Bachelorarbeit

Erkennung, Übersetzung und parallele Auswertung von Operatoren und Funktionen aus R in SQL

vorgelegt von Dennis Weu
Matr.-Nr. 212206684
am 03.03.2016

am Lehrstuhl für Datenbank- und Informationssysteme
der Universität Rostock

Erstgutachter
Prof. Dr. Andreas Heuer
Universität Rostock
Datenbank- und Informationssysteme

Betreuer
Dennis Marten
Universität Rostock
Datenbank- und Informationssysteme

Zweitgutachter
Dr. Holger Meyer
Universität Rostock
Datenbank- und Informationssysteme

Inhaltsverzeichnis

Abstrakt

Smarte Umgebungen, die Intentionen und Aktivitäten erkennen, sind derzeit im Trend. Mit diesem geht einher, dass große Datenmengen schnell und effizient ausgewertet werden müssen. Die für statistische Auswertungen beliebte Programmiersprache *R* weist auf diesem Gebiet Performanceschwächen auf, weshalb mittels Datenbanksystemen eine Unterstützung stattfinden soll. Hierfür werden Ansätze geprüft, um gezielt nützliche *R*-Operatoren und *R*-Funktionen zu erkennen und diese in die Sprache *SQL* zu übersetzen. Darüber hinaus werden Ansätze zur parallelen Auswertung der *SQL*-Ausdrücke vorgestellt. Im Rahmen dieser Arbeit findet eine Festlegung relationaler Strukturen für Matrix und Datenframe, Erkennung wesentlicher Operatoren und Funktionen, Übersetzung dieser und kurze Evaluierung der Resultate statt. Am Ende wird ein Ausblick anhand der gewonnenen Erkenntnisse gegeben.

Nowadays smart environments detecting intentions and activities are a developmental trend. Therefore, a fast and efficient analysis of data becomes necessary. The language *R* is a popular tool of statistical analysis, but lacks when processing big data. Hence we aimed at a database support for this purpose. Therefore, basic approaches for detecting and translating useful *R* operators and *R* functions will be evaluated. Furthermore, first steps for parallel processing of *SQL* statements are introduced. Within this thesis, there is a determination of relational structures for the *R* matrix and dataframe, as well as a detection and translation of certain operators. Subsequently, the created *SQL* statements are reviewed. Eventually, we give an outlook based on the gained insights.

1 Einleitung

In der heutigen Zeit ist ein Trend zur Erkennung von Aktivitäten und Intentionen zu beobachten, was in vielen Gebieten, wie z.B. bei smarten Umgebungen und der Medizin, in Form von Assistenzsystemen zum Einsatz kommt. Die dafür benötigten Daten können zunehmend einfacher durch leistungsfähige, kompakte und günstige Sensoren gesammelt werden, weshalb oftmals sehr große Datenmengen anfallen. Eine effiziente Auswertung solcher Datenmengen ist generell ein Ziel der Forschung und ist u.a. im Kontext der Intentions- und Aktivitätserkennung ein Forschungsgebiet an der Universität Rostock.

Zur Erkennung von Mustern müssen vorliegende Daten ausgewertet werden, was oft mit dem für statistische Auswertung beliebten Programm *R* geschieht. Dieses bietet viele einfach benutzbare statistische Werkzeuge an, wodurch es dem Anwender möglich ist, schnell zielführende Algorithmen zur Datenauswertung zu verfassen. Übersteigt jedoch die auszuwertende Datenmenge die Größe des Hauptspeichers, sind enorme Performanceeinbußen zu verzeichnen und andere Auswertungsansätze sind notwendig. Um erhöhte Datenmengen effizient handhaben zu können, wird für gewöhnlich versucht Algorithmen zu parallelisieren. Obwohl es Möglichkeiten gibt, *R*-Programme zu parallelisieren, liegt diese Kompetenz im Allgemeinen nicht im Anforderungsprofil eines *R*-Anwenders, was eine transparente parallele Unterstützung von *R*-Programmen, z.B. mittels paralleler Datenbanksysteme, wünschenswert macht.

Allgemein haben sich auf dem Gebiet der Datenauswertung Datenbanksysteme bewährt, weshalb eine solche Nutzung Nahe liegt. Damit einher geht die Nutzung der Datenbanksprache *SQL*, von der die *R*-Programmierer für gewöhnlich nicht die in dem geforderten Maße vorhandene Kenntnis haben. Stattdessen soll eine Möglichkeit geschaffen werden, aus üblichem *R*-Programmcode Passagen in *SQL* zu übersetzen und somit für Datenbanksysteme nutzbar zu machen.

Operatoren und Funktionen agieren auf Objekten mit bestimmten Eigenschaften. Solche sind beispielsweise die Struktur des Objektes und zugelassene Datentypen. Zur Festlegung der Entsprechenden im Datenbanksystem ist daher vor der Erkennung und Übersetzung das Festlegen relationaler Strukturen wichtiger *R*-Objekte nötig. Nach der Festlegung ist es möglich, *SQL*-Abfragen zu gestalten, die ausgewählten *R*-Operatoren und *R*-Funktionen semantisch entsprechen.

Oftmals werden bei der Intentions- und Aktivitätserkennung Matrixoperationen wie Matrixmultiplikation oder Transponierung einer Matrix, sowie aus der relationalen Algebra bekannte Funktionen wie Selektion, Projektion und Aggregatfunktionen, benutzt. Im Verlauf dieser Arbeit konzentrieren wir uns daher auf die Erkennung und Übersetzung von Operatoren und Funktionen aus diesen beiden Bereichen.

Erst nach Vorliegen von *SQL*-Ausdrücken kann eine Verarbeitung durch gewöhnliche, aber auch parallele, Datenbanksysteme stattfinden. Die vorliegende Arbeit beschäftigt sich daher vorrangig mit dem Festlegen relationaler Strukturen, sowie der Erkennung und Übersetzung erster Operatoren und Funktionen. Nichtsdestotrotz wird hier, mit Blick auf das Fernziel, auf Ansätze aus dem Gebiet der Parallelisierung eingegangen.

Diese Bachelorarbeit ist in sieben Kapitel strukturiert. Dabei schließt sich der *Einleitung* die *Problemstellung* in Kapitel 2 an, in der genauer die vorliegenden Problematiken und eine grobe Zielsetzung dieses Themas eine Darstellung finden. Eine Basis wird im Kapitel 3 *Grundlegende Konzepte* mit der Vorstellung der Sprachen *R* im Abschnitt 3.1 und *SQL* im Abschnitt 3.2 gelegt, bevor in den Kapiteln 4 mit dem *Stand der Technik* und 5 *Stand der Forschung* Ansätze und Methodiken vorgestellt werden, die eine verbesserte Datenauswertung anstreben. Das Kapitel 6 beschäftigt sich mit der Umsetzung oben genannter Ziele. Dafür werden zunächst in 6.1 schematisch der Ablauf von Erkennung und Übersetzung und in 6.2 Beispielprogramme vorgestellt, ehe in den Abschnitten 6.3 und 6.4 anhand der Exampel Erkennung und Übersetzung genauer betrachtet wird. Abschließend werden die Ergebnisse im Kapitel 7 zusammenfassend dargestellt und ein Ausblick anhand dieser gegeben.

2 Problemstellung

Um eine geeignete Intentions- und Aktivitätserkennung zu realisieren, ist es nötig, große Datenmengen statistisch auszuwerten. Diese Auswertung erfolgt in vielen Fällen mit der für die Manipulation, Kalkulation und graphischen Darstellung von Daten entwickelten Umgebung *R*. *R* ist jedoch nur auf Effizienz im Hauptspeicher und nicht auf Effizienz für die Verarbeitung großer Datenmengen auf externen Speichermedien ausgelegt, sodass es bei diesen zu einem Leistungsabfall kommt. Die effiziente Verwaltung von großen Datenmengen auf externen Speichermedien ist die Aufgabe von Datenbankmanagementsystemen (DBMS). Es liegt nahe, die Umgebung *R*, mit dem Vorteil der leichten Bedienbarkeit im Hinblick auf statistische Auswertung, und DBMS, mit dem Vorteil der effizienten Datenverwaltung, miteinander zu verknüpfen. In der *R*-Umgebung wird die gleichnamige Sprache verwendet, wohingegen sich *SQL* als Mittel zur Kommunikation mit dem DBMS bewährt hat.
Um den Vorteil der leicht zu bedienenden statistischen Funktionen von *R* zu erhalten, reicht die alleinige Nutzung einer Schnittstelle wie RJDBC, die es ermöglicht *SQL*-Anfragen aus *R* an ein DBMS zu senden, nicht aus. Der Programmierer muss eigenhändig den *SQL*-Ausdruck formulieren. Eine automatische Übersetzung von *R* in *SQL* wäre daher eine Bereicherung.
Eine generelle Übersetzung von einer Sprache in eine Andere ist möglich, jedoch ist das Problem zu überprüfen, ob das übersetzte Programm bei allen Eingaben das gleiche Ergebnis liefert unentscheidbar. Gleiche Problematik besteht zwischen dem übersetzten Programm und einem daraus optimierten Programm gleicher Sprache. Des Weiteren ist eine vollständige Übersetzung sehr aufwendig und nicht sinnvoll, da einige Berechnungen erwartungsgemäß in *R* aufgrund der vorliegenden Architektur kürzere Rechenzeiten haben als in *SQL*. Daher ist es praktikabel, nur bestimmte Operatoren und Funktionen, die im Hinblick auf die Intentions- und Aktivitätserkennung wichtig sind, im *R*-Programmcode zu erkennen und zu übersetzen. Es soll nur eine abschnittweise Übersetzung von *R* in *SQL* erfolgen.
Liegen die relevanten Programmabschnitte als *SQL*-Ausdrücke vor, können diese vom DBMS bearbeitet werden. Dabei gibt es auch für die Auswertung auf Datenbanken Ansätze zur Beschleunigung. Einer davon ist es, *SQL*-Anfragen parallel zu gestalten, weshalb zu prüfen ist, inwieweit Parallelität auf den automatisch generierten *SQL*-Anfragen realisiert werden kann.

3 Grundlegende Konzepte

Ziel dieser Arbeit ist es Erkennung und Übersetzung von Operatoren und Funktionen aus *R* in *SQL* zu untersuchen. Aus diesem Grund werden die beiden Sprachen in diesem Kapitel kurz vorgestellt.

3.1 R

Die Sprache *R* wurde von Ross Ihaka und Robert Gentleman 1996 als Sprache für statistische Datenanalyse und graphische Darstellung vorgestellt.
Beeinflusst ist *R* von zwei Sprachen - *S* und *Scheme*. Dabei ähnelt die Syntax der von *S* und die Semantik der von *Scheme*. Als Grund für die syntaktische Anlehnung an *S* nennen die Entwickler die umfassenden Möglichkeiten für Statistiker, gewünschte Berechnungen in einer von Ihnen gewohnten Art und Weise auszudrücken [1].

3.1.1 Vor- und Nachteile

R ist bereits seit 1996 als frei verfügbare Sprache für die statistische Auswertung bekannt. Mit dem Schritt, sich an der Syntax von *S* zu orientieren, wurde eine bereits bekannte Syntax adaptiert. Die Schreibweise von *S* und folglich von *R* ist Personen mit Hintergrund in der statistischen Auswertung sehr geläufig und dementsprechend einfach anzuwenden. Des Weiteren gibt *R* mit der Möglichkeit Daten graphisch darzustellen ein weiteres nützliches Werkzeug im Bereich der statistischen Auswertung. Über frei verfügbare Pakete des CRAN-Verzeichnisses lässt sich *R* mit nützlichen Funktionen bezüglich vieler Aufgabenbereiche erweitern.
Für den Zweck - statistische Auswertung und graphische Darstellung - entwickelt, wurde im Designprozess nicht primär auf die Optimierung der Performance Wert gelegt [2]. Das führt zu einem Nachteil von *R*: der Geschwindigkeit bei der Datenverarbeitung. *R* ist eine vergleichsweise langsame Sprache. Besonders deutlich wird dies bei der Verarbeitung großer Datenmengen.

3.1.2 Objekte

R bietet verschiedene Datenstrukturen an, die als Objekte bezeichnet und mit Symbolen benannt werden. Objekte in *R* sind zu unterscheiden nach ihren Dimensionen und danach, ob sie heterogen oder homogen sind. Heterogene Objekte können Daten verschiedener Typen aufnehmen - homogene nur Daten gleichen Typs. Die in *R* vorkommenden Objekte sind in der

	homogen	**heterogen**
eindimensional	atomarer Vektor	Liste
zweidimensional	Matrix	Datenframe
n-dimensional	Array	

Tabelle 3.1: Objekte in R

Tabelle 3.1 dargestellt. Als Standardobjekt wird der Vektor verwendet. Für diese Arbeit von Bedeutung sind die Objekte Vektor, Matrix und der Datenframe. Vektor und Matrix sind insofern wichtig, da sie bei der Intentions- und Aktivitätserkennung oft verwendet werden. Der Datenframe ist als matrixartige Struktur von Bedeutung und kapselt vorher definierte Objekte gleicher Zeilenlänge. Dadurch ist er im Aufbau einer Relation aus der Datenbankwelt ähnlich [3].
R bietet dem Programmierer keinen direkten Blick auf den Speicher, da auf unterer Ebene *C*-Code zur Speicherverwaltung verwendet wird. Jedes *R*-Objekt wird hier als Zeiger auf eine Struktur realisiert. Alle Datentypen von *R* werden dabei in *C* als `SEXPTYPE` repräsentiert. Eine Besonderheit von *R* ist, dass Symbole ebenfalls als Objekte angesehen werden. Sie können daher genauso verändert werden wie ein Objekt [4].
In *R* hat jedes Objekt einen `mode`, `typeof` und `storage.mode`, welche mit den gleichnamigen Funktionen abgerufen werden können. Die Funktion `typeof` beschreibt von welchem Typ das Objekt ist und `mode` gibt Informationen über den Modus des Objektes zurück, wie ihn die Entwickler von *S*, Chambers, Becker und Wilks, beschreiben, und sorgt daher für mehr Kompatibilität mit anderen *S*-Implementationen [4]. Der `storage.mode` hingegen sorgt für die Kompatibilität mit Sprachen wie *C* und *FORTRAN*. Dies ist nötig, da z.B. der `typeof integer` und `double` beide als `mode numeric` beschrieben werden. Eine Behandlung nur mit `mode` würde in *C* zu einem Fehlverhalten führen, da `integer` und `double` unterschiedliche Datentypen in *C* sind.
Objekte werden in *R* über verschiedene Funktionen erstellt, weshalb jene im Folgenden für die Objekte Vektor, Matrix und Datenframe vorgestellt werden. Innerhalb der Vorstellung des Vektors wird auf die möglichen Datentypen eingegangen, da die für den Vektor gültigen Datentypen analog zu denen der anderen Objekte sind.

Vektor

Der Vektor ist das Standardobjekt von *R* und ist eine eindimensionale homogene Datenstruktur. Dabei kann ein Vektor die verschiedenen mit der Funktion `typeof` abrufbaren Datentypen beinhalten [4]:

- `logical`
- `integer`

- `double`
- `complex`
- `character`
- `raw`

Im Programmbeispiel 3.1 werden die Vektoren `x`, `y`, und `z` auf drei verschiedene Arten mit dem Inhalt (1, 2, 3) erzeugt.

```
> x <- c(1,2,3)
> y <- 1:3
> z <- seq(1,3)
> x
[1] 1 2 3
> y
[1] 1 2 3
> z
[1] 1 2 3
```

Programmbeispiel 3.1: Erstellen eines Vektors

Die drei aufgeführten Arten einen Vektor zu erstellen sind hier semantisch äquivalent zueinander, unterscheiden sich jedoch in der Syntax. Die erste Funktion `c()`, die für "verbinden"(engl. concatenate) steht, erzeugt den Vektor `x`. Die beiden nachfolgenden Varianten sind zum einen verkürzende Schreibweisen für die `c()`-Funktion, zum Anderen ist `1:3` erneut eine verkürzende Schreibweise von `seq(1,3)`. Dieses Beispiel verdeutlicht, dass syntaktisch verschiedene Funktionen von gleicher Semantik sein können. Folglich können diese drei Varianten mit dem gleichen *SQL*-Ausdruck beschrieben werden. Es gibt noch weitere Möglichkeiten einen Vektor in *R* zu erstellen, die hier aber nicht betrachtet werden.

Matrix

Die Matrix ist ein zweidimensionaler Spezialtyp des Arrayobjektes und wird mit der Funktion `matrix()` erstellt. Dazu benötigt die Funktion einen Datenvektor, eine Zeilenanzahl und eine Spaltenanzahl. Optionale Attribute sind `byrow` und `dimnames` [3].

```
1  #Funktionskopf der Matrixfunktion
2  matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames =
       NULL)
3  #Matrix erstellen
4  > m <- matrix(c(1,2,3,4), nrow=2, ncol=2)
5  > m
6    [,1] [,2]
7  [1,]    1    3
8  [2,]    2    4
9
10 #Matrix mit byrow = TRUE erstellen
11 > m <- matrix(c(1,2,3,4), nrow=2, ncol=2, byrow=TRUE)
12 > m
13   [,1] [,2]
14 [1,]    1    2
15 [2,]    3    4
16
17 #Matrix mit byrow = TRUE und dimnames
18 > m <- matrix(c(1,2,3,4), nrow=2, ncol=2, byrow=TRUE, dimnames =
       list(c("row1","row2"),c("col1","col2")))
19 > m
20   col1 col2
21 row1    1    2
22 row2    3    4
```

Programmbeispiel 3.2: Matrix erstellen

In dem Programmbeispiel 3.2 ist der Datenvektor `c(1,2,3,4)`, die Anzahl der Zeilen ist `nrow = 2` und die Anzahl der Spalten ist `ncol = 2`. In Zeile 4 wird die Matrix `m` mit diesen Werten definiert und die Zeilen 6 bis 8 zeigen die Matrix. Standardmäßig wird ein Datenvektor spaltenweise in eine Matrix eingefüllt. Mit dem Argument `byrow = TRUE` kann dies auf eine zeilenweise Befüllung umgestellt werden (Zeile 11). Das Argument `dimnames` ermöglicht die Zuweisung von Spalten- und Zeilennamen. Dieses erwartet eine Liste mit zwei Vektoren, die die Namen beinhalten, wobei hier unabhängig vom `byrow`-Wert zuerst der Vektor mit den Zeilennamen und anschließend der Vektor mit den Spaltennamen gefordert wird.

Datenframe

Der Datenframe ist dem Konstrukt einer Relation aus der Datenbankwelt sehr ähnlich, da es als inhomogenes mehrdimensionales Objekt Vektoren, Matrizen und Listen gleicher Zeilenan-

zahl kapseln kann. Um einen Datenframe zu erstellen, müssen zunächst die Objekte definiert werden, die anschließend in jenem gekapselt werden. Ist dies geschehen, kann der Datenframe mit der Funktion `data.frame()` erzeugt werden. Notwendige Attribute sind die in Matrix, Vektor oder Listenform auftretenden Datenobjekte. Diese können miteinander in einen Datenframe gekapselt werden, solange sie die gleiche Zeilenanzahl haben [3].

```
1  #Funktionskopf von data.frame
2  data.frame(..., row.names = NULL, check.rows = FALSE, check.
       names = TRUE, fix.empty.names = TRUE, stringsAsFactors =
       default.stringsAsFactors())
3
4  #data.frame erstellen
5  > x <- c(1,2)
6  > y <- c("Meier", "Schulz")
7  > df <- data.frame(x,y)
8  > df
9    x    y
10 1 1   Meier
11 2 2   Schulz
```

Programmbeispiel 3.3: data.frame erstellen

Auflistung der Argumente und ihre Bedeutung für `data.frame()`:

`...`	Argumente die entweder die Form `value` oder `tag = value` haben
`row.names`	Zahl oder Name der eine Spalte auswählt, die als Zeilennamengeber dient, oder ein `character` bzw. `integer` Vektor der die Zeilennamen beinhaltet
`check.rows`	falls `TRUE` wird vorher geprüft, ob die Länge der Objekte übereinstimmt
`check.names`	falls `TRUE` wird geprüft, ob die Namen syntaktisch korrekte Variablennamen sind und es keine Duplikate gibt
`fix.empty.names`	generiert Namen, falls sie fehlen
`stringAsFactors`	falls TRUE, werden `character` Vektoren zu Faktoren umgewandelt

Wie in Zeile 7 des Programmbeispiels 3.3 zu sehen ist, reicht die Angabe von Daten um einen Datenframe zu erstellen. Die anderen Argumente sind optional und enthalten ihre Standardwerte wie sie in Zeile 2 zu sehen sind.

3.1.3 Operatoren

Bevor eine Erkennung von Operatoren vorgenommen werden kann, muss eine Feststellung der bereits für *R* definierten Operatoren erfolgen. In der Sprachdefinition von R sind die folgenden Operatoren angegeben [4].

Operator	**Erklärung**
`-`	Minus, binär und unär
`+`	Plus, binär und unär
`!`	Negation, unär
`~`	Tilde, zur Modellierung von Formel, binär und unär
`?`	Hilfe
`:`	Sequenz, binär
`*`	Multiplikation, binär
`/`	Division, binär
`^`	Exponent, binär
`%%`	Modulo, binär
`%*%`	Matrixmultiplikation, binär
`%/%`	ganzzahlige Division, binär
`%o%`	Kreuzprodukt, binär
`%x%`	Kronecker Produkt, binär
`%in%`	Matchingoperator, binär
`<`	kleiner als, binär
`>`	größer als, binär
`==`	gleich zu, binär
`>=`	größer gleich, binär
`<=`	kleiner gleich, binär
`&`	und, vektorisiert, binär
`&&`	und, nicht vektorisiert, binär
`|`	oder, vektorisiert, binär
`||`	oder, nicht vektorisiert, binär
`<-`	linke Zuweisung, binär
`->`	rechte Zuweisung, binär
`$`	Listenuntermenge, binär

Wie in der Übersicht erkenntlich, gibt es binäre und unäre Operatoren, welche für die Weitergabe von Informationen an die Übersetzungsfunktion wichtig ist.
Eine Besonderheit von *R* ist, dass jeder Operator als Funktionsaufruf angesehen wird. Im kommenden Abschnitt 3.1.4 wird dieser Zusammenhang genauer betrachtet.

3.1.4 Parsing

Zur Erleichterung der Erkennung von Operatoren und Funktionen ist es sinnvoll, einen Parsebaum des Programms zu erstellen, in dem anschließend gesucht wird.
In *R* tritt Parsing in drei verschiedenen Varianten auf: In der Lesen-Evaluieren-Ausgeben-Schleife, dem Parsen von Textdateien und dem von `character`-Strings [4].
Die Schleife bezieht sich auf die Verwendung in der Kommandozeile, bei der die Eingabe solange gelesen wird, bis ein vollständiger Ausdruck verfügbar ist. Als zweite Variante können mit der `parse()`-Funktion Textdateien und `character`-Strings geparst werden. Für diese Arbeit von Bedeutung ist die zweite Variante, da der Programmcode als Textdatei vorliegt.
Nach dem Parsen des Programmcodes liegt ein Objekt vor, welches für jeden Ausdruck den Parsebaum in Listenform beinhaltet. Dabei ist das Wurzelelement eines jeden Ausdrucks vom Modus `function call` und entspricht dem Ausdruck selbst. Die Kindeselemente sind entweder wieder von jenem Modus oder vom Modus `symbol`, wobei Kindeselemente vom Modus `function call` erneut einen Parsebaum der gleichen Struktur beginnen. Das erste Kindeselement ist der Name der Funktion oder das Operatorsymbol und die folgenden Kindeselemente sind die zugehörigen Argumente.
Mittels der `ast()`-Funktion aus dem Paket *pryr* ist es möglich, sich den Abstrakten-Syntax-Baum darstellen zu lassen. Zur Verdeutlichung des Aufbaus vom Parse- und Abstrakten-Syntax-Baum folgt ein Vergleich der Beiden.

```
#Parsebaum
p <- parse(text=x <- c(5,1))
> p[[1]]
x <- c(5, 1)
> p[[1]][[1]]
‘<-‘
> p[[1]][[2]]
x
> p[[1]][[3]]
c(5, 1)
> p[[1]][[3]][[1]]
c
> p[[1]][[3]][[2]]
[1] 5
> p[[1]][[3]][[3]]
[1] 1
```

```
#AST mit pryr
ast(text=x <- c(5,1))

\- ()

  \- ‘<-

  \- ‘x

  \- ()

    \- ‘c

    \-  5

    \-  1
```

Das Beispiel zeigt für `x <- c(5,1)` den Parse- und Abstrakten-Syntax-Baum an, wobei dem

Symbol x der zweidimensionale Vektor (5,1) zugewiesen wird. Anhand der beiden Ausgabevarianten wird deutlich, dass jeder Ausdruck ein `function call` darstellt, da an Position `[[1]]` im Parse-Baum die `expression x <- c(5,1)` steht und im Abstrakten-Syntax-Baum an gleicher Position mit `()` ein `call`. Bestätigt wird dies durch das Aufrufen der Funktion `mode(p[[1]])`, welche den Modus `call` zurück gibt. Da das erste Kindeselement eines `call` der Funktions- bzw. Operatorname ist, muss bei der Suche jenes betrachtet werden.

3.2 SQL

SQL, Structured Query Language, ist eine Datenbanksprache für relationale Datenbanksysteme und wird heutzutage von allen kommerziellen und frei verfügbaren relationalen Datenbanksystemen unterstützt. Jedoch bestehen beim Umfang und dem Anwenden des *SQL*-Standards zwischen den Datenbanksystemen Unterschiede, weshalb es zu verschiedenen *SQL*-Dialekten kommt. Daher wird im Anschluss an *SQL* das freie Datenbanksystem *PostgreSQL* kurz betrachtet.
Obwohl die Sprache *SQL* einen Datendefinitions-, Datenanfrage-, Datenänderungs-, Datenorganisationsteil und einen Abschnitt zur Definition von imperativen Konstrukten besitzt [5], sind hier nur die ersten Beiden von wesentlicher Bedeutung, da *R* im wesentlichen Objekte definiert und Funktionen auf denen ausführt. *SQL*-Funktionen aus dem Datenänderungsteil werden im geringen Umfang verwendet.

3.2.1 SQL als Datendefinitionssprache

Als Datendefinitionssprache bietet *SQL* eine Menge an Funktionen an, die sich in die Drei-Ebenen-Schemaarchitektur einordnen lassen: dem externen Schema, konzeptionellen Schema und internen Schema. Auf der konzeptionellen Ebene gibt es die Möglichkeit mit `create table` eine Relation zu definieren, welche anschließend mit Daten gefüllt wird. Darüber hinaus bietet die Funktion `create view` aus der externen Ebene an, eine Sicht zu definieren und auf der internen Ebene ist es möglich mit `create index` Zugriffspfade zu erstellen [5], die einen schnelleren Zugriff ermöglichen.

Die `create table`-Anweisung

Mit dieser Anweisung wird ein Relationenschema für eine Basisrelation definiert, welches in ihrer konzeptuellen Form wie folgt aussieht:

```
create table relationenname(
spaltenname1 wertebereich1 [not null],
...,
spaltennamek wertebereichk [not null])
```

Nach der Ausführung einer solchen Anweisung wird ein Relationenschema in der Datenbank abgelegt und eine leere Basisrelation erstellt.
In der `create table`-Anweisung können mit dem Primärschlüssel `primary key`, Fremdschlüssel `foreign key` und der `unique`-Anweisung Integritätsbedingungen festgelegt werden, wobei jeder Primärschlüssel zur eindeutigen Identifikation eines Tupels dient. Fremdschlüssel sind Primärschlüssel von fremden Relationen und dienen als eindeutiger Verweis auf ein Tupel in einer fremden Relation. Die Bedingung `unique` legt fest, dass in einer Spalte jeder Wert nur einmal vorkommen darf und lässt folglich auch keine `null`-Werte zu [5].
Mit der `drop table`-Anweisung wird eine Relation mit all ihren Inhalten gelöscht. Über `alter table` kann man Änderungen an Spalten der Tabelle vornehmen. Jenes bezieht sich im Wesentlichen auf das Ändern, Hinzufügen und Löschen von Spalten, sowie dem Hinzufügen und Löschen von Integritätsbedingungen. Es ist jedoch auch möglich im Nachhinein den Wertebereich einer Spalte zu ändern und neue Spalten hinzuzufügen [5].
Wie auch in *R* gibt es in *SQL* verschiedene Datentypen, die unter anderem bei der `create table`-Anweisung als Wertebereich angegeben werden müssen.

Datentyp	Erklärung
`smallint` `integer, int`	ganzzahlige Werte

`decimal(p,q)`
 	
`numeric(p,q)`
	Festkommazahlen mit p Genauigkeit
	und q Nachkommastellen

`float(p)`
`real`
`double precision`
	Gleitkommazahlen

`character(n), char(n)`	Zeichenketten mit fester Länge n
`character varying(n), varchar(n)`	Zeichenketten variabler Länge n
`boolean`	Wahrheitswerte
`bit(n)` `bit varying(n)`	Bitfolgen fester Länge n Bitfolgen variablen Länge n

`date`
`time`
`timestamp`
`interval`
	Datum
	Zeit
	Zeitstempel
	Zeitintervall

Tabelle 3.2: Datentypen in SQL aus [5]

Bereits erwähnt wurde, dass sich der Umfang, in welchem *SQL*-Standards eingehalten werden, zwischen den DBMS unterscheidet, wovon auch die Datentypen betroffen sind. Deshalb sind die in der Tabelle angegebenen Datentypen nur die im *SQL*-Standard vordefinierten. Es gibt je nach Datenbanksystem noch weitere Datentypen oder andere Bezeichnungen die von den aufgelisteten abweichen. Weiterhin ist keine genaue Angabe über den Wertebereich der einzelnen Datentypen gegeben. Dies wird demnach von den Datenbanksystemen und ihrer speziellen Architektur festgelegt und im Umsetzungskapitel im Abschnitt 6.4.1 für *PostgreSQL* betrachtet.

3.2.2 SQL als Datenanfragesprache

Bisher ist geklärt, wie eine Basisrelation erstellt und verändert werden kann. Jene Relation ist nach der Definition leer und kann über den `insert into`-Befehl mit Daten gefüllt werden. Der konzeptionelle Aufbau dieses Befehls sieht folgendermaßen aus:

```
1 insert into relationenname [(attribut_1, ..., attribut_n)]
2 values
3 (konstante_1, ..., konstante_n)
```

Die Angabe der Attributliste ist optional, womit das Einfügen unvollständiger Tupel ermöglicht wird.

Mittels der Anweisung `update` lassen sich bereits eingefügte Tupel bearbeiten:

```
update relationenname
set spalte = 'wert'
where bedingung
(konstante_1, ..., konstante_n)
```

Mit den genannten Möglichkeiten können Relationen definiert, Daten eingefügt und aktualisiert werden. Der nächste Schritt ist die Abfrage von Daten, was mit Hilfe des `select from where`-Blocks geschieht, welcher sich aus den Einzelnen `select`-, `from`- und `where`-Klauseln zusammen setzt [5].
In der `select`-Klausel wird die Projektionsliste angegeben. Darüber hinaus können arithmetische Operationen und Aggregatfunktionen angegeben werden.
Die `from`-Klausel legt Relationen fest aus denen die Daten stammen. Bei der Angabe mehrerer Relationen werden diese mit dem kartesischen Produkt verknüpft.
Für Abfragen ist die Angabe einer `where`-Klausel optional. In ihr werden Selektions- und Verbundbedingungen festgelegt. Gleichzeitig ist es möglich mit der Klausel verschachtelte Abfragen zu erstellen, da in ihr die Angabe weiterer `select from where`-Blöcke erlaubt sind. Formal ist der Block nachstehend beschrieben [5]:

```
select projektionsliste
from relationenliste
[ where bedingung ]
```

Die nachstehenden, aus der relationalen Algebra bekannten, Funktionen können mit dem `select from where`-Block dargestellt werden.

Projektion

Bei der Projektion werden ein oder mehrere Attribute für das Ergebnisschema ausgewählt, welche in der `select`-Klausel aufgelistet werden. Dabei können nur Attribute der Relationen angegeben werden, die in der `from`-Klausel stehen.

Selektion

Bei der Selektion werden einzelne Tupel anhand einer Selektionsbedingung ausgewählt, wobei diese in der `where`-Klausel steht. Eine Selektionsbedingung kann ein boolescher Ausdruck, wie beispielsweise ein bestimmter Attributwert sein, etwa: `where spalte='wert'`. Die Selektion

kann mit der Projektion kombiniert werden, sodass nur ausgewählte Attribute von bestimmten Tupeln ausgegeben werden.

Verbund

Obwohl innerhalb der `where`-Klausel die aus der relationalen Algebra bekannten Verbünde realisierbar sind, wurden sie in *SQL* vordefiniert. Die vordefinierten Verbünde werden in der `from`-Klausel angegeben und nachfolgend beschrieben.

Kartesisches Produkt: Das kartesische Produkt ist ein Verbund, bei dem jedes Tupel der ersten Relation mit jedem Tupel der zweiten Relation verknüpft wird. Angeben wird dieser Verbund als `select ... from relation_1 cross join relation_2`. Es ist ersichtlich, dass kein Verbundattribut nötig ist. Die Abfrage `select ... from relation_1, relation_2` erzeugt den gleichen Verbund.

Innerer Verbund: Der innere Verbund verknüpft die Datensätze zweier Relationen miteinander, die den gleichen Wert haben. Die dabei zu vergleichenden Attribute müssen angegeben werden. Dies kann wie folgt geschehen:

```
1 select ... from relation_1 inner join relation_2
2    on relation_1.attribut_x = relation_2.attribut_y
```

Natürlicher Verbund: Beim natürlichen Verbund werden die beiden Relation über die Gleichheit der Attributwerte bei gleichnamigen Attributen verbunden. Die gleichnamigen Attribute werden in der Ergebnisrelation nur einmal angezeigt. Sind keine gleichnamigen Attribute vorhanden, so mutiert der natürliche Verbund zum kartesischen Produkt.

```
1 select ... from relation_1 natural join relation_2
```

Linksseitiger Verbund: Dieser Verbund schließt alle Datensätze aus der linken Relation mit in das Ergebnis ein, auch wenn keine entsprechenden Werte in der rechten Relation existieren. Wenn keine existieren, dann werden diese mit `null` belegt. Die zu vergleichenden Attribute müssen angegeben werden.

```
1 select ... from relation_1 left join relation_2
2 on relation_1.attribut_x = relation_2.attribut_y
```

Rechtsseitiger Verbund: Der rechtsseitige Verbund definiert sich wie der linksseitige, mit dem Unterschied, dass alle Datensätze aus der rechten Relation mit in die Ergebnisrelation aufgenommen werden.

```
1 select ... from relation_1 right join relation_2
2 on relation_1.attribut_x = relation_2.attribut_y
```

Vollständiger Verbund: Ein vollständiger Verbund ist eine Kombination aus dem linksseitigen und rechtsseitigen Verbund. Hier werden daher alle Tupel beider Relationen miteinander verknüpft. Bezogen auf Primär-Fremdschlüsselverbünde bedeutet dies, dass im einen Extremfall die Ergebnisrelation genauso lang ist wie die erste bzw. zweite Relation und im Anderen die Ergebnisrelation genauso lang ist wie die beiden Relationen zusammen.

```
1 select ... from relation_1 full join relation_2
2 on relation_1.attribut_x = relation_2.attribut_y
```

Aggregation

In *SQL* (SQL-89) werden fünf Aggregatfunktionen angeboten. Diese arbeiten auf den Werten einzelner Attribute und werden in der `select`-Klausel definiert. Die Funktionen sind:

- `sum()`: Bildet die Summe der Werte über einem numerischen Attribut.
- `avg()`: Bildet den Durchschnitt der Werte über einem numerischen Attribut.
- `count()`: Zählt die Anzahl der Tupel. Mit `distinct` kann es auf die Anzahl der Werte beschränkt werden.
- `max()`: Findet das Maximum unter den Werten eines Attributs.
- `min()`: Findet das Minimum unter den Werten eines Attributs.

3.2.3 PostgreSQL

PostgreSQL ist ein freies objektrelationales Datenbankmanagementsystem (ORDBMS), das auf dem von der University of California geschriebenen Paket POSTGRES basiert.
Im Rahmen der Arbeit wurde dieses System aus mehreren Gründen verwendet. *PostgreSQL* ist Open Source und damit ohne finanziellen Aufwand nutzbar. Der aktuelle *SQL*-Standard wird zu einem sehr großen Umfang unterstützt und sorgt somit für die Bereitstellung aller

hier benötigten Operationen. Des Weiteren existiert eine vollständige Dokumentation zu *PostgreSQL* und für die folgenden Aufgaben ist es möglich *PostgreSQL* mit parallelen Ansätzen zu nutzen.

4 Stand der Technik

Innerhalb vieler Themenfelder, wie auch in der Intentions- und Aktivitätserkennung, kam die Notwendigkeit zur Verbesserung der bisherigen Datenauswertungsmethoden auf, da die Menge an Daten die Möglichkeiten überstieg jene schnell auszuwerten. In diesem Kapitel werden daher zwei Ansätze betrachtet, die eine Verbesserung bei der Auswertung von Daten darstellen. Zunächst ist mit *dplyr* ein *R*-Paket vorgestellt, dass bekannte Datenmanipulationsfunktionen in *R* zur Verfügung stellt. Weiterhin ist in dem Paket eine Funktion implementiert, die einfache *R*-Ausdrücke in *SQL* umwandelt. Im Folgenden wird ein Blick auf die parallele Auswertung ohne Datenbankunterstützung geworfen, indem das von Google-Mitarbeitern entwickelte Programmiermodel *MapReduce*, mit Möglichkeiten Daten ad-hoc parallel auszuwerten, vorgestellt wird.

4.1 Datenmanipulation mit dplyr

Das Paket *dplyr* ist die Weiterentwicklung vom Paket *plyr* und bietet Werkzeuge für die effiziente Bearbeitung von Datensätzen in *R*. Der Fokus dieser Weiterentwicklung liegt dabei auf der Bearbeitung von dem Objekt `data.frame` (siehe 3.1).
Im Grunde verfolgt *dplyr* drei Hauptziele. Zunächst sollen die wichtigsten Methoden zur Datenmanipulation in *R* einfach benutzbar werden. Für in-memory-Daten soll durch die Implementation von wichtigen Stücken in *C++* eine Leistungssteigerung erreicht und eine Schnittstelle geschaffen werden, die die Daten unabhängig vom Speicherort der Daten (z.B. Datenbank, Datenframe) mit den gleichen Methoden anspricht und bearbeitet.
Dieses Paket implementiert dafür die Methoden `select()`, `filter()`, `mutate()`, `summarise()` und `arrange()`. Diese können dabei unabhängig vom Speicherort der Daten benutzt werden. Lediglich die Performance unterscheidet sich in Abhängigkeit vom Speicherort. Dabei ist im Allgemeinen die Performance bei der Verwendung von Datenframes im Vergleich zu den externen Datenbanken besser. Eine externe Datenquelle ist jedoch angebracht, wenn die Datenmenge nicht in den internen Speicher passt [6].

Im Folgenden werden die Methoden beschrieben [6]:

`filter()`
: Wählt aus einer Tabelle Zeilen aus, die einen bestimmten Ausdruck erfüllen. Sie

ist vergleichbar mit der Selektion aus der Relationenalgebra.

`select()`
: Wählt aus einer Tabelle Spalten aus. Vergleichbar mit der Projektion.

`mutate()`
: Fügt zu einer Tabelle weitere Spalten hinzu, die sich dabei aus bereits vorhandenen Spalten berechnet.

`summarise()`
: bringt einen Datenframe in eine einzelne Zeile z.B. in Kombination mit einer Aggregatfunktion.

`arrange()`
: Ordnet eine Tabelle nach Spalten.

Weitere aus der relationalen Algebra bekannte Funktionen werden durch *dplyr* implementiert. Dazu gehören der Durchschnitt `intersect(x,y)`, Vereinigung `union(x,y)` und Differenz `setdiff(x,y)`. Ebenfalls implementiert sind die bekannten Verbünde `inner_join()`, `left_join()`, `semi_join()` und `anti_join()` [6].

translate_sql()-Funktion

Das Paket *dplyr* bietet weiterhin die Funktion `translate_sql()`. Diese ermöglicht es einfache mathematische Operationen von R in SQL zu übersetzen. Dazu zählen [6]:

- mathematische Operatoren: `+, -, *, /, %%, ^`
- mathematische Funktionen: `abs, acos, acosh, asin, asinh, atan, atan2, atanh, ceiling, cos, cosh, cot, coth, exp, floor, log, log10, round, sign, sin, sinh, sqrt, tan, tanh`
- logische Vergleichsoperationen: `<, <=, !=, >=, >, ==, %in%`
- boolesche Operatoren: `&, &&, |, ||, !, xor`

Zusammenfassend bietet das Paket *dplyr* eine Abstraktion von der Datenquelle indem es die gleiche Schnittstelle für Datenframes und entfernte Datenbanken bietet. Entgegen der Basisfunktionen von *R*, die auf der Verarbeitung von Vektoren basieren, basiert *dplyr* auf Datenframes. Im Vergleich zum Vorgänger *plyr* ist *dplyr* wesentlich schneller und bietet mehr `joins` [6]. Für den Verlauf dieser Arbeit ist im Wesentlichen nur die Funktion `translate_sql()` wichtig, da die anderen Funktionen zwar für Datenverarbeitung gedacht sind, aber entgegen unserem Ansatz nur eine Schnittstelle von *R* zur Datenbank für Datenbankerfahrene Personen bietet.

4.2 MapReduce

MapReduce ist ein Programmiermodell und eine damit assoziierte Implementation für die Verarbeitung von großen Datenmengen, die bei Google unter der Leitung von Jeffrey Dean und Sanjay Ghemawat entwickelt wurde. Als Beweggrund für die Entwicklung dieses Programmiermodells nennen die Beiden, dass die Verarbeitung großer Datenmengen eine Parallelisierung erforderte und damit eine Verkomplizierung selbst einfacher Berechnungen statt fand. Durch die *MapReduce*-Bibliothek soll es weiterhin möglich sein gewünschte Berechnungen einfach auszudrücken und gleichzeitig den komplexen Teil der Parallelisierung, Fehlertoleranz und des Loadbalancing zu verstecken. Der Ansatz geht dabei auf die aus funktionalen Programmiersprachen bekannten Map- und Reduce-Funktionen zurück. Die *MapReduce*-Bibliothek bildet somit eine einfache und leistungsstarke Schnittstelle für die automatische Parallelisierung und Verteilung großer Berechnungen [7].

Programmiermodel

Die Berechnung benötigt vom Benutzer definierte Map- und Reduce-Funktionen, die den problemlösendenden Algorithmus beinhalten. Zur Visualisierung des Programmiermodels ist dieses in der Abbildung 4.1 schematisch dargestellt.

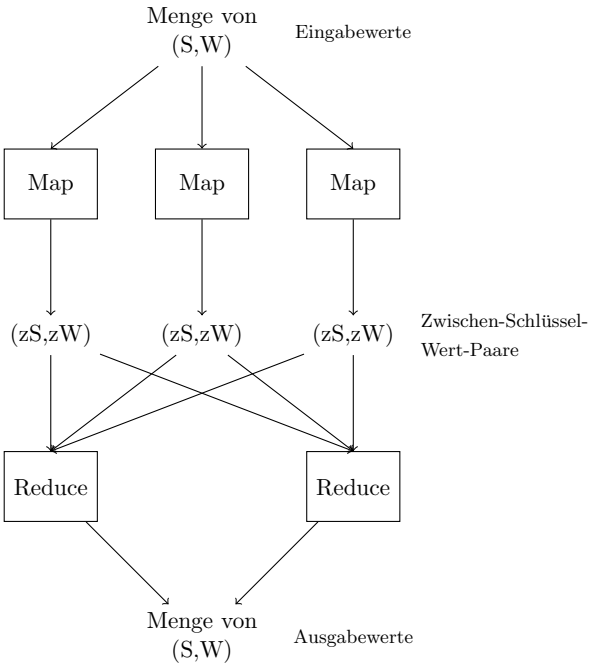


Abbildung 4.1: Programmiermodel MapReduce

Dort wird ersichtlich, dass sowohl die Ein-, als auch Ausgabewerte der Berechnung eine Menge

von Schlüssel-Wert-Paaren sind. In der Abbildung 4.1 wird die Eingabemenge in Teilmengen an drei Aufrufe der Map-Funktion gegeben, welche eine parallele Ausführung auf verschiedenen Maschinen erfahren können. Nach der Ausführung der Map-Funktion gibt es eine Menge von Zwischen-Schlüssel-Wert-Paaren, die mit Hilfe der MapReduce-Bibliothek über die Gleichheit der Zwischenschlüssel gruppiert werden. Die gruppierten Mengen werden anschließend an mehrere Ausführungen der Reduce-Funktion geschickt. Die Reduce-Funktion verbindet alle Zwischenwerte mit dem gleichen Zwischen-Schlüssel miteinander und liefert typischer Weise für jede Ausführung einen Wert oder eine kleine Menge von Werten zu einem Schlüssel [7].

Anwendungsbeispiel

Das vorgestellte Programmiermodel soll am Beispiel des Zählens der Häufigkeit des Auftretens eines Wortes in einer großen Menge von Texten verdeutlicht sein. Hierfür werden vom Benutzer die Map- und Reduce-Funktionen definiert, die im Folgenden als Pseudo-Code dargestellt sind [7]:

```
1  map(String key, String value):
2    // key: document name
3    // value: document contents
4    for each word w in value:
5      EmitIntermediate(w, "1");
6
7  reduce(String key, Iterator values):
8    // key: a word
9    // values: a list of counts
10   int result = 0;
11   for each v in values:
12     result += ParseInt(v);
13   Emit(AsString(result));
```

Programmbeispiel 4.1: Pseudo-Code Wörterzählen [7]

Das Eingabepaar der Map-Funktion besteht aus einer Zeichenkette, die dem Dokumentennamen und einer Zeichenkette, die dem Inhalt des Dokumentes entspricht. Für jedes vorkommende Wort in diesem Dokumentinhalt wird ein Zwischen-Schlüssel-Wert-Paar geniert, welches hier aus dem Wort als Schlüssel und der Zahl 1 als Wert besteht. Die Reduce-Funktion erhält als Eingabe ein solches Zwischen-Paar und summiert die Werte für ein bestimmtes Wort.
Zusätzlich werden in einem, hier nicht gezeigten, *MapReduce*-Spezifikations-Objekt die Namen der Ein- und Ausgabedateien und optionale Tuningparameter festgelegt.

MapReduce und Parallele DBMS

Befürworter vom *MapReduce*-Verfahren sehen in DBMS eine veraltete Technologie [8], wobei jedoch die Autoren von [8] beide Systeme als Ergänzung zueinander sehen. Aufgrund dieser Diskrepanz findet hier ein kurzer Vergleich der beiden Methoden anhand der Artikel [8] und [9] statt.
Zunächst soll mit den Vorteilen des *MapReduce*-Verfahren gegenüber parallelen DBMS begonnen werden. Dieses ist mittels Open Source Programmen, wie Hadoop, einfach zu installieren. Vor der Datenauswertung ist kein Laden dieser nötig und der Umgang mit Semistrukturierten Daten ist möglich. Ebenfalls bietet *MapReduce* eine gute Fehlertoleranz, da bei Ausfällen nur bestimmte Knotenpunkte eine erneute Ausführung erfahren [9],[8].
Auf Seiten der parallelen DBMS werden die folgenden Vorteile gelistet: Es sind Datenbank- und Indexstrukturen vorhanden. Die Sprache *SQL* ist leicht zu handhaben und das DBMS findet eigenständig die optimale Lösung zum Problem. Des Weiteren sind Mengenorientiere Anfragen möglich und die Effizienz liefert das System, nicht der Programmierer [9],[8].

4.3 Fazit

Die beiden vorgestellten Methoden sind Verbesserungen auf dem Gebiet der Datenverarbeitung. Das Paket *dplyr* erweitert *R* um Methoden zur Datenmanipulation und ermöglicht so Datenbankaffinen Anwendern mit bekannten Funktionsnamen zu agieren. Diese sind jedoch im Rahmen dieser Arbeit nicht relevant, da hier von *R*-affinen Anwendern ausgegangen wird. Hilfreich könnte hier die zur Übersetzung von einfachen *R*-Ausdrücken fähige Funktion `translate_sql()` sein.

Hiesige Vorstellung von *MapReduce* und der Vergleich mit parallelen DBMS zeigt, dass nur eine Datenauswertung in denen die Vorteile beider Ansätze miteinander verknüpft werden zielführend ist. So kann beispielsweise *MapReduce* aufgrund der fehlenden Ladezeit Einmal-Berechnungen Ausführen, oder Daten so formatieren, dass sie für ein DBMS strukturiert sind. Im Verlaufe der Arbeit wird jedoch dieser Ansatz der Komplexität halber nicht vertieft.

5 Stand der Forschung

Intentions- und Aktivitätserkennung erfordert Machine-Learning-Techniken auf großen Datenmengen. Die Formulierung solcher Techniken ist mit der Sprache *R* möglich, daher findet im Rahmen dieser Arbeit die Transformation von *R* in *SQL* Betrachtung. Dieses Problem ist dabei analog zu dem Problem andere Programmiersprachen in *SQL* zu übersetzen, weshalb hier mit *Database Supported Haskell* eine solche Problemlösung für die Sprache *Haskell* untersucht wird.

Im Weiteren werden gängige Konzepte zur parallelen Auswertung von *SQL*-Ausdrücken vorgestellt, die einen Einstiegspunkt in die Parallelisierung für Folgearbeiten liefern sollen.

5.1 Database Supported Haskell

Database Supported Haskell (*DSH*) ist eine *Haskell*-Bibliothek die eine Ausführung von Haskellprogrammen mittels Datenbankunterstützung ermöglicht. Hierbei kann ein relationales Datenbankmanagementsystem (RDBMS) als Koprozessor eingesetzt werden, um Programmabschnitte auszulagern. Dazu übersetzt *DSH* Haskellprogrammcode in *SQL*-Anfragen.
In [10] wird das Vorgehen von *DSH* anhand eines Beispielprogramms beschrieben, das in Standard-*Haskell* formuliert ist. Zur Weiterverwendung, dem Ausführen auf einer Datenbank, wird dieses Programm um einzelne Komponenten erweitert. Dadurch wird aus dem normalen ein *DSH*-Programm [10].
Die Modifizierungen beziehen sich auf die Typsignatur und die Einführung eines neuen Modifikators. Die Typsignaturen werden um den Buchstaben Q erweitert. Aus der Typsignatur `String -> [String]` wird die Typsignatur `Q String -> Q [String]`. Ein Auszug `Q [a]` wird als Anfrage, die den Typ `[a]` zurückgibt bezeichnet. Diese Modifizierung im Programmcode ist nötig, da sonst die Syntax und Semantik von Quasi-Quoted gleich dem der regulären *Haskell*-Listenkomprehension entspricht. Des Weiteren wird die Typsignatur für die Anfragefunktion um `IO` erweitert, welches dafür sorgt, dass der Rückgabewert innerhalb der `IO`-Monade gespeichert wird [10].
Mit `table` wird ein neuer Modifikator eingeführt, der anzeigt, dass das Programm nicht auf dem Heap, sondern auf den Daten der Datenbank arbeiten soll. Hinter `table` steht der Name der angefragten Tabelle. Mittels dieser Modifizierungen ist es möglich für *DSH* den Programmcode in *SQL* zu übersetzen. Dies geschieht im Groben in drei Schritten. Im ersten Schritt werden noch zur Übersetzungszeit die Listenkomprehensionen in listenverarbeitende Kombinatoren

umgewandelt. Die Kombinatoren wandeln im zweiten Schritt die Loop-Lifting-Technik in die Zwischendarstellung als `table algebra` um. Der Optimierer für `table algebra` und Codegenerator Pathfinder optimiert und kompiliert im dritten Schritte die Zwischendarstellung in relationale Algebra [10].

Obwohl *Database Supported Haskell* die Übersetzung von der Programmiersprache *Haskell* in *SQL* vollzieht, ist es nicht möglich, die vorgestellten Ansätze für diese Arbeit zu übernehmen, da hier vor der Übersetzung der *Haskell*-Programmcode modifiziert wird. Die *DSH*-Bibliothek erweitert demzufolge die Sprache um Komponenten, die bei der anschließenden Übersetzung helfen. Im Rahmen dieser Arbeit werden jedoch Möglichkeiten zur Übersetzung von unverändertem *R*-Programmcode in *SQL* untersucht.

5.2 Parallele Auswertung

Im Folgenden werden verschiedene Ansätze zur Parallelisierung von *SQL*-Anfragen vorgestellt, wobei hier auf die Inter- und Intraanfrageparallelität eingegangen wird. Erstere beschreibt die parallele Ausführung mehrerer verschiedener Anfragen, um den Datendurchsatz zu erhöhen und die letztere beschreibt die Möglichkeiten, innerhalb der Anfrage zu parallelisieren, um die Antwortzeit zu minimieren. Die Intraanfrage-Parallelität wird dazu unterteilt in die Intraoperator-Parallelität und Interoperator-Parallelität [11].

5.2.1 Intraoperator-Parallelität

Intraoperator-Parallelität basiert auf der Dekomposition eines Operators in mehrere unabhängige Suboperatoren, wobei jeder dieser Suboperatoren eine durch dynamische oder statische Methoden erstellte Partition einer Relation bearbeitet [11]. Bezogen auf eine horizontale Partitionierung und den Selektionsoperator lässt sich folgendes Beispiel darstellen: Eine Relation mit 10 Millionen Tupeln wird horizontal auf 50 Knoten, mit jeweils vier Platten, verteilt, wodurch auf jeder der 200 Platten eine Relation mit 50.000 Tupeln liegt [8] und es soll eine Selektion durchgeführt werden. Der Selektionsoperator wird demzufolge in 200 Suboperatoren unterteilt und jeder dieser bearbeitet genau eine der 200 Partitionen. Die Effizienz dieses Verfahrens ist von der Verteilung der Daten auf die einzelnen Partitionen abhängig, wie z.B. über Hashfunktionen, Ringverteilung oder Bereichsverteilung. Eine Hashfunktion bedingt z.B. bei einer Selektion, die genau ein Tupel auswählt, dass nur auf einer Relationspartition der Selektionsoperator ausgeführt werden muss [11]. In der folgenden Abbildung 5.1 ist schematisch eine Dekomposition des Selektionsoperators gezeigt.

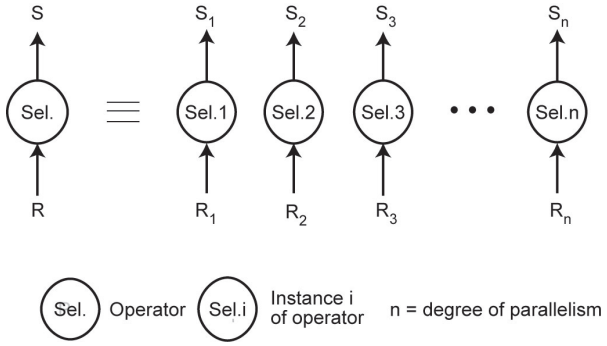


Abbildung 5.1: Schema Intraoperator Parallelität aus [11]

Wie beschrieben hängt die Effizienz von der Partitionierung ab, aber ebenfalls in Abhängigkeit vom Operator. Folglich kann eine Verteilung z.B. für den Selektionsoperator gut sein, jedoch für einen Joinoperator nicht. An dieser Stelle soll dennoch nicht genauer auf Möglichkeiten in diesem Bereich eingegangen werden.

5.2.2 Interoperator-Parallelität

Innerhalb der Interoperator-Paralleltität werden zwei Formen betrachtet: Pipeline-Parallelität und Unabhängigkeits-Parallelität [11]. Zur Erklärung der beiden nehmen wir den Operatorgraphen aus Abbildung 5.2 an.

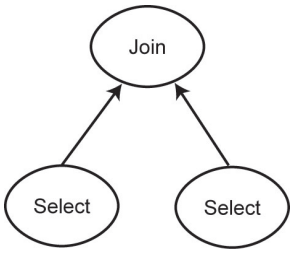


Abbildung 5.2: Operatorgraph aus [11]

Bei der Pipeline-Parallelität nehmen zwei Operatoren ein Erzeuger-Verbraucher-Verhältnis ein, wobei hier der Selektionsoperator Erzeuger und der Joinoperator Verbraucher ist. Damit beide Operatoren parallel ausgeführt werden, benötigt der Joinoperator Daten aus der Selektion, bevor diese vollständig ist. Die Daten aus der Selektion werden demnach nicht materialisiert, sondern z.B. während der Laufzeit tupelweise an den Joinoperator gegeben.
Die zweite Form ist gegeben, wenn zwischen den Operatoren keine Abhängigkeiten existieren. Da die beiden Selektionsoperatoren aus der Abbildung 5.2 auf unterschiedlichen Relationen selektieren, sind diese voneinander unabhängig und können demnach parallel agieren.

5.3 Fazit

Mit der Bibliothek *DSH* wurde ein Ansatz vorgestellt, der *Haskell* mit Datenbankunterstützung ausführen kann. Der Übersetzung in *SQL* geht hierbei jedoch eine Modifizierung des *Haskell*-Programmcodes voraus. Diese verhindern es, aus dem *DSH*-Konzept Ansätze für die vorliegende Arbeit zu übernehmen, da normaler *R*-Programmcode ohne weiteres Bearbeiten durch den Anwender in *SQL* übersetzt werden soll.

Die unter *Parallele Auswertungen* vorgestellten Ansätze sind auch für die hier entstehenden *SQL*-Ausdrücke relevant und können im weiteren Forschungsverlauf Verwendung finden, jedoch wird in dieser Ausarbeitung aufgrund der Komplexität nur kurz auf diese Thematik im Umsetzungskapitel eingegangen.

6 Umsetzung

Bisher wurden im Kapitel 3 die Grundlagen von *R* und *SQL* erläutert und anschließend in den Kapiteln 4 und 5 Ansätze zur verbesserten Datenauswertung vorgestellt. Nach der Vorstellung der Grundlagen ist es möglich, die Umsetzung von Erkennung und Übersetzung zu betrachten. Bevor sich jedoch Einzelheiten genauerer Betrachtung unterziehen, ist im Abschnitt 6.1 ein schematischer Ablaufplan gegeben. Zur Visualisierung der Inhalte werden in 6.2 zwei Programmbeispiele vorgestellt, an denen anschließend die Umsetzung gezeigt und in Abschnitt 6.5 überprüft wird.
Die Programmbeispiele greifen die in der Einleitung motivierten Operatoren und Funktionen aus dem Bereich der Datenbanksysteme und Matrizenrechnung auf und bilden folglich die Basis für die zu erkennenden und übersetzenden *R*-Gebilde. In 6.3 *Erkennen* wird dazu zunächst die Syntax und Semantik der ausgewählten Operatoren und Funktionen aufgezeigt, um anschließend die Suche nach diesen zu beschreiben.
Der Übersetzung bestimmter Operatoren und Funktionen in 6.4 geht die Erstellung von relationalen Strukturen im Abschnitt 6.4.2 voraus. Jene repräsentieren die *R*-Objekte in der Datenbank und ermöglichen somit die Formulierung von *SQL*-Anfragen.
Anschließend werden die zu den Programmbeispielen generierten *SQL*-Anfragen im Abschnitt 6.5 evaluiert, um mögliches Verbesserungspotenzial zu finden.

6.1 Schematischer Ablauf von Erkennung und Übersetzung

Die Abbildung 6.1 bietet einen schematischen Überblick über den Ablauf von Erkennung und Übersetzung eines *R*-Programms in *SQL*. Dafür werden die im R-Quellcode vorkommenden Ausdrücke mit der integrierten Funktion `parse()` in Parsebaumobjekte gespeichert. Das entstandene Objekt enthält alle Nicht-Terminale und Terminale und bietet daher die Suchgrundlage. In Bezug auf die spätere Übersetzung ist es von Vorteil, die entsprechenden Ebenen zu den Ausdrücken zu kennen. Daher werden die Objekte in einen Ausdrucksbaum umgewandelt. Beginnend bei der tiefsten Ebene im Ausdrucksbaum wird der Unterausdruck im zweiten Schritt in einen Symbolbaum umgewandelt. Auf diesem Symbolbaum beginnt der dritte Schritt: Das Suchen von Operatoren. Wurde eine übersetzbare Operation oder Funktion gefunden, so wird im vierten Schritt eine entsprechende Übersetzungsfunktion ausgelöst. Der durch die Funktion erzeugte *SQL*-Ausdruck ersetzt im Ausdrucksbaum den *R*-Ausdruck. Solange weitere Ausdrücke im Ausdrucksbaum vorhanden sind, werden die Schritte 2 bis 5 wiederholt.

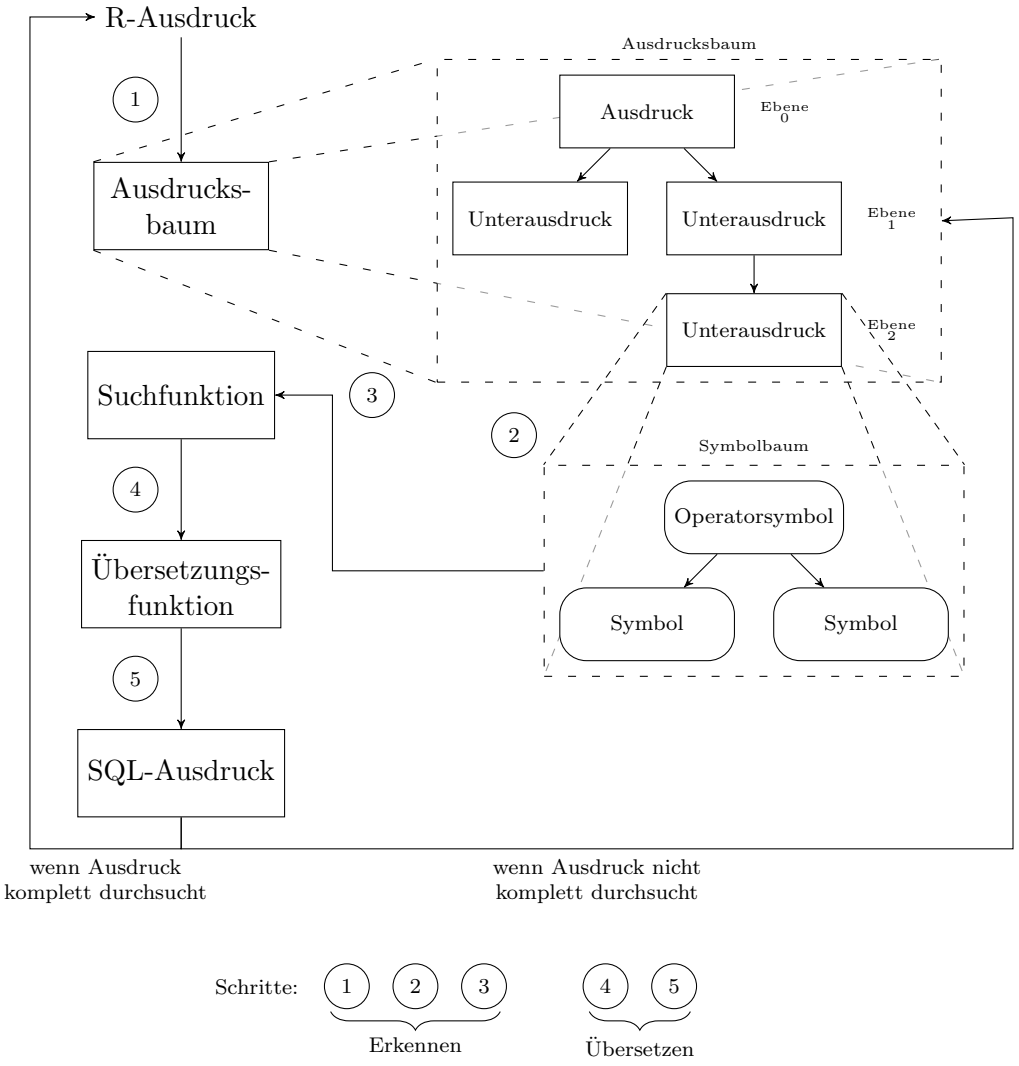


Abbildung 6.1: Schematischer Ablauf von Erkennung und Übersetzung

6.2 Einführung Programmbeispiele

Zur Verdeutlichung der Sachverhalte im Umsetzungskapitel werden hier zwei Beispiele verwendet, an denen im Anschluss auch die Übersetzung geprüft wird.
Das Programmbeispiel 6.1 mit dem zugehörigen Entity-Relationship-Modell aus Abbildung 6.2 greift die typischen Datenbankoperationen Selektion, Verbund und Aggregation auf. Das zweite Programmbeispiel 6.2 widmet sich hingegen den für Datenbanken untypischen Operationen Matrixmultiplikation und Transponierung.

6.2.1 Programmbeispiel für Selektion, Verbund und Aggregation

Wie im Entity-Relationship-Modell aus Abbildung 6.2 zu erkennen ist, handelt es sich um drei Relationen. Die vorhandenen Relationen sind die `produkt`-Relation mit den Attributen `id` (Primärschlüssel), `name` und `preis`, die Relation `kunde` mit den Attributen `id` (Primärschlüssel), `rolle` und die Relation `k_kauft_p`.

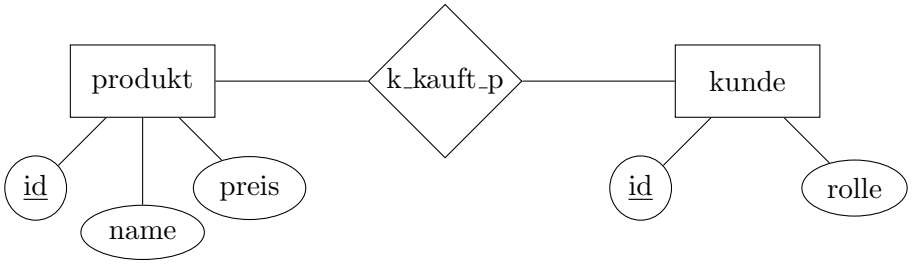


Abbildung 6.2: Entity-Relationship-Modell

In *R* wird dies über das `data.frame`-Objekte realisiert (siehe Kapitel 3). Die einzelnen Spalten werden hierbei zunächst als Vektor definiert und anschließend im Datenframe gekapselt (siehe Zeile 1 bis 12, Programmbeispiel 6.1). In den folgenden Zeilen werden die Funktionen `merge` (Zeile 15), `which`, `subset` (Zeile 19 und 20) und `sum` (Zeile 25) beschrieben.

```
1  kid <- c(1,4,5,2,3)
2  krolle <- c("Standard","Standard","Premium", "Standard", "
       Premium")
3  kunde <- data.frame(id=kid,rolle=krolle,row.names = NULL)
4
5  pname <- c("Rennrad", "Zeitfahrrad Stan", "Mountainbike", "
       Sattel", "Luftpumpe", "Zeitfahrrad Lux")
6  ppreis <- c(1499,2700,699,89,25,3400)
7  pid <- c(1,2,3,4,5,6)
8  produkt <- data.frame(id=pid,name=pname,preis=ppreis)
9
10 k_id <- c(1,5,4,2,5,3)
11 p_id <- c(1,2,3,4,5,6)
12 k_kauft_p <- data.frame(kunde=k_id,produkt=p_id)
13
14 #Verbund
15 kKp <- merge(k_kauft_p, kunde, by.x = "kunde", by.y = "id")
```

```
16 bestellungen <- merge(produkt, kKp, by.x = "id", by.y = "produkt
   ")
17
18 #Selektion
19 kunde5_kauft <- subset(bestellungen,kunde==5)
20
21 #Aggregation
22 kunde5_summe <- sum(kunde5_kauft[['preis']])
```

Programmbeispiel 6.1: Selektion, Verbund und Aggregation

6.2.2 Programmbeispiel für Matrixmultiplikation und Transponierung

Nachstehendes Programmbeispiel 6.2 definiert in den Zeilen 1 und 2 die Matrizen `m1` und `m2`. In Zeile 3 folgt die Matrixmultiplikation über den Operator `%*%` und in Zeile 4 die Transponierung über die Funktion `t()`.

```
1 m1 <- matrix(c(1,2,3,4,5,6),nrow=3,ncol=2,dimnames=list(c("row11
   ","row12","row13"),c("col11","col12")))
2 m2 <- matrix(c(1,2,3,4,5,6),nrow=2,ncol=3,dimnames=list(c("row21
   ","row22"),c("col21","col22","col23")))
3 mM <- m1 %*% m2 #Matrixmultiplikation
4 mT <- t(m1) #Transponierung
```

Programmbeispiel 6.2: Matrixoperationen

6.3 Erkennung

6.3.1 Ziele und Grenzen der Erkennung

Bevor auf die konkrete Umsetzung der Erkennung eingegangen wird, soll ein Anforderungsprofil hierfür erstellt werden.
Das Hauptaugenmerk liegt hier auf der Erkennung von Operatoren und Funktionen aus dem Bereich der Intentions- und Aktivitätserkennung. Im Allgemeinen ist es nicht praktikabel, alle in *R* vorhandenen Operatoren und Funktionen zu übersetzen, da zum einen durch Pakete immer weitere hinzugefügt werden können und zum anderen nicht jede in *SQL* darstellbar ist. Im Abschnitt 6.3.2 wird daher geklärt, welche Operatoren und Funktionen erkannt werden sollen. Die Operatoren und Funktionen werden hierbei durch syntaktische Vergleiche erkannt. Weiterhin wird jeder Ausdruck für sich behandelt. Somit wird deutlich, dass es nicht Ziel ist, semantische Zusammenhänge über mehrere Ausdrücke hinweg zu erkennen, obwohl diese möglicherweise

dadurch stark vereinfacht in *SQL* darstellbar wären.

6.3.2 Operatoren und Funktionen

In diesem Abschnitt werden die Operatoren und Funktionen vorgestellt, welche die Erkennung realisieren sollen. Ausgewählt wurden hierfür häufig genutzte Datenbankfunktionen, aber auch die für Datenbanken weniger häufig genutzten Funktionen Matrixmultiplikation und Transponierung.
Im Folgenden werden die Beziehungen zwischen *R*- und *SQL*-Funktionen beschrieben.

Selektion

Die Selektion wird mittels verschiedener Funktionen definiert: der `subset`- und der `which`-Funktion. Die Funktion `subset` gibt eine Untermenge eines Vektors, Matrix oder Datenframes zurück, die die Bedingung erfüllen. Hingegen gibt die Funktion `which` die Indizes zurück, bei denen die Selektionsbedingung wahr ist. Weiterhin sind die beiden Indexierungsfunktionen `object[index]` und `object$name` für den Vektor eine Selektionsfunktion.

Verbund

Mittels `merge` wird ein Verbund zwischen zwei Datenframes realisiert. Dabei können über das `all`-Attribut die verschiedenen bekannten Verbünde realisiert werden.

Projektion

Der Indexierungsfunktion obliegt die Besonderheit, dass sie auf allen Datentypen agieren kann, wodurch sie in Abhängigkeit vom Datentyp eine Selektion oder Projektion ist.

Beim Datentyp Vektor ist die Auswahl über `object[index]` immer eine Selektion. Hier wird aus der Attributmenge des einzig vorhandenem Attributes ein Element aus Zeile `index` selektiert.

Bei Matrizen ist die eindimensionale Anwendung über `matrix[index]` eine Projektion und Selektion, da das Element an der Stelle `index` gewählt wird. Das Hinzufügen der zweiten Dimension mittels `matrix[index-x][index-y]` führt zur Selektion des x-ten Elementes und der Projektion des y-ten Attributes.

Das Datenframeobjekt verhält sich hier wie Matrizen, mit der Ausnahme, dass die Angabe `matrix[index]` eine Projektion über die Spalte Index ist.

Aggregatfunktionen

In *SQL* gibt es nach dem Standard (SQL-89) die fünf Aggregatfunktionen `sum`, `avg`, `count`, `max` und `min`.

- `sum` wird in *R* über die gleichnamige Funktion `sum` realisiert
- `avg` ist in *R* die `mean` Funktion
- `count` entspricht der Längenfunktion `length`

- `max` und `min` sind in *R* zu *SQL* gleichnamig

Matrixmultiplikation

Matrixmultiplikation wird zwischen Matrixobjekten über den binären Operator `%*%` realisiert.

Transponierung

Transponierung einer Matrix geschieht über die Funktion `t()`.

Bereits bei der Selektion ist erkennbar, dass syntaktisch verschiedene Funktionen die gleiche Semantik besitzen können. Als *SQL*-Ausdruck werden diese jedoch nicht durch unterschiedliche Syntax repräsentiert. Neben diesen aufgelisteten Funktionen ist es wichtig, die Funktionen zu erkennen, welche die Objekte Vektor, Matrix und Datenframe erstellen. Diese Funktionen wurden im Kapitel 3 vorgestellt.

6.3.3 Vorbereitung der Suche

In der Abbildung 6.1 ist erkenntlich, dass vor der Suche nach Operatoren und Funktionen noch zwei Schritte durchgeführt werden müssen. Diese dienen der Umwandlung des Parsebaumes in den Ausdrucksbaum und anschließend in den Symbolbaum.
Für die sich anschließende Übersetzung ist es sinnvoll, den Ausdruck der untersten Ebene als erstes zu bearbeiten, da dieser keine weiteren Nicht-Terminale enthält. In dem Parseobjekt, das durch Ausführen von der `parse`-Funktion entsteht, ist die Ebene eines Ausdrucks schwer erkennbar. Daher wird mittels der Funktion `makeCallTree` ein Ausdrucksbaum produziert (vgl. Abbildung 6.1), der zu einem Ausdruck den Ausdruck selbst, alle Unterausdrücke und die zugehörigen Ebenen speichert. Hierbei steht der Ausdruck an Ebene 0, wobei sich der Ebenenindex mit fortschreitender Tiefe erhöht. Die unterste Ebene ist somit dem höchsten Index zugeordnet.

```
1  makeCallTree <- function(tt,eb) {
2    ttlist <- as.list(tt)
3    eb <- eb+1
4    for(e in 1:length(ttlist)) {
5      if(is.call(ttlist[[e]])) {
6        .GlobalEnv$callTree[[length(.GlobalEnv$callTree)+1]] <- c(
           list(object=ttlist[[e]],layer=eb))
7        makeCallTree(ttlist[[e]], eb)
8      }
9    }
10 }
```

Programmbeispiel 6.3: Ausdrucksbaum erzeugen

Die Funktion erhält als Argumente den Parsebaum eines Ausdrucks (`tt`) und einen Iterator für die Ebenen (`eb`), welcher mit 0 anfängt. Der Parsebaum ist ein Ausdruck, in dem nicht elementweise iteriert werden kann. Daher muss dieser zunächst in eine Liste umgewandelt werden. In dieser Liste wird nun nach Elementen mit dem Speichermodus `call` gesucht, da diese einem Unterausdruck entsprechen. Ist ein solches Element gefunden, so wird dieses zusammen mit der durch `eb` bereitgestellten Ebene in das Objekt `callTree` gespeichert. Da ein Unterausdruck wieder einen eigenen Parsebaum hat wird `makeCallTree` mit diesem Parsebaum und `eb` aufgerufen, wobei der Iterator `eb` bereits um eins erhöht wurde.
Mit diesem Ausdrucksbaum ist es möglich, zu jedem Ausdruck die entsprechende Ebene leicht zu erkennen und somit für das Erstellen des Symbolbaumes mit den Ausdrücken der untersten Ebene anzufangen. Für den Vektor `kid` des Programmbeispiels 6.1 (Zeile 1) sieht der `callTree` wie folgt aus:

```
1 [[1]]
2 [[1]]$object
3 c(1, 4, 5, 2, 3)
4
5 [[1]]$layer
6 [1] 1
```

Programmbeispiel 6.4: Ausdrucksbaum Vektor kid

Der zugehörige Symbolbaum wird dabei in der `findFunc`-Funktion erzeugt, welcher als Argument lediglich die aktuelle Ebene erhält. Der `callTree` muss nicht übergeben werden, da dieser als globales Objekt vorliegt. Dabei wird der Symbolbaum durch die einfache Umwandlung des Unterausdrucks in eine Liste mit `listExp <-as.list(layerExp[[e]])` erstellt, wobei `layerExp[[e]]` ein Ausdruck ist und `listExp` der Symbolbaum.

6.3.4 Suchen nach Operatoren und Funktionen

Im Abschnitt 6.3.1 wurde bereits angedeutet, dass hier die Erkennung von Operatoren und Funktionen über Vergleiche realisiert ist. Daher wird im Symbolbaum nach bestimmten Zeichenfolgen gesucht und wenn gefunden die zugehörige Übersetzungsfunktion aufgerufen. Zum Durchsuchen des Symbolbaumes, der als Liste vorliegt, wird für jedes Element überprüft, ob es einer gewünschten Zeichenfolge entspricht. Im Folgenden ist aufgelistet, welche Operatoren und Funktionen anhand welcher Zeichenfolge erkannt werden sollen.

- Vektor: `c`
- Matrix: `matrix`

- Datenframe: `data.frame`
- Matrixmultiplikation: `%*%`
- Transponierung: `t`
- Selektion: `subset`
- Aggregationen: `sum`, `mean`, `avg`, `count`, `min` und `max`
- arithmetische Operationen: `+`, `-`, `*`, `/`

6.4 Übersetzung

Nachdem Möglichkeiten zur Erkennung von Operatoren und Funktionen vorgestellt wurden, werden in diesem Abschnitt Ansätze zur Übersetzung aufgezeigt. Zum einen muss geklärt werden, wie die Objekte relational dargestellt werden, zum anderen, welcher *R*-Datentyp zu welchem *SQL*-Datentyp passt. In beiden Punkten ist es wichtig, die komplette Funktionalität beizubehalten, aber keine Seiteneffekte zu erzeugen.

6.4.1 R- und SQL-Datentypen

Es soll nun eine Vergleich der Datentypen von *R* und *SQL* vorgenommen werden. Dabei orientieren wir uns bei *R* an dem `storage.mode`, da dieser orientiert sich hierbei mehr an den Datentypen der Sprache *C*, als der normale Modus und Typ eines Objektes [4], weshalb er hier zur Datentyperkennung verwendet wird.
Der Vektor als Standardobjekt von *R* kann folgende Speichermodi annehmen: `logical`, `integer`, `double`, `complex`, `character` und `raw` [4]. Gleiches gilt für Matrizen und Datenframes, da dieses Objekt lediglich Vektoren, Matrizen und Listen kapselt.

`logical`
: Beschreibt den Datentyp für Wahrheitswerte, bei dem es nur die zwei Zustände `TRUE` und `FALSE` gibt [3].
Der dafür entsprechende Datentyp in *PostgreSQL* ist `boolean`, der neben `TRUE` und `FALSE` auch 1 und 0 und weitere Werte annehmen kann [12].

`double`
: Beschreibt einen Vektor, der Kommazahlen mit double-precision beinhaltet. *R* und *PostgreSQL* halten sich dabei an den IEEE-754-Standard, der eine Präzision von 53 bit vorgibt [3]. In *PostgreSQL* wird dieser mit `double precision` bezeichnet [12].

`integer`

R nutzt 32-bit-`integer`, was einen Wertebereich von circa $\pm 2 * 10^9$ ergibt [3]. Ebenfalls gleich verhält sich die gleichnamige 32-bit-Variante von *PostgreSQL*. Mit `smallint` und `bigint` gibt es noch die 16-bit und 64-bit-Varianten. Diese sind für die Repräsentation von *R*-Integerzahlen jedoch nicht zu verwenden, da der Wertebereich von `smallint` zu klein ist und `bigint` den Speicher belasten würde, ohne den zusätzlichen Wertebereich nutzen zu können.

`complex`

Bezeichnet komplexe Zahlen. In *PostgreSQL* gibt es keinen vergleichbaren Datentyp. Es besteht jedoch die Möglichkeit mit z.B. zwei `double`-Zahlen einen solchen Datentypen selbst zu definieren [12].

`character`

Bezeichnet einen String von Charaktersymbolen und in *PostgreSQL* ebenfalls unter dem Namen `character(n)`, aber auch `char(n)` zu finden ist. In beiden genannten Fällen wird der String durch `n` bestimmt. Fehlende Symbole werden leer aufgefüllt. Um Strings unterschiedlicher Länge zu speichern, bis zum Erreichen einer maximalen Länge, gibt es `character varying(n)` und `varchar(n)`.

`raw`

Hält rohe Datenbytes. In *PostgreSQL* gibt es dafür den Datentyp `bit` bzw. `bit varying(n)`.

6.4.2 Darstellung von R-Objekten in relationaler Algebra

Die drei Objekte Vektor, Matrix und Datenframe sollen in eine für ein Datenbanksystem nutzbare Form gebracht werden. Die Schwierigkeit besteht dabei, die Objekte relational so darzustellen, dass sie möglichst genauso viel Funktionalität ermöglichen wie das Objekt in *R* und gleichzeitig keine Seiteneffekte entstehen.

Matrix und Vektor

Obwohl der Vektor das Standardobjekt von *R* ist, wird hier der Vektor als Spezialform einer Matrix mit einer Spalte angesehen. Dies verhindert, dass die arithmetischen Operationen für Vektor und Matrix unterschiedlich gehandhabt werden müssen. Zudem ermöglicht es Übersetzungsfunktionen, die nur auf Vektoren und Matrizen agieren, auf die Überprüfung des zu behandelnden Objekttyps zu verzichten.
Im Allgemeinen finden die Operationen auf Matrizen und Vektoren elementweise statt, weshalb jedes Element eindeutig ansprechbar sein muss. Ansprechbar ist ein Objekt über die Indexierungsfunktion per numerischen Identifikator mit `obj[i]` und über Namen mit `obj$name`. Um dies in *SQL* zu realisieren, sind neben der Datenspalte ein Attribut für den numerischen Index und eines für den Namensindex erforderlich. Diese Darstellung ist für einen Vektor umsetzbar,

jedoch fehlt für die Darstellung von Matrizen die zweite Dimension.
In der gewohnten Darstellung ähnelt eine Matrix einer normalen Relation, was für die Matrix `m1` aus dem Programmbeispiel 6.2 den folgenden schematischen Aufbau aus drei Zeilen und zwei Spalten ergibt:

1	**2**
double	*double*
1	4
2	5
3	6

Da bei dieser Darstellung ein Schlüsselattribut fehlt, ist das eindeutige Ansprechen eines Elementes nicht möglich. Durch das Hinzufügen eines solchen Schlüsselattributs in Form eines numerischen Identifikators wird dies über die Kombination aus Identifikator und Attribut, welches ebenfalls eine Zahl ist, ermöglicht. Folglich entsteht das nachstehende Schema:

id	**1**	**2**
integer	*double*	*double*
1	1	4
2	2	5
3	3	6

In diesem Schema fehlt weiterhin die Möglichkeit, über den Namensindex ein Element auszuwählen. Zur Lösung dieses Problems gibt es zwei Ansätze, die ihrerseits nicht für alle Datentypen funktionsfähig sind. Für beide Ansätze gilt, dass ein Attribut hinzugefügt wird, welches die Namen der Zeilen speichert. Zuerst wird der Ansatz für Matrizen mit numerischen Datentypen vorgestellt:

id	**rownames**	**col11**	**col12**
integer	*varchar*	*double*	*double*
0		1	2
1	row11	1	4
2	row12	2	5
3	row13	3	6

In dieser Darstellung wird das 0-te Tupel als numerischer Index für die Matrizen verwendet. Entgegen der vorherigen Darstellungen sind hier die Attributnamen die angegebenen Spaltennamen, da bei Attributen mit numerischen Datentypen keine `character`-Strings im Wertebereich zugelassen sind, aber Attributnamen mit einem `character`-String benannt werden dürfen. Folglich müssen für Matrizen mit dem Datentyp `character` die Attributnamen dem

numerischen Index entsprechen und im 0-ten Tupel wird der Namensindex gespeichert. Für andere Datentypen ist diese Darstellung funktionsunfähig. Dementsprechend fiel die Entscheidung auf eine andere Darstellung, welche die Daten umstrukturiert.
Diese Darstellung orientiert sich an [13] und hat fünf Attribute, wobei die Attribute `rowid` und `colid` gemeinsam den primären Schlüssel bilden. Zusätzlich besitzt die Relation die Namensspalten `rownames`, `colnames` und die Datenspalte `dataCol`. Sind keine Namen angegeben, bleiben die beiden Namensspalten leer. Da R von sich aus nur eindeutige Namen zulässt, ist es nicht nötig, die Spalte mit der Integritätsbedingung `unique` zu belegen. Weiterhin würde es die Möglichkeit nehmen, die Namensspalten leer zu lassen. Die Spalte `dataCol` enthält den aus Abschnitt 3.1 entsprechenden Datentypen.
Bei dieser Darstellungsvariante ist es möglich, sowohl über den numerischen Index, als auch über den Namensindex die Elemente anzusprechen. Weiterhin birgt sie den Vorteil, dass bei dünnbesetzten Matrizen die Elemente mit dem Wert 0 nicht abgespeichert werden müssten. Der Komplexität halber wird jedoch auf diese Möglichkeit verzichtet. Beim Eintragen der Daten in die Matrix wird zuerst die Spalten-ID bis zum Maximalen erhöht und anschließend die Zeilen-ID.
Die Matrix `m1` stellt sich schematisch wie folgt dar:

rowid *integer*	**colid** *integer*	**rownames** *varchar*	**colnames** *varchar*	**dataCol** *double*
1	1	row11	col11	1
1	2	row11	col12	4
2	1	row12	col11	2
2	2	row12	col12	5
3	1	row13	col11	3
3	2	row13	col12	6

In PostgreSQL wird die Basisrelation für die Matrix `m1` mit folgendem Befehl erstellt:

```
1 CREATE TABLE m1 (rowid integer, colid integer, rownames varchar
    (50), colnames varchar(50), dataCol double precision, PRIMARY
    KEY (rowid,colid));
```

Für einen Vektor steht in der `colid` immer die Zahl 1. Die `colnames`-Spalte bleibt leer.

Datenframe

Der Datenframe kapselt Vektoren, Matrizen und Listen, die voneinander abweichende Datentypen haben können, aber von gleicher Zeilenanzahl sind. Aufgrund der Inhomogenität ist

es nicht möglich, die gleiche relationale Struktur wie bei der Matrix zu wählen, da einzelne Datenspalten in Relationen nur einen Datentyp aufnehmen können. Weil die Struktur des Datenframes in *R* einer Relation aus der Datenbank sehr ähnelt ist es sinnvoll, diese beizubehalten. Um den Zugriff auf einzelne Elemente der Relation zu ermöglichen, ist es notwendig, neben den Datentattributen ein Schlüsselattribut in der Relation zu haben. In *R* ist jedoch die Definition eines Datenframes bereits über die Angabe zweier Objekte möglich, was dazu führt, dass kein Schlüsselattribut vorhanden ist. Daher ist es hier notwendig, einen Schlüssel über eindeutige Identifikationsnummern hinzuzufügen. Wenn bei der Definition Attributnamen angegeben sind, werden diese verwendet, sonst sind die Objektnamen gleichzeitig die Attributnamen. Wird eine Matrix in einem Datenframe gekapselt, dann breitet sie sich über `m` Spalten aus. Die Spaltenbeschriftung ist in dem Fall: `Matrixname.Spaltenname1` oder `Matrixname.1`.

Das Relationenschema für den Datenframe stellt sich bisher wie folgt da:

Datenframe(id, col1, col2,..., coln).

Bei einer Relation von diesem Schema ist es möglich, eine Projektion über die Spaltennamen durchzuführen, eine Selektion über die Schlüssel-ID und über die Kombination beider die Auswahl eines Elementes. Die *R*-Umgebung bietet ebenfalls die Möglichkeit, über Zeilen- und Spaltennamen Elemente auszuwählen, weshalb hier der Relation eine Spalte `rownames` zugefügt wird:

Datenframe(id, rownames, col1, col2,..., coln).

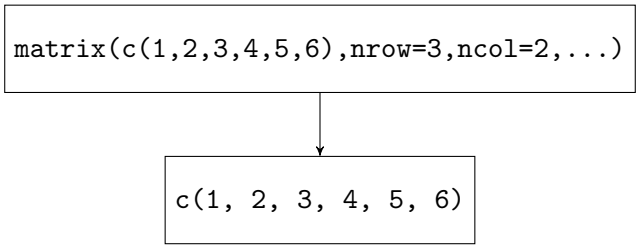
Die Angabe von Zeilennamen ist nicht Pflicht, weswegen diese Spalte leer bleibt, wenn keine angegeben sind. Mit dieser Struktur sind Selektionen und Projektionen über den Namensindex möglich und die Selektion eines Tupels ist über den numerischen Index möglich. Eine Projektion über die Angabe der Spaltennummer funktioniert hier nicht, da *SQL* keine Option bietet, im `select`-Teil das *i*-te Element einer Relation anzusprechen. Nachfolgend wird der Datenframe `produkt` aus dem Programmbeispiel 6.1 dargestellt:

id *serial*	**rownames** *varchar*	**id** *integer*	**rolle** *varchar*
1		col1	Standard
2		col2	Standard
3		col1	Premium
4		col2	Standard
5		col2	Premium

Obwohl nachfolgend die Operationen und Funktionen auf Basis dieser Struktur des Datenframes vorgestellt werden, gibt es noch andere Darstellungsformen, die hier vorgestellt werden. Eine Prüfung, ob diese praktikabler sind, ist nicht Bestandteil dieser Arbeit.
Die einfachste Veränderung wäre es, die Attribute mit Zahlen zu beschriften, wodurch eine Projektion und Selektion über den numerischen Index möglich ist, obgleich dies dazu führt, dass keine Projektion mehr über den Namensindex realisierbar ist. Eine weitere Variante ist in Betrachtung dessen, dass der Datenframe vorhandene Objekte kapselt, wodurch diese zum erstellen des Datenframes genutzt werden können. Hierfür besteht der Datenframe aus einer Relation mit Schlüssel-IDs und einer Spalte in der die Namen der gekapselten Objekte stehen, wodurch die Datenframe-Relation auf vorhandene Relationen referenziert. Für die Ausführung von Operationen auf dieser Struktur bedeutet das, dass, außer bei der Projektion über einen Vektor, die vorhandenen Relationen verbunden werden müssen.

6.4.3 Umgang mit Verschachtelung

In *R* ist es möglich innerhalb einer Funktion weitere Funktionen aufzurufen, wodurch Verschachtelungen entstehen. Diese Struktur lässt sich am einfachsten durch Ebenen darstellen, wo auf der untersten die innerste Funktion ist und die äußeren Funktionen auf den höheren. Jede Funktion einer höheren Ebene kann nur dann ausgeführt werden, wenn die in ihr aufgerufenen Funktionen unterer Ebenen ihre Resultate liefern. Folglich muss die Funktion der untersten Ebene bei einer Ausführung des *R*-Programms zu erst berechnet werden. Weiterhin fängt demnach auch die Erkennung und Übersetzung dieser an.
Verschachtelung ist auch in *SQL* möglich, indem im `where`- oder `from`-Teil der Anfrage eine weitere Anfrage gestellt wird. Wie im Kapitel 3 beschrieben gibt es in *SQL* einen Datendefinitions- und Datenanfrageteil. Der Datendefinitionsteil äußert sich durch die `create`-Anweisung und der Datenanfrageteil durch die `select from where`-Klausel. Durch diese Situation wird die Verschachtelung in *SQL* erschwert, da im `where`-Teil nur weitere Anfragen, aber keine Definitionen stehen dürfen. Ein oft vorkommendes Beispiel mit dieser Problematik ist das Erstellen einer Matrix in *R*. Die Matrix `m1` aus dem Programmbeispiel 6.2 führt zu folgendem Ausdrucksbaum:



Hinweis: Um die Übersicht zu waren wurde hier auf das Attribut dimnames verzichtet.

In diesem Funktionsaufruf sind die äußere Funktion `matrix()` und die innere Funktion `c(1, 2, 3, 4, 5, 6)` Datendefinitionen. Zuerst wird hier der Vektor über einen `create table`

-Aufruf erstellt und anschließend mit `insert into`-Befehlen mit Daten gefüllt. Diese beiden Anweisungen dürfen nicht im `where`-Teil einer Anfrage stehen, weshalb eine Verschachtelung schon aus diesem Grund nicht möglich ist. In diesem Beispiel ist eine Verschachtelung weiterhin nicht möglich, da auch die äußere Funktion einer Datendefinition entspricht und folglich kein `where`-Teil vorhanden ist. Um diese Inkonsistenz bei der Verschachtelung in *SQL* zu umgehen wird jeder Ausdruck, unabhängig von Datendefinition oder Datenanfrage, in einer temporären Relation gespeichert, wodurch beide Situationen gleich behandelt werden können. In Bezug auf die Matrix `m1` wird der Vektor mit dem Relationennamen `tbl0` erstellt. Mit dieser Relation kann nun auf der höheren Ebene gearbeitet werden, wodurch sich die Matrix aus Berechnungen des Vektors zusammensetzen lässt und als Relation `tbl1` gespeichert wird. Die zugehörogen *SQL*-Ausdrücke sind im Abschnitt 6.5 *Evaluation*.
Die temporären Relationen werden pro Ausdruck beginnend bei der untersten Ebene von links ausgehend vergeben. Im letzten Schritt jedes Ausdrucks der obersten Ebene folgt eine Zuweisung zu einem bestimmten Namen. Dafür wird die letzte temporäre Relation, im Beispiel `tbl1`, mit `select *into m1 from tbl1` in die Relation `m1` übernommen. Anschließend werden die temporären Relationen mit `drop table`-Anweisungen gelöscht.

6.4.4 Operationen in SQL

Nachdem die relationale Struktur der drei Objekte festgelegt ist, können ungeachtet vom Erkennen der Operatoren und Funktionen in *R* die typischen Operationen auf diesen Objekten mit *SQL* beschrieben werden. Sie dienen als Ziel, welches nach der Übersetzung erreicht werden soll.

Addition, Subtraktion, Multiplikation und Division

Sowohl auf Vektoren, als auch Matrizen ist die Addition, Subtraktion, Multiplikation (hier ist nicht die Matrixmultiplikation gemeint) und Division bei numerischen Datentypen möglich. Diese Operationen erfolgen dabei elementweise und das Ergebnis ist wieder ein Vektor, bzw. eine Matrix. Da der Vektor eine einspaltige-Matrix ist, werden die Rechnungen an Beispielmatrizen gezeigt. Die Ergebnismatrix soll die vorgestellte Struktur einer Matrix haben. Um dies zu realisieren wird eine Ergebnismatrix in der gewünschten Struktur erstellt und anschließend die Berechnung mit einem `INSERT INTO ... SELECT` Ausdruck in das zuvor definierte Ergebnisobjekt gespeichert.

```
1 CREATE TABLE resultMatrix (rowid integer, colid integer,
    rownames varchar(50), colnames varchar(50), dataCol double
    precision, PRIMARY KEY (rowid,colid));
2 INSERT INTO resultMatrix (rowid, colid, dataCol) SELECT m1.rowid
    ,m1.colid,(m1.dataCol + m2.dataCol) FROM m2, m1 WHERE m1.rowid
    =m2.rowid AND m1.colid=m2.colid;
```

Programmbeispiel 6.5: Addition zweier Matrizen

Die Subtraktion, Multiplikation und Division erfolgt bei Vektoren und Matrizen analog zur Addition und wird deshalb nicht vorgestellt.

Matrixmultiplikation

Die Matrixmultiplikation arbeitet auf zwei Matrizen, wobei die Spaltenanzahl der ersten Matrix mit der Zeilenanzahl der zweiten Matrix übereinstimmen muss. Entgegen der Multiplikation von zwei Skalaren ist die Matrixmultiplikation im Allgemeinen nicht kommutativ.
Die Ergebnismatrix entsteht aus komponentenweiser Multiplikation und Summation der Einträge der entsprechenden Zeile der ersten Matrix mit der entsprechenden Zeile in der zweiten Matrix und hat die Anzahl der Zeilen von der ersten Matrix und die Anzahl der Spalten von der zweiten Matrix.

Zuerst werden die nachstehende 3x2 Matrix und 2x3 Matrix mit Spalten- und Zeilennamen in *SQL* definiert:

$$m1 = \begin{array}{c} \\ row11 \\ row12 \\ row13 \end{array} \begin{array}{c} \begin{array}{cc} col11 & col12 \end{array} \\ \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} \end{array} \text{ und } m2 = \begin{array}{c} \\ row21 \\ row22 \end{array} \begin{array}{c} \begin{array}{ccc} col21 & col22 & col23 \end{array} \\ \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix} \end{array}.$$

```
1 CREATE TABLE m1 (rowid integer, colid integer, rownames varchar
    (50), colnames varchar(50), dataCol double precision, PRIMARY
    KEY (rowid,colid));
2
3 CREATE TABLE m2 (rowid integer, colid integer, rownames varchar
    (50), colnames varchar(50), dataCol double precision, PRIMARY
    KEY (rowid,colid));
```

Mit Daten gefüllt werden die Matrizen über die *SQL*-Befehle:

```
1 INSERT INTO m1 (rowid, colid, rownames, colnames, dataCol)
    VALUES (1,1,'row11','col11',1);
2 ...
3 INSERT INTO m1 (rowid, colid, rownames, colnames, dataCol)
    VALUES (3,2,'row13','col12',6);
4
5 INSERT INTO m2 (rowid, colid, rownames, colnames, dataCol)
    VALUES (1,1,'row21','col21',1);
6 ...
7 INSERT INTO m2 (rowid, colid, rownames, colnames, dataCol)
    VALUES (2,3,'row22','col23',6);
```

In *R* erhält die Ergebnismatrix die Zeilennamen der ersten Matrix und die Spaltennamen der zweiten Matrix. Dies soll mit *SQL* ebenfalls so sein.
Es wird eine Ergebnismatrix erstellt und anschließend mit `INSERT INTO ... SELECT` die Berechnung ausgeführt und in die Ergebnismatrix eingefügt.

```
1 --erstellen der Ergebnismatrix resultMatrix
2 CREATE TABLE resultMatrix (rowid integer, colid integer,
    rownames varchar(50), colnames varchar(50), dataCol double
    precision, PRIMARY KEY (rowid,colid));
3 --einfuegen und berechnen der Matrixmultiplikation
4 INSERT INTO resultMatrix (rowid,colid,rownames,colnames,dataCol)
     SELECT m1.rowid, m2.colid, m1.rownames, m2.colnames, sum(m1.
    dataCol * m2.dataCol) FROM m1,m2 WHERE m1.colid=m2.rowid GROUP
     BY m1.rowid, m2.colid, m1.rownames, m2.colnames ORDER BY
    rowid,colid;
```

Die Abfrage der Ergebnismatrix mit `SELECT *FROM resultMatrix` liefert folgende Relation:

rowid	colid	rownames	colnames	dataCol
1	1	"row11"	"col21"	9
1	2	"row11"	"col22"	19
1	3	"row11"	"col23"	29
2	1	"row12"	"col21"	12
2	2	"row12"	"col22"	26
2	3	"row12"	"col23"	40
3	1	"row13"	"col21"	15
3	2	"row13"	"col22"	33
3	3	"row13"	"col23"	51

Transponierung

Transponierung einer Matrix bedeutet nur, die Zeilen mit den Spalten zu tauschen. Die Beispielmatrix ist `m1` aus dem Abschnitt 6.4.4 Matrixmultiplikation. Es wird erneut eine Ergebnismatrix `resultMatrix` erstellt und anschließend das Ergebnis in ihr gespeichert. Für die Transponierung sind hier nur die Zeilen- und Spalten-IDs zu tauschen. Wie in *R* werden auch die Achsenbeschriftungen getauscht. Bedeutet, dass bei den Zeilen `'colxx'` und an den Spalten `'rowxx'` steht. Würde hier eine der anderen beschriebenen Darstellungsformen gewählt worden, würden die Attribute zu Schlüssel-IDs und Schlüssel-IDs zu Attributen, welches die Struktur der Relation zerstört.

```
1 --erstellen der Ergebnismatrix resultMatrix
2 CREATE TABLE resultMatrix (rowid integer, colid integer,
     rownames varchar(50), colnames varchar(50), dataCol double
     precision, PRIMARY KEY (rowid,colid));
3 --einfuegen und transponieren
4 INSERT INTO resultMatrix (rowid,colid,rownames,colnames,dataCol)
      SELECT colid,rowid,colnames,rownames,dataCol FROM m1;
```

Programmbeispiel 6.6: Transponierung einer Matrix

Die aus der relationalen Algebra bekannten Funktionen Selektion, Projektion, Verbund und Aggregation wurden im Abschnitt 3.2.2 vorgestellt. Hier wird darauf eingegangen wie diese Funktionen auf den im Abschnitt 6.4.2 Objekten realisiert werden.

Selektion und Projektion über den Indexierungsoperator

Der Abschnitt 6.3.2 beschreibt bereits die Diversität des Indexierungsoperators `[]`. Dabei ist die Ausgabe abhängig vom Objekt auf dem die Operation ausgeführt wird und ermöglicht folglich sowohl Selektionen, als auch Projektionen. Des Weiteren unterscheiden sich Matrix und

Datenframe in ihrer relationalen Struktur, weshalb im Sinne der Übersichtlichkeit hier zu erst das Verhalten auf den einzelnen Objekten und anschließend die zugehörigen *SQL*-Anfragen dargestellt werden.
Auf Vektoren ist über den Indexierungsoperator ausschließlich die Selektion möglich, da bei einer Datenspalte eine explizite Projektionsangabe für diese Spalte im Allgemeinen überflüssig ist. Hierbei soll nur die Auswahl eines Objektes über die Angabe des numeralen Indexes betrachtet werden. Folgendes Beispiel verdeutlicht die Auswahl des dritten Elementes des Vektors `kid` (Programmbeispiel 6.1) einmal in *R* und *SQL*.

```
1 kid[3] #R
2 SELECT dataCol FROM kid WHERE kid.rowid=3; #SQL
```

Bei der Matrix werden ebenfalls diese beiden Auswahlvarianten betrachtet, wobei durch die zweite Dimension der Operator als Selektion und Projektion fungieren kann. Dabei werden hier folgende Varianten verwendet:

1. Das Element r-ter Zeile und c-ter Spalte einer Matrix lässt sich über die zweidimensionale Angabe von `matrix[r,c]` ermitteln und entspricht der Selektion von der Zeile r und der Projektion über die Spalte c.

2. Spezialfälle von `matrix[r,c]` sind die Varianten `matrix[r,]` und `matrix[,c]`, welche entweder eine ganze Zeile `r` selektieren, oder eine ganze Spalte `c` projezieren.

3. Eine Sonderfall für die Matrix ist die Angabe von `matrix[i]`, welche das i-te Element von einer spaltenweise gelesenen Matrix auswählt.

Für diese drei Varianten werden hier die *SQL*-Anfragen am Beispiel der Matrix `m1` (Programmbeispiel 6.2) vorgestellt und gegebenenfalls erläutert.

```
1 m1[1,2] #R
2 SELECT dataCol FROM m1 WHERE m1.rowid=1 AND m1.colid=2; #SQL
```

Programmbeispiel 6.7: Selektion und Projektion des Elementes (1 / 2) von der Matrix m1

```
1 m1[1,] #R
2 SELECT dataCol FROM m1 WHERE m1.rowid=1; #SQL
```

Programmbeispiel 6.8: Selektion der ersten Zeile der Matrix m1

```
1 #R
2 m1[3]
```

Programmbeispiel 6.9: Auswahl des 3-ten Elementes der Matrix m1 in R

Die dritte Abfrage (Programmbeispiel 6.10) wählt für die Matrix `m1` das dritte Element aus. Dabei iteriert *R* zu erst über alle Zeilen der ersten Spalte und danach wird erst die nächste Spalte ausgewählt. Durch die Möglichkeit des `byrow` Argumentes ist dies nicht zwangsläufig das dritte Element des angegebenen Datenvektors, weshalb eine Sortierung der Relation notwendig ist. Mit Hilfe des `ORDER BY colid, rowid` Abschnitts wird die Relation zu erst nach Spalten-ID und dann nach Zeilen-ID aufsteigend sortiert. Da die `rowid` nicht mehr der Zeile in der Relation entsprechen muss, wird mit der *PostgreSQL*-Funktion `row_number()` eine eigene Spalte angefügt, die die Zeilennummer repräsentiert. Das dritte Element ist somit das an der Position `rnum=3`. Ausgehend von der gleichen relationalen Struktur von Vektor und Matrix können wir nicht entscheiden, ob es sich um eine Selektion auf einer Matrix oder einem Vektor handelt, weshalb auch die oben vorgestellte Methode zur Auswahl eines Elementes des Vektors über die Variante für die der Matrix (Programmbeispiel 6.10) erfolgen muss. Der *SQL*-Ausdruck ist nachstehend gezeigt:

```
1 #SQL
2 SELECT dataCol FROM (select row_number() OVER(ORDER BY colid,
     rowid) as rnum, dataCol FROM m1) as tmp WHERE tmp.rnum=3;
```

Programmbeispiel 6.10: Auswahl des 3-ten Elementes der Matrix m1 in SQL

Beim Datenframe ergeben sich weitere Besonderheiten: Die erste Variante der Matrix ist für den Datenframe möglich, aber die Zweite nicht. Ebenfalls nicht möglich ist die Auswahl einer Zeile des Datenframes, jedoch die Auswahl einer Spalte i, was über `datenframe[i]` realisiert wird. Analog dazu kann der Spaltenname anstelle des i stehen. Dementsprechend ergibt sich folgendes Problem: Angenommen es existieren zwei Objekte `a` und `b`, wobei `a` eine Matrix sei und `b` ein Datenframe. Die *R* Abfragen `a[i]` und `b[i]` geben syntaktisch keinen Aufschluss über das Objekt, sind jedoch semantisch abhängig von ihm. Dementsprechend muss beim Indexierungsoperator vorher geprüft werden, ob es sich um eine Matrix oder einen Datenframe handelt. Die Unterscheidung zwischen Matrix und Vektor wird aufgrund der gleichen relationalen Struktur und gleicher Semantik beim Indexierungsoperator nicht vorgenommen. Die Projektion über die Spaltennamen beim Datenframe ist trivial, wie an dem Datenframe `kunde` (Programmbeispiel 6.1) verdeutlicht wird:

```
1 #R
2 kunde[["rolle"]]
3 #SQL
4 SELECT rolle FROM kunde;
```

Programmbeispiel 6.11: Auswahl der Spalte rolle

Nicht trivial ist jedoch der Zugriff auf eine Spalte über den numerischen Index, da *SQL* keine Möglichkeit bietet die i-te Spalte in der `select`-Klausel anzugeben. Dies ist über zwei Varianten

möglich. Zum einen kann die Reihenfolge der Attributnamen mit der zugehörigen Ordinaten-ID im Data Dictionary abgefragt werden und zum anderen kann dynamisches *SQL* verwendet werden. Beides ist von der jeweiligen Implementierung des DBMS abhängig und wird hier nicht betrachtet.

Selektion über die subset()-Funktion

Neben dem Indexierungsoperator ist es möglich über die `subset()`-Funktion Daten aus Objekten zu selektieren. Obwohl diese Funktion auf Vektoren, Matrizen und Datenframes arbeiten kann, sei sich hier auf letztere beschränkt. Die `subset()`-Funktion stellt sich wie folgt dar:

```
1 subset(x, subset, select, drop = FALSE, ...)
```

Programmbeispiel 6.12: Funktionskopf der subset()-Funktion

Hierbei ist `x` das Objekt aus dem eine Teilmenge entnommen wird und `subset` entspricht einem logischen Ausdruck der angibt welche Elemente selektiert werden. Das Argument `select` bietet die Möglichkeit nur über bestimmte Spalten zu projezieren und `drop` legt fest, ob die Daten an den Indexierungsoperator übergeben werden sollen [3]. In dieser Funktion finden sich alle Bestandteile einer *SQL* `select from where`-Klausel wieder. Der `select`-Teil entspricht dem `select`-Argument, das Objekt `x` ist die Relation die im `from`-Teil angegeben wird und im `where`-Abschnitt steht der logische Ausdruck von `subset`. Folglich ergibt sich schematisch der *SQL*-Ausdruck:

```
1 SELECT select FROM x WHERE subset;
```

Programmbeispiel 6.13: Schematische SQL-Anfrage von subset()

Am Datenframe `produkt` (Programmbeispiel 6.1) soll die Vorgehensweise beim Einfügen der Bestandteile in die *SQL*-Anfrage verdeutlicht werden. Es wird von folgender Funktion ausgegangen:

```
1 subset(produkt,name=='Rennrad' | name=='Luftpumpe',select=c(name
    ,preis))
```

Programmbeispiel 6.14: subset()-Funktion auf dem Datenframe produkt

Der Name der Relation kann ohne Umformungen übernommen und im `from`-Teil eingefügt werden. Für den `select`-Teil müssen aus dem Vektor die Einzelnen Elemente selektiert und in der Projektionsliste angegeben werden. Dabei ist es möglich den Selektionsvektor auf jede Art die in *R* erlaubt ist zu definieren, was z.B. auch `c(id:preis)` ermöglicht. Es erfolgt jedoch nur eine Betrachtung von Vektoren, die eine einfache Auflistung der Spalten sind. Der unter `subset` angegebene Ausdruck kann nicht ohne Umformungen in den `where`-Teil übernommen werden, was beispielsweise an der unterschiedlichen Darstellung von den logischen Operatoren

UND und ODER liegt. Diese Umformung übernimmt die `translate_sql()`-Funktion aus dem im Abschnitt 4.1 vorgestelltem Paket *dplyr*:

```
> translate_sql(name=='Rennrad' | name=='Luftpumpe')
<SQL> "name" = 'Rennrad' OR "name" = 'Luftpumpe'
```

Programmbeispiel 6.15: Umformung mit der translatesql() Funktion

Folglich ergibt sich für unsere Beispielabfrage der *SQL*-Ausdruck:

```
SELECT name,preis FROM produkt WHERE "name" = 'Rennrad' OR "name
   " = 'Luftpumpe';
```

Verbund über die Funktion merge()

R bietet mit der `merge()`-Funktion die Möglichkeit Datenframes miteinander zu verbinden. Dabei stellt die Funktion genügend Optionen bereit, um die aus der relationalen Algebra bekannten Verbünde (siehe 3.2.2) zu realisieren. Die Funktion `merge()` hat folgenden Funktionskopf [3].

```
merge(x, y, by = intersect(names(x), names(y)), by.x = by, by.y
   = by, all = FALSE, all.x = all, all.y = all, sort = TRUE,
   suffixes = c(".x",".y"), incomparables = NULL, ...)
```

Programmbeispiel 6.16: Funktionskopf der Funktion merge()

Datenframes sind die Objekte `x` und `y`, welche über dem unter `by` angegebenem Attribut, oder den unter `by.x` und `by.y` angegebenen Attributen, verbunden werden. Über die Attribute `all`, `all.x` und `all.y` kommt es zur Realisation der verschiedenen Verbundarten, wobei `all = FALSE` einem natürlichen Verbund, `all.x = TRUE` einem linksseitigen Verbund und `all.y = TRUE` einem rechtsseitigen Verbund entspricht. Der vollständige Verbund lässt sich über `all = TRUE`, bzw. `all.x = TRUE, all.y = TRUE` erreichen. Mit dem Attribut `sort` kann die Reihenfolge der Ausgabe bestimmt werden, `suffixes` fügt den Spaltennamen Suffixe entsprechend des Herkunftsdatenframes an und `incomparables` sind Werte die nicht vergleichbar sind. Die letzten drei Attribute sind im Rahmen dieser Arbeit nicht zu betrachten. Schematisch ergibt sich folgender *SQL*-Ausdruck um einen Verbund zu realisieren:

```
SELECT * FROM x [NATURAL | FULL | LEFT | RIGHT] JOIN y [ON by.x=
   by.y];
```

Die aus dem Verbund entstehende Relation soll allerdings wieder einem Datenframe entsprechen und die Verbundsattribute sollen nur mit einem Attribut dargestellt werden. Wie dies erreicht wird ist im Abschnitt 6.4.8 beschrieben.

Aggregatfunktion

Die Aggregatfunktionen werden am Beispiel der Summenfunktion vorgestellt, dabei wird die Summe in *R* und *SQL* mit der `sum()`-Funktion über einem Attribut mit numerischen Attributwerten durch die Addition der einzelnen Werte gebildet. Obwohl jene Funktion auf Matrizen und Datenframes in *R* definiert ist, wird hier aufgrund der verschiedenen relationalen Strukturen dieser beiden nur auf den Datenframe eingegangen. Diese Heterogenität erfordert im Vorfeld der Übersetzung eine Prüfung, welches Datenobjekt vorliegt.
Zur Visualisierung wird der Abschnitt `kunde5_summe <-sum(kunde5_kauft[['preis])` aus dem Programmbeispiel 6.1 herangezogen. Hier wird die Funktion `sum()` auf der Spalte `preis` des Datenframes `kunde5_kauft` ausgeführt und in das Objekt `kunde5_summe` gespeichert. Diese Abschnitte können nach dem Schema `SELECT sum(Spalte)INTO NameDesZiels FROM NameDerQuelle` zu einer *SQL*-Anfrage zusammen gesetzt werden. Für das Beispiel ergibt sich daher folgender Befehl:

```
1 SELECT sum(preis) INTO kunde5_summe FROM kunde5_kauft;
```

6.4.5 Übersetzungsfunktion für objekterzeugende R-Funktionen

Im Abschnitt 6.3.4 sind die Operatoren und Funktionen aufgezeigt, die ein Vektor-, Matrix- oder Datenframeobjekt erzeugen. Beim Erkennen einer dieser Funktionen folgt ein entsprechender Aufruf der Übersetzungsfunktion. Obwohl Vektor und Matrix die gleiche Endstruktur haben, gibt es hier dennoch unterschiedliche Funktionen für die beiden Objekte, da sie über verschiedene *R*-Funktionen erstellt werden. Zusätzlich gibt es eine Funktion für den Datenframe, was zu den drei nachstehenden Funktionen führt:

- `createTableVector()`
- `createTableMatrix()`
- `createTableDataFrame()`

Für die Übersetzung in *SQL* wird in der `findFunc()`-Funktion eine der folgenden Funktionen mit dem Ausdruck in Form des Symbolbaumes, dem Ausdruck selbst und der Ebene als Argumente aufgerufen:

- `createTableVector()`, wenn die Funktion `c()` erkannt wird.

- `createTableMatrix()`, wenn die Funktion `matrix()` erkannt wird.
- `createTableDataFrame()`, wenn die Funktion `data.frame()` erkannt wird.

Innerhalb dieser Funktionen findet die Generierung des *SQL*-Programmcodes für das jeweilige Objekt statt. Dabei soll das Objekt den Strukturen aus Abschnitt 6.4.2 entsprechen.

Es gibt Programmabschnitte, die in jeder der drei Funktionen vorkommen, weshalb diese hier unabhängig vorgestellt werden. Dabei handelt es sich um die Funktion `createName()` und den Abschnitt, der den Datentypen bestimmt.

Generierung von Relationennamen mit `createName()`

Abschnitt 6.4.3 beschreibt bereits, dass jeder *SQL*-Ausdruck durch eine Relation mit dem durch `createName()` generierten Namen Repräsentation findet. Als Argument bekommt die Funktion die aktuelle Ebene in der sie aufgerufen wird. Nachfolgend ist diese gezeigt:

```
1 createName <- function(layer) {
2   tblname <- paste("tbl",.GlobalEnv$nameIterator,sep="")
3   .GlobalEnv$nameIterator <- .GlobalEnv$nameIterator+1
4   .GlobalEnv$tblnames[[length(.GlobalEnv$tblnames)+1]] <- c(list
      (tblname,layer))
5   return(tblname)
6 }
```

Programmbeispiel 6.17: Namen generieren mit createName()

Der Relationenname setzt sich als String aus `tbl` und der globalen Iteratorvariable `nameIterator` zusammen. Um den Zugriff auf die Namen im gesamten Programm zu gewährleisten, sind diese in der globalen Liste `tblnames` zusammen mit der Ebene, in der sie definiert wurden, gespeichert.

Bestimmung von Datentypen

Vor der Erstellung des *SQL*-Ausdrucks der Basisrelation für den Vektor muss der Datentyp für die Datenspalte geklärt sein. Dafür werden aus dem Symbolbaum die Elemente gefiltert, die weder den Speichermodus `symbol` noch `language` besitzen. Folglich bleiben nur noch Datenelemente übrig, welche in der Liste `layerObj` abgelegt werden. Da es sich beim Vektor um ein homogenes Objekt handelt, ist nur vom ersten Element der Speichermodus zu ermitteln und die Basisrelation in den Übersetzungsfunktionen mit dem entsprechenden Datentyp (siehe 6.4.1) zu generieren. Der Abschnitt zur Bestimmung des Datentyps `double` stellt sich wie folgt dar:

```
if(storage.mode(layerObj[[1]]) == "double") {
  sqlCreate <- paste("CREATE TABLE ",tblname," (rowid integer,
     colid integer, rownames varchar(50), colnames varchar(50),
     dataCol double precision, PRIMARY KEY (rowid,colid));", sep=
     "")
}
```

Programmbeispiel 6.18: Bestimmung des Datentyps double

Für den Datentyp `character` ergibt sich die Besonderheit, dass die maximale Länge `n` für `varchar(n)` angegeben werden muss, weshalb eine Suche nach dem längsten String in der Liste `layerObj` nötig wird.

SQL für Vektor generieren

Mittels der Funktion `createTableVector()` erfolgt die Erstellung der *SQL*-Ausdrücke, die bei Ausführung eine Relation der vorgegebenen Struktur (siehe Abschnitt 6.4.2) bilden und anschließend diese mit Daten befüllen. Als Argumente erhält die Funktion den Symbolbaum `listExp`, den Ausdruck `layerExp` und die aktuelle Ebene `layer`.
Im ersten Schritt wird über die `createName()`-Funktion der Name für die Relation generiert, um im Zweiten einen *SQL*-Ausdruck zu erstellen, der bei Ausführung eine Basisrelation mit diesem Namen und dem entsprechenden Datentyp (siehe 6.4.5) generiert.

```
CREATE TABLE tbl0 (rowid integer, colid integer, rownames
   varchar(50), colnames varchar(50), dataCol double precision,
   PRIMARY KEY (rowid,colid));
```

Programmbeispiel 6.19: SQL-Ausdruck Basisrelation des Vektors

Dieser *SQL*-Ausdruck wird als String zwischengespeichert und ist von der Form, der der Matrix gleich.
Nachfolgend sind mit einer `for`-Schleife, die über alle Elemente der Liste `layerObj` iteriert, die `insert into`-Befehle zu generieren. Diese werden am Ende der `createTableVector()` in die globale Liste `callTreeAfter` an die Stelle eingefügt, in der der zu übersetzende *R*-Ausdruck stand, und ersetzen jenen. Das Ergebnis erläuterter Vorgehensweise wird am Vergleich der Ausdrucksbäume des Vektor `kid` aus dem Programmbeispiel 6.1 vor und nach der Übersetzung gezeigt.

```
> callTree
[[1]]
[[1]]$object
c(1, 4, 5, 2, 3)
```

```
5
6 [[1]]$layer
7 [1] 1
```

Programmbeispiel 6.20: callTree des Vektors kid

Vor dem Aufruf der Übersetzungsfunktion besteht der Ausdrucksbaum aus dem Ausdruck `c(1, 4, 5, 2, 3)`, der in Ebene 1 vorkommt. Durch den Aufruf der Übersetzungsfunktion wird dieser Ausdruck durch die generierten *SQL*-Ausdrücke ersetzt, es entsteht der `callTreeAfter`:

```
1 [[1]]
2 [[1]]$object
3 [1] "CREATE TABLE tbl0 (rowid integer, colid integer, rownames
     varchar(50), colnames varchar(50), dataCol double precision,
     PRIMARY KEY (rowid,colid));INSERT INTO tbl0 (rowid,colid,
     dataCol) VALUES (1,1,1);INSERT INTO tbl0 (rowid,colid,dataCol)
      VALUES (2,1,4);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES
     (3,1,5);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (4,1,2);
     INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (5,1,3);"
4
5 [[1]]$layer
6 [1] 1
```

Programmbeispiel 6.21: callTreeAfter des Vektors kid

SQL für Matrix generieren

Obwohl Vektoren und Matrizen in dieser Arbeit die gleiche relationale Struktur erhalten, ist es notwendig, bei der Übersetzung zwischen ihnen zu unterscheiden. Grund hierfür ist, dass die beiden objekterzeugenden Funktionen verschieden sind. Eine Matrix wird über die Funktion `matrix()` definiert, welche die Argumente `data`, `nrow`, `ncol`, `byrow` und `dimnames` hat. Von diesen ist lediglich das Argument `data`, das einen Datenvektor erwartet, und entweder die Zeilenangabe über `nrow` oder die Spaltenangabe über `ncol` erforderlich. Die Argumente `byrow`, welches festlegt, wie der Datenvektor in die Matrix eingefügt wird, und `dimnames`, das den Spalten und Zeilen Namen gibt, sind optional.
Wie bereits bei der Erstellung des *SQL*-Ausdrucks für den Vektor ist es zuerst notwendig, einen Relationennamen mit `createName()` zu erzeugen und anschließend den Datentyp zu bestimmen. Hiernach unterscheidet sich jedoch diese Funktion von der `createTableVector`-Funktion, da bei einer Matrix, z.B. in Form des Datenvektors, innere Funktionen vorkommen können. Folglich gibt es zwei Möglichkeiten den Datenvektor in der `matrix()`-Funktion an zugeben:

```
mA <- matrix(c(1,2,3,4),nrow=2) #Vektor innerhalb definiert
dv <- c(1,2,3,4)
mB <- matrix(dv,nrow=2) #Vektor vorher definiert
```

Programmbeispiel 6.22: Übergabeformen des Datenvektors

Zum einen kann dies als innere Funktion, zum anderen als Namensreferenz auf einen bereits definierten Vektor geschehen. Diese beiden Fälle müssen beim Erstellen des *SQL*-Ausdrucks beachtet werden. Weiterhin ist beim Erstellen des Ausdrucks darauf zu achten, ob das Argument `byrow` gesetzt ist. Der Standardwert ist hier `false`, wodurch der Datenvektor spaltenweise in die Matrix eingefügt wird. Nach Klärung des Ursprungs vom Datenvektor und der Befüllungsreihenfolge ist die Berechnung der Zeilen- und Spalten-ID möglich. Ausgehend von einem Vektor mit dem Index 0 bis k und einer spaltenweise befüllten nxm Matrix kann für das i-te Element des Vektors die Spaltennummer mit $\lfloor (i/n) \rfloor$ und die Zeilennummer mit $i \bmod n$ berechnet werden. Da in R die Indexierung bei 1 anfängt, ergibt sich die Spaltennummer durch $\lfloor ((i-1)/n) + 1 \rfloor$ und Zeilennummer durch $((i-1) \bmod n) + 1$. Werden die Daten zeilenweise in den Vektor geschrieben, dann kommt es bei der Berechnung zur Ersetzung von n durch m. In Bezug auf die Matrix `m1` des Programmbeispiels 6.2 wird hier der Relationenname `tbl1` erzeugt und der auf einer tieferen Ebene erstellte Relationenname `tbl0` aus der globalen Liste `tblnames` gefiltert. Da das Einfügen in die Matrix `m1` spaltenweise erfolgt, ergibt sich nach Übersetzung folgender *SQL*-Ausdruck:

```
SELECT ((rowid-1)%3)+1 as rowid,FLOOR((rowid-1)/3)+1 as colid,
   rownames, colnames, dataCol INTO tbl1 FROM tbl0;
```

Jener *SQL*-Ausdruck erstellt, bei Ausführung, eine Relation mit dem Namen `tbl1` und fügt anhand der Berechnungsvorschriften die Daten aus dem Vektor `tbl0` ein. Im Ausdrucksbaum ersetzt er folgenden Ausdruck:

```
matrix(c(1, 2, 3, 4, 5, 6), nrow = 3, ncol = 2)
```

SQL für Datenframe generieren

In diesem Abschnitt wird das Erstellen der *SQL*-Befehle für den Datenframe betrachtet, welcher sich über die Funktion `data.frame` (siehe 3) definiert. Notwendige Argumente sind nur die Angabe von Objekten, die in dem Datenframe gekapselt werden sollen. Die Objekte sind bereits vorher definiert und folglich ebenfalls schon als Relationen vorhanden, wodurch die Verwendung dieser Objekte sinnvoll wäre. Obwohl jener Ansatz aufgrund der Komplexität nicht

implementiert wurde, soll dieser hier kurz schematisch vorgestellt werden. Ausgehend von einem Vektor `v <- c(1,2)` und einer Matrix `m <- matrix(c(1,2,3,4),nrow=2)` der Länge zwei, die im Datenframe `d <- data.frame(vector=v,matrix=m)` gekapselt wird, ergibt sich folgende Ausgabe:

```
> d
  vector matrix.1 matrix.2
1      1        1        3
2      2        2        4
```

Relational stellt sich der Vektor `v` so dar:

rowid *integer*	**colid** *integer*	**rownames** *varchar*	**colnames** *varchar*	**dataCol** *double*
1	1			1
2	1			2

Die Matrix `m` wird folgend repräsentiert:

rowid *integer*	**colid** *integer*	**rownames** *varchar*	**colnames** *varchar*	**dataCol** *double*
1	1			1
1	2			3
2	1			2
2	2			4

Ungeachtet der Namensgebung der Spalten und Namensspalten soll ein *SQL*-Befehl genutzt werden, der durch Verbünde die einzelnen Objekte zum Gesamtobjekt verknüpft. Bei Vektoren ist dies, um die gewünschte Struktur (siehe 6.4.2) zu erreichen, mit einem Verbund über die `rowid` möglich. Matrizen liegen hingegen in der umstrukturierten Variante vor, sodass die eine Datenspalte wieder auf mehrere verteilt werden muss. Realisierbar ist dies durch Verbünde mehrerer `SELECT FROM WHERE`-Klauseln, die im `FROM`-Teil auf `m` zugreifen und im `WHERE`-Teil über die Spalten-ID selektieren. Für unseren Beispieldatenframe ergibt sich folgende *SQL*-Anfrage:

```
SELECT v.rowid, v.dataCol, m1.dataCol, m2.dataCol FROM v
    JOIN (SELECT rowid,dataCol FROM m WHERE colid=1) m1 ON v.
       rowid=m1.rowid
    JOIN (SELECT rowid,dataCol FROM m WHERE colid=2) m2 ON v.
       rowid=m2.rowid;
```

Programmbeispiel 6.23: SQL-Anfrage Beispieldatenframe

Dieser Befehl liefert die nachfolgende Relation:

rowid *integer*	**dataCol** *double*	**dataCol** *double*	**dataCol** *double*
1	1	1	3
2	2	2	4

Obwohl die Anfrage händisch einfach zu erstellen ist, ergeben sich verschiedene Probleme bei der automatischen Erstellung dieses Befehls. Ein Grund hierfür ist, dass die Namensgebung von *R* nicht konsistent mit der in *SQL* ist, wie man an dem Namen `matrix.1` erkennt, was in *SQL* eine Anfrage der Spalte mit dem Namen "1"der Relation `matrix` entspricht. Somit muss hier eine Veränderung der Namen vorgenommen werden, die folglich auch bei Projektionen Beachtung finden muss. Eine einfache Lösung hierfür ist das Weglassen des trennenden Punktes. Es ergibt sich jedoch noch ein weiteres Problem: Vor dem Generieren der Anfrage muss geklärt sein, wie viele Spalten die Matrix hat, um die Projektionsliste korrekt anzugeben. Die Ermittlung der Spaltenanzahl geschieht mit der Aggregatfunktion `max(colid)`. Weiteres ist mit der Benutzung von statischem *SQL* nicht möglich, da nun dynamisch die Spaltennamen in die Projektionsliste geschrieben werden müssen. Dynamisches *SQL* ist jedoch abhängig vom genutzten DBMS. Neben der Nutzung von dynamischem *SQL* wäre es möglich, eine Schnittstelle wie *RJDBC* zu nutzen, um die maximale Spaltenanzahl zu erfragen und auf Basis von *R* den String weiter zusammenzusetzen. Gleiches Vorgehen gilt auch für die Verbundabschnitte der Abfrage.

Um dennoch die weiteren auf dem Datenframe agierenden Funktion implementieren und vorstellen zu können, wird in dieser Arbeit der Datenframe anders erstellt. Dies ist weniger eine Übersetzung und mehr eine Übernahme von Daten aus dem *R*-Objekt in das relationale Objekt. Hierfür wird der Abschnitt, der den Datenframe definiert, evaluiert, wodurch der Datenframe mit seinen Daten und Struktur auf *R*-Seite verfügbar ist. Folglich können die beinhalteten Daten abgefragt und mit `INSERT INTO`-Befehlen in die vorher mit `CREATE TABLE` erstellte Relation eingefügt werden. Das entstehende Datenframeobjekt hat somit die gewünschte Struktur und die Implementation weiterer Funktionen, wie Selektion und Projektion, ist möglich. Als endgültige Lösung ist dies nicht möglich, da keine Übersetzung der Funktionsstruktur in *SQL* erfolgt, sondern nur ein übernehmen der Daten von *R* in *SQL* geschieht. Darüber hinaus werden hier die bereits vorher definierten Vektor- und Matrixobjekte nicht benutzt.

6.4.6 Übersetzungsfunktionen für Addition, Subtraktion, Multiplikation und Division

Es wurde im Abschnitt 6.4.4 geklärt, wie die Addition, Subtraktion, Multiplikation und Division in *SQL* realisierbar ist. Dieses Kapitel beschäftigt sich demzufolge mit der Übersetzung jener aus *R* in *SQL*. Im Folgenden wird nur auf die Addition eingegangen, da die Übersetzung

der Subtraktion, elementeweise Multiplikation und Division analog zur Addition ist. Trotzdem sind hier die einzelnen Übersetzungsfunktionen genannt:

- `makeAddition()`
- `makeSubstraction()`
- `makeMultiplication()`
- `makeDivision()`

Die Argumente der Funktion sind der Symbolbaum `layerEle`, der Ausdrucksbaum `layerExp` und die aktuelle Ebene `layer`.
Bedingt durch die gleiche relationale Struktur von Matrix und Vektor ist es hier nicht nötig, zwischen den beiden Objekten bei der Ausführung dieser Funktion zu unterscheiden.
Vor der Generierung des *SQL*-Ausdrucks wird auch hier mit der Namensfunktion ein Relationenname erstellt und im Anschluss die auf tieferer Ebene erstellten Relationennamen aus der globalen Namensliste gefiltert und in einer lokalen Namensliste gespeichert.
Der Additionsoperator ist ein binärer Operator, wodurch in Bezug auf den Ursprung der Datenvektoren drei verschiedene Situationen auftreten können:

1. Beide Vektoren sind vordefiniert.
2. Einer der beiden Vektoren ist vordefiniert, der andere ist eine innere Funktion.
3. Beide Vektoren sind innere Funktionen.

Diese drei Situationen lassen sich über den Speichermodus der Elemente im Symbolbaum ermitteln, wobei sich im Symbolbaum `layerEle` an Position `layerEle[1]` der Operator `+` und an den Positionen `layerEle[2]` und `layerEle[3]` die beiden Datenvektoren befinden. Für den ersten Fall sind beide Elemente vom Speichermodus `symbol`. Im zweiten Fall ist eines der Elemente vom Speichermodus `symbol` und das andere `language` und im dritten Fall sind beide vom Speichermodus `language`. Für die Erstellung des *SQL*-Strings werden folglich die Relationennamen aus unterschiedlichen Quellen bezogen: Das Element vom Speichermodus `symbol` ist gleichzeitig der Name für die Relation, beim Element vom Speichermodus `language` ist der Name aus der lokalen Namensliste zu wählen.

6.4.7 Übersetzungsfunktion für Selektion

Der Abschnitt 6.4.4 beschreibt die Selektion über den Indexierungsoperator `[]`. Hier soll auf die Implementierung der Übersetzung dieser Funktion eingegangen werden, wobei diese nur auf dem Datenframe realisiert ist. Es folgt die Beschreibung der Selektion an der Programmzeile `kunde5_kauft2[['preis']]` aus dem Programmbeispiel 6.1, die eine Spalte mit dem Namen `preis` des Datenframes `kunde5_kauft` auswählt. Die zur Generierung des *SQL*-Ausdrucks

verantwortliche Funktion ist `makeSelection`. In ihr wird als erstes der Name der Relation mit `createName` (siehe 6.4.5) generiert. Dem folgt das Zusammensetzen des *SQL*-Ausdrucks, was in diesem Fall mit einer Zeile geschieht:

```
sqlSelect <- paste("SELECT tblid,rownames,",layerEle[[3]]," INTO
    ",tblname," FROM ",layerEle[[2]],";",sep="")
```

Das Objekt `layerEle` ist hier, bedingt durch den binären Operator, dreielementig: `layerEle [[1]]` ist das Symbol `[` oder `[[`, `layerEle[[2]]` ist der Name des Datenframes und `layerEle [[3]]` die auszuwählende Spalte. Der erzeugte Ausdruck wird im Anschluss im `callTreeAfter` an die entsprechende Stelle geschrieben, um den *R*-Ausdruck zu ersetzen.

6.4.8 Übersetzungsfunktion für Verbund

Die *R*-Funktion `merge` löst die zur Übersetzung verantwortliche `makeJoin`-Funktion aus. Diese beginnt erneut mit dem Erstellen eines Relationennamens über `createName`. Zum Verständnis der anschließenden Abschnitte wird hier die Ausgabe des *R*-Verbundes von `k_kauft_p` und `kunde` dargestellt:

```
kKp <- merge(k_kauft_p, kunde, by.x = "kunde", by.y = "id")
> kKp
  kunde produkt    rolle
1     1       1 Standard
2     2       4 Standard
3     3       6  Premium
4     4       3 Standard
5     5       2  Premium
6     5       5  Premium
```

An der Ausgabe wird deutlich, dass die Verbundattribute `kunde` aus `k_kauft_p` und `id` aus `kunde` nur einmal unter dem Namen `kunde` erscheinen und nach dieser Spalte sortiert ist. Beide Merkmale unterscheiden den *R*-Verbund von dem aus der relationalen Algebra bekannten Verbund. Die Ausgabe von `merge` ist erneut ein Datenframe.
Im Abschnitt 6.4.4 ist erklärt, dass die `merge`-Funktion die aus der relationalen Algebra bekannten Verbünde über Angabe der `all` Argumente erreichen kann. Implementiert ist hier der klassiche `INNER JOIN`, oder in Kurzform `JOIN`, der bei Angabe von `all = FALSE` ausgelöst wird.
Zu Generierung des *SQL*-Code reicht es wegen oben genannter Merkmale nicht aus, einen einfachen Verbund der beiden Relationen durchzuführen, da in dem Fall beide Verbundattribute

angegeben, die Verbundrelation nicht nach dem Verbundattribut sortiert wären und die relationale Struktur nicht der eines Datenframes entspräche. Die entstehende Struktur würde aus zwei `tblid`- und `rownames`-Attributen bestehen. Um das letztgenannte Problem zu lösen werden vor dem Verbund das `tblid`- und `rownames`-Attribut aus der zweiten Relation gelöscht. Dies ist hier dargestellt:

```
sqlAlterTable <- paste("SELECT * INTO tmp FROM ",layerObj[[3]],"
   ;ALTER TABLE tmp DROP COLUMN tblid;ALTER TABLE tmp DROP COLUMN
    rownames;",sep="")
```

Im Folgenden Abschnitt ist der Verbund zwischen den Relationen `k_kauft_p` und `tmp` visualisiert:

```
sqlJoin <- paste("SELECT * INTO ",tblname," FROM ",layerObj[[2]]," as a",sep="")
if(is.null(layerExp$all) & is.null(layerExp$all.x) & is.null(layerExp$all.y)) {
  if(is.null(layerExp$by.x) && is.null(layerExp$by.y)) {
    sqlJoin <- paste(sqlJoin, "NATURAL JOIN ")
  } else {
    sqlJoin <- paste(sqlJoin, "JOIN ")
  }
}
#...
sqlJoin <- paste(sqlJoin,"tmp as b ",sep="")
if(is.null(layerExp$all) & is.null(layerExp$all.x) & is.null(layerExp$all.y)) {
  if(is.null(layerExp$by.x) && is.null(layerExp$by.y)) {  natural join ohne on-teil }
      else {
    sqlJoin <- paste(sqlJoin, "ON a.",layerExp$by.x,"=b.",layerExp$by.y,";ALTER TABLE "
        ,tblname," DROP COLUMN ",layerExp$by.y,";","DROP TABLE tmp;",sep="")
  }
}
```

Im gezeigten Abschnitt wird über `if`-Anweisungen geprüft welcher Verbund vorliegt um den richtigen `character`-String auszuwählen. Vorher erfolgt die Generierung des `SELECT`-Teils und des ersten Abschnittes vom `FROM`-Teil. Anschließend wird der zweite Abschnitt des `FROM`-Teils, mit der Relation `tmp`, und dem `ON`-Teil des Verbundes erstellt. Die entstehende Relation hätte immer noch beide Verbundattribute, weshalb die Verbundspalte der zweiten Relation aus der Ergebnisrelation gelöscht wird. Der entstandene Ausdruck wird noch in den `callTreeAfter` geschrieben.

6.4.9 Übersetzungsfunktion für Aggregatfunktionen

Von den fünf genannten Aggregatfunktionen wird hier die Implementierung der Summation betrachtet, welche durch die Funktion `makeAggSum` realisiert ist. Die Summe wird über einer Spalte von einem Datenframe gebildet, weshalb innerhalb der Summenfunktion eine

Indexierungsfunktion steht, die jene auswählt. Die Funktion `makeSelection` übernimmt die Übersetzung der Indexierungsfunktion in *SQL* und hat folglich einen durch `createName` erzeugten Namen. Dieser ermittelt sich über die Abfrage von Namen, die in der vorherigen Ebene generiert wurden, was in diesem Fall nur einer ist. Nach der Ermittlung des Namens kann die Aggregatfunktion wie folgt zusammengesetzt werden:

```
1 sqlAgg <- paste("SELECT sum(",layerEle[[2]][[3]],") INTO ",
     tblname," FROM ",nameFrom[[1]],";",sep="")
```

Das Element `layerEle[[2]][[3]]` beinhaltet den Namen der Spalte, über den zu aggregieren ist.

6.4.10 Möglichkeiten der Parallelisierung

Das Themengebiet 5.2 betrachtet mit der Interanfrage- und Intraanfrage-Parallelität Ansätze *SQL*-Ausdrücke zu parallelisieren. Hier sollen diese Ansätze kurz in den Bezug zur vorliegenden Arbeit gesetzt werden.
Anhand der Struktur des Ausdrucksbaumes ist zu erkennen, dass die Operatoren und Funktionen gleicher Ebene voneinander unabhängig sind. Intraanfrage-Parallelität ist zwischen diesen Operatoren und Funktionen folglich realisierbar.
Auf dem Bereich der Interanfrage-Parallelität gibt es mit der Selektion die offensichtliche Möglichkeit Intraoperator-Parallelität zu erzeugen. Dazu ist beispielsweise eine Partitionierung der Matrixrelation und des Selektionsoperators notwendig.
Im Allgemeinen besteht zwischen alle Ebenen des Ausdrucksbaums ein Erzeuger-Verbraucher-Verhältnis, was eine Umsetzung der Interoperator-Parallelität hierfür nahe legt. Allerdings ist die derzeitige Umsetzung auf eine Zwischenspeicherung jedes *SQL*-Ausdrucks in eine gesamte Relation angewiesen. Dieses Konzept müsste hierfür überdacht und gegebenenfalls so angepasst werden, dass die Daten tupelweise an die höhere Ebene übergeben werden.

6.5 Evaluation

In diesem Abschnitt wird eine Prüfung der mit den in diesem Kapitel vorgestellten Funktionen generierten *SQL*-Ausdrücke zu den Programmbeispielen 6.1 und 6.2 vorgenommen. Dabei erfolgt ein Vergleich darüber, ob die bei Ausführung der *SQL*-Befehle entstehenden Objekte den beschriebenen relationalen Strukturen 6.4.2 entsprechen und semantisch äquivalent zu denen von R sind. Des Weiteren wird geprüft, ob die Ausgabe von Berechnungen und Zwischenergebnissen denen von R gleich sind. Die komplette Übersetzung beider Beispielprogramme befindet sich in den Programmabschnitten B.1 für das Programmbeispiel 6.1 und B.2 für das Programmbeispiel 6.2.

Programmbeispiel 6.1

Hier wird, stellvertretend für alle, auf bestimmte Abschnitte des Programmbeispiels 6.1 eingegangen. Zu jenen gehören die Definitionen eines Vektors, einer Matrix, eines Verbundes, einer Selektion und von einer Aggregatfunktionen. Mit diesen vier Abschnitten werden die im Beispiel genutzten Übersetzungsfunktionen überprüft. Folgend ist zunächst der *R*-Abschnitt gezeigt und nachstehend die Ausgabe der `translateExp`-Funktion. Die *SQL*-Befehle wurden auf einem lokalen *PostgreSQL*-System ausgeführt. Hierbei gilt für die `translateExp`-Ausgabe, dass zuerst *SQL*-Ausdrücke der tieferen Ebene ausgeführt werden. Bei Befehlen auf gleicher Ebene wird von oben nach unten ausgeführt. Hiesige Prüfung beginnt mit dem Vektor `kid`.

```
kid <- c(1,4,5,2,3)
```

Die Ausgabe der Funktion `translateExp(p[[1]])` ist folgende:

```
1 [[1]]
2 [[1]]$object
3 [1] "CREATE TABLE tbl0 (rowid integer, colid integer, rownames
     varchar(50), colnames varchar(50), dataCol double precision,
     PRIMARY KEY (rowid,colid));INSERT INTO tbl0 (rowid,colid,
     dataCol) VALUES (1,1,'1');INSERT INTO tbl0 (rowid,colid,
     dataCol) VALUES (2,1,'4');INSERT INTO tbl0 (rowid,colid,
     dataCol) VALUES (3,1,'5');INSERT INTO tbl0 (rowid,colid,
     dataCol) VALUES (4,1,'2');INSERT INTO tbl0 (rowid,colid,
     dataCol) VALUES (5,1,'3');"
4
5 [[1]]$layer
6 [1] 1
7
8 [1] "SELECT * INTO kid FROM tbl0;DROP TABLE tbl0;"
```

Der Aufruf von `kid` in *R* (links) und dem *SQL* Analogon (rechts) sind folgend dargestellt:

```
> kid
[1] 1 4 5 2 3
```

rowid	colid	rownames	colnames	dataCol
1	1			1
2	1			4
3	1			5
4	1			2
5	1			3

Die beiden dargestellten Objekte sind semantisch äquivalent zueinander. Darüber hinaus hat das relationale Objekt die in 6.4.2 betrachtete Struktur. Die Übersetzungsfunktion `createTableVector` bietet folglich die gewünschte Funktionalität.

Im nächsten Schritt wird stellvertretend der Datenframe `kunde <-data.frame(id=kid,rolle =krolle,row.names = NULL)` überprüft. Zunächst folgt wieder die Ausgabe der Funktion `translateExp(p[[3]])`:

```
17 [[1]]
18 [[1]]$object
19 [1] "CREATE TABLE tbl0 (tblid integer, rownames varchar(50), id
      double precision, rolle varchar(50));INSERT INTO tbl0 (tblid)
      VALUES (1);INSERT INTO tbl0 (tblid) VALUES (2);INSERT INTO
      tbl0 (tblid) VALUES (3);INSERT INTO tbl0 (tblid) VALUES (4);
      INSERT INTO tbl0 (tblid) VALUES (5);UPDATE tbl0 SET id='1'
      WHERE tblid=1;UPDATE tbl0 SET id='4' WHERE tblid=2;UPDATE tbl0
       SET id='5' WHERE tblid=3;UPDATE tbl0 SET id='2' WHERE tblid
      =4;UPDATE tbl0 SET id='3' WHERE tblid=5;UPDATE tbl0 SET rolle
      ='Standard' WHERE tblid=1;UPDATE tbl0 SET rolle='Standard'
      WHERE tblid=2;UPDATE tbl0 SET rolle='Premium' WHERE tblid=3;
      UPDATE tbl0 SET rolle='Standard' WHERE tblid=4;UPDATE tbl0 SET
       rolle='Premium' WHERE tblid=5;"
20
21 [[1]]$layer
22 [1] 1
23
24 [1] "SELECT * INTO kunde FROM tbl0;DROP TABLE tbl0;"
```

Nachstehend ist die Übersicht zur Ausgabe des Datenframes in *R* und *SQL* gegeben.

```
> kunde
  id    rolle
1  1 Standard
2  4 Standard
3  5  Premium
4  2 Standard
5  3  Premium
```

tblid	rownames	id	rolle
1		1	Standard
2		4	Standard
3		5	Premium
4		2	Standard
5		3	Premium

Der Vergleich macht deutlich, dass die beiden Ausgaben semantisch äquivalent zueinander sind und die im Abschnitt 6.4.2 vorgeschriebene Struktur eingehalten ist. Nachdem die zwei Abschnitte stellvertretend für Vektor- und Datenframeobjekte überprüft wurden, folgt die Prüfung des Verbundes anhand folgender Code-Zeile:

```
kKp <- merge(k_kauft_p, kunde, by.x = "kunde", by.y = "id")
```

Die Verarbreitung durch die Funktion `makeJoin` erzeugt nachstehenden *SQL*-Code:

```
81 [[1]]
82 [[1]]$object
83 [1] "SELECT * INTO tmp FROM kunde;ALTER TABLE tmp DROP COLUMN
      tblid;ALTER TABLE tmp DROP COLUMN rownames;SELECT * INTO tbl0
      FROM k_kauft_p as a JOIN tmp as b ON a.kunde=b.id ORDER BY a.
      kunde;ALTER TABLE tbl0 DROP COLUMN id;DROP TABLE tmp;"
84
85 [[1]]$layer
86 [1] 1
87
88 [1] "SELECT * INTO kKp FROM tbl0;DROP TABLE tbl0;"
```

Es folgt ein Vergleich der Ausgaben von *R* und *SQL*:

```
> kKp
  kunde produkt    rolle
1     1       1 Standard
2     2       4 Standard
3     3       6  Premium
4     4       3 Standard
5     5       2  Premium
6     5       5  Premium
```

tblid	rownames	kunde	produkt	rolle
1		1	1	Standard
2		2	4	Standard
3		3	6	Premium
4		4	3	Standard
5		5	5	Premium
6		5	2	Premium

Beide Ausgaben sind in ihrer Bedeutung gleich. Im Weiteren wird die Selektion mittels der `subset`-Methode geprüft. Dazu wird der Abschnitt `kunde5_kauft <-subset(bestellungen, kunde==5)` herangezogen, welcher nach der Übersetzung in *SQL* durch folgende Anweisungen repräsentiert wird:

```
[[1]]
[[1]]$object
[1] "SELECT * INTO tbl0 FROM bestellungen WHERE kunde = 5;"

[[1]]$layer
[1] 1

[[2]]
[[2]]$object
kunde == 5

[[2]]$layer
[1] 2

[1] "SELECT * INTO kunde5_kauft FROM tbl0;DROP TABLE tbl0;"
```

Die Ausgabe hierfür ist folgende:

```
> kunde5_kauft
  id             name preis kunde rolle
2 2 Zeitfahrrad Stan 2700   5  Premium
5 5        Luftpumpe   25   5  Premium
```

tblid	rownames	id	name	preis	kunde	rolle
2		2	Zeitfahrrad Stan	2700	5	Premium
5		5	Luftpumpe	25	5	Premium

Als letzter Abschnitt aus diesem Programmbeispiel soll die Aggregation anhand der Code-Zeile `kunde5_summe <-sum(kunde5_kauft[['preis']])` getestet werden, welche mit `kunde5_kauft[['preis']]` ebenfalls eine Projektion beinhaltet. Die Ausgabe der `translateExp`-Funktion ist:

```
112 [[1]]
113 [[1]]$object
114 [1] "SELECT sum(preis) INTO tbl1 FROM tbl0;"
115
116 [[1]]$layer
117 [1] 1
118
119 [[2]]
120 [[2]]$object
121 [1] "SELECT tblid,rownames,preis INTO tbl0 FROM kunde5_kauft;"
122
123 [[2]]$layer
124 [1] 2
125
126 [1] "SELECT * INTO kunde5_summe FROM tbl1;DROP TABLE tbl1;"
```

Nachstehend der Vergleich der *R*-Ausgabe (links) und *SQL*-Ausgabe (rechts).

```
> kunde5_summe
[1] 2725
```

sum
2725

Die beiden Abfragen erzeugen bei Ausgabe den gleichen Wert. Jedoch ist das Objekt `kunde5_summe` in *R* ein Vektor. Die durch den *SQL*-Code entstandene Relation nicht, wodurch die beiden Objekte nicht semantisch äquivalent sind.

Programmbeispiel 6.2

Analog zum obigen Vorgehen wird hier das Programmbeispiel 6.2 evaluiert. Hierbei ist zu erwähnen, dass die Namensgebung für Matrizen nicht implementiert ist, weshalb diese hier keine Betrachtung findet.
Zuerst wird der Ausdruck `m1 <- matrix(c(1,2,3,4,5,6),nrow=3,ncol=2)` mit `translateExp(p[[1]])`übersetzt. Es entsteht folgende Ausgabe:

```
1  [[1]]
2  [[1]]$object
3  [1] "SELECT ((rowid-1)%3)+1 as rowid,FLOOR((rowid-1)/3)+1 as
       colid, rownames, colnames, dataCol INTO tbl1 FROM tbl0;"
4  [[1]]$layer
5  [1] 1
6  [[2]]
7  [[2]]$object
8  [1] "CREATE TABLE tbl0 (rowid integer, colid integer, rownames
       varchar(50), colnames varchar(50), dataCol double precision,
       PRIMARY KEY (rowid,colid));INSERT INTO tbl0 (rowid,colid,
       dataCol) VALUES (1,1,'1');INSERT INTO tbl0 (rowid,colid,
       dataCol) VALUES (2,1,'2');INSERT INTO tbl0 (rowid,colid,
       dataCol) VALUES (3,1,'3');INSERT INTO tbl0 (rowid,colid,
       dataCol) VALUES (4,1,'4');INSERT INTO tbl0 (rowid,colid,
       dataCol) VALUES (5,1,'5');INSERT INTO tbl0 (rowid,colid,
       dataCol) VALUES (6,1,'6');"
9  [[2]]$layer
10 [1] 2
11 [1] "SELECT * INTO m1 FROM tbl1;DROP TABLE tbl1;"
```

Die Ausgabe in *R* und *SQL* ist unterhalb dargestellt:

```
> m1
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

rowid	colid	rownames	colnames	dataCol
1	1			1
2	1			2
3	1			3
1	2			4
2	2			5
3	2			6

Beide Darstellungen sind semantisch äquivalent. Die Relation ist entsprechend der in 6.4.2 vorgestellten Struktur.

Zur Überprüfung der Matrixmultiplikation wird der folgende Ausdruck verwendet:

```
mM <- m1 %*% m2
```

Durch die Bearbeitung mit der Übersetzungsfunktion ist folgender Abschnitt generiert worden:

```
[[1]]
[[1]]$object
[1] "SELECT a.rowid as rowid, b.colid as colid, a.rownames as
   rownames, b.colnames as colnames, sum(a.dataCol * b.dataCol)
   as dataCol INTO tbl0 FROM m1 as a, m2 as b WHERE a.colid=b.
   rowid GROUP BY a.rowid,b.colid,a.rownames,b.colnames;"

[[1]]$layer
[1] 1

[1] "SELECT * INTO mM FROM tbl0;DROP TABLE tbl0;"
```

Nach Ausführung auf *R* und *PostgreSQL* ergeben sich folgende Ausgaben:

```
> mM
     [,1] [,2] [,3]
[1,]    9   19   29
[2,]   12   26   40
[3,]   15   33   51
```

rowid	colid	rownames	colnames	dataCol
3	1			15
2	2			26
2	3			40
1	1			9
1	2			19
2	1			12
1	3			29
3	2			33
3	3			51

Den Abschluss der Überprüfung macht die Transponierung mittels der nachstehenden Beispielzeile:

```
mT <- t(m1)
```

Die Übersetzung erstellt folgende Ausgabe:

```
[[1]]
[[1]]$object
[1] "SELECT a.colid as rowid, a.rowid as colid, a.colnames as
   rownames, a.rownames as colnames, a.dataCol INSERT INTO tbl0
   FROM m1 as a;"

[[1]]$layer
[1] 1

[1] "SELECT * INTO mT FROM tbl0;DROP TABLE tbl0;"
```

Nachstehend ist auch diese Ausgabe für R und SQL visualisiert:

```
> mT
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

rowid	colid	rownames	colnames	dataCol
1	1			1
1	2			2
1	3			3
2	1			4
2	2			5
2	3			6

Hier ist ebenfalls zu erkennen, dass die beiden Ausgabe die gleiche Semantik haben.

Die Evaluation zeigte, dass die vorgestellten Methodiken eine Übersetzung von den ausgewählten R-Operatoren in SQL ermöglichen und folglich die gewählten Ansätze im vorgestellten Maße funktionsfähig sind.

Fehlererkennung

Vor und bei der Übersetzung der Programmabschnitte aus R in SQL wird der R-Programmcode nicht auf syntaktische oder semantische Fehler überprüft. Es wird hier davon ausgegangen, dass nur syntaktisch korrekter Programmcode, der in R ausführbar ist, der Übersetzungsfunktion übergeben wird. Wie sich die Übersetzungsfunktionen in diesem Fall verhalten, ist ungeklärt.

7 Zusammenfassung und Ausblick

Abschließend werden die Inhalte dieser Arbeit zusammengefasst dargestellt und ein Ausblick auf Basis der gewonnenen Erkenntnisse gegeben. Ziel war es, Ansätze zur Erkennung, Übersetzung und parallelen Auswertung von Operatoren und Funktionen aus *R* in *SQL* zu untersuchen.
Der Einstieg fand mit der Vorstellung von *R* und *SQL* statt. Dabei gab es auf Seiten von *R* einen kurzen geschichtlichen Überblick, sowie eine genauere Betrachtung der Objekte und Datentypen. Auf Seiten von *SQL* bezog sich die Vorstellung vorrangig auf den Datendefinitions- und Datenanfrageteil. Desweiteren wurde mit *PostgreSQL* kurz das hier vorrangig verwendete DBMS präsentiert.
Im Anschluss an die Klärung der Grundlagen und Vorstellung von bisherigen Ansätzen dieses Forschungsgebietes folgte mit der Erstellung eines, als Rahmenarchitektur dienenden, Ablaufplanes, der wesentliche Punkte im Verlauf der Erkennung und Übersetzung darstellt, der Umsetzungseintieg.
Für die Suche wurde eine Umwandlung des *R*-eigenen Parsebaums in einen Ausdrucksbaum programmiert. In diesem ist eine Ebenenstruktur leicht zu erkennen, was es ermöglicht, die in *R* vorkommenenden inneren Funktionen handzuhaben. Jene innere Funktionen in *SQL* darzustellen ist problematisch, da in *SQL* keine verschachtelten Datendefinitionen möglich sind. Ein Lösungsansatz wurde in 6.4.3 vorgestellt: Jede *SQL*-Anweisung wird in einer Relation zwischengespeichert.
Zur Generierung von *SQL*-Anfragen, die semantisch *R*-Operationen entsprechen, ist eine relationale Struktur nötig, welche ein *R*-Objekt im Datenbanksystem repräsentiert. Die Schwierigkeit lag darin, die in *R* für das Objekt angebotene Funktionalität beizubehalten. Repräsentationen wurden unter diesem Gesichtspunkt für die Objekte Vektor, Matrix und Datenframe generiert, wobei es sich als praktikabel erwies, den Vektor als degenerierte Matrix zu behandeln.
Es wurden *SQL*-Ausdrücke formuliert, die Relationen erstellen. Matrizen ergeben sich dabei durch Transformation von Datenvektorrelationen und Datenframes entstehen durch Verbünde von Matrixrelationen. Die notwendige Umstrukturierung der Matrixrelation für den Datenframe wurde konzeptionell vorgestellt. Eine Implementierung der Vektor- und Matrixrelationen ist vollständig realisiert.
Auf Basis der Objektrelationen gab es eine Betrachtung der konzeptionellen Erstellung und die partielle Implementierung von Selektion, Projektion, Verbund, Aggregatfunktionen, Ma-

trixmultiplikation und Transponierung. Der Übersetzung stellt sich die implementierte Suche nach den genannten Operatoren und Funktionen voran, die über syntaktische Vergleiche im Ausdrucksbaum realisiert ist.

Im Verlauf der Arbeit sind Themenstellungen offen geblieben und neue Erkenntnisse entstanden, anhand derer hier ein Ausblick für anschließende Arbeiten erfolgt.
Das Themengebiet Erkennung implementiert derzeit die Suche über den symbolweisen Vergleich der Syntax. Größere *R*-Gebilde, die möglicherweise im Gesamten eine einfache *SQL*-Anfrage darstellen, bleiben unentdeckt und werden in Kleinststrukturen übersetzt. Eine Erkennung solcher Gebilde kann ein weiteres Ziel sein.
Die bisherigen Theorien zur Erstellung der Operatoren und Funktionen können um weitere Funktionen und Kontrollstrukturen erweitert werden.
Weiterhin wurden keine Betrachtungen der Performance des aktuellen *SQL*-Codes durchgeführt. Hier sind Optimierungen wie die Umformung mit aus der relationalen Algebra bekannten Regeln und das Anlegen von Indexstrukturen zu betrachten. Nach Ausschöpfung dieser sollte eine Untersuchung der Ansätze zur parallelen Auswertung erfolgen. Hierfür können die Veröffentlichungen [14] und [15], die die Parallelisierung von Aggregatfunktionen genauer beleuchten, eine Grundlage sein. Neben der parallelisierten Auswertung mit einem Datenbanksystem kann die Nutzung von *MapReduce* als Datenzubringer Verbesserungen bringen. Ebenfalls ist die Performance der *SQL*-Anfragen auf verschiedenen DBMS, insbesondere auch spaltenorientierten, zu testen.
Entgegen des hier gewählten Ansatzes, könnten auch eigene Datenstrukturen mit ihr zugewiesenen Funktionen für *R* definiert werden, die sich nicht an der vorhandenen Struktur von *R* orientieren. Das ermöglicht es, eine für Datenbanken optimierte Datenstruktur zu generieren und diese von *R* aus nutzen zu können.

Im Gesamten ist mit den Betrachtungen und daraus gewonnenen Erkenntnissen dieser Arbeit ein erster Schritt auf dem Weg zur transparenten und parallelen, datenbankunterstützten Auswertung von *R* im Bereich der Intentions- und Aktitivätserkennung gelungen und somit eine Grundlage für fortführende Arbeiten gelegt worden.

A Literaturverzeichnis

[1] Ross Ihaka and Robert Gentleman. R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.

[2] Hadley Wickham. *Advanced R*. CRC Press, 2014.

[3] The R Core Team. R: A Language and Environment for Statistical Computing, 08 2015. (auf beiliegender CD).

[4] The R Core Team. R Language Definition, 2000. (auf beiliegender CD).

[5] Gunter Saake, Kai-Uwe Sattler, and Andreas Heuer. *Datenbanken: Implementierungstechniken*. mitp Verlags GmbH & Co. KG, 2011.

[6] RStudio Hadley Wickham, Romain Francois. Introduction to dplyr. https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html, August 2015. (auf beiliegender CD).

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] Michael Stonebraker, Daniel Abadi, David J DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[9] Jeffrey Dean and Sanjay Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[10] George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. Haskell boards the ferry. In Jurriaan Hage and MarcoT. Morazán, editors, *Implementation and Application of Functional Languages*, volume 6647 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2011.

[11] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[12] The PostgreSQL Global Development Group. PostgreSQL 9.5rc1 Documentation, 2015. (auf beiliegender CD).

[13] Joe Celko. Matrix Math in SQL. https://www.simple-talk.com/sql/t-sql-programming/matrix-math-in-sql/, September 2012. (auf beiliegender CD).

[14] Michael Jaedicke and Bernhard Mitschang. On Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.*, pages 379–389, 1998.

[15] Ambuj Shatdal and Jeffrey F. Naughton. Adaptive Parallel Aggregation Algorithms. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 104–114, 1995.

Abbildungsverzeichnis

Programmbeispielverzeichnis

B Programmbeispiele

```
[[1]]
[[1]]$object
[1] "CREATE TABLE tbl0 (rowid integer, colid integer, rownames varchar(50), colnames
    varchar(50), dataCol double precision, PRIMARY KEY (rowid,colid));INSERT INTO tbl0
    (rowid,colid,dataCol) VALUES (1,1,1);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES
    (2,1,4);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (3,1,5);INSERT INTO tbl0 (
    rowid,colid,dataCol) VALUES (4,1,2);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES
    (5,1,3);"

[[1]]$layer
[1] 1

[1] "SELECT * INTO kid FROM tbl0;DROP TABLE tbl0;"
[[1]]
[[1]]$object
[1] "CREATE TABLE tbl0 (rowid integer, colid integer, rownames varchar(50), colnames
    varchar(50), dataCol varchar(50), PRIMARY KEY (rowid,colid));INSERT INTO tbl0 (
    rowid,colid,dataCol) VALUES (1,1,Standard);INSERT INTO tbl0 (rowid,colid,dataCol)
    VALUES (2,1,Standard);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (3,1,Premium);
    INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (4,1,Standard);INSERT INTO tbl0 (
    rowid,colid,dataCol) VALUES (5,1,Premium);"

[[1]]$layer
[1] 1

[1] "SELECT * INTO krolle FROM tbl0;DROP TABLE tbl0;"
[[1]]
[[1]]$object
[1] "CREATE TABLE tbl0 (tblid integer, rownames varchar(50), id double precision, rolle
     varchar(50));INSERT INTO tbl0 (tblid) VALUES (1);INSERT INTO tbl0 (tblid) VALUES
    (2);INSERT INTO tbl0 (tblid) VALUES (3);INSERT INTO tbl0 (tblid) VALUES (4);INSERT
    INTO tbl0 (tblid) VALUES (5);UPDATE tbl0 SET id='1' WHERE tblid=1;UPDATE tbl0 SET
    id='4' WHERE tblid=2;UPDATE tbl0 SET id='5' WHERE tblid=3;UPDATE tbl0 SET id='2'
    WHERE tblid=4;UPDATE tbl0 SET id='3' WHERE tblid=5;UPDATE tbl0 SET rolle='Standard'
     WHERE tblid=1;UPDATE tbl0 SET rolle='Standard' WHERE tblid=2;UPDATE tbl0 SET rolle
    ='Premium' WHERE tblid=3;UPDATE tbl0 SET rolle='Standard' WHERE tblid=4;UPDATE tbl0
     SET rolle='Premium' WHERE tblid=5;"

[[1]]$layer
[1] 1

[1] "SELECT * INTO kunde FROM tbl0;DROP TABLE tbl0;"
[[1]]
[[1]]$object
```

```
27 [1] "CREATE TABLE tbl0 (rowid integer, colid integer, rownames varchar(50), colnames
       varchar(50), dataCol varchar(50), PRIMARY KEY (rowid,colid));INSERT INTO tbl0 (
       rowid,colid,dataCol) VALUES (1,1,Rennrad);INSERT INTO tbl0 (rowid,colid,dataCol)
       VALUES (2,1,Zeitfahrrad Stan);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (3,1,
       Mountainbike);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (4,1,Sattel);INSERT
       INTO tbl0 (rowid,colid,dataCol) VALUES (5,1,Luftpumpe);INSERT INTO tbl0 (rowid,
       colid,dataCol) VALUES (6,1,Zeitfahrrad Lux);"
28
29 [[1]]$layer
30 [1] 1
31
32 [1] "SELECT * INTO pname FROM tbl0;DROP TABLE tbl0;"
33 [[1]]
34 [[1]]$object
35 [1] "CREATE TABLE tbl0 (rowid integer, colid integer, rownames varchar(50), colnames
       varchar(50), dataCol double precision, PRIMARY KEY (rowid,colid));INSERT INTO tbl0
       (rowid,colid,dataCol) VALUES (1,1,1499);INSERT INTO tbl0 (rowid,colid,dataCol)
       VALUES (2,1,2700);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (3,1,699);INSERT
       INTO tbl0 (rowid,colid,dataCol) VALUES (4,1,89);INSERT INTO tbl0 (rowid,colid,
       dataCol) VALUES (5,1,25);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (6,1,3400);"
36
37 [[1]]$layer
38 [1] 1
39
40 [1] "SELECT * INTO ppreis FROM tbl0;DROP TABLE tbl0;"
41 [[1]]
42 [[1]]$object
43 [1] "CREATE TABLE tbl0 (rowid integer, colid integer, rownames varchar(50), colnames
       varchar(50), dataCol double precision, PRIMARY KEY (rowid,colid));INSERT INTO tbl0
       (rowid,colid,dataCol) VALUES (1,1,1);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES
       (2,1,2);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (3,1,3);INSERT INTO tbl0 (
       rowid,colid,dataCol) VALUES (4,1,4);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES
       (5,1,5);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (6,1,6);"
44
45 [[1]]$layer
46 [1] 1
47
48 [1] "SELECT * INTO pid FROM tbl0;DROP TABLE tbl0;"
49 [[1]]
50 [[1]]$object
51 [1] "CREATE TABLE tbl0 (tblid integer, rownames varchar(50), id double precision, name
       varchar(50) preis double precision,);INSERT INTO tbl0 (tblid) VALUES (1);INSERT
       INTO tbl0 (tblid) VALUES (2);INSERT INTO tbl0 (tblid) VALUES (3);INSERT INTO tbl0 (
       tblid) VALUES (4);INSERT INTO tbl0 (tblid) VALUES (5);INSERT INTO tbl0 (tblid)
       VALUES (6);UPDATE tbl0 SET id='1' WHERE tblid=1;UPDATE tbl0 SET id='2' WHERE tblid
       =2;UPDATE tbl0 SET id='3' WHERE tblid=3;UPDATE tbl0 SET id='4' WHERE tblid=4;UPDATE
        tbl0 SET id='5' WHERE tblid=5;UPDATE tbl0 SET id='6' WHERE tblid=6;UPDATE tbl0 SET
        name='Rennrad' WHERE tblid=1;UPDATE tbl0 SET name='Zeitfahrrad Stan' WHERE tblid
       =2;UPDATE tbl0 SET name='Mountainbike' WHERE tblid=3;UPDATE tbl0 SET name='Sattel'
       WHERE tblid=4;UPDATE tbl0 SET name='Luftpumpe' WHERE tblid=5;UPDATE tbl0 SET name='
       Zeitfahrrad Lux' WHERE tblid=6;UPDATE tbl0 SET preis='1499' WHERE tblid=1;UPDATE
       tbl0 SET preis='2700' WHERE tblid=2;UPDATE tbl0 SET preis='699' WHERE tblid=3;
       UPDATE tbl0 SET preis='89' WHERE tblid=4;UPDATE tbl0 SET preis='25' WHERE tblid=5;
       UPDATE tbl0 SET preis='3400' WHERE tblid=6;"
52
```

```
[[1]]$layer
[1] 1

[1] "SELECT * INTO produkt FROM tbl0;DROP TABLE tbl0;"
[[1]]
[[1]]$object
[1] "CREATE TABLE tbl0 (rowid integer, colid integer, rownames varchar(50), colnames
    varchar(50), dataCol double precision, PRIMARY KEY (rowid,colid));INSERT INTO tbl0
    (rowid,colid,dataCol) VALUES (1,1,1);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES
    (2,1,5);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (3,1,4);INSERT INTO tbl0 (
    rowid,colid,dataCol) VALUES (4,1,2);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES
    (5,1,5);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (6,1,3);"

[[1]]$layer
[1] 1

[1] "SELECT * INTO k_id FROM tbl0;DROP TABLE tbl0;"
[[1]]
[[1]]$object
[1] "CREATE TABLE tbl0 (rowid integer, colid integer, rownames varchar(50), colnames
    varchar(50), dataCol double precision, PRIMARY KEY (rowid,colid));INSERT INTO tbl0
    (rowid,colid,dataCol) VALUES (1,1,1);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES
    (2,1,2);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (3,1,3);INSERT INTO tbl0 (
    rowid,colid,dataCol) VALUES (4,1,4);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES
    (5,1,5);INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (6,1,6);"

[[1]]$layer
[1] 1

[1] "SELECT * INTO p_id FROM tbl0;DROP TABLE tbl0;"
[[1]]
[[1]]$object
[1] "CREATE TABLE tbl0 (tblid integer, rownames varchar(50), kunde double precision,
    produkt double precision,);INSERT INTO tbl0 (tblid) VALUES (1);INSERT INTO tbl0 (
    tblid) VALUES (2);INSERT INTO tbl0 (tblid) VALUES (3);INSERT INTO tbl0 (tblid)
    VALUES (4);INSERT INTO tbl0 (tblid) VALUES (5);INSERT INTO tbl0 (tblid) VALUES (6);
    UPDATE tbl0 SET kunde='1' WHERE tblid=1;UPDATE tbl0 SET kunde='5' WHERE tblid=2;
    UPDATE tbl0 SET kunde='4' WHERE tblid=3;UPDATE tbl0 SET kunde='2' WHERE tblid=4;
    UPDATE tbl0 SET kunde='5' WHERE tblid=5;UPDATE tbl0 SET kunde='3' WHERE tblid=6;
    UPDATE tbl0 SET produkt='1' WHERE tblid=1;UPDATE tbl0 SET produkt='2' WHERE tblid
    =2;UPDATE tbl0 SET produkt='3' WHERE tblid=3;UPDATE tbl0 SET produkt='4' WHERE
    tblid=4;UPDATE tbl0 SET produkt='5' WHERE tblid=5;UPDATE tbl0 SET produkt='6' WHERE
     tblid=6;"

[[1]]$layer
[1] 1

[1] "SELECT * INTO k_kauft_p FROM tbl0;DROP TABLE tbl0;"
[[1]]
[[1]]$object
[1] "SELECT * INTO tmp FROM kunde;ALTER TABLE tmp DROP COLUMN tblid;ALTER TABLE tmp
    DROP COLUMN rownames;SELECT * INTO tbl0 FROM k_kauft_p as a JOIN tmp as b ON a.
    kunde=b.id ORDER BY a.kunde;ALTER TABLE tbl0 DROP COLUMN id;DROP TABLE tmp;"

[[1]]$layer
```

```
[1] 1

[1] "SELECT * INTO kKp FROM tbl0;DROP TABLE tbl0;"
[[1]]
[[1]]$object
[1] "SELECT * INTO tmp FROM kKp;ALTER TABLE tmp DROP COLUMN tblid;ALTER TABLE tmp DROP
    COLUMN rownames;SELECT * INTO tbl0 FROM produkt as a JOIN tmp as b ON a.id=b.
    produkt;ALTER TABLE tbl0 DROP COLUMN produkt;DROP TABLE tmp;"

[[1]]$layer
[1] 1

[1] "SELECT * INTO bestellungen FROM tbl0;DROP TABLE tbl0;"
[[1]]
[[1]]$object
[1] "SELECT * INTO tbl0 FROM bestellungen WHERE kunde = 5;"

[[1]]$layer
[1] 1

[[2]]
[[2]]$object
kunde == 5

[[2]]$layer
[1] 2

[1] "SELECT * INTO kunde5_kauft2 FROM tbl0;DROP TABLE tbl0;"
[[1]]
[[1]]$object
[1] "SELECT sum(preis) INTO tbl1 FROM tbl0;"

[[1]]$layer
[1] 1

[[2]]
[[2]]$object
[1] "SELECT tblid,rownames,preis INTO tbl0 FROM kunde5_kauft2;"

[[2]]$layer
[1] 2

[1] "SELECT * INTO kunde5_summe FROM tbl1;DROP TABLE tbl1;"
```

Programmbeispiel B.1: Ausgabe des Beispielprogramms

```
[[1]]
[[1]]$object
[1] "SELECT ((rowid-1)%3)+1 as rowid,FLOOR((rowid-1)/3)+1 as colid, rownames, colnames,
     dataCol INTO tbl1 FROM tbl0;"

[[1]]$layer
[1] 1

[[2]]
[[2]]$object
[1] "CREATE TABLE tbl0 (rowid integer, colid integer, rownames varchar(50), colnames
    varchar(50), dataCol double precision, PRIMARY KEY (rowid,colid));INSERT INTO tbl0
    (rowid,colid,dataCol) VALUES (1,1,'1');INSERT INTO tbl0 (rowid,colid,dataCol)
    VALUES (2,1,'2');INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (3,1,'3');INSERT
    INTO tbl0 (rowid,colid,dataCol) VALUES (4,1,'4');INSERT INTO tbl0 (rowid,colid,
    dataCol) VALUES (5,1,'5');INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (6,1,'6');"

[[2]]$layer
[1] 2

[1] "SELECT * INTO m1 FROM tbl1;DROP TABLE tbl1;"
[[1]]
[[1]]$object
[1] "SELECT ((rowid-1)%2)+1 as rowid,FLOOR((rowid-1)/2)+1 as colid, rownames, colnames,
     dataCol INTO tbl1 FROM tbl0;"

[[1]]$layer
[1] 1

[[2]]
[[2]]$object
[1] "CREATE TABLE tbl0 (rowid integer, colid integer, rownames varchar(50), colnames
    varchar(50), dataCol double precision, PRIMARY KEY (rowid,colid));INSERT INTO tbl0
    (rowid,colid,dataCol) VALUES (1,1,'1');INSERT INTO tbl0 (rowid,colid,dataCol)
    VALUES (2,1,'2');INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (3,1,'3');INSERT
    INTO tbl0 (rowid,colid,dataCol) VALUES (4,1,'4');INSERT INTO tbl0 (rowid,colid,
    dataCol) VALUES (5,1,'5');INSERT INTO tbl0 (rowid,colid,dataCol) VALUES (6,1,'6');"

[[2]]$layer
[1] 2

[1] "SELECT * INTO m2 FROM tbl1;DROP TABLE tbl1;"
[[1]]
[[1]]$object
[1] "SELECT a.rowid as rowid, b.colid as colid, a.rownames as rownames, b.colnames as
    colnames, sum(a.dataCol * b.dataCol) as dataCol INTO tbl0 FROM m1 as a, m2 as b
    WHERE a.colid=b.rowid GROUP BY a.rowid,b.colid,a.rownames,b.colnames;"

[[1]]$layer
[1] 1

[1] "SELECT * INTO mM FROM tbl0;DROP TABLE tbl0;"
[[1]]
[[1]]$object
```

```
41 [1] "SELECT a.colid as rowid, a.rowid as colid, a.colnames as rownames, a.rownames as
       colnames, a.dataCol INSERT INTO tbl0 FROM m1 as a;"
42
43 [[1]]$layer
44 [1] 1
45
46 [1] "SELECT * INTO mT FROM tbl0;DROP TABLE tbl0;"
```

Programmbeispiel B.2: Ausgabe des Beispielprogramms Matrix

Selbstständigkeitserklärung

Hiermit erkläre ich, Dennis Weu, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

....

Ort, Datum Unterschrift