Georg-August-Universität
Göttingen
Institut für Informatik

ISSN 1612-6793
Nummer ZAI-BSC-2017-07

Bachelorarbeit

Im Studiengang „Angewandte Informatik“

Auswertung von XPath-Anfragen über relational gespeicherten XML-Daten

David Nicholas Schlicke

Fakultät für Mathematik und Informatik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen
24. Februar 2017

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Deutschland

Tel. +49 (5 51) 39-1 44 14

Fax +49 (5 51) 39-1 44 15

E-Mail office@informatik.uni-goettingen.de

WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Einbeck, den 24. Februar 2017

Abstract

Die vorliegende Bachelorarbeit behandelt die Abfrage XML-basierter Daten, die in das relationale Datenbankmodell überführt wurden. Dazu werden XPath-Ausdrücke in die Abfragesprache SQL übersetzt, um die dort abgespeicherten XML-Daten abzufragen. Als Basis dient der Datenbankkonverter [4], der die Datenbank aus einer DTD-Datei und der dazugehörigen XML-Datei erzeugt.

Es werden grundlegende Bestandteile von XPath behandelt. Das umfasst absolute Pfadausdrücke, die Dereferenzierungsfunktion, Aggregatfunktionen, Vereinigungen, Filter und absolute Pfadausdrücke in Filtern sein. Dabei werden nur die attribute::-, child::- und self::-Achsen unterstützt, da die zugrundeliegende Datenbank keine Abbildungsmetadaten für andere Achsen bereitstellt.

Die Bachelorarbeit ist für Studierende der Informatik von Interesse, die sich in Datenbanken spezialisieren.

Bachelor-Thesis

Auswertung von XPath-Anfragen über relational gespeicherten XML-Daten

David Nicholas Schlicke

24. Februar 2017

Betreut durch Prof. Dr. Wolfgang May

Datenbanken und Informationssysteme

Institut für Informatik

Georg-August-Universität Göttingen

Inhaltsverzeichnis

1. Einleitung

In der Welt der Datenbanken gibt es verschiedene Modelle zur Verwaltung von Daten:

- XML[1]
- RDF[2]
- SQL[3]

Es entstand die Idee ein Datenbankschema möglichst vollständig in ein anderes Schema zu transformieren. Dazu müssen Assoziationen zwischen den einzelnen Schemata gefunden werden. Daraus lassen sich Regeln ableiten, mit denen die Daten aus den einem Schema in das andere Schema übertragen werden können.

Die dieser Arbeit zugrundeliegende relationale Datenbank wurde aus einer DTD[4]- und der dazugehörigen XML-Instanz erzeugt. Dabei beschreibt die DTD eine hierarchische Datenbankstruktur. Die dazugehörige XML-Datei enthält sämtliche Daten der Datenbank, deren Baumstruktur bestimmt ist durch die DTD. Ein Einblick in die Datenbanktransformation und deren wichtige Bestandteile befindet sich in Kapitel *2.2*.

Auf Grundlage der Datenbanktransformation entstand die Idee dieser Arbeit: Auswertung von XPath-Anfragen über relational gespeicherten XML-Daten.

Die Datenbanktransformation bezieht sich dabei hauptsächlich auf die Datenvollständigkeit und Relationen zwischen den Daten. Das heißt, es existiert keine Dokumentordnung und dementsprechend ist es nicht möglich die Achsen following::, following-sibling::, preceding:: und preceding-sibling:: zu verarbeiten. Da die Abbildungsmetadaten keine Informationen über die Hierarchie der DTD enthält, entfallen die Achsen parent::, ancestor::, ancestor-or-self::, descendant:: und descendant-or-self::. Die Namensraum-Achse namespace:: ist nicht von Belang. Es bleiben nur die Achsen attribute::, child:: und self:: übrig.

Mit den Achsen attribute:: und child:: lassen sich absolute Pfadausdrücke ohne Filter realisieren. Wird zusätzlich die self::-Achse mit einbezogen, dann sind auch Filter respektive relative Pfade möglich. Die Möglichkeit relative Pfade nutzen zu können, ermöglicht es Aggregat- und Dereferenzierungsfunktionen zu verwenden.

Vereinigungen lassen sich nur auf absolute Pfadausdrücke anwenden. Da nur absolute Pfadausdrücke ausgewertet werden können, können Vereinigungen ebenfalls verarbeitet werden. Auf die Ausdrücke some, every, if (XPath 3.0), let und for wird in dieser Arbeit verzichtet.

Im folgenden Kapitel wird zuerst über die benötigten Grundlagen gesprochen. Ein Überblick auf das Datenmodell MONDIAL und in die Erzeugung der Datenbank wird gegeben. Die Abfragesprache XPath wird auf dem zuvor genannten zulässigen Funktionsumfang erläutert und SQL wird umrissen.

[1] Extensible Markup Language
[2] Resource Description Framework
[3] Structured Query Language
[4] Document Type Definition

Mit dem Wissen, wie die Datenbank aufgebaut wurde, ist es dann möglich sich Gedanken über den Übersetzungsprozess von Pfadausdrücken zu machen. Im Kapitel *4* werden die zuvor genannten Punkte behandelt. Besonderheiten, die bei der Implementierung auftraten, werden in Kapitel *5* behandelt. Abschließend wird noch eine kurze Zusammenfassung der wichtigsten Punkte dieser Arbeit gegeben.

2. Grundlagen

2.1. Die MONDIAL-Datenbank

Das verwendete Datenbankschema MONDIAL wurde aus verschiedenen geografischen Web-Quellen generiert. Es diente 1998 als Fallstudie zur Informationsextraktion und –integration (vgl. [5]). Im Verlauf der Jahre wurde das Schema in andere Schemata überführt. Ursprünglich wurde es unter F-Logic[5] erzeugt und später in SQL, XML, RDF und Datalog überführt. Verwendet wird die XML-Variante der MONDIAL-Datenbank, die in das relationale Datenbankmodell transformiert wurde.

Die Datenbank enthält geografische, politische und demographische Daten aus der ganzen Welt.

Für weitere Informationen siehe [5].

2.2. Der Aufbau der verwendeten Datenbank

In diesem Kapitel wird der Aufbau der XML-basierten relationalen Datenbank erläutert. Zum Generieren der Datenbank wird nur die DTD von MONDIAL benötigt. Für die Erzeugung der Datenbank müssen folgende Punkte behandelt werden:

- Wann wird aus einem Elementtyp eine Tabelle?
- Wann ist ein Elementtyp funktional und wann nicht?
- Wie werden Relationen zwischen Elementtypen dargestellt?

Um zu entscheiden ob ein Elementtyp zu einer Tabelle wird, muss es sich um einen komplexen Elementtyp handeln. Ein Elementtyp heißt *komplex*, falls einer der folgenden Punkte erfüllt ist:
Der Elementtyp enthält …

- … viele Unterelemente. Dabei spielt es keine Rolle ob diese einfach oder komplex sind.
- … einen ID-Attributtyp.
- … mehrwertige Attributtypen (IDREFS oder NMTOKENS).

Ansonsten heißt der Elementtyp *einfach*.

Alle komplexen Elementtypen erhalten eine eigene Tabelle. Einfache Elementtypen werden als Eigenschaften der jeweiligen Tabellen angesehen. Attribute werden ebenfalls der Tabelle hinzugefügt mit Ausnahme der mehrwertigen Attributtypen. Diese stellen NM-Beziehungen dar. Daher bekommen diese Attribute ihre eigenen NM-Tabellen. Auch erhalten einfache Elementtypen eine eigene 1N-Tabelle, falls diese mehr als einmal beim Vater-Elementtyp auftreten. Allgemein ausgedrückt erhalten alle einfachen Elementtypen eine 1N-Tabelle, falls deren Kardinalitäten größer als eins sind. Ansonsten werden sie der Vater-Elementtyp-Tabelle hinzugefügt und sind damit funktional. Wichtig hierbei ist, dass auf einen komplexen Elementtyp andere komplexe und einfache

[5] Frame Logic

Elementtypen folgen können, aber auf einen einfachen Elementtyp kann kein weiterer Elementtyp folgen.

Eine weitere Ausnahme bilden komplexe Elementtypen, die Kinder anderer komplexer Elementtypen sind. Durch die hierarchische Baumstruktur besitzt jedes Element genau ein Vater-Element. Daher erhält jedes komplexe Kind-Element eine zusätzliche Eigenschaft, die die eindeutige Beziehung zum Vater-Elementtyp repräsentiert.

Die Namen der Tabellen hängen vom jeweiligen Fall ab. Komplexe Elementtyp-Tabellen werden nach dem jeweiligen Elementtyp benannt. Alle NM-Tabellen bestehen aus einer Kombination der Namen des Vater-Elementtyps und dem des Kind-Elementtyps. Die Namen der Tabelleneigenschaften sind durch die Namen der Element- und Attributtypen bestimmt. Eine Ausnahme bilden manche Namen von Eigenschaften der NM-Tabellen. Das verwendete relationale Modell erzeugt für 1N-Tabellen mit genau einer Eigenschaft den Namen „VALUE“ anstatt dem Namen des Elementtyps. Von außen betrachtet ist nicht ersichtlich welcher Name verwendet wird.

Dafür gibt es das Mapping Dictionary (MD). Dort werden alle Namen der Elementtyp-Tabellen und deren dazugehörige Namen von Eigenschaften auf die intern verwendeten Namen der Tabellen und Eigenschaften abgebildet. Zum Beispiel wird das Paar (*city*, *name*) augebildet auf (*city_name*, *VALUE*). Das heißt der Elementtyp *city* mit der Eigenschaft *name* ist in der Datenbank abgespeichert in der Tabelle *city_name* unter dem Spaltennamen *VALUE*.

Zusätzlich zum MD gibt es Tabellen, die die Namen aller Tabellen in der Datenbank enthalten. Primärschlüssel werden nach demselben Prinzip wie das MD abgespeichert, denn eventuell sind die Namen aus der DTD anders als später in der Datenbank. Fremdschlüssel werden mit ihrem Tabellen- und Spaltennamen in einer Tabelle abgespeichert. Zusätzlich wird noch die Tabelle, wohin der Fremdschlüssel zeigt, gespeichert. Dies ist später für die id(.)-Funktion von XPath nützlich.

Diese speziellen Tabellen werden als *Abbildungsmetadaten* bezeichnet.

Für weitere Informationen siehe [4].

2.3. Extensible Markup Language

XML ist ein einfaches und sehr flexibles Datenmodell, welches vom SGML[6]-Standard abgeleitet wurde. Es wurde durch das W3C[7] standardisiert und dient unter anderem dem strukturierten Datenaustausch über das Internet.

[6] Standard Generalized Markup Language
[7] World Wide Web Consortium

2.3.1. Das Dokumentenformat XML

XML wurde als von Menschen und Maschinen lesbares, hierarchisches, flexibles und strukturiertes Datenmodell entwickelt. Damit wurde außerdem ein plattform- und implementationsunabhängiger Datenaustausch erreicht (vgl. [4], Kapitel 2.1.).

Ein XML-Dokument besteht aus Elementen, die als Metadaten dienen. Diese lassen sich beliebig ineinander verschachteln, solange die Syntax eingehalten wird. Um ein neues Element zu erzeugen, wird das Start-Tag verwendet. Zudem kann ein Element beliebig viele Attribute besitzen. Diese stehen direkt im Start-Tag des jeweiligen Elements. Der Inhalt eines Elements kann leer sein, aus einer Zeichenkette bestehen oder weitere Elemente enthalten. Um den Inhalt eines Elements zu schließen wird das End-Tag verwendet.

$$\underbrace{< elementtype \overbrace{[attribute = "value"]^{*}}^{beliebig\ viele\ Attribute} >}_{Start\text{-}Tag} \underbrace{content}_{\substack{Nichts,\ Text\ oder \\ weitere\ Element\text{-}typen}} \underbrace{</elementtype>}_{End\text{-}Tag}$$

```
<?xml version="1.0" encoding="UTF-8"?>
<documentroot>
    <child1 attribute1="123"> text-content </child1>
    <child2 attribute1="345" attribute2="type">
        <child3> text-content </child3>
        <child4 attribute1="text"/>
    </child2>
</documentroot>
```

XML 1 Beispiel einer XML-Instanz

Im *XML 1* Beispiel heißt die erste Zeile *Prolog*. Diese ist optional und muss am Anfang des Dokumentes stehen. Der Prolog gibt an, um welche XML-Version es sich handelt und welche Kodierung benutzt wird.

Dann folgt das eigentliche Dokument. Dieses besitzt eine hierarchische Baumstruktur, die durch die Elemente dargestellt wird. An dem Element `documentroot` hängt der gesamte Inhalt des XML-Dokumentes. Durch die Verschachtelung der `child`-Elemente entsteht die Baumstruktur, die am Wurzelelement `documentroot` beginnt.

Die Gültigkeit einer XML-Instanz wird mit Hilfe einer DTD sichergestellt, die auch zum weiteren Verständnis benutzt werden kann.

Für weitere Informationen siehe [8].

2.3.2. Document Type Definition

Die DTD beschreibt die Grammatik und Struktur von XML-Dokumenten. Dort werden erlaubte Element- und dessen Attributtypen festgelegt, die hierarchische Ordnung definiert und die Kardinalitäten von Unterelementen beschrieben.

$$< !ELEMENT \quad \underbrace{name}_{\text{Name des Elementtyps}} \quad \underbrace{(\ldots)}_{\text{Inhalt des Elementtyps}} >$$

$$< !ATTLIST \quad \overbrace{elementtypeName}^{\substack{\text{Name des Elementtyps} \\ \text{zu dem die Attribute gehören}}}$$
$$[attrName\ attrType\ attrDeclaration]^{+}$$
$$>$$

Elemente, die später im XML-Dokument stehen, werden über *!ELEMENT* definiert. Hier erhalten Elementtypen einen Namen und können als Inhalt Nichts, eine Zeichenkette (durch #PCDATA dargestellt) oder mehrere Unterelement enthalten. Die Kardinalitäten eines Unterelementes werden durch das anhängen von Zeichen dargestellt. Es gibt ϵ (kein Zeichen), ? (optional), + (mindestens eins) und * (0 bis beliebig viele). Hieraus lassen sich später die Beziehungen der Tabellen gewinnen.

Über *!ATTLIST* werden zu den jeweiligen Elementtypen Attribute definiert. Ein solches Tripel besteht aus einem Namen, einem Typ und einer Deklaration.

Im Folgenden werden die gängigsten Attributtypen aufgelistet:

- ID: ist ein global eindeutiger Identifikator
- IDREF(S): enthält einen oder mehrere ID-Werte anderer Elemente
- CDATA: String-Datentyp

Die Attributdeklarationen legen fest, ob ein solches Attribut vorausgesetzt wird (#REQUIRED), optional ist (#IMPLIED) oder einen festen Wert darstellt (#FIXED).

Die DTD bietet keine Möglichkeit Datentypen mit anzugeben. Es steht nur der Datentyp String zur Verfügung.

Für weitere Informationen siehe [7].

2.3.3. XML Path Language (XPath)

XPath ist eine Abfragesprache zum Zugriff auf Teile eines XML-Dokumentes und wurde ursprünglich als Basis für XSLT[8] und XPointer[9] entworfen. Später bildet es auch die Basis für XQuery[10].

Durch die hierarchische Baumstruktur kann der Baum Schritt für Schritt durchlaufen werden. Um die Auswahl der gewählten Äste einzuschränken, ist es in jedem Schritt möglich, Filter anzuwenden.

$$/Schritt_1[Filter_1]/Schritt_2[Filter_2]/\ \ldots/Schritt_n[Filter_n]$$

Der Filter bezieht sich auf den aktuellen Schritt, auch Kontextknoten genannt, und schränkt die Auswahl der nachfolgenden Knoten ein.

[8] Extensible Stylesheet Language Transformation
[9] XML Pointer Language
[10] XML Query Language

Im Folgenden werden die in der Einleitung genannten Punkte beleuchtet.

Für weitere Informationen zu XPath siehe [9].

2.3.3.1. Absolute Pfadausdrücke und Filter

Ein absoluter Pfad ist ein Ausdruck, der den Baum Schritt für Schritt in der gegebenen Reihenfolge auswertet. Der Einstiegspunkt dabei ist das Wurzelelement *rootElement*. Ausgehend vom Wurzelelement wird über alle Elemente des Typs $element_1$ iteriert. Es werden die Elemente ausgewählt, für die der Filter WAHR ist. Auf den Elementen, die übrig bleiben wird dies rekursiv bis zum Elementtyp $element_n$ fortgesetzt. Anders gesagt wird genau dem Pfad des Baumes gefolgt und bei der Auswertung nicht im Baum umhergesprungen.

$$/child :: rootElement/child :: element_1[filter_1]/ .../child :: element_n[filter_n]$$
$$\Leftrightarrow$$
$$/rootElement/element_1[filter_1]/ .../element_n[filter_n]$$

Anmerkung: Die child::-Achse wird abgekürzt durch ein einfaches „/“.

Ein Filter bezieht sich dabei immer auf den dazugehörigen Kontextknoten. Damit das Element in die weitere Auswertung gelangt, muss es der Bedingung des Filters genügen, das heißt die Bedingung muss WAHR sein. Dabei kann die eine Seite des Filters ebenfalls ein absoluter Pfad sein, der gegen den Kontextknoten ausgewertet wird. Dieser heißt dann *relativer Pfad*. Das heißt, für jedes Kind-Element des Kontextknotens wird rekursiv in die Tiefe gegangen und der Pfadausdruck mit der Bedingung ausgewertet. Ergibt die Auswertung WAHR, so wird der Elementtyp $element_{i+1}$ der neue Kontextknoten. Auf diesen wird dann wieder der Filter ausgewertet usw.

XPath bietet die Möglichkeit mehrere Filter auf einen Kontextknoten anzuwenden. Diese können in einem einzigen Filter durch Konjunktion zusammengefasst werden.

$$.../element_i[filter_1] ... [filter_n]$$
$$\Leftrightarrow$$
$$.../element_i[filter_1\ and\ ...\ and\ filter_n]$$

2.3.3.2. Vereinigungen

Vereinigungen führen mehrere verschiedene Pfade zusammen. Dabei können diese in kompakter Form angegeben werden, falls sie denselben Aufbau haben:

$$path_{left}/(element_i| ... |element_j)/path_{right}$$

Oder es sind unterschiedlich aufgebaute Pfadausdrücke:

$$path_1|\ path_2|\ ...|\ path_n$$

Dabei kann die kompakte Schreibweise auch wie zuletzt dargestellt werden. Im Weiteren wird diese Form als ausgedehnte Form bezeichnet.

2.3.3.3. Aggregatfunktionen

Diese Funktionen berechnen aus mehreren Eigenschaften oder Tabellen einen Wert. Es stehen die Funktionen avg(.), sum(,), max(.), min(.) und count(.) zur Verfügung. Während die ersten vier Funktionen auf Textinhalten oder Attributen von Elementen arbeiten, berechnet count(.) die Anzahl der Elemente.

Aggregatfunktionen können in Pfadausdrücken und Filtern vorkommen. Dabei ist zu unterscheiden, ob eine solche Funktion am Beginn eines Pfades steht oder mitten drin auftaucht.

$$.../element_i/aggr\text{-}fct(\underbrace{abs\text{-}path_2}_{relativ\ zu\ element_i})$$

Steht die Funktion am Beginn, so liefert diese einen einzigen Wert. Befindet sie sich an beliebiger Stelle, so liefert die Funktion mehrere Werte, da die Funktion gegen den Kontextknoten ausgewertet wird, zu dem die Aggregatfunktion relativ ist.

2.3.3.4. Die id(.)-Funktion

Die id(.)-Funktion dereferenziert Attribute vom Typ IDREF(S). Das heißt, es wird im Baum das Element gesucht, welches das Attribut vom Typ ID mit dem gleichen Wert besitzt. Diese Funktion kann auch in Pfadausdrücken sowie Filtern vorkommen.

$$.../element_i/id(\underbrace{@idref\text{-}attrib}_{relativ\ zu\ element_i})/...$$

Wird das Attribut vom Typ IDREF(S) des Elements $element_i$ dereferenziert, dann wird der nächste Kontextknoten das gefundene Element sein. Dieses kann an beliebiger Stelle im Baum stehen. Das heißt es wird im Baum umhergesprungen. Alle nachfolgenden Auswertungsschritte werden auf dem neuen Kontextknoten ausgeführt.

2.3.3.5. Relative und absolute Pfadausdrücke

Pfadausdrücke heißen *relativ*, wenn in Filtern Pfade stehen, die gegen dessen Kontextknoten ausgewertet werden. Dazu wird die self::-Achse verwendet.

$$.../element_i[\underbrace{self::*/child\text{-}path}_{relativ\ zu\ element_i}\ operator\ value]/...$$

$$\Leftrightarrow$$

$$.../element_i[\underbrace{./child\text{-}path}_{relativ\ zu\ element_i}\ operator\ value]/...$$

$$\Leftrightarrow$$

$$.../element_i[\underbrace{child\text{-}path}_{relativ\ zu\ element_i}\ operator\ value]/...$$

Anmerkung: Der Wert *value* kann auch ein Pfadausdruck sein. Dieser ist absolut und nicht relativ zu $element_i$ und kann auch als eigenständiger Pfadausdruck ausgewertet werden. Dies entspräche einem Semijoin in SQL.

Daraus folgen zwei Fälle: Ein Vergleich zwischen …

- … einem Wert und anderen Werten oder
- … einem Element und anderen Elementen

Werte mit Elementen zu vergleichen und umgekehrt ist möglich, aber es wird aus dem Element ein String generiert. Dieser String besteht aus sämtlichen Strings aller Unterelemente. Daher kann in Ausdrücken zum Großteil auf die text()-Funktion verzichtet werden.

2.4. Structured Query Language (SQL)

SQL ist eine Abfragesprache, die in den siebziger Jahren aus IBM[11]'s SEQUEL entstanden ist. Sie dient der Manipulation relationaler Datenbanken, dem Abfragen von Datensätzen und basiert auf der relationalen Algebra.

SQL lässt sich in mehrere Kategorien unterteilen:

1) Data Manipulation Language (DML): dient der Datensatzmanipulation (Abfragen, Einfügen, Löschen und Verändern)
2) Data Definition Language (DDL): dient der Datenbankmanipulation (Einfügen, Löschen und Verändern von Tabellen/Schemata)
3) Transaction Control Language (TCL): dient der Datenintegrität
4) Data Control Language (DCL): dient der Zugriffssicherheit (Rechteverwaltung)

Ein wichtiger Unterschied zu XML-basierten Datenbanken ist, dass es bei relationalen Datenbanken viele verschiedene Datentypen gibt.

Für weitere Informationen zu SQL siehe [6].

Im Folgenden wird nur die DML weiter ausgeführt. Die restlichen drei Punkte sind für diese Arbeit nicht von Belang.

2.4.1. SFW-Klauseln

SFW[12]-Klauseln sind die einfachsten Abfragen in SQL. Mit SELECT werden die Tabellen oder Eigenschaften gewählt, die ausgegeben werden sollen. Die FROM-Klausel gibt an, von welchen

[11] International Business Machines Corporation
[12] SELECT-FROM-WHERE Abkürzung

Tabellen die Ergebnismenge gewählt werden soll und in der WHERE-Klausel stehen Bedingungen, welche die Ergebnismenge einschränken sollen.

```
SELECT properties
FROM tables
WHERE conditions
```

SQL 1 Grundaufbau von SWF-Klauseln

Durch Konjunktion und Disjunktion lassen sich beliebig viele Bedingungen in einer WHERE-Klausel unterbringen. Negation von Bedingungen ist ebenfalls möglich.

2.4.2. Verknüpfungen von Tabellen

Es gibt zwei Arten Tabellen miteinander zu verknüpfen. Zum einen kann der Primär- und Fremdschlüssel gleichgesetzt in die WHERE-Klausel geschrieben werden oder sie werden mittels der JOIN-Klausel gleichgesetzet.

```
SELECT …
FROM table1, …, tablen
WHERE table1."ID"=table2."t1ID"
AND table2."ID"=table3."t2ID"
AND …
```

SQL 2 Verknüpfung der einzelnen Tabellen

```
SELECT …
FROM table1
JOIN table2 ON table1."ID"=table2."t1ID"
JOIN table3 ON table2."ID"=table3."t2ID"
JOIN …
```

SQL 3 Äquivalente Darstellung mittels JOIN

Anmerkung: `JOIN` ist die Kurzform von `INNER JOIN`. Für weitere Informationen zur JOIN-Klausel siehe [10].

2.4.3. Vereinigungen

Vereinigungen in SQL sind zwischen mindestens zwei SFW-Klauseln mit derselben Spaltenanzahl möglich. Außerdem müssen die jeweiligen Spalten die gleichen Datentypen haben und in der gleichen Reihenfolge auftreten.

```
(SELECT properties
 FROM table1
 WHERE conditions)
UNION
(SELECT properties
 FROM table2
 WHERE conditions)
```

SQL 4 Grundaufbau von Vereinigungen

Dies ist ein Unterschied zu der Vereinigung von XPath. XPath ist es egal welche Elementtypen vereinigt werden. Dort können zum Beispiel Städte mit Ländern vereinigt werden, während dies in SQL nur möglich wäre, falls die oben genannten Voraussetzungen erfüllt sind.

2.4.4. Aggregatfunktionen

Aggregatfunktionen berechnen aus den Werten einer Spalte einen einzelnen Wert. Zu den Aggregatfunktionen zählen AVG(.), MAX(.), MIN(.), SUM(.) und COUNT(.). Letztere berechnet die Anzahl der Elemente in der Ergebnismenge. NULL-Werte werden mitgezählt, bei den anderen Funktionen ignoriert.

Aggregatfunktionen können in der SELECT-Klausel oder als Bedingung auftreten. Als Bedingung ist es nicht möglich diese in die WHERE-Klausel zu schreiben. Dies wird von SQL nicht unterstützt. Stattdessen gibt es die HAVING-Klausel, die für diesen Zweck hinzugefügt wurde.

Mit der GROUP-BY-Klausel können Gruppierungen erstellt werden. Ohne eine Gruppierung wird eine Aggregatfunktion auf alle ausgewählten Spalten ausgeführt. Mit einer Gruppierung ist es möglich, dies auf bestimmte Spalten einzuschränken. Statt einem Wert über alle Spalten erhält man mehrere Werte für jede einzelne Gruppierung. Dabei kann man GROUP-BY-Klauseln ohne HAVING-Klauseln stehen haben und umgekehrt.

```
SELECT columns
FROM tables
WHERE conditions
GROUP BY columns
HAVING aggr-fct(column) operator value
```

SQL 5 Grundaufbau von SWFGBH-Klauseln

2.4.5. Unterabfragen

Unterabfragen lassen sich in zwei Kategorien unterteilen. Zum einen in korrelierte Unterabfragen, die in Bezug zu der Hauptabfrage stehen, oder als eigenständige SQL-Unterabfrage die nicht in Bezug zur Hauptabfrage steht.

```
SELECT columns
FROM table AS t1
WHERE EXISTS
       (SELECT column
        FROM table AS t2
        WHERE t2."id"=t1."t2id")
```

SQL 6 Grundaufbau einer korrelierten Unterabfrage

```
SELECT columns
FROM tables
WHERE property operator [ANY|ALL]
       (SELECT column
        FROM table)
```

SQL 7 Grundaufbau einer Unterabfrage

Anmerkung zu *SQL 6*: Es ist möglich, hier statt einem expliziten Vergleich mit der Unterabfrage, die EXISTS-Klausel zu verwenden.

Anmerkung zu *SQL 7*: Das Schlüsselwort `ANY` bzw. `ALL` sagt aus, dass die `property` mit einem beliebigen Element bzw. mit allen Elementen der Unterabfrage der Bedingung genügen muss. Die Angabe des Schlüsselwortes ist optional.

2.4.6. Datenbanken und Schemata

SQL bietet die Möglichkeit Tabellen aus verschiedenen Datenbanken und Schemata zu wählen. Dazu muss nur die FROM-Klausel erweitert werden.

```
SELECT columns
FROM [databaseName].[schemaName].tableName [AS tableAlias]
```

SQL 8 Verwendung von Tabellen anderer Datenbanken und Schemata

Vor jeder Tabelle werden der Name der Datenbank und des Schemas hinzugefügt.

In dieser Arbeit wird nur eine Datenbank verwendet, daher ist es nicht nötig diesen Namen mit anzugeben. Jedoch kann diese Datenbank mehrere Schemata besitzen. Das verwendete Schema heißt „xml“.

3. Verwendete Software

3.1. Saxon

Saxon ist eine API[13] zur Bearbeitung von XML-Dokumenten, die von Michael Kay in Java und .NET entwickelt wurde. Es enthält Implementationen von XSLT 2.0, XQuery 3.0, XPath 3.0 und XSD[14] 1.1, die alle durch das W3C festgelegt worden sind.

Es gibt mehrere Versionen von Saxon. Eine Home-Edition (HE), Professional-Edition (PE) und Enterprise-Edition (EE). Die HE wurde als MPL[15] veröffentlicht und die PE und EE wurden mit einer Saxonica-Lizenz veröffentlicht. Letztere bieten zusätzliche Features zum Arbeiten mit der Welt rund um XML, während die HE ein vollständiges Grundgerüst liefert.

Für weitere Informationen siehe [3].

3.2. PostgreSQL

PostgreSQL ist ein objektrelationales Datenbankmanagementsystem, welches durch die PostgreSQL Global Development Group entwickelt wird. Es wurde unter einer PostgreSQL-Lizenz veröffentlicht, einer Open-Source-Lizenz. Zusätzlich bietet es Programmierschnittstellen für C/C++, Java, .Net, und mehr.

Statt PostgreSQL kann jedes beliebige relationale DBMS verwendet werden.

Für weitere Informationen siehe [2].

[13] Application Programming Interface
[14] XML Schema Definition
[15] Mozilla Public License

4. Entwurf

In diesem Kapitel wird der grundlegende Entwurf des XPath-SQL-Konverters erläutert.

Begonnen wird mit einfachen absoluten Pfad-Ausdrücken, die im weiteren Verlauf dieses Kapitels an Komplexität gewinnen. Die Kernbestandteile wurden in Kapitel *2.3.3* erläutert.

4.1. Absolute Pfadausdrücke

Zuerst werden Pfade der Länge eins betrachtet und diese Stück für Stück erweitert. Dies ist der sogenannte Hauptpfad.

$$//element_1/\ .../element_n$$

Das Wurzelelement repräsentiert den gesamten Inhalt, ohne den Prolog eines XML-Dokuments. Es enthält keinerlei Attribute, nur Unterelemente. Für die Konvertierung in SQL spielt dieses Element keine Rolle. Es kann genauso gut weggelassen werden. In XPath wird daher meistens am Beginn jeden Pfades „//“ statt „/“ verwendet, um den Wurzelknoten nicht immer hinschreiben zu müssen. Alle Elemente an dem Wurzelknoten von MONDIAL sind komplexe Elementtypen, dementsprechend auch Tabellen.

Bei einem Pfad der Länge eins ohne Filter muss der Elementtyp komplex sein. Ansonsten kann nicht gesagt werden, zu welcher Tabelle die Eigenschaft gehört. In XPath kann zum Beispiel nach sämtlichen Namenselementen gesucht werden. In SQL hingegen ist dies deutlich schwieriger, da die Namen in vielen verschiedenen Tabellen stehen. Dies wird vom Aufbau der Datenbank nicht unterstützt, siehe Kapitel *2.2*.

Bei einem 1-Schritt Pfad wird geprüft, ob es sich um eine Tabelle handelt. Wenn es eine Tabelle ist, wird alles von dieser ausgewählt, da es keinen weiteren Schritt zur Verfeinerung gibt.

```
SELECT element₁.*
FROM element₁
```

SQL 9 Ein-Schritt Abfrage

Anmerkung: Bei manchen Datenbankmanagementsystemen (DBMS) muss die Tabelle in der SELECT-Klausel in Anführungszeichen stehen, da diese eventuell Groß- und Kleinschreibung berücksichtigen. Zum Beispiel beim DBMS Oracle müssen Anführungszeichen bei den Tabellen mit hingeschrieben werden.

Erweitert man diese Abfrage auf zwei Schritte, dann folgt durch die hierarchische Struktur von XML, das der zweite Elementtyp ein Kind vom ersten Elementtyp ist. Daher gibt es eine Relation zwischen diesen. Durch die Überführung in das relationale Modell entstehen dadurch unterschiedliche Möglichkeiten. Der zweite Elementtyp kann ebenfalls komplex sein oder es handelt sich um eine 1N-Tabelle oder er ist eine Eigenschaft der ersten Tabelle. Dabei bildet der hintere Elementtyp die Ergebnismenge. Bei zwei komplexen Elementtypen verknüpft man diese miteinander, da sie in

Relation zueinander stehen. So wird sichergestellt, dass jeder Eintrag der ersten Tabelle in Relation zu den richtigen Kind-Einträgen der zweiten Tabelle steht.

Der zweite Elementtyp kommt immer in die Auswahl. Daraus entsteht folgende Abfrage:

```
SELECT element₂.*
FROM element₁
JOIN element₂ ON element₁."id"=element₂."el₁id"
```

SQL 10 Zwei-Schritt Abfrage komplexer Elementtypen

Die Verknüpfung kann eigentlich weglassen werden, da kein Filter auf den ersten Elementtyp angewendet wird. Für später ist es jedoch wichtig diese zu erhalten.

Bei 1N-Tabellen muss der erste Elementtyp komplex sein. Daher werden beide Tabellen miteinander verknüpft, um mögliche Filter des ersten Elementtyps berücksichtigen zu können.

```
SELECT element₁_element₂.*
FROM element₁
JOIN element₁_element₂
ON element₁."id"=element₁_element₂."el₁id"
```

SQL 11 Zwei-Schritt Abfrage mit 1N-Tabellen

Handelt es sich stattdessen um eine Eigenschaft, so wird vom ersten Elementtyp genau die Eigenschaft gewählt.

```
SELECT element₁."element₂"
FROM element₁
```

SQL 12 Zwei-Schritt Abfrage nach einer Eigenschaft

Wird die Abfrage auf drei Schritte erweitert, ergeben sich zu den vorherigen Fällen zwei weitere Fälle. Der eine Fall betrifft nur die 1N-Tabellen. Die ersten zwei Elementtypen erzeugen eine 1N-Tabelle. Die Abfrage der jeweiligen Eigenschaften erfolgt über den dritten Elementtyp.

```
SELECT element₁_element₂."element₃"
FROM element₁
JOIN element₁_element₂
ON element₁."id"=element₁_element₂."el₁id"
```

SQL 13 Drei-Schritt Abfrage einer 1N-Tabelle nach einer Eigenschaft

Der andere Fall ignoriert den mittleren Elementtypen, da dieser einen einfachen Elementtyp mit nur einem Datenwert darstellt. Dieser wird beim Erzeugen der Datenbank der Tabelle des ersten Elementtyps hinzugefügt.

```
SELECT element₁."element₃"
FROM element₁
```

SQL 14 Drei-Schritt Abfrage einer Tabelle nach einer Eigenschaft, ohne den mittleren Elementtyp.

Bei den anderen Fällen stehen zu Beginn zwei komplexe Elementtypen. Einfache Elementtypen können zu Beginn nicht auftreten, da diese keine Relation zu nachfolgenden komplexen Elementtypen haben können. Anders ausgedrückt sind einfache Elementtypen in der Baumstruktur von XML Blattknoten. Daher beginnen alle weiteren n-Schritt Pfadausdrücke, $n \geq 3$, mit komplexen Elementtypen. Am Ende des Pfades steht einer der fünf Fälle.

$$//\underbrace{el_1/\ .../el_{n-3}}_{komplex}/\ \overbrace{\underbrace{el_{n-2}}_{einer\ der\ fünf\ Fälle}\ /el_{n-1}/el_n}^{komplex}$$

Die fünf Fälle kurz zusammengefasst:

1. `element`$_n$ bildet eine Tabelle.
2. `element`$_n$ ist eine Eigenschaft und `element`$_{n-1}$ bildet eine Tabelle.
3. `element`$_n$ und `element`$_{n-1}$ bilden eine 1N-Tabelle.
4. `element`$_{n-1}$ und `element`$_{n-2}$ bilden eine 1N-Tabelle und `element`$_n$ ist eine Eigenschaft der 1N-Tabelle.
5. `element`$_{n-2}$ bildet eine Tabelle und `element`$_n$ ist eine Eigenschaft der Tabelle. `element`$_{n-1}$ wird ignoriert.

Die ersten *n-3* Schritte werden aufsteigend miteinander verknüpft und mit dem Elementtyp `element`$_{n-2}$ verknüpft, um später auf alle Schritte die Filter anwenden zu können.

4.2. Relative Pfadausdrücke

Filterinhalte sind boolesche Ausdrücke, das heißt sie müssen WAHR oder FALSCH sein. Dementsprechend werden nur die wahren Aussagen der Ergebnismenge hinzugefügt. Da es für jeden Elementtyp des Hauptpfades Filter geben kann, müssen alle Verknüpfungen des Hauptpfades mitgenommen werden. Selbst dann, wenn ein Elementtyp keinen Filter besetzt, aber die benachbarten schon, dient der Elementtyp als Verbindung zwischen den anderen beiden.

Ein Elementtyp mit einem Filter sieht ganz allgemein wie folgt aus:

$$.../element_i[\underbrace{element_{i+1}/\ .../element_j}_{relativ\ zu\ element_i}\ operator\ value]/\ ...\ ,$$
$$operator \in \{>, <, \geq, \leq, !=, =\}$$

Der Hauptpfad und der relative Pfad lassen sich wie im Kapitel zuvor übersetzen, sofern die Elementtypen des relativen Pfades keine direkten Eigenschaften der Tabelle von `element`$_i$ sind. Bei der Übersetzung des relativen Pfades muss gegebenenfalls noch der Elementtyp `element`$_i$ mit einbezogen werden, um 1N-Tabellen berücksichtigen zu können. Beim Elementtyp `element`$_j$ muss es sich um eine Eigenschaft handeln, da komplexe Elementtypen nicht mit Werten verglichen werden können. Nach dem übersetzen des relativen Pfades fehlt noch die Einschränkung durch den Vergleichsoperator `operator` und den Wert `value`. Dazu wird die WHERE-Klausel auf die Eigenschaft mit der Bedingung angewendet.

```
SELECT table."elementj"
FROM table
WHERE table."elementj" operator value
```

SQL 15 Allgemeine Übersetzung des relativen Pfades

Auf diese Weise erhält man alle Tupel, für die diese Bedingung WAHR ist. Diese Unterabfrage muss dann nur noch in Verbindung mit dem übersetzten Hauptpfad gebracht werden. Da der relative Pfad

eine Relation zum Elementtyp $element_i$ besitzt, existiert ein Fremdschlüssel, der auf die Tabelle $element_i$ zeigt. Ändert man die SELECT-Klausel der Unterabfrage auf den Fremdschlüssel ab, muss der Primärschlüssel von $element_i$ in der Ergebnismenge der Unterabfrage enthalten sein.

```
SELECT element_i.*
FROM element_i
WHERE element_i."id" IN
   (SELECT table."el_iid"
    FROM table
    WHERE table."element_j" operator value)
```

SQL 16 Allgemeine Übersetzung von Filtern, dessen Eigenschaften in anderen Tabellen stehen

Dabei spielt es keine Rolle wie lang der relative Pfad ist. Dieser steht immer in Relation zum Kontextknoten $element_i$.

Enthält der Hauptpfad mehrere Filter dieser Art, so werden diese durch Konjunktion in die WHERE-Klausel der Hauptabfrage geschrieben. Dabei bezieht sich jeder Filter auf seinen dazugehörigen Kontextknoten.

Handelt es sich bei den Elementtypen des relativen Pfades um Eigenschaften die in der Tabelle von $element_i$ stehen, so werden diese Bedingungen ohne Unterabfragen der WHERE-Klausel hinzugefügt.

```
SELECT element_i.*
FROM element_i
WHERE element_i."element_j" operator value
```

SQL 17 Allgemeine Übersetzung von Filtern, dessen Eigenschaften in derselben Tabelle stehen

Es gibt einen Spezialfall, bei dem es weder Operator `operator` noch Wert `value` gibt. Dieser Filter bedeutet, dass ein solcher Elementtyp existieren muss beziehungsweise nicht leer sein darf. So wird bei Eigenschaften, die in der Tabelle des $element_i$ stehen, auf NOT NULL und bei Eigenschaften, die in anderen Tabellen stehen, mit der EXISTS-Klausel geprüft.

```
SELECT element_i.*
FROM element_i
WHERE element_i."element_j" IS NOT NULL
```

SQL 18 Allgemeine Übersetzung von Filtern, dessen Eigenschaften existieren müssen

```
SELECT element_i.*
FROM element_i
WHERE EXISTS
   (SELECT table."element_j"
    FROM table
    WHERE table."el_iid"=element_i."id")
```

SQL 19 Allgemeine Übersetzung von Filtern, dessen Eigenschaften anderer Tabellen existieren müssen

Jeder Filter kann aus mehreren Bedingungen bestehen, welche durch Konjunktion oder Disjunktion verknüpft sind. Zusätzlich kann man Bedingungen negieren. Dies wird beim Übersetzen 1:1 übernommen.

Anmerkung: Es werden hier zum Teil explizite Typumwandlungen benötigt. Siehe dazu Kapitel *4.7.2*.

4.3. Vereinigungen

Wie in Kapitel *2.4.3* beschrieben, ist die Vereinigung von Tabellen in SQL nur teilweise möglich. Es gibt zwei Schreibweisen für Vereinigungen in XPath. Um die Übersetzung in SQL zu vereinfachen wird die nicht kompakte Form verwendet. Es müssen alle kompakten Formen in die ausgedehnte Form überführt werden. Die einzelnen Teile lassen sich separat voneinander übersetzen und müssen nur noch mit dem UNION-Schlüsselwort verbunden werden.

$$// \ldots (\text{element}_i \mid \ldots \mid \text{element}_j) / \ldots$$
$$\Leftrightarrow$$
$$// \ldots / \text{element}_i / \ldots \mid \ldots \mid // \ldots / \text{element}_j / \ldots$$

XPath 1 Umformung von Vereinigungen der kompakten Form in die ausgedehnte Form

```
(SFW) UNION (SFW) UNION … UNION (SFW)
```

SQL 20 Übersetzung von XPath 1

Vereinigungen, die am Ende eines Pfadausdruckes stehen, werden anders behandelt. Diese bestimmen die zu wählenden Elemente der Ergebnismenge. Das heißt, alle Elementtypen in der Vereinigung, die am Ende eines Ausdruckes stehen, landen in der SELECT-Klausel.

$$// \ldots / \text{element}_i / (\text{element}_{i+1} \mid \ldots \mid \text{element}_j)$$

XPath 2 Beispiel eines UNION-Pfadausdruckes, der die SELECT-Klausel beeinflusst

```
SELECT elementᵢ."elementᵢ₊₁", …, elementᵢ."elementⱼ"
FROM elementᵢ
```

SQL 21 Übersetzung von XPath 2

4.4. Aggregatfunktionen

4.4.1. Aggregatfunktionen im Hauptpfad

Eine Aggregatfunktion, die sich über den gesamten absoluten Pfad erstreckt, liefert als Ergebnismenge eine einzige Zahl. Steht die Funktion an beliebiger Stelle des absoluten Pfades, so ist diese relativ zum Kontextknoten und wird gegen diesen ausgewertet. Daraus ergibt sich eine Ergebnismenge mit mehreren Zahlen.

Allgemeiner Aufbau:

$$\ldots / element_i / aggr\_fct(element_{i+1} / \ldots / element_n)$$

Dazu ein Beispiel:

$$1)\ //country/\underbrace{avg(population)}_{relativ\ zu\ country}$$
$$2)\ \underbrace{avg(//country/population)}_{absolut}$$

XPath 3 Beispiel avg(.)-Funktion

Dabei liefert 1) die durchschnittliche Bevölkerung eines jeden Landes und 2) die durchschnittliche Bevölkerung aller Länder zusammen. Das heißt die Aggregatfunktion ist bei 1) relativ zum Elementtyp country und bei 2) absolut.

Die absoluten Aggregatfunktionen lassen sich nahezu vollständig mit dem bisherigen vorgehen bearbeiten. Der kleine Unterschied liegt darin, die SELECT-Klausel mit der Aggregatfunktion zu erweitern. Zusätzlich muss wegen der fehlenden Datentypen in der DTD eine explizite Typumwandlung vorgenommen werden.

```
SELECT AVG(CAST(coupop."population" AS NUMERIC))
FROM xml."country_population" AS coupop
JOIN xml."country" AS cou
ON cou."car_code"=coupop."countryID"
```

SQL 22 Beispiel einer absoluten Aggregatfunktion. XPath 3 2)

Für die relativen Aggregatfunktionen wird zusätzlich die GROUP-BY-Klausel benötigt. Diese wird in Verbindung mit Aggregatfunktionen genutzt, um Ergebnismengen durch eine oder mehrere Spalten zu gruppieren. Ohne Gruppierung wird die Aggregatfunktion auf allen Werten ausgewertet.

```
SELECT AVG(CAST(coupop."population" AS NUMERIC))
FROM xml."country_population" AS coupop
JOIN xml."country" AS cou
ON cou."car_code"=coupop."countryID"
GROUB BY cou
```

SQL 23 Beispiel einer relativen Aggregatfunktion. XPath 3 1)

4.4.2. Aggregatfunktionen in Filtern

Steht eine Aggregatfunktion in einem Filter als Teil einer Bedingung, so sieht dies ganz allgemein wie folgt aus:

$$\ldots/element_i[\underbrace{\ldots/element_j/\overbrace{aggr\_fct(element_{j+1}/\ldots/element_n)}^{relativ\ zu\ element_j}\ operator\ value}_{relativ\ zu\ element_i}]/\ldots$$

Betrachtet man das allgemeine Schema, dann stimmt der relative Teil mit dem allgemeinen Aufbau aus Kapitel *4.4.1* überein. Daher wird der Teil separat übersetzt und in Verbindung mit dem Hauptpfad gebracht. Die Verbindung wird mittels EXISTS-Klausel und korrelierter Unterabfrage erzeugt, da die Unterabfrage relativ zum Elementtyp $element_i$ ist.

In der WHERE-Klausel sind keine Aggregatfunktionen zulässig. Dazu stellt SQL das Schlüsselwort HAVING bereit. Dies entspricht der WHERE-Klausel für Bedingungen in Zusammenhang mit Aggregatfunktionen.

Ist die Aggregatfunktion relativ zum Kontextknoten $element_i$, dann wird in der HAVING-Klausel die Aggregatfunktion ohne Gruppierung relativ zu $element_i$ ausgewertet. Beim zweiten Fall ist die Aggregatfunktion relativ zu $element_j$, während $element_j$ relativ zu $element_i$ ist. Daher ist zusätzlich zur HAVING-Klausel eine Gruppierung über $element_j$ nötig, da ansonsten über alle $element_i$ ausgewertet wird.

Das folgende Beispiel liefert alle Länder, die mindestens eine Stadt mit mehr als einer Millionen Einwohnern haben.

$$//country[.//city/\overbrace{\max(population)}^{relativ\ zu\ city} > 1000000]$$
$$\underbrace{\qquad\qquad\qquad\qquad}_{relativ\ zu\ country}$$

XPath 4 Beispiel einer Aggregatfunktion in einem Filter

Der relative Pfad wird nach Kapitel *4.4.1* zu:

```
SELECT MAX(CAST(citpop."population" AS NUMERIC))
FROM xml."city_population" AS citpop
JOIN xml."city" AS cit
ON cit."id"=citpop."cityID"
GROUP BY cit
```

SQL 24 Relativer Pfad von XPath 4 in SQL nach Kapitel 4.4.1

Um die Städte dem jeweiligen Land zuordnen zu können, muss die Unterabfrage mit der Hauptabfrage in Beziehung gesetzt werden.

```
SELECT MAX(CAST(citpop."population" AS NUMERIC))
FROM xml."city_population" AS citpop
JOIN xml."city" AS cit
ON cit."id"=citpop."cityID"
WHERE cit."country"=cou."car_code"
GROUP BY cit
```

SQL 25 Relativer Pfad von XPath 4 in SQL als korrelierte Unterabfrage

Es fehlt noch die Bedingung. Wie zuvor beschrieben, muss diese in der HAVING-Klausel stehen.

```
SELECT MAX(CAST(citpop."population" AS NUMERIC))
FROM xml."city_population" AS citpop
JOIN xml."city" AS cit
ON cit."id"=citpop."cityID"
WHERE cit."country"=cou."car_code"
GROUP BY cit
HAVING MAX(CAST(citpop."population" AS NUMERIC)) > 1000000
```

SQL 26 Relativer Pfad von XPath 4 in SQL als korrelierte Unterabfrage mit der Bedingung

Verbunden mit dem Hauptpfad liefert die Abfrage alle Länder, die mindestens eine Stadt mit mehr als einer Millionen Einwohner besitzt.

```
SELECT cou.*
FROM xml."country" AS cou
WHERE EXISTS
    (SELECT MAX(CAST(citpop."population" AS NUMERIC))
     FROM xml."city_population" AS citpop
     JOIN xml."city" AS cit
     ON cit."id"=citpop."cityID"
     WHERE cit."country"=cou."car_code"
     GROUP BY cit
     HAVING MAX(CAST(citpop."population" AS NUMERIC)) > 1000000)
```

SQL 27 XPath 4 in SQL

Konvention: Sollte eine Aggregatfunktion über den gesamten relativen Pfad reichen, dann sollte sie vor der self::-Achse stehen.

4.5. Filter mit absoluten Pfadausdrücken

Die absoluten Pfadausdrücke können entweder Elemente oder Strings in der Ergebnismenge enthalten. Entsprechend dazu muss der relative Pfad dieselben Ergebnistypen liefern um diese überhaupt miteinander vergleichen zu können, ansonsten ist die Bedingung für alle Elemente des Typs $element_i$ FALSCH. Vergleiche dieser Art sind genau dann WAHR, falls mindestens ein Element vom Typ $element_j$ vergleichbar ist mit einem Element/String der Ergebnismenge des absoluten Pfades und die Bedingung erfüllt.

$$.../element_i[\underbrace{element_{i+1}/.../element_j}_{relativ\ zu\ element_i}\ operator\ \underbrace{abs\text{-}path}_{nicht\ relativ\ zu\ element_i}]/\ldots,$$
$$operator \in \{>, <, \geq, \leq, !=, =\}$$

In SQL sind die Vergleiche auf Einträge einer Tabelle (zeilenweise) und einzelne Eigenschaften begrenzt. Daraus ergeben sich zwei mögliche Fälle: Zum einen können Tabellen miteinander verglichen werden oder Eigenschaften von Tabellen, aber Vergleiche zwischen Eigenschaften und Tabellen sind nicht möglich. Um Tabellen miteinander zu vergleichen, reicht es aus die Primärschlüssel zu vergleichen, da diese eindeutig sind.

Die einzelnen Teile - Hauptpfad, relativer Pfad und absoluter Pfad - lassen sich separat übersetzen. Für den relativen Pfad muss der WHERE-Klausel die Bedingung des Filters hinzugefügt werden, da der absolute Pfad den relativen Pfad einschränken soll. Die Elemente vom Typ $element_j$ müssen verglichen werden mit den Elementen der Ergebnismenge des absoluten Pfades. Dabei muss $element_j$ mit mindestens einem Element der Ergebnismenge die Bedingung erfüllen. Dies wird durch das Schlüsselwort ANY erreicht. Die Verbindung des Hauptpfades mit dem relativen Pfad wird durch die Erweiterung zur korrelierten Unterabfrage erreicht.

Zu den beiden Fällen wird je ein Beispiel betrachtet.

$$//country[\ \underbrace{.//city}_{relativ\ zu\ country} = \underbrace{//city[name/text() = 'Córdoba']}_{nicht\ relativ\ zu\ country}]/name$$

XPath 5 Beispiel eines Elementvergleiches

Man will den Namen jener Länder wissen, welche eine Stadt namens Córdoba enthalten. Dazu wird der Hauptpfad, der relative und absolute Pfad konvertiert. Man erhält:

```
SELECT counam.*
FROM xml."country_name" AS counam
JOIN xml."country" AS cou
ON cou."car_code"=counam."countryID"
```

SQL 28 Der Hauptpfad des Beispiels XPath 5 in SQL

```
SELECT cit.*
FROM xml."city" AS cit
```

SQL 29 Der relative Pfad des Beispiels XPath 5 in SQL

```
SELECT cit.*
FROM xml."city" AS cit
WHERE cit."id" IN (
   SELECT citnam."cityID"
   FROM xml."city_name" AS citnam
   JOIN xml."city" AS cit ON cit."id"=citnam."cityID"
   WHERE citnam."VALUE"='Córdoba')
```

SQL 30 Der absolute Pfad des Beispiels XPath 5 in SQL

SQL 30 liefert alle Städte, deren Name Córdoba ist. *SQL* 29 liefert alle Städte der Datenbank. Diese Menge soll eingeschränkt werden auf die Menge von *SQL* 30. Dazu wird die SELECT-Klausel von *SQL* 30 verändert auf die Primärschlüssel der Städte. Die Primärschlüssel von *SQL* 29 müssen in der Menge von *SQL* 30 enthalten sein. Daher wird die WHERE-Klausel von *SQL* 29 um die IN-Klausel erweitert. Da der relative Pfad noch keine Verbindung zum Hauptpfad besitzt, wird dieser zur korrelierten Unterabfrage erweitert. Dadurch ist es möglich die EXISTS-Klausel anzuwenden. Zusammengesetzt sieht dies wie folgt aus:

```
SELECT counam.*
FROM xml."country_name" AS counam
JOIN xml."country" AS cou
ON cou."car_code"=counam."countryID"
WHERE EXISTS (
   SELECT cit."country"
   FROM xml."city" AS cit
   WHERE cit."country"=cou."car_code"
   AND cit."id" IN (
       SELECT cit."id"
       FROM xml."city" AS cit
       WHERE cit."id" IN (
           SELECT citnam."cityID"
           FROM xml."city_name" AS citnam
           JOIN xml."city" AS cit
           ON cit."id"=citnam."cityID"
           WHERE citnam."VALUE"='Córdoba')
       )
   )
```

SQL 31 Komplette SQL-Abfrage von Beispiel XPath 5

Das zweite Beispiel soll alle Städtenamen liefern, die mehr Einwohner als Hamburg besitzen:

$$//\text{city}[\underbrace{\text{population}}_{relativ\ zu\ \text{city}} > \underbrace{\max(//city[name = 'Hamburg']/population)}_{nicht\ relativ\ zu\ \text{city},\ \text{absolut}}]/name$$

XPath 6 Beispiel eines Wertevergleichs

Die einzelnen Teile lassen sich wie bekannt erzeugen.

```
SELECT citnam.*
FROM xml."city" AS cit
JOIN xml."city_name" AS citnam
ON citnam."cityID"=cit."id"
```

SQL 32 Hauptpfad des Beispiels XPath 6 in SQL

```
SELECT citpop.*
FROM xml."city_population" AS citpop
JOIN xml."city" AS cit
ON cit."id"=citpop."cityID"
```

SQL 33 Relativer Pfad des Beispiels XPath 6 in SQL

```
SELECT MAX(CAST(citpop."population" AS NUMERIC))
FROM xml."city_population" AS citpop
JOIN xml."city" AS cit ON cit."id"=citpop."cityID"
WHERE cit."id" IN (
    SELECT citnam."cityID"
    FROM xml."city_name" AS citnam
    JOIN xml."city" AS cit ON cit."id"=citnam."cityID"
    WHERE citnam."VALUE"='Hamburg')
```

SQL 34 Absoluter Pfad des Beispiels XPath 6 in SQL

SQL 34 liefert die maximale Bevölkerung von Hamburg. Dies dient als eine Seite der Bedingung für den relativen Pfad. Daher wird der WHERE-Klausel des relativen Pfades der Vergleich von Einwohner-Elementen mit der maximalen Bevölkerung von Hamburg hinzugefügt. Es besteht die Möglichkeit, dass die Ergebnismenge des absoluten Pfades mehr als ein Element enthält. Deshalb wird zusätzlich das Schlüsselwort ANY verwendet.

```
SELECT cit.*
FROM xml."city_population" AS citpop
JOIN xml."city" AS cit_inner
ON cit."id"=citpop."cityID"
WHERE CAST(citpop."population" AS NUMERIC) > ANY (
    SELECT MAX(CAST(citpop."population" AS NUMERIC))
    FROM xml."city_population" AS citpop
    JOIN xml."city" AS cit
    ON cit."id"=citpop."cityID"
    WHERE cit."id" IN (
        SELECT citnam."cityID"
        FROM xml."city_name" AS citnam
        JOIN xml."city" AS cit
        ON cit."id"=citnam."cityID"
        WHERE citnam."VALUE"='Hamburg')
)
```

SQL 35 Relativer und absoluter Pfad des Beispiels XPath 6 in SQL

Da noch der Bezug zum Hauptpfad fehlt, wird der relative Pfad zur korrelierten Unterabfrage erweitert. Damit erhält man alle Städte, die mehr Einwohner als Hamburg maximal besitzen.

```
SELECT citnam.*
FROM xml."city_name" AS citnam
JOIN xml."city" AS cit
ON cit."id"=citnam."cityID"
WHERE EXISTS (
    SELECT cit_inner."id"
    FROM xml."city_population" AS citpop
    JOIN xml."city" AS cit_inner
    ON cit_inner."id"=citpop."cityID"
    WHERE cit_inner."id"=cit."id"
    AND CAST(citpop."population" AS NUMERIC) > ANY (
        SELECT MAX(CAST(citpop."population" AS NUMERIC))
```

```
        FROM xml."city_population" AS citpop
        JOIN xml."city" AS cit
        ON cit."id"=citpop."cityID"
        WHERE cit."id" IN (
            SELECT citnam."cityID"
            FROM xml."city_name" AS citnam
            JOIN xml."city" AS cit
            ON cit."id"=citnam."cityID"
            WHERE citnam."VALUE"='Hamburg')
    )
)
```

SQL 36 Komplette Abfrage des Beispiels XPath 6 in SQL

Anmerkung: Es müssen die Alias-Namen der korrelierten Unterabfrage verändert werden, da diese das gleiche Kürzel wie in der Hauptabfrage besitzen.

4.6. Die id(.)-Funktion…

Die id(.)-Funktion ist anwendbar auf IDREF(S)-Attribute von Elementen die andere Elemente referenzieren und dereferenziert diese. Das heißt, es wird im Verlauf der Auswertung des Pfadausdruckes nicht mehr auf dem Elementtyp $\texttt{element}_i$, zu dem das Attribut gehört, operiert sondern stattdessen auf den Elementen, die durch das Attribut referenziert werden. Es wird im XML-Baum gesprungen.

4.6.1. … im Hauptpfad

Konvention: Die id(.)-Funktion sollte am Anfang des jeweiligen Pfades stehen. Dies vereinfacht die Umformung.

$$id(//abs\text{-}path/@idref\text{-}attrib)/path_{deref}$$

XPath 7 id(.)-Funktion im Hauptpfad

In SQL gibt es keine solche Funktionalität. Die DTD liefert keinerlei Informationen, wohin ein IDREF(S)-Attribut verweist. Die XML-Instanz bietet Daten, mit denen die Verweise gesucht werden können. Diese Arbeit nimmt der Datenbankkonverter ab. Der Konverter sucht diese Verweise und ergänzt die Abbildungsmetadaten damit. Daher werden die Abbildungsmetadaten nach allen Elementtypen durchsucht, die durch das IDREF(S)-Attribut referenziert werden. Mit den Ziel-Elementtypen der IDREF(S)-Attribute kann ein äquivalenter Pfadausdruck erzeugt werden.

$$//(\underbrace{elem_1|\ldots|elem_n}_{Ziel-Elementtypen})[\underbrace{@id}_{ID-Attribut} = \underbrace{//abs\text{-}path/@idref\text{-}attrib}_{IDREF(S)-Attribut}]/path_{deref}$$

Anmerkung: Bei IDREFS-Attributen funktioniert dies nicht, jedoch wird es beim Konverter nach demselben Prinzip gehandhabt. Das liegt daran, dass das gesamte IDREFS-Attribut als ein String betrachtet wird. Die id(.)-Funktion zerteilt diese in die einzelnen IDREF-Attribute. Mit der Funktion *fn:tokenize(@attrib, "\s")* ist es möglich, die IDREFS-Attribute ebenfalls auszuwerten. Diese zerteilt den String nach dem Muster.

Die Auswertung liefert alle Elemente vom Typ $element_1, \ldots, element_n$ dessen ID-Attribut gleich dem IDREF-Attribut ist. Anschließend wird für jedes dieser Elemente der verbleibende Pfad $path_{deref}$ ausgewertet.

Nach dem umformen können diese Ausdrücke übersetzt werden. Siehe dazu Kapitel *4.5*.

Am Ende von Kapitel *4.2* wurde die Existenz von Eigenschaften und Tabellen beschrieben. Damit lässt sich die zuvor beschriebene Umformung für den Konverter weiter vereinfachen.

$$//(\underbrace{elem_1|\ldots|elem_n}_{Ziel-Elementtypen})[\underbrace{./abs\text{-}path/@idref\text{-}attrib}_{IDREF(S)-Attribut}]/path_{deref}$$

Dieser Ausdruck ist nicht valide zur DTD. Der Konverter wird dies später jedoch so verarbeiten. Es wird geprüft, ob für die Ziel-Elementtypen eine Relation existiert.

4.6.2. … in Filtern

Hierbei ist die eine id(.)-Funktion relativ zum Kontextknoten während die andere absolut ist.

$$\ldots/\text{element}_i[\underbrace{id(abs\text{-}path_{dep})/\ldots}_{relativ\ zu\ \text{element}_i}\ operator\ \underbrace{id(abs\text{-}path_{indep})/\ldots}_{nicht\ relativ\ zu\ \text{element}_i}]/\ldots$$

Wie in Kapitel *4.5* beschrieben, können nur die zwei Fälle auftreten. Jeder Teil lässt sich separat übersetzen und muss nur noch miteinander in Verbindung gebracht werden. Durch das umformen der id(.)-Funktion entstehen zwei Ausdrücke wie im Kapitel zuvor beschrieben, jedoch als Unterabfragen. Für den relativen Ausdruck muss noch die Verbindung zum Kontextknoten hergestellt werden. Dies wird mit einer korrelierten Unterabfrage erreicht. Zusätzlich muss der Abfrage noch der Vergleich mit dem absoluten Ausdruck hinzugefügt werden. Da hier ebenfalls eine Ergebnismenge mit mehr als einem Element auftreten kann, wird das Schlüsselwort ANY an den Operator *operator* gehangen.

Als Beispiel für den Element-Vergleich dient folgender Ausdruck:

$$//\text{country}[\underbrace{id(@capital)}_{relativ\ zu\ country} = \underbrace{id(//organization/@headq)}_{nicht\ relativ\ zu\ country}]/name$$

XPath 8 Beispiel der id(.)-Funktion im Filter

Dieser liefert alle Ländernamen, dessen Hauptstädte zugleich Organisationssitze sind. Der Hauptpfad lässt sich nach Kapitel *4.1* konstruieren. Die id(.)-Funktionen werden nach Kapitel *4.6.1* umgebaut und dann konvertiert. Dies erzeugt folgende SQL-Abfragen:

```
SELECT counam.*
FROM xml."country_name" AS counam
JOIN xml."country" AS cou
ON cou."car_code"=counam."countryID"
```

SQL 37 Der Hauptpfad von XPath 8

```
SELECT cit.*
FROM xml."city" AS cit
WHERE EXISTS (
```

```
        SELECT cou."capital"
        FROM xml."country" AS cou
        WHERE cit."id"=cou."capital")
```

SQL 38 Der relative Pfad von XPath 8

```
SELECT cit.*
FROM xml."city" AS cit
WHERE EXISTS (
        SELECT cou."capital"
        FROM xml."country" AS cou
        WHERE cit."id" = cou."capital")
```

SQL 39 Der absolute Pfad von XPath 8

Der absolute Pfad wird der WHERE-Klausel des relativen Pfades hinzugefügt. Da hier wieder viele Elemente die Ergebnismenge des absoluten Pfades bilden können, wird an den Operator ANY angehangen. Der relative Pfad wird zur korrelierten Unterabfrage erweitert.

```
SELECT counam.*
FROM xml."country_name" AS counam
JOIN xml."country" AS cou
ON cou."car_code"=counam."countryID"
WHERE EXISTS (
    (SELECT cit."country"
     FROM xml."city" AS cit
     WHERE EXISTS (
        SELECT cou."capital"
        FROM xml."country" AS cou
        WHERE cit."id" = cou."capital")
     AND cit."id" = ANY (
           SELECT cit."id"
           FROM xml."city" AS cit
           WHERE EXISTS (
              SELECT org."headq"
              FROM xml."organization" AS org
              WHERE cit."id" = org."headq")
        )
     AND cou."car_code"=cit."country"
    )
)
```

SQL 40 Fertige Abfrage des Beispiels XPath 8 in SQL

Der Aufbau zum zweiten Fall unterscheidet sich nur in einem Punkt, dass bei Werten die Möglichkeit besteht, eine Typumwandlung durchzuführen. Die Element-Vergleiche werden reduziert auf Wert-Vergleiche der Primärschlüssel.

4.7. Sonstiges

4.7.1. Die text()-Funktion

Die text()-Funktion greift nur auf den Textinhalt des jeweiligen Elementes zu. In der Datenbank werden Textinhalte unter dem Namen des Vaterelements abgelegt. Daher braucht „text()“ nur durch den Namen des dazugehörigen Elementes ersetzt zu werden.

Ein Beispiel:

$$//country[language/text() = 'German']/name$$
$$\Leftrightarrow$$
$$//country[language/language = 'German']/name$$

XPath 9 Beispiel text()-Funktion ersetzen

4.7.2. Typumwandlung

Die DTD bietet keine Möglichkeit Datentypen anzugeben. Daher werden alle Werte als String abgespeichert. In XPath wird dies durch implizite Typumwandlung umgangen. Das heißt bei numerischen Vergleichen (>, <, ≥, ≤) wird automatisch eine Typumwandlung vorgenommen.

SQL hingegen kennt Datentypen. Beim Erzeugen einer Datenbank wird für jede Spalte der Datentyp festgelegt. Daher werden im Allgemeinen Typumwandlungen eher selten benötigt.

Beim Überführen der Datenbank aus der DTD in das relationale Datenbankmodell bleibt nur die Möglichkeit, alles als String abzuspeichern. Daher muss bei numerischen Vergleichen eine explizite Typumwandlung vorgenommen werden, da dies sonst zu falschen Ergebnissen führen könnte.

```
CAST(column AS NUMERIC)
```

SQL 41 Explizite Typumwandlung in eine Zahl

4.7.3. OR-Äquivalent

Das OR-Äquivalent ist eine andere Darstellung für den OR-Ausdruck von SQL.

```
SELECT cou.*
FROM xml."country_name" AS counam
JOIN xml."country" AS cou
ON cou."car_code"=counam."countryID"
WHERE cou."car_code" = ANY ('D','E','F')
⇔
SELECT cou.*
FROM xml."country_name" AS counam
JOIN xml."country" AS cou
ON cou."car_code"=counam."countryID"
WHERE cou."car_code"='D'
   OR cou."car_code "='E'
   OR cou."car_code "='F'
```

SQL 42 OR-Äquivalent bei SQL

Oder auch in XPath:

$$//country[@car\_code = ('D','E','F')]/name$$
$$\Leftrightarrow$$
$$//country[@car\_code = 'D' \text{ or } @car\_code = 'F' \text{ or } @car\_code = 'F']/name$$

XPath 10 OR-Äquivalent bei XPath

Hier wird jeweils die letztere Variante genutzt, da diese beim Saxon-Compiler sonst in weiteren Objekten verpackt werden. Damit ist die Auswertung und Übersetzung einfacher.

5. Implementierung

In diesem Kapitel werden die Implementierung des Konverters und dessen Schnittstellen beschrieben.

5.1. Allgemeines

Der Konverter wurde in der Programmiersprache Java geschrieben. Dies ist notwendig, da Saxon und auch der Datenbank-Konverter ebenfalls in Java geschrieben worden sind.

Die Initialisierung passiert automatisch. Benötigte Daten werden aus der Datenbank geladen. Diese muss jedoch vom Datenbank-Konverter erzeugt worden sein, da dieser die Tabellen für die benötigten Abbildungsmetadaten anlegt.

Alle Methoden stehen in einer Klasse. Die einzige öffentliche Methode ist parseNext(.). Dieser wird ein XPath-Ausdruck übergeben. Zurückgeliefert wird eine rekursiv erzeugte SQL Abfrage.

5.2. Saxons abstrakte Syntaxbäume

Die Grundlage auf die dieser Konverter basiert, sind abstrakte Syntaxbäume. Diese werden in diesem Kapitel erklärt und auch die Implementierung in Saxon wird erläutert. Am Ende wird der Nutzen für den Konverter besprochen.

5.2.1. Abstrakte Syntaxbäume (AST[16])

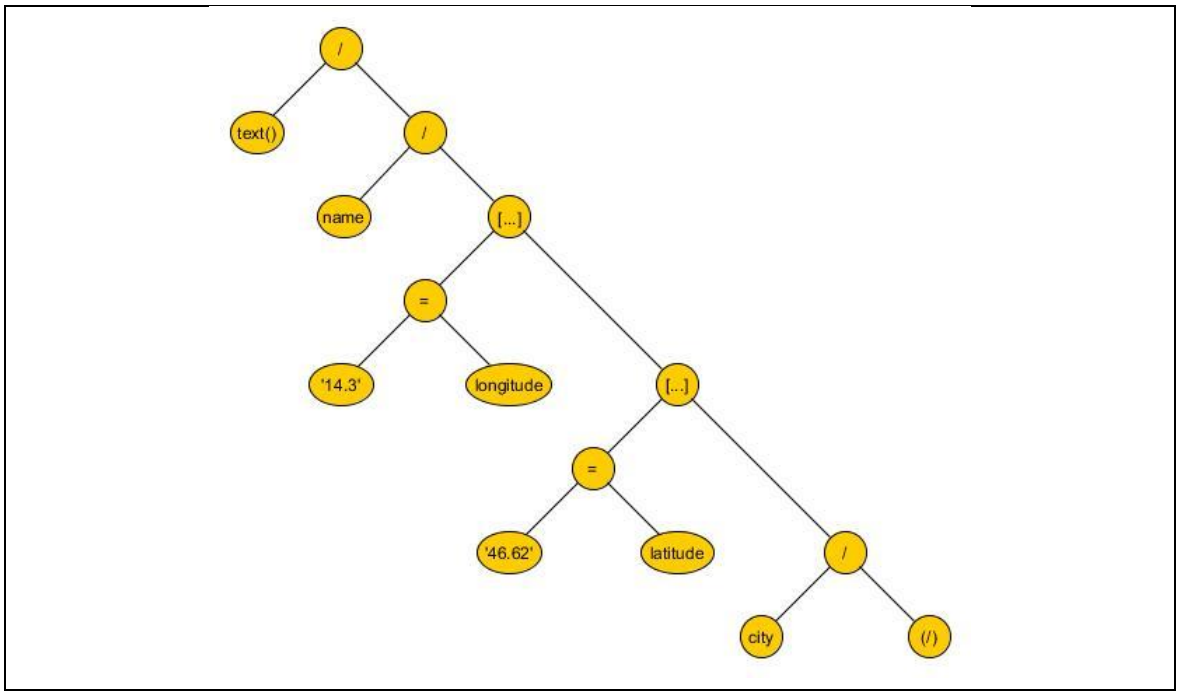
Ein AST ist die Darstellung eines Programmcodes, oder auch von Sprachen wie SQL, XPath, etc., als Baum. Dadurch erhält der gegebene Text eine strukturierte Darstellung, die durch eine Grammatik festgelegt ist. Im Gegensatz zu normalen Syntaxbäumen, werden im AST nicht die Produktionen der zugrundeliegenden Grammatik mit angegeben. Es werden nur die einzelnen Bestandteile des Codes/der Sprache abgebildet.

$$/city[latitude = '46.62'][longitude = '14.3']/name/text()$$

XPath 11 Beispiel eines Pfadausdruckes, nicht valide zur DTD

So lässt sich zum Beispiel der Ausdruck *XPath 11* darstellen als:

[16] Abstract Syntax Tree



AST 1Darstellung eines XPath-Ausdruckes, wie er durch Saxon erzeugt wird.

Abstrakte Syntax Bäume werden hauptsächlich in Compilern, Debuggern, Optimierern, und Validatoren verwendet.

Für weitere Informationen siehe [1], Kapitel 3.17.2.

5.2.2. Implementation in Saxon

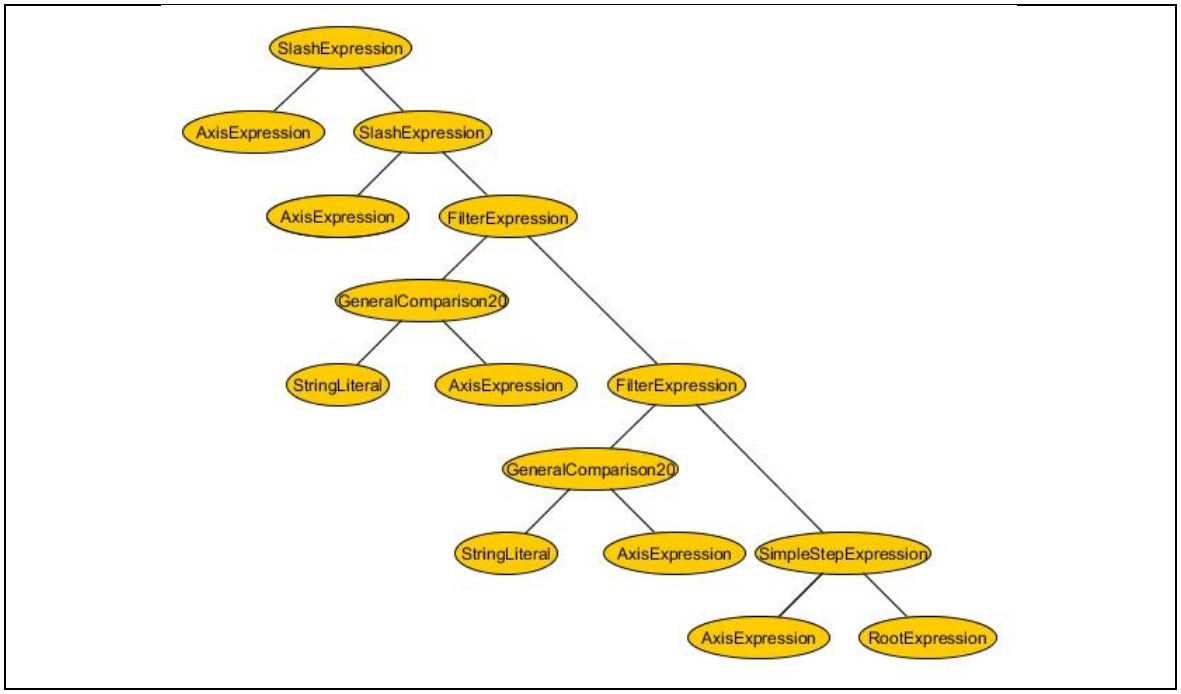
Der AST in Saxon wird erzeugt mit Instanzen der Expression-Klasse. Es gibt viele verschiedene Unterklassen der Klasse Expression. Die wichtigsten Klassen, die in dieser Arbeit verwendet werden, stehen in der folgenden Tabelle:

Klasse	Beschreibung
AndExpression	Stellt eine Konjunktion dar.
OrExpression	Stellt eine Disjunktion dar.
FilterExpression	Stellt einen Filter dar, dessen Kindknoten unter anderem GeneralComparsion20, And- oder OrExpression sein können.
AxisExpression	Stellt einen Elementtyp dar, wie city oder country.
SystemFunctionCall	Stellt einen Funktionsaufruf dar, der eine exists(.)-, not(.)- oder Aggregatfunktion enthält.
GeneralComparison20	Stellt einen Vergleich dar, der aus einem Vergleichsoperator und einer linken und rechten Seite besteht. Kann unter anderem verschachtelte And- und OrExpressions enthalten.

Operand	Stellt eine Beziehung der Vater- und dessen Kind-Expressions dar. Taucht nicht im AST auf.
SimpleStepExpression	Stellt den Beginn des Pfadausdruckes dar.
SlashExpression	Stellt einen Schritt dar.
(String-)Literal	Stellt einen Wert (String oder Zahl) dar.

Tabelle 1 Übersicht der wichtigsten Klassen aus Saxon

Das Beispiel *AST 1* durch Expression-Objekten dargestellt:



AST 2 Darstellung von AST 1 durch Expression-Objekte

Es fällt auf, dass der Wurzelknoten nicht die *RootExpression* ist sondern die *SlashExpression* mit der text()-Funktion. Das liegt daran, dass der AST in Saxon rückfällig Implementiert ist. Das bedeutet, es wird zuerst die maximale Tiefe des Baumes in einer Richtung gesucht und rückwärts bearbeitet. Der Pfadausdruck *XPath 11* wird rückwärts durchlaufen und die Schritte (das Slash „/“) und Filter (die Klammern „[…]“) in den Baum aufgenommen. Dadurch entsteht der ganz rechte Pfad des Baumes.

Ist der Beginn des Pfades erreicht, dann wird der Inhalt der einzelnen Teile des Pfades bearbeitet. Aus Elementtypen im Pfad wird eine *AxisExpression*. Diese werden als linkes Kind den *Slash*- bzw. *SimpleStepExpressions* hinzugefügt. Der Filterinhalt wird als Kind des jeweiligen Filters links angehängt. Nach jedem linken Schritt, wird direkt wieder die maximale rechte Tiefe gesucht. Bei *GeneralComparison20* steht der relative Pfad bei einem Vergleich immer rechts und das Literal immer links.

5.2.3. Der Nutzen für den Konverter

Der AST ist ein Hilfsmittel zum Übersetzen von XPath zu SQL.

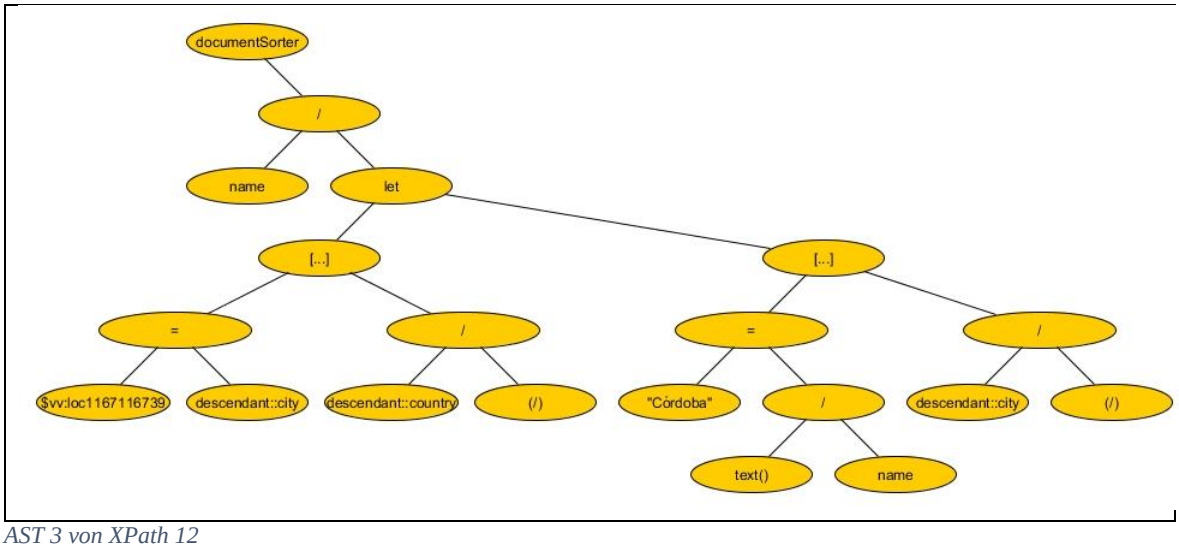
Der Hauptpfad entspricht den vom Wurzelknoten ausgehenden rechten Pfad. Dabei sind nur die Slash- und SimpleStepExpressions von Interesse, da diese die Tabellen darstellen. Die *FilterExpressions* werden der WHERE-Klausel hinzugefügt, wobei der rechte Pfad einer *FilterExpression* den relativen Pfad darstellt und der linke Pfad ein Literal. Daraus lässt sich die Unterabfrage mit der Bedingung erzeugen. Der Filter kann dabei weitere And- oder OrExpression- oder SystemFunctionCall-Objekte enthalten.

Problematisch sind Filter mit absoluten Pfadausdrücken. Der Saxon-XPathCompiler erzeugt LetExpression-Objekte für absolute Pfade in Filtern.

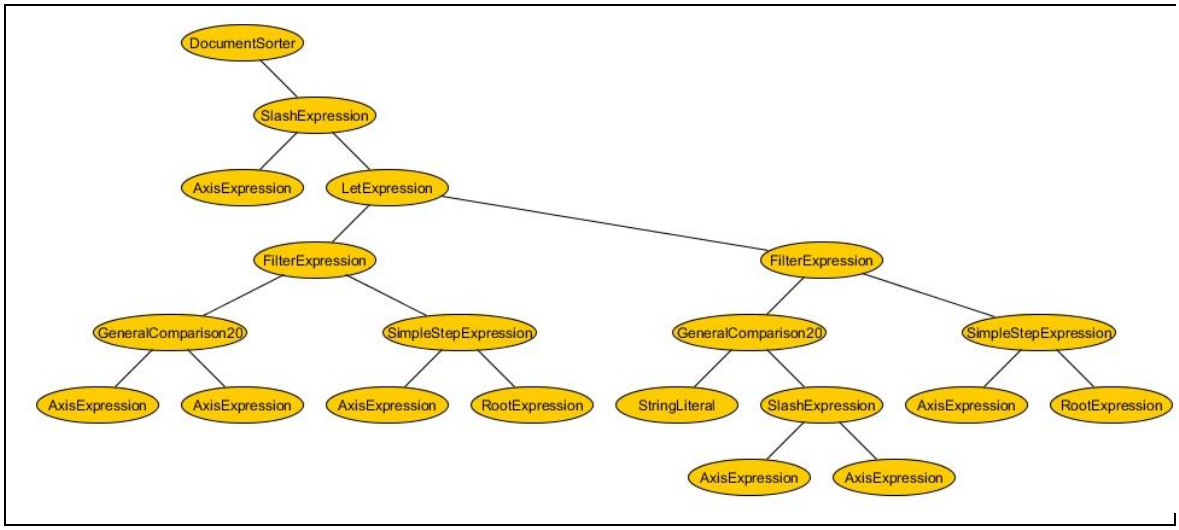
$$//country[.//city = //city[name/text() = 'Córdoba']]/name$$

XPath 12 Beispiel einer LetExpression-Umformung

Der erzeugte AST:



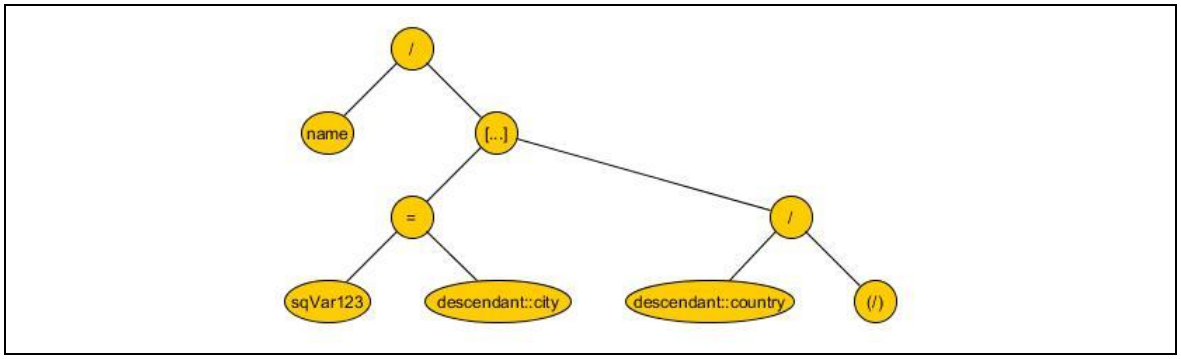
AST 3 von XPath 12



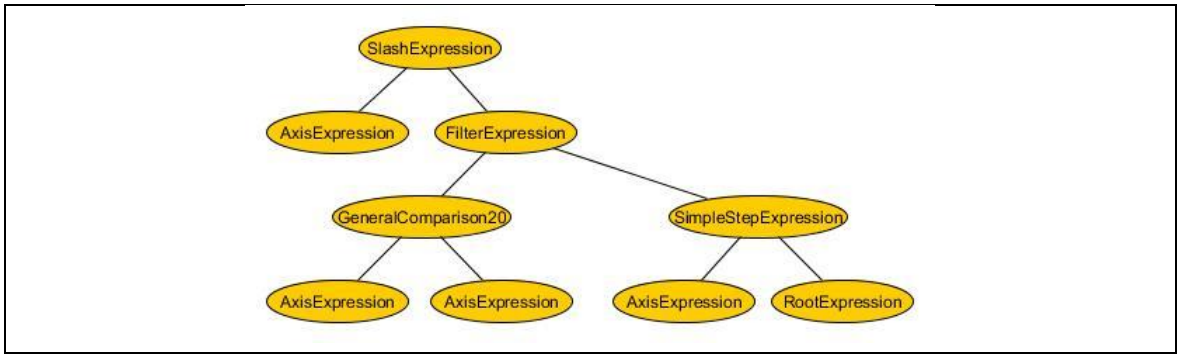
AST 4 von XPath 12, Expression-Ansicht

Da let-Ausdrücke in dieser Arbeit nicht behandelt werden, muss die Erzeugung von LetExpression-Objekten umgangen werden. String-Literale oder relative Pfade kommen direkt in den AST. Wird der absolute Pfad durch einen relativen Pfad ersetzt, dann wird dieser zur korrelierten Unterabfrage. Der absolute Pfad als String wird als StringLiteral behandelt. Beides ist ungeeignet. Durch Benutzung einer künstlichen Variable und der Feststellung, dass relative Pfade ohne LetExpression-Objekte auskommen, bietet dies eine gute Möglichkeit das Problem zu umgehen.

Ohne Let-Ausdruck sieht der Baum wie folgt aus:



AST 5 von XPath 12, ohne let-Ausdruck



AST 6 von XPath 12, ohne let-Ausdruck, Expression-Ansicht

Zur Umsetzung werden zwei Datenstrukturen benötigt, die die künstlichen Variablen und den absoluten Pfadausdruck zwischenspeichern. Zum einen wird eine Liste benötigt, worin alle bekannten Variablen stehen und eine Map, die jeder Variablen den absoluten Pfadausdruck zuordnet. Beim Auswerten wird geprüft, ob eine oder zwei Variablen in der Bedingung enthalten sind und dementsprechend werden diese separat verarbeitet und zu einer Bedingung zusammengesetzt.

Alle künstlichen Variablen beginnen mit dem Kürzel „sqVar“. Um den Variablen einen eindeutigen Namen zu verpassen, wird an das Kürzel „sqVar“ eine eindeutige Nummer gehängt. Diese kann zum Beispiel durch die Java-Klasse *java.util.UUID* und der *hashCode()*-Methode erzeugt werden.

Anmerkung: Zwei Variablen sind möglich, falls Aggregatfunktionen oder die id(.)-Funktion auf beiden Seiten einer Bedingung vorkommen. Dadurch ist es einfacher, solche Ausdrücke zu übersetzen.

5.3. Interne Methoden

Der Konverter benötigt keinen Konstruktor. Da es sich um eine statische Klasse handelt, wird keine Instanz dieser Klasse benötigt. Dementsprechend ist der Konstruktor privat, da es keinen Sinn macht, ein solches Objekt zu erzeugen.

Es gibt viele parse-Methoden. Jede hat ihre eigene Aufgabe. Die wichtigsten werden im Folgenden aufgelistet:

Methode	Funktion
parseNext(.)	Setzt interne Variablen zurück und ruft parse(.) auf.
parse(.)	Bereitet den gegebenen XPath-Ausdruck vor. Siehe dazu Kapitel *5.4*. Zerlegt den vorbereiteten Pfadausdruck in die einzelnen absoluten Pfade der Vereinigung und übersetzt diese. Jeder Pfad wird noch einmal auf nicht unterstützte Teilausdrücke hin untersucht. Ruft parse(.,.) oder parseSFC(.,.) auf.
parse(.,.)	Erzeugt den Hauptpfad in SQL. Ruft parseFilter(.) auf und hängt gegebenenfalls die GROUP-BY-Klausel an.
parseSFC(.,.)	Konvertiert SystemFunctionCall-Objekte (SFC) von Saxon. Dazu zählen exists(.), not(.), distinct-values(.) und die Aggregatfunktionen.
parseFilter(.)	Zerlegt einen Filter in seinen Filteranteil und dem dazugehörigen Kontextknoten. Ruft parseSide(.,.) zur Weiterverarbeitung auf.
parseSide(.,.)	Ruft je nach Art des Expression-Objektes eine bestimmte Methode auf. Es gibt And-, Or-, GeneralComparison20- und SFC-Objekte.
parseAndOr(.,.)	Erzeugt eine Konjunktion oder Disjunktion. Ruft parseSide(.,.) auf.
parseGC(.,.)	Liefert die transformierten Bedingungen in SQL zurück.

Tabelle 2 Übersicht der wichtigsten parse-Methoden

5.4. Die preprocessor(.)-Methode

Diese Methode bearbeitet einen XPath-Ausdruck, um diesen in eine Art Normalform zu bringen. Dazu zählen:

1) OR-äquivalente Ausdrücke umformen,
2) ersetzen der text()-Funktionen,
3) id(.)-Funktionen umformen,
4) Vereinigungen umformen,
5) absolute Pfadausdrücke und Pfade, die Funktionen enthalten, werden durch künstliche Variablen ersetzt.

5.5. Probleme und Lösungen

5.5.1. Saxon

Saxon als ein separates Projekt in Eclipse[17] angelegt, lieferte aus unbekannten Gründen Fehler. Daher liegt der Quellcode von Saxon im selben Projekt, wie der Datenbankkonverter und der XPath-Konverter.

5.5.2. Die Datenbank

Im Verlauf der Arbeit ist aufgefallen, dass die *river*-Tabelle zum Teil falsche Einträge besitzt. Nicht allen Flüssen werden die dazugehörigen *source*- und *estuary*-Elemente korrekt zugewiesen.

5.5.3. Das Ersetzen durch künstliche Variablen

Eine große Herausforderung war es, eine korrekte Ersetzung durch künstliche Variablen vorzunehmen. Es muss jeder Filter, Sub-Filter, Sub-Sub-Filter, … überprüft werden. Gegebenenfalls musste noch der Vater-Elementtyp hinzugezogen werden, falls der erste Elementtyp einer Variablen keine eigene Tabelle besitzt, sprich eine Eigenschaft ist.

Das Suchen eines Filters ist simpel. Dazu muss das Muster „[…]“ gesucht werden. Ein Problem hierbei ist, dass ein Regulärer Ausdruck nur auf Zeichen überprüft und keine Verschachtelung berücksichtigt. So kann ein Regulärer Ausdruck über einen Filter hinausgehen und beim darauffolgenden Filter enden.

$$\ldots/element_i[\overbrace{filter_i}^{korrekt}]/\ \underbrace{\ldots/element_j[\overbrace{filter_j}^{korrekt}]/\ \ldots}_{falsch}$$

Um das Problem zu umgehen, muss eine Methode entwickelt werden, die die Klammerung berücksichtigt. Dazu werden der Pfadausdruck, eine Start-Flag und die Art der Klammerung, die berücksichtigt werden soll, benötigt. Weiterhin ist ein Offset nötig, um denselben Pfadausdruck ab Index *x* durchlaufen zu können.

[17] Eine integrierte Entwicklungsumgebung (IDE) für Java

```
private static String getRegion(String xPathExpr, String startFlag,
            char add, char sub, int offset) {

    //Wenn keine entsprechende Klammer vorhanden ist,
    //gib Pfadausdruck zurück
    if(!xPathExpr.substring(offset).contains(startFlag))
        return xPathExpr;

    //Füge der Start-Flag ggf. Klammer hinzu
    if(!startFlag.endsWith("" + add))
        startFlag += add;

    //Klammer-Zähler mit eins initialisieren
    int counter = 1;

    //Index der Start-Flag ab dem offset
    int begin = xPathExpr.indexOf(startFlag, offset);
    int end = begin + startFlag.length();

    //Inkrementiere End-Index solange, bis Klammer-Zähler 0 ist
    while(counter > 0) {
        if(xPathExpr.charAt(end) == add)
            counter++;
        if(xPathExpr.charAt(end) == sub)
            counter--;
        end++;
    }
    //Gib den gefundenen Bereich zurück, ohne die Klammern
    return xPathExpr.substring(begin + startFlag.length(), end - 1);
}
```

Java Code 1 Methode zum Bestimmen eines Bereichs zwischen zwei Klammern.

Damit lässt sich jeder Filter suchen. Da jeder Filter Sub-Filter, jeder Sub-Filter Sub-Sub-Filter,… enthalten kann, wird jeder (Sub-)Filter auf (Sub-)Sub-Filter überprüft. Dies lässt sich rekursiv lösen. Um die Ersetzungen durch Variablen beizubehalten wird in jedem Schritt die gefundene Region des Filters durch den bearbeiteten Sub-Filter ersetzt.

$$path_i[\underbrace{\quad}_{sub-path_{i_j}[}\ \underbrace{filter_i}_{sub-filter_{i_j}}\ \underbrace{\quad}_{]/\dots}\ ]/\dots$$

$$\underbrace{\quad}_{sub-sub-path_{i_{j_k}}[}\ \underbrace{sub-sub-filter_{i_{j_k}}}_{\dots}\ \underbrace{\quad}_{]/\dots}$$

Dies wird rückfällig implementiert. Von jedem Filter wird erst der *n*-te Sub-Filter gesucht und bearbeitet. Dann wird der *n-1*-te Sub-Filter bearbeitet usw. So bleiben alle Änderungen der nachfolgenden Sub-Filter, im aktuellen Filter erhalten.

Nun werden die einzelnen Bedingungen der Filter betrachtet. Da diese Konjunktionen oder Disjunktionen enthalten können, wird der Filterinhalt an diesen zerteilt. Für jede einzelne Bedingung muss dann geprüft werden, ob der linke und/oder rechte Teil ersetzt werden muss. Um die Einzelteile der Bedingung zu erhalten, muss am Operator geteilt werden.

$$\dots\left[\underbrace{\dots/element_i[\dots]/\dots/element_j[\dots]}_{left}\ operator\ \underbrace{\{path\ ,value\}}_{right}\right]\dots$$

Die rechte Seite nimmt dabei nur Werte oder absolute Pfadausdrücke an. Problematisch ist die linke Seite. Sobald dort ein Filter mit einer Bedingung steht, wird an dieser geteilt und es liefert eine falsche Zerlegung der Bedingung. Indem von der Bedingung temporär alle Filter entfernt werden, bleiben nur noch der reine linke Pfad, der Vergleichsoperator und die rechte Seite stehen. Daraus wird $/element_j$ und $operator$ extrahiert und durch einen Regulären Ausdruck dieser Teile lässt sich die korrekte Zerteilung der Bedingung gewinnen. Abschließend werden die linke und rechte Seite geprüft, ob diese durch Variablen ersetzt werden müssen.

5.6. Weitere Beispiele

5.6.1. Beispiel 1

Gesucht sind die Namen jener Länder, die mehr Mitgliedschaften in Organisationen haben als Deutschland.

$$//country[\underbrace{count\big(id(@memberships)\big)}_{relativ} > \underbrace{count(id(//country[@car\_code = 'D']/@memberships))}_{absolut}]/name/text()$$

XPath 13 Beispiel 1

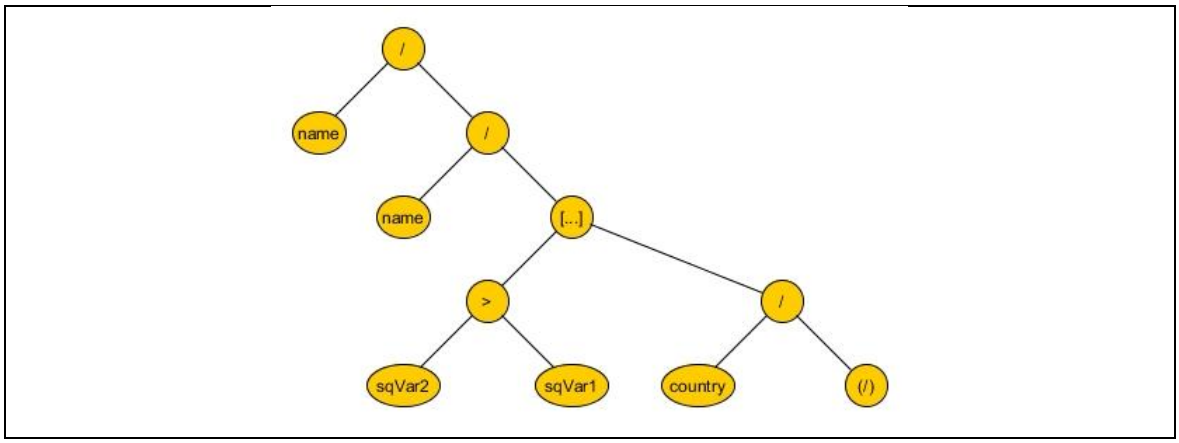
$$/country[\underbrace{sqVar1}_{relativ} > \underbrace{sqVar2}_{absolut}]/name/name$$

$$sqVar1 := count(/organization[country/@memberships])$$
$$sqVar2 := count(/organization[country[@car\_code = 'D']/@memberships])$$

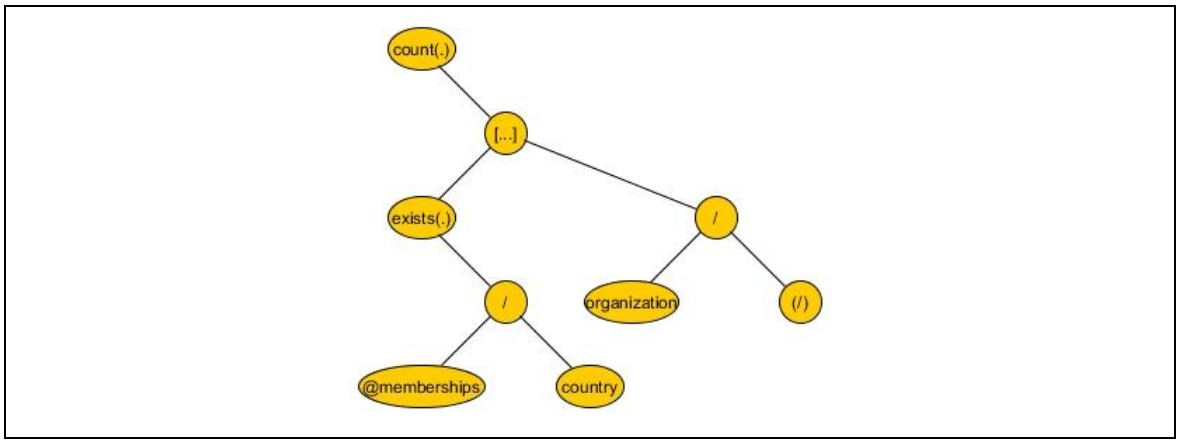
XPath 14 XPath 13 nach Bearbeitung durch die preprocessor(.)-Methode.

Anmerkung zu *XPath 14*: Es wird zuerst nur der Teilausdruck durch eine Variable ersetzt, ohne den Ausdruck zu verändern. Beim Auswerten einer Variablen wird im Verlauf der Auswertung die preprocessor(.)-Methode aufgerufen und die Variablen sehen danach wie zuvor definiert aus.

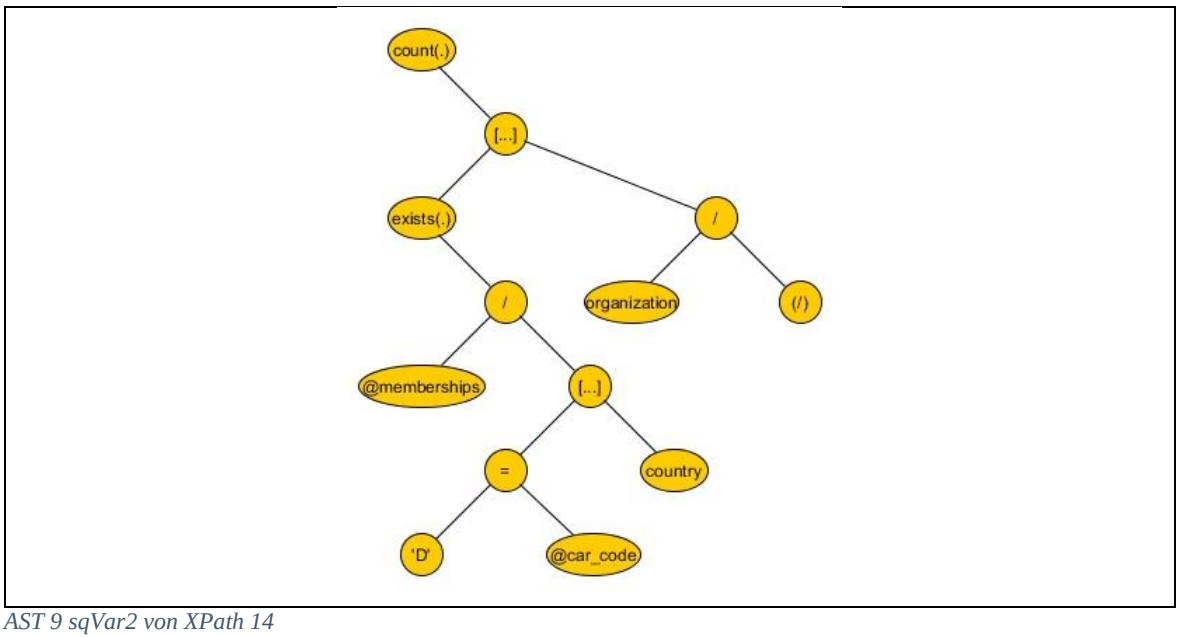
Dem XPath-Compiler von Saxon werden diese Pfadausdrücke übergeben. Dieser liefert unter anderem einen AST der jeweiligen Pfadausdrücke.



AST 7 von XPath 14



AST 8 sqVar1 von XPath 14



AST 9 sqVar2 von XPath 14

Begonnen wird mit *AST 7*. Daraus wird die Hauptabfrage erzeugt. An diese wird der Filter angehängt. Dabei wird geprüft, ob es sich um Variablen handelt. Es wird der relative und absolute Pfad aus dem Inhalt der Variablen bestimmt und diese werden dann einzeln übersetzt. Dabei wird die preprocessor(.)-Methode aufgerufen. Aus dem relativen Pfad wird eine korrelierte Unterabfrage generiert, da eine Aggregatfunktion verwendet wird und der absolute Pfad wird als Teil der

WHERE/HAVING-Klausel der korrelierten Unterabfrage hinzugefügt. Die korrelierte Unterabfrage wird dann mittels EXISTS der WHERE-Klausel der Hauptabfrage hinzugefügt.

```
SELECT COUNT(org.*)
FROM xml."organization" AS org
WHERE EXISTS (
   SELECT coumem.*
   FROM xml."country_memberships" AS coumem
   JOIN xml."country" AS cou
   ON cou."car_code"=coumem."countryID"
   WHERE cou."car_code"='D'
   AND org."id" = coumem."membershipsID")
```

SQL 43 Übersetzung von AST 9, der absolute Pfad.

```
SELECT COUNT(org.*)
FROM xml."organization" AS org
WHERE EXISTS (
   SELECT coumem.*
   FROM xml."country_memberships" AS coumem
   JOIN xml."country" AS cou_inner
   ON cou_inner."car_code"=coumem."countryID"
   WHERE org."id" = coumem."membershipsID"
   AND coumem."countryID"=cou."car_code")
HAVING CAST(COUNT(org.*) AS NUMERIC) > ANY (
   SELECT COUNT(org.*)
   FROM xml."organization" AS org
   WHERE EXISTS (
       SELECT coumem.*
       FROM xml."country_memberships" AS coumem
       JOIN xml."country" AS cou
       ON cou."car_code"=coumem."countryID"
       WHERE cou."car_code"='D'
       AND org."id" = coumem."membershipsID")
)
```

SQL 44 Übersetzung von AST 8 mit AST 9 in der HAVING-Klausel, der relative Pfad.

```
SELECT counam."VALUE"
FROM xml."country_name" AS counam
JOIN xml."country" AS cou
ON cou."car_code"=counam."countryID"
WHERE EXISTS (
   SELECT COUNT(org.*)
   FROM xml."organization" AS org
   WHERE EXISTS (
       SELECT coumem.*
       FROM xml."country_memberships" AS coumem
       JOIN xml."country" AS cou_inner
       ON cou_inner."car_code"=coumem."countryID"
       WHERE org."id" = coumem."membershipsID"
       AND coumem."countryID"=cou."car_code")
   HAVING CAST(COUNT(org.*) AS NUMERIC) > ANY (
       SELECT COUNT(org.*)
       FROM xml."organization" AS org
       WHERE EXISTS (
          SELECT coumem.*
          FROM xml."country_memberships" AS coumem
```

```
            JOIN xml."country" AS cou
            ON cou."car_code"=coumem."countryID"
            WHERE cou."car_code"='D'
            AND org."id" = coumem."membershipsID")
    )
)
```

SQL 45 Übersetzung von AST 7 mit AST 8 in der WHERE-Klausel, der Hauptpfad mit seinem Filter.

5.6.2. Beispiel 2

Gesucht sind alle Hauptstädte in Europa, in denen sich zugleich Organisationshauptsitze befinden und östlich von Berlin liegen.

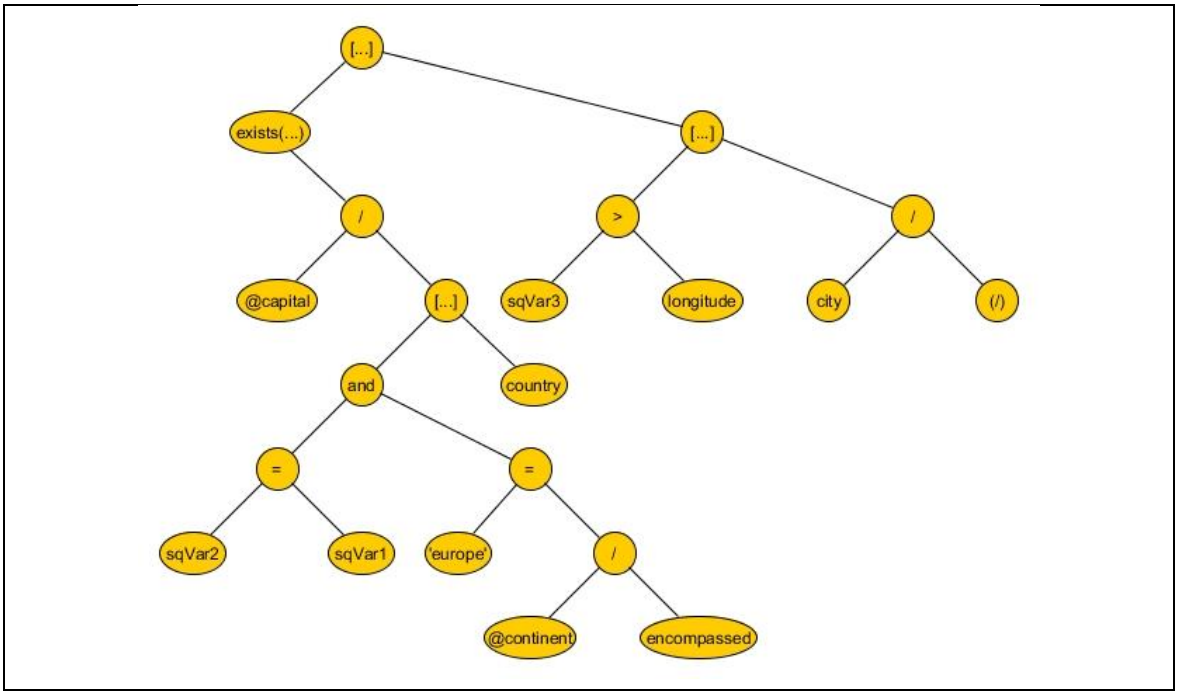
$$id(//country[\underbrace{encompassed/@continent}_{relativ} = 'europe' \; and \; \underbrace{id(@capital)}_{relativ} = \underbrace{id(//organization/@headq)}_{absolut}]/@capital)$$
$$[\underbrace{longitude}_{relativ} > \underbrace{//city[name = 'Berlin']/longitude}_{absolut}]$$

XPath 15 Beispiel 2

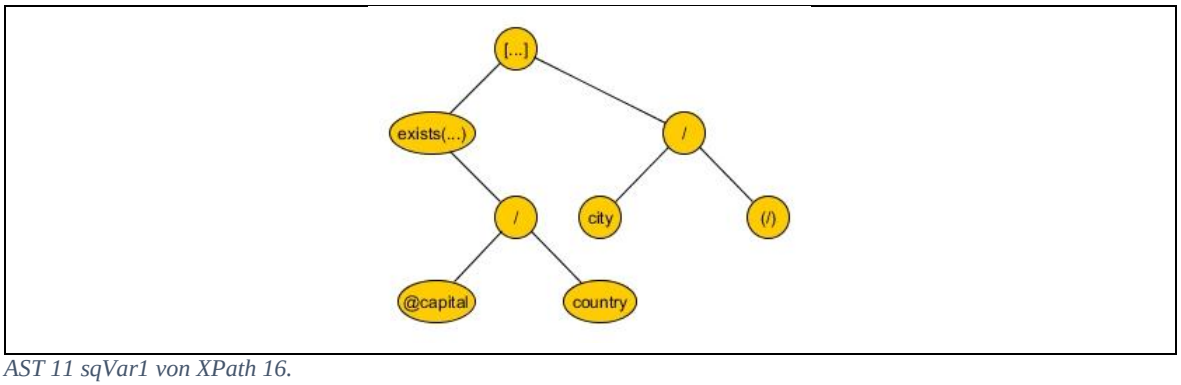
$$/city[./country[encompassed/@continent =' europe' and \underbrace{sqVar1}_{relativ} = \underbrace{sqVar2}_{absolut}]/@capital]$$
$$[longitude > \underbrace{sqVar3}_{absolut}]$$

$$sqVar1 := /city[./country@capital]$$
$$sqVar2 := /city[./organization/@headq]$$
$$sqVar3 := //city[name = 'Berlin']/longitude$$

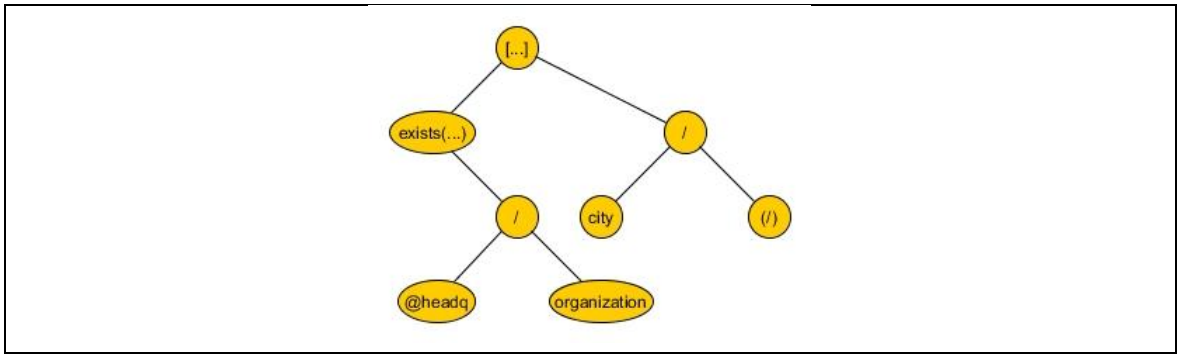
XPath 16 XPath 15 nach Bearbeitung durch die preprocessor(.)-Methode.



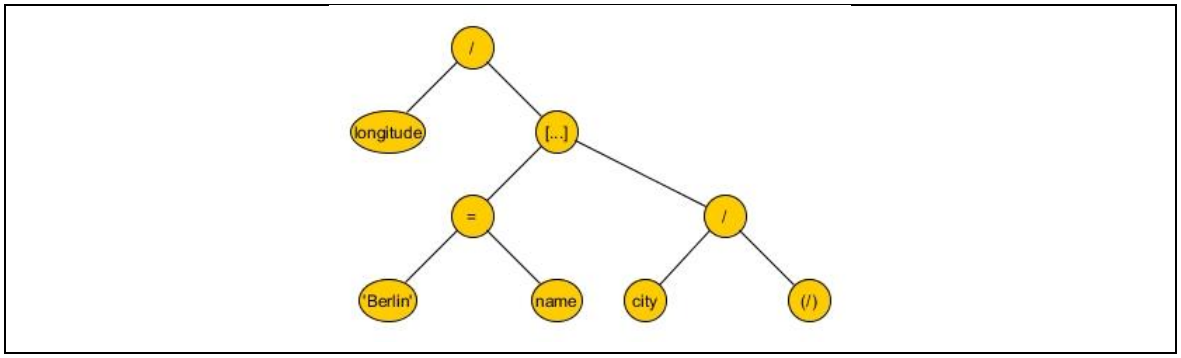
AST 10 von XPath 16.



AST 11 sqVar1 von XPath 16.



AST 12 sqVar2 von XPath 16.



AST 13 sqVar3 von XPath 16.

Nach Generierung der Hauptabfrage aus dem Hauptpfad des *AST 10* wird der exists(…)-Pfad bearbeitet. Diese korrelierte Unterabfrage besitzt drei Teile. Zwei Variablen und die Bedingung, dass die Länder in Europa liegen müssen.

```
SELECT couenc."countryID"
FROM xml."country_encompassed" AS couenc
JOIN xml."country" AS cou
ON cou."car_code"=couenc."countryID"
WHERE couenc."continent"='europe'
```

SQL 46 Liefert alle europäischen Länder.

```
SELECT cit.*
FROM xml."city" AS cit
WHERE EXISTS (
   SELECT cou."capital"
   FROM xml."country" AS cou
```

SQL 47 sqVar1 von XPath 15 liefert alle Hauptstädte.

```
SELECT cit."id"
```

```
FROM xml."city" AS cit
WHERE EXISTS (
   SELECT org."headq"
   FROM xml."organization" AS org
   WHERE cit."id" = org."headq")
```

SQL 48 sqVar2 von XPath 15 liefert alle Städte in denen Organisationssitze liegen.

Dazu wird die Hauptabfrage des exists(…)-Knotens erzeugt und die Bedingungen in dessen WHERE-Klausel geschrieben. *sqVar1* wird zur korrelierten Unterabfrage erweitert und erhält *sqVar2* als Bedingung. Hinzu kommt die Unterabfrage *SQL 46*.

```
SELECT cou."capital"
   FROM xml."country" AS cou
   WHERE EXISTS (
       SELECT cit.*
       FROM xml."city" AS cit
       WHERE EXISTS (
          SELECT cou."capital"
          FROM xml."country" AS cou
          WHERE cit."id" = cou."capital")
       AND cit."id" = ANY (
          SELECT cit."id"
          FROM xml."city" AS cit
          WHERE EXISTS (
             SELECT org."headq"
             FROM xml."organization" AS org
             WHERE cit."id" = org."headq")
       )
       AND cou."car_code"=cit."country"
   )
   AND cou."car_code" IN (
       SELECT couenc."countryID"
       FROM xml."country_encompassed" AS couenc
       JOIN xml."country" AS cou
       ON cou."car_code"=couenc."countryID"
       WHERE couenc."continent"='europe'
       AND cit."id" = cou."capital")
```

SQL 49 Die Bedingung der umgeformten id(.)-Funktion von AST 10 in SQL.

Der letzte Teil ist die Bedingung, dass die Städte östlich von Berlin liegen sollen.

```
SELECT CAST(cit."longitude" AS NUMERIC)
FROM xml."city" AS cit
WHERE cit."id" IN (
   SELECT citnam."cityID"
   FROM xml."city_name" AS citnam
   JOIN xml."city" AS cit
   ON cit."id"=citnam."cityID"
   WHERE citnam."VALUE"='Berlin')
```

SQL 50 sqVar3 von XPath 15 den Längengrad von Berlin.

Abschließend wird der WHERE-Klausel der Hauptabfrage *SQL 49* und *SQL 50* hinzugefügt. Dazu muss *SQL 49* zur korrelierten Unterabfrage erweitert werden, da es sich um die exists(…)-Funktion handelt.

```
SELECT cit.*
FROM xml."city" AS cit
WHERE EXISTS (
   SELECT cou."capital"
   FROM xml."country" AS cou
   WHERE EXISTS (
       SELECT cit.*
       FROM xml."city" AS cit
       WHERE EXISTS (
          SELECT cou."capital"
          FROM xml."country" AS cou
          WHERE cit."id" = cou."capital")
       AND cit."id" = ANY (
          SELECT cit."id"
          FROM xml."city" AS cit
          WHERE EXISTS (
             SELECT org."headq"
             FROM xml."organization" AS org
             WHERE cit."id" = org."headq")
       )
       AND cou."car_code"=cit."country"
   )
   AND cou."car_code" IN (
       SELECT couenc."countryID"
       FROM xml."country_encompassed" AS couenc
       JOIN xml."country" AS cou
       ON cou."car_code"=couenc."countryID"
       WHERE couenc."continent"='europe'
       AND cit."id" = cou."capital")
)
AND CAST(cit."longitude" AS NUMERIC) > ANY (
   SELECT CAST(cit."longitude" AS NUMERIC)
   FROM xml."city" AS cit
   WHERE cit."id" IN (
       SELECT citnam."cityID"
       FROM xml."city_name" AS citnam
       JOIN xml."city" AS cit
       ON cit."id"=citnam."cityID"
       WHERE citnam."VALUE"='Berlin')
)
```

SQL 51 Fertige Übersetzung von XPath 15.

6. Zusammenfassung

Der Konverter, welcher in dieser Arbeit entwickelt wurde, übersetzt XPath-Ausdrücke zu SQL-Abfragen. Es wurde auf der Grundlage der MONDIAL-Datenbank in deren XML-Form erarbeitet. Implementiert wurde das Programm in Java.

Die Java-Klasse des Konverters basiert auf den Überlegungen aus Kapitel 4 und Kapitel 5. Es unterstützt nicht die gesamte Funktionalität von XPath, deckt jedoch absolute und relative Pfadausdrücke, die Aggregatfunktionen sowie die Dereferenzierungsfunktion und Vereinigungen ab.

Ausgeführt werden die Abfragen über der MONDIAL-Datenbank, abgespeichert als relationales Modell. Diese Datenbank wurde aus der DTD-Datei von MONDIAL erzeugt und enthält alle Daten der dazugehörigen XML-Datei.

Das Programm ist nicht vollständig und befindet sich weiterhin in Entwicklung. Durch Erweiterung der vorhandenen Methoden lassen sich weitere Funktionalitäten wie some und every ergänzen.

7. Anhang

7.1. Die MONDIAL-DTD

```
<!-- XML DTD "mondial.dtd":
     (Wolfgang May, may@informatik.uni-freiburg.de, March 2000,      revised April 2009)
     a hierarchical DTD for the MONDIAL database,containing e.g.,
     - scalar reference attributes (city/capital)
     - multivalued reference attributes (organization/member/country)
     - cross-references in both directions (organization/member/country, country/memberships)
     - a "boolean"/flag attribute: city/is_country_cap
     - reference attributes with more than one target class
     (river/to, references rivers, lakes, and seas) -->

<!ELEMENT mondial (country*,continent*,organization*,
          sea*,river*,lake*,island*,mountain*,desert*,airport*)>

<!ELEMENT country (name+,localname?,population+,
          population_growth?,infant_mortality?,
          gdp_total?,gdp_agri?,gdp_ind?,gdp_serv?,inflation?,unemployment?,
          (indep_date|dependent)?,government?,encompassed*,
          ethnicgroup*,religion*,language*,border*,
          (province+|city+))>
<!ATTLIST country
          car_code ID #IMPLIED
          area CDATA #IMPLIED
          capital IDREF #IMPLIED
          memberships IDREFS #IMPLIED>

<!ELEMENT name (#PCDATA)>
<!ELEMENT localname (#PCDATA)>
<!ELEMENT area (#PCDATA)>
<!ELEMENT population (#PCDATA)>
<!-- note that population is also a subelement of city -->
<!ATTLIST population
          year CDATA #IMPLIED
          measured CDATA #IMPLIED>

<!ELEMENT population_growth (#PCDATA)>
<!ELEMENT infant_mortality (#PCDATA)>
<!ELEMENT gdp_total (#PCDATA)>
<!ELEMENT gdp_ind (#PCDATA)>
<!ELEMENT gdp_agri (#PCDATA)>
<!ELEMENT gdp_serv (#PCDATA)>
<!ELEMENT inflation (#PCDATA)>
<!ELEMENT unemployment (#PCDATA)>
<!ELEMENT indep_date (#PCDATA)>
<!ATTLIST indep_date
          from CDATA #IMPLIED>
          <!-- usually idref to a country, but not always (e.g. "Ottoman Empire") -->
<!ELEMENT dependent EMPTY>
<!ATTLIST dependent
```

```
        country IDREF #REQUIRED>
<!ELEMENT government (#PCDATA)>
<!ELEMENT encompassed EMPTY>
<!ATTLIST encompassed
        continent IDREF #REQUIRED
        percentage CDATA #REQUIRED>
<!ELEMENT ethnicgroup (#PCDATA)>
<!ATTLIST ethnicgroup
        percentage CDATA #REQUIRED>
<!ELEMENT religion (#PCDATA)>
<!ATTLIST religion
        percentage CDATA #REQUIRED>
<!ELEMENT language (#PCDATA)>
<!ATTLIST language
        percentage CDATA #REQUIRED>
<!ELEMENT border EMPTY>
<!ATTLIST border
        country IDREF #REQUIRED
        length CDATA #REQUIRED>

<!ELEMENT province (name+,localname?,area?,population*,city*)>
<!ATTLIST province
        id ID #REQUIRED
        country IDREF #REQUIRED
        capital IDREF #IMPLIED>

<!ELEMENT city (name+,localname?,latitude?,longitude?,elevation?,population*,
        located_at*,located_on*)>
<!ATTLIST city
        id ID #REQUIRED
        is_country_cap CDATA #IMPLIED
        is_state_cap CDATA #IMPLIED
        country IDREF #REQUIRED
        province IDREF #IMPLIED>
<!ELEMENT elevation (#PCDATA)>
<!ELEMENT longitude (#PCDATA)>
<!ELEMENT latitude (#PCDATA)>
<!ELEMENT located_at EMPTY>
<!ATTLIST located_at
        watertype (river|sea|lake) #REQUIRED
        river IDREFS #IMPLIED
        sea IDREFS #IMPLIED
        lake IDREFS #IMPLIED>
<!ELEMENT located_on EMPTY>
<!ATTLIST located_on
        island IDREF #REQUIRED>

<!ELEMENT organization (name,abbrev,established?,members*)>
<!ATTLIST organization
        id ID #REQUIRED
        headq IDREF #IMPLIED>
<!ELEMENT abbrev (#PCDATA)>
<!ELEMENT established (#PCDATA)>
<!ELEMENT members EMPTY>
<!ATTLIST members
```

```
        type CDATA #REQUIRED
        country IDREFS #REQUIRED>

<!ELEMENT continent (name,area)>
<!ATTLIST continent
        id ID #REQUIRED>

<!-- just as a pattern that is never used -->
<!ELEMENT geo (name+,islands?,mountains?,located*,to*,from?,
        area?,length?,latitude?,longitude?,elevation?,depth?,
        source?,estuary?)>
<!ATTLIST geo
        id ID #REQUIRED
        country IDREFS #IMPLIED
        type NMTOKEN #IMPLIED
        sea IDREFS #IMPLIED
        lake IDREFS #IMPLIED
        island IDREF #IMPLIED
        bordering IDREFS #IMPLIED>

<!ELEMENT river (name,located*,to?,through*,area?,length?,source?,estuary?)>
<!ATTLIST river
        id ID #REQUIRED
        country IDREFS #REQUIRED>

<!ELEMENT length (#PCDATA)>
<!ELEMENT depth (#PCDATA)>

<!ELEMENT to EMPTY>
<!ATTLIST to
        watertype (river|sea|lake) #REQUIRED
        water IDREF #REQUIRED>
<!ELEMENT through EMPTY>
<!ATTLIST through
        lake IDREFS #REQUIRED>

<!ELEMENT source (mountains?,located*,from?,latitude?,longitude?,elevation?)>
<!ATTLIST source
        country IDREFS #REQUIRED>

<!ELEMENT from EMPTY>
<!ATTLIST from
        watertype (river|sea|lake) #REQUIRED
        water IDREFS #REQUIRED>
        <!-- a river may evolve from one or more waters -->

<!ELEMENT estuary (located*,latitude?,longitude?)>
<!ATTLIST estuary
        country IDREFS #REQUIRED>

<!ELEMENT located EMPTY>
<!ATTLIST located
        country IDREF #REQUIRED
        province IDREFS #IMPLIED>
```

```
<!ELEMENT lake (name+,located*,to?,area?,latitude?,longitude?,elevation?,depth?)>
<!ATTLIST lake
        id ID #REQUIRED
        country IDREFS #REQUIRED
        island IDREF #IMPLIED
        type (salt|acid|artificial|dam|caldera|crater|impact|naturaldam|asphalt) #IMPLIED>

<!ELEMENT sea (name+,located*,area?,depth?)>
<!ATTLIST sea
        id ID #REQUIRED
        country IDREFS #REQUIRED
        bordering IDREFS #IMPLIED>

<!ELEMENT desert (name+,located*,area?,latitude?,longitude?)>
<!ATTLIST desert
        id ID #REQUIRED
        country IDREFS #REQUIRED
        type (sand|rocks|lime|salt|ice) #IMPLIED>

<!ELEMENT island (name+,islands?,located*,area?,latitude?,longitude?,elevation?)>
<!ATTLIST island
        id ID #REQUIRED
        sea IDREFS #IMPLIED
        lake IDREF #IMPLIED
        river IDREFS #IMPLIED
        country IDREFS #REQUIRED
        type (volcanic|coral|atoll|lime) #IMPLIED>
<!ELEMENT islands (#PCDATA)>
<!ELEMENT mountain (name+,mountains?,located*,latitude?,longitude?,elevation?)>
<!ATTLIST mountain
        id ID #REQUIRED
        country IDREFS #REQUIRED
        island IDREF #IMPLIED
        type (volcanic|volcano|monolith|granite) #IMPLIED
        last_eruption CDATA #IMPLIED>
<!ELEMENT mountains (#PCDATA)>

<!ELEMENT airport (name+,latitude,longitude,elevation?,gmtOffset,located_on?)>
<!ATTLIST airport
        iatacode NMTOKEN #REQUIRED
        city IDREF #IMPLIED
        country IDREF #REQUIRED>
<!ELEMENT gmtOffset (#PCDATA)>
```

8. Literaturverzeichnis

[1] Mogensen, Torben Ægidius. *Basics of Compiler Design.* Kopenhagen: lulu.com, 2010.

[2] *PostgreSQL.* PostgreSQl Global Development Group. kein Datum. www.postgresql.org (Zugriff am November 2016).

[3] *Saxonica.* kein Datum. www.saxonica.com/documentation/ (Zugriff am November 2016).

[4] Schlicke, David Nicholas. „Projektbericht - XML zum relationalen Modell.“ Göttingen, 2016.

[5] *The MONDIAL Database.* 1998. https://www.dbis.informatik.uni-goettingen.de/Mondial/ (Zugriff am Dezember/Januar 2016/17).

[6] Thomas, Jürgen. *Einführung in SQL.* Berlin: Jürgen Thomas, 2012.

[7] *W3C DTD.* kein Datum. www.w3.org/TR/xhtml1/dtds.html (Zugriff am Dezember 2016).

[8] *W3C XML.* kein Datum. www.w3.org/XML/ (Zugriff am Dezember 2016).

[9] *W3C XPath.* kein Datum. www.w3.org/TR/xpath/ (Zugriff am Dezember 2016).

[10] *w3schools.com.* kein Datum. http://www.w3schools.com/sql/sql_join.asp (Zugriff am Januar 2017).