Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik


Masterarbeit

# Parallelisierung in Query Compilern mit Lightweight-Modular-Staging

Autor:

Jan Düwel

20. Oktober 2015

Betreuer:

Prof. Dr. habil. Gunter Saake

M.Sc. David Broneske

M.Sc. Reimar Schröter

Institut für Technische und Betriebliche Informationssysteme

**Düwel, Jan:**
*Parallelisierung in Query Compilern mit Lightweight-Modular-Staging*
Masterarbeit, Otto-von-Guericke-Universität Magdeburg, 2015.

# Danksagung

Besonderer Dank geht an Herrn Prof. Dr. rer. nat. habil. Gunter Saake, der es mir ermöglichte meine Masterarbeit am Institut für Technische und Betriebliche Informationssysteme (ITI) zu verfassen. Des Weiteren möchte ich mich auch bei M.Sc. David Broneske und M.Sc. Reimar Schröter für die ausgezeichnete Betreuung, Verbesserungsvorschläge und die hilfreichen Diskussionen bedanken. Weiterer Dank geht auch an meine Freunde und meine Familie, die mich während der Masterarbeit moralisch unterstützt haben.

# Inhaltsangabe

Mehrkernprozessoren können das Antwortzeitverhalten bei der Anfrageverarbeitung in Datenbanken beschleunigen. In einigen Fällen gibt es nur eine geringe Beschleunigung oder sogar eine Verlangsamung der Anfrage. Aus diesem Grund wäre es wünschenswert, die Implementationen der Algorithmen anfragespezifisch tauschen zu können. Dabei ergeben die möglichen Ausprägungen der Parameter, wie Hardware oder Parallelisierungsgrad der Implementation, vielfältige Kombinationsmöglichkeiten. Diese große Anzahl an ähnlichen Varianten sind schwer händisch zu verwalten. Die automatische Generierung von Varianten aus einzelnen Bausteinen stellt eine Lösung dieser Problems dar.

In dieser Arbeit wurde die dynamische Generierung von parallelisierten Code für Datenbankanfragen in parallelen Datenbankmanagementsystemen untersucht. Wir stellen eine Umsetzung eines Programmgenerators für parallelisierte Programme im Kontext von Datenmanagement vor, die zeigt, dass der Code mit einer handgeschriebenen Variante vergleichbar sein kann.

Dazu werden Abstraktionen der impliziten Parallelisierung genutzt und mit Multi-Stage-Programming kombiniert, um den Leistungseinbußen der Abstraktion entgegenzuwirken und die Erstellung und Erweiterung des Programmgenerators zu vereinfachen.

In bisher bestehenden anfragekompilierenden Datenbankmanagementsystemen wird Code für eine Anfrage erzeugt, der nur auf eine sequentielle Abarbeitung ausgelegt ist. Das in dieser Arbeit vorgestellte Entwurfsmuster, ist in der Lage, Code zu generieren, der eine Anfrage parallel abarbeitet.

Die Ergebnisse sind ein Schritt in Richtung eines neuen Typs von Datenbankmanagementsystemen, welche Varianten von Algorithmen „on-the-fly“ generieren und intelligent auswählen können, um ein Anfrageergebnis so optimal zu berechnen, wie ein Experte es mit handgeschriebenen Code für diese spezifische Anfrage ebenfalls tun würde.

# Inhaltsverzeichnis

**Abbildungsverzeichnis** **ix**

**Tabellenverzeichnis** **xi**

**Quelltextverzeichnis** **xiii**

**Abkürzungsverzeichnis** **xv**

**1 Einleitung** **1**

**2 Grundlagen** **3**
2.1 Scala . . . 3
2.1.1 Typsystem . . . 3
2.1.2 Implicits . . . 4
2.1.3 Pattern Matching . . . 5
2.1.4 Traits . . . 6
2.1.5 Definieren eigener Kontrollstrukturen . . . 8
2.1.6 Virtualized Scala . . . 8
2.2 Domänenspezifische Sprachen . . . 8
2.2.1 Flache Einbettung . . . 9
2.2.2 Tiefe Einbettung . . . 9
2.2.3 Optimierungen . . . 12
2.2.4 Polymorphe Einbettung . . . 14
2.3 Lightweight Modular Staging . . . 15
2.4 Query Execution Engines . . . 17
2.5 Parallele Programmierung . . . 19

**3 Konzept** **21**
3.1 Übersicht . . . 21
3.2 Funktionsweise der Query Engine . . . 22
3.3 Randbedingungen . . . 24
3.4 Intraoperator Parallelisiserung mit Skeletons . . . 26
3.4.1 Skeletons . . . 26
3.4.2 Umsetzung von Operatoren mithilfe von Skeletons . . . 28
3.4.3 Abstraktionsebenen . . . 29
3.4.4 Nachteile von Skeletons . . . 29
3.5 Integration des Variantengenerators . . . 30
3.6 Zusammenfassung . . . 31

**4 Implementation** **33**
4.1 Ausgangspunkt . . . 33
4.2 OpenMP . . . 33
4.3 For-Loop . . . 34
4.4 Map . . . 37
4.5 Selektion . . . 38
4.6 Block-Nested-Loop-Join . . . 39
4.7 Sort-Merge-Join . . . 40
4.8 Herausforderungen . . . 41

**5 Evaluierung** **45**
5.1 Methodik . . . 45
5.2 Skeletons im Kontext von Datenbankmanagementsystem (DBMS) und Lightweight Modular Staging (LMS) . . . 46
5.3 Performance-Messungen . . . 49
5.3.1 Testsetup . . . 49
5.3.2 Erwartungen . . . 51
5.3.3 Ergebnisse . . . 52

**6 Verwandte Arbeiten** **61**

**7 Zusammenfassung** **63**

**8 Zukünftige Arbeiten** **65**

**A Anhang** **67**

**Literaturverzeichnis** **71**

# Abbildungsverzeichnis

2.1 Verhältnis von Kinds, Typen und Werten in Scala aus [MPO08] . . . 4

2.2 Umschreiben des Syntaxbaumes des Beispielausdrucks $5 * 2 + 5 * 10$ mit der optimierten Beispiel-DSL-Implementierung . . . . . 14

3.1 Die Anfrageverarbeitung im Überblick . . . . . 23

5.1 Vergleich der Antwortzeit bei der Selektion mit einem Selektivitätsfaktor von 0.5 . . . . . 52

5.2 Nested-Loop-Join mit Paralleliserung der äußeren Schleife . . . . . 54

5.3 Nested-Loop-Join mit Parallelisierung der inneren Schleife . . . . . 55

5.4 Vergleich der Laufzeit bei Sort . . . . . 56

5.5 Vergleich der Laufzeit beim Sort-Merge-Join (SMJ) . . . . . 57

5.6 Vergleich der Antwortzeit beim Map Skeleton bei der Multiplikation mit einer Konstanten . . . . . 58

# Tabellenverzeichnis

5.1 Übersicht der Entwurfsmuster und deren Bewertung . . . . . 49
5.2 Umrechnungstabelle 32 Bit Integer Tupel in Megabytes . . . . . 50
5.3 Durschnittliche Kompilierungszeiten der verschiedenen Tests . . . . . 58

A.1 Durschnittliche Laufzeiten Map Skeleton in ms . . . . . 67
A.2 Durschnittliche Laufzeiten Sort in ms . . . . . 68
A.3 Durschnittliche Laufzeiten Sort-Merge-Join in ms . . . . . 68
A.4 Durschnittliche Laufzeiten Selektion in ms . . . . . 69
A.5 Durschnittliche Laufzeiten Nested-Loop Join (NLJ) Variante 1 in ms 69
A.6 Durschnittliche Laufzeiten NLJ Variante 2 in ms . . . . . 70

# Quelltextverzeichnis


2.1 Beispiel für eine Anwendung von Implicits in Scala. . . . . 5
2.2 Einfaches Pattern Matching . . . . . 6
2.3 Pattern Matching auf einer Baumstruktur . . . . . 6
2.4 Beispiel für Traits in Scala . . . . . 7
2.5 Beispiel für eine Trait Linearisierung . . . . . 7
2.6 Überschreiben der If-Then-Else Kontrollstruktur mit Scala-Virtualized 9
2.7 Interface für eine DSL die arithmetische Ausdrücke erzeugt und berechnet . . . . . 10
2.8 Flache Einbettung einer Domain Specific Language (DSL) für arithmetische Berechnungen . . . . . 10
2.9 Einfache tiefe Einbettung einer DSL für arithmetische Berechnungen 11
2.10 Einfaches Programm für die arithmetische DSL . . . . . 11
2.11 Interpreter für die Beispiel-DSL . . . . . 12
2.12 Compiler für die Beispiel-DSL . . . . . 13
2.13 Generierte Funktion für den Ausdruck $5 * 2 + 5 * 10$ . . . . . 13
2.14 Beispiel-DSL Optimierung mit Regeln zum Ausklammern . . . . . 13
2.15 Beispiel-DSL Implementation mit Regeln zum Ausklammern . . . . . 15
2.16 Power Funktion mit und ohne Staging . . . . . 17
4.1 Interface der parallelen For-Loop . . . . . 34
4.2 Implementierung der parallelen For-Loop . . . . . 36
4.3 Generator-Trait der parallelen For-Loop . . . . . 37
4.4 Implementierung des Map Skeletons . . . . . 38
4.5 Generierter paralleler Code für ein einfaches DSL-Programm . . . . . 38
4.6 Filter mit Präfixsumme . . . . . 39
4.7 Paralleler Block-Nested-Loop-Join . . . . . 40
4.8 Parallele Sortierung mit OpenMP . . . . . 41
4.9 Sort-Merge-Join als DSL-Programm . . . . . 43
5.1 Auswahl des Algorithmus nach Anzahl der Threads . . . . . 56

# Abkürzungsverzeichnis

| | |
|---|---|
| API | Application Programming Interface |
| AST | Abstract Syntax Tree |
| BNLJ | Block-Nested-Loop-Join |
| DBMS | Datenbankmanagementsystem |
| DSL | Domain Specific Language |
| EPFL | Ecole Polytechnique Federale de Lausanne |
| GB | Gigabyte |
| Ghz | Gigahertz |
| HDD | Hard Disk Drive |
| IDE | Integrated Development Environment |
| JNI | Java-Native-Interface |
| JVM | Java Virtual Machine |
| KB | Kilobyte |
| LMS | Lightweight Modular Staging |
| MB | Megabyte |
| NLJ | Nested-Loop Join |
| OLAP | Online Analytical Processing |
| OLTP | Online-Transaction-Processing |
| PPL | Pervasive Parallelism Laboratory |

| | |
|---|---|
| RAM | Random Access Memory |
| SIMD | Single Instruction Multiple Data |
| SMJ | Sort-Merge-Join |

# 1. Einleitung

Mit dem stetigen Preisverfall von Random Access Memory (RAM)-Bausteinen, bei gleichzeitiger Vergrößerung der Kapazität, wird es immer günstiger, eine größere Menge an Arbeitsspeicher in Datenbankservern einzusetzen. Dies ermöglicht es, ganze Datenbanktabellen im Arbeitsspeicher vorzuhalten. Der Flaschenhals zwischen Hard Disk Drives (HDDs) und RAM wird damit beseitigt. Der neue Flaschenhals ist der Transportweg, der Daten vom Arbeitsspeicher zu den Prozessoren und zurück liefert. Die Geschwindigkeit des Arbeitsspeichers wächst nicht so schnell, wie die Abarbeitungsgeschwindigkeit von Prozessoren [BMK99]. Auch werden nun hardwareunabhängige Berechnungen zum Performanceproblem, da der Prozessor nicht mehr die meiste Zeit auf die Daten von der HDD warten muss.

Berechnungseinheiten in heutigen Computern sind heterogen und parallel [KPSW93]. Das liegt daran, dass eine Leistungssteigerung nicht mehr durch steigern der Taktraten durch die Prozessorhersteller erreichbar ist, da das Verhältnis von Taktrate und Abwärme immer ungünstiger wird.

Sequentielle Programme profitieren nicht von mehreren Prozessoren. Programme müssen angepasst oder neu geschrieben werden, um die Abarbeitung zu beschleunigen. Die Verwaltung der parallelisierten Berechnung erzeugt zusätzlichen Aufwand und kann in einigen Fällen die Berechnung sogar verlangsamen. Weiterhin erhöht die Parallelisierung die Komplexität der Software, im Vergleich zu einer sequentiellen Implementation [BNS04].

Um die Berechnung eines Anfrageergebnisses im Kontext von parallelen und heterogenen Prozessoren zu optimieren, muss Code abhängig von der Struktur der Anfrage und von den zur Verfügung stehenden Berechnungseinheiten generiert werden [BBHS14, Bro15]. Die Auswirkung der Parallelisierung auf das Antwortzeitverhalten von Anfragen ist dabei schwer einzuschätzen, da verschiedene Verarbeitungsalgorithmen auf strukturell unterschiedliche Anfragen, Hardware, Codeoptimierungen, Datenmengen und Datenverteilung unterschiedlich reagieren. Die Anzahl der möglichen Kombinationen ist enorm. Alle Varianten händisch zu programmieren und zu warten ist kaum möglich. Auch ergeben sich Probleme diese Varianten in

einem DBMS zu managen. Broneske [Bro15] stellt das Konzept eines DBMS vor, welches automatisch Varianten anhand von vorhergehenden Analysen generiert und auswählt.

Einsätze von Codegenerierung und Kompilierung vor der Anfrageverarbeitung in DBMSe konnten bereits gute Leistungen vorweisen, allerdings wurde eine Integration von Parallelität und Kostenmodellen noch nicht vorgestellt [KKRC14, KVC10, Neu11].

### Zielstellung der Arbeit

Das Hauptziel der Arbeit dient zur Beantwortung der folgenden Frage:

> *Wie müssen parallele Abstraktionen für Codegeneratoren in DBMS gestaltet werden, um Varianten erzuegen zu können, die möglichst viele verschiedene Kombinationen aus Hardware und Algorithmen unterstützen?*

Um dieses Ziel zu erreichen, präsentieren wir im folgenden den Hauptbeitrag dieser Arbeit:

1. Es wird eine Abstraktion, die bei der Parallelisierung genutzt wird, aus der Literatur extrahiert und in ein Entwurfsmuster integriert.
2. Es folgt eine Umsetzung paralleler Algorithmen für die Datenbankoperatoren. Die ausgewählten Algorithmen werden auf die zuvor beschriebenen Abstraktionen abgebildet.
3. Wir zeigen Probleme und potenzielle Lösungswege auf, die bei der Umsetzung mit LMS entstehen können.
4. Wir implementieren einen Codegenerator, bei dem das Entwurfsmuster angewendet wird.
5. Wir vergleichen das Verfahren mit anderen Verfahren, anhand aufgestellter Kriterien
6. Wir untersuchen die Performance unseres Ansatzes, mithilfe von ausgewählten Datenbankoperatoren und Implementationen.

### Gliederung der Arbeit

Zu Beginn der Arbeit werden in Kapitel 2, die zum Verständnis der Arbeit notwendigen Grundlagen, dargelegt. In Kapitel 3 folgt das Konzept, welches auf die Funktionsweise der vorgestellten Query-Engine und die Entwicklung des Entwurfsmusters eingeht. Dabei werden die Randbedingungen präzisiert. Im Anschluss folgt die Beschreibung der Implementierung einiger Operatoren mit dem vorgestellten Entwurfsmuster in Kapitel 4, welche auf die wichtigsten Details und Probleme eingeht. Kapitel 5 beschreibt die Testumgebung, stellt die Ergebnisse der Performanceuntersuchung der Implementierung vor und wertet diese aus. Weiterhin werden verschiedene Entwurfsmuster miteinander verglichen, die alternativen zu unserem Entwurfsmuster darstellen. Als Abschluss der Arbeit, wird auf verwandte Arbeiten eingegangen und eine Zusammenfassung, sowie ein Ausblick gegeben.

# 2. Grundlagen

Im folgenden präsentieren wir die Grundlagen dieser Arbeit. Es erfolgt zuerst eine Einführung in die Konzepte und Schlüsseleigenschaften der Programmiersprache Scala und dem Framework, welches zum Bau des Codegenerators benötigt werden. Weiterhin werden DSLs, Multi-Stage-Programmierung, parallele Programmierung und die Arbeitsweise von kompilierenden Query-Execution-Engines behandelt. Dieses sind spezielle Themen, die normalerweise nicht Pflichtteil eines Informatikstudiums sind und deshalb hier kurz beschrieben werden.

## 2.1 Scala

Scala [Oa04] ist eine objektorientierte Programmiersprache, die gleichzeitig typische Elemente der funktionalen Pogrammierung bereitstellt wie z.B. Pattern Matching, Funktionen höherer Ordnung, Algebraische Datentypen und Typklassen. Dem Entwickler steht es frei, zu welchen Anteilen er sich an den beiden Paradigmen bedient. Der Name leitet sich von „scalable language“ ab und bringt zum Ausdruck, dass der sehr kompakt gehaltene Sprachkern die Möglichkeit bietet, häufig verwendete Sprachelemente wie z.B. Operatoren oder zusätzliche Kontrollstrukturen in Benutzerklassen zu implementieren und dadurch den Sprachumfang zu erweitern und eigene DSLs zu erstellen. Scala wird auf der Java Virtual Machine (JVM) ausgeführt, wodurch Kompatibilität zu Java Klassen und Bibliotheken gewährleistet ist.

Im Weiteren werden Sprachkonzepte beschrieben, die im Rahmen dieser Arbeit relevant sind. Für ausführliche Erklärungen zur Scala-Programmierung sind [Oa04] oder [OSV11] zu konsultieren.

### 2.1.1 Typsystem

Scala besitzt ein statisches Typsysten, wodurch der Compiler Typfehler bei der Übersetzung erkennt und viele Laufzeitfehler automatisch vermieden werden. Einer String Variablen kann beispielsweise kein Int Wert zugewiesen werden. Typen werden vom Scala-Compiler automatisch in den meisten fällen geschlussfolgert, sodass sie

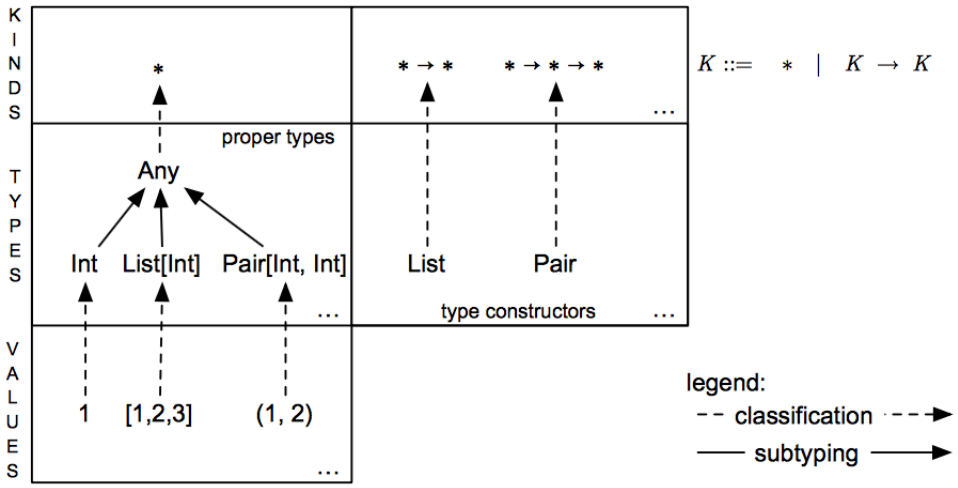


Abbildung 2.1: Verhältnis von Kinds, Typen und Werten in Scala aus [MPO08]

nicht explizit angegeben werden müssen. Die Anweisung `val x = 1` ist zulässig, wobei der Typ alternativ explizit angegeben werden darf, wie in `val x: Int = 1`.

Analog zu Funktionen höherer Ordnung, sind Typen höherer Ordnung definierbar. Das sind Typen, die Typen als Argument nehmen, um einen neuen vollwertigen Typen zu bilden, dem konkrete Werte zuweisbar sind [MPO08]. Der Typ Int nimmt keine Argumente und ist deshalb bereits ein vollwertiger Typ. Der Typkonstruktur List nimmt einen Typen, um einen vernüftigen Typen bilden zu können. Damit eine Liste von Ints, einer Variablen zugewiesen werden kann, muss dem Typkonstruktur von List der Typ Int übergeben werden. Abbildung 2.1 fasst die Zusammenhänge bildlich zusammen. Konkrete Werte sind auschließlich vollwertigen Typen zuzuordnen. Alle vollwertigen Typen sind eine Unterklasse von Any (ähnlich Object in Java). Jeder Typ gehört zu einem *Kind* (engl.). *Kinds* werden mit dem Stern Symbol notiert. Der Sachverhalt, dass der Typkonstruktur von List einen weiteren Typen benötigt, um einen vollwertigen Typen zu bilden wird mit $* \rightarrow *$ (gelesen „Nimm einen Typen als Eingabe und erzeuge einen neuen Typen“) notiert. $* \rightarrow * \rightarrow *$ bedeutet dann, dass zwei vollwertige Typen notwendig sind, um einen neuen zu erzeugen usf.

### 2.1.2 Implicits

Passt ein Wert nicht zum Typen einer Funktionsignatur oder einer Variablen, bricht ein Compiler einer statisch typisierten Sprache die Kompilierung normalerwiese ab. Der Scala-Compiler bietet einen Mechanismus an, um automatisch Ersetzungen vorzunehmen. Funktionen oder Klassen, denen das Schlüsselwort „implicit“ vorangestellt ist, signalisieren dem Compiler, dass er diese Funktionen oder Klassen für Ersetzungen verwenden darf. Sind mehrere passende Ersetzungsmöglichkeiten in der Umgebung, nimmt der Compiler keine Ersetzung vor und bricht die Kompilierung ab.

Quelltext 2.1 zeigt eine Anwendung von Implicits. Die Klasse `SomeClass` (Zeile 1) wird in Zeile 8 mit einer Methode aufgerufen, die sie nicht definiert. Das Programm kompiliert trotzdem, da in Zeile 3 eine Klasse `EnrichedSomeClass` existiert, der das

```
1  class SomeClass
2
3  implicit class EnrichedSomeClass(test: SomeClass) {
4    def enrichedMethod = println("enriched method")
5  }
6
7  val test = new SomeClass()
8  test.enrichedMethod
9
10 implicit def IntToArray(value: Int): Array[Int] = Array(
       value)
11 def printHead[T](array: Array[T]) = println(array.head)
12 printHead(1)
```

Quelltext 2.1: Beispiel für eine Anwendung von Implicits in Scala.

Schlüsselwort implicit vorangestellt wurde und einen Wert vom Typen `SomeClass` als Instanzierungsparameter nimmt. Eine Instanz dieser Klasse wird automatisch an der Stelle der `SomeClass` Instanz eingesetzt und deren `enrichedMethod` Methode aufgerufen. Damit können Klassen z.B. aus fremden Bibliotheken nachträglich um Verhalten erweitert werden, ohne sie modifizieren zu müssen. Dies folgt dem Offen-Geschlossen-Prinzip [Mey88].

Auf ähnliche Weise werden auch Methodenparameter ersetzt. In Zeile 12 soll die Methode `printHead` mit einem `Int`-Wert aufgerufen werden. Der Compiler sucht eine mögliche Ersetzung und findet sie mit der Methode in Zeile 10, die einen Int-Wert in ein Array umwandelt. Diese wird aufgerufen und ersetzt dann den Funktionsparameter in der `printHead` Methode.

### 2.1.3 Pattern Matching

In der funktionalen Programmierung gibt es keine If-Else oder Switch Anweisungen. Das funktionale Pendant dafür ist das sogenannte *Pattern Matching*. Es dient dazu, Strukturen zu identifizieren und diese dann spezifisch zu verarbeiten.

Ein einfaches Pattern Matching ist in Quelltext 2.2 zu sehen, welches einem Switch ähnelt. Das Schlüsselwort `match` gibt an, dass die Variable `x` auf die nachfolgend definierten Fälle geprüft werden soll. Es sind drei Fälle definiert, die je nach Höhe der Eingabe `x` einen entsprechenden String zurückgeben. Sobald ein Fall zutrifft, werden die restlichen Fälle nicht mehr abgeglichen. Die Reihenfolge der Fälle kann eine Auswirkung auf das Resultat haben. Der Unterstrich ist eine sogenannte *Wildcard* und trifft auf alles zu.

Quelltext 2.2 zeigt ein komplexeres Beispiel, mit einer binären Baumstruktur (Zeile 1-3). Diese ist rekursiv definiert (Algebraischer Datentyp). Ein Blattknoten ist ein Baum und hat einen Wert. Ein Knoten ist ebenfalls ein Baum und besitzt einen Wert und eventuell einen rechten und linken Teilbaum. Der Baum soll für das Debugging auf die Konsole geschrieben werden. Dazu definieren wir die Funktion `printTree`

```
1 def matchTest(x: Int): String = x match {
2     case 1 => "one"
3     case 2 => "two"
4     case _ => "many"
5 }
```

Quelltext 2.2: Einfaches Pattern Matching

```
1  abstract sealed class Tree
2  case class Leaf(value: Int) extends Tree
3  case class Node(value: Int, left: Option[Tree], right:
       Option[Tree]) extends Tree
4
5  def printTree(tree: Option[Tree]): Unit = tree match {
6    case Some(Node(_, Some(Leaf(x)), Some(Leaf(y)))) if x == y
         => println("Ding Ding Ding")
7    case Some(Node(value, left, right)) => {
8      println(value)
9      printTree(left)
10     printTree(right)
11   }
12   case Some(Leaf(x)) => println(x)
13   case _ =>
14 }
```

Quelltext 2.3: Pattern Matching auf einer Baumstruktur

(Zeile 5-14), die nur aus einem Pattern Matching besteht. Dieses erkennt Knotenstrukturen (Fall 2) und Blattstrukturen (Fall 3). Knotenstrukturen rufen rekursiv den linken und rechen Teilbaum auf und realisieren damit eine Traversierung. Für jeden Knoten wird dessen Wert auf die Konsole ausgegeben. Werden Blattknoten gefunden, wird die Rekursion gestoppt (Fall 3). Diese Muster können beliebig komplex werden und auch Bedingungen (sogenannte *Guards*) enthalten. Beispielweise um Geschwisterblattknoten mit dem gleichen Wert zu finden, sieht die Struktur aus wie in dem ersten Fall des Matchings. Es wird nach einem Knoten gesucht der links und rechts Blätter als Teilbäume hat und deren Werte miteinander übereinstimmen.

### 2.1.4 Traits

*Traits* sind in Scala das Pendant zu Java-Interfaces. Allerdings sind sie in Scala flexibler verwendbar. Im Unterschied zu Interfaces, können Traits die deklarierten Methoden direkt implementieren oder Properties definieren. Es ist modulares Verhalten, welches der Programmierer einer Klasse hinzufügt, ohne von einer Oberklasse erben zu müssen. Das Beispiel in Quelltext 2.4 zeigt eine einfach Trait Implementation. Es wird Verhalten für den Vergleich mit einem anderen Objekt angelegt. Dies kann in mehreren unabhängigen Klassen von nutzen sein. Dabei wird das Interface nur teilweise implementiert. `isNotSimilar` wird als Gegenteil von `isSimilar` im-

```
1  trait Similarity {
2    val someProperty = 0
3    def isSimilar(x: Any): Boolean
4    def isNotSimilar(x: Any): Boolean = !isSimilar(x)
5  }
```

Quelltext 2.4: Beispiel für Traits in Scala

```
1  trait Default {
2    def msg: String = "Default"
3  }
4
5  trait Foo extends Default {
6    override def msg: String = "Foo " + super.msg
7  }
8
9  trait Bar extends Default {
10   override def msg: String = "Bar " + super.msg
11 }
12
13 trait Baz extends Default{
14   override def msg: String = "Baz " + super.msg
15 }
16
17 object puzzler extends Default with Baz with Bar with Foo
18 println(puzzler.msg) //Foo Bar Baz Default
19 object puzzler2 extends Default with Foo with Bar with Baz
20 println(puzzler2.msg) //Baz Bar Foo Default
```

Quelltext 2.5: Beispiel für eine Trait Linearisierung

plementiert. isSimilar muss in der Klasse, die von Similarity erbt, implementiert werden.

Erbt eine Klasse von mehreren Traits, die die gleiche Methode implementieren, kommt es zu einem Problem, bei dem entschieden werden muss, welche Implementation gewählt wird. Im Scala-Compiler wird dieses Problem durch *Linearisierung* der Traits gelöst. Dabei werden die Traits in eine bestimmte Reihenfolge gebracht. Die Methode aus dem ersten Trait in der Reihenfolge, der diese Implementiert, wird genutzt. Allerdings ist zusätzlich mit super die Methode in einem höheren Trait aufrufbar. Dies ermöglicht eine einfache Umsetzung des Decorator-Patterns [GHJV94].

Quelltext 2.5 zeigt die Anwendung von Trait-Linearisierung und welche Rolle die Vererbungsreihenfolge spielt. Zuerst wird ein Basis-Trait „Default“ angelegt, welcher die Methode `msg` implementiert, die einen String zurückgibt. Danach werden drei neue Traits definiert, die von `Default` abgeleitet sind und die, die `msg` Methode überschreiben. Innerhalb dieser Methode rufen sie mit `super` die `msg` Methode einer höheren Vererbungsstufe auf. In Zeile 17 und 19 werden zwei Objekte angelegt

die diese Traits in jeweils unterschiedlichen Reihenfolgen erben und dann die `msg` Methode aufrufen. Die Konsolenausgaben zeigen, dass die `msg` Methode des Traits zuerst genutzt wird, der in der Vererbungsliste am weitesten rechts steht und danach schrittweise nach links weitergeht.

### 2.1.5 Definieren eigener Kontrollstrukturen

Ein Beispiel für die Vermischung der Programmierstile ist die klassische For-Schleife, die in Scala ausdrucksfähiger ist und dort For-Ausdruck genannt wird [OSV11]. In Scala kann ein For-Ausdruck in einem imperativen Stil geschrieben werden: $for (x <- expr1) expr2. Dieser Ausdruck wird in folgenden äquivalenten Ausdruck im funktionalen Stil umgeschrieben oder kann direkt vom Programmierer angegeben werden:`expr1.foreach(x => expr2)`

Der Ausdruck `expr1` muss einen Typen zurückgeben, der die `foreach` Methode implementiert. Es handelt sich dabei um eine Funktion höherer Ordnung. Diese nimmt eine einstellige Funktion, welche kein Ergebnis liefert. Selbstdefinierte Typen müssen lediglich die foreach Funktion implementieren, um die imperative Schreibweise nutzen zu können und ändern damit die Semantik der Kontrollstruktur. Das ist hilfreich bei der Implementation von DSLs.

### 2.1.6 Virtualized Scala

Virtualized Scala [RAM+12] ist eine Erweiterung des Scala Compilers. Diese bietet dem Programmierer die Möglichkeit, die Semantik weiterer Sprachelemente, neben des For-Ausdrucks, zu verändern, indem diese ebenfalls zu Methodenaufrufen umgeschrieben werden, wie z.B. Variablenzuweisungen, Variablendefinitionen, Do-While Schleifen oder return Anweisungen. Dadurch ist eine tiefere Einbettung von DSLs zu erreichen, als im herkömmlichen „Just-A-Library“ Ansatz, bei denen lediglich die Typen und Methoden Bezeichnungen aus der Domäne erhalten und mittels einer Bibliothek in das Programm eingeführt werden.

Quelltext 2.6 zeigt eine If-Then-Else Struktur, die so modifiziert wurde, dass immer der Then Block ausgeführt wird. Das Ergebnis der If-Bedingung wird lediglich auf der Konsole ausgegeben ausgegeben. Die Argumente der Methode sind die If-Bedingung, der Then-Block und der Else-Block. Diese sind als sogenannte „By-Name“ Parameter ausgeführt, welches am Pfeil zu erkennen ist, der dem Parameter vorangestellt ist. Damit wird verhindert, dass Ausdrücke bereits bei der Parameterübergabe ausgewertet werden, sondern erst bei der Verwendung innerhalb der Methode. In diesem Fall wird z. B. verhindert, dass beide Blöcke ausgewertet werden, obwohl nur einer verwendet wird.

## 2.2 Domänenspezifische Sprachen

Programmiersprachen wie Scala, Java, C++, Python, Ruby sind universelle Sprachen. Sie können Probleme aus verschiedenen Bereichen lösen. Diese Universalität bedeutet gleichzeitig, dass diese Sprachen für spezielle Probleme nur weniger effiziente Lösungen implementieren können. Die Idee ist, die Ausdrucksfähigkeit einer Sprache zu beschränken und gleichzeitig die Semantik zu spezialisieren, sodass nur

```
1 def __ifThenElse[T](cond: => Boolean, thenp: => T, elsep: =>
      T): T = {println("if: "+cond); thenp}
2 // scala> if (true) println(1) else println(2)
3 // if: true
4 // 1
5 // scala> if (false) println(1) else println(2)
6 // if: false
7 // 1
```

Quelltext 2.6: Überschreiben der If-Then-Else Kontrollstruktur mit Scala-Virtualized

eine Teilmenge aller Probleme gelöst werden können, diese dafür aber effizienter (im Sinne von Programmieraufwand und verkürzter Programmlaufzeit). Der Fokus bei der Programmerstellung verschiebt sich vom „Wie“ auf das „Was“. Solche Sprachen werden DSLs genannt. Fowler [Fow10] definiert eine DSL kurz als „computer programming language of limited expressiveness focused on a particular domain.“ Eine praktische Einführung in DSLs gibt Ghosh [Gho10].

DSLs werden in verschiedene Gruppen eingeteilt, je nach Art und Weise ihrer Umsetzung. Eine sogenannte externe DSLs wird wie eine herkömmliche Programmiersprache implementiert. Es muss ein gesamtes Ökosystem erstellt werden, bestehend aus Compiler, Integrated Development Environment (IDE) etc. Diese Infrastruktur bereitzustellen und zu warten ist mit einem hohem Aufwand verbunden. SQL ist ein Beispiel für ein externe DSL. Sie ist in ihrer Ursprungsform nicht Turing vollständig. Ein DSL-Programm in SQL entspricht einer Anfrage und muss von einem DBMS erst interpretiert bzw. kompiliert werden.

Eine weitere Umsetzungsart ist die interne DSL. Diese wird zusätzlich nach der Einbettungstiefe unterteilt. Dabei wird die DSL in eine universelle Sprache (Hostsprache) integriert. Die Infrastruktur und Werkzeuge der Hostsprache werden wiederverwendet. Der Arbeitsaufwand wird verringert.

### 2.2.1 Flache Einbettung

Bei einer flachen Einbettung sind Werte der DSL direkt von Werten der Hostsprache repräsentiert. In der Praxis erhalten Methoden und Typen Bezeichnungen, die an Konzepten der Problemdomäne angelehnt sind und so funktionieren, wie ein Domänenexperte es erwarten würde. Es ist eine Form von syntaktischem Zucker. Dadurch ist eine höhere Abstraktion erreicht und das arbeiten in einer Domäne erleichtert. Das Programm kann weiterhin direkt ausgeführt werden. Diese Form wird üblicherweise als Bibliothek implementiert, die in das Benutzerprogramm importiert wird [Hud96].

### 2.2.2 Tiefe Einbettung

Bei der tiefen Einbettung sind Werte der eingebetteten Sprache symbolisch in der Hostsprache dargestellt [BGG+92]. Das bedeutet, dass sie durch spezielle Datenstrukturen in der Hostsprache abgebildet werden.

```
1 trait ArithDSLInterface {
2   type TermRep
3   def plus(term1: TermRep, term2: TermRep): TermRep
4   def sub(term: TermRep, term2: TermRep): TermRep
5   def mult(term: TermRep, term2: TermRep): TermRep
6   def div(term: TermRep, term2: TermRep): TermRep
7   implicit def const(value: Double): TermRep
8 }
```

Quelltext 2.7: Interface für eine DSL die arithmetische Ausdrücke erzeugt und berechnet

```
1 trait ArithDSLShallowImpl extends ArithDSLInterface {
2   type TermRep = Double
3   def plus(term: TermRep, term2: TermRep) =  term + term2
4   def sub(term: TermRep, term2: TermRep) = term - term2
5   def mult(term: TermRep, term2: TermRep) = term * term2
6   def div(term: TermRep, term2: TermRep) = term / term2
7   implicit def const(value: Double): TermRep = value
8 }
```

Quelltext 2.8: Flache Einbettung einer DSL für arithmetische Berechnungen

Es sei eine DSL für arithmetische Ausdrücke, die bis zum Ende des Kapitels als Beispiel dient. Ihr Interface könnte wie in Quelltext 2.7 umgesetzt werden. Sie besitzt einen abstrakten Typen `TermRep`, der einen Term repräsentiert und erst in einer Implementation näher konkretisiert wird. Weiterhin sind auch die vier Grundoperationen auf diesem abstrakten Typen definiert. Es ist eine Möglichkeit *Polymorphie* in Scala umzusetzen. Die implizite Funktion `const` sorgt dafür, das bestimmte Werte der Hostsprache in Werte der DSL automatisch übersetzt werden.

Eine flache Einbettung ist in diesem Fall trivial und ist in Quelltext 2.8 umgesetzt. Wir nutzen den Typen `Double` der Hostsprache als Type-Alias für den abstrakten Typen `TermRep` (Zeile 2) und die arithmetischen Operatoren der Hostsprache, um die Operatoren der DSL umzusetzen (Zeile 3-6). Die `const` Funktion wird als Identität implementiert (Zeile 7).

Im Kontrast dazu, zeigt Quelltext 2.9 eine mögliche tiefe Einbettung der DSL. Dazu wird als Basis für die symbolische Darstellung, die abstrakte Klasse `Term` in Zeile 2 angelegt, die allgemein einen Term symbolisiert. Alle arithmetischen Operationen nehmen nicht mehr `Doubles` als Parameter, sondern die symbolischen Werte (Zeile 5-9). Die `const` Funktion wandelt automatisch `Double` Werte der Hostsprache in symbolische Konstanten der DSL (Zeile 16). In der Implementation erzeugen wir zunächst für alle Operationen Symbole, die vom Basisterm abgeleitet sind (Zeile 5-9). Die überschriebenen Funktionen des DSL-Interfaces konstruieren die zugehörigen Termsymbole (Zeile 12-16), statt sie direkt zu berechnen. Abstrakt betrachtet wird eine arithmetische Berechnung in einen Abstract Syntax Tree (AST) umgewandelt. Ein einfaches Programm, welches den Ausdruck $5*2+5*10$ darstellt, wird in Quelltext 2.10 umgesetzt. Anders als bei einer flachen Einbettung, wird der Wert nicht

```
1  trait ArithDSLImpl extends ArithDSLInterface {
2    type TermRep = Term
3    //Term Symbole
4    abstract class Term()
5    case class Const(number: Double) extends Term
6    case class Plus(term:Term,term2:Term) extends Term
7    case class Sub(term:Term,term2:Term) extends Term
8    case class Mult(term:Term,term2:Term) extends Term
9    case class Div(term:Term,term2:Term) extends Term
10
11   //DSL Operationen
12   def plus(term:Term, term2:Term): Term = Plus(term, term2)
13   def sub(term:Term, term2:Term): Term = Sub(term, term2)
14   def mult(term:Term, term2:Term): Term = Mult(term, term2)
15   def div(term:Term, term2:Term): Term = Div(term, term2)
16   implicit def const(value: Double) = Const(value)
17 }
```

Quelltext 2.9: Einfache tiefe Einbettung einer DSL für arithmetische Berechnungen

```
1  trait DSLProg extends ArithDSLInterface {
2    def f = plus(mult(5, 2), mult(5, 10))
3  }
```

Quelltext 2.10: Einfaches Programm für die arithmetische DSL

```
1  trait Interpreter {
2    val IR: ArithDSLImpl
3    import IR._
4    def interpret(term: Term): Double = term match {
5      case Const(number) => number
6      case Mult(term,term2) => interpret(term)*interpret(term2
         )
7      case Plus(term,term2) => interpret(term)+interpret(term2
         )
8      case Div(term,term2) => interpret(term)/interpret(term2)
9      case Sub(term,term2) => interpret(term)-interpret(term2)
10   }
11 }
```

Quelltext 2.11: Interpreter für die Beispiel-DSL

direkt berechnet, sondern die Funktion `f` erzeugt lediglich den gesamten Syntaxbaum für den Ausdruck. Bei einer tiefen Einbettung ist immer ein Interpreter oder Compiler notwendig. Das Prinzip ist dasselbe, wie beim Interpreter Pattern [GHJV94]. Ein einfacher Interpreter traversiert den Syntaxbaum und führt je nach Symbol die passende Operation in der Hostsprache aus. Der Interpreter in Quelltext 2.11 geht dabei rekursiv vor und nutzt Pattern-Matching. Ein Compiler für die Beispiel-DSL geht ähnlich vor. Quelltext 2.12 zeigt eine Implementation eines C-Compilers für die Beispiel-DSL. Die Cases generieren aus den symbolischen Werten einen äquivalenten Term in C als Strings der Hostsprache (Zeile 6 - 10). Zusätzlich wird ein Funktionsrumpf generiert, in dem der Term eingesetzt wird (Zeile 13) Der Beispielausdruck wird Die generierte Funktion in Quelltext 2.13 ist wiederum mit einem C-Compiler in Maschinencode übersetzbar und kann in das Scala-Hostprogramm über das Java-Native-Interface (JNI) eingebunden werden, um von der besseren Performance von Maschinencode zu profitieren.

### 2.2.3 Optimierungen

Indem bei DSLs der Umfang der Sprache eingeschränkt ist, können Annahmen getroffen werden, die nur in dieser Domäne zulässig sind, aber bei einer universellen Sprache die Allgemeinheit verletzen würden. Domänenspezifische Optimierungen sind mit der flachen Einbettung nicht möglich, da die spezielle Semantik der DSL nicht vom Compiler der Hostsprache berücksichtigt wird. Semantisches Verständnis wird hingegen bei der tiefen Einbettung erzeugt, indem das Benutzerprogramm in eine Zwischenrepräsentation umgewandelt wird. Geeignet dafür ist z. B. Syntaxbaum oder Graph. Der Optimierer ersetzt oder entfernt Teile in dieser Repräsentationform, mithilfe von domänenspezifischen Ersetzungsregeln.

Ein mögliche Optimierung für die Beispiel-DSL, aus der Domäne der Arithmetik, ist das Ausklammern von Ausdrücken. Dazu wird die Implementierung um ein passenden Satz an Regeln erweitert, die in Quelltext 2.14 von Zeile 3-7 zu sehen sind. Dabei werden alle möglichen Varianten abgebildet. Kann keine Optimierungsregel angewendet werden, wird die Funktion der Basisimplementation aufgerufen (Zeile

```
1  trait Compiler {
2    val IR: ArithDSLImpl
3    import IR._
4    def compile(term: Term): String = {
5      def compileTerm(term: Term): String = term match {
6        case Const(number) => number.toString
7        case Mult(term, term2) => "( " + compileTerm(term) + "
             * " + compileTerm(term2) + " )"
8        case Plus(term, term2) => "( " + compileTerm(term) +
           " + "+ compileTerm(term2) + " )"
9        case Div(term, term2) => "( " + compileTerm(term) + "
           / " + compileTerm(term2) + " )"
10       case Sub(term, term2) => "( " + compileTerm(term) + "
           - " + compileTerm(term2) + " )"
11     }
12     val compiledTerm = compileTerm(term)
13     return s"double f() {\n return $compiledTerm; \n}"
14   }
15 }
```

Quelltext 2.12: Compiler für die Beispiel-DSL

```
1 double f() {
2  return ( ( 5.0 * 2.0 ) + ( 5.0 * 10.0 ) );
3 }
```

Quelltext 2.13: Generierte Funktion für den Ausdruck $5 * 2 + 5 * 10$

```
1 trait ArithDSLImplFactorisationOpt extends ArithDSLImpl {
2   override def plus(term: Term, term2: Term): Term = (term,
      term2) match {
3     case (Mult(term1,term2), Mult(term3,term4)) if term1 ==
        term3 => Mult(term1, Plus(term2, term4))
4     case (Mult(term1,term2), Mult(term3,term4)) if term1 ==
        term4 => Mult(term1, Plus(term2, term3))
5     case (Mult(term1,term2), Mult(term3,term4)) if term2 ==
        term4 => Mult(term2, Plus(term1, term3))
6     case (Mult(term1,term2), Mult(term3,term4)) if term2 ==
        term3 => Mult(term2, Plus(term1, term4))
7     case (term,term2) => super.plus(term, term2)
8   }
9 }
```

Quelltext 2.14: Beispiel-DSL Optimierung mit Regeln zum Ausklammern

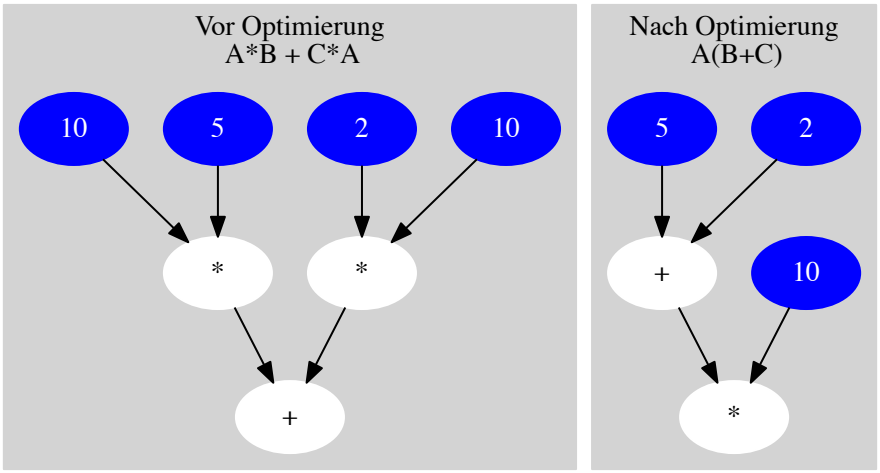


Abbildung 2.2: Umschreiben des Syntaxbaumes des Beispielausdrucks $5 * 2 + 5 * 10$ mit der optimierten Beispiel-DSL-Implementierung

7). Abbildung 2.2 zeigt den Syntaxbaum vor und nach der Optimierung, die durch entfernen und umschreiben von Knoten durchgeführt wurde.

Ein Nachteil gegenüber der flachen Einbettung, ist ein erhöhter Programmieraufwand und mehr Programmcode durch die Implementierung der symbolischen Repräsentation und der Optimierungen nötig. Allerdings gibt es bereits Ansätze automatisch symbolische Repräsetationen aus der flachen Einbettung zu erzeugen [JSS+14].

### 2.2.4 Polymorphe Einbettung

Ein Problem bei einer internen Umsetzung ist, das die Semantik (Optimierungen und Analysen) fix ist. Es gibt Anwendungsszenarien, die von einer anpassbaren Interpretation der DSL profitieren, beispielsweise um hardwarespezifische Optimierungen einzubinden. Hofer [HORM08] stellt ein Konzept vor, mit dem eine DSL modular in Scala aufgebaut wird. Die DSL hat genau ein Interface und beliebig viele Interpretationen (Implementierungen), die wiederum miteinander kombinierbar sind. Domänenspezifische Optimierungen und Analysen können nachträglich ergänzt oder ausgetauscht werden, um die DSL für einen speziellen Einsatzzweck (z. B. Hardwarekomponenten) anzupassen. In der Beispiel-DSL wurde diese Vorgehensweise bereits angewendet. Das Interface der DSL und eine mögliche Implementation wurden bereits voneinander getrennt. Dem Beispielprogramm aus Quelltext 2.10 werden bei der Instanzierung die gewünschten Traits der Implementation und der Optimierungen, sowie weitere Eigenschaften wie Logging etc. hinzugefügt, wie in Zeile 1 von Quelltext 2.15 zu sehen ist. Die Auslassungszeichen sollen andeuten, dass mehr Optimierungen und zusätzliches Verhalten hinzufügbar sind. Damit der Compiler/Interpreter und das Programm von den Typen her Kompatibel zueinander sind, wird der Typ der Zwischenrepräsentation in der Compiler/Interpreter Repräsentation konkre-

```
1 val prog = new DSLProg with ArithDSLImpl with
    ArithDSLImplFactorisationOpt with ArithDSLTermMoreOpt
    with ... with Logging
2 val interpreter = new Interpreter {val IR: prog.type = prog
    }
3 val compiler = new Compiler {val IR: prog.type = prog }
4 println(interpreter.interpret(prog.f))
5 println(compiler.compile(prog.f))
```

Quelltext 2.15: Beispiel-DSL Implementation mit Regeln zum Ausklammern

tisiert und zwar zu demselben Typen, den das DSL-Programm letztendlich bildet (Zeile 2-3), ohne dieselben Traits erben zu müssen.

## 2.3 Lightweight Modular Staging

Multi-Stage-Programming [Tah99] (kurz Staging) ist eine Form von Metaprogrammierung. Dabei werden Programmteile vor Ausführung bereits durch Vorabwissen ausgewertet, um die verbleibenden Teile effizienter gestalten zu können.

Das Ziel ist es, Berechnungen zu optimieren und wohlgeformten und korrekt typisierten Code zu erzeugen. Bekannt ist, das 10% des Programmcodes in 90% der Zeit abgearbeitet wird [HP11]. Deshalb ist es besonders Lohnenswert diese 10% zu optimieren. Das Vektorprodukt ist beispielsweise in zwei Stufen berechenbar. Zuerst wird die Länge des Vektors übergeben, um die Schleife im inneren der Berechnung zu entfernen. In der zweiten Stufe wird ein Vektor übergeben, um die Vektorwerte als Konstanten zu fixieren. Bei einer Matrixmultiplikation beschleunigt das die Berechnung, da eine Zeile mit allen anderen Spalten der anderen Matrix multipliziert wird.

Multi-Stage-Sprachen wie MetaOCaml integrieren Staging mithilfe von Staging Annotationen, speziellen Schlüsselwörtern der Programmiersprache (schwergewichtig). LMS [RO10] ist ein Staging Ansatz, ohne eine spezielle Staging Sprache zu benötigen, sondern mittels einer Bibliothek, in eine bereits bestehende Programmiersprache zu integrieren (deshalb leichtgewichtig), solange sie die Vorrausetzungen erfüllt. Dabei handelt es sich bisher um eine Scala Bibliothek, die den Scala Virtualized Compiler vorraussetzt, da dem puren Scala Compiler einige Vorrausetzungen für LMS fehlen (Beispielweise Überladungsmöglichkeiten für Kontrollstrukturen und Speicherzuweisungen).

Simple Programmgeneratoren konstruieren Programme, indem sie Strings zusammensetzen, die Quellcode repräsentieren. Diese Darstellungsform hat den Nachteil, dass es potenziell möglich ist, syntaktisch falsche Programme zu generieren. Analysen und Optimierungen sind schwierig umzusetzen, da zu einzelnen Codefragmenten keine Informationen vorliegen, da deren Semantik fehlt und nicht in dem String mitgespeichert werden können. Ein Verbesserung ist die Darstellung als AST. Dieser hat den Nachteil, dass wiederkehrende Programmteile nicht erkannt und zusammengefasst werden können. Eine weitere Verbesserung, ist die Darstellung als Datenabhängigkeitsgraphen (nicht zu verwechseln mit einem Kontrollflussgraphen). Sich

wiederholende Teilberechnungen können in einem Knoten dargestellt werden. Durch Kenntnis der Datenabhängigkeiten können Anweisungen verschoben werden, ohne das Berechnungsergebnis zu verändern. Eingehende Kanten sind in dem Graphen Parameter von Ausdrücken und ausgehende Kanten sind das Resultat deren Auswertung. Bei der Optimierung werden Knoten ausgetauscht, entfernt, hinzugefügt oder mit anderen Knoten verschmolzen. Dies kann in mehreren Phasen ablaufen um Schrittweise die Programmabstraktion zu verringern.

LMS führt einen abstrakten Datentypen `Rep[T]` ein, der Programmcode repräsentiert, wobei `T` für den Typen des Rückgabewerts des Programmcodes steht. Die Implementation dieses Typen ist frei wählbar (Modular), wobei LMS bereits eine Implementation liefert, die Programmcode als Datenabhängigkeitsgraphen mitliefert.

Um Code zu generieren, muss das betreffende Programmstück ausgeführt werden, indem es als Eingabe eine symbolische Eingabe bekommt und die Argumente der aktuellen Ausführungsstufe erhält. Dadurch wird der Graph aufgebaut und Berechnungen der aktuellen Stufe durchgeführt. Danach werden Optimierungen durchgeführt und der Graph topologisch sortiert, um die finale Ablaufreihenfolge der Anweisungen festzulegen. Die Codegeneration ist eine Traversierung des erzeugten AST, bei der jeder Knoten einen String ausgibt. Auch hier bietet LMS Standardimplementierungen für die Codeerzeugung in C und Scala. Die Objektsprache ist bei LMS dadurch konfigurierbar, indem die Implementation der Codegeneratoren ausgetauscht wird. Abstrakter Scala Code, der optimierten C Code generiert ist umsetzbar.

Als Beispiel soll eine Funktion zur Berechnung einer Potenz dienen.

Quelltext 2.16 zeigt zwei Varianten, die sich bis auf die Typen in der Signatur nicht unterscheiden. Sie sind beide rekursiv definiert und enthalten eine If-Abfrage. Durch das Vorabwissen des Exponenten kann die Rekursion und die If-Else Struktur entfernt werden. In der Signatur ist das an den Typen zu erkennen. Die Basis `b` bekommt den Typen `Rep[Int]` zugewiesen, da sie erst in der nächsten Stufe übergeben wird und beim Aufruf in der aktuellen Stufe ein symbolischer Wert genutzt wird. Der Exponent x hat den einfachen Typen Int, das bedeutet, dass dieser noch in der aktuellen Stufe übergeben wird.

Obwohl die beiden Funktionskörper identisch aussehen, ist die Semantik, aufgrund der unterschiedlichen Paramameter- und Rückgabetypen, verändert. Die direkte Variante hat die gewohnte Semantik. Die Variante mit Staging Typen hat eine andere Semantik. Es wird implizit ein Graph aufgebaut. Die Konstante 1 (Zeile 8) in der Abbruchbedingung wird in einen Konstantenknoten durch eine implizite Funktion umgewandelt. Bei der Multiplikation handelt es sich um eine überladene Variante, die einen Multiplikationsknoten erzeugt. Immer wenn ein Knoten erzeugt wird, wird dieser mit einer Nummer (Symbol) und einem Verweis auf abhängige Knoten mittels derer Nummer versehen und aufgezeichnet. Diese Nummern werden später in dem erzeugten Code genutzt für die Variablenbezeichner genutzt.

Die Funktion soll nun ausgeführt werden, um Code zu erzeugen. Zuerst müssen die Parameter übergeben werden. Der Exponent wird als Konstante übergeben. Für die Basis muss mit einer Funktion die `fresh` genannt wird, ein noch nicht verwendetes

```
1  //direkte Variante:
2    def power(b: Int, x: Int): Int =
3    if (x == 0) 1
4    else b * power(b, x-1)
5
6  //Variante mit Staging Typen:
7  def power(b: Rep[Int], x: Int): Rep[Int] =
8    if (x == 0) 1
9    else b * power(b, x-1)
10
11 //Nach auswerten der ersten Stufe mit b = 4
12 def power(x0: Int): Int = {
13 val x1 = x0 * 1
14 val x2 = x0 * x1
15 val x3 = x0 * x2
16 val x4 = x0 * x3
17 x4
18 }
```

Quelltext 2.16: Power Funktion mit und ohne Staging

Symbol erzeugt werden. Dieses Symbol verweist auf keinen Wert oder Ausdruck. Es dient nur als Platzhalter für den späteren Wert der Basis. Danach wird die Funktion ausgeführt, die Rekursion entfaltet und der Graph konstruiert. Nicht Teil des Graphen wird die If-Else Struktur, da sie keine Rep[T] Typen in der Bedingung verwendet, sondern nur das x Argument, welches in der aktuellen Stufe bekannt ist. Ab Zeile 12 ist die, aus dem Graphen erzeugte Funktion, zu sehen. Das Argument hat nun den Bezeichner x0 und war ursprünglich das unbenutzte Symbol, welches mit fresh erzeugt wurde. Die Multiplikationen von x1-x4 sind aus der entfalteten Rekursion $b * (b * (b * (b * 1)))$ hervorgegangen.

Wie gezeigt, ist der Anwendungsentwickler abgeschirmt von der Konstruktion und Komposition der Coderepräsentation, sowie der Codeoptimierung und Codegenerierung. Er nutzt die Operationen und Kontrollstrukturen der DSL, die durch Überladung eine andere Semantik erhalten und im Hintergrund in Knoten umgewandelt werden. Das erzeugte Objektprogramm ist automatisch wohlgeformt und typsicher, wenn das Metaprogramm auch wohlgeformt und typsicher ist.

## 2.4 Query Execution Engines

Die Query Execution Engine ist der Teil der Anfrageverabeitung in einem DBMS, der sich um die Berechnung des Anfrageergebnisses kümmert. Dazu erhält sie vom Query Optimizer einen Anfrageplan, der Operatoren der relationen Algebra enthält. Näheres zur Anfrageverarbeitung werden in dem Lehrbuch von Saake, Sattler und Heuer [SSH11] ausführlich behandelt.

SQL Anfragen können als deklarative domänenspezifische Programme betrachtet werden. Im Datenbankenkontext werden sie klassischerweise von der Query Execution Engine interpretiert und imperativ abgearbeitet. Dies ist mit den typischen

Nachteilen einer interpretierten Programmabarbeitung verbunden. Die SQL Anfrage wird erst zur Laufzeit analysiert. Neuere Ansätze (seit ca. 2000) vermeiden den Overhead der dabei entsteht und analysieren und kompilieren Anfragen vor der Ausführung, um die Bearbeitungszeit zu senken.

Verschiedene Autoren [RPML06, ADHW99, RDSF13, SZB11, KVC10, Neu11] stellen fest, dass das bisher vorherrschende Iterator Modell im Main Memory Kontext dazu führt, dass Prozessoren nicht optimal mit Ergebnisberechnungen ausgelastet werden und bieten verschiedene Alternativen an. Die Verwaltung der eigentlichen Berechnung nimmt einen großen Teil der Auslastung ein. Der Grund ist die tupelweise Berechnung des Ergebnisses. Ein Next-Aufruf erzeugt erneut einen Next-Aufruf in den Kind Operatoren bis hin zu einem Blatt Operator. Die Operatoren generisch gehalten, sodass sie zur Laufzeit z. B. an den Datentypen der Tabellenfelder angepasst werden müssen. Dadurch wird eine einfache Vergleichsoperation, die bei einer Selektion benötigt wird, zu einem Funktionsaufruf, der zuerst die passende Implementation mittels virtuellen Funktionen oder Funktionspointertabellen suchen muss. Das Grundproblem ist, das der Code, der für die Berechnung des Ergebnisses zuständig ist, erst über Umwege erreichbar ist, anstatt ihn komprimiert an einer Stelle abzuarbeiten. Jeder Operator ist statisch kompiliert. Inter-Operator Optimierungen, wie das Zusammenfassen von Operatoren, ist auf diese Weise nicht möglich. Postive Aspekte sind die Einfachheit, Flexibilität und die Vermeidung unnötiger Materalisierungen.

2010 stellt Krikellas [KVC10] das Konzept der Holistic Query Evaluation vor. Die Grundidee ist dabei, maßgeschneiderten spezialisierten Code für eine gesamte Anfrage zur Laufzeit zu erzeugen, mithilfe von C++ Templates. Dazu wird zwischen dem Optimierer und der Ausführungskomponente ein zusätzlicher Codegenerator eingebunden, der den Code für eine Anfrage aus dem Templates für die einzelnen Operatoren zusammenstellt. Die Operatorgrenzen sind im erzeugten Code sichtbar. Die Vorbereitungszeit der Anfrage (Codeerzeugung, Kompilierung) erhöht bei simplen Anfragen sogar die Ausführungszeit.

Neumann [Neu11] fasst Operatoren, die sich in einer Pipeline befinden, bei der Kompilierung zusammen. Damit können die Tupel länger im Prozessorcache verbleiben. Eine Pipeline bilden alle Operatoren, die sich zwischen „Pipelinebreakern“ befinden. Pipelinebreaker sind Operatoren die eingehende Tupel aus dem CPU Register nehmen, um sie zwischenzuspeichern. Performancekritische Abschnitte werden in LLVM Assembler geschrieben und reduzieren gleichzeitig die Kompilierungszeit gegenüber einem normalen optimierenden C++ Compiler.

Legobase [KKRC14] ist eine in Scala geschriebene Query Execution Engine und nutzt den erweiterbaren LMS Compiler, um nicht nur Anfragen, sondern auch Teile der Query Execution Engine zur Laufzeit zu generieren und wurde, wie Scala und LMS auch, am EPFL entwickelt. Es wird der Sachverhalt ausgenutzt, das ein „staged“ Interpreter ein Compiler ist. Das DSL-Programm (Die SQL Anfrage) wird dem Interpreter übergeben und ausgeführt, wodurch anfragespezifischer Code erzeugt und mit den Datenbankdaten ausgeführt wird. Ein Vorteil gegenüber anderen Lösungen ist die Flexibiliät und Abstraktion. Der Programmierer kombiniert Optimierungen, Datenstrukturen, Algorithmen, Codegeneratoren etc. beliebig miteinander. Dabei

sind mehrere Optimierungsebenen möglich. Neue Module können hinzugefügt werden oder andere ersetzen. Ein weiterer Vorteil ist die erhöhte Abstraktion gegenüber C++, C oder Assembler und führt zu geringeren Entwicklungs- und Wartungskosten. Normalerweise ist Performance und Abstraktion ein Trade-Off. Der in Scala eingebettete Compiler ermöglicht die von den Entwicklern propagierte „Abstraction without Regret“ und soll zum Umdenken in der Entwicklung und Forschung im Bereich von DBMSe führen [Koc13].

Das Ziel von kompilierenden Query Execution Engines ist es, Code zu erzeugen, der von der Bearbeitungszeit her, handoptimierten Code für eine spezielle Anfrage möglichst nahe kommt und die zur Verfügung stehende Hardware optimal nutzt.

## 2.5 Parallele Programmierung

Bei der parallelen Programmierung geht es darum, ein Programm auf mehreren Prozessoren gleichzeitig ablaufen zu lassen, mit dem Ziel die Abarbeitung zu beschleunigen oder größere Mengen an Daten in gleicher Zeit verarbeiten zu können.

Die Parallelität erhöht die Komplexität eines Programms und unterscheidet sich von der sequentiellen Programmierung, unteren anderem, durch die Folgenden Eigenschaften:

- Race Conditions, die zu Programmfehlern führen können, die schwer zu finden sind aufgrund des Nondeterminismus der Berechnung.
- Das Laufzeitverhalten ist schwieriger vorherzusagen, da mehr Faktoren beachtet werden müssen (Anzahl und Art der eingesetzten Prozessoren, Verteilung der Daten etc., Nutzung gemeinsamer Ressourcen)
- Verschiedene Programmiermodelle für unterschiedliche Hardware
- Kommunikation zwischen den Prozessoren wird benötigt.
- Daten müssen während des Programmablaufs aufgeteilt, verarbeitet und dann wieder zusammengefügt werden.
- Bestehende Algorithmen müssen angepasst oder neu entwickelt werden, um von mehreren Prozessoren gebrauch machen zu können
- Die Skalierung muss auf das Problem angepasst werden. Eine zu große Anzahl an Prozessoren kann die Abarbeitunsgeschwindigkeit durch einen erhöhten Kommunikationsaufwand verringern.

Auf diese muss bei der Programmierung zusätzlich geachtet werden, um maximale Performance zu gewährleisten.

McCool et al. [MRR12] gibt weiterführende Informationen zur parallelen Programmierung gibt.

# 3. Konzept

In diesem Kapitel wird vorgestellt, wie paralleliserter Code in einem DBMS, während der Anfrageverarbeitung, generiert werden kann. Dazu werden die benötigten Komponenten der Query Engine beschrieben. Weiterhin werden Skeletons erläutert, welche die zentrale parallele Abstraktion in dem vorgestellten Codegenerator darstellen. Anhand typischer relationaler Operatoren wird beschrieben, wie sie im Datenbankenkontext eingesetzt werden können und welche Probleme und Erweiterungsmöglichkeiten dabei bestehen.

## 3.1 Übersicht

Zuerst geben wir eine Übersicht indem wir die einzelnen Komponenten und deren Aufgabe in unserer Query-Engine Architektur kurz vorstellen.

1. **Interpreter** Führt den Anfrageplan aus und zeichnet die Berechnungsschritte auf, welche dabei in eine symbolische Zwischenrepräsentation überführt wird, auf der Optimierungen durchführbar sind. Dieser Interpreter basiert auf multi-stage Programmierung, was ihm zum Compiler macht.

2. **Generator** Der abstrakte Syntaxbaum der Zwischenrepräsentation wird durch den Generator interpretiert, wobei er die Symbole auf festgelegte Codeteile abbildet. Zusammen mit der Interpreter bilden diese beiden Komponenten den Anfrage-Compiler. D.h. eine SQL Anfrage wird in Quellcode übersetzt. Dieser ist komplett unabhängig vom Scala Compiler und durch den Programmierer erweiterbar.

3. **Archiv** Fertiger Maschinencode, der den Berechnungsweg für eine Anfrage festhält, wird hier abgelegt und ist bei Bedarf wieder abrufbar. Damit fällt der Overhead beim erstellen des Quellcodes und der Interpretation weg, wenn bereits Code im Archiv für eine gestellte Anfrage vorliegt. Die suche und das laden des Maschinencodes benötigt wiederum Zeit, sodass zu evaluieren ist, wie groß die Ersparnisse beim Antwortzeitverhalten wirklich sind. Nachteil ist,

dass der Maschinencode nur auf eine Anfrage fest angepasst ist. Operatoren und Prädikate dürfen nicht abweichen. Lediglich die Eingaberelationen dürfen sich ändern. Der Einsatz eines Archivs macht nur Sinn, wenn häufiger exakt dieselbe Anfrage abgefragt wird, um z. B. Veränderungen an den Relationen zu analysieren.

4. **Profiler** Sammelt Messdaten währen der Anfrageausführung und übergibt sie dem Variantengenerator zu weiteren Analyse.

5. **Variantengenerator** Der Variantengenerator kann mithilfe von Trait-Kompo sition verschiedene Query-Engines bereitstellen ohne das Programmierer Änderungen am Code vornehmen müssen. Die Idee basiert auf der Veröffentlichung von Broneske [Bro15]. Je nach Anfrage kann die Query-Engine durch den Variantengenerator ausgewechselt werden. Ein integrierter intelligenter Kostenschätzer dient dem Variantengenerator, um eine günstige Variante zu erstellen oder aus einem Pool an fertigen Varianten auswählen. Dieser schätzt die Kosten (Antwortzeit), die verschiedene Varianten benötigen würden und wählt dann eine günstige aus. Paralleliserung erschwert die Kostenschätzung, da zusätzlich Kosten hinzukommen die durch

   - das Aufteilen der Eingabe,
   - der Verteilung der Daten,
   - dem Zusammenfügen der Ergebnisse,
   - der Kommunikation zwischen den Threads,
   - und dem Lastausgleich

   entstehen.

## 3.2 Funktionsweise der Query Engine

Im Folgenden wird das Zusammenspiel der einzelnen Komponenten beschrieben. Dazu wird in Abbildung 3.1 gezeigt, wie die Abläufe innerhalb der Query Engine sind.

### Vorbereitung

Die vorgestellte Engine wird vom Variantengenerator mithilfe von Traits modular zusammengestellt oder aus einem Variantenpool ausgewählt. Die einzelnen Algorithmen und deren Implementation können aufeinander abgestimmt werden, sodass maximale Leistung erzielbar ist.

### Kompilierungsphase

Der vom Optimierer (nicht in der Übersicht, da es um die Query-Engine geht) erzeugte physische *Anfrageplan* wird vom *Interpreter* mithilfe einer *symbolischen Eingabe*, als Platzhalter für die realen Relationen, ausgeführt und die dabei benötigten Schritte werden aufgezeichnet. Dadurch wird das apriori Wissen über die Anfragestruktur

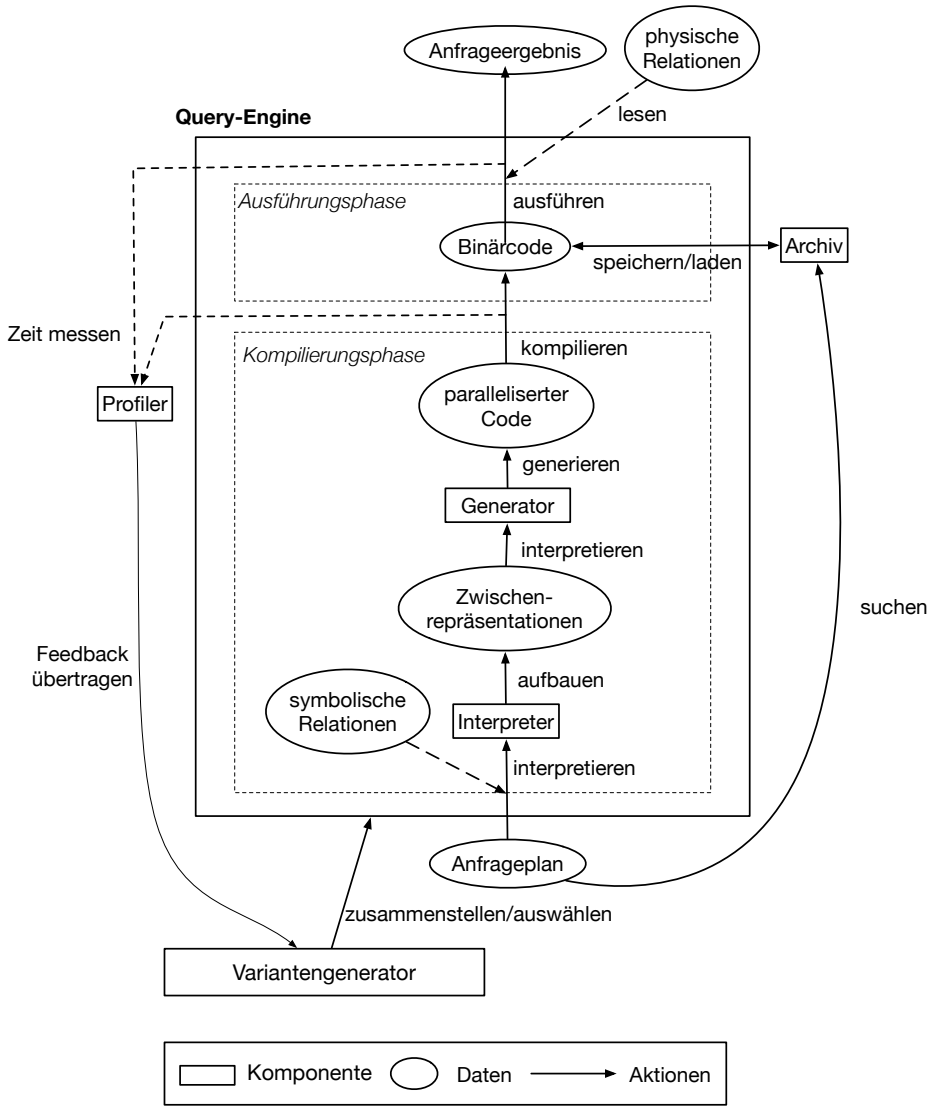


Abbildung 3.1: Die Anfrageverarbeitung im Überblick

zuerst ausgewertet. Die Abstraktionen der Paralleliserung werden in mehreren *Zwischenenrepräsentationen* konkretisiert, gemäß der hinterlegten Regeln optimiert und in einem finalen Durchlauf in parallelen Code umgewandelt und kompiliert. Dieser Code kann zur Wiederverwendung im *Archiv* abgespeichert werden, um bei einer erneuten Anfrage, die Interpretation, Analyse und Kompilierung einzusparen, die bei einigen Anfragen einen großen Anteil an dem Antwortzeithalten hat [KVC10], zu überspringen.

### Ausführungsphase

In der zweiten Phase wird der *Binärcode*, in der die Anfrage jetzt fest kodiert ist, mit den realen *physischen Relationen* aufgerufen und das *Anfrageergebnis* durch Nutzung paralleler Ressourcen berechnet. Die Ausführungszeit wird an dem *Profiler* weitergegeben, der diese Information, zusammen mit der Anfrage und der aktuellen Konfiguration der *Query-Engine*, speichert, um diese im Hintergrund analysieren zu

können und die Ergebnisse an den *Variantengenerator* zu übertragen. Diese Auswertung dient später als Entscheidungsgrundlage für eine geeignete Variante.

## 3.3 Randbedingungen

In diesem Abschnitt geht es darum, die einschränkenden Annahmen vorzustellen und diese näher zu erläutern. Diese sind entweder von konzeptueller Natur oder von praktischer Natur.

**Anfragetyp**

In der vorgestellten Query-Engine sollen ausschließlich Online Analytical Processing (OLAP) Anfragen verarbeitet werden und gehört zum Konzept. Das sind Anfragen, die nur lesend auf die Relationen zugreifen und eine lange Bearbeitungszeit haben, da auf viele Relationen zugegriffen und/oder komplexe Berechnungen auf den Relationen ausgeführt werden. In Data Warehouse Systemen kommen sie häufig vor und werden dort zur Datenanalyse eingesetzt. Da relativ viele Datensätze über mehrere Tabellen verteilt beteiligt sind, erscheint die Verarbeitung durch mehrere Prozessoren sinnvoll. Im Gegensatz dazu stehen Online-Transaction-Processing (OLTP) Anfragen, die von eher kurzer Dauer sind, da sie nur zum schreiben und lesen einzelner Datensätze durch viele Nutzer dienen. Diese werden meist zur Einbringung neuer Daten oder verändern, löschen oder abfragen weniger Datensätze genutzt. Hier ist der Einsatz von mehreren Prozessoren oder Systemen nicht sinnvoll, da der Overhead der Parallelisierung nicht im Verhältnis zur Datenmenge steht und daher das Antwortzeitverhalten gegenüber einer sequentiellen Abarbeitung sogar verlangsamt ist. Hier ist das Verteilen mehrerer Anfragen auf die Prozessoren sinnvoller, um den Durchsatz zu erhöhen. Mehr Informationen über den Einsatz von OLAP ist in dem Lehrbuch über Data Warehouse Systeme [KSS14] von Köppen et al. zu finden. Ein praktischer Aspekt ist, dass weniger Schreibzugriffe auf die Daten stattfinden, und dadurch die Implementierung der Parallelisierung erleichtern. Es müssen lediglich die Datenstrukturen bei der Berechnung mit Locks oder anderen Verfahren geschützt werden. Die Relationen aber nicht, da auf ihnen nicht schreibend zugegriffen wird.

Wenn mehrere solcher Anfragen gleichzeitig verarbeitet werden sollen, kann eine Runtime helfen den Leistungsabfall durch Interferenzen vorzubeugen. Diese würde den Zugriff auf die Ressourcen verwalten und auf die Anfragen verteilen, da es sonst zu unkontrollierten Blockierungen zwischen den einzelnen Anfragen kommen kann. In Rahmen dieser Arbeit liegt der Fokus vereinfachend auf nur einer Anfrage die gleichzeitig verarbeitet werden, wobei die Integration solcher Laufzeit nicht ausgeschlossen ist.

**Art der Parallelität**

Weiterhin steht die Datenparallelität im Zentrum der Betrachtung. Das ist die Art von Parallelität, bei der auf mehrere gleichartige Daten gleichzeitig dieselbe Operation ausgeführt wird.

Die Task-Parallelität (verschiedene Aufgaben werden gleichzeitig ausgeführt) und I/O Parallelität (lesen von mehreren Datenquellen gleichzeitig) wird nicht untersucht. Die Berechnung eines einzelnen Anfrageergebnisses soll durch Parallelität beschleunigt werden, dass wird als Intra-Query Parallelität bezeichnet, diese ist von der

Inter-Query Parallelität abzugrenzen, bei der mehrere Anfragen gleichzeitig auf mehreren Prozessoren parallel abgearbeitet werden. Bei der Intra-Query Parallelität wird zusätzlich in Intra- und Inter-Operator Parallelität unterschieden. Für diese Arbeit untersuchen wir die Intra-Operator Parallelität, also die Paralleliserung innerhalb eines Datenbankoperators. Diese ist von der Inter-Operator Parallelität abzugrenzen, welche mehrere Operatoren gleichzeitig auf mehreren Prozessoren ausführt, falls es die Datenabhängigkeiten zwischen den Operatoren erlauben. Die Einschränkungen sind von praktischer Natur und die Integration weiterer Parallelitätsarten kollidiert nicht direkt mit unserem Konzept. Weitere Untersuchungen müssen feststellen inwieweit diese anderen Parallelitäten genutzt werden können.

#### Art der Datenspeicherung

Die Daten der Relationen liegen vor Beginn der Anfrage bereits im Hauptspeicher, sodass lediglich die Datentransfers vom Hauptspeicher zum Prozessor die Berechnung beeinflussen. Damit wird die Verarbeitung durch mehrere Prozessoren erst sinnvoll, da die zu verarbeitenden Daten sonst nicht schnell genug herangeführt werden können. Diese müssten erst von der HDD in den Arbeitsspeicher geladen werden, wobei die Lese- und Schreibgeschwindigkeit dieser Hardware um Größenordnungen auseinanderliegen. Eine Übersicht über die Eigenschaften von Hauptspeicherdatenbanken gibt Garcia et al. [GMS92]. Das ist eine praktische Einschränkung. Es ist kein Problem die Daten auch von HDD's zu beziehen, obwohl dieses Vorgehen nicht mit den Zielen unseres Ansatzes vereinzubaren ist (Performance).

#### Verteilung der Daten

Ein weiteres Problem, welches aus praktischen Gründen vernachlässigt wird, ist die Auswirkung einer ungleichen Datenverteilung, die bei der Berechnung entstehen kann. Beispielsweise wird ein Datensatz auf mehreren Prozessoren aufgeteilt und gefiltert. Danach erfolgt direkt eine weitere Operation auf diesen Teildatensatz. Die Prozessoren die am meisten Daten im ersten Schritt herausgefiltert haben, werden nun diese Operation schneller ausführen können, da ihre Eingabegröße kleiner ist. Diese Prozessoren müssen zum Schluss der Berechnung auf die anderen Prozessoren mit größerer Eingabemenge warten. Optimalerweise sollten alle Prozessoren ungefähr gleich lang rechnen, um diese Wartezeit möglichst gering zu halten. Gegenmaßnahmen sind durch eine geeignete Implementationen der Algorithmen potenziell möglich, ist aber nicht Fokus unserer Arbeit.

#### Programmiersprachen

Die Metasprache, die zum erstellen von Code genutzt wird bzw. des Codegenerators, ist Scala. Dies ist durch die Verwendung von dem LMS-Frameworks vorgegeben, obwohl eine Umsetzung mit einer anderen Programmiersprache möglich ist, solange sie die Vorraussetzungen mitbringt. Eine Voraussetzung ist beispielsweise das eine Sprache virtualisierbar ist, dass bedeutet das alle Sprachkonstrukte (Loops, Variablenzuweisungen etc.) überschreibbar sind. Das trifft aber auf kaum eine populäre Programmiersprache zu, sodass es für diese kein LMS-Framework in naher Zukunft geben wird. Für die Testimplementation gehen wir der davon aus, dass die Objektsprache C ist, obwohl mithilfe von LMS auch in jede beliebige andere Sprache

übersetzt werden kann, solange eine Übersetzung der Scala Typen und Strukturen in Strukturen und Typen der Objektsprache bereitgestellt wird. Es handelt sich demnach um eine Einschränkung praktischer Natur.

**Operatoren**

Zur Veranschaulichung betrachten wir die wichtigsten relationalen Operatoren Projektion, Scan, Natural Joins und Agreggation. Verschiedene Join-Varianten, Umbennenung, Division etc. werden aus Zeitgründen nicht betrachtet. Sie sind aber im Prinzip auch mit Skeletons abbildbar.

**Hardware**

Bei der Auswahl der Hardware handelt es sich um eine praktische Einschränkung. Bei der Hardware wird von einem Mehrprozessorsystem ausgegangen, welches sich einen gemeinsamen Speicher teilt. Die Zugriffszeit auf den Speicher ist von allen Prozessoren aus gleich (Uniform Memory Access). Dies erleichtert die Synchronisation der Daten. Hardwarespezifikationen wie Cachegrößen werden außen vor gelassen. Der vorgestellte Codegenerator kann potenziell, die entsprechenden Implementationen vorausgesetzt, auch Code erzeugen, der Cachegrößen, spezielle Architekturen oder Hardware ausnutzt. Dies benötigt einen größeren Implementierungsaufwand und wurde aufgrund der Zeitrestriktionen für zukünftige Arbeiten offen gelassen.

## 3.4 Intraoperator Parallelisiserung mit Skeletons

### 3.4.1 Skeletons

Zur Abstraktion und Strukturierung der parallelen Berechnungen werden in der Query-Engine sogenannte Skeletons [Col04] genutzt. Das sind parallele Abarbeitungsmuster, die in Funktionen (meistens höherer Ordnung) gekapselt sind. Dabei bestimmt das Skeleton das allgemeine Vorgehen der Berechnung und die übergebene Funktion, die auf den Anwendungszweck bezogene Spezialisierung und folgen damit dem „Dont't Repeat Yourself“ Prinzip [HT99]. Die Prinzip bedeutet möglichst viel Code wiederzuverwenden und keine Code-Klone zu verwenden. Diese behindern die Wartung (Fehler werden mitkopiert) und Erweiterung (alle Kopien müssen angepasst werden) von Software.

Skeletons wurden entwickelt, da in vielen Anwendungsgebieten ähnliche Parallelisierungmuster sichtbar wurden und damit die Erstellung von parallelisiertem Code effizienter und flexibler gestaltet werden konnte als vorher. Sie sind so abstrakt, dass keine Details der Implementation, wie die Datenverteilung oder die Synchronisation zwischen den Recheneinheiten, nach außen sichtbar sind. Das ist nicht nötig, da der eingebettete Compiler die Details automatisch nach der gegebenen Konfiguration auswählt. Dabei können sie parallel, aber auch sequentiell implementiert werden. Bei den Implementationen für Skeletons muss darauf geachtet werden, dass gleiche Argumente gleiche Ergebnisse erzielen. Die Art und Weise der Berechnung ist nicht relevant, aber optimalerweise für die jeweilige Anfrage effizient. Ein weiterer Vorteil ist, dass das Ausgangsprogramm dadurch sequentiell zu gestalten ist. Entwickler die nicht an den Parallelisierungsmodulen arbeiten, werden von der Paralleliserung abgeschirmt. Weitere Informationen zu Skeletons können in dem Buch von Rabhi und Gorlatch gefunden werden [RG03].

### Beispiele für Skeletons

Als nächstes stellen wir einige der geläufigsten Skeletons kurz vor. Allen gemein ist, das sie auf Datenstrukturen operieren, die mehrere gleichartige Elemente in einer Sequenz speichern, wie beispielsweise einer Liste oder einem Array. Weiterhin müssen diese nicht zwingend parallel implementiert werden, sondern sind auch sequentiell umsetzbar.

1. **Map** Bei dem Map Skeleton wird eine einstellige Funktion mit der Signatur `Input: A => Output: B` auf alle Datenelemente unabhängig voneinander angewendet. Die Signatur bedeutet, dass die berechneten Elemente einen anderen Typen aufweisen können, für den Fall das der Typ B nicht gleich dem Typen A entspricht. Da es keine Abhängigkeiten zwischen den einzelnen Elementen gibt, können die Daten beliebig auf die Prozessoren verteilt werden und die Funktionswerte berechnet werden. Nach Funktionsanwendung werden die einzelnen Ergebnisse von einer Berechnungseinheit wieder zusammengefügt.

   *Map kann also kurz dadurch charakterisiert werden, dass die Anzahl der Elemente sich nicht ändern, sondern höchstens deren Typ.*

   Eine Variation stellt das Skeleton FlatMap dar, welches man nutzt, wenn die Eingabefunktion für einzelne Elemente der Grundmenge wieder eine Datenstruktur vom Typen der Grundmenge selbst erzeugt. Angenommen unsere Datenstruktur ist eine einfache Liste mit Elementen. Die Funktion erzeugt für ein Element wieder eine Liste. Das Ergebnisse wäre ein Liste von Listen. Bei FlatMap werden die inneren Listen nach der Funktionsanwendung zusammengefügt, sodass die Verschachtelung aufgelöst wurde. Das Ergebnis hat dann mehr Elemente als die Ausgangsmenge.

2. **Filter** Die Filter Operation nimmt als Parameter eine Funktion vom Typen `Input: A => Output: Bool`. Diese wird genutzt, um jedem Element einen booleschen Wert zuzuordnen. Je nach Ausgang der Auswertung des Prädikats wird das Element entfernt. Dazu verteilt ein Prozessor die Grundmenge wieder auf die zur Verfügung stehenden Berechnungseinheiten und sortiert in jeder Teilmenge Elemente aus. Danach fügt ein Prozessor die Teilergebnisse wieder zur gefilterten Gesamtmenge zusammen.

   *Filter ist dadurch charakterisiert, dass die Anzahl der Elemente sich verkleinert, aber nicht deren Typ.*

3. **Reduce** Reduce nimmt eine assoziative Funktion als Parameter, welche die Signatur `Input: (A,A) => Output: A` besitzt. Das bedeutet das immer zwei Elemente aus der Grundmenge zusammengefasst werden. Dieser Vorgang wird mit der Ergebnismenge wiederholt solange bis die ganze Datenstruktur auf einen Wert reduziert wurde. Dieses Verhalten ähnelt SQL Spaltenfunktionen, bei denen die Werte einer ganzen Spalte zu einem zusammengefasst werden. Möchte man Spaltenfunktionen umsetzen sollte man dieses Skeleton nutzen. Aus Zeitgründen wird eine Umsetzung nicht in dieser Arbeit gezeigt und für weitere Erweiterungen offen gelassen.

   *Charakteristisch für Reduce ist das Zusammenfassen mehrerer Elemente zu einem einzigen Wert desselben Typs.*

4. **Zip** Eine Besonderheit beim Zip-Skeleton ist, dass es das einzige Skeleton in dieser Auswahl ist, welches mit zwei Datenstrukturen arbeitet. Dabei werden die korrespondierenden Elemente (z. B. jeweils die beiden ersten, zweiten, etc.) aus beiden Datenstrukturen zu einem Element in der Ergebnisstruktur verknüpft. Die konkrete Art und Weise der Verknüpfung kann mithilfe einer Funktion festgelegt werden, die dem Skeleton übergeben wird.

Komplexere Muster ergeben sich beispielsweise durch geschicktes kombinieren dieser und weiterer Skeletons miteinander.

### 3.4.2 Umsetzung von Operatoren mithilfe von Skeletons

Als nächsten Schritt müssen die Skeletons den Datenbankoperatoren zugeordnet werden. Jeder Operator berechnet das Ergebnis für die gesamte Eingabe auf einmal und gibt es an den nächsten Operatoren weiter (Push). Mithilfe der Charakterisierungen der Skeletons aus dem vorhergehenden Ausschnitt, sind diese nun auf Operatoren abbildbar, indem deren typischen Eigenschaften untersucht werden, wie beispielsweise Eingaben und Ausgaben.

#### Projektion

Die Projektion kann auf das Map-Skeleton abgebildet werden. Die Menge der Tupel unterscheidet sich zwischen der Ausgangs- und Ergebnisrelation nicht. Weiterhin wirkt die Projektion auf jedes Tupel unabhängig voneinander und führt dieselbe Veränderungen an dem Tupel aus. Dabei muss die Funktion für das Map-Skeleton so konstruiert sein, dass sie aus dem Eingabetupel, ein neues Tupel konstruiert, von dem die nicht mehr benötigten Attribute entfernt wurden.

#### Scan

Der Scan Operator entspricht dem Filter-Skeleton. Die Anzahl der Tupel verringert sich oder bleibt gleich. Einzelne Tupel werden anhand bestimmter Eigenschaften aussortiert und nicht weiter modifiziert. Das Prädikat des Filter-Skeletons entspricht dann dem Prädikat des Scan Operators.

#### Natural-Joins

Natural-Joins können naiv mit einer Kombination aus Verkettung und Verschachtelung von flatMap, map und filter wie folgt dargestellt werden: `table1.flatMap (tuple1 => table2.map(tuple2 => (tuple1, tuple2))).filter(Vergleich)`. Diese funktionale Schreibweise bedeutet, dass zuerst das Kreuzprodukt der beiden Tabellen gebildet wird und danach erst die relevanten Tupel aussortiert werden. Das ist ineffizienter als ein Nested-Loop-Join. Aus diesem Grund wird Join ein eigenständiges paralleles Muster, um damit den zahlreichen bekannten parallelen Implementationen gerecht zu werden [BTAÖzsu14, AKN12].

#### Aggregation

Die Aggregation kann mit dem Skeleton GroupByReduce umgesetzt werden. Wie der Name bereits andeutet, handelt es sich um ein GroupBy zum gruppieren der Tupel nach einer bestimmten Eigenschaft. Das Reduce wird dann genutzt, um die einzelnen Gruppen mithilfe der geforderten Aggregationsfunktion auf einen Wert zu reduzieren.

### 3.4.3 Abstraktionsebenen

Skillcorn und Talia [ST98] unterteilen Modelle für parallele Berechnungen in sechs Abstraktionsebenen ein, die sich darin unterscheiden, wieviel Details der Parallelisierung sie nach außen sichtbar machen. Skeletons fallen dabei in die höchste Abstraktionsstufe. Diese sind deklarativ orientiert, sodass es um die Art der Berechnung geht und nicht um deren genaue Umsetzung. In den Zwischenrepräsentation des Anfragecompilers werden die Skeletons zu imperativen parallelen For-Loops transformiert, in denen die entfalteten Funktionen der Skeletons den Schleifenkörper darstellen. Diese müssen bei der Codegenerierung in ein Codefragment mit einem konkreten parallelen Programmiermodell (Cilk [BJK+95], Threading Building Blocks, OpenMP [DM98], Pthreads [oEE96], MPI [WD96]) transformiert werden. Solche Programmiermodelle sind auf den mittleren Abstraktionen angesiedelt und abstrahieren selbst nur Teilaspekte der Paralleliserung. Intern arbeiten sie direkt mit Threads. Diese sind im generierten Code nicht ersichtlich, da nur die Funktionen des verwendeten Programmiermodells enthalten sind, obwohl es potenziell möglich wäre Code direkt für Threads zu generieren.

### 3.4.4 Nachteile von Skeletons

Ein Problem bei Skeletons ist, das sie nur schwer zu kombinieren und verschachteln sind. Mit parallelem Code ist es schwieriger die Berechnungen der Operatoren zu Pipelines zusammenzufassen, so wie es Neumann [Neu11] bereits für sequentielle Berechnungen gezeigt hat. Einzelne parallele Codefragmente können sich gegenseitig beeinflussen. Ein einfaches Beispiel ist die Verschachtelung von zwei parallelen For-Schleifen die aus zwei Codefragmenten entstanden ist. Die Anzahl der erstellten Threads multipliziert sich, obwohl das eventuell nicht beabsichtigt war. Der Compiler hätte dies erkennen müssen und die Paraleliserung aus der inneren For Schleife entfernt. Das ist ein schwieriges Problem, welches aber gelöst werden muss, um den Paralleliserungsoverhead möglichst auf einen Niveau zu halten, welches mit der daraus gewonnen Leistung in einem Verhältnis steht.

#### Verkettung von Skeletons

Mithilfe von Fusionsregeln [GLPJ93] können nun bestimmte Kombinationen von Skeletons zu einem einzigen neuen Skeleton zusammengefasst werden, um Koordinationsaufwand zu reduzieren und das speichern Zwischenergebnissen einzuschränken. In der funktionalen Programmierung ist das Problem bereits seit Längerem ein Thema [Wad88].

**Verschachtelung von Skeletons**

Bei Joins und Unteranfragen kommt es zur Verschachtelung von Skeletons. Das führt aktuell dazu, dass die Joins und Unteranfragen einzeln nacheinander berechnet werden, anstatt sie effizient z. B. in einer verschachtelten Schleife zu berechnen.

Eine Möglichkeit dies zu beheben, ist ein neues Skeleton `join2` zu erstellen, welches die Verbindung von drei Relationen gleichzeitig parallel abarbeitet, anstatt zwei separater Joins. Dieses ist dann mit zwei Relationen als Argument wie folgt aufrufbar: `table1.join2(table2, table3)` und wird vom Compiler automatisch ausgetauscht wenn die passenden Regeln dafür implementiert wurden. Für vier Joins ist eine join3 Funktion nötig usw. Die Erweiterbarkeit ist dadurch limitiert und für jedes neue Muster wächst die Zahl der Fusionsregeln stark an. Coudarcher et al. [CDS+05] zeigen eine Lösung, die sie in der Bildverarbeitung angewendet haben. Sie bilden die Skeletons auf ein allgemeineres Skeleton ab, welches wiederum gut kombinierbar ist. Dieterle et al. [DHL10] versuchen das Problem durch das Austauschen von einzelnen Datensätzen zwischen den Skeletons zu verbessern.

## 3.5 Integration des Variantengenerators

Der Variantengenerator kann durch Komposition von Traits verschiedene Query-Engines bereitstellen. Dabei sind verschiedene konkrete Umsetzungsmöglichkeiten dieses Mechanismus mit unserem Ansatz denkbar.

Die einfachste ist, dass alle möglichen Kombinationen bereits bei der Kompilierung der Datenbank bekannt sind. Entweder durch manuelles zusammensetzen durch den Programmierer oder durch Makros [Bur13], die automatisch Zusammenstellungen nach einem vorgegebenen Muster erzeugen. Makros sind Programme die zur Kompilierzeit aufgerufen werden und automatisch Quelltext erzeugen. Ein naives Muster wäre beispielsweise: „Erzeuge alle möglichen Varianten“. Im Hintergrund werden für die Klassen, die mit Traits versehen wurden (also Varianten), neue Klassen erstellt, die kompiliert werden müssen. 100 verschiedene Kombinationen ergeben letztendlich 100 Klassen die kompiliert werden müssen. Entscheidet man sich für diese Lösung, sollte klar sein, dass zur Laufzeit lediglich aus den, bei der Kompilierung der Datenbank festgelegten Varianten, gewählt werden kann.

Zur Laufzeit der Datenbank ist das erstellen neuer Klassen nur schwieriger zu erreichen. Diese stützen sich auf das Decorator-Pattern [GHJV94]. Ein einfaches Beispiel dafür ist eine fiktive Klasse `A` und ein Trait `B`. Aus A und B wird jeweils ein Objekt instanziert. Diese beiden Objekte werden, in ein zur Laufzeit dynamisch erstelltes Objekt `C` als Property integriert, welches das kombinierte Interface von `A` und `B` besitzt. Wird eine `B`-Methode bei dem Objekt `C` aufgerufen, dann ruft `C` lediglich diese Methode bei Objekt `B` auf. Es ist aber fraglich, ob diese Möglichkeit für den Anwendungszweck ausreichend ist.

Eine andere Möglichkeit ist es, den Scala Compiler zur Laufzeit mit der gewünschten Komposition aufzurufen und dann mithilfe von Reflexion/Introspektion die kompilierte Klasse herauszufinden, zu laden und Objekte (also die konfigurierten Query-Engines) zu erzeugen.

## 3.6 Zusammenfassung

In diesem Kapitel wurde vorgestellt wie unsere Query-Engine den parallelisierten Code generiert, kompiliert und ausführt. Weiterhin wurden die Anbindungsmöglichkeiten an einen Variantengenerator aufgezeigt. Wir haben gezeigt, wie die Parallelisierung über verschiedene Ebenen hinweg, innerhalb eines Operators abstrahiert werden kann und welche Probleme es dabei gibt. Im nächsten Kapitel werden die Details der Umsetzung beschrieben.

# 4. Implementation

In diesem Kapitel geht es um die konkrete Umsetzung der Skeletons und Datenbankoperatoren mit dem LMS Framework. Dazu wird detailliert auf die Implementierung der einzelnen Operatoren und deren Grundbausteine eingegangen.

## 4.1 Ausgangspunkt

Als Grundlage wurde der DSLDriver aus dem LMS Tutorial genutzt. Dieses ist unter https://github.com/scala-lms/tutorials zu finden. Der DSLDriver bietet Möglichkeiten aus einfachen DSL-Programmen direkt primitiven C-Code generieren, kompilieren und auszuführen zu können. Das ist mit dem LMS Framework nicht direkt ohne weiteres möglich. Weiterhin nutzen wir das LMS Framework in der Version 0.3, dessen Sourcecode unter https://github.com/TiarkRompf/virtualization-lms-core zu finden ist. Als Scala Compiler kommt der Scala Virtualized Compiler in der Version 2.10.2 zum Einsatz.

## 4.2 OpenMP

Zur Umsetzung von des Parallelismus wird OpenMP [DM98] genutzt. Dabei handelt es sich um eine Application Programming Interface (API) zur Shared-Memory Programmierung für Multiprozessoren für C und C++. Wir haben uns für OpenMP entschieden weil damit Parallelisierung einfach und effizient umzusetzen ist, ohne manuelle Threadverwaltung betreiben zu müssen. OpenMP unterstützt Task- und Datenparallelität. Für die Datenparallelität ist die parallelisierte For-Schleife das wichtigste Programmierkonstrukt. Im folgenden Abschnitt wird noch ausführlich darauf eingegangen.

### Setup und Anpassungen an den DSLDriver

Um OpenMP nutzen können wird ein kompatibler Compiler benötigt. Der auf OS X mitgelieferte Clang Compiler beherrscht OpenMP nicht ohne weiteres. Es empfiehlt sich den GCC nachzuinstallieren, der OpenMP umsetzen kann. Bei der Auswertung

```
1 trait ParallelRangeOps extends RangeOps {
2   def range_parallel_foreach(r: Rep[Range], f: (Rep[Int]) =>
        Rep[Unit])(implicit pos: SourceContext): Rep[Unit]
3 }
```

Quelltext 4.1: Interface der parallelen For-Loop

der generierten Codefragmente muss gegenüber dem Tutorial DSLDriver der Compiler getauscht und ein spezielles Flag `-fopenmp` für OpenMP gesetzt werden. In der C Source-Datei, welche generiert wird, muss weiterhin der OpenMP Header importiert werden.

## 4.3 For-Loop

Da die Skeletons von deklarativer Natur sind, müssen sie in eine Form gebracht werden, sodass sie mit einer imperativen Programmiersprache wie C abarbeitbar sind. Dazu musste eine geeignete Übersetzung gefunden werden. Da es sich bei allem Skeletons um Operationen auf uniformen Daten handelt, erscheint die For-Schleife kombiniert mit Arrays in C als geeignetes Programmierkonstrukt. Ein parallelisiertes For-Konstrukt in der Metasprache dient als Grundbaustein für die Skeletons.

Bei der Parallelisierung von For-Schleifen mit OpenMP muss immer die Anzahl der Schleifeniterationen vorab bekannt sein. Das ist ein Problem, da dadurch nicht die For-Loop für Arrays in der Metasprache überschrieben werden kann. Arrays der Metasprache werden und müssen in dynamische Allokationen übersetzt werden, da für größere Datenbankanfragen der Stack-Speicher knapp werden kann und zum Absturz des Programms führt. Für diese dynamischen Array-Allokationen ist nun die Größe nicht einfach mittels des bekannten Konstrukts `sizeof(x)/sizeof(*x)` ermittelbar. Dieses teilt die Größe eines Arrays durch die Größe eines Pointers, welcher in diesem Fall die selbe Größe hat wie ein einzelnes Element des Arrays. Für dynamische Arrays ist nur die letztgenannte Größe bekannt.

Als Konsequenz muss bereits in dem Metaprogramm die Endgröße der Arrays festgelegt sein. Als Hilfsmittel nutzen wir Scala Ranges. Ranges repräsentieren Int Werte innerhalb eines geschlossenen Intervalls mit vorgegebener Schrittweite. Eine Range hat als Parameter immer einen Startwert, einen Endwert und eine Schrittweite. Beispielsweise würde die Range mit dem Parameter 2 als Startwert und 10 als Endwert mit der Schrittweite 2 die Zahlen 2,4,6,8,10 enthalten. Auf dieser Range kann nun in der Metasprache eine foreach Funktion definiert werden. Diese führt jedes Element in der Range ein Funktion aus, die mit diesem Element arbeitet. Die Elemente der Range sind quasi die Indexe der For-Loop in der Objektsprache. Start- und Endwert sind dann auf den initialen Wert der For-Loop und der Endwert wird in die Abbruchbedingung mittels vergleich umgewandelt. Soviel zum konzeptionellen Standpunkt der parallelen For-Loop. Als nächstes betrachten wir die genaue Umsetzung mit LMS.

### Interface

Zuerst wid das Interface betrachtet, so wie es in Quelltext 4.1 abgebildet ist. Zeile 1 sagt aus, dass die bereits bestehenden Operationen auf Ranges nun um ein paral-

leles foreach erweitert wird. An der Signatur ist abzulesen, das diese Operation eine Funktion höherer Ordnung ist, da sie neben der Range selbst, noch eine Funktion erwartet, die ein Eingabewert erwartet und keinen Ausgabewert hat. Diese nutzen wir später um Array Elemente zu verändern und dann direkt wieder in das Array einzufügen. Durch dieses In-Place verändern wird kein Rückgabewert benötigt.

Alle Typen der Funktion und der übergebenen Funktion wurden mit `Rep` versehen, da die For-Loop und die Argumente der übergebenen Funktion am Ende im generierten Code erscheinen müssen. Letztendlich sollen damit die Tupel der Relation verarbeitet werden, die zum Zeitpunkt der Codegenerierung noch nicht bekannt sind.

### Implementierung

Die Implementierung unterscheidet sich nicht von herkömmlichen For Loops auf Ranges in LMS (Zeile 4-7). Zuerst wird ein Symbol für die Index-Variable der Loop angelegt (Zeile 5). Die Funktion höherer Ordnung wird ausgeführt und die Effekte werden vom Framework aufgezeichnet (Zeile 6). Das ist nötig, da nicht garantiert werden kann, dass die Funktion die übergeben wird ohne Seiteneffekte ist. So wie sie später genutzt wird, hat sie immer Seiteneffekte, da Arrays In-Place verändert werden. Die ausgewertete Funktion wird nun zusammen mit dem Start- und Endwert, sowie der Indexvariablen genutzt, um das korrespondierende Symbol (also den Knoten) in der Graphenstruktur anzulegen und direkt einzufügen (Zeile 7).

Die verbleibenden Methoden werden benötigt um Operationen die auf den Graphen vom Framework benötigt werden, um Analysen auszuführen, wie z. B. welche Symbole der Knoten bindet und dadurch nicht an andere Stellen verschoben werden darf. Beispielsweise wäre es ungünstig die Indexvariable aus der For-Loop an eine andere Stelle zu verschieben. Deshalb muss sie an den For-Loop Knoten gebunden werden (Zeile 21-24).

### Generator

Nachdem der Graph mit den Zwischenrepräsentationen analysiert wurde, übersetzt der Code-Generator die Symbole in ausführbaren Code. Quelltext 4.3 ist der Trait, der für die Übersetzung des Symbols in C Code der parallelen For-Loop verantwortlich ist. Zeile 7 ist dabei die Zeile, die für die Parallelisierung relevant ist. Zu sehen ist das Compiler Pragma von OpenMP für eine parallele For-Loop. Komplizierte Parameter, die Regeln, welche Variablen privat (jeder Thread hat eigene Kopie) oder verteilt (alle Threads arbeiten auf derselben Kopie) sind, werden nicht benötigt. Wir nutzen die default Einstellungen, bei denen alle Variablen die im Body vorkommen auf allen Threads geteilt werden. Das führt zu keinen Konflikten, da bisher nur auf Arrays gearbeitet wird und jeder Thread eigene Indexes zugewiesen bekommen, sodass es niemals Überschneidungen gibt bei den Arrayzugriffen. Wenn eine Schleife beispielsweise 12 Iterationen beinhaltet und auf 4 Threads verteilt werden soll, dann bekommt Thread 1 die Iterationen 1,5,9 und Thread 2 bekommt die Indexes 2,6,10 etc.

Der Generator-Trait der For-Loop soll den bestehenden C-Code Generator erweitern und wird deshalb folgendermaßen hinzugefügt: `trait ParallelDSLGenC extends DslGenC with CGenParallelRangeOps`.

```
1  trait ParallelRangeOpsExp extends RangeOpsExp {
2    case class RangeParallelForeach(start: Exp[Int], end: Exp[
       Int], i: Sym[Int], body: Block[Unit]) extends Def[Unit]
3
4    def range_parallel_foreach(r: Exp[Range], block: Exp[Int]
       => Exp[Unit])(implicit pos: SourceContext) : Exp[Unit]
       = {
5      val i = fresh[Int]
6      val a = reifyEffects(block(i))
7      reflectEffect(RangeParallelForeach(r.start, r.end, i, a)
         , summarizeEffects(a).star)
8    }
9
10   override def mirror[A:Manifest](e: Def[A], f: Transformer)
       (implicit pos: SourceContext): Exp[A] = (e match {
11     case Reflect(RangeParallelForeach(s,e,i,b), u, es) =>
         reflectMirrored(Reflect(RangeParallelForeach(f(s),f(e
         ),f(i).asInstanceOf[Sym[Int]],f(b)), mapOver(f,u), f(
         es)))(mtype(manifest[A]), pos)
12     case _ => super.mirror(e,f)
13   }).asInstanceOf[Exp[A]]
14
15
16   override def syms(e: Any): List[Sym[Any]] = e match {
17     case RangeParallelForeach(start, end, i, body) => syms(
         start):::syms(end):::syms(body)
18     case _ => super.syms(e)
19   }
20
21   override def boundSyms(e: Any): List[Sym[Any]] = e match {
22     case RangeParallelForeach(start, end, i, y) => i ::
         effectSyms(y)
23     case _ => super.boundSyms(e)
24   }
25
26   override def symsFreq(e: Any): List[(Sym[Any], Double)] =
       e match {
27     case RangeParallelForeach(start, end, i, body) =>
         freqNormal(start):::freqNormal(end):::freqHot(body)
28     case _ => super.symsFreq(e)
29   }
30
31 }
```

Quelltext 4.2: Implementierung der parallelen For-Loop

```
1  trait CGenParallelRangeOps extends CGenEffect with
       ParallelGenRangeOps {
2    val IR: DslExp
3    import IR._
4
5    override def emitNode(sym: Sym[Any], rhs: Def[Any]) = rhs
       match {
6      case RangeParallelForeach(start, end, i, body) =>
7        gen"""#pragma omp parallel for
8             |for(int $i=$start; $i < $end; $i++) {
9               |${nestedBlock(body)}
10            |}"""
11
12     case _ => super.emitNode(sym, rhs)
13   }
14 }
```

Quelltext 4.3: Generator-Trait der parallelen For-Loop

## 4.4 Map

Mithilfe der definierten parallelen For-Loop kann nun eine parallele Variante des Map Skeletons implementiert werden. Quelltext 4.4 ist das korrespondierende Code Fragment dafür. Die Map Funktion nimmt ein Array und eine Funktion als Eingabeparameter, sowie die Größe des Arrays die wie bereits erwähnt leider für die Konvertierung in C benötigt wird, die allerdings in einer Table klasse kapselbar sind, welche die Properties data und size hat. Wäre Scala die Objektsprache würde dieser Parameter wegfallen, da dort nicht zwischen dynamisch und statischer Allokation unterschieden werden kann und die length Methode auf einem Array die Größe ermitteln kann. Das Array und dessen Größe sind bei der Codegenerierung noch nicht bekannt und werden somit als Variablen Teil des generierten Codes. Die übergeben Funktion allerdings ist bereits bekannt und kann ausgewertet werden (deren Werte allerdings nicht). Da die Funktion mit Rep Werten (also verzögerten) arbeitet kann man die Semantik leicht mit verwechseln mit Rep[Int => Int]. In diesem Fall ist die Funktion nicht bekannt und dementsprechend an dieser Stelle noch nicht auswertbar. Das sind allerdings zwei unterschiedliche Sachverhalte, welche nicht verwechselt werden dürfen.

Die Arraygröße wird genutzt um ein Range zu erstellen, das bei 0 beginnt und bis zur Arraygröße geht (Zeile 2), da Arrays in den gängisten Programmiersprachen bei 0 mit der Indexierung starten. Die Funktion für die parallele For-Schleife sieht vor, dass für jeden Index der Range, die Funktion auf das Array Element dieses Indexes ausgeführt wird und das Ergebnis sofort wieder bei diesem Index zugewiesen wird (In-Place).

In Quelltext 4.5 wurde ein kurzes Programm welches Map nutzt mit dem generierten Code gegenübergestellt. Es ist gut zu sehen, dass im generierten Code die ineinandergeschachtelten anonymen Funktionen vorab ausgewertet wurden ohne das Ergebnis des Programms zu ändern.

```
1 def map(array: Rep[Array[Int]], size: Rep[Int], function:
    Rep[Int] => Rep[Int]): Rep[Array[Int]] = {
2   range_parallel_foreach(0 until size, {i =>
3     array(i) = function(array(i))
4   })
5   return array
6 }
```

Quelltext 4.4: Implementierung des Map Skeletons

```
1  //Scala DSL Programm
2  val test2: Rep[Int] => Rep[Int] = {_ * 2}
3  val test: Rep[Int] => Rep[Int] = {x => test2(x * 2)}
4  val result = map(y, size, test)
5
6  //generierter Code
7  #pragma omp parallel for
8  for(int x7=0; x7 < 100; x7++) {
9  int32_t x8 = x1[x7];
10 int32_t x9 = x8 * 2;
11 int32_t x10 = x9 * 2;
12 x1[x7] = x10;
13
14 }
```

Quelltext 4.5: Generierter paralleler Code für ein einfaches DSL-Programm

## 4.5 Selektion

Der Datenbankoperator Selektion lässt sich auf das Filter Skeleton abbilden, so wie es in Quelltext 4.6 implementiert ist. Das Prädikat der Selektion wird als Funktion die einen booleschen Wert zurückgibt interpretiert, mit dem das Filter-Skeleton arbeiten kann.

Für jede Iteration wird der Ausgang der Prädikatsprüfung in dem Array `flags` vermerkt (Zeile 15-17). Da die Iterationen unabhängig voneinander sind, können sie mit einer parallelen For-Schleife genutzt werden.

Ein Problem bei der parallelen Filterung das die Ergebnisgröße vorab nicht bekannt ist und je nach Filterungsattribut unterschiedlich ausfallen kann.

Um ein Array passender Größe für das Berechnungsergebnis bereitstellen zu können muss die Präfixsumme aus den gespeicherten Prädikatsprüfungen ermittelt werden.

Die Präfixsumme einer Liste, ist eine Liste, die die selbe Größe hat wie die Ausgangsliste. Die Elemente der Ergebnisliste werden berechnet, indem die Summe aller Elemente bis zu dem gesuchten Index in der Ausgangsliste gebildet wird. Beispielsweise wird die Präfixsumme der Liste `(4,7,9,8)` berechnet, indem man die Liste mit den Termen `(4,4+7,4+7+9,4+7+9+8)` welches die Liste `(4,11,20,28)` ergibt.

```
1  def prefixSum(input : Rep[Array[Int]], size: Rep[Int]): Rep[
       Array[Int]] = {
2    var output: Rep[Array[Int]] = NewArray[Int](size)
3    output(0) = 0
4    range_foreach(1 until size, {i =>
5      output(i) = output(i - 1) + input(i - 1)
6    })
7    output
8  }
9
10 def filter(array : Rep[Array[Int]], size: Rep[Int], function
       : Rep[Int] => Rep[Boolean]): (Rep[Array[Int]], Rep[Int])
       = {
11   var position: Rep[Int] = unit(0)
12   var flags: Rep[Array[Int]] = NewArray[Int](size)
13
14   range_parallel_foreach(0 until size, {i =>
15     if (function(i)) {
16       flags(i) = 1
17     }
18   })
19
20   val sums = prefixSum(flags, size)
21
22   val output: Rep[Array[Int]] = NewArray[Int](sums(size - 1)
       )
23
24   range_parallel_foreach(0 until size, {i =>
25     if (flags(i) == 1) {
26       output(sums(i)) = array(i)
27     }
28   })
29   return (output, sums(size - 1))
30 }
```

Quelltext 4.6: Filter mit Präfixsumme

Weiterhin wird sie benötigt, um die Ergebnisse an die passende Stelle speichern zu können (Zeile 24-28).

## 4.6 Block-Nested-Loop-Join

Der Block-Nested-Loop-Join (BNLJ) lässt sich mithilfe der paralleliserten For-Loop ähnlich der sequentiellen Variante implementieren. An der Implementation in Quelltext 4.7 kann man erkennen, dass lediglich die äußere Schleife parallel ist und die innere Schleife sequentiell verbleibt. Es können auch beide parallelsiert werden, dann müssten zusätzliche OpenMP Anweisungen genutzt werden wie z. B. „omp thread

```
1 def join(table1: Table, table2: Table): Unit = {
2   range_parallel_foreach(0 until table1.size, {i =>
3     range_foreach(0 until table2.size, {j =>
4       if (table1.data(i) == table2.data(j)) {
5         println(table2.data(j))
6       }
7     })
8   })
9 }
```

Quelltext 4.7: Paralleler Block-Nested-Loop-Join

limit“ oder „omp nested“, die verschachtelte Paralleliserung deaktivieren oder die Anzahl der maximalen Threads einschränken. Dies soll verhindern, dass sonst zuviele Threads generiert werden bei großen Eingaberelationen. Der Verwaltungsaufwand für diese Threads kann sich negativ auf die Performance auswirken. Aus diesen Gründen haben wir uns dafür entschieden die innere Schleife von vornherein sequentiell zu gestalten.

## 4.7 Sort-Merge-Join

Der aufwendigste Teil beim Sort-Merge-Join ist die initiale Sortierung der Eingaberelationen. Zur effizienten Umsetzung eines Sort-Merge-Join ist es sinnvoll zuerst die Sortierung zu paralleliseren.

### Sort

Für die Sortierung wurde ein eigenes Symbol in der Graphensstruktur angelegt. Wird bei der Generation auf ein sort Symbol gestoßen, dann wird lediglich die Zeile `sort(pointerToArray, arraySize)` eingefügt. Der fertige Code ist schon statisch in dem C File enthalten und wird nicht dynamisch generiert, das erleichtert die Implementierung komplizierter Algorithmen, ist aber dafür auch nicht so flexibel. Im Falle des Sort Algorithmus wird auch keine Flexibilität benötigt. Das generierte C-Datei hat eine statische Komponente, die immer eingesetzt wird, wie z. B. der Import der C-Standard-Bibliothek, die immer benötigt wird. Weiterhin können gängige Algorithmen dort eingesetzt werden. In unserem Fall steht dort der Code aus Quelltext 4.8. Zum Schluss folgt dann der dynamisch generierte Teil, der dann die Algorithmen und Importe aus dem Anfangsteil benötigt. Der Nutzer des Sort-Symbols in dem Metaprogramm, muss darauf Vertrauen, dass im generierten Teil der Sort Algorithmus zu finden ist. Ansonsten kommt es zu einem Kompilierungsfehler. In der Testdatenbank des LMS-Tutorial wurde das z. B. bei den Hashing Algorithmen für den Hash-Join ebenfalls auf diese Weise umgesetzt. Die Hash Algorithmen liegen dort ebenfalls statisch in der generierten C Datei vor. Das verhindert, dass komplizierte Algorithmen mit in die DSL aufgenommen werden müssen.

Der parallele Code für die Sortierung ist in Quelltext 4.8 zu sehen. Zuerst werden die Anzahl an Threads ermittelt und gesetzt, die default der Anzahl entsprechen, die der Prozessor maximal verarbeiten kann (Zeile 5-6). Danach werden die Indizes auf die

```
1  int* sort(int *input, int size) {
2    int i;
3    int *a = input;
4    // set up threads
5    int threads = omp_get_max_threads();
6    omp_set_num_threads(threads);
7    int *index = (int *)malloc((threads+1)*sizeof(int));
8    for(i=0; i<threads; i++) index[i]=i*size/threads; index[
       threads]=size;
9
10   /* Main parallel sort loop */
11 #pragma omp parallel for private(i)
12   for(i=0; i<threads; i++) qsort(a+index[i],index[i+1]-index
       [i],sizeof(int),CmpInt);
13   /* Merge sorted array pieces */
14   if( threads>1 ) arraymerge(a,size,index,threads);
15   return input;
16 }
```

Quelltext 4.8: Parallele Sortierung mit OpenMP

Threads aufgeteilt, sodass jeder später ein zusammenhängendes Stück verarbeiten kann, was normaler bei der For-Loop nicht möglich wäre (Zeile 7-8). Danach wird jedem Thread das zusammenhängende Stück zugewiesen und parallel auf jedem Thread ein Quicksort aus der C-Standard-Bibliothek auf das Teilstück ausgeführt. Zum Schluss müssen die sortierten Teilstücke zu einem sortierten Gesamtergebnis zusammengefügt werden (Zeile 14).

#### Merge

Zum zusammenfügen der sortierten Relationen nutzen wir einen Standard Merge-Algorithmus, welcher in Quelltext 4.9 abgebildet ist (Zeile 24-46) und beispielsweise in dem Buch von Saake et al. [SSH11] zu finden ist. Dieser kann zur weiteren Performancesteigerung zusätzlich parallelisert werden, so wie von Cormen et al. [CLRS09] gezeigt wurde.

## 4.8 Herausforderungen

Bei der Übersetzung von Scala in C kommt es auch zu einem Wechsel der Art der Speicherverwaltung. Scala nutzt den JVM Garbage Collector. Dieser analysiert automatisch zur Laufzeit des Programms, welche Objekte noch referenziert werden. Wird ein Objekt nicht mehr referenziert, dann wird es freigegeben. In C ist dies ein manueller Prozess der vom Programmierer selbst vorgenommen werden muss. Das bedeutet, dass alle dynamisch vom Speicher angeforderten Objekte nicht gelöscht werden. Das betrifft in unserer Implementation alle Arrays die angelegt werden. Bei mehreren Datenbankanfragen hintereinander, kann das schnell den Speicher füllen und zum Programmabsturz führen. Eine Lösung wäre es passende Symbole für

die manuelle Speicherverwaltung für das DSL Programm anzulegen. Ist die Objektsprache und die Metasprache Scala können diese Symbole bei der Codegenerierung einfach ignoriert werden. Ist die Objektsprache C müssen die Symbole wie z. B. free zum freigeben von Speicher in den generierten an die richtige Stelle eingewoben werden. Dadurch ist die praktische Einsatzfähigkeit außerhalb von Testumgebungen derzeit noch nicht gegeben.

```
def join(S: Table, R: Table): Unit = {

  var sIndex = unit(0)
  var rIndex = unit(0)

  //Sort both input
  sort(S.data, size)
  sort(R.data, size)

  var s = S.data(varIntToRepInt(sIndex))
  var r = R.data(varIntToRepInt(rIndex))

  def nextR: Unit = {
    rIndex+=1
    r = R.data(varIntToRepInt(rIndex))
  }

  def nextS: Unit = {
    sIndex+=1
    s = S.data(varIntToRepInt(sIndex))
  }

  //Merge
  while (sIndex != S.size - 1 && rIndex != R.size - 1) {

    if (varIntToRepInt(r) > varIntToRepInt(s)) {
      nextS
    } else {
      if (varIntToRepInt(r) < varIntToRepInt(s)) {
        nextR
      } else {
        nextS
        while (sIndex != S.size - 1 && r == s) {
          //Ausgabe von (tr,ts)
          nextS
        }
        nextR
        while (rIndex != R.size - 1 && r == s) {
          //Ausgabe von (tr,ts)
          nextR
        }
        nextR
        nextS
      }
    }
  }

}
```

Quelltext 4.9: Sort-Merge-Join als DSL-Programm

# 5. Evaluierung

Im vorherigen Kapitel wurde die Implementierung, der vorgestellten Abstraktion, den Skeletons, zur Implementierung einiger Operatoren für die Query-Engine beschrieben. Dieser Prototyp dient in diesem Kapitel als Grundlage für die Evaluierungen, die die Eignung von Skeletons als Abstraktion in Query-Engines untersuchen.

Im folgenden Abschnitt wird dargestellt, wie wir bei der Evaluation vorgehen. Anschließend werden den gewählten Ansatz vergleichend diskutieren und die Messwerte und deren Interpretation präsentieren.

## 5.1 Methodik

In dieser Evaluation verfolgen wir eine zweigeteilte Strategie, um den entwickelten Ansatz näher zu untersuchen. Zur qualitativen Evaluation nutzen wir die Diskussion als typischen Vertreter dieser Art. Wir konzentrieren uns dabei auf die spezielle Konfiguration von Skeletons und LMS für DBMS. Aus Mangel an Zeit und geeignetem Personal können wir keine Langzeitstudien mit professionellen Datenbankentwickler und Parallelisierungsexperten durchführen, um Rückmeldung über die praktischen Einsatz zu erhalten wie es z. B. Nanz [NWDS13] für verschiedene Parallelisierungsmethodiken gemacht hat. Wir vergleichen auch nicht konkrete Parallelisierungsframeworks gegeneinander, wie beispielsweise OpenMP gegen PThreads. Solche Vergleiche wurden unter anderem von Tousimojarad und Vanderbauwhede [TV14], sowie Sacnehz et al. [SFS+13] angestellt. durchgeführt. Auch werden Probleme, Nachteile und Vorteile von Staged-Programming allgemein nicht diskutiert.

Zum Vergleich werden verschiedene Alternativen zu Skeletons vorgestellt, aus deren Sicht anhand von aufgestellten Kriterien argumentiert wird. Dies wird dabei immer auf den Datenbankkontext bezogen. Auf die Nachteile des LMS Frameworks allgemein gehen wir nicht ein. Das Hauptmotiv dafür ist, dass Performancetests in unserem Fall nicht ausreichen. Sobald Abstraktionen in Software genutzt werden, müssen sie auch auf das Einsatzgebiet abgestimmt sein.

Die zweite Strategie ist die quantitative Evaluation, die mithilfe von Testmessungen umgesetzt wird. Dort wollen wir überprüfen, ob der generierte Code auch korrekte Ergebnisse erzeugt und den Leistungserwartungen entspricht, indem wir sequentielle Versionen gegen die parallelen Varianten vergleichen.

## 5.2 Skeletons im Kontext von DBMS und LMS

In anderen Domänen wie physikalischen Simulationen oder Maschinellem Lernen werden Skeletons bereits mit großem Erfolg eingesetzt, jedoch liegt der Fokus in dieser Arbeit speziell auf parallele DBMS, welche mit LMS arbeiten.

### Alternativen

Skeletons sind nicht das einzige Entwurfsmuster um Parallelisierung mit LMS umzusetzen. Um unseren Ansatz besser einschätzen zu können, beschreiben wir zuerst die Alternativen die mit LMS umgesetzt werden können.

Eine mögliche Alternative ist, eine DSL für jedes Parallelisierungsframework zu erstellen, in denen deren Schnittstelle enthalten ist. Die Symbole der DSL werden in das C-Äquivalent übersetzt. Diese können direkt zur Implementierung der Operatoren verwendet werden. Das entspricht der Programmierweise die man in C verwenden würde, nur wie im unserem Fall aus Scala heraus. Im weiteren Verlauf nennen wir dies den direkten Ansatz, da er keinerlei parallele Abstraktionen nutzt (bis auf die des Frameworks, wenn welche Angeboten werden). Beispielsweise würde ein DSL Interface für OpenMP das Parallelisierungspragma anbieten, welches mit den gewohnten Parametern manipulierbar sein muss. Dieses muss dann in dem Scala DSL Programm auch vor einer For-Schleife verwendet werden, so wie man es in C auch verwenden würde. Die Entwickler der DSL-Implementation müssen allerdings beachten, dass die Pragma Methode als Effekt deklariert wird. Er wird sonst automatisch vom LMS-Framework entfernt oder eventuell verschoben, da er für sich kein Ergebnis liefert. Beides führt nicht zum gewünschten Parallelisierungsergebnis.

Ein andere Möglichkeit ist es, die Parallelität der Abstraktionen mit Parametern zu justieren. Beispielsweise könnte den Skeletons die Anzahl an der gewünschten Threads als Parameter hinzugefügt werden. Damit ist die Paralleliserung von den Nutzern der Abstraktion besser kontrollierbar. Dies wird gelegentlich als sogenannte „Leaky Abstractions“ [Spo04] bezeichnet.

In unserer Implementation sind die meisten Skeletons mithilfe von For-Loops definiert, die Code in dem jeweiligen gewünschten Framework generieren. Um die Skeleton-Abstraktion zu implementieren nutzen wir wiederum eine Abstraktion von For-Schleifen. Die Framework primitiven sind nicht von uns in eine DSL übertragen worden.

Darüberhinaus sind Mischformen möglich. Beispielsweise könnten wir die For-Loop mithilfe der Methoden aus der DSL eines Parallelisierungsframework wie bei der direkten Methode implementieren. Damit wären es dann insgesamt drei Abstraktionsschichten (Skeleton, For-Loop, DSL). Diese Art bezeichnen wir im folgenden als Layered, da mehrere Abstraktionschichten genutzt werden.

Die Möglichkeiten der Kombination verschiedener Ansätze sind groß und sicherlich sind weitere Abstraktionsstrategien möglich, allerdings sind Skeletons das abstrakteste, was an dieser Stelle theoretisch möglich ist, sodass auszuschließen ist, dass Parallelisierung auf einem höheren Abstraktionsniveau nutzbar gemacht werden kann. Nur Nutzer des Variantengenerators können später noch abstrakter arbeiten, indem sie die Paralleliserung für bestimmte Operatoren aktivieren oder deaktiveren.

Die folgende Auflistung fasst die, in diesen Abschnitt vorgestellten Entwurfsmuster kurz zusammen:

**Direkt/DSL** DSL mit einem Interface, dass sich an dem Interface von Parallelisierungsframeworks orientiert

**Skeletons** Implizite parallele Abstraktionen

**Leaky Abstractions** Abstraktionen mit Parallelisierungsparametern in der Signatur

**Layered** Parallelisierungsabstraktionen mithilfe von weniger abstrakten Parallelisierungsabstraktionen implementieren

### Kriterien

Nachdem Alternativen aufgezeigt wurden, stellen wir eine Auswahl von Kriterien vor, um verschiedene Entwurfsmuster bewerten zu können. Diese Zielen darauf ab, die Eignung für das Einsatzszenario in parallelen DBMS mit LMS einzuschätzen.

**Optimierungspotenzial** Mit diesem Kriterium soll Eingeschätzt werden, wie gut die regeln einer DSL genutzt werden können, um Berechnungen zu vereinfachen und zu beschleunigen.

**Aufwand** Eine relative Schätzung wie Aufwändig es ist, dieses Entwurfsmuster umzusetzen.

**Anbindungsmöglichkeiten an einen Variantengenerator** Wie gut kann das Entwurfsmuster mit einem Variantengenerator, so wie er im Implementationskapitel beschrieben wurde, zusammenarbeiten?

**Abstraktionsniveau** Auf welchen Level ist die Abstraktion angesiedelt? Dies ist relevant, da wir die Arbeit der Programmierer durch Abstraktion erleichtern möchten. Weiterhin bietet ein höheres Abstraktionslevel auch ein höheres Optimierunspotenzial.

**Flexibilität** Wie flexibel ist das Entwurfsmuster, im Sinne von Erweiterbarkeit der Programme, die dieses Nutzen.

### Diskussion

Im folgenden erläutern wir Vor- und Nachteile, die Skeletons speziell in der Kombination mit LMS und DBMS konzeptuell, im Vergleich zu den im vorhergehenden Abschnitt beschriebenen Entwurfsmustern, besitzen.

Aufgrund des hohen Abstraktionsniveaus von Skeletons, können Entwickler von DBMSe verschiedene Rollen einnehmen. Eine Gruppe sind die Nutzer der Skeletons, die Datenbankexperten, welche für eine hohe Produktivität bei der Entwicklung sorgen, da sie mit der Architektur von DBMS vertraut sind. Weiterhin gibt es die Gruppe der Parallelisierungsexperten. Da sich die jeweiligen Programmierer auf ihre Kernkompetenzen konzentrieren, werden Fehler die zu Programmabstürzen und Performanceproblemen führen aus Mangel an Kenntnis der Materie vermieden. Die Flexibilität ist gering, da man auf die bereitgestellten Skeletons begrenzt ist. Sobald ein Algorithmus umgesetzt werden muss, der auf kein typisches Parallelisierungsmuster passt, muss dieses erst analysiert und das Skeleton dafür erstellt werden. Im Datenbankkontext sind die Algorithmen allerdings relativ statisch, da die Operatoren vorgegeben sind und sich nicht ändern (Solange die Anfragesprache nicht erweitert oder geändert wird).

Beim direkten Ansatz hingegen ist diese Trennung nicht möglich und die Entwickler müssen mit den Parallelisierungsschnittstellen direkt arbeiten. Entwickler ohne Parallelisierungserfahrung werden wahrscheinlicher die typischen Fehler begehen. Der Aufwand mehrere Parallelisierungframeworks komplett in eine DSL zu übertragen ist aufwändig. Die Motivation dahinter ist, dass durch die abweichenden Perfromancecharakteristiken der Frameworks, es nützlich mehrere zu unterstützen und dann das geeigneteste für eine parallele Berechnung auszuwählen. Weiterhin kann es vorkommen, dass für spezielle Hardware auch spezielle Frameworks nötig sind. Die Möglichkeiten verschiedene Varianten mit einem Variantengenerator zu erstellen beschränkt sich auf die Auswahl des Parallelisierungsframeworks und die Auswahl der Algorithmen. Die Flexibilität ist allerdings hoch, da der direkte Zugriff des Programmierers auf ein Parallelisierungsframework die Umsetzung von weiteren Algorithmen ermöglicht.

Bei dem Ansatz mit den Leaky-Abstractions kommt es auf die Ausprägung der Leaks an. Je größer die Leaks sind, also die Einflussmöglichkeiten der Datenbankprogrammier, desto wahrscheinlicher ist eine fehlerhafte Nutzung, die von den Parallelisierungsexperten nicht vorgesehen ist. Dies führt zu Programmfehlern, falschen Ergebnissen oder schlechter Performance. Ein bekanntes Beispiel für Leaky-Abstractions ist SQL. Eine Anfrage kann auf mehrere Art und Weisen formuliert werden, um zum selben Ergebnis zu führen. Die Durchlaufzeit der Anfrage kann dabei erheblich variieren. Demnach brauch der Anfragesteller Wissen darüber, wie die Abstraktion funktioniert, um die Performance nicht zu beeinträchtigen. Durch die Parameter hat man ein gewisses Maß an Variabilität. In unserem Ansatz sollen die Varianten allerdings durch den Variantengenerator gesteuert werden und nicht durch den Datenbankprogrammierer, der Beispielsweise in Switches die passende Argumente in die parallelen Abstraktionen einsetzt.

Mit jeder Schicht multiplizieren sich die möglichen Varianten. Mehr Schichten erhöhen auch das Optimierungspotenzial, da auf jeder Ebene die ebenenspezifischen

| | Optimierungs-potenzial | Aufwand | Variationen | Abstraktions-niveau | Flexibilität |
|---|---|---|---|---|---|
| DSL/Direkt | gering | hoch | wenige | gering | hoch |
| Skeletons | hoch | gering | viele | hoch | gering |
| Leaky Abstractions | hoch | gering | wenige | gering | variiert |
| Layered | variiert | variiert | variiert | variabel | variiert |

Tabelle 5.1: Übersicht der Entwurfsmuster und deren Bewertung

Optimierungsregeln umsetzbar sind. Durch die Schichten und Optimierungsregeln kann der Aufwand relativ groß werden. Die Flexibilität und Abstraktionen wird durch die Höhe der höchsten verwendeten Abstraktion eingeschränkt.

#### Zusammenfassung

Die Untersuchung hat aufgezeigt das es einige Parameter gibt, mit denen man die Ausprägung der untersuchten Eigenschaften steuern kann.

Die Anzahl der Schichten steuert die Variantenvielfalt und Optimierungsmöglichkeiten, wobei ein Anstieg an Schichten auch ein Anstieg der möglichen Varianten und Optimierungen bedeutet (unter der Voraussetzung, dass es pro Schicht auch mindestens zwei Auswahlmöglichkeiten und Optimierungsmöglichkeiten gibt).

Je höher das Abstraktionsniveau ist, desto geringer wird die Flexibilität, aber gleichzeitig fördert es die Rollenverteilung innerhalb eines Entwicklerteams.

Tabelle 5.1 fasst die Ergebnisse der Diskussion kurz zusammen.

## 5.3 Performance-Messungen

Dieser Abschnitt befasst sich mit dem Aufbau der Testumgebung, der Präsentation der Testergebnisse, sowie der Interpretation der gemessenen Ergebnisse. Gemessen wurden die Performance der DB Operatoren die mit den Skeletons umgesetzt wurde.

### 5.3.1 Testsetup

#### Testdatensätze

Zur Auswertung werden ausschliesslich 32-Bit Integer Werte (`int32_t aus dem C stdint.h Header`) als Attribute genutzt. Die Testdatensätze wurden mit der Sample Funktion der Programmiersprache R [IG96] erzeugt. Die Sample Funktion erzeugt pseudo-zufällig Zahlen aus einer vorgegeben Menge an Zahlen. Diese Zahlen sind gleichverteilt, d.h. jede Zahl hat die gleiche Wahrscheinlichkeit ausgewählt zu werden. Die Parameter der Funktion ist die Anzahl an Zahlen die erzeugt werden soll, die Menge aus der Zahlen ausgewählt werden und ob mehrfach aus dieser Menge gezogen werden darf. In unserem Fall ist die Mehrfachziehung erlaubt und es wird aus den Zahlen von 1 bis 100 gezogen. Die Anzahl der erzeugten Werte variiert je nach Größe des gewünschten Verbrauchs an Speicherplatz im Hauptspeicher.

| Tupel | 131072 | 262144 | 524288 | 1048576 | 2097152 | 4194304 | 8388608 |
|---|---|---|---|---|---|---|---|
| Megabyte | 0,5 | 1 | 2 | 4 | 8 | 16 | 32 |
| | 16777216 | 33554432 | 67108864 | 134217728 | 268435456 | | |
| | 64 | 128 | 256 | 512 | 1024 | | |

Tabelle 5.2: Umrechnungstabelle 32 Bit Integer Tupel in Megabytes

Die Größe der Tabellenspalten wird im weiteren in Megabytes (genutzter Arbeitsspeicher) angegeben. Dies kann auch in die Anzahl an Werten in der Spalte umgerechnet werden. 64 MB sind umgerechnet 536870912 Bits die durch die Integerbreite von 32 Bit zu teilen ist. In dem Fall würde es 16777216 Attributwerten von Tupeln entsprechen. In Tabelle 5.2 sind alle Umrechnungen für alle verwendeten Größen enthalten.

Für die Tests haben wir Datensätze für Tupelmengen zwischen 0,5 MB und 1048 MB generiert.

### Zeitmessung

Das minimale DSL Programm enthält neben der eigentlichen Berechnung, auch einen Teil, der die Daten einer Spalte aus einer Datei in den Arbeitsspeicher einliest. Diese würde man zwangsläufig mit messen, wenn man die Laufzeit des Programmes ermittelt. Um genauer arbeiten zu können muss die Zeit beim Beginn und am Ende der Berechnung gemessen werden. Die beiden Werten werden subtrahiert und ergeben dann die Laufzeit der Berechnung, die diese beiden Kommandos einschließen. Die Zeitmessung muss in das DSL-Programm integriert werden. Dazu haben wir die DSL um die Methode `get_wtime()` erweitert. Diese wird bei der Codegenerierung in die von OpenMP bereitgestellte Funktion `omp_get_wtime()` übersetzt und misst die vergangene Zeit in Sekunden seit einem beliebigen fixen Zeitpunkts in der Vergangenheit. Dieser Zeitpunkt verändert sich nicht während der Programmausführung und kann aus diesem Grund für Vergleiche herangezogen werden.

Jeder Test wird 100 mal wiederholt und aus den gemessenen Werten wird der Durchschnitt gebildet. Dies soll die Schwankungen ausgleichen, die andere Prozesse verursachen, die ebenfalls auf dem Testgerät ausgeführt werden (Betriebsystem, Testsuite, IDE).

Jeder Test wird mit einem, zwei, vier und acht Thread/s ausgeführt, um den Speed-Up zu prüfen. Der Speed-Up wird berechnet, indem die Laufzeit des Tests für einen Thread geteilt wird durch die Zeit desselben Tests mit einer anderen Threadanzahl.

### Plattform

Alle Tests wurden mit dem Betriebsystem OS X 10.11 El Capitan mit 8 Gigabyte (GB) DDR3-1333 Arbeitsspeicher und einem Intel Core i7-2635QM Prozessor ausgeführt. Dieser verfügt über 4 physische bzw. durch Hyper-Threading 8 logische Kerne, die jeweils mit 2 Gigahertz (Ghz) getaktet sind. Die Caches sind in die Größen 6144 Kilobyte (KB), 1024 KB, 256 KB gestaffelt. Eine Cacheline ist 64 Byte groß.

Weiterhin haben wir den Turbo-Boost des Prozessors mithilfe des Turbo-Boost Switchers (erhältlich unter http://www.rugarciap.com/turbo-boost-switcher-for-os-x/) deaktiviert. Der Turbo-Boost sorgt dafür, dass die Taktfrequenz eines einzelnen CPU Kerns angehoben wird, wenn er komplett ausgelastet ist. Bei der Test-CPU ist der Takt von 2 Ghz auf bis zu 2,9 Ghz steigerbar. Damit sind 45% mehr Taktzyklen ausführbar. Mit der Deaktivierung soll ausgeschlossen werden, dass eine mögliche schnellere Abarbeitung mit einem einzelnen Thread zu großen Teilen durch den Turbo-Boost verursacht wird und nicht durch beispielsweise die fehlende Thread-synchronisierung.

### 5.3.2 Erwartungen

Von den Messergebnissen des generierten Codes erwarten wir, dass er sich wie handgeschriebener Code in den punkten Korrektheit und Performance verhält. Die Korrektheit lässt sich einfach überprüfen, indem die Ergebnisse der Operationen betrachtet werden oder während der Ausführung es zu Programmfehlern kommt. Bei der Bewertung der Performance analysieren wir die parallelen und sequentiellen Anteil eines Algorithmus, um den maximalen SpeedUp einschätzen zu können. Die Ergebnisse sollten sich bei korrekter Umsetzung gemäß des amdahlschen Gesetzes [Amd67] verhalten. Dieses sagt unter anderem aus, dass ein Programm einen parallelen und einen sequentiellen Anteil hat, von dem ausschließlich der parallele Anteil durch hinzufügen von Berechnungseinheiten beschleunigt werden kann. Die Synchronisationskosten steigen mit der Anzahl an Berechnungseinheiten.

Im folgenden stellen wir kurz die Tests vor und schätzen deren sequentiellen Anteil ein:

**Selektion** Der Datenbankoperator Selektion der eine Testabelle nach einem Prädikat filtert welches genau in 50% der Fälle zutrifft. Sequentiell bei der Umsetzung dieses Operators ist die Berechnung der Präfixsumme, welche nicht sonderlich aufwändig ist, sodass es nur zu leichten Einbußen bei der Skalierung kommen kann.

**Nested-Loop-Join** Der Datenbankoperator Join als Nested-Loop Variante, der ohne sequentiellen Anteil auskommt. Dabei wird die Testtabelle mit einer Schlüsseltabelle (enthält die Schlüsselwerte 1-100) zusammengefügt.

**Sort** Beim Sort läuft das zusammenfügen der sortieren Teilmengen sequentiell ab. Dies sollte das Skalierungsvermögen nicht stark einschränken, da der aufwändigste Teil die Sortierung der Teilmengen ist, welche parallel abläuft.

**Sort-Merge-Join** Enthält zweimal Sort für jeweils die beiden zu verbindenden Tabellen, also zweimal die sequentielle Merge-Phase. Weiterhin kommt nun das zusammenfügen der beiden sortierten Tabellen hinzu. Dies läuft sequentiell ab und hat einen relativ großen Anteil an der Berechnung, sodass die Skalierung gegenüber den einzelnen Sorts schlechter sein sollte. Auch hier wird die Testtabelle wieder mit der Schlüsseltabelle zusammengefügt.

**Map** Map Skeleton, welches für die Verarbeitung von Skalarfunktionen in SQL genutzt werden kann. Hat keinerlei sequentiellen Anteile und sollte gut skalieren, da keinerlei Abhängigkeiten zwischen den einzelnen Elementen der Spalte vorkommen.

### 5.3.3 Ergebnisse

#### Selektion

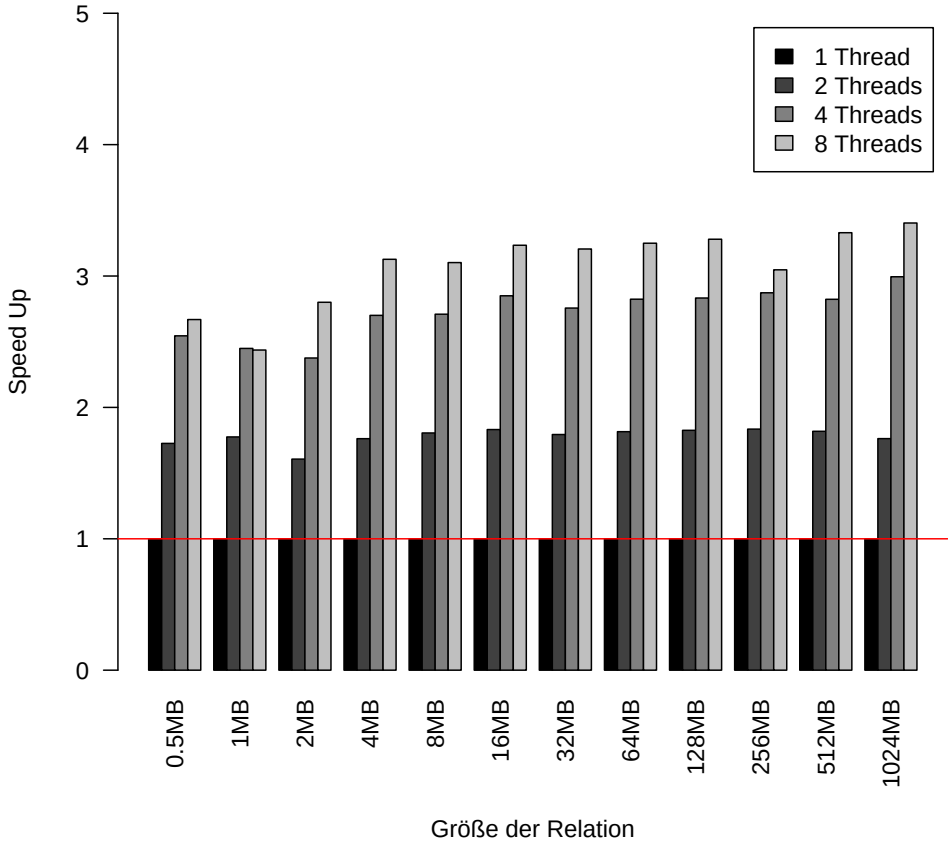


Abbildung 5.1: Vergleich der Antwortzeit bei der Selektion mit einem Selektivitätsfaktor von 0.5

Abbildung 5.1 zeigt den Speed-Up Vergleich für die Selektion. Die Skalierung ist durchwegs gut, wobei ab 8 Threads die Steigerung abnimmt, welches mit der Anzahl an verfügbaren Prozessorkernen zusammenhängt. Performanceabzüge ergeben sich durch den OpenMP Parallelisierungsoverhead. Weiterhin ist die Berechnung der Ergebnisgröße aus den markierten Indexen nicht parallelisiert. Der Selektivitätsfaktor bei der ausgeführten Selektion beträgt 50%, sodass einer Verfälschung der Ergebnisse durch Prozessorvorhersagen vermieden wird. Weiterhin sind die Werte in der Spalte gleichverteilt. Nach der Partition durch OpenMP sind sie ebenfalls gleichverteilt. Damit kann sichergestellt werden dass alle Threads ungefähr zur gleichen

Zeit beendet werden und der Master Thread die Ergebnisse verarbeiten kann. Bei ungleichen Datenverteilungen kann es dazu kommen, dass einige Threads schneller sind als andere und deshalb auf die anderen Threads gewartet werden muss, bevor die Berechnung fortgeführt werden kann. In diesem Fall würde der Speedup geringer ausfallen, sodass die Messwerte den Optimalfall beschreiben.

### Nested-Loop-Join

Beim Nested-Loop-Join gibt es drei Möglichkeiten der Parallelisierung die dadurch entstehen, dass zwei For-Schleifen ineinander verschachtelt werden. Die Möglichkeiten ergeben sich daraus wo die Parallelisierungsanweisungen gesetzt werden. Entweder an der inneren Schleife, der äußeren oder an beiden. Im folgenden zeigt sich beispielhaft, aus welchem Grund, die strikte Trennung zwischen Parallelisierungsentwickler und Datenbankentwickler sinnvoll ist.

Abbildung 5.2 zeigt die Messergebnisse für einen Nested-Loop-Join, bei dem nur die äußere Schleife parallelisiert wurde, wohingegen Abbildung 5.3 die Messergebnisse für einen Nested-Loop-Join zeigt, bei dem die innere Schleife parallelisiert wurde.

Bei der Parallelisierung mit der äußeren Schleife fällt auf, dass er unabhängig von der Eingabegröße immer ungefähr gleich skaliert. Das bedeutet das 2 Threads immer einen Speedup von ca 1,8 erreichen, 4 Threads schwanken um 2,5 und 8 Threads um ca. bei einem Speed-Up von 3 immer in Relation zu einem Thread. Die gesamte Arbeit wird einmalig auf mehrere Threads aufgeteilt und die innere Schleife wird jeweils sequentiell pro Thread abgearbeitet. Der Paralleliserungsoverhead ist dabei gering.

Bei kleinen Eingabegrößen ist der Speed-Up marginal bei der paralleliserten inneren Schleife. Selbst bei großen Eingabegrößen wird kaum ein Speed-Up erreicht der doppelt so groß ist, wie der eines Threads. Im Vergleich zu dem Nested-Loop-Join mit der äußeren Parallelisierung sind die werte wesentlich schlechter. Dies liegt daran, dass in diesem Fall für jede Iteration der äußeren Schleife ein Thread Team erstellt werden muss. Danach muss Fertigstellung aller Threads gewartet werden um dann mit der nächsten Iteration fortzufahren. Bei der äußeren Parallelisierung muss lediglich ein einziges Thread Team erstellt werden, welche alle Iterationen untereinander aufteilen. Dieser Overhead hat einen deutlichen Einfluss auf die Abarbeitungsgeschwindigkeit. Eine Steigerung ist es beide Schleifen zu parallelisieren. Hier würde die Threads des äußeren Threadteams wieder Threadteams für die innere Schleife erzeugen. Ein Einsteiger in die parallen Programmierung könnte meinen, dass mehr Parallelisierung in einer größeren Abarbeitungsgeschwindigkeit resultiert. Stattdessen kommt es auf die Parallelisierung an den richtigen Stellen an. Hier können Skeletons helfen Paralleliserungsfehler zu verhindern.

Auch wenn sich der Nested-Loop-Join einfach zu parallelisieren lässt, gibt es effizientere Algorithmen zur Berechnung des Joins.

### Sort

Um den Sort-Merge-Join umzusetzen kann die Sort-Phase parallelisiert werden.

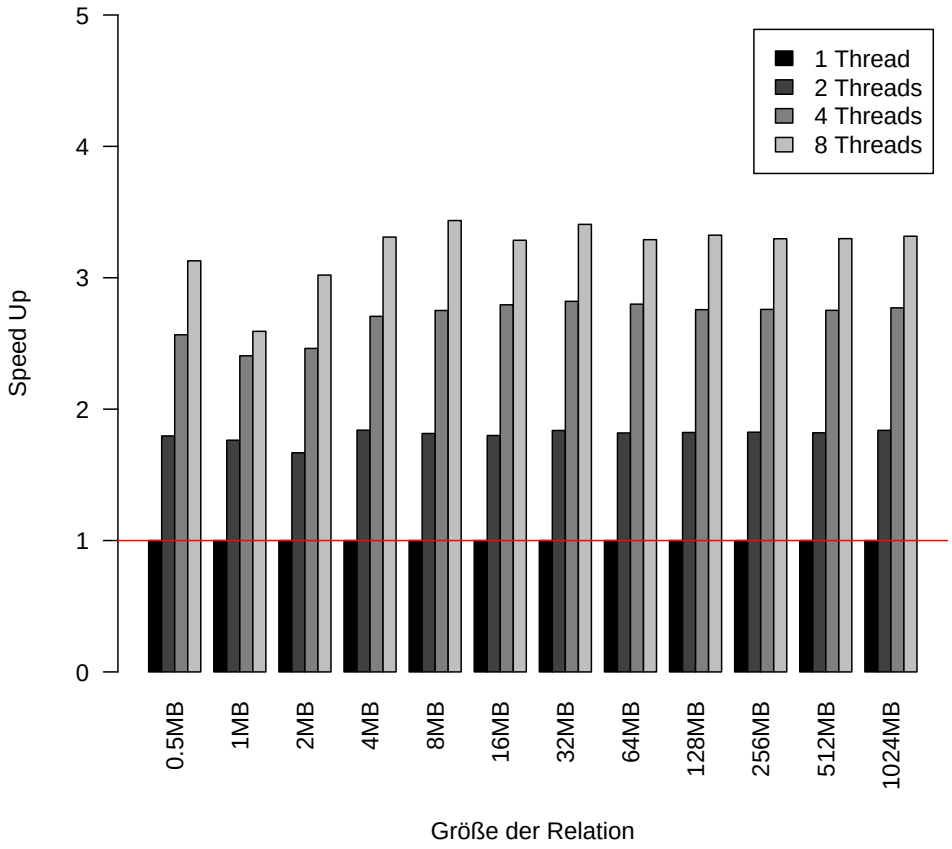


Abbildung 5.2: Nested-Loop-Join mit Paralleliserung der äußeren Schleife

Beim Sort weicht die Variante für ein Thread von denen für mehrere Threads ab. Bei einem Thread kommt eine sequentielle Variante zum Einsatz, die lediglich die qsort Funktion der C Standardbibliothek nutzt. Es fehlt der Overhead der parallelen Variante, welcher darin liegt die einzelnen sortierten Partitionen zusammenzufügen. Mit LMS lässt sich das einfach durch eine if-Abfrage umsetzen, die später nicht im generierten Code auftaucht, so wie in Quelltext 5.1 zu sehen ist.

Abbildung 5.4 zeigt den Speedup, der bei der Sortierung erreicht wurde.

### Sort-Merge-Join

Die Messergebnisse des Sort-Merge-Joins sind in Abbildung 5.5 zu sehen. Der Sort Merge besteht aus zwei parallelen Sorts, sowie der sequentiellen Merge Phase. Mit dem Hintergrund dieser Zusammensetzung ist auch das Messergebnis zu erklären. Der Speed-Up wird lediglich durch die Paralleliserung der Sortierung erzielt. Dadurch kann der Sort-Merge-Join unter keinen Umständen einen höheren Speed-Up als die Sortierung haben. Der sequentielle Merge Teil der hinzukommt vergrößert somit den sequentiellen Anteil der Berechnung gegenüber der reinen Sortierung. Demnach muss der Speed-Up beim Sort-Merge-Join nur leicht absinken, da das aufwändigste beim SMJ die Sortierung ist.

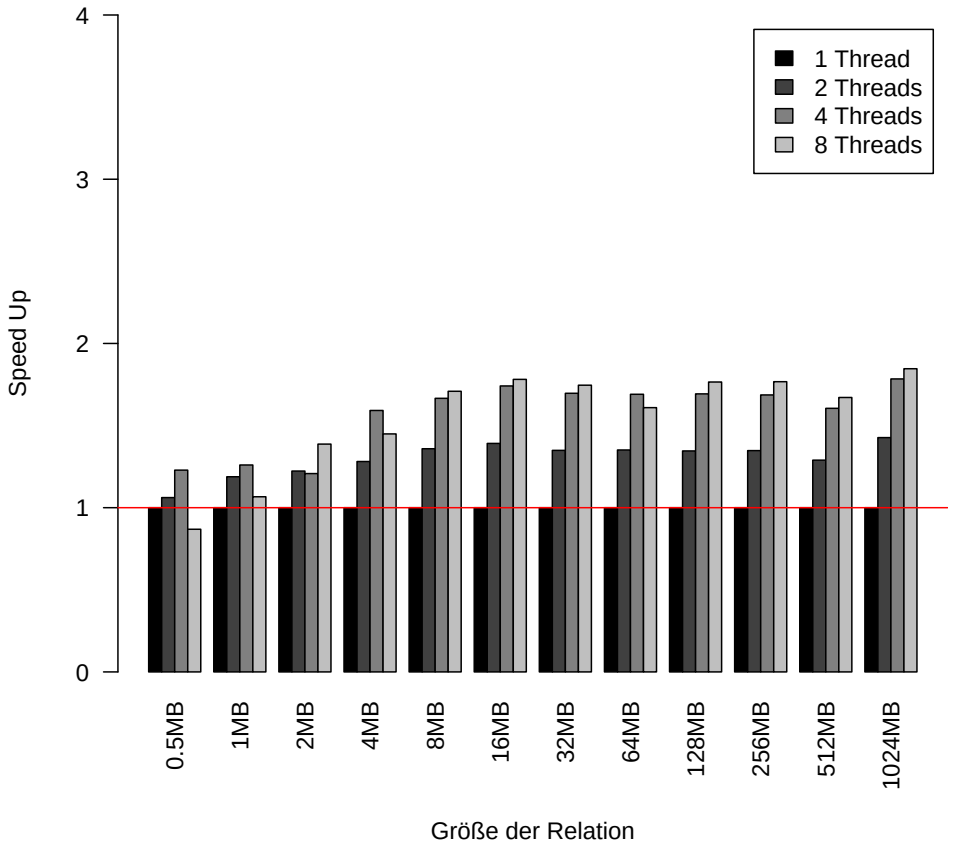


Abbildung 5.3: Nested-Loop-Join mit Parallelisierung der inneren Schleife

### Map

Das Map Skeleton wird zum Testen eine Funktion übergeben, die frei von Seiteneffekten ist, um die Korrektheit der Ergebnisse garantieren zu können. Diese multipliziert jeden Wert einer Spalte mit einer Konstanten. Die Messwerte, die in Abbildung 5.6 dargestellt sind, belegen, dass sich bei kleinen Eingabemengen (bis 2 Megabyte (MB)) keine Geschwindigkeitsvorteile ergeben. Es verlangsamt sich sogar die Abarbeitungsgeschwindigkeit. Der Overhead der Paralleliserung ist größer als die eigentliche Berechnung. Bei größeren Tabellenspalten wird kaum Speed-Up erzielt. Die übergebene Funktion ist nicht komplex genug, sodass die die Ergebnisse nicht so schnell in den Speicher zurückgeschrieben werden können. Der I/O Kanal zum Arbeitspeicher ist überfordert. Normalerweise ist der I/O Kanal bereits mit einem Prozessor überfordert bei einfachen Berechnungen. Eine Verdopplung der beteiligten Prozessoren, die den selben I/O Kanal nutzen, kann dementsprechend kaum Mehrleistung erbringen, da der Kanal bereits ausgereizt ist. Eine NUMA-Architektur schafft Abhilfe, da die Prozessoren über mehrere I/O Kanäle an den Arbeitsspeicher angebunden sind. Ein weiterer sinnvoller Anwendungsfall sind komplexere skalare Funktionen, die bei einer SQL Abfrage genutzt werden können, wie beispielsweise

```
1 if (threads == 1) {
2   sequential_sort(y, size)
3 } else {
4   sort(y, size)
5 }
```

Quelltext 5.1: Auswahl des Algorithmus nach Anzahl der Threads

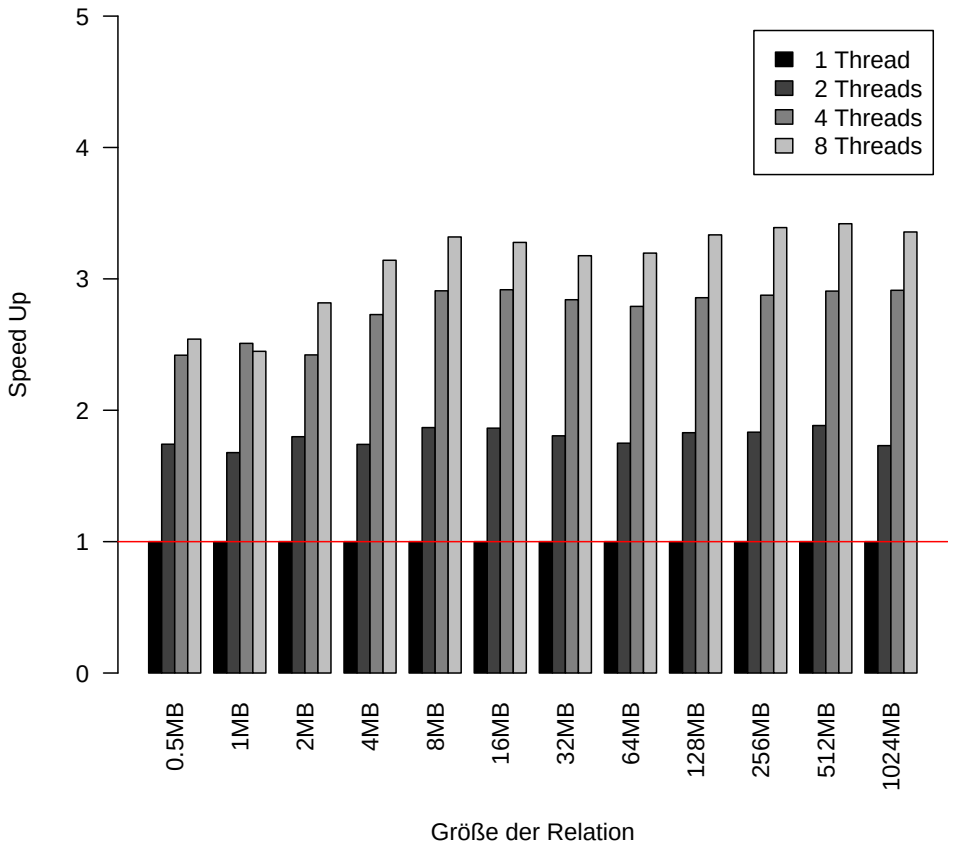


Abbildung 5.4: Vergleich der Laufzeit bei Sort

Stringkonvertierungen, welche wesentlich aufwändiger sind als einfache Multiplikationen. Die Berechnungszeit ist dann größer als die Zeit die zum schreiben in dem Arbeitsspeicher gebraucht wird, wodurch das Problem der Blockierung verringert wird.

### Kompilierungszeiten

Tabelle 5.3 zeigt eine Auswahl an Kompilierungszeiten. Vergleicht man diese mit den Laufzeiten der Testprogramme (siehe Anhang) wird klar, dass sich die vorher aufgebrachte Berechnungszeit in der Kompilierungsphase erst bei größeren Eingabemengen amortisiert. Die Testprogramme wurden mit dem `-O3` Flag des GCC

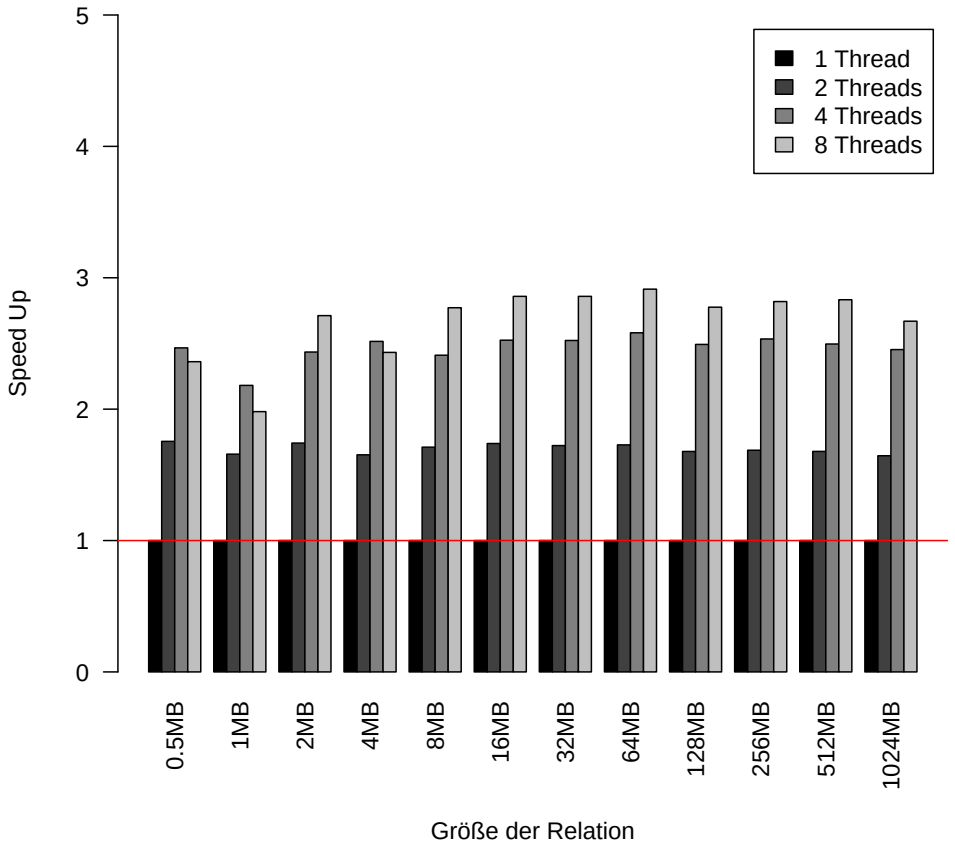


Abbildung 5.5: Vergleich der Laufzeit beim SMJ

Compilers kompiliert, sodass eine Verringerung der Kompilierungszeit möglich ist. Wie der Trade-Off von Kompilierungszeit und Ausführungszeit sich bei verschiedenen SQL-Anfragen verhält muss in weiteren Arbeiten festgestellt werden. Der Fokus in dieser Arbeit liegt ohnehin bei OLAP Anfragen, bei denen die Eingabegrößen generell sehr groß sind. Dort ist das Verhältnis von Ausführungszeit und Kompilierungszeit günstiger.

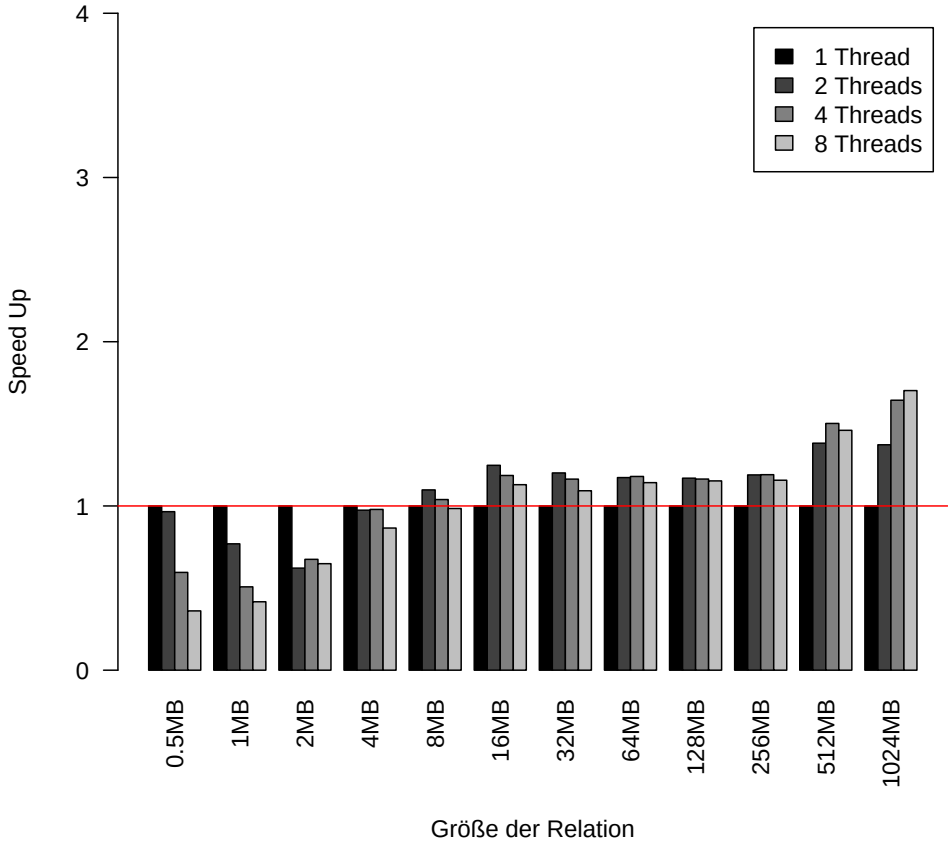


Abbildung 5.6: Vergleich der Antwortzeit beim Map Skeleton bei der Multiplikation mit einer Konstanten

| Test | Dauer |
|---|---|
| Map | 305.57ms |
| Selection | 279.42ms |
| NLJ | 270.28ms |
| NLJ2 | 278.41ms |
| Sort | 244.18ms |
| Sort-Merge-Join | 290.44ms |

Tabelle 5.3: Durschnittliche Kompilierungszeiten der verschiedenen Tests

### Zusammenfassung

Der generierte Code zeigt keine Auffälligkeiten bei der parallelen Performance. Das Map Skeleton kann aufgrund der Speichergebundenheit bei bestimmten Funktionen nicht wie die anderen Testprogramme skalieren. Ein handgeschriebenes Programm, mit denselben Voraussetzungen, würde sich allerdings ähnlich verhalten. Ansonsten verhalten sich die Tests wie erwartet. Die Kompilierungszeiten machen Staged-Programming ohne weitere Optimierungen des Kompilierungsprozesses für kleinere Eingabemengen unattraktiv.

# 6. Verwandte Arbeiten

Der Inhalt dieser Arbeit ist in dieser Form originell. Trotzdem lassen sich aktuelle Arbeiten finden, die ähnliche Zielstellungen verfolgen und vergleichbare Konzepte bei der Problemlösung und Herausforderungen aufweisen. Diese lassen sich grob zwei Themengebieten zuordnen.

Diese sind

- Codegeneratoren zur Paralleliserung
- Multi-Stage Programmierung bei der Implementierung von DBMS

und werden im folgenden kurz vorgestellt.

**Codegeneratoren zur Paralleliserung**

Um die parallele Programmierung zu vereinfachen werden zurzeit verschiedene Ansätze entwickelt die dafür DSLs nutzen. Im folgenden stellen wir zwei aktuelle Vertreter vor.

Der erste Vertreter ist das Delite Framework [CDM+10], welches am Pervasive Parallelism Laboratory (PPL) der Universität Stanford entwickelt wird. Unterstützt werden sie dabei von Mitarbeitern der Ecole Polytechnique Federale de Lausanne (EPFL) an der auch Scala, LMS und Legobase entwickelt wird. Delite bietet ebenfalls einen Satz an Skeletons an, welche dort als „parallel execution patterns“ bezeichnet werden und grundsätzlich dasselbe Konzept von sich wiederholt vorkommenden parallelen Abarbeitungsmustern bei verschiedenen Berechnungen beschreibt. DSL Operationen werden ebenfalls auf diese Patterns abgebildet. Programmierer nutzen die DSL Operationen um Anwendungen zu entwickeln, welche dadurch implizit parallel sind. Diese Programme sind ausschließlich mithilfe der Delite Laufzeit ausführbar. Diese entscheidet darüber wie parallelisert wird. Dies ist abhängig von der Ressourcenauslastung des Systems. Die kompilierung erfolgt demnach dynamisch. Das ist ein unterschied zu unserem Ansatz, bei dem die kompilierung statisch erfolgt. Dynamische Kompilierung ist konzeptionell durch den Einsatz des Variantengenerators

möglich, da die Query Engine oder Teile davon dynamisch erstellt werden und diese wiederum statisch Code generieren. Ebenfalls problematisch wie in unserem Ansatz ist die Verschachtelung von Patterns bei Delite. In diesem Fällen wird keine effiziente Variante der Berechnung generiert. Dieses Problem wurde bereits von den Entwicklern erkannt und an einer Lösung wird gearbeitet [LBS+14].

Das Framework Copperhead [CGK11] nutzt eine Teilmenge der Skriptsprache Python als Metasprache und führt speziell markierte Funktionen automatisch mithilfe von generierten und kompilierten Code in einem bereitgestellten parallelen Programmiermodell (zurzeit nur CUDA) aus. Dafür ist wieder eine spezielle Laufzeit nötig. Bei Copperhead wird das Konzept der Skeletons als parallele primitiven bezeichnet. Die zuvor erwähnten markierten Funktionen werden von den Entwicklern mithilfe der unterstützten parallelen Primitven implementiert.

Bei verschachtelten parallelen Primitiven nutzt Copperhead das sogenannte „Flattening“ [BS88], welches die Verschachtelung in eine äquivalente effizientere Variante umwandelt. Dazu werden die Operationen von der Copperhead Laufzeit analysiert und wenn möglich umgewandelt. In weiteren Arbeiten ist abzuklären, inwiefern Flattening mit LMS in unserer Lösung mithilfe von Umformungen am Syntaxbaum der DSL durchgeführt werden kann.

Ein weiterer Mechanismus, der die Anzahl an Synchronisationspunkten zwischen Threads und die Anzahl an Zwischenergebnissen verringert, ist die Fusion von mehreren Operatoren, die in Copperhead bereits primitiv umgesetzt wurde. Bei unserem Prototyp findet eine Synchronisierung nach jedem Skeleton statt, welches die Laufzeit um eine Größenordnung verschlechtern kann [CGK11].

### Multi-Stage Programmierung bei der Implementierung von DBMS

Die Verwendung von Multi-Stage Programmierung zur Implementierung von DBMS ist eine relativ neue Idee (2014) [KKRC14]. In dem DBMS Legobase, welches bisher nur der Forschung dient wird es eingesetzt um Anfragen zu beschleunigen und die Produktivität der Programmierer zu erhöhen. Dazu wird Scala als Metasprache und C als Objektsprache eingesetzt wie in unserem Prototypen. Dabei verfügen sie über eine ausgereiftere C-DSL die Möglichkeiten bietet um von Scala heraus Speicher zu allokieren und zu befreien oder beispielsweise pointer zu referenzieren. Diese Funktionalität ist in Legobase in einer extra Bibliothek ausgelagert, wovon lediglich die Binärdatei verfügbar ist. Die Sourcen sind für Entwickler außerhalb der Arbeitsgruppe die sich am EPFL damit beschäftigt nicht verfügbar. Mit Bereitstellung der Binärdatei kommen sie ein halbes Jahr nach Veröffentlichung des Papers der guten wissenschaftlichen Praxis nach Ergebnisse nachvollziehen zu können, behindern aber bewusst oder unbewusst die Weiterentwicklung anderer Forscher. Ein ähnliches Projekt welches ebenfalls an der EPFL entwickelt wurde ist die c-scala DSL. Um eine Vorstellung davon zu bekommen wie das Interface so einer C-DSL für Scala aussieht ist der Quellcode unter (https://github.com/Lewix/c-scala) einsehbar. Die meisten C Sprachkonstrukte können damit aus Scala mithilfe der passenden Methoden genutzt werden.

# 7. Zusammenfassung

Mit dem stetigen Preisverfall von RAM-Bausteinen wurde es immer attraktiver mehr davon in Datenbankserver einzusetzen. Damit wurde es ermöglicht ganze Tabellen in Arbeitsspeicher vorzuhalten. Der Flaschenhals von HDD zu RAM ist damit größtenteils weggefallen. Der neue Flaschenhals ist der Transportweg der Daten vom Arbeitsspeicher zu den Berechnungseinheiten. Diese sind heutzutage heterogen wie z. B. Prozessoren, Koprozessoren und Grafikprozessoren. Durch diese Ungleichverteilung der Rechenleistung nimmt die Bedeutung von Codeoptimierung zu. Eine Möglichkeit der Optimierung ist, durch Parallelisierung alle zur Verfügung stehenden Ressourcen effizient zu nutzen.

Besonders in Datenbankmanagmentsystemen, bei denen viele Daten uniform verarbeitet werden, kann dies zur erheblichen Senkung von Antwortzeiten beitragen oder die Verarbeitung von größeren Datenmengen ermöglichen.

Die Umsetzung von Codeoptimierungen sind aufwändig. Abstraktionen helfen die Produktivität zu steigern, kosten allerdings wiederum Performance, welche bei der Anfrageverarbeitung in Datenbanken nicht akzeptabel sind. Mit dem neuartigen LMS Ansatz können Abstraktionskosten verringert werden und zusätzliche Codeoptimierungen in den dazugehörigen Compiler integriert werden.

Die Möglichkeit Abstraktionen zu nutzen stellt nun die Frage, welche und wie diese am besten bei der Entwicklung von parallelen DBMS genutzt werden müssen, um eine möglichst kurze Antwortzeit zu gewährleisten.

Im Rahmen dieser Arbeit war das Ziel nun eine geeignete Abstraktion für die Parallelisierung zu finden, die sich mit LMS in einer Query-Engine integrieren lässt und das Erstellen verschiedener Varianten zulässt. Der generierte Code soll eine ähnliche Leistungscharakteristik wie handgeschriebener Code besitzen und dabei weiterhin korrekte Ergebnisse liefern.

Aus einer Menge von Paralleliserungsabstraktionen wurde eins ausgewählt und in ein Entwurfsmuster integriert. Das entwickelte Entwurfsmuster wurde mit alternativen Ansätzen verglichen und bewertet. Es wurde eine prototypische Implementation

nach dem Entwurfsmuster ausgeführt. Für ausgewählte Datenbankoperatoren, wie Selektion und Joins, wurde Code in verschiedenen Varianten mit der prototypischen Implementierung generiert. Dieser Code wurde dann mit verschiedenen Parametern auf dessen Leistungsfähigkeit untersucht.

Es ist ein Entwurfsmuster entstanden, in denen gekapselte funktionale Einheiten eine zentrale Rolle spielen, wobei deren Implementationen von einem Variantengenerator gesteuert wird. Sie dienen als Schnittstelle zwischen dem Datenbankentwickler und dem Parallelisierungsexperten.

Beim Vergleich mit alternativen Ansätzen konnte kein anderer Ansatz mehr Kriterien positiv erfüllen als unser Entwurfsmuster. Demnach ist dieses am geeignetesten für das konkrete Anwendungsszenario. Die praktische Untersuchung hat gezeigt, dass der erzeugte Code einwandfrei funktioniert und von der Leistung her sich wie handgeschriebenen Code verhält, solange die Skeletons nicht verschachtelt werden.

Auch wenn die Parallelisierung in DBMS von anderen Autoren als einfache Anpassungen ihrer bereits bestehenden sequentiellen Codegeneratoren eingeschätzt wird, konnten wir Hindernisse benennen, welche überwunden werden müssen, um maximale Leistung erreichen zu können. Paralleler Code ist wesentlich komplexer und damit auch schwieriger optimal zu erzeugen und sollte nicht unterschätzt werden. Die thematische Fusion von Paralleliserung, Multi-Stage-Programmierung und DBMS in dieser Arbeit stellt ein Ausgangspunkt für weitere Forschung dar.

# 8. Zukünftige Arbeiten

In dieser Arbeit wurde damit den Einsatz von Codegeneratoren in Datenbanken auf Basis von multi-stage Programmierung für die Paralleliserung zu untersuchen. In diesen Kapitel beschreiben wir die Ergebnisse dieser Arbeit erweitert werden können.

### Implementierung weiterer Parallelisierungsframeworks

In dieser Arbeit wurde eine mögliche Implementierung der Skeletons mit OpenMP vorgestellt und evaluiert. Die Implementierungen dieser Lösung können beliebig getauscht werden, solange sie dieselben Ergebnisse liefern. In weiteren Arbeiten könnten Codegeneratoren für CUDA, OpenCL, CilkPlus, TBB, PThreads entwickelt werden, um deren Eignung und Performance zu untersuchen und zu vergleichen.

### Nutzerauswertung und Produktivität

Ousterhaut [Ous98] untersuchte bereits die Zeitersparnis bei der Softwareentwicklung die mithilfe von Skriptsprachen gegenüber einer Implementierung in C oder C++ möglich ist. Scala ist keine Skriptsprache, bietet durch einige Features trotzdem ähnliche Möglichkeiten die Produktivität ebenfalls zu steigern und typische Fehlerquellen zu vermeiden. Eine Nutzerauswertung könnte evaluieren wie Entwickler mit den Skeletons umgehen und wieviel Zeit sie gegenüber einer eigenen Implementation von parallelen Code sparen.

Ein weiterer wichtiger Aspekt ist wie schnell neue Funktionen eingebaut und wieviel Bugs gefunden werden können im Vergleich zu einer Implementation in einer weniger abstrakten Sprachen. Das zielt darauf ab, möglichst schnell testfähige Prototypen bereitzustellen, die vor allem für die Forschung von Interesse ist.

Die Yin-Yang Bibliothek [JSS+14] soll die Nutzung und Erstellung von DSLs unter anderem mit LMS vereinfachen und beschleunigen. Die Integration dieser Bibliothek ist bei einer Weiterentwicklung des Prototypen in Betracht zu ziehen.

### Weiterentwicklung zu einem offenen Datenbankmanagementsystem

Eine weitere Möglichkeit ist es, den Prototypen zu einem vollständigen DBMS auszubauen. Der erste Schritt wäre dabei die Query Engine zu vervollständigen, indem die fehlenden Operatoren ergänzt werden wie Beispielsweise die Aggregation. Ebenfalls können weitere paralleliserbare Datenbankteile außerhalb der Query Engine gefunden und implementiert werden. Dabei wäre die Umsetzung des Variantengenerators etwas, was dieses DBMS von anderen DBMS abhebt und deshalb besonders erstrebenswert. Eine andere Möglichkeit wäre die Anpassung von Legobase, welches bereits über eine voll funktionsfähige Query Engine verfügt. Dies ist aber zurzeit noch nicht möglich, da der gesamte Quelltext noch nicht bereitgestellt wird.

### Performanceverbesserungen

An der Performance kann in weiteren Arbeiten an den folgenden Punkten gearbeitet werden.

Bei der Verschachtelung von Skeletons sollten Flattening- und Fusion Verfahren eingesetzt werden um Synchronisationspunkte und Zwischenergebnisse zu vermeiden. Der Hauptzweck ist dabei die Datentransfers möglichst zu verringern, da sie im Verhältnis kostbarer sind als Berechnungen auf dem Prozessor.

Die noch fehlende Implementation des Variantengenerators kann die Performance beeinflussen. Mithilfe der Kapselung der Parallelisierung in Skeletons kann ein Kostenmodell entwickelt werden, sodass für jede zur Verfügung stehende Implementierung, eines bestimmten Skeletons, die Performancecharakteristik mithilfe verschieden großen Testeingaben bestimmt wird. Mit diesem Wissen kann der Variantengenerator bei einer realen Eingabe anhand der zuvor ermittelten Ergebnisse eine gute Trait-Zusammensetzung für den Codegenerator bestimmen. Ein anderer Punkt ist die Ermittlung der aktuellen Systemauslastung und eine entsprechende Konfiguration der Threadgröße. Werden beispielsweise zwei von vier Kernen bereits mit einer Anfrage ausgelastet, dann sollten die Threadgrößen nicht mehr auf vier gesetzt werden. Wenn die Threads an die Kerne gebunden sind, würden zwei Threads eine relativ lange Zeit auf die anderen beiden Threads an der Synchronisationsbarriere warten müssen, da sie kaum Arbeit an den ausgelasteten Kernen verrichten können. Unter der Voraussetzung, dass die Eingabe gleich auf alle Threads verteilt wird.

Mithilfe von Single Instruction Multiple Data (SIMD) Instruktionen ist die Parallelisierung, zusätzlich zur Aufteilung und gleichzeitigen Bearbeitung der Daten auf mehreren Prozessoren, innerhalb eines Prozessor möglich, wenn die Berechnung dafür geeignet ist. Diese könnten bei dem Filter Skeleton zum Einsatz kommen, um mehrere Bedingungen gleichzeitig abzuprüfen oder beim Map Skeleton um mehrere Elemente gleichzeitig zu verarbeiten (je nach Funktionskörper). Es gibt über 1000 verschiedene SIMD Instruktionen, wodurch auch hier der Einsatz von generiertem Code die Programmierer entlasten kann. OpenMP und CilkPlus bieten auch hier wieder Vereinfachungen an, die bei der Codegeneration genutzt werden können.

# A. Anhang

Im Anhang präsentieren die Messwerte, die wir für die Evaluation gesammelt haben. Diese beinhalten Performancemessungen für verschiedene Varianten des Join Operators, der Selektion, sowie des Map Skeletons. Alle Tests wurden mit Threadgrößen zwischen 1 und 8 getestet und jeweils 100 mal wiederholt. Aus den gemessenen Zeiten wurde der Durchschnitt gebildet. Die Spaltengrößen der Testspalten variieren dabei von 0.5 MB bis 1024 MB und wurde zufällig aus Zahlen von 1-100 gleichverteilt zusammengestellt.

**Map Skeleton**

| Threads | Spaltengröße | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0.5MB | 1MB | 2MB | 4MB | 8MB | 16MB | 32MB | 64MB |
| 1 | 0.12 | 0.2 | 0.31 | 0.75 | 1.14 | 2.25 | 4.37 | 8.73 |
| 2 | 0.19 | 0.27 | 0.36 | 0.5 | 0.89 | 1.84 | 3.69 | 7.66 |
| 4 | 0.26 | 0.33 | 0.36 | 0.58 | 0.98 | 1.84 | 3.76 | 7.46 |
| 8 | 0.37 | 0.39 | 0.63 | 0.62 | 1.11 | 2.1 | 3.97 | 7.58 |

| | 128MB | 256MB | 512MB | 1024MB |
|---|---|---|---|---|
| 1 | 17.94 | 43.21 | 106.44 | 206.49 |
| 2 | 14.51 | 29.27 | 73.87 | 141.46 |
| 4 | 14.86 | 29.24 | 62.48 | 122.37 |
| 8 | 15.21 | 30.06 | 60.36 | 120.73 |

Tabelle A.1: Durschnittliche Laufzeiten Map Skeleton in ms

**Sort**

| Threads | Spaltengröße | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0.5MB | 1MB | 2MB | 4MB | 8MB | 16MB | 32MB | 64MB |
| 1 | 10.04 | 20.41 | 40.8 | 78.59 | 170.01 | 340.56 | 659.86 | 1,271.46 |
| 2 | 5.77 | 12.16 | 22.68 | 45.16 | 91.01 | 182.74 | 365.51 | 726.79 |
| 4 | 4.15 | 8.13 | 16.85 | 28.81 | 58.44 | 116.74 | 232.24 | 455.63 |
| 8 | 3.95 | 8.33 | 14.48 | 25.02 | 51.22 | 103.91 | 207.74 | 397.8 |

| | 128MB | 256MB | 512MB | 1024MB |
|---|---|---|---|---|
| 1 | 2,611.25 | 5,322.96 | 10,772.83 | 21,482.4 |
| 2 | 1,427.61 | 2,903.86 | 5,718.83 | 12,412.31 |
| 4 | 914.1 | 1,851 | 3,705.83 | 7,374.77 |
| 8 | 783.03 | 1,570.04 | 3,150.26 | 6,399.27 |

Tabelle A.2: Durschnittliche Laufzeiten Sort in ms

**Sort-Merge-Join**

| Threads | Spaltengröße | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0.5MB | 1MB | 2MB | 4MB | 8MB | 16MB | 32MB | 64MB |
| 1 | 11.57 | 20.55 | 42.3 | 81.94 | 166.54 | 339.47 | 676.85 | 1,355.45 |
| 2 | 6.59 | 12.4 | 24.29 | 49.59 | 97.35 | 195.26 | 392.89 | 784.42 |
| 4 | 4.69 | 9.42 | 17.37 | 32.57 | 69.09 | 134.45 | 268.31 | 525.12 |
| 8 | 4.9 | 10.37 | 15.6 | 33.69 | 60.08 | 118.76 | 236.79 | 465.27 |

| | 128MB | 256MB | 512MB | 1024MB |
|---|---|---|---|---|
| 1 | 2,625.01 | 5,257.19 | 10,698.29 | 21,961.08 |
| 2 | 1,564.42 | 3,115.59 | 6,374.27 | 13,347.19 |
| 4 | 1,053.05 | 2,074.59 | 4,286.54 | 8,951.07 |
| 8 | 945.55 | 1,864.97 | 3,776.59 | 8,227.07 |

Tabelle A.3: Durschnittliche Laufzeiten Sort-Merge-Join in ms

### Selektion

| Threads | Spaltengröße | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0.5MB | 1MB | 2MB | 4MB | 8MB | 16MB | 32MB | 64MB |
| 1 | 9.89 | 20.21 | 41.54 | 78.73 | 161.54 | 331.29 | 645.45 | 1,299.76 |
| 2 | 5.73 | 11.39 | 25.85 | 44.68 | 89.48 | 180.89 | 359.83 | 716.01 |
| 4 | 3.88 | 8.25 | 17.48 | 29.15 | 59.62 | 116.26 | 234.19 | 460.31 |
| 8 | 3.7 | 8.3 | 14.83 | 25.18 | 52.07 | 102.43 | 201.34 | 399.97 |

| | 128MB | 256MB | 512MB | 1024MB |
|---|---|---|---|---|
| 1 | 2,643.85 | 5,304.22 | 10,706.96 | 21,737.32 |
| 2 | 1,448.2 | 2,890.94 | 5,888.71 | 12,333.99 |
| 4 | 933.36 | 1,846.65 | 3,792.93 | 7,259.62 |
| 8 | 806.05 | 1,740.68 | 3,215.68 | 6,386.82 |

Tabelle A.4: Durschnittliche Laufzeiten Selektion in ms

### Nested-Loop-Join

| Threads | Spaltengröße | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0.5MB | 1MB | 2MB | 4MB | 8MB | 16MB | 32MB | 64MB |
| 1 | 15.54 | 30.42 | 60.84 | 127.42 | 266.43 | 533.72 | 1,072.76 | 2,124.51 |
| 2 | 8.65 | 17.25 | 36.48 | 69.24 | 146.82 | 296.62 | 583.74 | 1,168.06 |
| 4 | 6.05 | 12.64 | 24.71 | 47.08 | 96.85 | 191.02 | 380.34 | 758.96 |
| 8 | 4.97 | 11.74 | 20.14 | 38.5 | 77.55 | 162.45 | 314.92 | 645.76 |

| | 128MB | 256MB | 512MB | 1024MB |
|---|---|---|---|---|
| 1 | 4,265.01 | 8,533.16 | 17,127.16 | 34,439.31 |
| 2 | 2,340.16 | 4,676.71 | 9,410.3 | 18,730.91 |
| 4 | 1,546.89 | 3,092.68 | 6,223.5 | 12,427.83 |
| 8 | 1,283.02 | 2,588.36 | 5,193.38 | 10,385.36 |

Tabelle A.5: Durschnittliche Laufzeiten NLJ Variante 1 in ms

| Threads | Spaltengröße | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0.5MB | 1MB | 2MB | 4MB | 8MB | 16MB | 32MB | 64MB |
| 1 | 18.13 | 37.45 | 71.88 | 147.76 | 312.8 | 635.62 | 1,227.04 | 2,433.43 |
| 2 | 17.08 | 31.51 | 58.77 | 115.34 | 230.18 | 456.83 | 909.59 | 1,800.74 |
| 4 | 14.76 | 29.72 | 59.52 | 92.82 | 187.76 | 365.05 | 723.23 | 1,439.47 |
| 8 | 20.87 | 35.11 | 51.82 | 101.96 | 183.05 | 356.82 | 702.85 | 1,512.23 |

| | 128MB | 256MB | 512MB | 1024MB |
|---|---|---|---|---|
| 1 | 4,843.97 | 9,679.15 | 19,464.3 | 41,819.52 |
| 2 | 3,600.11 | 7,182.43 | 15,095.17 | 29,311.47 |
| 4 | 2,861.18 | 5,738.93 | 12,127.48 | 23,441.56 |
| 8 | 2,744.18 | 5,477.04 | 11,647.65 | 22,652.35 |

Tabelle A.6: Durschnittliche Laufzeiten NLJ Variante 2 in ms

# Literaturverzeichnis


[ADHW99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 266–277. Morgan Kaufmann, 1999. (zitiert auf Seite 18)

[AKN12] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the VLDB Endowment*, 5(10):1064–1075, 2012. (zitiert auf Seite 28)

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference 31*, pages 483–485. ACM, 1967. (zitiert auf Seite 51)

[BBHS14] David Broneske, Sebastian Breß, Max Heimel, and Gunter Saake. Toward hardware-sensitive database operations. In *Proceedings of the International Conference on Extending Database Techniques (EDBT)*, pages 229–234. OpenProceedings.org, 2014. (zitiert auf Seite 1)

[BGG+92] Richard J. Boulton, Andrew D. Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience (TPCD)*, pages 129–156. North-Holland Publishing, 1992. (zitiert auf Seite 9)

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 207–216. ACM, 1995. (zitiert auf Seite 29)

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 54–65. Morgan Kaufmann, 1999. (zitiert auf Seite 1)

[BNS04] David E. Bernholdt, Jarek Nieplocha, and P. Sadayappan. Raising level of programming abstraction in scalable programming models. In *Proceedings of the First Workshop on Productivity and Performance in High-End Computing (PPHEC)*, pages 76–84, 2004. (zitiert auf Seite 1)

[Bro15] David Broneske. Adaptive reprogramming for databases on heterogeneous processors. In *Proceedings of the SIGMOD PhD Symposium*, pages 51–55. ACM, 2015. (zitiert auf Seite 1, 2 und 22)

[BS88] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. In *The 2nd Symposium on the Frontiers of of Massively Parallel Computation (FMPC)*, pages 575–585. IEEE, 1988. (zitiert auf Seite 62)

[BTAÖzsu14] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on modern processor architectures. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 27(7):1754–1766, 2014. (zitiert auf Seite 28)

[Bur13] Eugene Burmako. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, page 3. ACM, 2013. (zitiert auf Seite 30)

[CDM+10] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 835–847. ACM, 2010. (zitiert auf Seite 61)

[CDS+05] Rémi Coudarcher, Florent Duculty, Jocelyn Serot, Frédéric Jurie, Jean-Pierre Derutin, and Michel Dhome. Managing algorithmic skeleton nesting requirements in realistic image processing applications: The case of the SKiPPER-II parallel programming environment's operating model. *EURASIP Journal on Applied Signal Processing*, 2005(7):1005–1023, 2005. (zitiert auf Seite 30)

[CGK11] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an embedded data parallel language. *Proceedings of the ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP)*, pages 47–56, 2011. (zitiert auf Seite 62)

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. (zitiert auf Seite 41)

[Col04] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004. (zitiert auf Seite 26)

[DHL10] Mischa Dieterle, Thomas Horstmeyer, and Rita Loogen. Skeleton composition using remote data. In *Practical Aspects of Declarative Languages*, pages 73–87. Springer, 2010. (zitiert auf Seite 30)

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. (zitiert auf Seite 29 und 33)

[Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley, 1st edition, 2010. (zitiert auf Seite 9)

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1st edition, 1994. (zitiert auf Seite 7, 12 und 30)

[Gho10] Debasish Ghosh. *DSLs in action*. Manning Publications Co., 1st edition, 2010. (zitiert auf Seite 9)

[GLPJ93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 223–232. ACM, 1993. (zitiert auf Seite 29)

[GMS92] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *Transactions on Knowledge and Data Engineering (TKDE)*, 4(6):509–516, 1992. (zitiert auf Seite 25)

[HORM08] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of DSLs. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 137–148. ACM, 2008. (zitiert auf Seite 14)

[HP11] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 5th edition, 2011. (zitiert auf Seite 15)

[HT99] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley, 1st edition, 1999. (zitiert auf Seite 26)

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996. (zitiert auf Seite 9)

[IG96] Ross Ihaka and Robert Gentleman. R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996. (zitiert auf Seite 49)

[JSS+14] Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. Yin-Yang: Concealing the deep

embedding of DSLs. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pages 73–82. ACM, 2014. (zitiert auf Seite 14 und 65)

[KKRC14] Ioannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment*, 7(10):853–864, 2014. (zitiert auf Seite 2, 18 und 62)

[Koc13] Christoph Koch. Abstraction without regret in data management systems. In *Conference on Innovative Data Systems Research (CIDR)*, page 149. www.cidrdb.org, 2013. (zitiert auf Seite 19)

[KPSW93] Ashfaq A. Khokhar, Viktor K. Prasanna, Muhammad E. Shaaban, and Cho-Li Wang. Heterogeneous computing: Challenges and opportunities. *Computer*, (6):18–27, 1993. (zitiert auf Seite 1)

[KSS14] Veit Köppen, Gunter Saake, and Kai-Uwe Sattler. *Data Warehouse Technologien*. mitp, 1st edition, 2014. (zitiert auf Seite 24)

[KVC10] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 613–624. IEEE, 2010. (zitiert auf Seite 2, 18 und 23)

[LBS+14] HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Tiark Rompf, and Kunle Olukotun. Locality-aware mapping of nested parallel patterns on GPUs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 63–74. IEEE, 2014. (zitiert auf Seite 62)

[Mey88] Bertrand Meyer. *Object oriented software construction*. Prentice-Hall, 1st edition, 1988. (zitiert auf Seite 5)

[MPO08] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In *Proceedings of the ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA)*, pages 423–438. ACM, 2008. (zitiert auf Seite ix und 4)

[MRR12] Michael D McCool, Arch D Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012. (zitiert auf Seite 19)

[Neu11] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011. (zitiert auf Seite 2, 18 und 29)

[NWDS13] Sebastian Nanz, Scott West, and Kaue Soares Da Silveira. Examining the expert gap in parallel programming. In *Euro-Par 2013 Parallel Processing*, pages 434–445. Springer, 2013. (zitiert auf Seite 45)

[Oa04] Martin Odersky and al. An overview of the Scala programming language. Technical report, EPFL, 2004. (zitiert auf Seite 3)

[oEE96] Institute of Electrical and Electronics Engineers. *IEEE Standard for information technology : portable operating system interface (POSIX)*. IEEE, 2nd edition, 1996. IEEE Std 1003.1-1996 (Incorporating ANSI/IEEE stds 1003.1-1990, 1003.1b-1993, 1003.1c-1995, and 1003.1i-1995). (zitiert auf Seite 29)

[OSV11] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide*. Artima, 2nd edition, 2011. (zitiert auf Seite 3 und 8)

[Ous98] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *Computer*, 31(3):23–30, 1998. (zitiert auf Seite 65)

[RAM+12] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-Virtualized: Linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, 25(1):165–207, 2012. (zitiert auf Seite 8)

[RDSF13] Hannes Rauhe, Jonathan Dees, Kai-Uwe Sattler, and Franz Faerber. Multi-level parallel query execution framework for CPU and GPU. In *Advances in Databases and Information Systems (ADBIS)*, pages 330–343. Springer, 2013. (zitiert auf Seite 18)

[RG03] Fethi Rabhi and Sergei Gorlatch. *Patterns and skeletons for parallel and distributed computing*. Springer, 2003. (zitiert auf Seite 26)

[RO10] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the international conference on Generative programming and component engineering (GPCE)*, pages 127–136. ACM, 2010. (zitiert auf Seite 15)

[RPML06] Jun Rao, Hamid Pirahesh, C. Mohan, and Guy Lohman. Compiled query execution engine using JVM. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 23–23. IEEE, 2006. (zitiert auf Seite 18)

[SFS+13] LuisMiguel Sanchez, Javier Fernandez, Rafael Sotomayor, Soledad Escolar, and J.Daniel. Garcia. A comparative study and evaluation of parallel programming models for shared-memory parallel architectures. *New Generation Computing*, 31(3):139–161, 2013. (zitiert auf Seite 45)

[Spo04] Joel Spolsky. *Joel on Software And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*. Apress, 1st edition, 2004. (zitiert auf Seite 46)

[SSH11] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken: Implementierungstechniken*. mitp, 2011. (zitiert auf Seite 17 und 41)

[ST98] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys (CSUR)*, 30(2):123–169, 1998. (zitiert auf Seite 29)

[SZB11] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 33–40. ACM, 2011. (zitiert auf Seite 18)

[Tah99] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. (zitiert auf Seite 15)

[TV14] Ashkan Tousimojarad and Wim Vanderbauwhede. Comparison of three popular parallel programming models on the intel xeon phi. In *Euro-Par 2014: Parallel Processing Workshops*, pages 314–325. Springer, 2014. (zitiert auf Seite 45)

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the European Symposium on Programming (ESOP)*, pages 344–358. Springer, 1988. (zitiert auf Seite 29)

[WD96] David W. Walker and Jack J. Dongarra. MPI: A standard message passing interface. *Supercomputer*, 12(1):56–68, 1996. (zitiert auf Seite 29)

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

Magdeburg, den 20. Oktober 2015