Entkopplung von Kernelementen der Anfrageverarbeitung*

Sebastian Bächle
Databases and Information Systems Group
University of Kaiserslautern, Germany
baechle@cs.uni-kl.de

Abstract: Diese Arbeit befasst sich mit der Konzeption, Optimierung und Realisierung einer erweiterbaren deklarativen Anfragesprache zur effizienten Verarbeitung strukturierter und semi-strukturierter Daten. Zur Vereinfachung von Optimierungs- und Übersetzungsprozessen strebt sie eine strikte Trennung von logischen mengenorientierten Konzepten und physischen Aspekten an. Die für diese Trennung sorgfältig abgestimmte Compiler-Architektur kann daher als effiziente Basis für die Implementierung, Erweiterung und Portierung von Anfragesystemen für unterschiedliche Sprachen, Datenmodelle und Systemarchitekturen herangezogen werden. Desweiteren wird ein neuartiges, auf das Operatorkonzept der Sprache abgestimmtes Ausführungsmodell vorgestellt, mit dem Anfragen zur Laufzeit automatisch parallelisiert werden können.

1 Übersicht

Die Bedeutung deklarativer Anfragesprachen als effiziente und produktivitätssteigernde Abstraktion für die Verarbeitung großer Datenmengen nimmt stetig zu. Die mit ihnen erzielte Entkopplung von der physischen Datenrepräsentation ermöglicht es, komplexe Zugriffs- und Verarbeitungsroutinen isoliert von der Anwendung selbst zu optimieren und auszuführen. Die aktuelle Tendenz zur immer tiefgreifenderen Erweiterung und Spezialisierung von Systemen für neue Anwendungsgebiete und deren Datenmodelle wirkt allerdings wie eine Zentrifugalkraft auf diese bislang von klassischen Unternehmens- und Geschäftsanwendungen dominierte Disziplin. Getrieben durch die zusätzlichen Anforderungen schnell wachsender Datenmengen, neuer Hardware und der Forderung nach unmittelbarer Verfügbarkeit, werden immer mehr Systeme, Sprachen und APIs für spezielle Anwendungsszenarien zugeschnitten. Ohne das relationale Modell und SQL als gegebene Kernelemente müssen jedoch bewährte Konzepte und Algorithmen immer wieder aufwändig adaptiert, portiert und implementiert werden.

Gegenstand dieser Arbeit ist die Zusammenfassung der in [Bä12] entwickelten Konzeption und Realisierung einer abstrakten Verarbeitungsprache für strukturierte und semistrukturierte Daten. Diese deklarative Sprache vereint in sich die zentralen Strategien und Techniken aus dem Bereich der Anfrageoptimierung und -auswertung. Sie kann deshalb als solide Basis für die Realisierung neuer Anfragesprachen dienen. Der Vorteil dieses

*Englischer Titel der Dissertation: “Separating Key Concerns in Query Processing”

Ansatzes besteht in der Möglichkeit, die für ein konkretes Anwendungsszenario zugeschnittenen Abstraktionen für Daten und Operationen auf Standardkonzepte abzubilden, zu optimieren und letztendlich effizient auf der konkreten Zielplattform umzusetzen.

Um den vielfältigen Herausforderungen eine Datenverarbeitungsprache auf logischer und physischer Ebene Rechnung tragen zu können, ist es wichtig, deren Komplexität durch eine klare Trennung von Kernzielen beherrschbar zu machen. Am Ausgangspunkt steht daher die Besinnung auf ihre vier Kernaufgaben: das *Filtern*, *Transformieren*, *Kombinieren* und *Aggregieren* von großen Datenmengen. Im Fokus der Betrachtungen liegen hierbei drei zentrale Herausforderungen: die Optimierung mengenorientierter Aspekte, die effiziente Ausnutzung plattformspezifischer Gegebenheiten und eine parallele Verarbeitung.

Anfragen oder Skripte bestehen üblicherweise aus Kombinationen mengenorientierter Operationen, die in einen möglichst effizienten Ausführungsplan übersetzt werden sollen. Art und Reihenfolge der einzelnen Verarbeitungsschritte sind dabei häufig die kritischsten Aspekte, wobei gängige Optimierungen in erster Linie unabhängig von der konkreten Ausprägung der Daten sind. Beim Transformieren und Filtern von Daten ist es beispielsweise sinnnvoll, den Filterschritt als erstes durchzuführen. Konzeptionell spielt es dabei jedoch keine Rolle, ob die Daten als Tabellen homogen-strukturierter Tupel oder Kollektionen semi-strukturierter Dokumente vorliegen. Physische Aspekte, beispielsweise der effiziente Zugriff auf die einzelnen Felder eines Tupels oder das Auswerten von Pfadmustern auf hierarchischen Strukturen, können isoliert davon betrachtet und für die jeweilige Plattform optimiert werden. Die effiziente Parallelisierung einer Anfrage zur gezielten Ausnutzung moderner Mehrkern-Architekturen stellt schließlich eine besondere Herausforderung dar, da neben logischen auch physische Aspekte wie Scheduling und Datenlokalität beachtet werden müssen.

Das Ergebnis dieser Arbeit ist die in Abbildung 1 schematisch dargestellte Compiler-Architektur. Sie ermöglicht es, unterschiedlichste Frontend-Sprachen auf einheitliche Konzepte abzubilden, diese auf rein logischer und konkreter physischer Ebene zu optimieren und schließlich auf unterschiedlichen Speicherungsarchitekturen auszuwerten. Die Laufzeitumgebung baut dazu auf einem Wrapper-Ansatz auf, der zur Effizienzsteigerung teilweise oder vollständig durch native Operationen ergänzt werden kann.

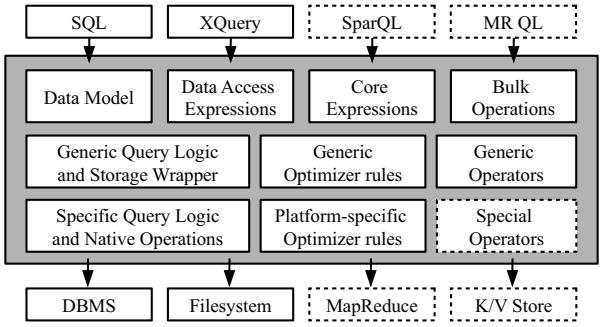


Abbildung 1: Schematischer Aufbau der Compiler Architektur.

2 Anatomie der Anfragesprache

Die konzipierte Anfragesprache besteht aus vier Kernelementen: einem abstrakten Modell zur Repräsentation von Daten, Operationen zum Erzeugen, Komponieren und Dekomponieren von Daten, elementaren Grundoperationen für Arithmetik, String-Manipulationen, Verzweigungen etc. sowie einer Menge von Operatoren zur Modellierung mengenorientierter Verarbeitungskonzepte.

2.1 Datenmodell

Anfragesprachen betrachten Datenbestände üblicherweise als Sammlung von mehr oder weniger homogenen, flach oder hierarchisch aufgebauten Datenelementen. Für einen portablen Übersetzungsprozess müssen unterschiedliche Datenmodelle aber auf einheitliche Strukturen zurückgeführt werden. Zur Abbildung der Datenarten und -strukturen verschiedenster Frontend-Sprachen auf eine abstrakte, einheitliche Ebene genügt bereits das in Abbildung 2 gezeigte minimalistische Grundmodell. Benötigt werden lediglich einige primitive Datentypen zur Repräsentation von Zahlen, Zeichenketten etc. sowie die zwei Kompositionstypen *Array* und *Map*. Ein Array ist eine geordnete Liste von Werten; eine Map ist eine geordnete Menge von Schlüssel-Wert-Paaren. Optional können zu diesem Grundmodell auch noch Funktoren hinzugenommen werden, was die Realisierung von mächtigeren Sprachen wie das in dieser Arbeit als Beispiel verwendete XQuery 3.0 ermöglicht.

Kompositionstypen können beliebig geschachtelt werden, um konkrete Datenstrukturen nachzubilden. Aus Effizienzgründen besitzen Werte jedoch keine Identität, was impliziert, dass Kompositionen strikt hierarchisch aufgebaut sind und auch keine direkten Verweise zwischen einzelnen Datenelementen möglich sind. Höherwertige, für Anwendungen benötigte Konzepte wie Identität, Beziehungen, Typen, Schemata und sonstige Invarianten werden durch verschiedene Mechanismen der jeweiligen Frontend-Sprache selbst realisiert. Ähnliches gilt auch für die andere Richtung. Wenn für Anfragen oder Datenmodelle bestimmte Aspekte des Grundmodells wie beispielsweise die Ordnung von Map-Werten keine Rolle spielen, kann deren Einhaltung aus Effizienzgründen ignoriert werden.

Wie in Abbildung 2 gezeigt, können relationale Daten recht einfach abgebildet werden. Ein Tupel entspricht einem Map-Wert, der Spaltennamen auf Werte abbildet. Tabellen sind Arrays von homogenen Tupeln. Auch das wesentlich komplexere Datenmodell von XQuery lässt sich auf ähnlich intuitive Weise auf Map- und Array-Werte zurückführen. Die Abbildung solcher Datenmodelle erfolgt jedoch teilweise über mehrere Kompositionsstufen, die zusätzlich bestimmten Invarianten unterliegen.

2.2 Datenzugriffe und Anfragelogik

Den Zugriff und das Erzeugen von Daten übernehmen dedizierte Konstruktoren und Zugriffsoperationen. Durch diese werden die jeweiligen Invarianten eines Datenmodells si-

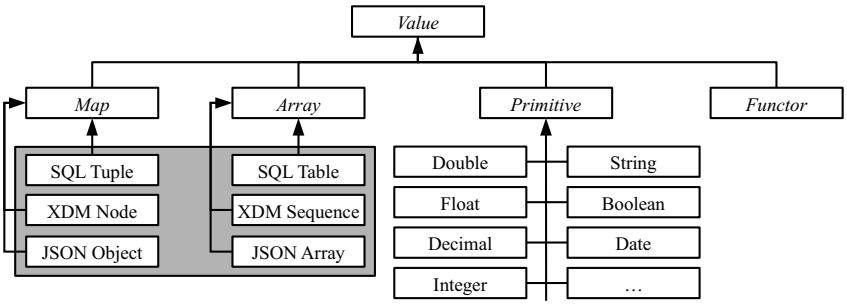


Abbildung 2: Aufbau des elementaren Datenmodells

chergestellt. Die Kapselung von logischen und letztlich auch physischen Datenoperationen in Form eigenständiger Ausdrücke erlaubt es, das Abstraktionsniveau und die Granularität der Frontend-Sprache direkt und sehr effizient auf die Fähigkeiten des Speicherungssystems abzubilden. In XQuery, zum Beispiel, kann eine Zugriffsoperation wie der Pfadschritt `./a//b/c` entweder direkt in Navigationsschritte im XML-Baum oder in eine noch effizientere Scan-Operation übersetzt werden, ohne dabei die konzeptionelle Sicht auf XML-Knoten als Kompositionen aus Map- und Array-Werten einnehmen zu müssen.

Die anderen Sprachbausteine bilden das Komplement zu Konstruktoren und Zugriffsoperationen, um sinnvoll Anfragen formulieren zu können. Sie implementieren Standardkonzepte wie Arithmetik, Boolesche Logik oder Funktionsaufrufe. Sie sind ebenfalls als Ausdrücke gekapselt und können daher bei Bedarf an sprach- oder systemspezifische Eigenschaften angepasst werden, ohne den Übersetzungsprozess an sich zu beeinflussen.

2.3 Operatoren

Mengenwertige Operationen werden als Funktionen über 2-dimensionale Arrays modelliert. Das macht es leichter, sie mit bewährten Verarbeitungskonzepten und Algorithmen relationaler Systeme umzusetzen. Jede dieser Funktionen, die aufgrund ihrer besonderen Rolle *Operatoren* genannt werden, akzeptiert ein 2-dimensionales Array als Eingaberelation und liefert als Ergebnis wieder ein 2-dimensionales Array. Operatoren lassen sich dadurch zu effizienten Verarbeitungspipelines komponieren. Tabelle 1 zeigt eine Auswahl der wichtigsten Operatoren. Beispiele zu einigen davon finden sich in Abbildung 3.

Operatoren werden in drei Gruppen unterschieden. Die Operatoren der ersten Gruppe verarbeiten die Tupel der Eingaberelation einzeln nacheinander. Der wichtigste Operator dieser Kategorie ist `ForEach`. Seine Funktionsweise entspricht einer klassischen *map*-Funktion mit Nested-Loops-Semantik. Die für jedes Eingabetupel ausgewertete Argumentfunktion *bind* liefert als Ergebnis selbst wieder eine Relation, deren Tupel im Verbund mit dem zugehörigen Eingabetupel zur Ausgaberelation hinzugefügt werden. `ForEach` ermöglicht so das Einspeisen von Daten in eine Pipeline und eignet sich unter ande-

Operator	Parameter		Rückgabetyp
`ForEach`	`in`	`[[T]]`	`[[T,S]]`
	`bind`	`[T]→[[S]]`	
`Project`	`in`	`[[T]]`	`[[S]]`
	`proj`	`[T]→[S]`	
`Select`	`in`	`[[T]]`	`[[T]]`
	`pred`	`[T]→bool`	
`OrderBy`	`in`	`[[T]]`	`[[T]]`
	`cmp`	`[T]×[T]→bool`	
`GroupBy`	`in`	`[[T]]`	`[[S]]`
	`grp`	`[T]→int`	
	`agg`	`[[T]]→[S]`	
`Count`	`in`	`[[T]]`	`[[T,int]]`
`Join / LeftJoin`	`in`	`[[T]]`	`[[T,S,R]]`
	`left`	`[T]→[[T,S]]`	
	`right`	`[T]→[[T,R]]`	
	`pred`	`[T,S]×[T,R]→bool`	

Tabelle 1: Auswahl der unterstützten Operatoren.

rem als Abstraktion für Scan-Operationen. Die Operatoren `Project` und `Select` sind äquivalent zu den entsprechenden Operationen im relationalen Fall.

Die zweite Gruppe von Operatoren verarbeitet Eingaberelationen als Ganzes. Zu ihr gehören die bekannten Sortier- und Agreggationsoperatoren `OrderBy` und `GroupBy` sowie der Operator `Count`, der die Tupel der Eingaberelation durchnummeriert.

Die Operatoren der dritten Gruppe kombinieren mehrere Relationen zu einer Ausgaberelation. Als Vertreter dieser Gruppe sind in Tabelle 1 die beiden Verbundoperationen `Join` und `LeftJoin` aufgeführt. Weitere Beispiele sind `Union` und `Intersect`. Wie alle Operatoren akzeptieren auch sie nur eine Eingaberelation. Die eigentliche Kombinationsoperation erfolgt nämlich für die zwei Relationen, die von den zwei Argumentfunktionen `left` und `right` für jedes Eingabetupel ausgegeben werden. Die Ausgabe des Operators selbst besteht aus der Konkatenation aller Teilergebnisse. Diese Funktionsweise stellt eine Verallgemeinerung entsprechender relationaler Operationen dar, wobei deren Algorithmen dennoch für die Realisierung herangezogen werden können. Der Grund für diese andere Struktur ergibt sich im Folgenden aus der Diskussion des Optimierungprozesses.

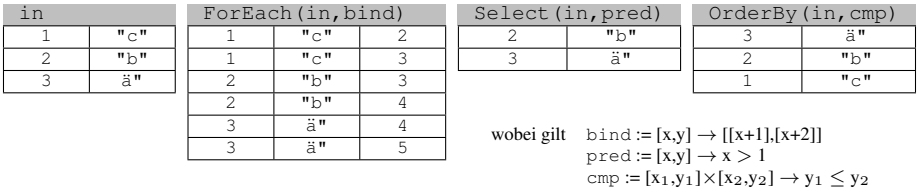


Abbildung 3: Beispiele für die Operatoren `ForEach`, `Select` und `OrderBy`.

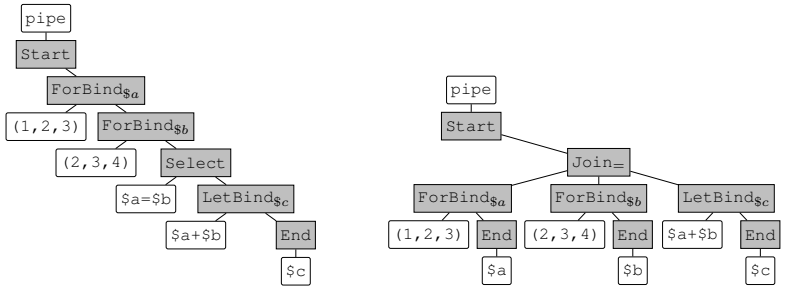
Um das Komponieren von Operatoren zu erleichtern und Hilfsfunktionen mit fixen Referenzen auf Tupel-Positionen wie in Abbildung 3 zu umgehen, werden die Spalten mit Hilfe von λ-Abstraktion wie in relationalen Umgebungen über explizite Spaltennamen

adressierbar gemacht. Abhängigkeiten zwischen Operatoren lassen sich so leichter erkennen, was insbesondere bei der Optimierung von Pipelines von Vorteil ist.

3 Übersetzung und Optimierung

Im Gegensatz zu traditionellen Ansätzen, die Anfragesprachen auf einer Algebra aufbauen, wird die Semantik einer Anfrage und damit auch der Suchraum für den Optimierer alleine über funktionale Konzepte definiert. Jede Anfrage wird durch einen Ausdrucksbaum repräsentiert, dessen Terme rekursiv ausgewertet werden. Da funktionale Werte stets unveränderlich sind und Anfragen im allgemeinen Fall ohnehin frei von Seiteneffekten sind, kann die Auswertung *eager* oder *lazy* erfolgen und beliebig parallelisiert werden. Abbildung 4(a) illustriert den prinzipiellen Aufbau einer Anfrage am Beispiel der einfachen XQuery-Anfrage:

```
for $a in (1,2,3)
return for $b in (2,3,4)
       where $a=$b
       let $c:= $a+$b
       return $c
```



(a) Einfache Pipeline.

(b) Pipeline mit Join-Operator.

Abbildung 4: Repräsentation und Optimierung eines XQuery-FLWOR-Ausdrucks.

Zur Übersetzungszeit wird diese Anfrage als `pipe`-Ausdruck dargestellt, der bei der Auswertung den aktuellen Kontext, in diesem Beispiel das leere Tupel `[]`, über den künstlichen Operator `Start` in die Pipeline einbringt. Die Variablenbindungen der beiden geschachtelten `for`-Schleifen werden in der Pipeline über Varianten des `ForEach`-Operators realisiert. Die beiden `ForBind`-Operatoren erzeugen für jedes Eingabetupel das Kartesische Produkt mit den Elementen ihrer jeweiligen Binding-Sequenz. Der `Select`-Operator filtert den so erzeugten Tupelstrom für das Prädikat `$a=$b`. Der `LetBind`-Operator bindet die Summe `$a+$b` schließlich als zusätzliche Variable `$c`. Der künstliche Operator `End` bildet das Ergebnis der Anfrage, in dem er die Teilergebnisse aller "Iterationen" zur Sequenz `(4,6)` konkateniert, die als Ergebnis des `pipe`-Ausdrucks zurückgeliefert wird. Die Ausgaben der einzelnen Operatoren sind Abbildung 5 nochmals dargestellt.

Die Auswertung eines Ausdrucks kann alleine von eventuell vorhandenen Variablenreferenzen abhängen. Diese bilden die Verknüpfung zwischen mengenorientierten Operatoren und Ausdrücken. Ein Ausdruck wird jeweils für eine bestimmte Variablenbelegung, den bereits erwähnten Kontext, ausgewertet. Dieser wird wiederum durch die von Operatoren produzierten und konsumierten Tupel repräsentiert, womit ein Pipeline-Tupel vergleichbar ist zu einem *stack frame* in einer prozeduralen Sprache.

Im Unterschied zu traditionellen Ansätzen, die Anfragen in *bottom-up*-Darstellung verarbeiten, werden Anfragen, wie gezeigt, in *top-down*-Form verwaltet. Die Operatoren werden dazu im *continuation-passing style* [SJ75] definiert, so dass der Datenfluss innerhalb einer Pipeline von oben nach unten verläuft. Diese Modellierung vereinfacht den Optimierungs- und Übersetzungsprozess erheblich, da sich die Sichtbarkeit von Variablen ganz natürlich mit der hierarchischen Struktur der Anfrage deckt.

ForBind$_{\$a}$
$a
1
2
3

ForBind$_{\$b}$	
$a	$b
1	2
1	3
...	...
3	4

Select	
$a	$b
2	2
3	3

LetBind$_{\$c}$		
$a	$b	$c
2	2	4
3	3	6

End			
$a	$b	$c	
2	2	4	4
3	3	6	6

Abbildung 5: Ausgaberelationen der Operatoren aus Abbildung 4(a).

3.1 Anfrageoptimierung

Die am Beispiel gezeigte Abbildung auf die interne Repräsentation ist einfach auf gängige Anfragekonstrukte verschiedenster Frontend-Sprachen übertragbar: mengenorientierte Passagen werden auf Operatoren abgebildet; die verbleibenden sprach- und modellspezifischen Artefakte werden in Ausdrücken gekapselt. Je besser es gelingt, große Bestandteile einer Anfrage zu einer Pipeline zusammenzufassen, desto effektiver sind Optimierungen.

Die natürliche Trennung von mengenorientierten und physischen Aspekten im Ausdrucksbaum erleichtert die Analyse und Optimierung von Datenflüssen. Hierbei werden die gleichen Ziele verfolgt wie im relationalen Fall. Filter sollten möglichst früh in der Pipeline ausgewertet werden, um die Anzahl der zu verarbeitenden Tupel schnell zu verringern. Für die durch `ForEach` häufig auftretenden Kartesischen Produkte gilt es, diese nach Möglichkeit wie in Abbildung 4(b) durch effizientere Join-Operatoren zu ersetzen.

Die formalen Grundlagen solcher Optimierungen sind in algebraischen Ansätzen direkt ersichtlich. Zur Herleitung entsprechender Transformationsregeln für das funktional aufgebaute Operatorenmodell kann man stattdessen auf Monaden zurückgreifen. Diese bilden ein auf den sogenannten Monadischen Gesetzen aufbauendes, formales Typkonstrukt und werden primär in funktionalen Programmiersprachen verwendet, um "nicht-funktionale" Aspekte wie sequentielle Abläufe, Indeterminismus oder Seiteneffekte zu modellieren. Die Möglichkeit, sie auch zur Spezifikation von Anfragen und Transformationsregeln zu verwenden, wurde bereits vor über 20 Jahren erkannt [JW07].

Da sich Operatoren auf Monadische Konstrukte zurückführen lassen, ist es möglich, über diese formal korrekte Transformationen für Operatorkompositionen herzuleiten [Gru99]. Monaden dienen somit als einer Algebra ebenbürtiges theoretisches Fundament, über das ein Großteil etablierter Optimierungstechniken, etwa für Joins oder Aggregate, als generische Pipeline-Optimierungen im Compiler verankert werden können.

Das Optimieren auf physischer Ebene wird durch die Kapselung von Datenzugriffen in dedizierten Ausdrücken vereinfacht. Im Normalfall erfolgen die Zugriffe über eine Wrapper-Architektur. Im Falle einer naïven Übersetzung von XQuery bedeutet das, dass über eine Schnittstelle auf den einzelnen XML-Knoten navigiert wird, ohne zu wissen, ob es sich dabei um im Hauptspeicher gehaltende Baumstrukturen oder serialisierte Instanzen auf dem Externspeicher handelt. Der Spielraum für Optimierungen ist entsprechend groß. Durch die Vereinfachung von Zugriffssequenzen zu gröberen Operationsgranulaten oder durch die Gruppierung von Zugriffen kann der relative Overhead pro Zugriff deutlich reduziert werden. Durch das Umgehen aufwändiger Speziallogik für Standardfälle können zudem die durchschnittlichen Kosten pro Zugriff reduziert werden. Desweiteren arbeiten alternative Zugriffswege, wie zum Beispiel Indexe, in vielen Situationen wesentlich effizienter. Direkte Zugriffe auf die Speicherungsstrukturen und andere Optimierungen können auf vielfältige Weise realisiert werden. Das Spektrum reicht von spezialisierten Ausdruckstypen, die native Erweiterungen von Wrappern nutzen oder die Wrapper-Schicht vollständig umgehen, bis hin zu umfangreichen Transformationen von korrelierten Zugriffen innerhalb einer Pipeline.

4 Parallele Verarbeitung

Der Datenfluss in einer Pipeline ähnelt stark dem Funktionsprinzip geschachtelter Schleifen. Wie in Abbildung 6 gezeigt, entspricht die Ausgabereihenfolge der tupelweise arbeitenden Operatoren der ersten Gruppe der Abfolge der Variablenbelegungen in geschachtelten Schleifen. Die Operatoren der zweiten Gruppe, die *pipeline blocker*, führen die aus den einzelnen Schleifendurchläufen resultierenden Ergebnisse wieder zusammen.

Die Iterationen jeder Schleife sind unabhängig voneinander. Da also keine Datenabhängigkeiten zwischen ihnen bestehen, können große Teile einer Anfrage durch die horizontale und vertikale Aufteilung der einzelnen Schleifendurchläufe in pipeline- und datenparallele Operationen aufgeteilt werden. Da sich die parallelisierbare Baumstruktur jedoch erst zur Laufzeit entwickelt und die Bestimmung einer effizienten Aufteilung a priori äußerst schwierig ist, kommt ein neuartiges Operatormodell zum Einsatz, welches diese Aufteilung dynamisch zur Laufzeit vornimmt. Das Modell basiert auf dem *fork/join*-Verfahren, einer sehr effektiven, nach dem divide-and-conquer-Prinzip arbeitenden Parallelisierungstechnik [Lea00]. Die Idee besteht darin, eine große Eingabe solange rekursiv in zwei unabhängige Teile aufzuspalten (“fork”), bis diese klein genug sind, um effizient seriell verarbeitet zu werden. Die während der Aufteilungsphase erzeugten Teile, die nicht direkt weiterverarbeitet werden, können zwischenzeitlich von freien Prozessen parallel verarbeitet werden, bis der initiierende Prozess zurückkehrt (“join”). Durch die Verwendung eines leichtgewichtigen Schedulers [SLS06] lässt sich so ein hoher Parallelitätsgrad erzielen.

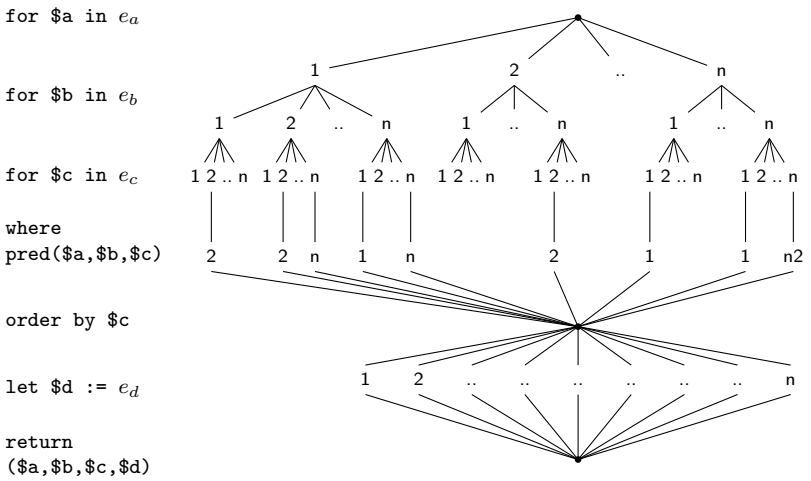
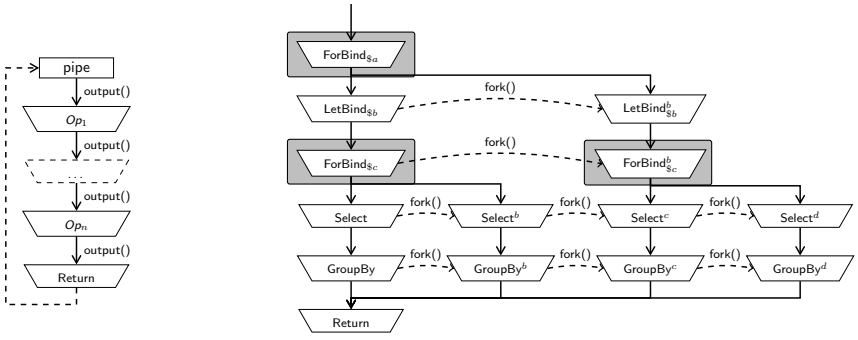


Abbildung 6: Iterationen eines XQuery-FLWOR-Ausdrucks.

Um dieses Prinzip in einer Pipeline anwenden zu können, müssen die Operatoren wie in Abbildung 7(a) *push*-basiert arbeiten. Das bedeutet, dass Operatoren im Gegensatz zum sonst üblichen *pull*-basierten Prinzip [Gra94], ihre Ergebnisse aktiv an den jeweils nachfolgenden Operator weiterreichen. Wenn ein `ForEach`-basierter Operator nun für ein Eingabetupel eine sehr große Binding-Sequenz verarbeiten muss, kann diese nach dem fork/join-Prinzip aufgeteilt und parallel verarbeitet werden. Damit die entstehenden Partitionen aber wirklich unabhängig voneinander verarbeitet werden können, muss der fork-Schritt auch auf Operatorebene erfolgen. So entwickelt sich die initial lineare Pipeline dynamisch zu einem Operatorgraphen wie in Abbildung 7(b). Der nächste blockierende Operator in der Pipeline führt die parallelen Pfade am Ende wieder zusammen.



(a) Lineare Pipeline. (b) Dynamischer Operatorgraph.

Abbildung 7: Push-basiertes Operatormodell mit Fork-Mechanismus.

5 Zusammenfassung

In diesem Rundgang wurden die tragenden Säulen der in [Bä12] entwickelten Konzepte und Techniken vorgestellt. Die abstrakte Anfragesprache setzt sich aus vier Bausteinen zusammen, mit denen man komplexe Datenmodelle auf die einheitlichen Kompositionstypen Map und Array zurückführen kann. Mengenorientierte und physische Aspekte werden getrennt voneinander modelliert, um sie einfacher und effektiver optimieren zu können. Transformationsregeln können dabei auf Monaden als zugrunde liegendes formales Konstrukt zurückgreifen. Für die dynamische Parallelisierung von Anfragen wurde ein push-basiertes Operatordesign entwickelt, das effiziente Partitionierungstechniken und Scheduling-Mechanismen aus dem HPC-Bereich für Datenverarbeitungsalgorithmen zugänglich macht. Eine empirische Evaluation der hier vorgestellten Aspekte findet sich ebenfalls in [Bä12] .

Literatur

[Bä12] Sebastian Bächle. *Separating Key Concerns in Query Processing – Set Orientation, Physical Data Independence, and Parallelism*. Dr. Hut Verlag, München, December 2012.

[Gra94] Goetz Graefe. Volcano: An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, Februar 1994.

[Gru99] Torsten Grust. Comprehending Queries. Bericht, 1999.

[JW07] Simon Peyton Jones und Philip Wadler. Comprehensive Comprehensions. In *Proc. SIGPLAN Workshop on Haskell*, Seiten 61–72, 2007.

[Lea00] Doug Lea. A Java Fork/Join Framework. In *Proceedings ACM Java Grande*, Seiten 36–43, 2000.

[SJ75] Gerald Jay Sussman und Guy L Steele Jr. Scheme: An Interpreter for Extended Lambda Calculus. In *MEMO 349, MIT AI LAB*, 1975.

[SLS06] William N. Scherer, III, Doug Lea und Michael L. Scott. Scalable Synchronous Queues. In *Proc. PPoPP*, Seiten 147–156, 2006.

Sebastian Bächle wurde am 11. November 1981 in Zweibrücken geboren. Nach dem Abitur 2001 trat er im Oktober 2002 ein Studium der Angewandten Informatik an der TU Kaiserslautern an. Im Anschluss an die abgeschlossene Diplomprüfung im September 2007 begann er seine Forschungsarbeit als wissenschaftlicher Mitarbeiter an der TU in der Arbeitsgruppe *Datenbanken und Informationssysteme* unter der Leitung von Prof. Dr.-Ing. Dr. h. c. Härder. Im Dezember 2012 schloss er dort seine Promotion über Anfrageverarbeitung für strukturierte und semi-strukturierte Daten ab. Seit Oktober 2012 arbeitet er bei der SAP AG in Walldorf.