Einführung in XQuery

Udo Kelter

11.06.2007

Zusammenfassung dieses Lehrmoduls

XPath ist eine grundlegende Abfragesprache für XML-Datenbestände, deren Funktionsumfang beschränkt ist und die es nicht erlaubt, beliebige Verbunde über Attribute oder Elementinhalte zu bilden, Gruppen zu bilden, Aggregationswerte zu berechnen und danach zu selektieren und Ausgaben zu sortieren. XQuery ist eine Erweiterung von XPath, die die vorgenannten Funktionsmerkmale bietet. In diesem Lehrmodule stellen wir die diesbezüglichen Teile von XQuery einführend vor.

Vorausgesetzte Lehrmodule:

obligatorisch:
- XSLT, Teil 1 (Stichworte)
- XPath
- Transportdateien und die XML

Stoffumfang in Vorlesungsdoppelstunden: 1.0

Inhaltsverzeichnis

1 Einordnung von XQuery

Historische Entwicklung von Abfragesprachen für XML-Datenbestände

- div. frühere Abfragesprachen (-prototypen): Quilt, XPath, XQL, XML-QL u.a.,
- stammen aus verschiedenen Denkwelten / Anwendungskontexten und verschiedenen wissenschaftlich/technischen "Communities":
  - Generierung von Links bei dynamischen WWW-Inhalten
  - Suche in schwach strukturierten Datenbeständen
  - Suche in streng strukturierten Datenbeständen (XQuery ist am ehesten hier zuhause)
  - ad-hoc Suche, Information Retrival
  - komplexe integrierte Informationssysteme
- iterativer Prozeß des gegenseitigen Beobachtens und Lernens voneinander

Ziele und Gestaltungskriterien von XQuery

- anwendbar sein auf XML-Daten, die aus unterschiedlichen Quellen stammen:
  1. semi-strukturierte Dokumente
  2. relationale / tabellarische Daten, die z.B. über eine Middleware nach XML konvertiert wurden (ggf. transiente Daten)
  3. Daten aus objektorientierten DB
  4. Mischungen aus allen vorstehenden
- 2 syntaktische Schreibvarianten von Abfragen / Transformationen:
  1. als XML-Element
  2. vernünftig lesbare Form, angelehnt an SQL (XSLT-Lektion gelernt)

- integriert sein mit anderen XML-Technologien, insb. XML-Dateien als Ausgabe erzeugen können
- Details s.: XML Query Requirements. W3C Working Draft, Jan. 31, 2000; `http://www.w3.org/TR/xmlquery-req`

Versionen und Entwicklungsstand:

- Mischung aus eher relationalen bzw. dokumentorientierten Konzepten → komplex (insb. Semantik)
- XPath 1.0 war mit leichten Modifikationen Teilmenge von XQuery
  diverse Erweiterungen in XPath 2.0, die von XQuery inspiriert waren
  inzwischen: XPath 2.0 komplett (mit minimalen Abweichungen..) Teilmenge von XQuery, d.h. jeder XPath-2.0-Ausdruck ist eine korrekte XQuery-Abfrage
- XPath 2.0 und XQuery 1.0 haben einheitliches Datenbankmodell ("der Syntaxbaum"), s.:
  XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Recommendation, 23 Jan. 2007. See
  `http://www.w3.org/TR/xpath-datamodel/`
- XQuery umfaßt halbe Programmiersprache: benutzerdefinierte Datentypen (XML Schema), Funktionen usw.; ufert ziemlich aus
- Entwicklung der Spezifikationen vor kurzem abgeschlossen (u.a. Semantikspezifikation durch XML Query Algebra)
  XQuery 1.0: An XML Query Language, W3C Recommendation 23 January 2007
- Implementierungen: s. WWW-Leitseite zum Lehrmodul

- Aussichten für die XQuery als (XML-) **DBMS-Technologie**:
  die "OODBMS-Lektion" ist bekannt ....
  XML-DBMS als primäres DVS im Vergleich zu RDBMS:
  - bieten in bestimmten Anwendungsfällen große Vorteile, weniger bei klassischen betrieblichen Anwendungen
  - komplexeres Datenbankmodell → komplexere + teurere Implementierungen
    unklar, ob Implementierungen auf Großanwendungen skalieren können

  → werden RDBMS nicht verdrängen, bleiben vermutlich Nischenmarktprodukt

- XQuery als **Middleware-Technologie**:
  Einsatz bei der "losen Integration" existierender Altanwendungen, die in Schnittstellen auf Basis von XML eingekapselt worden sind
  → nur transiente Daten, kleine Datenvolumina (Rolle ähnlich wie XSLT/XPath)

- XQuery im **Vergleich zu XSLT/XPath**:
  XQuery wurde früher als Ablösung von XSLT 1.0 / XPath 1.0 angesehen.
  XSLT/XPath haben in Version 2.0 sehr stark aufgeholt!
  → beide Technologien haben inzwischen mehr Gemeinsamkeiten als früher
  ... sind nicht generell vorteilhafter, nur fallweise
  ... werden noch lange koexistieren

2 Grundlegende Merkmale von XQuery

2.1 XQuery-Ausführungsmodell

- Ein- bzw. Ausgabe: Menge von "abstrakten" Syntaxbäumen (Dokumente oder Dokumentteile)

- keine Themen:
  - Entstehung der Syntaxbäume (Parsen von Dateien, Abfragen aus anderen Datenbanken, Generierung durch Applikationen)
  - Weiterverarbeitung der Syntaxbäume (Visualisierung, serielle Darstellung als XML-Datei, ...)
- Bearbeitungsphasen eines Ausdrucks:
  1. Analyse: Bearbeitung von Namensräumen, Schemadefinitionen, Variablendefinitionen, Funktionen u.a.
  2. Auswertung des Ausdrucks

2.2 Funktionale Sprache

- funktional, analog zur relationalen Algebra, also schachtelbare Ausdrücke;

  (fast) keine Seiteneffekte

  Ein- bzw. Ausgabe: Menge von Syntaxbäumen

  keine nicht-imperativen Konstrukte wie die Transformationsregeln von XSLT
- streng typisiert
- insg. > 10 verschiedene Arten von Ausdrücken, hier nur die wichtigsten
  - Path expressions
  - Element constructors
  - FLWOR expressions
  - .....

3 XQuery-Ausdrücke

3.1 Primary Expressions

```
[84] PrimaryExpr ::= Literal | VarRef
         | ParenthesizedExpr | ContextItemExpr
         | FunctionCall | Constructor
         | OrderedExpr | UnorderedExpr
```

`Literal`	Literale (Konstanten): Zahlen, Zeichenketten usw; sehr viele Details und Varianten
`VarRef`	Variablenreferenzierungen, s.u.
`ParenthesizedExpr`	Klammerung: wie üblich
`ContextItemExpr`	`::= '.'`
`Constructor`	Generierung von XML-Elementen, s.u.

3.2 XQuery-Kommentare

```
[151] Comment ::= "(:" (CommentContent |
                      Comment)* ":)"
[159] CommentContent ::= (Char+ -
             (Char* ( '(:' | ':)' ) Char*) )
```

Beispiel:

```
(: dies ist ein XQuery-Kommentar,
   der nicht ausgegeben wird       :)
<!-- dies erzeugt einen Kommentar-
     Knoten im XML-Ausgabebaum    -->
```

3.3 Sequenzen

sind geordnete Mengen von atomaren Werten oder Knoten
Keine Schachtelung (d.h. innere Sequenz als Element einer äußeren Sequenz)
Kein Unterschied zwischen Sequenz mit 1 Element und dem enthaltenen Element

Entstehung:

- "von Hand": Aufbau mittels Literalen; Syntax: außen (...), Elemente durch Komma getrennt
- Ergebnis der Auswertung von Pfaden
- übliche Mengenoperationen

sehr komplexe Typverträglichkeitsregeln

3.4 Eingabefunktionen

schaffen Verbindung zu Eingabedaten (einzelne Dokumente, Kollektionen)
Namenraumbereichner: `fn`

`doc` Beispiele: `fn:doc('kunden.xml')`,
`fn:doc('http://xyz.org/xmldb')`
Argument: URI
Bedeutung: liefert Wurzel (genau 1 Knoten!) des Syntaxbaums des Dokuments, das unter der angegebenen Quelle zu finden ist; dieser Knoten wir Kontextknoten

`collection` Argument: URI
Bedeutung: liefert alle Knoten des Syntaxbaums des Dokuments, das unter der angegebenen Quelle zu finden ist
wird i.d.R. durch Suchpfade weiter eingeschränkt, z.B.
`fn:collection('http://xyz.org/xmldb')/kunde`

3.5 Variablen

Variablennamen beginnen mit Dollarzeichen; Syntax:

```
[87] VarRef   ::= '$' VarName
[88] VarName  ::= QName
```

Wert einer Variablen kann eine *Sequenz von Knoten sein!!*.

Wertzuweisung: in LET-Klausel;
Ausgabe der Sequenz: in der Form {`$variablenname`}
(Details zu {`...`} später)

Beispiel:

```
let $s := (<one/>, <two/>, <three/>)
return <out>{$s}</out>
```

liefert ein Element `out`, dessen textuelle Darstellung wie folgt aussieht:

```
<out><one/><two/><three/></out>
```

Vorsicht! {`$s`} gibt die *ganze* Sequenz aus, nicht nur den ersten Knoten!!
(wird oft mit `<xsl:value-of select='{$s}' />` verwechselt)

3.6 Pfadausdrücke

fast alle Features von XPath (2.0):

- relative und absolute Pfadausdrücke
- beliebig viele Schritte
- pro Schritt: Navigationsrichtung (*axis*), Knotentest und weitere Selektionen

  Ausnahmen: Richtung `namespace` nicht unterstützt;

  optional: `ancestor`, `ancestor-or-self`, `following`, `following-sibling`, `preceding`, `preceding-sibling`

  Hintergrund: bei den vorstehenden Navigationsrichtungen muß eine Implementierung den Eingabebaum komplett puffern → hoher Speicherplatzbedarf in Pipeline-Architekturen
- div. vordefinierte Funktionen (`position()`, ...)
- div. Abkürzungen

3.7 Konstruktoren

mehrere Formen, können letztlich alle Arten von Knoten des XML-Ausgabebaums erzeugen

3.7.1 Direkte Konstruktoren

- sehen aus wie ein XML-Element
- sind eine Anweisung, diesen "Text" zu erzeugen und die inneren Ausdrücke auszuwerten (wie bei XSLT)

  → "direkte" Angabe der auszugebenden Elemente, Textknoten usw.

einfaches Beispiel (ohne innere Ausdrücke):

```
<Adresse>
   <Name>Meier</Name>
   <Vorname>Hans</Vorname>
   <Strasse>Hauptstr.</Strasse>
   <Hausnummer>5</Hausnummer>
   <PLZ>57076</PLZ>
   <Ort>Siegen</Ort>
</Adresse>
```

innere Ausdrücke: werden durch geschweifte Klammern markiert:

```
let $n = 'Meier'
let $v = 'Hans' ...
return  ...
   <Adresse>
      <Name>{$n}</Name> <Vorname>{$v}</Vorname>
      <Strasse>Hauptstr.</Strasse>
      <Hausnummer>5</Hausnummer>
      <PLZ>57076</PLZ> <Ort>Siegen</Ort>
   </Adresse>
```

weitere direkte Konstruktoren: Textknoten, CDATA-Sektionen, XML-Processing Instructions, XML-Kommentare

3.7.2 Berechnete Konstruktoren

Aufbau:

1. Schlüsselwort, das den Typ des zu erzeugenden Knotens angibt (alle 7 XML-Knotentypen):
   `element`,
   `attribute`,
   `document`,
   `text`,
   `processing-instruction`,
   `comment`,
   `namespace`.
2. danach ggf. Element- / Attributname
3. Inhalt des Knotens in {..}

Beispiel:

```
element Adresse {
   attribute ADrNr {'134'}
   element Name {'Meier'}
   element Vorname {'Hans'}
   element Strasse {'Hauptstr.'}
   element Hausnummer {'5'}
   element PLZ {'57076'}
   element Ort {'Siegen'}
}
```

`namespace`-Knoten müssen zuerst kommen, danach die `attribute`-Knoten.

4 FLWOR-Ausdrücke

4.1 Definition

benannt nach den darin auftretenden Schlüsselworten `for`, `let`, `where`, `order by`, `return` bzw. Klauseln
Syntax:

```
[33] FLWORExpr ::= (ForClause | LetClause)+
                   WhereClause?
```

```
                    OrderByClause?
                    'return' ExprSingle
```

Beispiel:

```
for $adr in fn:doc("adressen.xml")//Adresse
let $n = $adr/Name
where $adr/Ort = 'Siegen'
return
   <Adresse>
      <Name>{$n}</Name>
      <Vorname>{$adr/Vorname}</Vorname>
   </Adresse>
```

Bedeutung:

- eine `for`-Klausel = Schleife;
  iteriert durch die Sequenz, die sich aus dem Ausdruck ergibt, der hinter dem `in` angegeben ist

  "Rumpf" der `for`-Schleife: Rest der Abfrage

  mehrere `for`-Schleifen: sind geschachtelt, entsprechen Kreuzprodukt; produzieren "Tupel"-Strom
  Tupel: je ein Wert von jeder Sequenz, über die iteriert wird
- `let`-Klausel: führt genau 1 Wertzuweisung (i.a. mengenwertig) pro Durchlauf der äußeren Schleife(n) durch
- `where`-Klausel: enthält Booleschen Ausdruck, der i.a. Variablen enthält;

  Auswertung negativ → aktuelles Tupel wird nicht weiter bearbeitet
- `order by`-Klausel: gibt Sortierungswünsche an
- `return`-Klausel: legt die Ausgaben fest;
  beliebig komplexer Ausdruck (insb. auch innere FLWOR-Ausdrücke erlaubt), der ein Ausgabeelement generiert

Beispiel 1 (0 `for`-Klauseln, 1 `let`-Klausel):

```
let $s := (<one/>, <two/>, <three/>)
return <out>{$s}</out>
```

erzeugt:

```
<out><one/><two/><three/></out>
```

Beispiel 2 (1 `for`-Klausel, 0 `let`-Klauseln):

```
for $s in (<one/>, <two/>, <three/>)
return <out>{$s}</out>
```

erzeugt:

```
<out><one/></out>
<out><two/></out>
<out><three/></out>
```

Beispiel 3 (2 `for`-Klauseln):

```
for $i in (1, 2)
for $j in (3, 4)
```

oder

```
for $i in (1, 2), $j in (3, 4)
```

erzeugt folgenden Tupel-Strom:

```
($i = 1, $j = 3)
($i = 1, $j = 4)
($i = 2, $j = 3)
($i = 2, $j = 4)
```

4.2 Sortierung

Angabe in `order by`-Klausel; Beispiel:

```
for $adr in fn:doc("adressen.xml")//Adresse
order by  $adr/PLZ
return <Adresse>
          <Name>{$adr/Name}</Name>
          <Vorname>{$adr/Vorname}</Vorname>
       </Adresse>
```

Sortierkriterium braucht nicht ausgegeben zu werden
Optionen:

`stable`	bei gleichen Sortierwerten bleibt Eingabereihenfolge erhalten
`descending`	fallende Sortierung

4.3 Verbunde

- Verbunde werden durch geschachtelte `for`-Schleifen und entsprechende `where`-Bedingungen "von Hand" programmiert;
- keine kompakte Notation wie NATURAL JOIN (wo die Verbundattribute automatisch bestimmt werden)

Beispiel: gegeben Dateien `kunden.xml` und `lieferungen.xml` mit den üblichen Attributen;
gesucht: Name des Kunden und Datum der Lieferung für alle Lieferungen von Kunden aus Bonn

```
for $l in fn:doc('lieferungen.xml')//lieferung
for $k in fn:doc('kunden.xml')//kunde
where $l/@KdId = $k/@KdId
  and $k/Ort = 'Bonn'
return  <Lfg> attribute KdName { $k/Name }
              attribute LDatum { $l/Datum }
        </Lfg>
```

5 Gruppierung

Realisierung "von Hand", durch passende Schleifen und Schachtelung;

5.1 Grundlegendes Implementierungsmuster

Beispiel: gib die Adreßliste nach Städten gruppiert aus und pro Stadt die Namen der dort wohnenden Personen
gewünschtes Ergebnis:

```
<Staedteliste>
  <Ortsliste>
```

```
    <Ortsname>Bonn</Ortsname>
    <Adresse><Name>...</Name>...</Adresse>
    <Adresse><Name>...</Name>...</Adresse> ...
  </Ortsliste>
  <Ortsliste> ...  </Ortsliste>
</Staedteliste>
```

Lösung:

```
<Staedteliste> {
  for $o in fn:distinct-values
              ( fn:doc("adressen.xml") // Ort )
  order by $o
  return <Ortsliste>
    <Ortsname>{$o/text()}</Ortsname>
    { for $adr in fn:doc("adressen.xml")//adresse[Ort=$o]
      order by $adr/Name
      return <Adresse>
        <Name>{$adr/Name}</Name>
        <Vorname>{$adr/Vorname}</Vorname>
      </Adresse>
    }
  </Ortsliste>
} </Staedteliste>
```

Allgemeines Vorgehen:

- äußerste Funktion: ein direkter Konstruktor, erzeugt Wurzelelement der Ausgabeliste, passenden Wurzelelementtyp wählen
- Inhalt des Wurzelelements: durch äußere Schleife, diese iteriert über die Gruppen

  hierzu Bildung einer Liste **unterschiedlicher** Werte mit der Funktion `fn:distinct-values()`
- bei mehreren Gruppierungsfeldern: pro Feld eine Schleife
- im Rumpf der äußeren Schleife (also innerhalb von deren `return`-Klausel):

eingeschachtelte FLWORExpr bestimmt pro Gruppierungswert die *Mitglieder der diesser Gruppe*; Beispiel:
`fn:doc("adressen.xml")//adresse[ Ort = $o ]`
sucht nach allen Adressen, bei denen der Ort den gerade betrachteten Ortsnamen hat

- diese Sequenz kann einer Variablen zugewiesen werden oder direkt in einer inneren `for`-Schleife berechnet werden
- für jeden Gruppierungswert wird der *komplette* Datenbestand erneut nach zugehörigen Elementen durchsucht
  sehr flexibel, aber ineffizient

5.2 Aggregationen

Beispiel: Bestimme Orte und Zahl der zugörigen Adressen für Orte mit mindestens 10 zug. Adressen

```
<Staedteliste> {
  for $o in fn:distinct-values
              ( fn:doc("adressen.xml") // Ort )
  let $nl = fn:doc("adressen.xml") //
            adresse [ Ort=$o ] / Name
  where count($nl) >= 10
  order by $o
  return
  <Ortsliste>
    <Ort>{$o/text()}</Ort>
    <Adressenzahl>{count($nl)}</Adressenzahl>
  </Ortsliste>
} </Staedteliste>
```

Erläuterungen:

- Die Funktionen `count`, `avg` usw. arbeiten auf Sequenzen
- `let $nl = ...` weist `$nl` eine Liste von Namen zu
- die Bedingung `count($nl) >= 10` bezieht sich auf einen Aggregationswert!

- Vergleich mit SQL: dort stehen solche Bedingungen in der `having`-Klausel, nicht in der `where`-Klausel!

Gemischte Bedingungen (bei SQL: Bedingungen in der `having`-*und* der `where`-Klausel):
können in XQuery meist nachgebildet werden, indem die Bedingung an die Basisknoten *bei der Bestimmung der Elemente einer Gruppe* hinzugenommen wird;

Beispiel: wie oben, aber nur für Adressen, bei denen `Hausnummer = 5` ist:

```
<Staedteliste> {
  for $o in fn:distinct-values
            ( fn:doc("adressen.xml") // Ort )
  let $nl = fn:doc("adressen.xml") //
            adresse[ Ort=$o and Hausnummer=5 ] / Name
  where .....
} </Staedteliste>
```

In komplexeren Fällen muß man FLWOR-Ausdrücke schachteln

6 Rekursion

Beobachtungen zu den bisherige Ausgabemöglichkeiten und Breite / Tiefe des erzeugten Ausgabebaums:

- for-Schleifen können den Baum nur breiter machen
- Tiefe des erzeugten Ausgabebaums = Schachtelungstiefe der Abfrage! (daher statisch erkennbar ..... wie Tabellen!)

Konsequenz: mit bisherigen Mitteln kann man *keine identische Transformation schreiben*
Wie denn?

Lösung: benutzerdefinierte Funktionen, die sich rekursiv aufrufen:
– implementieren große Teile der Automatismen von XSLT
– sehr umständlich und fehlerträchtig

Daher: Erzeugung beliebig verschachtelter Ausgabebäume abhängig vom Eingabebaum (z.B. bei "Projektionen") ist eine prinzipielle Schwäche von XQuery
XSLT ist hierzu viel besser geeignet.

Literatur

[XDM] XQuery 1.0 and XPath 2.0 Data Model, W3C Candidate Recommendation 8 June 2006; World Wide Web Consortium, http://www.w3.org/TR/xpath-datamodel; 2006

[XP2] XML Path Language (XPath) 2.0, W3C Candidate Recommendation 8 June 2006; World Wide Web Consortium, http://www.w3.org/TR/xpath20; 2006

[XQ1] XQuery 1.0: An XML Query Language, W3C Candidate Recommendation 8 June 2006; World Wide Web Consortium, http://www.w3.org/TR/xquery/; 2006

[XSLT2] XSL Transformations (XSLT) Version 2.0, W3C Candidate Recommendation 8 June 2006; World Wide Web Consortium, http://www.w3.org/TR/xslt20; 2006

Index