THM
TECHNISCHE HOCHSCHULE MITTELHESSEN

Projektbericht

Projektphase bei INFORM GmbH - GB 70

Oktober 2022 - Dezember 2022

Madelaine Hestermann
madelaine.hestermann@mni.thm.de

Betrieb:
INFORM GmbH
Betreuer:
Dimitri Bohlender, M.Sc.
Dozent:
Prof. Dr. Michael Elberfeld

7. Februar 2023
Technische Hochschule Mittelhessen, Gießen

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Bereits in den 80-er und den frühen 90-er Jahren entstand im Bereich der Datenbankmanagement-Systeme großes Interesse an logisch deklarativen Programmierkonzepten wie etwa der Programmiersprache Datalog. Da diese Konzepte aber ersetzbar waren und nicht dringend umgesetzt werden mussten, versiegte das Forschungsinteresse daran bereits Ende der 90er Jahre, sodass die Entwicklung einiger Kernprojekte in diesem Zeitraum eingestellt wurde. In den letzten Jahren rückten diese Konzepte aber wieder vermehrt in den Vordergrund und finden Anwendung in vielen unterschiedlichen Applikationen, die sich beispielsweise mit Informationsgewinnung oder Optimierungen auseinandersetzen[1, S. 106].

Der Mehrwert, welchen deklaratives Programmieren bietet, liegt in der Abstrahierung von unwesentlichen Details. Hierbei wird deskriptiv die Aufgabe einer Anwendung beschrieben [1, S. 106]. Anders als bei imperativen Programmiersprachen, bei welchen detailliert ausgeführt werden muss, wie ein Programmablauf zu einem gewünschten Ergebnis führen kann, müssen Details wie Variablendeklarationen und Ähnliches in deklarativen Programmiersprachen nicht beachtet werden[2, S. 13].

Dies ist besonders nützlich bei Beweisführungen, die allgemeingültige Abstraktionen fordern sowie bei rapide wachsenden und komplexen Systemen, die eine hohe Anzahl an Daten organisieren und es der Programmierer*innen erlauben, sich auf die wesentliche Geschäftslogik zu konzentrieren. Außerdem erleichtert die hohe Abstraktionsebene Code Analysen, sodass Optimierungsmöglichkeiten und Sicherheitsmaßnahmen einfacher entwickelt und in Applikationen integriert werden können. Zuletzt ist es ebenfalls üblich insbesondere die Programmiersprache Datalog zur Abstraktion relationaler Strukturen und Datenabfragen zu verwenden [1, S. 107].

Aus diesen Vorteilen heraus ergab sich für den Projektpartner dieser Bachelorarbeit, den Geschäftsbereich 70 der INFORM GmbH, die Motivation für ihre Anwendung *WorkforcePlus* die deklarativ logische Sprache *Roxx* zu entwickeln, die ein Dialekt der Programmiersprache Datalog ist und dieser semantisch sehr ähnelt. Der Geschäftsbereich entwickelt die Software *WorkforcePlus* im Bereich des Workforce Managements. Workforce Management ist ein Unternehmensbereich des Personalwesens und bedeutet präzise ausgedrückt die Einteilung des richtigen Personals, mit der richtigen Qualifikation, zum richtigen Zeitpunkt, am richtigen Ort[3]. *WorkforcePlus* selbst ist also eine Applikation zur optimierten und automatisierten Personaleinsatzplanung. Sie organisiert und verarbeitet eine große Menge an Daten, weswegen der Einsatz einer Sprache mit hohem Abstraktionspotential, wie etwa Roxx, geeignet ist.

1.2 Problemstellung

Das Projekt, welches als Grundlage für diese Bachelorarbeit dient, beschäftigt sich mit der Erweiterung eines Sprachkonstruktes der Sprache *Roxx*, dem sogenannten *Partitionsprädikat*. Mithilfe des Partitionsprädikats ist es möglich eine Sortierfunktion auf eine Menge an Daten

auszuführen. Als Ausgangsbasis diente das Partitionsprädikat, welches allerdings nur nach einem einzigen gegebenen Attribut sortieren konnte und nur über eine bestimmte Sortierfunktion verfügte. Wird eine Sortierung auf Daten angewandt, welche für das entsprechende Sortierattribut dieselben Werte aufweisen, hat das, aufgrund der Funktionsweise dieser Sortierfunktion, die Konsequenz, dass die Tupel denselben Rang erhalten und die Sortierung nicht eindeutig ist[4].

Außerdem gibt es darüber hinaus einige Anwendungsfälle, bei welchem systematisch nach mehreren Attributen sortiert werden soll, um beispielsweise eine geordnete und übersichtliche Darstellungsart der Applikationsdaten zu gewährleisten. In Abbildung 1, welche einen Teil der Benutzeroberfläche der Applikation *WorkforcePlus Employee Portal* zeigt, werden Mitarbeiteranfragen dargestellt. Diese Anfragen, bei welchen Mitarbeiter beispielsweise Schichtwechsel mit Kolleginnen und Kollegen und Ähnliches anfragen können, bestehen stets aus einem Status (in der Abbildung dargestellt durch Icons), einer Dauer seit der letzten Statusänderung und einigen weiteren Informationen. Realisiert werden diese Anfragen durch das Partitionsprädikat, bei welchem zunächst nach dem Status und anschließend nach der Dauer seit der letzten Statusänderung sortiert werden muss, um die übersichtliche Darstellungsart, wie sie in Abbildung 1 zu sehen ist, zu erreichen.

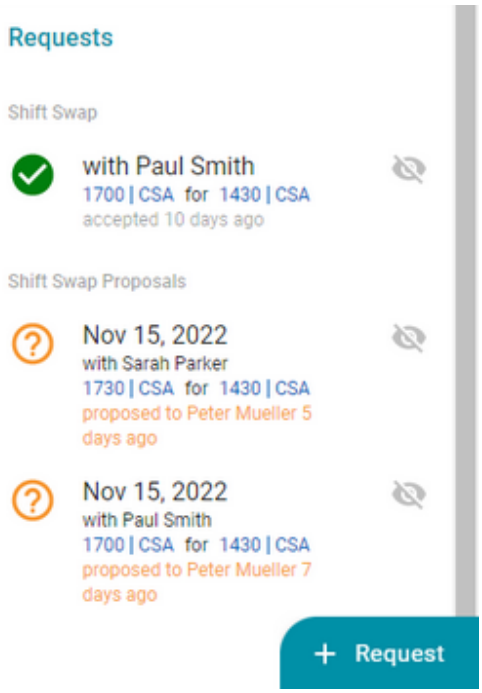

Abbildung 1: Mitarbeiter Requests der Applikation WorkforcePlus Employee Portal[5]

Jegliche Anwendungsfälle, welche Mehrfachsortierungen nutzen, sei es um Sortierungen eindeutig zu gestalten, oder weil ein Fall es konkret erfordert, konnten bereits vor Beginn des Projekts umgesetzt werden, allerdings nicht ohne die *Roxx* Konstrukte mit Programmcode der imperativen Programmiersprache *Groovy* zu ergänzen. Mithilfe sogenannter Makros war es möglich die Mehrfachsortierung aus einer Kombination mehrerer *Roxx* Konstrukte und imperativen Sprachkonstrukten erfolgreich zu simulieren, was allerdings unverhältnismäßig komplex sowie ineffizient war.

1.3 Ziele

Im Rahmen dieser Bachelorarbeit sollen das Partitionsprädikat im Kontext der logischen Programmiersprache Datalog wissenschaftlich aufgearbeitet und eine praktische Implementierung des Partitionsprädikats im Rahmen einer domänenspezifischen Sprache präsentiert werden.

1.4 Aufbau der Arbeit

In diesem Abschnitt soll die Struktur dieser Arbeit dargestellt werden.

Zuerst wird der theoretische Hintergrund zur Programmiersprache Datalog im entsprechenden Kapitel skizziert. Dabei werden zunächst die deduktiven Datenbanken und die wichtigsten Details zu Prädikaten entsprechend der mathematischen Prädikatenlogik in einem Abschnitt erläutert. Anschließend wird die Syntax von Datalog und Terminologien vorgestellt. Daraufhin wird erklärt wie die Berechnung rekursiver Datalog Regeln realisiert werden kann. Zuletzt werden, jeweils in eigenen Abschnitten, zwei spezielle Formen von Regeln vorgestellt, die Negation und die Aggregation. Hierzu wird erläutert, welche Probleme bei der Berechnung dieser Regeln auftreten können und wie diese Probleme zu lösen sind.

Im Kapitel des Partitionsprädikats wird dieses zunächst im Datalog Kontext vorgestellt. Hierbei wird eine Datalog Syntax für das Prädikat eingeführt und seine Semantik erläutert. Analog zu den Abschnitten der Negation und der Aggregation im Kapitel Datalog wird auch hier skizziert welche Probleme bei der Berechnung des Partitionsprädikats zu erwarten sind und welche Lösungsansätze es hierfür gibt. Anschließend wird das Partitionsprädikat im Kontext der domänenspezifischen Sprache Roxx beschrieben. Auch hier werden zuerst Syntax und daraufhin die Semantik erklärt.

Das Kapitel domänenspezifische Sprachen im Java Umfeld gibt einen Überblick darüber, was domänenspezifische Sprachen überhaupt sind, welche Eigenschaften sie haben, wieso es sich lohnt eine solche Sprache zu verwenden und welche Probleme mit ihr kommen können. Darüber hinaus wird die Java-verwandte Programmiersprache Groovy vorgestellt und erläutert wieso Groovy besonders geeignet ist um eine domänenspezifische Sprache zu realisieren.

Im Anschluss wird die konkrete Implementierung des Partitionsprädikats dargelegt. Dazu werden das Backend, in welchem die Sprache Roxx implementiert ist, skizziert und anschließend werden für die Implementierung nötige Voraussetzungen und Designentscheidungen, sowie die tatsächliche Umsetzung vorgestellt.

Abgeschlossen wird diese Bachelorarbeit mit einem Fazit, sowie einem Ausblick.

2 Datalog

Dieses Kapitel behandelt alle wesentlichen Grundlagen, die es bedarf um einerseits Problemstellung, Vorgehensweise und Lösung des Projekts nachvollziehen zu können und andererseits soll der theoretische Hintergrund zu einer fundierten Faktenlage beitragen, um schließlich die Diskussion mit der notwendigen inhaltlichen Tiefe und Korrektheit führen zu können. Sämtliche Beispiele, welche in diesem Kapitel genannt werden, sind angelehnt an die Beispiele des Papers *Datalog and Recursive Query Processing*[1].

2.1 Deduktive Datenbanken und Prädikate

Deduktive Datenbanken bilden die Grundlage von Datalog. Sie sind dazu in der Lage relationale Datenbanken durch regelbasierte Ausdrücke zu erweitern, sodass es möglich ist durch einige solcher Regeln eine große Faktenmenge abzuleiten und darzustellen[6, S. 1].

Datenbanksysteme profitieren von dieser Erweiterung, da das logische Ableiten von Fakten, die es erlauben abgeleitet zu werden, es erübrigt sämtliche Daten deskriptiv im jeweiligen System festzuhalten. Die Grundlage deduktiver Datenbanken bildet die Prädikatenlogik, da im logischen Datenmodell, sowohl zu verwaltende Daten als auch Operationen durch logische Formeln beschrieben werden[6, S. 17].

Prädikate sind im Kontext der Logik Gebilde, die auf Elemente aus einer Menge anhand von Regeln einen Wahrheitswert abbilden[7, S. 32]. Letztlich sind Prädikate also Funktionen mit einer booleschen Zielmenge.

Datalog verfügt über zwei unterschiedlich zu betrachtende Arten von Prädikaten: extensionale und intensionale Prädikate. *Extensionale Prädikate* entsprechen den Inhalten der Tabellen der Quelldatenbank (Beispielsweise eine relationale Datenbank, die als Grundlage verwendet wird). *Intensionale Prädikate* hingegen entsprechen hergeleiteten Daten, die sich aus errechneten Regeln ergeben[1, S. 114].

2.2 Syntax und Terminologie

Um ein Grundverständnis für die Sprache Datalog zu vermitteln, werden in diesem Abschnitt die Datalog Syntax vorgestellt und die wichtigsten Begriffe definiert und erläutert.

Ein Programm in der Programmiersprache Datalog besteht aus einer Ansammlung von Regeln, die der folgenden Syntax entsprechen:

$$A \text{ :- } B_1, B_2, ..., B_n$$

wobei n nicht negativ ist, A den *Kopf* und $B_1, B_2, ..., B_n$ den *Körper* der Regel darstellen. Semantisch stellt die hier aufgeführte Regel eine Konjunktion der Argumente B_i dar, die A implizieren[1, S. 113].

Ein *Term* ist in Datalog entweder eine Konstante oder eine Variable. Ein *Atom* ist ein Prädikat mit einer Liste an Argumenten, die aus Termen bestehen. In der oben aufgerührte Regel beispielsweise entsprechen sowohl die Bezeichner $B_1, B_2, ..., B_n$, als auch A solchen Atomen. Des weiteren werden Atome, welche nur aus konstanten Argumenten bestehen, auch als *Grundatome* bezeichnet. Eine *Datenbankinstanz* ist eine Menge von Grundatomen, wobei die *Quelldatenbankinstanz* die Instanz bezeichnet, welche ausschließlich Extensionale Prädikate als Atome beinhaltet. Die *Grundinstanz* ist eine solche Instanz, bei welcher alle Variablen einer Regel durch Konstanten ersetzt werden. Die Menge aller in der Datenbank vorkommenden Konstanten wird als *Aktive Domäne* einer Datenbankinstanz bezeichnet[1, S. 114].

Anhand der folgenden Datalog Regeln sollen konkrete Beispiele für die eingeführte Terminologie gegeben werden.

```
ancestor(X,Y) :- parent(X,Y)

ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y)
```

Sowohl `ancestor` als auch `parent` sind Atome, wobei `ancestor` ein Intensionales Prädikat und `parent` ein Extensionales Prädikat darstellt. Beide Atome besitzen mit den Termen `X`, `Y`, und `Z` in unterschiedlicher Kombination jeweils zwei Argumente. Die Quelldatenbankinstanz für dieses Beispiel ist eine Tabelle mit Elementen aus der Menge der Aktiven Domäne, die in Tupeln zusammengefasst werden. Sie könnte entsprechend der zu betrachtenden Regeln beispielsweise folgendermaßen aussehen:

Seien die Elemente der Aktiven Domäne die Personen {*Rose*, *Hugo*, *Ronald*, *Arthur*, *Septimus*}, ergibt sich daraus die folgende Beispieltabelle2.4:

Tabelle 1: *parent* (initiale Tupel)

X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus

Dabei stellt die Variable `Y` stets den Elternteil der Variable `X` dar.

Die Semantik der Regeln entspricht ihrer Bezeichnung. Sie sind in der Lage rekursiv den Vorfahren einer gegebenen Person zu berechnen, denn die Regel `ancestor` gilt dann als logisch wahr, wenn der Wert der Variable `Y` ein Vorfahre der Variable `X` ist. Dies geschieht indem die erste Regel zunächst über das Prädikat `parent` definiert wird, wodurch die Werte, welche sich für dieses Prädikat ergeben ebenfalls für das Prädikat `ancestor` gelten. In der zweiten

Regel wird der natürliche Verbund der Prädikate `ancestor` und `parent` über die Variable `Z` angewandt. Hierzu werden also neue Tupel aus den Variablen `X` und `Y` gebildet, bei welchen das zweite Argument des Prädikats `parent` und das erste Argument des Prädikats `ancestor` übereinstimmen.

Die aus den Regeln hergeleitete Ergebnistabelle würde auf Grundlage der Quelldatenbankinstanz die Folgende sein2:

Tabelle 2: *vorfahre*

X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus
Rose	Arthur
Hugo	Arthur
Ronald	Septimus
Rose	Septimus
Hugo	Septimus

Um die sichere Berechnung einer Datalog Regel zu garantieren, gibt es außerdem Beschränkungen, was die Definition einer Regel betrifft. Folgende Bedingung wird hierzu festgelegt:

Bedingung 2.2.1 *Jede Variable, welche im Kopf einer Regel auftaucht, muss auch selbst, oder als äquivalente Variable im Körper der Regel erscheinen[1, S. 115].*

Zuletzt ist es ebenfalls möglich, dass eine Regel einen leeren Körper hat, der in diesem Fall bedingungslos als wahr erachtet wird. Voraussetzung hierfür ist allerdings, dass die im Kopf auftretenden Terme allesamt Konstanten sind, um der Bedingung 2.2.1 gerecht zu werden. Eine solche Regel wird dann als *Fakt* bezeichnet[1, S. 115].

2.3 Berechnung rekursiver Regeln

In diesem Abschnitt wird erläutert, wie Datalog Regeln berechnet und ausgewertet werden. Die Auswertung einfacher Regeln geschieht offensichtlich schlicht durch Anwendung logischer Operationen, wie beispielsweise dem natürlichen Verbund, Konjunktionen, Disjunktionen und Ähnlichen. Beinhaltet der Körper einer Regel ein einziges Atom ohne Verwendung eines Operators, entspricht das Prädikat, welchem die Regel zugewiesen ist dem Atom im Körper. Und zuletzt ist, wie bereits im vorigen Abschnitt erläutert wurde, eine Regel mit leerem Körper ein Fakt, der immer als wahr gilt. Fraglich ist allerdings wie rekursive Regeln ausgewertet werden können. Zur Veranschaulichung wie etwa aus der initialen Tabelle 2.4 mithilfe der rekursiven Regel `ancestor` die Ergebnistabelle 2 entstehen kann, wird im Folgenden wieder das Beispiel verwendet, auf welcher beide Tabellen und die Regel basieren.

Die Grundlage für die Berechnung der Datalog Regeln bildet die *Kleinster-Fixpunkt-Semantik*[1, S. 117]. Sie basiert auf dem *Unmittelbaren Folgeoperator*[S. 103][6] T_P, wobei P ein Datalog Programm sei, welches die entsprechende Datenbankinstanz für den Operator spezifiziert. Sei I eine Datenbankinstanz, die dem Beispiel aus dem vorigen Abschnitt gleicht, so ergeben sich für sie die folgenden Tabellen3:

Tabelle 3: Initialbelegung Datenbankinstanz I

$I =$

parent	
X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus

,

ancestor	
X	Y

Wird der Unmittelbare Folgeoperator T_P einmal auf die Datenbankinstanz angewandt ergibt sich folgende Belegung für die Tabellen:

Tabelle 4: Datenbankinstanz $T_P(I)$

$T_P(I) =$

parent	
X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus

,

ancestor	
X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus

Zu erkennen ist, dass in der Tabelle `ancestor` dieselben Werte erscheinen, wie in der Tabelle `parent`. Dies geschieht durch Anwendung der ersten Regel, die der Operator T_P, angewandt auf I, auslöst. Diese erste Regel bestimmt das Elternteil `Y` der Variable `X` und dupliziert daher die Tabelle `parent`. Weitere Ergebnisse liefert der erste Schritt der Berechnung nicht, da der Unmittelbare Folgeoperator immer nur unmittelbare Ergebnisse errechnet. Das heißt jede Regel wird einmal durchlaufen. Auch rekursive Regeln werden je Schritt nur ein einziges mal angewandt. Die erste im Beispiel definierte Regel

```
ancestor(X,Y) :- parent(X,Y)
```

erzeugt damit das erkennbare Ergebnis des ersten Durchgangs. Auch die zweite Regel

```
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y)
```

wird angewandt. Sie hat allerdings noch keine Auswirkungen auf die Ergebnistabelle, da der natürliche Verbund der Prädikate `parent` und `ancestor` über `Z` auf Grundlage der Tabelle

`ancestor` zu Beginn des ersten Durchgangs durchgeführt wurde, als diese noch leer war und somit kein Tupel als Ergebnis liefert.

Die errechneten Tupel, wie sie in der Tabelle 8 dargestellt sind, ergeben nun den ersten Fixpunkt für T_P. In der Regel gibt es eine Menge an Fixpunkten für den Unmittelbaren Folgeoperator. Ziel der Kleinster-Fixpunkt-Semantik ist es aber eben den kleinsten dieser Fixpunkte zu errechnen. Der kleinste Fixpunkt für $T_P(I)$ ist eine Teilmenge jedes anderen Fixpunktes für $T_P(I)$, nämlich ein solcher der über keine weiteren hinzukommenden Tupel verfügt. Dieser Fixpunkt lässt sich konstruktiv dadurch errechnen, dass der Unmittelbare Folgeoperator wiederholt auf die Ergebnistupel des vorherigen Durchgangs angewandt wird bis der kleinste Fixpunkt erreicht ist [1, S. 118].

Führt man die Ergebnistabelle 8 mit dieser Rechenart fort, so lassen sich folgende Fixpunkte errechnen:

Tabelle 5: Datenbankinstanz $T_P(I)$

$T_P(I) =$

parent	
X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus

,

ancestor	
X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus

Tabelle 6: Datenbankinstanz $T_P^2(I)$

$T_P^2(I) =$

parent	
X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus

,

ancestor	
X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus
Rose	Arthur
Hugo	Arthur
Ronald	Septimus

Tabelle 7: Datenbankinstanz $T_P^3(I)$

$T_P^3(I) =$

parent	
X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus

,

ancestor	
X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus
Rose	Arthur
Hugo	Arthur
Ronald	Septimus
Rose	Septimus
Hugo	Septimus

Tabelle 8: Datenbankinstanz $T_P^4(I)$

$T_P^4(I) =$

parent	
X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus

,

ancestor	
X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus
Rose	Arthur
Hugo	Arthur
Ronald	Septimus
Rose	Septimus
Hugo	Septimus

Zu beachten ist hier, dass sich die Tabellen von $T_P^3(I)$ und $T_P^4(I)$ gleichen. Bei Anwendung des Unmittelbaren Folgeoperators auf $T_P^3(I)$ kommt also kein weiteres Tupel mehr hinzu. Der kleinste Fixpunkt wurde dementsprechend nach drei Iterationen gefunden. Außerdem ist zu beachten, dass die Ergebnistabelle `ancestor` der zu erwartenden Tabelle 2 entspricht. Mithilfe der Kleinster-Fixpunkt-Semantik konnte also die korrekte Belegung an Tupeln berechnet werden.

2.4 Negation

Die Negation ist eine Operation, welches grundlegendes Datalog um großen praktischen Nutzen erweitert. Dennoch birgt die Negation Probleme, die eine einfache intuitive Nutzung ausschließen. In diesem Abschnitt wird daher erläutert welche Problematik mit der Erweiterung um die Negation einhergeht und wie diese gelöst werden kann.

Die Erweiterung von Datalog um Negationen soll ausschließlich die Verwendung der Negation im Körper einer Regel umfassen. Die Negierung des Kopfes wird ausgeschlossen[S. 120][1].

Grundsätzlich lassen sich Datalog Regeln, welche Negationen in ihren Körpern beinhalten ebenso wie rekursive Regeln mithilfe der Kleinster-Fixpunkt-Semantik auswerten. Das heißt auf sie kann der Unmittelbare Folgeoperator ebenso angewendet werden, wie auf alle gewöhnlichen Regeln ohne Negation auch. Allerdings birgt die Negation zwei Probleme, welche im Folgenden anhand zweier Beispiele erläutert werden sollen.

Das erste Beispielprogramm zeigt zwei Regeln `p(x)` und `q(x)`, welche einander in einer Rekursion negiert bedingen, allerdings keine weiteren Informationen bereithält, sodass nicht festgelegt ist, woraus `p(x)` und `q(x)` eigentlich bestehen. Bei einem naiven Versuch diese Regeln anhand der Kleinsten-Fixpunkt-Semantik auszuwerten, wird schnell klar, dass dies ohne weitere Spezifizierung der Wertemenge zu keinem sinnvollen Ergebnis führen kann.

```
p(x) :- not q(x)
```

```
q(x) :- not p(x)
```

Das zweite Beispielprogramm zeigt eine einzige Regel anhand welcher das Problem des Alternierens deutlich wird. Die Regel `q(x)` ist bedingt durch ein Prädikat `e(x)`, welches in diesem Beispiel ein Extensionales Prädikat sein soll. Außerdem hängt `q(x)` aber zusätzlich von sich selbst negiert ab:

```
q(x) :- e(x) not q(x)
```

In diesem Beispiel ist das vorige Problem gelöst. Das Extensionale Prädikat `e(x)` liefert eine eindeutige Wertemenge, die als Grundlage für die Negation des Prädikats `q(x)` fungiert. Das Problem, welches sich hierbei ergibt, wird allerdings bei dem Versuch deutlich die Regel mithilfe der Kleinster-Fixpunkt-Semantik auszuwerten. Hierfür soll angenommen werden, dass das Exentsionale Prädikat `e(x)` drei numerische Werte $1, 2, 3$ enthält. Wird der Unmittelbare Folgeoperator mit dieser Belegung als Grundlage auf die Regel angewandt, so ergibt sich für die erstmalige Anwendung folgende Ergebnistabelle:

q(x)
1
2
3

Die Werte dieser Tabelle entstehen aus der Abhängigkeit zu dem Prädikat `e(x)`, sodass die Werte von `e(x)` in `q(x)` kopiert werden. Der zweite Teil der Regel `not q(x)` hat in diesem Schritt noch keine sichtbare Auswirkung, da `q(x)` vor Anwendung des Unmittelbaren Folgeoperators die leere Menge war und das Komplement der leeren Menge also alle beliebigen Werte außer der leeren Menge selbst ist, sodass von den Elementen aus `e(x)` nichts abgezogen wird.

Nach einer weiteren Anwendung des Unmittelbaren Folgeoperators kommt das Prädikat `q(x)` hingegen zum Tragen. Die Werte des Extensionalen Prädikats `e(x)` werden wiederum auf die Regel `q(x)` abgebildet. Diesmal hält `q(x)` aber bereits aus dem vorigen Schritt einige Werte, nämlich dieselben, welche wieder durch `e(x)` auf `q(x)` abgebildet werden sollten: $1, 2, 3$. Die Regel ist nun so zu verstehen, dass alle Werte aus `e(x)` aber nicht die aus `q(x)` als Ergebnis für diesen Schritt errechnet werden. Da `e(x)` und `q(x)` aber exakt dieselben Werte halten, ergibt sich hieraus dann die leere Menge, welche als Ergebnis für `q(x)` in diesem Schritt gilt.

Nun ist es möglich den Unmittelbaren Folgeoperator beliebig oft auf die Regel `q(x)` anzuwenden. Bereits nach zwei Anwendungen ist aber festzustellen, dass das Ergebnis nach dem zweiten Schritt dasselbe ist wie vor der erstmaligen Anwendung, nämlich die leere Menge. Das Ergebnis nach dem dritten Schritt entspricht dann offensichtlich dem Ergebnis nach dem ersten Schritt und es wird deutlich, dass das Ergebnis der Regel bei Ausführung nach jedem geraden Schritt die leere Menge und nach jedem ungeraden Schritt die Menge $1, 2, 3$ ist und abwechselnd zwischen diesen beiden Ergebnissen alterniert.

Nach Definition der Kleinster-Fixpunkt-Semantik, wie sie in Abschnitt 2.3 eingeführt wurde, ist es allerdings notwendig für die Berechnung rekursiver Regeln den Fixpunkt zu finden, der eine Teilmenge aller zu findenden Fixpunkte ist, was in dem erläuterten Beispiel offensichtlich nicht möglich ist, da die beiden Fixpunkte, welche das Programm erzeugt, jeweils keine Teilmenge des jeweils anderen Fixpunktes sind. Somit kann hier kein sinnvolles Endergebnis gefunden werden.

Um die beschriebenen Problematiken zu lösen gibt es zwei aufeinander aufbauende Konzepte, welche einige Regeln einführen, die eine sinnvolle Ausführung rekursiv definierter Negation in Datalog Programm garantieren. Das Konzept des *Semipositiven Datalog* fordert Folgendes:

Bedingung 2.4.1 *Negiert werden dürfen nur Extensionale Prädikate. Intensionale Prädikate sind von der Negation ausgeschlossen[1, S. 122].*

Bedingung 2.4.2 *Jede Variable im Körper einer Regel muss in mindestens einem nicht negierten Atom auftauchen[1, S. 122].*

Bedingung 2.4.1 verhindert ungewollte Effekte wie das Alternieren, welches im zweiten Beispiel demonstriert wurde. Das Alternieren kommt zustande, da das Intensionale Prädikat `q(x)` veränderbar ist und somit die Möglichkeit besitzt zwischen den beiden Ergebnismöglichkeiten wechselt. Die Einschränkung Negation, also das Wegnehmen von Elementen nur für Extensionale, also unveränderliche Prädikate zuzulassen, verhindert alternierendes Verhalten grundsätzlich. Bedingung 2.4.2 löst das Problem, welches mit dem ersten Beispiel skizziert wurde. Sie gewährleistet dass es im Falle einer Negation immer eine eindeutig definierte Grundmenge gibt, aus welcher Tupel durch die Negation entfernt werden. Dies garantiert dann, dass Datalog Programme terminieren und ihre Ergebnisse also endlich sind, da die Grundmenge selbst auch endlich ist. Außerdem garantiert es , dass die Ergebnisse ausschließlich auf dem tatsächlichen Inhalt der Datenbank beruhen, da die Grundmenge stets direkt auf Extensionalen Prädikaten oder indirekt auf Intensionalen Prädikaten beruht, die ihrerseits aber wieder von Extensionalen Prädikaten abgeleitet werden[1, S. 122].

Durch die Semantik des Negationsoperators wird, mit Rücksicht auf die oben genannten Bedingungen, schlicht das Komplement eines Prädikats erzeugt[S. 122][1]. Syntaktisch wollen wir für die Negation in Regeln das Wort *not* einführen.

Um dies am Eingangsbeispiel, welches Vorfahren von Personen errechnet, zu erläutern, sollen die Regeln daraus hier erweitert werden. Wir führen eine weitere Regeln zu den bereits bestehenden ein, sodass wir nun folgendes Datalog Programm erhalten:

```
ancestor(X,Y) :- parent(X,Y)

ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y)

grandparent(X,Y) :- ancestor(X,Y) not parent(X,Y)
```

Mit der neu eingeführten Regel ist das Programm in der Lage nur ein Großelternteil einer gegebenen Person zu berechnen, also eine verwandte Person, welche allerdings kein Elternteil ist. Dies geschieht über den natürlichen Verbund der Tupel, welche sich aus dem Prädikat `ancestor` und aus dem Komplement des Prädikats `parent`, also all jenen Tupeln, welche gerade nicht in dem Prädikat enthalten sind, ergeben.

Das Programm entspricht dem Konzept des Semipositiven Datalogs, da es Negation anwendet und es allen Bedingungen, welche das Semipositive Datalog definiert, erfüllt. Bedingung 2.4.1 wird dadurch erfüllt, dass das Prädikat `parent` dasjenige ist, welches negiert wird, da es sich bei diesem Prädikat entsprechend der Bedingung um ein Extensionales Prädikat handelt. Bedingung 2.4.2 wird erfüllt, da beide Variablen in dem negierten Prädikat `X` und `Y` in dem nicht negierten Prädikat `ancestor` ebenfalls vorkommen.

Zusätzlich zum Semipositiven Datalog gibt es ein Konzept, welches es erlaubt auch mit der Negation von Intensionalen Prädikaten umzugehen: die *Stratifizierte Negation*. Die Idee hinter der Stratifizierten Negation ist es ein Intensionales Prädikat für den Augenblick der Auswertung der Negation als ein Extensionales Prädikat zu betrachten, um damit die Bedingung 2.4.1 des Semipositiven Datalogs zu erfüllen[1, S. 123].

Um die Stratifizierte Negation realisieren zu können, sei davon auszugehen, dass ein Datalog Programm als eine Sequenz einzelner Programmabschnitte geschrieben werden kann. Diese Sequenz Partitionierung wird als *Stratifikation* und die einzelnen Sequenzabschnitte als *Strata* bezeichnet. Nach der Berechnung eines jeden Stratums werden dessen Intensionalen Prädikate für das nächste Stratum in Extensionale Prädikate umgewandelt[1, S. 123]. Des weiteren gelten zwei Bedingungen für die Stratifikation:

Bedingung 2.4.3 *Gibt es eine Regel A :- ..., B, ... in einem Datalog Programm P, welche in Stratum P_i ist, während B in Stratum P_j ist, dann ist $i \geq j$[1, S. 123].*

Bedingung 2.4.4 *Gibt es eine Regel A :- ..., not B, ... in einem Datalog Programm P, welche in Stratum P_i ist, während B in Stratum P_j ist, dann ist $i > j$[1, S. 123].*

Bedingung 2.4.3 ermöglicht es Regeln, welche keine Negation enthalten zunächst an beliebiger Stelle auszuwerten, wohingegen Bedingung 2.4.4 garantiert, dass Regeln, die Negation bein-

halten in jedem Fall nach allen Regeln ausgewertet wird, welche wiederum die negierte Regel beeinflussen.

Ein Datalog Programm, welches Stratifikation erlaubt wird als *stratifizierbar* bezeichnet.

Auch die Stratifizierte Negation soll anhand eines Beispiels illustriert werden. Hierzu erweitern wir das ursprüngliche Beispiel zur Berechnung von Vorfahren um drei weitere Regeln und erhalten folgendes Datalog Programm:

```
ancestor(X,Y) :- parent(X,Z)

ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y)

person(X) :- parent(X,Y)

person(Y) :- parent(X,Y)

notRelated(X,Y) :- person(X), person(Y), not ancestor(X,Y)
```

Dieses Programm kann zwei Personen errechnen zwischen welchen kein Verwandtschaftsverhältnis besteht. Hierbei handelt es sich außerdem um ein stratifizierbares Programm, was im späteren Teil diesen Abschnitts näher erläutert wird.

Die Vorgehensweise bei der Berechnung Stratifizierter Negation sieht vor, dass zunächst eine Stratifikation des gegebenen Programms berechnet wird, um anschließend die Strata nacheinander entsprechend des Konzepts des Semipositiven Datalog auszuwerten, wobei alle Intensionalen Prädikate eines vorangehenden Stratums als Extensionale Prädikate des darauffolgenden Stratums erachtet werden[1, S. 124].

Sowohl zur Berechnung einer Stratifikation als auch zur Feststellung, ob ein Programm stratifizierbar ist, dient ein endlicher *Präzedenzgraph*. Die Knoten dieses Präzedenzgraphen bestehen aus den Intensionalen Prädikaten des Programms. Die Kanten des Graphs setzen sich aus den Regeln des Programms zusammen. Eine Regel ohne Negation, wie etwa

$$A \text{ :- } ...B...$$

entspricht der Kante (B, A). Eine Regel, welche Negation beinhaltet, beispielsweise

$$A \text{ :- } not\ ...B...$$

entspricht der Kante (B, A) mit einem Label $\neg$, welches die Negation symbolisiert[1, S. 124].

Um mithilfe des Präzedenzgraphen herauszufinden ob ein Programm stratifizierbar ist, oder nicht, muss geprüft werden ob der Graph Zyklen beinhaltet. Falls es keine Zyklen gibt, ist das

Programm in jedem Fall stratifizierbar. Falls es allerdings Zyklen innerhalb des Graphen gibt, muss außerdem geprüft werden, ob der entsprechende Zyklus Kanten beinhaltet, welche mit dem Label ¬ versehen sind. Ist dies der Fall, ist das Programm nicht stratifizierbar. Andernfalls können für das Programm Stratifikationen gefunden werden[1, S. 124].

Der Präzedenzgraph kann außerdem dazu genutzt werden eine sinnvolle Reihenfolge der Stratifikationen festzulegen. Abgesehen von Bedingung 2.4.4, welche bereits Einschränkungen bezüglich der Reihenfolge bei Negation mitbringt, gilt es die *Starke Zusammenhangskomponenten* des Präzedenzgraphen zu finden und zu sortieren. Starke Zusammenhangskomponenten sind Teilmengen eines stark zusammenhängenden gerichteten Graphs. Ein gerichteter Graph heißt stark zusammenhängend, wenn für jedes Paar e_i, e_j gilt, dass sowohl ein Weg von e_i nach e_j, als auch umgekehrt von e_j nach e_i existiert. Die Menge aller Ecken des gerichteten Graphs E können dann in disjunkte Teilmengen zerlegt werden, für die wiederum gilt, dass jede dieser Teilmengen stark zusammenhängend ist, dass es also wieder jeweils für alle Paare e_i, e_j einen Weg e_i nach e_j und von e_j nach e_i gibt[8, S. 99]. Diese Teilmengen sind dann die entsprechenden Zusammenhangskomponenten des Graphs[8, S. 100]. Ein Beispiel für die Zusammenhangskomponenten stellt folgende Abbildung dar:

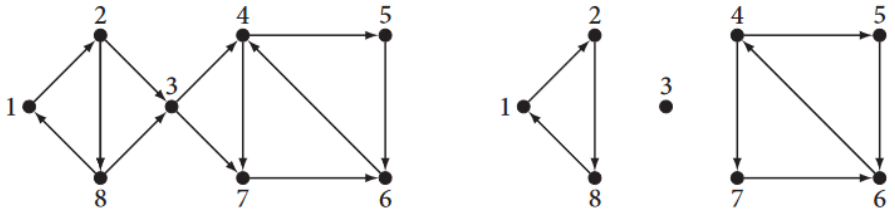

Abbildung 2: Gerichteter Graph mit seinen starken Zusammenhangskomponenten[8, S. 100]

In Abbildung 2 ist links ein stark zusammenhängender, gerichteter Graph mit acht Knoten zu erkennen. Rechts sind die einzelnen Starken Zusammenhangskomponenten des Graphen ohne die jeweiligen Kanten zueinander zu erkennen. Hier sind also all diejenigen Knoten in Gruppen dargestellt, die Wege zu anderen Knoten besitzen und rekursiv über diese anderen Knoten letztlich wieder zu sich selbst zurückführen.

Sei der Graph in Abbildung 2 nun ein Präzedenzgraph für ein Datalog Programm P, so ist es möglich anhand der Starken Zusammenhangskomponenten eine Aussage darüber zu treffen welche Regeln von P zuerst ausgeführt werden müssen, da sie sich durch Rekursion gegenseitig beeinflussen und andere aber wiederum nicht. In diesem Beispiel etwa müssen die Regeln, welche für Knoten 1, 2 und 8 stehen vor den Regeln der Knoten 3, 4, 5, 6 und 7 und Knoten 3 vor den Knoten 4, 5, 6 und 7 ausgeführt werden. Dies führt dann zu der sogenannten *Topologischen Sortierung*. Also zu der Nummerierung der einzelnen Komponenten, welche unter Berücksichtigung ihrer Beeinflussung aufeinander in eine Reihenfolge gebracht werden[8, S. 98].

Um dies an einem Beispiel zu skizzieren, soll das Programm dienen, welches berechnen kann, dass zwei Personen kein Verwandtschaftsverhältnis zueinander haben. Ein entsprechender Präzedenzgraph für dieses Programm sieht folgendermaßen aus:

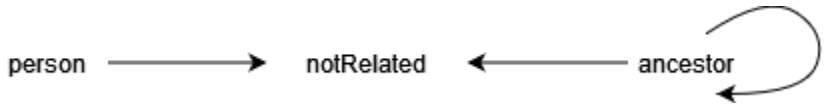

Abbildung 3: Präzedenzgraph

Anhand des Graphen ist zu erkennen, dass das Programm tatsächlich stratifizierbar ist. Der Graph beinhaltet zwar einen Zyklus, allerdings ist die entsprechende Kante nicht mit einem Negations-Label versehen.

Der Graph besitzt außerdem drei Starke Zusammenhangskomponenten. Da es sich um ein einfaches Programm mit nur einer rekursiven Regel handelt, ist dies anhand des Graphen relativ leicht auszumachen. Da der Graph beispielsweise vom Knoten `person` aus einen Kante zum Knoten `notRelated` besitzt, aber nicht zurück und vom Knoten `person` keine Kanten zum Knoten `ancestor` führen, ist der Knoten `person` als eine Zusammenhangskomponente zu betrachten. Analog gilt dasselbe für die Knoten `notRelated` und `ancestor`. Zuletzt lässt sich anhand des Graphen auch eine sinnvolle Topologische Sortierung finden. Da sowohl vom Knoten `person` als auch vom Knoten `ancestor` aus ausgehende Kanten zum Knoten `notRelated` führen, der Knoten `notRelated` selbst aber nur eingehende Knoten besitzt, müssen die Regeln `person` und `ancestor` in jedem Fall vor der Regel `notRelated` ausgewertet werden. Da es außerdem aber keine Kanten zwischen den Knoten `person` und `ancestor` gibt, kann auch die Aussage getroffen werden, dass es irrelevant ist, ob die Regel `person` oder die Regel `ancestor` zuallererst ausgewertet wird.

Wird entschieden, dass zunächst die Regel `ancestor` und anschließend die Regel `person` ausgewertet werden soll, wäre eine mögliche Stratifikation des Programms also die Folgende: (1) `ancestor`, (2) `person`, (3) `notRelated`. Soll das Programm anhand dieser Stratifikation auch gänzlich ausgewertet werden, können die einzelnen Strata schrittweise nacheinander errechnet werden, indem das jeweilige Intensionale Prädikat ausgewertet und anschließend als Extensionales Prädikat für das Stratum im nächsten Schritt erachtet wird, bis alle Prädikate ausgewertet wurden und ein Endergebnis liefern.

Für eine beispielhafte Berechnung sollen wieder die Daten aus Tabelle 2.4 verwendet werden. Da diese allerdings nur Tupel mit Personen beinhalten, welche alle in einem Verwandtschaftsverhältnis zueinander stehen, wird der Tabelle 2.4 ein weiteres Tupel (*Edward*, *Dora*) hinzugefügt, sodass diese nun aussieht wie folgt:

Tabelle 9: *parent*

X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus
Edward	Dora

Für die Berechnung muss zunächst das Prädikat `ancestor` ausgewertet werden. Dies geschieht entsprechend der Erläuterung in Abschnitt 2.3 und ergibt mit dem neuen zusätzlichen Tupel nun die folgende Tabelle:

Tabelle 10: *ancestor*

X	Y
Rose	Ronald
Hugo	Ronald
Ronald	Arthur
Arthur	Septimus
Rose	Arthur
Hugo	Arthur
Ronald	Septimus
Rose	Septimus
Hugo	Septimus
Edward	Dora

Nach der Berechnung des Prädikats stehen seine Ergebnistupel für das nächste Stratum so zur Verfügung als handele es sich bei dem Prädikat um ein Extensionales Prädikat. Danach wird das Prädikat `person` für die Berechnung des zweiten Stratums ausgewertet. Hierfür ergibt sich Tabelle 11.

Tabelle 11: *person*

X	Y
Rose	Ronald
Hugo	Arthur
Ronald	Septimus
Arthur	Dora
Edward	

Auch dieses Prädikat wird nach seiner Berechnung als ein Extensionales erachtet, sodass zum Schluss auf Grundlage der beiden nun Extensionalen Prädikate `ancestor` und `person` das Prädikat `notRelated` errechnet werden kann. Das Ergebnis welches hieraus resultiert, ist in der folgenden Tabelle festgehalten:

Tabelle 12: *notRelated*

X	Y
Rose	Dora
Hugo	Dora
Ronald	Dora
Ronald	Ronald
Arthur	Dora
Arthur	Arthur
Edward	Ronald
Edward	Arthur
Edward	Septimus

2.5 Aggregation

Neben der Berechnung von Negation und gewöhnlichen Datalog Regeln, fordern viele Anwendungen außerdem die Möglichkeit weitere Berechnungen durchzuführen, wie beispielsweise das Zählen von Ergebnistupeln einer Regel oder das Aufsummieren dieser, sofern es sich um eine numerische Domäne handelt, und Ähnliches[S. 126][1]. In diesem Abschnitt wird daher die Aggregation eingeführt. Es werden Syntax und Semantik erläutert, sowie weitere Bedingungen, welche garantieren, dass Datalog Regeln, die Aggregation nutzen, sicher evaluiert werden können. Zuletzt wird die Berechnungsweise der Aggregation in Datalog theoretisch und anhand eines Beispiels erklärt.

Formal betrachtet ist die *Aggregationsfunktion* eine Funktion, welche *Multimengen* von Werten der entsprechenden Domäne auf Werte der Domäne abbildet[1, S. 127]. Multimengen wiederum besitzen dieselben Eigenschaften wie gewöhnliche Mengen, mit dem Unterschied, dass sie Mehrfachbelegungen derselben Elemente erlauben[9, S. 3]. In Datenbanken üblicherweise verwendete Aggregationsfunktionen sind beispielsweise die `count`, `sum`, `max`, `min` und `average` Funktionen. Ein *Aggregierter Term* ist ein Ausdruck $f < t_1, ..., t_k >$ wobei f eine Aggregationsfunktion mit k Argumenten ist und $t_1, ..., t_k$ gewöhnliche Terme sind. Variablen die in gewöhnlichen Termen im Kopf einer Datalog Regel mit Aggregation auftreten, werden als *Gruppierungsvariablen* bezeichnet[1, S. 127]. Diese Gruppierungsvariablen definieren, anders als bei Sprachen wie SQL wo dies explizit passiert, implizit nach welchen Variablen aggregiert wird[1, S. 128].

Syntaktisch ändern sich Datalog Regeln, wie sie bislang erläutert wurden, insofern als dass mit der Aggregation Aggregierte Terme neben gewöhnlichen Termen im Kopf einer Regel vorkommen dürfen. Das folgende Beispielprogramm stellt ein Datalog Programm mit einer Regel, welche Aggregation beinhaltet dar:

```
ancestor(X,Y) :- parent(X,Y)

ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y)
```

```
sumAncestors(X,count<Y>) :- ancestor(X,Y)
```

Wie auch die Negation und gewöhnliche Datalog Regeln zuvor, soll die Aggregation einige Einschränkungen erhalten um eine sichere und konsistente Evaluierung der Regeln zu gewährleisten:

Bedingung 2.5.1 *Jede Variable, die im Kopf einer Regel auftaucht, muss auch im Körper der Regel auftauchen. Dies gilt im Falle von Aggregation auch für das Auftreten von Variablen innerhalb eines Aggregierten Terms[1, S. 127].*

Bedingung 2.5.2 *Variablen, die in einem Aggregierten Term im Kopf einer Regel auftreten, dürfen in keinem anderen gewöhnlichen Term im Kopf der Regel auftreten[1, S. 127].*

Bedingung 2.5.1 sorgt wieder dafür, dass auch die Variablen in der Aggregationsfunktion, durch das Vorkommen derselben Variable im Körper einer Regel, aus einer eindeutig definierten Menge stammen. Damit kann garantiert werden, dass die Aggregationsfunktion auf Werte angewandt wird, die auch tatsächlich Inhalt einer Datenbankinstanz sind. Bedingung 2.5.2 wird hier festgelegt um die Definition der Regeln möglichst einfach zu gestalten.

Die Berechnung der Aggregation wirft allerdings wieder dieselben Probleme auf, die bereits bei der Berechnung der Negation aufgetreten sind.

Bei Betrachtung des folgenden Beispielprogramms[1, S. 128]

```
p(X) :- q(X)
```

```
p(sum<X>) :- p(X)
```

ist festzustellen, dass das Programm, welches eigentlich die Summe der Variable `X` errechnen sollte, aufgrund der Rekursion in der zweiten Regel nicht endlich ist und daher keine sinnvolle Evaluierung möglich ist.

Um dies an einem Beispiel zu zeigen, soll angenommen werden, dass `q(x)` ein Extensionales Prädikat ist und die Werte $\{1, 2\}$ hält. Wird dann der Unmittelbare Folgeoperator angewandt, ergeben sich für `p(x)` nach einem ersten Schritt ebenfalls die Werte $\{1, 2\}$, da `q(x)` seine Tupelmenge auf `p(x)` abbildet und `p(x)` selbst vor Ausführung des ersten Schritts aus der leeren Menge bestand, sodass hieraus nichts hinzugefügt wird. Bei erneuter Anwendung des Unmittelbaren Folgeoperators werden im zweiten Schritt erneut die Tupel des Prädikats `q(x)` auf `p(x)` abgebildet. Außerdem kommt nun aber auch die zweite Regel für `p(x)` zum Tragen, in welcher die Werte $\{1, 2\}$ aus dem vorherigen Schritt durch die Aggregationsfunktion miteinander addiert werden. Das Ergebnis von `p(x)` ist dann also die Menge $\{1, 2, 3\}$. In einem weiteren Schritt kommen dann wieder $\{1, 2\}$ hinzu und die vorige Ergebnismenge wird addiert, was zu dem Ergebnis $\{1, 2, 6\}$ führt. Dieses Muster setzt sich praktisch unendlich fort, in welchem Werte aus dem Extensionalen Prädikat kontinuierlich hinzugefügt und die Werte aus dem vorigen Ergebnis von `p(x)` addiert werden, sodass hier niemals der Kleinste Fixpunkt gefunden werden kann.

Lösbar ist dieses Problem, indem sich auch die Aggregation der Stratifikation von Programmen bedient. Analog zur Stratifizierten Negation handelt es sich hierbei um die *Stratifzierte Aggregation*[1, S. 128].

Für die Stratifizierte Aggregation gelten die für die Stratifizierte Negation definierten Bedingungen 2.4.3 und 2.4.4 entsprechend[1, S. 128]. Zusätzlich wird außerdem die folgende Bedingung eingeführt:

Bedingung 2.5.3 *Gibt es eine Regel A :- ...B,... in einem Datalog Programm P, welche einen Aggregierten Term enthält und A ist in Stratum P_i, während B in Stratum P_j ist, dann ist $i > j$[1, S. 128].*

Analog zur Bedingung 2.4.4 der Negation soll auch für die Aggregation gelten, dass Regeln, welche Aggregierte Terme enthalten, nach Regeln ausgewertet werden, welche keine Aggregierten Terme enthalten. Hierzu wird Bedingung 2.5.3 festgelegt.

Äquivalent zur Stratifizierten Negation kann mithilfe des dort eingeführten Präzedenzgraphen auch bei der Stratifizierten Aggregation eine Stratifikation gefunden werden[1, S. 128].

Um eine sinnvolle Berechnung von Datalog Programmen mit Stratifizierter Aggregation zu gewährleisten, erweitern wir den Unmittelbaren Folgeoperator aus Abschnitt 2.3 wie folgt. Sei I eine Datenbankinstanz, P ein Datalog Programm und r eine Grundinstanz einer Datalog Regel in P mit einer Aggregationsfunktion in der Form

$$R(f < t_1, ..., t_n >) \text{ :- } B_1, ..., B_n$$

Außerdem sei Θ die Menge der Substitutionen θ aller Variablen von r, welche keine Gruppierungsvariablen sind, durch Konstanten, sodass $\theta B_1, ..., B_n$ in I enthalten sind. Dann ist ein Fakt $R(c_1, ..., c_n)$ eine Unmittelbare Folge der Regel für die Datenbankinstanz I wenn außerdem die folgenden Bedingungen erfüllt sind[1, S. 129]:

Bedingung 2.5.4 *Die Menge Θ ist nicht leer[1, S. 129].*

Bedingung 2.5.5 *Für $1 \leq i \leq n$ muss gelten, dass wenn t_i eine Konstante ist, dann entspricht $t_i = c_i$[1, S. 129].*

Bedingung 2.5.6 *Für $1 \leq i \leq n$ muss gelten, dass wenn t_i ein Aggregierter Term $f(u_1, ..., u_k)$ ist, dann sei*

$$c_i = f((\theta u_1, ..., \theta u_k) | \theta \in \Theta),$$

wobei die Menge Θ in diesem Fall eine Multimenge darstellt[1, S. 129].

Bei der Aggregation wird also sichergestellt, dass sämtliche Variablen innerhalb einer Aggregationsfunktion durch Konstanten substituiert werden, um zu gewährleisten, dass sie sich im Laufe der Berechnungen nicht ändern. Der Unmittelbare Folgeoperator ist abgesehen davon für die Aggregation so definiert, wie er bereits im Abschnitt 2.3 definiert wurde.

Die Berechnung der stratifizierten Aggregation funktioniert auf Grundlage des beschriebenen theoretischen Hintergrunds dann äquivalent zur stratifizierten Negation[1, S. 129].

Als Illustration eines einfachen Beispiels soll erneut dieses Programm dienen:

```
ancestor(X,Y) :- parent(X,Y)

ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y)

sumAncestors(X,count<Y>) :- ancestor(X,Y)
```

Die Berechnung der Regel `ancestor` funktioniert hier offensichtlich genau wie in Abschnitt 2.3 beschrieben. Als Datengrundlage sollen wieder diejenigen Daten dienen, welche vor Einführung der Negation angenommen wurden, sodass die Ergebnistabelle des Prädikats `ancestor` der Tabelle 2 entspricht.

Der Präzedenzgraph für das Programm ist der folgende Graph:

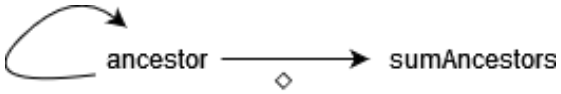

Abbildung 4: Präzedenzgraph

Hieraus ergibt sich nach Topologischer Sortierung, entsprechend der Erläuterungen in Abschnitt 2.4, die Stratifikation (1) `ancestor`, (2) `sumAncestors`.

Da Schritt (1) bereits errechnet wurde (siehe Tabelle 2), wird dies an dieser Stelle nicht nochmals wiederholt. Unter der Annahme dass das Prädikat `ancestor` nach der Berechnung als ein Extensionales Prädikat gilt, kann das Prädikat `sumAncestors` in Schritt (2) errechnet werden. Hierzu werden zunächst alle Variablen `Y` durch Konstanten mit denselben Werten substituiert, um sicherzustellen, dass sich die Terme in der Aggregationsfunktion auch im Falle einer weiteren Rekursion des Prädikats `sumAncestors` nicht ändern können. Anschließend kann das Prädikat `sumAncestors` unter Anwendung der Aggregationsfunktion evaluiert werden und liefert folgendes Ergebnis:

Tabelle 13: *sumAncestors*

X	count<Y>
Rose	3
Hugo	3
Ronald	2
Arthur	1

3 Partitionsprädikat

Der Kern dieser Arbeit besteht darin ein neues Prädikat, das sogenannte *Partitionsprädikat*, einzuführen. Das *Partitionsprädikat* ist analog zum Aggregationsprädikat ein intensionales Prädikat, welches Daten aus einem extensionalen Prädikat oder anderen intensionalen Prädikaten ableitet. Zusätzlich dazu ist es in der Lage mithilfe spezieller Funktionen weitere Informationen zu generieren. Diese Funktionen des Partitionsprädikats besitzen die Fähigkeit Werttupel nach gegebenen Kriterien zu sortieren und damit Ränge zu errechnen. Hierzu soll erläutert werden inwiefern ein solches Partitionsprädikat syntaktisch aufgebaut ist, wie es semantisch funktioniert und welche Problematiken die Anwendung des Prädikats mit sich bringt und wie diese gelöst werden können.

3.1 Partitionsprädikat in Datalog

Die Syntax des Partitionsprädikats ist in Anlehnung an das in Abschnitt 2.5 vorgestellte Aggregationsprädikat die Folgende:

$$A(X, asc\ rank < Z > by\ Y) \text{ :- } B(X, Y, Z)$$

Dabei seien A und B gewöhnliche Atome. Auch bei der Partition, analog zur Aggregation, soll es erlaubt sein Funktionen im Kopf einer Regel anzuwenden. Diese Funktionen nennen sich hier *Partitionsfunktionen*. Vorgestellt werden zwei unterschiedliche Partitionsfunktionen `rank`, welche das sogenannte *lückenlose Ranking* implementiert und `uniqueRank`, welche das sogenannte *eindeutige Ranking* implementiert. Prinzipiell ist es aber denkbar jede mögliche Art des Sortierens und Nummerierens in Form einer eigens erdachten Partitionsfunktion zu implementieren. Die Partitionsfunktion $rank < Z >$ ist außerdem ein *partitionierter Term*, welcher, entsprechend der Aggregation, formal durch den Term $f < t >$ ausgedrückt werden kann. t sei auch hier ein gewöhnlicher Term. Der Unterschied zum Aggregierten Term ist der, das zunächst nur ein einziger Term als Argument der Partitionsfunktion erlaubt wird. Die Erweiterung des Partitionierten Terms um die Form $f < t_1, ..., t_n >$ wird im späteren Verlauf erläutert. Auch die Variable X ist als ein gewöhnlicher Term zu erachten. Sie gehört weder zum Partitionierten Term, noch zu den *Partitionierungsvariablen* und dient schlichtweg dazu Tupel aus dem

Atom B abzubilden. Die *Partitionierungsvariablen* sind hier formal durch die Variable Y vertreten und können eine leere Menge oder eine Menge von Variablen mit beliebiger Mächtigkeit sein. Anders als bei der Aggregation sind die Partitionierungsvariablen im Partitionsprädikat durch ein Schlüsselwort `by` explizit ausgewiesen. Diese explizite Nennung ist notwendig um zu gewährleisten, dass Terme zum Abbilden von Tupeln genutzt werden können ohne dass diese einen Einfluss auf die Partitionierung selbst haben. Zuletzt gilt das Schlüsselwort `asc` als Sortierrichtung und kann alternativ durch das Schlüsselwort `desc` ersetzt werden, wobei `asc` (ascending, zu deutsch: aufsteigend) eine aufsteigende und `descending` (descending, zu deutsch: absteigend) eine absteigende Sortierrichtung bewirken.

Semantisch bewirkt das Partitionsprädikat, dass sämtliche Tupel, welche sich aus Atom B ergeben, zunächst entsprechend der Partitionierungsvariablen in Partitionen eingeteilt werden. In dem Fall, dass die Partitionierungsvariablen eine leere Menge sind, entsteht eine einzige Partition in welcher alle Tupel enthalten sind. Andernfalls werden die Tupel anhand der Partitionierungsvariablen in Partitionen aufgeteilt, welche dann abgeschlossene Einheiten darstellen, sodass jede Partition für sich steht. Anschließend wird die Partitionsfunktion auf die Tupel in den Partitionen angewandt und der Rang daraus errechnet.

Die beiden Partitionierungsfunktionen `rank` und `uniqueRank` implementieren, wie bereits genannt, zwei unterschiedliche Sortierfunktionen. Die Funktionen unterscheiden sich aber erst dann, wenn mehrere Tupel für die gewählte Variable im Partitionierungsterm dieselben Werte aufweisen. Die Funktion `rank` weist den Tupeln nach dem *lückenlosen Ranking* denselben Rang zu, der dann entsprechend auch innerhalb derselben Partition mehrfach vergeben wird. Die Funktion `uniqueRank` wählt für jedes der Tupel rein zufällig unterschiedliche Ränge, sodass aber in jedem Fall kein Rang innerhalb derselben Partition mehrfach vergeben wird. Zu beachten ist hierbei allerdings, dass die zufällige Vergabe der Ränge nichtdeterministisch ist. Möglichkeiten dieses Konzept in der praktischen Implementierung etwas deterministischer zu gestalten, werden allerdings im entsprechenden Kapitel 5 diskutiert.

Um das Partitionsprädikat anhand von Beispielen zu illustrieren, soll zunächst angenommen werden, dass es ein extensionales Prädikat *person*(*Name*, *Geschlecht*) gibt, welche die folgenden Werttupel aufweist:

Tabelle 14: *person*

Name	Sex
Rose	female
Hugo	male
Ronald	male
Arthur	male
Dora	female

Hierbei kann das Partitionsprädikat genutzt werden um alle Personen der Beispieldaten anhand ihrer Namen zu sortieren. Die Regel sowie eine Ergebnistabelle für ein entsprechendes Partitionsprädikat sehen folgendermaßen aus:

```
peopleByName(Name,asc rank<Name>) :- person(Name,Sex)
```

Tabelle 15: *peopleByName*

Name	Rank
Rose	4
Hugo	2
Ronald	3
Arthur	0
Dora	1

Hierbei ist zu beachten, dass die Regel keine Partitionierungsvariablen aufweist. Daher fallen alle Tupel unter dieselbe Partition, auf welche dann die Partitionsfunktion angewendet wird. Da die Regel außerdem eine aufsteigende Sortierung angibt, erhält dasjenige Tupel, dessen Name alphabetisch zuerst kommt den niedrigsten Rang. Bei Verwendung der absteigenden Sortierung mit dem Schlüsselwort `desc`, würde die Zählung bei dem Namen, welcher alphabetisch zuletzt kommt, mit dem niedrigsten Rang beginnen und von dort aus absteigend weiter zählen.

Ein Beispiel für ein Partitionsprädikat, welches die Partitionierung auch tatsächlich nutzt, stellt die folgende Regel dar:

```
peopleByName(Name,Sex,asc rank<Name> by Sex) :- person(Name,Sex)
```

Dieses Prädikat partitioniert nach dem Geschlecht der Personen, sodass sich aus der Gesamtmenge an Tupeln zwei Partitionen bilden, auf welche jeweils die Partitionsfunktion angewendet wird. Das Ergebnis, welches sich daraus ergibt, ist in der folgenden Tabelle zu betrachten:

Tabelle 16: *peopleByName*

Name	Geschlecht	Rang
Rose	female	1
Dora	female	0
Hugo	male	1
Ronald	male	2
Arthur	male	0

In zwei weiteren Beispielen sollen außerdem die Unterschiede zwischen dem lückenlosen und dem einzigartigen Ranking illustriert werden. In den beiden vorherigen Beispielen wurde die Partitionsfunktion `rank` verwendet. Tatsächlich ergibt sich bei diesen Beispielen bei Anwendung der `uniqueRank` Funktion dasselbe Ergebnis, da keine Tupel dieselben Werte für die Partitionierungsvariable aufweisen. Aus der Regel

```
peopleByName(Name,asc uniqueRank<Name>) :- person(Name,Sex)
```

würde also ebenfalls die Ergebnistabelle 15 resultieren. Analog dazu gilt für die Regel

```
peopleByName(Name,Sex,asc uniqueRank<Name> by Sex) :- person(Name,Sex)
```

und die Tabelle 16 das Gleiche. Es soll nun allerdings ein leicht verändertes Extensionales Prädikat `person(Name,Sex,Birthday)` angenommen werden, welches als Grundlage für die folgenden Beispiele dient. Für dieses Prädikat sollen die Werttupel wie folgt dienen:

Tabelle 17: *birthdayRank*

Name	Sex	Birthday
Rose	female	19.03.
Hugo	male	06.02.
Ronald	male	01.03.
Arthur	male	06.02.
Dora	female	06.02.

Wie zu erkennen ist, besitzen nun drei Tupel denselben Wert für die Variable `Birthday`. Wird diese Variable auch als Sortiervariable genutzt, führt das zu unterschiedlichen Ergebnissen bei Anwendung beider Partitionsfunktionen `rank` und `uniqueRank`.

Zunächst soll der Fall betrachtet werden bei welchem `rank` als Partitionsfunktion verwendet, nach dem Geschlecht partitioniert und nach dem Geburtstag der Personen sortiert wird. Eine entsprechende Regel hierfür könnte so aussehen und die Ergebnistabelle 18 liefern:

```
birthdayRank(Name,Sex,Birthday,asc rank<Birthday> by Sex) :- person(Name,Sex,Birthday)
```

Tabelle 18: *birthdayRank*

Name	Sex	Birthday	Rank
Rose	female	19.03.	1
Dora	female	06.02.	0
Hugo	male	06.02.	0
Ronald	male	01.03.	1
Arthur	male	06.02.	0

Zu beachten seien hier die Tupel $(Hugo, m, 06.02., 0)$ und $(Arthur, m, 06.02., 0)$, welche tatsächlich die gleichen Ränge in derselben Partition aufweisen.

Wird eine solche Regel durch das eindeutige Ranking definiert, sieht sie so aus:

```
birthdayRank(Name,Birthday,asc uniqueRank<Birthday> by Sex) :- person(Name,Sex,Birthday)
```

Und liefert die folgende Ergebnistabelle:

Tabelle 19: *birthdayRank*

Name	Geburtstag	$Rang_1$	$Rang_2$
Rose	19.03.	1	1
Dora	06.02.	0	0
Hugo	06.02.	1	0
Ronald	01.03.	2	2
Arthur	06.02.	0	1

Hierbei entstehen zwei alternative Ränge, welche in der Tabelle als $Rang_1$ und $Rang_2$ bezeichnet sind. Diese stellen die beiden möglichen Alternativen dar, die sich für den tatsächlichen Rang letztendlich ergeben. Festzustellen ist, dass sich tatsächlich keine Dopplungen der Ränge ergeben, alle Ränge innerhalb einer Partition also einzigartig und eindeutig sind.

Um zu erlauben, dass Partitionsfunktionen, analog zu Aggregationsfunktionen, auch Partitionierte Terme der Form $f < t_1, ..., t_n >$ beinhalten können, ist es notwendig eine Sortierung nach beliebig vielen Kriterien zu realisieren. Semantisch hat eine solche Sortierung die Konsequenz, dass immer dann, wenn Tupel für die Variable im gegebenen Partitionierten Term dieselben Werte aufweisen, diese Tupel ebenfalls nach der darauffolgenden Variable sortiert werden, solange bis entweder ein eindeutiger Rang gefunden wurde, oder alle angegebenen Variablen im Partitionierten Term einmal zur Sortierung verwendet wurden.

Als Beispiel für eine Sortierung nach beliebig vielen Kriterien soll die Regel dienen, aus welcher Tabelle 18 entstand. Diese wird allerdings dahingehend verändert, als dass der Partitionierte Term nicht mehr nur die Variable `Birthday`, sondern nun zusätzlich auch die Variable `Name` hält, sodass die Regel nun folgendermaßen aussieht:

```
birthdayRank(Name,Birthday,asc rank<Birthday,Name> by Sex) :- person(Name,Sex,Birthday)
```

Diese Regel wird zunächst genauso ausgewertet, wie die, welche ausschließlich die Variable `Birthday` als Sortierkriterium nutzt und führt in einem ersten Schritt zu demselben Ergebnis, welches in Tabelle 18 betrachtet werden kann. Anschließend wird die Variable `Name` aber ebenfalls als Sortierkriterium für all diejenigen Tupel in Betracht genommen, welche aus der ersten Sortierung heraus denselben Rang erhalten. Dies entspricht im konkreten Beispiel den Tupeln $(Hugo, m, 06.02.)$ und $(Arthur, m, 06.02.)$. Diese beiden Tupel werden anhand der Werte, welche sie für die Variable `Name` halten, ein weiteres Mal verglichen sodass schlussendlich folgende Ergebnistabelle aus der gesamten Regel resultiert:

Tabelle 20: *birthdayRank*

Name	Sex	Birthday	Rank
Rose	female	19.03.	1
Dora	female	06.02.	0
Hugo	male	06.02.	1
Ronald	male	01.03.	2
Arthur	male	06.02.	0

Wie zu erkennen ist, erhalten die beiden Tupel $(Hugo, m, 06.02.)$ und $(Arthur, m, 06.02.)$ nun unterschiedliche Ränge anhand der alphabetischen Reihenfolge ihrer Namen. Damit eignet sich das Sortieren nach mehreren Kriterien beim lückenlosen Ranking neben dem eindeutigen Ranking dazu, eindeutige Ränge für sämtliche Tupel einer Partition zu generieren. Äquivalent kann das Sortieren nach mehreren Kriterien auch beim eindeutigen Ranking genutzt werden und führt dazu, dass Ränge solange deterministisch eindeutig errechnet werden, bis die Anzahl an angegebenen Sortierkriterien erschöpft ist.

Mit den vorangegangen Beispiele konnten Wirkungsweise und Eigenschaften des Partitionsprädikats unter Berücksichtigung der verschiedenen Partitionsfunktionen für einfache Regeln illustriert werden. Nun stellt sich allerdings wiederum die Frage, ob Rekursion ebenfalls zu Problemen führen kann, wie das etwa bei Negation und Aggregation geschieht.

Grundsätzlich geschieht die Rekursion bei der Partitionierung ebenfalls durch die Kleinster-Fixpunkt-Semantik. Tatsächlich zeigen sich bei bestimmten Konfigurationen des Partitionsprädikats unter Verwendung von Rekursion aber ähnliche Effekte, wie das beim Aggregationsprädikat der Fall ist.

Zur Illustration der Problematik sollen folgende Regeln angenommen werden:

```
p(X,Y) :- e(X,Y)

p(asc uniqueRank<Y>,Y) :- p(X,Y)
```

Das Prädikat `e` soll dabei ein Extensionales Prädikat sein und folgende Tupel beinhalten:

Tabelle 21: *e*

X	Y
1	1
1	2

Hierbei werden dann also durch Anwendung der ersten Regel stets die Werttupel des Prädikats `e` auf `p` projiziert. Hier ist dies bei erstmaliger Anwendung des Unmittelbaren Folgeoperators das einzige was `p` beeinflusst, da die rekursive Regel, welche zusätzlich einen Rang berechnet

auf eine noch leere Tupelmenge von `p` stößt und auf diese keine Auswirkungen hat, sodass die Ergebnismenge von `p` nach dem ersten Schritt der Tupelmenge von `e` gleicht. Im darauffolgenden Schritt haben dann beide Regeln einen Effekt auf die Ergebnismenge, sodass die numerischen Werte in Y sortiert werden und entsprechende Ränge errechnen. Das Ergebnis ist das Folgende:

Tabelle 22: p

Rank	Y
1	1
2	2
1	2

Bei erneuter Anwendung des Unmittelbaren Folgeoperators, wird wieder sortiert, sodass der erste Wert aller Tupel von `p` jeweils den errechneten Rang erhält, welcher aus der Sortierung des zweiten Werts aller Tupel von `p` resultiert. Da hier die Partitionsfunktion `uniqueRank` verwendet wird, erhalten die beiden Tupel, $(2, 2)$ und $(1, 2)$ in jedem Fall unterschiedliche Ränge, sodass eines der Tupel dann den Rang 2 erhält und der jeweils andere den Rang 3. Anschließend wird wieder das Tupel $(1, 2)$ aus dem extensionalen Prädikat `e` hinzu projiziert, sodass die Ergebnismenge dann folgendermaßen aussieht:

Tabelle 23: p

Rank	Y
1	1
2	2
3	2
1	2

Es ist leicht vorstellbar wie weitere Anwendungen des Unmittelbaren Folgeoperators sich auf die Ergebnismenge auswirken. Im nächsten Schritt wird eines der drei Tupel, dessen Y Wert eine 2 enthält den Rang 4 erhalten, die anderen beiden Tupel mit Y Werten von 2 die Ränge 2 und 3. Anschließend wird erneut das Tupel $(1, 2)$ hinzu projiziert. Dies wird dann, ähnlich wie das bei rekursiven Regeln mit Aggregation der Fall ist, endlos weitergehen, dass sich die Ränge der Tupel durch Sortierung neu ordnen und stets ein weiteres Tupel durch das extensionale Prädikat hinzugefügt wird. Hierbei kann also nie ein kleinster Fixpunkt gefunden werden.

Um eine sichere Evaluierung von Datalog Programmen mit Partitionierung zu garantieren, wird daher auch hier eine Bedingung für die korrekte Definition, sowie die *stratifizierte Partitionierung* eingeführt:

Bedingung 3.1.1 *Jede Variable, die im Kopf einer Regel auftaucht, muss auch im Körper der Regel auftauchen. Dies gilt im Falle von Partitionierung auch für das Auftreten von Variablen innerhalb eines partitionierten Terms*

Bedingung 3.1.1 sorgt abermals dafür, dass auch Variablen in der Partitionsfunktion von einer Variable im Körper der Regel projiziert wird und damit aus einer eindeutig definierten Menge stammen, welche sich tatsächlich in der Datenbank befinden.

Auch für die *stratifizierte Partitionierung* gelten die Bedingungen 2.4.3 und 2.4.4 entsprechend der stratifizierten Negation und Aggregation. Darüber hinaus wird hier eine weitere Bedingung eingeführt:

Bedingung 3.1.2 *Gibt es eine Regel A :- ...B,... in einem Datalog Programm P, welche einen partitionierten Term enthält und A ist in Stratum P_i, während B in Stratum P_j ist, dann ist $i > j$*

Bedingung 3.1.2 gewährleistet, dass Strata, welche Partitionierung enthalten zuletzt ausgewertet werden.

Um eine sinnvolle und valide Stratifikation zu finden, bedienen wir uns außerdem wieder dem Präzedenzgraphen, welcher bei der Partitionierung äquivalent definiert ist, wie bei der Negation und Aggregation. Zusätzlich wird ein entsprechendes Label für die Partition festgelegt.

Der unmittelbare Folgeoperator ist bei der Partitionierung so definiert, wie bei der Negation. Eine Erweiterung um die Substitution von Variablen durch Konstanten, wie bei der Aggregation, ist hier nicht nötig, da die Endlichkeit des entsprechenden Datalog Programms hierdurch nicht herbeigeführt werden kann, denn während bei der Aggregation die Veränderlichkeit der einzelnen Variablen dazu führt, dass die Aggregationsfunktion mit ständig neuen Tatsachen konfrontiert wird, liegt das Problem bei der Partitionierung darin, dass extensionale Prädikate immer neue Tatsachen zu einer Tupelmenge hinzufügt und davon abhängige intensionale Prädikate dadurch neu evaluiert werden müssen. Dies lässt sich nicht mithilfe von Konstanten lösen, stattdessen aber durch einfache Stratifikation, wie sie im Abschnitt 2.4 definiert wurde.

Zur Illustration der Stratifikation bei der Partitionierung erweitern wir das Eingangsbeispiel um zwei weitere Regeln und erhalten damit folgendes Datalog Programm:

```
p(X,Y) :- e(X,Y)

p(X,Y) :- e(X,Z), p(Z,Y)

q(asc uniqueRank<Y>,Y) :- p(X,Y)
```

Abbildung 5 zeigt den Präzedenzgraphen des Programms. Zu erkennen ist darauf der Zyklus des Prädikats `p`, sowie eine Kante von `p` nach `q`, welches mit dem Partitionslabel versehen ist.

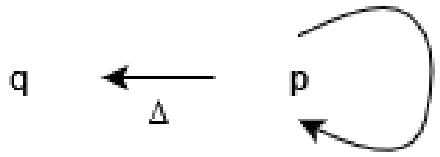

Abbildung 5: Präzedenzgraph

Nach topologischer Sortierung ergibt sich daraus folgende Stratifikation: (1) `p`, (2) `q`.

Es wird angenommen, dass das extensionale Prädikat `e` in diesem Beispiel folgende Tupelmenge enthält:

Tabelle 24: *e*

X	Y
1	1
1	2
2	3

Bei der Evaluierung des Prädikats `p` wird, entsprechend der Berechnung rekursiver Regeln, wie in Abschnitt 2.3 erläutert, zunächst die Tupelmenge von `e` auf `p` projiziert. Daraus ergibt sich eine Ergebnismenge von `p`, welche der Tupeln des Prädikats `e` entspricht. Anschließend wird bei erneuter Anwendung des unmittelbaren Folgeoperators der Verbund der beiden Prädikate `e` und `p` über die Variable `Z` gebildet, sodass das Prädikat `p` dann die Ergebnismenge, wie in Tabelle 25 dargestellt, enthält.

Tabelle 25: *p*

X	Y
1	1
1	2
2	3
1	3

Sobald diese Ergebnismenge von `p` feststeht, wird das Prädikat `p` wieder als ein extensionales Prädikat angenommen, auf dessen Grundlage `q` errechnet werden kann. Hierzu wird dann der aufsteigende einzigartige Rang nach der Variable `Y` errechnet, woraus folgendes Ergebnis resultiert:

Tabelle 26: q

Rank	Y
1	1
2	2
3	3
4	3

Neben der Notwendigkeit zur Stratifikation, welche die Regelberechnung bei der Partitionierung zwangsläufig mit sich bringt, resultiert aus der Anwendung der Stratifikation außerdem ein praktischer Nutzen. Sie kann mithilfe des Präzedenzgraphen und der schrittweisen Evaluierung voneinander abhängiger Regeln dazu beitragen, dass Regeln in einer sinnvollen Reihenfolge ausgeführt werden. Wenn eine Regel R beispielsweise von vielen anderen Regeln abhängt, kann mithilfe der topologischen Sortierung eine Reihenfolge festgelegt werden, sodass erst alle Prädikate, von welchen R abhängt, berechnet werden, bevor R selbst berechnet wird, anstelle dass dies in einer zufälligen Reihenfolge geschieht und das Prädikat R aufgrund zwischendurch neu hinzugewonnener Tupel mehrfach ausgewertet werden muss. In der konkreten Implementierung des Partitionsprädikats führt dies zu einer nennenswerten Effizienzerhöhung, da die Berechnung des Partitionsprädikats sehr aufwendig ist.

3.2 Partitionsprädikat in Roxx

Das Partition Predicate stellt ein fortgeschrittenes Feature der Sprache Roxx dar[4]. Es ist auch hier ein spezielles Intensionales Prädikat, das sich syntaktisch und semantisch aber von anderen gewöhnlichen Intensionalen Prädikaten abhebt.

Die folgende Abbildung zeigt die Syntax eines beispielhaften Partition Predicates.

```
partitionPredicate {
    employeeBirthdayRank('emplGroup', 'employee', 'birthday', 'birthdayRank') {
        from = employeesBirthdays('emplGroup', 'employee', 'birthday')
        partitionBy = ['emplGroup'] // Can also be more than one attribute. Can also be empty.
        birthdayRank = Rank(OrderDesc('birthday')) // "OrderAsc" is also possible.
    }
}
```

Abbildung 6: Beispiel Partition Predicate[4]

Im Allgemeinen strukturiert Roxx einige sogenannte High-Level Konstrukte, wie das Partition Predicate, durch eine ganz bestimmte, festgelegte Syntax, die es erlaubt diese Konstrukte, die allesamt Intensionale Prädikate sind, von anderen gewöhnlichen Intensionalen Prädikaten zu unterscheiden. Die Syntax dieser Prädikate folgt also dem Prinzip der Schlüsselwörter, die

insbesondere in imperativen Programmiersprachen verwendet werden.

Semantisch erlaubt das Partition Predicate Partitionen anhand der gewählten Attribute aus der gesamten Menge von Elementen des angegebenen Prädikats zu erstellen.

Dabei gibt die Zeile

```
from = employeesBirthdays('emplGroup', 'employee', 'birthday')
```

das Prädikat mit entsprechenden Attributen an, das als Grundlage für das Partitionieren dienen soll.

Die Zeile

```
partitionBy = ['emplGroup']
```

erwartet eine Liste von Attributen, die einstellig, mehrstellig oder auch leer sein kann. Diese Attribute bestimmen dann wonach partitioniert wird. In diesem Beispiel ist das eine Gruppe von Arbeitnehmerinnen und Arbeitnehmern, definiert über das Attribut *emplGroup*.

Zuletzt gibt es eine Operation, die ein Partition Predicate ausführen kann.
Über die *Rank* Operation ist es der*dem Modellierer*in möglich innerhalb der einzelnen Partitionen nach einem Attribut zu sortieren. Jedes Element, erhält daraufhin eine von 0 beginnend aufsteigende Nummerierung, die dann dem *Rank*, also einem Rang entspricht Hierzu wählt die*der Modellierer*in einen beliebigen Bezeichner und weist diesem dann das Ergebnis der *Rank* Operation zu. Diese *Rank* Operation entspricht dem sogenannten *Dense-Ranking*. Das *Dense-Ranking* ist eine Sortiermethode, bei welcher die Rangvergabe lückenlos erfolgt und die mehrfache Vergabe desselben Rangs erlaubt ist. (*Dense* zu deutsch: dicht.) Das bedeutet, dass bei dem Fall bei welchem derselbe Rang mehrfach vergeben wird, darauffolgende Ränge dennoch lückenlos an vorangegangene Ränge anschließen. Im oben aufgeführtem Beispiel entspricht die *Rank* Operation folgender Zeile:

```
birthdayRank = Rank(OrderDesc('birthday'))
```

Die Sortierung nach dem gewählten Attribut kann mit der Funktion *OrderAsc()* sowohl aufsteigend als auch absteigend durch die Funktion *OrderDesc()* ausgewertet werden[4].

Zur Veranschaulichung der Funktionalität des Partition Predicates wird das in Abbildung 2 aufgeführte Partition Predicate im Folgenden anhand von Beispieldaten evaluiert.

Das Prädikat, welches als Grundlage für das Partition Predicate verwendet wurde, *employeesBirthday*, enthält drei Attribute: die Gruppe von Arbeitnehmerinnen und Arbeitnehmern

emplGroup, die entsprechende Mitarbeiterin oder der entsprechende Mitarbeiter *employee*, sowie der Geburtstag der Mitarbeiterin oder des Mitarbeiters *birthday*. Daraus ergeben sich für das Prädikat *employeesBirthday* Beispieldaten in einer Tabelle wie in Abbildung 3 aufgeführt.

emplGroup	employee	birthday
EmplGr3	Jake	21.02.1978
EmplGr2	Haley	05.09.1988
EmplGr2	Ron	03.04.1986
EmplGr1	Anne	11.12.1996
EmplGr1	Sofia	07.01.1974
EmplGr2	Tina	03.04.1986
EmplGr1	Peter	22.08.1990

Abbildung 7: Beispieldaten [4]

Wie in Abbildung 4 erkennbar ist, wurden die Beispieldaten nun nach dem Attribut *emplGroup* partitioniert. Um die einzelnen Partitionen in der Abbildung besser erkennen zu können, wurden die zu einer Partition zugehörigen Zeilen und Spalten außerdem in einer einheitlichen Farbe markiert.

emplGroup	employee	birthday
EmplGr3	Jake	21.02.1978
EmplGr2	Haley	05.09.1988
EmplGr2	Ron	03.04.1986
EmplGr2	Tina	03.04.1986
EmplGr1	Anne	11.12.1996
EmplGr1	Sofia	07.01.1974
EmplGr1	Peter	22.08.1990

Abbildung 8: Beispieldaten nach der Partitionierung [4]

Nach der Auswertung der *Rank* Operation wird nun jeder Zeile der Tabelle und somit jeder Mitarbeiterin und jedem Mitarbeiter ein Rang anhand der absteigenden Reihenfolge des Sortierattributs *birthday* zugeordnet. Der Rang gilt für alle Elemente innerhalb einer Partition und beginnt daher bei jeder neuen Partition wieder beim Startwert 0.

Wichtig zu beachten ist allerdings auch der Fall, bei welchem derselbe Wert des Sortierattributs für mehrere Spalten auftritt. Dieser ist in der unten aufgeführten Abbildung in den Spalten des

Mitarbeiters *Ron* und der Mitarbeiterin *Tina* zu beobachten. Beide Mitarbeiter besitzen den Wert 03.04.1986 für das Attribut *birthday*. Die Konsequenz daraus ist, dass beide nach dem *Dense-Ranking* denselben Rang, den Startwert 0, erhalten und die nächste unterscheidbare Zeile den um eins erhöhten Rang erhält.

emplGroup	employee	birthday	birthdayRank
EmplGr3	Jake	21.02.1978	0
EmplGr2	Ron	03.04.1986	0
EmplGr2	Tina	03.04.1986	0
EmplGr2	Haley	05.09.1988	1
EmplGr1	Sofia	07.01.1974	0
EmplGr1	Peter	22.08.1990	1
EmplGr1	Anne	11.12.1996	2

Abbildung 9: Beispieldaten nach Evaluierung der *Rank*-Operation[4]

4 Domänenspezifische Sprachen im Java Umfeld

In diesem Kapitel soll erläutert werden was domänenspezifische Sprachen sind, welche Formen sie annehmen können und wie diese konkret mit Java und Java verwandten Sprachen realisiert werden können.

Eine *domänenspezifische Sprache* ist eine Programmiersprache mit verminderter Ausdrucksfähigkeit, welche sich auf eine bestimmte Domäne spezialisiert[10, S. 27]. Ihre verminderte Ausdrucksfähigkeit führt insbesondere dazu, dass domänenspezifische Sprachen nicht Turing-vollständig sind[10, S. 30]. Domäne meint im Kontext der domänenspezifischen Sprachen die Fachgebiete, zu welchen Applikationen entwickelt werden. Als Programmiersprache dient eine domänenspezifische Sprache dazu Programmcode einfach schreib- und lesbar und zusätzlich von Computern ausführbar zu gestalten. Im Gegensatz zu Allzweck-Programmiersprache, die eine Vielzahl an Möglichkeiten bieten Daten eines Programms zu manipulieren oder Funktionalität durch Kontroll-, Abstraktionsstrukturen und Ähnlichem zu implementieren um sämtliche lösbaren Probleme bearbeiten zu können. konzentrieren sich domänenspezifische Sprachen auf ein Minimum an Funktionalität, die unbedingt benötigt wird um die tatsächlich zu lösenden Probleme einer entsprechenden Domäne zu behandeln[10, S. 27].

Aufgrund dessen können einzelne domänenspezifische Sprachen nur für einen kleinen in sich abgeschlossenen Themenbereich sinnvoll eingesetzt werden und werden nicht über verschiedene Domäne hinweg verwendet. Diese Eigenschaften machen domänenspezifische Sprachen sehr einfach, allerdings sind sie aufgrund ihrer Einfachheit nicht Turing-vollständig, da sie eben über eine Vielzahl an Funktionalität, welche Turing-vollständige Allzweck-Sprachen besitzen, nicht verfügen. Die Konsequenz daraus ist, dass eine domänenspezifische Sprache niemals dazu genutzt werden kann eine vollständige Applikation zu schreiben. Sie dient immer

nur als Ergänzung und kann damit Teile der Implementierung einer Applikation abdecken. Ein wesentlicher Vorteil, der damit erreicht wird, ist dass domänenspezifische Sprachen nicht zwangsläufig von Personen genutzt werden müssen, die über ein umfangreiches Verständnis über Allzweck-Programmiersprachen und Programmierkonzepten verfügen. Die Einfachheit einer domänenspezifischen Sprache führt nämlich dazu, dass Spezialisten, welche sich mit der Domäne, nicht aber mit Programmierung und Softwareentwicklung auskennen, nicht auch den Code verstehen müssen, welcher restliche Teile einer Applikation implementiert[10, S. 13]. Um die genannte Einfachheit zu erreichen, wird die Syntax domänenspezifischer Sprachen häufig so gewählt, dass sie außerdem, im Gegensatz zu imperativen Allzweck-Programmiersprachen, einen eher deklarativen Charakter hat, welcher sich an die gesprochene Sprache anlehnt[10, S. 16]. Syntaktisch fügt die domänenspezifische Sprache üblicherweise also Teilstücke aus dem Gesamtkontext so zusammen, dass sie eher an vollständige Sätze gleichen und keine zusammenhangslosen Befehle darstellen, wie dies bei imperativen Allzweck-Sprachen in der Regel der Fall ist[10, S. 30].

Bezüglich der Implementierung und konkreten Umsetzung einer domänenspezifischen Sprache gibt es zwei wesentliche Konzepte, welche sich gegenüberstehen.

Eine dieser Formen ist die *externe domänenspezifische Sprache*. Die Programmiersprache in welcher eine domänenspezifische Sprache und die Applikation, in der sie eingebettet ist, geschrieben ist, soll im weiteren Verlauf als *Grundsprache* bezeichnet werden. Eine externe domänenspezifische Sprache zeichnet sich dann dadurch aus, dass sich ihre Repräsentation von der Grundsprache unterscheidet. Eine externe domänenspezifische Sprache besitzt damit entweder eine eigens für sie definierte Syntax, oder sie bedient sich wiederum anderen syntaktischen Repräsentationen, wie beispielsweise die der *Extensible Markup Language* (XML)[10, S. 15]. Häufig wird neben der externen domänenspezifischen Sprache selbst, zusätzlich Parser Funktionalität in der Grundapplikation eingebaut, um den Text eigens definierter Sprachen zu parsen[10, S. 28].

Neben den externen domänenspezifischen Sprachen existieren auch die *internen domänenspezifischen Sprachen*. Bei dieser Form entspricht die syntaktische Repräsentation der Grundsprache und ist tatsächlich ein Teil dieser. Die Grundsprache wird hierfür nur für einen spezifischen Anwendungszweck stilisiert, bleibt aber grundsätzlich dieselbe. Dies führt dazu, dass ein Skript der internen domänenspezifischen Sprache valider Code seiner Grundsprache ist, welche aber nur teilweise verwendet wird und den Programmierstil dahingehend einschränkt und anpasst, dass der für domänenspezifische Sprachen typische flüssige und natürlichsprachige Stil resultiert[10, S. 28]. Damit erhält die interne domänenspezifische Sprache dann eine eigene Grammatik[10, S. 29] und erscheint wie eine neue Sprache, welche von der Grundsprache abzugrenzen ist[10, S. 28]. Die Einschränkung der Ausdrucksfähigkeit kommt bei der internen domänenspezifischen Sprache nicht von der eigentlichen Grundsprache, diese ist immer Turing-vollständig, sondern ausschließlich von der Art und Weise wie die Grundsprache im Rahmen der internen domänenspezifischen Sprache verwendet wird. Nämlich indem die domänenspezifische Sprache auf komplizierte Strukturen, wie beispielsweise Kontrollstrukturen, Schleifen und Ähnliches, verzichtet[10, S. 30].

Bezüglich der konkreten Ausführung von domänenspezifischen Sprachen gibt es ebenfalls zwei Konzepte die unterschiedliche Realisierungen bereithalten.

Eine Art der Ausführung ist es die Sprache *interpretieren* zu lassen. Das heißt, dass Text der domänenspezifischen Sprache geparst wird, der dann im Anschluss direkt zur Laufzeit mitsamt des Codes der restlichen Applikation ausgeführt wird. In diesem Fall liefert ein einziger Prozess ein Ergebnis.

Die andere Art der Ausführung ist die *Code Generation durch Kompilierung*. Dabei parst ein Compiler den Text der domänenspezifischen Sprache und liefert als Ergebnis generierten Code, welcher in einem anderen Prozess erst ausgeführt wird[10, S. 19]. Der Vorteil der Code Generation ist der, dass die Sprache in welchem der Parser geschrieben ist, eine andere sein kann, als die des generierten Codes, so wie es bei Compilern von Allzweck-Programmiersprachen ohnehin üblich ist. In diesem Fall, kann eine zweite Kompilierung des generierten Codes auch wegfallen, wenn es sich um eine rein dynamische Sprache handelt, die grundsätzlich nicht kompiliert wird. Andernfalls ist der Bau Prozess des Codes der domänenspezifischen Sprache bei Verwendung von Code Generation sehr viel komplizierter als die reine Interpretation des Codes, da immer zuerst Logik und Parser kompiliert werden müssen, der Parser anschließend ausgeführt werden muss um den Code zu generieren und der generierte Code zum Schluss nochmals kompiliert werden muss[10, S. 21].

Die Form in welcher eine domänenspezifische Sprache in den Kontext ihrer Applikation eingebaut werden kann, lässt sich in die *fragmentierte domänenspezifische Sprache* und die *eigenständige domänenspezifische Sprache* unterteilen[10, S. 32].

Die fragmentierte domänenspezifische Sprache zeichnet sich dadurch aus, dass nur Teile ihres Codes innerhalb anderen Codes in der Hauptsprache eingepackt sind. Die domänenspezifische Sprache bereichert den Code der Hauptsprache dann nur. Wesentlich für diese Art der Nutzung ist, dass es ohne Wissen über die Hauptsprache kaum möglich ist den Gesamtkontext zu verstehen, weshalb oft nur Personen mit Programmiererfahrung sie verwenden. Ein klassisches Beispiel für eine domänenspezifische Sprache, welche häufig als fragmentierte domänenspezifische Sprache verwendet wird, ist die Structured Query Language (SQL) beispielsweise im Kontext von Webanwendungen mit persistenter Datenspeicherung[10, S. 32].

Eigenständige domänenspezifische Sprachen sind dahingegen reine Code Teile der domänenspezifischen Sprache ohne jegliche Vorkommnisse der Hauptsprache[10, S. 32].

Gründe warum es überhaupt sinnvoll ist, eine domänenspezifische Sprache in ein Projekt zu involvieren, gibt es einige.

Einerseits verspricht die Verwendung von domänenspezifischen Sprachen eine Verbesserung der Produktivität in der Entwicklung, da die syntaktische Einfachheit der Sprache zu klarem Code führt, bei welchem Fehler schneller gefunden werden können und Systeme leichter manipulierbar sind. Dies führt insgesamt zu einer höheren Softwarequalität, da die Konzentration der Entwickler*innen auf der wesentlichen Geschäftslogik der Applikation liegt, komplexe inhaltliche Probleme also zuverlässiger und schneller gelöst werden können und Zeit bei Fehlersuche und -behebung gespart wird[10, S. 33].

Ein weiterer Grund, warum sich der Einbau einer domänenspezifischen Sprache in eine Applikation lohnen kann, ist die Tatsache, dass domänenspezifische Sprachen die Fähigkeit besitzen

Kommunikation anzureichern und als ein Kommunikationsmedium zwischen Softwareentwickler*innen und Domänenexpert*innen dienen können[10, S. 34]. In der Softwareentwicklung stellt sich die Kommunikation zwischen Kunden und Programmierer*innen häufig als die größte Herausforderung dar, da Softwareentwickler*innen Expertise in der Programmierung besitzen, es ihnen aber häufig an fachlichem Wissen zu der Domäne mangelt und Domänenexpert*innen andersherum häufig nur begrenztes oder gar nicht vorhandenes technisches Wissen mitbringen. Die domänenspezifischen Sprachen können aufgrund ihrer Einfachheit und des deklarativen Charakters viel einfacher von Personen gelesen und verstanden werden, die keinen Bezug und keine tiefgehenden Kenntnisse zum Programmieren besitzen. Firmen können domänenspezifische Sprachen daher auch nutzen Domänenexpert*innen direkt in den Entwicklungsprozess miteinzubeziehen und Teile der Applikationen, welche die domänenspezifische Sprache nutzen, selbst schreiben zu lassen, anstelle dass sie ihr Wissen an die Softwareentwickler*innen weitergeben, welche sich anschließend um die Umsetzung bemühen. Selbst wenn eine domänenspezifische Sprache nicht implementiert werden, kann ihre Definition doch wesentlich dazu beitragen gegenseitiges Verständnis zu verbessern, sodass alleine die reine Konzeption einer domänenspezifischen Sprache häufig schon einen Mehrwert liefert[10, S. 35].

Domänenspezifische Sprachen können darüber hinaus auch dazu genutzt werden Ausführungskontexte zu ändern. Ist es beispielsweise von Interesse die Ausführung der Logik eines Teils einer Applikation aus der Compilezeit in die Laufzeit zu verschieben, dann kann dies realisiert werden, indem Code in einer Extensible Markup Language Konfigurationsdatei in Form einer domänenspezifischen Sprache geschrieben ist, der dann erst zur Laufzeit und nicht schon zur Compilezeit ausgeführt wird.[10, S. 35].

Kritik wird an domänenspezifischen Sprachen ebenfalls geäußert.

Ein Problem welches aus der Verwendung von domänenspezifischen Sprachen resultiert ist, dass dann mehrere Sprachen erlernt werden, beziehungsweise beherrscht sein müssen[10, S. 37].

Ebenfalls problematisch kann es sein wenn Unternehmen eine große Anzahl ihrer Systeme wesentlich mit domänenspezifischen Sprachen bereichern, welche in anderen Unternehmen nicht verwendet werden. Für solche Unternehmen kann es schwierig werden neues Personal für Projekte zu finden, da das Wissen, welches sich Mitarbeiter*innen bezüglich der domänenspezifischen Sprachen aneignen häufig auf andere Unternehmen nicht übertragbar sind, wenn diese die selbst gebaute Sprache weder kennen noch nutzen[10, S. 38].

Des weiteren kann es für Unternehmen, welche domänenspezifische Sprachen betreiben zur Herausforderung werden diese Sprache regelmäßig den neusten technologischen Entwicklungen nachzukommen. Der Wartungsaufwand für solche Sprachen ist entsprechend hoch, wenn Unternehmen von neuesten Standards profitieren wollen. Gleichzeitig muss bei dieser Wartung und bei Erweiterung der domänenspezifischen Sprache darauf geachtet werden, dass die Sprache durch eine Vielzahl von Erweiterungen nicht zu mächtig wird, sodass sie letztendlich nicht mehr nur Teile einer Applikation realisiert oder plötzlich Turing-vollständig wird und äquivalent zu einer Allzweck-Programmiersprache ist. Hierbei müssen Unternehmen also stringent darauf achten die Sprache gegen die Probleme zu entwickeln, wegen welcher sie ursprünglich ins Leben gerufen wurde und Teile dessen was sie löst in andere Sprachen auszulagern, sollte die domänenspezifische Sprache zu groß werden[10, S. 38].

Unerfahrene Entwickler*innen können auch insofern zum Problem beim Bau einer domänenspezifischen Sprache werden, da schlecht gebaute Sprachen das Potential besitzen die anschließende Programmierung damit eher zu verkomplizieren und zu verschlechtern, anstatt sie einfacher verständlich und programmierbar zu gestalten[10, S. 38].

Insgesamt muss ohnehin abgewogen werden, ob sich der Bau einer domänenspezifische Sprache für ein Unternehmen lohnt. Sowohl Wartung als auch Bau einer solchen Sprache sind aufwendig und stellen häufig Herausforderungen insbesondere für Entwickler*innen dar, die wenig Erfahrung mit domänenspezifischen Sprachen haben. Gerade der Bau einer externen domänenspezifischen Sprache muss gut überlegt sein, da hier der zusätzliche Aufwand einen eigenen Parser zu schreiben mit beachtet werden muss[10, S. 37].

Zuletzt müssen Entwickler*innen bei Verwendung einer domänenspezifischen Sprache auch immer dazu bereit sein die Sprache und ihre Abstraktion der wesentlichen Inhalte der Domäne zu verändern und anzupassen, wenn sich aus der Domäne neue Tatsachen ergeben. Dies kann insbesondere für Domänenexpert*innen schwierig sein, sobald sie sich an eine Abstraktionsform gewöhnt haben und diese gut verstehen und mit ihr umgehen können. Ist diese Bereitschaft nicht vorhanden, kann es viel Zeit und Mühe kosten neue Tatsachen anzupassen um diese mit einer veralteten Abstraktionsform kompatibel zu gestalten. Entwickler*innen müssen sich also darüber im Klaren sein, dass eine domänenspezifische Sprache nie ein fertiges Produkt ist, sondern immer neu überarbeitet werden muss um seiner Domäne gerecht zu werden[10, S. 39].

Eine Programmiersprache, welche sich besonders gut dazu eignet domänenspezifische Sprachen zu entwickeln, ist Groovy. Groovy ist eine Open-Source-Sprache, welche für die Java Virtual Machine (JVM) gebaut wurde. Die Entwicklung an Groovy begann im Jahr 2003 und wurde seitdem von einer Vielzahl an Programmierer*innen entwickelt, wobei Groovy inzwischen ein Projekt der Apache Software Foundation ist. Groovy besitzt einige Kerneigenschaften, welche die Sprache definieren. Im Gegensatz zu zum stark statisch typisierten Java kann Groovy sowohl dynamisch als auch statisch typisiert genutzt werden, je nach Bedarf oder Vorlieben. Besonders die dynamische Typisierung zeichnet Groovy allerdings aus, die es erlaubt Typen von Variablen beliebig zu variieren. Eine weitere Kerneigenschaft ist die Meta-Programmierung, die es erlaubt Klassen zur Laufzeit zu erweitern und verändern. Eine Fähigkeit über welche beispielsweise Java ebenfalls nicht verfügt. Zuletzt ist die enge Zusammengehörigkeit zu Java allerdings auch eine dieser Kerneigenschaften der Sprache Groovy. Syntaktisch ist Groovy Java sehr ähnlich. Grundsätzlich ist nämlich jeder valide Java Code auch valider Groovy Code. Darüber hinaus besitzt Groovy aber eben noch weitere Eigenschaften, wie etwa auch die bereits genannten, welche sehr nützlich sein können[11, S. 5]. Die Eigenschaften, wegen welcher Groovy sich besonders als Entwicklungssprache für domänenspezifische Sprachen eignet, werden hier im weiteren Verlauf genannt und beschrieben.

Die Eigenschaften, welche Groovy, zu einer geeigneten Sprache zur Entwicklung einer domänenspezifischen Sprache machen, sind im Wesentlichen dazu in der Lage die Syntax von Groovy Code so zu manipulieren, dass aus dem Groovy Code der für domänenspezifische Sprachen notwendige deklarative Charakter entsteht und der Code insgesamt eine weitaus einfachere Struktur erhält, als klassischer imperativer Programmcode ihn besitzt. Groovy ist also besonders gut geeignet um interne domänenspezifische Sprachen zu realisieren, da tatsächlich keine eigene Syntax mitsamt Parser entwickelt wird, sondern valider Groovy Code ausgenutzt werden kann um mit

syntaktischen Tricks eine andere, einfachere Sprache zu generieren.

Beispielsweise sind viele Zeichen in Code syntaktisch notwendig, weil sie eine semantische Bedeutung besitzen, welche für Compiler oder Interpreter essentiell sind um die Semantik eines ganzen Ausdrucks erfassen zu können. Das Semikolon etwa gibt üblicherweise an, wann ein Ausdruck endet. Für Domänenexpert*innen, welche wenig Berührungspunkte mit Programmcode haben, sind gerade solche Zeichensetzungen irreführend, da diese in der natürlichen Sprache nicht so verwendet werden, wie das im Programmieren der Fall ist. Da das Ziel der domänenspezifischen Sprache ist eine Syntax zu entwickeln, welche gesprochenen Sprachen möglichst nahe kommt, ist es daher naheliegend solche Zeichensetzungen einfach entfallen zu lassen. Groovy ist im Gegensatz zu Java eine sehr flexible Sprache. Dort ist es an vielen Stellen erlaubt genau das zu tun. Semikola müssen in Groovy grundsätzlich nicht gesetzt werden[12, S. 677]. Klammern, welche Argumente bei einem Methodenaufruf umklammern, sind in Groovy optional[11, S. 35].

Neben entfallenen Zeichenketten, welche in der gesprochenen Sprache unüblich sind, profitiert Groovy von einer weiteren syntaktischen Eigenschaft, welcher beim Bau von domänenspezifischen Sprachen äußerst nützlich sein kann. Die sogenannten *benannten Argumente* sind Argumente, welche beim Aufruf einer Methode eine zusätzliche Information zu jedem übergebenen Argument erlaubt, welche dem Namen des Arguments entspricht. Groovy erlaubt zudem benannte Argumente, sowie unbenannte Argumente je nach Belieben in ihrem Auftreten und ihrer Reihenfolge zu mischen. Mit der Fähigkeit damit umzugehen, ist es möglich Methodenaufrufe wie den folgenden in Groovy zu tätigen[12, S. 693]:

```
foo argOne, keyOne: valueOne, argTwo, keyTwo: valueTwo
```
[12, S. 693]

Die aufgerufene Methode trägt den Bezeichner `foo`. Alles weitere stellen Argumente dar, wobei `keyOne` sowie `keyTwo` die Namen zweier benannter Argumente sind. Der gezeigte Methodenaufruf ist sehr abstrakt und soll ausschließlich verdeutlichen wie benannte Argumente syntaktisch in Groovy verwendet werden können. Hieraus lässt sich noch kein wesentlicher Vorteil ablesen, welcher die Entwicklung einer domänenspezifischen Sprachen bereichern kann. Deutlicher wird dies in dem folgenden Beispiel:

```
move right, by: 3.meters
```
[12, S. 693]

An diesem Beispiel ist gut zu erkennen welche Möglichkeiten sich mit den benannten Argumenten eröffnen. Grundsätzlich ist die Bezeichnung der Namen von Argumenten völlig frei. Daher ist es möglich von der üblichen Programmierkonvention Bezeichnungen zu wählen, die Informationen darüber bereithalten worum genau es sich bei der zu bezeichnenden Sache handelt, und stattdessen die Namen der Argumente so zu wählen, dass der geschriebene Code nahezu gesprochenen Sätzen natürlicher Sprache gleicht[12, S. 693].

Ebenfalls auffällig ist die Anwendung des Punktoperators eines Zahlenwerts auf eine Einheit, welche in einem Gesamtkontext beispielsweise als Enumeration kodiert sein könnte. In Java wäre es nötig gewesen eine Instanz eines Objekts als zweites Argument zu übergeben, bei welcher der Zahlenwert, sowie die konkrete Einheit, im Konstruktor des instanziierten Objekts übergeben würden. Auch in Groovy wäre es möglich das mit Verwendung einer Instanz realisieren und

würde wie folgt aussehen:

```
move right, by: new Distance(3, meters)
```
[12, S. 691]

Diese Schreibweise eines Methodenaufrufs gleicht syntaktisch viel mehr der klassischen Java Syntax, wenn auch Klammern um die Argumente herum fehlen. Gleichzeitig entspricht es nun aber wieder viel weniger einem Ausdruck, welcher in natürlicher Sprache verwendet würde um die gewünschte Semantik auszudrücken. Insbesondere das Schlüsselwort zur Instantiierung `new` kann für Personen ohne Programmierkenntnisse irreführend sein, weshalb es bei der Entwicklung der Syntax einer domänenspezifischen Sprache nützlich sein kann entsprechende Teile des Codes entfallen zu lassen.

Nun stellt sich allerdings die Frage welche Fähigkeiten Groovy besitzt, dass es in der Lage ist seine Syntax derart zu verändern. Wie bereits eingangs genannt ist eine der Haupteigenschaften von Groovy die sogenannte Meta-Programmierung, welche es erlaubt Klassen, Methoden und damit auch die Struktur von Attributen noch zur Laufzeit zu verändern. Dies gilt auch für Typen aller Art, entsprechend auch für numerische Typen[12, S. 692]. Mit der Anweisung

```
Number.metaClass.getMeters = {new Distance(delegate, Unit.meters)}
```

kann numerischen Typen die Einheit Meter in Form eines Werts der Enumeration `Unit` als Attribut hinzugefügt werden, sodass es daraufhin möglich ist mit dem Punktoperator von jedem numerischen Wert aus auf die Einheit zuzugreifen[12, S. 692]. Dieses Beispiel illustriert dabei nur eine Möglichkeit wie die Meta-Programmierung von Groovy genutzt werden kann um Syntax flüssiger zu gestalten. Gerade dieser Aspekt von Groovy bietet Entwickler*innen einen großen Raum an Gestaltungsmöglichkeiten die Syntax ihrer domänenspezifischen Sprache auf die Ansprüche der Domäne maßzuschneidern.

Eine weitere Möglichkeit weitaus leserlichen Code mithilfe von Groovy zu entwickeln, bieten die sogenannten *Delegates*[11, S. 35]. Ein Delegate ist eine implizite Variable einer Closure. Mit einem Delegate ist es möglich implizit Klassen und Closures zu referenzieren, wobei ein Delegate im Standardfall stets die nächste umschließende Klasse oder Closure referenziert. Das Delegate erlaubt es allerdings auch die zu referenzierende Klasse explizit zu bestimmen[11, S. 9].

Was damit erreicht werden kann, wird das folgende Beispiel illustrieren. Hierbei soll eine Klasse `SMS` angenommen werden, welche vier Methoden, `from(String fromNumber)`, `to(String toNumber)`, `body(String body)` und `send()`, implementiert. Diese Klasse soll das Versenden von SMS Kurznachrichten simulieren, wobei die Methoden `from`, `to` und `body` Versender- und Empfängernummern und den Inhalt der Kurznachricht setzen. Die Methode `send` implementiert das tatsächliche Versenden[11, S. 35]. In einer Java Anwendung würde für die Anwendung dieser Methoden zunächst eine Instanz der Klasse SMS erstellt werden, mit welcher dann über den Punktoperator auf die Methoden zugegriffen werden kann um diese auszuführen[11, S. 36]. Als ausgeschriebener Code würde dies folgendermaßen aussehen:

```
1  SMS msg = new SMS();
2  msg.from("01234");
3  msg.to("56789");
```

```
4  msg.body("Hello");
5  msg.send();
```

Wie zu erkennen ist, muss die Instanz `msg` einzeln jede Methode sequentiell hintereinander mit dem Punktoperator aufrufen. Dies ist nicht nur viel duplizierter Code, sondern außerdem nicht sonderlich flüssig zu lesen. Groovy bietet die Möglichkeit dies mithilfe des Delegates kürzer und leserlicher zu gestalten. Hierzu soll die folgende Methode angenommen werden:

```
1  def static send(Closure block) {
2          SMS msg = new SMS()
3          block.delegate = msg
4          block()
5          msg.send()
6  }
```

Diese statische Methode erzeugt zuerst eine Instanz der Klasse `SMS`, worin sich der Code zunächst also nicht wesentlich von dem Java Beispiel unterscheidet. Anschließend wird allerdings durch die Zeile `block.delegate = msg` die als Argument übergebene Closure an die erstellte Instanz delegiert. Das bedeutet, dass alle Methodenaufrufe innerhalb der Closure nun stets als Methodenaufrufe der Instanz `msg` interpretiert werden können. Damit ist es möglich eine Closure wie die folgende zu nutzen:

```
1  SMS.send {
2          from '01234'
3          to '56789'
4          body 'Hello'
6  }
```

Diese Closure wird als Argument `block` an die oben definierte statische Methode übergeben und innerhalb der Methode ausgeführt, nachdem sie an die Instanz `msg` delegiert wurde. Zuletzt wird die Methode `send` der `SMS` Klasse aufgerufen, sodass dann semantisch genau dasselbe passiert ist wie im Java Beispiel[11, S. 36]. Syntaktisch bietet die Closure der Java Variante gegenüber aber einen großen Vorteil. Innerhalb solcher Closures kann in Kombination mit den Delegates ein sehr deklarativer Charakter erreicht werden, der außerdem sehr leicht zu lesen ist und von jeder Person unabhängig ihrer Programmierkenntnisse verstanden werden kann.

Eine weitere Möglichkeit, die Groovy bietet um seine Syntax zu konfigurieren, ist das Überschreiben von Operatoren. Dabei müssen schlicht Methoden definiert werden, dessen Bezeichner jeweils der englische Name der entsprechenden Operation ist. Soll beispielsweise der Operator `+` überschrieben werden, muss lediglich eine Methode `plus` mit der gewünschten Funktionalität implementiert werden. Dies gilt für alle vordefinierten Operatoren, welche Groovy bietet, und führt dazu, dass auch Zeichen je nach Nutzen in domänenspezifische Sprachen eingebaut werden können[11, S. 37].

Zuletzt verfügt Groovy über eine weitere sehr nützliche Funktionalität im Bezug auf domänenspezifische Sprachen. Mit den Methoden `methodMissing(String name, args)` und `propertyMissing(String name)` hat Groovy eine Möglichkeit mit Methoden und Attributen umzugehen, welches es überhaupt nicht gibt, die also nie definiert wurden. Voraussetzung dafür ist, dass man solche Methoden und Attribute in der jeweils korrekten Syntax verwendet. Die Methoden `methodMissing` und `popertyMissing` werden getriggert, wenn Bezeichner im Code auftauchen, welche nicht definiert wurden. Die Funktionalität der Methoden kann innerhalb eine Klasse für Methoden und Attribute dieser Klasse implementiert werden, sodass ein beliebiger Umgang mit nicht definierten Methoden und Attributen realisiert werden kann[11, S. 38].

Um dies an einem Beispiel zu zeigen soll der folgende Code angenommen werden:

```
 1  class Chemistry {
 2       public static void exec(Closure block) {
 3            block.delegate = new Chemestry()
 4            block()
 5       }
 6       def propertyMissing(String name) {
 7            def comp = new Compound(name)
 8            (comp.elements.size() == 1 && comp.elements.values()[0] == 1) ?
 9             comp.elements.keySet()[0] : comp
10       }
11  }
```

Dieses Beispiel ist ein Ausschnitt aus einer domänenspezifischen Sprachen mithilfe welcher Atomgewichte von chemischen Verbindungen berechnet werden können. Die Klasse `Chemistry` nutzt hier zunächst wieder ein Delegate, mit welchem sie eine Closure an eine erstellte Instanz delegieren kann. Anschließend wird die `propertyMissing` Methode implementiert. In diesem Beispiel soll im Fall eines nicht definierten Attributs entweder ein einzelnes Element zurückgegeben werden, sofern die Verbindung aus nur einem Element besteht, oder andernfalls die gesamte Verbindung. Die statische Funktion `exec` kann dann folgendermaßen verwendet werden[11, S. 38 f.]:

```
1  Chemistry.exec {
2       def water = H2O
3       println water
4  }
```

Hierbei soll angenommen werden, dass das Attribut `H2O` zuvor nicht definiert wurde und entsprechend die `propertyMissing` Methode triggert. Wie zu erkennen ist, kann mithilfe der `propertyMissing` und `methodMissing` Methoden nahezu sämtliche beliebige Bezeichnungen in den Code einer domänenspezifischen Sprache inkludiert werden, ohne dass diese explizit definiert wurden. Dies gibt Entwickler*innen die Freiheit neue Bezeichnungen beliebig in einem flüssigen, möglichst natürlichsprachigen Code einzuführen und direkt mit sinnvollen Werten oder Implementierungen zu versehen.

5 Implementierung des Partitionsprädikats

5.1 Backend

5.1.1 Technologien

Das Back-end nutzt einige Frameworks als Hilfsmittel zur Implementierung.

Dazu gehören das Framework *Spring*, welches als *Web-Application* Framework genutzt wird um die Kommunikation mit den WorkforcePlus Clients zu erleichtern, das *Akka Actor* Framework, welches zur Vereinfachung von Threading und Synchronisierungsproblemen genutzt wird, das *Spock* Framework, welches zur Umsetzung von Unit Tests verwendet wird und außerdem, wie eingangs schon beschrieben, den *Apache Tomcat Server*, sowie *PostgreSQL* und *Oracle* als Relationale-Datenbank-Managementsysteme, die als Datenbankgrundlage für die deduktive Datenbank, auf welcher Roxx basiert, fungieren. (C. Briem, persönliche Kommunikation, 20. Dezember 2022)

5.1.2 Architektur des Partition Predicates im Backend

Die Architektur des Partition Predicate im Back-end umfasst alle in der Abbildung 6 aufgeführten Klassen.

Hierbei ist zu beachten, dass die Klassen entsprechend ihrer Zugehörigkeit zu den einzelnen Komponenten, in welchen das gesamte Back-end aufgebaut ist, farblich markiert sind.

Grundsätzlich ist das Back-end in einer Schichtenarchitektur organisiert und besitzt vier Komponenten. Diese sind die Ebene der DSL, auf welcher die genaue Syntax der Sprache Roxx definiert wird. Auf der DSL basiert die Ebene der *Runtime*, der Laufzeit, welche der Semantik des Roxx Programms entspricht. Hier werden logische Regeln organisiert, die etwa gültigen Datalog Regeln entsprechen können. Basierend auf diesen Regeln folgt darauf die Ebene des Compilers, der zwar kein richtiger Compiler im herkömmlichen Sinne ist, da dieser keinen Maschinencode erzeugt, allerdings ist diese Ebene dafür zuständig, dass Elemente wie Regeln oder sogenannte *Commands* durch entsprechende Klassen gebaut werden. Die letzte dieser vier Komponenten ist die Ebene der Datenbank, *Database*, auf welcher ein Prädikat nur noch aus Zeilen und Spalten besteht, welche die im Prädikat enthaltenen Tupel darstellen. Anhand dieser Zeilen und Spalten wird hier die Sortierlogik implementiert und der Roxx Code somit interpretiert.

Die Abbildung zeigt ebenfalls vier Klassen, die zwar zunächst keine direkten Auswirkungen auf die Funktionsweise zu Partition Predicates haben, allesamt aber dennoch Teile davon beinhalten, die im Laufe des Projekts geändert werden mussten.

Dazu zählen zwei Klassen der *Runtime* Ebene. Eine der beiden, *ToDatalogWithIdsVisitor*, generiert aus Roxx Regeln äquivalente Datalog Regeln. Die andere der beiden, *RuleUtils* stellt eine

Hilfsklasse dar, die Hilfsmethoden für Regeln über verschiedene Arten von Prädikaten hinweg bereitstellt.

Des Weiteren sind für die Änderungen am Partition Predicates ebenfalls die im Kapitel Grundlagen bereits erläuterten *Constraints* und ein vom Back-end erzeugter Graph relevant.

Der erzeugte Graph ist dazu in der Lage Abhängigkeiten von Prädikaten zueinander zu modellieren und darzustellen.

Ebenfalls erwähnenswert, ist die Tatsache, dass das unterste Element in der Architektur des Back-ends von Roxx immer ein Template ist, das zur Laufzeit des Back-ends mit entsprechenden Werten gefüllt wird und damit dann Roxx Code interpretieren kann.

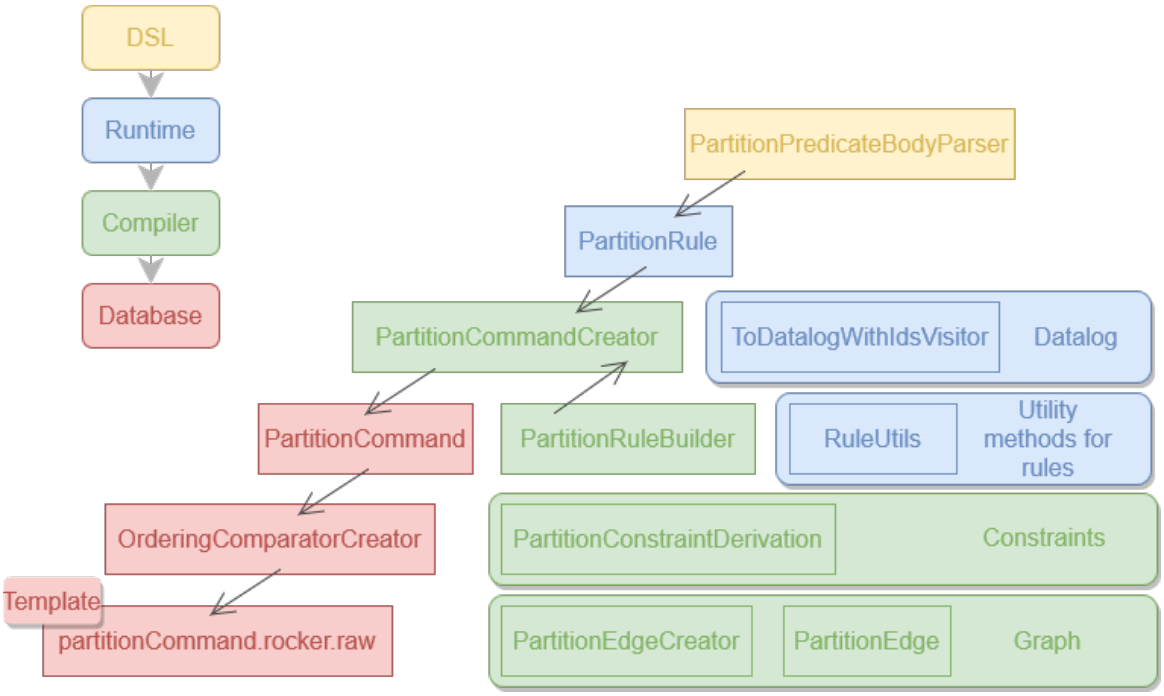

Abbildung 10: Architektur Partition Predicate (Eigene Abbildung)

5.2 Voraussetzungen der Mehrfachsortierung

5.3 Umsetzung

6 Funktionale Abhängigkeit

6.1 Kardinalitäten

Die Kardinalitäten oder auch *Constraints*, wie sie in *Roxx* genannt werden, werden durch sogenannte *Completeness Constraints* und *Uniqueness Constraints* definiert. Diese Kardinalitäten entsprechen Einschränkungen der Anzahl einzelner Elemente eines Prädikats, indem sie angeben wie viele Tupel gültiger Elemente mindestens oder maximal bei Auswertung eines Prädikats vorkommen dürfen. *Completeness Constraints* dienen hierbei dazu eine Untergrenze der Anzahl zu definieren, wohingegen *Uniqueness Constraints* dazu dienen eine Obergrenze der Anzahl zu definieren. In der Sprache *Roxx* werden die Kardinalitäten bei der Definition eines Extensionalen Prädikats mitangegeben. Bei Intensionalen Prädikaten wird neben weiteren Fakten auch die Kardinalität aus logischen Zusammenhängen heraus abgeleitet[4].

7 Fazit & Ausblick

7.1 Fazit

7.2 Ausblick

Anhang

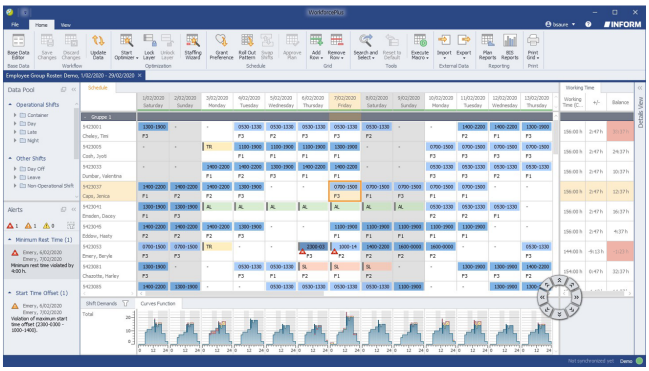

Abbildung 11: WorkforcePlus Windows Application Client[?]

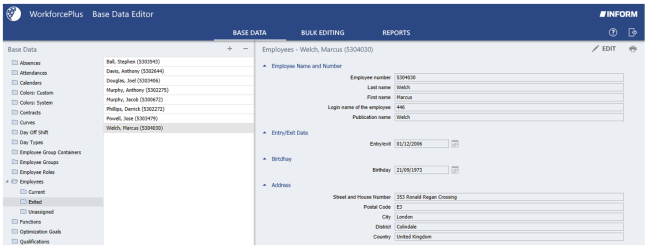

Abbildung 12: WorkforcePlus Base Data Editor[?]

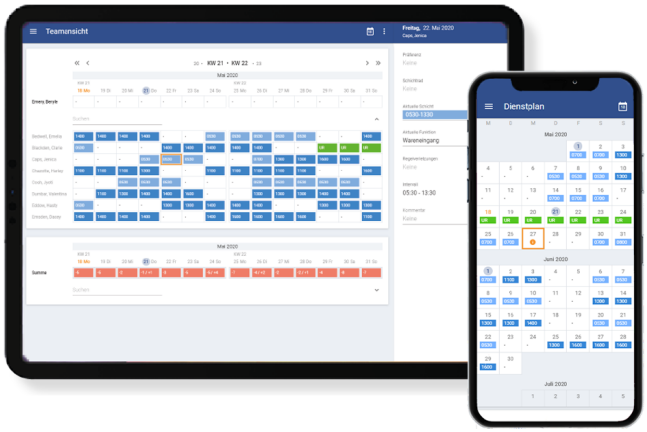

Abbildung 13: WorkforcePlus Employee Portal[?]

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou, "Datalog and Recursive Query Processing," *Foundations and Trends® in Databases*, vol. 5, no. 2, 2012.

[2] H. Kleine Büning and S. Schmitgen, *Prolog*, 2000.

[3] "Workforce Management - Komplexität einfach beherrschen," zuletzt geprüft am 22.12.2022. [Online]. Available: https://www.inform-software.de/workforcemanagement

[4] "Roxx - GB70 Workforce Management - INFORM Confluence," zuletzt geprüft am 22.12.2022. [Online]. Available: https://confluence.inform-software.com/display/WFM/Roxx

[5] "Sprint Review - 22 S15 - Declare It!" zuletzt geprüft am 23.12.2022. [Online]. Available: https://confluence.inform-software.com/pages/viewpage.action?pageId=117836769

[6] A. B. Cremers, U. Griefahn, and R. Hinze, *Deduktive Datenbanken*, 1994.

[7] W. Kohn and U. Tamm, *Mathematik für Wirtschaftsinformatiker*, 2019.

[8] V. Turau and C. Weyer, *Algorithmische Graphenthoerie*, 2015.

[9] S. Schäffler, *Die Kunst des Zählens*, 2019.

[10] M. Fowler and R. Parsons, *Domain-Specific Languages*, 2011.

[11] A. L. Davis, *Learning Groovy*, 2016.

[12] D. König and P. King, *Groovy in Action*, 2015.