr,e.wi,e.lwi,e.ei,e.lei
        FROM element e, path p,
                (SELECT docID FROM  document
                 WHERE name ='test.xml') AS d
        WHERE   e.pathID=p.pathID AND e.docID = d.docID
                AND p.docID=e.docID
```

```
                AND e.wi = 0 AND e.ei=1 AND
                p.pathexpr LIKE '/semester') AS parents
WHERE e.pathID=p.pathID AND e.docID=p.docID
        AND p.docID = parents.docID
        AND p.pathexpr LIKE CONCAT(parents.pathexpr,'/%')
        AND p.pathexpr NOT LIKE CONCAT(parents.pathexpr,'/%/%'
        AND e.wi >= parents.wi
        AND e.lei <= parents.lwi
        AND p.pathexpr LIKE '%/studenten'
ORDER BY e.wi,e.ei
```

Bei der Abbildung von Pfadausdrücken nach diesem Verfahren könnte ein erheblicher Performancegewinn erzielt werden. Leider scheitert die Implementierung mit *IBM DB2*, weil *DB2* diese Query nicht bearbeiten kann und mit einer Fehlermeldung abbricht. Grund dafür ist der "LIKE"–Operator. In Verbindung mit der Funktion "CONCAT(varchar,varchar)" und einem Attribut aus der Schachtelung als Parameter der Funktion, erzeugt *DB2* eine Fehlermeldung mit dem Inhalt, daß die rechte Seite des "LIKE"–Operators, welche die "CONCAT"-Funktion enthält, keine Zeichenkette liefert.

5.4 Die Programmierumgebung

Die Implementierung von *XQuery* erfolgte unter Verwendung folgender Produkte:

- SAX–Parser in der Implementierung von Xerces Version 1.1.2

- Jflex 1.3.1

  JFlex [15] ist ein Generator für lexikalische Analysatoren (*Scanner*) für *Java*.

- Jay

  Jay ist ein ist ein Generator für syntaktische Analysatoren (*Parser*) und ist eine weitgehend originalgetreue Umsetzung von *Yacc*. Damit ist es möglich, die einmal erlernten Techniken von *YACC* auch für die *Java*–Programmierung einzusetzen.

- *Java 1.2*

  Die Implementierung sämtlicher Module wurde mit *Java 1.2* vorgenommen.

- *DB2 UDB Version 7*

  Wie in der Aufgabenstellung gefordert, wurde *XQuery* unter Verwendung von IBMs *DB2 Universal Database* vorgenommen.

Kapitel 6

Zusammenfassung und Ausblick

XQuery ist eine sehr umfangreiche Anfragesprache. Aus diesem Grund konnte die Sprache in dieser Arbeit nur prototypisch implementiert werden. Es wurden die gebräuchlichsten Ausdrücke und deren Features implementiert, sodaß die Sprache ohne erhebliche Einschränkungen eingesetzt werden kann. Einige der Restriktionen ergeben sich bereits aus dem gewählten Datenbankschema. Andere Ausdrücke wurden nicht implementiert, um wichtigeren Ausdrücken den Vorang zu geben.
Der wichtigste *XQuery*–Ausdruck ist der Pfadausdruck. Er stellt den grundlegenden Ausdruck dar, auf den die gesamte Anfragesprache aufbaut. Er findet in fast allen *XQuery*–Ausdrücken Verwendung, zum Beispiel in *FLWR*–Ausdrücken, Elementkonstruktoren oder innerhalb von quantifizierten Ausdrücken. Bei jeder Verwendung von Pfadausdrücken innerhalb einer *XQuery*–Anfrage werden Datenbankzugriffe ausgeführt. Die Anzahl der Zugriffe ist stark abhängig von Struktur und Größe des Dokumentes und ist damit ein entscheidender Performancefaktor. Eine Verbesserung der Performance bei der Auswertung von Pfadausdrücken wurde im Abschnitt 5.3 diskutiert.

In dieser Arbeit wird aufgrund des generischen Datenbankschemas kein Dokument auf seine Richtigkeit bezüglich einer DTD überprüft. Ebenso wenig geschieht das bei der Erzeugung der Ergebnisse. Als Weiterführung der Arbeit könnten in einem nächsten Schritt die einzufügenden Dokumente von einem validierenden Parser überprüft werden. Auch die Ausgabe des Ergebnisses kann so durch einen Parser auf seine Richtigkeit überprüft werden. Dazu müßte dann das Datenbankschema erweitert werden, um die dazugehörenden DTDs mit aufzunehmen. Darüber hinaus könnten die Pfadausdrücke, die innerhalb einer *XQuery*–Anfrage Verwendung finden, während der Query–Auswertung überprüft werden, ob sie gültige Ausdrücke bezüglich der damit assozieierten DTD sind. Für den Fall eines nicht gültigen Pfadausdruckes kann anschließend in einer gewünschten Art und Weise darauf reagiert werden, z.B. mit einem Hinweis, einer Fehlermeldung oder mit dem Abbruch der Query-Bearbeitung.

Ein weitere Vorschlag zur Weiterführung der Implementation wäre jedoch die Vervollständigung des *XQuery 1.0*–Sprachumfangs. Ebenso bietet die Implementierung von *XPath* mit dessen Vielfalt von Ausdrucksmöglichkeiten noch viel Raum für Eränzungen. Ein erster Schritt für die Vervollständigung von *XPath* ist die Ergänzung der fehlenden Achsen. Dazu sind jedoch die nächsten Vorschläge in der Entwicklung von *XQuery* abzuwarten.

Außerdem könnten die Einschränkungen, die das gewählte Datenbankschema mit sich bringt, beseitigt werden. Es könnte um die Relationen "comment","pi" und "namespace" ergänzt werden, um die Knoten *Comment, Processing Instruction* und *Name Space* zu speichern. Diese könnten dann auch in der Implementierung der Sprache beachtet werden.

Anhang A

Anhang

A.1 Definition DB2 Transform Functions

```
create function stringvaluetofunc(a stringvalue)
returns varchar(260)
language sql
return values (a..stringval);

create function floatvaluetofunc(a floatvalue)
returns real
language sql
return values (a..floatval);

create function functostringvalue(str varchar(260))
returns stringvalue
language sql
return stringvalue()..stringval(str);

create function functofloatvalue(i real)
returns floatvalue
```

```
language sql
return floatvalue()..floatval(i);

create transform for stringvalue
stringgroup (from sql with function stringvaluetofunc,
            to sql with function functostringvalue);

create transform for floatvalue
floatgroup (from sql with function floatvaluetofunc,
            to sql with function functofloatvalue);

CREATE FUNCTION val_to_client (a stringvalue )
RETURNS varchar(260)
EXTERNAL NAME 'UDFsrv!strval_to_str'
LANGUAGE java
PARAMETER STYLE db2general
NOT VARIANT
NOT FENCED
NO SQL
NO EXTERNAL ACTION
ALLOW PARALLEL
NO DBINFO
TRANSFORM GROUP stringgroup;

CREATE FUNCTION val_to_client (a floatvalue )
RETURNS varchar(260)
EXTERNAL NAME 'UDFsrv!floatval_to_str'
LANGUAGE java
PARAMETER STYLE db2general
```

```
NOT VARIANT
NOT FENCED
NO SQL
NO EXTERNAL ACTION
ALLOW PARALLEL
NO DBINFO
TRANSFORM GROUP floatgroup;

CREATE FUNCTION c_strval (strstream varchar(260))
RETURNS stringvalue
EXTERNAL NAME 'UDFsrv!str_to_strval'
LANGUAGE java
PARAMETER STYLE db2general
NOT VARIANT
NOT FENCED
NO SQL
NO EXTERNAL ACTION
ALLOW PARALLEL
NO DBINFO
TRANSFORM GROUP stringgroup;

CREATE FUNCTION c_floatval (strstream varchar(260))
RETURNS floatvalue
EXTERNAL NAME 'UDFsrv!str_to_floatval'
LANGUAGE java
PARAMETER STYLE db2general
NOT VARIANT
NOT FENCED
NO SQL
```

```
NO EXTERNAL ACTION
ALLOW PARALLEL
NO DBINFO
TRANSFORM GROUP floatgroup;

--Funktionen, die  den dynamischen Typeauswahl realisiert

CREATE FUNCTION val_stream (ab stringvalue)
RETURNS VARCHAR(260)
LANGUAGE SQL RETURN
WITH test(value) AS
        (SELECT val_to_client(ta.a)
                FROM TABLE (VALUES (ab)) AS ta(a)
                WHERE ta.a IS OF (ONLY stringvalue)
        UNION ALL
        SELECT val_to_client(TREAT (tb.a AS floatvalue))
                FROM TABLE (VALUES (ab)) AS tb(a)
                WHERE tb.a IS OF (ONLY floatvalue))
        SELECT value FROM test;

CREATE FUNCTION stream_val (strstream varchar(260))
  RETURNS stringvalue
  LANGUAGE SQL
RETURN
  (CASE(SUBSTR(strstream,2,POSSTR(strstream,']')-2))
        WHEN 'string'
                THEN c_strval(strstream)
        WHEN 'int'
                THEN c_floatval(strstream)
```

```
        ELSE NULL
END);

create transform for stringvalue DB2_PROGRAM
        (TO SQL WITH FUNCTION stream_val,
        FROM SQL WITH FUNCTION val_stream);
```

A.2 Installation

A.2.1 Installierte Produkte

Folgende Produkte werden für die Installation und das Ausführen der *XQuery* Engine vorausgestetzt:

- JDK 1.2
- Xerces als Implementierung des SAX Parser
  Wenn eine andere Implementierung verwendet werden soll, muß die *URL* im Code der Datei Sax/XMLFill.java geändert werden.
- *IBM DB2 UDB* Die DB2-Version muß *User Defined Types* unterstützen. Die Datenbank muß *Java Enabled* werden. Dies ist zur Ausführung der in *Java* geschriebenen *UDFs* zwingend erforderlich.
- Zum Compilieren des Projektes werden außerdem die Programme *Jflex* und *Jay* , zur Generierung des *Scanners* und des *Parsers*, benötigt.

A.2.2 Das Makefile

Zur Ausführung und Compilierung müssen die Makefiles in den Verzeichnissen `.../Diplomarbeit/XQuery/xquery` und `.../Diplomarbeit/Sax` editiert werden. In dem Makefile müssen muß die "CLASSPATH" Variable der lokalen Umgebung angepaßt werden. Außerdem müssen die Programmpfade von *Java, Javac, Jflex* und *Jay* angepaßt werden.

A.2.3 Generierung des Datenbankschemas

Die beötigten SQL–Scripte befinden sich im Verzeichnis `'.../Diplomarbeit/XQuery/xquery/sql'`. Zur Erzeugung einer funktionierenden Datenbank müssen folgende Schritte ausgeführt werden.

1. Anlegen der Datenbank "xmldocs" (Rechtevergabe beachten, der Nutzer am Client muß Rechte zum Zugriff auf die Datenbank haben)

2. *UDTs* erzeugen

   Den Befehl `'db2 -tf typeCreate.sql'` ausführen.

3. *DB2 Transform Functions* erzeugen

   Den Befehl `'db2 -tf typeFunctionCreate.sql'` ausführen.

4. *UDF* installieren

   Die Datei "UDFsrv.class" in das Verzeichnis $INSTANCEOWNER$/sqllib/function kopieren

Literaturverzeichnis

[1] A. Bonifati, S. Ceri: *Comparative Analysis of Five XML Query Languages*; 15 September 1999

[2] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, David Maier, Dan Suciu: *Querying XML Data*

[3] M. Fernandez, J. Simeon, P. Wadler: *XML Query Languages: Experiences and Examples*

[4] Bernd Bartke: *Untersuchung und Vergleich der XQL mit SQL*; Studienarbeit Wintersemester 1999/2000; *http://userwst2.fh-reutlingen.de/ bartke/*

[5] Takeyuki Shimura, Masatoshi Yoshikawa, Shunsuke Uemura: *Storage and Retrieval of XML Documentes using Object–Relational Databases*

[6] *XML Path Language (XPath)*; W3C Recommendation 16 November 1999; *http://www.w3.org/TR/xpath*

[7] *XQuery 1.0: An XML Query Language*; W3C Working Draft 07 June 2001 *http://www.w3.org/TR/2001/WD-xquery-20010607*

[8] *XQuery: A Query Language for XML*; W3C Working Draft 15 February 2001; *http://www.w3.org/TR/2001/WD-xquery-20010215*

[9] *XQuery 1.0 and XPath 2.0 Data Model*; W3C Working Draft 7 June 2001; *http://www.w3.org/TR/2001/WD-query-datamodel-20010607/Overview.html*

[10] *XQuery 1.0 and XPath 2.0 Functions and Operators Version 1.0*; W3C Working Draft 27 August 2001; *http://www.w3.org/TR/2001/WD-xquery-operators-20010827*

[11] *XQuery 1.0 Formal Semantics*; W3C Working Draft 07 June 2001; *http://www.w3.org/TR/2001/WD-query-semantics-20010607*

[12] *XML Query Requirements*; W3C Working Draft 15 February 2001; *http://www.w3.org/TR/2001/WD-xmlquery-req-20010215*

[13] Jay; Bernd Kühl, Axel–Tobial Schreiner, Fachbereich Mathematik/Informatik der Universität Osnabrück: *Compiler bauen mit yacc und Java*; *http://www.jay.de*

[14] Xerces: *http://xml.apache.org/dist/xerces-j*

[15] Gerwin Klein: *JFlex*; *http://www.jflex.de*

[16] Justin Campbell, Daniel Grossman, Ana–Maria Popescu: *Qilt2Sql – An XML Storage Schema and Query Engine for the Quilt Query Language*; 5 June 2000; *http://www.cs.washington.edu/homes/grossman/projects/544project*

Eidesstattliche Erklärung

Hiermit versichere ich, daß ich meine Diplomhausarbeit

Thema:
"Zur Implementierung von XQuery auf einem Objekt-relationalem XML–Speicher"
selbständig und ohne fremde Hilfe angefertigt habe und daß ich alle von anderen Autoren wörtlich übernommenen Stellen wie auch sich an Gedankengänge anderer Autoren eng anlehnenden Ausführungen meiner Arbeit besonders gekennzeichnet habe.

Rostock, den 22. Mai 2003