XML-Datenbanktechnologien mit XQuery – Vergleichende Bewertung von XML-Datenbanksystemen

Bastian Schenke

22.10.2004

Inhaltsverzeichnis

Kapitel 1

Die XML-Technologiefamilie

XML ist das Schlagwort in der IT-Branche. Inzwischen scheint XML in jedem Bereich ob Datenhaltung oder Telekommunikation, ob Softwareentwicklung oder Multimedia, ob Chemie oder Mathematik, ob Journalismus oder Einzelhandel Verwendung zu finden. XML wird sogar in Forschungsgebieten wie der mittelalterlichen Literatur verwendet. Seit das W3C Anfang 1998 XML zum offiziellen Standard erklärte, hat sich XML in kürzester Zeit in jeglichen Anwendungsbereichen der Informationstechnik eingeschlichen und häufig sogar durchgesetzt.
Anfänglich war die (Extensible Markup Language) eine Technologie unter vielen. In den letzten Jahren hat sich XML jedoch zu der Technologie entwickelt, auf die die Welt scheinbar gewartet hat. Ein regelrechter Hype ist ausgebrochen. Ein Softwareentwickler, der kein XML beherrscht ist kein echter Softwareentwickler mehr. HTML ist nicht mehr State of the Arts, heute wird XHTML verwendet. Einfache Metainformationen werden nicht mehr in einer Webseite eingearbeitet, sondern werden mit RDF beschrieben.
Inzwischen ist XML als Datenformat fast überall eingeführt. Es ist eine einfache, flexible und mächtige Technologie. Neben den Erweiterungen, die zu XML entwickelt wurden und werden, gibt es schon jetzt eine unüberschaubare Fülle an XML-Datenformaten. XML selbst hat durch angelehnte Technologien wie XLinks, XSLT, XPointer, XPath, XML Namespaces und XML Schema noch an Flexibilität und Mächtigkeit zugenommen und kann somit in immer mehr Bereichen angewendet werden. Die bekanntesten XML-Formate sind SVG (Scalable Vector Graphics), MathML (Mathematical Markup Language), SMIL (Synchronized Multimedia Integration Language), CML (Chemical Markup Language) und viele, viele andere, von denen mehrere hundert durch das W3C oder andere Organisationen wie OASIS oder OMG eingeführt und standardisiert wurden.

1.1 XML 1.1

XML ist eine der wichtigsten Entwicklungen der Dokumentsyntax in der Geschichte der IT-Branche. XML ist das für die Datenformate gelungen, was Java für Programmiersprachen erreichen sollte: Verbreitung, Flexibilität und Portabilität verbunden mit Effizienz.

1.1.1 Was ist XML?

XML steht für Extensible Markup Language, ist also eine erweiterbare Auszeichnungssprache für strukturierte Daten jeglicher Art. Strukturierte Informationen lassen sich nach Syntax und Semantik trennen. XML bietet eine Möglichkeit, um Strukturen in Informationen darzustellen.
Meist spricht man von einem XML-Dokument, obwohl XML für weitaus mehr verwendet werden kann. Mit XML können strukturierte Datenströme oder komplexwertige Attribute beschrieben werden, sowie mathematische Formeln oder der zeitliche Ablauf einer Multimediapräsentation.

XML vs. HTML

XML und HTML ähneln sich in der Syntax der einzelnen Tags (Markierungen). Der markante Unterschied ist jedoch, dass HTML für einen ganz speziellen Zweck entwickelt wurde, während XML nur eine Form vorgibt. Die Menge von Tags, die HTML zur Verfügung stehen ist fest definiert. Das gleiche gilt in Bezug auf deren Bedeutung. XML erlaubt es jedoch komplett eigene Tags und zugehörige Attribute und deren strukturellen Beziehungen untereinander zu definieren. Damit lässt sich HTML mit den Wegen und Mitteln von XML beschreiben.
XML ist also eine Metasprache, mit der andere Auszeichnungssprachen definiert werden können.

XML vs. SGML

Auch dieser Vergleich mag naheliegend sein. SGML steht für Standard Generalized Markup Language und ist wie XML eine Metasprache zum definieren neuer Auszeichnungssprachen. HTML ist eine Anwendung oder ein Dokumenttyp von SGML, so wie z.b. SVG eine Anwendung von XML ist.
Obwohl beide scheinbar die selbe Funktion erfüllen enthält XML nur einen Bruchteil der Spezifikationen von SGML. XML ist also wesentlich einfacher als SGML (Zum Vergleich: Die Definition von SGML umfasst mehr als 150 Seiten, die von XML nur ca. 55.). Das hat den Vorteil, dass die Dokumente bzw. Dokumenttypen wesentlich leichter zu erstellen sind und dass die Parser für XML-Dokumente wesentlich kleiner und einfacher zu implementieren sind, als die für SGML.
XML ist also eine echte Untermenge von SGML, die sich auf die wichtigen

(also die benötigten) Features beschränkt. XML ist dadurch vollständig durch einen SGML-Parser interpretierbar. Dabei gilt nur eine Einschränkung, SGML-Dokumente verlangen nach einer Document Type Definition (DTD, Dokument-Typ-Definition), XML verlangt das nicht. XML verlangt nur die Wohlgeformtheit (siehe 1.1.3 ... Wie ist XML aufgebaut?).

1.1.2 Wozu dient XML?

Der ursprüngliche Gedanke hinter XML war, ein Datenformat zu schaffen, mit dem sehr stark strukturierte Daten über das Web ausgetauscht werden konnten. Zum damaligen Zeitpunkt gab es nur zwei Alternativen HTML und SGML.
HTML war zu unflexibel. Die Menge an Tags war begrenzt und der eigentliche Sinn bestand darin Daten anzuzeigen, zu formatieren. SGML auf der anderen Seite war jedoch viel zu umständlich, um sie über das Web durch einen Browser interpretieren zu lassen. Dieser Aufwand wäre zu groß gewesen.
Natürlich wird XML SGML nicht ablösen können. Für viele komplexe Probleme und Strukturen wird SGML wohl noch eine ziemlich lange Zeit benötigt werden, aber um diese Strukturen im Internet zugänglich zu machen werden sie meist in XML übersetzt und dann verschickt.
XML hat sich auch in der Datenkommunikation etabliert. Da XML-Dokumente sehr schnell und einfach zu parsen sind und standardisierte Schnittstellen (APIs) in fast allen gängigen Programmiersprachen existieren, ist die Nutzung von XML-Dokumenten zur Übertragung die naheliegendste und meist auch einfachste Möglichkeit.

1.1.3 Wie ist XML aufgebaut?

Ein XML-Dokument besteht eigentlich aus zwei Teilen: Erstens der Definition (der Form), die durch eine sogenannte DTD bereitgestellt wird, und zweitens den eigentlichen Daten (dem Inhalt) dem XML-Dokument.

Wohlgeformtheit von XML-Dokumenten

Ein XML-Dokument kann sowohl in einer Datei vorliegen, als auch nur virtuell in einer Zeichenkette. Wichtig ist nur, dass das Dokument wohlgeformt ist, damit es verarbeitet werden kann. Im Unterschied zu HTML kann bei XML nicht auf eine bestimmte Interpretation eines Ausdrucks geschlossen werden, da der Kontext von XML-Elementen (im Unterschied zu HTML-Elementen) nicht bekannt ist. Jeder Versuch ein nicht-wohlgeformtes zu parsen, wird eine Fehlermeldung hervorrufen und zum Abbruch des Parsingvorgangs führen.
Jedes Dokument ist als Baum strukturiert. Daraus ergeben sich folgende Eigenschaften für die Wohlgeformtheit:

- Jedes XML-Dokument enthält ein oder mehrere Elemente. Genau eines dieser Elemente ist das Root-Element.
  Das Root-Element zeichnet sich dadurch aus, dass es vollständig außerhalb andere Elemente liegt. Öffnendes und schließendes Tag des Root-Elementes dürfen nicht von anderen Elementen umschlossen werden.

- Tags müssen (anders als bei HTML) immer geschlossen werden.
  Auf ein öffnendes Tag `<Element>` muss auch ein schließendes Tag `</Element>` folgen. Zwischen öffnendem und schließendem Tag befindet sich der Inhalt des Elementes. Die Bezeichnung von öffnendem und schließendem Tag muss gleich sein. XML-Bezeichner sind case-sensitive, es muss also Groß-/ Kleinschreibung beachtet werden (`<ELEMENT>` ist nicht gleich `<element>`).
  Leere Elemente können mit `<LeeresElement/>` notiert werden, was als öffnendes und schließendes Tag interpretiert wird.

- Sowohl das öffnende als auch das schließende Tag müssen Teil desselben Vaterelements sein.
  Verschiedene Elemente müssen also korrekt verschachtelt sein und dürfen nicht ineinander greifen.

Weiterhin ist folgendes durch die XML-Empfehlung des W3C vorgeschrieben, um die Wohlgeformtheit zu gewährleisten:

- XML-Namen wie Element- und Attributnamen müssen folgende Eigenschaften haben:

  - Sie bestehen aus alphanumerischen Zeichen ('`A`'-'`Z`', '`a`'-'`z`' und '`0`'-'`9`') oder anderen nicht-englischen Buchstaben, Zahlen und Ideogrammen. Sie können weiterhin die Interpunktionszeichen '`_`', '`-`' und '`.`' enthalten.
  - Erlaubt ist auch der Doppelpunkt '`:`', wobei dessen Verwendung für die Nutzung in Verbindung mit Namensräumen (namespaces) reserviert ist.
  - Sie enthalten keine Leerzeichen. Dazu zählen normale Leerzeichen, Zeilenumbruch (carriage return), Zeilen- und Seitenvorschub (line/form feed) oder geschützte Leerzeichen.
  - Sie beginnen mit einem Buchstaben, einem Ideogramm oder einem Unterstrich. Ziffern, Bindestrich und Punkt sind nicht erlaubt.
  - Es gibt keine Längenbeschränkung für XML-Namen.

  Zu XML-Namen gehören auch alle Namen, die von anderen Technologien der XML-Familie verwendet werden. Diese Namensbeschränkungen gelten also global.

- Ein XML-Element kann kein, ein oder mehrere Attribute enthalten.
  Für Attribute gelten die Regeln für XML-Namen. Sie werden innerhalb des öffnenden Tags in der Form `<Element attributname = "attributwert">` notiert. Der Wert des Attributes muss in einfachen oder doppelten Anführungszeichen eingeschlossen werden. Enthält der Attributwert einfache oder doppelte Anführungszeichen, so müssen diese entweder maskiert (siehe Maskierung von Sonderzeichen weiter unten) oder der Attributwert von den jeweils anderen Anführungszeichen umschlossen werden.

- Einige reservierte Sonderzeichen, die von XML verwendet werden, müssen in Zeichendaten innerhalb eines Elementes und in Attributwerten durch sogenannte Entity-Referenzen maskiert werden.
  Verpflichtend ist die Maskierung für:

  – das Kleiner-als-Zeichen (`<`), das mit `&lt;` , und
  – das Ampersand (`&`), das mit `&amp;` maskiert wird.

  Besteht die Gefahr, dass in Attributwerten enthaltene Anführungszeichen mit den umschließenden Anführungszeichen verwechselt werden, ist die Maskierung sinnvoll für:

  – die doppelten Anführungszeichen (`"`), die mit `&quot;`, und
  – das einfache Anführungszeichen (`'`), das mit `&apos;` maskiert wird.

  Aus Symmetriegründen zu `<` ist die Maskierung auch erlaubt für:

  – das Größer-als-Zeichen (`>`), das mit `&gt;` maskiert werden kann.
    Diese Entity-Referenz hat außer den kosmetischen Gründen keinen praktischen Nutzen.

  Bei umfangreicheren Texten innerhalb eines Elementes ist es häufig sinnvoll den gesamten Abschnitt mit `<![CDATA[` und `]]>` zu umschließen.

- Erlaubt sind Kommentare, die mit `<!--` beginnen und mit `-->` enden. Innerhalb von Kommentaren ist die Zeichenkette `--` nicht erlaubt.

- Innerhalb eines XML-Dokumentes können auch sogenannte Processing Instructions eingefügt werden.
  Diese werden als Anweisungen für den Parser interpretiert. Es gibt aber keine Garantie dafür, dass sie beachtet werden. Sie werden mit `<?` eingeleitet und mit `?>` abgeschlossen.

- Jedes XML-Dokument kann und sollte, muss aber nicht mit einer XML-Deklaration beginnen. Die XML-Deklaration hat die Form einer PI mit dem Namen "xml", ist aber keine. Sie ist schlicht und einfach eine Deklaration.

Gültigkeit von XML-Dokumenten

Neben der Wohlgeformtheit, die von jedem XML-Dokument vorausgesetzt wird, kann ein XML-Dokument auch auf Gültigkeit geprüft werden. Um diese Prüfung durchzuführen wird eine Definition benötigt, aus der hervorgeht, wie ein gültiges Dokument in diesem speziellen Fall auszusehen hat.
Um die Gültigkeitsbedingungen zu formulieren, wurden die sogenannten DTDs (Document Type Definitions) von SGML übernommen. Mit Hilfe dieser Definitionen kann die Form eines XML-Dokumentes mehr oder weniger exakt beschrieben werden.

XML-Elemente ,die erlaubt sind, und deren erlaubte Inhalte werden mit der Notation `<!ELEMENT Elementname (Elementinhalt)>` definiert. `Elementinhalt` steht für den möglichen Inhalt des Elementes. Elementinhalt kann eine einzelnes Element oder eine Folge von anderen Elementen sein. Möglich sind auch folgende Werte:

- `#PCDATA`
  Parsed Character Data bedeutet soviel wie einfacher Text.
- `ANY`
  Jedes deklarierte Element und beliebiger Text kann Inhalt des Elements sein.
- `EMPTY`
  Das Element darf keinen Inhalt enthalten. So deklarierte Elemente können bedenkenlos mit der abgekürzten Schreibweise `<Element/>` notiert werden.

Die Reihenfolge der Elemente ist bindend. Wichtig ist, dass nur die direkten Nachfolger aufgezählt werden. Mehrere Kindelemente werden, durch Kommata getrennt, aufgezählt.
Möglich ist auch eine `entweder oder` Unterscheidung, die mittels des `|`-Operators getroffen wird. Dieser Operator wird anstatt des Kommas gesetzt.
Für jedes Kindelement (oder auch geklammerte Folgen von Kindelementen) können Kardinalitäten angegeben werden. Elemente ohne explizite Angabe von Kardinalitäten gelten als `genau einmal` vorhanden. Folgende Kardinalitäten
können direkt an das jeweilige Element geschrieben werden:

- `Kindelement?`
  Das Element `kann einmal` vorkommen.
- `Kindelement+`
  Das Element muss `mindestens einmal` vorkommen.
- `Kindelement*`
  Das Element kann `beliebig oft` vorkommen.

XML-Attribute für bestimmte XML-Elemente definiert man mit der Notation `<!ATTLIST Element attribute>`. Die Attribute werden nacheinander mit Angabe des Typs und eines Modifikators angegeben. Die Reihenfolge ist bei Attributen nicht relevant. Die drei häufigsten DTD-Typen sind:

- `CDATA`: Dieser Typ unterliegt den wenigsten Restriktionen. Erlaubt sind alle Zeichenketten.
- `NMTOKEN`: Dieser Typ unterliegt den selben Bedingungen wie XML-Namen ohne die Beschränkung des Anfangszeichens.
  Abgeleitet davon ist `NMTOKENS`, der eine Folge von `NMTOKEN`, getrennt durch Leerzeichen, enthält.
- Aufzählung: Dies ist der am häufigsten genutzte Typ. Hier werden alle erlaubten Werte in Klammern, durch den `|`-Operator getrennt, aufgezählt.

Neben diesen drei (bzw. vier) Attributtypen gibt es noch sieben (bzw. sechs) weitere Typen, die hier nicht weiter behandelt werden können.
Die Modifikatoren geben an, ob ein Attribut existieren muss (`#REQUIRED`) oder nicht (`#IMPLIED`). Ein dritter Modifikator (`#FIXED`) legt einen Wert fest, den das Attribut annehmen muss.

1.1.4 Ein XML-Beispieldokument

Folgende DTD soll als Grundlage für ein Beispiel dienen:

```
1   <!-- Elementdeklarationen -->

    <!ELEMENT Kundendaten (KUNDE*)>
      <!ELEMENT Kunde (Vorname+, Nachname+, Bankverbindung+)>
5       <!ELEMENT Vorname (#PCDATA)>
        <!ELEMENT Nachname (#PCDATA)>
        <!ELEMENT Bankverbindung (Konto+)>
          <!ELEMENT Konto (Kontoinhaber?)>
            <!ELEMENT Kontoinhaber (Vorname+, Nachname+)>
10
    <!-- Attributdeklaration -->

    <!ATTLIST Kunde
      kundennummer NMTOKEN #REQUIRED>
15  <!ATTLIST Konto
      kontonummer NMTOKEN #REQUIRED
      blz NMTOKEN #REQUIRED>
```

Das einzige Element, dass von keinem anderen Element umschlossen wird ist `Kundendaten` (Zeile 3). Es ist also das Root-Element. Es darf nur Kunden enthalten, die jedoch in beliebiger Anzahl (`*`-Kardinalität). Jedes `Kunden`-element (Zeile 4) besteht wiederum aus mindestens einem Vornamen, mindestens einem

Nachnamen und mindestens einer Bankverbindung (+-Kardinalität). Vor- und Nachname (Zeilen 5 und 6) bestehen jeweils aus Text (#PCDATA).
Die Bankverbindung (Zeile 7) wiederum besteht aus mindestens einem Konto (Zeile 8), was wiederum einen Kontoinhaber (Zeile 9) mit Vor- und Nachnamen haben kann, aber nicht muss (?-Kardinalität). Es kann genauso auch leer sein, was in diesem Beispiel so interpretiert werden könnte, dass der Kunde gleichzeitig auch der Kontoinhaber ist.

Gültig im Kontext dieser DTD wäre zum Beispiel folgendes XML-Dokument:

```
1  <?xml version="1.0"?>
   <!DOCTYPE Kundendaten SYSTEM "kunden.dtd">

   <Kundendaten>
5    <Kunde kundennummer="b417">
       <Vorname>Willi</Vorname>
       <Nachname>Meier</Nachname>
       <Bankverbindung>
         <Konto kontonummer = "8319" blz = "82303">
10         <Kontoinhaber>
             <Vorname>Willi</Vorname>
             <Nachname>Meier</Nachname>
           </Kontoinhaber>
         </Konto>
15     </Bankverbindung>
     </Kunde>
     <Kunde kundennummer="v023">
       <Vorname>Holger</Vorname>
       <Nachname>Schulze</Nachname>
20     <Bankverbindung>
         <Konto kontonummer = "2348" blz = "40293">
           <Kontoinhaber>
             <Vorname>Willi</Vorname>
             <Nachname>Meier</Nachname>
25         </Kontoinhaber>
         </Konto>
         <Konto kontonummer = "8341" blz = "82303"/>
       </Bankverbindung>
     </Kunde>
30 </Kundendaten>
```

Die Wohlgeformtheit würde für jedes XML-Dokument geprüft werden. Durch die <!DOCTYPE>-Deklaration in Zeile 2 wird der Parser angewiesen, das Dokument auch auf Gültigkeit zu prüfen. In diesem Zusammenhang spricht man auch von Validation oder validieren. Trotz dieser Anweisung muss der Parser das Dokument nicht validieren. Im Gegensatz zur Wohlgeformtheit ist die Gültigkeit optional.

1.2 XML Schema

Die XML Schema Definition Language wurde vom W3C standardisiert ([04] und [05]). Es dient dazu, XML-Dokumente zu beschränken, ähnlich wie es DTDs tun. Im Rahmen dieser Arbeit ist eine erschöpfende Behandlung nicht möglich. Es wurde aufgenommen, da das Typsystem fast vollständig von XQuery mitbenutzt wird. Viele Details und einige weniger wichtige Konzepte wurden ausgelassen. Wer sich diese trotzdem erarbeiten möchte sei auf "XML Schema Part 0: Primer" ([03] bzw. in deutsch [23]) verwiesen.

1.2.1 Was ist XML Schema?

Der DTD-Ansatz zur Beschränkung von Dokumenten ist ein Erbe von SGML. SGML wurde ursprünglich entwickelt, um Dokumente, die vormals in Papierform existierten in digitaler Form zu beschreiben. Aus diesem Grund sind die DTDs auch gut geeignet, "echte" Dokumente zu beschränken. Nachteilig wirkt sich das jedoch bei datenorientierten XML-Dokumenten aus, die eine genauere Beschreibung von Struktur und Quantität verschiedener Elemente verlangen.
Im Unterschied zu den DTDs wurde bei der Entwicklung von XML Schema großer Wert darauf gelegt, eine bessere Schemabeschreibung für datenorientierte XML-Dokumente zu ermöglichen. So ist es z.B. möglich, einen genauen Wertebereich für die Anzahl eines Elementes anzugeben. Die wichtigste Neuerung, die durch XML Schema eingeführt wurde, ist jedoch die Definition und Verwendung von vorgegebenen und auch selbstdefinierten, komplexen Datentypen.

1.2.2 Wozu dient XML Schema?

Wie bereits erwähnt dient XML Schema, ähnlich wie die DTDs zum beschränken von XML-Dokumenten. Durch XML Schemata werden, genau wie durch DTDs, Dokumentklassen erzeugt.
XML Schema versucht dabei jedoch die Funktionalität von DTDs noch zu erweitern. Die klassischen DTDs haben einige gravierende Nachteile, die hier kurz genannt werden sollen:

Rudimentäre Datentypunterstützung: In den klassischen DTDs gibt es vier "Typen" von Elementinhalten: Kindelemente, PCData, gemischter Inhalt und 'EMPTY'. Für Attribute gibt es zwar mehr Möglichkeiten, aber bei genauerer Betrachtung fällt auf, dass es sich ausnahmslos um Zeichenketten handelt.

Ungenaue Kardinalitäten: Die Angabe von Kardinalitäten ist etwas "kryptisch" und erlaubt keine genaue Angabe von Häufigkeiten bestimmter Elemente.

Keine Wiederverwendbarkeit von Definitionen: Einmal definierte Konstruktionen von Inhaltselementen lassen sich nicht auf ein anderes Element übertragen. Möglich ist hier nur die Definition eines expliziten Elementes, das als Inhalt in mehreren anderen Elementen aufgeführt werden kann. Gleichgeformte Elemente mit unterschiedlichem Namen müssen explizit deklariert werden.

Starres Typsystem: Der Anwender kann keine eigenen Datentypen erstellen. Das Typsystem ist nicht erweiterbar.

Keine Namensraumunterstützung: Das Konzept der XML-Namespaces wurde erst mit XML 1.1 eingeführt. Aus Gründen der Abwärtskompatibilität wurden die DTDs jedoch nicht um dieses Konzept erweitert. Eine Realisierung von Namensräumen ist mittels DTDs zwar möglich, aber unübersichtlich und kompliziert.

Nicht XML-Syntax: Nachteilig wurde gesehen, dass die Definition eines Schemas mit DTDs eine eigene Syntax verlangt.

Zur Vermeidung dieser Probleme wurden viele verschiedene Ansätze entwickelt. Durchgesetzt hat sich letztendlich jedoch der Vorschlag des W3Cs, die XML Schema Definition Language.

1.2.3 Wie ist XML Schema aufgebaut?

Ein XML Schema ist eine normale XML-Datei. Das hat erst mal zur Folge, dass es selbst auf Gültigkeit geprüft werden kann, was bei klassischen DTDs nicht möglich ist.
Zur Prüfung von XML-Schemata gibt es ein Metaschema, das alle XML Schema beschreibt. Da dieses Metaschema wiederum ein XML-Dokument ist, kann es wiederum validiert werden. Um eine unendliche Validierungskaskade zu vermeiden, wurde vom W3C der Ansatz der Selbstvalidierung gewählt. Das heißt, das Metaschema beschreibt sich selbst. Zum Übergang existiert ebenfalls eine Schema-DTD, die zur Validierung von Schemata herangezogen werden kann.

Das Grundgerüst eines XML Schemas

Der Aufbau eines XML Schemas entspricht dem eines normalen XML-Dokumentes. In erster Linie heißt das, dass jedes XML Schema mit einer XML-Deklaration beginnen sollte.
Das Wurzelelement sollte `<xsd:schema>` sein (wobei das Namensraumpräfix optional ist). Es enthält alle Element- und Attributdefinitionen. Alle Elemente von XML Schema gehören zum Namensraum `http://www.w3.org/2001/XMLSchema`.

Um diesem Umstand Rechnung zu tragen, wird ein Namensraumpräfix vorangestellt. Per Konvention ist dieses Präfix immer `xsd:`. Dieses Präfix muss jedem Wort aus dem XML Schemavokabular vorangestellt werden. Das umfasst Element- und Attributnamen, aber auch Typbezeichner.
Soll das erstellte "XML-Vokabular" einem Namensraum zugeordnet werden, so muss das Attribut `targetNamespace` eingefügt werden. Das hat zur Folge, dass jedes erstellte Element oder Attribut automatisch mit diesem Namensraum assoziiert wird.
Das Grundgerüst eines Schemas sollte also folgendermaßen aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="www.example.com/targetNamespace">

  <!-- Element- und Attributdefinitionen -->

</xsd:schema>
```

Element- und Attributdeklaration

Die Definition eines Elementes erfolgt mit dem Element `<xsd:element>`. Jedem Element wird ein Bezeichner mit dem `name`-Attribut und (falls gewünscht) ein Typ mit dem `type`-Attribut zugewiesen. Wird das `type`-Attribut weggelassen, ist jeder Inhalt erlaubt.
Attribute werden mit dem `<xsd:attribute>`-Element definiert. Dabei wird – wie bei einem Element – der Bezeichner und der Typ mit dem `name`- bzw. `type`-Attribut zugewiesen. Einzige Einschränkung hier ist: Attribute dürfen keine komplexen Datentypen enthalten, da sie keine anderen Elemente und Attribute enthalten können.
Mit den Attributen `minOccurs` und `maxOccurs` lassen sich beliebige Kardinalitäten für Elemente angeben. Der `default`-Wert der beiden Attributes ist `1`. Jedes Element kann also ohne weitere Angaben genau einmal vorkommen. Eine unbegrenzte Anzahl wird mit `maxOccurs = "unbounded"` angegeben.
Da ein Attribut maximal einmal auftreten kann, wird hier das `use`-Attribut verwendet. Mögliche Werte sind:

- `optional` ist der `default`-Wert des `use`-Attributes. Ist das `use`-Attribut nicht vorhanden oder enthält den Wert `optional`, kann es auftreten, muss aber nicht.
- `required` bedeutet, dass das Attribut auftreten muss.
- `prohibited` bedeutet, dass das Attribut verboten ist.

Ist das `use`-Attribut `optional` oder nicht angegeben, kann über das Attribut `default` ein Standardwert für dieses Attribut angegeben werden. Wird das ents-

sprechende Attribut, für das ein `default`-Wert definiert wurde, im Instanzdokument nicht angegeben, würde es ein schemaprüfender Parser (was optional ist) einfügen.
Als Alternative zum `default`-Attribut kann auch das `fixed`-Attribut verwendet werden. Tritt das entsprechende Attribut auf, muss es den Wert des `fixed`-Attributes annehmen. Ein schemaprüfender Parser würde, wie beim `default`-Attribut, den Wert des Attributes auf den Wert des `fixed`-Attributes setzen.

Globale Elemente und Attribute

Elemente und Attribute, die als direkte Kinder des `<xsd:schema>`-Elementes deklariert wurden, gelten als globale Element bzw. globale Attribute. Alle globalen Elemente eines Schemas kommen in Instanzdokumenten als Wurzelelement in Frage.
Globale Elemente und Attribute dürfen keine Referenzen enthalten, ihnen muss also direkt ein Typ zugewiesen werden. Da in einem Instanzdokument die Wurzel genau einmal vorkommt, ist es nicht möglich globalen Elementdeklarationen Kardinalitäten und globalen Attributdeklarationen ein `use`-Attribut zuzuweisen.

Referenzieren globaler Elemente und Attribute Die eigentliche Besonderheit an globalen Elementen und Attributen ist jedoch die Möglichkeit, auf sie zu referenzieren. In jeder nichtglobalen Element- bzw. Attributdeklaration kann mit dem `ref`-Attribut auf ein globales Element oder Attribut verwiesen werden. Dem `ref`-Attribut wird einfach der zu referenzierende Bezeichner als Wert zugewiesen.
Dem referenzierenden Element oder Attribut können dann auch Kardinalitäten bzw. ein `use`-Attribut zugewiesen werden.

Definition von Element- und Attributgruppen Es gibt die Möglichkeit Elemente und Attribute zu einer Gruppe zusammenzufassen, um gemeinsam auf sie zu referenzieren. Dazu wird das `<xsd:group>`-Element verwendet. Über das `name`-Attribut wird der Gruppe ein Bezeichner zugewiesen, der als Wert für das `ref`-Attribut des referenzierenden `<xsd:group>`-Elementes dient.

```
<!-- Gruppendefinition -->
<xsd:group name = "strukturierterName">
  <xsd:element name = "Vorname" type = "xsd:string"/>
  <xsd:element name = "Nachname" type = "xsd:string"/>
</xsd:group>

<!-- Referenz auf Gruppe Name -->

<xsd:element name = "Name">
  <xsd:choice>
```

```
      <xsd:group ref="strukturierterName"/>
      <xsd:element name = "einfacherName" type = "xsd:string"/>
    </xsd:choice>
  </xsd:element>
```

Das `<xsd:choice>`-Element bewirkt nur, das eines der beiden Inhaltselemente im Instanzdokument vorkommen darf.
Analog zur Definition von Elementgruppen existiert das `<xsd:attributeGroup>`-Element, das die gleiche Funktionalität für Attribute bereitstellt.

Typdefinition

Einfache Typen wie Zeichenketten und Zahlenbereiche in den verschiedenen Wertebereichen bilden an sich schon ein echtes Novum für XML. Wurden von DTDs nur zeichenkettenbasierte Typen unterstützt, gibt es in XML Schema 44 primitive Datentypen und mächtige Konzepte zum definieren komplexer Datentypen und vererbungsähnlicher Strukturen.

Definition primitiver und abgeleiteter Datentypen Einfache Datentypen in XML sind alle Typen, die sich als Zeichenkette darstellen lassen. Dazu gehören die bekannten Typen wie `decimal`, `float`, `double`, `boolean` und `string`, aber auch andere "einfache" Datentypen wie `date`, `time` und `duration`. Daraus ergeben sich noch weitere vordefinierte abgeleitete Datentypen, die den Wertebereich der einzelnen Typen einschränken oder erweitern (z.B. `long`, `unsignedlong`, `positiveInteger` und `negativeInteger`).
Die Definition einfacher Datentypen erfolgt mit dem `<xsd:simpleType>`-Element. Mit dem `name`-Attribut wird ihm ein Bezeichner zugewiesen. Die Definition eines eigenen Datentyps kann durch eine der drei Möglichkeiten `restriction`, `list` oder `union` erfolgen.

Einschränkungen mithilfe des `<xsd:restriction>`-Elements sind die einfachste Möglichkeit einen neuen Typ zu definieren. Dabei wird innerhalb des `<xsd:simpleType>`-Elements das `<xsd:restriction>`-Element eingefügt. Mit dem `base`-Attribut wird der Basistyp angegeben. Die eigentlichen Einschränkungen des Basistypes wird durch sogenannte Fassetten dargestellt. Es gibt 12 solcher Fassetten, die den Wertebereich auf verschiedene Art und Weise einschränken. Eine genaue Beschreibung kann den W3C-Empfehlungen über XML Schema-Datentypen ([05] oder [25]) entnommen werden. Die wichtigsten sind `pattern`, `enumeration` und jeweils `max-` und `min- Exclusive` bzw. `Inclusive`.

Die `pattern`-Fassette erlaubt die Einschränkung der Werte mittels regulärer Ausdrücke. So ist es beispielsweise möglich den Wertebereich für Postleitzahlen durch folgende Einschränkung zu beschreiben:

```
<xsd:simpleType name = "PLZ">
  <xsd:restriction base = "xsd:string">
    <xsd:pattern value = "\d{5}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Als Basistyp dient `string`, der durch den regulären Ausdruck `\d{5}` auf eine Folge von 5 Ziffern beschränkt wird.
Eine genaue Beschreibung der von XML Schema verwendeten regulären Ausdrücke befindet sich im Anhang F der W3C-Empfehlungen ([05] bzw. [25]).

Die `enumeration`-Fassette ermöglicht die Definition eines Aufzählungstypes. Eine Aufzählung erlaubter Schulnoten wäre z.B. durch folgendes Beispiel möglich:

```
<xsd:simpleType name = "Schulnote">
  <xsd:restriction base = "xsd:integer">
    <xsd:enumeration value = "1"/>
    <xsd:enumeration value = "2"/>
    <xsd:enumeration value = "3"/>
    <xsd:enumeration value = "4"/>
    <xsd:enumeration value = "5"/>
    <xsd:enumeration value = "6"/>
  </xsd:restriction>
</xsd:simpleType>
```

Als Basistyp dient integer. Erlaubt als Wert wären nur die Schulnoten 1 bis 6, jedoch keine Abstufungen wie 2,3 oder 3,7.

Die Fassetten `maxInclusive`, `maxExclusive`, `minInclusive` und `minExclusive` erlauben es den Wertebereich intervallweise zu beschränken. Ein Zahlenbereich aller dreistelliger Zahlen könnte z.B. durch folgendes Beispiel realisiert werden:

```
<xsd:simpleType name = "dreistelligeZahl">
  <xsd:restriction base = "xsd:integer">
    <xsd:minInclusive value = "100"/>
    <xsd:maxInclusive value = "999"/>
  </xsd:restriction>
</xsd:simpleType>
```

Der Wertebereich würde in diesem Beispiel auf das Intervall [100,999] beschränkt sein.

Listen und Vereinigung In XML Schema lassen sich Listen als einfache Datentypen darstellen. Mit dem `<xsd:list>`-Element als Inhalt des `<simpleType>`-Elements wird ein Listentyp definiert. Als Attribut des `<xsd:list>`-Elementes

legt `itemTyp` den Typ der einzelnen Elemente fest. Vorsicht ist hierbei bei `string`-Listen angebracht, da das normale Leerzeichen als Trennzeichen zwischen den Listenelementen verwendet wird. Es sollte also sichergestellt werden, dass die einzelnen Listenelemente keine Leerzeichen enthalten.

```
<xsd:simpleType name = "Zahlenliste">
  <xsd:list itemType = "xsd:integer"/>
</xsd:simpleType>
```

Dieses Beispiel definiert eine Liste von Ganzzahlen. In Verbindung mit der Aufzählung mittels `enumeration`- oder Fassettenrestriktionen lassen sich so fast beliebige Wertelisten definieren.

Das letzte Konzept zur Definition einfacher Typen ist die Vereinigung einfacher Typen. Hierzu dient das `<xsd:union>`-Element, welches als Inhalt von `<xsd:simpleType>` notiert wird. Die zu vereinigenden Wertemengen werden als Liste innerhalb des `memberTypes`-Attributes angegeben. Will man z.B. ein XML-Element definieren, dass als Inhalt entweder eine Ortsbezeichnung oder eine PLZ enthält, kann das unter Verwendung des oben definierten PLZ-Typs folgendermaßen realisiert werden:

```
<xsd:simpleType name = "Ortsangabe">
  <xsd:union memberTypes = "PLZ xsd:string"/>
</xsd:simpleType>
```

Wie oben bereits erwähnt, werden die einzelnen Listenelemente durch ein Leerzeichen getrennt notiert. Dieses Beispiel ist natürlich wenig sinnvoll, da PLZ selbst ein von `string` abgeleiteter Typ ist.

Besonders an den letzten Beispielen ist zu erkennen, dass mit Hilfe einfacher Datentypen schon relativ komplexe Datenstrukturen wie reguläre Ausdrücke, Listen oder Vereinigungsmengen realisiert werden können. Was sie zu einfachen Datentypen macht ist aber die Eigenschaft, dass sie durch eine Stringrepräsentation eindeutig beschrieben werden können. Die wichtigste Einschränkung ist, dass einfache Datentypen keine Elemente oder Attribute enthalten dürfen.

Definition komplexer Datentypen Elementen werden in XML Schema ebenfalls Datentypen zugeordnet, um ihren Inhalt zu bestimmen. Um dieses Konzept zu ermöglichen wurden sogenannte komplexe Datentypen eingeführt, die als Inhalt sowohl andere Elemente, als auch Attribute enthalten können (aber nicht müssen). Ein komplexer Typ wird mit dem Element `<xsd:complexType>` definiert. Es enthält meist Elementdeklarationen, Elementreferenzen und Attributdeklarationen. Mit dem `name`-Attribut wird dem Typ ein Bezeichner zugewiesen, damit dieser später Elementen zugeordnet werden kann.

```
<xsd:complexType name = "...">

  <!-- Element- und Attributdeklarationen bzw. -referenzen -->

</xsd:complexType>
```

Am gebräuchlichsten ist die direkte Angabe von Kindelementen innerhalb des `<xsd:complexType>`-Elementes. Dafür stehen drei verschiedene Inhaltselemente zur Verfügung:

- `sequence`-Gruppe
  Die Elementreihenfolge wird bei DTDs implizit vorausgesetzt, in XML Schema gibt es ein explizites Inhaltselement. Alle Elemente innerhalb dieses `<xsd:sequence>`-Elementes müssen, unter Beachtung der Kardinalitäten, in der selben Reihenfolge im Instanzdokument erscheinen.

  ```
  <xsd:complexType name = "Adresse">
    <xsd:sequenze>
      <xsd:element name = "Name" type = "xsd:string"/>
      <xsd:element name = "Straße" type = "xsd:string"/>
      <xsd:element name = "Hausnummer" type = "xsd:positiveInteger"/>
      <xsd:element name = "Ort" type = "xsd:string"/>
      <xsd:element name = "Postleitzahl" type = "xsd:positiveInteger"/>
    </xsd:sequence>
  </xsd:complexType>
  ```

  Eine Instanz dieser Adresse enthält genau einen Namen, dann genau eine Straße, Hausnummer, Ort und Postleitzahl. In genau dieser Reihenfolge.

- `choice`-Gruppe
  Mit der Definition von Auswahlgruppen mit Hilfe des `<xsd:choice>`-Elementes wird dieselbe Struktur erzeugt wie mit der Oder-Verknüpfung der DTDs `(...|...|...)`. Genau eines der Kindelemente der Auswahlgruppe darf in einem Instanzdokument auftreten.

  ```
  <xsd:complexType name = "Farbtopf">
    <xsd:choice>
      <xsd:element name = "Rot"/>
      <xsd:element name = "Gruen"/>
      <xsd:element name = "Blau"/>
    </xsd:choice>
  </xsd:complexType>
  ```

  Eine Instanz dieses Farbtopfs enthält entweder Rot, Gruen oder Blau. Das Auftreten mehrerer dieser Elemente ist nicht erlaubt.

- `all`-Gruppe
  Das `<xsd:all>`-Element definiert eine Menge von Elementen, die in beliebiger Reihenfolge auftreten dürfen. Die `all`-Gruppe darf nur Elemente enthalten. Anderen Gruppendefinitionen als Inhalt sind nicht erlaubt. Außerdem darf sie nur auf oberster Ebene des Inhaltsmodells auftreten und jedes enthaltene Element darf höchstens einmal vorkommen (`minOccurs` $\leq$ 1, `maxOccurs` $\leq$ 1).

```
<xsd:complexType name = "Datum">
  <xsd:all>
    <xsd:element name = "Tag"/>
    <xsd:element name = "Monat"/>
    <xsd:element name = "Jahr"/>
  </xsd:all>
</xsd:complexType>
```

  Hierbei gehen wird davon ausgegangen, dass einer verarbeitenden Applikation die Reihenfolge der Daten egal ist, sie muss nur unterscheiden können, welches Element den Tag, den Monat oder das Jahr repräsentiert.

Mit Ausnahme der letztgenannten `all`-Gruppe, die nur Elemente enthalten darf und nicht innerhalb anderer Auswahlgruppen auftreten kann, können diese Inhaltsmodelle beliebig geschachtelt und mit `min`-/`maxOccurs`-Attributen versehen werden.
Die Attributdeklarationen werden hinter den Gruppendefinitionen aufgelistet. Da Attribute maximal einmal auftreten können, und ihre Reihenfolge ebenfalls unerheblich ist, gibt es keine Gruppendefinitionen für Attribute, die eine ähnliche semantische Bedeutung hätten.

Primitive Typen mit Attributen Wie bereits oben erwähnt, können primitive Typen keine Attribute tragen. Ein Preiselement vom Typ `xsd:decimal` mit einem Attribut `währung`, ist nicht als einfacher Datentyp realisierbar.
Da nur komplexe Datentypen Attribute definieren können, müssen wir das Element `<xsd:complexType>` verwenden. Hier ein einfaches Beispiel:

```
<xsd:complexType name = "Preis">
  <xsd:simpleContent>
    <xsd:extension base = "xsd:decimal">
      <xsd:attribute name = "währung" type = "xsd:string"/>
    <\xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

Diese Vorgehensweise erweitert den einfachen Typ `xsd:decimal` um ein Attribut namens `währung`. Das `<xsd:simpleContent>`-Element drückt hierbei aus, dass ein einfacher Inhalt erweitert wird.

Gemischter Inhalt Gemischter Inhalt in Dokumenten meint Text in dem Elemente auftreten können. Das ist sehr häufig bei Dokumenten der Fall, in denen bestimmte Hervorhebungen gemacht wurden (z.B. XHTML). Diese Mischung war in DTDs nur unvollständig zu validieren. Man konnte nur festlegen das ein Element auftritt, nicht jedoch wie oft oder in welcher Reihenfolge. Mit dem mixed Type Konzept von XML Schema ist die Validierung nun vollständig möglich.
Dem `<complexType>`-Element wird einfach das `mixed`-Attribut mit dem Wert `true` zugewiesen. Damit ist es erlaubt, das zwischen zwei Elementen auf der selben Ebene Zeichendaten auftreten.

Element ohne Inhalt Von den DTDs her ist bekannt, dass Elemente leer sein können (Schlüsselwort: `EMPTY`). Dieses Konzept wird in XML Schema einfach durch ein leeres `<xsd:complexType/>`-Element ausgedrückt. Sollen dem Element noch Attribute zugeordnet werden, werden diese normal angegeben. Solange das `mixed`-Attribut nicht auf `true` gesetzt wird, ist ein leeres Element mit Attributen deklariert.

Anonyme Typen Nicht immer ist es sinnvoll einen Datentyp explizit zu definieren und ihm einen Bezeichner zuzuweisen. In vielen Fällen wird ein Datentyp nur ein einziges Mal verwendet. In solchen Fällen ist es angebracht, die Datentypdefinition einfach ohne Bezeichner und ähnliches durchzuführen.
Um einen Datentyp anonym zu definieren wird er einfach innerhalb des `element`-Elementes definiert:

```
<xsd:element name = "Laenge">
  <xsd:complexType>
    <xsd:sequence>
    <xsd:element name = "Anzahl" type = "xsd:string"/>
    <xsd:element name = "Einheit" type = "xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Das `name`-Attribute des `<xsd:complesType>`-Elementes entfällt hierbei. Zu beachten ist jedoch, dass ein anonymer Datentyp nicht von anderer Stelle aus mit dem `type`-Attribut referenziert werden kann. Der Datentyp wird nur in diesem einen Element angewendet.

Schema- und Instanzdokument

Bisher wurde erläutert, wie man mit XML Schema XML-Dokumente beschreiben und beschränken kann. Was jedoch noch fehlt, ist ein `DOCTYPE`-ähnliches Konzept, das einem Instanzdokument ein Schema zuordnet.
XML Schema bietet dafür das `schemaLocation`-Attribut bzw. in manchen Fällen

das `noNamespaceSchemaLocation`-Attribut. Definiert das Schema einen Zielnamensraum (`targetNameSpace`-Attribut), wird das `schemaLocation`-Attribut genutzt, definiert es keinen, wird das `noNamespaceSchemaLocation`-Attribut verwendet. Das entsprechende Attribut wird am Wurzelelement des Instanzdokumentes notiert. Um Konflikte mit dem Namensraum des Schemas zu vermeiden muss das entsprechende Attribut mit dem Instanznamensraum assoziiert werden. Dazu wird per Konvention ein Namensraumpräfix `xsi` (XML Schema Instanz) mit dem Namensraum `http://www.w3.org/2001/XMLSchema-instance` definiert und dem Attribut vorangestellt.
Der Wert des `schemaLocation`-Attributes ist der definierte Zielnamensraum des Schemas und, durch Leerzeichen getrennt, einem URI, der auf den Ort des Schema weist. Das `noNamespaceSchemaLocation`-Attribut enthält nur den URI zum Schema.
Definiert das Schema einen eigenen Zielnamensraum, sieht das Wurzelelement des Instanzdokuments folgendermaßen aus:

```
<?xml version="1.0" encoding="UTF-8"?>

<Wurzelelement xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://www.example.com/Zielnamensraum
    http://www.example.com/schema.xsd">

  <!-- Dokumentinhalt -->

</Wurzelelement>
```

Wird kein Namensraum definiert, wird das `noNamespaceSchemaLocation`-Attribut verwendet und nur der URI zum Schema angegeben.

1.3 Weitere XML-Technologien

Durch die schnelle Verbreitung und Flexibilität von XML, ist auch der Anwendungsbereich immer mehr gewachsen. Mit der Zeit wurden immer mehr XML-Technologien entwickelt, die entweder die Möglichkeiten von XML ergänzen oder XML als Grundlage nutzen.
Ein Überblick über die wichtigsten bzw. verbreitetsten XML-Technologien soll an dieser Stelle gegeben werden.

1.3.1 XML-Namensräume

XML-Namensräume bieten die Lösung für ein Problem, dass bei der Verwendung von XML-Dokumenten aus verschiedenen Quellen auftreten kann.
Für jedes XML-Dokument können beliebige Elemente und Attribute durch eine DTD oder ein XML-Schema definiert werden. Dabei kann es vorkommen, dass zwei DTDs oder XML-Schemata die selben XML-Namen verwenden. Für XML-verarbeitende Applikationen kann das zu einem nicht lösbaren Konflikt führen.
Durch XML-Namensräume können verschiedene Dokumentklassen, die durch DTDs und/oder XML-Schemata beschrieben sind, gleiche Namen verwenden. Dabei werden die XML-Namen bestimmten Namensräumen zugeordnet, die durch beliebige URIs repräsentiert werden. Zur weiteren Verwendung wird ihnen ein Namensraumpräfix getrennt durch einen Doppelpunkt vorangestellt. Elementnamen werden also mit `<Namensraum:Elementnamen>` notiert.
Dadurch kann eine verarbeitende Applikation genau unterscheiden, zu welcher DTD oder welchem XML-Schema ein Element oder Attribut gehört.
Genauere Informationen zu XML Namensräumen findet man in den entsprechenden W3C-Empfehlungen ([02]).

1.3.2 XSL

Die Extensible Stylesheet Language (XSL) besteht aus den beiden anderen Sprache XSL Transformations (XSLT) und XSL Formating Objects. Häufig werden XSL und XSLT synonym verwendet, was jedoch nicht richtig ist.

XSLT

XSLT ist selbst eine XML-Anwendung, unterliegt also XML-Syntax. Mit XSLT lassen sich Regeln für die Transformation eines XML-Dokumentes von einem XML-Vokabuar in ein anderes definieren. Als wichtigste Anwendung von XSLT kann die Transformation von XML-Dokumenten in XHTML-Dokumente betrachtet werden.
Grundlegend kann gesagt werden, dass XSLT den Inhalt eines Elements oder Attributs aus dem Quelldokumentbaum in ein bestimmtes Element oder Attribut

des Ergebnisdokumentbaums transferiert. So kann z.B. folgende Liste von Namen im XML-Format:

```
<Namensliste>
  <Name email = "klaus@example.com">
    <Vorname>Klaus</Vorname>
    <Nachname>Meier</Nachname>
  </Name>
  <Name email = "emil@example.com">
    <Vorname>Emil</Vorname>
    <Nachname>Schmidt</Nachname>
  </Name>
</Namensliste>
```

in folgendes (X)HTML-Dokument transformiert werden:

```
<table>
  <tr>
    <td>Klaus</td>
    <td>Meier</td>
    <td>klaus@example.com</td>
  </tr>
  <tr>
    <td>Emil</td>
    <td>Schmidt</td>
    <td>emil@example.com</td>
  </tr>
</table>
```

Einfach gesagt wurde das Element `<Namensliste>` in `table` und die Elemente `<Vorname>` und `<Nachname>` in `<td>` übersetzt. Der Inhalt des Attributes `email` aus dem `<Name>`-Element wurde in ein Element `<td>` umgewandelt. Zur Selektion der verschiedenen Elemente wird XPath verwendet, was auch Teil der Spezifikation von XQuery ist. Die Transformation eines XML-Dokumentes anhand einer XSLT-Definition übernehmen sogenannte XSLT-Parser.
Die Sprache XSLT ist selbst so vielseitig und mächtig, dass inzwischen eine eigene Spezifikation für sie existiert.

XSL-FO

Während XSLT die Sprache zur Transformation von XML-Dokumenten in XML-Dokumente ist, ist XSL-FO für die Transformation "nach außen" zuständig. XSL-FO ist eine vollständige Seitenbeschreibungssprache. Mit XSL-FO ist es möglich XML-Repräsentationen für Bildschirme und vor allem Printmedien zu erzeugen. Die eigentliche Arbeit macht dabei ein sogenannter FO-Prozessor, der aus der XSL-FO-Beschreibung und dem XML-Quelldokument ein fertiges Dokument eines möglicherweise komplett anderen Formates erstellt. Die unterstützten Formate reichen dabei von PDF und RTF über SVG zurück zu XML.

Kapitel 2

Datenbanksprachen für XML

Sprachen für Datenbanken teilen sich in drei große Bereiche: Datendefinitions-, Datenmanipulations- und Datenabfragesprachen. Bei den relationalen Datenbanken hat sich SQL bewährt, das alle drei Sprachbestandteile vereint.
Eine Datendefinitionssprache ist in XML schwierig zu realisieren. Eine solche Sprache definiert die Struktur der Daten, um die Form der Speicherung festzulegen. Da XML-Daten jedoch alles andere als gleichmäßig strukturiert sind, ist eine solche Sprache nicht notwendig. Am ehesten werden dieser Aufgabe XML Schema bzw. die einfachen DTDs gerecht. Da XML-Daten jedoch beliebige Struktur haben können, wird ein XML Schema oder eine DTD meist nicht vom XML-Datenbanksystem verlangt. Für die beiden anderen Komponenten in XML-Datenbanksystemen gibt es viele Ansätze, jedoch noch keinen, der Abfrage- und Manipulationskomponente vereint.
Die meisten bisherigen XML-Datenbanksysteme benutzen als Abfragesprache XPath. Dieser Ansatz birgt jedoch einige Nachteile in sich. XPath ist nicht als Datenbankabfragesprache entworfen worden. Es kann zwar genutzt werden, Abfragen über einzelnen Dokumenten durchzuführen, mehrere Dokumente lassen sich mit XPath jedoch nicht verarbeiten. Joins mehrerer Dokumente sind mit XPath nicht möglich.
Seitdem XML-Datenbanken existieren, arbeiten viele Institutionen und Projekte an einer besser geeigneten Alternative zu XPath als Datenbankabfragesprache. Eine der erfolgversprechensten Abfragesprachen ist XQuery. Diese befindet sich gerade im Standardisierungsprozess beim W3C und gilt dort als Working Draft. Einen Datenmanipulationsteil hat XQuery (zumindest in der jetzigen Version) nicht. Als Manipulationskomponente für Datenbanken gibt es mehrere Ansätze: XUpdate und SiXDML. XUpdate ist ein Arbeitsentwurf der XML:DB Initiative und soll alle von relationalen Datenbanken her bekannten Operationen auf XML-Dokumenten ermöglichen. Viele XML-Datenbanksysteme unterstützen XUpdate. SiXDML ist ebenfalls ein Entwurf der XML:DB Initiative, wird bisher aber kaum unterstützt.

2.1 XPath 1.0

XPath ist eine eigene Empfehlung des W3C vom 16.11.1999 [06]. XPath ist eine Sprache in Nicht-XML-Syntax, die zur Adressierung von Teilen eines XML-Dokumentes dient.
Ursprünglich wurde XPath als Teilmenge von XSL und XPointer entwickelt. In beiden Sprachen ist XPath der Bestandteil, der einzelne Elemente bzw. Teilmengen des XML-Dokumentes identifiziert. XPath ist auf eine ähnliche Weise wie in XSL und XPointer auch in XQuery integriert. Im Zuge der Entwicklung von XQuery wurde parallel an XPath 2.0 gearbeitet, um den Anforderungen von XQuery und zukünftigen Technologien besser gerecht werden zu können.

2.1.1 Was ist XPath?

XPath ist das Ergebnis der Bemühungen, eine einheitliche Semantik für den Zugriff auf XML-Dokumente bereitzustellen. Die Arbeitsgruppen, die sich mit XSLT bzw. XPointer beschäftigten, einigten sich darauf, eine einheitliche Semantik zu verwenden. Ergebnis dieser Bemühungen ist die XPath-Empfehlung.
Diese Arbeit kann nur einen kleinen Einblick in XPath geben. Einen kompletten Einblick in XPath kann man den Empfehlungen des W3Cs entnehmen ([06] und [26]).

2.1.2 Wozu dient XPath?

Ziel von XPath ist es, Teile von XML-Dokumenten zu referenzieren. XPath dient dabei anderen XML-Technologien als Subsprache.
XSLT zum Beispiel dient zur Umwandlung von XML-Dokumenten in andere Formate, wie z.B. HTML. Dabei wird ein Templatemechanismus verwendet. XPath-Ausdrücke dienen dabei zum Identifizieren bestimmter Elemente eines Eingabedokumentes, auf die ein bestimmtes Template angewendet werden soll.
XPointer dient zur Referenzierung von Elementen in anderen XML-Dokumenten. Dabei werden XPath-Ausdrücke genutzt, um das Ziel der Referenz zu bestimmen. XPath-Ausdrücke ermöglichen neben der eigentlichen Ergebnismenge auch die Repräsentation von einfachen Datentypen, wie Strings, Ziffern oder booleschen Werte. Damit wird XSLT ermöglicht, durch Stylesheets eine einfache Nummerierung bspw. von Kapitelüberschriften oder Tabellen- und Abbildungsverzeichnissen bereitzustellen.

2.1.3 Wie ist XPath aufgebaut?

XML-Dokumente werden in XPath als Baum modelliert. Das grundlegende Konstrukt ist ein Ausdruck. Jeder Ausdruck wird zu einem der vier Grundtypen Knotenmenge (node set), Fließkommazahl (number), Zeichenkette (string) oder

Wahrheitswert (boolean) ausgewertet. Anwendungen von XPath (XSLT, XPointer und XQuery) definieren unter Umständen weitere Typen.
Weiterhin enthält die XPath-Spezifikation eine Menge an Funktionen, die je nach verarbeitender Applikation noch erweitert sein kann.

Das Datenmodell von XPath

Jedes XML-Dokument wird in XPath als ein Baum interpretiert. Dieser Baum kann insgesamt sieben verschiedenen Knotentypen enthalten:

- Wurzelknoten
- Elementknoten
- Textknoten
- Attributknoten
- Kommentarknoten
- Steueranweisungsknoten
- Namensraumknoten

Konstrukte wie CDATA-Abschnitte, Entity-Referenzen und die Deklarationen des Dokumenttyp kennt XPath nicht. Der Grund hierfür ist einleuchtend, wenn man bedenkt, dass XPath auf einem wohlgeformten, geparsten XML-Dokument arbeitet. In einem solchen Dokument sind Entity-Referenzen aufgelöst, CDATA-Abschnitte in Form von einfachen Textknoten repräsentiert und Dokumenttyp-Deklarationen eventuell schon validiert. Die XML-Deklaration ist ebenfalls nicht mehr im XPath-Datenmodell vorhanden.
Der Wurzelknoten in XPath entspricht jedoch nicht dem Wurzelelement nach XML-Empfehlung (vgl. [01]). In XPath ist der Wurzelknoten ein abstraktes Konstrukt, das den gesamten Inhalt eines XML-Dokumentes enthält, nicht nur den Inhalt des XML-Wurzelelementes. Der Wurzelknoten kann Element-, Kommentar- und Steueranweisungsknoten enthalten. Im Gegensatz zu Entity-Referenzen könnten diese nämlich noch wichtige Informationen für verarbeitende Applikationen oder Nutzer enthalten.
Nur der Wurzelknoten und die Elementknoten können Kindknoten haben. Alle anderen müssen Blätter des Dokumentbaumes sein. Alle Knoten des Baumes sind geordnet in so genannter Dokumentordnung. Das heißt, die Reihenfolge der Knoten wird durch ihr Erscheinen im Dokument bestimmt. Demzufolge ist der Wurzelknoten immer der erste Knoten des Dokumentes.
Zu beachten ist, dass Attribut- und Namensraumknoten keine Kinder der zugehörigen Elementknoten sind. Sie werden jeweils über so genannte Achsen (siehe unten) selektiert.

Eine genauere und ausführlichere Beschreibung des Datenmodells von XPath kann man der W3C-Empfehlung entnehmen ([06] bzw. in deutsch [26]).

Lokalisierungspfade

Der Kernbestandteil von XPath sind Lokalisierungspfade. Deren grundlegende Syntax ähnelt auf den ersten Blick dem Dateisystem von Unix. Jeder Lokalisierungspfad besteht aus einem oder mehreren Schritten, mit denen man ähnlich den Verzeichnissen im Dateisystem die Ergebnismenge einengt, im Falle von Lokalisierungspfaden jedoch auch erweitern kann. Zwei Schritte im Dokumentbaum werden durch einen Schrägstrich (/) getrennt. Jeder Schritt liefert sozusagen eine Menge an Knoten, die dem nächsten Schritt als Ausgangsmenge dient. Damit kann in jedem Schritt die Knotenmenge, die der Vorgängerschritt geliefert hat, noch weiter eingeschränkt, aber auch erweitert werden.
Der einfachste Fall ist die normale Navigation von der Wurzel des Dokumentbaumes ausgehend zum gewünschten Teilbaum. Dieser kann sowohl aus einem einzelnen Knoten, aber auch aus einer Menge von Knoten bestehen. Beginnt der Pfad bei der Dokumentwurzel, spricht man auch von einem absoluten Pfad, der mit einem `/` beginnt. Jeder Pfad, der nicht mit einem `/` beginnt, ist ein relativer Pfad, er beginnt immer beim aktuellen Kontextknoten. Relative Pfade sind meist Teile von Ausdrücken, die als Prädikat eines Lokalisierungsschrittes dienen.
Folgendes XML-Dokument soll helfen, die Pfadmechanismen zu verdeutlichen:

```
1   <?xml version = "1.0"?>

    <Leute>
      <Person name = "Thales">
5       <Wohnort>Milet</Wohnort>
        <Beruf>Mathematiker</Beruf>
      </Person>
      <Person name = "Pythagoras">
        <Wohnort>Samos</Wohnort>
10      <Beruf>Mathematiker</Beruf>
      </Person>
    </Leute>
```

Der Pfad `/Leute/Person/Wohnort` selektiert alle `<Wohnort>`-Elemente des Dokumentes. Es ergibt also diese Knotenmenge als Ergebnis:

```
<Wohnort>Milet</Wohnort>
<Wohnort>Samos</Wohnort>
```

Ausgehend vom Wurzelknoten `/` des Dokumentes werden alle `<Leute>`-Elemente selektiert. In diesem Fall gibt es nur ein solches Element. Der Kontextknoten für den nächsten Schritt ist also dieses eine `<Leute>`-Element. Im nächsten Schritt werden alle `<Person>`-Elemente ausgewählt, die im aktuellen Kontextknoten liegen, also Kind des `<Leute>`-Elementes sind. Von dieser Knotenmenge wiederum werden alle `<Wohnort>`-Elemente selektiert und als Ergebnismenge zurückgegeben. In diesem Fall gibt es mehrere Kontextknoten, die alle Elemente der vorherigen Ergebnismenge nacheinander auf `<Wohnort>`-Elemente geprüft werden.

Analog würde der Pfad `/Leute/Person/Beruf` alle `<Beruf>`-Elemente selektieren.

Diese einfache Beispiel stellt aber nur einen Spezialfall von Lokalisierungspfaden dar. Es nutzt nur eine abgekürzte Notation und stellt kaum die Möglichkeiten dar, die Lokalisierungspfade bieten.

Ein Lokalisierungpfad besteht meist aus mehreren Lokalisierungsschritten. Ein Lokalisierungsschritt besteht normalerweise aus drei Teilen:

- einer Achse,
- einem Knotentest und
- beliebig vielen Prädikaten.

In obigem Beispiel wurde nur der Knotentest explizit angegeben. Die Achse wurde nicht benannt und daher die Defaultachse verwendet. Prädikate können, müssen aber nicht vorkommen. Die allgemeine Form für Lokalisierungsschritte sieht also folgendermaßen aus:

```
Achse::Knotentest[Prädikat_1][Prädikat_2]...
```

Die Achse bestimmt, welcher Teil des Dokumentes getestet wird, der Knotentest die Knoten aus der Achse, die getestet werden sollen und die Prädikate geben die Auswahlkriterien an, nach denen geprüft wird, ob ein Knoten zum Kontext des nächsten Schrittes gehört oder nicht.

Die Achse ist der erste Bestandteil eines Lokalisierungsschrittes. Sie wird durch einen doppelten Doppelpunkt (`::`) getrennt vom Knotentest notiert. Die Achse gibt an welche Knoten überprüft werden. Einfach ausgedrückt geben sie die Richtung an, in der (ausgehend von Kontextknoten) die Knotentests durchgeführt werden. Es gibt 13 verschiedene Achsen:

- `child`-Achse
- `self`-Achse
- `ancestor`- und `ancestor-or-self`-Achse
- `descendant`- und `descendant-or-self`-Achse
- `following`- und `preceding`-Achse
- `parent`-Achse
- `following-sibling`- und `preceding-sibling`-Achse
- `attribute`-Achse und
- `namespace`-Achse.

Die `child`-Achse führt den Knotentest auf alle Kinder des Kontextknotens aus. Dies ist die Standardachse und wird implizit angenommen, falls keine andere Achse explizit angegeben wurde. Die `child`-Achse kann auch explizit angegeben werden, das ist jedoch unüblich. Der Pfad `/Leute/Person` ist gleichzusetzen mit `/child::Leute/child::Person`.

Die `self`-Achse führt den Knotentest auf dem Kontextknoten selbst aus. In Verbindung mit Prädikaten kann eine Knotenmenge selbst eingeschränkt werden.

Die `ancestor`-Achse enthält alle Vorfahren des Kontextknotens, zusätzlich dazu enthält die `ancestor-or-self`-Achse den Kontextknoten selbst.

Die `descendant`-Achse enthält alle Nachfahren des Kontextknotens, also alle Elementknoten aus dem Inhalt des Kontextknotenelementes (so der Kontextknoten ein Elementknoten ist). Die `descendant-or-self`-Achse umfasst zusätzlich noch den Kontextknoten selbst. Zu beachten ist, dass nur der Wurzelknoten und die Elementknoten Nachfahren haben können, für alle anderen Knotentypen liefern diese Achsen die leere Menge bzw. den Kontextknoten selbst als Ergebnis.

Die `following`- bzw. `preceding`-Achse liefern alle Knoten, die nach bzw. vor dem Kontextknoten liegen. Hierbei ist die Dokumentordnung entscheidend. Ein Element z.B. liegt vor einem anderen Element, wenn sein schließendes Tag vor dem öffnenden Tag des zweiten Elements liegt.

Die `parent`-Achse führt den Knotentest auf dem Elternelement des aktuellen Kontextknotens aus.

Die `following-sibling`- und die `preceding-sibling`-Achse wählen alle vorangehenden bzw. nachfolgenden Knoten aus, die den selben Vaterknoten haben wie der Kontextknoten.

Mit der `attribute`-Achse lassen sich Attribute selektieren. Hierbei ist die abkürzende Schreibweise `@attributname` zulässig und wird meist auch verwendet. Gleichbedeutend sind hier `attribute::name` und `@name`, um ein Attribut zu selektieren.

Die `namespace`-Achse enthält alle Namensraumknoten des Kontextknotens.

Die Achsen `ancestor`, `descendant`, `following`, `preceding` und `self` betrachtet zu jedem beliebigen Kontextknoten partitionieren das gesamte Dokument überlappungsfrei, wenn man von Attribut- und Namensraumknoten absieht.

Der Knotentest ist der einzige Teil eines Lokalisierungsschritt, der zwingend angegeben werden muss. Häufig besteht er nur aus einem Knotenbezeichner. Der angegebene Bezeichner dann wird mit jedem Knotennamen verglichen, der durch die Achse nicht ausgeschlossen wurde. Normalerweise ist es der Name eines Elementes oder eines Attributes. In beiden Fällen wird für jeden Knoten aus der angegebenen Achse überprüft, ob sein Bezeichner mit dem Knotentest übereinstimmt. Ist das der Fall, wird er zur Kontextknotenmenge des Folgeschrittes hinzugefügt, sonst verworfen.
Zusätzlich besteht die Möglichkeit bestimmte Knotentypen auszuwählen. Mit

dem Wildcardzeichen * werden jeweils alle Knoten ausgewählt, die von der entsprechenden Achse eingeschlossen werden. Der Schritt `attribute::*` wählt alle Attribute des Kontextknotens und der Schritt * alle Kindelemente (abgekürzte Schreibweise für `child::*`).
Manche ´Kotentests folgen auch einer Notation, die an Funktionsaufrufe erinnern. Mit dem Knotentest `/node()` werden alle Knoten ausgewählt, die Kindknoten des Wurzelknoten sind, mit den Knotentests `text()`, `comment()` und `processing-instruction()` werden jeweils alle Knoten des entsprechenden Typs in die Ergebnismenge übernommen. Der Knotentest `processing-instruction()` bietet hierbei noch die Möglichkeit, einen Namen als Parameter zu übergeben. In diesem Fall werden nur Processing-Instructions mit dem angegebenen Ziel ausgewählt (z.B. würde der Knoten `<?xml-stylesheet href=... ?>` mit dem Knotentest `processing-instruction('xml-stylesheet')` ausgewählt).

Ein Prädikat filtert die Knotenmenge, die durch Achse und Knotentest bestimmt wurde. Jeder Lokalisierungsschritt kann eine beliebige Anzahl an Prädikaten enthalten. Diese werden in eckigen Klammern umschlossen hinter dem Knotentest notiert.
Ein Prädikat ist immer ein Ausdruck. Das Ergebnis dieses Ausdrucks wird immer in einen Wahrheitswert konvertiert. Ergibt sich der Ausdruck zu einer Zahl, wird er mit der Position des Knoten in der Kontextknotenmenge verglichen. Ist dieser Wert wahr, wird der Knoten in die nächste Kontextknotenmenge übernommen, sonst nicht.

Abkürzende Schreibweisen erleichtern die Notation und Lesbarkeit von XPath-Ausdrücken. Die beiden wichtigsten sind sicher die abkürzenden Schreibweisen für die `child`- bzw. die `attribute`-Achse. Neben diesen häufig verwendeten Abkürzungen gibt es noch drei weitere.

Statt `/descendant-or-self::node()/` kann auch `//` notiert werden. Das ist hilfreich, wenn man beispielsweise alle Elemente eines bestimmten Typs auswählen möchte. Mit dem Pfad `//table` auf einem XHTML-Dokument würden alle Knoten mit dem Bezeichner `table` selektiert werden. Dieser Pfad steht abkürzend für `/descendant-or-self::node()/table`.

Abkürzend für `self::node()` kann auch `.` notiert werden. Mit `.//td` würden alle `td`-Elemente selektiert werden, die Nachfahren des Kontextknoten sind.

Ähnlich kann `parent::node()` durch `..` abgekürzt werden. Mit dieser abkürzenden Schreibweise würde der Elternknoten des Kontextknotens selektiert werden.

Ausdrücke

Ein Ausdruck ist das allgemeinste Konstrukt in XPath. Jeder Ausdruck wird zu einem der vier Grundtypen (node set, number, string und boolean) von XPath

ausgewertet. XPath selber bietet keine Möglichkeit, Variablen zu deklarieren, jedoch können diese von den verwendenden Wirtssprachen (z.b. XSLT oder XQuery) bereitgestellt werden. Der Variablenbezeichner wird dabei einfach mit einem vorangestellten Dollarzeichen notiert.

Ein Lokalisierungspfad ist ein solcher Ausdruck und liefert eine Knotenmenge, die auch leer sein kann. Die Ergebnisknotenmengen zweier Lokalisierungspfade können mit dem Vereinigungsoperator (`|`) verknüpft werden. Der Vereinigungsoperator verknüpft immer zwei Ausdrücke, die eine Knotenmenge als Ergebnis liefern, und liefert die Vereinigungsmenge beider Knotenmengen.

```
/document/chapter[ 2 ] | /document/chapter[ 3 ]
```

Dieser Ausdruck würde zwei `chapter`-Elemente (das zweite und dritte Kapitel) enthalten.

Eine andere Form des Ausdruckes ist ein Funktionsaufruf. XPath stellt eine Funktionsbibliothek zur Verfügung, die weiter unten kurz beschrieben wird. Die bereitgestellten Funktionen liefern immer einen Wert, der vom Typ node set, number, string oder boolean sein muss. Einer Funktion können Argumente übergeben werden, die wiederum wieder durch Ausdrücke repräsentiert werden können. Die Ergebnisse dieser Ausdrücke werden in den entsprechenden Typ, den das Argument haben muss, konvertiert. Eine Funktion kann also durchaus auch einen Lokalisierungspfad als Argument enthalten.

Neben dem Vereinigungsoperator gibt es in XPath noch eine Vielzahl anderer Operatoren. An dieser Stelle wird nur eine kleine Auswahl kurz vorgestellt, eine komplette Auflistung findet man in der Spezifikation.
Die Operatoren `or` und `and` tun genau das, was man vermuten würde. Sie werten die beiden Ausdrücke rechts bzw. links aus, konvertieren ihre Ergebnisse in einen Wahrheitswert und bestimmen das Ergebnis indem die entsprechende logische Operation ausgeführt wird. Als Ergebnis liefern diese beiden Operatoren immer einen Wahrheitswert.
Ausdrücke, die number als Wert liefern, können mit den Operatoren `+`, `-`, `*`, `div` oder `mod` verknüpft werden. Diese Operatoren verhalten sich genau so, wie es aus Programmiersprachen bekannt ist. Die Operanden werden gegebenenfalls in number konvertiert und dann das Ergebnis entsprechend berechnet. Der `-`-Operator sollte immer in Leerzeichen eingeschlossen werden, damit keine Verwechslung mit einem Knotentest auftreten kann. In XML-Namen ist das Zeichen `-` nämlich erlaubt.
Weiterhin werden in XPath noch Vergleichsoperationen (`=`, `!=`, `<=`, `<`, `>=`, `>`) definiert, die je nach verglichenen Knotentypen ein anderes Verhalten aufweisen. Sind beide Vergleichsoperanden Knotenmengen, werden alle diese Operatoren ähnlich ausgewertet. Sie liefern wahr, falls ein Knoten der ersten Knotenmenge den Vergleich für einen beliebigen Knoten der zweiten Knotenmenge erfüllt.
Ist nur einer der Operanden eine Knotenmenge, wird jeder Knoten der Knotenmenge in den Typ des anderen Operanden konvertiert. Ist der Vergleich eines der Knoten mit dem zweiten Operanden wahr, ist der Ausdruck wahr.

Dieses ungewöhnlich anmutende Verhalten der Vergleichsoperatoren hat zur folge, dass `not($a=$b)` nicht zwangsläufig das selbe ist wie `$a!=$b`.

Ausdrücke haben eine besonders wichtige Bedeutung bei der Verwendung in Prädikaten von Lokalisierungspfaden. Hierbei werden die Ausdrücke ausgewertet und das Ergebnis danach in einen booleschen Wert konvertiert, wie weiter oben bereits erläutert wurde.

Funktionsbibliothek

In XPath ist eine Menge von Standardfunktionen enthalten, die als Teil von Ausdrücken auftreten können. Jede Wirtssprache kann noch weitere Funktionen definieren, die nicht in den Spezifikationen von XPath auftreten. Viele verarbeitenden Prozessoren von XPath definieren ebenfalls noch weitere Funktionen.
Einige der Standardfunktionen sollen hier kurz aufgeführt werden.

last() Die Funktion liefert einen Wert vom Typ number. Dieser Wert entspricht der letzten Position im aktuellen Kontextknoten. Häufig findet diese Funktion in Prädikaten Verwendung, um das letzte Element einer Menge auszuwählen.

```
/document/chapter[2]/section[ last() ]
```

Hier wird das letzte `section`-Element selektiert, das in Kapitel 2 auftritt.

count() Die `count`-Funktion erwartet als Argument eine Knotenmenge und liefert als Rückgabewert die Anzahl aller Knoten in der Argumentknotenmenge.

```
count(/document/chapter)
```

Dieser Ausdruck würde zu einer Zahl ausgewertet werden, die die Anzahl aller Kapitel darstellt.

string() Die `string`-Funktion erwartet als Argument ein beliebiges Objekt und liefert einen string als Rückgabewert. Das genaue Verhalten bei verschiedenen Argumenttypen kann man der Spezifikation entnehmen.

```
string(true())
```

Die Funktion true() tut nichts weiter, als den booleschen Wert "wahr" zurückzugeben. Demzufolge ergibt dieser Ausdruck die Zeichenkette `true`.

starts-with() Diese Funktion erwartet zwei Zeichenketten als Argumente. Sie ergibt "wahr", wenn die erste Zeichenkette mit der zweiten beginnt.

```
starts-with("document_27", "doc")
```

Dieser Ausdruck, wird zu "wahr" ausgewertet.

contains() Als Argument erwartet diese Funktion zwei Zeichenketten. Sie liefert "wahr", falls die zweite Zeichenkette Teil der ersten ist.

```
contains("XPath Spezifikation", "XQuery")
```

Dieses Beispiel würde zu "falsch" ausgewertet werden, da "XQuery" nicht Teil des ersten Strings ist.

boolean() Die Funktion `boolean()` erwartet ein beliebiges Argument und konvertiert es in einen Wahrheitswert. Das genaue Verhalten bei verschiedenen Argumenttypen kann man der Spezifikation entnehmen.

```
boolean(/document[2]/chapter)
```

Enthält das zweite Dokument Kapitel, so ist dieser Ausdruck "wahr", sonst "falsch".

number() Die Funktion `number()` erwartet ein beliebiges Argument und konvertiert es in ein Wert vom Typ number. Das genaue Verhalten bei verschiedenen Argumenttypen kann man der Spezifikation entnehmen.

```
/document/chapter[number("3")]
```

Dieser Ausdruck liefert das Kapitel an dritter Position.

round() Diese Funktion erwartet einen Wert vom Typ number und rundet ihn auf eine Ganzzahl ohne Nachkommastellen. Der Rückgabetyp ist number.

```
/document/chapter[round("3.22")]
```

Dieser Ausdruck liefert das dritte Kapitel des Dokumentes.

Eine komplette Auflistung aller Funktionen befindet sich in den Empfehlungen des W3C zu XPath. Sprachen wie XSLT oder spezielle XPath-Prozessoren definieren unter Umständen noch weitere proprietäre Funktionen. Diese werden dann in den entsprechenden Dokumentationen beschrieben.

2.1.4 XPath 2.0

Im Zuge der Entwicklung von XQuery 1.0 und XSLT 2.0 wurde versucht einige Unzulänglichkeiten von XPath auszumerzen. Parallel zu XQuery 1.0 wird in Zusammenarbeit mit der XSLT 2.0 Arbeitsgruppe intensiv an XPath 2.0 gearbeitet, um XPath 2.0 optimal auf die "Wirtssprachen" abzustimmen. Tatsächlich ist es so, dass ca. 80% des XPath 2.0 Sprachumfangs Teil von XQuery sind. Das hat zur Folge, dass viele XPath-Ausdrücke gleichzeitig auch XQuery-Ausdrücke sind. Während XPath 1.0 nur vier Grundtypen kannte, sind es bei XPath 2.0 wesentlich mehr. Durch die Unterstützung von XML Schema in XQuery nutzt auch XPath 2.0 das selbe Typsystem.

Die starken Gemeinsamkeiten von XPath 2.0 und XQuery werden deutlich, wenn man sich die Arbeitsentwürfe des W3C ansieht. So gibt es neben dem XQuery 1.0 [08] und XPath 2.0 Working Draft [07] noch acht weitere Entwürfe, die Teile von XQuery 1.0 und XPath 2.0 beschreiben.

Neben der Unterstützung für mehr als die bekannten vier Datentypen ist eine der wesentlichen Änderungen, dass der Datentyp §node set§ durch §sequence§ ersetzt wurde. Während eine Knotenmenge ungeordnet war und jedes Element maximal einmal enthalten konnte, ist eine Sequenz geordnet und kann ein Element auch mehrmals enthalten. Die Arbeit mit Sequenzen wird im Kapitel über XQuery genauer beschrieben.
Das rudimentären Typsystem von XPath 1.0 wurde durch das Typensystem von XML Schema ersetzt. Für die Funktion von XPath hat das jedoch erstmal weniger Bedeutung. Erst die Verwendung der neuen Datentypen in Verbindung mit XQuery bringt einen wirklichen Vorteil.

Neben den Änderungen, die das Datenmodell, die Funktionsbibliothek und die Operatoren betreffen wurden auch neue Schlüsselwörter eingeführt. Da diese Schlüsselwörter ebenfalls zum Sprachumfang von XQuery gehören, werden sie erst im nächsten Abschnitt näher erläutert.

2.2 XQuery

XQuery ist ein Arbeitsentwurf des W3C und hat bereits eine sehr lange Entwicklung hinter sich.
Bereits kurz nach Fertigstellung der XML 1.0 Spezifikation war erkennbar, dass sich XML nicht nur als Datenaustausch-, sondern auch als Datenablageformat verwenden lässt. Bereits im Dezember 1998 veranstaltete das W3C einen Workshop zum Thema "Query-Sprache für XML". In dieser Konferenz wurden bereits viele unterschiedliche Sprachvorschläge gemacht und in deren Folge die "XML Query Working Group" gegründet. Ziemlich schnell wurde klar, dass eine Lösung auf Basis von SQL nur schwer möglich sei, da die Strukturen der Daten in XML zu inhomogen sind.
Unstrittig war ebenfalls, dass die Arbeit der XML Query Working Group mit den Arbeiten anderer Arbeitsgruppen koordiniert werden mussten. Besondere Bedeutung erlangten hier XML Schema, XML Namespaces, XML Information Set und XSLT. Im Zuge der Zusammenarbeit mit den anderen Arbeitsgruppen wurde beschlossen, XPath einen wichtigen Bestandteil von XQuery sein zu lassen.
Die eigentliche Arbeit an XQuery erfolgte seit 1999 und basierte neben den vielen anderen Sprachen des W3C auf der Sprache Quilt, die wiederum viele der Vorschläge umfasste.
Die aktuellen Working Drafts aller XQuery relevanten Arbeitsentwürfe stammen zur Zeit von November 2003. Wie umfangreich die Arbeiten an XQuery sind, zeigt allein schon die Tatsache, dass insgesamt zehn Einzeldokumente für die unterschiedlichen Bereiche von XQuery 1.0 bzw. XPath 2.0 existieren.

2.2.1 Wie ist XQuery aufgebaut?

XQuery basiert auf Ausdrücken, die jeweils zu einem bestimmten Ergebnis ausgewertet werden. Das hat zur Folge, dass `2+3` einen vollwertigen und gültigen XQuery Ausdruck darstellt, der zu `5` ausgewertet wird.
Die Kernbestandteile von XQuery sind XPath Ausdrücke, FLWOR Ausdrücke, Elementkonstruktoren und viele andere Ausdrucksformen wie Funktionsdefinitionen, Schemaimport und Namensraumdeklarationen. Eine genaue Beschreibung aller Sprachkonzepte findet sich z.b. in [37] oder in einer der zahlreichen Onlinequellen, die inzwischen verfügbar sind.

2.2.2 Aufbau eines XQueryskriptes

Ein XQueryskript besteht in der Regel aus einem Skriptprolog und einem Skriptkörper. Der Prolog ist optional. Hier werden normalerweise globale Einstellung (der so genannte `compile-time context`) und Definitionen vorgenommen.
Im Skriptkörper steht der eigentliche Ausdruck, der ausgewertet wird, um das Ergebnis des gesamten Ausdruckes zu ermitteln. Der Ausdruck im Skriptkörper

kann Variablen- oder Funktionsdefinitionen aus dem Prolog verwenden, genau wie das mit Standardfunktionen möglich wäre.

Prolog

Die wichtigsten Elemente im Prolog sind die Definition von Funktionen, die Definition von globalen Variablen und die Definition von XML Namensräumen. Es stehen noch weitere Features wie Import von Schematypen bzw. ganzen XML Schemata, Import von externen Variablen oder Funktionen zur Verfügung. Da diese Features für Implementierungen jedoch teilweise nur empfohlen sind, sollen sie in dieser Stelle nicht behandelt werden. Für diese Features soll auf [37] verwiesen werden.
Definitionen erfolgen mit Hilfe von Prologausdrücken. Diese werden einfach mit einem Semikolon abgeschlossen, um anzuzeigen, dass sie zum Prolog gehören.

Namensraumdefinitionen Mit der Anweisung `declare namespace` wird ein Namensraumpräfix definiert. Nach dem Schlüsselwort folgt der Name des Präfix und nach einem `=` der mit dem Präfix assoziierte URI in doppelten Anführungszeichen (als Stringliteral).

```
declare namespace x = "http://www.example.com/namespace";
```

Dieses Statement steht für den allgemeinen Fall. Es ist weiterhin möglich Defaultnamensräume, entweder für Funktionen, Variablen oder für Elemente zu deklarieren. In vielen Skripten ist es sinnvoll, einen Standardnamensraum zu definieren, der dem Namensraum der zu verarbeitenden XML Dokumente entspricht.

```
declare default element namespace "http://www.example.com/default";
```

Standardmäßig ist der Standardnamensraum als leere Zeichenkette definiert, was dazu führt, das kein Namensraum angenommen wird.

Funktionsdefinition Die Definition von Funktionen in XQuery erfolgt über das Statement `declare function`, gefolgt vom Funktionsnamen, der Parameterliste und dem Rückgabetyp. Der Funktionskörper wird in geschweifte Klammern eingeschlossen und besteht aus einem Ausdruck.

```
declare function addition ($a as xs:integer,
                           $b as xs:integer) as xs:integer {
  $a + $b
};
```

Die Angabe von Typen der Parameter bzw. des Rückgabewertes ist optional. Wird sie weggelassen ist der Rückgabewert eine einfache Sequenz, die aus beliebigen Elementen bestehen darf.
Wie an diesem Beispiel auch zu sehen ist, haben Typen ein Namensraumpräfix `xs`. Da das Typsystem von XQuery auf XML Schema aufbaut, ist das Präfix `xs` standardmäßig an den XML Schema Namensraum gebunden.

Statt eines Funktionskörpers kann auch das Schlüsselwort `extern` auftreten. Das zeigt an, dass die entsprechende Funktion extern definiert wurde (möglicherweise in einer anderen Programmiersprache). Das genaue Verhalten in einem solchen Fall ist jedoch implementierungsabhängig.

Definition globaler Variablen Die Definition von globalen Variablen erfolgt analog zu den beiden vorherigen Definitionen.

```
declare  variable $a as xs:integer := 12;
```

Mit dieser Definition wird die Variable `$a` im gesamten XQuery-Skript verfügbar und hat den Wert `12`.

Body

Im Skriptkörper eines XQuery-Skriptes steht jeweils ein Ausdruck, zu dessen Wert das Skript ausgewertet wird. Innerhalb dieses Ausdrucks sind alle syntaktischen Konstrukte erlaubt, die im nächsten Teilabschnitt beschrieben sind. Innerhalb dieses Ausdrucks stehen alle definierten Namensräume, Funktionen und Variablen zur Verfügung.

Ausdrücke in XQuery

Kommentare und Leerzeichen Als das Wichtigste in jeder Programmier- oder Skriptsprache sollte der Kommentar gelten. Da er ein grundlegendes Mittel ist, um Code zu dokumentieren, soll er gleich zu Anfang vorgestellt werden. Kommentare in XQuery werden mit `(: :)` notiert. Alle Abschnitte die in "Doppelpunktklammern" eingeschlossen sind, werden von XQuery-Prozessoren ignoriert. Zur besseren Lesbarkeit von Skripten ist es sinnvoll den Quellcode an manchen Stellen einzurücken. Im Allgemeinen ist das in XQuery durch Leerzeichen und Tabulatoren sowie Zeilenumbrüche erlaubt. Leerzeichen und Kommentare dürfen in XQuery fast überall eingefügt werden. Kommentare und Leerzeichen sind nur innerhalb von Sringliteralen und Elementkonstruktoren nicht erlaubt. In einem solchen Fall würden sie als Text interpretiert werden.

Einfache Ausdrücke Die einfachste Form von Ausdrücken in XQuery sind Literale. Unterschieden werden Zahlwert- und Zeichenkettenliterale.
Zeichenkettenliterale werden entweder durch einfache (`'`) oder durch doppelte Anführungszeichen (`"`) eingeschlossen. Ist es nötig, innerhalb einer durch `"` umschlossenen Zeichenkette ein weiteres `"` als Zeichen einzufügen, wird dieses doppelt notiert (analog bei `'`). Stringliterale können auch Entitätsreferenzen enthalten, die bei der Auswertung des Ausdrucks durch die entsprechenden Zeichen ersetzt werden.
Zahlwertliterale werden in der gewohnten Schreibweise ohne einschließende Zeichen notiert. Die Auswertung erfolgt in einen der drei numerischen Typen `integer`,

`decimal` oder `double`, die durch XML Schema definiert sind. Ein Zahlwertliteral wird immer in den einfachsten der drei möglichen Typen umgewandelt.

Variablenreferenzen sind die zweite grundlegende Ausdrucksform in XQuery. Eine Variablenreferenz wird immer zum Wert der Variablen aufgelöst. Sie besteht aus einem einleitenden `$` und dem Namen der Variablen.

Funktionsaufrufe sind ebenfalls Ausdrücke, die einfach aus dem Namen der Funktion gefolgt von der Argumentliste, eingeschlossen in `( )`, bestehen. Bei der Auswertung eines Funktionsaufrufes werden erst die Argumentausdrücke ausgewertet und dann die Funktion selbst. Die Ergebnisse der Argumentausdrücke werden gegebenenfalls in den von der Funktion verlangten Typ konvertiert

Die Auswertungsreihenfolge mehrerer Ausdrücke lässt sich durch runde Klammern `( )` beeinflussen. Ausdrücke innerhalb von Klammern werden immer zuerst ausgewertet.

Arithmetische Ausdrücke Arithmetische Ausdrücke in XQuery erlauben die selben Operationen, die auch im vorhergehenden Abschnitt für XPath beschrieben wurden (`+`, `-`, `*`, `div`, `idiv`, `mod`,). Zusätzlich erlaubt XQuery noch einen weiteren Divisionsoperator `idiv`. Dieser Operator führt eine Division der Operanden wie gewohnt durch, schneidet jedoch alle Nachkommastellen ab und liefert einen Integerwert. Neben den bekannten binären Operatoren ist auch der unäre Operator `-` erlaubt, der einer Multiplikation mit `(-1)` entspricht.
Die Operatoren können beliebige Ausdrücke sein, die sich allerdings zu numerischen Werten konvertieren lassen müssen. Ergibt sich einer der Operanden zu einer leeren Sequenz, wird der gesamte Ausdruck zu einer leeren Sequenz. Sequenzen mit mehr als einem Element sind als Operanden nicht zulässig.

Vergleichsoperatoren In XPath 1.0 waren Vergleichsoperatoren etwas ungewohnt zu handhaben. Die aus XPath 1.0 her bekannten Operatoren weisen immer noch das selbe Verhalten auf. Ein Vergleichsausdruck ist wahr, wenn er für mindestens ein Paar von Elementen (je ein Element pro Operand) wahr ist. Das führt dazu, dass Vergleiche nicht transitiv sind.
Zusätzlich zu den bekannten Operatoren sind in XQuery jeweils die gleichen Operanden als Schlüsselwörter erlaubt: `eq` (equal), `ne` (not equal), `lt` (lesser than), `le` (lesser or equal), `gt` (greater than) und `ge` (greater or equal). Diese Operanden dürfen jedoch nur in Verbindung mit einzelnen Werten als Operanden verwendet werden, sind also nur für Wertevergleiche erlaubt. Ist einer der Operanden eine leere Sequenz, ergibt der ganze Ausdruck eine leere Sequenz.
Zusätzlich zu den normalen und den Wertvergleichsoperatoren gibt es drei Knotenvergleichsoperatoren. Der Operator `is` ist wahr, wenn ein Knoten mit sich selbst verglichen wird. Das kann immer dann vorkommen, wenn ein Knoten durch zwei verschiedene Pfadausdrücke ausgewählt wird. Die anderen beiden Vergleichsoperatoren `<<` und `>>` vergleichen die Dokumentordnung, ergeben also wahr, wenn

der erste Knoten vor dem zweiten im Dokument auftaucht bzw. nach dem zweiten für den `>>` Operator.

Logische Operatoren Zwei Sequenzen, also Ergebnisse von Ausdrücken, können durch die logischen Operatoren `and` und `or` verknüpft werden. Das Verhalten dieser Operatoren dürfte klar sein. Die Operanden werden in einen booleschen Wert konvertiert und dann wird die logische Operation ausgewertet. Im Allgemeinen wird jeder Ausdruck als wahr ausgewertet, wenn er irgendetwas enthält. Als falsch werden nur leere Sequenzen oder Zeichenketten bzw. numerische Werte vom Wert `0` oder `NaN` (Not a Number) ausgewertet.

Bedingte Auswertung XQuery bietet die Möglichkeit der bedingten Auswertung. Mit dem `if-then-else` Konstrukt lassen sich Ausdrücke in Abhängigkeit von einer Bedingung auswerten. Nach dem Schlüsselwort `if` wird ein Ausdruck in runden Klammern notiert, der als Bedingung fungiert und zu einem booleschen Wert ausgewertet wird. Ergibt dieser Wert wahr, wird der Ausdruck hinter dem `then` Schlüsselwort ausgewertet, andernfalls der Ausdruck hinter dem `else` Schlüsselwort.

```
if (3 < 5) then <wahr/> else <falsch/>,
if (3 > 5) then <wahr/> else <falsch/>
```

Dieser Ausdruck liefert zwei Element. Das erste Element ist ein Knoten `<wahr/>` und das zweite Element ein Knoten `<falsch/>`. Als Bedingung bzw. `then` oder `else` Ausdruck kann jeder beliebige Ausdruck stehen.

XPath-Ausdrücke Jeder XPath-Ausdruck ist auch ein XQuery-Ausdruck. Diese Verwandtschaft zwischen den beiden Sprachen ist gewollt. Im Unterschied zu XPath, wie es im vorhergehenden Abschnitt behandelt wurde, verwendet XQuery jedoch XPath 2.0. Wenn man genau betrachtet, welche Sprachfeatures sowohl in XQuery als auch in XPath 2.0 definiert sind, stellt man fest, dass XPath die eigentliche Abfragesprache ist. XQuery setzt praktisch nur weitere Funktionalität auf XPath auf (Funktionsdefinition, Namensraumdeklaration, Schemaimport, etc.). Selbst die oft erwähnten "FLWOR"-Ausdrücke sind zum Teil (`for` und `return`) auch schon durch XPath 2.0 spezifiziert. Im folgenden Abschnitt wird kein expliziter Unterschied zwischen XPath 2.0 und XQuery 1.0 gemacht. Dazu haben beide Sprachen zu viele Gemeinsamkeiten. XPath 2.0 wird hier als Untermenge von XQuery betrachtet.

Im Unterschied zu XPath 1.0 sind in XQuery nicht alle Achsen verpflichtend zu implementieren. Wenn ein XML Datenbanksystem alle aus XPath 1.0 her bekannten Achsen implementiert, erfüllt es das so genannte Full-Axis-Feature.

Elementkonstruktoren Innerhalb eines XQuery-Ausdrucks besteht die Möglichkeit, komplett neue Elemente zu konstruieren. Ein gültiger Ausdruck, der nur aus einem Elementkonstruktor besteht, wäre dieser:

```
<document title="newDocument">
  <chapter title="chapter1">
    <section>11</section>
    <section>12</section>
  </chapter>
  <chapter title="chapter2">
    <section>11</section>
  </chapter>
  <chapter title="chapter3">
    <section>11</section>
    <section>12</section>
    <section>13</section>
  </chapter>
</document>
```

Obwohl in diesem Beispiel keinerlei Schlüsselwörter oder Pfade vorhanden sind, ist diese reine Zeichenkette ein gültiger XQuery-Ausdruck. In diesem Fall würde ein wohlgeformtes XML-Dokument erschaffen werden, dass selbst wieder Teil eines anderen Ausdrucks sein könnte. Wichtig hierbei ist jedoch, dass das Ergebnis dieses Ausdrucks eigentlich eine Sequenz mit einem Inhaltselement ist. Dieses Inhaltselement ist das `document` Element.

Weit interessanter ist die Elementkonstruktion jedoch in Verbindung mit Variablen. Geht man davon aus, dass `$title` eine Sequenz mit drei Zeichenketten ist, die die einzelnen Kapitelüberschriften (`chapter1` bis `3`) tragen, würde folgender Ausdruck genau dasselbe Ergebnis liefern:

```
<document title="newDocument">
  <chapter title="{ $title [1] }">
    <section>11</section>
    <section>12</section>
  </chapter>
  <chapter title="{ $title [2] }">
    <section>13</section>
  </chapter>
  <chapter title="{ $title [3] }">
    <section>14</section>
    <section>15</section>
    <section>16</section>
  </chapter>
</document>
```

Zuerst werden die inneren Ausdrücke `$title [...]` ausgewertet. Diese Ausdrücke sind durch `{ }` eingeschlossen, was dazu führt, dass sie als Ausdrücke und nicht als Zeichenketten ausgewertet werden. Dann wird wie im obigen Beispiel das `document` Element konstruiert und als Sequenz mit einem Inhaltselement ermittelt.

Eine besonders wichtige Bedeutung haben Elementkonstruktoren bei der Darstellung und Umformung von Daten. Auf diese Weise könnte z.b. ein komplettes XHTML-Dokument generiert werden, das als Antwort auf eine HTTP-Anfrage gesendet werden könnte.

Neben diesen "aufgeschriebenen" Elementkonstruktoren können Elemente auch berechnet werden. Für jeden Knotentypen gibt es ein Schlüsselwort (`element`, `attribute`, `document`, `text`, `processing-instruction` und `comment`). Für die Element- und Attributkonstruktoren wird jeweils das Schlüsselwort, gefolgt von einem Bezeichner, notiert und der Inhalt in geschweiften Klammern (`{ }`) eingefügt. Für Dokument-, Text- und Kommentarknoten wird kein Bezeichner, sondern nur der Inhalt in `{ }` erwartet. Der Steueranweisungsknoten verlangt nach zwei Inhaltsausdrücken, der erste enthält das Ziel der Steueranweisung und der zweite den eigentlichen Inhalt.

Dieses Beispiel konstruiert das selbe `document` Element, wie die beiden anderen Konstruktoren:

```
element document { attribute title { "newDocument" },
  (: chapter1 :)
  element chapter { attribute title { "chapter1" },
    element section {"11"},
    element section {"12"}
  },
  (: chapter2 :)
  element chapter { attribute title { "chapter2" },
    element section {"13"}
  },
  (: chapter3 :)
  element chapter { attribute title { "chapter3" },
    element section {"14"},
    element section {"15"},
    element section {"16"}
  }
}
```

Sequenzausdrücke Die Sequenz hat in XQuery eine sehr wichtige Funktion. Oben wurde bereits erwähnt, dass jeder Pfadausdruck eine Sequenz liefert. Das trifft aber nicht nur auf Pfadausdrücke sondern auch auf Ausdrücke allgemein zu. Jeder Ausdruck wird zu einer Sequenz ausgewertet. Ausdrücke, die scheinbar einen atomaren Wert liefern, liefern eigentlich eine Sequenz mit einem Element. In XQuery ist alles eine Sequenz.
Manchmal ist es sinnvoll eine Sequenz selbst zu definieren. Der einfachste Fall ist, alle Elemente der neuen Sequenz in Klammern, durch Kommata getrennt zu notieren. Es würde auch genügen, die Elemente einfach nur durch Kommata getrennt aufzulisten. Diese Vorgehensweise ist jedoch nicht zu empfehlen, da der Kommaoperator die niedrigste Auswertungspriorität hat. Das erste und letzte Element der Sequenz würden somit erst mit den benachbarten Ausdrücken zusammen ausgewertet werden und danach erst als Elemente der Sequenz eingefügt. Indem man die Sequenzen prinzipiell einklammert, umgeht man diese Fehlerquelle.

```
for $a in ( 12, 13, 23, 27, 31 )
return
<zahl>{$a}</zahl>
```

Dieser Ausdruck definiert eine Sequenz von Integerzahlen und wandelt sie in eine Sequenz von `zahl` Elementen um. Statt einfache Integerliterale aufzuzählen, können als Elemente der Sequenz auch beliebige Ausdrücke verwendet werden. Das können sowohl Pfadausdrücke aber auch Variablen oder Elementkonstruktoren sein. Ein Ausdruck der Form `(<eins/>, <zwei/>)` würde eine Sequenz mit zwei Elementen definieren.
Eine Folge von Integerzahlen kann auch mit so genannten Bereichsausdrücken der Form `( 1 to 5 )` definiert werden. Das Ergebnis wäre eine Sequenz mit den natürlichen Zahlen von 1 bis 5.
Genau wie in Schritten von Lokalisierungspfaden können auch Sequenzen mit einem Prädikat versehen werden. Dazu wird die selbe Notation in genutzt. Der Ausdruck `(2 to 4)[2]` würde `3` ergeben, da sich die 3 an zweiter Position der Sequenz befindet.
Auf Sequenzen von Knoten können auch Operationen ausgeführt werden, wie sie zum Teil schon aus XPath bekannt sind. XQuery definiert drei solcher Operationen, die über Schlüsselwörter realisiert sind. Der bekannte Operator `union` (Vereinigung) und zusätzlich `intersect` (Schnittmenge) und `except` (Mengendifferenz).
Wenn diese Variablen gegeben sind:

```
let $a := <A/>, $b := <B/>, $c := <C/>
```

ergibt sich der Ausdruck `($a, $b) union ($b, $c)` zu `(<A/>, <B/>, <C/>)`, der Ausdruck `($a, $b) intersect ($b, $c)` ergibt sich zu `<B/>` und der dritte Ausdruck `($a, $b) except ($b, $c)` ergibt sich zu `<A/>`.

Flower-Ausdrücke Flower ist eine etwas blumige Umschreibung für "FLWOR" und steht abkürzend für `for`, `let`, `where`, `order by` und `return`, die wichtigsten Schlüsselwörter von XQuery.
Die Bedeutung der einzelnen Schlüsselwörter soll kurz erklärt und dann an einem Beispiel verdeutlicht werden.

- `for`
  Mit `for` wird eine geordnete Liste von Elementen generiert, die nacheinander einer Variablen zugewiesen werden. Dieses Schlüsselwort ist vergleichbar mit einer `for each` Schleife, die aus einigen Programmiersprachen bekannt ist.

- `let`
  Das Schlüsselwort `let` weißt einer Variablen eine Sequenz zu. Im Unterschied zu `for` wird jedoch nicht über diese Sequenz iteriert, sondern der Wert bleibt konstant. Während `for` jeweils den Wert eines Elementes der Sequenz annimmt, nimmt `let` die gesamte Sequenz als Wert an.

- `where`
  `where` ist vergleichbar mit dem selbigen Schlüsselwort in SQL. Mit `where` lassen sich die einzelnen Werte, die durch `for` zugewiesen wurden filtern. Dazu wird der Ausdruck, der auf `where` folgt zu einem Wahrheitswert ausgewertet und, je nach Ergebnis, das Element in die Ergebnismenge weitergegeben oder nicht.

- `order by`
  Wiederum vergleichbar mit dem SQL-Äquivalent ist `order by`. Es wird nach der `where` Klausel notiert und wählt einen Schlüssel aus der Ergebnismenge, nach der sortiert wird. Nach dem Sortierkriterium erfolgt die Angabe der Sortierreihenfolge durch `descending` oder `ascending`.

- `return`
  Mit `return` wird für jedes durch `for` definierte und `where` gefilterte Tupel die Ergebnissequenz bestimmt. Häufig werden in der `return` Klausel ein oder mehrere Elementkonstruktoren mit Hilfe von Variablen gefüllt.

Ein Ausdruck in XQuery kann mehrere `for` bzw. `let` Schlüsselworte enthalten, jedoch nur eine `where` Klausel, die nach `for` und `let` auftreten darf. Nach der `where` Klausel dürfen noch beliebig viele Sortierkriterien (`order by`) folgen.
Folgender Ausdruck soll über dem oben generierten Dokument ausgewertet werden:

```
1   for $a in /document/chapter
    let $b := string($a/section[1])
    where count($a/section) < 3
    order by count($a/section) ascending
5   return
      <chapter>
      <first-section>{$b}</first-section>
      { string($a/@title) }
      </chapter>
```

Dieser Ausdruck würde zu folgender Sequenz ausgewertet werden:

```
1   <chapter>
      <first-section>11</first-section>
      chapter2
    </chapter>
5   <chapter>
      <first-section>11</first-section>
      chapter1
    </chapter>
```

Die genaue Auswertungsreihenfolge ist hier kurz beschrieben.
Zuerst wird der Pfadausdruck in Zeile 1 ausgewertet. Als Ergebnis liefert er eine Sequenz mit allen `chapter` Elementen innerhalb des `document` Elementes. Das `for in` Schlüsselwort sorgt jetzt dafür, dass jedes Element dieser Ergebnissequenz

einzeln der Variablen `$a` zugewiesen wird. Die `let` Klausel weist als nächstes der Variablen `$b` die Anzahl der `section` Elemente von `$a` zu. Die `let` Klausel wird nur einmal ausgewertet und bleibt konstant, auch wenn die Variable `$a` für das nächste `chapter` Element einen anderen Wert annimmt.
Als nächstes wird für jedes `chapter` Element, das der Variablen `$a` zugewiesen wurde, die `where` Klausel geprüft. Enthält das entsprechende `chapter` Element aus `$a` weniger als drei `section` Elemente, wird das aktuelle `chapter` Element in die Ergebnismenge aufgenommen.
Als bisherige Ergebnissequenz sind noch zwei `chapter` Elemente übrig. Diese werden jetzt durch die `order by` Klausel nach der Anzahl ihrer enthaltenen `section` Elemente absteigend (Schlüsselwort `ascending`) sortiert.
Der `return` Ausdruck wird für jedes übrig gebliebene `$a` genau einmal ausgewertet. Innerhalb des `return` Ausdrucks wird jeweils ein neues `chapter` Element konstruiert, das ein neuen Elementkonstruktor und einen weiteren Ausdruck umschließt. Hier werden jeweils die inneren Ausdrücke ausgewertet. Da die Variable `$b` mit der `let` Klausel definiert wurde, ändert sich ihr Wert nicht. In diesem XQuery ist sie eigentlich falsch, soll aber zeigen, wie das `let` Schlüsselwort ausgewertet wird.
Der zweite innere Ausdruck innerhalb des `chapter` Elementes liefert den Wert des `title` Attributes und konvertiert ihn in eine Zeichenkette. Demzufolge wird für jedes `chapter` Element das Attribut `title` ausgelesen und als Inhalt des konstruierten `chapter` Elementes eingefügt.

2.2.3 Optionale Features in XQuery

In der XQueryspezifikation sind einige Sprachkonzepte als optional markiert. Es bleibt den Anbietern von XML Datenbanksystemen also überlassen, ob sie diese Features unterstützen oder nicht. Wie genau diese Features angewendet werden (so sie denn unterstützt werden), kann man z.b. dem Buch von Michael Brundage "XQuery The XML Query Language" [37] oder direkt den Spezifikationen des W3C ([11]) entnehmen.

Pragmas Die XQueryspezifikation definiert Pragmas als Anweisung für verarbeitende Applikationen innerhalb eines Queries. Pragmas werden als Kommentar mit doppeltem Doppelpunkt notiert `(:: do something ::)`. Die Spezifikation sieht vor, Pragmas zu ignorieren, wenn sie nicht implementiert sind. Sind sie nicht implementiert, geschieht dies automatisch, da sie ja innerhalb eines Kommentars auftreten.

Module Module haben in XQuery eine ähnliche Funktion wie in anderen Programmiersprachen. Sie enthalten Funktionen, die von XQueryskripten importiert und benutzt werden können. Modulen wird immer ein eindeutiger Namensraum

zugewiesen, über den sie identifiziert werden.

XML Schema Features Optional bietet XQuery die Möglichkeit, Schema zu importieren, um mit diesen Teildokument oder Knotensequenzen "on-the-fly" zu validieren oder benutzerdefinierte Typen zusätzlich zu den XML Schema Basistypen zu benutzen.

Weiteres In XQuery sind noch viele andere, kleinere Features optional, die jedoch nur im Detail von Bedeutung sind. An dieser Stelle sollen sie nicht behandelt werden.

2.2.4 XQuery in der Zukunft

XQuery ist noch keine Empfehlung des W3C. Zur Zeit befindet sich die Spezifikation noch in der "Working Draft" Phase. Einige Features, die hier in diesem Abschnit beschrieben wurden, können sich also noch ändern oder gar ganz verworfen werden. Aller Wahrscheinlichkeit nach, wird das jedoch nur mit einzelnen Details geschehen. Der feste Sprachumfang wird sich – in dieser Version – kaum noch ändern.

Zu erwarten ist, dass in einer späteren Version von XQuery Konzepte zur Datenmanipulation enthalten sein werden. Es existieren bereits viele Vorschläge, die sicher nicht vom W3C ignoriert werden können. Wie genau diese Konzepte dann jedoch aussehen, wann sie in die Spezifikation aufgenommen werden oder ob das überhaupt geschieht ist noch nicht mit Sicherheit zu sagen.

2.3 XUpdate

Die XML Update Language ist keine Empfehlung des W3C. XUpdate wurde von einer Arbeitsgruppe der XML:DB Initiative entwickelt.

2.3.1 Was ist XUpdate?

XUpdate dient als Datenmanipulationssprache für XML-Dokumente. Ursprünglich wurde XUpdate von der Info-Zone Gruppe entwickelt, aber an die XML:DB Initiative abgegeben. Der letzte Working Draft stammt bereits aus dem Jahre 2000 und ist unvollständig und recht frei zu interpretieren.
XUpdate bietet eine syntaktisch einfache Variante Änderungen für XML-Dokumente zu beschreiben. Es ist intuitiv zu verstehen und wenig komplex. Wahrscheinlich wegen seiner Einfachheit unterstützen viele native XML-Datenbanken XUpdate als Standard zur Datenmanipulation, obwohl es anscheinend nicht mehr weiterentwickelt wird.

2.3.2 Wie ist XUpdate aufgebaut?

XUpdate verwendet XML-Syntax und wird als eigenes XML-Dokument notiert. Das hat zur Folge, dass ein XUpdate-Prozessor zwei Eingabedokumente verarbeitet und ein Ergebnisdokument produziert.
Zur Lokalisierung von Elementen wird XPath verwendet. Damit ist jeder Teil eines XML-Dokumentes erreichbar. XUpdate unterstützt folgende Knotentypen:

- Elementknoten
- Attributknoten
- Textknoten
- Steueranweisungsknoten
- Kommentarknoten

Damit sind – bis auf Namensraum- und Wurzelknoten – alle Knotentypen veränderbar, die durch XPath referenzierbar sind.
Das Wurzelelement eines XUpdate-Dokumentes ist immer `<xupdate:modifications>`. Wie jedes andere XUpdate-Element gehört auch das Wurzelelement zum Namensraum `www.xmldb.org/xupdate`. Als Namensraumpräfix haben sich mehrere Varianten etabliert: `lexus` in Anlehnung an die Referenzimplementierung und `xu` bzw. `xupdate`. Als Attribut muss eine Versionsnummer angegeben werden. Dies ist immer `version = "1.0"`, da noch keine weitere Entwicklung erfolgt ist.
Alle Knoten können mit Update-Elementen aus dem XUpdate-Namensraum bearbeitet werden. Folgende Operationen in Form von Elementen werden von XUpdate bereitgestellt:

- `<xupdate:insert-before>`
  Einfügen eines Knotens vor einem ausgewählten Knoten.
- `<xupdate:insert-after>`
  Einfügen eines Knotens nach einem ausgewählten Knoten.
- `<xupdate:append>`
  Einfügen eines Knotens als Kind eines ausgewählten Knotens.
- `<xupdate:update>`
  Ändern des ausgewählten Knotens.
- `<xupdate:remove>`
  Entfernen des ausgewählten Knotens.
- `<xupdate:rename>`
  Umbenennen des ausgewählten Knotens.

Für jedes dieser Elemente ist ein `select`-Attribut vorgeschrieben, das als Inhalt einen XPath-Ausdruck enthält. Dieser Ausdruck muss zu einer Knotenmenge ausgewertet werden können. Die Operation wird immer auf der selektierten Knotenmenge ausgeführt.
Das erstellen neuer Knoten wird in XUpdate über Elemente für jeden Knotentyp beschrieben. Sie werden innerhalb der entsprechenden Update-Elemente notiert (insert-before/-after und append). Entsprechend für die fünf unterstützten Knotentypen existieren folgende Elemente:

- `<xupdate:element>`
- `<xupdate:attribute>`
- `<xupdate:text>`
- `<xupdate:processing-instruction>`
- `<xupdate:comment>`

Das `<xupdate:element>`-Element definiert ein neues Element. Der Name des Attributes wird mit dem `name`-Attribut zugewiesen. Der Inhalt des neuen Elementes wird einfach innerhalb des Elementes notiert. Das kann entweder normaler Text oder eine Folge neuer Knotendefinitionen sein.
Das `<xupdate:attribute>`-Element definiert ein neues Attribut. Der Inhalt des Elementes ist der Attributwert und der Name des neuen Attributes wird mit dem `name`-Attribute angegeben. Das `<xupdate:attribute>`-Element kann zusätzlich noch innerhalb eines `<xupdate:element>`-Elementes auftreten.
Die Elemente `<xupdate:text>` und `<xupdate:comment>` definieren einen Text-

bzw. Kommentarknoten. Der Inhalt wird jeweils in das Ergebnisdokument übernommen. Ein `name`-Attribut gibt es selbstverständlich nicht.
Das Element `<xupdate:processing-instruction>` erzeugt einen neuen Steueranweisungsknoten innerhalb des Dokumentes. Mit dem `name`-Attribut wird das Ziel der Steueranweisung angegeben. Der Inhalt entspricht der Steueranweisung.

Neben den insert-Elementen gibt es noch die Möglichkeit neue Knoten als Kindelemente an bestehende Knoten "anzuhängen" und ihnen eine bestimmte Position zuzuweisen. Das `<xupdate:append>`-Element erfüllt die selbe Funktion wie die insert-Elemente, bietet jedoch die Möglichkeit mit einem optionalen `child`-Attribut die Position zu bestimmen, an der der Inhalt eingefügt werden soll. Das `child`-Attribut enthält einen XPath-Ausdruck, der sich zu einem `integer` auflösen lassen muss. Wird das Attribut nicht angegeben, wird es durch `child = "last()"` ersetzt. Der Inhalt des `<xupdate:append>`-Elementes wird genauso interpretiert wie der der insert-Elemente.

Das `<xupdate:remove>`-Element erlaubt es, Knoten komplett zu löschen. Der Inhalt dieses Elementes ist leer

Das Element `<xupdate:update>` ersetzt den Inhalt des selektierten Knotens. Mit diesem Element können alle Knoten verändert werden.

Das Element `<xupdate:rename>` kann nur auf Element- und Attributknoten angewendet werden. Es ersetzt den Namen des Knotens durch den eigenen Inhalt.

Mit den Elementen `<xupdate:variable>` und `<xupdate:value-of>` können Variablen deklariert und deren Wert wieder abgefragt werden. So ist es möglich, Knoten in einem Baum zu kopieren oder zu bewegen.
Ersteres Element definiert eine Variable mit einem Namen, der über das `name`-Attribut zugewiesen wird. Der Wert der Variable wird mit dem `select`-Attribut bestimmt und ist immer die Ergebnismenge eines XPath-Ausdrucks. Der Wert der Variable wird mit dem zweiten Element abgefragt. Der Name der abzufragenden Variable wird über das `select`-Attribut angegeben, wobei dem Variablenbezeichner ein Dollarzeichen vorangestellt wird.

Kapitel 3

XML-DBS

Mit der Verbreitung von XML in vielen Bereichen der IT wie Kommunikation, Repräsentation und Präsentation bekam die performante Speicherung von XML-Daten eine immer größere Bedeutung. Viele etablierte relationale Datenbanksysteme wurden um XML-Funktionalität erweitert, um auf dem neuen Markt XML Fuß zu fassen.
Die Nutzung von relationalen Datenbanken mit XML Erweiterungen ist für einige Anwendungen ausreichend, jedoch für andere viel zu unperformant. Je nach Strukturierung der Daten, ist es meist sinnvoller native XML Datenbanksysteme zu verwenden.

3.1 XML in Datenbanken

Wesentlich in der Betrachtung von Lösungen für die persistente Speicherung von XML Dokumenten in Datenbanken ist die Struktur der Dokumente. Hierbei lassen sich grob zwei Klassen von Dokumenten nennen.

- Datenzentrierte Dokumente und
- Dokumentorientierte Dokumente

Datenzentrierte Dokumente haben eine regelmäßige Struktur. Das heißt sie bestehen aus einer Folge von gleich oder zumindest ähnlich strukturierten Elementen. Die Reihenfolge dieser Elemente ist nicht relevant, da sie nur Aufzählungen repräsentieren (ähnlich einer Zeile in einer Tabelle). Gemischter Inhalt kommt in Ihnen nicht vor.
Ein Dokument wie dieses zählt als datenzentriertes Dokument.

```
1   <?xml version = "1.0"?>
    <Leute>
      <Besonderheit>schon tot</Besonderheit>
      <Person name = "Thales">
5       <Wohnort>Milet</Wohnort>
        <Beruf>Mathematiker</Beruf>
```

```
    </Person>
    <Person name = "Pythagoras">
      <Wohnort>Samos</Wohnort>
      <Beruf>Mathematiker</Beruf>
    </Person>
  </Leute>
```

Die Struktur ist regelmäßig, Da jedes Personelement die gleichen Kindelemente (Wohnort und Beruf) hat. Bei einer Anfrage ist es nicht von Bedeutung, welches de beiden Personelemente zuerst zurückgegeben wird.
Solche Datenzentrierten Dokumente lassen sich leicht in ein relationales Datenbankschema transformieren. Eine Tabelle würde genügen. Jeder Person werden ein Name, ein Wohnort, ein Beruf und eine Besonderheit zugeordnet.
Datenzentrierte Dokumente lassen sich in der Regel immer leicht in ein relationales Schema transformieren.

Dokumentzentrierte Dokumente hingegen weisen meist eine vollkommen unvorhersehbare Struktur auf. Beispiele für dokumentorientierte Dokumente sind Briefe, Bücher, Werbung und alle Dokumente, die keine regelmäßigen Strukturen aufweisen. In dokumentzentrierten Dokumenten ist die Reihenfolge der Elemente von großer Bedeutung. So sollte Kapitel 1 immer auch vor Kapitel 2 erscheinen, wenn es nicht explizit anders verlangt wird. Häufig ist in solchen Dokumenten auch gemischter Inhalt vorhanden, der ein Mapping in ein relationales Schema schwierig macht.
Eine Email ist ein typisches Beispiel für solche dokumentzentrierte Dokumente.

```
<?xml version = "1.0"?>
<email gesendet="02.04.2003">
  <Absender>Herr P</Absender>
  <Empfänger>Herr Q</Absender>
  <Betreff>wichtige Nachricht</Betreff>
  <Text>
    <Anrede>Hallo Q,</Anrede>
    ich habe hier eine <Fett>wichtige</Fett> Nachricht für dich.

    Bitte antworte bis zum <Termin>10.04.2003</Termin>!

    <Gruß>MfG</Gruß>, P
  </Text>
</email>
```

Allein durch die Folge von Text und enthaltenen Elementen (gemischter Inhalt) ist ein Mapping in relationale Schemata schwierig.

Daten- bzw. dokumentorientierte Dokumente kommen nur selten in reiner Form vor. Wesentlich häufiger sind Mischformen anzutreffen, die Elemente beider Klassen enthalten.
Beim Mapping solcher halbstrukturierter Daten in relationale Schemata würden viele Spalten entstehen, die meist Nullwerte enthalten. Das bringt einen unverhältnismäßig hohen Speicherverbrauch mit sich. Wenn man versucht, dieses Problem mit dem Erstellen von entsprechend mehr Tabellen zu umgehen, würde

das zu einer erheblichen Verlangsamung von Abfrageoperationen führen, da die Berechnung der Abfragealgebraen durch viele Tabellen erschwert wird.
Die Probleme beim Mapping von XML Dokumenten in relationale Schemata lassen sich aus folgenden Überlegungen ableiten:

- Ein relationales Schema muss für jedes Dokument angelegt werden, da verschieden Dokument zu viele Unterschiede aufweisen können.
- Selbst bei einfachen Änderungsoperationen, die die Struktur betreffen, muss unter Umständen ein neues relationales Schema erstellt werden.
- Die große Fülle an Kombinationsmöglichkeiten von gemischtem Inhalt würde eine Tabellenstruktur verlangen, die nicht mehr sinnvoll zu verwalten wäre.
- Kommentare und Steueranweisung können an beliebiger Stelle im Dokument auftauchen. Sollen solche Elemente ebenfalls gespeichert werden, verkompliziert sich das Tabellenschema noch um ein Vielfaches.

Um die Schwierigkeiten des Mappings zu umgehen, kann man XML Dokumente auch als CLOBs oder BLOBs in einer relationalen Datenbank abspeichern. Bis auf ein sehr grobgranulares Transaktionsmanagment ergeben sich jedoch keine Vorteile gegenüber der Verwendung eines normalen Dateisystems (vgl. [36]).

Daten- bzw. dokumentorientierte Dokumente kommen nur selten in reiner Form vor. Wesentlich häufiger sind Mischformen anzutreffen, die Elemente beider Klassen enthalten.
Trotzdem ist eine Lösung mit Hilfe eines relationalen Datenbanksystems selten sinnvoll. Es ist also notwendig, bereits bei Entwurf und Implementierung eines XML Datenbanksystems andere Ansätze zu verwenden.

3.1.1 Native XML Datenbanksysteme

Was sind native Datenbanksysteme? Aufgeworfen wurde dieser Begriff von der deutschen Firma Software AG im Rahmen einer Vermarktungskampagne für ihr XML Datenbanksystem Tamino. Darauf folgten viele Diskussionen in Mailinglisten und Foren um den Begriff Native XML Datenbank zu definieren. Folgende Punkte müssen laut XML:DB mailing list von einer nativen XML Datenbank erfüllt sein (vgl. [35]):

- Ein natives Datenbanksystem definiert ein Datenmodell für XML Dokumente. Entsprechend zu diesem Datenmodell werden Dokumente gespeichert und wiederhergestellt. Minimal muss ein solches Datenmodell Elemente, Attribute, PCDATA und Dokumentordnung definieren.
- Die grundlegende (logische) Speichereinheit sind Dokumente.

- Das zugrunde liegende physikalische Speichermodell ist nicht festgeschrieben.

Diese Definition ist zwar nicht in allen Belangen befriedigend, soll aber an dieser Stelle genügen.

Mögliche Features

Um auf ein natives XML Datenbanksystem zu nutzen bedarf es an Gründen, ihnen den Vorzug vor bekannten und bewährten relationalen Datenbanken zu geben. An dieser Stelle soll einfach eine kurze Übersicht über mögliche Eigenschaften oder Fähigkeiten von nativen XML Datenbanken gegeben werden, größtenteils sind diese Konzepte schon von anderen Datenbanktypen her bekannt.

- Collections oder Libraries
  Mithilfe von Collections bzw. Libraries können die in der Datenbank enthaltenen XML Dokumente sortiert werden. So können Abfragen über mehreren spezifischen Dokumenten durchgeführt werden.

- Zugriffsberechtigung
  Auf der Basis von Collections lassen sich Berechtigungen für Nutzer und Nutzergruppen festlegen.

- Abfragesprachen
  In der Praxis hat sich gezeigt, dass SQL als Abfragesprache für XML Dokumente, seien sie auch in relationale Schemata gemappt, viel zu inperformant ist. Native XML Datenbanken bieten XML-spezifische Abfragesprachen, um den Anforderungen gerecht zu werden. Dazu gehören z.b. XPath, XQuery oder XSL mit seinen beiden Komponenten XSLT und XSL-FO.

- Datenmanipulation
  Ist SQL schon für Abfragen schlecht geeignet, so trifft das erst recht auf Manipulationsoperationen zu. Beim W3C sind derzeit keine Bemühungen erkennbar, eine XML Updatesprache zu spezifizieren. Dieser Umstand kann sich jedoch ändern, wenn sich die derzeitigen Arbeitsentwürfe dem Ende des Standardisierungsprozesses nähern.
  Bis dahin gibt es mehrere Vorschläge anderer Institutionen oder Personengruppen, die bereits heute von vielen XML Datenbanken unterstützt werden (XUpdate, SiXDML, Erweiterungen zu XQuery).

- API-Unterstützung
  Viele native XML Datenbanksysteme unterstützen die verschiedensten XML-relevanten APIs. Der einzige Ansatz einer Standardisierung ist die "XML:DB API" von der XML:DB Initiative. Viele proprietäre Produkte unterstützen jedoch eine eigene API.

- Indizes
  Wie aus relationalen Datenbanken bekannt, kann die Ausführungszeit von Anfragen durch Indizes erheblich verkürzt werden. In XML Datenbanksystemen haben sich drei Arten von Indizes etabliert: Werteindizes über Text- oder Attributwerte, Strukturindizes über bestimmte Elemente zur schnelleren Navigation und Volltextindizes zur Unterstützung von Volltextsuchen.

- Transaktionsmanagment
  Das Transaktionsmanagment ist in XML Datenbanksystemen etwas schwierig. Viele Operationen auf Knoten führen dazu, dass ganze Dokumente gesperrt sind. Diese grobe Granularität wird von den meisten Systemen geboten. Trotzdem unterstützen einige System Sperrungen auch auf Teilbaum oder sogar Knotenebene.

Speichermodelle

Um XML Dokumente in der Datenbank zu speichern gibt es zwei grundlegende Ansätze, textbasierte Speicherung und modellbasierte Speicherung.

Textbasierte Speicherung XML Dokumente als reinen Text zu Speichern hat einen großen Vorteil: Es können keine Informationen verloren gehen. Die Art der Speicherung geht hierbei von Dateien in einem Dateisystem, BLOBs oder CLOBs in relationalen Datenbanken bis zu proprietären Textformaten.
Die Speicherung als Text ermöglicht das anlegen von Indizes. Beim Abfragen ganzer Dokumente oder Dokumentfragmenten ergeben sich erhebliche Geschwindigkeitsvorteile, da die XML Dokumente nicht in viele kleine Teile aufgeteilt sind. In diesem Fall reicht ein einzelner Lesevorgang meist aus, um die benötigten Daten für die Anfragen zu erhalten. Nachteilig wirkt sich diese Struktur jedoch aus, wenn viele unterschiedliche Elemente abgefragt werden sollen, die über unterschiedliche Dokumente verteilt sind, oder wenn die Struktur der bestehenden Dokumente komplett verändert werden soll.

Modellbasierte Speicherung Der entgegengesetzte Ansatz ist der, jedes XML Dokument in ein internes Datenformat umzuwandeln und so abzuspeichern. Eine Baumstruktur wie DOM (Document Object Modell) würde sich anbieten. Diese Herangehensweise hat den Vorteil, dass XML Dokumente nicht erst aus Text geparst werden müssen, sondern schon in Element- oder Knotenform vorliegen. Das kann unter Umständen zu Performancevorteilen gegenüber textbasierten Speichermodellen führen.

3.2 X-Hive DB 6.0

X-Hive DB 6.0 ist ein kommerzielles Produkt der X-Hive Corporation aus Rotterdam. Hauptaugenmerk der Firma liegt auf XML Datenbanken und Contentmanagmentsystemen auf Basis von XML. Die X-Hive Corporation ist Mitglied des W3Cs und maßgeblich an Test und Entwicklung von XQuery beteiligt.

Die Datenbank ist in Java programmiert. Sie lässt sich stand-alone, als Client-Server-Applikation oder als Web-Service nutzen.

Angeboten werden verschiedene Lizenzmodelle – unter anderem ein Entwicklungslizenz mit der das System in dritte Anwendungen eingebaut werden kann, eine normale Nutzungslizenz und eine Lizenz für Zwecke wie Lehre und Forschung.

Die Informationen in diesem Artikel stammen aus dem X-Hive/DB Manual ([42]).

3.2.1 Datenbankfeatures

Abfrage- und Änderungssprachen

X-Hive unterstützt als Abfragesprachen XQuery bzw. XPath. Als Updatesprache wird XUpdate als Paket zur nachträglichen Installation angeboten. Darin ist die XUpdate-Referenzimplementierung Lexus enthalten.

Unterstützung von Standards

Folgende Technologien des W3C werden von X-Hive unterstützt bzw. implementiert.

- Document Object Model (DOM) Level 1
- DOM Level 2 (nur die Abschnitte "Core' und "Traversal")
- DOM Level 3 (nur die Abschnitte "Load/Save" und "Validation")
- XSLT
- XQuery
- XPath
- XLink
- XPointer

Weiterhin wird über ein separates Paket XUpdate von der XML:DB Initiative unterstützt.

API-Unterstützung

X-Hive/DB unterstützt eine eigene proprietäre API. Der grundlegende Ablauf der Nutzung dieser API ist folgender:

1. Erstellen eines Datenbanktreibers,
2. Erstellen einer Session,
3. Verbinden zur Datenbank,
4. Starten einer Transaktion,
5. Ausführen von DB-Operationen,
6. Abschließen der Transaktion (commit oder rollback) und
7. Schließen der Session und des Datenbanktreibers.

Die beiliegende HTML-Dokumentation bietet eine gute Einführung und Erklärung der API. Außerdem befindet sich eine JavaDoc-Dokumentation im Lieferumfang. Weitere Pakete z.b. für die Unterstützung von SOAP, WebDAV und J2EE Connector sind wie die XUpdate-Erweiterung über die Webseite der X-Hive Corp. zu beziehen.

3.2.2 Logische Struktur

Eine einzelne X-Hive Applikation besteht in der Regel aus mehreren Datenbanken. Die Summe aller Datenbanken wird als Federation bezeichnet. Eine laufende Instanz von X-Hive/DB bezieht sich immer auf eine solche Federation und die enthaltenen Datenbanken. Auf einem Rechner können also gleichzeitig mehrere Federations in Benutzung sein. Zu jeder Federation gehört ein Superuseraccount. Dieser Superuser kann Datenbanken anlegen und löschen sowie datenbankübergreifende Operationen (Backup) durchführen. Dieser Account hat keinen Zugriff auf die Daten innerhalb der einzelnen Datenbanken.
Zu jeder Datenbank gehören:

- Nutzer
- Nutzergruppen
- Indizes
- Libraries
- Dokumente
- Kataloge und Schemata

- Blobs (binary large objects)

Diese Elemente sind für jede Datenbank spezifisch. Nutzer lassen sich durch den Datenbankadministrator (wird beim Anlegen der Datenbank festgelegt) erstellen. Ihnen können verschiedene Rechte zugeteilt werden. Der Datenbankadministrator hat kompletten Zugriff auf "seine" Datenbank und die zugehörigen Nutzer bzw. Nutzergruppen.
Nutzer lassen sich auch in Gruppen zusammenfassen, um eine einfachere Rechteverwaltung zu ermöglichen.
Indizes beschleunigen die Suche in bestimmten Dokumenten. X-Hive/DB unterstützt folgende Indexarten: jeweils einen Id- bzw. Namensindex für Libraries, AttributIdindex, Werteindex, Elementnamensindex bzw. Elementnamensindex für bestimmte Elementnamen und Volltextindex.
Jede Datenbank besteht aus mindestens einer Library (der `root-library`). Diese Library kann Dokumente bzw. weitere Libraries enthalten (wie jede andere Library auch). Die Struktur ist vergleichbar mit einem Dateisystem.
Für jede Library kann ein so genannter Katalog angelegt werden, der DTDs und XML Schemata aufnimmt.

3.2.3 Interne (Speicher-)architektur

Die interne Speicherstrategie von X-Hive/DB basiert auf der objektorientierten Datenbank Objectivity/DB (diese Information stammt aus dem Jahre 2002 und wurde [35] entnommen). In welcher Form die XML Dokumente in Objekte gemappt werden ist dem Autor dieser Arbeit leider nicht bekannt.
Die folgenden Informationen wurden dem "X-Hive/DB 6.0 - Manual" entnommen, das der Datenbank in HTML-Form beiliegt.

logische Speicherstruktur: Segmente

Die Speicherung der verschiedenen Datenbanken erfolgt in Segmenten. Das sind logische Speichereinheiten für die Daten der Datenbank. Jede Datenbank besitzt mindestens ein Segment (das `default` Segment). Dieses kann nur entfernt werden, wenn die gesamte Datenbank gelöscht wird. Das `default` Segment wird während der Erstellung der Datenbank angelegt.

physikalische Speicherstruktur: Dateien

Zu jedem Segment gehören eine oder mehr Dateien. Ähnlich wie bei den Segmenten, gibt es eine `default` Datei, die bei Erstellung des Segmentes angelegt wird. Jede Datei besteht aus verschiedenen Seiten (pages), deren Größe bei Erstellung der Federation festgelegt wird. Eine Seite wird immer in genau einer Datei gespeichert, verschiedene Seiten eines Dokumentes können aber auch auf mehrere Seiten verteilt werden.

Aus Performancegründen ist es sinnvoll, die Seitengröße bei der Erstellung der Federation auf einen Wert gleich der Clustergröße (unter Windows) bzw. der Blockgröße (unter Linux) zu setzen. Wenn allerdings viele kleinere Dokumente in die Datenbank aufgenommen werden sollen, sollte eine kleinere Seitengröße gewählt werden, um nicht unnötig Speicherplatz zu belegen.

3.2.4 XQuery in X-Hive/DB

Die Spezifikation von XQuery ist nicht in allen Belangen bindend. Einige Features müssen nicht implementiert werden bzw. ist nicht festgelegt, wie sie implementiert werden sollen. Zusätzlich kann jede Datenbank natürlich noch proprietäre Erweiterungen implementieren, um den Funktionsumfang zu erweitern.

Erweiterung der Funktionsbibliothek

X-Hive/DB hat die Funktionsbibliothek von XQuery noch um einige Funktionen ergänzt. Diese Funktionen sind mit dem Namensraum `http://www.x-hive.com/2001/08/xquery-functions` asoziiert. Standardmäßig wird dieser Namensraum durch das Präfix `xhive:` repräsentiert. Hier werden nur drei dieser zusätzlichen Funktionen beschrieben, weitere Informationen kann man dem Handbuch [42] entnehmen.

xhive:evaluate() Die `evaluate` Funktion dient dazu, aus einem Query heraus einen anderen Query auszuwerten, der als Zeichenkette als Argument übergeben wird. Sinnvoll ist diese Funktion z.b. bei XML Dokumenten, die mehrere Queries enthalten, die nacheinander ausgeführt werden sollen.

```
1  for $query in document("queries.xml")/queries/query/text()
   return
   <result>
     xhive:evaluate($query)
5  </result>
```

xhive:java() Die Funktion `java()` ruft eine beliebige Javafunktion auf. Diese Funktion kann separat implementiert werden. Als erstes Argument wird die Klasse übergeben, die die Funktion enthält, folgen können noch beliebig viele andere Parameter, die der Javafunktion als Parameter weitergegeben werden. Der Implementiervorgang ist in der Dokumentation zu X-Hive/DB [42] kurz angedeutet und wird in der ebenfalls beiliegenden Javadoc genau beschrieben.

xhive:setoptions() Mit dieser Funktion besteht die Möglichkeit, während der Auswertung eines Queries Einfluss auf die zugrunde liegende Datenbank zu nehmen. Als Parameter erwartet die Funktion zwei Zeichenketten als Optionsname bzw. -wert. Eine Auflistung aller unterstützten Optionen befindet sich im Handbuch [42].

Einschränkungen des XQuery-Sprachumfangs

In diesem Abschnitt sollen die Features oder Sprachkonzepten von XQuery aufgelistet werden, die nicht in X-Hive/DB implementiert sind. Einige dieser Feature sind als implementationsabhängig in der XQueryspezifikation aufgeführt, andere sollten eigentlich zum Sprachumfang gehören.
Sprachkonzepte, die zum Sprachumfang von XQuery dazugehören sollten:

- Variablendeklarationen im Prolog
  X-Hive/DB unterstützt keine Deklarationen von Variablen im Prolog. Variablen müssen also innerhalb des Skriptkörpers eingeführt werden (z.b. mit dem Schlüsselwort `let`).

- `castable as` Ausdruck
  Der `castable as` Ausdruck prüft, ob ein Argument (vor dem Schlüsselwort) in einen Typ gecastet werden kann, der nach dem Schlüsselwort notiert wird. X-Hive unterstützt dieses Ausdruck nicht.

Folgende Funktionen, Typen und Operationen sind zwar in der XQuery- bzw. XPathspezifikation vorgesehen, aber werden von X-Hive/DB nicht unterstützt:

- xs:base64Binary
- xs:hexBinary
- op:base64Binary-equal
- op:hexBinary-equal
- fn:default-collation
- fn:document-uri
- fn:implicit-timezone
- fn:trace

Die Spezifikation von XQuery bietet einige optionale Features. Hier ist eine Auflistung der optionalen Features, die X-Hive/DB nicht unterstützt:

- Pragmas
- Module
- Schemaimport und andere XML Schema Features

3.3 eXist XML Database

Die eXist XML Database ist ein Open Source Projekt, dass von Wolfgang Meier ins Leben gerufen wurde. Die gesamte Datenbank ist in Java geschrieben. Sie kann sowohl als stand-alone Datenbankserver, als eingebettete Datenbankbibliothek oder als Servlet innerhalb eines Webservers laufen. Für letztere Variante wird der HTTP-Server und Servletcontainer Jetty mitgeliefert.

3.3.1 Datenbankfeatures

Abfrage- und Änderungssprachen

Als Abfragesprachen wird von eXist XQuery bzw. XPath unterstützt. XUpdate dient als DML-Komponente und ist bereits integriert.

Unterstützung von Standards

Folgende Standards des W3C werden von eXist unterstützt:

- XQuery
- XPath
- XInclude
- XPointer
- XSL/XSLT

eXist unterstützt außerdem XUpdate von der XML:DB Initiative als Datenmanipulationskomponente.

API-Unterstützung

eXist unterstützt die XML:DB-API der XML:DB Initiative. Darin enthalten ist die Unterstützung für SAX und DOM. Die Treiber zur Unterstützung der XML:DB-API und deren Erweiterungen (XQueryausführung, Nutzerverwaltung, Indexverwaltung, etc.) nutzen XML-RPC. Wenn eXist im Servletmodus betrieben wird, steht SOAP zur Verfügung. Zur Zeit wird an einer kompletten WebDAV-Unterstützung gearbeitet.

3.3.2 Logische Struktur

Folgende Komponenten gehören zum logischen Datenmodell der eXist Datenbank:

- Collections

- Dokumente
- Nutzer
- Nutzergruppen
- Indizes
- Blobs (binary large objects)

Die Organisation der eXist Datenbank basiert auf Collections. Jede Collection kann Dokumente und andere Collections enthalten.
Das Nutzer- und Zugriffsmanagment ist dem von Unix nachempfunden. Nutzer können einer Nutzergruppe zugeordnet werden. Jede Collection und jedes Dokument hat einen Nutzer als "Besitzer". Die Zugriffsberechtigungen sind in read-, write- und update-Zugriff jeweils für den Nutzer selbst, die Nutzergruppe und alle Nutzer eingeteilt. Die voreingestellte Gruppe `dba` enthält alle Administratorrechte, um Nutzerdaten und Datenbankinhalte zu ändern.
Die Indexstruktur in eXist stellt Strukturindizes für Elemente und Attribute und Volltextindizes für Textknoten und Attributwerte zur Verfügung.

3.3.3 Interne Architektur

eXist verwendet zur Speicherung aller enthaltenen Dokumente vier Indexdateien. Die eigentlichen Daten liegen alle in der Datei `dom.dbx`. Die verschiedenen Knoten werden nach dem Datenmodell von DOM erstellt und gespeichert. Die Datei `collections.dbx` enthält Indizes für Collections und Dokumente. Jedes Dokument bzw. jede Collection, die der Datenbank hinzugefügt wird, wird automatisch indiziert. Die Datei `elements.dbx` enthält Indizes für Elemente und Attribute, die Datei `words.dbx` enthält alle Volltextindizes.

Die Struktur aller Indexdateien basiert auf B+ Bäumen. Für jede Collection wird in jeder Datei ein B+ Baum angelegt. Das ist insofern ungewöhnlich, dass viele andere XML Datenbanksysteme die Indizes dokumentweise anlegen. Durch die collectionweise Zusammenfassung wird die Zahl der Bäume jedoch niedrig gehalten, was der Performance zugute kommt.
Die Indizes der einzelnen Baumelemente basieren auf einem Zahlenschema und lassen sich direkt in die Speicherposition abbilden. Deshalb ist es nicht notwendig, Verbindungen zwischen verschiedenen Knoten (z.b. Eltern-Kindbeziehungen) explizit anzugeben, was zu Speichereinsparungen führt. Die Baumstruktur der DOM-Elemente lässt sich so aufgrund der Indexstruktur eindeutig wiederherstellen.

3.3.4 XQuery in eXist

Wie auch in X-Hive sind in eXist einige Features von XQuery nicht oder nicht vollständig implementiert bzw. neue Features implementiert, die nicht zum Standardsprachumfang von XQuery gehören.

Erweiterungen für Volltextsuche

eXist verfügt über eine sehr ausgefeilte Indexstruktur sowohl für Struktur- als auch für Volltextindizes. Zur vollen Ausnutzung der Volltextfeatures von eXist sind ein Reihe von Operatoren und Funktionen implementiert, die nicht in den Spezifikationen auftauchen.

Operatoren Zwei neue Operatoren für die Volltextsuche sind in eXist definiert: `&=` und `|=`. Links der Operatoren steht eine Knotenmenge und rechts von ihnen eine Zeichenkette, die zu suchende Wörter getrennt durch Leerzeichen enthält. Der erste Operator liefert alle Knoten aus der Knotenmenge, die alle Wörter in der Zeichenkette in beliebiger Reihenfolge enthalten. Der zweite Operator verhält sich analog, wenn eines der Suchwörter enthalten ist.
Innerhalb der Suchwortzeichenkette sind einfache reguläre Ausdrücke erlaubt (Kardinalitäten `?` und `*` bzw. Auswahl `[abc]`).

Funktionen Eine etwas komfortablere Version der zusätzlichen Operatoren wird durch einige zusätzliche Funktionen geboten.

near() Die Funktion `near` hat ein ähnliches Verhalten wie der Operator `&=`. Sie erwartet zwei optional auch drei Argumente, wobei das erste Argument eine Knotenliste ist und das zweite wiederum eine Zeichenkette mit den Suchbegriffen. Bei Verwendung der `near` Funktion ist jedoch die Reihenfolge der Suchbegriffe entscheidend. Das optionale dritte Argument kann eine Zahl enthalten, die die Anzahl der Worte angibt, die maximal zwischen zwei Suchbegriffen stehen dürfen.

match-all()/match-any() Diese beiden Funktionen erlauben die Suche nach regulären Ausdrücken. Sie unterstützen wesentlich mehr Formulierungen als die oben beschriebenen Operatoren. Eine genaue Beschreibung findet man unter [40].

Erweiterung der Funktionsbibliothek

eXist definiert unheimlich viele Funktionen, die nicht im Standardsprachumfang enthalten sind. Eine komplette Auflistung aller Funktionen, die in eXist implementiert sind findet sich unter [41].

XMLDB Funktionen Die hier erläuterten Funktionen gehören alle zum Namensraum `http://exist-db.org/xquery/xmldb`. Mit Hilfe dieser Funktionen können direkt Änderungen an der Datenbank auf Collection- bzw. Dokumentebene vorgenommen werden.

register-database() Da diese Funktionen auf der XML:DB-API basieren, werden sie auch ähnlich verwendet. Die `register-database()` Funktion erstellt einen Datenbanktreiber, ähnlich wie er in einem Javaprogramm initialisiert werden würde.

collection() Die Funktion `collection()` wählt die entsprechende Collection innerhalb von eXist aus. Als Argument werden ein String mit dem Pfad zur gewünschten Collection, ein Nutzername und das entsprechende Passwort als Zeichenkette übergeben. Der Pfad zur Collection basiert wiederum auf der XML:DB und ist von der Form "xmldb:exist:///db", wird also notiert, als wenn der Zugriff von Außerhalb stattfinden würde.

create-collection() Die `create-collection()` Funktion erstellt eine neue Collection, dabei werden zwei Argumente übergeben. Als erstes die "Vatercollection" und als zweites den Namen der neuen Collection.
Diese Funktion liefert eine Referenz auf ein Javaobjekt, das in XQuery nicht weiter bearbeitet werden kann. Dieses Objekt enthält wiederum eine Referenz auf die erstellte Collection und kann als Argument an andere Funktionen übergeben werden.

store() Diese Funktion speichert das Ergebnis eines Ausdrucks in die Datenbank. Sie erwartet drei Argumente: Ein Collectionobjekt, wie es von der `create-collection` Funktion geliefert wird, einen Namen für das neu anzulegende XML Dokument (z.b. "abc.xml") und natürlich den Inhalt des neuen Dokuments.

Utility Funktionen Die Utilityfunktionen von eXist gehören zum Namensraum `http://exist-db.org/xquery/util`. Die Funktionen definieren eine Reihe von Funktionalitäten, die in Programmiersprachen unter Umständen zur Standardbibliothek gehören.

md5() Diese Funktion berechnet einen MD5-Hashwert für jedes beliebige Element, das als Argument übergeben wird.

eval() Die `eval` Funktion führt dynamisch einen XQuery aus. Siehe auch `evaluate` Funktion von X-Hive/DB.

exclusive-lock() bzw. shared-lock() Diese Funktionen erwarten zwei Argumente: eine Knotenmenge und einen beliebigen Ausdruck. Alle Dokumente, die einen der Knoten aus der ersten Knotensequenz enthalten werden gesperrt. Dann wird der Ausdruck ausgewertet und danach die Sperrung wieder zurückgenommen.

HTTP Funktionen In eXist sind Funktionen definiert, mit denen Zugriff auf HTTP Funktionalitäten möglich ist. In Verbindung mit dem XSLT-Feature ist es somit möglich ganze Webanwendungen nur mit eXist, XSLT und XQuery zu implementieren. Der Namensraum, mit dem diese Funktionen assoziiert sind, ist `http://exist-db.org/xquery/request`.

create-session() Diese Funktion erstellt eine HTTP Session, wenn noch keine aktiv ist.

session-attributes() Liefert eine Liste aller gespeicherter Sessionattribute.

get-session-attribute() bzw. set-session-attribute() Mit der `get-...` Funktion kann der Wert eines Sessionattributes abgefragt werden (Attributbezeichner als Argument). Mit der `set-...` Funktion kann ein Bezeichner - Wert Paar als Sessionattribut speichern.

get-request-data() Liefert den Inhalt einer POST - Anfrage als Zeichenkette.

Einschränkungen des XQuery-Sprachumfangs

eXist unterstützt nicht den gesamten Sprachumfang, der durch die XQueryspezifikation beschrieben ist. eXist bietet keine Unterstützung der folgenden Schlüsselwörter:

- `typeswitch`
- `treat as`
- `instance of`
- `castable as`

Diese vier Schlüsselworte gehören in ihrer Funktionalität in den selben Bereich. Sie bearbeiten Elemente im Zusammenhang mit Datentypen, entweder werden Elemente gecastet oder der Typ wird bestimmt.

Weiterhin bietet eXist – ähnlich wie X-Hive – keine Unterstützung für Schemafeatures. Damit werden folgende Schlüsselwörter nicht unterstützt:

- `validate`
- `import schema`
- `declare validation`

Die Implementierung der Schemafeatures ist jedoch bereits in Arbeit. Zur Zeit werden Knoten intern immer als `xs:untypedAtomic`, dem allgemeinsten einfachen Datentyp in XML Schema bzw. XQuery, behandelt.

Im Typsystem der XQueryimplementierung von eXist fehlen außerdem die Typen:

- xs:gYear,
- xs:gMonth,
- xs:gDay,
- xs:gYearMonth und
- xs:gMonthDay.

Als wichtigstes Feature, das in eXist nicht implementiert wurde, gehört sicher die fehlende Unterstützung für direkte Elementkonstruktoren. Die Elementkonstruktoren können natürlich normal verwendet werden, allerdings werden sie wie Text behandelt und können damit nicht als Ausgangsmenge für weitere Queryverarbeitung dienen. Zur Zeit ist die Queryausführung auf dem DOM-Modell im Speicher nicht so möglich, wie das beim persistenten DOM-Baum der Fall ist.

Kapitel 4

Benchmark von XML DBS

Im Rahmen dieser Arbeit wurden die beiden XML Datenbanksysteme, die im vorherigen Kapitel kurz vorgestellt wurden, auf Ihre Anfrageverarbeitungsleistung hin getestet. Verwendet wurde dazu der Benchmark XMach-1.

Benchmarks für XML Datenbanksysteme müssen mehrere Anwendungsdomänen abdecken. Durch die verschiedenartige Struktur von XML Dokumenten sind sehr unterschiedliche Abfrageoperationen vonnöten. Man kann grob in drei solcher Domänen einteilen:

- Anfragen über dokumentzentrierten XML Dokumenten,
- Anfragen über datenzentrierten XML Dokumenten und
- Anfragen über gemischten Daten in XML Dokumenten.

Erstere beiden Datentypen wurden bereits im vorhergehenden Kapitel beschrieben. Es ist relativ einfach die Abfrageengine auf einen dieser beiden Fälle hin zu optimieren. Der dritte Fall geht jedoch von Daten aus, die beiden ersteren Domänen zugeordnet werden können.
Anfragen über dokumentzentrierten Daten beinhalten Operationen zur Navigation in die Tiefe als auch in die Breite, zur Volltextsuche in Elementinhalten und Operationen zum Auslesen ganzer Dokumente bzw. ganzer Dokumentfragmente ohne die Hierarchie der Daten zu verändern. Im Gegensatz dazu umfassen Anfragen über datenzentrierten Dokumenten andere Operationen wie z.b. Suche über Attributwerten, Sortieren, Umordnen und Joinen mehrerer Dokumente oder Dokumentfragmente und Umstrukturierung der Ergebnismengen in neue Klassen von Dokumenten. Ein Benchmark muss Operationen und Daten für alle diese Fälle bereitstellen, um XML Datenbanksysteme vergleichbar in ihrer Anfrageverarbeitungsleistung zu machen.

4.1 Beschreibung von XMach-1

Der Benchmark XMach-1 wurde an der Universität Leipzig von Thimo Böhme und Erhard Rahm entwickelt. Das Ziel der Entwicklung von XMach-1 war einen Benchmark zu schaffen, der die einfachen Operationen auf XML Daten auf die Performance hin testen. Weiterführende Features, die durch Abfragesprachen wie XQuery definiert sind, sind nicht berücksichtigt, da zur Zeit der Entwicklung von XMach-1 viele Datenbanksysteme über keine vollständige Implementierung des XQuery-Standards verfügten. (Dieser und folgende Abschnitte: vgl. [38])

4.1.1 Struktur des Benchmark

Der Benchmark XMach-1 versucht ein Typisches Anwendungsszenario für XML DBS zu modellieren. Der Benchmark wurde als Webapplikation implementiert. Als Server fungieren hierbei das zu testende Datenbanksystem sowie, das über ein Datenbankmodul angesprochen wird. Die spezifischen Anfragen werden von Clients erstellt und an dieses Datenbankmodul übertragen. Damit wird ein Intra- oder Internetserver simuliert, der XML Dokumente in einer Datenbank zur Verfügung stellt.

4.1.2 Datenbasis

Die Anwendungsdomäne für die XMach-1 entwickelt wurde, sind die gemischten Daten. Die Datenbasis für den Benchmark besteht sowohl aus daten- als auch aus dokumentzentrierten Dokumenten. Der Benchmark enthält eine große Menge an dokumentzentrierten Dokumenten, die Fließtext enthalten, und ein datenzentriertes Dokument (directory), das Metadaten über alle Textdokumente des Benchmark enthält.

Das directory-Dokument

Die Struktur des `directory` Dokumentes ist durch folgende DTD beschrieben:

```
<!ELEMENT directory (host+) >
<!ELEMENT host (host+ | path+) >
<!ATTLIST host
  name CDATA #REQUIRED >
<!ELEMENT path (path+ | doc_info) >
<!ATTLIST path
  name CDATA #REQUIRED >
<!ELEMENT doc_info EMPTY >
<!ATTLIST doc_info
  doc_id ID #REQUIRED
  loader CDATA #REQUIRED
  insert_time NMTOKEN #REQUIRED
  update_time NMTOKEN #IMPLIED >
```

Die einzelnen Textdokumente werden nach imaginären URIs sortiert. Alle Textdokumente werden in dieser Hierarchie durch ein `doc_info` Element repräsentiert, dass einem Host zugeordnet ist, auf dem sie über einen bestimmten Pfad erreichbar sind. Jede Hostadresse besteht aus einer dreistufigen Hierarchie von `host` Elementen, die jeweils durch ein `name` Attribut identifiziert werden. Auf der obersten Hierarchiestufe wird zwischen drei verschiedenen Hosts unterschieden, in der zweiten Hierarchiestufe zwischen zehn Hosts und in der dritten Stufe zwischen fünf.
Unterhalb der Hostadressen folgt eine Pfadhierarchie, die eine Tiefe von einem bis vier `path` Elementen aufweist. Die `path` Elemente werden wiederum durch ein `name` Attribut identifiziert. Das unterste dieser `path` Elemente hat als Namen jeweils einen XML Dateinamen. Innerhalb dieses Pfadelementes ist ein `doc_info` Element enthalten, mit Metainformationen zum jeweiligen Dokument in Form von Attributen (`doc_id`, `loader`, `insert-time` und `update-time`; letzteres ist optional). Über den Wert des `doc_id` Attributes werden die einzelnen Dokumente identifiziert.

Die document-Dokumente

Die `document` Dokumente sind durch folgende DTD beschrieben:

```
<!ELEMENT document (title, chapter+)>
<!ATTLIST document
  author CDATA #IMPLIED
  doc_id ID #IMPLIED>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT chapter (author?, head, section+)>
<!ATTLIST chapter
  id ID #REQUIRED>
<!ELEMENT section (head, paragraph+, section*)>
<!ATTLIST section
  id ID #REQUIRED>
<!ELEMENT head (#PCDATA)>
<!ELEMENT paragraph (#PCDATA | link)*>
<!ELEMENT link EMPTY>
<!ATTLIST link
  xlink:type (simple) #FIXED "simple"
  xlink:href CDATA #REQUIRED>
```

Jedes Dokument hat eine eindeutige Id. Die Dokumente bestehen jeweils aus mehreren `chapter`, `section` und `paragraph` Elementen, die ineinander verschachtelt sind. Zusätzlich sind noch Informationen über Titel und Autor enthalten. Untereinander können die Dokumente mittels XLink aufeinander verweisen.

4.1.3 Die Benchmarkqueries

Der Benchmark Xmach-1 umfasst acht verschiedene Anfrageoperationen:

Anfrageoperation 1 Diese Anfrage soll ein Dokument auswählen, dass unter einem ganz bestimmten Pfad zu finden ist, und es komplett ausgeben. Dieser Fall testet, wie schnell ein gesamtes Dokument wiederhergestellt werden kann.

```
1
      /host[@name='chost5']/path[@name='apath2']/path[@name='bpath2']
      /path[@name='cpath3']/path[@name='d883.xml']/doc_info/@doc_id,
      $b := /*[@doc_id = $a]
5   return
      $b
```

Parametrisiert sind jeweils die Werte für Host- und Pfadangaben. Die Anzahl der Pfadangaben kann variieren.

Anfrageoperation 2 Diese Anfrage liefert die DokumentIds aller Dokumente, die eine bestimmte Textphrase enthalten. Diese Anfrage zeigt, wie schnell eine Volltextsuche durchgeführt werden kann.

```
1   for $a in /*[@doc_id]
    where
      some $p in $a//paragraph
      satisfies contains($p, 'tried gonzalez lens')
5   return
      distinct-values($a/@doc_id)
```

Parametrisiert ist hierbei die gesuchte Phrase.

Anfrageoperation 3 Diese Anfrage navigiert rekursiv jeweils zum ersten `section` Element und liefert dabei das tiefste `section` Element. Diese Operation simuliert eine sequentielle Navigation.

```
1   declare function deepestFirstSection ($e as element()) as element() {
      if (empty($e/section2))
        then ($e)
        else deepestFirstSection($e/section2[1])
5   };
    deepestFirstSection(/document2[@doc_id='d12']/chapter2[1]/section2[1])
```

Parametrisiert sind in dieser Anfrage das Suffix des `document` und `chapter` Elementes und der `section` Elemente, sowie die DokumentId.

Anfrageoperation 4 Diese Anfrage liefert eine Liste aller `head` Elemente eines Dokumentes, die Kind eines `section` Elementes sind. Getestet wird hierbei die Fähigkeit, Knoten umzustrukturieren.

```
1   for $a in /document2[@doc_id='d12']//section2/head2
    return
      $a
```

Parametrisiert sind hierbei die DokumentId, sowie das Suffix von `documten`, `section` und `head`.

Anfrageoperation 5 Dieser Query liefert die Namen aller Dokumente unterhalb eines bestimmten Hosts. Hier wird Geschwindigkeit überprüft, mit der durch die Baumstruktur von strukturierten, ungeordneten Daten navigiert wird.

```
1  for $a in /directory/host[@name='ahost2']/host[@name='bhost6']
       /host[@name='chost3']//path[doc_info]
   return
     $a/@name
```

Parametrisiert sind die Angaben zum Host.

Anfrageoperation 6 Diese Anfrage liefert die DokumentId und die Id des Elternknotens eines `author` Elementes mit bestimmtem Inhalt (Autorenname). Mit dieser Anfrage soll die Effizienz der Indexstruktur getestet werden.

```
1  for $a in /*[@doc_id]
   where $a//*/author='Levi Deedee'
   return
     <document>
5      <doc_id>{string($a/@doc_id)}</doc_id> {
         for $e in $a//*[author='Levi Deedee']
         return
           <parent_id> { string($e/@id) } </parent_id>
       }
10   </document>
```

Parametrisiert ist hier der Autorenname.

Anfrageoperation 7 Diese Anfrage liefert alle DokumentIds derjenigen Dokumente, auf die vier oder mehrmals verwiesen wird.

```
1  declare namespace xlink='http://www.w3.org/1999/xlink';
   for $refID in distinct-values(/*//link/@xlink:href)
   let $refDocs := distinct-values(/*[.//link/@xlink:href=$refID]/@doc_id)
   where count($refDocs) >= 4
5  return
     <docid>{string($refID)}</docid>
```

Anfrageoperation 8 Diese Anfrage liefert die DokumentId der 100 zuletzt geänderten Dokumente, die ein Attribut `author` haben. Die Dauer dieser Anfrage hängt besonders von der Effizienz

```
1  let $b := (
     for $a in /directory//doc_info[@update_time]
     where not(empty(/*[@doc_id=$a/@doc_id]/@author))
     order by $a/@update_time descending
5    return
       <docid>{string($a/@doc_id)}</docid>)
   return subsequence($b, 1, 100)
```

Der Benchmark XMach-1 umfasst eigentlich noch drei weitere Operationen, die die Updateleistung prüfen sollen. Da diese Arbeit sich jedoch vorwiegend mit XQuery beschäftigt, wurden die Operationen im Rahmen dieser Arbeit nicht beachtet.

Operationszusammenstellung

Um einen Vergleichswert zwischen den Messungen an verschiedenen Datenbanken zu erhalten, müssen bei jedem Benchmarkdurchlauf ähnliche Verhältnisse zwischen den Operationen vorliegen.
XMach-1 definiert folgende Häufigkeitsverhältnisse für die verschiedenen Operationen (angegeben in Prozent von allen Operationen):

- Query 1: 30%
- Query 2: 20%
- Query 3-6: je 10%
- Query 7+8: je 4%

Die Summe der Werte ergeben nicht 100%, da die Updateoperationen vernachlässigt wurden.
Ein Durchlauf des Benchmarks umfasst 400 Operationen, womit Q1 120 mal, Q2 80 mal, Q3 bis Q6 40 mal und Q7 bzw. Q8 je 16 mal durchgeführt wird. Auf die Updateoperationen würden 8 Operationen entfallen.

4.2 Implementierung der Datenbankmodule

Der XMach-1 Benchmark ist ein Client/Serversystem, das Datenbankabfragen über ein Intra- oder Internet simuliert. Der serverseitige Teil wird als Servlet ausgeführt, während der clientseitige Teil als Javaapplikation gestartet läuft.
Auf Clientseite werden jeweils nur die Operationen ausgewählt und entsprechende Parameter für die Abfrage generiert und an den Server geschickt. Auf Serverseite werden diese Parameter in eine passende Abfragesprache –in diesem Fall XQuery– umgewandelt und die Queries ausgeführt.
Die serverseitige Kommunikation übernimmt eine Klasse namens `GenericServer`. Diese Klasse erstellt ein Objekt des entsprechenden Datenbankmoduls. Jedes Datenbankmodul implementiert das Interface `QueryInterface`, das die entsprechenden Methoden definiert, mit denen die Queries ausgeführt werden. Für jeden Query ist eine Methode `doQueryN()` zu implementieren, wobei N die Nummer des Query ist. Die Parameter werden über eine HashMap übergeben. Sind keine Parameter für den Query vorgesehen, ist die HashMap leer.
Hier eine Auflistung aller auftretenden Parameter, die

Query	Schlüssel	Bedeutung
1	ahost, bhost, chost	Teile des Hosts
	apath, bpath, cpath	treten abhängig von der Pfadtiefe auf
	name	der Dateiname des Dokumentes
2	phrase	die gesuchte Textphrase
3	suffix	das Suffix für die einzelnen Elemente
	docid	die DokumentId des surchsuchten Dokumentes
4	suffix	das Suffix für die einzelnen Elemente
	docid	die DokumentId des durchsuchten Dokumentes
5	ahost, bhost, chost	Teile des Hosts
6	author	Name des gesuchten Autoren

Für die Queries 7 und 8 werden keine Parameter benötigt.

Aus den einzelnen `doQueryN()` Methoden wird jeweils eine Methode `doQuery()` aufgerufen, die als Parameter eine Zeichenkette bekommt, die den fertigen XQuery enthält. Diese Methode führt die eigentliche Datenbankanfrage durch.
Für die Anbindung der Datenbank ist außerdem noch die Methoden `init()` von Bedeutung. Diese Methode stellt die Verbindung zum Datenbankserver her und legt die Zugangsinformationen wie z.B. Accountname und Passwort fest, damit diese zur Verfügung stehen, wenn die Queries ausgeführt werden.

4.3 Ergebnisse

Die Queries wurden in beiden Datenbanken über 1000 generierten Dokumenten (plus `directory` Dokument) ausgeführt. Die Daten wurden nach der Generierung erst als Dateien abgespeichert und dann separat in die Datenbanken importiert.

4.3.1 X-Hive/DB

Der Benchmark der X-Hive/DB ergab folgende Werte (in Millisekunden):

Query	min	max	avg
1	191	871	624
2	2093	2563	2198
3	160	420	304
4	190	430	317
5	190	401	319
6	1652	5658	1845
8	11636	13049	12131

Die Gesamtlaufzeit betrug 578 Sekunden für alle Operationen.

4.3.2 eXist

Der Benchmark der eXist Datenbank ergab folgende Werte (in Millisekunden):

Query	min	max	avg
1	100	2234	226
2	14681	38545	19100
3	10	561	60
4	20	871	132
5	20	200	45
6	140	1702	358
8	1843	9974	2944

Die Gesamtlaufzeit betrug 1648 Sekunden für alle Operationen.

4.3.3 Auswertung

Urteilt man nur nach den Gesamtausführungszeiten, liegt die X-Hive/DB klar vor der eXist XML Database. eXist brauchte mit 1648 Sekunden fast drei mal so lange wie die X-Hive/DB.

Schaut man sich jedoch die Ergebnisse der einzelnen Queries an, entsteht ein anderes Bild: Die eXist Datenbank liegt abgesehen von Q2 bei allen Anfragen

klar vorn. Q2 ist eine Volltextsuche in allen Dokumenten. Bei beiden Datenbanken wurden exakt die selben Queries verwendet. Der Volltextindex, den eXist anlegt, wird aber nur über spezielle Funktionen von eXist genutzt, die nicht zum Standard von XPath/XQuery gehören. X-Hive nutzt den Volltextindex hingegen auch mit den Standardfunktionen. eXist könnte hier sicher bessere Ergebnisse erzielen, wenn dieser spezielle Query mit den eXist-eigenen Funktionen optimiert werden würde.

Alle anderen Queries wurden von der eXist Datenbank meist um ein Vielfaches schneller abgearbeitet. Bei genauer Betrachtung der Implementierung des X-Hive Datenbankmoduls fällt auf, dass für jede Anfrage eine neue Session erstellt wurde, was sicher eine sehr teure Operation ist. Doch selbst wenn man einen konstanten Wert bei allen Queries abziehen würde, um diesen Nachteil auszugleichen, kommt eXist bei vielen Operationen noch weit schneller mit der Verarbeitung der Queries voran.
Der Grund hierfür könnte die automatische Indexerstellung der eXist Datenbank sein. Hier werden Indizes aufgrund des Speichermodells immer angelegt und werden deshalb auch bei der Auswertung von Anfragen genutzt. Die X-Hive/DB hingegen legt automatisch keine Indizes an, was die langen Ausführungszeiten erklärt.

Als Ergebnis des Benchmarks muss leider gesagt werden, dass die Ergebnisse kaum eine verlässliche Aussage über die Leistung der Datenbanken erlauben.

Query 7

Für beide Datenbanken sind keine Werte für die Ausführungszeit der siebten Anfrage aufgenommen worden. Die Ausführungszeiten waren bei beiden DBS so hoch, dass sie nicht mehr vom Benchmark erfasst werden konnten. Bei einem separaten Versuch, die siebte Anfrage durchzuführen, brauchten beide DBS fast eine halbe Stunde. Der Grund hierfür liegt in der Formulierung der Anfrage. Für jedes der 1000 Dokumente müssen alle 1000 Dokumente nach Links durchsucht werden.

4.3.4 Testsystem

Der Benchmark wurde auf einem Rechner mit Pentium 4 2,6 GHz und 512 MB Arbeitsspeicher ausgeführt. Als Betriebssystem kam Windows XP Home zum Einsatz. Der Benchmarkserver wurde innerhalb von Tomcat 5.0 ausgeführt. Die installierte Javaversion ist 1.4.2.

Kapitel 5

Zusammenfassung

Die Arbeit gibt einen Überblick über XML, XML-relevante Technologien wie XML Schema, sowie XML-datenbankspezifische Sprachen und Technologien wie XPath, XQuery und XUpdate. Die Beschreibung dieser Sprachen fasst grundlegendes Wissen zusammen, um produktiv mit XML Datenbanksystemen arbeiten zu können.

Im Anschluss daran wurde eine grobe Einführung in native XML Datenbanksysteme gegeben. Hierbei wurden erst allgemeine Informationen über verschiedene Architekturen, Speicherstrukturen und Features eingeführt.
Diese allgemeinen Betrachtungen wurden dann anhand zweier Datenbanksysteme am Fallbeispiel diskutiert. Die diskutierten XML Datenbanksysteme waren "X-Hive/DB" ein kommerzielles Projekt der X-Hive Corp. und "eXist XML Database" ein Opensourceprojekt. Beide Datenbaksysteme sind ausgereift, wobei eXist als Opensourceprojekt wahrscheinlich nie wirklich fertig implementiert sein wird.

Im Anschluss an die Vorstellung der beiden XML Datenbanksysteme wurde XMach-1 erläutert. XMach-1 ist ein Benchmark, der speziell für XML Datenbanksysteme entwickelt wurde. Der Benchmark wurde beschrieben und an beiden Datenbanken durchgeführt.

W3C-Recommendations und Working-Drafts

[01] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, F. Yergeau, J. Cowan:
Extensible Markup Language (XML) 1.1;
http://www.w3.org/TR/2004/REC-xml11-20040204/; 10.08.2004
Die aktuelle W3C-Empfehlung für XML 1.1.

[02] T. Bray, D. Hollander, A. Layman:
Namespaces in XML 1.1;
http://www.w3.org/TR/2004/REC-xml-names11-20040204/;
10.08.2004
Die aktuelle W3C-Empfehlung für XML-Namensräume 1.1.

[03] Fallside D.C.:
XML Schema Part 0: Primer;
http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/;
10.08.2004
Einführung in XML Schema, nichtnormativer Teil der XML Schema-Recommendation.

[04] H.S. Thompson, D. Beech, M. Maloney, N. Mendelsohn:
XML Schema Part 1: Structures;
http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/;
10.08.2004
Die aktuelle W3C-Empfehlung für XML Schema: Structures. Dieser Teil beschreibt, wie Strukturen und Beschränkungen des Inhalts definiert werden.

[05] P.V. Biron, A. Malhotra:
XML Schema Part 2: Datatypes;
http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/;
10.08.2004
Die aktuelle W3C-Empfehlung für XML Schema: Datatypes.

Dieser Teil beschreibt, wie Datentypen für die Verwendung mit XML-Schema definiert werden.

[06] J. Clark, S. DeRose:
XML Path Language (XPath), Version 1.0;
http://www.w3.org/TR/1999/REC-xpath-19991116/; 10.08.2004
Die aktuelle W3C-Empfehlung für XPath 1.0.

[07] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernández, M. Kay, J. Robie, J. Siméon:
XML Path Language (XPath) 2.0;
http://www.w3.org/TR/2004/WD-xpath20-20040723/; 10.08.2004
Der aktuelle Arbeitsentwurf von XPath 2.0.

[08] S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, J. Siméon:
XQuery 1.0: An XML Query Language;
http://www.w3.org/TR/2004/WD-xquery-20040723/; 10.08.2004
Der aktuelle Arbeitsentwurf von XQuery 1.0.

[09] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, N. Walsh:
XQuery 1.0 and XPath 2.0 Data Model;
http://www.w3.org/TR/2004/WD-xquery-20040723/; 10.08.2004
Der aktuelle Arbeitsentwurf von XQuery 1.0.

[10] A. Malhotra, J. Melton, N. Walsh:
XQuery 1.0 and XPath 2.0 Functions and Operators;
http://www.w3.org/TR/2004/WD-xquery-20040723/; 10.08.2004
Der aktuelle Arbeitsentwurf von XQuery 1.0.

[11] Marchiori, Massimo:
XML Query (XQuery);
http://www.w3.org/XML/Query; 10.08.2004
Übersicht über XQuery beim W3C.

W3C-Recommendations in deutscher Sprache

[21] S. Mintert (Hrsg.):
Viele W3C-Spezifikationen in deutscher Übersetzung und Kommentierung;
http://www.edition-w3c.de/; 10.08.2004
Vom W3C legitimierte Übersetzungen der W3C-Empfehlungen in die deutsche Sprache.

[22] S. Mintert (Hrsg.):
Extensible Markup Language (XML) 1.1, Deutsche Übersetzung;
http://edition-w3c.de/TR/2004/REC-xml11-20040204/; 10.08.2004
Übersetzung zu [01].

[23] S. Mintert (Hrsg.):
XML Schema Teil 0: Einführung, Deutsche Übersetzung;
http://www.edition-w3c.de/TR/2001/REC-xmlschema-0-20010502/; 10.08.2004
Übersetzung zu [03].

[24] S. Mintert (Hrsg.):
XML Schema Teil 1: Strukturen, Deutsche Übersetzung;
http://www.edition-w3c.de/TR/2001/REC-xmlschema-1-20010502/; 10.08.2004
Übersetzung zu [04].

[25] S. Mintert (Hrsg.):
XML Schema Teil 2: Datentypen, Deutsche Übersetzung;
http://www.edition-w3c.de/TR/2001/REC-xmlschema-2-20010502/; 10.08.2004
Übersetzung zu [05].

[26] S. Mintert (Hrsg.):
XML Path Language (XPath) Version 1.0, Deutsche, kommentierte Übersetzung;

http://www.obqo.de/w3c-trans/xpath-de-20020226/; 10.08.2004
Übersetzung zu [06].

Literatur

[31] ZVON.org - The Guide to the XML Galaxy
http://www.zvon.org/; 10.08.2004
Verschiedene Tutorien zu fast allen XML-Technologien und mehr.

[32] M. Jeckle:
Skriptum zur Vorlesung XML
http://www.jeckle.de/vorlesung/xml/script.html; 10.08.2004
Überblick über XML inkl. vieler XML-Sprachen

[33] E. Lenz:
What's New in XPath 2.0
http://www.xml.com/lpt/a/2002/03/20/xpath2.html; 10.08.2004
Eine kurze Übersicht über Veränderungen durch XPath 2.0.

[34] R. Bourret:
XML and Databases
http://www.rpbourret.com/xml/XMLAndDatabases.htm;
10.08.2004
Übersicht über XML Datenbanken und Datenbanktechnologien

[35] R. Bourret:
XML Database Products
http://www.rpbourret.com/xml/XMLDatabaseProds.htm;
10.08.2004
Ubersicht über verschiedene XML Datenbanksysteme

[36] Th. Fiebig, C.-Ch. Kanne, G. Moerkotte:
Natix - ein natives XML-DBMS
in: Datenbankspektrum 1 (2001), 5-13, 2001
http://mordor.prakinf.tu-ilmenau.de/papers/dbspektrum/dbs-01-
05.pdf Design und Implementierungsbeschreibung zu Natix

[37] M. Brundage:
XQuery – The XML Query Language, Addison Wesley, 2004, Boston...

[38] T. Böhme, E. Rahm:
XMach-1: A Benchmark for XML Data Management
Proceedings of BTW2001, Oldenburg, 7.-9. März, Springer, 2001, Berlin

[39] W. Meier:
Open Source Native XML Database
http://www.exist-db.org; 16.09.2004 Projektseite der eXist XML Database.

[40] W. Meier:
Open Source Native XML Database – XQuery
http://www.exist-db.org/xquery.html; 16.09.2004
Beschreibung von XQueryfeatures in eXist.

[41] W. Meier:
Open Source Native XML Database – Builtin Functions
http://demo.exist-db.org/exist/xquery/functions.xq; 16.09.2004
Übersicht über alle von eXist unterstützten Funktionen.

[42] X-Hive Corp.
X-Hive/DB 6.0 - Manual
Release 6 vom 19.04.2004
Handbuch im HTML-Format, wird mit X-Hive/DB ausgeliefert.