

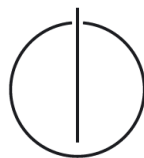


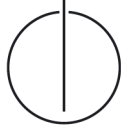
Fakultät für Informatik
der Technische Universität München

Diplomarbeit in Informatik

**Pfad-Indexe im Pathfinder
XQuery Compiler**

Alexander Gafriller





Fakultät für Informatik
Technische Universität München
Lehrstuhl für Datenbanken



Diplomarbeit in Informatik

Pfad-Indexe im Pathfinder XQuery Compiler

Path Summaries for the Pathfinder
XQuery Compiler

Alexander Gafriller

17. Dezember 2007

Aufgabensteller: Prof. Dr. Torsten Grust
Betreuer: Jan Rittinger, M.Sc.
Manuel Mayr, Dipl.-Inf.

Erklärung

Ich versichere, diese Diplomarbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet zu haben.

Alexander Gafriller,
17. Dezember 2007

Inhaltsverzeichnis

1	Einleitung	1
1.1	<i>Pathfinder</i>	2
1.2	Thema und Zielsetzung	3
1.3	Aufbau der Diplomarbeit	3
2	Grundlagen	5
2.1	XML	5
2.2	XPath	7
2.2.1	Knotentypen von XPath	7
2.2.2	Lokalisierungspfade	8
2.2.3	Achsen	8
2.2.4	Knotentests	10
2.2.5	Abkürzungen	10
3	Kodierung von XML	11
3.1	Begriffe	11
3.2	Tabellen-basierte Repräsentation	12
3.3	Beispiel	14
3.4	Kodierung der XPath-Achsen	15
4	Data Guides	17
4.1	Einführung	17
4.2	Definitionen	18
4.3	Partitionierung von XPath-Pfaden	20
5	Optimierung	25
5.1	Einführendes Beispiel	25
5.2	Optimierung von XPath-Pfaden	26
5.2.1	Probleme mit XPath-Achsen	26
5.2.2	Achsenrichtung	28
5.2.3	Resultierende XPath-Achse	29

5.2.4	Algorithmus	34
5.3	Zusatzinformationen von XML-Dokumenten	38
5.3.1	<i>Data Guide</i> -Attribute	38
5.3.2	Optimierungen durch min/max	39
5.3.3	Optimierung durch <i>count</i>	41
6	Experimente	42
6.1	XMark-Queries	42
6.2	Komponenten der Testumgebung	44
6.2.1	Computer	44
6.2.2	Datenbanksystem	45
6.3	Test-Vorbereitung	47
6.4	Indizes	49
6.5	Auswertung	50
6.5.1	Einfluss der <i>Data Guides</i>	50
6.5.2	Einfluss des <i>Statistical Views</i>	52
6.6	Beobachtungen	53
7	Fazit	61
	Literaturverzeichnis	62
	Abbildungsverzeichnis	65
	Tabellenverzeichnis	67
A	Shredder	68
A.1	Aufbau des Shredders	68
A.2	Ausgabe des <i>Shredders</i>	69
B	XMark-Queries	70
C	Testzeiten der XMark-Queries	76
D	Algebra-Pläne von Query Q06	80

Kapitel 1

Einleitung

Die Einsatzgebiete von XML sind sehr vielfältig. Sie reichen vom Datenaustausch zwischen verschiedenen Anwendungen über die Verwendung als Basisformat für weitere Ausgabeformate bis hin zur Bioinformatik (E.C05). Je nach Anwendungsgebiet sind dabei die Datenmengen, welche zur Verarbeitung kommen, unterschiedlich groß. Für diese Diplomarbeit sind Datenmengen von mehreren Megabyte bis über Gigabytegröße hinaus von Interesse. Solche Datenmengen fallen in vielen Bereichen der Informatik an: so befasst sich die Europäische Zentralbank mit der Umsetzung eines einheitlichen bargeldlosen Euro-Zahlungsverkehrsraums (Single Euro Payments Area, SEPA) (ezb06). Ein weiterer Bereich, in dem XML-Daten von bis zu Terabytegröße anfallen, ist die Bioinformatik (FGE01).

Um die benötigten Informationen in XML-Dokumenten zu finden, wurde vom World Wide Web Consortium (W3C) eine Abfragesprache entwickelt: XQuery (BCF07). Mit ihrer Hilfe haben wir die Möglichkeit, Informationen aus XML-Dokumenten zu extrahieren. Dazu wird ein XML-System benötigt, welches die XQuery-Anfragen interpretiert und das Ergebnis zurück liefert. Ein alternativer Ansatz nutzt die Abbildung von XML-Dokumenten auf ein relationales Datenbank-Managementsystem (RDBMS) sowie die Transformation von XQuery nach SQL - oder in eine andere Sprache, die von RDBMS interpretiert werden kann. Dieses Konzept wird mit *Pathfinder* verfolgt.

Die Auswahl bestimmter Knoten eines XML-Dokumentes erfolgt durch XPath (CD99). XPath wird auch von XQuery verwendet und beeinflusst dessen Laufzeit. Eine Idee zur schnelleren Auswertung von XQuery-Ausdrücken wäre in der Beschleunigung der XPath-Abfragen. Dieser Zeitgewinn soll durch eine geschickte Partitionierung von XML erfolgen. Das zugrunde liegende Konzept für eine verbesserte Auswertung von XPath nennen wir *Data Guides*. In der vorliegenden Arbeit wird dieses Konzept für die auf das RDBMS abgebildeten XML-Dokumente erarbeitet. *Data Guides* basieren auf der di-

rekten Adressierung äquivalenter Teile von XML sowie dem schnellen Zugriff auf den partitionierten Knoten durch das RDBMS.

1.1 *Pathfinder*

*Pathfinder*¹, ist ein an der Technischen Universität München entwickelter XQuery-Compiler, welcher die Transformation von XQuery nach SQL ermöglicht. Somit kann ein RDBMS auch XQuery-Abfragen auf ein XML-Dokument simulieren. Hierzu müssen wir jedoch das XML-Dokument in eine für das RDBMS darstellbare Form bringen. Besonders bei größeren XML-Dokumenten bietet diese Art der Verarbeitung eine schnellere Ausführungszeit, als dies auf XML-Systemen mit XQuery der Fall wäre (Mon).

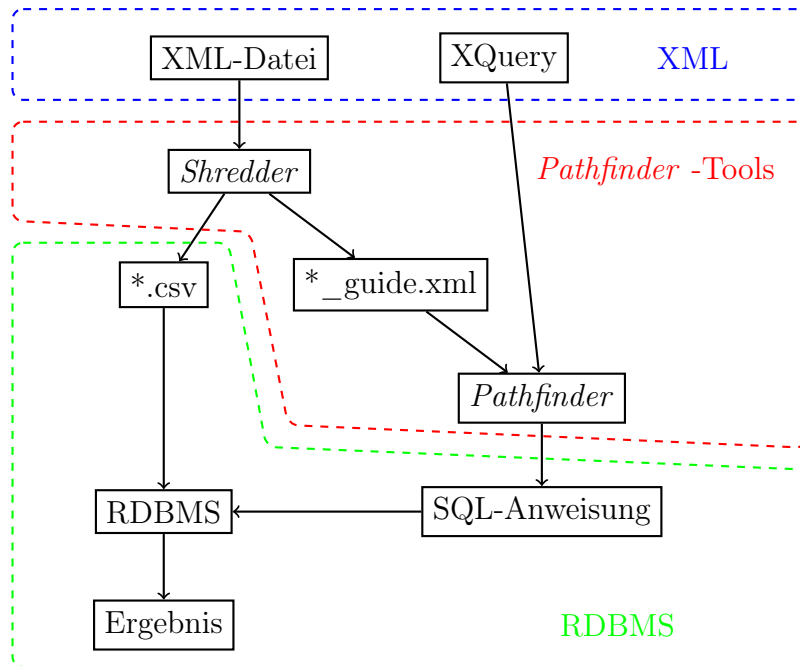


Abbildung 1.1: Konvertierung von XML/XQuery nach RDBMS/SQL

In Abbildung 1.1 sind die Schritte skizziert, mit denen eine XML-Datei für das RDBMS aufbereitet und XQuery nach SQL kompiliert wird. Der *Shredder* (siehe Anhang A) wandelt die Darstellung eines XML-Dokumentes in eine tabellen-basierte Repräsentation für das RDBMS um, damit wir die Daten ohne Informationsverlust verarbeiten können. Zusätzlich werden die

¹<http://pathfinder-xquery.org/>

für die Optimierung benötigten Informationen generiert und in die Datei `*_guide.xml` geschrieben. *Pathfinder* verwendet einen Teil der Resultate des *Shredders*, um das Konzept der *Data Guides* zu verwirklichen und um die XQuery-Abfragen zu optimieren. Das Resultat ist ein ausführbarer SQL-Code. Diese SQL-Anweisung können wir auf dem RDBMS ausführen und erhalten das gewünschte Ergebnis.

1.2 Thema und Zielsetzung

Für die Adressierung bestimmter Teile eines XML-Dokumentes verwendet XQuery das vom W3C entwickelte XPath (CD99). XPath ermöglicht die Adressierung ausgewählter Knoten eines XML-Dokumentes, welche die benötigten Informationen für die Weiterverarbeitung enthalten. Das Ermitteln dieser gewünschten Knoten durch XPath kann, abhängig von der verwendeten DTD des XML-Dokumentes, lange Pfadauswertungen beinhalten.

Das Ziel der vorliegenden Arbeit ist die Optimierung der Adressierung von XPath-Pfaden, um dadurch eine schnellere Auswertung derselben zu ermöglichen. Zu diesem Zweck werden wir ein Konzept erarbeiten, welches wir *Data Guide* nennen. Das Konzept der Indizierung von XML-Daten, welche in einem RDBMS abgespeichert sind, wurde bereits in (PCS⁺04) dargestellt. Die dahinter stehende Idee sind ORDPATHs (OOP⁺04), welche die Zuweisung von eindeutigen Bezeichnern zu den hierarchisch strukturierten Daten von XML beinhalten. Dieses Modell werden wir in der vorliegenden Diplomarbeit auf die Kodierung von *Pathfinder* übertragen.

1.3 Aufbau der Diplomarbeit

Die Diplomarbeit ist in fünf Teile gegliedert, in welchen wir schrittweise die Verbesserung der Pfadauswertung für XPath erarbeiten. In Kapitel 2 gehen wir kurz auf die Grundlagen von XML und XPath ein, um damit ein grundlegendes Vokabular zu schaffen. Beide Themen werden nur oberflächlich aufgegriffen da sie bereits in vielen Büchern ausgiebig behandelt wurden.

In Kapitel 3 befassen wir uns mit der Konvertierung eines XML-Dokumentes in die tabellen-basierte Repräsentation des RDBMS. Zu Beginn werden die benötigten Begriffe eingeführt, um mit ihnen in einem zweiten Schritt die relationale Repräsentation zu erarbeiten. Den Abschluss bildet die Kodierung der XPath-Achsen anhand der erarbeiteten Repräsentation.

Im Anschluss an einige allgemeine Bemerkungen zum Thema *Data Guide* werden wir in Kapitel 4 das Konzept der *Data Guides* definieren. Am

Ende des Kapitels schließen wir die Lücke zwischen den *Data Guides* und XML.

Das Kapitel 5 bildet den Kern der Diplomarbeit. Zu Anfang wird das Annotieren von *Data Guides* an Lokalisierungsschritten beschrieben. Es folgen die Bedingungen, die wir bei der Optimierung beachten müssen. Nachdem wir diese Bedingungen beschrieben haben, geben wir einen Algorithmus für die Optimierung von XPath Ausdrücken an. Des Weiteren werden wir Vereinfachungen am algebraischen Plan von *Pathfinder* kennen lernen, die durch das Konzept der *Data Guides* ermöglicht werden.

Um Auswirkungen der *Data Guides* auf die Laufzeiten der XQuery zu beobachten, werden wir in Kapitel 6 mehrere Experimente durchführen. Vorher beschreiben wir noch den verwendeten XMark-Benchmark sowie die Umgebung, in der die Tests ausgeführt werden. Abschließend visualisieren wir die Ergebnisse der Testläufe und arbeiten Vor- und Nachteile der *Data Guides* heraus.

Kapitel 2

Grundlagen

In diesem Kapitel wollen wir einen kurzen Einblick in XML und XPath geben. Beide Themen sind grundlegend für die Diplomarbeit und aus diesem Grund betrachten wir die wichtigsten Teile, um eine Grundlage für die weitere Arbeit zu schaffen.

2.1 XML

Extensible Markup Language (XML) ging aus der Standard Generalized Markup Language (SGML) hervor. Sie ist eine Meta-Markup-Sprache - was bedeutet, die Benutzer können die Tags und Elemente selber benennen - und dient ausschließlich der Strukturierung von Textdokumenten oder Datenmengen. XML-Dokumente enthalten immer reinen Text und können deshalb mit jedem beliebigen Editor oder Programm geöffnet werden, welches in der Lage ist Textdateien zu verarbeiten.

Beispiel 2.1. *Kleines XML-Dokument.*

```
<computer anbotbeginn='03.08.2006' anbotende='09.08.2006'>
  <prozessor>Intel 3,0 GHz</prozessor>
  <hauptspeicher>1024MB RAM</hauptspeicher>
  <massenspeicher>
    <festplatte>Maxtor 200 GB</festplatte>
    <festplatte>Maxtor 300 GB</festplatte>
  </massenspeicher>
</computer>
```

Der Aufbau eines XML-Dokumentes wird in Beispiel 2.1 skizziert. Die Bestandteile, aus denen ein XML-Dokument besteht, sind Knoten, Elemente, Tags, Text und Attribute. Bei einem Tag wird zwischen Start- und End-Tag

unterschieden. Ein Start-Tag beginnt mit '<' und endet mit '>', wohingegen ein End-Tag mit '</' beginnt, aber wie das Start-Tag endet. Beispiele dafür sind <computer> und </computer>. Elemente werden durch Start- und End-Tags eingeschlossen. Im Beispiel kommen mehrere Elemente wie `computer`, `prozessor`, `Hauptspeicher` usw. vor. Zeichendaten sind die Nutzinformation, die im XML-Dokument vorkommt. Dies ist der Text zwischen Start- und End-Tags. Elemente können Attribute enthalten. Attribute bestehen aus einem Namen und einem Wert und werden im Start-Tag der Elemente angegeben. Der Wert wird zwischen einfachen oder doppelten Anführungszeichen angegeben und mit einem Gleichheitszeichen vom Namen getrennt. Im Beispiel hat das Element `computer` zwei Attribute `angebotbeginn` und `angebotende`. Aus dem Beispiel wird auch der schematische Aufbau ersichtlich.

Ein wohlgeformtes XML-Dokument besitzt genau ein Element, welches keinen Elternknoten besitzt. Dieses Element wird Wurzelement oder Dokumentelement genannt. Eine weitere wichtige Eigenschaft ist, dass die Start- und End-Tags nicht ineinander geschachtelt werden dürfen, d.h. sie müssen ebenengetreu verschachtelt werden. Zudem darf ein Element keine zwei Attribute mit gleichem Namen besitzen.

Die Datenstruktur eines XML-Dokumentes kann auch als Baumstruktur dargestellt werden. Als Wurzel wird das äußerste Element, das Wurzelement, verwendet und die enthaltenen Knoten werden als Kinder betrachtet. Dabei können die Kindknoten wiederum eines oder mehrere Elemente beinhalten. Auch Attribute können als Kinder von Elementen, welche die Attribute beinhalten, dargestellt werden. Als Veranschaulichung dient Beispiel 2.1, welches in Abbildung 2.1 als Baumdiagramm dargestellt ist.

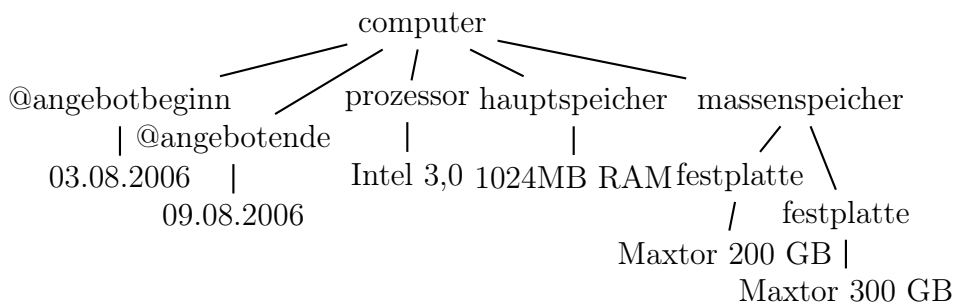


Abbildung 2.1: Baumstruktur eines XML-Dokumentes.

2.2 XPath

XPath wird verwendet um Teile eines XML-Dokumentes mit bestimmten Kriterien auszuwählen und diese dann anderen Konzepten wie XPointer oder XSLT zur Verfügung zu stellen. Im Nachfolgenden wird näher auf die Konstrukte von XPath eingegangen, wobei die Pfadauswertung am genauesten betrachtet wird.

2.2.1 Knotentypen von XPath

XML-Dokumente bestehen aus Knoten, welche wiederum Knoten enthalten können. Wenn diese Dokumente grafisch dargestellt werden, ergibt sich die Struktur eines Baumes. In XPath können die Knoten einen der folgenden Typen haben:

- **Wurzel (root)**: Jedes Dokument hat genau einen Wurzelknoten, welcher einen Kommentar-Knoten, einen Verarbeitungsanweisungs-Knoten und ein Wurzelement enthält. Der Knoten vom Typ Verarbeitungsanweisung beinhaltet die Anweisungen für Kommentare und Elemente außerhalb des Wurzelementes. Das Wurzelement selbst besitzt keinen Elternknoten.
- **Element (element)**: Ein Element besteht aus einem Namen, Namensraum-URI, Elternknoten, einer Liste von Kindknoten und Attributen. Attribute und Namensräume werden nicht als Kinder des Elementes angesehen.
- **Attribut (attribute)**: Das Attribut hat einen Namen, einen Namensraum-URI, einen Wert und einen Elternknoten. Obwohl Elementknoten Eltern von Attributen sind, gilt nicht, dass Attributknoten Kinder desselben sind.
- **Text**: Besteht aus einem Bereich von Zeichen oder Text, der durch andere Knotentypen wie z.B. Verarbeitungsanweisungen oder Kommentare eingegrenzt ist.
- **Namensraum (namespace)**: Knoten von diesem Typ haben wie Attribute einen Elternknoten, werden aber nicht als Kinder dessen betrachtet. Sie geben den Namensraum des Elternknotens und dessen Nachfahren, sofern diese keine eigenen Namensräume definieren, an.
- **Verarbeitungsanweisung (processing-instruction)**: hat einen Elternknoten - aber keine Kindknoten - ein Ziel und Daten.

- **Kommentar (comment)**: Kommentare besitzen einen Elternknoten aber keine Kind-Knoten. Der Wert des Kommentars ist die Zeichenkette innerhalb den Begrenzern '`<!--`' und '`-->`'.

2.2.2 Lokalisierungspfade

Lokalisierungspfade können wir uns wie Pfadangaben unter UNIX vorstellen. Sie bestehen aus mehreren Lokalisierungsschritten, die hintereinander gestellt werden. Ein Lokalisierungsschritt besteht wiederum aus einer Achse und einem Knotentest, der durch zwei Doppelpunkte getrennt angegeben wird.

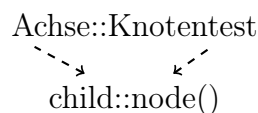


Abbildung 2.2: Aufbau von Lokalisierungsschritten

Die Achse bestimmt die Richtung, in der Knoten gesucht werden. Der Startpunkt ist ein vorgegebener Knoten: der Kontextknoten. Der Knotentest wählt die entsprechenden Knoten entlang der Achse aus, die einen bestimmten Typ haben. Die Lokalisierungspfade entstehen, indem Lokalisierungsschritte durch einen Schrägstrich '/' getrennt angegeben werden. Dabei bestimmt der weiter links stehende Lokalisierungsschritt den oder die Kontextknoten des nächsten Schrittes.

`/descendant-or-self::node()/@id`

Abbildung 2.3: Absoluter Lokalisierungspfad

Beginnt ein Lokalisierungspfad mit einem '/', so nennt man diesen Pfad „absoluter Pfad“. Absolute Lokalisierungspfade beginnen immer beim Wurzelement mit der Pfadauswertung. Weiterhin können Lokalisierungspfade Prädikate enthalten, die eine detailliertere Auswahl von Knoten zulassen.

2.2.3 Achsen

Ausgehend von einem ausgezeichneten Knoten, dem Kontextknoten, können wir uns in XPath auf 13 verschiedenen Achsen bewegen. Diese Achsen werden nachfolgend aufgelistet und näher beschrieben.

1. **child**: Die Kinder des Kontextknotens. Dabei gibt es einige Sonderfälle zu beachten. Es können nur der Wurzelknoten und die Elementknoten Kinder haben, wobei Attribut- oder Namensraumknoten keine Kinder sind und keine Kinder haben, obwohl diese Elternknoten besitzen.
2. **descendant**: Dies sind alle Nachfahren des Wurzelknotens oder Elementknotens. Anders ausgedrückt, sind es die Kinder, Kinder der Kinder usw. Wiederum bilden die Knoten vom Typ Attribut und Namensraum eine Ausnahme, sie sind nicht enthalten und haben auch keine Nachfolger.
3. **descendant-or-self**: Wie **descendant** nur inklusive des Kontextknotens.
4. **parent**: Elternknoten des Kontextknotens, d.h. der Knoten, der den Kontextknoten enthält. Abgesehen vom Wurzelknoten besitzt jeder Knoten einen Elternknoten.
5. **ancestor**: Das sind die Vorfahren des Kontextknotens, oder anders ausgedrückt: der Elternknoten, Elternknoten des Elternknotens usw.
6. **ancestor-or-self**: Wie **ancestor**, nur dass der Kontextknoten selbst auch enthalten ist.
7. **following**: Das sind alle Knoten die nach dem Kontextknoten beginnen und weder Attribut- und Namensraumknoten noch Kinder des Kontextknotens selbst sind.
8. **preceding**: Wie **following**, nur dass die Knoten vor dem Start des Kontextknotens enden und keine Vorfahren desselben sind.
9. **following-sibling**: Sind alle Knoten die in der Reihenfolge nach dem Kontextknoten kommen und den gleichen Elternknoten besitzen. Ausgenommen sind wieder die Attribut- und Namensraumknoten, welche keine Geschwister besitzen.
10. **preceding-sibling**: Wie **following-sibling**, nur dass die Knoten in der Reihenfolge vor dem Kontextknoten stehen müssen.
11. **attribute**: Die Attribute des Kontextknotens. Nur Elementknoten können Attribute enthalten.
12. **namespace**: Gibt alle gültigen Namensräume eines Kontextknotens vom Typ **element** an. Andernfalls ist die Achse leer.
13. **self**: Der Kontextknoten selbst.

2.2.4 Knotentests

Wie bereits erwähnt besteht der Lokalisierungsschritt aus einer Achse und einem Knotentest, die durch zwei Doppelpunkte getrennt sind. Der Knotentest dient zur Einschränkung der durch die Achse ausgewählten Knoten. Insgesamt gibt es sieben verschiedene Knotentests, welche in der Tabelle aufgelistet sind.

1. **name**: Es werden alle Elementknoten mit dem angegebenen Namen ausgewählt. Bei der Attributachse werden alle Attributknoten, die dem Namen `name` entsprechen, ausgewählt.
2. **prefix:***: Liefert alle Elementknoten zurück, die denselben Namensraum, wie im Prefix angegeben, haben. Bei der Attributachse werden alle Attribute zurückgegeben, die dem Namensraum von `prefix` entsprechen.
3. **comment()**: Findet alle Kommentare.
4. **text()**: Findet die Textknoten.
5. **processing-instruction()**: Es werden alle Verarbeitungsanweisungen entlang der Achse ausgewählt.
6. **node()**: Wählt alle Knoten jeglichen Typs aus.
7. *****: Wählt alle Elementknoten unabhängig vom Namen aus. Bei der Attribut- bzw. Namensraumachse werden alle Attribute bzw. Namensräume ausgewählt.

2.2.5 Abkürzungen

Um die viel verwendeten Achsen und Knotentests kürzer ausdrücken zu können, werden Abkürzungen eingeführt. Diese Schreibweisen werden zusammen mit ihren eigentlichen Bedeutungen in Tabelle 2.1 aufgelistet.

Abkürzung	Beschreibung
.	Kontextknoten
..	Elternknoten
name	Kindelemente mit entsprechenden Namen
//	Entspricht der Achse <code>descendant-or-self</code>
@name	Attribut mit angegebenen Namen

Tabelle 2.1: Abkürzungen von Lokalisierungsschritten

Kapitel 3

Kodierung von XML

In diesem Kapitel werden wir die Kodierung von XML-Dokumenten besprechen, die es uns ermöglicht, XML auf ein relationales Datenbankmanagementsystem (RDBMS) abzubilden und XQuery-Abfragen darauf auszuwerten. Um Daten in einem RDBMS abspeichern zu können, müssen diese in Form von Tabellen vorliegen. Da wir aber bei XML-Dokumenten die Namen der Tags selbst aussuchen können - und es deshalb eine unendliche Vielfalt gibt - müssen wir uns ein geeignetes Konzept erarbeiten um die Informationen abspeichern zu können. Weiterhin ist es wichtig, dass wir alle XPath-Achsen von jedem beliebigen Kontextknoten aus darstellen können.

3.1 Begriffe

Um die tabellen-basierte Repräsentation von XML-Dokumenten beschreiben zu können, müssen wir zuerst einige Konzepte und Begriffe erklären. Am Anfang werden wir die Traversierung von Binärbäumen betrachten. Es gibt verschiedene Arten, jedoch werden wir nur die *pre-order* und die *post-order* betrachten. Bei der *pre-order* wird als erstes die Wurzel, anschließend der linke Teilbaum und zum Schluss der rechte Teilbaum durchlaufen. Die Zuweisung der Werte an einen Knoten v werden wir mit $pre(v)$ schreiben. Bei der *post-order* wird als erstes der linke Teilbaum und dann der rechte Teilbaum durchlaufen. Zum Schluss wird die Wurzel betrachtet. Die Werte die dem Knoten v bei dieser Traversierung zugewiesen werden, stellen wir mit $post(v)$ dar. In Abbildung 3.1(a) sehen wir die Zuordnung der *pre/post*-Werte an die Knoten. Links oben sind die *pre*-Werte annotiert und rechts unten die *post*-Werte.

Des weiteren benötigen wir die Anzahl der Knoten im Teilbaum, der sich unter jedem Knoten v befindet. Diese Wertzuweisung stellen wir mit $size(v)$

dar. Dabei wird der Knoten, für den die *size* bestimmt wird, nicht mitgezählt. Beispielsweise gilt für das Element *f* in Abbildung 3.1(b) $size(f) = 4$, und die *size* für jedes Blatt ist gleich 0. Die *size*-Werte werden wir bei den Knoten oben rechts notieren.

Damit wir die Kinder- und Elternknoten von einem Kontextknoten bestimmen können, benötigen wir noch weitere Informationen. Aus diesem Grunde speichern wir noch zusätzlich das Niveau (*level*) der Knoten, die sie innerhalb des Baumes haben, ab. Das Niveau ist die Anzahl der Kanten, die durchlaufen werden müssen, um vom Knoten zum Wurzelement zu gelangen, und sei durch $level(v)$ definiert. In Abbildung 3.1(b) sehen wir das Niveau am rechten Rand abgebildet.

Wie wir bereits im Abschnitt 2.2.2 gesehen haben, besteht ein Lokalisierungsschritt eines XPath-Ausdrucks aus einer Achse und einem Knotentest. Damit wir die Knoten eines XML-Dokumentes unterscheiden können, müssen wir den Typ abspeichern. In *Pathfinder* unterscheiden wir sechs Arten: Elemente, Attribute, Textknoten, Kommentare, Verarbeitungsanweisungen und das Dokument. In Zukunft werden wir diese Unterscheidung anhand der Abkürzung $kind(v)$ machen, die jedem Knoten ein Element aus der Menge $\{\text{elem}, \text{attr}, \text{text}, \text{comm}, \text{pi}, \text{doc}\}$ zuweist.

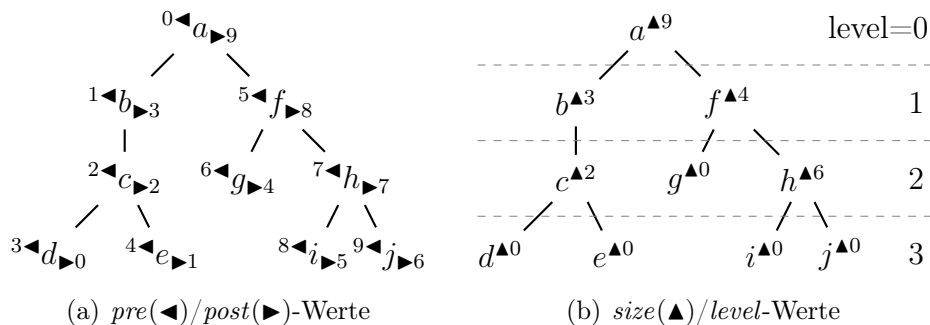


Abbildung 3.1: *pre*, *post*, *size* und *level*-Werte für ein XML-Dokument

3.2 Tabellen-basierte Repräsentation

Wir haben bereits im vorhergehenden Abschnitt gesehen, dass XML-Dokumente als Bäume dargestellt werden können. Diese Darstellungsform werden wir ausnutzen, um die Kodierung einzuführen. Eine geeignete Kodierung zur Darstellung der XPath-Achsen basiert auf den Werten *pre* und *post* der Knoten (GT04). Um jedoch die Optimierungen des RDBMS besser ausnutzen zu können, verwenden wir in *Pathfinder* *pre/size/level*. Dies bereitet keinen

Unterschied für die Funktionalität, weil *post* auch durch *size* und *level* ausdrückbar ist (BMR05).

$$post = pre + size - level$$

Dies sind auch schon die ersten drei Werte für die tabellen-basierte Repräsentation der XML-Dokumente.

Zudem benötigen wir noch die Unterscheidung der Knotentypen. Das machen wir mit *kind*, welche Eigenschaft wir im vorhergehenden Abschnitt eingeführt haben und die keiner weiteren Erklärung bedarf.

Die bis jetzt vorhandene tabellen-basierte Repräsentation kann bereits alle XPath-Achsen darstellen, jedoch noch keine Nutzinformationen abspeichern. Außerdem kann keine Unterscheidung zwischen den Knoten des selben Typs gemacht werden. Um das zu realisieren werden wir nachfolgend zwei weitere Spalten zu der Tabelle hinzufügen. Zuerst wenden wir uns dem Abspeichern von Tags zu: um diese abspeichern zu können führen wir die Spalte *name* in der Tabelle ein. Jedoch nicht jeder Typ hat einen Tag, so bleibt bei Wurzelementen, Kommentaren und Texten diese Spalte leer. Bei dem Elementknoten wird der Tagname, bei den Attributen und Verarbeitungsanweisungen die jeweiligen identifizierenden Namen eingetragen.

Zum Schluss müssen wir noch die Nutzinformationen abspeichern. Diese werden wir durch die Spalte *value* realisieren. Bei einem Text- und Kommentarknoten wird der gesamte Inhalt abgespeichert. Wurzelemente haben auch keinen *value* Wert, jedoch in *Pathfinder* wird bei diesem Knoten der Dateiname zugeordnet und in der Spalte *value* eingetragen. Bei Attributen wird der String nach dem Gleichheitszeichen in *value* abgelegt. Elementknoten hingegen haben in dieser Spalte keinen Wert. In Tabelle 3.1 werden noch einmal die einzelnen Spalten der tabellen-basierten Repräsentation von XML-Dokumenten zusammen mit einer Beschreibung als Überblick aufgelistet.

Name	Beschreibung
<i>pre</i>	<i>pre</i> -Wert des Knotens
<i>size</i>	Anzahl der Knoten im Teilbaum unter dem jeweiligem Knoten
<i>level</i>	Niveau des Knotens im Baum
<i>kind</i>	Typ des Knotens
<i>value</i>	Text
<i>name</i>	Tagname der Knoten

Tabelle 3.1: Tabellen-basierte Repräsentation der XML-Dokumente für ein RDBMS

3.3 Beispiel

Für das Beispiel einer Kodierung verwenden wir das XML-Dokument, welches in Abbildung 3.2 dargestellt und unter dem Dateinamen 'doc2.xml' abgespeichert ist. Im Baum sehen wir bereits die Werte *pre/size/level* eingetragen. Wie im Abschnitt 3.1 beschrieben, befinden sich links oben die *pre*-Werte, rechts oben die *size*-Werte und am rechten Rand die *level*-Werte.

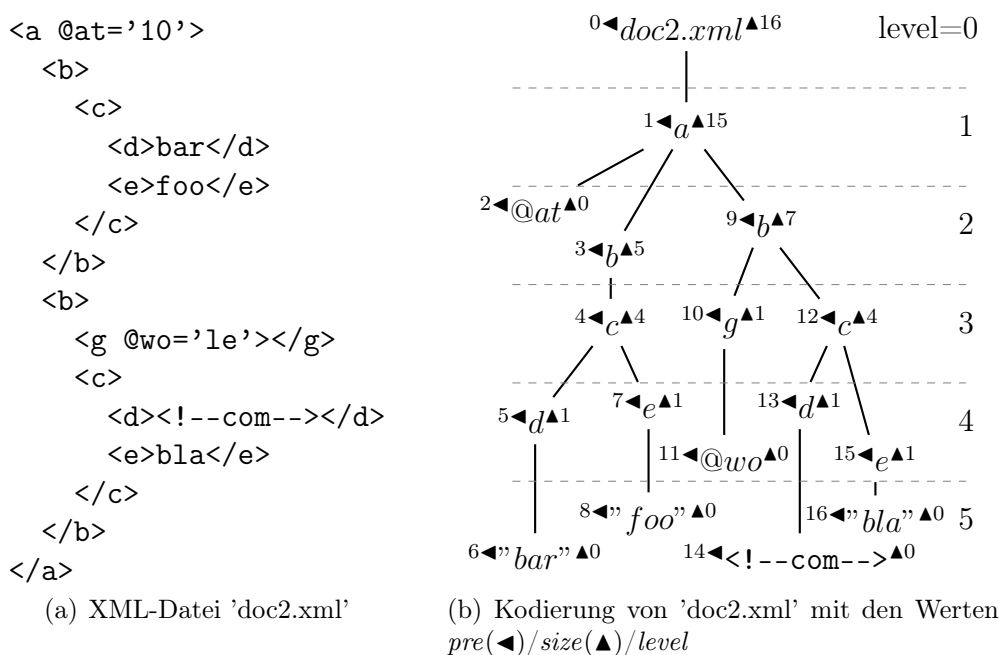


Abbildung 3.2: Baumstruktur eines XML-Dokumentes.

Wie wir sehen sind die neu hinzugekommenen Textknoten im Baum als Blätter dargestellt. Nicht nur Textknoten sondern auch Kommentare und Verarbeitungsanweisungen stellen die Blätter des Baumes dar, da sie keine Kinder besitzen können. Die Attribute dagegen sind auf dem gleichen *level* wie die Kinder der Elementknoten. Allerdings bilden sie eine getrennte Achse. Des weiteren haben wir den Dokumentknoten als Wurzel im Baum eingefügt.

Bilden wir nun das XML-Dokument aus Abbildung 3.2(a) auf die relationale Repräsentation ab, welche wir im Abschnitt 3.2 erarbeitet haben. Das Ergebnis sehen wir in der Tabelle 3.2.

<i>pre</i>	<i>size</i>	<i>level</i>	<i>kind</i>	<i>name</i>	<i>value</i>
0	16	0	doc	-	'doc2.xml'
1	15	1	elem	a	-
2	0	2	attr	at	'10'
3	5	2	elem	b	-
4	4	3	elem	c	-
5	1	4	elem	d	-
6	0	5	text	-	'bar'
7	1	4	elem	e	-
8	0	5	text	-	'foo'
9	7	2	elem	b	-
10	1	3	elem	g	-
11	0	4	attr	wo	'le'
12	4	3	elem	c	-
13	1	4	elem	d	-
14	0	5	comm	-	'com'
15	1	4	elem	e	-
16	0	5	text	-	'bla'

Tabelle 3.2: Tabellen-basierte Repräsentation des XML-Dokumentes aus Abbildung 3.2(a)

3.4 Kodierung der XPath-Achsen

Nachdem wir im vorherigen Abschnitt die Kodierung von XML-Dokumenten eingeführt haben, beschäftigen wir uns nun mit den XPath-Achsen. Ausgehend von der tabellen-basierte Repräsentation der XML-Dokumente, welche wir in Tabelle 3.1 vorfinden, überlegen wir uns, wie wir die Achsen auf RDBMS abbilden können. Nachfolgend wollen wir aber nur die Achsen behandeln, welche im weiteren Verlauf für uns noch eine Bedeutung haben. Diese Achsen haben wir in Tabelle 3.3 aufgelistet. Der Knoten *ctx* stellt dabei immer den Kontextknoten dar, von welchem aus die entsprechenden Knoten gefunden werden sollen; *v* hingegen bildet die Menge der Knoten, welche auf der jeweiligen Achse vorkommen und die gewünschten Eigenschaften besitzen.

Zur Veranschaulichung betrachten wir nun eine Achse genauer. Nehmen wir die *descendant*-Achse und den Kontextknoten *ctx*. Um in die Ergebnismenge *v* aufgenommen zu werden, müssen die Knoten bestimmte Eigenschaften erfüllen. So müssen bei der *descendant*-Achse die Ergebnisknoten einen größeren *pre*-Wert besitzen als der Kontextknoten, da sie in der Dokumentordnung nach dem Kontextknoten auftreten müssen - und genau das

Achse	<i>pre/size/level</i> -Darstellung
self	$v \in \text{ctx/self} \Leftrightarrow \text{pre}(v) = \text{pre}(\text{ctx})$
descendant-or-self	$v \in \text{ctx/ancestor-or-self} \Leftrightarrow$ $\text{pre}(\text{ctx}) \leq \text{pre}(v) \leq \text{pre}(\text{ctx}) + \text{size}(\text{ctx})$
descendant	$v \in \text{ctx/ancestor} \Leftrightarrow$ $\text{pre}(\text{ctx}) < \text{pre}(v) \leq \text{pre}(\text{ctx}) + \text{size}(\text{ctx})$
child	$v \in \text{ctx/child} \Leftrightarrow$ $\text{descendant} \wedge \text{level}(v) = \text{level}(\text{ctx}) + 1$
ancestor-or-self	$v \in \text{ctx/ancestor-or-self} \Leftrightarrow$ $\text{pre}(v) \leq \text{pre}(\text{ctx}) \leq \text{pre}(v) + \text{size}(v)$
ancestor	$v \in \text{ctx/ancestor} \Leftrightarrow$ $\text{pre}(v) < \text{pre}(\text{ctx}) \leq \text{pre}(v) + \text{size}(v)$
parent	$v \in \text{ctx/parent} \Leftrightarrow$ $\text{ancestor} \wedge \text{level}(v) = \text{level}(\text{ctx}) - 1$
attribute	$v \in \text{ctx/attribute} \Leftrightarrow \text{child} \wedge \text{kind}(v) = \text{attr}$

Tabelle 3.3: Darstellung der XPath-Achsen mit *pre/size/level*

gewährleisten die *pre*-Werte. Weiterhin müssen die Ergebnisknoten im Teilbaum unterhalb des Kontextknoten liegen. Wie wir wissen, gibt *size* die Anzahl der Knoten im Teilbaum an, welcher sich unterhalb dessen befindet. Addieren wir nun *size* zum *pre*-Wert des Kontextknotens, so erhalten wir den maximalen *pre*-Wert für den letzten Knoten innerhalb des Teilbaumes. Alle Knoten welche die Eigenschaft $\text{pre}(\text{ctx}) < \text{pre}(v) \leq \text{pre}(\text{ctx}) + \text{size}(\text{ctx})$ erfüllen, kommen in der Dokumentordnung nach dem Kontextknoten und befinden sich innerhalb des Teilbaumes. Sie werden in die Ergebnismenge aufgenommen.

Die Überlegungen für die restlichen Achsen von XPath funktionieren analog und werden aus diesem Grund nicht genauer betrachtet.

Kapitel 4

Data Guides

In diesem Kapitel beschäftigen wir uns mit den *Data Guides*. Zu Beginn werden wir diese konzeptionell beschreiben, damit wir einen Überblick bekommen. Anschließend werden wir Schritt für Schritt die Definition der *Data Guides* herleiten und anhand eines Beispiels veranschaulichen.

4.1 Einführung

Wir betrachten eine beliebige Menge unterschiedlicher Elemente, wie sie in Abbildung 4.1(a) gezeigt ist. Die Elemente dieser Menge sollten gemeinsame Eigenschaften besitzen z.B. Form, Farbe, Inhalt, usw., anhand dessen wir eine Unterteilung in Partitionen vornehmen können. Zur Zeit interessiert uns noch nicht welche Attribute dies genau sind, welche die Unterteilung ermöglichen.

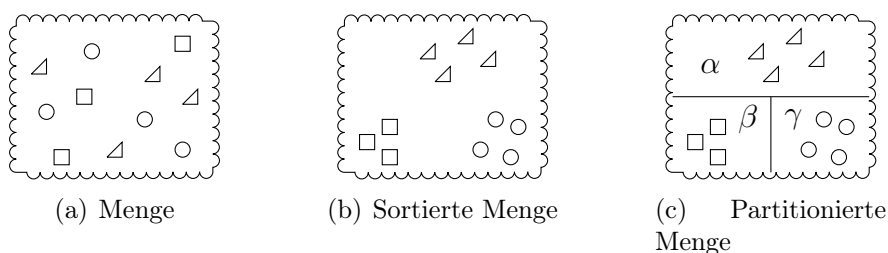


Abbildung 4.1: Schritte der *Data Guide*-Konstruktion: (b) Ordnen der Elemente und (c) Zuweisung von eindeutigen Symbolen an die unterschiedlichen Gruppierungen der Elemente

Nun ordnen wir die Elemente nach den ausgewählten Gemeinsamkeiten und bekommen verschiedene Äquivalenzklassen, die die gleichen Merkmale

aufweisen. In Abbildung 4.1(b) sind die Elemente beispielsweise nach der Form sortiert.

Jetzt können wir die einzelnen Partitionen mit einem Symbol versehen, so dass jedes stellvertretend für eine Äquivalenzklasse steht. Diese zu den Äquivalenzklassen zugewiesenen Symbole nennen wir auch *Data Guides*. Es entsteht ein Bild wie in 4.1(c). Betrachten wir nun ein Element einer Partition, so besitzt dieses die gleichen Eigenschaften wie die übrigen. Stellvertretend können wir auch den *Data Guide* als Repräsentant verwenden, indem wir ihn mit den jeweiligen Eigenschaften der Partitionen in Verbindung bringen. So können wir beispielsweise dem *Data Guide* α aus Abbildung 4.1(c) die Eigenschaft Dreieck, β das Rechteck und γ den Kreis zuordnen. Wollen wir nun alle Kreise aus der Menge bekommen, so müssen wir nur die Elemente mit dem *Data Guide* γ suchen. Zu diesem Zeitpunkt ist noch kein großer Vorteil ersichtlich, wenn wir die Partitionen mit künstlichen Surrogaten versehen. Sobald wir aber mehr als eine Eigenschaft zur Partitionierung der Menge verwenden, ändert sich dieser Sachverhalt.

Zusammenfassend ist ein *Data Guide* nichts anderes als ein Symbol, das ein Repräsentant für eine Äquivalenzklasse und deren Eigenschaften ist.

4.2 Definitionen

Nachdem wir im letzten Abschnitt die Grundlagen entwickelt haben, was *Data Guides* überhaupt sind, werden wir diese jetzt für XPath definieren.

Dazu werden wir den Begriff *Wurzelbaum* aus der Graphentheorie einführen, da, wie wir bereits gesehen haben, XML-Dokumente als Bäume dargestellt werden können. Bäume in der Graphentheorie sind nichts anderes als Graphen mit speziellen Eigenschaften. Aus diesem Grund beginnen wir mit der Definition von Graphen.

Definition 4.1. *Ein Graph $G = (V, E)$ ist ein Paar zweier endlicher Mengen V und E , wobei V die Menge der im Graph enthaltenen Knoten und E die Menge der Kanten des Graphen bezeichnet. Weiters besitzt ein Graph eine auf der Knotenmenge E definierte Abbildung Ψ mit*

$$\Psi = E \mapsto V \times V$$

*Ein Graph $G = (E, V)$ heißt **gerichtet**, falls zu jeder Kante $e \in E$ das zugeordnete Paar (v_1, v_2) mit $v_1, v_2 \in V$ gerichtet ist.*

Nun können wir den Graphen zu einem Baum verfeinern, indem wir bestimmte Kanten zwischen den Knoten verbieten. So darf in einem Baum kein Zyklus oder Kreis vorkommen.

Definition 4.2. Ein **Baum** T ist ein Graph $G = (V, E)$, in welchem es zwischen zwei Knoten $v_1, v_2 \in V$ genau einen Weg gibt.

Beispiele zu den letzten beiden Definitionen sehen wir in Abbildung 4.2. Auf der linken Seite ist ein Graph mit mehreren Kreisen abgebildet, wogegen auf der rechten Seite ein Graph ohne Kreise, also ein Baum, abgebildet ist. Die Eigenschaft, dass es zwischen zwei Knoten immer genau einen Weg gibt, ist erfüllt.

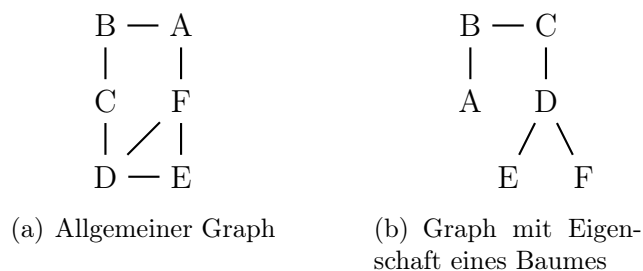


Abbildung 4.2: Unterschiedliche Graphentypen: links ein Graph mit mehreren Zyklen (Beispiel: D-E-F); rechts ein Graph mit Baumeigenschaft, allerdings ohne Wurzel

Kennzeichnen wir einen Knoten eines Baumes speziell, so erhalten wir einen Wurzelbaum (siehe Definition 4.3). Der gekennzeichnete Knoten wird auch Wurzel genannt. Aus einem Baum können mehrere verschiedene Wurzelbäume entstehen, indem wir die Wurzel variieren. In Abbildung 4.3 sehen wir zwei verschiedene Wurzelbäume, die aus demselben Baum in Abbildung 4.2(b) entstanden sind - nur mit dem Unterschied dass der Knoten, welcher als Wurzel ausgewählt wurde, ein anderer ist.

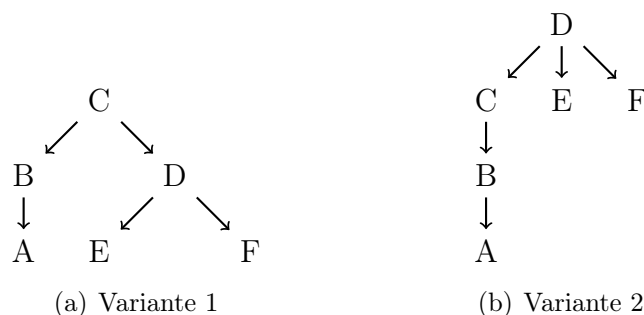


Abbildung 4.3: Varianten eines Wurzelbaumes von 4.2(b): links mit Wurzel C, rechts mit Wurzel D

Definition 4.3. Ein **Wurzelbaum** W ist ein Baum, von dem ein Knoten als Wurzel und die Endknoten als Blätter bezeichnet werden (andere Knoten werden als innere Knoten bezeichnet). Jeder Ast ist von der Wurzel weggerichtet.

Definition 4.4. Ein **Pfad** in einem Graphen $G = (V, E)$ ist eine Knotenfolge $(v_{i_0}, v_{i_1}, \dots, v_{i_n})$ mit $(v_{i_j}, v_{i_{j+1}}) \in E$ im ungerichteten Fall bzw. $(v_{i_j}, v_{i_{j+1}}) \in E$ im gerichteten Fall für $j = 0, 1, \dots, n - 1$.

Bei einem Wurzelbaum bietet es sich an, Pfade für die Einteilung der jeweiligen Äquivalenzklassen auszuwählen. Sind die Pfade zu zwei oder mehr Knoten gleich (bis auf die letzte Kante) und sind die Knoten selbst noch äquivalent bezüglich bestimmter Eigenschaften, so können wir einen der Knoten benutzen um die gesamte Partition zu beschreiben. Die Äquivalenz von Pfaden geben wir in der anschließenden Definition wieder.

Definition 4.5. Gegeben sei der Wurzelbaum $T = (V, E)$ und zwei Pfade $p_1 = (v_0, v_1, \dots, v_n)$ und $p_2 = (w_0, w_1, \dots, w_n)$. Die Pfade p_1 und p_2 sind genau dann **äquivalent**, wenn

- p_1 und p_2 dieselbe Länge haben.
- p_1 und p_2 bei der Wurzel r beginnen. Es gilt: $v_0 = w_0 = r$.
- alle Kanten existieren. Es gilt $(v_0, v_1), (v_1, v_2), \dots, (v_{n-1}, v_n) \in E$ und $(w_0, w_1), (w_1, w_2), \dots, (w_{n-1}, w_n) \in E$.
- die Knoten $v_i = w_i$ für $i = 0, 1, \dots, n$ äquivalent sind.

In Definition 4.5 haben wir die Äquivalenz von Pfaden in Wurzelbäumen definiert. Was fehlt sind die Eigenschaften der XML-Dokumente, welche die Einteilung von XPath-Pfaden in Äquivalenzklassen ermöglichen.

4.3 Partitionierung von XPath-Pfaden

Nachdem wir die Äquivalenz von Pfaden eingeführt haben, müssen wir uns noch überlegen, welche Informationen wir von den XML-Dokumenten benötigen, um die Lokalisierungspfade in Partitionen einteilen zu können. Dazu fehlen uns Eigenschaften von Knoten. Diese Attribute werden wir nachfolgend Schritt für Schritt erarbeiten. Sie sind alle aus den Eigenschaften von XML herleitbar.

Die naheliegendste Partitionierungseigenschaft eines Baumes ist das Niveau (Definition 4.6). Das Niveau eines Baumes wird in *Pathfinder* mit *level* bezeichnet. An diese Terminologie werden wir uns auch nachfolgend halten. Dies alleine genügt allerdings noch nicht, da in diesem Fall alle Knoten, welche sich auf einem Niveau befinden, zu einer Äquivalenzklasse zusammengefasst würden, und nützliche Informationen verloren gingen.

Definition 4.6. *Unter dem Niveau eines Knotens v in einem Wurzelbaum versteht man den Abstand von v zur Wurzel (die Wurzel hat das Niveau 0)*

Wie wir bereits wissen, gibt es in XPath sieben verschiedene Knotentypen (Kapitel 2.2.1). Allerdings werden in *Pathfinder* keine namespaces behandelt, somit bleiben noch sechs Typen zur Differenzierung übrig. Diese Knotenunterscheidung verwenden wir nun auch für die Erzeugung der Partitionen. Dazu definieren wir uns eine Funktion, die diese Unterscheidung der Knoten auf ihre Typen hin machen kann.

Definition 4.7. *Gegeben sei ein Wurzelbaum $T = (V, E)$, welcher aus einem XML-Dokument generiert wurde. Die Funktion $kind(v)$ weist jedem Knoten $v \in V$ einen Typ aus der Menge $\{elem, attr, text, comm, pi, doc\}$ zu. Die Bedeutung der Elemente ist wie folgt gegeben:*

- *elem: Elemente*
- *attr: Attribute*
- *text: Nutzinformation*
- *comm: Kommentar*
- *pi: Verarbeitungsanweisung*
- *doc: Wurzelement*

Auch in diesem Fall werden wir nachfolgend die Terminologie von *Pathfinder* verwenden, wo statt Knotentyp von *kind* die Rede ist. Betrachten wir nun Abbildung 4.4(c), so fällt auf, dass im rechten Blatt zwei Elementknoten mit verschiedenen Namen (C und D) zusammengefasst werden und deshalb Information verloren geht. Aus diesem Grund wollen wir den Namen als weitere Partitionierungseigenschaft hinzunehmen.

Der Name von Knoten ist eine Zeichenkette und dient der Strukturierung von XML-Dokumenten, spiegelt allerdings nicht die Nutzinformation wieder. Die Kombination aus *level* und *kind* ergibt bereits sinnvolle Ergebnisse, fasst aber noch zu viele Knoten zusammen, wie aus Abbildung 4.4(c) ersichtlich

wird. Es werden immer noch zu viele Knoten zu einer Partition verschmolzen, wie z.B. der Elementknoten C mit dem Elementknoten D. Um dies zu verhindern müssen wir eine zusätzliche Unterscheidung der Namen einführen.

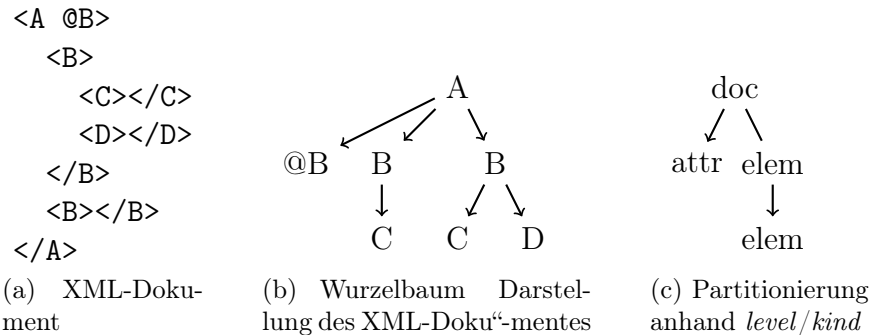


Abbildung 4.4: (a) und (b) zeigen zwei unterschiedliche Darstellungsformen eines XML-Dokumentes. (c) zeigt das partitionierte XML-Dokument durch *level/kind*

Dazu müssen wir aber zuerst festlegen, was wir unter dem Namen von Knoten verstehen. Wir unterscheiden nun für jedes *kind* eines XML-Knoten getrennt, was wir in *Pathfinder* unter dem Namen verstehen.

- **elem**: Der Name eines Elementknotens ist die im Tag angegebene Zeichenkette. Liegt beispielsweise das Tag `<A>` vor, so ist der Name `A`.
- **attr**: Attribute werden im Start-Tag eines Elementknotens definiert und werden durch `@` eingeleitet. Bsp. `<A @anzahl='10'>`. In diesem Fall verstehen wir unter dem Namen die Zeichenkette zwischen dem `@` und dem Gleichheitszeichen, also `anzahl`.
- **pi**: Verarbeitungsanweisungen haben die Form `<?pi-name...>`. Dementsprechend wird `pi-name` dem Namen der Verarbeitungsanweisungsknoten zugeordnet.
- **doc**, **text** und **comm**: Das Wurzelement, der Text und die Kommentare besitzen keinen Namen, können deshalb nicht danach partitioniert werden.

Nachdem die einzelnen Knoten in Äquivalenzklassen von *level/kind/name* eingeteilt sind, fehlt nur noch die Darstellung der Dokumenthierarchie. Um diese darzustellen, werden wir der Zuweisung von *Data Guide* Einschränkungen auferlegen. Nur Knoten unter demselben *Data Guide* dürfen in die gleiche Äquivalenzklasse aufgenommen werden. Dies hat zur Folge, dass zwei

Knoten nicht unbedingt in die selbe Äquivalenzklasse eingeteilt werden müssen, auch wenn die Werte von *level/kind/name* gleich sind. Die endgültigen Eigenschaften für die Äquivalenzklassen der Knoten von XML-Dokumenten bestehen aus den Werten von *level/kind/name* und dem *Data Guide*-Wert des Elternknotens.

Nun haben wir alle Attribute aufgelistet die eine sinnvolle Partitionierung von Pfaden in XML-Dokumenten zulassen. In *Pathfinder* verwenden wir die Werte von *level/kind/name* um einen geeigneten *Data Guide* zu erzeugen und die Pfadauswertungen zu beschleunigen. Um dies zu ermöglichen müssen wir das Schema aus Tabelle 3.1 um eine neue Spalte *guide* erweitern. Nachdem wir die Eigenschaften für die Partitionierung festgelegt haben, können wir uns nun ein Beispiel betrachten. Wir verwenden das Dokument aus Abbildung 3.2 um die Zuweisung der *Data Guides* zu veranschaulichen. Zur Kodierung des XML-Dokumentes haben wir eine weitere Spalte *guide* hinzugefügt. Alle Knoten, welche dieselben *level/kind/name*-Werte besitzen, haben wir zu einer Äquivalenzklasse zusammengefasst und mit einem *Data Guide* versehen. Dies können wir in Tabelle 4.1 beobachten.

<i>pre</i>	<i>size</i>	<i>level</i>	<i>kind</i>	<i>name</i>	<i>value</i>	<i>guide</i>
0	16	0	doc	-	'doc2.xml'	0
1	15	1	elem	a	-	1
2	0	2	attr	at	'10'	2
3	5	2	elem	b	-	3
4	4	3	elem	c	-	4
5	1	4	elem	d	-	5
6	0	5	text	-	'bar'	6
7	1	4	elem	e	-	7
8	0	5	text	-	'foo'	8
9	7	2	elem	b	-	3
10	1	3	elem	g	-	9
11	0	4	attr	wo	'le'	10
12	4	3	elem	c	-	4
13	1	4	elem	d	-	5
14	0	5	comm	-	'comm'	11
15	1	4	elem	e	-	7
16	0	5	text	-	'bla'	8

Tabelle 4.1: Kodierung des XML-Dokumentes aus Abbildung 3.2(a) und mit *Data Guides* aus den Werten *pre/size/level*

Die dazugehörige Baumdarstellung der *Data Guides* ist in Abbildung 4.5

dargestellt. Die jeweiligen *Data Guides* der Knoten sind links unten angegeben. Im Vergleich zum originalen Dokument, besitzt dieser Baum nur mehr 12 Knoten, anstatt 17.

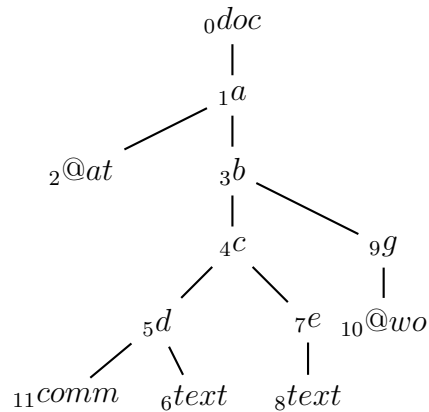


Abbildung 4.5: Die *Data Guides* des XML-Dokumentes aus Abbildung 3.2(a)

Bemerkung 4.1. Wenn im Folgenden von *Data Guide* die Rede ist, so meinen wir damit die Einteilung in die Äquivalenzklassen anhand des Tripels *level/kind/name* sowie den *Data Guide* des Elternknotens, welcher zur Partitionierung der Knoten in *Pathfinder* verwendet wird. Wird das Konzept der *Data Guides* angesprochen, so ist das aus dem Kontext ersichtlich.

Kapitel 5

Optimierung

Nachdem wir in Kapitel 4 die *Data Guides* als Konzept der Äquivalenzklassen eingeführt haben, werden wir uns nun mit den Vorteilen dieser Zusatzinformation befassen. Das Ziel ist die Verbesserung der Auswertung von XPath-Ausdrücken durch die zusätzliche Information der *Data Guides*.

In diesem Kapitel werden wir die Lücke zwischen den *Data Guides* und der Kodierung von Lokalisierungspfaden schließen. Das Schema für die Abbildung von XML-Dokumenten haben wir bereits in Kapitel 3.2 kennen gelernt. Die Abbildung der XPath-Achsen auf das RDBMS ist in Tabelle 3.3 aufgelistet.

Die Werte *pre/size/level* werden für das Darstellen der XPath-Achsen benötigt, die Werte *kind/name* für die Auswahl gezielter Knoten. *Data Guides* bestehen aus den Werten *level/kind/name*, wodurch wir mit der Kombination von *pre/size* und einem *Data Guide*-Wert die Knoten eindeutig bestimmen können.

Unser Ziel ist, so viele *Data Guides* wie möglich den Lokalisierungsschritten zuzuweisen. Wie wir noch sehen werden, bringt diese Zuweisung einen Geschwindigkeitszuwachs bei der Auswertung. Das Zuweisen von *Data Guide*-Werten an Lokalisierungsschritten nennen wir *annotieren* (von *Data Guides*). Jedoch können die *Data Guides* nicht immer annotiert werden, da es einige Ausnahmen gibt.

5.1 Einführendes Beispiel

Lokalisierungspfade bestehen aus mehreren Lokalisierungsschritten (Kapitel 2.2.2). Wir versuchen nun, nicht mehr alle Lokalisierungsschritte nacheinander auszuführen um schlussendlich die richtigen Knoten der Ergebnismenge zu erhalten, sondern den Pfad „abzukürzen“. Hierzu haben wir die *Data Guides* eingeführt. Wie diese Optimierung aussieht, betrachten wir am besten

an einem Beispiel.

Beispiel 5.1. *Betrachten wir die XMark Q15 und eine passende XML-Datei `auction.xml`. In der XQuery-Abfrage muss ein langer Pfad ausgewertet werden, bis schließlich der Text ausgegeben werden kann. Dieser ist vom Elementknoten `keyword` eingeschlossen. Die Auswertung geht folgendermaßen vor sich: Wir beginnen beim Dokumentknoten und suchen die Kindknoten mit dem Namen `site`. Danach verwenden wir diese Knoten als Kontextknoten und durchsuchen dessen Kinder auf Elementknoten mit Namen `closed_auctions`, usw. bis wir die Elementknoten `keyword` erreicht haben. Von diesen Knoten verwenden wir schließlich den Text für die Ausgabe.*

Versehen wir aber die einzelnen Knoten mit Data Guides, so können wir mit deren Hilfe und einer einzigen Abfrage alle Texte finden, die wir für die Ausgabe benötigen. Dafür ermitteln wir den entsprechenden Data Guide und benutzen ihn für die Abfrage im RDBMS, wodurch die Suche auf die Knoten mit dem entsprechenden Data Guide-Wert eingeschränkt wird.

Allerdings können nicht alle Lokalisierungsschritte beliebig für diese Art der Optimierung zusammengefasst werden. Die Einschränkungen werden wir im Nachfolgenden genauer betrachten.

5.2 Optimierung von XPath-Pfaden

In diesem Abschnitt werden wir den Algorithmus für die Optimierung von XPath-Pfaden kennen lernen. Bevor wir diesen aufzeigen, müssen die im Algorithmus verwendeten Überlegungen genauer betrachtet werden.

5.2.1 Probleme mit XPath-Achsen

In XPath werden insgesamt 13 Achsen definiert. Nicht jede eignet sich jedoch für die Optimierung mit Hilfe von *Data Guides*. Die Gründe warum das so ist, werden wir in diesem Abschnitt näher kennen lernen.

Die Achse „Namensraum“ wird von XQuery nicht unterstützt und aus diesem Grunde in *Pathfinder* nicht behandelt. Außerdem bildet die Attribut-Achse eine eigenständige Achse und kann nicht mit den anderen verknüpft werden. Attribute können allerdings mit *Data Guides* versehen werden, was zu Optimierungen führt. Bei folgenden vier Achsen funktioniert das Optimieren der Lokalisierungspfade durch Einführen von *Data Guides* nicht:

- `following`
- `following-sibling`

- preceding
- preceding-sibling

Die Ursache warum die Optimierung mit *Data Guides* nicht funktioniert, verdeutlichen wir am besten mit folgendem Beispiel.

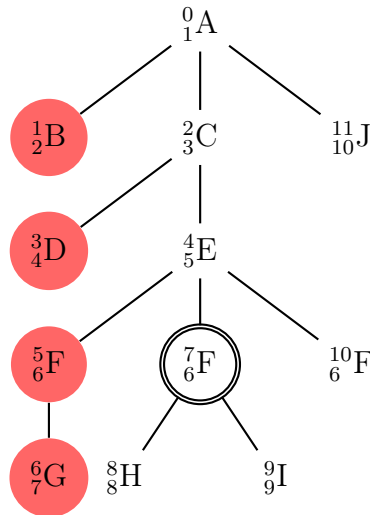


Abbildung 5.1: Die *preceding*-Achse für den Kontextknoten \odot werden durch \bullet hervorgehoben

Beispiel 5.2. Gegeben sei die Abbildung 5.1. Der Knoten mit dem Doppelkreis stellt den Kontextknoten dar. Wir wollen nun für diesen Kontextknoten alle *preceding*-Knoten herausfinden. Zur Veranschaulichung sind diese bereits farbig (\bullet) unterlegt. Alle Knoten mit dem Namen F besitzen denselben Data Guide, da sie den gleichen kind und den gleichen Namen haben, und sich auf demselben level wie der Kontextknoten befinden. Ist das der Fall, so können wir nicht mehr ohne Weiteres anhand der Data Guides unterscheiden, ob ein Knoten im Baum vor oder nach dem Kontextknoten vorkommt. Also wissen wir nicht, welche Knoten in die Ergebnismenge aufgenommen werden müssen.

Wenn wir nun diese Beobachtung aus dem Beispiel 5.2 auf das Konzept der Äquivalenzklassen übertragen, so merken wir, dass die oben genannten Achsen eine detailliertere Einteilung der Knoten vornehmen als in Kapitel 4. Dies bedeutet, dass die Unterteilung der Knoten anhand von *level/kind/name* nicht ausreicht, um die Auswahl der Elemente für die Ergebnismenge durchführen zu können. Wir benötigen eine Ordnung der Elemente

innerhalb der Äquivalenzklassen selbst. Diese Ordnung der Elemente wird durch die *pre*-Werte dargestellt, welche aber nicht Teil der Eigenschaften der eingeführten *Data Guides* sind.

Dieselbe Argumentation können wir bei den anderen drei Achsen anwenden und kommen zur gleichen Schlussfolgerung: diese sind durch den *Data Guide*, der aus den Eigenschaften *level/kind/name* besteht, nicht darstellbar. Um diese trotzdem darstellen zu können, benötigen wir zusätzlich den *pre*-Wert.

5.2.2 Achsenrichtung

Durch das Einführen von *Data Guides* ist es nun möglich, Optimierungen bezüglich der Pfadauswertung von XQuery-Abfragen vorzunehmen. Folgen zwei oder mehrere Lokalisierungsschritte aufeinander, so können diese im besten Falle zu einer vereinfachten Abfrage zusammengefasst werden (siehe Abschnitt 5.2.4). Wie wir bereits wissen, können nicht alle Achsen, die in XPath definiert sind, durch Verwendung von *Data Guides* vereinfacht werden. Wir müssen uns auf folgende sieben Achsen beschränken:

- `self`
- `child`
- `parent`
- `descendant`
- `descendant-or-self`
- `ancestor`
- `ancestor-or-self`

Folgen nun zwei solcher Lokalisierungsschritte aufeinander, können diese gegebenenfalls zu einem einzigen verschmolzen werden. Dabei müssen wir aber einige Sonderfälle beachten. Wir teilen dazu die Achsen in zwei Richtungen auf. Eine verläuft in der Baumstruktur nach unten und beinhaltet `child`, `descendant`, `descendant-or-self` und `self`. Die andere verläuft nach oben und besteht aus `parent`, `ancestor`, `ancestor-or-self` und `self`. Warum wir diese Unterscheidung machen, verdeutlichen wir an einem Beispiel:

Beispiel 5.3. *Gegeben sei ein XML-Dokument, welches in Abbildung 5.1 als Baum dargestellt ist. Nun wollen wir den Pfad `'/A/C/E/F/G/..'` auswerten. Dieser Pfad beginnt mit der Suche beim Knoten A. Anschließend*

werden die Kindknoten mit Namen **C**, von diesen die Kindknoten **E** sowie dessen Kinder mit Tag **F** gesucht. Von diesen Knoten wiederum suchen wir die Kinder mit Namen **G**. Damit ist der Abstieg im Baum zu Ende. In der bisherigen Ergebnismenge befindet sich ein Knoten mit dem pre-Wert 6. Im letzten Schritt wird der Elternknoten von **G** ermittelt. In die endgültige Ergebnismenge kommt der Knoten mit dem pre-Wert 5 und dem Data Guide 6.

Um die Pfadauswertung zu beschleunigen, weisen wir den Lokalisierungsschritten Data Guides zu. Würden wir allerdings dem gesamten XPath einen Data Guide zuweisen (entspricht dem Wert 6), so bekämen wir eine falsche Ergebnismenge. Es gibt nämlich nicht nur einen Elementknoten, der diesem Wert entspricht, sondern drei. Es existiert aber nur ein Elementknoten, der einen Kindknoten mit dem Namen **G** besitzt.

Auch hier gibt es Situationen, in denen die Äquivalenzklassen nicht mehr der Ergebnismenge entsprechen. Aus diesem Grunde genügt der *Data Guide* mit den Eigenschaften *level/kind/name* nicht für die eindeutige Identifizierung, wenn zwei Lokalisierungsschritte mit entgegengesetzten Achsenrichtungen aufeinander folgen.

Die Schlussfolgerung ist, dass wir zwei entgegengesetzte Achsenrichtungen nicht zusammenfassen und mit *Data Guides* annotieren dürfen. Es ist nicht gewährleistet, dass alle Knoten genau die gleichen Kindknoten, Attribute, usw. besitzen und dementsprechend nicht in das Ergebnis aufgenommen werden dürfen.

5.2.3 Resultierende XPath-Achse

Bevor wir den Algorithmus für die Vereinigung von Lokalisierungsschritten angeben, beschreiben wir einen wichtigen Punkt, der dabei beachtet werden muss. Gegeben seien zwei aufeinanderfolgende Lokalisierungsschritte, welche alle Bedingungen für das Zusammenfassen erfüllen. Um jedoch das Ergebnis nicht zu verfälschen, müssen wir die richtige Achse des resultierenden Lokalisierungsschrittes ermitteln. Dazu beachten wir die beiden Achsenrichtungen (Abschnitt 5.2.2) getrennt.

Als erstes werden wir die Achsen betrachten, welche die Auswertungen in der Baumstruktur nach unten vornehmen. In dieser Richtung gibt es vier verschiedene Achsen, woraus 16 verschiedene Kombinationen erzeugt werden können. Alle Kombinationen mit den daraus resultierenden Ergebnisachsen sind in Tabelle 5.1 aufgelistet. Wie die Ergebnisse entstehen, werden wir nachfolgend genauer betrachten. Da wir uns in diesem Kapitel nur mit der Achse beschäftigen, sind die Eigenschaften wie *kind* und *name* nicht von Bedeutung und werden deshalb nicht beachtet.

Achse 1	Achse 2	Ergebnisachse
child	child	descendant
child	descendant	descendant
descendant	child	descendant
descendant	descendant	descendant
child	descendant-or-self	descendant
descendant-or-self	child	descendant
descendant-or-self	descendant-or-self	descendant-or-self
descendant	descendant-or-self	descendant
descendant-or-self	descendant	descendant
self	child	child
self	descendant	descendant
self	descendant-or-self	descendant-or-self
self	self	self
child	self	child
descendant	self	descendant
descendant-or-self	self	descendant-or-self

Tabelle 5.1: Zusammenfassen von zwei XPath-Achsen

Unser Ziel ist die Ermittlung der resultierenden Achse bei der Vereinigung zweier Schritte. Beim Wert der resultierenden Achse ist es irrelevant, in welcher Reihenfolge die Auswertung der Achsen für die zu vereinigenden Lokalisierungsschritte erfolgt. Dies bedeutet, wenn wir zwei Lokalisierungsschritte durch das Konzept der *Data Guides* vereinigen, macht es für die Achse des neuen erzeugten Lokalisierungsschrittes keinen Unterschied, welche zuerst ausgewertet wurde.

Wir beginnen mit dem Achsenpaar `child` und `descendant`. Ausgehend von einem Kontextknoten `ctx` werten wir zuerst die `child`-Achse und anschließend die `descendant`-Achse aus. Es ergibt sich folgender schematischer Pfad: `ctx/child/descendant`. Für nachfolgende Überlegungen ist `ctx` ein einzelner Knoten, was jedoch keine Einschränkung darstellt. Sollte `ctx` aus einer Menge verschiedener Knoten bestehen, so müssen wir die Auswertung des ersten Schrittes auf alle weiteren Knoten ausdehnen. Das Ergebnis für den zweiten Lokalisierungsschritt wäre wiederum eine Menge.

Übertragen wir `ctx/child/descendant` in die Kodierung `pre/size/level`, dann erhalten wir für die Auswertung des `child`-Schrittes die Menge Z . Diese Menge stellt gleichzeitig die Kontextknoten für den zweiten Lokalisierungsschritt dar, und die enthaltenen Knoten erfüllen folgende Bedingung:

$$z \in Z \Leftrightarrow (pre(ctx) < pre(z) \leq pre(ctx) + size(ctx)) \wedge level(z) = level(ctx) + 1$$

R ist die gesamte Ergebnismenge die sich nach dem Auswerten beider Lokalisierungsschritte ergibt und dessen Knoten die nachfolgende Bedingung erfüllen:

$$\forall z \in Z : r \in R \Leftrightarrow pre(z) < pre(r) \leq pre(z) + size(z)$$

Um die Knoten von R zu erhalten, müssen wir für alle Knoten $z \in Z$ die **descendant**-Achse auswerten. Also ermitteln wir alle Knoten unterhalb der Kinder vom Knoten **ctx**. Anders formuliert enthält R alle Knoten vom Teilbaum von **ctx**, dessen Niveau die Bedingung $level(r) > level(\mathbf{ctx}) + 1$ erfüllt.

Werden wir die Achsen in umgekehrter Reihenfolge aus, **ctx/descendant/child**, so erhalten wir für die Menge Z alle Knoten des Teilbaumes unterhalb von **ctx**.

$$z \in Z \Leftrightarrow pre(\mathbf{ctx}) < pre(z) \leq pre(\mathbf{ctx}) + size(\mathbf{ctx})$$

Weiterhin müssen wir im zweiten Lokalisierungsschritt auf jeden dieser Knoten einen **child**-Schritt ausführen, und wir erhalten die Ergebnismenge R .

$$\forall z \in Z : r \in R \Leftrightarrow pre(z) < pre(r) \leq pre(z) + size(z) \wedge level(r) = level(z) + 1$$

R beinhaltet die Kindknoten aller Knoten des Teilbaumes, die sich unterhalb von **ctx** befinden und $level(r) > level(\mathbf{ctx}) + 1$ erfüllen. Das Ergebnis ist das gleiche wie bei der Auswertung von **ctx/child/descendant**. Zur Veranschaulichung verweisen wir auf das Beispiel 5.4.

Als Ergebnis für den resultierenden Lokalisierungsschritt erhalten wir die Achse **descendant** mit $level(r) > level(\mathbf{ctx}) + 1$. Die *level*-Eigenschaft wird bereits durch die annotierten *Data Guides* erfüllt und muss nicht erneut überprüft werden.

Beispiel 5.4. *Zur Veranschaulichung betrachten wir Abbildung 5.2. Auf der linken Seite sehen wir die Pfadauswertung **ctx/child/descendant**. Die blau umrandeten Knoten sind die Kinder des Kontextknotens **ctx** welche durch den ersten Lokalisierungsschritt ermittelt werden. Von diesen Knoten müssen wir die **descendant**-Achse auswerten und erhalten für die Ergebnismenge die rot umrandeten Knoten.*

*In Abbildung 5.2(b) werten wir den Pfad **ctx/descendant/child** aus. Als Zwischenmenge erhalten wir die blau umrandeten Knoten. Auf jeden dieser Knoten müssen wir nun den **child**-Schritt ausführen. Die Ergebnismenge ist wiederum rot unterlegt und beinhaltet genau dieselben Knoten wie in 5.2(a).*

Als nächstes betrachten wir die Ausführung zweier **child**-Schritte hintereinander, beginnend beim Kontextknoten **ctx**. Wie wir aus Tabelle 5.1

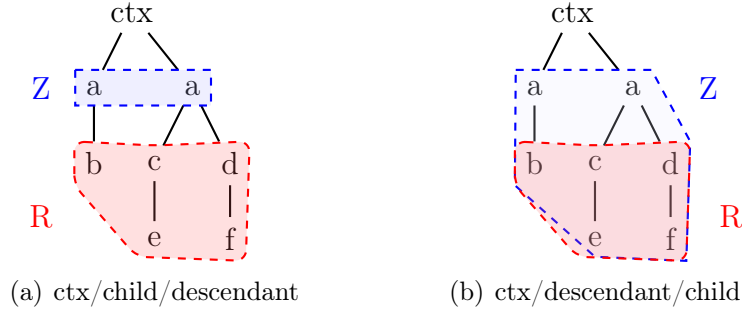


Abbildung 5.2: Auswerten der Lokalisierungsschritte **child** und **descendant** nacheinander. Z bildet die Menge der Knoten für den ersten Schritt, R ist die Ergebnismenge

sehen, besitzt die Achse des resultierenden Lokalisierungsschrittes den Wert **descendant**. Für die Begründung spalten wir die Auswertung von **ctx/child/child** wieder in zwei Abfragen auf. Die Menge Z enthält die Zwischenergebnisse mit der folgenden Eigenschaft:

$$z \in Z \Leftrightarrow (pre(ctx) < pre(z) \leq pre(ctx) + size(ctx)) \wedge level(z) = level(ctx) + 1$$

Für jeden Knoten in Z müssen wir den **child**-Schritt anwenden und erhalten die Ergebnismenge R .

$$\forall z \in Z : r \in R \Leftrightarrow (pre(z) < pre(r) \leq pre(z) + size(z)) \wedge level(r) = level(z) + 1$$

Alle Knoten in der Menge R befinden sich im Teilbaum unterhalb von **ctx** und erfüllen die Bedingung $level(r) = level(ctx) + 2$. Diese Bedingung kann mit einem **child**-Schritt nicht umgesetzt werden, da die Überprüfung des $level$ auf $level(r) = level(ctx) + 1$ festgesetzt ist. Damit wir diese Funktionalität nicht ändern müssen und Nebeneffekte erhalten, verwenden wir die **descendant**-Achse als resultierende Achse und überprüfen die $level$ -Bedingung anhand der *Data Guides*.

Beispiel 5.5. *Betrachten wir Abbildung 5.3. Wir erkennen dass zwei **child**-Lokalisierungsschritte nacheinander abgearbeitet werden. Ausgehend vom Kontextknoten **ctx** erhalten wir im ersten Schritt die blau umrandeten Knoten, auf denen wir den zweiten Lokalisierungsschritt auswerten. Als Ergebnis erhalten wir die Knoten, die rot eingerahmt sind. Wir können sehr gut erkennen, dass sich alle Knoten der Ergebnismenge auf demselben Niveau befinden und dieses die Bedingung $level(ctx) + 2$ erfüllt. Würden wir diese beiden Lokalisierungsschritte vereinigen und verwendeten **child** als Ergebnis-Achse,*

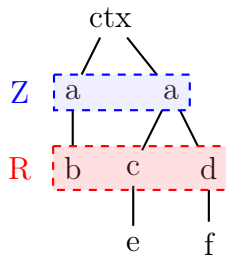


Abbildung 5.3: Auswerten zweier `child`-Schritte nacheinander. `Z` bildet die Menge der Knoten für den ersten Schritt, `R` ist die Ergebnismenge

so wäre dies ein Widerspruch zu dessen `level`-Bedingung. Die Lösung bietet die *descendant*-Achse zusammen mit der Überprüfung des Niveaus auf den korrekten Wert.

Die Herleitung der restlichen Kombinationen von Achsen erfolgt konzeptuell auf gleiche Weise und wird aus diesem Grund nicht extra behandelt. Die Auflistung der zu betrachtenden Achsen-Paare und deren Ergebnisse finden wir in Tabelle 5.2.

Achse 1	Achse 2	Ergebnisachse
child	child	descendant
descendant	child	descendant
descendant	descendant	descendant
child	descendant-or-self	descendant
descendant-or-self	descendant-or-self	descendant-or-self
descendant	descendant-or-self	descendant
self	child	child
self	descendant	descendant
self	descendant-or-self	descendant-or-self
self	self	self

Tabelle 5.2: Resultierende Achsenrichtung beim Zusammenfassen zweier im Baum nach unten gerichteter Lokalisierungsschritte; Redundante Kombinationen wurden entfernt

Bis jetzt haben wir nur die Achsen mit einer Auswertung nach unten betrachtet. Die Gegenrichtung mit den Achsen `self`, `parent`, `ancestor` und `ancestor-or-self` erfolgt analog zu den oben betrachteten Überlegungen. Aus diesem Grunde geben wir nur die Auflistung der Kombinationen und deren Ergebnis-Achsen in Tabelle 5.3 an.

Achse 1	Achse 2	Ergebnisachse
parent	parent	ancestor
ancestor	parent	ancestor
ancestor	ancestor	ancestor
parent	ancestor-or-self	ancestor
ancestor-or-self	ancestor-or-self	ancestor-or-self
ancestor	ancestor-or-self	ancestor
self	parent	parent
self	ancestor	ancestor
self	ancestor-or-self	ancestor-or-self
self	self	self

Tabelle 5.3: Resultierende Achsenrichtung beim Zusammenfassen zweier im Baum nach oben gerichteter Lokalisierungsschritte; Redundante Kombinationen wurden entfernt

5.2.4 Algorithmus

Nachdem wir jetzt wissen, welche Achsen überhaupt mit *Data Guides* optimiert werden können und welche Einschränkungen bezüglich der Achsenrichtung bestehen, werden wir nun die Vereinigung von zwei oder mehreren Lokalisierungsschritten vornehmen. Dazu benutzen wir den schematischen Pfad $L_1/L_2/L_3/\dots/L_n$ wobei L_1, L_2, \dots, L_n für einzelne Lokalisierungsschritte stehen. Um die annotierten *Data Guides* darzustellen, benutzen wir folgende Mengen-Schreibweise: $L_i\{\text{Data Guides}\}$. Nach dem Lokalisierungsschritt werden in geschweiften Klammern die einzelnen *Data Guides* durch Beistrich getrennt aufgezählt. Ein Beispiel dafür ist $L_1\{1\}/L_2\{2,15,17,24\}/L_3\{23, 45, 67\}/\dots$

Bemerkung 5.1. *Um die Unterscheidung zwischen den Lokalisierungsschritten mit und ohne Data Guides machen zu können, werden wir im weiteren Verlauf die Lokalisierungsschritte an denen Data Guides annotiert wurden, mit dem Begriff „Guide-Steps“ bezeichnen. Sollten uns die Werte der einzelnen Data Guides für die Lokalisierungsschritte nicht interessieren, so verwenden wir die Schreibweise $G_1/G_2/G_3\dots/G_n$ wobei für $i = 1 \dots n$ gilt $G_i = L_i\{\text{Data Guides}\}$.*

Nachfolgend werden wir die zwei Schritte angeben, welche notwendig sind, um das Zusammenfassen von aufeinanderfolgenden Lokalisierungsschritten zu ermöglichen und diese dann ausführlicher erklären.

1. Der erste Schritt besteht aus dem Annotieren der *Data Guides* an die einzelnen Lokalisierungsschritte.

2. Nach der Annotation können wir die Vereinfachung des Pfades vornehmen. Folgen zwei *Guide-Steps* aufeinander und besitzen sie die selbe Achsenrichtung, so können wir diese vereinigen. Der Algorithmus wird als Pseudocode in Algorithmus 1 aufgelistet.

Zum besseren Verständnis betrachten wir ein Beispiel, welches wir sukzessive erweitern, um die einzelnen Schritte anschaulicher zu erklären.

Beispiel 5.6. Wir verwenden für das Beispiel eine XQuery-Abfrage, die der Q15 von den XMark-Queries (Anhang B) ähnlich ist. Das passende Dokument sei durch `auction.xml`, und die dazugehörige Data Guide-Datei durch `auction_guide.xml` gegeben.

```

declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $a in
  $auction/site/closed_auctions/closed_auction/
  annotation/description/parlist/listitem/parlist/
  listitem/../../../../listitem/parlist/listitem/
  text/emph/keyword/text()
return $a

```

Als erstes müssen wir die Information der *Data Guides* an die Lokalisierungsschritte binden. Dies geschieht, indem wir beim ersten Schritt beginnen und die entsprechenden *Data Guides* aus den Werten *level/kind/name* und dem Elternknoten ermitteln. Ist der erste Schritt der Wurzelknoten, so entfällt die Überprüfung auf den Elternknoten. Ist die Zuweisung an L_1 erfolgreich, so können wir mit der Annotation am nächsten Schritt fortfahren. Dies führen wir solange durch, bis wir auf einen Lokalisierungsschritt stoßen, an dem wir keine *Data Guides* annotieren können. Danach können wir mit der Vereinigung der *Guide-Steps* beginnen.

Eine wichtige Eigenschaft für das Erzeugen der *Guide-Steps* ist die Achse selbst. Entspricht sie nicht einer der sieben genannten Achsen aus Abschnitt 5.2.2, dann können wir diesen Lokalisierungsschritt überspringen und auch am nachfolgendem Pfad keine *Data Guides* annotieren. Wir sind also mit der Annotation fertig und können zum zweiten Schritt übergehen.

Beispiel 5.7. *Nachfolgend sehen wir die Abfrage aus dem Beispiel 5.6, jedoch mit der Information der Data Guides angereichert.*

```

declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $a in
  $auction{1}/site{2}/closed_auctions{766}/closed_auction{767}/
  annotation{784}/description{788}/parlist{800}/listitem{802}/
  parlist{810}/listitem{812}/..{810}/..{802}/..{800}/
  listitem{802}/parlist{810}/listitem{812}/
  text{814}/emph{818}/keyword{840}/text(){841}
return $a

```

Eine weitere Pfadauswertung, wo die Data Guides nicht überall annotiert werden können, ist im folgenden Beispiel gegeben.

```

declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $a in
  $auction{1}/site{2}/regions{4}/
  preceding-sibling::africa//item
return $a

```

Wie wir sehen, können die Data Guides nur bis zum dritten Lokalisierungsschritt ermittelt werden, denn nach der Auswertung von `preceding-sibling::africa` können wir keine Annotation vornehmen.

Nachdem wir die *Data Guides* annotiert und dadurch die *Guide-Steps* erhalten haben, können wir die Optimierungen vornehmen. Die Vorgehensweise ist im Algorithmus 1 angegeben.

Bemerkung 5.2. *Im Folgenden verwenden wir den Begriff Ergebnis-Pfad. Dieser Begriff steht für einen XPath, welcher aus einem anderen Pfad hervorgeht, indem Guide-Steps zusammengefasst wurden. Es ist also ein Pfad mit annotierten Data Guides und eliminierten Lokalisierungsschritten.*

Wir betrachten immer nur zwei aufeinander folgende Lokalisierungsschritte auf einmal, die wir durch L_E und L_i ansprechen. Bevor wir mit der Schleife beginnen, initialisieren wir L_E mit dem ersten Lokalisierungsschritt. Innerhalb der Schleife weisen wir den nachfolgenden Step an L_i zu. Nun überprüfen wir ob an beiden Lokalisierungsschritten *Data Guides* annotiert sind. Ist dies nicht der Fall, so werden L_E und L_i in den Ergebnis-Pfad eingetragen und die Optimierung endet.

Im anderen Fall müssen wir überprüfen ob beide *Guide-Steps* dieselbe Achsenrichtung besitzen. Trifft dies zu, so müssen wir die Achse von L_i anpassen (Abschnitt 5.2.3). Anschließend können wir L_E aus dem Ergebnis-Pfad entfernen und L_i an L_E zuweisen, um die Optimierung der nächsten Lokalisierungsschritte zu ermöglichen. Haben die zu überprüfenden Lokalisierungsschritte zwei unterschiedliche Achsenrichtungen, so müssen wir L_E in

Algorithm 1 Zusammenfassen von *Guide-Steps*

Require: XPath-Pfad mit annotierten *Data Guides***Ensure:** Ergebnis-Pfad mit den vereinigten *Guide-Steps*

```
1:  $L_E = L_1$ 
2: for  $i = 2$  to  $n$  do
3:   if  $L_E$  und  $L_i$  sind Guide-Steps then
4:     if  $L_E$  und  $L_i$  haben gleiche Achsenrichtung (Kapitel 5.2.2) then
5:       Resultierende Achse ermitteln (Kapitel 5.2.3)
6:        $L_E = L_i$ 
7:     else
8:       trage  $L_E$  in Ergebnis-Pfad ein
9:        $L_E = L_i$ 
10:    end if
11:  else
12:    trage  $L_E$  in Ergebnis-Pfad ein
13:    trage  $L_i$  in Ergebnis-Pfad ein
14:  beende Schleife
15: end if
16: end for
```

den Ergebnis-Pfad eintragen, können aber mit der Optimierung fortfahren. Dies bedeutet, dass für eine korrekte Auswertung die Ermittlung der Ergebnisknoten an L_E nötig ist, der restliche Pfad allerdings weiter optimiert werden kann.

Wenden wir den vorhin beschriebenen Optimierungs-Algorithmus auf die Abfragen aus 5.7 an, so erhalten wir die vereinfachte Abfrage wie in Beispiel 5.8 dargestellt. Der Lokalisierungspfad besteht nun nur noch aus drei Schritten und nicht wie zuvor aus zwanzig.

Beispiel 5.8. *Nachfolgend ist dieselbe Abfrage wie in 5.7 dargestellt, nur dass der Algorithmus zur Optimierung der XPath-Pfade angewandt wurde. Der Pfad selbst besteht nur noch aus drei Schritten.*

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $a in
  $//listitem{812}/ancestor::*{800}//text(){841}
return $a
```

5.3 Zusatzinformationen von XML-Dokumenten

Die *Data Guides* können nicht nur zum Beschleunigen der XPath-Pfade benutzt werden, sondern auch für weitere Konzepte. Dazu benötigen wir zusätzliche Informationen über den Aufbau des XML-Dokumentes. In diesem Abschnitt werden wir als erstes beschreiben, um welche Informationen es sich handelt. Anschließend werden wir die Optimierungen näher erläutern.

5.3.1 *Data Guide*-Attribute

Um die Optimierungen erklären zu können, müssen wir zuerst die Begriffe min/max und *count* einführen und definieren. Diese Informationen werden zur gleichen Zeit wie die *Data Guides* aus dem XML-Dokument ermittelt, also bei der Transformierung des XML-Dokumentes in die tabellen-basierte Repräsentation.

Zur Veranschaulichung der Eigenschaften betrachten wir Abbildung 5.4. Dort sehen wir ein XML-Dokument in der Baumdarstellung, wobei das Dokument selbst nicht dargestellt ist, da es hier nicht von Bedeutung ist. In Abbildung 5.4(b) ist der *Data Guide*-Baum zum Dokument 5.4(a) dargestellt und in Tabelle 5.4 finden wir die Darstellung der *Data Guides* in Tabellenform, zusätzlich mit den min/max-Werten. Die Bedeutung dieser Werte werden wir nachfolgend diskutieren.

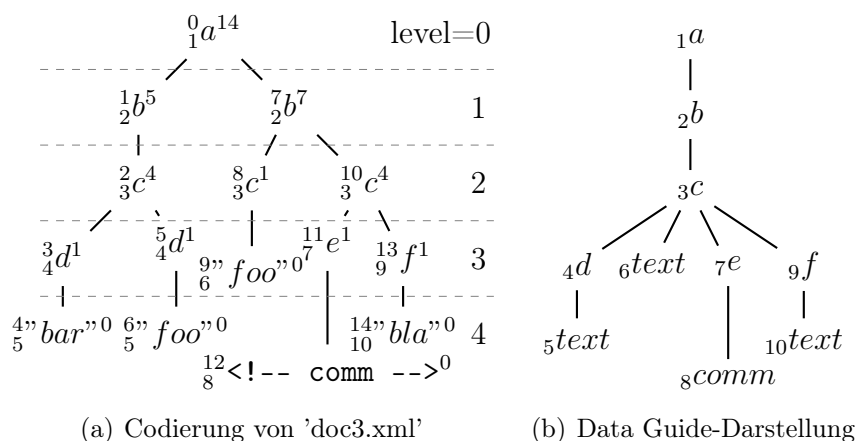


Abbildung 5.4: Baumstruktur eines XML-Dokumentes.

Der min-Wert gibt das minimale Vorkommen der Knoten mit demselben *Data Guide* unterhalb eines Elternknotens an, der max-Wert das maximale. Betrachten wir nun beispielsweise den Knoten C mit dem *Data Guide* 3

aus 5.4(a): wir sehen, dass unter dem linken Knoten B nur ein Knoten vorkommt, beim rechten dagegen zwei. Aus diesen Beobachtungen ermitteln wir die Werte min/max für den *Data Guide* 3, und setzen den min-Wert auf 1 und den max-Wert auf 2. Wir geben diese Werte für alle *Data Guides* an und erhalten damit die Kodierung in Tabelle 5.4.

Gleichzeitig wird noch der *count*-Wert ermittelt. Dieser Wert spiegelt die Anzahl der Elemente der einzelnen Äquivalenzklassen wieder, d.h. für die Kombination von *level/kind/name* werden die im XML-Dokument auftretenden Knoten gezählt und dem *Data Guide* als Zusatzinformation zugewiesen. Dabei macht es keinen Unterschied, unter welchem Elternknoten die Elemente der Äquivalenzklasse vorkommen, im Gegensatz zu den min/max-Werten. Als Beispiel wollen wir den Knoten C aus Abbildung 5.4(a) heranziehen. Dieser Knoten kommt dreimal auf *level* zwei vor. Da die Unterscheidung zwischen den einzelnen Elternknoten nicht von Bedeutung ist, erhalten wir als Resultat den Wert drei. Die gesamten Informationen, welche während des Parsens vom XML-Dokument aus Abbildung 5.4(a) ermittelt werden, sind in Tabelle 5.4 aufgelistet.

<i>guide</i>	<i>count</i>	min	max	<i>kind</i>	<i>name</i>
1	1	1	1	elem	a
2	2	1	1	elem	b
3	3	1	2	elem	c
4	2	0	2	elem	d
5	2	1	1	text	-
6	1	0	1	text	-
7	1	0	1	elem	e
8	1	0	1	com	-
9	1	0	1	elem	f
10	1	0	1	text	-

Tabelle 5.4: *Data Guide*-Kodierung mit min/max und *count*-Werten

5.3.2 Optimierungen durch min/max

Durch die Zusatzinformation von min/max können wir Optimierungen an den XQuery-Abfragen zur Kompilzeit vornehmen, welche wir nachfolgend beschreiben.

string-joins

Die XQuery Funktion `string-join` (MMW07) erzeugt aus einer Liste von Zeichenketten ein einziges Element, wobei die einzelnen Elemente durch einen String getrennt sind.

```
fn:string-join($arg1 as xs:string*, $arg2 as xs:string) as xs:string
```

Der Parameter `$arg1` ist die Sequenz von Strings, die miteinander verknüpft werden sollen. `$arg2` stellt den Trennstring dar, durch welchen die einzelnen Elemente voneinander getrennt werden sollen. Befinden sich keine Elemente in der Liste, so wird der leere String als Ergebnis geliefert. Ist der Trennstring `$arg` leer, so werden die Elemente aus der Liste ohne Trennung aneinander gereiht und als Ergebnis zurückgeliefert. Befindet sich hingegen nur ein Element in der Liste, dann besteht das Ergebnis aus dem Element ohne das Trennzeichen.

Wollen wir nun diese Funktion an einem Knoten des XML-Dokumentes ausführen, der mit einem *Data Guide* versehen ist, so können wir bereits während des Kompilierens der XQuery-Anfrage einige Optimierungen vornehmen. Ist der zum *Data Guide* ermittelte max-Wert gleich eins, so wissen wir, dass der Knoten höchstens einen String im entsprechenden Teilbaum besitzt. In diesem Fall müssen wir die Funktion `string-join` nicht mehr anwenden, da das Ergebnis gleich dem Element selbst ist. In diesem Fall können wir die Funktion weglassen.

merge-adjacent

Die Datenmodelle von XQuery und XPath sind reicher an Konstrukten und weniger zwingend als das von XML. So beinhaltet das Datenmodell der ersteren eine Typunterscheidung, Listen, usw., was es in XML nicht gibt. Wenn nun ein solches Datenmodell umgewandelt werden soll, müssen verschiedene Konvertierungen in den Datentyp String vorgenommen werden. Der genaue Vorgang der Serialisierung ist nicht vom W3C (KW03) vorgegeben, sondern nur die Idee und das Resultat. Bei dieser Serialisierung entstehen mehrere String Werte, welche zusammengefasst werden können (merge-adjacent). Dieses „mergen“ von benachbarten Strings kann auf dieselbe Art und Weise wie das Joinen von Strings vom vorhergehenden Abschnitt realisiert werden. Aus diesem Grunde wird nicht näher darauf eingegangen.

exactly-one

Mit der XQuery-Funktion `exactly-one` (MMW07) wird die Kardinalität einer Sequenz überprüft. Ist die Anzahl der Elemente der Liste nicht genau

eins, so wird ein Fehler ausgelöst.

```
fn:exactly-one($arg as item(*) as item())
```

Die Funktion `exactly-one` bekommt als Parameter eine Liste von Knoten. Sie überprüft, ob genau ein Element in dieser Liste vorhanden ist. Trifft dies zu, so wird `$arg` zurückgeliefert, andernfalls der Fehler `err:FORG0005` ausgelöst. Dieser beinhaltet die Information, dass entweder null oder mehr Elemente in der Liste vorhanden sind.

Werten wir nun die Funktion `exactly-one` auf Knoten aus, an welchen wir *Data Guides* annotiert haben, so können wir bereits zur Kompilzeit einige Optimierungen vornehmen. Die Bedingung der XPath-Funktion können wir jedoch auch mit den Werten `min/max` ausdrücken. Wenn beide den Wert eins haben, so befindet sich unter jedem Teilbaum des Elternknotens genau ein Knoten, der die gesuchten Eigenschaften besitzt. Durch diese Zusatzinformation können wir die Funktion `exactly-one` aus dem Auswertungsplan streichen, da die Überprüfung dann keinen Sinn mehr macht.

5.3.3 Optimierung durch *count*

`count` ist eine XQuery-Funktion, welche die Anzahl der Elemente einer Liste ermittelt.

```
fn:count($arg as item(*) as xs:integer
```

Der Parameter `$arg` ist eine Liste von Elementen, wovon die Anzahl ermittelt werden soll. Der Rückgabewert einer leeren Liste ist Null. An die Liste `$arg` können auch Attribute angehängt werden, welche eine Einschränkung der Elemente ermöglichen. In diesem Fall wird jedes Element auf die Erfüllung der Eigenschaft getestet und bei positivem Ergebnis wird der Zähler um eins erhöht.

Folgt ein `count` ohne Attributeinschränkung auf einen *Guide-Step*, können wir bereits zur Kompilzeit den Wert der `count`-Funktion ermitteln. Diese Kombination von Operatoren zählt die Knoten, welche durch den *Guide-Step* ausgewählt wurden. Die Anzahl steht bereits bei der Kompilierung der XQuery-Abfrage fest und kann mit Hilfe der Zusatzinformation `count` ermittelt werden. Dafür müssen wir von jedem *Data Guide*, der im *Guide-Step* enthalten ist, den `count` addieren und erhalten das abschließende Ergebnis.

Kapitel 6

Experimente

In diesem Kapitel beschreiben wir am Anfang die Komponenten, welche verwendet wurden, um die Tests auszuführen. Als erstes wird der verwendete Benchmark näher betrachtet. Anschließend gehen wir auf das System ein, unter welchem die Tests ausgeführt wurden, sowie das zugrundeliegende RDBMS DB2[®] mit dessen Werkzeugen. Um die Reproduzierbarkeit der Tests zu gewährleisten, werden dessen Vorbereitungen genau aufgeschlüsselt. Die erhaltenen Ausführungszeiten der einzelnen Testdurchläufe werden im letzten Abschnitt genauer miteinander verglichen.

6.1 XMark-Queries

Für den Vergleich von unterschiedlichen XML-DBMS wurde am *National Research Institute for Mathematics and Computer Science* (CWI) in den Niederlanden ein aus 20 XQuery-Abfragen bestehender Benchmark (SWK⁺⁰²) ausgearbeitet. Diese XQuery-Abfragen werden auch **XMark-Queries** genannt und sind im Anhang B aufgelistet. Für die Unterscheidung der einzelnen Abfragen werden diese mit Q01 bis Q20 durchnummeriert.

Damit eine standardisierte Auswertung gemacht werden kann, wurde das Schema des XML-Dokumentes festgelegt (Abbildung 6.1). Es stellt die Datenspeicherung eines Internet-Auktionshauses dar. Für das Erzeugen der Dokumente wurde ein Generator `xmlgen` programmiert, welcher Dateien verschiedener Größe zwischen wenigen Megabyte und mehreren Gigabyte generieren kann.

Um ein breites Spektrum der XQuery-Sprache abzudecken, wurden zu verschiedenen Konzepten **XMark-Queries** erstellt. Die Unterteilung geschieht anhand von 14 Klassen:

1. Genaue Übereinstimmung (Exact Match): Es wird ein bestimmter Kno-

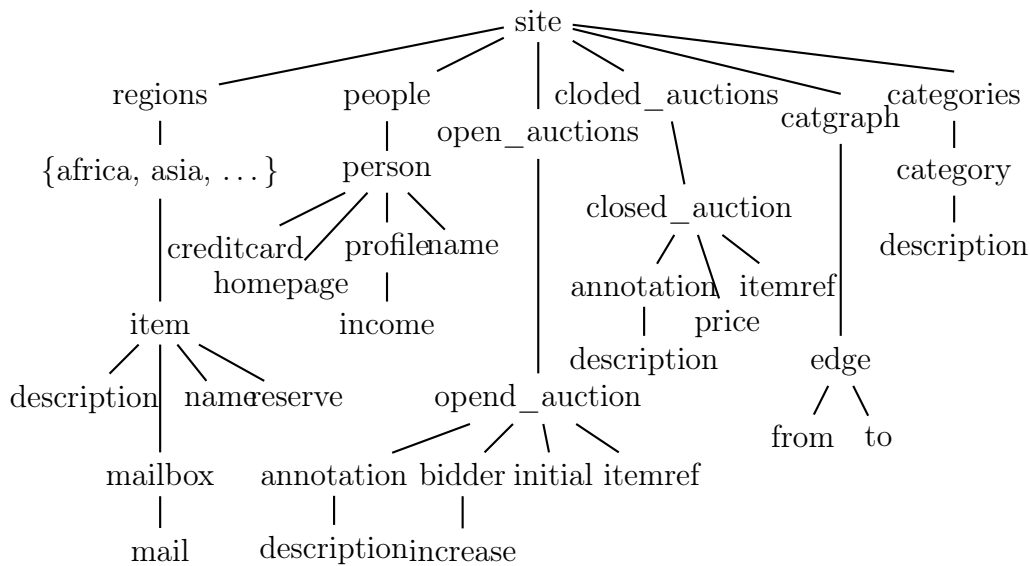


Abbildung 6.1: DTD der von `xmlgen` erzeugten XML-Dateien

ten im Dokument gesucht. Diese Abfrage dient zur Erstellung einer Performance Grundlage (Q01).

2. Dokumentordnung (Ordered Access): Die Dokumentordnung ist eine fundamentale Eigenschaft von semistrukturierten Datenmengen. Um die Performance der Realisierung zu testen werden drei Abfragen verwendet (Q02, Q03, Q04).
3. Typumwandlung (Casting): In XML ist String der generische Typ. Es wird getestet, wie gut die Typumwandlung in andere Datentypen funktioniert (Q05).
4. Reguläre Pfadausdrücke (Regular Path Expressions): Es wird getestet wie gut die Optimierung der Pfade funktioniert und ob dadurch nicht relevante Pfade übergangen werden (Q06, Q07).
5. Quer-Verweise (Chasing References): Quer-Verweise sind in XML erlaubt. Ihre Bearbeitung durch das System wird durch zwei Abfragen getestet (Q08, Q09).
6. Komplexe Ergebnisse (Construction of Complex Results): Die Erzeugung von komplexen Ergebnissen kann das DBMS an seine Grenzen bringen, besonders wenn das Ergebnis materialisiert werden soll (Q10).

7. Joins anhand von Werten (Joins on Values): Der Umgang mit großen Zwischenergebnissen beansprucht auch das DBMS stark und zeigt, wie gut der Optimierer funktioniert (Q11, Q12).
8. Wiederherstellung (Reconstruction): Die Wiederherstellung des aufgebrochenen XML-Dokumentes wird getestet (Q13).
9. Volltextsuche (Full Text): Das Durchsuchen von String nach bestimmten Zeichenketten wird getestet (Q14).
10. Pfad Traversierung (Path Traversal): Die Auswertung langer Pfade ohne Wildcards wird getestet (Q15, Q16).
11. Fehlende Knoten (Missing Elements): Der Umgang mit den semistrukturierten Daten von XML, im Speziellen mit optionalen Elementen, wird getestet (Q17).
12. Benutzerdefinierte Funktionen (Function Application): Es wird die Handhabung von benutzerdefinierten Funktionen getestet (Q18).
13. Sortieren (Sorting): Die Geschwindigkeit bei der Sortierung der Daten ist hier das Haupt-Augenmerkmal (Q19).
14. Aggregatbildung (Aggregation): Die Aggregation von semistrukturierten Daten wird getestet (Q20).

6.2 Komponenten der Testumgebung

In diesem Abschnitt werden wir kurz die Testumgebung skizzieren: den verwendeten Rechner, das Datenbanksystem und seine Werkzeuge.

6.2.1 Computer

Der Rechner auf dem wir die Tests durchführen, besitzt vier Intel[®] Xeon[™] Prozessoren mit 3.20 GHz. Bei zwei Prozessoren wurde Hyper-Threading aktiviert. Der verfügbare Hauptspeicher beträgt 8 GB mit einem zusätzlichen 8 GB Swap-Bereich. Der Massenspeicher besteht aus 6 Festplatten zu je 146 GB von denen jeweils zwei zu einem RAID 1 zusammengeschlossen sind.

6.2.2 Datenbanksystem

Das zugrunde liegende Datenbankmanagementsystem für die Testausführungen ist DB2[®] (Version 9.1) von IBM. DB2[®] stellt eine Reihe von Werkzeugen zur Verfügung, mit denen die Testvorbereitungen und -ausführungen erleichtert werden. Diese sollen nun näher beschrieben werden.

db2adv

db2adv (DB2[®] Design Advisor Command) (ibm07a) ist ein Hilfsmittel für DB2[®] Benutzer, um damit materialisierte Tabellen (MQT) und Indizes automatisch ermitteln zu lassen. Durch das Anwenden von Indizes auf Datenbanktabellen wird die Abarbeitung der SQL-Befehle beschleunigt. Für die genauere Erklärung der Begriffe MQT und Index und deren Vorteile sei auf die weiterführende Literatur verwiesen (KE04).

Dem Befehl *db2adv* werden als Parameter ein oder mehrere SQL-Befehle übergeben, für welche die Indizes ermittelt werden sollen. Die Sammlung von SQL-Abfragen wird auch als *workload* bezeichnet. Als Ergebnis liefert *db2adv* ein Skript, in dem `CREATE INDEX`, `CREATE SUMMARY TABLE (MQT)` und `CREATE TABLE` Befehle enthalten sind. Somit wird das Erzeugen der entsprechenden Indizes und Tabellen vereinfacht.

db2batch

db2batch (Benchmark Tool Command) (ibm07a) ist ein Befehl zum Ausführen von Benchmarks, mit welchem wir aussagekräftige Zeiten ermitteln können. Als Eingabe werden eine oder mehrere SQL-Anweisungen erwartet, die nacheinander ausgeführt werden. Das Resultat besteht aus dem SQL-Code, gefolgt von dessen Ergebnis und verschiedenen Zeitmessungen. *db2batch* ermittelt die Gesamtdauer der Ausführung aller übergebenen SQL-Anweisungen, die minimale und maximale Zeit, sowie das arithmetische und geometrische Mittel der Zeiten aller SQL-Anfragen. Damit ist es möglich, einen relativ genauen Benchmark zu erzeugen, dessen Zeiten aussagekräftig sind.

db2expln

Das Werkzeug *db2expln* (SQL Explain Command) (ibm07a) ermöglicht die visuelle Darstellung von Zugriffsplänen, welche für die Abarbeitung der SQL-Anweisung verwendet wird. Für das Abrufen eines Zugriffsplans müssen keine EXPLAIN-Daten erzeugt werden. Sie umfassen aus diesem Grunde nur einen vereinfachten Plan, der vom SQL-Compiler verwendet wird. Mit diesen Informationen können wir nachvollziehen, wie die einzelnen SQL-Anweisungen

abgearbeitet werden. Gegebenenfalls können Leistungsverbesserungen vorgenommen werden. Genügen die erhaltenen Daten nicht, so kann mit *db2exfmt* ein detaillierter Zugriffsplan erzeugt werden.

db2exfmt

Die Darstellung und visuelle Aufbereitung der EXPLAIN-Daten wird mit dem Befehl *db2exfmt* (Explain Table Format Command) (ibm07a) ermöglicht. Bevor dies geschehen kann, müssen diese Daten allerdings gesammelt werden, indem beim Ausführen einer SQL-Anweisung der `CURRENT EXPLAIN MODE` auf `EXPLAIN` gesetzt und die Anweisung anschließend aufgerufen wird. Zu den gesammelten Informationen gehören, unter anderem, folgende:

- Reihenfolge der Operationen für die Verarbeitung einer SQL-Abfrage
- Vergleichselemente und dazugehörige Selektivitätsschätzwerte
- Statistiken für Objekte, auf welche in der SQL-Anweisung verwiesen wird

Als Ergebnis liefert *db2exfmt* einen detaillierten Zugriffsplan mit den vorgenommenen Schätzungen von Zugriffszeiten auf die Festplatte, den Prozessorkosten und den Filterfaktoren für Prädikate. Anhand dieser Informationen kann der Zugriffsplan genau verfolgt und gegebenenfalls mit entsprechenden Mitteln beeinflusst werden.

Statistiken

Um eine SQL-Abfrage auf DB2[®] ausführen zu können, muss diese zuerst vom Datenbank-Optimierer in einen Zugriffsplan übersetzt werden. Um eine schnelle Ausführung der Abfrage zu gewährleisten, benötigt der Datenbank-Optimierer Informationen über die Verteilung der in den Tabellen enthaltenen Daten. In DB2[®] wird ein kostenbasierter Optimierer für das Erzeugen der Zugriffspläne benutzt. Die Statistiken hierfür können mit dem Befehl `RUNSTAT` erzeugt werden. Es wird zwischen zwei Arten von Statistiken unterschieden: grundlegende Statistiken (*basic statistics*) und Werteverteilung (*distribution statistics*) (Fec06).

Bei den *basic statistics* werden Informationen über die Tabelle und dessen Spalten gesammelt. Gesammelte Informationen über die Tabelle sind beispielsweise die Anzahl der benutzten Seiten, die Anzahl der Seiten die Tupel enthalten usw. Bei den Spalten wird die Kardinalität, die mittlere Länge der Spalte, der zweitgrößte und zweitkleinste Wert innerhalb der Spalte usw. ermittelt.

distribution statistics werden wiederum in zwei verschiedene Teile eingeteilt. Einerseits gibt es Häufigkeitsstatistiken und andererseits die Quantile. Bei den ersteren werden Informationen über die Häufigkeit der auftretenden Werte innerhalb einer Spalte erzeugt, also wie oft die einzelnen Werte in einer Spalte auftreten. Der Datenbank-Optimierer benutzt diese, um den Filterfaktor von Prädikaten bei den Tests auf (Un-)Gleichheit von Werten korrekt zu berechnen. Bei Quantile (SS02) geht es um die Bestimmung, wie viele Tupel unterhalb oder oberhalb eines bestimmten Wertes auftreten. Solche Informationen ermöglichen eine bessere Bestimmung der Filterfaktoren von Prädikaten, bei denen Abfragen die Form kleiner(-gleich), größer(-gleich) haben oder beim **BETWEEN**-Befehl.

Statistical View

Auch unter Verwendung der oben genannten Statistiken gibt es Situationen, in denen sich der Optimierer irren kann. Bei dynamischen Abfragen, die zur Kompilzeit keine Auswahlkriterien bereit stellen können, benötigen wir ein Hilfsmittel, um Prognosen für einen Join erstellen zu können. Ein *Statistical View* kann in diesem Fall eine bessere Planung ermöglichen, indem Kardinalitäten von bestimmten Konstrukten als statistische Daten zur Verfügung gestellt werden. Damit ein *Statistical View* für statistische Zwecke bei der Planung verwendet werden kann, genügt es ein **VIEW** anzulegen, welches den entsprechenden Join beinhaltet. Dieses wird mit der Option **ENABLE QUERY OPTIMIZATION** versehen. Dadurch wird es dem Datenbank-Optimierer ermöglicht, die gesammelten Informationen für die Planung der Zugriffspläne zu benutzen. Für ausführlichere Informationen verweisen wir auf (Che06).

6.3 Test-Vorbereitung

Nach der Darstellung des Systems, auf dem die Tests ausgeführt werden, skizzieren wir nun kurz das Vorgehen beim Testen. Die zugrunde liegende Tabelle im RDBMS (nachfolgend **XMLDOC** genannt) besitzt ein ähnliches Schema wie es in Tabelle 3.1 aufgelistet ist. Um jedoch die Vorteile der *Data Guides* nutzen zu können, müssen wir dafür noch eine Spalte in das Schema aufnehmen. Zudem führen wir eine berechnete Spalte **pre+size** ein, damit die Addition nicht bei jedem Lokalisierungsschritt von neuem durchgeführt werden muss. Das endgültige Schema für **XMLDOC** ist in Tabelle 6.1 aufgelistet.

Wir wenden uns nun dem Erzeugen der XML-Dokumente zu: Hierzu benutzen wir das Programm **xmlgen** und generieren drei XML-Dateien mit den

Name	Beschreibung
<i>pre</i>	<i>pre</i> -Wert des Knotens
<i>size</i>	Anzahl der Knoten im Teilgraphen unter dem jeweiligem Element
<i>level</i>	Niveau des Knotens im Baum
<i>kind</i>	Typ des Knotens
<i>value</i>	Text
<i>name</i>	Tagname der Knoten
<i>guide</i>	Werte der <i>Data Guides</i>
<i>pre+size</i>	Berechnete Spalte für die Werte $pre(v) + size(v)$

Tabelle 6.1: Schema für die RDBMS-Tabelle XMLDOC

Faktoren 0.01, 0.1 und 1. Dies entspricht den Dateigrößen 1.2 MB, 12 MB und 112 MB. Diese Dateien müssen wir noch mit dem *Shredder* (siehe Anhang A) in die tabellen-basierte Repräsentation transformieren, damit sie in die Datenbank geladen werden können. Bei jeder der drei verschiedenen Dateien führen wir den XMark-Benchmark aus.

Der nächste Schritt ist das Erzeugen von Indizes mit Hilfe von *db2advis*. Als Eingabe für *db2advis* dienen uns alle XQuery-Anfragen des XMark-Benchmarks, die von *Pathfinder* nach SQL kompiliert wurden. Dabei gibt es zwei Versionen pro XMark-Query: einmal mit dem Konzept der *Data Guides* und ein anderes mal ohne diese. Das Werkzeug *db2advis* liefert vorgefertigte Data Definition Language (DDL) Konstrukte, die das Einfügen der Indizes in das RDBMS vereinfachen.

Nach diesen Vorbereitungen können wir den Benchmark starten. Der Benchmark besteht aus den mit *Pathfinder* kompilierten XMark-Queries. Um die Abweichungen für die Queries mit dem Konzept der *Data Guides* von denjenigen ohne *Data Guides* zu ermitteln, werden beide Varianten auf demselben RDBMS unter gleichen Bedingungen ausgeführt. Dieses Vorgehen gewährleistet den Erhalt vergleichbarer Zeiten.

Die nachfolgenden Schritte beschreiben, wie die Laufzeiten der XMark-Queries ermittelt werden und müssen auf jede der drei erzeugten XML-Dokumente angewandt werden. Die Messung der Laufzeiten besteht aus zwei Teilen und wird für jede XMark-Query ausgeführt. Bevor wir die Laufzeiten der SQL-Anweisungen ermitteln, führen wir die SQL-Anweisung einmal auf dem RDBMS aus. Dies bewirkt, dass die Abfrage von DB2[®] kompiliert und der dazugehörige Zugriffsplan abgespeichert wird. Führen wir jetzt dieselbe SQL-Anweisung erneut aus, so entfällt die Kompilezeit und wir erhalten nur die Laufzeit der Query. Im zweiten Schritt benutzen wir *db2batch*, um dieselbe XMark-Query mehrmals nacheinander auszuführen. Wir haben die Anzahl

der Wiederholungen aufgrund der hohen Laufzeiten einiger XMark-Queries auf drei festgelegt. Es traten dabei auch nie größere Schwankungen auf, die eine höhere Anzahl gerechtfertigt hätten. Die Zeiten, die wir für die Auswertung benutzen, sind jeweils das arithmetische Mittel dieser drei erhaltenen Werte.

Um die weiteren Vor- und Nachteile der *Statistical View* aufzuzeigen, führen wir erneut Testläufe durch. Dazu erzeugen wir ein *Statistical View* das im Listing 6.1 dargestellt ist. Nachdem das *Statistical View* erzeugt und die Statistiken erneuert wurden, führen wir die selben Zeitmessungen wie im letzten Absatz durch.

Listing 6.1: Statistical View

```
create view xmlview as
  select a1.*
from xmldoc as a1, xmldoc as a2
where(a1.kind = 6) and
  (a2.pre between (a1.pre + 1) and (a1.pre + a1.size))
```

Die ermittelten Zeiten der einzelnen XMark-Queries in den verschiedenen Szenarien sind die Grundlage für das Kapitel 6.5 und 6.6. Alle Zeiten der einzelnen Testszenerien sind im Anhang C aufgelistet.

6.4 Indizes

Das Konzept der Indizes spielt im Zusammenhang mit den RDBMS eine wichtige Rolle. Durch Verwendung der richtigen Indizes lässt sich eine schnellere Ausführungszeit von SQL-Befehlen erreichen. DB2[®] stellt ein Werkzeug (*db2advsi*) zur Verfügung, mit welchem die automatische Ermittlung von Indizes für SQL-Abfragen möglich ist. Mit *db2advsi* lassen wir Indizes für den XMark-Benchmark (siehe Kapitel 6.1) kreieren. Die wichtigsten Indizes zeigt Tabelle 6.2. Diese Indizes sind speziell für SQL-Befehle mit der *Data Guide*-Optimierung von Bedeutung.

Verwendete Spalten
<guide,pre>
<guide,pre,pre+size>
<value,guide,pre>

Tabelle 6.2: Auszug aus den von *db2advsi* ermittelten Indizes

Wie groß der Einfluss der Indizes auf die Auswertung der Pfade von XPath ist, wird in (GRT07) aufgezeigt. Insbesondere ist der Vergleich zwischen den

Indizes `<pre,level>` und `<level,pre>` interessant. Die beschriebenen Indizes basieren auf dem Konzept der *Partitioned B-Trees* (Gra). Die Auswertung von Lokalisierungspfaden mit den Achsen `child` und `parent` wird entscheidend durch den Index beeinflusst. Als Beispiel dient der Pfad `site/regions/africa`. Während sich die Laufzeit für den Index `<pre,level>` bei größeren XML-Dokumenten verlangsamt, bleibt sie für die Auswertung mit `<level,pre>` konstant (GRT07).

Die in Tabelle 6.2 gezeigten Indizes beinhalten alle die Spalte *guide*. Wie in Kapitel 4.3 beschrieben, ist der Wert von *guide* gleichbedeutend, wie die Information von *level*, *kind* und *name* zusammen. Aus diesem Grund besitzt ein Index mit der Spalte *guide* mindestens dieselbe Selektivität wie *level*. Der zusätzliche Vorteil eines *Data Guides* liegt darin, dass wir bereits zur Kompilzeit der XQuery-Abfrage diese Zuweisung machen können. Für die Auswertung eines XPath-Pfades mit annotierten *Data Guides* genügt ein Index mit dem Wert *guide*, anstelle von *level*, *kind* und *name*. Dies bedeutet, der Vergleich reduziert sich auf den richtigen *guide*-Wert. Die Spalten der Indizes können wir dann ebenfalls verringern.

Besonders für die Darstellung der Achsen ist der Index `<guide,pre,pre+size>` wichtig. Er spiegelt genau die benötigten Informationen wieder, um einen mit *Data Guide* versehenen Lokalisierungspfad auszuwerten. Betrachten wir die Zugriffspläne, welche DB2[®] für die Abarbeitung der XMark-Queries verwendet, so bestätigt sich dieser Sachverhalt.

6.5 Auswertung

In diesem Abschnitt wenden wir uns der Auswertung zu. Als erstes beschäftigen wir uns mit dem Einfluss der *Data Guides* auf die Laufzeiten des XMark-Benchmarks in Abhängigkeit zu verschiedenen Dateigrößen. Anschließend betrachten wir den Einfluss der *Statistical View* auf die Laufzeiten der einzelnen XMark-Queries ohne und mit *Data Guides*.

6.5.1 Einfluss der *Data Guides*

Der interessanteste Aspekt bei der Auswertung ist der Einfluss der *Data Guides* auf die Laufzeit des XMark-Benchmarks. Damit aussagekräftige Ergebnisse erzeugt werden, müssen wir verschiedene Größen von XML-Dokumenten verwenden, um die Skalierung der *Data Guides* beobachten zu können. Dazu lassen wir die XMark-Queries auf den verschiedenen Dateigrößen laufen. Wir erhalten interessante Zeitunterschiede zwischen den XMark-Queries ohne *Data Guides* und jenen mit annotierten *Data Guides*.

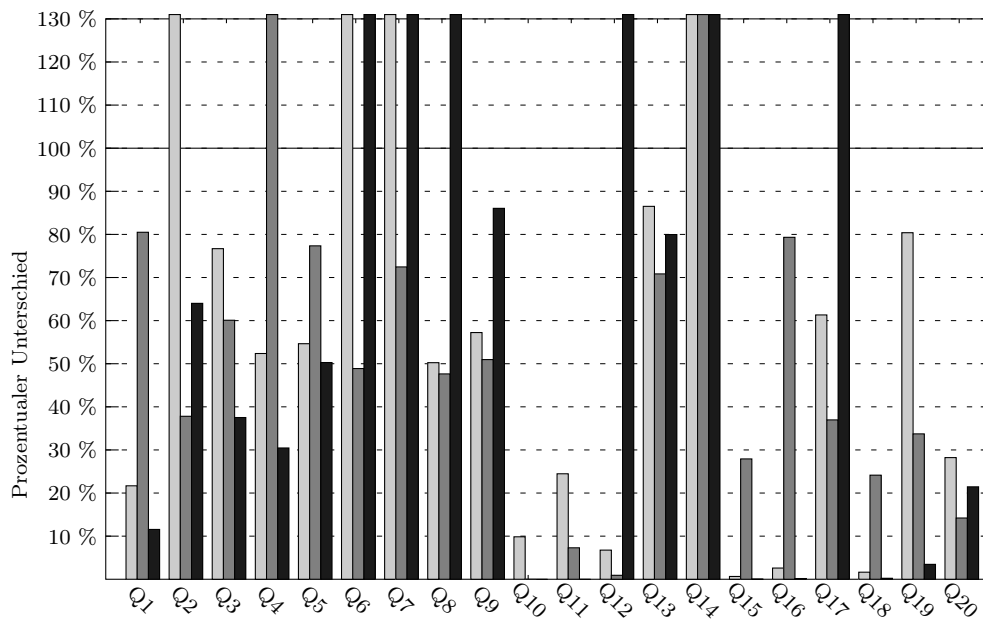


Abbildung 6.2: Prozentualer Unterschied der einzelnen XMark-Queries ohne und mit *Data Guides* auf den Dateigrößen 1,2 MB (□), 12 MB (■) und 112 MB (■). Die 100% Marke stellt die Queries ohne *Data Guides* dar

In Abbildung 6.2 sind auf der X-Achse die XMark-Queries aufgelistet. Zu jeder Abfrage sehen wir drei verschieden eingefärbte Balken. Die hellgrauen (□) Balken stehen für das XML-Dokument mit 1,2 MB, die dunkelgrauen (■) für die Datei mit 12 MB und die schwarzen (■) für die Laufzeiten auf der Größe von 112 MB. Die einzelnen Balken selbst stellen den prozentualen Unterschied zwischen den Laufzeiten der XMark-Queries mit und ohne *Data Guides* dar. Dabei ist die 100% Marke die Laufzeit der Queries ohne *Data Guides*. Alle Balken die sich darunter befinden bedeuten eine Verbesserung der Laufzeit durch *Data Guides*. Zu beachten ist, dass die Balken welche über die 130% Marke hinausreichen sehr viel mehr Zeit benötigen, als die 130%. Aus Platzgründen jedoch wird dies nur angedeutet. Der genaue Prozentsatz kann aus den Tabellen im Anhang C entnommen werden.

Wie wir erkennen können, fällt das Ergebnisse mit *Data Guides* nicht immer positiv aus. Je nach Dateigröße verschlechtern sich Queries erheblich. Bei dem XML-Dokument von 1.2 MB sind dies die Abfragen Q02, Q06, Q07 und Q14. Am besten schneidet die Datei mit 12 MB ab. Bei dieser Größe verschlechtern sich nur zwei Queries: Q04 und Q14. Beim Dokument von 112 MB, auf welches wir unser stärkstes Augenmerk legen wollen, hat sich die Ausführungszeit von sechs Queries verschlechtert: Q06, Q07, Q08, Q12, Q14 und Q17. Andererseits gibt es einige Queries, die sich bedeutend verbessert

haben. Da sticht besonders Q15 und Q16 auf dem größten Dokument ins Auge. Diese beiden befassen sich vorwiegend mit der Auswertung von langen Pfadausdrücken und profitieren am besten von den *Data Guides*. Q10 und Q11 hingegen lieferten auf den größeren Dokumenten noch nach zehn Stunden kein Ergebnis zurück. Aus diesem Grund können wir keine Aussage machen.

Auffällig ist, dass sich die Abfragen nicht immer nur auf dem größten Dokument verschlechtert haben, sondern auch auf den kleineren. Daraus können wir schließen, dass nicht das Konzept der *Data Guides* alleine die Ursache für die Laufzeiten ist, sondern auch die Abarbeitung durch DB2[®]. Abhängig von der Dateigröße werden die Anfragen unterschiedlich kompiliert und es kommt zu den verschiedenen Abweichungen. Ein Konzept für die Verbesserung der Laufzeiten betrachten wir im nächsten Abschnitt 6.5.2, wo es um den Einfluss des *Statistical Views* geht. Andere Gründe für die Fehlplanung von DB2[®] werden wir im Abschnitt 6.6 aufzeigen.

6.5.2 Einfluss des *Statistical Views*

Für den Einfluss des *Statistical Views* betrachten wir nun den direkten Vergleich der Laufzeiten mit und ohne *Statistical View* auf dem XML-Dokument von 112 MB. Des Weiteren machen wir die Unterscheidung zwischen den XMark-Queries mit und ohne *Data Guides*.

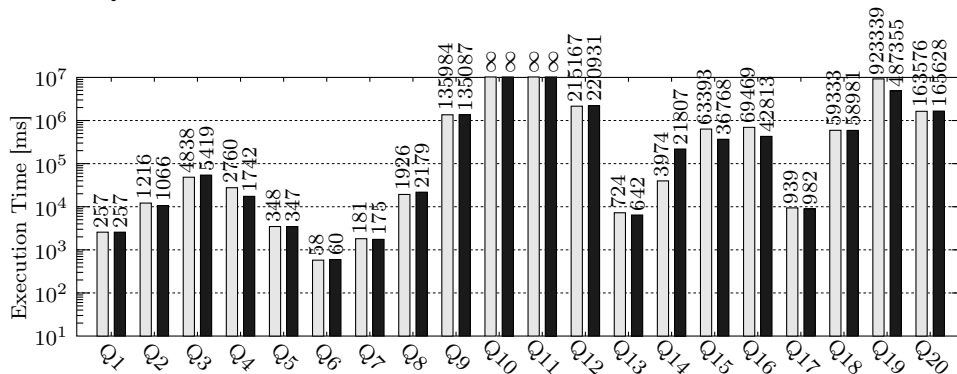


Abbildung 6.3: XMark-Benchmark ohne *Data Guides* auf 112 MB mit *Statistical View* (■) und ohne *Statistical View* (□)

In Abbildung 6.3 sehen wir die Ausführungszeiten der XMark-Queries ohne *Data Guides*. In den grauen Balken ist die Laufzeit ohne das *Statistical View* dargestellt, wogegen die schwarzen Balken die Laufzeit mit dem *Statistical View* zeigen. Wie wir erkennen können, verändert sich die Laufzeit für den Großteil der XMark-Queries nur marginal. Eine Ausnahme bildet Q14, dessen Zeit mehr als fünf mal langsamer wurde. Im Gegensatz dazu haben

sich Q04, Q15, Q16 und Q19 deutlich verbessert. Keinen Vergleich haben wir für die Q10 und Q11, da sie auch nach zehn Stunden noch keine Ergebnisse geliefert haben.

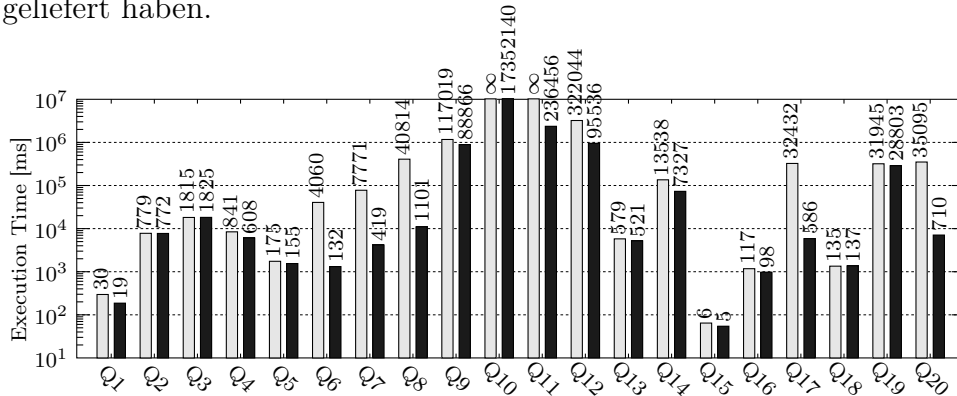


Abbildung 6.4: XMark-Benchmark mit *Data Guides* auf 112 MB mit *Statistical View* (■) und ohne *Statistical View* (■)

Betrachten wir nun die XMark-Queries mit annotierten *Data Guides*. In Abbildung 6.4 sind die Zeiten für die Testläufe mit (graue Balken) und ohne (schwarze Balken) *Statistical View* gegenübergestellt. Bis auf eine Ausnahme (Q18) sind alle Queries mit *Statistical View* schneller. Besonders die Queries Q06, Q07, Q08, Q17 und Q20 haben sich erheblich verbessert. Diese benötigen alle nur noch unter 10% der vorherigen Abarbeitungszeit. Ein weiterer positiver Effekt ist bei Q10 und Q11 ersichtlich, da diese beiden mit den Optimierungen überhaupt zu einem Resultat kommen, was vorher nicht der Fall war (für Details sei auf den Anhang C verwiesen).

Für den direkten Vergleich sind in Abbildung 6.5 die prozentualen Verbesserungen nochmals angeführt. 100% stehen auch hier wieder für: ohne *Statistical View*. Bei den XMark-Queries mit *Data Guides* ist fast durchgehend eine erheblich stärkere Verbesserung zu beobachten als ohne *Data Guides*. Der Umfang der Verbesserung ist allerdings bei den einzelnen Queries verschieden. Sowie beim XMark-Benchmark ohne *Data Guides* als auch mit *Data Guides* treten keine erheblichen Verschlechterungen der Ausführungszeiten auf. Aus diesem Grund können wir guten Gewissens das Konzept der *Statistical View* bei beiden Varianten anwenden, ohne eine zu benachteiligen.

6.6 Beobachtungen

In diesem Abschnitt wollen wir anhand der Query Q06 die einzelnen Schritte eines Testlaufs erarbeiten. Wir beginnen mit der Kompilierung der XMark-Query in einen Algebra-Plan durch *Pathfinder* und dessen Umwandlung in

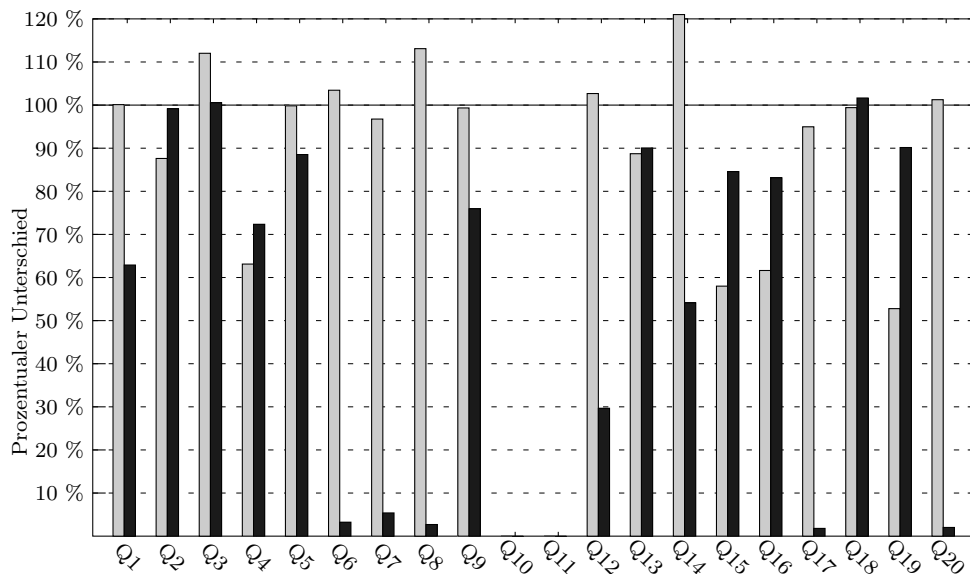


Abbildung 6.5: Prozentuale Veränderung pro XMark-Query durch die *Statistical View*; die 100% stellen die Abfragen ohne *Statistical View* dar, Queries deren Balken isch darunter befindet bedeutet eine Verbesserung; helle Balken (□) stellen Queries ohne *Data Guides* dar, die schwarzen Balken (■) Queries mit *Data Guides*

SQL. Anschließend wird der von DB2[®] erzeugte Zugriffsplan ohne *Statistical View* aufgezeigt und danach die Veränderung durch die Verwendung von *Statistical View*. Des Weiteren werden wir noch Überlegungen für die Verbesserung der Laufzeiten machen.

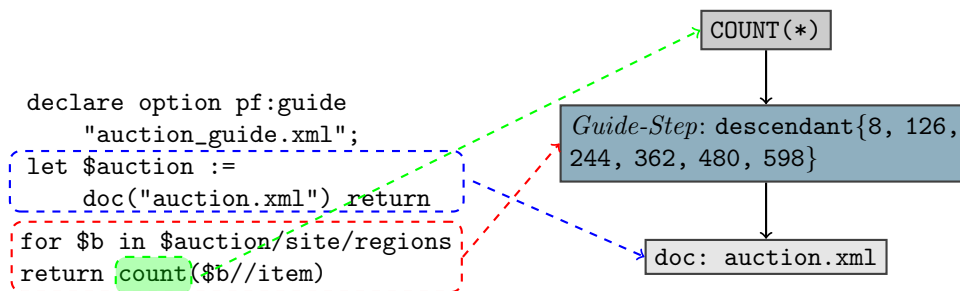


Abbildung 6.6: Kompilierung von XMark-Query Q06 in eine vereinfachte Algebra-Darstellung von *Pathfinder*

In Abbildung 6.6 können wir den Übergang von der Query zum Algebra-Plan sehen. Das Dokument selbst wird auf einen Knoten `doc` abgebildet und dient als Quelle für die Daten. Es ist dementsprechend der erste Kontextknoten der Abfrage und wird im Algebra-Baum ganz unten dargestellt.

Der gesamte rot umrandete Teil der Query kann mit Hilfe der Annotation von *Data Guides* zu einem einzigen Lokalisierungsschritt zusammengefasst werden. Dazu werden alle `item` Knoten von den Teilbäumen der Knoten `$auction/site/regions` ermittelt und deren *Data Guides* bestimmt. Die Achse hat den Wert `descendant`, ausgehend vom Knoten `doc`. Abschließend müssen wir noch die Anzahl der Knoten der Ergebnismenge berechnen. Dies geschieht mit dem Befehl `count` und ist der letzte Schritt der Abfrage.

Bemerkung 6.1. *Der in Abbildung 6.6 dargestellte Algebra-Plan von Q06 ist nur ein Auszug aus dem von Pathfinder erstellten Plan. Aus Gründen der Übersichtlichkeit haben wir ihn kompakter dargestellt und hier nur die relevanten Knoten aufgenommen. Der vollständige Plan ist im Anhang D zu finden.*

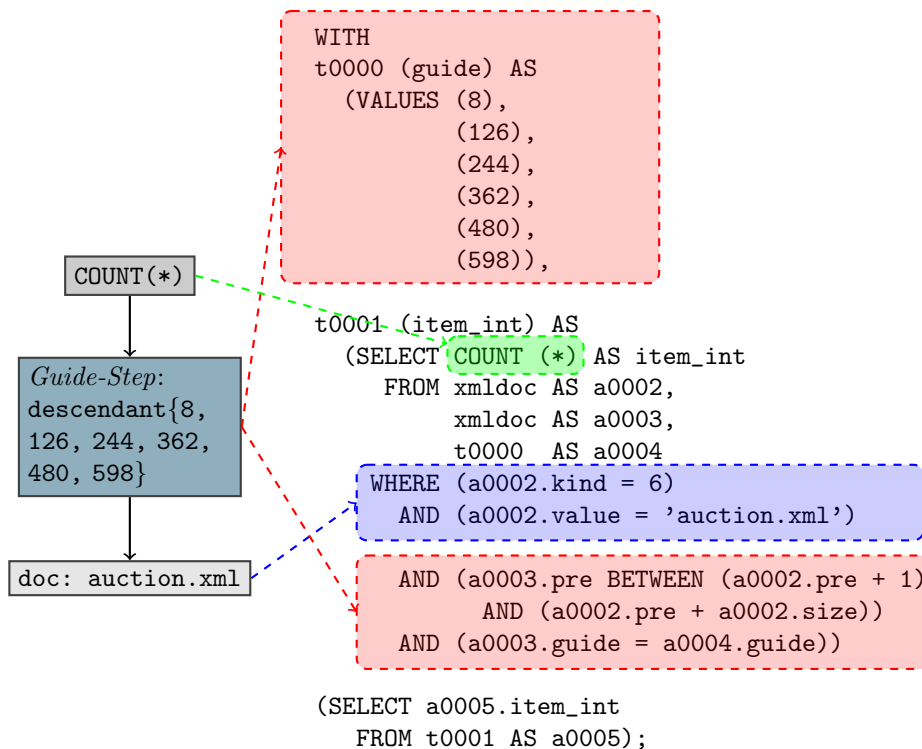


Abbildung 6.7: Transformation der Algebra-Darstellung von *Pathfinder* nach SQL für XMark-Query Q06

Wir haben im vorherigen Abschnitt den Algebra-Plan von *Pathfinder* kennen gelernt. Wir lassen uns nun den SQL-Code von *Pathfinder* generieren, um die einzelnen Schritte genauer zu betrachten. Die Abbildung 6.7 zeigt

die Zuordnung der einzelnen Teile des Algebra-Planes auf SQL. Die gesamte Abfrage wird in ein `WITH`-Statement gekapselt. Dieses enthält mehrere `SELECT`-Anweisungen für die Ermittlung des Ergebnisses. Die einzelnen Schritte für die Umwandlung eines Algebra-Plans nach SQL kann in (GST04; GMR⁺07; May07) nachgelesen werden. Nachfolgend werden wir nur auf einige Konzepte bei der SQL-Code Generierung eingehen.

Der Inhalt des XML-Dokuments `auction.xml` wurde in der Tabelle `XMLDOC` des RDBMS abgelegt, worauf die SQL-Anfragen zugreifen. Der Knoten `doc` greift auf die Tabelle `XMLDOC` zu und überprüft, ob ein Tupel mit dem richtigen Dateinamen (in dieser Beispielabfrage `'auction.xml'`) und `kind` gleich 6 existiert. In Kapitel 3.1 haben wir die Zuordnung von `kind(v)` zu den Elementen `{elem, attr, text, comm, pi, doc}` gesehen. Um Festplattenplatz einzusparen und eine schnellere Auswertung der SQL-Anfragen zu gewährleisten, haben wir den Datentyp von `kind` als Integer realisiert und die Elemente folgendermaßen zugeordnet: `elem=1, attr=2, text=3, comm=4, pi=5` und `doc=6`.

Der *Guide-Step* selbst wird in zwei Teile aufgespalten. Als erstes werden alle *Data Guides* der verschiedenen *Guide-Steps* in getrennte temporäre Tabellen abgelegt. Diese Tabellen besitzen eine Spalte mit dem Namen `guide` und speichern hierin jeden *Data Guide* als separates Tupel ab. In DB2[®] werden die temporären Tabellen mit dem Befehl `GENROW` realisiert. Dieser Operator generiert eine Tabelle mit Zeilen, verwendet aber keine Eingabe aus Tabellen, Indizes oder anderen Operatoren. Bei der Q06 tritt ein einziger *Guide-Step* mit sechs Werten auf und wird mit `t0000` innerhalb des `WITH`-Befehl realisiert. Als zweites muss die Achse, wie in Tabelle 3.3 beschrieben, dargestellt werden. Die entsprechende Kodierung in SQL geschieht mit dem `BETWEEN`-Operator und der Überprüfung der *Data Guides* auf Übereinstimmung.

Der `count`-Operator des Algebra-Plans wird in sein Pendant von SQL übersetzt und liefert bereits das endgültige Ergebnis der XQuery-Abfrage.

Nachdem wir jetzt den von *Pathfinder* generierten SQL-Code kennen, wenden wir uns der Abarbeitung durch DB2[®] zu. Wir werden den Zugriffsplan von XMark-Query Q06 genauer betrachten. Wiederum beginnen wir mit der Auswertung ohne *Statistical View*.

Bemerkung 6.2. Für eine detailliertere Beschreibung der Zugriffspläne von DB2[®] sei auf (ibm07b) verwiesen. Als einzigen Operator betrachten wir hier `IXSCAN` näher. Dieser Operator durchsucht den Index einer Tabelle. Auf welchen Index zugegriffen wird, zeigt der Knoten unterhalb von `IXSCAN`. In Abbildung 6.8 kommen drei verschiedene Indizes vor, dessen dazugehörige Spalten wir für ein besseres Verständnis in Tabelle 6.3 aufgeschlüsselt haben.

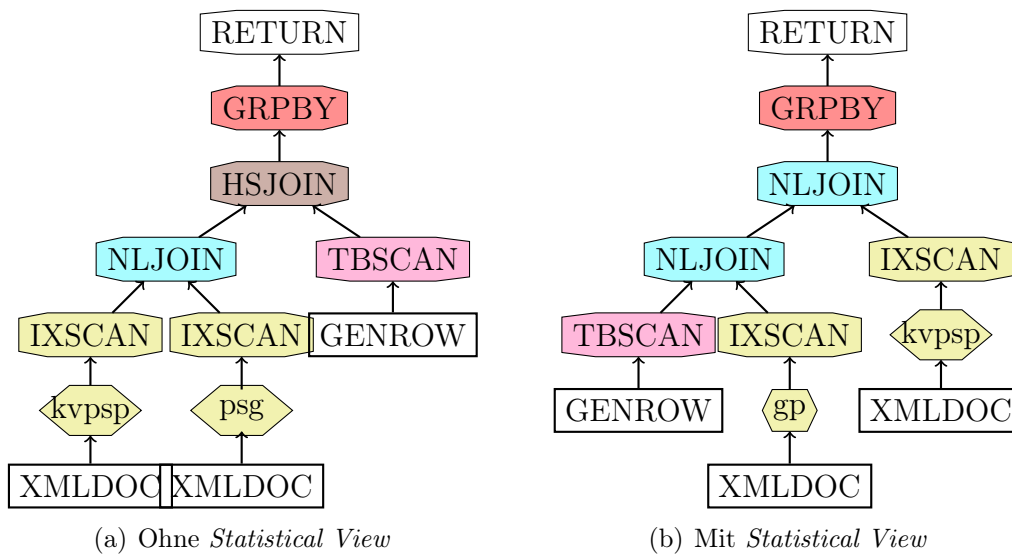


Abbildung 6.8: Zugriffspläne von DB2[®] für Q06 ohne und mit *Statistical View*

Index-Name	Spalten des Index
kvpsp	<i>kind, value, pre+size, pre</i>
psg	<i>pre, size, guide</i>
gp	<i>guide, pre</i>

Tabelle 6.3: Aufschlüsselung der Index-Abkürzungen

Abbildung 6.8(a) zeigt den von *db2expln* erzeugten Plan. Es wird als erstes die Überprüfung auf den richtigen doc Knoten durchgeführt, und alle darin enthaltenen Tupel werden als Ergebnis geliefert. Dies bedeutet, dass das gesamte XML-Dokument in das Zwischenergebnis aufgenommen wird. Anschließend wird mit dem Hash Join auf die richtigen Werte der *Data Guides* überprüft. Diese Auswertungsstrategie ist jedoch eher schlecht, da die Selektivität der *Data Guides* hierbei überhaupt nicht ausgenutzt wird und kein Geschwindigkeitsvorteil entstehen kann.

Dagegen ist in Abbildung 6.8(b) der Zugriffsplan mit *Statistical View* zu sehen. Wie auf den ersten Blick erkennbar, ist der für diesen Fall langsame Hash Join durch einen Nested Loop Join ersetzt worden. Auch die Reihenfolge der Abarbeitung wurde umgestellt: es werden jetzt zuerst die Tupel mit den richtigen *Data Guides* ermittelt und anschließend auf die Dokumentordnung hin überprüft. Der dadurch erhaltene Geschwindigkeitszuwachs wurde bereits im Abschnitt 6.5.2 gezeigt.

Wenn wir die Ausführungszeiten von Q06 auf einem XML-Dokument von

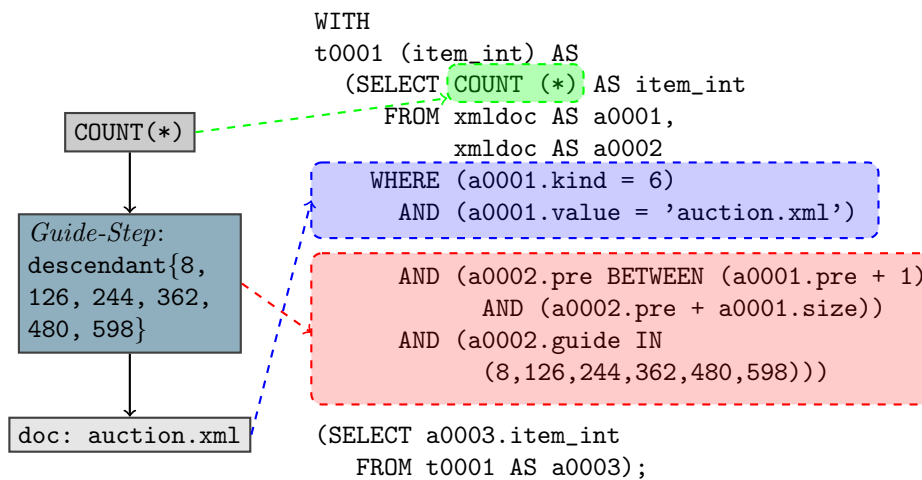


Abbildung 6.9: SQL-Code von Q06 mit dem IN-Operator

112 MB Größe betrachten, kommen wir zu interessanten Beobachtungen: die Ausführungszeit ohne *Data Guides* und ohne *Statistical View* beträgt 57,62 Millisekunden, hingegen die Variante mit *Data Guides* satte 4059,52 Millisekunden. Das entspricht einer 70-fach langsameren Ausführungszeit. Diese Diskrepanz können wir verkleinern, indem wir eine *Statistical View* benutzen und somit die Zeit auf 131,55 Millisekunden drücken, was allerdings immer noch mehr als der doppelten Laufzeit in der ursprünglichen Variante entspricht.

Diesen Umstand ändern wir, indem wir eine andere Kodierung des *Guide-Steps* in SQL verwenden. Um keine temporären Tabellen für die *Data Guides* zu erzeugen, benutzen wir den IN-Operator. Der IN-Operator findet alle Tupel, bei denen die *Data Guides* mit jenen des *Guide-Steps* übereinstimmen. Der daraus erzeugte SQL-Code ist in Abbildung 6.9 dargestellt. Der *Guide-Step* wird in zwei SQL-Befehle übersetzt: in einen BETWEEN-Operator für die Darstellung der Achse und in einen IN-Operator für die Überprüfung der *Data Guides*.

Durch die Verwendung des IN-Operators anstelle von temporären Tabellen in den SQL-Abfragen reduziert sich die Ausführungszeit der Abfrage auf 65,77 Millisekunden. Dies entspricht fast dem anfänglichen Ergebnis der Variante ohne *Data Guides*. Allerdings ist der IN-Operator nicht immer von Vorteil. Besitzt ein *Guide-Step* eine große Anzahl von *Data Guides*, so ist die SQL-Variante mit den temporären Tabellen und der Abbildung auf GONROW schneller. Unsere durchgeführten Tests reichen allerdings nicht dazu aus, exakt angeben zu können, bei welcher Anzahl von *Data Guides* pro *Guide-Step* der Umschwung passiert.

Im nachfolgenden Beispiel zeigen wir eine eher künstliche SQL-Anweisung mit dem UNION ALL-Operator. Dabei teilen wir den *Guide-Step* auf mehrere SELECT-Anweisungen auf. Für jeden *Data Guide* erzeugen wir ein SELECT-Statement, welches die Tupel für einen einzelnen *Data Guide* ermittelt. Des Weiteren werden diese Tupel auf ihre korrekte Position innerhalb der Achse überprüft. Anschließend vereinigen wir die Ergebnisse mit dem UNION ALL-Operator und erhalten so alle gesuchten Tupel für den entsprechenden *Guide-Step*. Der SQL-Code ist im Listing 6.2 aufgezeigt.

Listing 6.2: SQL-Code von Q06 auf Basis des UNION ALL-Operators

```

WITH
t0000 (pre) AS
  ((SELECT a0002.pre AS pre
    FROM A3_AGA.xmldoc AS a0001 ,
         A3_AGA.xmldoc AS a0002
   WHERE (a0001.kind = 6)
        AND (a0001.value = '../.. / document/A3/A3.xml ')
        AND (a0002.pre BETWEEN (a0001.pre + 1)
            AND (a0001.pre + a0001.size))
        AND (a0002.guide = 8))
UNION ALL

... für jeden Data Guide ein eigenes SELECT-Statement ...

UNION ALL
  (SELECT a0002.pre AS pre
    FROM A3_AGA.xmldoc AS a0001 ,
         A3_AGA.xmldoc AS a0002
   WHERE (a0001.kind = 6)
        AND (a0001.value = '../.. / document/A3/A3.xml ')
        AND (a0002.pre BETWEEN (a0001.pre + 1)
            AND (a0001.pre + a0001.size))
        AND (a0002.guide = 598)),

t0001 (item_int) AS
  (SELECT COUNT(*) AS item_int
   FROM t0000 AS a0003)

(SELECT a0004.item_int
  FROM t0001 AS a0004);

```

Erzeugen wir von Listing 6.2 den Zugriffsplan mit DB2[®], so stellen wir eine interessante Eigenschaft fest: in Abbildung 6.10 sehen wir, dass der GRPBY-Operator am UNION ALL-Operator vorbei nach unten versetzt wurde. Diese Optimierung bringt einen deutlichen Vorteil gegenüber dem IN- und GENROW-Operator. Führen wir diese Variante von Q06 auf dem Dokument mit 112 MB aus, so ergibt sich eine Laufzeit von 15,47 Millisekunden gegenüber den ursprünglichen 57,62 Millisekunden. Dies entspricht einer Steigerung um das Dreifache.

Wie sich die Variante mit dem UNION ALL-Operator ohne Aggregatfunktion verhält, wurde hier nicht getestet. Aus diesem Grunde können wir keinen direkten Vergleich zu den SQL-Anweisungen mit IN- oder GENROW-Befehl machen. Wir erkennen aus diesem Experiment, dass DB2[®] eine bessere Optimierung des Zugriffsplans ermöglicht, indem der GRPBY-Operator nach unten verschoben werden kann.

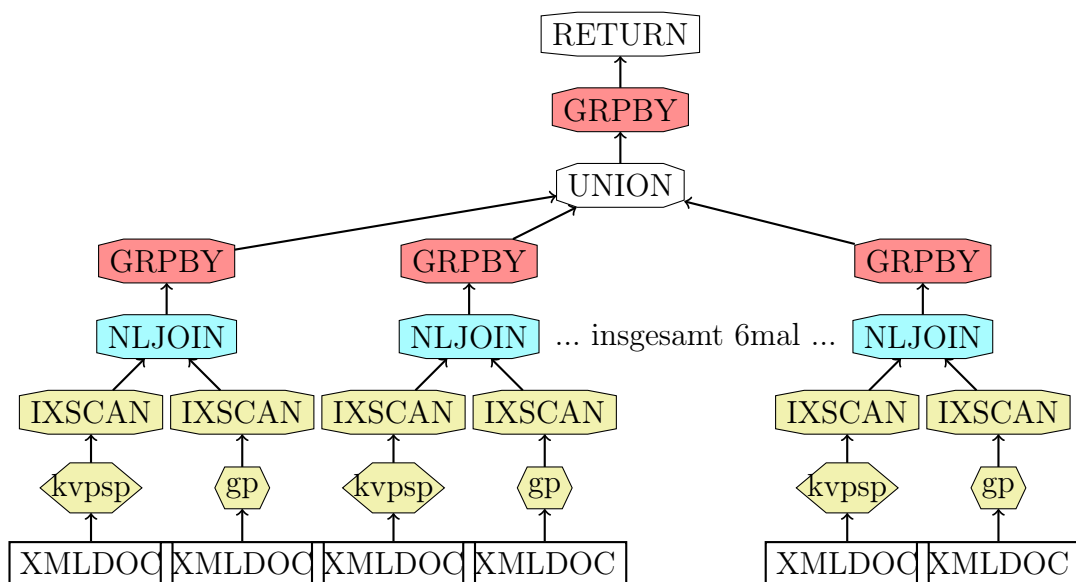


Abbildung 6.10: Zugriffsplan von Q06 mit UNION ALL

Kapitel 7

Fazit

Das Ziel dieser Diplomarbeit ist die Verbesserung der Pfadauswertung von XPath. Wenn wir nun einen Blick auf die Performance-Tests von Kapitel 6 werfen, so wird offenbar, dass wir das Ziel, eine Performance-Verbesserung, erreicht haben. Der XMark-Benchmark deckt einen breiten Teil der XQuery-Funktionalitäten ab, weshalb unsere Testergebnisse für XQuery einen durchaus allgemeingültigen Charakter haben. Auch die Unterschiede in der Verarbeitungsgeschwindigkeit fallen deutlich aus, besonders bei den XMark-Abfragen, mit besonderem Augenmerk auf der Pfadauswertung (Q15, Q16). Je umfangreicher die XML-Dokumente werden, umso mehr fallen die Pfadauswertungen ins Gewicht und beeinflussen die Verarbeitungszeit.

Nachdem wir das Konzept der *Data Guides* definiert und erörtert haben, wurde der Algorithmus für die Vereinigung von Lokalisierungsschritten erklärt. Die *Data Guides* ermöglichen nicht nur eine Optimierung der XPath-Pfade, sondern auch die Optimierung anderer Konzepte. Einige dieser Optionen haben wir in Kapitel 5.3 kennen gelernt. In *Pathfinder* sind allerdings zur Zeit noch nicht alle Optimierungsmöglichkeiten implementiert. So könnten in einem weiteren Schritt der `string-join`, `merge-adjacent` und die Optimierung mit dem `count`-Operator implementiert werden.

Zusammenfassend können wir festhalten, dass mit den *Data Guides* ein elegantes Konzept, nicht nur für die Optimierung der XPath-Pfadauswertung, sondern auch für die Vereinfachung anderer Konstrukte, entwickelt wurde.

Literaturverzeichnis

- [BCF07] BOAG, S. ; CHAMBERLIN, D. ; FERNÁNDEZ, M.: *XQuery 1.0: An XML Query Language*. W3 Consortium. <http://www.w3.org/TR/xquery/>. Version: 2007
- [BMR05] BONCZ, P. ; MANEGOLD, S. ; RITTINGER, J.: Updating the Pre/Post Plane in MonetDB/XQuery. In: *Proceedings of the ACM SIGMOD/PODS 2nd International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P 2005)*. Baltimore, MD, USA, 2005
- [CD99] CLARK, J. ; DEROSE, S.: *XML Path Language (XPath) Version 1.0*. W3 Consortium. <http://www.w3.org/TR/xpath>. Version: 1999
- [Che06] CHEN, K. K.: Influence query optimization with optimization profiles and statistical views in DB2 9. Version: December 2006. <http://www.ibm.com/developerworks/db2/library/techarticle/dm-0612chen/index.html>. 2006. – Forschungsbericht
- [D.B02] D.BROWNELL: *SAX2*. O'Reilly Media, 2002
- [E.C05] E.CERAMI: *XML for Bioinformatics*. Springer, 2005
- [ezb06] *Der einheitliche EURO-Zahlungsverkehrsraum (SEPA)*. Europäische Zentralbank. http://www.ecb.int/pub/pdf/other/sepa_brochure_2006de.pdf. Version: 2006
- [Fec06] FECHNER, D.: *Distribution Statistics uses with the DB2 optimizer*. <http://www.ibm.com/developerworks/db2/library/techarticle/dm-0606fechner/index.html>. Version: June 2006

- [FGE01] F.ACHARD ; G.VAYSSEIX ; E.BARILLOT: XML, bioinformatics and data integration. In: *Bioinformatics* (2001), February, S. 115–125
- [GMR⁺07] GRUST, T. ; MAYR, M. ; RITTINGER, J. ; SAKR, S. ; TEUBNER, J.: A SQL:1999 Code Generator for the Pathfinder XQuery Compiler. In: *Proceedings of the 26th ACM SIGMOD International Conference on Management of Data*. Beijing, China, June 2007
- [Gra]
- [GRT07] GRUST, T. ; RITTINGER, J. ; TEUBNER, J.: Why Off-The-Shelf RDBMS are Better at XPath Than You Might Expect. In: *Proceedings of the 26th ACM SIGMOD International Conference on Management of Data, 2007*
- [GST04] GRUST, T. ; SAKR, S. ; TEUBNER, J.: XQuery on SQL Hosts. In: *Proceedings of the 30th International Conference on Very Large Databases*. Toronto, Canada, August/September 2004
- [GT04] GRUST, T. ; TEUBNER, J.: Relational Algebra: Mother Tongue – XQuery: Fluent. In: *TDM'04, the first Twente Data Management Workshop on XML Databases and Information Retrieval, 2004*
- [ibm07a] *DB2 Systembefehle*. IBM. <http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/core/r0002452.htm>. Version: 2007
- [ibm07b] *EXPLAIN-Operatoren*. IBM. <http://publib.boulder.ibm.com/infocenter/db2luw/v9/topic/com.ibm.db2.udb.admin.doc/doc/c0021198.htm>. Version: 2007
- [KE04] KEMPER, A. ; EICKLER, A.: *Datenbanksysteme - Eine Einführung*. Oldenbourg Wissenschaftsverlag GMBH, 2004
- [KW03] KAY, M. ; WALSH, N.: *XSLT 2.0 and XQuery 1.0 Serialization*. W3 Consortium. <http://www.w3.org/TR/2003/WD-xslt-xquery-serialization-20030502>. Version: 2003
- [May07] MAYR, M.: *Ein SQL:99 Codegenerator für Pathfinder*, Technische Universität München, Diplomarbeit, 2007
- [MMW07] MALHOTRA, A. ; MELTON, J. ; WALSH, N.: *XQuery 1.0 and XPath 2.0 Functions and Operators*. W3 Consortium. <http://www.w3.org/TR/xquery-operators/>. Version: 2007

- [Mon] MONETDB: *Comparison of MonetDB with other XQuery systems and previous MonetDB versions.* MonetDB. <http://monetdb.cwi.nl/projects/monetdb/XQuery/Benchmark/XMark/index.html>
- [OOP⁺04] O'NEIL, P. ; O'NEIL, E. ; PAL, S. ; CSERI, I. ; SCHALLER, G.: ORDPATHs: Insert-Friendly XML Node Labels. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2004
- [PCS⁺04] PAL, S. ; CSERI, I. ; SEELIGER, O. ; SCHALLER, G. ; GIAKOU MAKIS, L. ; ZOLOTOV, V.: Indexing XML Data Stored in a Relational Database. In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, 2004
- [SS02] SCHICKINGER, T. ; STEGER, A.: *Diskrete Strukturen Band 2*. Springer, 2002
- [SWK⁺02] SCHMIDT, A. R. ; WAAS, F. ; KERSTEN, M. L. ; CAREY, M. J. ; MANOLESCU, I. ; BUSSE, R.: XMark: A Benchmark for XML Data Management. In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. Hong Kong, China, August 2002, S. 974–985

Abbildungsverzeichnis

1.1	Konvertierung von XML/XQuery nach RDBMS/SQL	2
2.1	Baumstruktur eines XML-Dokumentes.	6
2.2	Aufbau von Lokalisierungsschritten	8
2.3	Absoluter Lokalisierungspfad	8
3.1	<i>pre</i> , <i>post</i> , <i>size</i> und <i>level</i> -Werte für ein XML-Dokument	12
3.2	Baumstruktur eines XML-Dokumentes.	14
4.1	Schritte der <i>Data Guide</i> -Konstruktion	17
4.2	Unterschiedliche Graphentypen	19
4.3	Varianten eines Wurzelbaumes	19
4.4	Partitioniertes XML-Dokument durch <i>level/kind</i>	22
4.5	Die <i>Data Guides</i> des XML-Dokumentes aus Abbildung 3.2(a)	24
5.1	<i>preceding</i> -Achse	27
5.2	Auswerten der Lokalisierungsschritte <i>child</i> und <i>descendant</i>	32
5.3	Auswerten zweier <i>child</i> -Schritte nacheinander	33
5.4	Baumstruktur eines XML-Dokumentes.	38
6.1	DTD der von <i>xmlgen</i> erzeugten XML-Dateien	43
6.2	Prozentualer Unterschied der einzelnen <i>XMark</i> -Queries ohne und mit <i>Data Guides</i>	51
6.3	<i>XMark</i> -Benchmark ohne <i>Data Guides</i>	52
6.4	<i>XMark</i> -Benchmark mit <i>Data Guides</i>	53
6.5	Prozentuale Veränderung pro <i>XMark</i> -Query durch die <i>Statistical View</i>	54
6.6	Kompilierung von <i>XMark</i> -Query Q06 nach Algebra-Darstellung	54
6.7	Transformation der Algebra-Darstellung nach SQL	55
6.8	Zugriffspläne von DB2 [®] für Q06 ohne und mit <i>Statistical View</i>	57
6.9	SQL-Code von Q06 mit dem IN-Operator	58
6.10	Zugriffsplan von Q06 mit UNION ALL	60

D.1	Algebra-Plan von Q06 mit <i>Data Guides</i>	80
D.2	Algebra-Plan von Q06 ohne <i>Data Guides</i>	81

Tabellenverzeichnis

2.1	Abkürzungen von Lokalisierungsschritten	10
3.1	Tabellen-basierte Repräsentation der XML-Dokumente für ein RDBMS	13
3.2	Tabellen-basierte Repräsentation des XML-Dokumentes aus Abbildung 3.2(a)	15
3.3	Darstellung der XPath-Achsen mit <i>pre/size/level</i>	16
4.1	Kodierung des XML-Dokumentes aus Abbildung 3.2(a) und mit <i>Data Guides</i> aus den Werten <i>pre/size/level</i>	23
5.1	Zusammenfassen von zwei XPath-Achsen	30
5.2	Resultierende Achsenrichtung beim Zusammenfassen zweier im Baum nach unten gerichteter Lokalisierungsschritte	33
5.3	Resultierende Achsenrichtung beim Zusammenfassen zweier im Baum nach oben gerichteter Lokalisierungsschritte	34
5.4	<i>Data Guide</i> -Kodierung mit <i>min/max</i> und <i>count</i> -Werten	39
6.1	Schema für die RDBMS-Tabelle XMLDOC	48
6.2	Auszug aus den von <i>db2advsi</i> ermittelten Indizes	49
6.3	Aufschlüsselung der Index-Abkürzungen	57
C.1	Laufzeit der XMark-Queries auf 1,2 MB ohne <i>Statistical View</i>	76
C.2	Laufzeit der XMark-Queries auf 12 MB ohne <i>Statistical View</i>	77
C.3	Laufzeit der XMark-Queries auf 112 MB ohne <i>Statistical View</i>	77
C.4	Laufzeit der XMark-Queries auf 1,2 MB mit <i>Statistical View</i>	78
C.5	Laufzeit der XMark-Queries auf 12 MB mit <i>Statistical View</i>	78
C.6	Laufzeit der XMark-Queries auf 112 MB mit <i>Statistical View</i>	79

Anhang A

Shredder

Wie in Kapitel 3.2 beschrieben, müssen XML-Dokumente auf relationalen Datenbanken abgebildet werden, um die Vorteile dieser Datenbanken ausnutzen zu können. Diese Aufbereitung, auch transformieren genannt, von XML-Dokumenten wird mit einem speziellen Programm bewerkstelligt, dem *Shredder*. In diesem Kapitel werden wir einen Einblick in die Arbeitsweise des *Shredders* vermitteln und die erzeugten Ausgabedaten betrachten.

A.1 Aufbau des Shredders

Um die XML-Dokumente einzulesen und in ihre Teile aufzuspalten, verwenden wir einen *SAX*-Parser. *SAX* (Simple API for XML) ist eine Programmierschnittstelle (API) für das Parsen von XML-Dokumenten und ist frei erhältlich. Dieser Parser liest ein XML-Dokument sequentiell ein und ruft sogenannte *callback functions* auf, welche durch den Benutzer angepasst werden können. Für ausführliche Informationen sei auf (D.B02) verwiesen.

Mit Hilfe des Parsers wandeln wir ein XML-Dokument in die entsprechende tabellen-basierte Repräsentation für das RDBMS um, ohne dass damit Informationen verloren gehen (Kapitel 3.2). Dabei genügt uns der sequentielle Zugriff auf das XML-Dokument. Bei großen Dokumenten bringt das einen erheblichen Vorteil, da wir nicht die gesamte Datei auf einmal im Hauptspeicher halten müssen, sondern diese schrittweise einlesen können.

Jetzt skizzieren wir, wie ein XML-Dokument auf die Tabelle abgebildet wird. Wir beginnen mit dem *pre*-Wert. Immer wenn durch den *SAX*-Parser eine *callback function* aufgerufen wird, welche einen neuen Knoten öffnet, zählen wir eine globale Variable nach oben und weisen den Wert dem aktuellen Knoten zu. Dasselbe geschieht bei *level*, nur dass dieser Wert beim Schließen eines Knotens wieder dekrementiert werden muss. Die Werte für die Spalten *kind*, *value* und *name* können direkt aus den Parametern der *callback function* ermittelt werden und bedürfen keiner näheren Erklärung. Die Größe des Teilbaumes eines Knotens (*size*) muss

aus den *pre*-Werten berechnet werden. Dazu werden die Knoten, von denen ein Start-Tag jedoch kein End-Tag vorgefunden wurde, auf einen Stack abgelegt. Wird ein Knoten durch ein End-Tag geschlossen, so können wir die Größe des Teilbaums berechnen.

Um die *Data Guides* zu ermitteln, wird während der Transformierung des Dokumentes ein *Data Guide*-Baum aufgebaut, in welchem alle wichtigen Informationen über die Struktur der *Data Guides* abgespeichert werden. Mit Hilfe des *Data Guide*-Baumes wird überprüft, ob ein Knoten dieselben Eigenschaften besitzt wie ein vorhergehender und dementsprechend den selben *Data Guide* zugeordnet bekommt. Existiert der Knoten noch nicht, so wird er im *Data Guide*-Baum neu angelegt. Auch die Werte *min*, *max* und *count* (genauere Beschreibung in Kapitel 5.3.1) werden an dieser Stelle ermittelt.

A.2 Ausgabe des *Shredders*

Nachdem wir die Funktionsweise des *Shredders* betrachtet haben, widmen wir uns der Ausgabe. Der Befehl für das Shredden von XML-Dokumenten lautet `pfshred` und wurde in *Pathfinder* integriert. Als Parameter wird mindestens eine XML-Datei und ein Präfix für die Ausgabedateien benötigt. Ein Beispielaufwurf ist in Listing A.1 angegeben.

Listing A.1: Beispielaufwurf für den *Shredder*

```
pfshred -f auction.xml -o auction
```

Mit dem Befehl aus A.1 werden drei neue Dateien generiert. Die Datei "`auction.csv`" beinhaltet die Informationen des XML-Dokumentes, allerdings für das Einlesen in ein RDBMS aufbereitet. Die Namen der Knoten sind hier nicht enthalten, sondern nur „Fremdschlüssel“, welche wir den unterschiedlichen Knotennamen zuordnen können. Die Namen der Knoten werden getrennt in der Datei "`auction_names.csv`" ausgegeben. In einem RDBMS können diese Informationen durch einen Join verknüpft werden.

Die dritte Datei "`auction_guide.xml`" beinhaltet die Informationen über die *Data Guides*. Das Format ist wiederum XML; die dazugehörige DTD ist in A.2 angeführt. Sie beinhaltet für jeden *Data Guide*-Knoten die Informationen *guide*, *count*, *min*, *max*, *kind* und *name*. Um die Optimierung in *Pathfinder* zu ermöglichen wird diese Datei eingelesen und für die Annotation von *Data Guides* verwendet.

Listing A.2: DTD für die `*_guide.xml`

```
<!ELEMENT node (EMPTY | node*)>
<!ATTLIST node guide CDATA #REQUIRED>
<!ATTLIST node count CDATA #REQUIRED>
<!ATTLIST node min CDATA #REQUIRED>
<!ATTLIST node max CDATA #REQUIRED>
<!ATTLIST node kind (1|2|3|4|5|6) #REQUIRED>
<!ATTLIST node name CDATA #IMPLIED>
```

Anhang B

XMark-Queries

XMark Q01

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $b in $auction/site/people/person[@id = "person0"]
return $b/name/text()
```

XMark Q02

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
return <increase>{$b/bidder[1]/increase/text()}</increase>
```

XMark Q03

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
where zero-or-one($b/bidder[1]/increase/text()) * 2
  <= $b/bidder[last()]/increase/text()
return
  <increase
  first="{ $b/bidder[1]/increase/text()}"
  last="{ $b/bidder[last()]/increase/text()}" />
```

XMark Q04

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
where
  some $pr1 in $b/bidder/personref[@person = "person20"],
  $pr2 in $b/bidder/personref[@person = "person51"]
  satisfies $pr1 << $pr2
return <history>{$b/reserve/text()}</history>
```

XMark Q05

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
count(
  for $i in $auction/site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price
)
```

XMark Q06

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $b in $auction/site/regions
return count($b//item)
```

XMark Q07

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $p in $auction/site
return
  count($p//description) + count($p//annotation) + count($p//emailaddress)
```

XMark Q08

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $a :=
  for $t in $auction/site/closed_auctions/closed_auction
  where $t/buyer/@person = $p/@id
  return $t
return <item person="{ $p/name/text() }">{count($a)}</item>
```

XMark Q09

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
let $ca := $auction/site/closed_auctions/closed_auction return
let $ei := $auction/site/regions/europe/item
for $p in $auction/site/people/person
let $a :=
  for $t in $ca
  where $p/@id = $t/buyer/@person
  return
  let $n := for $t2 in $ei where $t/itemref/@item = $t2/@id return $t2
  return <item>{$n/name/text()}</item>
return <person name="{ $p/name/text()}">{$a}</person>
```

XMark Q10

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $i in
  distinct-values($auction/site/people/person/profile/interest/@category)
let $p :=
  for $t in $auction/site/people/person
  where $t/profile/interest/@category = $i
  return
  <personne>
    <statistiques>
      <sexe>{$t/profile/gender/text()}</sexe>
      <age>{$t/profile/age/text()}</age>
      <education>{$t/profile/education/text()}</education>
      <revenu>{fn:data($t/profile/@income)}</revenu>
    </statistiques>
    <coordonnees>
      <nom>{$t/name/text()}</nom>
      <rue>{$t/address/street/text()}</rue>
      <ville>{$t/address/city/text()}</ville>
      <pays>{$t/address/country/text()}</pays>
      <reseau>
        <courrier>{$t/emailaddress/text()}</courrier>
        <pagePerso>{$t/homepage/text()}</pagePerso>
      </reseau>
    </coordonnees>
    <cartePaiement>{$t/creditcard/text()}</cartePaiement>
  </personne>
return <categorie>{<id>{$i}</id>, $p}</categorie>
```

XMark Q11

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $l := for $i in $auction/site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
return <items name="{ $p/name/text() }">{count($l)}</items>
```

XMark Q12

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $l := for $i in $auction/site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i where $p/profile/@income > 50000
return <items person="{ $p/profile/@income }">{count($l)}</items>
```

XMark Q13

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $i in $auction/site/regions/australia/item
return <item name="{ $i/name/text() }">{ $i/description }</item>
```

XMark Q14

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $i in $auction/site//item
where contains(string(exactly-one($i/description)), "gold")
return $i/name/text()
```

XMark Q15

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $a in
  $auction/site/closed_auctions/closed_auction/annotation/description/
  parlist/listitem/parlist/listitem/text/emph/keyword/text()
return <text>{ $a }</text>
```

XMark Q16

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $a in $auction/site/closed_auctions/closed_auction
where
  not(
    empty(
      $a/annotation/description/parlist/listitem/parlist/listitem/
      text/emph/keyword/text()
    )
  )
return <person id="{ $a/seller/@person }"/>
```

XMark Q17

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
where empty($p/homepage/text())
return <person name="{ $p/name/text() }"/>
```

XMark Q18

```
declare namespace local = "http://www.foobar.org";
declare function local:convert($v as xs:decimal?) as xs:decimal?
{
  2.20371 * $v (: convert Dfl to Euro :)
};
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $i in $auction/site/open_auctions/open_auction
return local:convert(zero-or-one($i/reserve))
```

XMark Q19

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
for $b in $auction/site/regions//item
let $k := $b/name/text()
order by zero-or-one($b/location) ascending empty greatest
return <item name="{ $k }">{ $b/location/text() }</item>
```

XMark Q20

```
declare option pf:guide "auction_guide.xml";
let $auction := doc("auction.xml") return
<result>
  <preferred>
    {count($auction/site/people/person/profile[@income >= 100000])}
  </preferred>
  <standard>
    {
      count(
        $auction/site/people/person/
        profile[@income < 100000 and @income >= 30000]
      )
    }
  </standard>
  <challenge>
    {count($auction/site/people/person/profile[@income < 30000])}
  </challenge>
  <na>
    {
      count(
        for $p in $auction/site/people/person
        where empty($p/profile/@income)
        return $p
      )
    }
  </na>
</result>
```

Anhang C

Testzeiten der XMark-Queries

Bemerkung C.1. Falls eine XMark-Abfrage aus Gründen der Performance nicht innerhalb von 10 Stunden zu einem Ende gekommen ist, haben wir sie abgebrochen und das entsprechende Feld in der Tabelle mit einem '*' versehen.

Abfrage	ohne Data Guides [ms]	mit Data Guides [ms]	Unterschied in %
Q01	9,32	2,02	21,68
Q02	13,27	450,77	3395,92
Q03	58,60	44,94	76,69
Q04	17,63	9,23	52,37
Q05	3,69	2,02	54,64
Q06	1,47	42,34	2878,45
Q07	4,09	74,67	1824,88
Q08	30,64	15,39	50,24
Q09	48,42	27,71	57,23
Q10	37780,77	3721,57	9,85
Q11	148,84	36,42	24,47
Q12	354,88	24,00	6,76
Q13	6,63	5,73	86,51
Q14	39,40	110,15	279,58
Q15	123,52	0,80	0,65
Q16	104,39	2,71	2,60
Q17	9,21	5,65	61,32
Q18	87,48	1,44	1,64
Q19	113,47	91,22	80,39
Q20	30,49	8,61	28,22

Tabelle C.1: Laufzeit der XMark-Queries auf 1,2 MB ohne *Statistical View*

Abfrage	ohne <i>Data Guides</i> [ms]	mit <i>Data Guides</i> [ms]	Unterschied in %
Q01	414,27	333,51	80,50
Q02	1045,68	395,42	37,81
Q03	1754,02	1053,92	60,09
Q04	579,07	1779,77	307,35
Q05	411,55	318,35	77,35
Q06	734,42	359,00	48,88
Q07	934,90	677,43	72,46
Q08	2881,54	1372,86	47,64
Q09	3600,71	1834,76	50,96
Q10	*	*	-
Q11	43960,06	3209,23	7,30
Q12	136516,55	1254,55	0,92
Q13	1303,39	923,39	70,84
Q14	599,39	1047,57	174,77
Q15	1089,02	304,09	27,92
Q16	744,73	590,77	79,33
Q17	289,02	106,85	36,97
Q18	1327,20	320,48	24,15
Q19	9325,07	3145,09	33,73
Q20	2091,41	297,17	14,21

Tabelle C.2: Laufzeit der XMark-Queries auf 12 MB ohne *Statistical View*

Abfrage	ohne <i>Data Guides</i> [ms]	mit <i>Data Guides</i> [ms]	Unterschied in %
Q01	256,76	29,68	11,56
Q02	1216,30	778,60	64,01
Q03	4837,82	1815,02	37,52
Q04	2759,73	840,68	30,46
Q05	347,61	174,78	50,28
Q06	57,62	4059,52	7044,85
Q07	180,97	7770,90	4293,98
Q08	1926,37	40813,69	2118,68
Q09	135983,94	117018,78	86,05
Q10	*	*	-
Q11	*	*	-
Q12	215167,40	322043,98	149,67
Q13	723,66	578,71	79,97
Q14	3973,97	13537,83	340,66
Q15	63393,03	6,44	0,01
Q16	69469,86	117,41	0,17
Q17	938,91	32492,07	3460,61
Q18	59333,17	134,74	0,23
Q19	923338,65	31944,83	3,46
Q20	163576,45	35095,02	21,45

Tabelle C.3: Laufzeit der XMark-Queries auf 112 MB ohne *Statistical View*

Abfrage	ohne <i>Data Guides</i> [ms]	mit <i>Data Guides</i> [ms]	Unterschied in %
Q01	3,89	0,60	15,51
Q02	415,35	8,81	2,12
Q03	59,90	14,53	24,25
Q04	17,68	5,04	28,52
Q05	3,75	1,86	49,71
Q06	1,29	0,65	50,15
Q07	4,44	1,44	32,43
Q08	31,18	28,74	92,18
Q09	48,34	41,89	86,65
Q10	38190,99	12299,21	32,20
Q11	149,37	37,03	24,79
Q12	353,41	37,33	10,56
Q13	6,30	5,59	88,69
Q14	39,32	34,23	87,06
Q15	124,21	83,20	66,98
Q16	104,91	1,41	1,35
Q17	8,99	15,50	172,30
Q18	5,71	1,22	21,39
Q19	124,41	7172,82	5765,33
Q20	30,42	7,17	23,57

Tabelle C.4: Laufzeit der XMark-Queries auf 1,2 MB mit *Statistical View*

Abfrage	ohne <i>Data Guides</i> [ms]	mit <i>Data Guides</i> [ms]	Unterschied in %
Q01	436,24	14,77	3,38
Q02	703,29	896,88	127,53
Q03	1820,43	162,29	8,91
Q04	545,29	15,24	2,79
Q05	407,17	15,16	3,72
Q06	8,56	13,84	161,60
Q07	952,31	43,93	4,61
Q08	2830,98	493,72	17,44
Q09	2567,67	1290,25	50,25
Q10	187908,30	2032063,84	1081,41
Q11	43068,09	3943,65	9,16
Q12	268139,85	1658,45	0,62
Q13	1298,62	73,11	5,63
Q14	600,18	356,99	59,48
Q15	1231,04	1,32	0,11
Q16	2189,82	12,24	0,56
Q17	248,54	151,12	60,80
Q18	1025,39	341,93	33,35
Q19	9997,10	4327,10	43,28
Q20	2151,63	561,64	26,10

Tabelle C.5: Laufzeit der XMark-Queries auf 12 MB mit *Statistical View*

Abfrage	ohne <i>Data Guides</i> [ms]	mit <i>Data Guides</i> [ms]	Unterschied in %
Q01	257,04	18,67	7,26
Q02	1065,71	771,86	72,43
Q03	5419,26	1825,03	33,68
Q04	1741,83	608,19	34,92
Q05	347,02	154,73	44,59
Q06	59,61	131,55	220,68
Q07	175,11	419,16	239,37
Q08	2178,50	1100,87	50,53
Q09	135087,40	88865,76	65,78
Q10	*	17352140,21	-
Q11	*	236456,19	-
Q12	220930,88	95536,18	43,24
Q13	642,05	521,20	81,18
Q14	21806,98	7327,38	33,60
Q15	36768,41	5,44	0,01
Q16	42813,45	97,62	0,23
Q17	891,63	586,25	65,75
Q18	58980,97	136,96	0,23
Q19	487355,35	28802,60	5,91
Q20	165627,80	709,90	0,43

Tabelle C.6: Laufzeit der XMark-Queries auf 112 MB mit *Statistical View*

Anhang D

Algebra-Pläne von Query Q06

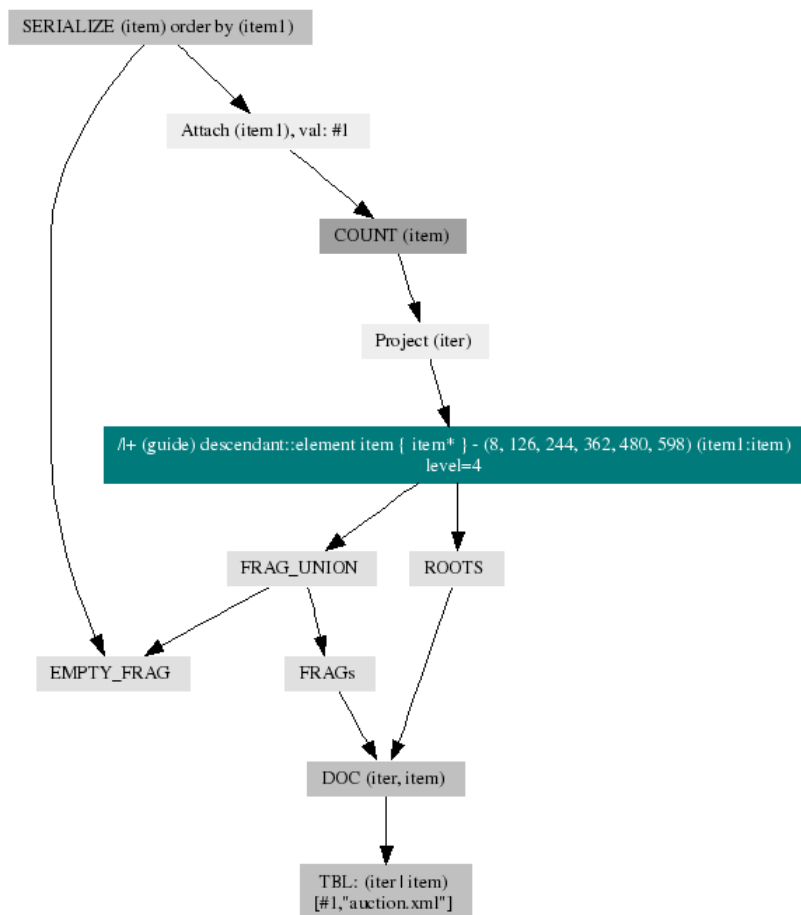


Abbildung D.1: Algebra-Plan von Q06 mit *Data Guides*

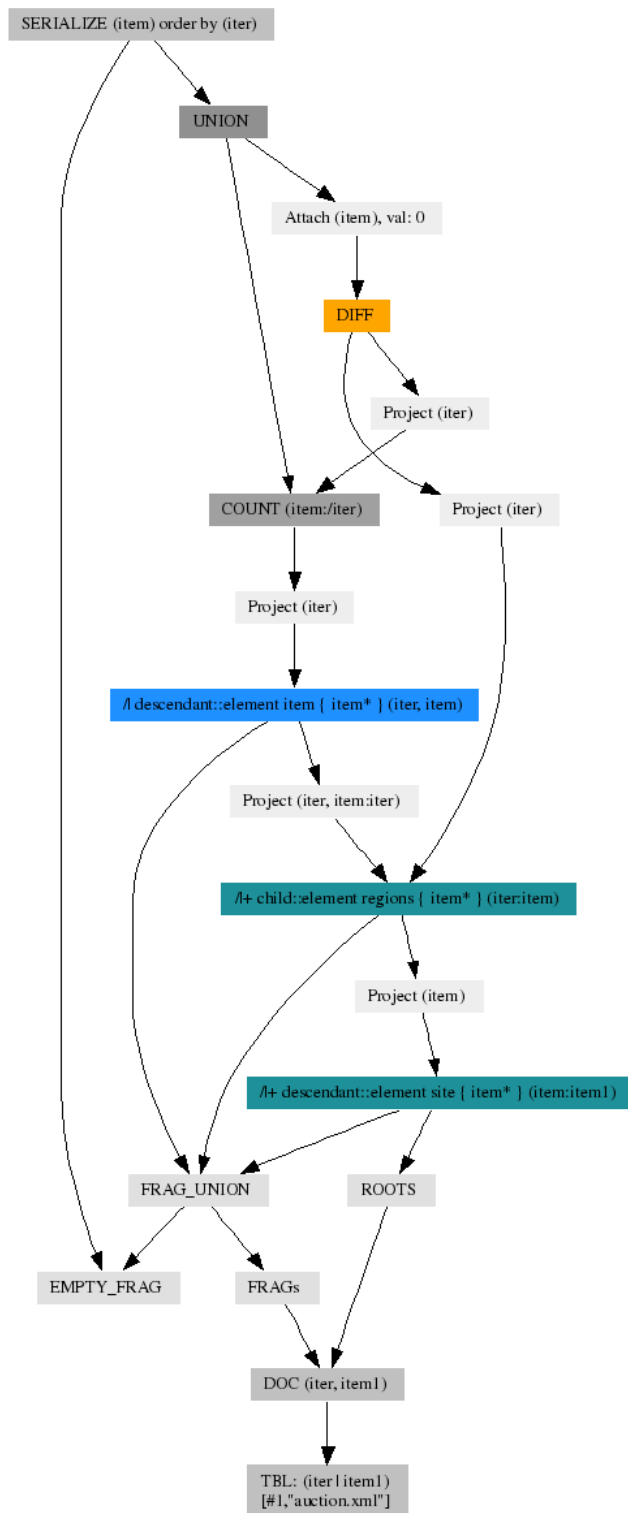


Abbildung D.2: Algebra-Plan von Q06 ohne *Data Guides*