Seminar Multimediale Informationssysteme

Thema: Suche in semi-strukturierten Dokumenten

Bearbeitet von: Christoph Thelen

Zusammenfassung

Diese Ausarbeitung befasst sich mit der Informationssuche in semi-strukturierten Dokumenten, wobei speziell XML-Dokumente und Methoden für den Zugriff auf darin gespeicherte Daten dargestellt werden.
Zunächst erfolgt eine kurze Darstellung der Informationssuche in unstrukturierten Dokumenten. Danach folgen Beispiele für Möglichkeiten der Strukturierung. Als Notation für semi-strukturierte Dokumente wird XML vorgestellt und einige Zugriffssprachen mit ihren Einsatzgebieten und Vor- und Nachteilen werden erklärt.

1. Volltextsuche – Information Retrieval

Dokumente, die dem Austausch von Daten zwischen Menschen dienen, sind meistens nicht oder nur sehr grob strukturiert. Ein Beispiel dafür ist diese Ausarbeitung: Die Information steht im Text und muss ausgewertet und interpretiert werden. Die Überschriften geben nur eine recht ungenaue Hilfestellung bei der Einordnung der Information.
Eine automatische Analyse und korrekte Beurteilung und Einordnung ist extrem schwierig, unter Anderem auch weil Zusammenhänge meist nicht genau beschrieben, sondern nur durch den Kontext definiert werden. Es fehlt ein durchgängiges Schema, das den Daten eine Bedeutung gibt und sie damit zur Information macht. Deswegen kann ein Rechner nur eine Suche über Begriffe im ganzen Text (→ Volltextsuche) ausführen und eine Rangordnung von Dokumenten nach vorgegebenen Kriterien erstellen. [IR]

Boolesches Retrieval

Die einfachste Form der Volltextsuche ist die Boolesche Anfragesprache oder Boolesches Retrieval. Dabei werden Dokumente nach Wörtern oder allgemein nach Zeichenketten durchsucht und als Ergebnis eine Menge von Dokumenten erstellt, welche die Query erfüllen.
Zu den Nachteilen dieser Methode gehört, dass sich die Größe der Ergebnismenge nur sehr schwer kontrollieren lässt und keine Rangfolge der Dokumente nach ihrer Relevanz erstellt wird.
Bei zusammengesetzten Querys mit UND-Verknüpfungen wird ein Dokument abgelehnt, wenn nur eine der Bedingungen nicht erfüllt wird. Damit ist z. B. keine Unterscheidung möglich, wie viele Bedingungen ein Dokument erfüllt oder nicht. Für Anwendungen, in denen keine exakten Query möglich sind ist diese Form der Suche zu streng, da unter Umständen relevante Dokumente nicht in der Antwortmenge enthalten sind. Außerdem ist die unnatürliche Art der Frageformulierung oft ein Problem für unerfahrene Benutzer.

Unscharfe Anfrage

Eine Verbesserung stellt das Fuzzy Retrieval oder auch Ranked Retrieval dar. Dabei wird nicht nur zurückgegeben, welche Dokumente sich für die Antwortmenge qualifiziert haben, sondern auch eine Bewertung ihrer Relevanz nach den angegebenen Bedingungen.
Damit ist zum einen eine Rangordnung der Dokumente gegeben, die dem Benutzer eine gezielte Auswahl ermöglicht. Zum anderen lässt sich damit die Größe der Antwortmenge kontrollieren („zeige alle Dokumente mit Relevanz > 0,5“, „zeige die 10 besten Dokumente“).

Wie zu sehen war, ist die Volltextsuche kaum für die automatische Verarbeitung von Informationen in Dokumenten zu gebrauchen. Sie stellt lediglich eine Hilfestellung für den menschlichen Benutzer dar, die ihm zeigt, wo die Informationen zu finden sind. Vor allem die Erstellung und Verwaltung von Abhängigkeiten und Referenzen ist mit reiner Volltextsuche in unstrukturierten Dokumenten extrem schwierig bis unmöglich.

2. Strukturieren bei Dokumenten

Unter einem Dokument versteht man zusammengehörige Daten und Informationen, die in einer Einheit zusammengefasst sind. Diese Einheit folgt generell keiner festgelegten Form. Die Reihenfolge der Daten ist allerdings wichtig und durch die Reihenfolge innerhalb des Dokuments festgelegt, wie z. B. in einem Buch. Verknüpfungen von Daten erfolgen implizit z. B. durch Einteilung in Abschnitte mit Überschrift oder durch die Daten selber.
Ein Dokument zu strukturieren heißt, ihm einen systematischen Aufbau zu geben. Dabei muss zwischen den Empfängern der Information unterschieden werden, für einen Menschen eignet sich eine andere Struktur als für einen Rechner. Die Ziele eines entsprechenden Dokumentes unterscheiden sich dabei ebenfalls:

Menschliche Sicht

Ein Mensch will den Zweck des Dokumentes schnell erkennen und die relevanten Informationen schnell finden. Außerdem muss das Dokument in sich selbst verständlich sein. Weitergehende Anforderungen an ein Dokument sind Verwaltbarkeit und Wiederverwendbarkeit, d. h. es soll nicht an ein bestimmtes Medium gebunden sein.
Ein strukturiertes Dokument erfüllt diese Anforderungen und erleichtert so die Aufnahme und Verarbeitung der in ihm enthaltenen Informationen durch den menschlichen Benutzer sowie seine Verwaltung und automatische Aufbereitung für jedes Zielmedium (Papier, PDF, HTML...).
Zusätzlichen können Metainformationen zum automatischen Aufbau eines Inhaltsverzeichnisses und der gezielten selektiven Suche nach Dokumenten enthalten sein.
(Beispiel: HTML <meta> Tag)
Die Plattform-Unabhängigkeit wird von dem verwendeten Medien-/Dateiformat bestimmt – ein Dokument aus Papier ist für einen Rechner ohne Wert (Texterkennung via Scanner ausgenommen) während eine HTML-Datei mittlerweile auf fast allen Betriebssystemen von einer Vielzahl von Programmen unterstützt und so dargestellt wird, wie der Ersteller es vorgesehen hat (im Fall von HTML zumindest sehr ähnlich).
Als Beispiel für eine Methode zur Mensch-orientierten Strukturierung von Dokumenten wird hier „Information Mapping“ vorgestellt.

Information Mapping (s. Abbildung1) [IM] ist eine Sammlung von Regeln und Vorgehensweisen, die es dem Autor erleichtern „strukturierte“, d. h. für den Menschen leicht lesbare Dokumente zu erstellen. Grundbegriffe sind dabei zwei neu definierte Informationseinheiten – der Begriff des „Blocks“ und der „Map“.

Ein **Block** setzt sich aus einem oder mehreren Absätzen, Sätzen, Tabellen, Diagrammen oder sonstigen Informationsübermittlern zusammen. Er befasst sich mit einem abgegrenzten Thema und wird entsprechend betitelt. Zu diesem Thema stellt der Block Informationen auf für den Anwender verständliche Art und Weise dar.

Eine **Map** umfasst mehrere zusammenhängende Blöcke, die ein gemeinsames Oberthema haben. Eine Map hat ebenfalls einen Titel, der den gemeinsamen Inhalt und den Zweck der gesamten Map beschreibt.

Weiterhin werden folgende Informationstypen festgelegt: Prozeduren, Prozesse, Strukturen, Begriffe, Prinzipien, Fakten und Klassifikationen. Ein Block behandelt immer genau einen Informationstyp.
Zuletzt werden sieben Prinzipien angegeben, nach denen das Dokument erstellt werden soll:

- Gliederung: Die Information muss in übersichtliche und leicht zu verarbeitende Einheiten gegliedert werden.
- Relevanz: Informationseinheiten enthalten immer nur thematisch zusammengehörige und in Bezug auf ihren Zweck für den Leser wesentliche Informationen.
- Betitelung: Jede Einheit wird mit einem Titel versehen, der klar den Zweck, die Funktion oder den Inhalt der Informationseinheit bezeichnet.
- Einheitlichkeit: Ähnliche Informationen, Gliederungen oder Formate werden ähnlich behandelt.
- Gleichwertigkeit der Informationsträger: Diagramme, Bilder, Tabellen und Text können autonom als Informationsträger dienen oder sich gegenseitig ergänzen.
- Verfügbarkeit von Einzelheiten: Alle notwendigen Einzelheiten werden dem Leser an der Stelle zur Verfügung gestellt, an der er sie braucht.
- Systematische Gliederung und Betitelung: Die Prinzipien der Gliederung und Betitelung werden systematisch auf allen Ebenen des Dokuments angewendet.

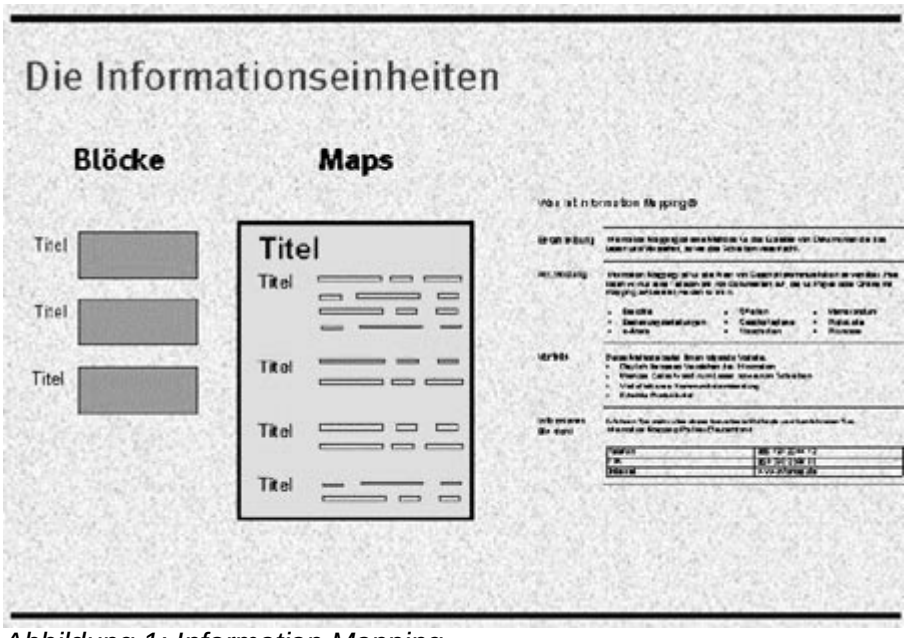

Abbildung 1: Information Mapping

Rechnersicht

Für den Rechner sind strukturierte Dokumente Datensammlungen, die genau zu einem entsprechenden Schema passen. Das „natürliche“ Beispiel ist hier die Datenbank mit verschiedenen Tabellen, deren Aufbau und Art der gespeicherten Daten teilweise zusammen mit ihrer Semantik genau definiert ist. Ein einfaches Beispiel zeigt Tabelle 1.

PNR	Name	Vorname	Geburtsdatum	Beruf	Vorgesetzter
123	Meier	Martin	04.05.1963	Zeichner	235
573	Maier	Michael	11.12.1971	Programmierer	235
235	Mayr	Markus	22.02.1955	Manager	-
...	...	...	...	...	...

Tabelle 1: Daten mit definiertem Schema und Semantik (Fremdschlüssel „Vorgesetzter“)

Diese genaue Schemadefinition führt allerdings zu verminderter Plattform-Unabhängigkeit – jedes Programm bzw. DBMS verwendet ein eigenes Dateiformat sowohl für die Schemainformation als auch für die eigentlichen Daten. Dadurch treten nicht nur bei der Datenintegration zwischen verschiedenen DBMS Probleme auf, sondern auch beim „einfachen“ Austausch von Informationen zwischen Programmen, die unterschiedliche Dateiformate oder Schemadefinitionen benutzen.

3. Semi-strukturierte Dokumente

Aus der ständig zunehmenden Vernetzung und Verknüpfung von Datenquellen entsteht ein Bedarf an einem besseren Datenaustauschformat, in dem sowohl die Daten als auch das zugrunde liegende Schema erfasst werden.

In semi-strukturierten Dokumente sind die eigentlichen Daten in eine Struktur eingebettet, aus der sich nicht nur Schemainformationen, sondern auch semantische Informationen wie z. B. Verknüpfungen und Abhängigkeiten ableiten lassen. Diese Art der Vermischung von Daten und zugehörigem Schema nennt man auch Strukturtragende Daten. [BKOS]
Dabei ist deutlich mehr Flexibilität gegeben, als in den starren Schemata wie man sie von Datenbanken her kennt. Z.B können sehr komplexe Strukturen definiert werden, die im Relationenmodell nur über große Umwege und weniger natürlich nachgebildet werden können. Hinzu kommen erlaubte Irregularitäten wie redundante oder fehlende Elemente.

Verknüpfungen werden in der Struktur selber dargestellt, indem Elemente entsprechend geschachtelt werden. Wenn also die Strukturierung in einer Art von Hierarchiebildung besteht, dann würde ein Datum *a*, das in dieser Hierarchie unterhalb eines Datums *b* steht, mit diesem *b* verknüpft.

Das Schema kann aber auch in einer separaten Datei abgelegt werden, was die syntaktische und grammatikalische Überprüfung eines Dokumentes und den einfachen Austausch des Schemas ermöglicht.
Wichtig ist auch, dass ein Dokument immer als Ganzes gesehen wird – nicht nur die reinen Daten sind von Interesse, sondern z. B. auch die Reihenfolge, in der sie gespeichert sind. Eine Anfrage auf ein ganzes Dokument sollte dieses möglichst unverändert zurückgeben.

Diese Art der Sicht auf Informationen nennt man auch **Dokumenten-Perspektive**. Im Gegensatz dazu steht die **DB-Perspektive**, die von einem festen Schema ausgeht und die Daten mit Hilfe dieses Schemas auswertet und zu Informationen aufbereitet.
In der DB-Perspektive stehen die Daten klar im Vordergrund, die Reihenfolge ist unwichtig. Verknüpfungen müssen explizit im festen Schema modelliert werden. Ein Beispiel für die DB-Perspektive ist das Relationenmodell.

Hauptmerkmal des Relationenmodells sind die sogenannten „flachen“ Tupel von Daten. Damit sind Datentupel gemeint, die ein Objekt von einem bestimmten Typ beschreiben. Assoziationen und Verschachtelungen müssen über Fremdschlüssel realisiert werden, n:m-Verknüpfungen erfordern zusätzliche Datentypen ($\rightarrow$ Tabellen), da keine mehrwertigen Attribute möglich oder zumindest nicht erwünscht sind ($\rightarrow$ Normalformen).

Ein Datenbank-Schema (→ Relationenschema) existiert getrennt von den Daten und ist unabhängig von ihnen. Eine Selektion von Daten wird von einer sehr mächtigen Anfragesprache durchgeführt (z. B. SQL). Die Anfragen werden dabei auf Konsistenz bezüglich des Schemas überprüft, optimiert und dann ausgewertet. Dabei wird allerdings reines Boolesches Retrieval durchgeführt, eine unscharfe Suche ist nur in Ansätzen möglich und nur für Textdaten. Eine Unschärfe bezüglich der Struktur kann nicht erzeugt werden, wegen der genannten Überprüfung auf Konsistenz mit dem Schema.

In der DB-Perspektive stehen die reinen Daten im Vordergrund. Ein Anfrageergebnis ist nicht von vornherein auf eine bestimmte Reihenfolge festgelegt. Die Sortierung der Tupel spielt keine Rolle, sofern dies bei der Anfrage nicht vorgegeben wird. Der Kontext eines Tupels muss aus den Verknüpfungen und der im Schema festgelegten Semantik ausgelesen und abgeleitet werden.

Beispiel Dokumenten-/DB-Perspetive:

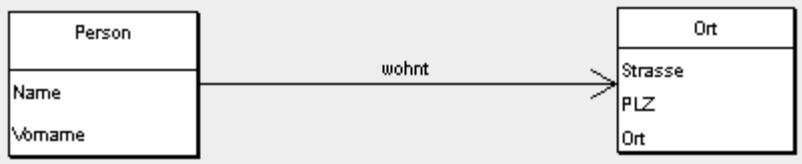

Abbildung 2

Die Assoziation „wohnt“ in dem Beispiel, das im UML-Diagramm in Abb. 2 bechrieben ist, wird im Relationenschema als Fremdschlüssel auf die Relation Ort umgesetzt. Als Folge davon muss ein geeigneter Primärschlüssel für Ort gefunden werden, was sehr oft als ID-Feld realisiert wird. Dadurch wird das Tupel zwar absolut eindeutig, aber um „künstliche“ Informationen erweitert – die gespeicherten Daten haben nicht alle direkte Aussagekraft über das beschriebene Objekt.

In semi-strukturierten Dokumenten (hier XML als Notation) wird diese Verknüpfung viel natürlicher in der Struktur selber dargestellt (Codebeispiel 1)

```
<person>
<name>Müller</name>
<vorname>Klaus</vorname>
<wohnort>
   <ort>
      <strasse>Gerhard-Hauptmann Str. 123</strasse>
      ...
   </ort>
</wohnort>
...
</person>
```

Codebeispiel 1: Verknüpfung durch Verschachtelung

Klar ist aber auch, dass durch diese Art der Verknüpfung und Kontexterzeugung Redundanzen auftreten, wenn bestimmte Objekte mehrfach vorkommen.

XML

Als spezielles Format semi-strukturierter Dokumente hat sich in den letzten Jahren XML durchgesetzt. XML ist eine Textbasierte Sprache für den Austausch und die Speicherung von Daten.
XML ([ML]) ist eine Untermenge des SGML (Standard Generalized Markup Language) Standards des World Wide Web Consortiums (W3C) mit besonderer Hervorhebung der

Erweiterbarkeit. Die Struktur wird durch Tags (z. B. `<name>`) erzeugt, mit denen Daten in das Schema eingeordnet werden und ihnen so Bedeutung verliehen wird. Die Tags treten immer paarweise und korrekt verschachtelt auf, d. h. auf ein Start-Tag folgt immer das zugehörige End-Tag und es treten keine Überlappungen von Bereichen auf, die mit einem Start-Tag eingeleitet und von einem End-Tag abgeschlossen werden.
Die einzige Ausnahme stellen leere Tags dar, wobei `<verheiratet/>` die Abkürzung für `<verheiratet></verheiratet>` ist.
Tags können zusätzliche Informationen in Form von Attribute enthalten: `<link protokoll="ftp"> ... </link>`

Um die Informationen in einem XML-Dokument auszulesen benutzt man sog. „Parser“, die durch Auswerten der Tags die Struktur des Dokuments in einem Baum (dem Ableitungsbaum) repräsentieren. In jedem XML-Dokument gibt es genau ein äußeres Tag-Paar, das Wurzelelement im Ableitungsbaum ist. Dieser Baum kann allerdings auch Querverweise enthalten, entspricht also eher einem gerichteten Graphen.

Zu einem XML-Dokument gibt es zwei wichtige Eigenschaften:

1. Wohlgeformtheit (wellformedness): Ein wohlgeformtes XML-Dokument befolgt die XML-Syntax, benutzt also beispielsweise < und > richtig und schachtelt die Elemente korrekt.
2. Gültigkeit (validity): Ein wohlgeformtes XML-Dokument wird als gültig oder valide bezeichnet, wenn es den Regeln einer DTD folgt.

Die DTD (engl. DokumentType Definition) ist eine externe Datei, in welcher die Grammatik, oder auch das Schema eines XML-Dokuments abgelegt ist. Die Grammatik legt fest, welche Strukturen (Art der Verschachtelungen, optionale Elemente, etc.) erlaubt sind. Wenn mehrere Dokumente dieselbe DTD benutzen, so sagt man sie besitzen den gleichen Dokumententyp.
Für das obige Beispiel könnte die DTD wie in Codebeispiel 2 aussehen.

```
<!ELEMENT personen         (person*)>

<!ELEMENT person  (name, vorname?,  wohnort?)>

 <!ELEMENT name             (#PCDATA)>
 <!ELEMENT vorname          (#PCDATA)>
 <!ELEMENT wohnort          (ort)>

<!ELEMENT ort              (strasse, plz, ortsname)>
 <!ELEMENT strasse          (#PCDATA)>
 <!ELEMENT plz              (#PCDATA)>
 <!ELEMENT ortsname         (#PCDATA)>
```

Codebeispiel 2: Beispiel DTD

Die Elemente `personen`, `person` und `ort` sind aus anderen Elementen zusammengesetzt, die in der jeweiligen Klammer angegeben sind, wobei `personen` das Wurzelelement ist. Die Kommaliste innerhalb dieser Klammern besteht aus regulären Ausdrücken, d. h. es können Multiplizität, Optionalität, usw. modelliert werden.
Elemente wie z. B. `name`, `vorname` oder `plz` sind atomar und enthalten Daten vom Typ „Parsed CDATA“. CDATA steht dabei für „Character Data“, d. h. eine Folge von beliebigen Zeichen. Eventuell darin auftretende Steuerzeichen werden ignoriert.
Codebeispiel 3 zeigt einen Teil einer DTD, in der ein Element mit Attribut beschrieben wird.

```
<!ELEMENT preis             (#PCDATA)>
<!ATTLIST preis waehrung CDATA #REQUIRED>
```
Codebeispiel 3 Element mit Attribut

Mit dem Zusatz `#REQUIRED` wird festgelegt, dass dieses Attribut immer belegt werden muss. Ansonsten ist die Benutzung des Attributs optional. Außerdem kann festgelegt werden welche Werte angegeben werden dürfen (z. B. `(EUR|USD|SFR)`) , welcher Wert standardmäßig ausgewählt wird wenn das Attribut nicht belegt wird oder ob das Attribut einen konstanten Wert erhält.
Darüber hinaus gibt es auch die Möglichkeit, Referenzen auf Elemente zu setzen, auf die dann von anderen Elementen aus zugegriffen werden kann.
Für den Fall, dass XML-Steuerzeichen (wie „<“ oder „>“) innerhalb eines XML-Dokuments auftreten werden diese gegen spezielle, an die HTML-Notation angelehnte Zeichenketten („<“, „>“) ausgetauscht

4. Anfragesprachen

XML bietet eine mächtige Form der Datenspeicherung und –Übertragung. Um aber mit diesen Daten arbeiten zu können, müssen Möglichkeiten geschaffen werden diese Daten je nach Anwendung nach verschiedenen Kriterien geordnet auszulesen. Dazu müssen die Daten selber, aber auch die Struktur gelesen und verarbeitet werden. Für diese Aufgabe sind die Anfragesprachen zuständig.
Es gibt noch andere Ansätze, auf Daten in XML-Dokumenten zuzugreifen. Eine einfache Implementierung einer Suche ignoriert die Struktur, sowie die Tag-Namen und führt im Prinzip eine normale Volltextsuche auf den Daten des Dokuments durch, mit allen beschriebenen Vor- und Nachteilen.
Eine weitere Möglichkeit ist die manuelle oder halbautomatische Integration in eine Datenbank, um dann mit den vorhandenen Anfragesprachen zu arbeiten. Dabei muss die Struktur des XML-Dokuments auf ein globales Schema abgebildet und die Daten dann entsprechend in Tupel eingeordnet werden.
Die flexibelste Variante ist die der speziellen Anfragesprachen, auf die im Folgenden näher eingegangen wird. Nur durch die Anpassung an die Eigenschaften von XML können Daten zusammen mit ihrem Kontext aus einem Dokument ausgelesen und sinnvoll in anderen Anwendungen weiterverwendet werden.
Als Erweiterung zu den Standard-Anfragesprachen wird zum Schluß die textuelle und strukturelle Ähnlichkeitssuche an Beispielen vorgestellt.

Zunächst müssen die Anforderungen an eine solche Anfragesprache festgelegt werden:
Mit ihrer Hilfe müssen Daten aus großen XML-Dokumenten extrahiert werden können, sie soll möglichst flexible Anfragen zulassen und bearbeiten, es sollen verschiedene XML-Datenquellen integriert werden können und nach Möglichkeit sollten Daten zwischen Relationalen/OO-Datenbanken und XML-Dokumenten konvertiert werden können. Außerdem soll man nicht das genaue Schema kennen müssen, um eine Anfrage zu stellen, ein vorhandenes Schema soll aber zur Überprüfung der Anfrage (sind die gefragten Attribute/Strukturen vorhanden/zulässig, usw.) ausgenutzt werden. Reihenfolge und Verknüpfungen der Daten sollen bei der Ausgabe erhalten bleiben.

Anders als bei SQL ist eine genaue Definition einer Anfrage wegen der beschriebenen Flexibilität von XML oft nicht möglich. Daher muss eine Anfragesprache über Sprachkonstrukte verfügen, die es ermöglichen, eine unbekannte Struktur oder eine unbestimmte Tiefe auszudrücken.

Da XML besonders im Internet zu finden ist, wo eine große Menge zusammenhängender Quellen mit großer Streuung und entsprechend unterschiedlichen, wenn auch teilweise

ähnlichen Strukturen zu finden ist, sollte eine Anfragesprache idealerweise nicht nur eine Auswahl von sich qualifizierenden Dokumenten treffen, sondern auch eine Rangliste von Dokumenten entsprechend ihrer Wichtigkeit festlegen (→ Ranked Retrieval).

XPath

Die Grundlage der "höheren" Anfragesprachen für XML wird von XPath gebildet. [XPA] Die Hauptaufgabe von XPath ist, Teile eines XML-Dokuments anzusprechen. Dazu wird jedes XML-Dokument als Baum entsprechend seiner logischen Struktur angelegt. Die Knoten dieses Baumes werden von Elementen, Attributen und Text gebildet. Einige Knoten besitzen einen lokalen Namen, entsprechend ihrer Benennung im Dokument. Zusätzlich kann ein Knoten einen „Expanded Name" haben, der aus dem lokalen Teil erweitert um eine Namespace URI besteht.
Es gibt sieben verschiedene Typen von Knoten:

- Wurzelknoten
- Elementknoten
- Textknoten
- Attributknoten
- Namespace Knoten
- Processing instruction Knoten (Informationen für den Parser bzw. die Anwendung)
- Kommentarknoten

Mit Hilfe eines nach diesen Vorgaben aufgebauten Baumes ermöglicht XPath den Zugriff auf alle Elemente im entsprechenden Dokument. Auf die zugehörige Syntax wird im Folgenden kurz eingegangen.

Zentrales Konstrukt in XPath ist der „**Ausdruck"**. Die Auswertung eines Ausdrucks gibt ein Objekt zurück, welches eine Knotenmenge, ein boolescher Wert, eine Gleitkommazahl oder ein String sein kann, wobei Knotenmenge eine ungeordnete Sammlung von Knoten ohne doppelte Knoten bezeichnet.
Die Auswertung eines Ausdrucks erfolgt jeweils innerhalb eines bestimmten Kontextes. Der **Kontext** besteht aus

- einem Knoten (Kontextknoten)
- zwei positiven Integer-Zahlen (die Position des Kontexts von Kontextknoten aus und die Kontextgröße)
- einer Menge von Variable Bindings (also Zuordnungen von Variablennamen zu Werten)
- einer Funktionsbibliothek (Funktionen, auf die innerhalb des Kontextes zugegriffen werden kann)
- der Menge von Namespace Deklarationen im Bereich des Ausdrucks

Die wichtigste, weil am häufigsten benutzte Art von Ausdrücken ist der **Location Path**. Er wählt eine Menge von Knoten relativ zum Kontextknoten aus und kann rekursiv weitere Ausdrücke zur Filterung der ausgewählten Knoten enthalten. Rückgabe ist die sich qualifizierende Menge von Knoten.
Die Syntax eines Location Path lässt sich am besten an Beispielen verdeutlichen:

- `child::para` : wählt das `para` Element, das Kind des Kontextknotens ist
- `child::*` : wählt alle Kinder des Kontextknotens
- `child::text()` : wählt alle Kinder des Kontextknotens von Typ text

- `attribute::name` : wählt das Attribut `name` des Kontextknotens
- `ancestor::div` : wählt alle Vorgänger vom Typ `div` des Kontextknotens
- `self::para` : wählt den Kontextknoten wenn er vom Typ `para` ist, sonst nichts
- `child::*/child::para` : wählt alle `para` Enkel des Kontextknotens
- `/descendant::para` : wählt alle `para` Elemente im selben Dokument wie der Kontextknoten
- `/descendant::olist/child::item` : wählt alle `item` Elemente im selben Dokument wie der Kontextknoten die einen `olist` Vater haben
- `child::para[position()=1]` : wählt das erste `para` Element, das Kind des Kontextknotens ist
- `child::para[position()=last()-1]` : wählt das vorletzte `para` Element, das Kind des Kontextknotens ist
- `following-sibling::chapter[position()=1]` : wählt den nächsten `chapter` Nachfolger des Kontextknotens
- `child::para[attribute::type="warning"]` : wählt alle `para` Elemente, die Kinder des Kontextknotens sind und deren „type“-Attribut den Wert „warning“ hat
- `child::chapter[child::title='Introduction']` : wählt das erste `chapter` Element, das Kind des Kontextknoten ist und ein oder mehr Kindelemente mit dem Stringwert „Introduction“ hat

Um oft benutzte Location Paths zu vereinfachen ist folgende, verkürzte Schreibweise möglich:

- `para` : wählt das `para` Element, das Kind des Kontextknotens ist
- `@name` : wählt das Attribut `name` des Kontextknotens
- `*/para` : wählt alle `para` Enkel des Kontextknotens
- `//olist/item` : wählt alle `item` Elemente im selben Dokument wie der Kontextknoten die einen `olist` Vater haben
- `.` : wählt den Kontextknoten
- `.//para` : wählt alle `para` Nachfolger des Kontextknotens
- `..` : wählt den Vater des Kontextknotens

Zusammenfassend gesagt kann man `child::` aus einem Zwischenschritt weglassen, Attribute mit @ ansprechen, mit // alle Nachfolger und den Kontextknoten selbst auswählen und „.“ für `self::node()` und „..“ für `parent::node()` schreiben.

Über diese „Suchpfade“ (→Location Path) werden passende Knoten ausgewählt und zurückgeliefert. In den Beispielen hat man unter Anderem auch schon Aufrufe der Kern-Funktionsbibliothek gesehen (z. B. `last()`).

XPath wird von den höheren Sprachen XSLT und XPointer verwendet, von welchen letztere jetzt näher betrachtet werden soll.

XPointer

In XPath können nur einzelne Knoten, bzw. Mengen von Knoten referenziert werden, XPointer [XPO] führt als Ergebnistyp zusätzlich das "Location-Set" ein. Ein Location-Set erweitert eine Knotenmenge dahingehend, dass benutzerdefinierte Auswahlen wie z. B. "vom dritten Wort des ersten Absatzes bis zum fünften Wort des vorletzten Absatzes" möglich sind. Ein solches Ergebnis wird auch **range** genannt. Eine range hat einen Startpunkt (**point**, z. B. wie oben „drittes Wort im ersten Absatz“) und einen Endpunkt („fünftes Wort des vorletzten Absatzes“). Alle dazwischen liegenden Elemente oder Teile(!) von Elementen werden ausgewählt.

Formell besteht ein point aus einem Knoten (dem Containerknoten) und einem nicht-negativen Index, der die Position des points nach einem Zeichen oder einem Element festlegt.

In XPointer werden dazu zwei neue Funktionen definiert:

- *location-set* `range-to (location-set)`
- *location-set* `string-range (location-set, string, number, number)`

Mit `range-to` wird der Bereich vom aktuellen Kontextknoten bis zur angegebenen Location ausgewählt. Zum Beispiel gibt

```
//para[2]/range-to(following::chapter)
```

alles zwischen dem zweiten `para` Element und dem darauffolgenden `chapter` Element zurück.

`string-range` gibt einen Location-Set zurück, in dem alle Vorkommen des übergebenen Strings innerhalb des string-values der Position `location` angegeben sind.

```
string-range(//para, "Ziel", 2, 0)
```

gibt alle Positionen des Wortes Ziel im string-value aller `para` Elemente im Dokument zurück. Genauer gesagt legt die 2 fest, dass die zurückgegebenen points vor dem zweiten Buchstaben stehen, also hier dem „i“.

Da die Funktion nur auf dem string-value des Elements arbeitet wird auch mixed Content, also mit Tags durchsetzter Text richtig erkannt. Das obige Beispiel findet also auch folgendes Vorkommen des Suchstrings:

```
<para> ...  Z<b>iel</b> ... </para>
```

Weitere Methoden für den Zugriff auf Ranges sowie die Funktionen `here()` und `origin()`, die das übergeordnete Element des aktuellen Elements, bzw. den Ursprung eines Links aus einem anderen Dokument heraus bei Zugriff auf verschiedene Dokumente zurückgeben, sind ebenfalls in XPointer definiert.

Für den gezielten Zugriff auf Elemente erlaubt XPointer die direkte Auswahl über unique ID-Attribute. Im Beispiel sieht das so aus:

```
<person id="1234">
<name>Müller</name>
...
</person>
```

Eine Ausdruck `#1234/name` würde das Name Element der Person mit der ID 1234 zurückliefern, wobei `#1234/name` die verkürzte Form von `#xpointer(//id("1234")/name`) ist.

Allgemein fällt auf, dass die Grammatik von XPointer mit Hilfe einer einfachen Extended Backus-Naur Form aufgebaut ist, wie es in der „XML-Recommendation“ beschrieben ist.

XPointer ermöglicht die Untersuchung hierarchischer Strukturen und Auswahl von Teilen dieser Strukturen über Vergleich von Attributwerten, Elementtypen, Textinhalten und relativer Position in der Hierarchie. Dadurch werden die Anfragen einfach und vor allem stabil, da sie sich nicht an IDs orientieren, sondern nur am Element-Baum. Das bedeutet, dass eine

XPointer-Location so lange erhalten bleibt, bis sich die Reihenfolge der Elemente im Dokument ändert, was recht selten passiert.
Darüber hinaus wird es durch die Unabhängigkeit von ID-Attributen einfacher, Referenzen auf Daten in fremden Dokumenten anzulegen und zu verwalten. XPointer bietet plattform-unabhängiges Boolesches Retrieval in XML-Dokumenten.

XML Fragment Interchange

Die bisher vorgestellten Sprachen sind Anfragesprachen, die Daten in einem Dokument relativ zu einer bestimmten Location (einem bestimmten Kontext) finden und auslesen können. Wenn man allerdings die Daten auch ändern, oder eine bestimmte, nicht durch ihre genaue Position bekannte Location angeben will, dann muss der aktuelle Kontext bekannt sein. Dies ist zwar durch Analyse des Dokuments leicht möglich, aber in vielen Fällen möchte man nicht das ganze Dokument übertragen müssen, um an wenige, wenn auch Kontext-bezogene Daten zu gelangen.
Das Problem ist, dass ein Parser den Kontext kennen muss, um ein Fragment, also einen (evtl. kleinen) Teil eines XML-Dokuments richtig in das zugehörige Schema einordnen und die Information entsprechend verwenden zu können. Dieser Kontext ergibt sich normalerweise aus dem Zustand des Parsers, also z. B. Verschachtelungstiefe/-Weg usw. Wenn allerdings nur ein Fragment eines Dokumentes übertragen wird, das nicht bis auf Wurzelebene zurück reicht, kann der Parser die empfangenen Informationen nicht mit den übrigen verknüpfen und im Zusammenhang nutzbar machen.
Es muss also eine Möglichkeit bereitgestellt werden, den Kontext eines XML-Fragments zu übermitteln, ohne den entsprechenden Teilbaum oder sogar noch mehr Daten an den Empfänger zu schicken und ihn die Auswertung machen zu lassen, zumal dies oft weder erwünscht noch machbar ist.
Diese Funktionalität wird von XML Fragment Interchange (FI) vorgestellt. [XF] FI definiert einen Weg, kleine Teile eines XML-Dokuments sinnvoll zu bearbeiten, ohne alle darüber liegenden Elemente durchgehen zu müssen. Dabei ist es unwichtig, ob die entsprechenden Teile Entitäten sind oder nicht und ob die Teile direkt verwendet oder für spätere Bearbeitung gesammelt werden.

FI ist keine implementierte Sprache, sondern eine Empfehlung, wie eine solche Sprache aussehen sollte. In FI werden drei Begriffe definiert: **Fragment context specification**, also eine Beschreibung des Kontextes in dem sich das Fragment befindet, **Fragment body**, der eigentliche Inhalt des Fragments, aus dem Dokument herausgelöst und **Fragment Entity**: Das Container-Object, in dem der Fragment body, evtl. zusammen mit der Fragment context specification übertragen wird.

Außerdem werden genaue Bedingungen festgelegt, die Fragmente erfüllen müssen um übertragen werden zu können. So muss ein Bereich eines XML-Dokuments (wohl-) balanciert sein, d. h. wenn der Bereich einen Teil eines Konstrukts beinhaltet, dann ist das ganze Konstrukt enthalten, um ein Fragment werden zu können. (Wenn z. B. ein Start-Tag enthalten ist muss auch das entsprechende End-Tag enthalten sein.)
Weiterhin definiert FI die Menge der Informationen über ein Fragment, die erfolgreiches, sinnvolles Parsen sowie die Betrachtung und Änderung der Daten im Fragment ermöglichen und die Notation, in der diese Information (also die Fragment context specification) beschrieben wird. Zusätzlich werden noch Mechanismen genannt, mit denen die Information mit dem Fragment body verknüpft wird.

Als Beispiel betrachten wir einen Fragment body, der aus listitem 2 und 3 einer sortierten Liste (orderedlist) innerhalb der zweiten sect1 im ersten chapter im ersten part eines Buches book besteht. Die DTD befindet sich in der Datei Docbook.dtd auf dem OASIS Open Web Server, das Dokument in der Datei mybook.xml auf dem Acme Web Server. Die Fragment body specification sieht dann z. B. wie im Codebeispiel4 aus.

```
  <f:fcs xmlns:f="http://www.w3.org/2001/02/xml-fragment"
         extref="http://www.oasis-
open.org/docbook/docbook/3.0/docbook.dtd"
         parentref="http://www.acme.com/~me/mydocs/mybook.xml"
         xmlns="http://www.oasis-open.org/docbook/DocbookSchema">
    <book>
      <part>
        <chapter>
          <sect1/>
          <sect1>
            <orderedlist numeration="arabic">
              <listitem/>
              <f:fragbody/>
            </orderedlist>
          </sect1>
        </chapter>
      </part>
    </book>
  </f:fcs>
```

Codebeispiel 4: Ein Kontext eines XML-Fragments

Der Parser erhält also alle Informationen die er benötigt. An der Stelle `<f:fragbody/>` wird dann der Fragment body eingesetzt.

XLink

Wir können jetzt Informationen aus einem XML-Dokument auslesen und sinnvoll bearbeiten. Was wir noch nicht können, ist Verknüpfungen zwischen verschiedenen Dokumenten aufzubauen. Diese Funktionalität liefert XLink. [XL]

Codebeispiel 5 zeigt, wie ein ssolcher Link aussehen könnte.

```
<my:crossReference                    -- Der Name des Links
  xmlns:my="http://example.com/"
  xmlns:xlink="http://www.w3.org/1999/xlink"      -- der Standard-
                                                     Namespace von XLink
  xlink:type="simple"
  xlink:href="students.xml"                  -- das Zieldokument
  xlink:role="http://www.example.com/linkprops/studentlist"
  xlink:title="Student List"
  xlink:show="new"
  xlink:actuate="onRequest">
Current List of Students
</my:crossReference>
```

Codebeispiel 5: XLink

Weiter soll darauf nicht eingegangen werden, wichtig ist die Fähigkeit der Verknüpfung von beliebig verteilten Dokumenten.

XML-Query Language (XQuery)

Nachdem jetzt die Grundlagen des Zugriffs auf XML-Domukente vorhanden sind, kann eine höhere Anfragesprache vorgestellt werden. XQuery [XQ] baut auf allen bisher beschriebenen Hilfsmitteln auf und stellt Funktionalität zur Verfügung, mit der Informationen aus beliebigen XML-Datenquellen, wie z. B. XML-Dokumenten oder auch XML-erzeugende Middleware, ausgelesen und ausgewertet werden können.

XQuery ist als kleine, leicht zu implementierende Sprache entworfen, die Anfragen sind prägnant und leicht zu verstehen. Die Syntax ist auf Lesbarkeit für den Menschen ausgerichtet. Ansonsten lehnt sich XQuery stark an die XML-Spezifikationen an. Das Typen-System z. B. basiert auf dem von XML und soll bald komplett angepasst werden.

Wie in XPath ist der **Ausdruck** das zentrale Objekt der Sprache. Ein Ausdruck gibt entweder eine Sequence oder einen Error-Wert zurück. Eine „**Sequence“** ist eine sortierte Menge von „**Items“** (inklusive der leeren Menge). Ein Item kann entweder ein **atomarer Wert** oder ein **Knoten** von einem der folgenden sieben **Knotentypen** sein, die schon in XPath definiert wurden:

- Document
- Element
- Attribute
- Text
- Namespace
- Processing Information
- Comment

Zu jedem Ausdruck gehört ein Kontext, der aus allen Informationen besteht, die das Ergebnis des Ausdrucks beeinflussen können. Es gibt einen statischen Kontext und einen Auswertungs-Kontext.

Der **statische Kontext** besteht aus folgenden Komponenten:

- In-scope namespaces: Eine Menge von (Präfix, URI) Paaren, die zur Auflösung der Präfixe innerhalb des Ausdrucks dienen
- Default namespace for element and type names: Eine URI mit dem standard namespace für alle Namen von Elementen und Typen die ohne Präfix angegeben wurden
- Default namespace for function names: s.o., der namspace für Funktionen
- In-scope schema definitions: Eine Menge von (QName, type definition) Paaren (QName = Namenskonvention von XQuery), die alle im Ausdruck verfügbaren Typen enthält, inklusive den eingebauten Schema-Typen und global definierten Typen von importierten Schemata.
- In-scope variables: (Qname, type) Paare mit den Variablendeklarationen des Ausdrucks
- ...

Im **Auswertungs-Kontext** finden sich alle Komponenten des statischen Kontextes sowie dem **Focus** des Ausdrucks, der dem Prozessor ermöglicht, die bearbeiteten Knoten zu verfolgen. Für den äußersten Ausdruck wird der Focus von der Umgebung geliefert, in der er ausgeführt wird. Es gibt aber Sprachkonstrukte, die einen Sub-Ausdruck erzeugen, wie z. B. der Pfadausdruck `E1/E2`. Der Ausdruck `E2` wird für jedes von `E1` gelieferte Item neu ausgewertet und erhält bei jeder Auswertung einen neuen Focus. Der Focus von `E2` wird dabei „innerer Focus“, der von `E1` „äußerer Focus“ genannt.
Ein Focus enthält:

- Kontext-Item: Der atomare Wert oder Knoten (in diesem Fall auch Kontext-Knoten gennant), der aktuell bearbeitet wird

- Kontext-Position: Ein Integer-Wert, der die aktuelle Position innerhalb der gerade bearbeiteten Items angibt
- Kontext-Größe: Die Anzahl der gerade bearbeiteten Items
- Kontext Dokument: Das Dokument, repräsentiert als Dokument-Knoten, in dem der Ausdruck gerade ausgewertet wird

In XQuery können arithmetische und logische Operationen, sowie Vergleiche durchgeführt werden, wie es aus normalen Programmier-/Anfragesprachen bekannt ist.
Eine Anfrage folgt generell der FLWR (sprich: „Flower“) Syntax (Codebeispiel 6). [KST]

```
FOR $v IN pfad
LET $w IN pfad
WHERE bedingung
RETURN <element>$v</element>
```
Codebeispiel 6: FLWR

Der `FOR`-Teil iteriert über die Menge von Knoten, die im Pfadausdruck angegeben ist. Diese Knoten werden dann jeweils der Variablen `$v` zugewiesen, mit der dann in diesem Schritt weiter gearbeitet wird. Im `LET`-Teil wird der Variablen `$w` ein Knoten zugewiesen, auf den der zugehörige Pfad zeigt, wobei natürlich auch atomare Werte zurückgegeben werden können. Eine Selektion der Rückgaben findet im `WHERE`-Teil statt: Nur die Iterationen, bei denen die Bedingung erfüllt ist führen zu einer Ausgabe. Diese wird schließlich nach dem reservierten Wort `RETURN` zusammengestellt, wobei auf den Inhalt aller verwendeten Variablen zugegriffen werden kann.
Optional können die LET und WHERE Klauseln auch weggelassen werden (Codebeispiel 7).

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

hat als Rückgabe:

```
<out>
   <one/>
</out>
<out>
   <two/>
</out>
<out>
   <three/>
</out>
```
Codebeispiel 7: Verinfachte FLWR-Anfrage

Der Variablen `$s` in obigem Beipielwerden nacheinander die Werte `<one/>, <two/>` und `<three/>` zugewiesen und jeweils einmal `return` aufgerufen.

In folgendem Beispiel (Codebeispiel 9) soll eine Liste der Autoren und der Bücher, die sie geschrieben haben aus einem XML-Dokument (Codebeipiel 8) mit etwas anderer Struktur ausgelesen und erzeugt werden.

```
<bib>
  <book>
    <title>TCP/IP Illustrated</title>
    <author>
      W. Stevens
    </author>
```

```
    <publisher>Addison-Wesley</publisher>
  </book>
  <book>
    <title>Advanced Programming in the Unix environment</title>
    <author>
      W. Stevens
    </author>
    <publisher>Addison-Wesley</publisher>
  </book>
</bib>
```

Codebeispiel 8: XML-Dokument für das Beipiel

```
<authlist>
 {
   let $input := document("bib.xml")
   for $a in distinct-values($input//author)
   return
     <author>
      {
        <name>
           { $a/text() }
        </name>,
        <books>
         {
           for $b in $input//book
           where $b/author = $a
           return $b/title
         }
        </books>
      }
     </author>
 }
</authlist>
```

erzeugt dieses Ergebnis:

```
<authlist>
  <author>
    <name>
      W. Stevens
    </name>
    <books>
      <title>TCP/IP Illustrated</title>
      <title>Advanced Programming in the Unix environment</title>
    </books>
  </author>
</authlist>
```

Codebeispiel 9

Eine weiter sehr wichtige Eigenschaft von XQuery ist die Fähigkeit alle Arten von Joins auszuführen. Auch Gruppierung, Sortieren und Filtern sind mit XQuery möglich. Die Rückgabe eines Ausdrucks besteht aus Daten im XML-Format und kann damit auch wieder als Datenquelle für übergeordnete Anfragen dienen.
Auch XQuery bietet allerdings nur Boolesches Retrieval an, es wird keine Bewertung der Suchergebnisse vorgenommen.

XIRQL

Diese Schwäche wird mit XIRQL, der „XML Information Retrieval Query Language“ (sprich „circle“) behoben. [FG]
XIRQL erweitert XQL, eine weitere Anfragesprache für XML, um:

- wahrscheinlichkeits-gesteuertes Retrieval mit gewichteten Suchergebnissen
- eine Relevanz-orientierte Suche, bei der allerdings keine Rücksicht auf die Struktur genommen wird und
- erweiterbare Datentypen mit unscharfen Prädikaten.

Die Suche läuft dabei über Teile des Dokuments ab, wobei die Gewichtung von Zwischenergebnissen von ihrem Kontext, also ihrer Position im Dokument abhängt. Diese Eigenschaft ist interessant, da hier keine definierten Pfade für die Anfrage benötigt werden, sondern während der Auswertung dieser Anfrage entschieden wird, wie „gut“ ein Pfad zu dieser Anfrage passt. D. h. es wird ein Ähnlichkeitsvergleich über der Struktur ausgeführt.

Bei der Relevanz-orientierten Suche wird die Ergebnismenge beschränkt, es sollen nur die spezifischsten Elemente, die die Query erfüllen, zurückgegeben werden.
Ein Beispiel für unscharfe Prädikate wäre eine Anfrage

```
Suche einen Künstler namens Ulbrich, der vor etwa 100 Jahren in der
Rhein-Main Gegend lebte
⇒ Ernst Olbrich, Darmstadt, 1899
```

Wie zu sehen ist, wird dabei sowohl unscharf über dem Zeitraum, der geographischen Lage und dem Namen (Ulbrich↔Olbrich) gesucht.

XXL (flexible XML search language)

Einen etwas anderen Ansatz für Ranked Retrieval auf XML-Dokumenten verfolgt XXL. [STW] XXL erweitert XML QL um Textähnlichkeitsvergleiche mit einer Bewertungsfunktion. Dabei ist „Ähnlichkeitsvergleich“ weder nur auf atomare Werte, noch auf Begriffe ähnlicher Schreibweise begrenzt. Es wird eine Ontologie, also eine Wissensdatenbank mit Informationen über die jeweilige Miniwelt (im Prinzip also ein „besserer Thesaurus“) benutzt, um auch Begriffe mit ähnlicher Bedeutung als Ergebnis zuzulassen.

```
SELECT P
FROM urlaubsorte.xml
WHERE Region.~Ort AS P
AND P.#.(~Schwimmen)? ~ "Schwimmen"
AND P.#.~Saison ~ "Sommer"
```

Codebeispiel 10

Ein Anfrage wie in Codebeispiel 10 würde z. B. eine Sehenswürdigkeit, bei der man ganzjährig Schnorcheln kann als mit Ähnlichkeitsmaßen bewertetes Ergebnis zurückliefern. Mit dem ~-Operator wird sowohl der Inhalt von Attributen und Elementen (Saison ~ „Sommer“) als auch der Name des Attributs/Elements selber verglichen, wobei in beiden Fällen die Ontologie zu Hilfe genommen wird. # steht für einen beliebigen Pfad.

In XXL ist es dagegen nicht möglich, einen Ähnlichkeitsvergleich der Struktur, also z.B. der Art der Verschachtelung durchzuführen. Die Bewertung der Ergebnisse erfolgt nur über den Grad der Textähnlichkeit, wobei auch die Elementnamen ähnlich sein dürfen.

Literatur:

[BKOS] „Semi-strukturierte Daten"
Francois Bry, Michael Kraus, Dan Olteanu und Sebastian Schaffert, Universität München

[FG] XIRQL: A Query Language for Information Retrieval in XML Documents,
Norbert Fuhr, Kai Großjohann, Universität Dortmund

[IR] Skript “Information Retrieval”, Wolfgang Lenski

[KST] Datenbanken und XML,
Wassilios Kazakos, Andreas Schmidt, Peter Tomczyk

[MH] Information Mapping: doculine Verlags GmbH, Martin Holzmann
http://www.doculine.com/news/2000/Januar/infomap.htm

[ML] Seminarvortrag „XML“, Matthias Liebisch, Friedrich-Schiller-Universität Jena

[STW] Ähnlichkeitssuche auf XML-Daten,
Sergej Sizov, Anja Theobald, Gerhard Weikum, Universität des Saarlandes

[XF] XML Fragment Interchange, http://www.w3.org/TR/xml-fragment

[XL] XML Linking Language (XLink), http://www.w3.org/TR/xlink/

[XPA] XML Path Language (XPath), http://www.w3.org/TR/xpath

[XPO] XML Pointer Language (XPointer), http://www.w3.org/TR/xptr/

[XQ] XML Query Language (XQuery), http://www.w3.org/TR/xquery