UNIVERSITÄT LEIPZIG
Fakultät für Mathematik und Informatik
Institut für Informatik

Implementierung der XPath-Anfragesprache für XML-Daten in RDBMS unter Ausnutzung des Nummerierungsschemas DLN

Diplomarbeit

Aufgabenstellung und Betreuung:

Prof. Erhard Rahm
Timo Böhme

Leipzig, Dezember 2005

vorgelegt von:

Schmidt, Oliver
geb. am: 09.03.1981

Studiengang Informatik

Aktualisierte Fassung vom 02.02.2005. Ein Fehler bei der Übertragung der Benchmarkergebnisse hatte eine Korrektur der Seiten 69 - 72 und 85 zur Folge.

XML-Dokumente haben sich in den letzten Jahren zu einem wichtigen Datenformat für die standardisierte Übertragung von vielfältigen Informationen entwickelt. Dementsprechend gibt es einen großen Bedarf nach Speicherlösungen für XML-Daten. Neben den nativen XML-Datenbanken bieten sich aber zunehmend auch relationale Datenbanksysteme mit unterschiedlichen Ansätzen für die Speicherung der Dokumente an. Die Art der Speicherung ist jedoch nur ein Aspekt der XML-Datenhaltung - die Anwender wollen auch mit ihren gewohnten XML-Schnittstellen auf die Daten zurückgreifen.

Das XMLRDB-Projekt bietet dafür ein Nummerierungsschema für XML-Knoten an, welches es erlaubt, aus den in Relationen gespeicherten Daten Strukturinformationen zu gewinnen. In dieser Diplomarbeit werden diese Informationen für eine XPath-Schnittstelle in XMLRDB genutzt, welche dadurch in der Lage ist, XPath-Anfragen nach SQL zu konvertieren und effizient deren Lösungsmenge zu bestimmen.

Für diese Schnittstelle werden verschiedene Verfahren für die Umsetzung der XPath-Strukturen vorgestellt. An Hand einer Implementierung wird gezeigt, wie die Fähigkeiten von unterschiedliche Datenbanksystemen gewinnbringend in das Schema integriert werden können. Mittels eines Benchmarks findet schließlich eine Analyse der XPath-Umsetzung hinsichtlich Effizienz und Performanz statt.

Inhaltsverzeichnis

1. Einführung **1**
1.1. Das XMLRDB-Projekt 2
1.2. Aufgabenstellung 3
1.3. Gliederung der Arbeit 3

2. Grundlagen **4**
2.1. XML und Anfragesprachen 4
2.1.1. XML 4
2.1.2. XPath 8
2.1.3. XQuery 1.0 10
2.1.4. XPath 2.0 11
2.2. Optimierung von XPath-Ausdrücken durch Umformung 12
2.3. Speicherung, Abfrage und Änderungen von XML-Daten in RDBMS . 13
2.3.1. Schemaabhängige Speicherverfahren 13
2.3.2. Schemaunabhängige Speicherverfahren 16
2.4. Das Nummerierungsschema DLN 23
2.4.1. Grundlagen 23
2.4.2. Die DLN-DOM-ID 25
2.4.3. Die DLN-Prefix-ID 26
2.4.4. Vorteile der DLN-Nummerierung 28

3. Konzeption und Architektur **30**
3.1. Das Projekt XMLRDB 30
3.1.1. Konzept des modularen Mediators 30
3.1.2. Arbeiten im Vorfeld dieser Diplomarbeit 32
3.1.3. Das relationale Schema 33

3.2. Konzeptionsphase 36
3.2.1. Grundlagen der XPath-zu-SQL-Konvertierung 36
3.2.2. Übersetzung eines XPath-Ausdruckes nach SQL 39
3.2.3. Die abstrakte SQL-Darstellung 41
3.2.4. Die spezielle SQL-Darstellung 41
3.2.5. Umsetzung der XPath-Achsen mit Hilfe von DLN 42
3.2.6. Realisierung der Knotentests und Prädikate 45
3.2.7. Umsetzung von XPath-Funktionen 46
3.3. Abgrenzung 48

4. Umsetzung **51**
4.1. Die Software-Basis 51
4.2. Einbindung als Modul in XMLRDB 52
4.3. UDFs für die descendant- und ancestor-Achse 53
4.3.1. UDFs in PostgreSQL 54
4.3.2. UDFs in MySQL 55
4.4. Zweistufige Konvertierung von XPath nach SQL 56
4.4.1. Die abstrakte SQL-Darstellung 56
4.4.2. Die spezielle SQL-Darstellung 59
4.4.3. Optimierungen der SQL-Darstellung 60
4.4.4. Datenbankspezifische Anpassungen und Optimierungen 62
4.5. Ergebnisse 63
4.5.1. Testdaten 63
4.5.2. Benchmarkeinstellungen 65
4.5.3. Benchmarkresultate und Interpretation 66
4.5.4. Fazit 71

5. Zusammenfassung **73**

Quellenverzeichnis **75**

Abbildungsverzeichnis **79**

Tabellenverzeichnis **79**

A. Inhalt der beiliegenden CD **80**

B. Das XPath-Modul **81**

B.1. Verwendung des XPath-Moduls 81

B.2. User Defined Functions kompilieren und laden 82

C. Benchmark-Resultate **85**

D. Danksagung **86**

E. Erklärung **87**

1. Einführung

Mit der zunehmenden Bedeutung von XML-Daten in Anwendungen und vor allem beim Datenaustausch zwischen unterschiedlichen Quellen wächst auch die Nachfrage nach Speicherlösungen für die strukturierten Daten. Doch obwohl es schon diverse native XML-Datenbank-Lösungen gibt, besteht ein Bedarf danach, Relationale Datenbank-Management-Systeme (RDBMS) zur Speicherung von XML-Daten zu verwenden. Dies hat seine Ursachen zum Einen darin, dass RDBMS nach über 25 Jahren Entwicklung sehr ausgereift und performant bezüglich der Speicherung und Abfrage von Daten sind. Sie skalieren hervorragend auch bei großen Datenmengen und bieten mit ihren typischen Merkmalen wie dem ACID-Modell genügend Sicherheit für die Speicherung sensibler Dokumente.

Zum Anderen sind RDBMS in vielen Unternehmen bereits im Einsatz und die Mitarbeiter erfahren im Umgang damit, so dass eine Lösung für die bestehenden Systeme aus ökonomischer Sicht bevorzugt wird. Da XML-Dokumente in der Wirtschaft an Bedeutung gewinnen, haben folgerichtig auch alle großen Datenbankhersteller Erweiterungen ihrer DBMS für XML-Daten im Angebot (für einen Vergleich siehe [S02]).

Zur effektiven Nutzung von in RDBMS gespeicherten XML-Dokumenten ist es aber nötig, die Daten nicht nur zu speichern, sondern auch darauf zurückzugreifen und Anfragen, Updates und die Ausgabe mit XML-typischen Schnittstellen wie XPath, XUpdate oder DOM durchzuführen. Die meisten Erweiterungen der bekannten Datenbanken leisten dies aber noch nicht oder nur teilweise [S02], so dass für ein System, welches mit beliebigen Schnittstellen auf den XML-Daten arbeiten soll, auf die allgemeine Basis der RDBMS zurückgegriffen werden muss.

Dies bedeutet, die XML-Daten in einem relationalen Schema zu speichern und Anfragen und Updates mit der Sprache SQL darauf auszuführen. Für die Umwandlung der XML-Schnittstellen auf die relationale Ebene bzw. die Umformung in SQL-Ausdrücke ist dann eine generische Zwischenschicht zwischen dem Anwender und der Datenbank verantwortlich.

Ein entsprechendes System sollte dabei einer Reihe von Anforderungen genügen:

- automatische, verlustfreie Speicherung von beliebigen XML-Daten ohne Angabe eines Schemas (DTD, XML Schema) in einem RDBMS
- Unterstützung des sequentiellen Einlesens von XML-Dokumenten (Streaming)
- effiziente Aktualisierung der gespeicherten Daten ohne aufwendige Strukturanalysen der bestehenden Daten
- Erhalten der Struktur der XML-Daten und schnelle Rekonstruktion von Teilbäumen oder ganzen Dokumenten
- Zugriff über Standard-XML-Schnittstellen wie zum Beispiel XPath unter Einbeziehung von optimierten Zugriffsstrukturen

1.1. Das XMLRDB-Projekt

Das XMLRDB-Projekt an der Abteilung Datenbanken der Universität Leipzig hat es sich zum Ziel gesetzt, ein System vorzustellen, welches in der Lage ist, die genannten Anforderungen zu erfüllen. Eine Mediator genannte Zwischenschicht übernimmt dabei sämtliche Aufgaben bei der Vermittlung zwischen Anwender und Datenbank (siehe Kapitel 3.1.1). Nach außen stellt der Mediator verschiedene Schnittstellen wie DOM oder XPath[1] zur Verfügung; die Kommunikation mit der Datenbank erfolgt jedoch ausschließlich über die Datenbank-Anfragesprache SQL. Für jede der XML-Schnittstellen übernimmt ein eigenes Modul innerhalb des Mediators die Konvertierung nach SQL.

Ein wichtiger Grundstein dieser Umwandlung ist das Nummerierungsschema DLN (Dynamic Level Numbering) für XML-Daten, welches in Kapitel 2.4 näher vorgestellt wird. Es vergibt an jeden Knoten im XML-Baum eine feste ID, aus welcher sich die Dokumentenordnung und andere Strukturinformationen der Daten ablesen lassen. Die Ausnutzung dieser Informationen ist ein essentieller Bestandteil sowohl beim Einlesen der Dokumente als auch bei der Wiederherstellung und Abfrage von Teilen der Daten oder ganzer Dokumente.

[1] für eine Erläuterung der Schnittstellen wird auf Kapitel 2 verwiesen

1.2. Aufgabenstellung

Das Ziel dieser Diplomarbeit ist die Implementierung der XPath-Anfragesprache als Modul in das XMLRDB-Projekt. Nach der Eingabe einer XPath-Anfrage soll das Modul eine SQL-Query generieren, welche auf der Datenbank ausgeführt und deren Ergebnismenge schließlich in eine gültige Antwort auf die XPath-Query umgewandelt wird.

Mittels XPath können so ausgewählte Teile der gespeicherten XML-Daten abgefragt werden. Als grundlegender Bestandteil von XQuery und XSLT ist ein XPath-Modul außerdem ein wichtiger Baustein für weitere Schnittstellen. In Hinblick auf die aktuelle Entwicklung von XPath 2.0 (siehe Kapitel 2.1.4) soll das Modul auf diesen neuen Standard vorbereitet sein.

Mit der Umsetzung von XPath für das XMLRDB-Projekt soll außerdem gezeigt werden, welche realen Vorteile sich aus der Speicherung von XML-Daten mittels des Nummerierungsschemas DLN ergeben. Praktische Tests sollen schließlich aufzeigen, wie sich verschiedene Varianten der DLN-ID und des relationalen Schemas auf die Performanz auswirken und wo es eventuelle Schwachstellen gibt.

1.3. Gliederung der Arbeit

Die Diplomarbeit gliedert sich wie folgt: In Kapitel 2 werden nach einer Einführung in XML und die dazu gehörende Anfragesprache XPath verschiedene Ansätze für das Problem der Speicherung von XML-Daten in einem RDBMS verglichen und in Bezug auf die Zielstellung dieser Diplomarbeit bewertet. Danach wird ausführlich das Nummerierungsschema DLN und das damit verbundene relationale Schema vorgestellt, welche zusammen die oben genannten Anforderungen erfüllen.

Kapitel 3 beschäftigt sich danach mit der Frage, wie die Konventierung von XPath-Anfragen nach SQL mit Hilfe von DLN aussehen kann. Die daraus folgende Umsetzung und die praktischen Tests werden in Kapitel 4 beschrieben. Die Diplomarbeit schließt mit einer Zusammenfassung der gewonnenen Erkenntnisse und einem Ausblick auf mögliche fortführende Arbeiten in Kapitel 5.

2. Grundlagen

In diesem Kapitel wird einen Überblick über die wichtigsten Grundlagen dieser Diplomarbeit gegeben. Als Erstes folgt eine Einführung in das Datenformat XML und die dazu gehörende Anfragesprache XPath (Kapitel 2.1). Danach wird sich der Frage gewidmet, auf welche Art und Weise eine Speicherung von XML-Daten in einem RDBMS erfolgen kann (Kapitel 2.3). Schließlich stellt Abschnitt 2.4 mit DLN ein Nummerierungsschema vor, welches bei der Erfüllung der in Kapitel 1 gestellten Anforderungen eine wichtige Rolle spielt und bei der Umsetzung dieser Diplomarbeit zum Einsatz kam.

2.1. XML und Anfragesprachen

2.1.1. XML

XML (eXtensible Markup Language) [WWW98] ist eine formale Beschreibungssprache für textbasierte Datenformate, die für den einfachen Austausch und die Verarbeitung von strukturierten Daten entwickelt wurde. Dies bedeutet, dass die Dokumente nicht nur den reinen Inhalt, sondern auch zusätzliche Metadaten wie Struktur und Eigenschaften darüber enthalten. Im Gegensatz zu Komma-separierten Dateien sind XML-Dokumente damit flexibler und deshalb auch vielseitiger anwendbar.

Die XML-Spezifikation definiert die Regeln für den Aufbau solcher Dokumente, die das Format sowohl maschinenlesbar, mittels eines Schemas kontrollierbar, als auch automatisiert verarbeitbar machen sollen. Neben der Syntax (die Verwendung von Zeichendaten und Tags `<...>` als Markup für die Trennung von Inhalt und Struktur) sieht die XML Recommendation [WWW98] auch ein Datenmodell für den transportierten Inhalt der Daten vor. Das Dokument wird darin als Textrepräsentation eines Baumgraphen angesehen, der aus verschiedenen Knotentypen besteht. Der Baum wird durch das Parsen des Dokumentes gewonnen; die Abbildung ist jedoch nicht absolut eindeutig (dazu später mehr). Als Beispiel ist in Abbildung

```
<?xml version="1.0"?>
<?xml-stylesheet href="style.xsl" type="text/xml"?>
<rezept>
   <zutat id="mehl">200g Mehl</zutat>
   <!--weitere Zutaten-->
   <anleitung>
      Zuerst nehmen Sie das
      <zutat>Mehl</zutat>
      und mischen es mit ...
   </anleitung>
</rezept>
```

Abbildung 2.1.: Ein einfaches XML-Dokument

2.1 ein einfaches XML-Dokument aus der deutschen Version von [WWW99] gegeben und in Abbildung 2.2 seine Baumdarstellung.

Der wichtigste Knotentyp im XML-Datenmodell ist das Element, welches in der Textform als

```
<elementname attributname="attributwert">Inhalt</elementname>
```

dargestellt wird. Es besteht aus einem öffnenden und einem schließenden Tag, welche für Elemente ohne Inhalt auch zusammenfallen können. Desweiteren kann ein Element wie abgebildet Attribute für die Definition zusätzlicher Eigenschaften enthalten. Der Element- bzw. Attributname sollte dabei als Bezeichner für den Inhalt fungieren, also z.B.

```
<zutat id="mehl">200g Mehl</zutat>.
```

Die Namen der Knoten (QName für qualifizierter Name) bestehen dabei aus einem optionalen Namensraumanteil und dem eigentlichen Bezeichner, getrennt durch einen Doppelpunkt (z.B. `xml:lang`). Die Angabe eines Namensraumes ermöglicht es, den gleichen Bezeichner in unterschiedlichen Kontexten zu verwenden. Ansonsten können zwei Elemente mit identischem Bezeichner nicht in ihrer Bedeutung unterschieden werden. Um einen Namensraumbezeichner zu benutzen, muss dieser jedoch vorher definiert werden. Dies geschieht in einer Namensraumdeklaration, in der dem Namensraumbezeichner eine eindeutige URI[1] zugeordnet wird (zum Beispiel `xmlns:xs="http://www.w3.org/2001/XMLSchema"`).

[1]Uniform Resource Identifier (URI): eine zur Identifizierung einer abstrakten oder physikalischen Ressource dienende Zeichenkette

In der Baumdarstellung wird der (einfache) Inhalt eines Elementes als ein Textknoten und jedes Attribut als ein einzelner Attributknoten angesehen. Elemente können auch beliebig geschachtelt werden, solange die Hierarchie beibehalten wird. Alle Elemente, die innerhalb des öffnenden und schließenden Tags eines anderen Elementes vorkommen, sind dessen Nachfahren. Wenn sie genau eine Hierarchieebene unter dem Element liegen, sind sie ebenfalls dessen Kinder.

Auch das als Mixed Content bezeichnete benachbarte Auftreten von unterschiedlichen Knotentypen ist erlaubt (als Beispiel sei das Element `anleitung` in Abbildung 2.1 genannt). Attributknoten sind aber per Definiton keine Kinder des zugeordneten Elementes, obwohl dieses der Vater aller seiner Attribute ist. Attribute können auch nur Elementen zugeordnet werden.

Die weiteren im XML-Datenmodell erlaubten Knotentypen sind Kommentarknoten (in `<!--...-->` eingeschlossene Anmerkungen), Processing-Instructions (Verarbeitungsanweisungen für Anwendungen innerhalb von `<?...?>`) und Namensraumknoten, die jedem Element, dessen Name oder Attribut einen Namensraumbezeichner enthält, die vorher definierte URI zuordnen. Außerdem gibt es im Datenmodell zusätzlich zu den in der Textform vorhandenen Knoten einen Wurzelknoten, welcher das logische oberste Element eines XML-Dokument darstellt und damit der Vater aller Knoten auf der obersten Hierarchieebene ist.

Um auf die Baumstruktur eines XML-Dokumentes zurückgreifen und z.B. XPath-Anfragen (siehe Kapitel 2.1.2) beantworten zu können, muss es jedoch erst analysiert werden. Dies erledigt ein als Parser bezeichnetes Programm, welches aus dem flachen Text die Struktur extrahieren kann. Weit verbreitet sind dabei SAX-Parser. Diese erzeugen beim Lesen von bestimmten Syntax-Elementen wie den Tags oder den Attributdeklarationen ein entsprechendes SAX-Event, welches dann durch eine weitere Software interpretiert werden kann.

Um die für das Markup verwendeten oder in dem genutzten Zeichensatz[2] nicht enthaltenen Zeichen in XML-Dokumenten darzustellen, werden Entitäten verwendet. Diese können nach ihrer Deklaration über die Syntax `&Name;` aufgerufen werden; einige wie z.B. `&amp;` für `&` sind schon standardmäßig definiert und werden vom Parser durch das entsprechende Zeichen ersetzt, so dass sie in der Baumdarstellung nicht mehr auftauchen. Innerhalb von sogenannten CDATA-Sektionen (`<![CDATA[...]]>`) können dagegen Textinhalte dargestellt werden, ohne dass der Parser versucht, darin Markup zu erkennen. Diese Sektionen sind in der resultieren-

[2] `UTF-8` ist als Standard vorgesehen

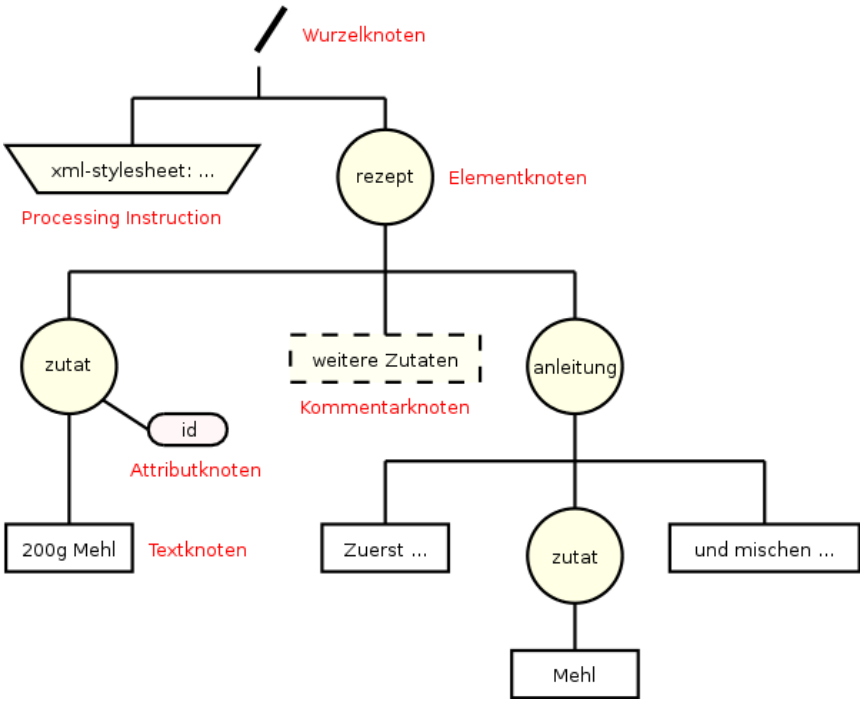

Abbildung 2.2.: Die Baumdarstellung des Datenmodells für das XML-Dokument aus Abbildung 2.1

den Baumdarstellung nicht mehr zu finden - ihr Inhalt wird als Textknoten abgebildet.

Für diese Baumdarstellung gibt es ein standardisiertes Format namens DOM (Document Object Model). Es enthält für jeden Knoten im Baum eine Objektrepräsentation, welche über Referenzen zu seinen Vater-, Kinder- und direkten Nachbarknoten sowie Attributen verfügt. Damit wird innerhalb des DOM-Baumes die Struktur der XML-Daten nachgebildet. Diese kann nun genutzt werden, um weitere Beziehungen von Knoten zu analysieren, welche aus der Textform nicht sofort ersichtlich sind.

Entspricht ein Textdokument der in der XML-Spezifikation vorgegebenen Syntax, so wird es als wohlgeformt bezeichnet. Für den Datenaustausch zwischen Anwendungen muss ein Dokument jedoch weiteren Anforderungen genügen betreffend der festgelegten Anordnung von Elementen oder deren grundsätzlichem Auftreten. Für die Beschreibung dieser Anforderungen gibt es die mehrere Möglichkeit, die eine Kontrolle der Dokumente durch den Vergleich mit einem Schema ermöglichen. Die beiden bekanntesten Schemadefinitionen, DTD und XML Schema, werden nun kurz vorgestellt.

Eine DTD (Document Type Definition) ist eine kontextfreie Grammatik für XML-

Dokumente. Über Regeln kann darin beschrieben werden, wie ein der DTD entsprechendes XML-Dokument aufgebaut sein muss bezüglich der enthaltenen Knoten und deren Auftreten im XML-Baum. Jede DTD definiert damit eine eigene Klasse von Dokumenten, zu welcher genau jene Dokumente gehören, die in der von der DTD-Grammatik erzeugten Sprache enthalten sind. Durch diese formelle Definition können XML-Dokumente automatisiert überprüft werden, ob sie dem in der DTD vorgegebenen Schema entsprechen. Dieser Vorgang wird als Validierung bezeichnet.

DTDs beschränken sich auf die wichtigsten Einheiten des XML-Datenmodells: Elemente, Attribute und Textinhalte. Dies ist aber nicht für alle Anwendungen ausreichend. So kann es vorkommen, dass nach einer erfolgreichen Validierung noch manuell geprüft werden muss, ob ein Wert innerhalb des Dokumentes einem festen Datentyp entspricht oder ein Element aus einem bestimmten Namensraum stammt. Um diesen Nachteil zu beheben wurde XML Schema entwickelt, eine weitaus mächtigere und damit auch komplexere Beschreibungssprache für typorientierte XML-Daten.

Ein XML Schema ist selber ein XML-Dokument und hat so den Vorteil, in XML-Datenbanken zusammen mit den damit beschriebenen Dokumenten gespeichert werden zu können und keinen zusätzlichen Parser wie für die Verarbeitung von DTDs zu benötigen. Es erlaubt die Defintion von Datentypen und bringt bereits eine Anzahl erweiterbarer Typen mit (z.B. `xs:date` für das Datum). Für jeden Datentyp lassen sich Einschränkungen des Wertebereiches und Muster für die Belegung durch reguläre Ausdrücke festlegen. Außerdem kann mit XML Schema explizit beschrieben werden, welche Namensräume verwendet werden dürfen.

2.1.2. XPath

Die XML Path Language (XPath) [WWW99] ist eine auf dem XML-Datenmodell arbeitende Anfragesprache, die über Pfadausdrücke eine Teilmenge der Knoten eines Dokumentes adressieren kann. Aus diesem Grund ist XPath die Grundlage verschiedener weiterer Standards wie XSLT, XPointer und XQuery, welche die Ergebnisse der Pfadausdrücke weiterverarbeiten.

Auf der geparsten und geordneten Baumdarstellung des XML-Dokumentes definiert XPath eine Reihe von Achsen, die von einem Kontextknoten (Referenzknoten) aus auf Pfaden entlang der Baumstruktur navigieren. Neben einfachen Achsen wie `parent` für die Auswahl des Vaterknotens oder `attribute` für die Einschränkung auf Attributknoten sind aber auch komplexere Achsentypen vorgesehen. In Tabelle 2.1 werden alle verfügbaren Achsen kurz vorgestellt.

Achsenname	*Beschreibung*
`self`	wählt den Kontextknoten aus
`child`	liefert die Kinder des Kontextknotens
`parent`	wählt den Vaterknoten des Kontextknotens aus
`descendant`	liefert alle Nachfahren des Kontextknotens
`descendant-or-self`	liefert alle Nachfahren des Kontextknotens und den Kontextknoten selber
`ancestor`	liefert alle Vorfahren des Kontextknotens
`ancestor-or-self`	liefert alle Vorfahren des Kontextknotens und den Kontextknoten selber
`following-sibling`	liefert alle Nachbarknoten des Kontextknotens, die in Dokumentenordnung folgen
`following`	liefert alle in Dokumentenordnung dem Kontextknoten folgenden Knoten
`preceding-sibling`	liefert alle Nachbarknoten des Kontextknotens, die in Dokumentenordnung vor dem Kontextknoten auftreten
`preceding`	liefert alle in Dokumentenordnung vor dem Kontextknoten auftretenden Knoten
`namespace`	liefert die Namensraumknoten des Kontextknoten
`attribute`	liefert die Attributknoten des Kontextknoten

Tabelle 2.1.: Die Achsen der XPath-Anfragesprache

Ein XPath-Ausdruck besteht aus einzelnen Lokationsschritten, getrennt durch `/`-Zeichen. Jeder Schritt liefert abhängig von der Kontextknotenmenge eine neue Menge von Knoten des Dokumentes. Ein Schritt kann dabei neben der Angabe einer Achse auch einen Knotentest und Prädikate für die weitere Einschränkung der selektierten Knoten enthalten. Achse und Knotentest werden dabei durch `::` voneinander getrennt. Der Knotentest ermöglicht es, die ausgewählte Knotenmenge auf einen bestimmten Namen oder Typ (z.B. `comment()` für Kommentare) einzuschränken. Prädikate in eckigen Klammern (`[...]`) dürfen zur weiteren Selektion u.a. komplette XPath-Schritte und eine Reihe von auf der Kontextknotenmenge arbeitenden Funktionen enthalten. Neben der Abfrage der Dokumentenordnung (`[position() = 2]` oder einfach `[2]`) stehen auch logische, mathematische und String-Funktionen zur Verfügung. Eine Übersicht der umzusetzenden Funktionen bietet das Kapitel 3.2.7.

Eine gültige XPath-Anfrage für die Auswahl aller Rezepte mit der Zutat Zucker wäre

```
/descendant-or-self::rezept[child::zutat = 'Zucker'].
```

Sie wählt in einem Schritt vom Wurzelknoten ausgehend (das erste `/`) alle Elemente mit dem Namen `rezept` aus. Diese Knotenmenge wird eingeschränkt auf diejenigen Rezepte, die ein Kind mit dem Elementnamen `zutat` haben, dessen Wert gleich `Zucker` ist.

Da einige Achsen häufiger als andere benutzt werden, sind für sie Kurzformen vorgesehen. So kann ein Schritt `/descendant-or-self::node()/` mit `//` abgekürzt werden, `self::node()` mit `.` und `child` als Standard-Achse wird bei allen Schritten verwendet, die keine Achse spezifizieren. Außerdem sind die Kurzformen `@` für die `attribute`-Achse und `..` für `parent::node()` erlaubt. Die Anfrage für die Auswahl der Rezepte, welche die Zutat Zucker enthalten, würde in der Kurzform dann so aussehen:

```
//rezept[zutat = 'Zucker'].
```

2.1.3. XQuery 1.0

Die XML Query Language (kurz XQuery) in der Version 1.0 [WWW05Q] ist eine auf XPath basierende Anfragesprache für XML-basierte Datenbanken. Während XPath für Anfragen auf Dokumenten ausreichend mächtig ist, stößt es bei großen Datenkollektionen schnell an seine Grenzen. Joins oder Quantifizierer sind ebenso wenig möglich wie eine Sortierung oder Restrukturierung von (Zwischen-) Ergebnissen. Außerdem ist XPath beschränkt auf die vier Datentypen Knotenmenge, boolescher Wert, numerischer Wert und Zeichenkette mit einer dementsprechend schmalen Funktionsbibliothek.

XQuery behebt diese Limitierungen, benutzt aber das parallel weiterentwickelte XPath 2.0 (dazu mehr in Abschnitt 2.1.4) für die Beschreibung der Pfadausdrücke. Anfragen in XQuery erlauben die Verwendung von Variablen und erweiterten Konstrukten wie Schleifen (`for $var in expr`) und Bedingungen (`if ... then ... else`). Wegen dem Hauptanwendungsgebiet in Datenbanksystemen ähnelt die XQuery-Syntax mit ihren `FLWOR`-Ausdrücken (gesprochen: Flower) der von SQL. `FLWOR` steht dabei für

```
FOR ... LET ... (WHERE ...) (ORDER BY ...) RETURN ...,
```

wobei die Klauseln in Klammern optionale Teile darstellen. Diese Syntax ermöglicht Gruppierungen und Verbundoperationen auf Knotenmengen (`LET`), eine Sortierung

unabhängig von der Dokumentenordnung (`ORDER BY`) und sogar die Konstruktion von neuen Elementen zur Ausgabe des Ergebnisses (`RETURN`). Ein Beispiel für eine XQuery-Anfrage auf dem Dokument 2.1:

```
FOR $r := //rezept
LET $z := $r/zutat
WHERE count($z) < 3
RETURN <vorschlag> {$r} </vorschlag>
```

Diese Anfrage prüft für jedes Rezept, ob die Anzahl der Zutaten kleiner als drei ist (dazu werden Zwischenergebnismengen in `$z` gebildet) und konstruiert als Ausgabe ein neues `vorschlag`-Element für jedes einzelne Ergebnis mit den Rezepten als Inhalt.

Neben der erweiterten Syntax kann XQuery auch mit den vordefinierten Datentypen aus XML Schema umgehen und kennt aus diesem Grund ein deutlich erweitertes, typisiertes Spektrum an Funktionen - der Nutzer kann sogar eigene Funktionen definieren. Damit ist XQuery eine sehr mächtige Anfragesprache, die allerdings keine Einfüge- und Änderungsoperationen auf den Daten vorsieht.

2.1.4. XPath 2.0

In XPath 2.0 [WWW05P], welches sich momentan zusammen mit XQuery 1.0 in der Entwicklung zum Standard befindet, wurden einige der für Anfragen auf Dokumenten interessanten Weiterentwicklungen aus XQuery 1.0 übernommen. So kennt die Version 2.0 jetzt auch das Typsystem von XML Schema mit dessen vordefinierten Datentypen und unterstützt Dokumentenkollektionen[3]. Sämtliche Funktionen von XQuery haben ebenfalls den Weg in den XPath-Nachfolger gefunden. Durch die vielen möglichen Typen als Ergebnis eines XPath-Ausdruckes wird die Rückgabemenge nun als eine Sequenz von Elementen mit möglicherweise unterschiedlichen Datentypen definiert. Daraus folgt, dass Knotenmengen nicht mehr zwangsweise duplikatfrei sind und eine eigene, von der Dokumentenordnung möglicherweise unterschiedliche Ordnung besitzen können.

Eine weitere wichtige Neuerung ist die Einführung von Variablen und die damit ermöglichte Bildung von `for`-Schleifen und `if`-Bedingungen in einer Anfrage.

[3]Dies äußert sich u.a. darin, dass es keine Wurzelknoten mehr gibt, sondern nur noch Dokumentenknoten.

Ebenfalls gibt es nun Möglichkeiten für Vereinigungen, die Bildung von Schnittmengen (`union, intersect, except`) und Unschärfekondition (`some / every $var satisfies ...`) zwischen Sequenzen, die z.B. das Verhalten bei Vergleichen von Knotenmengen mit Werten[4] eindeutig festlegen.

2.2. Optimierung von XPath-Ausdrücken durch Umformung

Die Optimierung von XPath-Ausdrücken kann auf vielfältige Art und Weise erfolgen. Neben der Ausnutzung von semantischen Informationen durch die gewählte Speicherform der XML-Daten gibt es aber auch Ansätze für die Umformung der XPath-Ausdrücke auf syntaktischer Ebene mit dem Ziel der Vereinfachung.

In [CFZ04] wird dabei versucht, Wildcards aus XPath-Ausdrücken (z.B. `/child::*` oder `/*/child::name`) durch Umordnung der Schritte zu entfernen und damit die Anfragen auf nativen XML-Datenbanken zu beschleunigen. Die umgeformten Ausdrücke haben jedoch eine höhere Komplexität als zuvor und bestehen meist aus mehreren zusätzlichen `descendant`-Schritten.

[OMFB02] dagegen stellt ein umfangreiches theoretisches Regelwerk für die semantisch korrekte Umformung von XPath-Ausdrücken vor, welches es u.a. erlaubt, Umkehrschritte bei der Verwendung der `descendant`-Achse zu vermeiden. So lassen sich zum Beispiel die beiden XPath-Ausdrücke

```
/descendant-or-self::name1/parent::name2
/descendant::name3/preceding::name4
```

ersetzen durch:

```
/descendant::name2[child::name1]
/descendant::name4[following::name3]
```

Dieses Regelwerk wurde ursprünglich als Hilfe für das Verarbeiten von XML-Streams konzipiert, wie sie zum Beispiel bei der Verwendung eines SAX-Parsers als

[4] Der Ausdruck `//zutat = 'Mehl'` wurde unter XPath 1.0 als `true` ausgewertet, wenn wenigstens ein Element der Knotenmenge den Wert 'Mehl' hatte.

XPath-Prozessor auftreten. Dabei ist es von Vorteil, das Dokument nicht mehrmals parsen zu müssen, um die rückwärts gerichteten Achsen auszuführen. Aber auch bei der relationalen Speicherung kann es ein Vorteil sein, nicht erst große Zwischenmengen an Knoten in tiefergelegenen Bereichen der Baumstruktur zu erzeugen. Das XMLRDB-Projekt, auf dem diese Diplomarbeit basiert, nutzt deshalb dieses Regelwerk bei der optionalen Optimierung der gestellten XPath-Anfragen [B05].

2.3. Speicherung, Abfrage und Änderungen von XML-Daten in RDBMS

Für die Speicherung bzw. Abfrage von XML-Daten bei Verwendung eines relationalen Datenbanksystemes existieren eine große Anzahl von verschiedenen Verfahren. Grundsätzlich gibt es die Möglichkeit, ganze Dokumente in Zeichenkettenform als CLOB oder Textdatei in der Datenbank abzulegen oder die XML-Daten modellbasiert auf Relationen herunterzubrechen. Das erste Verfahren ist zwar sehr performant wenn es darum geht, ein ganzes Dokument aus der Datenbank zu laden, benötigt jedoch aufgrund der fehlenden Informationen über die Struktur der Daten vor jeder Anfrage oder Aktualisierung des Dokumentes einen kompletten, zeitaufwändigen Parser-Durchlauf [S02].

Deshalb wurden für diese Diplomarbeit nur modellbasierte Verfahren untersucht. Dieses Kapitel gibt nun einen Überblick über Ansätze zur relationalen Speicherung von XML-Dokumenten und bewertet diese in Hinblick auf die Zielstellung dieser Diplomarbeit und die in Kapitel 1 genannten Punkte. Nach der Vorstellung von einigen schemaabhängigen Speicherverfahrung und ihren Techniken folgen schließlich in Kapitel 2.3.2 diejenigen Ansätze, welche sämtliche XML-Daten in ein festes Schema abbilden.

2.3.1. Schemaabhängige Speicherverfahren

Für die Konvertierung von relationalen Daten nach XML und umgekehrt wird immer ein bestimmtes Schema benötigt, welches die XML-Struktur auf relationale Tabellen und umgekehrt abbildet. Die schemaabhängigen Speicherverfahren zeichnen sich dadurch aus, dass sie unterschiedliche XML-Daten auch auf unterschiedliche Tabellenstrukturen abbilden. Dadurch kann für jedes Dokument eine optimale Darstellung gewählt werden.

Um dennoch mit einheitlichen Schnittstellen auf die Daten zurückgreifen zu können, wird häufig eine XML-Ansicht (View) oberhalb der Tabellen des RDBMS verwendet[5], auf der dann die Anfragen ausgeführt werden.

Dies ist z.B. bei der Middleware Silkroute [FMS01] der Fall. Hier wird eine virtuelle Baumdarstellung von relationalen Daten mit Hilfe der dazu entwickelten Anfragesprache RXL (Relational to XML transformation Language) erzeugt. RXL kombiniert zu diesem Zweck verschiedene Elemente von SQL und XML-QL[6] miteinander. RXL-Anfragen werden von Silkroute mit Hilfe von heuristischen Plänen in eine oder mehrere SQL-Anfragen transformiert und die Ergebnismengen dann in ein XML-Dokument umgewandelt. Mehr als die Ausgabe der relationalen Daten als XML-Dokument beherrscht das System jedoch nicht, d.h. es existiert keine Abbildung von XML-Daten in eine relationale Struktur.

In [DFS99] wurde mit STORED (Semistructured TO Relational Data) eine deklarative Sprache vorgestellt, welche die Speicherung von semistrukturierten Dokumenten in RDBMS spezifiziert und damit auch XML als Spezialform davon speichern kann. Das relationale Schema wird dabei aus den vorhandenen Daten und eventuell vorliegenden Anfragen generiert und kann bei Bedarf angepasst werden (schemabasierte Speicherung). Die Anfragen und anfallenden Updates werden an Hand des aktuellen Schemas von STORED nach SQL umgewandelt. Der Algorithmus für die Erzeugung des relationalen Schemas verlangt allerdings Wissen über die zu speichernden Daten und das Schema muss bei jeder Speicherung von neuen XML-Dokumenten komplett angepasst und verändert werden, so dass dieser Ansatz für diese Diplomarbeit nicht in Frage kommt.

Das Rainbow-System [ZPR02] benutzt dagegen eine beschränkte Anzahl an relationalen Schemas, die je nach Art der zu speichernden Dokumente ausgewählt werden. Zwischen den relationalen Daten und der virtuellen XML-Sicht für den Benutzer vermittelt eine Algebra (XAT - XML Algebra Tree), mit deren Hilfe Anfragen auf den XML-Daten nach SQL umgewandelt werden. Dazu trennt die Algebra SQL-Operatoren von anderen, nicht in SQL umsetzbaren Anweisungen und optimiert den Algebrabaum nach heuristischen Regeln. So ist das System zwar in der Lage, XQuery-Anfragen zu beantworten, aber Updates sind nicht vorgesehen.

Ein auf Relationen innerhalb des XML-Baumes spezialisierter Ansatz, genannt

[5] meist in einer Mittelschicht außerhalb des Datenbanksystems

[6] `XML-QL: A Query Language for XML` war einer der Vorschläge für eine standardisierte XML-Anfragesprache; viele Elemente von XML-QL flossen dann in die Entwicklung von XQuery mit ein

EDGE, wird in [FK99] vorgestellt. Hier werden für alle Knoten der Inhalt und die ausgehenden Kanten gespeichert. Je nach Ausprägung geschieht dies in einer großen Tabelle oder in jeweils einer kleineren Tabelle pro vergebener Name. Letzteres kann bei großen und unstrukturierten Dokumenten zu einer nicht sehr performanten Partitionierung der Datenbank führen, wodurch weiterer Aufwand nötig ist, um neue Tabellen anzulegen bzw. leere zu löschen. Andererseits wird das XPath-Modell mit seinen Schritten von Knotenmenge zu Knotenmenge über traversierende Achsen sehr gut abgebildet und Updates wie das Einfügen eines neuen Teilbaumes werden erleichtert, da immer nur einzelne, kleine Knoten-Tabellen bearbeitet werden müssen. Allerdings müssen bei langen Pfaden oder `descendant`-Anfragen auch sehr viele Tabellen durchlaufen werden, was zu schlechter Anfragenperformanz führen kann.

Ein anderes Problem ist das Speichern des Inhaltes als Text zusammen mit den Kanten, da Textknoten beliebig lang sein können und es nötig ist, den Typ des Inhaltes zusätzlich zu speichern. Dafür bieten die Autoren zwei Lösungen an: Mit separaten Wertetabellen pro Inhaltstyp können die Inhalte zumindest außerhalb der oft abgefragten Knotentabelle(n) gespeichert werden, während beim Werte-Inling für jeden möglichen Wertetyp eine eigene Spalte angelegt wird und der nicht zutreffende Typ mit NULL belegt wird. Der letztgenannte Ansatz hat sich aufgrund der Fähigkeit aktueller RDBMS, NULL-Werte effizient zu speichern, in den Tests der Autoren als die perfomantere Variante erwiesen.

Auch im MONET-Schema [SKWW00] stehen die Beziehungen zwischen den Knoten im Vordergrund. Dazu wird ein vereinfachtes XML-Datenmodell, der Syntaxbaum, als Abbild dieser Beziehungen eingeführt. Jeder Knoten bekommt darin eine eindeutige ID zugeordnet, welche es ermöglicht, jede Kante eindeutig zu identifizieren.

Der Baum kennt genau wie DTDs ausschließlich Zeichenketten als Datentypen. Er wird vor der Speicherung mit der MONET-Transformation in ein relationales Schema überführt. Dazu wird für jeden möglichen Knoten im Baum eine eigene Tabelle angelegt und dort dann jede ausgehende Kante gespeichert. Außer der Kante Knoten-Knoten (für `parent-child`-Beziehungen) werden so auch Knoten-Attribut und Knoten-Textinhalt als Relationen abgelegt. Die natürliche Dokumentenordnung bleibt dabei in jeder Relation gewahrt.

Bei der Performanz für Anfragen gelten ähnliche Einschränkungen wie bei dem EDGE-Ansatz: Lange Pfade und `descendant`-Anfragen verlangen nach einem Durchlauf von vielen Tabellen, in denen in komplex strukturierten XML-Dokumenten

durchschnittlich nur wenige Datensätze enthalten sind. Auf der anderen Seite beschleunigt diese Tatsache die bei Pfad-Anfragen sehr häufig auftretenden Joins zwischen den Tabellen. Bei den Updates entsteht jedoch ein zusätzlicher Aufwand durch das Anlegen bzw. Löschen von vielen Tabellen.

In [MFK01] schließlich beschreiben die Autoren das AGORA-System, welches Ansätze aus schemaabhängigen Verfahren mit einem festen relationalen Schema verbindet. AGORA speichert dazu alle XML-Daten in einem Schema, kann bei Vorlage einer DTD jedoch zusätzlich optimierte XML-Ansichten für einzelne Dokumente generieren. Diese beschleunigen auf den betroffenen Dokumenten die für Anfragen implementierte XQuery-Schnittstelle.

Auf relationaler Ebene gibt es für jeden Knotentyp (Element, Attribut, etc.) eigene Tabellen. Zudem werden `parent-child`-Beziehungen gespeichert und es gibt eine `descendant`-Tabelle für beschleunigte Anfragen auf dieser Achse, die sonst aufgrund vieler benötigter Joins nicht sehr perfomant ausgeführt werden kann. Um die Ordnung der Elemente innerhalb des XML-Dokumentes zu erhalten, werden feste, durchnummerierte Element-IDs vergeben. Diese erhöhen jedoch die Komplexität von Updates, da bei dem Einfügen von Teilbäumen möglicherweise in großen Teilen des Dokumentes die ID geändert werden muss.

2.3.2. Schemaunabhängige Speicherverfahren

Ein zweiter Ansatz für die Speicherung von XML-Daten in relationalen Tabellen sind die schemaunabhängigen Verfahren. Diese ermöglichen es, jegliche XML-Daten zu speichern, da kein Schema für die Erstellung des relationalen Datenbankschemas benötigt wird. Zudem können Streaming-Daten direkt in die Datenbank eingelesen werden, ohne vorher das gesamte Dokument zwischenzuspeichern und analysieren zu müssen.

Die zu speichernden Daten müssen dafür in ein festes Relationenschema überführt werden, welches es erlaubt, die Strukturinformationen der XML-Dokumente zu erhalten - sonst wäre die Wiederherstellung nicht möglich. Zusätzlich zu den zwingend benötigten Vater-Kind-Beziehungen können aber noch mehr Informationen aus den Daten gewonnen werden, wie zum Beispiel Bereichsinformationen für jeden Knoten.

Diese ergeben sich, indem beim Lesen des Dokumentes die Knoten in der Reihenfolge des Auftretens in Dokumentenordnung nummeriert und ihre Reichweite vom einleitenden bis zum abschließenden Tag notiert wird. In der einfachsten Form vergibt der Parser dabei für jeden gelesenen Tag eine eindeutige, linear aufsteigende

Knoten	*Starttag*	*Endtag*	*Bereich*
`rezept`	1	16	15
`zutat`	2	5	3
200g Mehl ...	3	4	1
`anleitung`	6	15	9
Zuerst nehmen ...	7	8	1
`zutat`	9	12	3
Mehl	10	11	1
und mischen ...	13	14	1

Tabelle 2.2.: Bereichsnummerierung für das XML-Dokument aus Abbildung 2.1

Nummer. Diese IDs für den öffnenden und schließenden Tag eines Elementes werden dann als Start- und Endwerte gespeichert. Alternativ kann auch an Stelle des Wertes für den schließenden Tag die Größe des Bereiches notiert werden, so dass für einen Knoten X gilt:

`Nummer Starttag (`X`) + Bereich (`X`) = Nummer Endtag (`X`)`

In Tabelle 2.2 ist als Beispiel die Nummerierung des Dokumentes aus Abbildung 2.1 gegeben - sowohl mit den Start- und Endtags, als auch mit einer entsprechenden Bereichsangabe.

Die so gewonnenen Daten erlauben einfache `descendant`-Abfragen, da alle nachfolgenden Knoten eines Kontextknotens mit ihren zwei Bereichsnummern zwischen den Nummern des Kontextknotens liegen müssen[7]. Bei einem geeigneten Index auf der relationalen Tabelle müssen so nur noch wenige Knoteneinträge geprüft werden. Im Beispiel 2.2 wäre also `//anleitung` gleichbedeutend mit dem Suchen aller Knoten, welche die Bedingung `Starttag >= 6` und `Endtag <= 15` (die Tagwerte des `anleitung`-Elementes) erfüllen.

Ein Problem dieses Nummerierungsschemas sind jedoch die Updates. Beim Einfügen neuer Knoten in ein bereits nummeriertes Dokument wird die bestehende Ordnung verändert, so dass im Extremfall alle Knoten neu nummeriert werden müssen. Im Beispiel 2.2 müssten beim Einfügen eines weiteren Elementes `zutat` (z.B. für

[7]Die Nummer des Endtags lässt sich immer mit der Formel `Nummer des Starttags + Bereich = Nummer des Endtags` bestimmen, wenn nur Starttag und Bereich gegeben sind

Zucker) hinter der ersten Zutat die Werte aller folgenden Knoten plus des Wurzelknotens `rezept` geändert werden, da sich die folgenden Tags in Dokumentenordnung alle verschieben.

Das INTERVAL ENCODING [DTCÖ03] ist ein Vertreter dieser Bereichsnummerierung, mit dessen Hilfe die Autoren XQuery für Anfragen auf den XML-Daten implementieren. Die Knoten werden hier als Zeichenkette zusammen mit ihren linken und rechten Endpunkten gespeichert. Für jede XQuery-Anfrage erzeugt das System dann dynamische Sichten für die verwendeten Strukturen wie Achsen und Schleifen, so dass die Anfragen nicht auf den kompletten Datentabellen ausgeführt werden müssen. Bei vielen verschiedenen Anfragen entsteht daraus jedoch ein Performanznachteil aufgrund der nicht persistenten View-Generierung. Außerdem hat auch das INTERVAL ENCODING das nummerierungsbedingte Problem mit dem hohen Änderungsaufwand bei Einfügeoperationen.

Für eine schnellere Abarbeitung der Bereichsanfragen speichert [ZNDLL01] zusätzlich zu den zwei Knotennummern noch eine Levelnummer für die jeweilige Tiefe des Elementes im Baum und hält eine Tabelle mit `parent-child`-Beziehungen vor. Schritte über die Vater- oder Kind-Achse werden so bedeutend beschleunigt. Von den Levelnummern profitieren aber auch die `following-sibling`- und `preceding-sibling`-Achse, da die Knotenrelation nicht mehr sequentiell durchlaufen werden muss, um alle Nachbarknoten zu finden.

Zusätzlich schlagen die Autoren einen effizienterer Join[8] für Indexsuchen auf mehreren Spalten vor, der die bei der Umsetzung von XPath zu SQL häufig auftretenden Tabellen-Joins beschleunigen soll, und die Verwendung von invertierten Listen aus dem Information Retrieval für die Optimierung von `contains()`-Anfragen. Diese Listen werden ebenfalls in der Datenbank gespeichert und können so von der höheren Performanz des vorgeschlagenen MPMGJN profitieren. Trotz der auf diese Weise beschleunigten Anfragen bleiben jedoch die schon benannten Probleme mit den Aktualisierungen der Bereichsnummerierung bestehen.

Der XREL-Ansatz [TVBSSZ02] dagegen versucht, mit der zusätzlichen Speicherung von Pfaden XPath-Anfragen zu beschleunigen. Grundsätzlich ist dieser Ansatz aber ein Vertreter der Bereichsspeicherung und hält diese Daten zusammen mit der Nummer des dazu gehörenden Pfades für Elemente, Attribute und Textknoten in jeweils eigenen Tabellen vor. In einer zusätzlichen Pfadtabelle speichert XREL dann die einfachen Pfade (bestehen nur aus der `child`- und `attribute`-Achse) von der

[8] MPMGJN - Multi-Predicate Merge Join

Wurzel bis zum Blatt für jeden einzelnen Knoten im XML-Baum.

Damit kann z.B. die Anfrage `//` auf der `descendant`-Achse durch den Stringvergleich `LIKE '/%/'` in SQL[9] auf der Pfadtabelle abgekürzt werden. Dabei werden die üblicherweise bei einer großen Anzahl von XPath-Schritten anfallenden vielen Tabellen-Joins vermieden. Treten in Anfragen aber Prädikate oder andere Achsen auf, wird der XPath-Ausdruck in einfache und komplexe Pfade zerlegt und für den komplexeren Teil auf die Bereichsinformationen, hier genannt Regionen, zurückgegriffen.

Im Vergleich der Autoren mit dem EDGE-Ansatz [FK99] auf sehr flachen XML-Dokumenten kann XREL dann auch bei mehreren `descendant`-Schritten oder generell XPath-Ausdrücken mit vielen Schritten die Anfragen deutlich beschleunigen. Sind die XPath-Ausdrücke jedoch komplexer oder bestehen aus mehreren Prädikaten, lässt sich kein Vorteil mehr aus der Pfadspeicherung gewinnen und die Anfragen werden sogar langsamer bearbeitet. Zudem kann bei sehr tief verschachtelten Dokumenten die Pfadlänge extrem wachsen.

Zusammen mit der Voraussetzung, dass die Datenbank besonders effektiv String-Vergleiche durchführen können bzw. einen speziellen Textindex besitzen muss, um die SQL-Anfragen auf der Pfadtabelle effizient auszuführen, eignet sich dieser Ansatz also nur in Spezialfällen oder als optionale Optimierung von bestimmten, einfachen Anfragen.

Die Autoren von [LM01] versuchen das Problem der Neunummerierung bei Updates zu lösen, indem sie beim Einfügen der Dokumente und der damit verbundenen ersten Nummerierung Freiraum lassen für später einzufügende Knoten. Sie verwenden dazu das Schema mit der expliziten Angabe des Bereiches, den sie dann bewusst großzügig wählen. Die eindeutige Nummer nennen sie *`order`*, den Bereich *`size`*, so dass für jeden Knoten ein Wertepaar `<`*`order,size`*`>` gespeichert wird.

In der Datenbank werden die Knoten dann in Tabellen für Elemente, Attribute, Strukturen, Namen und Werte gespeichert. Die Namentabelle enthält die Bezeichner für Knoten und Attribute, die Wertetabelle ihre Inhalte. Elemente und Attribute werden zusammen mit ihrem Wertepaar `<`*`order,size`*`>` und den Verweisen auf Namen und Inhalt in eigenen Relationen abgelegt. Die Strukturtabelle ist der wichtigste Teil des Schemas. Hier finden sich alle Elemente und Attribute wieder, zusammen mit den *`order`*-Nummern ihrer Vaterknoten, ersten Kind-, Nachbar- und Attri-

[9] XREL speichert die Pfade in einer leicht angepassten XPath-Syntax, um falsche Suchergebnisse zu vermeiden

butknoten. Mit der Hilfe dieser Strukturen können die Anfragen auf den `parent`-, `attribute`- und `sibling`-Achsen beantwortet werden.

Für eine begrenzte Anzahl an Einfügeoperationen bietet dieses Verfahren eine Lösung für das Updateproblem. Sobald jedoch größere Mengen Knoten (z.B. ein großer Teilbaum) eingefügt werden kann es passieren, dass nicht genügend Platz in dem entsprechenden Bereich zur Verfügung steht[10]. Das grundsätzliche Problem, dass die Nummern zweier aufeinander folgender Knoten einen festen Abstand haben und sich dazwischen nur eine begrenzte Anzahl an neuen Knoten einordnen lässt, bleibt also bestehen.

In [GRUST02] wird die XML-Baumstruktur mit *preorder*-Nummern (Knoten in der Reihenfolge der Tiefensuche) und *postorder*-Nummern (Breitensuche) auf einen zweidimensionalen R-Baum[11] abgebildet. Unterstützt das Datenbanksystem keine R-Bäume, können alternativ auch B*-Bäume genutzt werden, die allerdings eine schlechtere Performanz aufweisen.

Für alle Achsen kann über die *preorder*/*postorder*-Nummern des Referenzknotens ein Bereich (Fenster genannt, da zweidimensional) angegeben werden, der die gewünschten Knoten des XPath-Schrittes enthält. Bei der Bereichssuche (Spatial Search) nach Knoten in diesen Fenstern können dann die Vorteile des R-Baumes auf diesem Gebiet genutzt werden. Dies beschleunigt vor allem die `parent`- und `ancestor`-Achsen, die in Tests unabhängig von der Pfadlänge und der Dokumentengröße fast keine Zeit bei Anfragen verbrauchten.

Mit einem speziellen, in das RDBMS eingebauten Join-Algorithmus (Staircase-Join) [GKT03], der das Wissen über die Baumstruktur der XML-Daten nutzt, kann die Performanz weiter erhöht werden. Aber auch bei dieser Variante der Bereichsnummerierung bleibt der Nachteil betreffend der Updates erhalten, die die Aktualisierung der Werte einer großen Anzahl von Knoten bedeuten können und damit auch des R-Baum-Indexes.

Das in [TVBSSZ02] vorgestellte Dewey Order versucht das Problem der Updates auf eine andere Art zu lösen, indem es die Bereichsinformationen in nur einer einzigen Nummer ablegt. Die Position eines Knotens innerhalb des Baumes wird dabei unter Ausnutzung des UTF-8-Schemas[12] in diese Nummer kodiert. Dazu haben al-

[10] Alle Nachfolger eines Knotens müssen in dessen Bereich liegen. Ist dieser z.B. mit 100 gewählt und werden mehr als 100 Nachfolger eingefügt, bedarf es einer Neunummerierung

[11] R-Baum: Eine räumliche Indexstruktur, die die schnelle Suche in mehrdimensionalen Daten erlaubt

[12] die maximale Länge der Positonsnummer kann durch Anhängen von neuen Ziffern beliebig verlängert werden

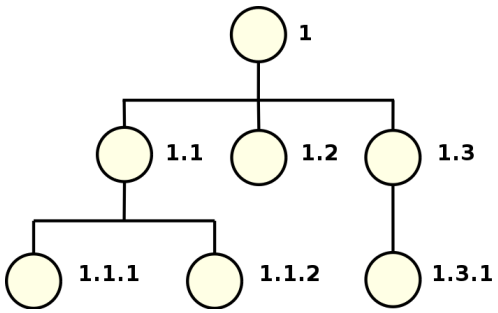

Abbildung 2.3.: Nummerierung eines XML-Baumes mit Dewey Order: Die Knoten auf jeder Ebene mit demselben Vaterelement werden der Reihe nach durchnummeriert.

le Knoten des Baumes die Nummer des Vaterknotens als Präfix gefolgt von einer Nummer für die aktuelle Ebene, die die lokale Reihenfolge innerhalb der Menge der Kinderknoten zu dem Vaterknoten angibt (Abbildung 2.3).

Die Speicherung im UTF-8-Schema erlaubt es, Teilbäume hinter jedem letzten Kindelement anzuhängen, ohne bestehende Knoten neu nummerieren zu müssen. Allerdings ist das Einfügen von neuen Kindelementen an einer anderen Position als der letzten immer noch mit der Umbenennung der folgenden Knoten auf dieser Ebene (und deren Nachfolgern) verbunden.

Auf relationaler Ebene werden in einer Tabelle die Pfade zu den einzelnen Knoten und in einer zweiten Tabelle die Knoten selbst mit ihrer Dewey Order ID, der Pfad-ID und dem Wert (Inhalt) gespeichert. Bei Vorgabe eines Schemas können die Elemente auf mehrere dem Schema entsprechende Tabellen partitioniert werden, wodurch aufgrund der kleineren Relationengrößen Joins beschleunigt werden und damit auch die Queryausführung.

XPath-Anfragen profitieren von den kodierten Informationen in der Dewey Order ID, da über Präfix-Vergleiche die in der Knotennummer kodierten Strukturinformationen erschlossen werden können. Alle Nachfolger eines Knotens (`//`) haben zum Beispiel dessen Nummer als Präfix, alle rechten Nachbarn (`following-sibling`) haben einen höheren Ebenenwert bei gleichem Präfix. Auch Prädikate wie `[2]` bzw. `[position() = 2]` sind durch die vorhandene Speicherung der Knotennummern von Nachbarknoten in der Reihenfolge des Auftretens einfach zu bestimmen.

Das ORDPATH [OOPCSW04] getaufte Nummerierungsschema des Microsoft SQL Server verfolgt bei der Erstellung der Knoten-ID ebenfalls das bei Dewey Order verwendete Konzept mit der vorangestellten Vater-ID gefolgt von einem Levelwert.

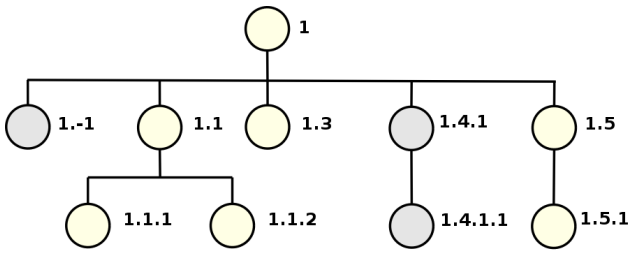

Abbildung 2.4.: Einfügen eines neuen Teilbaumes mit der Hilfe von Einschaltungszeichen und eines einzelnen Elementes als ersten Nachbarknoten in einen mit ORDPATH nummerierten XML-Baum.

Allerdings werden für Knoten auf einer Ebene nur ungerade Nummern vergeben (1, 3, 5, ...)[13], so dass zwischen zwei Knoten immer noch Platz für das Einfügen eines neuen Teilbaumes bei einem Update ist (2, 4, 6, ...). Diese geraden Werte werden allerdings nicht als Positionswerte angesehen, sondern als Einschaltungszeichen (carets) bezeichnet und in der Hierarchie nicht mitgezählt. Die einzufügenden Knoten erscheinen dann als 'Kinder' dieser Einschaltungszeichen (wieder mit ungerader Nummer), befinden sich logisch jedoch zwischen den linken und rechten Nachbarn der geraden ID[14]. Dadurch können beliebig viele neue Knoten an jeder Stelle im Baum eingefügt werden, ohne dass bereits vergebene IDs verändert und bestehende Datensätze angepasst werden müssten.

Um als Beispiel in den Baum aus Abbildung 2.3 ein neues Element vor dem Knoten 1.1 einzufügen, reicht es, einen negativen Levelwert zu verwenden (1.-1). Beim Einfügen eines neuen Wertes zwischen 1.3 und 1.5 muss jedoch auf ein Einschaltungszeichen zurückgegriffen werden; der neue Knoten erhält dann die ID 1.4.1 (siehe Abbildung 2.4).

ORDPATH-IDs werden in der Datenbank als variabel lange, binäre Zeichenketten gespeichert. Jeder Levelwert besteht dabei aus einem Teil L_i, der die Anzahl an Bits festlegt, mit der der folgende, eigentliche Ebenenwert O_i kodiert wird. Die Werte von L_i sind dabei als Pfade in einem Binärbaumes präfixfrei kodiert, so dass die Länge von L_i eindeutig bestimmt werden kann und damit auch die Länge von O_i. Obwohl die geraden Nummern als Einschaltungszeichen nicht als logische Ebenen angesehen

[13]Diese Nummern können auch negative Werte annehmen, so dass das Einfügen beliebig vieler Werte vor dem ersten Knoten einer Reihe von Nachbarknoten kein Problem mehr darstellt

[14]Durch die Bedingung, dass der letzte Wert einer ORDPATH-ID immer ungerade ist, ist garantiert, dass die ID auf eine 1 endet und damit ihre Länge leicht bestimmt werden kann

werden, können sie bei der binären Darstellung doch ebenso kodiert werden (da $1.3 < 1.4.* < 1.5$). Eine komplette ID als Aneinanderreihung von allen Levelwerten $0, 1, \ldots, i$ von der Wurzel bis zum Kontextknoten sieht damit wie folgt aus:

L_0	O_0	L_1	O_1	$\ldots$	L_i	O_i

Auf relationaler Ebene werden zusammen mit der ORDPATH-ID für jeden Knoten zusätzlich der Typ und die Ebene plus Verweise auf den Namen und den Inhalt gespeichert, welche in seperaten Tabellen abgelegt werden. Die durch bitweise Vergleiche der IDs entstehende Ordnung ist dank der Anforderungen an die ORDPATH-ID gleichzeitig die Dokumentenordnung, wodurch das Auslesen ganzer Dokumente beschleunigt wird. Für XPath-Anfragen schließlich gelten die gleichen Vorteile wie bei Dewey Order, die sich durch die möglichen Präfix-Vergleiche der IDs ergeben.

Im nächsten Abschnitt wird nun eine andere Weiterentwicklung von Dewey Order vorgestellt, welche auf eine ähnliche Weise wie das parallel entwickelte ORDPATH das Problem angeht, beliebig neue Kinderknoten in einen bereits nummerierten XML-Baum einfügen zu können, ohne eine Neunummerierung der vorhandenen Knoten vornehmen zu müssen.

2.4. Das Nummerierungsschema DLN

2.4.1. Grundlagen

In [BR04] wird mit DLN (Dynamic Level Numbering) eine verbesserte Version des hierarchischen Nummerierungsschemas Dewey Order [TVBSSZ02] vorgestellt. Die Grundlagen bleiben dabei die Gleichen: Jeder Knoten des XML-Baumes bekommt eine innerhalb der Nachbarknoten eindeutige Nummer zugeordnet, welche zudem die Position in der Reihenfolge der Knoten angibt. Außerdem wird die Nummer des Vaterknotens als Präfix vorangestellt, getrennt durch ein Symbol für den Ebenenwechsel (siehe Abbildung 2.3). Dewey Order erlaubt aufgrund seiner geradlinigen Nummerierung jedoch nur, neue Kinderknoten an der letzten Position einzufügen. Dieses Problem behebt DLN durch die sogenannten Zwischenwerte (subvalues), die es erlauben, zwischen zwei Knotennummern eines Baumes jeweils noch eine weitere einzufügen, ohne andere Knoten neu nummerieren zu müssen. Dadurch können neue

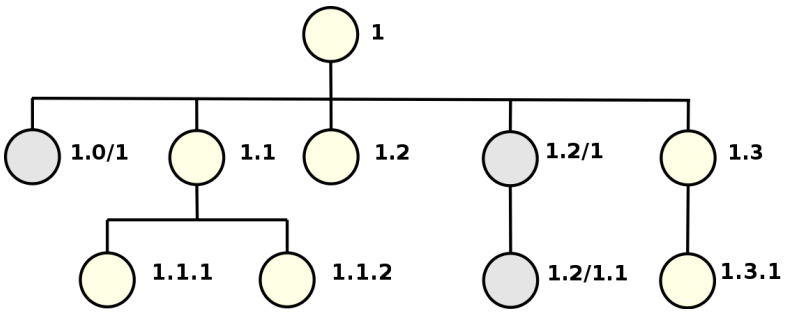

Abbildung 2.5.: Einfügen eines neuen Teilbaumes und eines einzelnen Elementes als ersten Nachbarknoten mit der Hilfe von Zwischenwerten in einen mit DLN nummerierten XML-Baum

Knoten und ganze Teilbäume in bereits in der Datenbank gespeicherte Dokumente eingefügt werden, ohne dass die vorhandenen Daten angepasst werden müssten.

Die Idee dahinter ist dieselbe wie bei den Reellen Zahlen. Zwischen der Zahl 1 und der Zahl 2 kann immer noch eine weitere Zahl durch Anhängen eines Kommas und einer neuen Zahl eingefügt werden, also z.B. $1,1$. Auf dieselbe Art und Weise wird auch bei DLN verfahren. Um zwischen zwei Nachbarknoten a und b auf derselben Ebene das Element c einzufügen, wird an die Nummer von a eine neue Nummer als Suffix angehängt, die durch ein bestimmtes Symbol von der Nummer von a getrennt ist. Dieses Symbol muss größer sein als das Symbol, welches die Ebenen trennt, damit c automatisch größer ist als alle Kinder von a (ein Beispiel ist in Abbildung 2.5 abgebildet mit Punkten als Symbol für den Ebenenwechsel und Schrägstrichen als Symbol für Zwischenwerte).

Eine wichtiger Aspekt für das beliebige Einfügen von Knoten ist dabei, dass die Nummerierung immer mit 1 beginnt. So kann auch vor dem ersten Knoten ein weiterer eingefügt werden durch Verwendung eines Zwischenwertes zwischen 0 und 1. Da dieser wiederum die 1 als Levelwert erhält, können beliebig viele neue Elemente vor dem jeweils ersten Knoten eingefügt werden durch sukzessives Erzeugen von neuen Zwischenwerten, was bei der Verwendung der 0 als erste Zahl nicht möglich wäre.

Sollen die neuen Knoten jedoch nicht an der ersten Stelle eingefügt werden, können auch vorhandene Zwischenwerte weiter genutzt werden. Der Zwischenwert wird dazu einfach weiter hoch gezählt. So passen zwischen den Knoten 1.2/1 und 1.3 aus Abbildung 2.5 beliebig viele Knoten mit den Nummern 1.2/2, 1.2/3, 1.2/4, usw. Die Zwischenwerte verhindern also, dass es zu einem Überlauf der darstellbaren

Nummern kommt.

Für das Konzept der Zwischenwerte gibt es beim Dynamic Level Numbering zwei verschiedene Umsetzungen, die nun vorgestellt werden.

2.4.2. Die DLN-DOM-ID

Für die Speicherung der DLN-Nummern wurde eine binäre Darstellung entwickelt, da aktuelle relationale Datenbanken Integer-Zahlen schneller vergleichen können als Zeichenketten gleicher Länge [BR04]. In der Binärform besteht eine DLN-DOM-Nummer aus einer Reihe von Levelwerten, die durch das Symbol für den Ebenenwechsel 0 voneinander getrennt sind. Jeder Levelwert wiederum besteht aus einer Reihe von Zwischenwerten getrennt durch 1. Die Anzahl der für jeden einzelnen Zwischenwert verwendeten Stellen richtet sich nach einem Array mit Längenabgaben, der nur für einzelne Dokumente oder für die gesamte Datenbank gelten kann. Je nach Aufbau des Dokumentes ist es damit möglich eine Darstellung zu wählen, die die Länge der DLN-DOM-Nummer so klein wie möglich hält[15].

	Subvalue 0	***Subvalue 1***	***Subvalue 2***
Level 1	1	2	
Level 2	2		
Level 3	3	3	2

Tabelle 2.3.: Beispiel für ein Array mit Längenangaben für die Zwischenwerte

Das Array mit den Längenangaben für die Zwischenwerte enthält für die ersten `m` Level[16] eine levelabhängige Anzahl von `n` Werten für die möglichen `n` Zwischenwerte, die jedoch nicht zwangsweise alle auftreten müssen. Für weitere Subvalues größer als `n` wird die Längenangabe des letzten Zwischenwertes (`n`) verwendet. Ebenso übernehmen Levels jenseits der Zeilenanzahl `m` des Längenarrays die Werte des letzten definierten Levels `m`. Im Beispielarray 2.3 würden also alle Levels größer als `3`

[15] Liegt das zu speichernde XML-Dokument bei der Einfügeoperation nicht komplett vor (z.B. als DOM-Baum), sondern ist nur als Stream verfügbar, kann zum Zeitpunkt der Speicherung keine Aussage über die Anzahl von Elementen auf einer Ebene getroffen werden. Dieses Problem behandelt der darauf spezialisierte DLN Streaming Algorithmus, der eine große Anzahl von möglichen Nachbarknoten addressiert, ohne übermäßig lange DLN-Nummern zu produzieren durch fortwährendes Einfügen von Zwischenwerten [BR04]. In Abschnitt 2.4.3 wird diese DLN-Stream-Prefix-ID genauer vorgestellt.

[16] meist reicht die Angabe eines einzigen Levels

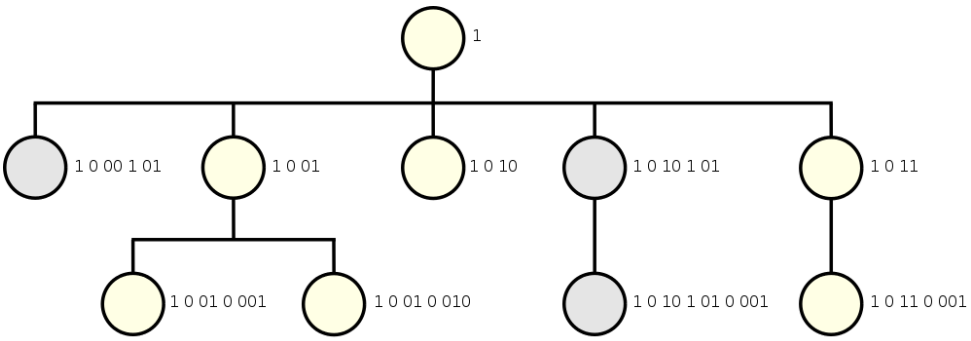

Abbildung 2.6.: Binäre Darstellung eines mit DLN-DOM nummerierten XML-Baumes

die Zwischenwerte des dritten Levels verwenden und weitere Zwischenwerte (sofern benötigt) auf diesem Level die Länge 2 aufweisen.

Als Beispiel für die DLN-Nummerierung folgt nun einer Entwicklung der binären Darstellung des Baumes aus Abbildung 2.5. Der oberste Level des Baumes hat nur einen einzigen Knoten, so dass nur der Subvalue mit der Länge 1 für die Speicherung benötigt wird. Die ursprünglichen Knoten auf Level zwei passen ebenfalls in die zwei Bits des ersten Subvalues dieser Ebene. Für die neueingefügten Knoten wird jedoch ein weiterer Zwischenwert benötigt, der laut dem Längenarray 2.3 auch mit zwei Bits kodiert wird.

So hat Knoten 1 nun die binäre Darstellung 1 und die Knoten eine Ebene tiefer bekommen die Werte 1 $\underline{0}$ 01 bis 1 $\underline{0}$ 11 zugewiesen mit $\underline{0}$ als Trennzeichen zwischen den Levels. Die später hinzugefügten Knoten verwenden dann einen zweiten Subvalue nach dem Trennzeichen $\underline{1}$. 1.2/1 wird damit als 1 $\underline{0}$ 10 $\underline{1}$ 01 kodiert. Analog geht es weiter auf den folgenden Ebenen: 1.1.2 hat die binäre Form 1 $\underline{0}$ 01 $\underline{0}$ 010 und 1.2/1.1 wird von der Bitfolge 1 $\underline{0}$ 10 $\underline{1}$ 01 $\underline{0}$ 001 repräsentiert (siehe Abbildung 2.6).

2.4.3. Die DLN-Prefix-ID

Wenn das zu speichernde XML-Dokument bei der Einfügeoperation nicht komplett vorliegt (z.B. beim Streaming von XML-Daten), kann die Nummerierung der Knoten beim Einfügen in die Datenbank durch den spezialisierten DLN Streaming Algorithmus erfolgen. Dieser ist darauf optimiert, die Länge der IDs auch bei einer großen Anzahl von Nachbarknoten nicht übermäßig wachsen zu lassen, indem die Wertebereiche der Zwischenwerte möglichst optimal genutzt werden.

Dazu ist die Präfix-ID etwas anders aufgebaut als die DOM-ID. Die einzelnen

durch 0 voneinander getrennten Ebenen bestehen nun aus einer Anzahl von Subleveln (an Stelle der Subvalues, aber mit derselben Semantik), die wiederum aus Zwischenwerten bestehen. Sublevels werden durch das Trennzeichen 1 auseinander gehalten und die Anzahl der Zwischenwerte wird durch die Anzahl der führenden Nullen oder Einsen des jeweiligen Sublevels bestimmt. Das in Kapitel 2.4.2 vorgestellte Array gibt auch hier die Länge der einzelnen Zwischenwerte vor.

Der DLN Streaming Algorithmus kann mit diesem Aufbau der DLN-ID nun bei der suksessiven Anforderung von neuen Sublevelnummern automatisiert die Subvalues anpassen, ohne sehr lange und uneffektive IDs durch Benutzung vieler Zwischenwerte zu generieren. Der Algorithmus funktioniert dabei nach folgendem Schema (Beispiele mit einer festen Zwischenwertlänge von 4):

Erhöhen der Sublevelnummer um 1:

(1) der erste Zwischenwert enthält 1-Bits:

Die führenden 1-Bits (gefolgt von einer 0) bestimmen die Anzahl der auf den ersten Zwischenwert folgenden weiteren Zwischenwerte. Ist die Menge der Zwischenwerte ausgeschöpft, wird ein weiterer 0...0-Zwischenwert an den Sublevel angehängt.

```
0001            ... 0111
1000 0000       ... 1011 1111
1100 0000 0000 ... 1101 1111 1111
```

(2) der erste Zwischenwert ist gleich 0 (keine 1-Bits):

Die führenden 0-Bits ab dem zweiten Zwischenwert bestimmen die Anzahl der folgenden Zwischenwerte. Sind alle folgenden Bits mit 1 belegt, wird das letzte führende 0-Bit und der letzte Zwischenwert gestrichen. Mit zunehmender Anzahl von Subleveln wird aber ein Punkt erreicht (0000 1111 zu 0001), wo wieder Fall **(1)** angewendet werden muss.

```
0000 0010 0000 0000 ... 0000 0011 1111 1111
0000 0100 0000       ... 0000 0111 1111
0000 1000            ... 0000 1111
```

Senken der Sublevelnummer um 1:

(1) der erste Zwischenwert ist gleich 0 (keine 1-Bits):

Wenn die erste mit 1 belegte Stelle nach dem ersten Zwischenwert von 1 zu 0 wechselt, wird ein 1...1-Zwischenwert angehängt. Das Einfügen eines Knotens vor 0000 1000 ergibt also die DLN-ID 0000 0111 1111.

(2) der erste Zwischenwert enthält 1-Bits:

Wenn die letzte Stelle der führenden Einsen zu Null wechselt, wird der letzte Zwischenwert entfernt. Das Einfügen eines Knotens vor 1100 0000 0000 ergibt dann die DLN-ID 1011 1111. Mit zunehmender Anzahl von Subleveln wird aber ein Punkt erreicht (0001 zu 0000 1111), wo wieder Fall **(1)** angewendet werden muss.

2.4.4. Vorteile der DLN-Nummerierung

Die DLN-IDs haben ebenso wie bei den ORDPATH-IDs den Vorteil, dass die natürliche Dokumentenordnung der XML-Daten leicht wieder hergestellt werden kann. Dazu müssen alle Nummern der Knoten des Baumes links ausrichtet, nach rechts bis auf die Länge der längsten ID mit Nullen aufgefüllt und danach der Größe nach aufsteigend geordnet werden.

Dies zeigt erneut ein Beispiel an Hand des Baumes aus der Abbildung 2.6 und seinen DLN-DOM-IDs. Die längste binäre Darstellung hat hier die Nummer 1.2/1.1 mit 11 Stellen; alle anderen Nummern werden nach rechts mit Nullen auf diese Anzahl aufgefüllt. Nach der Sortierung der IDs ergibt sich dann die korrekte Dokumentreihenfolge in Tabelle 2.4.

Für das zu DLN gehörende relationale Schema werden alle Hauptknotentypen (Elemente, Textknoten, Kommentare und Processing-Instructions) in einer Tabelle zusammen mit ihren DLN-IDs gespeichert, um eine schnelle Ausgabe des gesamten Dokumentes über die geordneten DLN-IDs zu ermöglichen. Damit diese Tabelle nicht zu groß wird, werden die Inhalte der Textknoten in eine seperate Tabelle ausgelagert; ebenso werden Attribute in einer eigenen Tabelle abgelegt[17]. Zu jedem Hauptknoten speichert das Schema zudem je einen Verweis auf den Vaterknoten

[17] Attribute gehören nicht dem Hauptknotentyp an und passen damit nicht in das Vater-Kind-Schema der sonstigen Knoten

binäre DLN-Darstellung	***DLN-ID als Zeichenkette***
10000000000	1
1$\underline{0}$00$\underline{1}$010000	1.0/1
1$\underline{0}$010000000	1.1
1$\underline{0}$01$\underline{0}$010000	1.1.1
1$\underline{0}$01$\underline{0}$100000	1.1.2
1$\underline{0}$100000000	1.2
1$\underline{0}$10$\underline{1}$010000	1.2/1
1$\underline{0}$10$\underline{1}$01$\underline{0}$001	1.2/1.1
1$\underline{0}$110000000	1.3
1$\underline{0}$11$\underline{0}$001000	1.3.1

Tabelle 2.4.: Erhaltung der Dokumentenordnung durch DLN-IDs

und den in Dokumentenordnung folgenden Nachbarn, um Anfragen nach den Eltern, Kindern oder benachbarten Elementen des Kontextknotens zu beschleunigen (siehe dazu auch Kapitel 3.1.3).

Über die DLN-ID wird auch eine Verbesserung von Anfragen auf der `descendant`- und `ancestor`-Achse erreicht, da diese auf das Bearbeiten der ID bzw. das Suchen in einem minimalen Bereich reduziert werden. Durch die Verwendung der Strukturinformationen in den DLN-IDs profitieren aber auch die Achsen `preceding (-sibling)` und `following (-sibling)`. In Kapitel 3.2.5 wird für jede Achse genau ausgeführt, wie die DLN-ID und das dazu gehörende relationale Schema bei der Umsetzung genutzt werden können.

3. Konzeption und Architektur

Dieses Kapitel beschreibt den Entstehungsprozess und die Konzeption dieser Diplomarbeit. Zuerst wird das Projekt XMLRDB etwas genauer vorgestellt, da es die Grundlage für diese Diplomarbeit bildet. Dazu gehört eine Analyse der Schnittstellen und vorhandenen Vorarbeiten, die zum Teil für diese Diplomarbeit genutzt werden konnten, sowie die Festlegung der genauen Aufgabenstellung und des Rahmens dieser Arbeit.

Die Sektion 3.2 widmet sich dann der Fragestellung, wie ein der Aufgabenstellung entsprechendes XPath-Modul funktionieren soll und auf welche Weise der zentrale Aspekt, die Konvertierung von XPath nach SQL, überhaupt umgesetzt werden kann.

In Kapitel 3.3 schließlich wird in einer Abgrenzung erläutert, welche Abstriche während des Voranschreitens der Arbeit gemacht werden mussten und welche Auswirkungen dies zur Folge hat.

3.1. Das Projekt XMLRDB

3.1.1. Konzept des modularen Mediators

Wie in Kapitel 1.1 schon kurz ausgeführt, hat es sich das XMLRDB-Projekt an der Abteilung Datenbanken der Universität Leipzig zum Ziel gesetzt, ein System zu entwickeln, welches beliebige XML-Daten in einem RDBMS speichern kann. Als Basis für die verlustfreie Speicherung und die Wiederherstellung der Daten sollen dabei die Nummerierungsschemas aus Kapitel 2.3.2 dienen, da sie die Strukturinformationen der XML-Daten auf vergleichbare Art und Weise über Knoten-IDs abbilden. Als Standard wird jedoch DLN aus Kapitel 2.4 eingesetzt, da es ein Ziel des XMLRDB-Projektes ist, die Vorteile dieses Schemas aufzuzeigen.

Um die Anfordungen vom Anfang dieser Diplomarbeit (Kapitel 1) zu erfüllen, wurde eine generische, in JAVA programmierte Zwischenschicht oberhalb der Datenbank geschaffen, welche die Kommunikation zwischen dem Nutzer und der Datenbank

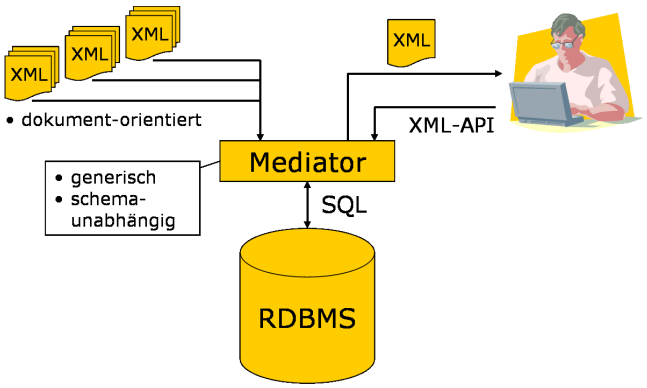

Abbildung 3.1.: Das Konzept des Mediators (aus [B05])

übernimmt und so für die Transformation der XML-Schnittstellen in SQL-Anfragen verantwortlich ist. In der Abbildung 3.1 ist die Arbeitsweise dieses als Mediator bezeichneten Konzeptes zu erkennen: Der Nutzer übergibt der Middleware die zu speichernden Dokumente und kann danach über standardisierte XML-Schnittstellen darauf zurückgreifen, ohne sich über die Art der Speicherung oder der Kommunikation Gedanken machen zu müssen.

Der Mediator verbirgt neben dem Nummerierungsschema auch den genauen Typ der Datenbank, so dass theoretisch alle RDBMS, welche gewisse Grundanforderungen erfüllen, als Unterbau für das XMLRDB-Projekt dienen können. Zu diesen Anforderungen gehört neben der Unterstützung von Subqueries für Existenzbedingungen auch die Möglichkeit, die Nummerierungsschema-IDs per User Defined Functions zu verarbeiten.

Innerhalb des Mediators sind verschiedene Module für die unterschiedlichen Aufgaben zuständig. Für jedes eingesetzte Nummerierungsschema gibt es Implementierungen von vier verschiedenen Interfaces, die einen einheitlichen Zugriff auf die Daten unabhängig von dem real verwendeten Schema erlauben (siehe Abbildung 3.2). Ein `ConnectionManager` sorgt dafür, dass alle auf die Daten zurückgreifenden Bestandteile des Mediators über dieselbe Schnittstelle auf jeder mögliche Datenbank arbeiten können, die von dem Projekt unterstützt wird.

Im Mittelpunkt steht dementsprechend der `SchemaHandler`, welcher die Informationen über das relationale Schema und die Art und Eigenschaften der verwendeten Knoten-IDs verwaltet und den anderen Modulen zur Verfügung stellt. Die für die Kommunikation zuständigen Module benötigen diesen `SchemaHandler`, um die Transformationen der Nutzeranfragen in ein schemaabhängiges SQL durchführen zu

können.

Zuständig für die Verwaltung der gespeicherten Daten ist das `Update`-Modul, welches mit Hilfe des `SchemaHandlers` sowohl neue XML-Knoten in der Datenbank anlegen als auch bestehende Tabelleneinträge aktualisieren oder entfernen kann. Der auf das Einlesen neuer Dokumente spezialisierte `Loader` ruft nach der Zerlegung der übergebenen Daten in einzelne XML-Knoten den `Updater` auf, um die geparsten Daten zu sichern.

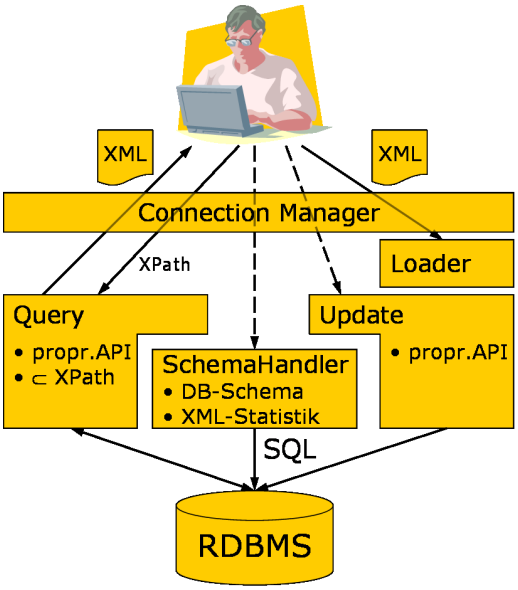

Abbildung 3.2.: Der interne Aufbau des Mediators (aus [B05])

Damit ein Nutzer Anfragen über eine XML-Schnittstelle an die gespeicherten Daten stellen kann, muss ein `Query`-Modul erstellt werden. Dieses ist dafür verantwortlich, mit Hilfe des `SchemaHandlers` die Anfragen an die Schnittstelle nach SQL umzuwandeln und die Ergebnismenge schließlich wieder als XML-Daten aufzubereiten. Die Zielstellung dieser Diplomarbeit ist nun, ein Modul zu entwickeln, welches XPath-Anfragen an den Mediator beantworten kann.

3.1.2. Arbeiten im Vorfeld dieser Diplomarbeit

Das XMLRDB-Projekt enthielt für erste Tests von DLN bereits ein rudimentäres `Query`-Modul namens `DLNQuery`. Für dieses wurde parallel zu dieser Diplomarbeit innerhalb des Projektes ein XPath-Parser entwickelt, welcher aus dem übergebenen String eine objektorientierte Darstellung (XPPath) generiert. Auf Basis dieses Parsers arbeitet das XPath-Optimierungs-Modul aus [B05], so dass bei der Konzeption dieser Diplomarbeit entschieden wurde, den Parser bzw. die optimierte Objektdarstellung der XPath-Anfrage auch für das zu implementierende XPath-Modul zu verwenden.

Während der parallelen Entwicklung des Parsers und des XPath-Moduls wurden auch bestimmte Konventionen festgelegt, um die Schnittstelle des XPPath-Objektes genauer zu spezifizieren. So werden äquivalente Ausdrücke wie

`attribute::attribute(name)` und `attribute::name`

schon vom Parser auf eine feste Darstellungsform reduziert, was eine Erleichterung der Transformation von XPath nach SQL aufgrund einer geringeren Anzahl zu überprüfender Ausdrücke darstellt.

Für die Generierung der Ergebnismenge besaß das `Query`-Modul außerdem bereits Methoden auf einem Pool von `PreparedStatements`, welche zu einer Menge von gegebenen Ergebnisknoten-IDs ein XML-Dokument als Antwortmenge auf die XPath-Query generieren können. Diese Funktionsweise wurde für das neue XPath-Modul übernommen.

Das im nächsten Abschnitt vorgestellte relationale Datenbankschema ist als Bestandteil des Nummerierungsschemas DLN bereits in [BR04] ausgearbeitet worden. Im Rahmen dieser Diplomarbeit wurde es noch um die `doublevalue`-Spalten und eine Dokumententabelle erweitert (siehe Kapitel 3.1.3), um dem geplanten Umfang der XPath-Implementierung gerecht zu werden. Ebenfalls leicht angepasst wurde das `Loader`-Modul, welches bis dahin Knoten vom Typ `namespace`, `comment` und `processing-instruction` nicht verarbeitete. Diese wurden nun in das Schema von DLN integriert, um eine komplette Speicherung der gegebenen XML-Daten zu ermöglichen.

3.1.3. Das relationale Schema

Das zum Nummerierungsschema DLN gehörende relationale Schema (Grafik 3.3) sieht eine zentrale Tabelle `xmlnode` für das Speichern aller Hauptknotentypen vor. Eine Ausnahme bilden die Attributknoten, da diese zwar das zugeordnete Element als den Vaterknoten ansehen, aber formal nicht zu der Menge von dessen Kinderknoten gehören und nur über die `attribute`-Achse von XPath adressierbar sind. Deshalb werden sie separat in der Relation `xmlattr` abgelegt. Die dritte Knotentabelle, `xmltext`, dient zur Speicherung des Inhaltes der Textknoten. Dieser ist in seiner Länge nicht beschränkt und würde die zentrale `xmlnode`-Relation sprengen, weshalb er in einem Textfeld `value` der `xmltext`-Tabelle abgelegt wird. Alle Textknoten erscheinen trotzdem parallel in der Knotentabelle, da sie dem Hauptknotentyp angehören und damit bei typischen Anfragen auf den XPath-Achsen im Kontext berücksichtigt werden müssen. Ein Tabellen-Join zwischen der `xmltext`- und `xmlnode`-Relation ist aber nur notwendig, wenn gezielt nach Textinhalten wie bei der `fn:contains`-Funktion gefragt wird.

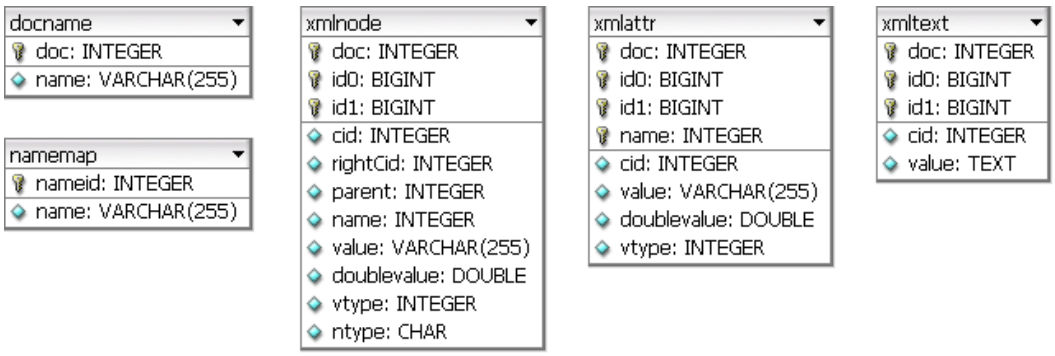

Abbildung 3.3.: Das relationale Schema

Zu jedem Knoten speichert das Schema die Dokumenten-ID und die DLN-ID. Jedes in der Datenbank abgelegte Dokument besitzt einen Eintrag in der Relation `docname`, so dass in anderen Tabellen die als Primärschlüssel auftretende `doc`-ID referenziert kann.

Für die Speicherung der DLN-ID sind zwei BIGINT-Spalten `id0` und `id1` mit je 64 Binärstellen vorgesehen[1], welche nur für das jeweilige Dokument eindeutig sind - deshalb besteht der Primärschlüssel zusätzlich aus der Dokument-ID. Mit insgesamt 128 Stellen sollte die DLN-ID selbst bei der Verwendung von XML-Daten mit einer tiefen, weit verzweigten Baumstruktur nicht so schnell an ihre Grenzen geraten.

Für die Referenzierung eines Knotens ist ein Primärschlüssel aus einer INTEGER und zwei BIGINT-Spalten jedoch zu unhandlich, weshalb zusätzlich eine datenbankweit eindeutige ID namens `cid` vergeben wird. Auf diese ID beziehen sich auch die beiden Spalten `rightCid` und `parent`. Erstere speichert, falls vorhanden, die `cid` des rechten Nachbars eines Knotens oder den Wert `NULL`. Dieser Eintrag ist für die Navigation durch den XML-Baum per DOM sehr nützlich. Die `parent`-Spalte enthält analog die `cid` des Vaterknotens oder `NULL`, wenn sich der Knoten auf der obersten Hierarchieebene befindet[2]. Diese Information ist bei der Umsetzung einiger XPath-Achsen von Vorteil (siehe Kapitel 3.2.5).

Die Namen der Elemente werden wie die Dokumentennamen zentral in einer Tabelle (`namemap`) gespeichert. Darin erscheinen dann alle qualifizierten Namen von At-

[1]Leider unterstützen nicht alle Datenbanken ein Byte-Array. Dieser würde die Verarbeitung der DLN-ID im Gegensatz zu der Abbildung auf große Integer-Zahlen vereinfachen, da er die entsprechende Semantik bereits mitbringt.

[2]Das Nummerierungsschema DLN bildet den Dokumentenknoten nicht explizit ab, weshalb alle Kinder dieses Knotens den Eintrag `NULL` in der `parent`-Spalte besitzen.

tributen und Elementen, welche über XPath angesprochen werden können. Zusätzlich werden in dieser Relation auch die `target`-Werte der Processing-Instruction-Knoten abgelegt. Analog zu der Dokumententabelle erscheint in der `xmlnode`-Relation nur der Primärschlüssel von `namemap` in der Spalte `name`. Bei Knoten ohne eigenen Namen beträgt der Wert des `name`-Feldes -1.

Die eindeutige Erkennung des genauen Knotentyps in der `xmlnode`-Tabelle erfolgt über die `ntype`-Spalte. Elemente sind mit `'e'` gekennzeichnet, Textknoten mit `'t'`, Processing-Instructions mit `'p'` und Kommentare mit `'c'`. Namespace-Knoten werden aufgrund der ähnlichen Syntax (Zugriff nur über eigene Achse, keine Kinder von Elementknoten) wie Attribute behandelt und erscheinen deshalb in der `xmlattr`-Tabelle. Der Namensraum `xmlns` sorgt dafür, dass es zu keinen Kollisionen mit echten Attributen kommt.

Die übrigen Spalten dienen zur Speicherung des Inhaltes von Knoten. Besitzt ein Knoten als Inhalt eine Zeichenkette mit einer maximalen Länge von 255 Zeichen, so wird diese nicht nur als Textknoten, sondern auch in dem VARCHAR-Feld `value` des Elementes abgelegt. Für Prädikate wie z.B. `[url='http://www.uni-leipzig.de']` muss so bei der Beantwortung kein Textknoten überprüft werden. Das Element kann direkt getestet werden und die Anzahl der notwendigen Tabellenjoins verringert sich. Lässt sich der Inhalt eines Knotens dagegen als Zahl parsen, wird er zusätzlich in der Spalte `doublevalue` gespeichert. Beim Verarbeiten von Zahlenvergleichen in Prädikaten (`[einkommen > 100.000]`) kann so das Casting aller relevanten Textinhalte gespart werden, da nur Knoten mit gesetztem `doublevalue`-Feld betrachtet werden müssen.

Dies gilt sowohl für Elemente als auch für Attribute. Kommentare speichern dagegen ihren Inhalt ausschließlich in der `value`-Spalte[3] und Processing-Instruction-Knoten nutzen dieses Feld, um den `data`-Teil abzulegen. Die `target`-Werte erscheinen wie weiter oben erwähnt in der `namemap`-Tabelle und damit als Verweis in der `name`-Spalte.

Die Spalte `vtype` schließlich ist für eine spätere Implementierung des Typmodells von XML Schema und XQuery 1.0 bzw. XPath 2.0 vorgesehen. In diesem Feld kann dann eine Typangabe für die gespeicherten Inhalte der Elemente und Attribute erfolgen.

[3]Kommentare länger als 255 Zeichen werden abgeschnitten

3.2. Konzeptionsphase

3.2.1. Grundlagen der XPath-zu-SQL-Konvertierung

Den wichtigsten Teil dieser Diplomarbeit stellt die Transformation einer XPath-Anfrage nach SQL dar. Das Ziel war, genau eine Datenbank-Abfrage zu erzeugen, welche die Ergebnismenge des XPath-Ausdruckes bestimmt. Damit wird ein Overhead durch die Kommunikation ausgeschlossen und sich auf die Fähigkeiten des RDBMS verlassen, optimal mit Zwischenergebnissen umzugehen. Ein Verarbeiten von großen Mengen an Knotendaten im Mediator würde die Vorteile durch die Nutzung einer relationalen Datenbank wieder zunichte machen. Sobald die Lösungsmenge der XPath-Anfrage dann eindeutig bestimmt ist, können die für den Aufbau eines Ergebnisdokumentes benötigten Knotendaten aus der Datenbank extrahiert und zusammengesetzt werden.

Die Auswahl von Knoten aus einer Kontextmenge durch einen XPath-Schritt geschieht theoretisch analog zu der Bestimmung einer Auswahl von Datensätzen in einer Datenbanktabelle: Mit einer Anzahl von Bedingungen wird die Menge so lange eingeschränkt, bis die gewünschte Auswahl erreicht ist. Allerdings ermöglicht XPath mit dem Konzept der Achsen auch komplexere Bedingungen, welche zum Beispiel die Knotenmenge erweitern können[4]. Deren Umsetzung kann dadurch nur über Tabellenjoins erfolgen, indem die Knotentabelle mit sich selbst vereinigt wird und diejenigen Datensätze auswählt, welche der Auswahl durch die Achsennavigation entsprechen.

Durch diese Analogie ist nun ein Werkzeug gegeben, um verschiedene XPath-Schritte miteinander zu kombinieren und einen längeren Ausdruck umzusetzen. Durch fortwährendes Einschränken der Datensätze der Knotentabelle mittels Knotentests und Prädikaten und einem anschließendem Join mit der gleichen Tabelle für die Auswahl einer neuen Kontextmenge kann am Ende eine Auswahl derjenigen Knotendatensätze erzeugt werden, welche dem XPath-Ausdruck entspricht.

Hierzu folgt ein Beispiel eines typischen XPath-Schrittes in Langform :

```
.../child::kindname[. = 15]/...
```

Dieser Schritt besteht aus der Achse `child`, welche alle Kinder von Knoten aus der Kontextmenge bestimmt, gefolgt von einem Knotentest. Da `kindname` keine der an

[4]Generell erfolgt die Auswahl von Knoten durch einen XPath-Ausdruck aufgrund der XML-Baumstruktur vom Speziellen ins Allgemeine, während bei SQL eine Auswahl durch fortwährende Bedingungen immer weiter eingeschränkt wird.

dieser Stelle erlaubten Funktionen darstellt, handelt es sich um einen Namenstest, d.h. die von der Achse ausgewählten Knoten werden nach Einträgen gefiltert, welche den Bezeichner `kindname` haben. Da die Achse `child` als Hauptknotentyp Elemente enthält, wird nur nach Elementknoten gesucht. Das Prädikat in den eckigen Klammern schränkt diese Auswahl noch so weit ein, dass der Inhalt der gesuchten Elemente gleich der Zahl `15` sein soll.

Die Menge der Kontextknoten wird wie oben beschrieben durch eine Auswahl von Datensätzen auf der Knotentabelle `xmlnode` ausgedrückt. Ein Ausschnitt einer SQL-Anfrage, welche diesen XPath-Schritt abbildet, sieht dann wie folgt aus:

```
SELECT ...
FROM ... xmlnode kontext, xmlnode neu ...
WHERE ... kontext.cid = neu.parent AND neu.ntype = 'e'
      AND neu.name = 7 AND neu.doublevalue = 15 ...
```

Das erste WHERE-Prädikat wählt für die neue Kontextknotenmenge (abgebildet in der Tabelleninstanz `neu`) alle diejenigen Datensätze (== Knoten) aus, welche als Vater einen Verweis auf eine Zeile aus der alten Kontextmenge `kontext` besitzen. Damit ist ein Join zwischen den beiden Tabelleninstanzen durch die Achse `child` durchgeführt worden. Nun folgen weitere Einschränkungen der neuen Kontextmenge: Zuerst wird die Auswahl auf Elementknoten beschränkt, danach auf Zeilen mit dem Namen `kindname`. An dieser Stelle ist schon eine Ersetzung des Bezeichners durch seine ID aus der Tabelle `namemap` vorgenommen worden (dazu mehr in Kapitel 4.4.2). Zum Schluss folgt noch ein Restriktion auf alle diejenigen Knoten, welche als Inhalt einen Zahlenwert besitzen und dieser gleich `15` ist.

Für Anfragen betreffend Attributen oder Textknoten ist es meist nötig, die als Kontextmenge bestimmte Knotenauswahl durch einen weiteren Tabellenjoin mit den Tabellen `xmlattr` oder `xmltext` zu vereinigen. Prädikate mit komplexeren Pfaden als dem Kontext `'.'` bringen außerdem den Nachteil mit sich, bei zusätzlichen Joins die Kontextmenge durch Duplikate zu vervielfachen. Als Beispiel sei der folgende XPath-Schritt genannt:

```
.../child::kindname[kindname]/...
```

Das Prädikat `[kindname]` überprüft nun die Kontextmenge nach der Existenz mindestens eines Kindknotens mit dem Bezeichner `kindname`. Dies hätte nach obigem Schema folgenden SQL-Ausdruck zur Folge:

```
SELECT ...
FROM ... xmlnode kontext, xmlnode neu, xmlnode predicate ...
WHERE ... kontext.cid = neu.parent AND neu.ntype = 'e'
      AND neu.name = 7 AND neu.cid = predicate.parent
      AND predicate.type = 'e' AND predicate.name = 7 ...
```

Dabei würde wie von SQL gewohnt für jeden existierenden Kindknoten mit dem Bezeichner `kindname` eine Zeile in der Auswahl angelegt, so dass die Knoten aus der Tabelleninstanz `neu` mehr als einmal in der Auswahl auftauchen könnten. Da die Tabelleninstanz `neu` aber die Kontextmenge für einen eventuell folgenden Schritt darstellt, ist dieses Verhalten nicht erwünscht.

Einen Ausweg bieten Subqueries mit einer EXIST-Bedingung. Alle einschränkenden XPath-Prädikate, welche nur über einen Tabellenjoin und nicht als `1:1`-Beziehung angebunden werden können, müssen in diese Subquery verschoben werden, wodurch eine Vervielfachung der Kontextmenge durch Duplikate ausgeschlossen ist. Das Ergebnis sieht dann wie folgt aus:

```
SELECT ...
FROM ... xmlnode kontext, xmlnode neu ...
WHERE ... kontext.cid = neu.parent AND neu.ntype = 'e'
      AND neu.name = 7 AND EXISTS
      (
         SELECT *
         FROM xmlnode predicate
         WHERE neu.cid = predicate.parent AND
               predicate.type = 'e' AND
               predicate.name = 7
      ) ...
```

Auf diese Art und Weise können XPath-Ausdrücke also Schritt für Schritt in SQL umgesetzt werden. Ausgegangen wird zu Beginn einer XPath-Anfrage immer von der Kollektion der gespeicherten Dokumente, d.h. die erste Kontextmenge enthält alle Dokumentenknoten. Da diese im DLN-Schema nicht abgebildet werden, finden sich die Knoten auch nicht in der Datenbank wieder. Es können aber alle Knoten auf der obersten Ebene eines Dokumentes mit `parent = -1` bestimmt werden, so dass

Anfragen mit absoluten Pfaden oder einem ersten `//`-Schritt trotzdem übersetzt werden können.

Allerdings lassen sich nicht alle Ausdrücke, Prädikate und Achsen so einfach konvertieren wie in dem gezeigten Beispiel; bestimmte Aspekte der XPath-Anfragesprache sind auf diese Weise gar nicht umsetzbar[5]. Und wie sich schon an der Umsetzung der `child`-Achse gezeigt hat, hängt die Transformation von XPath nach SQL stark von dem verwendeten Nummerierungsschema und dessen relationalen Schema ab.

Aus diesem Grund wurde bei der Konzeption dieser Diplomarbeit beschlossen, bei der Konvertierung einen Zwischenschritt einzulegen und die grundsätzliche Übersetzung nach SQL von den schema- und datenbankabhängigen Teilen zu trennen. Den gesamten Ablauf der Transformation von XPath nach SQL werden die nächsten Abschnitte genau erläutern.

3.2.2. Übersetzung eines XPath-Ausdruckes nach SQL

Die Erzeugung einer SQL-Query aus einer gegebenen XPath-Anfrage erfolgt in mehreren Schritten (siehe Abbildung 3.4). Zuerst wird die auf den Daten auszuführende XPath-Anfrage als String an den XPath-Parser übergeben, welcher daraus eine objektorientierte Darstellung, das `XPPath`-Objekt, generiert. Dieses bildet die ausführliche Form der XPath-Anfrage ab und besteht aus einer Liste von XPath-Schritten (`XPStep`), wobei für jeden Schritt wieder auf seine Bestandteile zurückgegriffen werden kann[6].

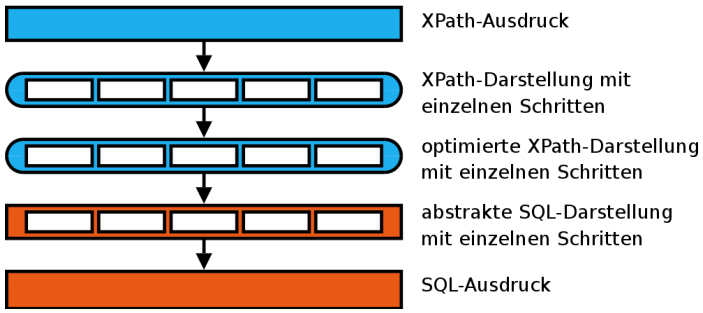

Abbildung 3.4.: Verarbeitung eines XPath-Ausdruckes

[5] die damit verbundenen Einschränkungen betreffen hauptsächlich die fehlende Unterstützung von Schleifen (siehe auch Kapitel 3.3)

[6] bei einem Achsenschritt sind dies z.B. Achsenangabe, Knotentest und Prädikate, bei einem Funktionsschritt der Funktionsname und die Argumente

Auf diesem `XPPath`-Objekt können nun verschiedene im XMLRDB-Projekt getestete Optimierungen stattfinden (Umstellung der Anfrage, Ersetzung von Schritten durch Views und Pfadtabellen, siehe [B05]), bevor die endgültige Konvertierung in eine SQL-Anfrage erfolgt. Für diese Umformung wurde der Umweg über eine abstrakte, ebenfalls objektorientierte SQL-Darstellung (`SQLQuery`-Objekt) gewählt. Diese arbeitet auf einem allgemeinen Relationenschema und ist deshalb nicht auf das Nummerierungsschema DLN beschränkt. Die gemeinsame Grundlage aller damit abbildbaren Speichermethoden von XML-Daten in RDBMS bildet die Kennzeichnung von Knoten eines XML-Baumes mit Hilfe einer Bereichsnummerierung (siehe Kapitel 2.3.2).

Ebenfalls abstrahiert wird in der objektorientierten SQL-Darstellung die Umsetzung für eine spezielle Datenbank. Es werden nur Annahmen getroffen über die Mindestanforderungen der zu nutzenden RDBMS. Neben der Unterstützung von Subqueries und Views betrifft dies vor allem die Möglichkeit, UDFs (User Defined Functions) zu erstellen, da diese für die Extraktion der Strukturinformationen aus den Nummerierungsschema-IDs gebraucht werden.

Mit dieser Zwischenversion der SQL-Anfrage ist es nun möglich, verschiedene Nummerierungsschemas zu verwenden, ohne jeweils eine komplettes XMLRDB--Modul zur Konvertierung von XPath nach SQL zu entwickeln. Stattdessen kann auf die abstrakte SQL-Darstellung zurückgegeriffen werden und nur die nummerierungs- und datenbankabhängigen Teile müssen für die Nutzung eines neuen Schemas oder eines neuen RDBMS angepasst werden.

Das `SQLQuery`-Objekt selber besteht aus einer Reihe von SQL-Prädikaten, welche wie in Kapitel 3.2.1 beschrieben meist aus den einzelnen XPath-Schritten generiert wurden. Zum Schluss der Verarbeitungs-Pipeline wird aus dieser Liste von SQL-Prädikaten wieder ein String erzeugt, der die finale SQL-Anfrage enthält. Dazu werden die noch fehlenden schema- und datenbankabhängigen Teile der XPath-Anfrage übersetzt und die Objektdarstellung der abstrakten SQLQuery in die Textform überführt.

Der erzeugte SQL-String ist nun angepasst an das verwendete Nummerierungsschema und die genutze Datenbank und kann deshalb direkt an selbige übergeben und dort ausgeführt werden.

3.2.3. Die abstrakte SQL-Darstellung

Damit durch die als Zwischenergebnis erzeugte abstrakte SQL-Darstellung alle in Kapitel 2.3.2 beschriebenen Nummerierungsschemas abgedeckt werden, wurde versucht, die verwendeten relationalen Schemas auf ein Einziges abzubilden. Dieses geht von nur einer Tabelle `SQLNode` aus, in der alle Knotentypen gemeinsam gespeichert werden. Zu jedem Knoten ist bekannt, welchem Typ er angehört und welchen Wert und Namen er besitzt. Eindeutig identifiziert wird jeder Eintrag in `SQLNode` über die Angabe des Dokumentes und einer ID, welche für jeden Knoten durch das jeweilige Nummerierungsschema erzeugt wird.

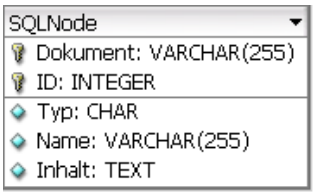

Abbildung 3.5.: Die Tabelle SQLNode

Dieses Schema geht weiterhin davon aus, dass alle Strukturinformationen aus der ID des Knotens abgelesen werden können. Deshalb sind die Achsen in der abstrakten SQL-Darstellung noch nicht umgesetzt und werden als spezielle SQL-Prädikate weitergereicht, da sie auf der Struktur des XML-Baumes navigieren. Ansonsten werden aber alle Elemente der XPath-Anfrage nach diesem Schema konvertiert, um eine einheitliche Form zu erreichen.

3.2.4. Die spezielle SQL-Darstellung

Für das im XMLRDB-Projekt genutzte Nummerierungsschema DLN soll die abstrakte SQL-Darstellung nun in eine fertige SQL-Anfrage umgewandelt werden, welche auf der verwendeten Datenbank ausführbar ist und möglichst effektiv deren Funktionen nutzt.

Ein wichtiger Aspekt ist dabei die Transformation der XPath-Achsen nach SQL. In einem RDBMS gibt es keine vergleichbaren Konzepte zu einem XML-Baum und der Navigation über Achsen darauf, weshalb diese Strukturen mit Hilfe des relationalen Schemas und der DLN-ID abgebildet werden müssen. Im Abschnitt 3.2.5 wird genau beschreiben, wie jede Achse mit Hilfe des Nummerierungsschemas umgesetzt werden kann.

Die Konvertierung des abstrakten Schemas auf das tatsächlich verwendete relationale Schema ist dagegen recht einfach. Die Tabelle `SQLNode` wird je nach verwendeten Knotentyp auf die Tabellen `xmlnode`, `xmlattr` und `xmltext` gemappt, wobei bei letzteren beiden der Knotentyp wegfällt und Textknoten auch keine Namen besitzen. Die restlichen Attribute von `SQLNode` sind auch im relationalen Schema von DLN enthalten, wenn auch das Dokument durch eine ID und die Knoten-ID durch zwei DLN-IDs repräsentiert werden.

In diesem Teil der Transformation von XPath nach SQL wird auch erkannt, ob es sich bei den Vergleichen mit einem Knoteninhalt um Zahlen- oder Stringtypen handelt und welche Spalte dementsprechend gewählt werden muss. Die Anpassungen an die jeweilige Datenbank schließlich, welche ebenfalls hier erfolgen, werden in Kapitel 4 ausführlich beschreiben, so dass sich an dieser Stelle weiter auf die Theorie hinter der Umsetzung beschränkt wird.

3.2.5. Umsetzung der XPath-Achsen mit Hilfe von DLN

Mit Hilfe des Nummerierungsschemas DLN aus Kapitel 2.4 lassen sich die in XPath definierten Achsen (siehe Tabelle 2.1) unter Ausnutzung der durch die IDs gegebenen Strukturinformationen beschreiben. Da diese DLN-IDs immer nur in einem Dokument eindeutig sind, wird die Einschränkung der gesuchten Knoten auf das gewünschte Dokument bei der Navigation über eine XPath-Achse vorausgesetzt. Im Kontext von SQL ist dies aufgrund der datensatzweisen Verarbeitung auch bei Verwendung von `fn:collection()` kein Problem - entweder wird die Spalte `cid`, welche global eindeutig ist, für Verweise genutzt, oder aber die Dokumenten-ID wird zusätzlich zu der DLN-ID übernommen. Zusammen sind diese IDs ebenso eindeutig in der Datenbank.

Der Ausgang von einem der folgenden Achsenschritte ist immer die Sequenz der Kontextknoten K, wobei jeweils nur ein Element k gleichzeitig betrachtet wird und alle Ergebnismengen dann zu einer neuen Sequenz vereinigt werden. Jeder nach Bearbeitung der Achse zurückgegebene Knoten wird vorher jedoch noch auf Erfüllung der möglichen Bedingungen aus Knotentests und Prädikaten geprüft - in diesem Abschnitt geht es aber erst einmal um die theoretische Umsetzung der Achsen:

`self:` Diese Achse wählt den Kontextknoten selber aus und liefert deshalb k zurück.

`child:` Das verwendete relationale Schema speichert zu jedem Knoten den dazu gehörenden Vaterknoten, so dass diese Achse alle diejenigen Knoten zurücklie-

fert, die als Vaterknoten den Kontextknoten k eingetragen haben. Diese Variante ist performanter als die alternative Bestimmung des Vaterknotens durch Extraktion des Präfixes wie bei der `ancestor`-Achse, da bei Verwendung eines Indexes auf der `parent`-Spalte das Parsen der ID gespart wird und nur auf einer INTEGER-Spalte verglichen werden muss.

`parent:` Die parent-Achse liefert diejenigen Knoten zurück, die bei dem Kontextknoten k als Vaterknoten eingetragen sind.

`descendant:` Die Bestimmung aller Nachfolger eines Kontextknotens k bei einer Bereichsnummerierung erfolgt über die Bestimmung der Knoten, die in dem Bereich vom öffnenden bis zum schließenden Tag von k liegen (siehe Kapitel 2.3.2). Da DLN den Endtag nicht speichert und die Bestimmung des folgenden Knotens eine vorherige Sortierung in Dokumentenordnung verlangt, wird alternativ das Kind von k mit der größten Nummer als dessen direkter Vorgänger bestimmt - dies erfordert nicht mehr als eine Bearbeitung der DLN-ID des Kontextknotens.

Das maximale Kind von k hat wie alle Nachfolger von k dessen DLN-ID als Präfix gefolgt von einem Ebenentrennzeichen $\underline{0}$. Da über die Anzahl der folgenden Zwischenwerte keine Annahmen getroffen werden können, wird die ID nach rechts bis auf die maximale Länge mit Einsen aufgefüllt, d.h. mit maximalen Zwischenwerten verbunden durch das Subvalues-Trennzeichen $\underline{1}$. Somit ist ausgeschlossen, dass ein Knoten existiert, der k als Vater (und damit als Präfix) hat und größer als dessen maximales Kind ist.

Die `descendant`-Achse eines k liefert also alle diejenigen Knoten zurück, deren DLN-ID größer als die von k ist und kleiner-gleich der ID des maximalen Kindes von k[7]. Für die Bestimmung des Kindes mit der größten Nummer aus der DLN-ID des Kontextknotens muss das verwendete RDBMS um eine User-Defined Function erweitert werden, da entsprechende Funktionen im SQL-Standard nicht vorgesehen sind.

`descendant-or-self:` Diese Achse schließt zusätzlich zu den Nachfolgern des Kontextknotens k auch noch k selber mit ein, so dass sie eine Vereinigung von k mit den `descendant` von k zurückliefert.

`ancestor:` Diese Achse liefert alle direkten Vorfahren eines Kontextknotens k. Auch

[7]Bei Vergleichen von DLN-IDs müssen beide IDs zunächst auf gleiche Länge gebracht werden, d.h. die kürzere ID wird auf die Länge der längeren ID erweitert indem sie nach rechts mit Nullen aufgefüllt wird.

dies wird über eine Bearbeitung der DLN-ID realisiert, da jeder Vorfahre von k als Präfix in dessen ID enthalten ist. Die Präfixe werden bestimmt, indem von der letzten Stelle der ID an rückwärts nach dem nächsten Ebenentrennzeichen 0 gesucht wird. Sollte k auf Level `n` vorkommen, dann ist der Teil seiner ID vom Anfang bis Level `n - 1` der Vater von k und damit dessen erster Vorfahre. Das Verfahren wird dann analog auf dessen ID angewendet und so lange rekursiv fortgeführt, bis der Wurzelknoten erreicht wurde, also die DLN-ID die Länge `0` hat (siehe das Beispiel in Tabelle 3.1).

	DLN-ID
Kontextknoten k	`1 0 10 1 01 0 01`
erster Schritt	`1 0 10 1 01`
zweiter Schritt	`1`

Tabelle 3.1.: Beispiel für die Extraktion der Vorfahren eines Knotens aus seiner DLN-ID

`ancestor-or-self:` Bei dieser Achse wird eine Vereinigung aus den `ancestor`-Knoten von k mit k selber zurückgegeben.

`following-sibling:` Für die Bestimmung der in Dokumentenordnung folgenden Nachbarn des Kontextknotens k wird wiederum der gespeicherte Vaterknoten verwendet. Alle Knoten, welche denselben Vater besitzen wie k und eine größere DLN-ID haben, werden von dieser Achse zurückgegeben.

`following:` Die `following`-Achse profitiert ebenfalls von der Erzeugung der maximalen Kind-ID als Alternative für die Bestimmung des in Dokumentenordnung folgenden Knotens, da alle Knoten mit einer größeren DLN-ID als die des maximalen Kindes des Kontextknotens k nach k selber folgen. Damit beschreibt diese Bedingung die zurückzugebenden Knoten.

`preceding-sibling:` Diese Achse ist das Gegenstück zu `following-sibling` und liefert deshalb alle Knoten, welche denselben Vater besitzen wie das jeweilige k und eine kleinere DLN-ID haben.

`preceding:` Die `preceding`-Achse liefert alle Vorgänger des Kontextknotens k exklusive der direkten Vorgänger von k. Zurückgegeben werden also alle Knoten mit einer kleiner DLN-ID als der des Kontextknotens, abzüglich der von der `ancestor`-Achse spezifizierten Vorgängerknoten von k.

`attribute:` Da Attribute in einer eigenen Tabelle abgelegt und dort unter der DLN-ID des zugeordneten Elementes gespeichert werden, muss diese Tabelle nur nach den IDs des Kontextknotens k durchsucht werden.

3.2.6. Realisierung der Knotentests und Prädikate

Die meisten Prädikate werden über das erläuterte Subquery-Schema realisiert. Im Standardfall `[XPath Operator XPath]` wird dann zuerst geprüft, ob die beiden zu vergleichenden XPath-Ausdrücke den gleichen Rückgabetyp haben[8]. Ist dies gegeben, können die beiden Pfade miteinander verglichen werden. Dazu werden die Pfade innerhalb der Subquery nach SQL konvertiert und zum Schluss die beiden Kontextmengen entsprechend dem Operator zueinander in Relation gesetzt.

Knoten- und Namenstest wie `element(name)` sind aufgrund des relationalen Schemas einfach zu realisieren, da jeder Knoten vom Hauptknotentyp einen Typeintrag in der Spalte `ntype` enthält. Elemente und Processing-Instructions können zusätzlich auf einen Namen getestet werden, welcher als Referenz auf die `namemap`-Tabelle im `name`-Feld abgelegt ist. Für die Attribute-Achse erfolgt ein Join mit der `xmlattr`-Tabelle, die exklusiv nur Attribute enthält, wodurch ein weiterer Test nicht notwendig ist. Aufgrund der Tatsache, dass DLN den Dokumentenknoten nicht explizit nachbildet, ist es jedoch nicht möglich, eine Sequenz über die Funktion `document-node()` auf diesen Knotentyp zu testen.

Weiterhin werden jedoch Positionsprädikate unterstützt. Diese Konstrukte bestehen aus einem optionalen Aufruf der XPath-Funktion `fn:position()` und einer Positionsangabe und erlauben es, bestimmte Teile des geordneten Kontextes auszuwählen. Dieser besteht aus einer Sequenz der gerade verarbeiteten Kontextelemente, von denen jedes eine feste Position im Kontext hat. Eine Positionsangabe kann nun eine Zahlangabe wie `1` für den ersten Knoten sein, aber auch ein Intervall `10 to 20` oder die Funktion `fn:last()`, welche den letzten Knoten des Kontextes auswählt.

Realisiert werden Positionsprädikate über SQL-IN-Ausdrücke mit einer Subquery. Dazu wird der Kontext eines XPath-Schrittes innerhalb der Subquery bestimmt und über ORDER-BY- und LIMIT-Ausdrücke auf die gewünschte Menge eingeschränkt. Dies wird erneut an dem Beispiel aus Abschnitt 3.2.1 gezeigt, in welchem nur das Prädikat geändert wurde:

[8] Dazu wird der letzte Schritt des XPath-Ausdruckes betrachtet - der Datentyp der dort bestimmten Elemente ist auch der globale Rückgabetyp.

```
.../child::kindname[10 to 15]/...
```

wählt alle Kinder des Kontextes aus, welche den Namen `kindname` besitzen. Von dieser neuen Kontextmenge werden jedoch nur die sechs Elemente genommen, welche sich an den Positionen `10` bis `15` des Kontextes befinden[9]. Die neun vorherigen und alle folgenden Elemente werden verworfen. Ist die Kontextmenge nicht groß genug, bleibt die Auswahl leer.

In SQL sieht dieser XPath-Schritt dann wie folgt aus:

```
SELECT ...
FROM ... xmlnode kontext, xmlnode neu ...
WHERE ... xmlnode neu IN
      (
         SELECT predicate.cid
         FROM xmlnode predicate
         WHERE kontext.cid = predicate.parent AND
               predicate.type = 'e' AND
               predicate.name = 7
         ORDER BY predicate.id0 ASC, predicate.id1 ASC
         LIMIT 6 OFFSET 9
      ) ...
```

3.2.7. Umsetzung von XPath-Funktionen

Das in dieser Diplomarbeit zu erstellende XPath-Modul für das XMLRDB-Projekt beschränkt sich auf einige wenige Beispielumsetzungen der wichtigsten XPath-Funktionen (siehe auch Kapitel 3.3). Die folgende Liste stellt nun vor, wie diese Funktionen mit DLN und dessen relationalen Schema umgesetzt werden können.

`fn:collection()`: Diese Funktion stellt den Standardfall einer XPath-Anfrage über einer Kollektion von Dokumenten dar und hat deshalb keine Umsetzung in SQL. Auch ohne `fn:collection()` als ersten Schritt wird zu Beginn keine Einschränkung der gespeicherten Dokumente vorgenommen. Wenn dies gewünscht ist, kann die als nächstes vorgestellte Funktion `fn:doc()` verwendet werden.

[9] Die Reihenfolge der Elemente ist bei Knoten durch die Dokumentenordnung gegeben. Bei rückwärts gerichteten Achsen befinden sich die Knoten aber in umgekehrter Reihenfolge!

`fn:collection()` wird aber nur ohne Argumente unterstützt, da eine Verwaltungsstruktur von Dokumenten über benannte Kollektionen im XMLRDB-Projekt nicht vorgesehen ist.

`fn:doc()`: muss als Argument einen Dokumentennamen enthalten, damit die Funktion ausgeführt werden kann. In der Tabelle `docname` wird dann geprüft, ob dieses Dokument in der Datenbank enthalten ist. Bei einem positiven Ergebnis wird die Knotentabelle auf alle diejenigen Einträge beschränkt, welche die docID des Dokumentes enthalten.

Dies muss nur einmal zu Beginn einer SQL-Anfrage getan werden, denn fortan wird die Auswahl der Kontextmenge immer aus Knoten gebildet, welche mit dem vohergehenden Kontext in einer Beziehung stehen, die über eine Achse bestimmt werden kann. XPath-Achsen verlassen jedoch niemals das gegebene Dokument.

`fn:count()`: Diese XPath-Funktion hat den Vorteil, einen entsprechenden Gegenpart in SQL zu besitzen. Somit ist es ausreichend, den als Parameter gegebenen XPath in eine SQL-Query umzuwandeln und den SELECT-Teil zu ersetzen durch die SQL-Funktion COUNT() über einer Spalte der Ergebnisknotenmenge. Vergleiche in Prädikaten werden über Subqueries nach dem Schema `(SELECT COUNT(*) ...) Operator Value` realisiert.

`fn:contains()`: Dies ist die komplexeste der unterstützten XPath-Funktionen. Eigentlich überprüft `fn:contains()` nur, ob der zweite Parameter ein Teilstring des ersten Parameters ist (Collations werden noch nicht unterstützt). Allerdings können auch Knoten und ganze Pfade übergeben werden, wodurch erst der Stringwert des jeweiligen XPath-Ausdruckes bestimmt werden muss. Bei der Implementierung wurde sich auf die Übergabe von Strings als zweiten Parameter beschränkt. Bei dem ersten Argument kann es sich jedoch auch um Knoten handeln, da deren Stringwerte fest definiert sind:

- für Attribute ist der Stringwert der Inhalt des Knotens, also das `value`-Feld im relationalen Schema
- für Processing-Instructions und Kommentare ist der Stringwert definiert als der Inhalt (content), so dass in diesen Fällen das `value`-Feld der `xmlnode`-Tabelle geprüft wird
- der Stringwert eines Textknoten wird ebenfalls durch den content beschrieben, weshalb die `value`-Spalte der `xmltext`-Tabelle durchsucht wer-

den muss

- der Inhalt eines Elementes schließlich ist komplizierter zu bestimmen. Er besteht aus allen Inhalten der nachfolgenden Textknoten, d.h. der Inhalt des Dokumentenknotens besteht aus allen Textknoten des Dokumentes. Im Idealfall besitzen die nachfolgenden Knoten eines Kontextknotens so wenig Inhalt (maximal 255 Zeichen), dass es ausreichend ist, das `value`-Feld der `xmlnode`-Tabelle zu überprüfen. In allen anderen Fällen müssen die nachfolgenden Textknoten über einen Join auf der `descendant`-Achse mit der `xmltext`-Tabelle bestimmt und jeder davon überprüft werden, ob er den zweiten Parameter von `fn:contains()` enthält.

Je nach Typ des übergebenen Knotens verhält sich `fn:contains()` also anders, was mit einer SQL-OR-Bedingung umgesetzt wird. Ansonsten wird in der abstrakten SQL-Darstellung ein einfaches `LIKE '%value%'` verwendet, welches ja nach Datenbank auch durch effizientere Zugriffstrukturen ersetzt werden kann.

3.3. Abgrenzung

Da es in dieser Diplomarbeit um den Beweis des praktischen Nutzens der Speicherung von XML-Daten in einem RDBMS unter Ausnutzung des Nummerierungsschemas DLN geht, wurden an einigen Stellen Abstraktionen vorgenommen. Die meisten der fehlenden Teile würden den Rahmen dieser Diplomarbeit sprengen, können aber in zukünftigen Arbeiten vervollständigt werden. Andere Limitierungen haben ihre Ursache in der Aufgabenstellung und den gestellten Anforderungen.

So ist das Modul zum Beispiel für XPath-2.0-Ausdrücke vorbereitet, behandelt aber nur XPath-1.0-Anfragen im Kontext von XPath 2.0. Dies bedeutet, dass sich auf Pfadausdrücke beschränkt wird und alle anderen Funktionen der Sprache wie Schleifen vorerst ausgelassen werden. Die Gründe dafür liegen zum Einen darin, dass in SQL das Konzept der Schleifen nicht vorhanden ist. Ansätze wie temporäre Tabellen für die Zwischenergebnisse und Schleifenvariablen würden jedoch bedeuten, die XPath-Anfrage nicht mehr in eine einzelne SQL-Query konvertieren zu können. Da dies eine der Zielstellungen dieser Diplomarbeit ist, wird auf die Umsetzung von Schleifen verzichtet.

Zum Anderen sind Schleifen natürlich auch ein Konstrukt, welches seine Anwendung hauptsächlich in den komplexeren Anfragen von XQuery findet. Da XPath ein

wichtiger Grundstein von XQuery ist, wird auf eine spätere Implementierung dieser Anfragesprache verwiesen, für deren Umsetzung das XPath-Modul einen wichtigen Bestandteil darstellt. Da der im XMLRDB-Projekt entwickelte XPath-Parser zu diesem Zeitpunkt ebenfalls keine Schleifen erkennt, ist so eine Kontinuität innerhalb des Projektes gegeben.

Unter dieselbe Thematik fällt die Implementierung des von XQuery geerbten Typsystems in XPath 2.0. Das relationale Schema ist prinzipiell darauf vorbereitet, zu jedem Element und Attribut einen genauen Datentyp anzugeben. Da aber keine Verarbeitung von Schemas zu XML-Daten erfolgt und auch schemaunabhängige Dokumente gespeichert werden sollen, sind zum Zeitpunkt des Einlesens der Daten keine Typinformationen verfügbar. Aufgrund dieser eingeschränkten Typisierung sind auch viele Funktionen aus XPath 2.0 nutzlos, so dass nur ein kleiner, aber wichtiger Teil davon umgesetzt wird. Diese hauptsächlich noch aus XPath 1.0 stammenden Funktionen werden allerdings im Kontext von Version 2.0 der Pfadanfragesprache ausgeführt.

Für die Typverarbeitung bedeutet dies, dass sich auf die XPath-1.0-Typen beschränkt wird (Knotenmenge, Zahl, String, Boolean), diese intern allerdings einen Typ aus XPath 2.0 zugewiesen bekommen (Sequenz, xs:double, xs:string, xs:boolean).

Nicht unterstützt wird damit ebenfalls die Namensraum-Semantik. Präfixe werden zwar zusammen mit den lokalen Bezeichnern als qualifizierte Namen (QName) in der Datenbank abgelegt, jedoch sind ihnen keine URIs zugeordnet. Diese werden als pure Namensraumknoten ohne Prüfung auf Korrektheit oder semantische Interpretation in die Datenbank übernommen. Der Grund dafür ist wiederum die nicht implementierte Unterstützung für Schemas, wodurch keine geeignete Kontrolle der gegebenen Daten möglich ist. Allerdings kann diese optional vor dem Speichern der Daten in der Datenbank durch den Benutzer erfolgen; ein XMLRDB-Modul mit dieser Funktionalität ist denkbar.

Ein weiterer Aspekt der Namensraum-Problematik ist, dass die `namespace`-Achse in den Entwürfen für den neuen Standard XPath 2.0 als deprecated markiert ist [WWW05P]. Dies bedeutet, dass diese Achse veraltet ist und nicht implementiert werden muss. Da das bei dieser Diplomarbeit zu entwickelnde XMLRDB-Modul Anfragen im Kontext von XPath 2.0 ausführen soll, wird die `namespace`-Achse nicht umgesetzt.

Eine weitere Einschränkung betrifft das in Kapitel 2.4 vorgestellte Längenarray für die DLN-IDs. Dieses kann theoretisch für jedes Dokument separat festgelegt

werden, je nachdem, ob die Struktur der zu speichernden XML-Daten bekannt ist oder nicht. Diese Diplomarbeit beschränkt sich aber auf ein festes, global für die gesamte Datenbank geltendes Array. Für die Zukunft könnten diese Längenangaben auch in der Datenbank zu jedem Dokument gespeichert und bei Anfragen für die Bearbeitung der DLN-IDs herangezogen werden.

4. Umsetzung

Diesem Kapitel beschreibt nun, wie das Konzept der XPath-Übersetzung nach SQL als XMLRDB-Modul realisiert wurde. Dafür wird in Abschnitt 4.1 kurz die für die praktischen Tests verwendete Datenbanksoftware vorgestellt und zum Schluss nach einigen Benchmarks eine Auswertung der Leistungsfähigkeit vorgenommen (Kapitel 4.5).

4.1. Die Software-Basis

Das XMLRDB-Projekt wurde entworfen und umgesetzt in der objektorientierten Programmiersprache JAVA. Bei der Realisierung des XPath-Moduls wurde deshalb ebenfalls damit gearbeitet. Als Datenbanken kamen die beiden OpenSource-RDBMS PostgreSQL [PSQL05] und MySQL [MYSQL05] zum Einsatz (siehe Tabelle 4.1). Beide Datenbankserver sind unter den angegebenen Lizenzen für die Nutzung frei gegeben und können von den Homepages der Hersteller (siehe Quellenverzeichnis) bezogen werden. Dort finden sich auch die JDBC-Treiber und Administrations-Werkzeuge, welche für diese Diplomarbeit in den angegebenen Versionen genutzt wurden.

Datenbank	MySQL	PostgreSQL
Version	5.0.11-beta[1]	8.1
Lizenz	GNU General Public License	BSD license
Treiber	mysql-connector-java-3.1.11-bin.jar	postgresql-8.1-404.jdbc3.jar
Admintool[2]	MySQL Administrator 1.1.4	pgAdmin III 1.4.0

Tabelle 4.1.: Spezifikationen der verwendeten Datenbanken

[1] Die finale Version 5.0.15 erschien zu spät, um noch in dieser Diplomarbeit getestet zu werden. Die Unterschiede zwischen den Versionen sind jedoch minimal und betreffen keinen Aspekt dieser Diplomarbeit.

[2] hauptsächlich eingesetzt für das User-Management

Für das Erzeugen der SAX-Events beim Einlesen der Dokumente wurde der SAX-Parser Xerces [XJP01] in der Version 1.4.4 verwendet. Dieser steht ebenfalls unter einer OpenSource-Lizenz (Apache Software License) und kann über die Homepage der Apache Software Foundation bezogen werden.

4.2. Einbindung als Modul in XMLRDB

Der in Kapitel 3.2.2 vorgestellte Ablauf der Konvertierung eines XPath-Strings in einen SQL-String und die Bestimmung des DOM-Ergebnisbaumes mit Hilfe der Datenbank wird durch das XPath-Modul vollkommen verborgen. Der Nutzer muss nur noch eine Objektinstanz der `DLNXPathQuery`-Klasse erstellen, diese mit einem `XMLDBConnection`-Objekt initialisieren[3] und erhält nach der Übergabe des XPath-Anfrage-Strings an die Methode `executeXPath` ein DOM-Element zurück, welches den Wurzelknoten des Ergebnisdokumentes darstellt.

Intern geht das Modul aber nach folgendem Schema vor:

1. ein DOM-Dokumentenknoten für das Lösungsdokument wird erstellt
2. der Parser `xmlrdb.xpath.XPParser` erzeugt aus dem übergebenen XPath-Anfragestring ein `XPPath`-Objekt
3. die Klasse `xmlrdb.xpath2sql.XPPathProcessing` konvertiert den `XPPath` in die abstrakte SQL-Darstellung (ein `xmlrdb.sql.SQLQuery`-Objekt) und bestimmt den Rückgabetyp der Ergebnismenge[4]
4. daraus erstellt der `xmlrdb.mapping.dln.DLNSQLConverter` schließlich die finale Textform der SQL-Anfrage
5. die Anfrage wird nun an die Datenbank übergeben, welche dafür ein `ResultSet` erstellt
6. schließlich wird je nach gewählter Komplexität für jedes Ergebniselement ein entsprechender Knoten an das Lösungs-Dokument angehangen[5] oder nur ein Attribut mit der Kardinalität der Ergebnismenge gesetzt

[3] Das `XMLDBConnection`-Objekt stellt eine zentrale Management-Klasse für XMLRDB-Module dar, welche u.a. eine Implementierung des `SchemaHandlers` vorhält. Ein Programmierbeispiel für das XPath-Modul findet sich in Anhang B.1.

[4] dies ist wichtig um zu entscheiden, welche Daten für die Erstellung der Ergebnismenge noch aus der Datenbank extrahiert werden müssen

[5] über `PreparedStatements` werden die genauen Eigenschaften der Lösungsknoten ermittelt

7. der DOM-Dokumentenknoten wird als Lösung zurückgegeben

Der Nutzer kann nun den erschaffenen DOM-Baum weiterverarbeiten (Ausgabe als Dokument, Ausgabe auf dem Monitor, ...) und das `DLNXPathQuery`-Objekt für weitere Anfragen nutzen.

4.3. UDFs für die descendant- und ancestor-Achse

Wie in Kapitel 3.2.5 beschrieben, wird für die Umsetzung der XPath-Achsen `descendant(-or-self)` und `ancestor(-or-self)` die Möglichkeit der Datenbanken genutzt, eigene Funktionen zu programmieren. Für diese User Defined Functions (UDF) des XMLRDB-Projektes wurde das Präfix `xrdb_` gewählt, da z.B. die Datenbank DB2 von IBM maximal 18 Zeichen lange Bezeichner unterstützt und die Funktionsnamen `getAncestors` und `getMaxChildx` allein schon auf zwölf Zeichen kommen.

Die `descendant`-Umsetzung als Bestimmung des maximalen Kindes einer DLN-ID hat das Problem, dass die ID im relationalen Schema nicht als ein langes Feld gespeichert wird, sondern aufgeteilt auf mehrere Spalten. Damit bedarf es für einen Vergleich `Node1.ID < Node2.ID` mit zwei `id`-Feldern die SQL-Umsetzung `(Node1.id0 < Node2.id0) OR (Node1.id0 = Node2.id0 AND Node1.id1 < Node2.id1)`.

Dementsprechend gibt es auch mehrere UDFs `xrdb_getMaxChild0`, `xrdb_getMaxChild1` etc., welche die jeweilige Teil-ID des maximalen Kindes bestimmen. Dazu benötigen sie aber als Parameter alle Teile der Referenzknoten-ID, da intern erst die komplette DLN-ID des maximalen Kindes bestimmt werden muss.

Das resultierende SQL für eine `descendant`-Anfrage mit zwei DLN-ID-Spalten sieht dann wie folgt aus[6]:

```
SELECT ...
FROM ... xmlnode kontext, xmlnode neu ...
WHERE ... ((neu.id0 > kontext.id0 AND
      neu.id0 <= xrdb_getMaxChild0(kontext.id0, kontext.id1))
      OR (neu.id0 = kontext.id0 AND neu.id1 > kontext.id1 AND
      neu.id1 <= xrdb_getMaxChild1(kontext.id0, kontext.id1))
      ...
```

[6]Eine `descendant-or-self`-Anfrage wird daraus, wenn alle `>` durch `>=` ersetzt werden.

Die `ancestor`-Umsetzung kommt dagegen mit einer einzigen Funktion aus, da die Zielmenge der XPath-Achse direkt aus der übergebenen DLN-ID generiert werden kann. So beschränkt sich der Anfrage-Syntax auf:

```
SELECT ...
FROM ... xmlnode kontext, xmlnode neu ...
WHERE ... (neu.id0, neu.id1) IN
      xrdb_getAncestors(kontext.id0, kontext.id1, 0) ...
```

Der dritte Parameter der UDF gibt an, ob es sich um die `ancestor`- (0) oder `ancestor-or-self`-Achse (1) handelt. In letzterem Fall gibt die Funktion auch den Kontextknoten mit aus.

Während der Arbeit an dieser Diplomarbeit wurde die Standard-DLN-ID von der DOM- auf die Präfix-Variante umgestellt. Deshalb finden sich auf der beigelegten CD zu dieser Diplomarbeit auch zwei Umsetzungen der UDFs, da die IDs einen leicht unterschiedlichen Aufbau haben und damit verschiedene Parsing-Algorithmen notwendig sind.

Die Programmierung der UDFs erfolgte durchgängig in C und die Quelltexte mussten jeweils als dynamisch ladbare Bibliotheken compiliert werden, um in den Datenbanksystemen genutzt werden zu können. Die Beschreibung der Umsetzung beschränkt sich auf die grundsätzlichen Möglichkeiten der Datenbanken - genauere Anweisungen für die Integration der UDFs in die Datenbanken finden sich in Anhang B.2 und die Quelltexte auf der dieser Diplomarbeit beiliegenden CD.

4.3.1. UDFs in PostgreSQL

PostgreSQL erlaubt das Programmieren von User Defined Functions in C über zwei verschiedene Aufruf-Konventionen. Die neuere Version `1` erlaubt den Zugriff auf Funktionsparameter und die Ausgabe der Rückgabevariablen mittels Makros, welche einen besseren Umgang mit Nullwerten ermöglichen. Außerdem wird nur bei Konvention `1` die Rückgabe von Mengen unterstützt; die Ausgabe von selbstdefinierten Typen gab es schon vorher. Damit lassen sich sehr komfortabel UDFs programmieren - die `xrdb_getMaxChild`-Funktionen beschränken sich auf wenige Zeilen Code plus eine Hilfsfunktion für die Bestimmung der genauen Länge der DLN-ID[7].

[7] diese ist nicht automatisch bekannt, da die IDs mit Nullen aufgefüllt werden und so alle die gleiche Länge aufweisen

Leider ließ sich die Aufruf-Syntax der `xrdb_getAncestors`-Funktion nicht ganz wie beschrieben umsetzen. Eine Funktion, welche in PostgreSQL mehrere Zeilen zurückgibt, wird in Wirklichkeit für jede Zeile einmal ausgeführt, bis sie kein Ergebnis mehr zurückliefert. Eine interne Struktur sorgt dafür, dass bei jedem neuen Aufruf die alten Variablen wieder zur Verfügung stehen bzw. Zwischenergebnisse gespeichert werden.

Dadurch entsteht als Rückgabemenge ein tabellenartiges Konstrukt, auf welches PostgreSQL nicht in der WHERE-Klausel zurückgreifen kann. Für die Übergabe der aktuellen Tabellenauswahl musste der Aufruf von `xrdb_getAncestors` deshalb in eine Subquery verpackt werden. Das auch dies nicht ganz frei von Problemen war, wird in Abschnitt 4.4.4 näher ausgeführt.

4.3.2. UDFs in MySQL

MySQL erlaubt es zwar seit Version 4.x, User Defined Functions zu programmieren, jedoch sind deren Möglichkeiten sehr begrenzt. So erlaubt das RDBMS nur drei verschiedene Rückgabetypen (String, Integer, Float) und kennt keine tabellenwertigen Rückgabeformate. Eine Umgehung dieser Beschränkungen ist auf Grund der fehlenden Unterstützung für User Defined Types und der strengen Typüberprüfungen nicht möglich, so dass eine Umsetzung der `xrdb_getAncestors`-Funktion fehlschlug.

Nach einigen Tests blieb deshalb nur die Lösung, eine Auswahl von Datensätzen Zeile für Zeile mit einer Funktion `xrdb_isAncestor` auf Vorgängerschaft zu prüfen[8]. Das resultierende SQL sieht wie folgt aus:

```
SELECT ...
FROM ... xmlnode kontext, xmlnode neu ...
WHERE ... neu.ntype = 'e' AND ((neu.id0 < kontext.id0) OR
      (neu.id0 = kontext.id0 AND neu.id1 < kontext.id1))
      AND xrdb_isAncestor(neu.id0, neu.id1,
      kontext.id0, kontext.id1, 0) ...
```

Vor dem Aufruf der Funktion wird die Menge der zu testenden Knoten noch auf diejenigen Elemente reduziert, welche vor dem Kontextknoten liegen. Dies sollte eine

[8] Zwar kann MySQL auch um native Funtionen erweitert werden, jedoch bedarf dies einer Kompilierung der gesamten MySQL-Distribution zusammen mit der neuen Funktion. Dies wäre ein zu großer Aufwand, um ein funktionierendes XMLRDB-System aufzusetzen

sequentielle Prüfung der gesamten Tabelle verhindern, wenngleich die Performanz dieser Lösung trotzdem schlechter ausfallen dürfte verglichen mit der effizienten PostgreSQL-Umsetzung.

Ansonsten wurden auch die MySQL-UDFs in C programmiert und es konnte bei den `xrdb_getMaxChild`-Funktionen fast der identische Quellcode wie für die PostgreSQL-Versionen verwendet werden.

4.4. Zweistufige Konvertierung von XPath nach SQL

4.4.1. Die abstrakte SQL-Darstellung

Die Konvertierung des vom XPath-Parser generierten `XPPath`-Objektes in ein abstraktes `SQLQuery`-Objekt übernimmt wie beschrieben die Klasse `XPPathProcessing`. Es folgt nun eine genaue Vorstellung der abstrakten SQL-Darstellung, da diese als Grundlage für die Transformation von XPath in ein auf verschiedenen Nummerierungsschemas und Datenbanken ausführbares SQL dient.

Aus diesem Grund wurde auch ein eigenes Paket `xmlrdb.sql` angelegt, welches die Klassen für die Abbildung einer SQL-Anfrage in ein Objektmodell enthält. Das Unterpaket `xmlrdb.sql.interfaces` bietet drei Schnittstellen an, welche an Hand des Aufbaus einer SQL-Query erklärt werden sollen. Eine typische Anfrage in der strukturierten Anfragesprache hat folgende, auf die für dieses Kapitel wichtigsten Elemente beschränkte Syntax:

```
SELECT expression
FROM table_references
WHERE condition
ORDER BY expression [ ASC | DESC ]
LIMIT count [ OFFSET start ]
```

Mehr bedarf es nicht für die Umsetzung von XPath nach SQL. Bestimmte Aspekte der Syntax werden nicht für alle Anfragen benötigt; zwingend notwendig ist theoretisch nur die SELECT-Klausel. Hinter den kursiven Ausdrücken wie *expression* können sich jedoch die unterschiedlichsten Konstrukte bis hin zu einer weiteren SQL-Anfrage verbergen, weshalb folgende Interfaces definiert wurden:

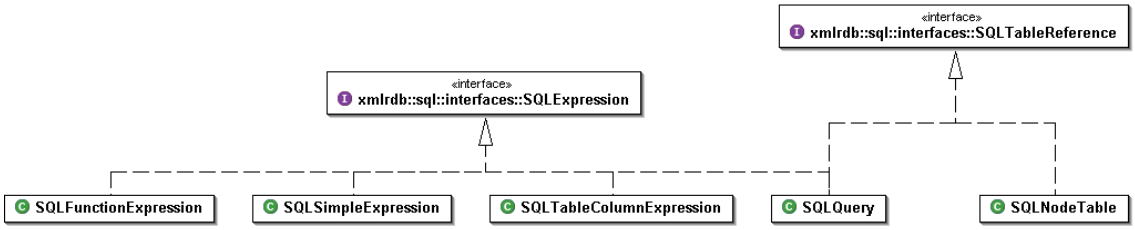

Abbildung 4.1.: Die Interfaces `SQLExpression` und `SQLTableReference` und implementierende Klassen

`SQLTableReference:` Diese Schnittstelle steht für ein Tabellenkonstrukt, welches in der FROM-Klausel einer SQL-Anfrage erscheinen kann. Dies sind typischerweise Tabellen oder Subqueries, welche über einen Alias verfügen, über den sie in anderen Ausdrücken referenziert werden können.

`SQLExpression:` SQLExpressions erscheinen fast überall in einer SQL-Anfrage - von der SELECT-Klausel über ORDER-BY-Statements bis hin zu den WHERE-Prädikaten. Sie können sowohl einfache Literale darstellen, als auch Funktionen oder sogar komplette SQL-Anfragen. Jede `SQLExpression` verfügt über einen eindeutigen Bezeichner, der in SQL mittels `as` definiert wird und über welchen der SQL-Ausdruck referenziert werden kann.

`SQLPredicate:` Die WHERE-Klausel in SQL besteht aus einer Reihe von Prädikaten, welche mit ANDs oder ORs verknüpft und durch Klammern gruppiert werden können. Typische Prädikate sind zum Beispiel Vergleiche zwischen `SQLExpressions`, doch dazu gleich mehr.

Eine `SQLQuery` kann damit beschrieben werden durch eine Liste von `SQLExpressions` (SELECT), eine Menge von `SQLTableReferences` (FROM), eine `SQLCondition` (WHERE) bestehend aus geordneten `SQLPredicates` und schließlich durch eine sortierte Liste von `SQLOrderByStatements` und einen `LIMIT`-Ausdruck. `SQLTableReferences` sind dabei entweder `SQLQuery`-Objekte oder vom Typ `SQLNodeTable`, welcher die abstrakte Tabelle `SQLNode` aus Kapitel 3.2.3 repräsentiert und die verfügbaren Spalten des Schemas als statische Variablen allen Objekten zugänglich macht.

Es folgt nun die Vorstellung der `SQLExpressions`, da diese die Grundlage für verschiedene andere SQL-Konstrukte darstellen. Vier verschiedene Klassen implementieren dieses Interface (siehe Grafik 4.1). Die `SQLSimpleExpression` steht dabei für ein einfaches Literal wie eine Zahl oder ein String.

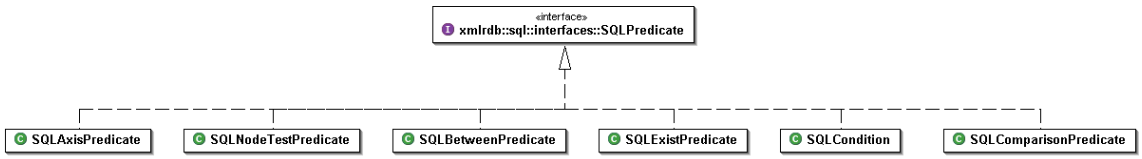

Abbildung 4.2.: Das Interface `SQLPredicate` und implementierende Klassen

Mit Hilfe einer `SQLTableColumnExpression` kann auf einzelne Spalten von bereits definierten Tabellenteferenzen über die Syntax `tabellenalias.spaltename` zurück gegriffen werden. Die `SQLFunctionExpression` bildet dagegen einen Aufruf einer Funktion nach. Sie besitzt daher eine Liste von Parametern und einen Funktionsnamen. Das `SQLQuery`-Objekt schließlich ist selber ein SQL-Ausdruck, weshalb es ebenfalls diese Schnittstelle umsetzt.

`SQLOrderByStatements` können nun durch einen der soeben vorgestellten SQL-Ausdrücke beschrieben werden, wobei zumeist die `SQLTableColumnExpression` genutzt wird. Zusätzlich enthält jedes Objekt dieser Klasse noch eine Ordnung, welche entweder aufsteigen (ASC, Standard) oder absteigend (DESC) definiert sein kann.

Kommen wir nun zu der SQL-WHERE-Klausel. Sie besteht wie beschrieben aus einer Reihe von SQL-Prädikaten, welche durch bestimmte Symbole verkettet und gruppiert werden können. Diese Syntax bildet die `SQLCondition`-Klasse ab. Sie besteht aus einer Reihe von `SQLPredicates`, welche entweder durch OR oder AND (Standard) miteinander verbunden sind. Außerdem kann eine `SQLCondition` Klammern um den gesamten Ausdruck besitzen. Da `SQLCondition` selber die Schnittstelle `SQLPredicate` implementiert, kann es beliebig geschachtelt werden. Damit ist die Erstellung einer komplexen WHERE-Klausel möglich.

Zum Schluss folgen nun alle Klassen, welche außer `SQLCondition` noch das Interface `SQLPredicate` enthalten und damit in WHERE-Klauseln vorkommen können. Der einfachste und häufigste Vertreter ist das `SQLComparisonPredicate`, welches zwei `SQLExpressions` mit einem Operator verbindet; also zum Beispiel `tabellenalias.spaltename = literal`. Es können aber auch Konstrukte wie `tabellenalias.spaltename IN subquery` oder `funktion(parameter) != NULL` dargestellt werden.

Weitere in SQL definierte und in `xmlrdb.sql` als Klassen umgesetzte Ausdrücke sind das `SQLBetweenPredicate` (`SQLExpression BETWEEN SQLExpression AND SQLExpression`) und das `SQLExistPredicate` (`[ NOT ] EXISTS SQLExpression`), welches vor allem bei der Konvertierung von XPath-Prädikaten zum Einsatz kommt.

Zu guter Letzt gibt es noch zwei Prädikate vorzustellen, welche keinen Gegenpart in SQL haben. Die beiden Klassen `SQLAxisPredicate` und `SQLNodeTestPredicate` stellen Wrapper-Objekte für die nicht komplett in die abstrakte SQL-Darstellung konvertierbaren XPath-Elemente Achsangabe und Knotentest dar. Sie enthalten nur Angaben über die verwendete Achse und die Art des Knotentests und werden bei der Transformation des `SQLQuery`-Objektes in die finale Textform durch ihre SQL-Repräsentationen ersetzt. Dies erklärt ausführlich das folgende Kapitel.

4.4.2. Die spezielle SQL-Darstellung

Den Hauptteil der Konvertierung des abstrakten `SQLQuery`-Objekts zu einem SQL-String erledigt die Klasse `DLNSQLConverter` des Paketes `xmlrdb.mapping.dln`. Hier werden aus den Objekten des `xmlrdb.sql`-Paketes die entsprechenden Textrepräsentationen auf dem realen Schema gewonnen, vorerst noch gekapselt in einem Objekt `DLNSQLQueryString`, welches über die Verwendung der genutzten Tabelleninstanzen und ihrer Aliase wacht[9].

Vorher wird jedoch mit Hilfe des `DLNSchemaHandlers` bestimmt, welche Datenbank als Ziel für die Anfrage agiert. Davon abhängig wird eine Implementierung des Interfaces `xmlrdb.mapping.dln.DBImplementation` geladen, welche die datenbankabhängigen Objektkonvertierungen durchführt. Dies betrifft alle diejenigen Elemente, welche nicht dem SQL-Standard unterliegen und deshalb von Datenbank zu Datenbank anders umgesetzt werden - wie zum Beispiel die im Abschnitt 4.4.4 noch vorzustellende Nutzung von Textindizes oder die Umsetzung der Achsen, welche über UDFs realisiert werden.

Die Methode `rewriteQuery` des Interfaces erlaubt es außerdem, die abstrakte `SQLQuery` noch vor der endgültigen Umwandlung in einen String logisch umzustellen. Dies ist z.B. bei PostgreSQL notwendig, da dort der QueryPlanner[10] nicht optimal arbeitet und ihm durch eine Umstellung der Anfrage auf die Sprünge geholfen werden kann.

Mit der Klasse `Standard_DBImplementation` gibt es eine Umsetzung des Interfaces `DBImplementation`, welche standardisierte Umsetzungen der Konvertierungsfunktionen und einige Hilfsmethoden bietet. Weitere Implementationen können von dieser Klasse abgeleitet werden und müssen so nur die Methoden überladen, welche

[9] alle Tabellen erhalten einen Alias nach dem Muster `x` + eine eindeutige ID

[10] Ein QueryPlanner ist ein Programm, welches verschiedene Strategien zur Ausführung von Anfragen vergleicht und die Beste davon auswählt.

einer Änderung bedürfen.

Ansonsten sorgt der `DLNSQLConverter` hauptsächlich dafür, dass ein korrekter SQL-Anfrage-String erzeugt wird. Bei Vergleichen auf der `value`-Spalte wird an dieser Stelle geprüft, ob eine Zahl als Vergleichsparameter vorliegt und in diesem Fall der Audruck auf einen Vergleich mit dem `doublevalue`-Feld geändert.

Die Knotentests sind unabhängig von der verwendeten Datenbank und werden durch den Ausdruck `ntype = 'x'` ersetzt, wobei `x` für das jeweilige Kürzel des Knotentyps steht (siehe dazu Kapitel 3.1.3). Ebenfalls an dieser Stelle werden die Bezeichner der Knoten durch ihre `doc`-IDs ersetzt. Dazu hält der `DLNSchemaHandler` die Namenstabellen des Schemas in einer Liste im Speicher vor, welche auch bei Updates genutzt und so ständig aktuell gehalten wird. Da XML-Dokumente meist nur über eine geringe Anzahl von Bezeichnern verfügen, sollte dieses Verfahren auf jeden Fall schneller sein, als den Namen über eine Anfrage auf der Datenbank zu bestimmen oder gar einen Join mit der `namemap`-Tabelle für jeden Bezeichner durchzuführen.

Über eine öffentliche Methode kann im `DLNSQLConverter` auf diese Namensliste zurückgegriffen werden, so dass noch vor der Ausführung der Anfrage auf der Datenbank die Ersetzung 'Name durch ID' erfolgt.

Nachdem alle Objekte in ihre Stringrepräsentationen umgewandelt wurden, kann das `DLNSQLQueryString`-Objekt endlich den finalen SQL-Ausdruck generieren, welcher anschließend an die Datenbank weitergereicht wird.

4.4.3. Optimierungen der SQL-Darstellung

An verschiedenen Stellen der Transformation von XPath nach SQL kann durch kleine Anpassungen dafür gesorgt werden, dass die SQL-Anfrage etwas kürzer ausfällt und damit schneller ausgeführt wird. So können teilweise Schritte der ausführlichen XPath-Form ausgelassen oder abgekürzt und bestimmte WHERE-Prädikate vereinfacht oder ganz weggelassen werden.

So steht zum Beispiel das `//` zu Beginn einer XPath-Anfrage für `/descendant-or-self::node()/`, also für alle Knoten des XML-Baumes. Da dies keinerlei Einschränkung der Knoten der `xmlnode`-Tabelle darstellt, kann dieser Schritt einfach weggelassen werden. Dies trifft auch zu, wenn vorher eine Einschränkung durch `fn:doc()` oder `fn:collection()` stattgefunden hat.

Ebenfalls weggelassen werden kann ein Schritt `/self::node()/`, wenn er nicht gerade auf die Auswahl eines Attributes oder Literals folgt. Denn dieser Schritt hat

keinerlei einschränkende Wirkung auf die Kontextmenge der Knoten und ist damit überflüssig. Generell hat die Achse `self` den Vorteil, nicht nach SQL übersetzt werden zu müssen, da sie keine Einschränkung darstellt.

Weitere Optimierungen betreffen die Verarbeitung von Prädikaten. Wie in Kapitel 3.2.1 beschrieben, werden diese im Standardfall über EXISTS-Subqueries in SQL realisiert. Der Grund dafür (ungewollte Vervielfachung der Kontextknoten durch Duplikate) ist jedoch nicht in allen Fällen gegeben. Einer diese Fälle sind Prädikate mit Attribut-Einschränkungen wie `[@nr < 100]` oder Attribut-Existenzbedingungen (`[@url]`). Da es von Attributknoten nur maximal einen pro Element und Bezeichner gibt, können Prädikate, welche Attribute eines Kontextelementes betreffen, ohne Subquery als einfache WHERE-Bedingungen umgesetzt werden.

Analog lassen sich Prädikate mit Einschränkungen des Kontextknotens übersetzen: `[. = 12]` wird so zu `WHERE kontext.doublevalue = 12`. Im Allgemeinen lässt sich sagen: Bei `1:1`-Beziehungen zwischen einem Knoten der Kontextmenge und einer Knotenauswahl im Prädikat können die Prädikate ohne Subqueries umgesetzt werden, da die Gefahr der Duplikatbildung nicht besteht.

Ebenfalls unter das Thema Optimierung fällt die Entscheidung, den Dokumentenknoten nicht explizit mit dem Nummerierungsschema DLN abzubilden. Durch den Wegfall des Knotens kann nämlich ein Tabellenjoin gespart werden, indem eine Anfrage zu Beginn nicht erst eine Menge von Dokumentenknoten und dann deren Kinder bestimmen muss. Desweiteren besitzt der Dokumentknoten einige Eigenschaften (enthält z.B. keine Attribute), deren Umsetzung die Anfragen komplexer gestaltet hätte.

Aus diesem Grund werden alle Kontextknotenmengen, welche den Dokumentknoten enthalten können[11], innerhalb des XPath-Modules angepasst behandelt. Da der Anwender meist gar nicht beabsichtigt, den Dokumentenknoten mit der Anfrage zu adressieren, wird versucht, die als Zwischenergebnis bestimmte Knotenmenge bei folgenden Schritten auf vorwärts gerichteten Achsen wieder um diesen Knoten zu bereinigen. Schon nach einem `child`-Schritt kann sich kein Dokumentknoten mehr in der Kontextmenge befinden.

[11] Die Startsequenz und bestimmte Kontexte nach rückwärts gerichteten Achsen können den Dokumentknoten enthalten. Der Knotentest `document-node()` wird jedoch nicht unterstützt.

4.4.4. Datenbankspezifische Anpassungen und Optimierungen

Leider werden User Defined Functions von Datenbank zu Datenbank unterschiedlich realisiert (siehe Abschnitt 4.3), so dass sich auch die Aufrufkonventionen in SQL unterscheiden. Das Interface `DBImplementation` sieht deshalb Methoden für die Umwandlung derjenigen `SQLAxisPredicate`-Objekte vor, welche eine der UDFs nutzen[12].

Probleme bereitete davon vor allem die `getAncestors`-Funktion. Während MySQL aufgrund seiner Limitierungen (siehe Kapitel 4.3.2) nur Tests nach dem Muster `Knoten1 ist Vorgänger von Knoten2` unterstützt und so für jeden Knoten aus der Kontextmenge fast die gesamte Tabelle sequentiell nach Vorgängern durchsuchen muss, steht bei PostgreSQL der QueryPlanner einer schnellen Abarbeitung im Wege. Er prüft für jeden Eintrag der Tabelle, ob eine Übereinstimmung mit dem Ergebnis der UDF vorliegt, da er annimmt, dass diese eine große Rückgabemenge erzeugt. Von dieser falschen Annahme kann er nur abgebracht werden, wenn vorher eine Art temporäre Tabelle als Zwischenergebnis mit der Rückgabemenge der `getAncestors`-Funktion gebildet wird.

Die `rewriteQuery`-Funktion muss deshalb die SQLQuery so umstellen, dass über eine Subquery eine Menge von Zeilen mit Dokumenten- und DLN-ID erzeugt wird, welche die Vorgänger aller als Kontextmenge vorliegenden Knoten enthält. Alle bisher bearbeiteten Schritte müssen dafür in die Subquery verschoben werden. Dies ist nur auf dem abstrakten SQLQuery-Objekt möglich, da während der Verarbeitung der Query durch den `DLNSQLConverter` die bereits umgewandelten Objekte nicht mehr vorliegen und so auch nicht in die Subquery verschoben werden können.

Ein weiteres Problem gab es mit MySQL und Subqueries, welche `IN` als Operator verwenden. Das RDBMS ist leider nicht in der Lage, solcherart Subqueries zu ordnen, was jedoch wichtig ist für `fn:position()`-Prädikate. Auch hier hilft eine Umstellung der Query, denn wenn die Subqueries unter FROM als Tabelle aufgeführt werden, kann die Ergebnismenge plötzlich geordnet werden. So muss nur noch eine Verknüpfung mit dieser Subquery-Tabelle den WHERE-Prädikaten hinzugefügt werden, und die Abfrage funktioniert wie gewünscht.

Unter die datenbank-spezifischen Optimierungen fällt auch die Verwendung der jeweiligen Textindizes der RDBMS für die `xmltext`-Tabelle. Unter MySQL muss diese leider im transaktionslosen Format MyISAM vorliegen, um einen FULLTEXT-Index zu nutzen. Dieser beschleunigt Anfragen dafür deutlich im Vergleich zu LIKE-

[12] `descendant(-or-self), ancestor(-or-self), preceding, following`

Vergleichen auf einer mit den ersten 10 Buchstaben indizierten InnoDB-Tabelle.

PostgreSQL kennt von Haus aus keinen Textindex, bringt jedoch eine Erweiterung namens TSearch V2 mit, welche eine invertierte Liste[13] in einer zusätzlichen Spalte anlegt und diese mit dem speziellen GisT-Index indiziert. Suchstrings müssen für Vergleiche damit erst in den speziellen Datentyp `tsquery` umgewandelt werden, was jedoch trotzdem eine höhere Performanz als ein normaler Index auf der `value`-Spalte aufweist.

4.5. Ergebnisse

4.5.1. Testdaten

Für die Tests des entwickelten XMLRDB-Moduls wurden jeweils zwei Datensätze mit unterschiedlichen Schwerpunkten in die vorgestellten Datenbanken eingelesen. Datensatz I besteht aus einer großen Datei (`standard.xml`), erzeugt durch den Datengenerator `xmlgen` des XMark-Benchmarks [XMARK01]. Dieser Generator ist in der Lage, beliebig große XML-Dokumente über einer DTD zu erzeugen, welche den Katalog eines Auktionshauses abbildet. Auf der XMark-Homepage findet sich aber auch eine bereits generierte, 111MB große Datei, welche für diesen Test ausgewählt wurde.

Die Daten zeichnen sich durch eine große Anzahl von Knoten aus (siehe Tabelle 4.2), welche einen sehr breiten und maximal zehn Ebenen tiefen XML-Baum bilden. Die Herausforderung an diesem Datensatz ist, dass die Datenbank keine Einschränkung auf ein Dokument vornehmen kann, sondern die Knotendatensätze des nächsten Joins über die gesamte Tabelle verteilt sein können.

Datensatz II besteht dagegen aus einer großen Anzahl von kleineren Dokumenten aus dem XMach-1-Benchmark [XMACH01]. Die Dokumente wurden mit dem Datengenerator des Benchmarks generiert, welcher zusammen mit diesem auf der Projekt-Homepage zur Verfügung steht. Alle der so erzeugten Dokumente weisen eine ähnliche Struktur auf, da sie Dokumente mit Kapitel, Sektionen und Titeln abbilden, haben aber trotzdem teilweise unterschiedliche Knotennamen[14]. Zusätzlich

[13] Eine invertierte Liste enthält alle Wörter eines Textes mit Verweisen auf ihre Positionen innerhalb des Textes und eignet sich deshalb gut für eine Volltextsuche.

[14] Die Dokumente wurden mittels 320 DTDs erzeugt, welche sich nur durch Präfixe der Elementnamen voneinander unterscheiden (`document1` bis `document320`). Keine DTD erzeugte mehr als 100 Dokumente, so dass durch das Verwenden der Elementnamen in Anfragen eine gewisse Lokalität gegeben ist.

enthält Datensatz II noch ein Dokument `directory.xml`, welches die 5000 generierten Dokumente in einer Art virtuellem XML-Verzeichnisbaum anordnet. Bei diesen Daten war die Fragestellung interessant, ob die Datenbanken von Einschränkungen auf wenige Dokumente profitieren können, da sie nur auf einem geringen Teil der Tabelle nach Datensätzen suchen müssen.

	Datensatz I	**Datensatz II**
Anzahl Dateien	1	5001
Größe des Datensatzes	111 MB	75 MB
Anzahl Knoten	2840047	1088308
davon Textknoten	1173732	463101
Anzahl Attribute	381878	189476
Speicherbedarf MySQL	335 MB (845 MB)	172 MB (404 MB)
Speicherbedarf PostgreSQL	597 MB (1240 MB)	285 MB (467 MB)

Tabelle 4.2.: Datensätze für die Benchmarks (die Werte in Klammern geben die Größen der Indizes an)

Nach dem Einlesen der Datensätze wurden für beide Datenbanken zusätzlich zu den Primärschlüsseln Indizes auf den für Anfragen relevanten Spalten erzeugt. Für die `xmlnode`-Tabelle sind dies die Felder `cid`, `parent`, `name`, `value`, `doublevalue` und `ntype`; bei `xmltext` und `xmlattr` kommen die `cid` und `value`-Spalten hinzu[15]. Alle diese Spalten werden bei typischen XPath-Anfragen zu Einschränkungen der Knotenmenge gebraucht und profitieren deshalb von der Indizierung[16].

Bei PostgreSQL ist es nach der Indizierung nützlich, den ANALYZE-Befehl auf den drei relevanten Tabellen auszuführen. Dieser sammelt statistische Informationen über die Daten und Indizes und hilft so dem QueryPlanner, eine günstige Ausführungsreihenfolge zu finden bzw. die richtigen Indizes zu verwenden. Leider arbeitet dieses Verfahren bei vielen Tabellen-Joins und verschachtelten Subquery-Anfragen nicht mehr sehr effizient.

MySQL besitzt ebenfalls ein ANALYZE-Werkzeug, welches die Verteilung der Schlüssel von Tabellen analysiert und der Datenbank hilft zu entscheiden, welche

[15] für Informationen zu den genutzten Textindizes auf der Tabelle `mxltext` siehe Kapitel 4.4.4

[16] Erste Tests mit Datensatz I ohne Indizes auf allen Spalten mussten abgebrochen werden, da bereits ein sequentieller Durchlauf der `xmlnode`-Tabelle die Anfrage minutenlang verzögerte. Deshalb werden die Indizes inzwischen bei der Erzeugung des Schemas mit angelegt.

Indizes es nutzen soll und wie Tabellen-Joins optimal ausgeführt werden. Bei beiden Datenbanken wurden diese Funktionen nach dem Einlesen der Daten aufgerufen.

4.5.2. Benchmarkeinstellungen

Für Datensatz I wurden probeweise neben den Indizes noch Fremdschlüssel definiert und alle Anfragen mit und ohne diese ausgeführt. Dieser Vergleich soll zeigen, ob sich durch die Verwendung von Fremdschlüsseln zusätzlich zu den indizierten Spalten Vorteile ergeben. Neben den zur Verknüpfung der Tabellen genutzen `cid`-Feldern von `xmltext` und `xmlattr`, welche die `cid`-Spalte von `xmlnode` referenzieren, wurde auch ein Fremdschlüssel von `parent` auf `cid` innerhalb der Hauptknotentabelle definiert. Dieser könnte bei den häufigen Anfragen auf der Standardachse `child` eine positive Auswirkung auf die Ausführgeschwindigkeit haben.

Bei den Anfragen auf Datensatz II wurden zwei Varianten der Anfragen getestet. Bei der ersten Version kamen SQL-Queries zum Zuge, welche die Joins zwischen den Tabellen des Schemas hauptsächlich auf Grundlage des `cid`-Feldes eines Knotens durchführen. Bei der zweiten Version wurde jedoch bei jedem Join zusätzlich die Bedingung `oldtable.doc = newtable.doc` eingefügt, wo dies nicht automatisch geschieht. Damit soll geprüft werden, ob die Datenbanken von zusätzlichen Bereichsangaben profitieren, was bei Datensatz I aufgrund des einzelnen Dokumentes nicht ermittelt werden konnte.

Probleme bereitete beim Benchmarken vor allem die Tatsache, dass das Cachen der Datenbanken und auch des Betriebssystemes nicht unterbunden werden kann. Bei MySQL kann zwar über eine Option der Querycache auf `0KB` gesetzt und per `SQL_NO_CACHE` für jede Anfrage einzeln deaktiviert werden, allein eine Auswirkung zeigten diese Optionen bei den Tests nicht. Ab der zweiten Abfrage kam es immer zu niedrigeren Anfragezeiten, die zum Einen auf das Zwischenspeichern der Ergebnisse deuten (bei Anfragezeiten im Bereich von wenigen Millisekunden) bzw. auf das Vorhalten der zuletzt abgefragten Seiten der Datenbank (meist eine Beschleunigung von 30 bis 40% der Anfragezeiten).

Damit kann natürlich nicht ermittelt werden, wie schnell die Datenbanken effektiv die Anfragen ausführen. In einer Arbeitsumgebung kann es zwar durchaus vorkommen, dass ständig ähnliche Anfragen gestellt werden, aber davon kann nicht ausgegangen werden. Da ein Neustart der Datenbanken (wahrscheinlich aufgrund des Cachen des Betriebssystemes) ebenfalls keine Besserung brachte, musste für jeden Benchmarklauf der PC neu gestartet werden.

Die im nächsten Abschnitt präsentierten Benchmarkergebnisse stellen Mittelwerte aus je fünf Läufen dar. Die genauen Listen finden sich in Anhang C dieser Diplomarbeit; die in diesem Kapitel dargestellten Abbildungen dienen nur zur Einordnung und zum Vergleich der Ergebnisse. Die abgebildeten Zeiten stellen den Zeitrahmen dar, in welchem die SQL-Anfrage an die Datenbank per `Statement.executeQuery()` übermittelt wurde und diese eine Ergebnismenge als `ResultSet`-Objekt zurücklieferte. Die Konvertierung der XPath-Anfrage in einen SQL-String läuft innerhalb weniger Millisekunden ab und kann deshalb vernachlässigt werden.

Ebenfalls nicht enthalten in den Zeiten ist das Auslesen der kompletten Knoteninformationen aus der Datenbank für die Erzeugung eines Ergebnis-DOM-Baumes. Dies bedarf einer annähernd konstanten Zeit pro Ergebnisknoten und hängt deshalb mehr von der Ergebnismenge ab als von der Anfrage bzw. deren Formulierung.

Vor jeder Anfrage wurden Datenbank und Datenbanktreiber bereits mit einer kurzen Query initialisiert, um Einflüsse von dieser Seite auf die Ergebnisse zu vermeiden. Dies entspricht auch einer realen Arbeitsumgebung, in welcher regelmäßig Anfragen an die Datenbank gestellt werden. Dabei wurde darauf geachtet, dass die Test-Anfrage nicht auf den XML-Daten arbeitete, um ein Laden von Seiten oder Indizes in den Cache zu verhindern[17].

Allen Benchmarks wurden schließlich auf einem Rechner mit 512MB RAM und einem Pentium IV mit 2,53GHz durchgeführt. Als Betriebssystem kam Windows 2000 zum Einsatz und JAVA lag in Version 1.4.2-b28 des Sun Runtime Environment (Standard Edition) vor.

4.5.3. Benchmarkresultate und Interpretation

Die meisten simplen XPath-Anfragen über die `child-` oder `parent`-Achse beantwortet das Modul in wenigen hundert Millisekunden. Anfragen über die Achsen `preceding(-sibling)` und `following(-sibling)` sind aufgrund der Nutzung des `parent`-Feldes und der DLN-ID-Ordnung ebenso schnell ausgeführt, wenn sich die Ergebnismenge in Grenzen hält.

Die Queries in diesem Benchmarkt wurden deshalb bewusst etwas komplexer gewählt, um Unterschiede zwischen den Datenbanken und den gewählten Anfragevarianten aufzuzeigen. Dabei wurde darauf geachtet, möglichst kleine Rückgabemengen zu erzeugen, damit die Datenbanken nicht den Großteil der Zeit damit

[17] Die Anfrage ist `SELECT xrdb_getMaxChild0(1,0)` und testet gleichzeitig, ob die UDFs geladen werden konnten.

verbringen, die Lösungsmenge von der Festplatte zu lesen und an den Mediator zu übertragen.

Query	Anfrage	#
1	`//person[profile/business='Yes']/address[country='Canada']/city`	2
2	`/site/people/person[profile/business='Yes']/address[country='Canada']/city`	2
3	`//closed_auction[price>750]/seller`	3
4	`/site/closed_auctions/closed_auction[price>750]/seller`	3
5	`//profile[@income>140000]`	6
6	`/site/people/person/profile[@income>140000]`	6
7	`//address[city='Panama']/zipcode`	54
8	`/site/people/person/address[city='Panama']/zipcode`	54
9	`//open_auction[1to10]/seller/@person`	10
10	`//open_auction[fn:count(bidder)>50]`	1
11	`//date[.='06/18/1999']/ancestor::item`	28

Tabelle 4.3.: XPath-Anfragen und die Kardinalität der Lösungsmengen für die Benchmarks auf Datensatz I

Query	Anfrage	#
12	`/directory/host[fn:count(.//path/doc_info[@insert_time>20051214113124000])>5]/@name`	3
13	`fn:count(//doc_info[@insert_time>20051214113124000])`	1
14	`fn:doc('02673_doc_154_test.xml')//chapter154[fn:contains(.,'number')]/@id`	3
15	`/document154[@doc_id='d2673']//chapter154[fn:contains(.,'number')]/@id`	3
16	`/*[fn:count(.//link)>25]/@doc_id`	11

Tabelle 4.4.: XPath-Anfragen und die Kardinalität der Lösungsmengen für die Benchmarks auf Datensatz II

Es folgt nun eine Vorstellung der einzelnen Anfragen und die Auswertung der erzielten Ergebnisse:

`Queries 1 bis 11:` Über alle Anfragen hinweg ließ sich kein Vorteil durch die zusätzliche Nutzung von Fremdschlüsselbedingungen feststellen. Weder bei Joins auf der `xmlnode`-Tabelle noch bei Verknüpfungen mit der Attributtabelle gab es einen sichtbarer Vorteil für die Fremdschlüssel. Unter MySQL sind die Anfragezeiten im Durchschnitt sogar minimal höher als ohne Fremdschlüssel, so dass auf eine Verwendung dieser Funktion verzichtet werden kann.

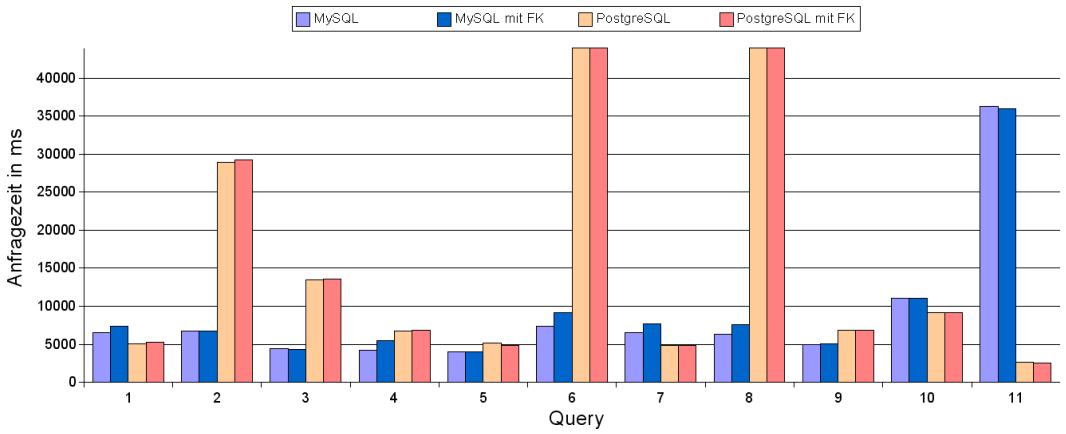

Abbildung 4.3.: Benchmarkergebnisse für Datensatz I jeweils mit und ohne Fremdschlüsselbeziehungen (FK)

Die Ursache dafür ist wahrscheinlich darin begründet, dass die Datenbank für den Zugriff auf den UNIQUE-Index der Fremdschlüsselbeziehung genauso viel Zeit braucht wie um zu erkennen, dass es nur genau einen Eintrag zu dieser ID im Index der referenzierten Spalte gibt.

`Queries 1 bis 8:` Diese Anfragen bestehen aus Paaren von jeweils zwei inhaltlich identischen Queries. Die ungeraden Anfragen nutzen ausgiebig die `descendant`-Achse, während die Gegenparts mit geraden Nummern den gesamten Pfad bis zu den gesuchten Elementen angeben. Eigentlich müsste den Datenbanken eine weitere Einschränkung der Auswahl durch die `xrdb_getMaxChild`-UDFs besser liegen als ein zusätzlicher Join mit einer neuen `xmlnode`-Tabelleninstanz.

Unter MySQL sind jedoch keine größeren Unterschiede feststellbar, was auf eine sehr effiziente Umsetzung von Tabellenjoins mit Instanzen einer einzelnen Tabelle deutet. Erst bei einer Verknüpfung mit der Attributtabelle in den Anfragen 5 & 6 kann sich die `descendant`-Umsetzung absetzen - da hilft auch keine Fremdschlüsselbeziehung.

Bei PostgreSQL zeigt sich ein gemischtes Bild. Während die Queries 2, 6 und 8 ohne die `descendant`-Achse durch deutlich höhere Anfragezeiten auffallen, ist Anfrage 4 schneller mit dem ausführlichen Weg. Wahrscheinlich kann PostgreSQL Tabellen-Joins bis zu einer bestimmten Anzahl ebenfalls sehr gut optimieren, so dass diese kurze Anfrage im Gegensatz zu den anderen Queries davon profitiert.

Generell ist jedoch eine Nutzung der `descendant`-Achse vorzuziehen, da zu viele Tabellen-Joins vor allem unter PostgreSQL die Laufzeiten der Anfragen deutlich verlängern können.

`Query 9:` Diese Anfrage stellt einen Test der Positionsprädikate dar. Hier muss die Datenbank an Hand des Primärschlüssels erkennen können, dass nicht erst alle `open_auction`-Elemente gesucht werden müssen, da in der Positions-Subquery nur die ersten zehn davon ausgewählt werden. Beide Datenbanken scheinen dies zu leisten; der leichte Unterschied der Anfragezeiten ist vermutlich mit der etwas höheren Join-Performanz von MySQL zu erklären.

`Query 10:` Anfrage 10 muss alle `bidder`-Kindknoten der `open_auction`-Elemente zählen. Die langen Anfragezeiten sind wahrscheinlich mit der großen Anzahl an `open_auction`-Knoten und der noch größeren Anzahl an `bidder`-Elementen zu begründen. Diese sind noch dazu über die gesamte Tabelle verteilt, so dass die Datenbanksysteme aufwendig für jedes `open_auction`-Elemente alle `bidder`-Knoten bestimmen und dann diese auf Vaterschaft durch den Kontextknoten testen müssen. Diese Schleife lässt sich aufgrund der Zählung in der Subquery nicht wegoptimieren.

An dieser Stelle kann PostgreSQL zeigen, dass es auf großen Datenmengen und bei schwierigen Anfragen dem Konkurrenten MySQL überlegen ist.

`Query 11:` Bei der Umsetzung der `ancestor`-Achse zeigt sich der in Kapitel 4.3.2 erwartete Einbruch unter MySQL. Aufgrund der fehlenden Möglichkeit, durch eine UDF tabellenwertige Rückgabemengen umzusetzen, muss eine große Anzahl von Tabelleneinträgen auf Vorgängerschaft getestet werden. Damit kann leider nur von der Nutzung dieser Achse unter MySQL abgeraten werden.

PostgreSQL dagegen kann optimal von den Strukturinformationen der DLN-ID profitieren und zeigt ein entsprechend gutes Ergebnis bei den Anfragezeiten.

`Queries 12 bis 16:` Auf Datensatz II wurden die zwei vorgestellten Varianten der Tabellenverknüpfung getestet. MySQL zeigt nur bei Anfrage 16, dass es diese zusätzliche Hilfe benötigt, und liefert bei allen anderen Queries vergleichbare Anfragezeiten. Der QueryPlanner weiß also in den meisten Fällen die definierten Indizes effizient zu nutzen.

PostgreSQL profitiert trotz der statistischen Analyse der Daten und Indizes bei Anfrage 12 deutlich von den zusätzlichen Bereichsinformationen, während diese bei der letzten Anfrage eher eine Belastung darstellen. Bei sehr seltenen

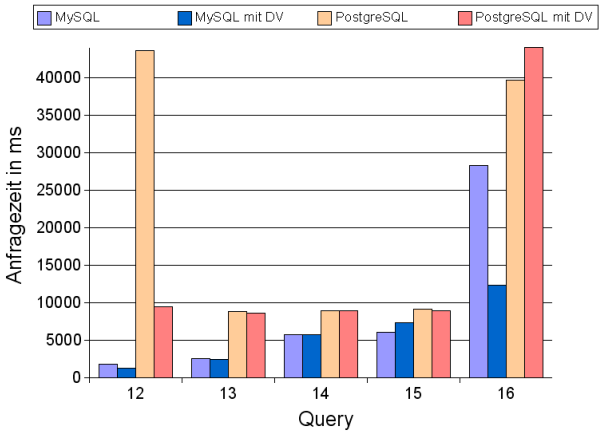

Abbildung 4.4.: Benchmarkergebnisse für Datensatz II jeweils mit und ohne die Verknüpfung per Dokumenten-ID bei Tabellen-Joins (DV)

und lokalen Elementnamen kann sich die statistische Analyse somit negativ auf die Anfragezeiten auswirken, was sich durch die Angabe des Dokumentes kompensieren lässt.

Da sich nur sehr geringe Nachteile, aber zum Teil deutliche Vorteile aus der Verknüpfung über die Dokumentenspalte ergeben, wurde dieser zusätzliche Join-Parameter in der finalen Version des XPath-Moduls standardmäßig aktiviert.

`Query 12:` Bei dieser Anfrage zeigt PostgreSQL kein gutes Ergebnis. Während MySQL aufgrund der Elementnamen genau weiß, dass es nur einen sehr geringen Teil der `xmlnode`-Tabelle überhaupt zu durchsuchen braucht, muss die PostgreSQL-Datenbank durch die Angabe des Dokumentes darauf hingewiesen werden. Dazu kommt die generell nicht sehr gute Performanz dieses DBMS bei ineinander verschachtelten Subqueries (gesteigert durch einen `descendant`-Schritt), so dass auf den Einsatz von Prädikaten innerhalb anderer Prädikate unter PostgreSQL verzichtet werden sollte.

`Query 13:` Da eine äußere `fn:count()`-Funktion sich nur durch eine veränderte SELECT-Anweisung bemerkbar macht, zeigt diese Anfrage die Performanz einer `descendant`-Anfrage auf einem einzigen Dokument. An dieser Stelle benötigt PostgreSQL nicht die Hilfe der Dokumenten-ID und braucht in etwa die gleiche Zeit wie bei der vorhergehenden Frage, da die Elementnamen ausreichend einschränkend wirken.

`Query 14 & 15:` Diese beiden Anfragen sind wieder inhaltlich identisch. Bei Query 14 wird die Auswahl über die Angabe des Dokumentes beschränkt, in Query 15 wird dazu das Attribut des `document`-Elementes verwendet. In beiden Versionen scheint die Einschränkung auf Elemente mit der Nummer 154 für die QueryPlanner ausreichend zu sein, um nicht auf der gesamten Tabelle suchen zu müssen.

Ein weiterer Aspekt bei diesen Anfragen ist die Nutzung der Textindizes auf der `xmltext`-Relation, welche über die Funktion `fn:contains()` explizit angesprochen werden. Dabei ist es von Vorteil, diese Funktion nur auf einer kleinen Menge von Knoten ohne viele Nachfolger auszuführen, da sie alle `descendant`-Textknoten eines Elementes durchsuchen muss (siehe Kapitel 3.2.7). Unter MySQL schlägt sich dies gegenüber den zwei nicht viel komplexeren Anfragen 12 & 13 deutlich in den Anfragezeiten nieder.

`Query 16:` Die letzte XPath-Anfrage des Benchmarks zeigt als einzige einen größeren Nachteil der zusätzlichen Dokumentenverknüpfung. PostgreSQL bricht um 20% ein gegenüber der auch nicht sehr schnellen Antwort ohne die Verknüpfung, während MySQL zum ersten Mal deutlich davon profitiert.

Das Datenbanksystem muss für die Beantwortung der Anfrage zu dem obersten Element von jedem der 5000 Dokumente eine Subquery auflösen, welche die nachfolgenden `link`-Elemente zählt. An dieser Stelle ist es wichtig zu wissen, dass diese Elemente nur in der Lokalität des Dokumentes des aktuellen Kontextknotens zu suchen sind. Dies geschieht bei der Nutzung der `descendant`-Achse automatisch, jedoch bei dem in der ausführlichen Form des XPath-Ausdruckes folgenden `child::link`-Schritt nicht.

Die DBS greifen bei diesem Join nun zusätzlich zu den Indizes auf der `cid`- und `parent`-Spalte auch auf den Primärschlüssel zu, um die Auswahl auf das Dokument einzuschränken. Dies hat positive Auswirkungen bei MySQL, da der Elementname keine Lokalität aufweist, während sich bei PostgreSQL aufgrund der gestiegenen Komplexität der Subquery die Anfragezeit erhöht.

4.5.4. Fazit

Simple Anfragen wie `/name1/name2/name3` stellen das XPath-Modul auch bei großen Datenmengen vor keine Probleme. Bei der Angabe von langen Pfaden mit vielen Prädikateinschränkungen gibt es aber einen Schwellenwert, bei dessen Überschrei-

ten die Datenbanksysteme die Tabellenjoins nicht mehr effizient durchführen können und die Anfragezeiten deutlich steigen - bei PostgreSQL liegt dieser Wert niedriger als bei MySQL. Deshalb sollte auf jeden Fall die `descendant`-Achse eingesetzt werden, wenn dies möglich ist.

Von der Verwendung der `ancestor`-Achse unter MySQL muss leider abgeraten werden, da dort die Implementierung als UDF aufgrund der Limitierungen der Datenbank nicht wie gewünscht erfolgen konnte. Bei PostgreSQL dagegen werden auch Anfragen über diese Achse aufgrund der Struktur der DLN-ID sehr effizient ausgeführt.

Eine Definition von Fremdschlüsseln bringt bei der Anfragenperformanz hingegen keinen sichtbaren Vorteil, da die QueryPlanner genug Informationen aus den definierten Indizes gewinnen können. Die zusätzliche Verknüpfung von zwei zu joinenden Tabellen über die Dokumenten-ID kann jedoch unter PostgreSQL und MySQL teilweise erhebliche Vorteile bringen, weshalb diese nun standardmäßig durchgeführt wird.

Bei Verwendung der Funktion `fn:contains` sollte darauf geachtet werden, dass eine angegebene Knotenmenge nicht zu viele Textknoten enthält. Denn trotz der eingesetzten Textindizes sind die Datenbanken nicht in der Lage, Joins mit einer ausgewählten Menge von Datensätzen aus der Tabelle `xmltext` in einer vertretbaren Zeit durchzuführen.

5. Zusammenfassung

Im Rahmen dieser Diplomarbeit wurde eine Implementierung der XPath-Anfragesprache als Modul für das XMLRDB-Projekt entwickelt. Das Modul ist in der Lage, die Vorteile von DLN bei einer Anfrage auf den mit diesem Nummerierungsschema in einem RDBMS gespeicherten Daten zu nutzen und kann so effizient die Antwortmengen zu gestellten XPath-Queries bestimmen.

Damit existiert eine Anfrage-Schnittstelle für das Projekt, welche den Zugriff auf die XML-Daten wie bei einer nativen Speicherung erlaubt, ohne dass der Anwender sich Gedanken um den relationalen Hintergrund machen muss. Er stellt seine Anfrage an das System und erhält einen XML-DOM-Baum als Antwortmenge auf seine Query zurück.

Das Anfragemodul unterstützt dabei sämtliche von der XPath 2.0-Spezifikation verlangten Achsen und einen Großteil des Funktionsumfanges der alten Version der Anfragesprache. An Funktionen wurde zunächst nur eine Teilmenge umgesetzt, welche unter anderem die gebräuchlichen Funktionen `fn:count()`, `fn:position()` und `fn:contains()` umfasst. Weitere Funktionen können ohne viel Aufwand in einer fortführenden Arbeit hinzugefügt werden, müssten sich jedoch aufgrund der noch fehlenden Schemaintegration mit der Beschränkung auf die vier Datentypen von XPath 1.0 begnügen. Das Modul selber ist vorbereitet auf eine zukünftige Erweiterung der Fähigkeiten von XMLRDB um die Typhierarchie von XML Schema und XQuery und das damit verbundene Namensraumkonzept. Die momentane Auslassung dieser Funktionen ermöglicht es aber dem Projekt, sämtliche dokumentenzentrierten XML-Daten zu speichern und XPath-Anfragen darauf auszuführen.

Das Modul nutzt für diese Anfragen die Vorteile von DLN aus, das Ziel von bestimmten Achsen wie `ancestor` allein durch das Parsen der Knoten-ID zu bestimmen und häufig genutzte Elemente von XPath wie die `descendant`-Achse durch die Realisierung als Nummernsuche auf einem Bereich zu beschleunigen. Weiterhin können die Anfragen bei der Umwandlung nach SQL für das verwendete Datenbanksystem optimiert werden, indem die speziellen Funktionen der RDBMS genutzt und

Schwachstellen bei der Bearbeitung von bestimmten Ausdrücken umgangen werden.

Dies wurde examplarisch an Hand von zwei OpenSource-Datenbanken vorgeführt. Die Anforderungen aus Kapitel 1 sind also in der Praxis umsetzbar und können mit geringem Aufwand auch auf weitere relationale Datenbanksysteme übertragen werden.

Auf Basis des vorliegenden XPath-Moduls ist zudem eine Integration der XQuery-Sprache in das XMLRDB-Projekt denkbar. Diese könnte dann die fehlenden neuen Funktionen von XPath 2.0 wie Schleifen aufgreifen, müsste dafür aber wahrscheinlich von der effizienten Bearbeitung der Anfragen durch die Datenbank in einem einzigen Schritt abrücken. Auch andere XML-Schnittstellen wie XSL benötigen XPath für die Adressierung von XML-Knoten, so dass ein wichtiger Schritt für die Nutzung von relationalen Datenbanken als Speicherlösung für XML-Daten erfolgt ist.

Quellenverzeichnis

[ANT05] The Apache Ant Project: *Apache Ant*. http://ant.apache.org/, 2005

[B05] Böhme, Timo: *Optimierung von XPath-Anfragen in XMLRDB*. Oberseminar Datenbanken, Universität Leipzig, 2005

[BR04] Böhme, Timo; Rahm, Erhard: *Supporting Efficient Streaming and Insertion of XML Data in RDBMS*. Proceedings of the International Workshop on Data Integration over the Web, Riga, Latvia, 2004

[CFZ04] Chan, Chee-Yong; Fan, Wenfei; Zeng, Yiming: *Taming XPath Queries by Minimizing Wildcard Steps*. Proceedings of the VLDB Conference, Toronto, Canada, 2004

[DFS99] Deutsch, Alin; Fernandez, Mary; Suciu, Dan: *Storing Semistructured Data with STORED*. Proceedings of the ACM SIGMOD Conference, Philadelphia, USA, 1999

[DTCÖ03] DeHaan, David; Toman, David; Consens, Mariano P.; Özsu, M. Tamer: *A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding*. Proceedings of the ACM SIGMOD Conference, San Diego, USA, 2003

[FK99] Florescu, Daniela; Kossmann, Donald: *Storing and Querying XML Data using an RDMBS*. IEEE Data Engineering Bulletin 22(3), 1999

[FMS01] Fernandez, Mary; Morishima, Atsuyuki; Suciuy, Dan: *Efficient Evaluation of XML Middleware Queries*. Proceedings of the ACM SIGMOD Conference, Santa Barbara, USA, 2001

[GKT03] Grust, Torsten; van Keulen, Maurice; Teubner, Jens: *Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps*. Proceedings of the VLDB Conference, Berlin, Germany, 2003

[GRUST02] Grust, Torsten: *Accelerating XPath Location Steps*. Proceedings of the ACM SIGMOD Conference, Madison, USA, 2002

[LM01] Li, Quanzhong; Moon, Bongki: *Indexing and Querying XML Data for Regular Path Expressions*. Proceedings of the VLDB Conference, Roma, Italy, 2001

[MFK01] Manolescu, Ioana; Florescu, Daniela; Kossmann, Donald: *Answering XML Queries over Heterogeneous Data Sources*. Proceedings of the VLDB Conference, Roma, Italy, 2001

[MGW05] MinGW Team: *MinGW and MSYS*. http://www.mingw.org/, 2005

[MVC05] Microsoft Corporation: *Microsoft Visual Studio C++ 2005 Express Edition*. http://msdn.microsoft.com/vstudio/express/visualC/, 2005

[MYSQL05] MySQL AB: *MySQL 5*. http://www.mysql.com/, 2005

[OMFB02] Olteanu, Dan; Meuss, Holger; Furche, Tim; Bry, Francois: *Xpath: Looking forward*. Workshop on XML-based Data Management, Prague, Czech Republic, 2002

[OOPCSW04] O'Neil, Patrick; O'Neil, Elizabeth; Pal, Shankar; Cseri, Istvan; Schaller, Gideon; Westbury, Nigel: *ORDPATHs: Insert-Friendly XML Node Labels*. Proceedings of the ACM SIGMOD Conference, Paris, France, 2004

[PSQL05] PostgreSQL Global Development Group: *PostgreSQL 8.1*. http://www.postgresql.org, 2005

[S02] Schröder, Peter: *Evaluierung relationaler Datenbanksysteme hinsichtlich ihrer Eignung zur Verwaltung von XML-Daten*. Diplomarbeit, Universität Leipzig, 2002

[SKWW00] Schmidt, Albrecht; Kersten, Martin; Windhouwer, Menzo; Waas, Florian: *Efficient Relational Storage and Retrieval of XML Documents*. International Workshop on the Web and Databases (Selected Papers), Dallas, USA, 2000

[TVBSSZ02] Tatarinov, Igor; Viglas, Stratis D.; Beyer, Kevin; Shanmugasundaram, Jayavel; Shekita, Eugene; Zhang, Chun: *Storing and Querying Ordered XML Using a Relational Database System*. Proceedings of the ACM SIGMOD Conference, Madison, USA, 2002

[WWW98] World Wide Web Consortium: *Extensible Markup Language (XML) 1.0 (Third Edition) W3C Recommendation*. http://www.w3.org/TR/REC-xml, 2004

[WWW99] World Wide Web Consortium: *XML Path Language (XPath) Version 1.0 W3C Recommendation*. http://www.w3.org/TR/xpath, 1999 (deutsche Übersetzung unter http://www.edition-w3c.de/TR/xpath)

[WWW05P] World Wide Web Consortium: *XML Path Language (XPath) Version 2.0 W3C Working Draft*. http://www.w3.org/TR/xpath20/, 2005

[WWW05Q] World Wide Web Consortium: *XQuery 1.0: An XML Query Language W3C Working Draft*. http://www.w3.org/TR/xquery/, 2005

[XJP01] The Apache Xerces Project: *Xerces Java Parser*. http://xerces.apache.org/xerces-j/, 2001

[XMACH01] Böhme, Timo; Rahm, Erhard: *XMach-1: A Benchmark for XML Data Management*. http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html, 2001

[XMARK01] Busse, Ralph; Carey, Mike; Florescu, Daniela; Kersten, Martin; Manolescu, Ioana; Schmidt, Albrecht; Waas, Florian: *XMark - An XML Benchmark Project*. http://www.xml-benchmark.org, 2001

[YAU99] Yoshikawa, Masatoshi; Amagasa, Toshiyuki; Uemura, Shunsuke: *XRel - A Path-Based Approach to Storage and Retrieval of XML documents using Relational Databases*. Proceedings of the International Conference on Database and Expert Systems Applications, Florence, Italy, 1999

[ZNDLL01] Zhang, Chun; Naughton, Jeffrey; DeWitt, David; Luo, Qiong; Lohman, Guy: *On Supporting Containment Queries in Relational Database Management Systems*. Proceedings of the ACM SIGMOD Conference, Santa Barbara, USA, 2001

[ZPR02] Zhang, Xin; Pielech, Bradford; Rundensteiner, Elke A.: *Honey, I Shrunk the XQuery! - An XML Algebra Optimization Approach*. International Workshop on Web Information and Data Management, Virginia, USA, 2002

Abbildungsverzeichnis

2.1. Ein einfaches XML-Dokument 5
2.2. Die Baumdarstellung des Datenmodells für das XML-Dokument aus Abbildung 2.1 7
2.3. Nummerierung eines XML-Baumes mit Dewey Order: Die Knoten auf jeder Ebene mit demselben Vaterelement werden der Reihe nach durchnummeriert. 21
2.4. Einfügen eines neuen Teilbaumes mit der Hilfe von Einschaltungszeichen und eines einzelnen Elementes als ersten Nachbarknoten in einen mit ORDPATH nummerierten XML-Baum. 22
2.5. Einfügen eines neuen Teilbaumes und eines einzelnen Elementes als ersten Nachbarknoten mit der Hilfe von Zwischenwerten in einen mit DLN nummerierten XML-Baum 24
2.6. Binäre Darstellung eines mit DLN-DOM nummerierten XML-Baumes 26

3.1. Das Konzept des Mediators (aus [B05]) 31
3.2. Der interne Aufbau des Mediators (aus [B05]) 32
3.3. Das relationale Schema 34
3.4. Verarbeitung eines XPath-Ausdruckes 39
3.5. Die Tabelle SQLNode 41

4.1. Die Interfaces `SQLExpression` und `SQLTableReference` und implementierende Klassen 57
4.2. Das Interface `SQLPredicate` und implementierende Klassen 58
4.3. Benchmarkergebnisse für Datensatz I jeweils mit und ohne Fremdschlüsselbeziehungen (FK) 68
4.4. Benchmarkergebnisse für Datensatz II jeweils mit und ohne die Verknüpfung per Dokumenten-ID bei Tabellen-Joins (DV) 70

Tabellenverzeichnis

2.1. Die Achsen der XPath-Anfragesprache 9
2.2. Bereichsnummerierung für das XML-Dokument aus Abbildung 2.1 . . 17
2.3. Beispiel für ein Array mit Längenangaben für die Zwischenwerte . . . 25
2.4. Erhaltung der Dokumentenordnung durch DLN-IDs 29

3.1. Beispiel für die Extraktion der Vorfahren eines Knotens aus seiner DLN-ID 44

4.1. Spezifikationen der verwendeten Datenbanken 51
4.2. Datensätze für die Benchmarks (die Werte in Klammern geben die Größen der Indizes an) 64
4.3. XPath-Anfragen und die Kardinalität der Lösungsmengen für die Benchmarks auf Datensatz I 67
4.4. XPath-Anfragen und die Kardinalität der Lösungsmengen für die Benchmarks auf Datensatz II 67

C.1. Benchmarkergebnisse für Datensatz I in ms 85
C.2. Benchmarkergebnisse für Datensatz II in ms 85

A. Inhalt der beiliegenden CD

Die dieser Diplomarbeit beiliegende CD hat folgende Ordnerstruktur:

`/bin` enthält die übersetzte Version des XMLRDB-Projektes (`xmlrdb.jar`)

`/doc` enthält die Java-Dokumentation für das XMLRDB-Projekt

`/settings` enthält Beispiel-`Properties`-Dateien für MySQL und PostgreSQL

`/src` enthält die Quelltexte des XMLRDB-Projektes

`/test` enthält die Quelltexte zu zwei JAVA-Klassen, welche exemplarisch die Funktionsweise des Projektes aufzeigen

`/thesis` enthält die PDF-Version dieser Diplomarbeit

`/udf` enthält die UDF-Quelltexte und weitere nützliche Dateien

- `/mysql` enthält neben einer `xmlrdb.def`-Datei für die Erstellung einer DLL-Bibliothek unter Windows auch die Anweisungen für das Laden der Funktionen in MySQL. In den Unterordnern befinden sich die Quelltexte für die UDFs mit DOM-DLN-ID und Prefix-DLN-ID.
- `/postgresql` enthält neben einem `Makefile` und einer `README.xmlrdb`-Datei für die Erstellung der UDF-Bibliothek auch die Anweisungen für das Laden der Funktionen in PostgreSQL. In den Unterordnern befinden sich die Quelltexte für die UDFs mit DOM-DLN-ID und Prefix-DLN-ID.

Auf der CD befindet sich außerdem noch die Datei `build.xml`, welche das bequeme Kompilieren des Projektes mittels des Tools ANT [ANT05] ermöglicht.

B. Das XPath-Modul

B.1. Verwendung des XPath-Moduls

Für das Einlesen eines XML-Dokumentes in die Datenbank bedarf es der auf der beiliegenden CD enthaltenen XMLRDB-Distribution und des Xerces-Parsers (siehe Kapitel 4.1)[1]. Informationen über die verwendete Datenbank (inklusive LogIns) und die grundsätzlichen Einstellungen des Projektes müssen über eine `Properties`-Datei angegeben werden. Für die beiden Datenbanken MySQL und PostgreSQL befinden sich zwei Beispieldateien im Verzeichnis `settings` der beiliegenden CD.

Danach kann über folgenden JAVA-Quellcode ein XML-Dokument in die Datenbank eingelesen werden:

```
import java.io.FileReader;
import xmlrdb.main.XMLDBConnection;
import xmlrdb.mapping.dln.DLNLoader;

...

String doc = "Pfad + Dateiname";
String properties_file = "Pfad + Dateiname";

XMLDBConnection xCon = new XMLDBConnection(properties_file);
DLNLoader loader = (DLNLoader) xCon.getLoader();

loader.parseDoc(new FileReader(doc), doc);
```

[1] beide müssen in einem Verzeichnis verfügbar sein, welches in der Umgebungsvariable `CLASSPATH` aufgeführt ist

Eine Anfrage auf den eingelesenen XML-Daten ist ebenso einfach durchzuführen:

```
import xmlrdb.main.XMLDBConnection;
import xmlrdb.mapping.dln.DLNXPathQuery;
import org.w3c.dom.Element;

....

String xpath = "XPath-Anfrage";

DLNXPathQuery queryEnv = new DLNXPathQuery();
queryEnv.init(xCon);

Element elem = queryEnv.executeXPath(xpath);
```

Im Verzeichnis `test` der CD befinden sich zwei kommentierte Beispiel-Klassen, welche die Funktionalität aufzeigen und als Basis für eigene Anwendungen dienen können. Bevor jedoch eine Anfrage an die Datenbank gestellt werden kann, sollten die User Defined Functions, welche für die Umsetzung von einigen XPath-Achsen benötigt werden, für die verwendete Datenbank kompiliert und danach in diese eingefügt werden.

B.2. User Defined Functions kompilieren und laden

Vor der Kompilierung sollten die richtigen UDFs von der CD ausgewählt werden. Im Verlaufe dieser Diplomarbeit wechselte das XMLRDB-Projekt von der DOM- zu der Prefix-DLN-ID (siehe Kapitel 2.4.3). Diese wird bei der Distribution als Standard verwendet und dementsprechend sollten auch die `xmlrdb.c`-Dateien aus den Unterordnern `prefix` gewählt werden, damit keine falschen Knotenmengen bestimmt werden.

Für die Übersetzung der UDF-Quelltexte als dynamisch ladbare Bibliotheken wird sowohl bei MySQL als auch bei PostgreSQL eine Quelldistribution benötigt, da nur

diese die verwendeten Headerdateien für Makros und Datentypen mitbringen. Unter Unix/Linux verläuft das Kompilieren naturgemäß etwas einfacher, da diese Betriebssysteme die entsprechende Software wie den GCC-Compiler gleich mitliefern. Die Dokumentationen der Datenbanken beschreiben für diese Systeme ausführlich das Vorgehen, so dass auf diese Quellen verwiesen wird. Die entstandene Biblliothek hat die Endung `.so` und kann nun von anderen Anwendungen dynamisch eingebunden werden.

Unter Windows zeigt sich bei der Erstellung der `.dll`-Bibliothek[2] jedoch kein einheitliches Bild. Die Quelldistribution von PostgreSQL ist darauf ausgelegt, mit Hilfe der OpenSource-Pakete MSYS und MinGW [MGW05] kompiliert und verlinkt zu werden. Diese bilden zusammen eine einfache GNU-POSIX-Umgebung für Windows, welche aus Unix bekannte Werkzeuge wie `make` mitliefert und native Windows-Anwendungen erzeugen kann. Da für das Linken der PostgreSQL-UDFs kompilierte Bibliotheken benötigt werden, muss zuerst die gesamte Quelldistribution übersetzt werden[3].

PostgreSQL bietet eine einfache Möglichkeit, eigene Erweiterungen in die Datenbank einzubringen. Dafür existiert das Verzeichnis `contrib` der Quelldistribution, welches auch die mitgelieferten Erweiterungen enthält. Dort muss ein neues Verzeichnis `xmlrdb` für dieses Projekt erstellt und die Dateien aus dem Ordner `postgresql` dort hinein kopiert werden. Zuletzt folgt noch die ausgewählte `xmlrdb.c`-Datei und nun kann die UDF durch Aufruf von `make` in diesem Verzeichnis kompiliert werden. Die erstellte DLL muss schließlich in den Unterordner `lib` der PostgreSQL-Server-Installation kopiert werden.

MySQL dagegen verlangt nach dem Microsoft-Compiler Visual C++ [MVC05]. Dieser kann momentan in der aktuellen Version 2005 als kostenlose Express Edition bezogen werden, deren eingeschränkter Funktionsumfang für ein Kompilieren der UDF aber ausreichend ist. Ebenfalls gebraucht wird das Windows Platform SDK, welches einige benötigte Bibliotheken enthält. Auch dieses kann kostenlos von der Homepage des Herstellers bezogen werden. Bei der Verwendung des Microsoft-Compilers ist es jedoch notwendig, im Quelltext bei Zahlenangaben für 64Bit-Integer-Typen das Suffix von `ll` unter GCC auf `i64` zu ändern.

Die Windows-Quellpakete von MySQL enthalten einen Ordner namens `examples/udf_example`, welcher ein UDF-Projekt mit einigen Beispielen enthält.

[2] Dynamic-Link Library
[3] Dies funktioniert analog zu Unix/Linux, wofür ausführliche Beschreibungen mitgeliefert werden.

Dieses kann als Basis für eigene Funktionen dienen. Auf der Homepage von MySQL findet sich in der Bug-Datenbank unter Eintrag #334 auch eine ausführliche Anleitung, welche Bibliotheken und Header-Dateien unter Visual C++ an welcher Stelle eingetragen werden müssen, damit eine DLL erstellt werden kann.

Der Ordner `mysql` der beiliegenden CD enthält neben den UDF-Quelltexten auch die benötigte `.def`-Datei. Diese teilt dem Compiler mit, welche Funktionen die DLL nach außen bereitstellen soll. Nachdem das Visual C++-Projekt fertig eingerichtet ist, kann mit dem Befehl `Build Solution` die dynamische Bibliothek erzeugt werden, welche nun nur noch in einen Systemordner wie `system32` kopiert werden muss. Damit MySQL aber auch die UDFs annimmt, muss die Datenbank vorher mit der Einstellung `allow-suspicious-udfs`[4] dazu überredet werden. Ansonsten blockiert der Server das Laden von UDFs, welche keine der optionalen `xxx_init()` oder `xxx_deinit()`-Funktionen zusätzlich zu der eigentlichen Hauptfunktion `xxx()` definieren.

Das Einfügen der Funktionen in die Datenbanken ist dann schnell geschehen: Die beiden Dateien `xmlrdb.sql` in den Verzeichnissen `mysql` und `postgresql` der beiliegenden CD enthalten die Aufrufsyntax, um die nötigen UDFs den Datenbanken bekannt zu machen. Voraussetzung ist, dass der User genügend Rechte für das Einfügen von User Defined Functions in das RDBMS besitzt und dieses auf die Bibliothek zurückgreifen kann[5].

[4] Diese Option muss entweder beim Starten des Servers als Kommandozeilenparameter übergeben werden oder im MySQL Configuration File unter den `[mysqld]`-Serveroptionen eingetragen werden.

[5] MySQL quittiert eine nicht gefunden Bibliothek gerne mit einem Absturz. In den SQL-Dateien muss deshalb der komplette Name der erstellten Datei angegeben werden; dessen Endung variiert je nach System.

C. Benchmark-Resultate

Query	MySQL	MySQL + FK	PostgreSQL	PostgreSQL + FK
1	6553	7369	5022	5284
2	6753	6713	28888	29219
3	4388	4372	13469	13572
4	4244	5441	6772	6809
5	3997	3972	5178	4872
6	7344	9144	114575	114290
7	6534	7700	4878	4828
8	6331	7544	112185	112257
9	4903	5069	6794	6891
10	11037	11019	9141	9200
11	36297	35947	2647	2519

Tabelle C.1.: Benchmarkergebnisse für Datensatz I in ms

Query	MySQL	MySQL + DV	PostgreSQL	PostgreSQL + DV
12	1841	1216	43647	9494
13	2534	2425	8797	8575
14	5725	5722	8918	8897
15	6087	7294	9157	8953
16	28281	12281	39694	48259

Tabelle C.2.: Benchmarkergebnisse für Datensatz II in ms

D. Danksagung

Ich möchte mich bei Prof. Erhard Rahm und Timo Böhme für die Bereitstellung und Betreuung dieser Diplomarbeit und die Unterstützung und Anregungen während der Erstellung bedanken.

E. Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Ort	Datum	Unterschrift