

Lehrstuhl für Datenbanken
Fakultät für Informatik
Technische Universität München



Diplomarbeit
in Informatik

Ein SQL:99 Codegenerator für Pathfinder

A SQL:99 Codegenerator
for Pathfinder

Manuel Mayr
`manuel.mayr@mytum.de`

12. April 2007

Aufgabensteller: Prof. Dr. Torsten Grust
Betreuer: Dr. Jens Teubner
Jan Rittinger, M.Sc.

Erklärung

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

Manuel Mayr,
12. April 2007

Zusammenfassung

Diese Diplomarbeit beschäftigt sich mit der Implementation eines SQL:99 Compilers als Backend-Lösung für *Pathfinder*. Ziel ist es, eine XQuery-Anfrage in semantisch äquivalente SQL-Statements zu transformieren. Diese können für encodierte XML-Dokumente auf relationalen Datenbankmanagementsystemen (RDBMS) ausgeführt werden. Diese Datenbanksysteme spielen eine tragende Rolle bei der Speicherung und Abfrage von Informationen. Durch die in dieser Arbeit vorgestellten Konzepte wird es möglich, relationale Datenbanksysteme in schnelle XML-Prozessoren zu verwandeln.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	2
1.2	Pathfinder	2
1.2.1	Kompilationsphasen von Pathfinder	3
1.2.2	Die Relationale Algebra als Quellsprache	4
1.3	Ziel der Diplomarbeit	4
2	Relationale Algebra	7
2.1	Die Operatoren	8
2.1.1	Tabelle	8
2.1.2	Attach $@_{a:v}$	9
2.1.3	Projektion ($\pi_{a_1:b_1, \dots, a_n:b_n}$)	9
2.1.4	Vereinigung (\cup)	9
2.1.5	Differenz	9
2.1.6	Selektion (σ_p)	10
2.1.7	Kartesisches Produkt (\times)	10
2.1.8	Join (\bowtie)	10
2.1.9	Distinct	10
2.1.10	Der generische Operator (fun_1to1)	11
2.1.11	Bool'sche Ausdrücke	11
2.1.12	Rownumber ($\rho_{b:\langle a_1, \dots, a_n \rangle / p}$)	11
2.1.13	Pfadausdrücke	11
3	SQL:99	14
3.1	Die Sprachkonstrukte	14
3.1.1	Das WITH-Statement und <code>common table expressions</code>	14
3.1.2	Rownumber	15
3.1.3	If-then-else Logik in SQL	16
3.1.4	Duplikateliminierung	17

4	Übersetzung	19
4.1	Die Struktur des Abarbeitungsplans	19
4.2	WITH-Statement	20
4.3	Lazy Binding	21
4.3.1	Auswirkungen der GAG-Struktur auf die Bindung	25
4.4	Document Relation	25
4.5	Die Inferenzregeln	27
4.5.1	Tabelle	29
4.5.2	Attach	30
4.5.3	Vereinigung	30
4.5.4	Selektion	31
4.5.5	Projektion	31
4.5.6	Equi-Join	31
4.5.7	Der generische Operator (fun_1to1)	32
4.5.8	Bool'sche Ausdrücke	32
4.5.9	Rownumber	33
4.5.10	Fragment	34
4.5.11	Pfadausdrücke	34
4.5.12	Elementkonstruktion	36
4.6	Metaregeln	40
4.6.1	Ausnahme bei bool'schen Ausdrücken	41
5	Optimierung	43
5.1	Bündelung	43
5.1.1	Indizes bei Pfadausdrücken	44
5.2	Indizes bei Elementen	45
6	Implementierung	48
6.1	Baumstrukturen	48
6.2	Lazy Binding	50
6.3	Mustererkennung	52
6.3.1	Muster bei der Bündelung von <i>Path-Steps</i>	53
7	Experimente	55
7.1	Vorbereitungen	55
7.1.1	Strategie bei der Auswertung	57
7.2	Bündeln	57
7.3	Keine Attribute	61
7.4	Indizes	62
7.5	Analyse von Q08.xq	63
7.6	Verschiedene Größen	65

8	Ausblick	67
8.1	Unterstützung von Attributen	67
8.2	Selektivität	67
8.3	Materialisierung	68
A	Beispielübersetzung	70
B	Beigelegte CD	77

Kapitel 1

Einleitung

XQuery ist die Zukunft! Während in der Vergangenheit vorwiegend relationale Datenbanksysteme im Bereich der Informationsverarbeitung verwendet wurden, wird heute zunehmend XML eingesetzt. Die Anwendungen von XML reichen von der simplen Strukturierung von Daten, bis hin zu komplexen E-Commerce-Systemen oder Schlagwörtern wie *Semantic Web*. Dank dieser Popularität von XML genießt XQuery, als Anfragesprache für XML-Dokumente denselben Status, wie SQL auf relationalen Datenbanken.

XQuery wird seit 2001 vom World Wide Web Consortium (W3C) entwickelt und ist mittlerweile zur Standardanfragesprache für XML-Dokumente geworden. Während viele Anstrengungen dahingehen XQuery-Implementationen von Grund auf neu zu entwickeln, verfolgen wir mit dem *Pathfinder* einen anderen Ansatz. Wir wollen XQuery auf gängigen relationalen Datenbanken zur Verfügung stellen. Dabei ist es ein Anliegen von *Pathfinder* den Kern der Datenbank so wenig wie möglich zu verändern.

Ein erster Schritt in diese Richtung wurde mit der Unterstützung von XQuery auf *MonetDB*¹ gemacht. Eine XQuery-Anfrage wird hierbei in eine Zwischensprache, namens MIL (Monet Interpreter Language) transformiert, die dann auf *MonetDB* ausgeführt werden kann. Das Resultat ist, dass die relationale Datenbank *MonetDB* nun auch als hoch skalierbarer, performanter XML-Prozessor gebraucht werden kann.

In dieser Diplomarbeit wollen wir einen Schritt weitergehen. Wir distanzieren uns von einer spezifischen Schnittstelle, wie MIL und wollen XQuery auf **jeder beliebigen relationalen Datenbank** unterstützen. Relationale Datenbanken spielen, nach wie vor, eine tragende Rolle bei der Speicherung und Ver-

¹<http://monetdb.cwi.nl>

arbeitung von Informationen. Die Anwendung von neuen Technologien wie XQuery und XML ist für manche Unternehmen unerlässlich. Mit unserem SQL:99-Codegenerator für *Pathfinder* kann ein bewährtes Datenbanksystem zu einem effizienten und hoch skalierbaren XML-Prozessor werden.

1.1 Motivation

Pathfinder wird in dieser Diplomarbeit durch einen neuen Codegenerator erweitert, der **jede** beliebige SQL:99-kompatible Datenbank in schnelle XML-Prozessoren verwandeln kann. Relationale Datenbanken sind ausgereifte und hochgradig komplexe Systeme. Der Ansatz, sich solche etablierten Systeme zu Nutze zu machen, bringt zahlreiche Vorteile mit sich. Viele Jahre der Forschung und Entwicklung machen sie zu den effizientesten Systemen im Bereich der Datenspeicherung und -anfrage. Mit der Entscheidung relationale Datenbanken für unsere Zwecke zu nutzen, übertragen wir dieses reichhaltige Wissen aus der Datenbankforschung direkt auf die Performanz unserer Anfragen.

Pathfinder transformiert XQuery-Ausdrücke in relationale Algebra-Pläne. Die Algebra wird in vielen Schritten zahlreichen Optimierungen unterzogen, die auf SQL-Seite nur sehr schwer durchführbar sind (siehe auch [7]). Die Pläne werden vom SQL-Codegenerator in SQL:99-Ausdrücke verwandelt. Abbildung 1.1 zeigt vereinfacht die Architektur von *Pathfinder*. *Pathfinder*

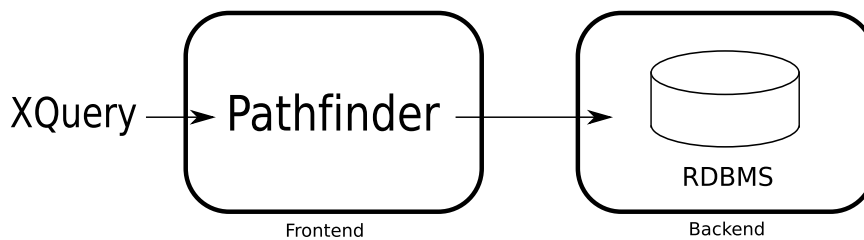


Abbildung 1.1: Architektur von *Pathfinder*.

hat eine typische Frontend-Backend-Architektur. In unserem Fall ist das Backend eine relationale Datenbank.

1.2 Pathfinder

Während Abbildung 1.1 nur einen groben Gesamtüberblick über die Funktionsweise des *Pathfinder*-Compilers liefert, wird im Folgenden Schritt für

Schritt nachvollzogen, wie `Pathfinder` valide XQuery-Anfragen transformiert.

1.2.1 Kompilationsphasen von Pathfinder

In diesem Abschnitt werden wir die wichtigen Schritte genauer betrachten, die eine XQuery-Anfrage durchlaufen muss, bis sie in einen SQL-Ausdruck transformiert werden kann.

Lexikalische Analyse

Bevor mit der Transformation begonnen wird, muss in einem ersten Schritt überprüft werden, ob der übergebene XQuery-Ausdruck gültig ist. Ist dies nicht der Fall, wird der Compilationsprozess abgebrochen und entsprechende Fehlermeldungen werden generiert. Die lexikalische Analyse erfolgt mit `flex` – dessen Ausgabe wird als Eingabe für den Parser verwendet, der wiederum die komplexere Grammatik von XQuery erkennt.

Parzen

Das Parsen wird durch `bison` erleichtert und am Ende dieses Vorgangs steht ein abstrakter Syntaxbaum zur Weiterverarbeitung bereit. Die lexikalische Analyse und das anschließende Parsen gewährleistet die syntaktische Validität der Anfrage.

Core Language

Nachdem die Anfrage auf etwaige Ungereimtheiten geprüft worden ist, wird der abstrakte Syntaxbaum in eine Form gebracht, die der weiteren Verarbeitung entgegenkommt. Der Terminus für diese Repräsentation nennt sich *Core* (zu deutsch: Kern). Diese Kern-Sprache verzichtet auf den *syntaktischen Zucker* von XQuery und macht einige implizite Semantiken explizit.

In einem weiteren Schritt wird der oft unnötig komplexe *Core*-Code normalisiert und mit Hilfe von Static Typing weiter vereinfacht.

Logische Algebra

Die Motivation von `Pathfinder` liegt darin, XQuery-Anfragen auf reinen relationalen Backends auszuführen (siehe auch [10]). Die algebraischen Grundlagen liegen seit den Arbeiten Codd's in Form der relationalen Algebra vor. `Pathfinder` übersetzt den vorliegenden *Core*-Syntaxbaum zum Zweck der

Ausführung auf relationalen Backends in die generischen Ausdrücke der relationalen Algebra (weitere Informationen finden sich in [11]). Hierzu wurde die Algebra für einige spezielle Aufgaben, wie zum Beispiel den XPath-Achsenauswertung erweitert.

Optimierung der logischen Algebra

Die Optimierung der logischen Algebra ist wahrscheinlich der wichtigste Punkt, um Anfragen noch effizienter zu gestalten. So enthält die Algebra, die im vorhergehenden Schritt generiert wurde enthält viele Redundanzen. Um eine mehrmalige Auswertung von gleichen Unterausdrücken zu vermeiden, wird der algebraische Ausdruck in einen äquivalenten Graphen transformiert (GAG), der identische Unterausdrücke nur einmal enthält.

SQL

Der letzte Schritt ist die Generierung des SQL-Ausdrucks aus der logischen Algebra. Diesen Teil werden wir in den folgenden Abschnitten detaillierter behandeln. Abbildung 1.2 zeigt den Vorgang der Kompilation noch einmal schematisch.

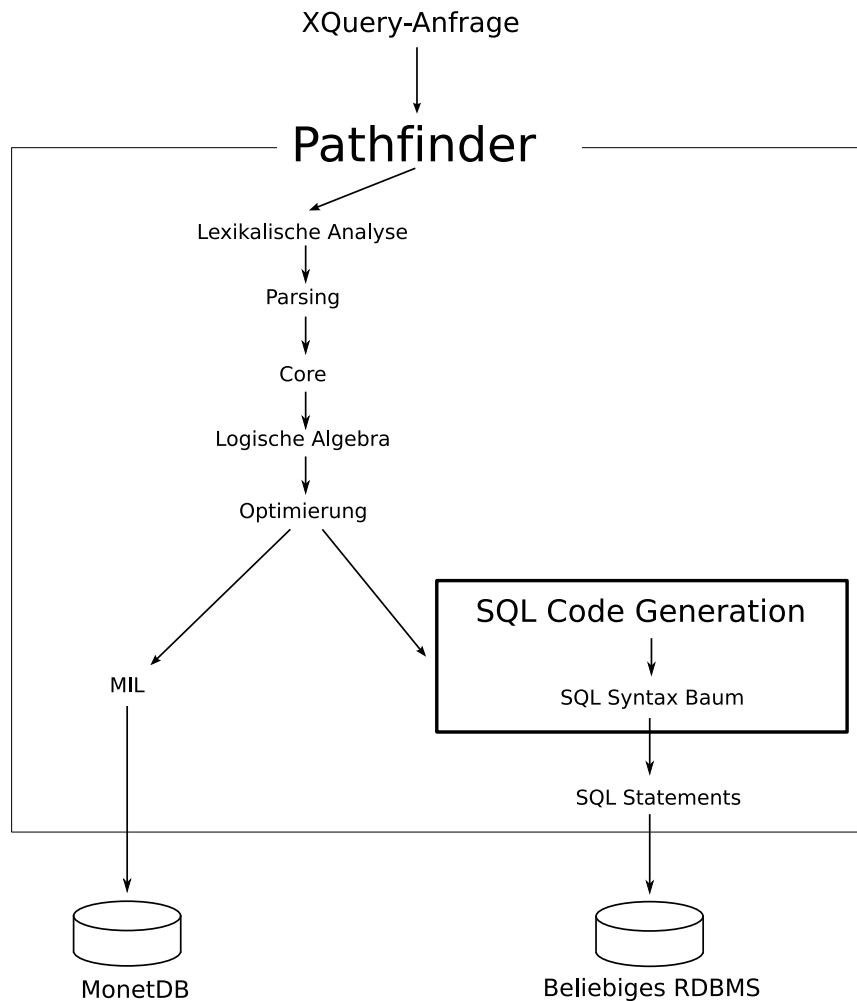
1.2.2 Die Relationale Algebra als Quellsprache

Unser Codegenerator kann von den zahlreichen Optimierungen profitieren, denen die Algebra unterzogen wurde. Zudem macht die große Ähnlichkeit zwischen der relationalen Algebra und SQL sie zu einem guten Ausgangspunkt, um das SQL-Backend direkt an dieser Stelle anzusetzen. Die relationale Algebra wird also als Quellsprache für die Codegenerierung verwendet. Die Anfragepläne der Algebra werden dabei in semantisch äquivalente SQL-Statements transformiert.

1.3 Ziel der Diplomarbeit

In dieser Diplomarbeit wollen wir für XQuery-Anfragen semantisch äquivalente SQL-Anfragen generieren. Damit distanzieren wir uns von einer spezifischen Schnittstelle, wie MIL. Wir beschränken uns auf Sprachkonstrukte der SQL:99-Spezifikation, um die Kompatibilität zu den meisten gängigen RDBMS zu gewährleisten.

In Kapitel 2 und 3 wollen wir uns mit den Eigenheiten der Quell- und Zielsprache vertraut machen. Dabei werden wir uns vorwiegend auf Operatoren

Abbildung 1.2: Architektur von *Pathfinder* (detailliert).

beschränken, die nicht zum eigentlichen Wortschatz der relationalen Algebra gehören, sondern im Zuge der effizienten Umsetzung von XQuery hinzugefügt wurden.

In einem zweiten Schritt, wollen wir in Kapitel 4 die Struktur der Anfragepläne untersuchen und die Übersetzung unter diesem Aspekt konkretisieren. SQL ist als Standardanfragesprache für RDBMS wohlbekannt und wir werden eine Reihe von Konzepten, in Form von strukturellen Untersuchungen der Anfragepläne und Übersetzungsregeln, vorstellen um Operatoren der relationalen Algebra in äquivalente SQL-Ausdrücke transformieren.

Kapitel 5 wird auf einige Optimierungsmethoden eingehen. Wir behandeln die Bündelung von *Path-Step*-Operatoren und den Umgang mit mehreren

Fragmenten, in Zusammenhang mit dem Elementkonstruktor. Außerdem werden wir noch erläutern, wie man die Duplikateliminierung bei *Path-Step*-Operatoren unter Umständen vermeiden kann.

In Kapitel 6 werden wir die Implementierung des Codegenerators unter dem Gesichtspunkt einiger wichtiger Konzepte genauer veranschaulichen. Dazu gehören die Mustererkennung mit Burg (siehe [4]) und jene Datenstrukturen, welche beim *Lazy Binding* zum Einsatz kommen. Auch den Umgang mit Baumstrukturen für den Aufbau einer Grammatik werden wir kurz besprechen.

In Kapitel 7 werden wir die Performanz der generierten SQL-Anfragen auf einem konkreten System, in unserem Fall DB2, untersuchen. Wir werden unter anderem analysieren, wie gut unsere generierten Anfragen skalieren und wie schnell unser Codegenerator im Vergleich zu einem nativen XML-System ist. In Kapitel 8 werden wir noch einige zukünftige Verbesserungen vorstellen, die dabei helfen könnten die SQL-Anfragen noch performanter zu gestalten.

Kapitel 2

Einführung in die Relationale Algebra

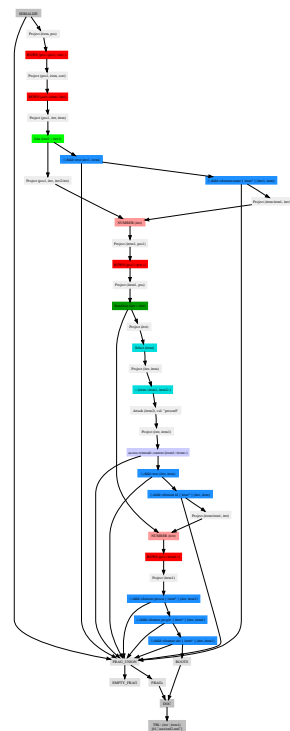
Die relationale Algebra ist *prozedural* orientiert und hat ihre Bedeutung als theoretische Grundlage aller modernen Anfragesprachen in relationalen Datenbanken. Ein relationalalgebraischer Ausdruck impliziert einen Abarbeitungsplan, der bereits Informationen darüber gibt, wie die Anfrage auszuwerten ist. So ist es auch nicht verwunderlich, weshalb die relationale Algebra eine starke Verbreitung bei der Realisierung von Datenbanksystemen findet.

Bei naiver Handhabung kann ein Abarbeitungsplan in bottom-up Manier ausgewertet werden.

Die relationale Algebra operiert auf Mengen, die unter bestimmten schematischen Voraussetzungen miteinander kombinierbar sind.

Definition 2.1. *Schema einer relationalen Algebra.* Sei R eine Relation, wir bezeichnen mit $sch(R)$ das Schema von R . $++$ konkateniert zwei Schemas.

Definition 2.2. *Typkompatibilität.* Die meisten Operatoren der relationalen Algebra stellen Voraussetzungen an die zu verknüpfenden Mengen. Seien Q und R zwei Relationen, so wird $sch(Q) \equiv sch(R)$ als Vereinigungskompati-



bilität bzw. Typkompatibilität bezeichnet.

Diese Voraussetzung erfordert folgende Eigenschaften:

- *R und Q haben den gleichen Grad, also dieselbe Anzahl an Tupelelementen.*
- *Die Tupelnamen von R und Q müssen übereinstimmen.*

2.1 Die Operatoren

Neben den gängigen Operatoren der relationalen Algebra, wie Vereinigung, Projektion, etc. wurde die relationale Algebra hinsichtlich der effizienten Weiterverarbeitung um einige Operatoren erweitert, auf die wir uns im Folgenden beschränken wollen. Wir stellen auch einige der Standardoperatoren vor, wobei wir auf eventuelle Unterschiede zwischen der Standardsemantik und der Semantik in Pathfinder hinweisen werden. In Tabelle 2.1 werden einige der üblichen Operatoren der relationalen Algebra aufgezeigt.

$a b$	Tabelle
$@_{a:v}$	Attach
$\pi_{a_1:b_1, \dots, a_n:b_n}$	Projektion
σ_p	Selektion
$\dot{\cup}$	disjunkte Vereinigung
\times	Kartesisches Produkt
$\bowtie_{a=b}$	Equi-Join
$\rho_{b:\langle a_1, \dots, a_n \rangle / p}$	Tupelnumerierung
$\sqcup_{\alpha, n}$	<i>Path-Step</i> -Operator (Achse α , Knotentest n)
ϵ	Element-Konstruktion

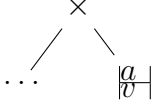
Tabelle 2.1: Operatoren der relationalen Algebra

2.1.1 Tabelle

Der Tabellen-Operator hat keine Eingaberelation und definiert eine Tabelle von Literalen. Sie enthält eine Liste von Tupeln, sowie die Kardinalität der Tabelle. Meistens besteht die Tabelle nur aus einer Spalte mit einem einzigen Tupel.

2.1.2 Attach $@_{a:v}$

Der Operator Attach $@_{a:v}$ hat eine Eingaberelation und erweitert deren Schema um die Spalte a . Alle Tupel enthalten nach der Anwendung dieselbe Konstante v in a . Der Attach-Operator kann auch als

...  geschrieben werden.

2.1.3 Projektion $(\pi_{a_1:b_1, \dots, a_n:b_n})$

Neben der bekannten Funktionalität des Projektions-Operators unterstützt der Operator in *Pathfinder* auch eine Spaltenumbenennung. Dabei wird eine Spalte b_1 nach der Anwendung von $\pi_{a_1:b_1}$ in a_1 umbenannt.

2.1.4 Vereinigung (\cup)

Bei der Vereinigung zweier Ausdrücke $R \cup Q$ werden alle Tupel der Relation R mit allen Tupeln der Relation Q zu einer einzigen Relation vereint. Voraussetzung ist die Typkompatibilität der zu verknüpfenden Operanden R und Q .

$$R \cup Q := \{t \mid t \in R \vee t \in Q\}$$

Die Semantik der Mengenoperation von *Pathfinder* unterscheidet sich jedoch von der gängigen Semantik. Während die mathematisch motivierte Vereinigung stets eine Elimination von Duplikaten fordert, können diese in *Pathfinder*, nach Anwenden einer Vereinigung sehr wohl auftreten. Der Grund hierfür liegt darin, dass eine Duplikatelimination eine kostspielige Operation darstellt und deshalb so oft wie möglich vermieden wird. Die Duplikatelimination wird explizit durch den Distinct-Operator (Abschnitt 2.1.9) eingeleitet.

2.1.5 Differenz

Die Operation $R - Q$ entfernt aus R alle Tupel, die in der zweiten Relation Q vorhanden sind. Voraussetzung ist hier wieder die Typkompatibilität von R und Q

$$R - Q := \{t \mid t \in R \wedge t \notin Q\}$$

Hier stellt bereits die logische Algebra sicher, dass keine Duplikate im Resultat vorhanden sind.

2.1.6 Selektion (σ_p)

Die Selektion $\sigma_p(R)$ wählt als unärer Operator alle Tupel der Relation R , die ein Prädikat p erfüllen.

$$\sigma_p(R) := \{t \mid t \in R \wedge t \in p\}$$

Das Prädikat p kann hier nur *true* oder *false* annehmen, was eine Restriktion bezüglich der Standardalgebra darstellt.

2.1.7 Kartesisches Produkt (\times)

Die Anwendung des kartesischen Produkts $R \times Q$ resultiert in der Menge aller Kombinationen der Tupel aus R und Q . Jedes Tupel der Relation R wird mit jedem Tupel der Relation Q kombiniert. $sch(R)$ und $sch(Q)$ sind disjunkt und so umfasst das Resultat alle Merkmale der Ausgangsrelationen $sch(R \times Q) \equiv sch(R) + sch(Q)$.

2.1.8 Join (\bowtie)

Die relationale Algebra von Pathfinder führt lediglich den Gleichverbund (Equi-Join) und den Semijoin explizit an. Ein Theta-Verbund (Theta-Join) kann implizit durch zahlreiche Konstellationen der Pläne zustande kommen.

Equi-Join ($\bowtie_{a=b}$)

Der Equi-Join Operator verbindet zwei Eingaberelationen basierend auf einer einzigen Äquivalenzrelation. Es wird vorausgesetzt, dass die Elemente der beiden Spalten, welche an der Äquivalenzrelation teilhaben, paarweise verschieden sind.

Semi-Join

Der Semi-Join prüft für jedes Tupel in der linken Eingaberelation, ob es in der rechten Relation mindestens ein korrespondierendes Tupel gibt. Das Schema der Ausgabe entspricht dem der linken Eingaberelation.

2.1.9 Distinct

Die Eliminierung von Duplikation ist auf größeren Relationen sehr kostspielig. In der Regel wird eine Entfernung von identischen Tupeln nicht bei jeder Operation durchgeführt, sondern durch den speziellen *Distinct*-Operator eingeleitet. SQL (Abschnitt 2.1.9) bietet als Äquivalent zum *Distinct*-Operator

bei jeder Anfrage eine optionale `DISTINCT`-Klausel, welche Duplikate eliminiert.

2.1.10 Der generische Operator (`fun_1to1`)

`fun_1to1` ist ein generischer Operator, der Arithmetik und Built-In Funktionen von XQuery elegant zusammenfasst. Er hat eine Eingaberelation und erweitert deren Schema um eine Spalte, welche das Ergebnis der Funktion enthält. In `fun_1to1` werden `+`, `-`, ... sowie die Built-In Funktionen `fn:concat`, `fn:abs`, `fn:contains`, ... vereint. Für die Übersetzung von `fun_1to1` siehe Abschnitt 4.5.7.

2.1.11 Bool'sche Ausdrücke

Die Operatoren `num_eq`, `num_gt`, `num_or`, ... arbeiten auf einer Eingaberelation und erweitern das Schema um eine *bool'sche* Spalte, die das entsprechende Ergebnis enthält. Da in der `SELECT`-Liste eines SQL-Ausdrucks keine bool'schen Ausdrücke angegeben werden können, umgehen wir dieses Problem recht elegant mit einem `CASE`-Statement. Für Erklärung des `CASE`-Statements siehe Abschnitt 3.1.3. Eine Übersetzung am Beispiel des `num_gt`-Operators findet sich in Abschnitt 4.5.8, sowie in 4.6.1.

2.1.12 Rownumber ($(\rho_{b:\langle a_1, \dots, a_n \rangle} / p)$)

Hier wird das Schema der bestehenden Relation durch das Attribut b erweitert. Diese Spalte enthält die Enumerations-Werte der Tupel. Zusätzlich werden die Tupel nach den bestehenden Attributen a_1, \dots, a_n sortiert. Eine weitere Eigenheit ist das Partitionierungsattribut p , welches garantiert, dass für jede Partition die Enumeration der Tupel wieder bei 1 beginnt.

Ähnlich wie der Rownumber-Operator ist auch die Semantik des Number-Operators. Der einzige Unterschied besteht darin, dass keine Sortierung der Tupel stattfindet.

2.1.13 Pfadausdrücke

Die Eleganz eines XML-Dokuments liegt in seiner Baumstruktur. Mittels sogenannten Pfadausdrücken ist es möglich XML-Dokumente zu traversieren. Hierfür gibt es schon seit längerem die vom World Wide Web Consortium spezifizierte Sprache XPath. XQuery verwendet das Vokabular von XPath als Teilsprache.

Das zentrale Konzept von XPath sind die sogenannten Lokalisierungspfade.

Ein Lokalisierungspfad selektiert, ausgehend von einem Referenzknoten, eine Menge von Knoten. In einer Sequenz von Lokalisierungsschritten fungiert jeder Knoten in dieser Menge als neuer Referenzknoten für den nächsten Lokalisierungsschritt. Im letzten Schritt werden alle selektierten Knoten vereinigt und bilden das Ergebnis des gesamten Lokalisierungspfads.

Ein Lokalisierungsschritt besteht aus zwei Teilen

- einer Achse,
- einem Knotentest

Die Achse in einem Lokalisierungsschritt orientiert sich an der Baumstruktur des XML-Dokuments und bestimmt die Richtung innerhalb des Baums.

Path-Step ($\vartriangleleft_{\alpha:n}$)

Die Umsetzung des Achsenkonzepts von XPath wird vom *Path-Step*-Operator verwirklicht. Wir kapseln die Enkodierung des XML-Dokuments und den Zugriff auf XML-Knoten im Operator $\vartriangleleft_{\alpha:n}$. Der *Path-Step*-Operator operiert auf zwei Eingaberelationen. Die erste Relation zeigt auf die aktiven Dokumente, während die zweite Relation die Kontext-Knoten enthält. Der Achsentest α findet also die gewünschte Achse, während mit dem Knotentest n auf den gewünschten Namen und den Knotentyp getestet wird. Dabei sei α eine der in Tabelle 2.2 gezeigten Achsen.

<code>descendant-or-self</code> :	wie <code>descendant</code> , nur dass hierzu auch der Referenzknoten gehört
<code>descendant</code> :	hierunter fallen alle direkten und indirekten Unterelemente, also die Kinder und deren Kinder etc.
<code>ancestor-or-self</code> :	der Referenzknoten selbst oder einer seiner Vorgänger
<code>ancestor</code> :	alle Vorgänger des Referenzknotens fallen unter diese Achse
<code>child</code> :	diese Achse bestimmt alle direkten Unterelemente
<code>following</code> :	alle Knoten, die in der Dokumentreihenfolge nach dem Referenzknoten aufgeführt sind
<code>following-sibling</code> :	alle Knoten, die in der Dokumentreihenfolge nachfolgende Kinder des Vaterknotens von <code>self</code> sind
<code>parent</code> :	der Vaterknoten wird durch diese Achse ermittelt
<code>preceding</code> :	alle Knoten, die in der Dokumentreihenfolge vor dem Referenzknoten vorkommen
<code>preceding-sibling</code> :	alle Knoten der in der Dokumentreihenfolge vorangehenden Kinder des Referenzknotens
<code>self</code> :	der Referenzknoten

Tabelle 2.2: XPath-Achsen

Der Knotentest n kann dabei die Formen in Abbildung 2.3 annehmen. Formal betrachtet führt der *Path-Step*-Operator einen Theta-Join auf den

<code>element()</code>	... ein XML-Tag
<code>text()</code>	... ein Textknoten

Tabelle 2.3: Knotentest

Kontextknoten der Relationen ctx und $frag$ aus. Dabei sind α und n die Join-Prädikate. Der Operator fordert zudem implizit die Eliminierung von gleichen Tupeln.

Kapitel 3

Einführung in die Zielsprache SQL:99

Wie bei der relationalen Algebra werden wir auch hier nicht das Standardvokabular von SQL erläutern, sondern uns auf jene Funktionalitäten beschränken, die charakteristisch für die Übersetzung sind.

Hierzu gehört insbesondere auch die erweiterten OLAP-Funktionalitäten von SQL. Für eine vollständige Einführung in die Fähigkeiten von SQL sei auf [13] verwiesen.

3.1 Die Sprachkonstrukte

3.1.1 Das WITH-Statement und common table expressions

Der SQL-Standard bietet mit `common table expressions` und dem `WITH`-Statement die Möglichkeit äußerst komplexe Anfragen an die Datenbank zu stellen. Eine `common table expression` ist einer `nested table expression` sehr ähnlich, wird aber durch das Schlüsselwort `WITH` eingeleitet. Darauf folgt eine Sequenz von benannten Tabellendefinitionen. Eine solche benannte Tabellendefinition ist innerhalb der gesamten Anfrage sichtbar und kann von den folgenden Anfragen, innerhalb des `WITH`-Statements, gemeinsam benutzt werden. In ihrer Semantik ist eine `common table expression` einer temporären View-Definition ähnlich. Sie ist durch ihren Korrelationsnamen überall dort einsetzbar, wo Relationen benutzt werden dürfen. Jede Referenz auf eine spezifische `common table expression` greift auf dieselbe temporäre View-Definition zurück. Im Gegensatz zu einer View-Definition bewirkt die transiente Eigenschaft einer `common table expression`, dass nach Abarbeitung der komplexen Anfrage die Datenbank denselben Status, wie vor deren Aus-

führung hat.

Der Einsatz von `common table expression` führt zu einem sehr intuitiven Umgang mit der Graphenstruktur unseres Algebraplans. In Kapitel 4 werden wir noch genauer auf diesen Sachverhalt eingehen.

Beispiel 3.1. *Das folgende, etwas künstliche Beispiel selektiert aus einer Relation `employee` in der ersten Anfrage `employee_1` alle Mitarbeiter mit einem Gehalt größer als 50000. In der zweiten Anfrage `employee_2` werden daraus noch alle Mitarbeiter, die älter als 42 sind, gefiltert.*

```

1 WITH
2 employee_1(id, name, age) AS
3 (
4     SELECT id, name, age
5     FROM employee
6     WHERE salary > 50000
7 ),
8 employee_2(id, name) AS
9 (
10    SELECT id, name
11    FROM employee_1
12    WHERE age > 50
13 )
14 SELECT id, name FROM employee_2

```

Die temporären View-Definitionen `employee_1` und `employee_2` in Form von `common table expressions` verlieren nach Ausführung der Anfrage ihre Gültigkeit und sind auch in der Datenbank nicht mehr vorhanden.

3.1.2 Rownumber

Die `ROWNUMBER()`-Funktion ordnet jedem Tupel einer Ergebnisrelation eine aufsteigende Zahl zu. Die Sortierung bestimmt die Reihenfolge, in der den Tupeln die Zahl zugeordnet wird. Die erste Zahl, die von der `ROWNUMBER()`-Funktion vergeben wird, ist die 1, danach wird die Zahl inkrementiert und jeweils dem nächsten Tupel¹ in der Relation zugeordnet. Mit den `PARTITION BY`- und `ORDER BY`-Klauseln, auf die wir noch eingehen werden, können wir das Verhalten von `ROWNUMBER()` weiter beeinflussen.

Bei der Implementierung des SQL-Codegenerators wird diese Funktion bei der Übersetzung des Number- und Rownumber-Operators (Abschnitt 4.5.9) verwendet.

PARTITION BY

Bestimmt die Partition innerhalb der Ergebnisrelation, auf die die Funktion angewendet wird. Für alle Tupel mit demselben Wert in a fängt der Zählprozess wieder bei 1 an und nimmt dann seinen gewohnten Lauf.

¹bestimmt durch die Sortierung

ORDER BY

Definiert die Ordnung der Tupel in einer Partition. Je nach Sortierung zieht die `ROW_NUMBER` Funktion für die Vergabe der Zahl ein anderes Tupel heran. Für die Sortierung können mehrere Spalten herangezogen werden.

Beispiel 3.2. *Wenden wir die `ROWNUMBER`-Funktion nun auf eine Relation v an. Die Anweisung, die wir auf v anwenden, sieht folgendermaßen aus:*

```
SELECT
  ROWNUMBER() OVER (
    PARTITION BY int1
    ORDER BY int2) AS row,
  int1, int2
FROM v
```

Wir partitionieren die Tabelle nach dem Attribut int_1 und sortieren zuerst über int_1 und dann über int_2 .

(a) Relation v	(b) v nach der Anwendung des <code>ROWNUMBER</code> -Operators			
int ₁	int ₂	row	int ₁	int ₂
1	2	1	1	2
2	5	2	1	3
1	3	1	2	1
2	4	2	2	4
2	1	3	2	5

Tabelle 3.1: Anwendung des `ROWNUMBER`-Operators

3.1.3 If-then-else Logik in SQL

Die `CASE`-Anweisung in SQL bietet die Möglichkeit Bedingungen auszuwerten. So evaluiert es eine Liste von Bedingungen und gibt eine von mehreren möglichen Ergebnissen aus. `CASE` gibt es in zwei Formaten, wobei wir bei der Generierung von SQL nur Folgendes benötigen.

```

CASE
  WHEN <ausdruck_1> THEN <ergebnis_1>
  WHEN <ausdruck_2> THEN <ergebnis_2>
  ...
  WHEN <ausdruck_n> THEN <ergebnis_n>
  ELSE <ergebnis_else>
END

```

Abbildung 3.1: Die Syntax einer CASE-Anweisung

Beispiel 3.3. *Es sei v die Relation, auf der wir unsere Beispiel-Anweisung anwenden. Die Anweisung, die auf v angewendet wurde, sieht wie folgt aus:*

```

SELECT
  CASE
    WHEN int1 < int2 THEN 1
    ELSE 0
  END AS case,
  int1, int2
FROM v

```

(a) Relation v	
int ₁	int ₂
1	2
2	1
3	3
2	5

(b) Relation v nach der Anwendung von CASE		
case	int ₁	int ₂
1	1	2
0	2	1
0	3	3
1	2	5

Tabelle 3.2: Anwendung der CASE-Anweisung

3.1.4 Duplikateliminierung

Als Teil der **SELECT**-Klausel leitet das Schlüsselwort **DISTINCT** eine Duplikateliminierung ein. Das bedeutet konkret, dass alle ausgewählten Tupel paarweise verschieden sind. Diese Operation ist auf großen Relationen sehr aufwendig und wird so oft wie möglich vermieden (siehe dazu auch Abschnitt 5.1).

Beispiel 3.4. *Es sei v die Relation, auf der wir unsere Beispiel-Anweisung anwenden. Die Anweisung, die auf v angewendet wurde, sieht wie folgt aus:*

```
SELECT DISTINCT int1, int2
FROM v
```

(a) Relation v		(b) Relation v nach der Anwendung von DISTINCT	
int ₁	int ₂	int ₁	int ₂
1	2	1	2
2	1	2	1
1	2		
2	1		

Tabelle 3.3: Anwendung der CASE-Anweisung

Kapitel 4

Übersetzung

Bei näherer Betrachtung eines Ausführungsplans stellt man fest, dass seine Struktur nicht mehr dem eines gewöhnlichen Syntaxbaums entspricht, sondern dem eines gerichteten azyklischen Graphen (GAG). Diese Graphenstruktur resultiert aus der Kompilation der Algebra, in der sich mehrere Operatoren ihre Kinder teilen. Die intuitive Handhabung solcher Pläne führt direkt zu einer Graphenstruktur, in der Teilanfragen innerhalb der Übersetzung immer wieder benutzt werden können, siehe auch Abschnitt 3.1.1. Eine weitere Besonderheit ist die hochgradig kompositionale Struktur der Pläne. Im Allgemeinen setzt also ein relationenalgebraischer Operator voraus, dass seine Nachfolger bereits ausgewertet wurden. Diese strukturelle Eigenschaft kommt formal einer topologischen Sortierung gleich. Dies impliziert, dass ein Knoten existiert, der den Eingangsgrad 0 besitzt.

Definition 4.1. *Eingangsgrad.* Sei $G(V, E)$ ein gerichteter Graph und $v \in G(V, E)$ ein Knoten, so bezeichnen wir $d_G^-(v)$ als den Eingangsgrad von v in G .

Definition 4.2. Sei $G(V, E)$ ein gerichteter azyklischer Graph. G sei zusammenhängend bezüglich eines Knotens $r \in V$, das heißt $\exists_1 r \in G(V, E)$ mit $d_G^-(r) = 0$. Wir bezeichnen r im Weiteren als Wurzel.

4.1 Die Struktur des Abarbeitungsplans

Ein naiver Ansatz wäre es eine post-order Traversierung, ausgehend von der Wurzel durchzuführen, jeden Operator auszuwerten und anschließend zu materialisieren. Jeder Knoten würde also in Form einer Tabelle in der Datenbank existieren. Diese Strategie steht im Einklang mit der kompositionalen Struktur des Plans, hat allerdings den Nachteil, dass durch die Materialisierung

ein nicht zu unterschätzender Overhead durch das Anlegen der Tabellen bei jedem Operator anfallen würde. Zudem hinterließe diese Methode Spuren in der Datenbank, was der Semantik einer Anfrage widerspräche. Als Semantik einer Anfrage wird in diesem Kontext die Tatsache bezeichnet, dass eine Anfrage nur lesenden Zugriff auf eine Datenmenge haben sollte. Das Anlegen von Tabellen im Datenbanksystem hätte zur Folge, dass diese in einer Transaktion am Ende der Anfrage wieder aus dem System entfernt werden müssten. Die Einbettung der Anfrage in eine Transaktion würde seitens des Datenbanksystems weitere Logging-Maßnahmen nach sich ziehen, die eine weitere Verlangsamung zur Folge hätten. Zudem entsteht durch die kompositionale Struktur sehr viel unnötiger Kopieraufwand.

Beispiel 4.1. *Wir wollen nun versuchen, unter dieser Methode den Plan in Abbildung 4.1 zu übersetzen.*

Listing 4.1: Übersetzung mittels Materialisierung der einzelnen Operatoren

```

1 — Übersetzung des Tabellen-Operators
2 CREATE TABLE a0000 (pos_nat INTEGER NOT NULL);
3 INSERT INTO TABLE a0000(pos_nat)
4     SELECT 1 AS pos_nat from sysibm.sysdummy1;
5 — Übersetzung des Attach-Operators
6 CREATE TABLE a0001 (pos_nat INTEGER NOT NULL,
7     item_int INTEGER NOT NULL);
8 INSERT INTO TABLE a0001(pos_nat, item_int)
9     SELECT pos_nat, 1 as item_int from a0000;
10 — Auslesen der gesamten Relation
11 SELECT pos_nat, item_int FROM a0001 ORDER BY pos_nat;
```

Wir sehen schnell, dass eine ganze Reihe von CREATE TABLE- und INSERT-Befehlen notwendig ist, um die Relationen für die einzelnen Operationen physisch verfügbar zu machen. Diese Vorgehensweise führt bei größeren Plänen zu einer unüberschaubaren und ineffizienten Übersetzung.

Im Folgenden wollen wir nun Schritt für Schritt diesen Ansatz verbessern und eine befriedigende Lösung vorstellen.

4.2 Bindung mit dem WITH-Statement

Die Materialisierung hinterlässt, wie bereits verdeutlicht, Spuren in der Datenbank. Das WITH-Statement bietet uns die Möglichkeit eine Sequenz von SQL-Anweisungen auszuführen, ohne sie in physischen Tabellen materialisieren zu müssen (für eine kurze Einführung in das WITH-Statement, sei auf Kapitel 3.1.1 verwiesen). Hierbei wird eine Anweisung durch eine Referenz identifiziert, deren Sichtbarkeitsbereich sich über die folgende Sequenz, innerhalb des WITH-Statements, erstreckt. Dies kommt unserer verschachtelten Struktur zu Gute, da wir nun jeden Knoten explizit übersetzen und in den

Vorgänger-Knoten verwenden können. Durch dieses Vorgehen binden wir al-

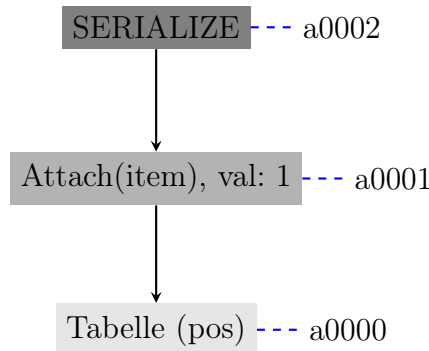


Abbildung 4.1: Jeder Operator wird an eine Referenz gebunden.

so jeden relationalen Operator an eine Referenz, die wir wiederverwenden können. Abbildung 4.1 skizziert diesen Vorgang. Diese Bindung muss aber nicht explizit für jeden Knoten vorgenommen werden.

Beispiel 4.2. *Durch das WITH-Statement erhalten wir die Möglichkeit, unsere Pläne ohne den Einsatz von CREATE TABLE- und INSERT-Statements zu übersetzen. Dadurch erreichen wir, dass die Datenbank invariant bezüglich ihres Inhalts bleibt, also keine neuen Relationen hinzugefügt werden.*

Listing 4.2: Einsatz des WITH-Statements

```

1 WITH
2 — Uebersetzung des LIT_TBL-Operators
3 a0000(item_int) AS (
4   SELECT
5     1 AS item_int
6   FROM
7     sysibm.sysdummy1 AS c0000),
8 — Uebersetzung des ATTACH-Operators
9 a0001(item_int, pos_nat) AS (
10  SELECT
11    c0000.item_int,
12    1 AS pos_nat
13  FROM
14    a0000 AS c0001)
15 — Auslesen der gesamten Relation
16 select item_int, pos_nat from a0001 ORDER BY pos_nat;
  
```

4.3 Lazy Binding

Die Bindung eines Operator-Knotens an eine Referenz bewirkt, dass die damit verbundene SQL-Anweisung ausgeführt und deren Ergebnis unter diesem eindeutigen Namen abrufbar wird. Aber nicht jeder Knoten muss gebunden

werden. Vielmehr können einige Knoten, unter gewissen Aspekten, zu einer komplexeren SQL-Anweisung zusammengefasst werden. Eine Bindung an eine Tabellenreferenz soll also nur herbeigeführt werden, wenn dies erforderlich ist.

Um diesen Vorgang anhand eines Beispiels zu motivieren betrachten wir den Plan in Abbildung 4.3, der aus der Übersetzung von $2 * 3$ stammt.

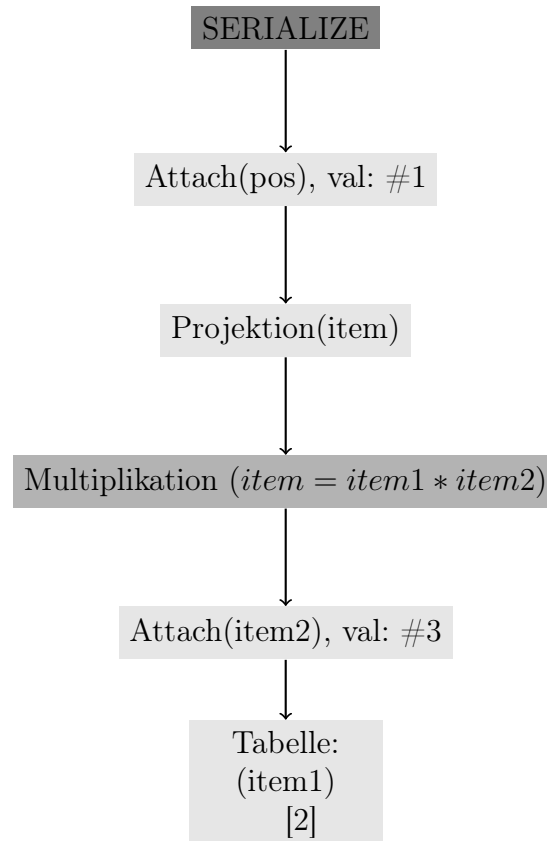


Abbildung 4.2: Relationaler Plan

Sehen wir uns zunächst einmal an, wie dieser Plan ohne *Lazy Binding* übersetzt werden würde. Wir traversieren den Baum in Bottom-Up-Manier. Die Reihenfolge, in der die Operatoren abgelaufen werden, ist folgende:

1. Tabelle: (item1)
2. Attach(item2), val: #1
3. Multiplikation ($item = item1 * item2$)
4. Projektion (item)

5. Attach (pos), val: #1

Wir erhalten ohne Lazy-Binding also folgenden Ausdruck:

Listing 4.3: Übersetzung ohne Lazy-Binding

```

1 WITH
2 — Uebersetzung des Tabellen-Operators
3 a0000(item1_int) AS (
4   SELECT
5     2 AS item1_int
6   FROM
7     sysibm.sysdummy1 AS c0000),
8 — Uebersetzung des ATTACH-Operators
9 a0001(item1_int, item2_int) AS (
10  SELECT
11    c0001.item1_int,
12    3 AS pos_nat
13  FROM
14    a0000 AS c0001),
15 — Uebersetzung der Multiplikation
16 a0002(item1_int, item2_int, item_int) AS (
17  SELECT
18    c0002.item1_int,
19    c0002.item2_int,
20    (c0002.item1_int * c0002.item2_int) AS
21    item_int
22  FROM a0001 AS c0002),
23 — Uebersetzung der Projektion
24 a0003(item_int) AS (
25  SELECT
26    c0003.item_int
27  FROM a0002 AS c0003),
28 — Uebersetzung von Attach
29 a0004(item_int, pos_nat) AS (
30  SELECT
31    c0004.item_int,
32    1 AS pos_nat
33  FROM a0003 AS c0004)
34 — Auslesen der gesamten Relation
35 select item_int, pos_nat from a0004 ORDER BY pos_nat;

```

Wir sehen an diesem Beispiel, dass jeder Operator einzeln übersetzt und gleich mittels `common table expression` an einen eindeutigen Namen gebunden wird, auf den wir in der folgenden Übersetzungssequenz referenzieren. Die Übersetzung jedes einzelnen Operators erscheint uns nicht intuitiv, da der gesamte Ausdruck in einem einzigen Statement zusammengefasst werden könnte. Außerdem werden die Ausdrücke sehr schnell unhandlich und wachsen zu enormer Größe heran.

Listing 4.4: Übersetzung mit Lazy-Binding

```

1 WITH
2 — Uebersetzung aller Operatoren
3 a0000(pos_nat, item_int) AS (
4   SELECT 1 AS pos_nat,
5          (2 * 3) AS item_int
6   FROM sysibm.sysdummy1 AS c0000),
7 — Auslesen der gesamten Relation

```

```
8 SELECT pos_nat, item_int FROM a0000 ORDER BY pos_nat;
```

Anhand von Listing 4.4 sehen wir, wie kompakt und elegant der Plan in Abbildung 4.3 zusammengefasst werden kann. Tatsächlich werden die meisten Zwischenergebnisse durch die Projektion auf *item* aus dem Plan entfernt, sodass am Ende lediglich *pos_nat* und *item_int* im Ergebnis vorhanden sind. Dieser Plan ist nicht nur viel intuitiver, sondern schont auch die Ressourcen des Datenbanksystems.

Zur Umsetzung des *Lazy-Bindings* generieren wir nicht sofort gültige SQL-Ausdrücke, sondern sammeln Ausdrücke der *select-list* in einem Container, der die Spaltennamen auf gültige Ausdrücke abbildet. Abbildung 4.3 zeigt

$$\{a_1 \rightarrow e_1, \dots, a_n \rightarrow e_n\}$$

Abbildung 4.3: Datenstruktur zum Aufsammeln von Ausdrücken

eine Menge, in der Abbildungen abgelegt werden können. a_i entspricht dabei einem Spaltennamen, während e_i einem Ausdruck der *select-list* (siehe [13]) entspricht.

Nach der Übersetzung wird geprüft, ob ein Ausdruck gebunden werden muss, oder ob die gesammelten Informationen für eine spätere Bindung weiter propagiert werden können.

Hinreichende Kriterien für die sofortige Durchführung einer Bindung sind ein Eingangsgrad $d_G^-(v) > 1$ oder, wenn der Gesamtausdruck keinem *SELECT FROM WHERE*-Statement entspricht. Die Bindung einiger Operatoren der relationalen Algebra nicht aufgeschoben werden, weshalb diese markiert werden können, um eine Bindung zu forcieren.

Beispiel 4.3. *In diesem Beispiel wollen wir uns ansehen, wie sich die eben definierte Abbildungsmenge bei der Generierung des Plans in Abbildung 4.3 verhält.*

1. *Tabelle (item1)*
 $\{item1_int \rightarrow 2\}$
2. *Attach (item1), val: #3*
 $\{item1_int \rightarrow 2, item2_int \rightarrow 3\}$
3. *Multiplikation (item = item1 * item2)*
 $\{item1_int \rightarrow 1, item2_int \rightarrow 3, item_int \rightarrow (2 * 3)\}$
4. *Project(item)*
 $\{item_int \rightarrow (2 * 3)\}$

5. *Attach(pos), val: #1*
 $\{item_int \rightarrow (2 * 3), pos_nat \rightarrow 1\}$

Hier sehen wir konkret, wie sich die Abbildungsmenge bei der Übersetzung verhält. Am Ende sind nur *pos_nat* und *item_int* in der Menge vorhanden. Die restlichen Zwischenergebnisse werden durch die Projektion aus der Liste entfernt.

4.3.1 Auswirkungen der GAG-Struktur auf die Bindung

Wir haben bereits angesprochen, dass unser Abarbeitungsplan die Form eines gerichteten azyklischen Graphens besitzt. Ein Knoten $v \in G(V, E)$ hat also im Allgemeinen einen Eingangsgrad $d_G^-(v) \geq 1$. Das bedeutet für unsere Übersetzung, dass wir den doppelten Aufwand verzeichnen, wenn wir den Knoten lediglich zu seinem Vorgänger propagieren und die Bindung später durchführen. Ein *Lazy Binding* ist in diesem Fall also nicht sinnvoll.

Regel 1. Für alle Knoten $v \in G(V, E)$ für die gilt, dass $d_G^-(v) > 1$, führe eine sofortige Bindung durch.

Der Gedanke hinter diesem Vorgehen ist, dass SQL-Code in seiner Optimierung sehr stark davon abhängig ist wie er eingegeben wird. Das bedeutet, dass sich die Optimierung indirekt durch den generierten Code beeinflussen lässt.

Ein Eingangsgrad $d_G^-(v) > 1$ bedeutet, dass ein Operator von mehreren seiner Vorgänger benötigt wird. Die Forcierung einer Bindung in einer solchen Situation, gibt dem Optimierer Hinweise auf die GAG-Struktur in den Ausführungsplänen. Diese zusätzliche Information kann dazu genutzt werden, um effizientere Pläne zu erzeugen.

4.4 Die Document- und Result-Relation

Um XML in einer RDBMS zu speichern, müssen die Daten in eine wohldefinierte Ordnung gebracht werden, um die Baumstruktur auf eine flache Relation abbilden zu können. In [11] wurde ein Indexschema, basierend auf `pre/size/level` Werten, entwickelt. Während ein XML-Dokument geparkt wird, werden den Elementen die zugehörigen Werte zugewiesen. Abbildung 4.5(a) zeigt ein XML-Fragment, in Abbildung 4.4(b) wird dessen zugehörige relationale Enkodierung illustriert. Die Markierungen `pre`, `size` und `level` entsprechen dabei folgenden Definitionen:

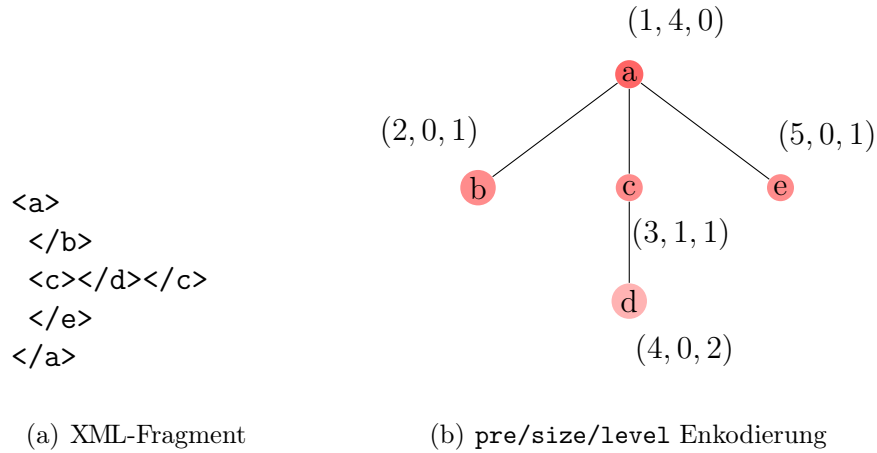


Abbildung 4.4: Relationale Enkodierung

Definition 4.3. Es sei $T(V, E)$ ein Baum, so ist $level_T(v)$ eines Knotens $v \in T(V, E)$ die Anzahl Vorgänger ohne v von v in T .

Definition 4.4. Es sei $T(V, E)$ ein Baum, so ist $size_T(v)$ eines Knotens $v \in T(V, E)$ die Anzahl der Nachfolger ohne v von v in T .

Definition 4.5. Es sei $T(V, E)$ ein Baum, so ist $pre_T(v)$ eines Knotens $v \in T(V, E)$ der Index, der v bei einer pre-order Traversierung von T zugewiesen wird.

Die Attribute *pre*, *size* und *level* speichern strukturelle Informationen über die Gestalt des Baumes. Das *level*-Attribut erlaubt zudem eine effiziente Bestimmung der Position von Vater-, Geschwister- und Kind-Knoten. Diese Art der Enkodierung wird bei der Implementierung des Codegenerators verwendet. In [6] wird eine weitere mögliche Enkodierung auf Basis von *pre*- und *post*-Werten vorgeschlagen. Die Semantik eines Knotens, wie dessen Tagname, Text oder Zahl wird im *prop*-Attribut abgelegt. *prop* ist ein reiner Textknoten, was bedeutet, dass ohne Rücksicht auf den Datentyp der Wert eines Knotens immer in einer Stringrepräsentation abgelegt wird. Dies führt zu einer kompakten, wenn auch etwas unsaubereren Lösung, die aber durchaus konform mit der Handhabung von Textknoten in XML/XQuery ist.

	pre	size	level	kind	prop
<beispiel>	0	8	0	6	"beispiel.xml"
<zahlen>	1	7	1	1	"beispiel"
<intzahl>42</intzahl>	2	4	2	1	"zahlen"
<floatzahl>3.14	3	1	3	1	"intzahl"
</floatzahl>	4	0	4	3	"42"
</zahlen>	5	1	3	1	"floatzahl"
<text>Beispieltext</text>	6	0	4	3	"3.14"
</beispiel>	7	1	2	1	"text"
	8	0	3	3	"Beispieltext"

(a) Beispiel XML-Fragment

(b) Document-Enkodierung

Abbildung 4.5: Relationale Enkodierung von XML-Dokumenten

4.5 Die Inferenzregeln

Nachdem wir die Enkodierung der XML-Dokumente in Abschnitt 4.4 eingeführt haben, wenden wir uns den konkreten Regeln zur Übersetzung zu. Die Regeln folgen dabei der Notation

$$G, \mathcal{V}, \mathcal{M} \vdash v \Leftrightarrow (\mathcal{S}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \Delta)$$

- G sei der Ausführungsplan der relationalen Algebra in Form eines Graphen.
- \mathcal{V} sei jene Menge, die die Namen der bereits definierten `common table expressions` beinhaltet.
- \mathcal{M} sei die Menge, welche den konkreten Ausdruck enthält, der zur Bindung eines SQL-Statements an eine `common table expression` nötig ist. Außerdem ist \mathcal{M} genau jene Menge, die die Ausgabe enthält, welche am Ende der Übersetzung die Anfrage enthält.

Der Compilationsprozess wird mit der *top-level-expression* G gestartet. Die Mengen \mathcal{V} und \mathcal{M} sind am Anfang noch leer, das heißt $\mathcal{V} = \emptyset$ und $\mathcal{M} = \emptyset$. Auf den Knoten $v \in G$ wird auf zweierlei Weise referenziert. Einerseits wird v zur Bestimmung des Eingangsgrads als Knoten behandelt, andererseits referenzieren wir damit auf den in v annotierten algebraischen Ausdruck. Wir verzichten hier auf weitere Formalismen um die Regeln einfach zu halten. Das *Lazy Binding* führt dazu, dass wir nicht sofort SQL-Statements generieren können. Um die Ausdrücke zu anderen Operatoren zu propagieren verwenden

wir die Mengen \mathcal{S} , \mathcal{F} und den bool'schen Ausdruck $w \in \mathcal{W}$ aus der Menge aller möglichen bool'schen Ausdrücke. w besteht nur aus Konjunktionen.

In \mathcal{S} werden Ausdrücke gesammelt, die später für die Erstellung der *select-list* des `SELECT FROM WHERE`-Statements erforderlich sind. Dabei werden in \mathcal{S} Ausdrücke der Form

$$\text{Attribut} \rightarrow \text{Ausdruck}$$

abgelegt. Damit ist es möglich Ausdrücke durch ihren Attributnamen zu finden. Die Menge \mathcal{F} speichert indessen alle Tabellen, beziehungsweise Relationen, auf die ein SQL-Ausdruck referenziert. Bei der Generierung des konkreten SQL-Codes landet der Inhalt von \mathcal{F} im `FROM`-Teil. Der bool'sche Ausdruck w wird schließlich für die Generierung des *where*-Teils benötigt.

Im Zuge einer Übersetzung können neue *common table expressions* entstehen. Diese werden in $\hat{\mathcal{M}} = \mathcal{M} \dot{\cup} \mathcal{M}_{new}$ abgelegt. Der an die *common table expression* gebundene Name wird in $\hat{\mathcal{V}} = \mathcal{V} \dot{\cup} \mathcal{V}_{new}$ aufbewahrt.

Einige Operatoren der relationalen Algebra besitzen auch die Fähigkeit neue Fragmente zu kreieren. Dabei wird der SQL-Code zum Generieren des neuen Fragments, von elementerzeugenden Operatoren, in Δ abgelegt.

Die Angabe einer Inferenzregel erfolgt durch die in Abbildung 4.6 dargestellte Notation. Über der horizontalen Linie werden alle zur Übersetzung

$$\frac{G, \mathcal{V}, \mathcal{M} \vdash v_1 \Leftrightarrow (\mathcal{S}_1, \mathcal{F}_1, w_1, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \hat{\Delta}) \quad G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash v_2 \Leftrightarrow (\mathcal{S}_2, \mathcal{F}_2, w_2, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \hat{\Delta})}{G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash \begin{array}{c} v_3 \\ / \quad \backslash \\ v_1 \quad v_2 \end{array} \Leftrightarrow (\mathcal{S}_3, \mathcal{F}_3, w_3, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \hat{\Delta})}$$

Abbildung 4.6: Notation zur Darstellung einer Inferenzregel

notwendigen Voraussetzungen angegeben. Unter der Linie steht die eigentli-

che Übersetzung des Knotens v_3 . Die baumähnliche Darstellung $\begin{array}{c} v_3 \\ / \quad \backslash \\ v_1 \quad v_2 \end{array}$

drückt die Tatsache aus, dass die Operation v_3 zwei Operanden v_1 und v_2 benötigt, welche im Anfrageplan als Kinder dargestellt werden. Die spezielle Bedeutung von \Leftrightarrow werden wir erst in Abschnitt 4.6 genauer betrachten. Fürs Erste nehmen wir an, dass \Leftrightarrow für den SQL-Ausdruck immer eine Bindung vornimmt. Wir verwenden zudem die Symbole $_$ und \perp . $_$ steht für einen

beliebigen Ausdruck, während \perp entweder ein leeres Fragment oder einen boolschen Ausdruck kennzeichnet, der dem bool'schen Wert *true* entspricht.

4.5.1 Tabelle

$$\begin{array}{c}
 \hat{\mathcal{M}} \equiv \left\{ \begin{array}{l}
 \mathfrak{t}(a, b) \text{ AS} \\
 ((\text{SELECT} \\
 \quad a_1 \text{ AS } a, b_1 \text{ AS } b \\
 \text{FROM} \\
 \quad \text{sysibm.sysdummy1}) \\
 \text{UNION} \\
 (\text{SELECT} \\
 \quad \dots \text{ AS } a, \dots \text{ AS } b \\
 \text{FROM} \\
 \quad \text{sysibm.sysdummy1}) \\
 \text{UNION} \\
 (\text{SELECT} \\
 \quad a_n \text{ AS } a, b_n \text{ AS } b \\
 \text{FROM} \\
 \quad \text{sysibm.sysdummy1}))
 \end{array} \right.
 \end{array}$$

$$G, \mathcal{V}, \mathcal{M} \vdash \begin{array}{c|c}
 a & b \\
 \hline
 a_1 & b_1 \\
 \dots & \dots \\
 a_n & b_n
 \end{array} \Rightarrow \left(\{a \rightarrow a, b \rightarrow b\}, \{\mathfrak{t}\}, \perp, \mathcal{V} \dot{\cup} \{\mathfrak{t}\}, \mathcal{M} \dot{\cup} \hat{\mathcal{M}}, \perp \right)$$

Der Tabellen-Operator (`lit_tbl`) hat keine Eingaberelation und definiert, wie bereits in Abschnitt 2.1.1 erläutert, eine Tabelle von Literalen. Da `lit_tbl` keine Eingaberelation hat stehen oberhalb der horizontalen Linie keine Voraussetzung.

Die Tabelle wird einfach durch SQL-Befehle nachempfunden. Bei mehreren Tupeln kommt es dabei zu sehr unhandlichen Ausdrücken. In diesem Beispiel kommt es, aufgrund mehrere Tupel in der Tabelle, zu einer Bindung, da der Ausdruck aus mehreren `UNION`-Befehlen zusammengesetzt ist.

Besteht die Tabelle aus einem Tupel, muss keine Bindung stattfinden und die Ergebnisse der Übersetzung müssen an den nächsten Operator propagiert werden.

4.5.2 Attach

$$\frac{G, \mathcal{V}, \mathcal{M} \vdash v \Rightarrow (\mathcal{S}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, _)}{G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash \underset{v}{\underset{\uparrow}{\textcircled{a:c}}} \Rightarrow (\mathcal{S} \dot{\cup} \{a \rightarrow c\}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \perp)}$$

Der Attach-Operator $\textcircled{a:c}$ erweitert das Schema der Relation v um ein Attribut a . Diese neue Spalte wird mit der Konstante c gefüllt; alle anderen Attribute von v werden in die neue Relation übernommen.

4.5.3 Vereinigung

$$\frac{\begin{array}{l} G, \mathcal{V}, \mathcal{M} \vdash v_1 \Rightarrow (\{a_1 \rightarrow e_1, \dots, a_n \rightarrow e_n\}, \mathcal{F}_1, w_1, \hat{\mathcal{V}}, \hat{\mathcal{M}}, _) \\ G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash v_2 \Rightarrow (\{b_1 \rightarrow d_1, \dots, b_n \rightarrow d_n\}, \mathcal{F}_2, w_2, \hat{\mathcal{V}}, \hat{\mathcal{M}}, _) \end{array}}{G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash \underset{v_1}{\swarrow} \cup \underset{v_2}{\searrow} \Rightarrow \left(\{a_1 \rightarrow a_1, \dots, a_n \rightarrow a_n\}, \{\mathbf{t}\}, \perp, \hat{\mathcal{V}} \dot{\cup} \{\mathbf{t}\}, \hat{\mathcal{M}} \dot{\cup} \hat{\hat{\mathcal{M}}}, \perp \right)}$$

$$\hat{\hat{\mathcal{M}}} \equiv \left\{ \begin{array}{l} \mathbf{t}(a_1, \dots, a_n) \text{ AS} \\ ((\text{SELECT } e_1 \text{ AS } a_1, \\ \dots, \\ e_n \text{ AS } a_n \\ \text{FROM } \mathcal{F}_1 \\ \text{WHERE } w_1) \\ \text{UNION ALL} \\ (\text{SELECT } d_1 \text{ AS } b_1, \\ \dots, \\ d_n \text{ AS } b_n \\ \text{FROM } \mathcal{F}_2 \\ \text{WHERE } w_2)) \end{array} \right\}$$

Der Vereinigungsoperator in Pathfinder verhält sich wie UNION ALL in SQL, das heißt Duplikate werden nicht eliminiert. Die einzige Voraussetzung für die Verknüpfung zweier Relationen ist ihre Typkompatibilität (siehe Definition 2.2).

Analog hierzu wird die Differenz zweier Mengen mit EXCEPT ALL übersetzt.

4.5.4 Selektion

$$\frac{\mathcal{S} \equiv \{\dots, a \rightarrow p, \dots\} \quad G, \mathcal{V}, \mathcal{M} \vdash v \Rightarrow (\mathcal{S}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, _)}{G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash \begin{array}{c} \sigma_a \\ | \\ v \end{array} \Rightarrow (\mathcal{S}, \mathcal{F}, (w) \wedge (p), \hat{\mathcal{V}}, \hat{\mathcal{M}}, \perp)}$$

Der σ -Operator hat die Aufgabe alle Zeilen aus einer Relation zu selektieren, welche ein Prädikat p erfüllen. Dieses Prädikat muss bereits zu einem früheren Zeitpunkt in die Menge \mathcal{S} unter dem Namen a eingetragen worden sein. Der σ_a -Operator muss nun eine Konjunktion zwischen dem existierenden Prädikaten w und b herstellen, um die gewünschte neue Eigenschaft der Tupel zu garantieren.

4.5.5 Projektion

$$\frac{G, \mathcal{V}, \mathcal{M} \vdash v \Rightarrow (\{\dots, b_1 \rightarrow c_1, \dots, b_n \rightarrow c_n, \dots\}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, _)}{G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash \begin{array}{c} \pi_{a_1:b_1, \dots, a_n:b_n} \\ | \\ v \end{array} \Rightarrow (\{a_1 \rightarrow c_1, \dots, a_n \rightarrow c_n\}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \perp)}$$

Die Aufgabe einer Projektion beschränkt sich darauf, eine Teilmenge von Ausdrücken $\hat{\mathcal{S}} \subseteq \{b_1 \rightarrow c_1, \dots, b_n \rightarrow c_n\}$ in den neuen Listencontainer aufzunehmen und den Ausdrücken c_i gegebenenfalls neue Namen a_i zuzuweisen. Die Projektion kann als eine Art Attributbeschränkung verstanden werden – eine Selektion auf Spaltenebene.

Interessant ist hier die Tatsache, dass kein Code für eine Projektion erzeugt wird. Die Umbenennung, sowie die eigentliche Projektion findet lediglich zur Zeit der Kompilierung statt und hat somit keine Auswirkungen auf die Performanz des produzierten Codes.

4.5.6 Equi-Join

$$\frac{G, \mathcal{V}, \mathcal{M} \vdash \Rightarrow (\{a_1 \rightarrow e_1, \dots, a_n \rightarrow e_n\}, \mathcal{F}, w_1, \hat{\mathcal{V}}, \hat{\mathcal{M}}, _)}{G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash v_2 \Rightarrow (\{b_1 \rightarrow d_1, \dots, b_n \rightarrow d_n\}, \mathcal{F}_2, w_2, \hat{\mathcal{V}}, \hat{\mathcal{M}}, _)}{\hat{\mathcal{S}} \equiv \{a_1 \rightarrow e_1, \dots, a_n \rightarrow e_n, b_1 \rightarrow d_1, \dots, b_n \rightarrow d_n\}}{G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash \begin{array}{c} \bowtie_{a_i=b_j} \\ / \quad \backslash \\ v_1 \quad v_2 \end{array} \Rightarrow (\hat{\mathcal{S}}, \mathcal{F}_1 \dot{\cup} \mathcal{F}_2, w_1 \wedge w_2 \wedge (e_i = d_j), \hat{\mathcal{V}}, \hat{\mathcal{M}}, \perp)}$$

Wir zeigen die Übersetzung von Join-Operatoren am Beispiel des Equi-Joins. Der Join stellt eine triviale Aufgabe in SQL dar. In Pathfinder gilt, dass

$a_i \neq b_j$ für alle $i, j \in \{1, \dots, n\}$. Die bool'schen Ausdrücke können durch eine Konjunktion verknüpft werden. Hinzu kommt das Join-Prädikat $e_i = d_j$ mit $i, j \in \{1, \dots, n\}$

4.5.7 Der generische Operator (fun_1to1)

$$\begin{array}{c} \mathcal{S} \equiv \{\dots, a_1 \rightarrow e_1, a_2 \rightarrow e_2, \dots\} \quad G, \mathcal{V}, \mathcal{M} \vdash v \Rightarrow (\mathcal{S}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, _) \\ \hline G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash \begin{array}{c} r = f(a_1, \dots, a_n) \\ | \\ v \end{array} \Rightarrow (\mathcal{S} \dot{\cup} \{r \rightarrow \hat{f}(e_1, \dots, e_n)\}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \perp) \end{array}$$

Die Funktion f ist hier ein generischer Operator für eine konkrete Operation wie $+$, $-$, etc. Er nimmt mehrere Argumente a_1 bis a_n entgegen und verknüpft, die damit verbundenen Ausdrücke, auf die gewünschte Weise. Dabei wird das Schema der Relation um ein neues Attribut r erweitert, welches das Ergebnis der Operation enthält. Built-In Funktionen, wie `fn:contains` oder `fn:abs` fordern nur ein Argument. Die Funktion f wird aufgrund ihrer generischen Eigenschaft auf ihr SQL-Äquivalent \hat{f} abgebildet.

Beispiel 4.4. *Wir zeigen die Anwendung von $+(a_1, a_2)$ auf eine existierende Relation v . Damit die Operation erfolgreich ist, muss v bereits die geforderten Argumente a_1 und a_2 beinhalten.*

$v \equiv$	<table style="border-collapse: collapse; text-align: center;"> <tr><td style="border-right: 1px solid black; padding: 2px 5px;">a_1</td><td style="padding: 2px 5px;">a_2</td></tr> <tr><td style="border-right: 1px solid black; padding: 2px 5px;">1</td><td style="padding: 2px 5px;">23</td></tr> <tr><td style="border-right: 1px solid black; padding: 2px 5px;">2</td><td style="padding: 2px 5px;">34</td></tr> <tr><td style="border-right: 1px solid black; padding: 2px 5px;">3</td><td style="padding: 2px 5px;">42</td></tr> </table>	a_1	a_2	1	23	2	34	3	42	$f(a_1, a_2)$	\equiv	<table style="border-collapse: collapse; text-align: center;"> <tr><td style="border-right: 1px solid black; padding: 2px 5px;">$r = f(a_1, a_2)$</td><td style="border-right: 1px solid black; padding: 2px 5px;">a_1</td><td style="padding: 2px 5px;">a_2</td></tr> <tr><td style="border-right: 1px solid black; padding: 2px 5px;">24</td><td style="border-right: 1px solid black; padding: 2px 5px;">1</td><td style="padding: 2px 5px;">23</td></tr> <tr><td style="border-right: 1px solid black; padding: 2px 5px;">36</td><td style="border-right: 1px solid black; padding: 2px 5px;">2</td><td style="padding: 2px 5px;">34</td></tr> <tr><td style="border-right: 1px solid black; padding: 2px 5px;">45</td><td style="border-right: 1px solid black; padding: 2px 5px;">3</td><td style="padding: 2px 5px;">42</td></tr> </table>	$r = f(a_1, a_2)$	a_1	a_2	24	1	23	36	2	34	45	3	42
a_1	a_2																							
1	23																							
2	34																							
3	42																							
$r = f(a_1, a_2)$	a_1	a_2																						
24	1	23																						
36	2	34																						
45	3	42																						
(a) Relation v vor der Anwendung von $+(a_1, a_2)$		(b) Relation v nach der Anwendung von $+(a_1, a_2)$																						

Abbildung 4.7: Anwendung des fun_1to1-Operators

4.5.8 Bool'sche Ausdrücke

$$\begin{array}{c} \mathcal{S} \equiv \{\dots, a_1 \rightarrow e_1, a_2 \rightarrow e_2, \dots\} \quad G, \mathcal{V}, \mathcal{M} \vdash v \Rightarrow (\mathcal{S}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, _) \\ \hline G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash \begin{array}{c} a = a_1 > a_2 \\ | \\ v \end{array} \Rightarrow (\mathcal{S} \dot{\cup} \{a \rightarrow e_1 > e_2\}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \perp) \end{array}$$

Hier zeigen wir am Beispiel des $>$ -Operator, wie bool'sche Ausdrücke übersetzt werden. Der Operator $>$ platziert einen bool'schen Ausdruck in \mathcal{S} . Im

Speziellen wird hier ein Vergleich auf zwei bestehenden Elementen durchgeführt. Beispiel 4.5 zeigt die Anwendung auf eine Relation v .

Beispiel 4.5. *Der Vergleich wird auf den Attributen a_1 und a_2 ausgeführt.*

a_1	a_2	$a = a_1 > a_2$ v	a	a_1	a_2
100	10		true	100	10
$v \equiv 30$	50		false	30	50
5	1		true	5	1
40	3		true	40	3
(a) Relation vor der Anwendung von $a = a_1 > a_2$		(b) Relation nach der Anwendung von $a =$ $a_1 > a_2$			

Abbildung 4.8: Anwendung des num_gt-Operators

Wenn \mathcal{S} einen bool'schen Ausdruck enthält, so müssen wir bei einer Bindung eine Ausnahme machen, da SQL keinen bool'schen Ausdruck im SELECT-Teil akzeptiert. Wir überbrücken diese Restriktion durch das in Abschnitt 4.6.1 beschriebene Verfahren.

4.5.9 Rownumber

$$\frac{\mathcal{G}, \mathcal{V}, \mathcal{M} \vdash v \Rightarrow \left(\{a_1 \rightarrow e_1, \dots, a_i \rightarrow e_i, \dots, a_j \rightarrow e_j, \dots, a_n \rightarrow e_n\}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, _ \right)}{\hat{\mathcal{M}} \equiv \left(\begin{array}{l} t(b, a_1, \dots, a_n) \text{ AS} \\ (\text{SELECT ROWNUMBER() OVER (PARTITION BY } p \\ \text{ORDER BY } e_i, \dots, e_j) \text{ AS } b, \\ e_1 \text{ AS } a_1, \\ \dots, \\ e_n \text{ AS } a_n \\ \text{FROM } \mathcal{F}) \\ \text{WHERE } w \\ \mathcal{S} \equiv \{b \rightarrow b, a_1 \rightarrow a_1, \dots, a_n \rightarrow a_n\} \end{array} \right)}{\mathcal{G}, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash \begin{array}{c} \rho_{b:\langle a_i, \dots, a_j \rangle / p} \\ | \\ v \end{array} \Rightarrow \left(\hat{\mathcal{S}}, \{t\}, \perp, \hat{\mathcal{V}} \dot{\cup} \{t\}, \hat{\mathcal{M}} \dot{\cup} \hat{\mathcal{M}}, \perp \right)}$$

Der Rownumber-Operator erzeugt ein neues Attribut in der EingabeRelation. Dieses Attribut enthält Enumerationwerte, die bei 1 starten. Zusätzlich kann man dem Operator eine Partitionierungsspalte mitteilen. Diese bewirkt, dass die Enumeration in jeder Partition wieder bei 1 startet. Die optionale Sortierung, mit der man die Zuweisung der Enumerationswerte beeinflussen

kann, wird als eine Liste von Spalten mitgeteilt. Für den äquivalenten Ausdruck in SQL sei auf Abschnitt 3.1.2 verwiesen.

Aufgrund der ähnlichen Funktionalität wurde auf die Übersetzung des Number-Operators verzichtet.

4.5.10 Fragment

$$\frac{G, \mathcal{V}, \mathcal{M} \vdash v \Leftrightarrow (\mathcal{S}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \Delta')}{G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash \underset{v}{\text{frags}} \Leftrightarrow (\emptyset, \emptyset, \perp, \emptyset, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \Delta')}$$

Eine Eigenschaft von XQuery ist, neben dem Zugriff auf physische Dokumente, die Erzeugung von Elementen. Diese Besonderheiten werden in *Pathfinder* als Fragmente behandelt. So erzeugt die Elementkonstruktion zusätzlich zu den üblichen Result-Relationen, welche die Ergebnisse eines Ausdrucks weiterpropagieren, Fragmentrelationen, die dieselbe Enkodierung aufweisen wie die Document-Relation in Abschnitt 4.4.

Über den *frags*-Operator erfolgt der Zugriff auf ein solches Fragment.

4.5.11 Pfadausdrücke

Für die Übersetzung von Pfadausdrücken müssen wir ein paar neue Notationen einführen, welche die Inferenzregel etwas vereinfachen. Diese Notationen wurden bereits in [11] verwendet.

Definition 4.6. *Achsen.* Um die Semantik des Achsenkonzepts in XML zu repräsentieren verwenden wir ein Prädikat $axis(d_1, d_2, \alpha) \Leftrightarrow v \in ctx/\alpha$.

Definition 4.7. *Knotentest.* Die Unterstützung für Namen und Knotentest wird durch ein weiteres Prädikat gewährleistet. Der Test wird auf den Attributen $v.kind \in \{ 'elem', 'text' \}$ und den Tagnamen, der in $v.prop$ gespeichert wird, ausgeführt. Für einen Element-Knoten \bar{v} mit dem Tagnamen 'element', testen wir auf $\bar{v}.kind = 'elem'$ und $\bar{v}.prop = 'element'$. Diesen Sachverhalt fassen wir in einem einzigen Prädikat $test(d, n)$ zusammen.

$$\begin{array}{c}
G, \mathcal{V}, \mathcal{M} \vdash ctx \Rightarrow \left(\{iter \rightarrow e_1, item \rightarrow e_2\}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, _ \right) \\
G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash frag \Rightarrow \left(_, _, _, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \Delta \right) \\
\hline
\hat{\sqsupset} \equiv \begin{array}{c} \sqsupset_{\alpha:n}(iter, item) \\ \swarrow \quad \searrow \\ frag \quad \quad ctx \end{array} \\
\bar{\mathcal{F}} \equiv \mathcal{F} \dot{\cup} \{ \Delta \text{ AS doc1}, \Delta \text{ AS doc2} \} \\
\bar{w} \equiv (w \wedge (doc1.pre = e_2) \wedge axis(doc1, doc2, \alpha) \wedge test(doc2, n)) \\
G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash \hat{\sqsupset} \Rightarrow \left(\{iter \rightarrow e_2, item \rightarrow doc2.item\}, \bar{\mathcal{F}}, \bar{w}, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \perp \right)
\end{array}$$

Die Inferenzregel zur Übersetzung eines Pfadausdrucks nimmt einen Lokalisierungsschritt entgegen und generiert den SQL-Code für die Achse α mittels `pre`, `size` und `level` Werten und den Knotentest n . Alle Achsen der XPath Spezifikation können mit einfachster Arithmetik durch die `pre`, `size`, `level` Attribute ausgedrückt werden.

Für weitere Optimierungen bei der Übersetzung von Pfadausdrücken verweisen wir auf Abschnitt 5.1 und 5.1.1.

Beispiel 4.6. Für das folgende Beispiel wollen wir das äußerst einfache Dokument in Abbildung 4.9 verwenden.

```

<?xml version="1.0"?>
<recipes>
  <recipe>
    <title>Sushi</title>
  </recipe>
  <recipe>
    <title>Pizza</title>
  </recipe>
</recipes>

```

Abbildung 4.9: Dokument `recipes.xml`

```
fn:doc("recipes.xml")/child::recipes
```

Mit dem Prädikat für den Achsentest `axis` ($\alpha, doc1, doc2$) in Abbildung 4.10 filtern wir die `child`-Achse aus dem Dokument.

```
(doc2.level = (doc1.level + 1)) AND
((doc1.pre + doc1.size) >= doc2.pre) AND
(doc2.pre > doc1.pre)
```

Abbildung 4.10: Prädikat für den Achsentest

Der Knotentest $\text{test}(\text{doc2}, n)$ wird in SQL durch das Prädikat in Abbildung 4.11 formuliert.

```
doc2.kind = 1 AND doc2.prop = 'recipes'
```

Abbildung 4.11: Prädikat für den Knotentest

Mit `doc2.kind = 1` werden zusätzlich zum Achsentest nur jene Knoten gefiltert, welche einem XML-Element entsprechen.

Die Äquivalenz `doc.prop = 'recipes'` garantiert zusätzlich, dass nur jene Elemente gefiltert werden, welche mit `recipes` ausgezeichnet wurden.

Das Schlüsselwort `DISTINCT` ist nötig, um Duplikate, resultierend aus den Joins im `FROM`-Teil, zu entfernen. Dies ist ein Schwachpunkt dieser Implementierung, da sich `DISTINCT`-Operationen sehr negativ auf die Laufzeiten der Anfragen auswirken. In Abschnitt 5.1 werden wir nochmal dieses Thema aufgreifen.

4.5.12 Elementkonstruktion

$$\begin{array}{c}
 v_1\text{-QUERY} \\
 \hline
 G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash v_1 \Rightarrow \left(\{a_1 \rightarrow e_1, \dots, a_n \rightarrow e_n\}, \mathcal{F}, w, \dot{\mathcal{V}}, \dot{\mathcal{M}}, \Delta \right) \\
 \left. \begin{array}{l}
 \ddot{\mathcal{M}} \equiv \left\{ \begin{array}{l}
 t_1(a_1, \dots, a_n) \text{ AS } (\\
 \text{SELECT } e_1 \text{ AS } a_1, \\
 \dots, \\
 e_n \text{ AS } a_n \\
 \text{FROM } \mathcal{F} \\
 \text{WHERE } w)
 \end{array} \right\} \\
 \ddot{\mathcal{M}} \equiv \dot{\mathcal{M}} \cup \ddot{\mathcal{M}}
 \end{array} \right\} \\
 \ddot{\mathcal{V}} \equiv \dot{\mathcal{V}} \cup \{t_1\} \\
 G, \dot{\mathcal{V}}, \dot{\mathcal{M}} \vdash v \Rightarrow \left(\{a_1 \rightarrow a_1, \dots, a_n \rightarrow a_n\}, \{t_1\}, w, \ddot{\mathcal{V}}, \ddot{\mathcal{M}}, \Delta \right)
 \end{array}$$

$$\begin{array}{c}
v_2\text{-QUERY} \\
\hline
G, \bar{\mathcal{V}}, \bar{\mathcal{M}} \vdash v_2 \Rightarrow (\{a_1 \rightarrow e_1, \dots, a_n \rightarrow e_n\}, \mathcal{F}, w, \bar{\mathcal{V}}, \bar{\mathcal{M}}, \Delta) \\
\bar{\mathcal{M}} \equiv \left\{ \begin{array}{l} t_2(a_1, \dots, a_n) \text{ AS } (\\ \text{SELECT } e_1 \text{ AS } a_1, \\ \dots, \\ e_n \text{ AS } a_n \\ \text{FROM } \mathcal{F} \\ \text{WHERE } w) \end{array} \right\} \quad \hat{\mathcal{M}} \equiv \bar{\mathcal{M}} \cup \bar{\mathcal{M}} \\
\hat{\mathcal{V}} \equiv \bar{\mathcal{V}} \dot{\cup} \{t_2\} \\
G, \bar{\mathcal{V}}, \bar{\mathcal{M}} \vdash v \Rightarrow (\{a_1 \rightarrow a_1, \dots, a_n \rightarrow a_n\}, \{t_2\}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \Delta)
\end{array}$$

$$\text{newroots} \equiv \left\{ \begin{array}{l} \text{SELECT} \\ \text{scope.iter AS iter,} \\ 0 \text{ AS pos,} \\ -2 \text{ AS pre,} \\ \text{COALESCE(SUM(size + 1), 0) AS size,} \\ 0 \text{ AS level, 0 AS kind,} \\ \text{CAST(e1.item_qname AS CHAR(32)) AS prop} \\ \text{FROM} \\ (\text{SELECT * FROM } t_1) \\ \text{AS e1 INNER JOIN } \Delta \text{ AS doc ON} \\ \text{e1.item = doc.pre) AS e2} \\ \text{RIGHT OUTER JOIN} \\ (\text{SELECT * FROM } t_2) \text{ AS scope ON} \\ \text{scope.iter = e2.iter} \\ \text{GROUP BY scope.iter} \end{array} \right\} \quad (4.1)$$

$$\text{subcopy} \equiv \left\{ \begin{array}{l} \text{SELECT} \\ \text{e1.iter AS iter, e1.pos AS pos,} \\ \text{d2.pre AS pre, d2.size AS size,} \\ \text{d2.level - d1.level + 1 AS level,} \\ \text{d2.kind AS kind, d2.prop AS prop} \\ \text{FROM} \\ (\text{SELECT *} \\ \text{FROM (SELECT * FROM } t_2) \text{ AS e1,} \\ \Delta \text{ AS d1, } \Delta \text{ AS d2} \\ \text{WHERE} \\ \text{d1.pre = e1.item AND d2.pre } \geq \text{d1.pre AND} \\ \text{d2.pre } \leq \text{d1.pre + d1.size}) \end{array} \right\} \quad (4.2)$$

$$\text{newelems} \equiv \left\{ \begin{array}{l} \text{SELECT } e.\text{iter}, \\ \text{dmax.pre} + \text{ROW_NUMBER()} \text{ OVER} \\ \quad (\text{ORDER BY } \text{iter}, e.\text{pos}, e.\text{pre}) \text{ AS } \text{pre}, \\ e.\text{size} \text{ AS } \text{size}, e.\text{level} \text{ AS } \text{level}, \\ e.\text{kind} \text{ AS } \text{kind}, e.\text{prop} \text{ AS } \text{prop}, \\ \text{FROM} \\ \quad (\text{newroots} \\ \quad \text{UNION} \\ \quad \text{subcopy}) \text{ AS } e, \\ \quad (\text{SELECT MAX}(\text{pre}) \text{ AS } \text{pre} \\ \quad \text{FROM } \Delta) \text{ AS } \text{dmax} \end{array} \right\} \quad (4.3)$$

$$G, \mathcal{V}, \mathcal{M} \vdash \text{frag} \Rightarrow \left(_ , _ , _ , \hat{\mathcal{V}}, \hat{\mathcal{M}}, \Delta \right) \quad v_1 - \text{QUERY} \quad v_2 - \text{QUERY}$$

$$\begin{array}{c} \epsilon \\ \swarrow \quad \downarrow \quad \searrow \\ \hat{\epsilon} \equiv \text{frag} \quad v_1 \quad v_2 \end{array}$$

$$\Delta_{\text{new}} = \left(\begin{array}{l} \text{SELECT } \text{pre}, \text{size}, \text{level}, \\ \quad \text{kind}, \text{prop} \\ \text{FROM } t \end{array} \right) \quad \bar{w} \equiv (e.\text{level} = 0)$$

$$\hat{\mathcal{S}} \equiv \{ \text{iter} \rightarrow e.\text{iter}, \text{item} \rightarrow e.\text{pre} \}$$

$$\hat{\mathcal{M}} \equiv \left(\begin{array}{l} t(\text{iter}, \text{pre}, \text{size}, \text{level}, \text{kind}, \text{prop}) \text{ AS } \\ \quad (\text{newelems}) \end{array} \right)$$

$$G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash \hat{\epsilon} \Rightarrow \left(\bar{\mathcal{S}}, \{t \text{ AS } e\}, \bar{w}, \hat{\mathcal{V}} \dot{\cup} \{t\}, \hat{\mathcal{M}} \dot{\cup} \hat{\mathcal{M}}, \Delta_{\text{new}} \right)$$

XQuery ist nicht auf ein einziges Dokument limitiert, sondern kann in eine Anfrage mehrere Dokumente und Fragmente involvieren. Solche Fragmente können während der Laufzeit mit dem XQuery-Elementkonstruktor erstellt werden. Ein Beispiel dazu findet sich in Beispiel 4.7.

Der Elementkonstruktor erzeugt neben der Ergebnisrelation mit *iter*- und *pre*-Attributen auch ein neues Fragment namens Δ_{new} , in dem die neu erzeugten XML-Knoten und deren Nachfolger gespeichert werden. Δ_{new} wird in gewisser Weise an das persistente Dokument in der Datenbank angehängt und muss deshalb auch dasselbe Schema, beziehungsweise dieselbe Enkodierung aufweisen. Betrachtet man Gleichung 4.3 etwas genauer, so sieht man, dass der FROM-Teil der obersten Anweisung aus einer Vereinigung (UNION) zusammengesetzt ist.

Der erste Teil dieser Vereinigung, Gleichung 4.1, erzeugt die neuen Wurzeln eines Fragments – jene Elemente, die während der Elementkonstruktion neu erzeugt werden. Ein neues Wurzel-Element steht am Anfang eines jeden neuen Fragments, weshalb das Attribut *pos* mit 0 und *pre* mit -2 ausgezeichnet

iter	pos	pre	size	level	kind	prop
1	0	-2	4	0	1	menu

Tabelle 4.2: Ergebnis von *newroots*

Das Ergebnis von *newroots* stellt das Wurzelement als Fragment, zusammen mit seinem *iter* Wert.

iter	pos	pre	size	level	kind	prop
1	1	5	1	1	1	title
1	1	6	0	2	3	Sushi
1	2	11	1	1	1	title
1	2	12	0	2	3	Pizza

Tabelle 4.3: Ergebnis von *subcopy*

Mittels der Ergebnisrelation in Tabelle 4.1 werden in der *subcopy*-Teilanfrage mit einem *Join* jene Teilfragmente kopiert, welche als Unterelemente des neuen Wurzelknotens fungieren. Diese werden zusammen mit ihren Nachfolgern in die Relation aufgenommen.

iter	pre	size	level	kind	prop
1	15	4	0	1	menu
1	16	1	1	1	title
1	17	0	2	3	Sushi
1	18	1	1	1	title
1	19	0	2	3	Pizza

Tabelle 4.4: Ergebnisse von *newelems*

Die umschließende *newelems*-Anfrage vereinigt nun Tabelle 4.3 und 4.2 und numeriert die *pre* Werte auf der Basis des maximalen *pre* Wertes neu. Um die Dokumentordnung der Elemente zu garantieren wird zuerst eine Sortierung nach den Attributen *iter*, *pos* und *pre* eingeleitet.

4.6 Metaregeln

Die Integration des Lazy-Bindings schlägt sich auch bei der Formulierung der Inferenzregeln in Abschnitt 4.5 nieder. Dabei wird bei der Notation der Konsequenz fast durchgehend das Symbol \Rightarrow verwendet. Die folgenden Regeln

zeigen, wie man *Lazy Binding* formal mit Hilfe von Inferenzregeln formuliert. Dabei erfolgt eine Bindung, wenn der Eingangsgrad eines Knotens v des Anfrageplans $d_G^-(v) > 0$ ist.

$$\frac{G, \mathcal{V}, \mathcal{M} \vdash v \Rightarrow \left(\{a_1 \rightarrow e_1, \dots, a_n \rightarrow e_n\}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, _ \right) \quad d_G^-(v) > 1}{\hat{\mathcal{M}} \equiv \left\{ \begin{array}{l} t(a_1, \dots, a_n) \text{ AS } (\\ \text{SELECT } e_1 \text{ AS } a_1, \\ \dots, \\ e_n \text{ AS } a_n \\ \text{FROM } \mathcal{F} \\ \text{WHERE } w) \end{array} \right\}} G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash v \Rightarrow \left(\{a_1 \rightarrow a_1, \dots, a_n \rightarrow a_n\}, \{t\}, \perp, \hat{\mathcal{V}} \dot{\cup} \{t\}, \hat{\mathcal{M}} \cup \hat{\mathcal{M}}, \perp \right)$$

Ein Eingangsgrad $d_G^-(v) \leq 1$ hat zur Folge, dass das Symbol \Rightarrow nur durch \Rightarrow substituiert und die Regel in seiner ursprünglichen Form belassen wird. In Worten bedeutet dies, dass keine Bindung erfolgt, sondern der Inhalt des Operators an weitere Operatoren weiterpropagiert werden kann.

$$\frac{G, \mathcal{V}, \mathcal{M} \vdash v \Rightarrow \left(\mathcal{S}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, _ \right) \quad d_G^-(v) \leq 1}{G, \mathcal{V}, \mathcal{M} \vdash v \Rightarrow \left(\mathcal{S}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \perp \right)}$$

4.6.1 Ausnahme bei bool'schen Ausdrücken

Wie bereits in Abschnitt 4.5.8 verdeutlicht, akzeptiert SQL keine bool'schen Ausdrücke im **SELECT**-Teil eines Statements. Einen Ausweg aus diesem Dilemma bietet der **CASE**-Ausdruck (siehe Abschnitt 3.1.3).

```

CASE
  WHEN  $b$  THEN 1
  ELSE      0
END

```

Gesetzt den Fall, dass ein bool'scher Ausdruck bei einer Bindung in \mathcal{S} vorkommt reagieren wir wie folgt:

$$\begin{array}{c}
\mathcal{S} \equiv \{a_1 \rightarrow e_1, \dots, a_i \rightarrow b, \dots, a_n \rightarrow e_n\} \\
G, \mathcal{V}, \mathcal{M} \vdash v \Rightarrow (\mathcal{S}, \mathcal{F}, w, \hat{\mathcal{V}}, \hat{\mathcal{M}}, \perp) \quad d_G^-(v) > 1 \\
\hline
\hat{\mathcal{S}} \equiv \{a_1 \rightarrow a_1, \dots, a_i \rightarrow a_n = 1, \dots, a_n \rightarrow a_n\} \\
\hat{\mathcal{M}} \equiv \left\{ \begin{array}{l}
t(a_1, \dots, a_i, \dots, a_n) \text{ AS (} \\
\text{SELECT} \\
e_1 \text{ AS } a_1, \\
\dots, \\
\text{CASE} \\
\text{WHEN } b \text{ THEN } 1 \\
\text{ELSE } 0 \\
\text{AS } a_i, \\
\dots, \\
e_n \text{ AS } a_n \\
\text{FROM } \mathcal{F} \\
\text{WHERE } w) \\
\end{array} \right\} \\
G, \hat{\mathcal{V}}, \hat{\mathcal{M}} \vdash v \Rightarrow (\hat{\mathcal{S}}, \{t\}, \perp, \hat{\mathcal{V}} \dot{\cup} \{t\}, \hat{\mathcal{M}} \cup \hat{\mathcal{M}}, \perp)
\end{array}$$

Wir verschachteln den bool'schen Ausdruck b in einem **CASE**-Statement und binden dieses an das Attribut a_n . Sollte auf a_n in einem späteren Operator zugegriffen werden, wie zum Beispiel im Selektionsoperator σ_a , so muss in der **WHERE**-Liste auf $a_n = 1$ getestet werden, um die Korrektheit des Ausdrucks zu gewährleisten.

Kapitel 5

Optimierung

5.1 Bündelung von Pfadausdrücken

Der *Path-Step*-Operator ist verantwortlich für die Umsetzung des Achsenkonzepts von XPath, daher hat seine Implementierung große Auswirkungen auf die Performanz der Übersetzung.

Eine XQuery-Anfrage besteht häufig aus mehreren direkt aufeinander aufbauenden *Path-Step*-Operatoren.

Definition 5.1. *Sequenz von Pfadausdrücken.* Sei $1 \leq k \in \mathbb{N}$ und ctx eine Menge von Kontextknoten, so ist

$$c/\alpha_1 :: n_1/\alpha_2 :: n_2/\dots/\alpha_k :: n_k \quad (5.1)$$

eine Sequenz von Pfadausdrücken (Abbildung 5.1 zeigt diesen Sachverhalt).

Die in Gleichung 5.1 gezeigte Situation, wird vom Compiler als

$$((((c/\alpha_1 :: n_1)/\alpha_2 :: n_2)/\dots)/\alpha_n :: n_k)$$

wahrgenommen und schrittweise übersetzt. Der Grund für dieses Vorgehen ist die Duplikateliminiierung, die jeder *Path-Step*-Operator impliziert. Deshalb muss hierbei jeder Operator einzeln übersetzt werden, was zu einer großen Anzahl von einzelnen `common table expressions` führt, die jeweils einem einzigen XPath-Schritt zugeordnet sind.

Außerdem gehen bei der Übersetzung einzelner *Path-Step*-Operatoren Informationen verloren, die für einen weiteren *Path-Step*-Operator von Nutzen sind: Die Übersetzung eines *Path-Step*-Operator resultiert in einer Knotenreferenz, die Kontextknoten für den nächsten *Path-Step*-Operator in der Sequenz liefert. Die Ermittlung der `size`-, `level`-, `kind`- und `prop`-Spalten

müsste hier wieder mühsam durch einen Join ermittelt werden, da sie für die Berechnung des nächsten *Path-Step*-Operators in der Liste vonnöten sind. Im folgenden Abschnitt wollen wir diese Schritte in einem einzigen komplexen SQL-Statement zusammenfassen. Der *Path-Step*-Operator setzt implizit eine

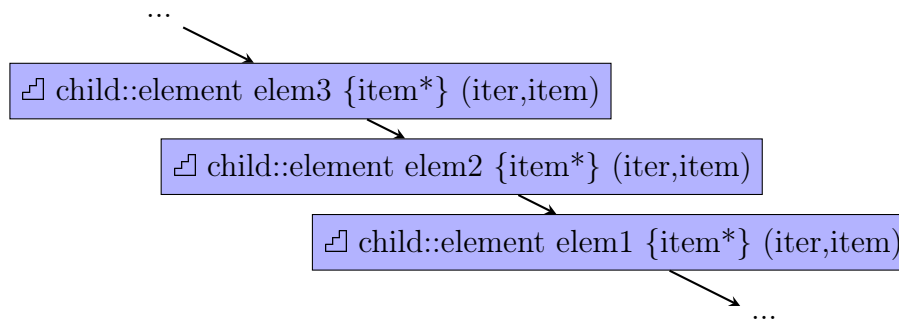


Abbildung 5.1: Mehrere *Path-Step*-Operatoren direkt hintereinander

Duplikateliminierung voraus, die es erfordert jeden Operator einzeln zu übersetzen. Es ist jedoch möglich, diese Eliminierung von Duplikaten lediglich im letzten *Path-Step*-Operator einer Sequenz durchzuführen. Unter Umständen können wir auch dort auf die Duplikateliminierung verzichten.

Alle Operatoren der logischen Algebra werden mit einer Struktur annotiert, welche spezifische Informationen zu den Eigenschaften des jeweiligen Operators enthält. Diese Struktur enthält auch eine Eigenschaft namens **Set**. Ist die **Set**-Eigenschaft gesetzt, so spielt es für einen Operator keine Rolle, ob in der Relation Duplikate vorhanden sind. In unserer SQL-Generierung wird die **Set**-Eigenschaft dazu verwendet um eine Duplikateliminierung (**DISTINCT**) im Zusammenhang mit *Path-Step*-Operatoren, wann immer es möglich ist, zu unterbinden.

5.1.1 Ausnützen von vorhandenen Index-Strukturen bei Pfadausdrücken

Indizes können nur auf physischen Relationen angewendet werden. Im Laufe einer XQuery-Anfrage arbeitet der *Path-Step*-Operator aber nicht nur auf XML-Dokumenten, die bereits physisch in der Datenbank vorliegen. Innerhalb einer Anfrage ist es auch möglich, dynamisch neue XML-Elemente hinzuzufügen. Solche Situationen machen es unmöglich für den *Path-Step*-Operator die Indizes der physisch vorliegenden Relation zu nützen. Formal

erhalten wir die in Gleichung 5.2 gezeigte Situation.

$$c \sqcup_{\alpha,n} (doc \dot{\cup} \Delta) \quad (5.2)$$

Dabei ist doc die physische Relation und Δ seien durch Konstruktoren in der Anfrage erzeugte Fragmente, bestehend aus transienten Knoten. Durch die disjunkte Vereinigung kann der Join-Operator die Indizes, die auf doc angelegt wurden, nicht nutzen. Eine befriedigende Lösung des Problems bietet das Distributivgesetz in Gleichung 5.3. Dadurch wird es möglich den Satz an Indizes für die Relation doc vollständig zu nutzen.

$$c \sqcup_{\alpha,n} (doc \dot{\cup} \Delta) = (c \sqcup_{\alpha,n} doc) \dot{\cup} (c \sqcup_{\alpha,n} \Delta) \quad (5.3)$$

Wir erhalten also eine effiziente Implementierung des *Path-Step*-Operators, da es sich bei Δ üblicherweise um ein sehr kleines XML-Fragment handelt, während doc den, mit Indizes versehenen, großen Teil an Daten beinhaltet¹. Abbildung 5.2 stellt das Vorgehen schematisch dar.

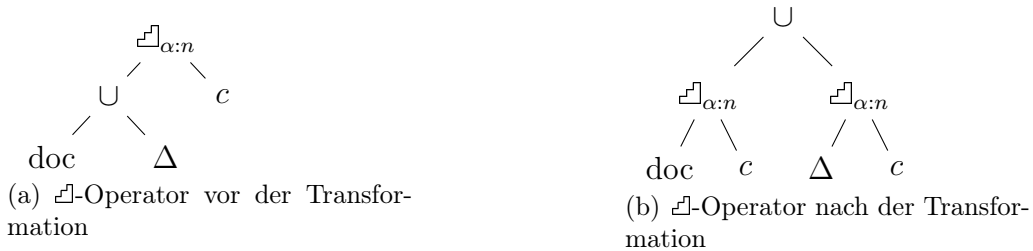


Abbildung 5.2: Transformation des *Path-Step*-Operators zur Ausnutzung von Index-Strukturen

5.2 Ausnützen von Index-Strukturen bei der Konstruktion von Elementen

Auch bei der Konstruktion von Elementen können wir dieselben Konzepte benutzen, die wir bereits in Abschnitt 5.1.1 verwendet haben. Der Operator hat nun also folgendes Muster:

¹Typischerweise gilt, $|\Delta| \ll |doc|$

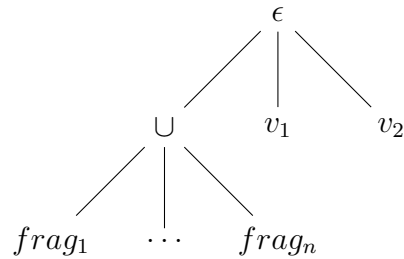


Abbildung 5.3: Elementkonstruktion mit mehreren Fragmenten

Die Anfragen für *subcopy* (Gleichung 4.2) und *newroots* (Gleichung 4.1) der Elementkonstruktion in Abschnitt 4.5.12 ändern sich dadurch wie folgt:

$$\text{newroots} \equiv \left\{ \begin{array}{l} \text{SELECT} \\ \text{scope.iter, 0 AS pos,} \\ \text{-2 AS pre,} \\ \text{COALESCE(SUM(size + 1), 0) AS size,} \\ \text{0 AS level, 0 AS kind,} \\ \text{CAST(e1.item_qname AS CHAR(32)) AS prop} \\ \text{FROM} \\ \text{(((SELECT * FROM t_1) AS e1 INNER JOIN \bar{\Delta} AS doc ON} \\ \text{e1.item = doc.pre)} \\ \text{UNION} \\ \text{(...)} \\ \text{UNION} \\ \text{((SELECT * FROM t_1) AS e1 INNER JOIN \bar{\Delta} AS doc ON} \\ \text{e1.item = doc.pre)) AS e1} \\ \text{RIGHT OUTER JOIN} \\ \text{(SELECT * FROM t_2) AS scope ON} \\ \text{scope.iter = e1.iter} \\ \text{GROUP BY scope.iter} \end{array} \right. \quad (5.4)$$

$$\text{subcopy} \equiv \left\{ \begin{array}{l}
 \text{SELECT} \\
 \quad \text{e1.iter, e1.pos, d2.pre,} \\
 \quad \text{d2.size, d2.level - d1.level + 1 AS level,} \\
 \quad \text{d2.kind, d2.prop} \\
 \text{FROM} \\
 \quad ((\text{SELECT} * \\
 \quad \quad \text{FROM (SELECT * FROM } t_2) \\
 \quad \quad \text{AS e1, } \bar{\Delta} \text{ AS d1, } \bar{\Delta} \text{ AS d2} \\
 \quad \quad \text{WHERE} \\
 \quad \quad \quad \text{d1.pre = e1.item AND d2.pre} \geq \text{d1.pre AND} \\
 \quad \quad \quad \text{d2.pre} \leq \text{d1.pre + d1.size}) \\
 \quad \text{UNION} \\
 \quad \dots \\
 \quad \text{UNION} \\
 \quad (\text{SELECT} * \\
 \quad \quad \text{FROM (SELECT * FROM } t_2) \\
 \quad \quad \text{AS e1, } \bar{\Delta} \text{ AS d1, } \bar{\Delta} \text{ AS d2} \\
 \quad \quad \text{WHERE} \\
 \quad \quad \quad \text{d1.pre = e1.item AND d2.pre} \geq \text{d1.pre AND} \\
 \quad \quad \quad \text{d2.pre} \leq \text{d1.pre + d1.size}))
 \end{array} \right\} \quad (5.5)$$

Die Struktur der restlichen Inferenzregel ändert sich, bis auf das Hinzufügen weiterer Fragmente, nicht und kann übernommen werden. Hier nutzen wir in Gleichung 5.4 den Index auf `doc.pre` im `INNER JOIN`-Teil und einen weiteren Index in Gleichung 5.5 für den `descendant`-Step auf `d1` und `d2`.

Kapitel 6

Die Implementierung des SQL-Backends

Das *Lazy Binding* ist eine zentrale Strategie, welche sich durch die gesamte Struktur des Übersetzungsmechanismus zieht. In diesem Kapitel werden wir auf einige implementierungsspezifische Details näher eingehen, die sowohl das *Lazy Binding*, als auch andere umgesetzte Konzepte näher beleuchten.

6.1 Baumstrukturen

Eine der zentralen Datenstrukturen des Backends ist `PF_sql_t` (dargestellt in Abbildung 6.1). Durch sie erfolgt die Umsetzung der Grammatik von SQL.

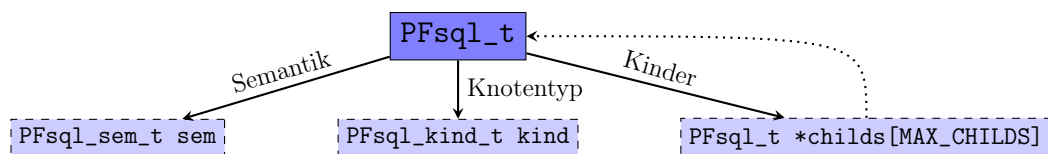


Abbildung 6.1: Die zentrale Datenstruktur der SQL-Generierung

Die Bedeutungen der Attribute dieser Struktur sind folgende:

1. `PFsql_sem_t sem` repräsentiert die Semantik der in Form einer weiteren Struktur.
2. `PFsql_kind_t kind` definiert den Typ des SQL-Knotens, zum Beispiel eine Zahl oder einen *SELECT ... FROM* Knoten.
3. `PFsql_t *child[MAX_CHILD]` referenziert ein Array, welches auf die Kinder des Knotens zeigt.

`PFsql_kind_t` wird als sogenannter Aufzählungstyp deklariert und beinhaltet jeden Typ der benötigten SQL-Knoten. Jeder Knotentyp hat zudem seine eigene Knotenkonstruktionsfunktion (Listing 6.1).

Listing 6.1: Verallgemeinerter Knotenkonstruktor

```
1 PFsql_t* PFsql_node_constr(PFsql_t *child_1, ..., PFsql_t *child_n);
```

Mehrere Hilfsfunktionen, die sogenannten *wiring-functions* bewerkstelligen die eigentliche Konstruktion, während die Konstruktionsfunktion darüber entscheidet, aus wie vielen Teilausdrücken sie zusammengesetzt werden soll. Die Anzahl der Teilausdrücke entscheidet darüber, welche *wiring-function* notwendig ist, um den gesamten Ausdruck darzustellen. Beispiel 6.1 zeigt an der konkreten Implementierung das Zusammenspiel der Funktionen.

Beispiel 6.1. *Anhand eines Subtraktions-Knotens lässt sich gut zeigen, welche Rolle die einzelnen Funktionen bei der Erzeugung eines Knotens spielen. Im Folgenden wollen wir den arithmetischen Ausdruck $54 - 12$, anhand eines Auszugs aus dem Code, erläutern.*

Listing 6.2: Basisfunktionen zur Erzeugung eines arithmetischen Ausdrucks

```
1 ...
2
3 PFsql_t* PFsql_op_leaf (PFsql_kind_t kind);
4
5 static PFsql_t* wire1(PFsql_kind_t k, const PFsql_t *n);
6
7 static PFsql_t*
8 wire2(PFsql_kind_t k, const PFsql_t *n1, const PFsql_t *n2);
9
10 /**
11 * Create a SQL tree node representing a literal integer.
12 *
13 * @param i The integer value to represent in SQL.
14 */
15 PFsql_t*
16 lit_int(int i)
17 {
18     PFsql_t *ret = leaf( sql_lit_int );
19     ret->sem.atom.val.i = i;
20     return ret;
21 }
22
23 /**
24 * Create a SQL tree node representing an arithmetic
25 * subtract operator.
26 *
27 * @param a The minuend.
28 * @param b The subtrahend.
29 */
30 PFsql_t*
31 sub(const PFsql_t *a, const PFsql_t *b)
32 {
33     return wire2(sql_sub, a, b);
```

```

34 }
35
36 ...

```

Nachdem wir mit Listing 6.2 die notwendige Vorarbeit geleistet haben, können wir mit dem in Listing 6.3 den arithmetischen Ausdruck $54 - 12$ in einer internen Baumdarstellung repräsentieren.

Listing 6.3: Subtraktion

```

1 PFsql_t *sub = sub(lit_int(54), lit_int(12));

```

Nachdem die Funktion `sql_op_leaf` Speicher für die beiden Integer-Knoten reserviert hat, wird die Kontrolle an die `wire2`-Funktion übergeben, welche ihrerseits wieder Speicher für den Vaterknoten, in diesem Fall die Subtraktion, durch `sql_op_leaf` allokiert. Die beiden Integer-Knoten werden als Kinder angehängt.

Die Instruktionen erzeugen die in Abbildung 6.2 dargestellte Datenstruktur.

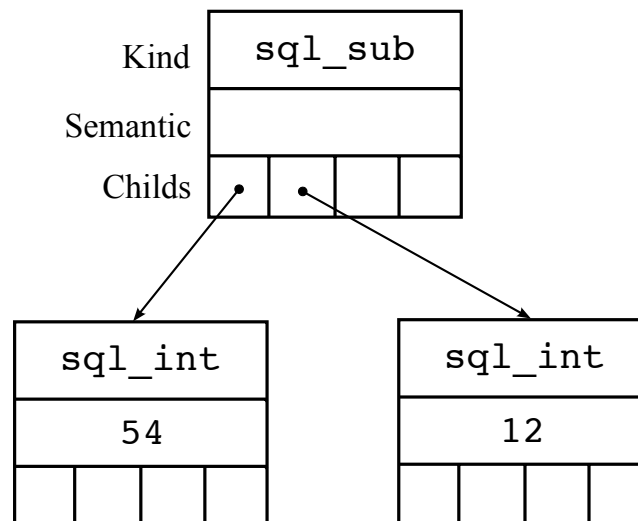


Abbildung 6.2: Baumdarstellung einer Subtraktion.

6.2 Implementierungsaspekte des *Lazy Binding*'s

Für die Unterstützung dieser Strategie muss jeder Knoten der relationalen Algebra mit Annotationen versehen werden. Hierzu definieren wir die in Abbildung 6.3 skizzierte Datenstruktur.

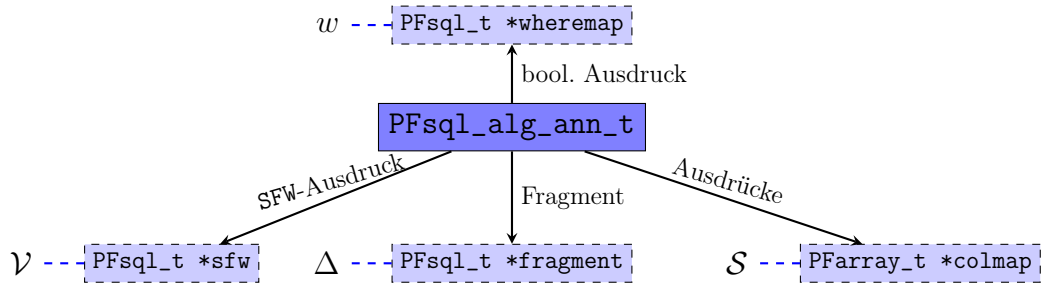


Abbildung 6.3: Algebra Annotationen (jede Datenstruktur ist mit dem entsprechenden Äquivalent aus den Inferenzregeln gekennzeichnet). \mathcal{M} ist in dieser Abbildung nicht sichtbar, da es, im Zuge des *Lazy Bindings*, Schritt für Schritt aufgebaut wird.

1. `PFsql_t *sfw` enthält nach jedem Übersetzungsschritt einen gültigen `SELECT FROM WHERE`-Ausdruck, auf den zurückgegriffen werden kann.
2. `PFsql_t fragment` wird dazu benötigt, eine Referenz auf ein Fragment zu speichern. Operatoren wie der Elementkonstruktor halten zusätzlich zu den üblichen Relationen Fragmente, die neue Elemente beinhalten, welche während einer Anfrage erzeugt werden. `fragment` hält eine Referenz auf einen validen SQL-Ausdruck, welcher dieses Fragment erzeugt.
3. `PFarray_t *colmap` speichert, wie bereits in Abschnitt 4.3 verdeutlicht, Abbildungen von Attributnamen auf SQL-Ausdrücke. Diese Ausdrücke werden zu weiteren Operatoren propagiert und weiterverarbeitet. Bei Bindungen werden diese Ausdrücke im `SELECT`-Teil verwendet.
4. `PFarray_t *wheremap` enthält eine Referenz auf einen bool'schen Ausdruck, welcher bei einer Bindung im `WHERE`-Teil platziert wird.

Im Wesentlichen wollen wir die Bindung an eine Tabellenreferenz solange wie möglich verzögern. Dazu müssen wir die Ausdrücke, welche in einem Übersetzungsschritt generiert werden, in einer Struktur sammeln, um sie im nächsten Übersetzungsschritt verfügbar zu machen. Zu diesem Zweck erfolgt nach der Übersetzung einer Regel eine Prüfung, ob der generierte Ausdruck einer Bindung bedarf. Hinreichende Kriterien dafür sind ein Eingangsgrad $d_G^-(v) > 0$ oder, wenn das generierte Statement keinem `SELECT-FROM-WHERE` Ausdruck entspricht. Zudem können Knoten explizit als `dirty` markiert werden, um eine Bindung zu erzwingen.

6.3 Mustererkennung mit Burg

Burg ist ein Übersetzerbau-Werkzeug, das sehr gute Dienste bei der Codeerzeugung leistet. Akzeptiert wird dabei ein in einer kontextfreien Grammatik formuliertes Problem. Burg kreiert ein C-Programm, das eine lineare Instruktionssequenz findet, welche äquivalent zur übergebenen Graphenstruktur ist. Eine Grammatik kann zudem noch mit Kosten annotiert werden. Burg findet dann, bei mehrdeutigen Grammatiken, eine kostenoptimale Überdeckung der Instruktionssequenz. Eine Einführung in Burg findet sich in [4].

Der Compilationsprozess von Pathfinder kann, durch den Einsatz von Burg, als eine Reihe von Transformationen, auf der Basis eines azyklischen Graphen, interpretiert werden. Der von Pathfinder ermittelte Anfrageplan in relationaler Algebra wird in einer Reihe von Transformationen in eine Sequenz von `common table expressions` transformiert.

Eine zentrale Rolle bei der Transformation spielen die einzelnen Regeln, die im Zuge der Problemformulierung im Code plaziert werden müssen. Jedem Knoten der relationalen Algebra ist eine Zahl zugeordnet, welche den Knoten eindeutig identifiziert. Zuallererst muss Burg diese eineindeutige Abbildung von Knoten auf den entsprechenden Identifier mitgeteilt werden. Die allgemeine Syntax wird in Abbildung 6.4 dargestellt. Mit einer solchen Regel definieren wir ein der relationalen Algebra entsprechendes Terminal.

```
%term label_id = id
```

Abbildung 6.4: Burg-Notation für die Zuordnung von Identifiern

`label_id` kann völlig frei gewählt werden, während `id` jener Zahl entspricht die einem Knoten der relationalen Algebra zugeordnet wurde.

Beispiel 6.2. *In diesem Beispiel wollen wir uns anhand eines konkreten Codeauszugs diese Zuordnung verdeutlichen. Jeder Knoten der relationalen Algebra entspricht dabei einem der aufgelisteten Identifier. Für die Definition der Grammatik kann später `label_id` verwendet werden.*

```

1 ...
2 %start Query
3 /**
4  * Node identifiers, corresponding to the node kinds in
5  * include/logical.h
6  */
7 %term serialize      = 1 /**< serialize algebra expression
8                      (Placed on the very top of the tree.) */
9 %term lit_tbl       = 2 /**< literal table */
10 %term empty_tbl    = 3 /**< empty literal table */
11 %term attach       = 4 /**< attach constant column */

```

```

12 %term cross          = 5 /**< cross product(Cartesian product) */
13 %term eqjoin        = 6 /**< equi-join */
14 %term semijoin      = 7 /**< semi-join */
15 ...

```

Burg startet seine Mustererkennung mit dem Nichtterminal `Query` und weist in der ersten Phase des Compilationsprozesses jedem Knoten seinen Identifier zu.

Die Nichtterminale der kontextfreien Grammatik werden erst später, durch die in Abbildung dargestellte Notation, definiert.

```
label_id: tree_pattern = id (cost);
```

Abbildung 6.5: Burg-Notation für Grammatik-Regeln

Eine gültige Regel enthält mindestens eine `label_id` und ein `tree_pattern`. Die Angabe der Kosten (`cost`) ist optional und bei Nichtangabe verwendet *Burg* Standardkosten. Die Muster, auch *subject trees* genannt, werden in einer Prefix-Notation formuliert.

```
cross(Re1, Re1)
```

6.3.1 Muster bei der Bündelung von *Path-Steps*

Um das Konzept in Abschnitt 5.1.1 umzusetzen verwenden wir wieder *Burg*. Abbildung 6.6 zeigt eine Sequenz von Pfadausdrücken und entspricht `ctx/elem1/elem2/elem3`.

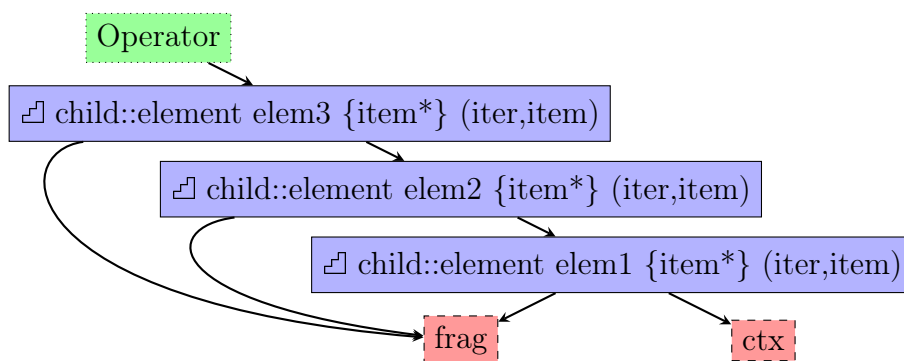


Abbildung 6.6: Sequenz von Pfadausdrücken

Der erste, mit `elem1` annotierte Ausdruck, muss als erstes von *Burg* erfasst werden. Hierbei darf noch keine Bindung erfolgen. Bei dem mittleren,

mit `elem2` versehenen Pfadausdruck, bedarf es auch noch keiner Bindung. Eine Bindung muss erst dann erzwungen werden, wenn der letzte *Path-Step*-Operator in einer Sequenz erfasst wird. Erst dann erfolgt auch die Eliminierung der Duplikate mittels `DISTINCT`-Schlüsselwort. In `Burg` kann genau diese Linearisierung der Abarbeitung von Pfadausdrücken recht elegant herbeigeführt werden.

```

1 Rel:      ScjRel          = 52 (10);
2 ScjRel:  scjoin (Frag, Rel) = 53 (0);
3 ScjRel:  scjoin (Frag, ScjRel) = 54 (0);

```

Mit diesen Ausdrücken wird das gewünschte Verhalten von `Burg` herbeigeführt. Um die Anwendung der Regeln nachzuvollziehen, beziehen wir uns auf Abbildung 6.6.

Der mit `elem1` gekennzeichnete \sqcup -Operator ist der erste in dieser Sequenz. Er baut offensichtlich direkt auf einem Fragment und einer weiteren Relation namens `ctx` auf. Die Regel in Zeile 2 ist hier also die richtige Wahl, um das Muster zu überdecken. Diese Regel führt den Ausdruck in eine Relation namens `ScjRel` über. Für den zweiten Ausdruck stellt sich nun die Frage, ob die Regel in Zeile 1 und anschließend wieder die Regel in Zeile 2 angewendet werden soll, oder ob die Regel in Zeile 3 die richtige Wahl ist. Hier kommt die kostenoptimierende Eigenschaft von `Burg` ins Spiel. Man beachte, dass Zeile 1 mit einem Kostenwert von 10 annotiert ist, während Zeile 3 mit Kosten 0 versehen wird. `Burg` erkennt also, dass die erste unserer beiden Optionen die teurere wäre und entscheidet sich deshalb für die Regel in Zeile 3. Diese wird schlussendlich auch, bei dem mit `elem3` ausgezeichneten, Operator verwendet.

Der mit Punkten umrandete Operator fordert eine Relation (`Rel`) als Eingabe und so bleibt uns nun keine andere Wahl, die teure Regel in Zeile 1 anzuwenden und `ScjRel` in `Rel` zu überführen. Die Regel in Zeile 1 führt keinen weiteren Achsensschritt durch, sondern führt eine Bindung herbei, sollte dies vonnöten sei (siehe auch Abschnitt 5.1).

Kapitel 7

XQuery auf DB2

Die Anfragen für den Test auf DB2 stellt das **XMark**-Benchmark (siehe [12]). In insgesamt zwanzig Anfragen werden eine Vielzahl an Eigenschaften von XQuery abgedeckt. Zudem wird ein Generator für XML-Dokumente zur Verfügung gestellt, welches Dokumente in beliebiger Größe erzeugt. Dieses Werkzeug namens `xmlgen` produziert XML-Dokumente, welche Auktionen modellieren. Anzahl und Typ der Elemente werden dabei anhand einer Vorlage unter einer bestimmten Wahrscheinlichkeitsverteilung gewählt. Abbildung 7.1 zeigt ein solches mit `xmlgen` erzeugtes Dokument.

Insgesamt werden die Tests auf neun Dokumenten durchgeführt, die sich in ihrer Größe von 1 MB bis zu 112 MB erstrecken.

Die Tests werden auf eine Linux-basierendem Server mit zwei 3.2 Ghz Intel Xeon Prozessoren, 8 GB Hauptspeicher und 280 GB SCSI Sekundärspeicher ausgeführt.

Jedes Dokument wird mittels `xmlshred` in eine `pre/size/level` Enkodierung transformiert und dann mit Hilfe des *bulkloaders* von DB2 in die Datenbank eingespeist.

```
[bash]$ xmlshred -f auction.xml -o A_sql_exp.csv
```

Die Optionen `-f` und `-o` von `xmlshred` geben jeweils das zu verarbeitende XML-Dokument und dessen Ausgabe an.

7.1 Vorbereitungen

Pathfinder ist nicht direkt in das Datenbanksystem integriert. Je eine XQuery-Anfrage wird also zuerst übersetzt und der generierte SQL-Code wird in

```
<?xml version="1.0"?>
<site>
  <regions>
    <africa>
      <item><id>item0</id>
      <location>United States</location>
      <quantity>1</quantity>
      <name>duteous nine eighteen </name>
      <payment>Creditcard</payment>
      .
      .
      .
    </africa>
    <europe>
      .
      .
      .
    </europe>
    .
    .
    .
  </regions>
</site>
```

Abbildung 7.1: Beispiel eines von `xmlgen` generierten XML-Dokuments

einzelne `.sql`-Dateien geschrieben. Anschließend wird eine Verbindung zu DB2 hergestellt und die Anfrage mit `db2batch` ausgewertet.

```
[bash]$ pf -g query.xq > query.sql
[bash]$ db2 "connect to xml"
[bash]$ db2batch -d xml -f query.sql -i complete \
    &> query.batch
```

`db2batch` wird mit der Option `-d xml` ausgeführt, wobei `xml` die Datenbank repräsentiert, welche die enkodierten XML-Dokumente beinhaltet.

Das in der Datenbank, in enkodierter Form, vorliegende XML-Dokument liegt in den generierten SQL-Anfragen als `doc` vor und muss vor der Ausführung mit dem Namen der entsprechenden Relation im Datenbanksystem ersetzt werden. Dies kann einfach mit dem Streameditor `sed` erledigt werden.

```
[bash]$ cat query.sql | sed "s/doc/A_sql_exp.doc/g" \  
> A_query.sql
```

Die Option `-f query.sql` von `db2batch` übergibt die SQL-Anfrage an das Datenbanksystem. Die Option `-i complete` bewirkt, dass die einzelnen Phasen, in denen das Datenbanksystem die Anfrage auswertet hinsichtlich der verbrauchten Zeit aufgelistet werden.

Die physischen Tabellen müssen noch mit Indizes versehen werden, damit die generierten SQL-Anfragen davon profitieren können. Diese lassen wir von `db2advise` vorschlagen. Dazu müssen wir dem Werkzeug alle auszuführenden Anfragen übergeben. `db2advise` schlägt dann eine Reihe von Indizes vor, welche die Ausführungszeit der Anfragen minimieren. Für einen Einblick in die Auswahl der Indizes, welche `db2advise` vorschlägt, verweisen wir auf [9] und [5]. Nach dem Anlegen der, von `db2advise` vorgeschlagenen, Indizes wird das Kommando `runstats` zur Ermittlung aktueller Statistiken ausgeführt.

7.1.1 Strategie bei der Auswertung

Beim Testen und Auswerten der Anfragen wird eine *warm-cache* Strategie verwendet. Bevor eine Anfrage ausgewertet wird, erfolgt eine Ausführung durch die Datenbank mit dem Kommando `db2`. Daraufhin wird der Test auf fünf Anfragen gestartet und deren Ausführungszeit gemessen. Das arithmetische Mittel dieser gemessenen Ausführungszeiten $t = \frac{t_1 + \dots + t_5}{5}$ wird als effektives Testergebnis herangezogen.

Zudem hat diese Strategie den Vorteil, dass der Anfrageoptimierer von DB2 nur für die erste Anfrage, die nicht in die Messung integriert wird, einen optimalen Plan finden muss. DB2 erkennt gleiche Anfragen und zieht die bereits ermittelten Pläne zur Rate, um sie auszuwerten. Das Datenbanksystem verschwendet also keine zusätzliche Zeit für die Optimierungsphase und beginnt sofort mit der Auswertung. Im Schnitt verbringt DB2 circa 200ms mit der Findung, beziehungsweise Optimierung des Plans. Wenn die Query schon bekannt ist verringert sich diese Zeit auf nicht einmal eine Millisekunde.

7.2 Bündeln von Lokalisierungspfaden

Durch die Bündelung von *Path-Step*-Operatoren kommt es zu einer Ansammlung von Tabellen im `FROM`-Teil des `SQL-SELECT`-Statements. Der Optimierer in DB2 versucht die Anfrage bezüglich der Kosten zu optimieren. Um dies zu bewerkstelligen ordnet er die Relationen im `FROM` Teil in einem Binärbaum an. Dabei muss der Optimierer, in einem Enumerationsalgorithmus,

alle möglichen Binärbäume auf Kostenoptimierungen hin testen. Es kommt zu einer kombinatorischen Explosion, deren Hintergrund in den Catalan-Zahlen (Gleichung 7.1) zu finden ist.

$$C_{n+1} = \frac{1}{n+1} \binom{2n}{n} \quad (7.1)$$

C_n kann man dabei als die Anzahl aller möglichen Binärbäume mit $n + 1$ Blättern interpretieren. Zur Erinnerung: Ein Binärbaum mit n inneren Knoten hat $n + 1$ Blätter.

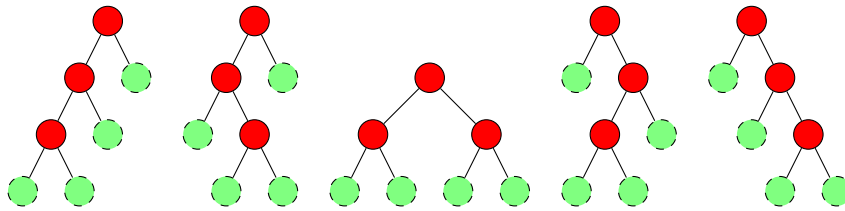


Abbildung 7.2: Mögliche Binärbäume mit vier Blättern

n	0	1	2	3	4	5	6	7	8
C_n	1	1	2	5	14	42	132	429	1430

Tabelle 7.1: Tabelle mit einigen Werten für C_n

Die Werte für C_n steigen, wie Abbildung 7.1 zeigt, sehr schnell an. Der Algorithmus zur Kostenoptimierung von DB2 gibt, durch diese enorme kombinatorische Hürde, nach einer Reihe von Tests auf und gibt sich auch mit suboptimalen Auswertungsplänen zufrieden.

Ausgang für den Test ist die in XQuery-Anfrage Q15.xq aus dem XMark-Benchmark. In insgesamt 12 Testfällen wurde die Anfrage in Abbildung 7.4 auf die Anfrage in Abbildung 7.3 reduziert. Wir variieren die *Path-Steps* und erweitern die Anfrage von einem *Path-Step* auf zwölf *Path-Steps*.

```
let $auction := doc("auctionG.xml") return
for $a in
  $auction/site/text()
return {$a}
```

Abbildung 7.3: Abfrage scj1.xq

In zwei Szenarien wird die Codegenerierung sowohl mit Unterstützung der Bündelung von *Path-Steps*, als auch ohne getestet. Abbildung 7.2 zeigt

```

let $auction := doc("auctionG.xml") return
for $a in
  $auction/site/
    closed_auctions/closed_auction/annotation/
    description/parlist/listitem/parlist/
    listitem/text/emph/keyword/text()
return {$a}

```

Abbildung 7.4: Abfrage scj12.xq

das Testergebnis mit Bündelung der *Path-Steps*. Die x -Achse zeigt hierbei die Anzahl der Lokalisierungsschritte, während die y -Achse die Ausführungszeit der Anfrage in Sekunden angibt. Während die Anfrage mit acht Lokalisierungsschritten noch in einem Bruchteil einer Sekunde ausgewertet werden kann, zeigt sich bei neun Schritten ein erster Sprung in der Auswertungszeit. Diese Tendenz setzt sich auch in den folgenden Anfragen fort.

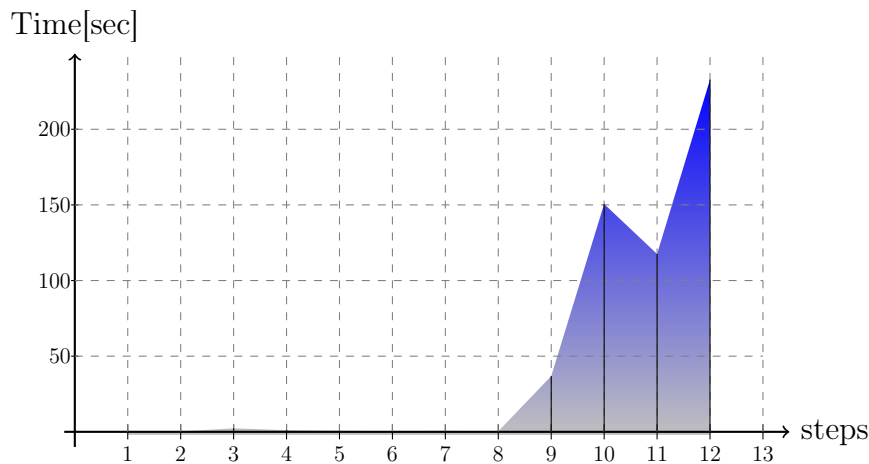


Tabelle 7.2: Anfragen mit Bündelung

Abbildung 7.3 zeigt denselben Test ohne Bündelung. Auch hier ist der Sprung bei neun Lokalisierungsschritten deutlich zu sehen. Ein mögliche Erklärung hierfür ist, dass der Optimierungsalgorithmus von DB2, durch den Einsatz von `common table expressions`, in der Auswertungsreihenfolge nicht restringiert wird und freie Hand bei potenziellen Optimierungsschritten hat. Hier tritt zutage, dass die Anfrage in einem einzigen komplexen Plan ausgewertet und, durch die Bindung an Tabellenreferenzen, keine sofortige Auswertung der Teilanfrage forciert wird. Dadurch wird es möglich, dass DB2, in einer Reihe von Optimierungsschritten, dieselben Verbesserungen vornimmt

wie der Codegenerator.

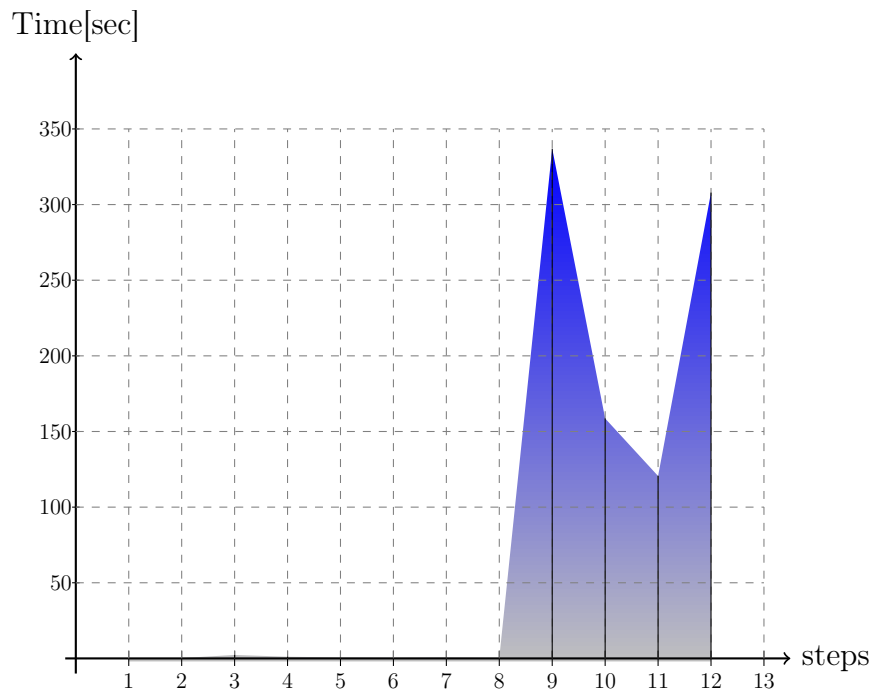


Tabelle 7.3: Anfragen ohne Bündelung

Eine mögliche Lösung dieses Problems wäre eine Materialisierung oder eine View-Definition nach der Bündelung von acht Lokalisierungsschritten. Damit würde das Datenbanksystem zu einer sofortigen Auswertung der Anfrage gezwungen und würde damit das Problem der Ansammlung von Tabellen im FROM-Teil unterbinden.

Der große Verlust von Performanz kann auch in den von DB2 generierten Plänen beobachtet werden. Bei sehr langen Pfadausdrücken degenerieren diese von indexnutzenden *left-deep-trees* zu *bushy trees*, welche die vorhandenen Indizes nicht optimal nutzen.

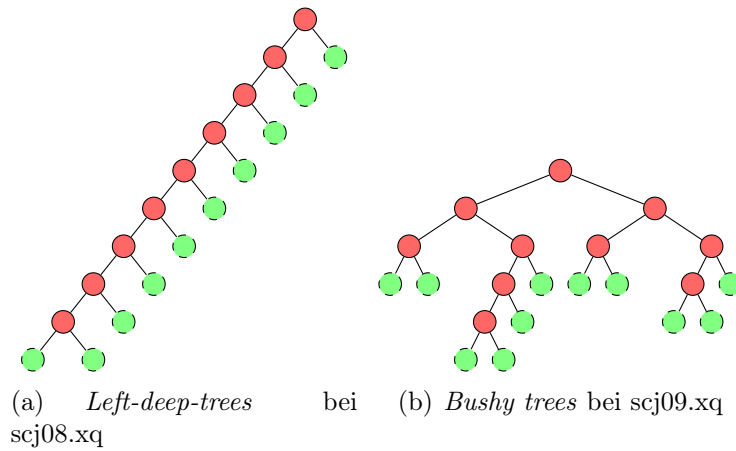


Abbildung 7.5: Auswertungspläne

Abbildung 7.5 zeigt die Auswertungspläne von `scj08.xq` und `scj09.xq`. Die Blätter des Baumes bezeichnen jene physischen Dokumente, auf denen ein vorhandener Index angewendet werden kann. Im Plan von `scj09.xq` erkennt man deutlich die Suboptimalität des Plans, die es nicht mehr erlaubt Indizes auf Zwischenergebnisse anzuwenden, während dies bei `scj08.xq` noch ohne Probleme möglich ist. Während für manche Anfragen *bushy-join-trees* die nötige Performanz schaffen, ist diese Join-Reihenfolge für diese Art von Anfragen nicht optimal. Dies bringt uns zur Erkenntnis, dass DB2 bei zunehmender Pfadlänge Probleme hat die Kosten richtig zu schätzen. In Abschnitt 8.3 und 8.2 werden wir eine mögliche Lösung dieses Problems vorschlagen.

7.3 Keine Unterstützung von Attributen

Attribute werden in der aktuellen Version der SQL-Erzeugung noch nicht unterstützt. Um die Anfragen des XMark-Benchmark trotzdem zu unterstützen behelfen wir uns eines Tricks. Wir behandeln Attribute als Kinder des aktuellen Elements. Die Dokumente müssen in dieser Hinsicht vorbereitet werden. Abhilfe schafft dabei ein XSLT-Skript, welches Attribute dahingehend transformiert, dass sie als Kinder des aktuellen Element-Knotens betrachtet werden können.

Diese Vorgehensweise wirkt sich auch auf die Anfragen aus, da an manchen Stellen zusätzliche Elemente erzeugt werden, welche die Anfragen in ihrer Auswertungszeit negativ beeinflussen. Folgendes Experiment in Abbildung 7.6 zeigt das Potential, welches die Unterstützung von Attributen mit sich bringen kann. In den XMark-Queries wurde dabei auf die Generierung von zusätzlichen Elementen verzichtet und stattdessen Textknoten verwendet.

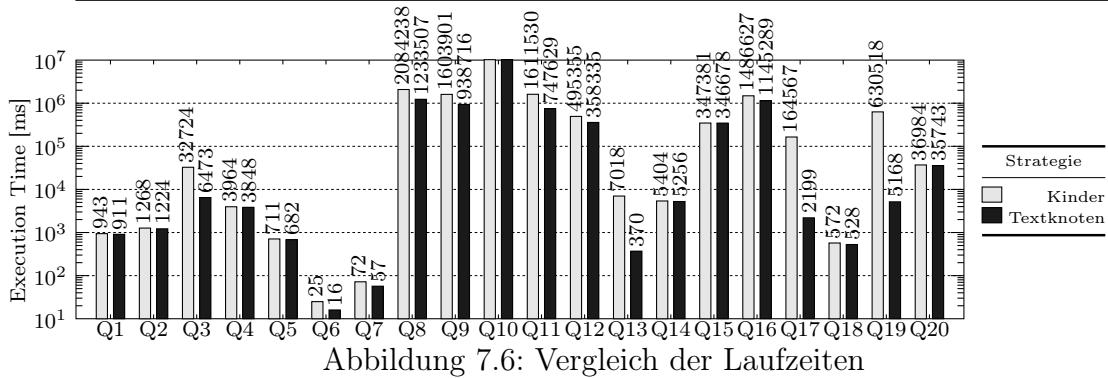


Abbildung 7.6: Vergleich der Laufzeiten

Bei Q13, Q17 sowie Q19 wird die Laufzeit der Queries, ohne die Generierung von zusätzlichen Elementen, maßgeblich verbessert. In der Unterstützung von echten Attributen liegt also noch sehr großes Potenzial zur Verbesserung der Anfragen.

7.4 Indizes

Ein wesentlicher Bestandteil unserer Optimierung ist die intensive Nutzung von Indexstrukturen auf physischen Relationen. In Abbildung 7.7 wollen wir uns die Unterschiede der XMark-Benchmark Anfragen mit und ohne Indizes verdeutlichen.

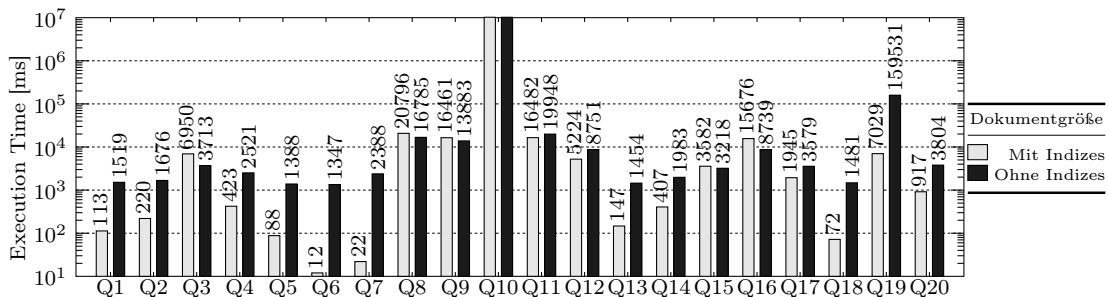


Abbildung 7.7: XMark-Benchmark Anfragen auf einer 10 MB Relation mit und ohne Indizes

Während die Indizes die meisten Anfragen deutlich verbessern, scheinen sie sich auf Anfragen wie Q3 nachteilig auszuwirken. Dies könnte auf unvermeidbare Zwischenergebnisse von Q3 zurückzuführen sein, die es erst gar nicht möglich machen, von den Indizes zu profitieren. Im Gegenteil, der zusätzliche Verwaltungsaufwand wirkt sich negativ auf die Laufzeiten der Anfragen aus.

7.5 Analyse von Q08.xq

Hier soll eine genauere Analyse von Q08.xq zu Tage bringen, welche Teile der Anfrage sehr viel Zeit verlieren. Eine Übersetzung der Anfrage findet sich in Anhang A.

Tabelle 7.4 zeigt, wieviel Zeit die einzelnen `common table expressions` benötigen.

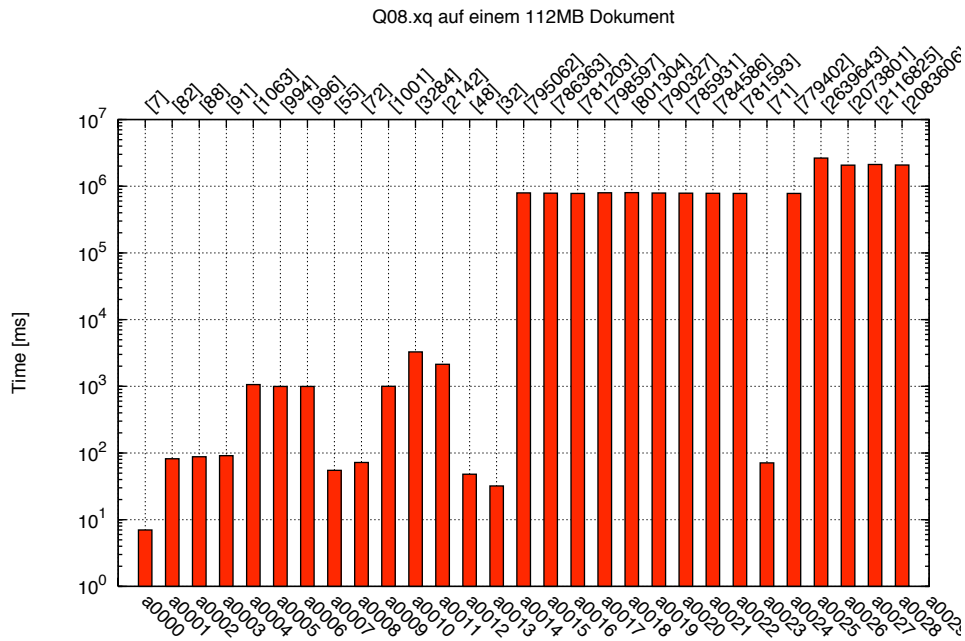


Tabelle 7.4: Zeiten der einzelnen `common table expressions` von Q08.xq

Tabelle 7.5 zeigt die Anzahl der Knoten, die die einzelnen `common table expressions` zurückliefern.

7.6 Test auf Dokumenten verschiedener Größe

In diesem Abschnitt wollen wir die Laufzeiten der Anfragen des XMark-Benchmarks auf Relationen verschiedener Größen testen. Wir beginnen bei einer Relation von 1.1 und beenden unser Experiment auf einer Relation mit 112.1 MB. Dabei werden die Relationen stets um einen Faktor 10 exponentiell vergrößert.

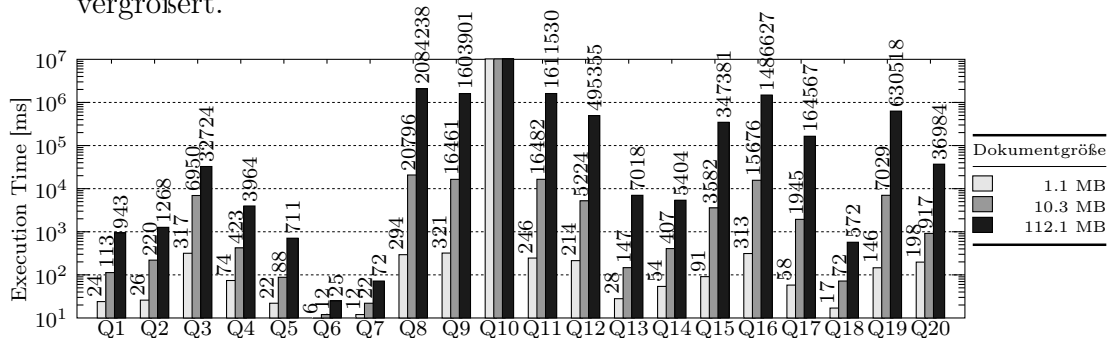


Abbildung 7.8: XMark-Benchmark Anfragen auf Dokumenten verschiedener Größe

In Abbildung 7.8 werden die Laufzeiten der XMark-Queries auf Dokumenten verschiedener Größe dargestellt. Dies gibt uns einen ersten Überblick, wie die Queries in Abhängigkeit der Größe der physischen Relation skalieren. Es ist zu beachten, dass wir für die Experimente modifizierte Queries (gemäß Abschnitt 7.3) verwendet haben, was die Laufzeiten, wegen der Elementkonstruktoren, negativ beeinflusst.

Elementkonstruktoren wirken sich insgesamt (siehe auch Abschnitt 7.5) sehr negativ auf die Gesamtp Performanz der Anfragen aus. Wie [8] zeigt, können wir in diesem Punkt mit einer Materialisierung, in einem nächsten Entwicklungsschritt sehr gute Ergebnisse erzielen. Mit einer Materialisierungsstrategie könnten wir auch lange *Path-Step*-Operatoren in ihrer Laufzeit verbessern (siehe auch Abschnitt 8.3).

An den Daten können wir ohne Ausnahme eine lineare Skalierung beobachten. Das einzige Manko ist, dass wir für Q10 keine Auswertung erzielen konnten, was auf den häufigen Gebrauch von Elementkonstruktoren zurückzuführen ist.

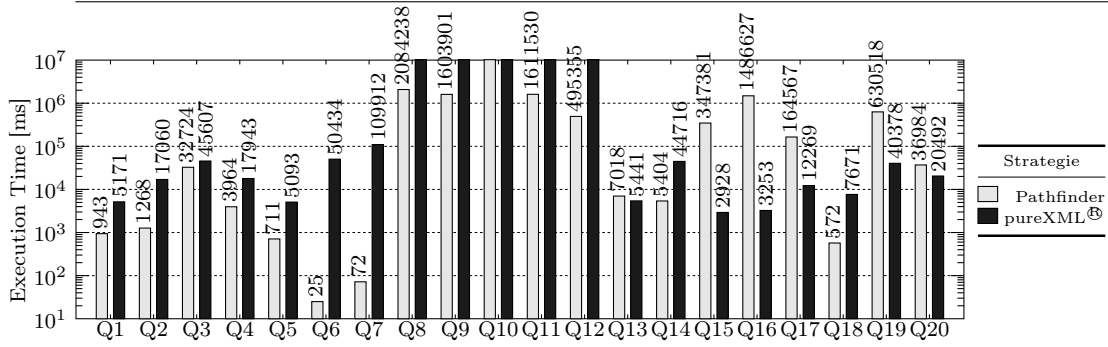


Abbildung 7.9: XMark-Benchmark Anfragen auf einem 112.1 MB Dokument

Der Vergleich mit einem nativen XML-System, wie pureXML[®] von IBM, zeigt in aller Deutlichkeit die Stärken unseres Codegenerators. Besonders die Anfragen Q1 bis Q7 verzeichnen eine enorme Verbesserung der Laufzeiten bei unserem Ansatz, im Gegensatz zum nativen System. Es ist zu beachten, dass die Anfragen Q8 bis Q12 auch nach einigen Stunden von pureXML[®] nicht ausgewertet werden können.

Kapitel 8

Ausblick

Dieses Kapitel stellt noch einige Konzepte vor, welche für die zukünftige Entwicklung und Verbesserung der SQL-Generierung wichtig sind.

8.1 Unterstützung von Attributen

Um valide XQuery-Anfragen zu stellen, wird es in Zukunft erforderlich sein Attribute zu integrieren. Dadurch wird nicht nur die Vorbehandlung des Dokuments und der Anfragen durch ein XSLT-Skript unnötig, sondern auch die Laufzeiten der meisten Anfragen können stark verbessert werden.

8.2 Selektivität

Der Optimierer von DB2 verwendet vielseitige Kriterien, um optimale Pläne zu generieren. So ist zum Beispiel die Selektivität ein Maß, welches DB2 mitteilt, wieviele Tupel, im Mittel, in einer Relation denselben Schlüsselwert besitzen. Die Selektivität ist, auf der Suche nach optimalen Plänen, ein wichtiges Maß für den Optimierer.

Nehmen wir einmal an, die zu betrachtende Relation beinhaltet 1,000,000 Tupel. Nehmen wir weiterhin an, unsere Relation besitze zwei Indizes. Der erste Index wird über ID1, der zweite über ID2 erzeugt. ID1 sei unser Primärschlüssel und somit eindeutig, damit besitzt er eine Selektivität von 1,000,000. Für den zweiten Index sei dieser Wert 100. Das heißt für den Schlüssel ID2 existieren 100 verschiedene Werte in der Datenbank. Damit weiß DB2, dass bei einer Suche nach einem bestimmten Wert von ID2 im Mittel 100 Werte zurückgeliefert werden. DB2 weiß aber nichts über die aktuelle

Verteilung des ID2-Attributs. Es besitzt nur Informationen über die durchschnittliche Verteilung.

In DB2 kann die Verteilung von Selektivitäten manuell über eine spezielle Syntax mitzuteilen. Dies könnte, besonders bei langen Lokalisierungspfaden, große Vorteile eröffnen, die dem Optimierer bei der Suche nach dem optimalen Plan unterstützen. So könnten wir zum Beispiel ein Prädikat mit einer hohen Selektivität auszeichnen, um das Verhalten des Optimierers positiv zu beeinflussen.

```
SELECT *  
FROM table AS t  
WHERE t.id = ? selectivity 0.01
```

Abbildung 8.1: Anfrage mit Angabe einer Selektivität

In Abbildung 8.1 wird das Prädikat `t.id = ?` mit einer hohen Selektivität ausgezeichnet. Der Optimierer wird nun mit dieser neuen Information versuchen den Plan noch weiter zu optimieren.

Diese Technik kann DB2 vor allem bei der Findung von Plänen bei langen Lokalisierungspfaden (Abschnitt 7.2) helfen. DB2 hat hier, wie bereits erläutert, Probleme die Kosten richtig zu schätzen, was in einem Performanzverlust resultiert.

8.3 Materialisierung

Bereits bei den Experimenten mit sehr langen Lokalisierungspfaden in Abschnitt 7.2 wird klar, dass eine sehr hohe Ansammlung von Relationen im FROM-Teil negative Auswirkungen auf die Optimierung des Anfrageplans hat. Die Pläne degenerieren in ihrer Struktur von den typischen *left-deep-trees* in *bushy-trees*, die durch Restriktionen des DB2-Optimierers zustande kommen. Diese Repräsentationen führen zu enorm schlechten Laufzeiten bei zu langen Lokalisierungspfaden. Solche degenerierten Pläne treten aber nicht nur bei Lokalisierungspfaden auf, sondern werden indirekt durch das *Lazy Binding* gefördert und können bei jeglicher Konstellation der Operatoren zustande kommen. Die Vermeidung von solchen Plänen kann hierbei nicht einmal, wie Abschnitt 7.2 zeigt, durch das zusätzliche Kreieren von weiteren *common table expressions* in eine adäquatere Form gebracht werden.

Ein Ausweg aus dieser etwas misslichen Lage ist eine Materialisierung von Zwischenergebnissen in Form von echten physischen Tabellen oder in Form einer *View*. Dadurch würde das Datenbanksystem gezwungen, das Zwischener-

gebnis direkt auszuwerten. Die Tabelle, oder die *View* könnte dann zusätzlich noch mit Indizes versehen werden, um die Laufzeiten zu verbessern. Ähnliche Ansätze wurden mit Erfolg bereits in [8] angewendet. Dieselbe Materialisierungsstrategie könnte auch bei der Umsetzung der Elementkonstruktoren verwendet werden. Dabei könnten wir wiederum die Indizes auf Zwischenergebnisse anlegen, um unsere Anfragen noch effizienter zu gestalten.

Anhang A

Beispielübersetzung

Beispielübersetzung der Anfrage Q08.xq des XMark-Benchmarks. Da unsere SQL-Generation keine Attribute benutzt, wurde Q08.xq gemäß Abschnitt 7.3 geändert.

```
let $auction := doc("auctionG.xml") return
for $p in $auction/site/people/person
let $a :=
  for $t in $auction/site/closed_auctions/closed_auction
  where $t/buyer/child::person/text() = $p/child::id/text()
  return $t
return
  <item>
    <person>{$p/name/text()}</person>{count($a)}
  </item>
```

Abbildung A.1: Anfrage Q08.xq

```
1 — !! START SCHEMA INFORMATION ** DO NOT EDIT THESE LINES !!
2 — document-relation: document
3 — document-pre: pre
4 — document-size: size
5 — document-level: level
6 — document-kind: kind
7 — document-prop: prop
8 — document-tag: tag
9 — result-relation: result
10 — result-pos_nat: pos_nat
11 — result-item_pre: item_pre
12 — !! END SCHEMA INFORMATION ** DO NOT EDIT THESE LINES !!
13 WITH
14 — ScjRel: scjoin(Frag, Rel)
15 — Binding due to: refctr > 1
16 a0000(item1_pre, iter_nat) AS
17 (SELECT DISTINCT c0002.pre, 1 AS iter_nat
```

ANHANG A. BEISPIELÜBERSETZUNG

```
18 FROM doc AS c0002 ,
19 doc AS c0001 ,
20 sysibm.sysdummy1 AS c0000
21 WHERE ((c0002.level = (c0001.level + 1)) AND (((c0001.pre + c0001.size) >=
22 c0002.pre) AND ((c0002.pre > c0001.pre) AND ((c0002.kind = 1) AND
23 ((c0002.prop = 'site') AND ((c0001.prop = 'auctionG.xml') AND
24 (c0001.kind = 6))))))))),
25
26 — Rel: ScjRel
27 — Binding due to: dirty node
28 a0001(item1_pre, iter_nat) AS
29 (SELECT DISTINCT c0006.pre, c0003.iter_nat
30 FROM doc AS c0006 ,
31 doc AS c0004 ,
32 doc AS c0005 ,
33 a0000 AS c0003
34 WHERE ((c0006.level = (c0004.level + 1)) AND (((c0004.pre + c0004.size) >=
35 c0006.pre) AND ((c0006.pre > c0004.pre) AND ((c0006.kind = 1) AND
36 ((c0006.prop = 'person') AND ((c0004.level = (c0005.level + 1)) AND
37 ((c0005.pre + c0005.size) >= c0004.pre) AND ((c0004.pre > c0005.pre)
38 AND ((c0004.kind = 1) AND ((c0004.prop = 'people') AND (c0005.pre =
39 c0003.item1_pre))))))))))))),
40
41 — Rel: rownum(Rel)
42 — Binding due to: dirty node
43 a0002(pos1_nat, item1_pre) AS
44 (SELECT ROWNUMBER () OVER (ORDER BY c0007.item1_pre ASC) AS pos1_nat ,
45 c0007.item1_pre
46 FROM a0001 AS c0007),
47
48 — Rel: number(Rel)
49 — Binding due to: refctr > 1 and dirty node
50 a0003(iter_nat, pos1_nat, item1_pre) AS
51 (SELECT ROWNUMBER () OVER () AS iter_nat, c0008.pos1_nat, c0008.item1_pre
52 FROM a0002 AS c0008),
53
54 — Rel: ScjRel
55 — Binding due to: dirty node
56 a0004(item_pre, iter1_nat) AS
57 (SELECT DISTINCT c0013.pre, c0010.iter_nat AS iter1_nat
58 FROM doc AS c0013 ,
59 doc AS c0011 ,
60 doc AS c0012 ,
61 a0003 AS c0010
62 WHERE ((c0013.level = (c0011.level + 1)) AND (((c0011.pre + c0011.size) >=
63 c0013.pre) AND ((c0013.pre > c0011.pre) AND ((c0013.kind = 3) AND
64 ((c0011.level = (c0012.level + 1)) AND (((c0012.pre + c0012.size) >=
65 c0011.pre) AND ((c0011.pre > c0012.pre) AND ((c0011.kind = 1) AND
66 ((c0011.prop = 'name') AND (c0012.pre = c0010.item1_pre))))))))))))),
67
68 — Rel: rownum(Rel)
69 — Binding due to: dirty node
70 a0005(pos_nat, item_pre, iter1_nat) AS
71 (SELECT ROWNUMBER () OVER
72 (PARTITION BY c0014.iter1_nat ORDER BY c0014.item_pre ASC) AS pos_nat ,
73 c0014.item_pre, c0014.iter1_nat
74 FROM a0004 AS c0014),
75
76 — Constr: merge_adjacent(Frag, Rel)
77 — Binding due to: refctr > 1
78 a0006(item_pre, pos_nat, iter1_nat) AS
79 (SELECT c0015.item_pre, c0015.pos_nat, c0015.iter1_nat
```

ANHANG A. BEISPIELÜBERSETZUNG

```

80     FROM a0005 AS c0015),
81
82 — Rel: project(Rel)
83 — Binding due to: refctr > 1
84 a0007(iter1_nat) AS
85   (SELECT c0018.iter_nat AS iter1_nat
86     FROM a0003 AS c0018),
87
88 — Rel: attach(Rel)
89 — Binding due to: dirty node
90 a0008(item_qname, iter1_nat) AS
91   (SELECT 'person' AS item_qname, c0020.iter1_nat
92     FROM a0007 AS c0020),
93
94 — Rel: roots_(Constr)
95 — Binding due to: dirty node
96 a0009(item_pre, pos_nat, iter1_nat) AS
97   (SELECT c0022.item_pre, c0022.pos_nat, c0022.iter1_nat
98     FROM a0006 AS c0022),
99
100 — =====
101 — = ELEMENT CONSTRUCTOR =
102 — =====
103 a0010(iter_nat, pre, size, level, kind, prop) AS
104   (SELECT c0025.iter_nat AS iter_nat,
105     (c0024.pre + ROWNUMBER () OVER
106     (ORDER BY iter_nat, c0025.pos_nat, c0025.pre)) AS pre,
107     c0025.size AS size, c0025.level AS level, c0025.kind AS kind,
108     c0025.prop AS prop
109   FROM ((SELECT c0030.iter1_nat AS iter_nat, 0 AS pos_nat, -2 AS pre,
110     COALESCE (SUM ((size + 1)), 0) AS size, 0 AS level, 1 AS kind,
111     c0030.item_qname AS prop
112     FROM (SELECT *
113       FROM a0009 AS c0026
114       INNER JOIN doc AS c0027
115         ON (c0026.item_pre = c0027.pre)) AS c0026
116     RIGHT OUTER JOIN a0008 AS c0030
117       ON (c0030.iter1_nat = c0026.iter1_nat)
118     GROUP BY c0030.item_qname, c0030.iter1_nat)
119   UNION ALL
120   (SELECT c0026.iter1_nat AS iter_nat, c0026.pos_nat AS pos_nat,
121     c0029.pre AS pre, c0029.size AS size,
122     ((c0029.level - c0028.level) + 1) AS level, c0029.kind AS kind,
123     c0029.prop AS prop
124   FROM doc AS c0029,
125     doc AS c0028,
126     a0009 AS c0026
127   WHERE ((c0028.pre = c0026.item_pre) AND ((c0029.pre >= c0028.pre) AND
128     ((c0028.pre + c0028.size) >= c0029.pre)))) AS c0025,
129   (SELECT MAX (pre) AS pre
130     FROM doc AS c0031) AS c0024),
131
132 — Constr: element(Frag, element_tag(Rel, Rel))
133 — Binding due to: refctr > 1 and dirty node
134 a0011(item_pre, iter1_nat) AS
135   (SELECT c0025.pre, c0025.iter_nat AS iter1_nat
136     FROM a0010 AS c0025
137   WHERE (c0025.level = 0)),
138
139 — Rel: ScjRel
140 — Binding due to: dirty node
141 a0012(item1_pre, iter_nat) AS

```

ANHANG A. BEISPIELÜBERSETZUNG

```

142 (SELECT DISTINCT c0037.pre, c0003.iter_nat
143     FROM doc AS c0037,
144          doc AS c0035,
145          doc AS c0036,
146          a0000 AS c0003
147     WHERE ((c0037.level = (c0035.level + 1)) AND (((c0035.pre + c0035.size) >=
148            c0037.pre) AND ((c0037.pre > c0035.pre) AND ((c0037.kind = 1) AND
149            ((c0037.prop = 'closed_auction') AND ((c0035.level = (c0036.level +
150            1)) AND (((c0036.pre + c0036.size) >= c0035.pre) AND ((c0035.pre >
151            c0036.pre) AND ((c0035.kind = 1) AND ((c0035.prop = 'closed_auctions')
152            AND (c0036.pre = c0003.item1_pre))))))))))))),
153
154 — Rel: number(Rel)
155 — Binding due to: dirty node
156 a0013(iter_nat, item1_pre) AS
157 (SELECT ROWNUMBER () OVER () AS iter_nat, c0038.item1_pre
158     FROM a0012 AS c0038),
159
160 — Rel: distinct(Rel)
161 — Binding due to: dirty node
162 a0014(iter1_nat, iter_nat) AS
163 (SELECT DISTINCT c0045.iter_nat AS iter1_nat, c0039.iter_nat
164     FROM doc AS c0049,
165          doc AS c0048,
166          doc AS c0046,
167          doc AS c0047,
168          a0003 AS c0045,
169          doc AS c0044,
170          doc AS c0043,
171          doc AS c0042,
172          doc AS c0040,
173          doc AS c0041,
174          a0013 AS c0039
175     WHERE ((c0044.prop = c0049.prop) AND ((c0049.kind = 3) AND ((c0048.pre =
176            c0049.pre) AND ((c0048.level = (c0046.level + 1)) AND (((c0046.pre +
177            c0046.size) >= c0048.pre) AND ((c0048.pre > c0046.pre) AND
178            ((c0048.kind = 3) AND ((c0046.level = (c0047.level + 1)) AND
179            (((c0047.pre + c0047.size) >= c0046.pre) AND ((c0046.pre > c0047.pre)
180            AND ((c0046.kind = 1) AND ((c0046.prop = 'id') AND ((c0047.pre =
181            c0045.item1_pre) AND ((c0044.kind = 3) AND ((c0043.pre = c0044.pre)
182            AND ((c0043.level = (c0042.level + 1)) AND (((c0042.pre + c0042.size)
183            >= c0043.pre) AND ((c0043.pre > c0042.pre) AND ((c0043.kind = 3) AND
184            ((c0042.level = (c0040.level + 1)) AND (((c0040.pre + c0040.size) >=
185            c0042.pre) AND ((c0042.pre > c0040.pre) AND ((c0042.kind = 1) AND
186            ((c0042.prop = 'person') AND ((c0040.level = (c0041.level + 1)) AND
187            (((c0041.pre + c0041.size) >= c0040.pre) AND ((c0040.pre > c0041.pre)
188            AND ((c0040.kind = 1) AND ((c0040.prop = 'buyer') AND (c0041.pre =
189            c0039.item1_pre))))))))))))))))))))))))),
190
191 — Rel: count(Rel)
192 — Binding due to: refctr > 1 and dirty node
193 a0015(iter1_nat, item1_int) AS
194 (SELECT c0050.iter1_nat, COUNT (*) AS item1_int
195     FROM a0014 AS c0050
196     GROUP BY c0050.iter1_nat),
197
198 — Rel: difference(Rel, Rel)
199 — Binding due to: kind != sql_select
200 a0016(iter1_nat) AS
201 ((SELECT c0053.iter1_nat
202     FROM a0007 AS c0053)
203 EXCEPT ALL

```

ANHANG A. BEISPIELÜBERSETZUNG

```

204 (SELECT c0052.iter1_nat
205 FROM a0015 AS c0052)),
206
207 — Rel: disjunction(Rel, Rel)
208 — Binding due to: kind != sql_select
209 a0017(iter1_nat, item1_int) AS
210 ((SELECT c0055.iter1_nat, c0055.item1_int
211 FROM a0015 AS c0055)
212 UNION ALL
213 (SELECT c0054.iter1_nat, 0 AS item1_int
214 FROM a0016 AS c0054)),
215
216 — =====
217 — = TEXTNODE CONSTRUCTOR =
218 — =====
219 a0018(pre, size, level, kind, prop, item2_str, iter1_nat, item1_int) AS
220 (SELECT (c0058.pre + ROWNUMBER () OVER ()) AS pre, 0 AS size, 0 AS level,
221 3 AS kind, CAST(c0056.item1_int AS CHAR(100)) AS prop,
222 CAST(c0056.item1_int AS CHAR(100)) AS item2_str,
223 c0056.iter1_nat AS iter1_nat, c0056.item1_int AS item1_int
224 FROM (SELECT MAX (pre) AS pre
225 FROM a0010 AS c0059) AS c0058,
226 a0017 AS c0056),
227
228 — Constr: textnode(Rel)
229 — Binding due to: refctr > 1
230 a0019(item_pre, item2_str, iter1_nat, item1_int) AS
231 (SELECT c0057.pre, c0057.item2_str, c0057.iter1_nat, c0057.item1_int
232 FROM a0018 AS c0057),
233
234 — Rel: disjunction(Rel, Rel)
235 — Binding due to: kind != sql_select
236 a0020(pos1_nat, item_pre, iter1_nat) AS
237 ((SELECT 1 AS pos1_nat, c0063.item_pre, c0063.iter1_nat
238 FROM a0011 AS c0063)
239 UNION ALL
240 (SELECT 2 AS pos1_nat, c0064.item_pre, c0064.iter1_nat
241 FROM a0019 AS c0064)),
242
243 — Rel: rownum(Rel)
244 — Binding due to: dirty node
245 a0021(pos_nat, pos1_nat, item_pre, iter1_nat) AS
246 (SELECT ROWNUMBER () OVER
247 (PARTITION BY c0065.iter1_nat ORDER BY c0065.pos1_nat ASC) AS pos_nat,
248 c0065.pos1_nat, c0065.item_pre, c0065.iter1_nat
249 FROM a0020 AS c0065),
250
251 — Constr: merge_adjacent(Frag, Rel)
252 — Binding due to: refctr > 1
253 a0022(item_pre, pos_nat, iter1_nat) AS
254 (SELECT c0066.item_pre, c0066.pos_nat, c0066.iter1_nat
255 FROM a0021 AS c0066),
256
257 — Rel: attach(Rel)
258 — Binding due to: dirty node
259 a0023(item_qname, iter1_nat) AS
260 (SELECT 'item' AS item_qname, c0069.iter1_nat
261 FROM a0007 AS c0069),
262
263 — Rel: roots_(Constr)
264 — Binding due to: dirty node
265 a0024(item_pre, pos_nat, iter1_nat) AS

```

ANHANG A. BEISPIELÜBERSETZUNG

```

266 (SELECT c0071.item_pre, c0071.pos_nat, c0071.iter1_nat
267 FROM a0022 AS c0071),
268
269 —=====
270 — = ELEMENT CONSTRUCTOR =
271 —=====
272 a0025(iter_nat, pre, size, level, kind, prop) AS
273 (SELECT c0074.iter_nat AS iter_nat,
274 (c0073.pre + ROWNUMBER () OVER
275 (ORDER BY iter_nat, c0074.pos_nat, c0074.pre)) AS pre,
276 c0074.size AS size, c0074.level AS level, c0074.kind AS kind,
277 c0074.prop AS prop
278 FROM ((SELECT c0079.iter1_nat AS iter_nat, 0 AS pos_nat, -2 AS pre,
279 COALESCE (SUM ((size + 1)), 0) AS size, 0 AS level, 1 AS kind,
280 c0079.item_qname AS prop
281 FROM ((SELECT *
282 FROM a0024 AS c0075
283 INNER JOIN (SELECT pre, size, level, kind, prop
284 FROM a0018 AS c0060) AS c0076
285 ON (c0075.item_pre = c0076.pre))
286 UNION ALL
287 (SELECT *
288 FROM a0024 AS c0075
289 INNER JOIN (SELECT pre, size, level, kind, prop
290 FROM a0010 AS c0032) AS c0076
291 ON (c0075.item_pre = c0076.pre))) AS c0075
292 RIGHT OUTER JOIN a0023 AS c0079
293 ON (c0079.iter1_nat = c0075.iter1_nat)
294 GROUP BY c0079.item_qname, c0079.iter1_nat)
295 UNION ALL
296 ((SELECT c0075.iter1_nat AS iter_nat, c0075.pos_nat AS pos_nat,
297 c0078.pre AS pre, c0078.size AS size,
298 ((c0078.level - c0077.level) + 1) AS level, c0078.kind AS kind,
299 c0078.prop AS prop
300 FROM (SELECT pre, size, level, kind, prop
301 FROM a0018 AS c0060) AS c0078,
302 (SELECT pre, size, level, kind, prop
303 FROM a0018 AS c0060) AS c0077,
304 a0024 AS c0075
305 WHERE ((c0077.pre = c0075.item_pre) AND ((c0078.pre >= c0077.pre) AND
306 ((c0077.pre + c0077.size) >= c0078.pre))))
307 UNION ALL
308 (SELECT c0075.iter1_nat AS iter_nat, c0075.pos_nat AS pos_nat,
309 c0078.pre AS pre, c0078.size AS size,
310 ((c0078.level - c0077.level) + 1) AS level, c0078.kind AS kind,
311 c0078.prop AS prop
312 FROM (SELECT pre, size, level, kind, prop
313 FROM a0010 AS c0032) AS c0078,
314 (SELECT pre, size, level, kind, prop
315 FROM a0010 AS c0032) AS c0077,
316 a0024 AS c0075
317 WHERE ((c0077.pre = c0075.item_pre) AND ((c0078.pre >= c0077.pre) AND
318 ((c0077.pre + c0077.size) >= c0078.pre)))) AS c0074,
319 (SELECT MAX (pre) AS pre
320 FROM (SELECT pre, size, level, kind, prop
321 FROM a0018 AS c0060) AS c0080) AS c0073),
322
323 — Constr: element(Frag, element_tag(Rel, Rel))
324 — Binding due to: refctr > 1 and dirty node
325 a0026(item_pre, iter1_nat) AS
326 (SELECT c0074.pre, c0074.iter1_nat AS iter1_nat
327 FROM a0025 AS c0074

```

```

328     WHERE (c0074.level = 0)),
329
330 — Rel: rownum(Rel)
331 — Binding due to: dirty node
332 a0027(pos_nat, item_pre, pos1_nat) AS
333 (SELECT ROWNUMBER () OVER (ORDER BY c0085.pos1_nat ASC) AS pos_nat,
334     c0084.item_pre, c0085.pos1_nat
335     FROM a0026 AS c0084,
336     a0003 AS c0085
337     WHERE (c0085.iter_nat = c0084.iter1_nat)),
338
339 — Rel: project(Rel)
340 — Binding due to: dirty node
341 a0028(pos_nat, item_pre) AS
342 (SELECT c0086.pos_nat, c0086.item_pre
343     FROM a0027 AS c0086),
344
345 — =====
346 — = RESULT RELATIONS =
347 — =====
348 result(pos_nat, item_pre) AS
349 (SELECT c0087.pos_nat, c0087.item_pre
350     FROM a0028 AS c0087),
351
352 document(pre, size, level, kind, prop, tag) AS
353 (SELECT pre, size, level, kind, prop, prop
354     FROM (SELECT pre, size, level, kind, prop
355           FROM a0025 AS c0081) AS c0088)

```


Anhang B

Beigelegte CD

Datei	Zweck
<code>/thesis/thesis.pdf</code>	Die Diplomarbeit als PDF
<code>/experiments/documents/</code>	Die XML-Dokumente in verschiedenen Größen
<code>/experiments/tables/</code>	Die enkodierten XML-Dokumente in verschiedenen Größen
<code>/experiments/xqueries</code>	Die modifizierten XMark-Queries
<code>/experiments/sqlqueries</code>	Experimente mit den XMark-Queries auf Dokumenten verschiedener Größe
<code>/experiments/scjtest</code>	Experimente mit und ohne Bündelung von <i>Path-Step</i> -Operatoren
<code>/experiments/q08_G_test</code>	Experiment mit Q08.xq
<code>/cvs/compiler/include</code>	Header Dateien
<code>./sql.h</code>	Strukturen und Definition für die SQL-Knotenkonstruktion
<code>./sql_mnemonic.h</code>	Abkürzungen für die bessere Handhabung von Funktionen
<code>./sqlprint.h</code>	Definitionen für Funktionen zur Ausgabe des SQL-Codes
<code>./prettysql.h</code>	Definitionen für das Prettyprinting des SQL-Codes
<code>/cvs/compiler/debug</code>	
<code>./prettysql.c</code>	Funktionen für das Prettyprinting von SQL.
<code>/cvs/compiler/sql</code>	Implementierung der konkreten Funktionen
<code>./sql.c</code>	Hilfsfunktionen für die SQL-Knotenkonstruktion
<code>./sqlprint.c</code>	Funktionen für die Ausgabe des SQL-Codes
<code>./lalg2sql.brg</code>	Transformation der relationalen Algebra nach SQL

Außerdem kann man den aktuellen Quellcode von Pathfinder stets über <http://pathfinder-xquery.org> beziehen.

Literaturverzeichnis

- [1] A. V. Aho, M. Ganapathi, and S. W. K. Tjang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, 1989.
- [2] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, IL, USA, June 2006.
- [3] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a Simple, Efficient Code Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.
- [4] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG – Fast Optimal Instruction Selection and Tree Parsing. Technical Report CS-TR-1991-1066, AT&T, 1991.
- [5] G. Graefe. Sorting and Indexing with Partitioned B-Trees. In *Proceedings of the 1st Int’l Conference on Innovative Data Systems (CIDR)*, Asilomar, CA, USA, June 2003.
- [6] T. Grust. Accelerating XPath Location Steps. In *Proceedings of the 21 International ACM SIGMOD Conference on Management of Data, Madison, Wisconsin, USA*, pages 109–120, June 2002.
- [7] T. Grust. Purely Relational FLOWRS. In *Proceedings of the ACM SIGMOD/PODS 2nd International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P 2005)*, Baltimore, MD, USA, June 2005.
- [8] T. Grust, M. Mayr, J. Teubner, and J. Rittinger. A SQL:1999 Code Generator for the Pathfinder XQuery Compiler. In *Proceedings of the 26th ACM SIGMOD Int’l Conference on Management of Data (SIGMOD)*, Beijing, China, June 2007.

- [9] T. Grust, J. Rittinger, and J. Teubner. Why Off-The-Shelf RDBMS are Better at XPath Than You Might Expect. In *Proceedings of the 2007 SIGMOD International Conference on Management of Data*, June 2007.
- [10] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. Technical report, Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004.
- [11] T. Grust and J. Teubner. Relational Algebra: Mother Tongue – XQuery: Fluent. Technical report, TDM’04, the first Twente Data Management Workshop on XML Databases and Information Retrieval, 2004.
- [12] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, June 2001.
- [13] IBM Staff. *SQL Reference - Version 7*. International Business Machines Corporation, 2000.
- [14] J. Teubner. Pathfinder: XQuery Compilation Techniques for Relational Database Targets, 2006.

Abbildungsverzeichnis

1.1	Architektur von <i>Pathfinder</i>	2
1.2	Architektur von <i>Pathfinder</i> (detailliert).	5
3.1	Die Syntax einer CASE-Anweisung	17
4.1	Jeder Operator wird an eine Referenz gebunden.	21
4.2	Relationaler Plan	22
4.3	Datenstruktur zum Aufsammeln von Ausdrücken	24
4.4	Relationale Enkodierung	26
4.5	Relationale Enkodierung von XML-Dokumenten	27
4.6	Notation zur Darstellung einer Inferenzregel	28
4.7	Anwendung des fun_1to1-Operators	32
4.8	Anwendung des num_gt-Operators	33
4.9	Dokument <code>recipes.xml</code>	35
4.10	Prädikat für den Achsentest	36
4.11	Prädikat für den Knotentest	36
5.1	Mehrere <i>Path-Step</i> -Operatoren direkt hintereinander	44
5.2	Transformation des <i>Path-Step</i> -Operators zur Ausnutzung von Index-Strukturen	45
5.3	Elementkonstruktion mit mehreren Fragmenten	46
6.1	Die zentrale Datenstruktur der SQL-Generierung	48
6.2	Baumdarstellung einer Subtraktion.	50
6.3	Algebra Annotationen (jede Datenstruktur ist mit dem ent- sprechenden Äquivalent aus den Inferenzregeln gekennzeich- net). \mathcal{M} ist in dieser Abbildung nicht sichtbar, da es, im Zuge des <i>Lazy Bindings</i> , Schritt für Schritt aufgebaut wird.	51
6.4	Burg-Notation für die Zuordnung von Identifiern	52
6.5	Burg-Notation für Grammatik-Regeln	53
6.6	Sequenz von Pfadausdrücken	53

7.1	Beispiel eines von <code>xmlgen</code> generierten XML-Dokuments	56
7.2	Mögliche Binärbäume mit vier Blättern	58
7.3	Abfrage <code>scj1.xq</code>	58
7.4	Abfrage <code>scj12.xq</code>	59
7.5	Auswertungspläne	61
7.6	Vergleich der Laufzeiten	62
7.7	XMark-Benchmark Anfragen auf einer 10 MB Relation mit und ohne Indizes	62
7.8	XMark-Benchmark Anfragen auf Dokumenten verschiedener Grö- ße	65
7.9	XMark-Benchmark Anfragen auf einem 112.1 MB Dokument	66
8.1	Anfrage mit Angabe einer Selektivität	68
A.1	Anfrage <code>Q08.xq</code>	70

Tabellenverzeichnis

2.1	Operatoren der relationalen Algebra	8
2.2	XPath-Achsen	13
2.3	Knotentest	13
3.1	Anwendung des ROWNUMBER-Operators	16
3.2	Anwendung der CASE-Anweisung	17
3.3	Anwendung der CASE-Anweisung	18
4.1	Das Ergebnis des Lokalisierungspfads	39
4.2	Ergebnis von <i>newroots</i>	40
4.3	Ergebnis von <i>subcopy</i>	40
4.4	Ergebnisse von <i>newelems</i>	40
7.1	Tabelle mit einigen Werten für C_n	58
7.2	Anfragen mit Bündelung	59
7.3	Anfragen ohne Bündelung	60
7.4	Zeiten der einzelnen <code>common table expressions</code> von Q08.xq .	63
7.5	Anzahl der Knoten, die von den einzelnen <code>common table expressions</code> zurückgeliefert werden (insgesamt befinden sich 5, 241, 961 Kno- ten in der Tabelle)	64

Index

- Übersetzung, 19
 - Attach, 30
 - Bindung, 20
 - Differenz, 30
 - Document-Relation, 25
 - Elementkonstruktion, 36
 - Fragment, 34
 - fun_1to1, 32
 - Index, 44
 - Lazy Binding, 21
 - lit_tbl, 29
 - num_gt, 32
 - Number, 33
 - Pfadausdrücke, 34
 - Projektion, 31
 - Relationale Enkodierung, 26
 - Result-Relation, 25
 - Rownumber, 33
 - Selektion, 31
 - Vereinigung, 30
- SQL
 - CASE, 16
 - WITH-Statement, 14
 - Common Table Expressions, 14
 - Distinct, 17
 - OLAP, 14
 - Rownumber, 15
- Burg, 52
- XMark-Benchmark, 55
- db2advis, 57
- pre/size/level-Enkodierung, 25
- runstats, 57
- Abarbeitungsplan, 7
- Algebra Annotationen, 50
- Beispielübersetzung, 70
- bushy trees, 60
- Catalan-Zahlen, 58
- Core Language, 3
- Equi-Join, 10
- Experimente, 55
 - Auswertungsstrategie, 57
 - Lokalisierungspfad, 57
 - Relationen, 65
 - Vorbereitungen, 55
- GAG, 4, 19
- gerichteter Graph, 19
- Graphen-Traversierung, 19
- Implementierung
 - Bündelung, 53
 - Baumstruktur, 48
 - Lazy Binding, 50
 - Mustererkennung, 52
- Kompilationsphasen, 3
- left-deep-trees, 60
- Lexikalische Analyse, 3
- Lokalisierungspfad, 11
- Materialisierung, 60, 68
- Parsen, 3
- Pathfinder, 1
- Quellsprache, 4

Relationale Algebra, 7
 Differenz, 9
 Distinct, 10
 Equi-Join, 10
 fun_1to1, 11
 Kartesisches Produkt, 10
 lit_tbl, 8
 Operatoren, 8
 Path-Step, 12
 Pfadausdrücke, 11
 Rownumber, 11
 Selektion ($\sigma_p(R)$), 10
 Semi-Join, 10
 Vereinigung, 9

Schema, 7
Selektivität, 67

Theta-Join, 10
Typkompatibilität, 7

XML, 1

Zielsprache SQL, 14