THM | CAMPUS GIESSEN | MNI
Mathematik, Naturwissenschaften und Informatik
TECHNISCHE HOCHSCHULE MITTELHESSEN

# Erweiterung von Datalog um die Berechnung von Rängen

## Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)
vorgelegt dem
Fachbereich Mathematik, Naturwissenschaften und Informatik
der Technischen Hochschule Mittelhessen
von

**Madelaine Tatjana Hestermann**

im März 2023

Referent: Prof. Dr. Michael Elberfeld
Korreferent: Dr. Dimitri Bohlender

# Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

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

Gießen, 28. Februar 2023

# Abstract

Deklarative Programmiersprachen sind in der Lage durch Abstraktion der Kontroll- und Datenflüsse eines Programms auf komplexe technische Details zu verzichten. Dies kann Personen ohne Programmierkenntnisse dazu befähigen entsprechenden Programmcode besser zu verstehen. Zudem sind insbesondere logische Programmiersprachen gut geeignet Applikationen zu implementieren, deren Domäne aus Regeln, wie beispielsweise gesetzlichen Bestimmungen, bestehen. Aufgrund dessen wurde für die Applikation WorkforcePlus, eine Applikation zur Organisation der Personaleinsatzplanung in Unternehmen, der INFORM GmbH eine eigene domänenspezifische Sprache entwickelt. Die Sprache Roxx ist eine solche logische Sprache und soll die genannten Vorteile des Paradigmas implementieren.

Gegenstand dieser Bachelorarbeit ist die Erweiterung eines speziellen Prädikats der Sprache Roxx, dem Partitionsprädikat. Dabei wird der bisherige Stand der theoretischen Arbeit zur Programmiersprache Datalog, der Roxx als semantische Grundlage dient, skizziert. Anschließend wird das Partitionsprädikat im Kontext von Datalog definiert. Zudem wird der Begriff der domänenspezifischen Sprache erläutert und in Zusammenhang mit Roxx erklärt. Außerdem wird eine Implementierung des Partitionsprädikats vorgestellt. Im Fokus liegen dabei die Erweiterung um die Mehrfachsortierung des Partitionsprädikats und eine neu implementierte Partitionsfunktion, welche im Rahmen dieser Bachelorarbeit entwickelt wurden. Abschließend folgt ein Ausblick inwiefern Roxx zusätzlich erweitert werden kann.

# Inhaltsverzeichnis

# 1 Einleitung

Die vorliegende Arbeit handelt von der Erweiterung der logisch deklarativen Programmiersprache Datalog um ein Prädikat, welches in der Lage ist Ränge für Tupel anhand von Sortierungen zu errechnen und zweier Partitionsfunktionen dieses Prädikats, die unterschiedliche Sortierweisen implementieren.

## 1.1 Motivation für diese Arbeit

Green et al. berichten in ihrem Überblicksartikel [1], dass bereits in den 80er und den frühen 90er Jahren im Bereich der Datenbankmanagementsysteme großes Interesse an logisch deklarativen Programmierkonzepten wie etwa der Programmiersprache Datalog entstand. 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. Ein aktuelles Beispiel für die Verwendung von Datalog ist die Sprache QL, eine objektorientierte Sprache, welche Datenabfragen auf relationale Daten tätigt [2, S. 1]. QL bietet dabei objektorientierte Elemente wie Klassen und Methoden, die in logische Terme interpretiert und schließlich zu Datalog kompiliert werden.

Der Mehrwert, welchen deklaratives Programmieren bietet, liegt in der Abstrahierung des Kontroll- und Datenflusses eines Programms. Hierbei wird deskriptiv die Aufgabe einer Anwendung beschrieben [1]. Anders als bei imperativen Programmiersprachen, bei welchen detailliert ausgeführt werden muss, wie ein Programmablauf zu einem gewünschten Ergebnis führen kann, muss dies in deklarativen Programmiersprachen nicht beachtet werden [3, S. 13].

Wie Green et al. [1] erläutern, ist dies besonders nützlich bei Beweisführungen, die allgemeingültige Invarianten 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 Komplexität zu konzentrieren, welche aus der für die Applikation essentiellen Bestandteile resultiert. Zudem erleichtert die hohe Abstraktionsebene Codeanalysen, welche genutzt werden können um Optimierungsmöglichkeiten und Sicherheitsmaßnahmen zu entwickeln und in Applikationen zu integrieren. Zuletzt ist es ebenfalls üblich insbesondere die Programmiersprache Datalog zur Abstraktion relationaler Strukturen und Datenabfragen zu verwenden.

Aus diesen Vorteilen heraus ergab sich für den Partner der Projektphase dieser Bachelorarbeit, den Geschäftsbereich 70 der INFORM GmbH, die Motivation für ihre Anwendung WorkforcePlus die deklarativ logische Sprache *Roxx* zu entwickeln. Roxx ist ein Dialekt der Programmiersprache Datalog und ähnelt dieser semantisch sehr. Der Geschäftsbereich entwickelt

die Software WorkforcePlus im Bereich des Workforce Managements. Workforce Management ist ein Unterbereich des Personalwesens, das die Einteilung des richtigen Personals, mit der richtigen Qualifikation, zum richtigen Zeitpunkt, und am richtigen Ort behandelt [4]. WorkforcePlus ist eine Applikation zur optimierten und automatisierten Personaleinsatzplanung. Sie organisiert große Datenmengen effizient, weswegen der Einsatz einer Sprache mit hohem Abstraktionspotential, wie etwa Roxx, passend ist.

## 1.2 Problemstellung dieser Arbeit

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 Partitionsfunktion zur Sortierung von Daten auf Partitionen anzuwenden. Als Ausgangsbasis dient das Partitionsprädikat, welches allerdings nur nach einem einzigen gegebenen Attribut sortieren kann und nur über eine bestimmte Partitionsfunktion verfügt. Wird eine Sortierung auf eine Menge an Datentupeln angewandt, welches für das Sortierattribut dieselben Werte aufweisen, hat das die Konsequenz, dass die Tupel denselben Rang erhalten [5]. Die Sortierung ist dann nicht eindeutig.

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.

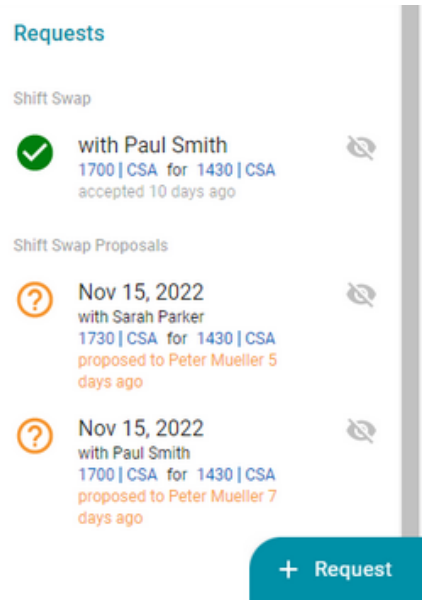


Abbildung 1: Mitarbeiter Requests der Applikation WorkforcePlus Employee Portal [6]

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* [7] zu ergänzen. Mithilfe sogenannter Makros [8] war es möglich die Mehrfachsortierung aus einer Kombination mehrerer *Roxx*-Konstrukte und imperativen Sprachkonstrukten, welche Funktionalität zur Kompilierzeit beigetragen haben, erfolgreich zu simulieren und den gewünschten *Roxx*-Code zu erzeugen, was allerdings unverhältnismäßig komplex sowie ineffizient war. Des weiteren konnte die Berechnung des Rangs mithilfe des Makros bei großen Tupelmengen und vielen Sortiervariablen zu einem Ganzzahlenüberlauf führen, sodass die Nutzung hinsichtlich der Datenmenge eingeschränkt war.

## 1.3 Beiträge dieser Arbeit

Im Rahmen dieser Bachelorarbeit wird das Partitionsprädikat, welches bislang in keiner wissenschaftlichen Arbeit behandelt wurde, im Kontext der logischen Programmiersprache Datalog eingeführt. Hierzu werden Syntax und Semantik des Partitionsprädikats definiert und erläutert. Zudem werden zwei Partitionsfunktionen des Partitionsprädikats eingeführt, welche unterschiedliche Arten der Rangberechnung implementieren. Des weiteren werden Probleme bei der Evaluierung von Datalogprogrammen, welche die Partition rekursiv nutzen, beschrieben. Hierzu wird die Stratifikation der Partition eingeführt, welche als Lösungsansatz für die aufgezeigten Probleme gilt. Zum Schluss wird die konkrete Implementierung des Partitionsprädikats im Rahmen einer Applikation vorgestellt.

## 1.4 Aufbau dieser Arbeit

In diesem Abschnitt wird die Struktur dieser Arbeit dargestellt. Zuerst wird der theoretische Hintergrund zur Programmiersprache Datalog im Kapitel 2 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 Prädikate 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 gelöst werden können.

Kapitel 3 stellt das Partitionsprädikat zunächst im Datalog Kontext vor. 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.

In Kapitel 4 wird ein Überblick darüber gegeben, was domänenspezifische Sprachen sind, welche Eigenschaften sie haben, wieso es sich lohnt eine solche Sprache zu verwenden und welche Probleme sie birgt. 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 in Kapitel 5 das Partitionsprädikat im Kontext der domänenspezifischen proprietären Sprache Roxx beschrieben. Hierzu werden zuerst Syntax und daraufhin die Semantik des Prädikats erklärt. Dann wird die konkrete Implementierung des Partitionsprädikats dargelegt. Dazu wird zuerst die domänenspezifische Sprache Roxx genauer vorgestellt. Anschließend wird die Architektur des Backends, in welchem die Sprache Roxx implementiert ist, sowie dort verwendete Technologien skizziert und zuletzt wird die konkrete Umsetzung der Mehrfachsortierung und der neu eingeführten Partitionsfunktion `uniqueRank` beschrieben.

Abgeschlossen wird diese Bachelorarbeit in Kapitel 6 mit einer Zusammenfassung über das Geschriebene, sowie einem Ausblick um welche Funktionalität das Partitionsprädikat künftig außerdem erweitert werden kann.

# 2 Datalog

Dieses Kapitel behandelt Grundlagen, die es bedarf, um einerseits Problemstellung, Vorgehensweise und Lösung des Projekts nachvollziehen zu können. Sämtliche Beispiele, welche in diesem Kapitel genannt werden, sind angelehnt an die Beispiele einer Publikation von Green et al. [1].

## 2.1 Deduktive Datenbanken und Prädikate

Deduktive Datenbanken bilden die Grundlage von Datalog. Sie sind in der Lage relationale Datenbanken um Regeln zu erweitern, sodass es möglich ist, durch einige solcher Regeln eine große Faktenmenge abzuleiten und darzustellen [9, 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 [9, S. 17].

Prädikate sind im Kontext der Logik Gebilde, die durch Eigenschaften von Elementen die Zugehörigkeit zu einer Menge definieren und aufgrund dieser Wahrheitswerte auf die Elemente abbilden [10, S. 32]. Letztlich sind Prädikate also Funktionen mit einer booleschen Zielmenge. Laut Green et al. [1] verfügt Datalog 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) und gelten als Fakten. *intensionale Prädikate* hingegen entsprechen hergeleiteten Daten, die sich aus ausgewerteten Regeln ergeben. Sie werden durch extensionale Prädikate impliziert.

## 2.2 Syntax von Datalog

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.

In der Form eines Prädikats würde die Regel folgendermaßen notiert werden:

$$\forall X.A(X) \leftarrow B_1(X) \wedge B_2(X) \wedge ... \wedge B_n(X)$$

Wie von Green et al. [1] erläutert, ist ein *Term* 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 aufgeführten Regel entsprechen beispielsweise sowohl die Bezeichner $B_1, B_2, ..., B_n$, als auch $A$ Atomen. Atome, welche nur aus konstanten Argumenten bestehen, werden als *Grundatome* bezeichnet. Eine *Datenbankinstanz* ist eine Menge von Grundatomen, wobei die *extensionale Datenbankinstanz* die Instanz bezeichnet, welche ausschließlich extensionale Prädikate als Atome beinhaltet. Entsprechend beinhaltet die *intensionale Datenbankinstanz* ausschließlich intenrsionale Prädikate. Die Menge aller in der Datenbank vorkommenden Konstanten wird als *aktive Domäne* einer Datenbankinstanz bezeichnet.

**Beispiel 2.2.1** Anhand der folgenden Datalog-Regeln wird die eingeführte Terminologie konkret erläutert.

```
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 ist. Das Prädikat `parent` soll in diesem Beispiel ein extensionales Prädikat sein. Beide Atome besitzen mit den Termen `X`, `Y` und `Z` in unterschiedlicher Kombination jeweils zwei Argumente. Die extensionale Datenbankinstanz 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:

Die Menge der Personen $\{Rose, Hugo, Ronald, Arthur, Septimus\}$ seien die Elemente der aktiven Domäne. Das extensionale Prädikat `parent` sei ein Prädikat, welches die Relation zwischen den Elementen der Personen ausdrückt, sodass eine Person als Elternteil einer anderen gilt. Dann ergibt sich daraus die folgende Beispieltabelle 1:

| Child | Parent |
|---|---|
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |

Tabelle 1: *parent* (initiale Tupel)

Dabei stellt die Variable `Y` stets den Elternteil der Variable `X` dar. Die Semantik der Regeln entspricht in unserem Beispiel der Bedeutung des Bezeichners. Sie sind in der Lage rekursiv den Vorfahren einer gegebenen Person zu charakterisieren, denn das Prädikat `ancestor` gilt dann als logisch wahr, wenn der Wert der Variable `Y` ein Vorfahre des Werts der Variable `X` ist. Dies geschieht indem die erste Regel zunächst über das Prädikat `parent` definiert wird, wodurch ein Paar `(X,Y)` dann als Element der `ancestor` Relation gilt, wenn es auch ein Element der `parent` Relation ist. 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 extensionale Datenbankinstanz die Folgende sein 2:

| Descendant | Ancestor |
|---|---|
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |
| Rose | Arthur |
| Hugo | Arthur |
| Ronald | Septimus |
| Rose | Septimus |
| Hugo | Septimus |

Tabelle 2: *ancestor*

Um die Auswertung von Datalog-Regeln sicher zu gestalten, gibt es außerdem Beschränkungen zur Definition von Regeln. Folgende Definition wird hierzu festgelegt:

**Definition 2.2.1** *Ein Datalogprogramm ist dann sicher, wenn jede Variable, welche im Kopf einer Regel auftaucht, auch im Körper der Regel erscheint [1].*

Beispiele für sichere und unsichere Regeln zeigt das folgende Beispiel:

**Beispiel 2.2.2** Die Regel

```
a(X,Y) :- b(X), c(Y,Z)
```

ist eine sichere Regel, da alle Variablen im Kopf auch im Körper auftauchen.

Die Regel

```
a(X) :- b(Y)
```

ist keine sichere Regel, da die Variable X im Kopf der Regel, nicht aber in ihrem Körper auftaucht.

## 2.3 Semantik von Datalog

In diesem Abschnitt wird erläutert, wie Regeln berechnet und ausgewertet werden können. Die Auswertung einfacher Regeln geschieht durch Anwendung logischer Operationen. 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 1 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 nach Green et al. [1] die *Fixpunkt-Semantik*. Sie basiert auf dem *(unmittelbaren) Konsequenzoperator* [9, S. 103] $T_P$, wobei $P$ ein Datalogprogramm ist, welches eine Datenbankinstanz für den Operator spezifiziert. Der Konsequenzoperator ist dazu in der Lage das unmittelbare Resultat der einmaligen Anwendung

von Regeln eines Datalogprogramms zu bestimmen. Die Fixpunkt-Semantik ist ein Konzept, bei welchen der Konsequenzoperator solange wiederholt auf ein Datalogprogramm angewandt wird, bis ein Fixpunkt erreicht wird. Die Funktionsweise der Fixpunkt-Semantik wird im Folgenden anhand eines Beispiels illustriert:

**Beispiel 2.3.1** Sei $I$ eine Datenbankinstanz, die dem Beispiel aus dem vorigen Abschnitt gleicht, so ergeben sich für sie die folgenden Tabellen 3:

$I =$

| *parent* | |
|---|---|
| Child | Parent |
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |

,

| *ancestor* | |
|---|---|
| Descendant | Ancestor |
| $\emptyset$ | $\emptyset$ |

Tabelle 3: Initialbelegung Datenbankinstanz $I$

Wird der Konsequenzoperator $T_P$ einmal auf die Datenbankinstanz angewandt, ergibt sich folgende Belegung für die Tabellen:

$T_P(I) =$

| *parent* | |
|---|---|
| Child | Parent |
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |

,

| *ancestor* | |
|---|---|
| Descendant | Ancestor |
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |

Tabelle 4: Datenbankinstanz $T_P(I)$

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 Konsequenzoperator immer nur unmittelbare Ergebnisse errechnet. Das heißt jede Regel wird einmal durchlaufen. Auch rekursive Regeln werden je Schritt nur ein einziges Mal ausgewertet. 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 das erste Zwischenergebnis für $T_P$. Der gewünschte Fixpunkt lässt sich konstruktiv dadurch ermitteln, dass der Konsequenzoperator wiederholt auf die Ergebnistupel des vorherigen Durchgangs angewandt wird bis der Fixpunkt erreicht ist [1].

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

$T_P(I) =$

| *parent* | |
|---|---|
| Child | Parent |
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |

,

| *ancestor* | |
|---|---|
| Descendant | Ancestor |
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |

Tabelle 5: Datenbankinstanz $T_P(I)$

$T_P^2(I) =$

| *parent* | |
|---|---|
| Child | Parent |
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |

,

| *ancestor* | |
|---|---|
| Descendant | Ancestor |
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |
| Rose | Arthur |
| Hugo | Arthur |
| Ronald | Septimus |

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

$T_P^3(I) =$

| *parent* | |
|---|---|
| Child | Parent |
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |

,

| *ancestor* | |
|---|---|
| Descendant | Ancestor |
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |
| Rose | Arthur |
| Hugo | Arthur |
| Ronald | Septimus |
| Rose | Septimus |
| Hugo | Septimus |

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

$T_P^4(I) =$

| *parent* | |
|---|---|
| Child | Parent |
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |

,

| *ancestor* | |
|---|---|
| Descendant | Ancestor |
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |
| Rose | Arthur |
| Hugo | Arthur |
| Ronald | Septimus |
| Rose | Septimus |
| Hugo | Septimus |

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

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. Ein Fixpunkt wurde dementsprechend nach drei Iterationen gefunden. Außerdem ist zu beachten, dass die Ergebnistabelle `ancestor` wie zu erwarten den Ergebnissen der Tabelle 2 entspricht. Nämlich der korrekten Relation zwischen den Personen, welche charakterisiert, welche Person ein Vor- oder Nachfahre einer anderen Person ist.

## 2.4 Syntax und Semantik der Negation

Die Negation ist eine Operation, welche Datalog um 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.

Syntaktisch wollen wir für die Negation der Regeln das Wort *not* einführen. Grundsätzlich lassen sich Datalog-Regeln, welche Negationen in ihren Körpern beinhalten, ebenso wie Regeln ohne Negation mithilfe der Fixpunkt-Semantik auswerten. Das heißt, auf sie kann der Kon-

sequenzoperator 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.

**Beispiel 2.4.1** Das erste Beispielprogramm zeigt zwei Regeln `p(x)` und `q(x)`, welche einander in einer Rekursion negiert bedingen. Da beide Regeln kein extensionales Prädikat beinhalten, besitzen sie keine weiteren Informationen über die Wertemenge der Variable X, sodass nicht festgelegt ist, woraus `p(x)` und `q(x)` eigentlich bestehen. Bei einem naiven Versuch diese Regeln anhand der 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)
```

**Beispiel 2.4.2** Das zweite Beispielprogramm zeigt eine einzige Regel, anhand welcher das Problem des Alternierens deutlich wird. Das heißt, dass die Ergebnismenge des Prädikats immer zwischen zwei verschiedenen Mengen hin und her springt, aber nie zu einem Fixpunkt kommt. 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 Problem aus Beispiel 2.4.1 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 Fixpunkt-Semantik auszuwerten. Hierfür soll angenommen werden, dass das extensionale Prädikat `e(x)` drei numerische Werte $1, 2, 3$ enthält. Wird der Konsequenzoperator 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. Formal ergibt sich $T_P^1(I)(q) = T_P^0(I)(e) = I(e) = \{1, 2, 3\}$. 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 leer ist und das Komplement der leeren Menge alle beliebigen Werte außer der leeren Menge selbst enthält, sodass von den Elementen aus `e(x)` nichts abgezogen wird. Grundsätzlich wird also $T_P^1(I)(q) = T_P^0(I)(e) \setminus T_P^0(I)(q)$ angewandt. Da allerdings $T_P^0(I)(q) = \emptyset$ ist, sind die Resultate von $T_P^1(I)(q) = T_P^0(I)(e)$ und $T_P^1(I)(q) = T_P^0(I)(e) \setminus T_P^0(I)(q)$ zu diesem Zeitpunkt der Auswertung äquivalent.

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: $I(q) = \{1, 2, 3\}$. 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. Formal notiert entspricht das dem Folgendem: $T_P^2(I)(q) = T_P^1(I)(e) \setminus T_P^1(I)(q) = I(e) \setminus I(q) = \{1, 2, 3\} \setminus \{1, 2, 3\} = \emptyset$.

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 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. Die Zwischenergebnisse bis Schritt 5 kann in Tabelle 9 nachverfolgt werden.

$$T_P(I) = \begin{array}{|c|}\hline \text{q(x)} \\ \hline 1 \\ 2 \\ 3 \\ \hline \end{array}, T_P^2(I) = \begin{array}{|c|}\hline \text{q(x)} \\ \hline \emptyset \\ \hline \end{array}, T_P^3(I) = \begin{array}{|c|}\hline \text{q(x)} \\ \hline 1 \\ 2 \\ 3 \\ \hline \end{array}, T_P^4(I) = \begin{array}{|c|}\hline \text{q(x)} \\ \hline \emptyset \\ \hline \end{array}, T_P^5(I) = \begin{array}{|c|}\hline \text{q(x)} \\ \hline 1 \\ 2 \\ 3 \\ \hline \end{array}$$

Tabelle 9: Alternierendes Ergebnis

Nach Definition der 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 Bedingungen einführen, die eine sinnvolle Ausführung rekursiv definierter Negation in Datalogprogramm garantieren.

**Definition 2.4.1** *Im semipositiven Datalog dürfen nur extensionale Prädikate negiert werden. Intensionale Prädikate sind von der Negation ausgeschlossen. Außerdem muss jede Variable im Körper einer Regel des semipositiven Datalog in mindestens einem nicht negierten Atom auftauchen.*

Die Bedingung aus Definition 2.4.1 [1], dass nur extensionale Prädikate bedingt werden dürfen, verhindert ungewollte Effekte wie das Alternieren, das im Beispiel 2.4.2 demonstriert wird. 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. Die Bedingung, dass jede Variable im Körper einer Regel in einem oder mehreren nicht negierten Atomen der Regel

auftauchen muss, löst das Problem, welches mit Beispiel 2.4.1 skizziert wird. 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 Datalogprogramme 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. Im Folgenden entsprechen alle Beispiele dem semipositiven Datalog.

Semantisch entspricht der Negationsoperator, mit Rücksicht auf die oben genannten Bedingungen, dem Komplement eines Prädikats [1].

> **Beispiel 2.4.3** Um dies am Eingangsbeispiel 2.3.1 zu erläutern, sollen die Regeln daraus hier erweitert werden. Wir führen eine weitere Regeln zu den bereits bestehenden ein, so dass wir nun folgendes Datalogprogramm 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 fällt ins Fragment des semipositiven Datalogs, da es Negation anwendet und es allen Bedingungen, welche das semipositive Datalog entsprechend der Definition 2.4.1 definiert, erfüllt. Das Prädikat im Programm, welches negiert wird, ist das Prädikat `parent`. Da dieses Prädikat ein extensionales ist, entspricht dies insofern dem semipositiven Datalog. Zudem kommen beide Variablen `X` und `Y` sowohl im negierten Prädikat `parent` als auch im nicht negierten Prädikat `ancestor` vor. Auch dies entspricht den Bedingungen des semipositiven Datalog.

## 2.5 Stratifizierte Negation

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].

Um die stratifizierte Negation realisieren zu können, legen wir entsprechend der Definition von Abiteboul et al. [11, S. 378 f.] Folgendes fest:

**Definition 2.5.1** *Die Stratifikation eines Datalogprogramms $P$ ist eine Sequenz von Datalogprogrammen der Form $P^1, ..., P^n$.*

**Definition 2.5.2** *Sei $P^1, ..., P^n$ eine Stratifikation für ein Datalogprogramm $P$, dann wird jedes $P^i$ als Stratum dieser Stratifikation bezeichnet.*

Nach der Berechnung eines jeden Stratums werden dessen intensionalen Prädikate für das nächste Stratum in Extensionale Prädikate umgewandelt. Des weiteren gelten zwei Bedingungen [1] für die Stratifikation:

**Bedingung 2.5.1** *Gibt es ein Prädikat ohne Negation A :- ..., B, ... in einem Datalogprogramm P, welche in Stratum $P_i$ ist, während B in Stratum $P_j$ ist, dann ist $i \geq j$.*

**Bedingung 2.5.2** *Gibt es eine Prädikat mit Negation A :- ..., not B, ... in einem Datalogprogramm P, welche in Stratum $P_i$ ist, während B in Stratum $P_j$ ist, dann ist $i > j$.*

Bedingung 2.5.1 ermöglicht es Regeln, welche keine Negation enthalten zunächst an beliebiger Stelle auszuwerten, wohingegen Bedingung 2.5.2 garantiert, dass Regeln, die Negation beinhalten in jedem Fall nach allen Regeln ausgewertet wird, welche wiederum die negierte Regel beeinflussen.Ein Datalogprogramm, welches Stratifikation erlaubt wird als *stratifizierbar* bezeichnet.

**Beispiel 2.5.1** Auch die stratifizierte Negation illustrieren wir anhand eines Beispiels. Hierzu erweitern wir das ursprüngliche Beispiel zur Berechnung von Vorfahren um drei weitere Regeln und erhalten folgendes Datalogprogramm:

```
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)

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

Dieses Programm kann zwei Personen ermitteln zwischen denen 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 Auswertung stratifizierter Negation sieht gemäß Green et al. [1] vor, dass zunächst eine Stratifikation des gegebenen Programms ermittelt wird, um anschließend die Strata nacheinander entsprechend des Semipositiven Datalog auszuwerten, wobei alle Intensionalen Prädikate eines vorangehenden Stratums als Extensionale Prädikate des darauffolgenden Stratums erachtet werden.

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.

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 $\neg$ versehen sind. Ist dies der Fall, ist das Programm nicht stratifizierbar. Andernfalls können für das Programm Stratifikationen gefunden werden.

Der Präzedenzgraph kann außerdem dazu genutzt werden, eine Reihenfolge der Stratifikationen festzulegen. Abgesehen von Bedingung 2.5.2, 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 Graphen. Ein gerichteter Graph heißt stark zusammenhängend, wenn für jedes Paar $v_i$, $v_j$ einer Eckenmenge $V$ ein Weg von $v_i$ nach $v_j$, als auch umgekehrt ein Weg von $v_j$ nach $v_i$ existiert. [12, S. 99]. Um die starken Zusammenhangskomponenten eines Graphen zu bestimmen, muss ein Graph in Untergraphen aufgeteilt werden, sodass die Untergraphen stark zusammenhängend sind [12, S. 100]. Ein Beispiel für die Zusammenhangskomponenten eines Graphen stellt folgende Abbildung dar:

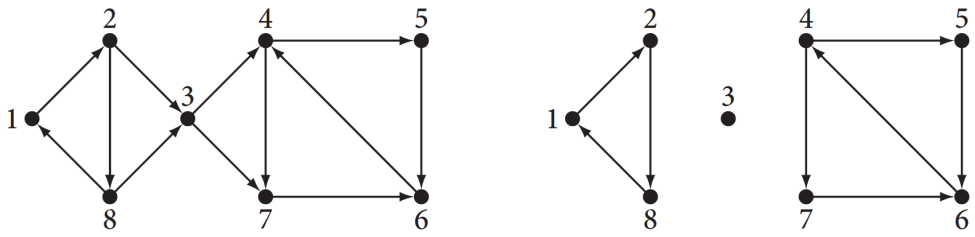


Abbildung 2: Gerichteter Graph mit seinen starken Zusammenhangskomponenten [12, S. 100]

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 Datalogprogramm $P$, 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 [12, S. 98].

Um dies an einem Beispiel zu skizzieren, soll Beispiel 2.5.1 dienen. 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. Eine beispielhafte Stratifizierung wird im Folgenden errechnet.

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. Von jedem Knoten im Graphen aus führt ein Knoten zu einem anderen Knoten hin, aber nicht mehr zurück. Daher sind alle Prädikate `person`, `unrelated` und `ancestor` einzelne starke Zusammenhangskomponenten. Nach der topologischen Sortierung müssen die Prädikate `ancestor` und `person` vor dem Prädikat `unrelated` ausgewertet werden. Diese Notwendigkeit resultiert aus den Kanten, die von den Prädikaten `ancestor` und `person` zum Prädikat `unrelated` führt. Außerdem existieren aber keine weitere Abhängigkeiten, sodass die übrige Sortierung beliebig erfolgen kann.

Wird entschieden, dass zunächst die Regel `ancestor` und anschließend die Regel `person` ausgewertet werden soll, wäre eine mögliche Stratifikation des Programms die Folgende: (1) `ancestor`, (2) `person`, (3) `unrelated`. 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.

**Beispiel 2.5.2** Für eine beispielhafte Berechnung verwenden wir wieder die Daten aus Tabelle 1 verwendet. Da diese allerdings nur Tupel mit Personen beinhalten, welche alle in einem Verwandtschaftsverhältnis zueinander stehen, wird der Tabelle 1 ein weiteres Tupel (*Edward*, *Dora*) hinzugefügt, sodass diese nun aussieht wie folgt:

$T_P(I) =$

| Child | Parent |
|---|---|
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |
| Edward | Dora |

Tabelle 10: *parent*

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:

$T_P(I) =$

| Descendant | Ancestor |
|---|---|
| Rose | Ronald |
| Hugo | Ronald |
| Ronald | Arthur |
| Arthur | Septimus |
| Rose | Arthur |
| Hugo | Arthur |
| Ronald | Septimus |
| Rose | Septimus |
| Hugo | Septimus |
| Edward | Dora |

Tabelle 11: *ancestor*

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 12.

$T_P(I) =$

| Person |
|---|
| Rose |
| Hugo |
| Ronald |
| Arthur |
| Edward |
| Septimus |
| Dora |

Tabelle 12: *person*

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 `unrelated` errechnet werden kann. Das Ergebnis welches hieraus resultiert, ist in der folgenden Tabelle festgehalten:

$T_P(I) =$

| Person X | Person Y |
|---|---|
| Rose | Dora |
| Hugo | Dora |
| Ronald | Dora |
| Ronald | Ronald |
| Arthur | Dora |
| Arthur | Arthur |
| Edward | Ronald |
| Edward | Arthur |
| Edward | Septimus |

Tabelle 13: *unrelated*

## 2.6 Syntax und Semantik der Aggregation

Wie Green et al. [1] beschreiben, fordern viele Anwendungen neben der Berechnung von Negation und Standard Datalog-Regeln, 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. 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 erklärt.

Die *Aggregationsfunktion* ist eine Funktion, die Mengen von Werten der entsprechenden Domäne auf Werte der Domäne abbildet . In Datenbanken üblicherweise verwendete Aggregationsfunktionen sind beispielsweise die `count`, `sum`, `max`, `min` und `average` Funktionen. Entsprechend der Definitionen von Green et al. [1] ist ein *aggregierter Term* ein Ausdruck $f < t_1, ..., t_k >$ wobei $f$ eine Aggregationsfunktion mit $k$ Argumenten ist. Variablen die in nicht aggregierten Termen im Kopf einer Datalog-Regel mit Aggregation auftreten, werden als *Gruppierungsvariablen* bezeichnet. Diese Gruppierungsvariablen definieren, anders als bei Sprachen wie SQL wo dies explizit passiert, implizit nach welchen Variablen aggregiert wird.

Im Gegensatz zu Datalog-Regeln, wie sie in vorigen Abschnitten erläutert werden, ändert sich die Syntax bei der Aggregation insofern, als dass auch aggregierte Terme im Kopf einer Regel vorkommen dürfen.

**Beispiel 2.6.1** Das folgende Beispielprogramm stellt ein Datalogprogramm 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)
```

Dieses Beispielprogramm wird zum am Ende diesen Abschnitts nochmals anhand einer Beispielauswertung genauer erklärt. Die Definition eines sicheren Datalogprogramms aus Abschnitt 2.3 wollen wir um eine Bedingung erweitern:

**Definition 2.6.1** *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.*

Die Bedingung der Definition 2.6.1 sorgt wieder dafür, dass auch die Variablen in der Aggregationsfunktion, durch das Vorkommen derselben Variable im Körper einer Regel, aus einer durch den Körper festgelegten Menge stammen. Damit kann garantiert werden, dass die Aggregationsfunktion nur auf Werte angewandt wird, die auch tatsächlich Inhalt einer Datenbankinstanz sind.

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

**Beispiel 2.6.2** Bei Betrachtung dieses Beispielprogramms [1]

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

kann festgestellt werden, dass das Programm, welches eigentlich die Summe der Variable `X` errechnen sollte keine sinnvolles Ergebnis liefert. Der Grund dafür ist die Rekursion in der zweiten Regel, welche nicht endlich ist.

Um dies zu illustrieren, soll angenommen werden, dass `q(x)` ein extensionales Prädikat ist und die Werte $\{1, 2\}$ hält.

Wird dann der Konsequenzoperator angewandt, ergibt sich für `p(x)` nach einem ersten Schritt die Tabelle 14 $T_P(I)$. Dieses Zwischenergebnis kommt zustande, 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 entspricht dann dem Inhalt der Tabelle 14 $T_P^2(I)$. In einem weiteren Schritt kommt dann wieder die Menge $\{1, 2\}$ hinzu und die vorige Ergebnismenge wird addiert, was zu dem Ergebnis, wie in Tabelle 14 $T_P^3(I)$ abgebildet, führt.

$$T_P(I) = \begin{array}{|c|}\hline \text{p(X)} \\ \hline 1 \\ 2 \\ \hline \end{array}\ , T_P^2(I) = \begin{array}{|c|}\hline \text{p(X)} \\ \hline 1 \\ 2 \\ 3 \\ \hline \end{array}\ , T_P^3(I) = \begin{array}{|c|}\hline \text{p(X)} \\ \hline 1 \\ 2 \\ 6 \\ \hline \end{array}$$

Tabelle 14: $p(X)$

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 ein Fixpunkt gefunden werden kann.

## 2.7 Stratifizierte Aggregation

Das in Beispiel 2.6.2 beschriebene Problem kann, analog zur stratifizierten Negation, ebenfalls durch Stratifizierung des Programms gelöst werden. Im Fall der Aggregation handelt es sich dann nach Green et al. [1] um die *stratifzierte Aggregation*.

Für die stratifizierte Aggregation gelten die für die stratifizierte Negation definierten Bedingungen 2.5.1 und 2.5.2 entsprechend. Zusätzlich wird eine weitere Bedingung getroffen:

**Bedingung 2.7.1** *Gibt es eine Prädikat A :- ...B,... in einem Datalogprogramm P, welche einen aggregierten Term enthält und A ist in Stratum $P_i$, während B in Stratum $P_j$ ist, dann ist $i > j$.*

Analog zur Bedingung 2.5.2 der Negation soll auch für die Aggregation gelten, dass Prädikate, welche aggregierte Terme enthalten, nach Prädikaten ausgewertet werden, welche keine aggregierten Terme enthalten. Hierzu wird Bedingung 2.7.1 festgelegt.

Äquivalent zur stratifizierten Negation kann mithilfe des dort eingeführten Präzedenzgraphen auch bei der stratifizierten Aggregation eine Stratifikation gefunden werden. Die Berechnung der stratifizierten Aggregation funktioniert auf Grundlage des beschriebenen theoretischen Hintergrunds dann äquivalent zur stratifizierten Negation [1].

**Beispiel 2.7.1** 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:

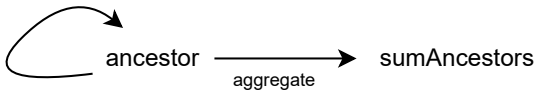


Abbildung 4: Präzedenzgraph

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. Anschlie-

ßend kann das Prädikat `sumAncestors` unter Anwendung der Aggregationsfunktion evaluiert werden und liefert folgendes Ergebnis:

| Person | Nr of Ancestors |
|---|---|
| Rose | 3 |
| Hugo | 3 |
| Ronald | 2 |
| Arthur | 1 |

Tabelle 15: *sumAncestors*

# 3 Partitionsprädikat in Datalog

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 Syntax und Semantik der Partition

Die Syntax des Partitionsprädikats ist in Anlehnung an das in Abschnitt 2.6 vorgestellte Aggregationsprädikat die Folgende:

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

$A$ und $B$ sind wiederum Atome. Auch bei der Partition, analog zur Aggregation, ist es erlaubt Funktionen im Kopf einer Partitionsregel 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 $\mathit{rank} < Z >$ ist außerdem ein *partitionierter Term*. Ein partitionierter Term kann formal durch den Term $f < t >$ ausgedrückt werden. Der Unterschied zum aggregierten Term ist, dass 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 weiteren Verlauf erläutert. Die Variable $X$ gilt als ein gewöhnlicher Term. Sie gehört weder zum partitionierten Term, noch zu den *Partitionierungsvariablen* und dient 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 `desc` (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 Arten Ränge zu errechnen. Die Zählung des Rangs beginnt bei beiden Funktionen mit der Zahl 1. Sie unterscheiden sich erst dann, wenn mehrere Tupel für die gewählte Variable im Partitionierungsterm dieselben Werte aufweisen. Unter der Annahme, dass eine Sortierung anhand der Werte zweier Variablen $X, Y$ gefunden werden soll, wobei $X = Y$ gilt, errechnet die Funktion `rank` nach dem *lückenlosen Ranking* für die Tupel beider Variablen denselben Rang, der dann entsprechend auch innerhalb derselben Partition mehrfach vergeben wird, sodass dann gilt $Rank_X = Rank_Y$. Die Funktion `uniqueRank` wählt nach dem *eindeutigen Ranking* für jedes der Tupel rein zufällig unterschiedliche Ränge, sodass aber in jedem Fall kein Rang innerhalb derselben Partition mehrfach vergeben wird. Hier gilt also stets $Rank_X \neq Rank_Y$. 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 $person(Name, Majority)$ gibt, welche die folgenden Werttupel aufweist:

| Name | Majority |
|---|---|
| Rose | minor |
| Hugo | minor |
| Ronald | adult |
| Arthur | adult |
| Dora | adult |

Tabelle 16: *person*

Hierbei kann das Partitionsprädikat genutzt werden um alle Personen der Beispieldaten anhand ihrer Namen zu sortieren.

**Beispiel 3.1.1** Die Regel sowie eine Ergebnistabelle für ein entsprechendes Partitionsprädikat sehen folgendermaßen aus:

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

| Name | Rank |
|---|---|
| Rose | 5 |
| Hugo | 3 |
| Ronald | 4 |
| Arthur | 1 |
| Dora | 2 |

Tabelle 17: *peopleByName*

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:

**Beispiel 3.1.2** Die Regel

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

partitioniert nach der Volljährigkeit 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:

| Name | Majority | Rank |
|---|---|---|
| Rose | minor | 2 |
| Hugo | minor | 1 |
| Ronald | adult | 3 |
| Arthur | adult | 1 |
| Dora | adult | 2 |

Tabelle 18: *peopleByName*

In zwei weiteren Beispielen sollen außerdem die Unterschiede zwischen dem lückenlosen und dem eindeutigen 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,Majority)`

würde also ebenfalls die Ergebnistabelle 17 resultieren. Analog dazu gilt für die Regel

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

und die Tabelle 18 das Gleiche. Es soll nun allerdings ein leicht verändertes extensionales Prädikat `person(Name,Majority,Birthday)` angenommen werden, welches als Grundlage für die folgenden Beispiele dient. Für dieses Prädikat sollen die Werttupel wie folgt dienen:

| Name | Majority | Birthday |
|---|---|---|
| Rose | minor | 19.03. |
| Hugo | minor | 06.02. |
| Ronald | adult | 01.03. |
| Arthur | adult | 06.02. |
| Dora | adult | 06.02. |

Tabelle 19: *person*

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 der Volljährigkeit partitioniert und nach dem Geburtstag der Personen sortiert wird.

**Beispiel 3.1.3** Eine entsprechende Regel hierfür könnte so aussehen und die Ergebnistabelle 20 liefern:

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

| Name | Majority | Birthday | Rank |
|---|---|---|---|
| Rose | minor | 19.03. | 2 |
| Hugo | minor | 06.02. | 1 |
| Ronald | adult | 01.03. | 2 |
| Arthur | adult | 06.02. | 1 |
| Dora | adult | 06.02. | 1 |

Tabelle 20: *birthdayRank*

Zu beachten seien hier die Tupel (*Dora*, *adult*, *06.02.*, *1*) und (*Arthur*, *adult*, *06.02.*, *1*), welche tatsächlich die gleichen Ränge in derselben Partition aufweisen.

Nun soll der Fall betrachtet werden, bei welchem `uniqueRank` als Partitionsfunktion verwendet, nach der Volljährigkeit partitioniert und nach dem Geburtstag der Personen sortiert wird.

**Beispiel 3.1.4** Wird eine solche Regel durch das eindeutige Ranking definiert, sieht sie so aus:

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

Und liefert die folgende Ergebnistabelle:

| Name | Birthday | $Rank_1$ |
|---|---|---|
| Rose | 19.03. | 2 |
| Hugo | 06.02. | 1 |
| Ronald | 01.03. | 3 |
| Arthur | 06.02. | 2 |
| Dora | 06.02. | 1 |

,

| Name | Birthday | $Rank_2$ |
|---|---|---|
| Rose | 19.03. | 2 |
| Hugo | 06.02. | 1 |
| Ronald | 01.03. | 3 |
| Arthur | 06.02. | 1 |
| Dora | 06.02. | 2 |

Tabelle 21: *birthdayRank*

Hierbei entstehen zwei alternative Ränge, welche in zwei Tabellen als $Rank_1$ und $Rank_2$ bezeichnet sind. Diese stellen die beiden möglichen Alternativen dar, die sich für den tatsächlichen Rang

letztendlich ergeben. Es ist festzustellen, 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 aus Beispiel 3.1.3 dienen. Diese wird allerdings dahingehend verändert, als dass der partitionierte Term nicht mehr nur die Variable `Birthday`, sondern nun zusätzlich auch die Variable `Name` hält.

**Beispiel 3.1.5** Somit sieht die Regel nun folgendermaßen aus:

```
birthdayRank(Name,Birthday,asc rank<Birthday,Name> by Majority)
:- person(Name,Majority,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 20 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 (*Dora*, *adult*, *06.02.*) und (*Arthur*, *adult*, *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:

| Name | Majority | Birthday | Rank |
|---|---|---|---|
| Rose | minor | 19.03. | 2 |
| Hugo | minor | 06.02. | 1 |
| Ronald | adult | 01.03. | 3 |
| Arthur | adult | 06.02. | 1 |
| Dora | adult | 06.02. | 2 |

Tabelle 22: *birthdayRank*

Wie zu erkennen ist, erhalten die beiden Tupel (*Dora*, *adult*, *06.02.*) und (*Arthur*, *adult*, *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.

## 3.2 Stratifizierte Partition

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 Fixpunkt-Semantik. Tatsächlich zeigen sich bei bestimmten Konfigurationen des Partitionsprädikats unter Verwendung von Rekursion aber ähnliche Effekte, wie das beim Aggregationsprädikat der Fall ist.

**Beispiel 3.2.1** 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:

| X | Y |
|---|---|
| 1 | 1 |
| 1 | 2 |

Tabelle 23: $e$

Hierbei werden durch Anwendung der ersten Regel in jedem Anwendungsschritt 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:

| Rank | Y |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 1 | 2 |

Tabelle 24: $p$

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:

| Rank | Y |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 2 |
| 1 | 2 |

Tabelle 25: $p$

Es ist 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, ähnlich wie bei rekursiven Regeln mit Aggregation, endlos weitergehen. In jedem Schritt werden die Ränge der Tupel durch Sortierung neu geordnet und es wird ein weiteres Tupel durch das extensionale Prädikat hinzugefügt. Hierbei kann also nie ein kleinster Fixpunkt gefunden werden.

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

**Bedingung 3.2.1** *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 3.2.1 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 2.5.1 und 2.5.2 entsprechend der stratifizierten Negation und Aggregation. Darüber hinaus wird hier eine weitere Bedingung eingeführt:

**Bedingung 3.2.2** *Gibt es eine Regel A :- ...B,... in einem Datalogprogramm 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 3.2.2 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 Konsequenzoperator 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 Datalogprogramms 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.

**Beispiel 3.2.2** Wir erhalten damit folgendes Datalogprogramm:

```
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.

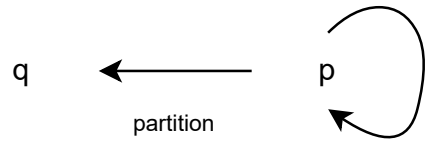


Abbildung 5: Präzedenzgraph

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:

| X | Y |
|---|---|
| 1 | 1 |
| 1 | 2 |
| 2 | 3 |

Tabelle 26: *e*

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 27 dargestellt, enthält.

| X | Y |
|---|---|
| 1 | 1 |
| 1 | 2 |
| 2 | 3 |
| 1 | 3 |

Tabelle 27: $p$

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:

| Rank | Y |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 3 |

Tabelle 28: $q$

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.

# 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 mithilfe Java-verwandter Sprachen realisiert werden können.

## 4.1 Was ist eine domänenspezifische Sprache?

Die Beispiele dieses Kapitels sind dem Buch *Learning Groovy* [13, S. 35-39] entnommen.

Entsprechend der Definition von Fowler und Parsons [14, Kap. 2] ist eine *domänenspezifische Sprache* eine Programmiersprache mit verminderter Ausdrucksfähigkeit, welche sich auf eine bestimmte Domäne spezialisiert. Ihre verminderte Ausdrucksfähigkeit führt insbesondere dazu, dass domänenspezifische Sprachen nicht Turing-vollständig sind. Domäne meint im Kontext der domänenspezifischen Sprachen die Fachgebiete, zu welchen Applikationen entwickelt werden. Eine domänenspezifische Sprache dient dazu Programmcode einfach schreib- und lesbar und zusätzlich von Computern ausführbar zu gestalten. Im Gegensatz zu Allzweck-Programmiersprache, nutzen domänenspezifische Sprachen eine Vielzahl an Funktionalität in Form von beispielsweise Kontroll-, Abstraktionsstrukturen und Ähnlichem nicht, weswegen sie nicht in der Lage sind sämtliche lösbaren Probleme bearbeiten zu können. Stattdessen konzentrieren sich domänenspezifische Sprachen auf ein Minimum an Funktionalität, die unbedingt benötigt wird um die tatsächlich zu lösenden Probleme einer entsprechenden Domäne zu behandeln.

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. Die Konsequenz aus dieser Einfachheit ist, dass eine domänenspezifische Sprache niemals dazu genutzt werden kann eine vollständige Applikation zu schreiben. Sie dient immer nur als Ergänzung und kann damit nur Teile der Implementierung einer Applikation abdecken. Ein wesentlicher Vorteil, der mit der Simplizität erreicht wird, ist dass domänenspezifische Sprachen nicht zwangsläufig von Personen genutzt werden müssen, die über ein umfangreiches Verständnis über Allzweck-Programmiersprachen und Programmierkonzepte 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 [14, S. 13]. Um die genannte Einfachheit zu erreichen, wird die Syntax domänenspezifischer Sprachen häufig so gewählt, dass sie einen eher deklarativen Charakter hat, welcher sich an die gesprochene Sprache anlehnt [14, S. 16]. Syntaktisch fügt die domänenspezifische Sprache üblicherweise Teilstücke aus dem Gesamtkontext so zusammen, dass sie eher an vollständige Sätze erinnern und keine zusammenhangslosen Befehle darstellen, wie dies bei imperativen Allzweck-Sprachen in der Regel der Fall ist.

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

**Externe domänenspezifische Sprache** Eine dieser Formen ist die *externe domänenspezifische Sprache*. Die Programmiersprache in welcher eine domänenspezifische Sprache und die Applika-

tion, in der sie eingebettet ist, geschrieben sind, 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) [14, S. 15]. Häufig wird neben der externen domänenspezifischen Sprache selbst, zusätzlich Parser Funktionalität in die Grundapplikation eingebaut, um den Text eigens definierter Sprachen zu parsen.

**Interne domänenspezifische Sprache** Neben 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. Damit erhält die interne domänenspezifische Sprache dann eine eigene Grammatik und erscheint wie eine neue Sprache, welche von der Grundsprache abzugrenzen ist. 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 komplexe Strukturen, wie beispielsweise Kontrollstrukturen, Schleifen und Ähnliches, verzichtet.

Bezüglich der konkreten Ausführung von domänenspezifischen Sprachen gibt es laut Fowler und Parsons [14, Kap. 1] ebenfalls zwei Konzepte die unterschiedliche Realisierungen bereithalten.

**Ausführung durch Interpretation** 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.

**Ausführung durch Code Generation** 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. Ein 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. Wenn dies der Fall ist, kann eine zweite Kompilierung des generierten Codes außerdem wegfallen, wenn der generierte Code in einer rein dynamischen Sprache geschrieben ist, 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, bevor er ausgeführt werden kann.

Die Form in welcher eine domänenspezifische Sprache in den Kontext ihrer Applikation eingebaut werden kann, lässt sich nach Fowler und Parsons [14, Kap. 2] wiederum in zwei unterschiedliche Arten unterteilen: in die *fragmentierte domänenspezifische Sprache* und die *eigenständige domänenspezifische Sprache*.

**Fragmentierte domänenspezifische Sprache** Die fragmentierte domänenspezifische Sprache zeichnet sich dadurch aus, dass nur Teile ihres Codes innerhalb anderen Codes in der Grundsprache eingepackt sind. Die domänenspezifische Sprache bereichert den Code der Grundsprache dann nur. Wesentlich für diese Art der Nutzung ist, dass es ohne Wissen über die Grundsprache kaum möglich ist den Gesamtkontext zu verstehen, weshalb oft nur Personen mit Programmiererfahrung diese Einsatzart 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.

**Eigenständige domänenspezifische Sprachen** Eigenständige domänenspezifische Sprachen sind dahingegen reine Code Teile, geschrieben in der domänenspezifischen Sprache, ohne jegliche Vorkommnisse der Grundsprache.

## 4.2 Vor- und Nachteile domänenspezifischer Sprachen

Gründe warum es überhaupt sinnvoll ist, eine domänenspezifische Sprache in ein Projekt zu involvieren, gibt es einige. Die Vor- und Nachteile, welche Fowler und Parsons [14, Kap. 2] beschreiben, werden in diesem Abschnitt erläutert.

**Vorteile domänenspezifischer Sprachen** 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.

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 zu erleichtern und als ein Kommunikationsmedium zwischen Softwareentwickler:innen und Domänenexpert:innen dienen können. 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, dafür aber tiefgehendes Wissen über die Domäne besitzen. 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 über das 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 wird, kann allein ihre Definition wesentlich dazu beitragen gegenseitiges Verständnis zu verbessern, sodass die reine Konzeption einer domänenspezifischen Sprache häufig schon einen Mehrwert liefert.

**Nachteile domänenspezifischer Sprachen** Kritik wird an domänenspezifischen Sprachen ebenfalls geäußert.

Ein Problem welches aus der Verwendung von domänenspezifischen Sprachen resultiert ist, dass mehrere Sprachen erlernt werden, beziehungsweise beherrscht sein müssen.

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 ist, wenn diese die selbst gebaute Sprache weder kennen noch nutzen.

Des weiteren kann es für Unternehmen, welche domänenspezifische Sprachen betreiben zur Herausforderung werden diese Sprache regelmäßig den neusten technologischen Entwicklungen entsprechend zu entwickeln und warten. 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.

Unerfahrene Entwickler:innen können auch insofern zum Problem bei der Konzeption einer domänenspezifischen Sprache werden, als dass schlecht gebaute Sprachen das Potential besitzen die anschließende Programmierung eher zu verkomplizieren und zu verschlechtern, anstatt sie einfacher verständlich und programmierbar zu gestalten.

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.

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.

## 4.3 Groovy als Grundsprache

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 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 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 um domänenspezifische Sprachen zu entwickeln [13, 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 besonders 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 und semantischen Tricks eine andere, einfachere Sprache zu generieren.

Beispielsweise gibt es viele Zeichen im Code klassischer imperativer Programmiersprachen, die dort syntaktisch notwendig sind, weil ihnen eine semantische Bedeutung zugewiesen wurde und sie für Compiler oder Interpreter dann essentiell sind um die Semantik ganzer Ausdrücke erfassen zu können. Das Semikolon gibt zum Beispiel 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 in Programmiersprachen der Fall ist. Da es 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 [15, S. 677]. Klammern, welche Argumente bei einem Methodenaufruf umklammern, sind in Groovy optional und können daher ebenfalls weggelassen werden [13, S. 35].

Neben entfallener Zeichensetzung, die in der gesprochenen Sprache unüblich ist, profitiert Groo-

vy von einer weiteren syntaktischen Eigenschaft, welche beim Bau von domänenspezifischen Sprachen äußerst nützlich sein kann. Laut König und King [15, Kap. 19] sind die sogenannten *benannten Argumente* Argumente, die beim Aufruf einer Methode eine zusätzliche Information zu jedem übergebenen Argument erlauben, 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:

```
foo argOne, keyOne: valueOne, argTwo, keyTwo: valueTwo
```

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. Ausgenutzt wird diese Fähigkeit aber in folgendem Beispiel:

```
move right, by: 3.meters
```

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 abzusehen 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.

Ebenfalls auffällig ist die Anwendung des Punktoperators eines Zahlenwerts auf eine Einheit, welche in einem Gesamtkontext beispielsweise als Enumeration kodiert sein könnte und die ebenfalls von König und Kind erläutert wird [15, Kap. 19]. 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 dies mit Verwendung einer Instanz realisieren und würde wie folgt aussehen:

```
move right, by: new Distance(3, meters)
```

Diese Schreibweise eines Methodenaufrufs gleicht syntaktisch viel mehr der klassischen Java Syntax, wenn auch Klammern um die Argumente der Methode 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 nach König und King [15, Kap. 19] 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. 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. 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 laut Davis [13, Kap. 7] die sogenannten *Delegates*. 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 [13, 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. 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. Als ausgeschriebener Code würde dies folgendermaßen aussehen:

```
SMS msg = new SMS();
msg.from("01234");
msg.to("56789");
msg.body("Hello");
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:

```
def static send(Closure block) {
    SMS msg = new SMS()
    block.delegate = msg
    block()
    msg.send()
}
```

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:

```
SMS.send {
    from '01234'
    to '56789'
    body 'Hello'
}
```

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. 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, welche 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 [13, S. 37].

Zuletzt verfügt Groovy nach Davis [13, Kap. 7] über eine weitere sehr nützliche Funktionalität im Bezug auf domänenspezifische Sprachen. Die beiden Groovy Methoden `methodMissing(String name,` und `propertyMissing(String name)` bieten eine Möglichkeit mit Methoden und Attributen umzugehen, welche es überhaupt nicht gibt, die also nie definiert wurden. Voraussetzung dafür ist, dass man solche Methoden und Attribute in der jeweils korrekten Syntax verwendet, also in dem jeweiligen Kontext eines Attributs oder einer Methode. Die Methoden `methodMissing` und `popertyMissing` werden getriggert, wenn Bezeichner im Code auftauchen, welche nicht definiert wurden. Die Funktionalität der Methoden `methodMissing` und `propertyMissing` kann innerhalb einer Klasse für nicht definierte Methoden und Attribute dieser Klasse implementiert werden, sodass ein beliebiger Umgang mit unbekannten Methoden und Attributen realisiert werden kann.

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

```
class Chemistry {
    public static void exec(Closure block) {
    block.delegate = new Chemestry()
        block()
    }
    def propertyMissing(String name) {
        def comp = new Compound(name)
        (comp.elements.size() == 1 && comp.elements.values()[0] == 1) ?
        comp.elements.keySet()[0] : comp
    }
}
```

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:

```
Chemistry.exec {
    def water = H2O
    println water
}
```

Hierbei soll angenommen werden, dass das Attribut `H2O` zuvor nicht definiert wurde und entsprechend die `propertyMissing` Methode triggert. Wie zu erkennen ist, können 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

In diesem Kapitel wird die konkrete Implementierung des Partitionsprädikats in der Sprache Roxx erläutert. Hierzu wird zunächst auf einige Eigenschaften der Sprache Roxx im Allgemeinen eingegangen. Anschließend wird das Backend der Applikation WorkforcePlus in dem Rahmen skizziert, welcher für das Partitionsprädikat von Relevanz ist. Zuletzt erfolgt außerdem eine Beschreibung der konkreten Umsetzung der Mehrfachsortierung und Implementierung der Partitionsfunktion des eindeutigen Rankings.

## 5.1 Die Sprache Roxx

Die Sprache Roxx wird genutzt um konkrete Kundenmodelle in der Anwendung WorkforcePlus zu erstellen. Sie ist eine domänenspezifische Sprache, welche dem logisch deklarativen Programmierparadigma folgt und baut darüber hinaus auf den Konzepten der deduktiven Datenbanken auf. Roxx kann daher als ein Dialekt der Programmiersprache Datalog erachtet werden.

Roxx ist in zwei Grundsprachen geschrieben: Java und Groovy. Die Logik, mit welcher die Prädikate von Roxx implementiert sind, ist in Java geschrieben. Die Syntax von Roxx wurde mithilfe der in Kapitel 4 erläuterten Konzepte in der Sprache Groovy konzipiert. Konkret wird hier insbesondere das Delegate verwendet, das es erlaubt die verschiedenen Prädikate von Roxx wie etwa das Partitionsprädikat oder das Aggregationsprädikat in Klassen zu organisieren, welche an eine Closure delegiert werden, die anschließend ausgeführt werden kann, sodass es möglich ist eine Syntax zu verwenden, wie sie in Listing 1 dargestellt ist.

Roxx ist damit eine interne domänenspezifische Sprache, denn sie besteht aus validem Groovy Code, welcher auf die spezifische Roxx Grammatik eingeschränkt wird. Außerdem wird Roxx Code nicht interpretiert, sondern generiert wiederum neuen Java Code durch Kompilierung.

Verwendet wird Roxx sowohl als fragmentierte, wie auch als eigenständige domänenspezifische Sprache. Es ist möglich Roxx Dateien zu verfassen, welche ausschließlich Roxx Code beinhalten, was dem Konzept der eigenständigen domänenspezifischen Sprache entspricht. Da es sich bei Roxx um eine interne domänenspezifische Sprache handelt und Roxx Code damit auch Groovy Code ist, ist es aber durchaus auch möglich Roxx Code in Groovy Code einzufügen, was zum Zwecke von Code Modularisierung und dem Hinzufügen von Funktionalität, der Roxx nicht mächtig ist, in so genannten Makro Dateien tatsächlich passiert.

## 5.2 Partitionsprädikat in Roxx

Das Partitionsprädikat in Roxx stellt eine Implementierungsmöglichkeit des in Abschnitt 3.1 vorgestellten Partitionsprädikats in Datalog dar und besitzt dieselbe Semantik wie dieses. Innerhalb der Sprache Roxx stellt es, aufgrund der Partitionsfunktion, die es von gewöhnlichen intensionalen Prädikaten unterscheidet, ein fortgeschrittenes Prädikat der Sprache Roxx dar [5].

Die Syntax des Partitionspärdikats in Roxx unterscheidet sich jedoch stark von der des Partitionsprädikats in Datalog.

Der folgende Code zeigt die Syntax eines beispielhaften Partitionsprädikats:

Listing 1: Beispiel eines Partitionsprädikats

```
partitionPredicate {
    employeeBirthdayRank('emplGroup','empl','birthday','birthdayRank') {
        from = employeesBirthdays('emplGroup','empl','birthday')
        partitionBy = ['emplGroup']
        birthdayRank = Rank(OrderDesc('birthday'))
    }
}
```

Dabei gibt die Zeile

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

das Prädikat mit entsprechenden Attributen an, das als Grundlage für das Partitionieren dienen soll. Mit dem `from` Schlüsselwort wird also stets das Atom gleichgesetzt, welches in Datalog den Körper der Regel eines Partitionspärdikats ausmacht.

Die Zeile

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

erwartet eine Liste von Attributen, die einstellig, mehrstellig oder auch leer sein kann. Diese Attribute entsprechen den Partitionierungsvariablen in Datalog und bestimmen daher wonach partitioniert wird. In diesem Beispiel ist das eine Gruppe von Arbeitnehmerinnen und Arbeitnehmern, definiert über das Attribut `emplGroup`.

Zuletzt wird die Operation definiert, welche das Partitionsprädikat ausführen soll. Hierbei stehen, analog zum Datalog Partitionsprädikat, die beiden Funktionen `rank` und `uniqueRank` zur Verfügung, welche dieselbe Funktionsweise besitzen, wie sie in im Abschnitt des Partitionspärdikats in Datalog erläutert sind. Der einzige Unterschied besteht darin, dass die Zählung der Ränge bei der Partitionsfunktion in Roxx bei der Zahl 0 beginnen, wohingegen die Partitionsfunktion in Datalog in Abschnitt 3.1 so definiert sind, dass die Zählung bei 1 beginnt.

Um der Syntax in Roxx gerecht zu werden, muss ein beliebiger Bezeichner gewählt werden, welchem das Ergebnis der Sortieroperation zugewiesen wird. Im oben aufgeführtem Beispiel entspricht dies folgender Zeile:

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

Die Sortierrichtung wird in Roxx mit den Funktionen `OrderAsc` und `OrderDesc` spezifiziert, wobei `OrderAsc` dem Schlüsselwort `asc` und `OrderDesc` dem Schlüsselwort `desc` der Datalog Variante entsprechen.

Zur Veranschaulichung der Funktionalität des Partitionsprädikats wird das Beispielprädikat im Folgenden anhand von realistischen Beispieldaten, wie sie in der Applikation WorkforcePlus vorkommen können, evaluiert.

> **Beispiel 5.2.1** Dieses Beispiel dient der Demonstration des Partitionsprädikat mit realistischen Daten mit Bezug zur Domäne. Beispieldaten und Regel sind entnommen aus Übungsinhalten, welche die INFORM GmbH seinen Mitarbeitern zu Zwecken des Trainings zur Verfügung stellt.

Das Prädikat `employeesBirthday`, welches als Grundlage für das Partitionsprädikat verwendet wird, enthält drei Variablen: die Gruppe von Arbeitnehmerinnen und Arbeitnehmern `emplGroup`, die entsprechende Mitarbeiterin oder der entsprechende Mitarbeiter `empl`, sowie der Geburtstag der Mitarbeiterin oder des Mitarbeiters `birthday`. Für die Berechnung des Partitionsprädikats soll folgende Beispielbelegung dieser Variablen angenommen werden:

| 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 |

Tabelle 29: Beispieldaten als Grundlage eines Partitionsprädikats

In der folgenden Tabelle wird zunächst die Partitionierung dieser Daten nach der Variable `emplGroup` dargestellt. 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 |

Tabelle 30: Beispieldaten nach der Partitionierung

Nach Berechnung der Partitionsfunktion wird nun jeder Zeile der Tabelle und somit jeder Mitarbeiterin und jedem Mitarbeiter ein Rang anhand der absteigenden Reihenfolge der Sortiervariable `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 auch hier der Fall, bei welchem derselbe Wert des Sortierattributs für mehrere Spalten auftritt. Dieser ist in Tabelle 31 in den Spalten des Mitarbeiters `Ron` und der Mitarbeiterin `Tina` zu beobachten. Beide Mitarbeiter besitzen den Wert 03.04.1986 für die Variable `birthday`. Die Konsequenz daraus ist, dass beide nach dem eindeutigen Ranking denselben Rang, den Startwert 0, erhalten und die nächste unterscheidbare Zeile den um eins erhöhten Rang erhält.

| emplGroup | employee | birthday | Rank |
|---|---|---|---|
| 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 | Anne | 11.12.1996 | 0 |
| EmplGr1 | Sofia | 07.01.1974 | 1 |
| EmplGr1 | Peter | 22.08.1990 | 2 |

Tabelle 31: Beispieldaten nach Evaluierung der Partitionsfunktion

Wie dieses Beispiel zeigt, ist die Funktionsweise des Partitionsprädikats in Roxx also tatsächlich die gleiche wie die des Partitionsprädikats in Datalog, mit dem Unterschied der abweichenden Startzahlen bei der Zählung des Rangs. Diese gleiche Funktionsweise gilt analog auch für alle weiteren in Abschnitt 3.1 gezeigten Konfigurationsmöglichkeiten des Partitionsprädikats.

## 5.3 Backend der Applikation WorkforcePlus

Das Backend von WorkforcePlus implementiert sowohl die domänenspezifische Sprache Roxx, als auch die Logik der Prädikate. Es ist ein überwiegend in der Programmiersprache Java geschriebener Anwendungs-Webserver, der über HTTP Anfragen mit den Komponenten von WorkforcePlus kommuniziert. Des Weiteren organisiert das Backend die Benutzerumgebung von WorkforcePlus in Sessions, handelt diese und verwaltet die Daten der Applikation durch das Schreiben und Lesen von und in externe Datenbanken wie PostgreSQL und Oracle Database, welche der Applikation als Quelldatenbanken dienen [16].

Als Hilfsmittel zur Implementierung nutzt das Backend einige Frameworks. 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 den *Apache Tomcat Server*, welcher für die Implementierung der HTTP Kommunikation zuständig ist, sowie *PostgreSQL* und *Oracle* als Relationale-Datenbank-Managementsysteme, die, wie bereits beschreiben, als Quelldatenbanken für die deduktive Datenbank, auf welcher Roxx basiert, fungieren. (C. Briem, persönliche Kommunikation, 20. Dezember 2022)

Die Architektur des Partitionsprädikats im Backend ist in Abbildung 6 dargestellt.

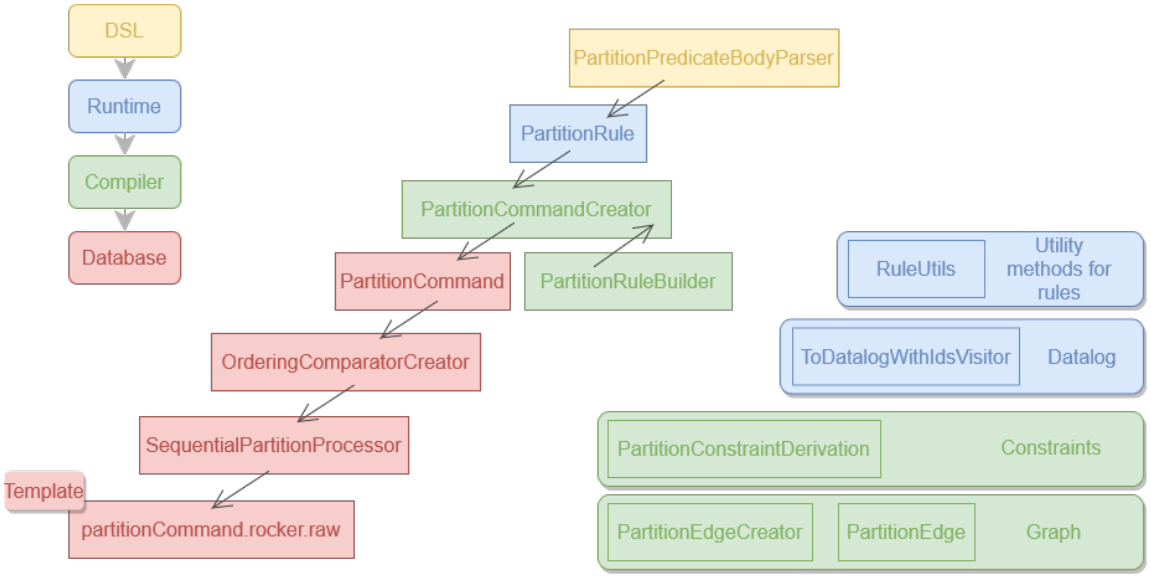


Abbildung 6: Architektur Partitionsprädikat

Grundsätzlich ist das Backend in einer Art Schichtmodell durch Module aufgebaut. Diese Module `DSL` (Abk.: domain-specific language, zu deutsch: domänenspezifische Sprache), `Runtime`, `Compiler` und `Database` sind in der Abbildung farbig markiert, sodass erkennbar ist, welche Klassen zu welchem Modul und entsprechend in welche Schicht gehören. Außerdem sind die Module selbst oben links in der Abbildung aufgeführt. Hier ist auch ihre Hierarchie anhand der Pfeile dargestellt. Auf unterster Ebene liegt die Schicht der Datenbank, welche die Logik der Partitionierung und Sortierung des Partitionsprädikats implementiert. Hier bestehen Prädikate nur aus Reihen und Spalten mit entsprechenden Werten, welche die Werttupel der Prädikate darstellen. Sie ist abhängig von Daten aus der Schicht des Compilers. Auf dieser Ebene befinden sich Klassen, welche Abstraktionsformen von Prädikaten bauen, die alle Informationen besitzen, die benötigt werden, um die Logik auf Ebene der Datenbank zu realisieren. Die Ebene des Compilers erhält alle nötigen Informationen wiederum aus der Laufzeitebene, welche die Prädikate auf logische Regeln gemäß ihrer Datalog-Repräsentation abstrahiert. Auf der obersten Ebene der domänenspezifischen Sprache wird die genaue Syntax der Sprache Roxx definiert. Außerdem stellt diese Ebene alle Informationen, welche aus einem Roxx Prädikat extrahiert werden können, für Klassen der Laufzeitebene bereit.

In der Abbildung sind außerdem sämtliche Klassen zu dargestellt, welche in der entsprechenden Ebene die erläuterte Funktionalität implementieren. Auch hier werden Abhängigkeiten durch Pfeile dargestellt, sodass die Hierarchie hier nun feingranularer zu erkennen ist.

Zunächst ist auf der obersten Ebene, der Ebene der domänenspezifischen Sprache, die Groovy Klasse `PartitionPredicateBodyParser` dafür verantwortlich die Roxx Syntax zu definieren. Außerdem werden alle Daten, welche für die Weiterverarbeitung benötigt werden in lokalen Variablen gesammelt. Zu diesen nötigen Daten gehören das Quellprädikat, von welchem die Partitionsregel abgeleitet wird, eine Liste von Partitionierungsvariablen, die Partitionsfunktion, sowie die Sortiervariablen und dazugehörige Sortierrichtungen.

Auf der Laufzeitebene stellt eine Instanz der Java Klasse `PartitionRule` das Partitionsprädikat in Form einer Datalog-Regel dar. Die Syntax der hier implementierten Datalog-Regel weicht leicht von der Regel des Partitionsprädikats, wie sie in Kapitel 2 definiert wurde, ab. Während

die Syntax dort der Beispielregel

```
A(X,Y,Z, asc rank<Z> by Y) :- B(X,Y,Z)
```

entsprach, würde eine äquivalente Regel entsprechend der Implementierung wie folgt aussehen:

```
A(X,Y,Z,RANK) :- partition(B(X,Y,Z) by (Y), rank(Z ASCENDING AS RANK)).
```

Wesentliche Unterschiede der jeweiligen Syntax sind das Schlüsselwort `partition`, welches die implementierte Syntax verwendet, sowie eine andere Anordnung der Regelbestandteile. Im Kopf der implementierten Regel befinden sich, wie in gewöhnlichen intensionalen Prädikaten, ausschließlich die Variablen, welche projiziert werden sollen. Die Partitionsfunktion steht zusammen mit dem Quellprädikat im Körper der Regel. Außerdem wird der Ergebnisrang in der implementierten Syntax explizit mit einem Bezeichner versehen, welcher beliebig gewählt werden kann, wohingegen die Projektion des Ergebnisrangs in der theoretisch definierten Partitionsregel nur implizit durch die Angabe der Partitionsfunktion geschieht.

Die Java Klassen `PartitionCommandCreator` und `PartitionRuleBuilder` auf der Compiler Ebene nehmen ausschließlich die Informationen entgegen, welche sich aus den Instanzen der `PartitionRule` Klasse ergeben und erzeugen damit, wie bereits erläutert, Objekte, welche Instanzen des `PartitionCommand`, der simpelste Abstraktionsform des Partitionsprädikats, erzeugen können.

Eine Instanz der Klasse `PartitionCommand` enthält als einfachste Abstraktion des Partitionsprädikats ausschließlich Informationen darüber, mit welchen konkreten Werten Methoden getriggert werden müssen, um die korrekte Partitionierung und Sortierung auszuführen. Sowohl Eingabewerte als auch die Ergebnismenge bestehen hierbei bereits ausschließlich aus Reihen von Werten, welche die Werttupel des Prädikats darstellen. Neben der Klasse `PartitionCommand` befinden sich im Modul `Database` außerdem zwei weitere Klassen, `OrderingComparatorCreator` und `SequentialPartitionProcessor`. In der Klasse `OrderingComparatorOperator` wird die Sortierlogik entsprechend der vorgesehenen Funktionalität der beiden Partitionsfunktionen `rank` und `uniqueRank` implementiert. Die Klasse `SequentialPartitionProcessor` implementiert die Partitionierung, führt anschließend mithilfe einer Instanz der Klasse `OrderingComparator` die Sortierung durch und errechnet die aus der Sortierung resultierenden Ränge der Tupel des Prädikats. Auch die Rangberechnung wird in dieser Klasse je nach Partitionsfunktion entsprechend gehandhabt. Zuallerletzt wird bei der Evaluierung eines Partitionsprädikats ein Template, `partitionCommand.rocker.raw` ausgeführt, welches in der Lage ist durch eine Template-Sprache die Code Generation durchzuführen. Hierzu entnimmt es alle nötigen Daten aus den Klassen `OrderingComparatorOperator` und `SequentialPartitionProcessor` und generiert Java Code, welcher dann im Rahmen der vollständigen WorkforcePlus Applikation kompiliert und ausgeführt werden kann.

Abbildung 6 zeigt ebenfalls vier Klassen, die zwar zunächst keine direkten Auswirkungen auf die Funktionsweise des Partitionsprädikats haben, allesamt aber dennoch Teile von diesem beinhalten, sodass es durchaus Berührungspunkte mit dem Partitionsprädikat gibt. Dazu zählen zwei Klassen der Laufzeitebene. Eine dieser Klassen, `ToDatalogWithIdsVisitor`, generiert aus Roxx Regeln äquivalente Datalog-Regeln in einer Repräsentation aus Zeichenketten. Die andere der beiden, `RuleUtils` stellt eine Hilfsklasse dar, die Hilfsmethoden für Regeln über verschiedene Arten von Prädikaten hinweg bereitstellt. Des Weiteren sind auch die Klassen `PartitionConstraintDerivation`, `ParitionEdgeCreator` und `PartitionEdge` im Modul `Compiler` für das Partitionsprädikat von Relevanz. Die Klasse `PartitionConstraintDerivation`

berechnet erlaubte Ober- und Untergrenzen für die Anzahl unterschiedlicher Werte, welche sich für Variablen in den Ergebnistupeln des Partitionsprädikats ergeben dürfen. Die Klassen `PartitionEdgeCreator` und `PartitionEdge` erzeugt aus dem Partitionsprädikat einen Knoten für den Präzedenzgraphen eines Roxx Programms, mit welchem es, entsprechend der Erläuterung in Kapitel 2, möglich ist, Zyklen in Programmen zu erkennen, eine Aussage darüber zu treffen, ob ein Programm stratifzierbar ist, oder nicht und eine valide Stratifikation zu finden, sofern es eine gibt.

## 5.4 Umsetzung des Partitionsprädikats

Im Rahmen dieser Arbeit wurde das Partitionsprädikat um Funktionalität erweitert. Vor Beginn der Implementierung war das Partitionsprädikat ausschließlich dazu in der Lage Tupel nach dem lückenlosen Ranking zu sortieren und eine Sortierung nach mehr als einer Sortiervariablen war nicht möglich. Der Implementierungsgegenstand, welcher in dieser Arbeit skizziert werden soll, besteht also aus der Erweiterung des Partitionsprädikats nach beliebig vielen Variablen sortieren zu können, sowie der Erweiterung um die Partitionsfunktion `uniqueRank`, welche das einzigartige Ranking implementiert.

Die Sortierlogik des Partitionsprädikats ist mit dem funktionalen Java Interface `Comparator` realisiert und basiert hauptsächlich auf dessen Methoden `comparing` und `reversed`, wobei `comparing` einen `Comparator` erzeugt, der nach einer natürlichen Ordnung sortiert und die Methode `reversed` dazu in der Lage ist, diese natürliche Ordnung umzudrehen, sodass dadurch die absteigende Sortierung realisiert werden kann. Die Mehrfachsortierung kann darauf aufbauend durch eine weitere Methode des `Comparator` Interfaces `thenComparing` implementiert werden. Diese Methode erlaubt es mehre Instanzen von Objekten, welche das `Comparator` Interface implementieren, zu sequenzieren. Auch die Methode `thenComparing` lässt durch den darauffolgenden Methodenaufruf der Methode `reversed` eine absteigende Sortierrichtung zu, sodass die Richtung in einer Fallunterscheidung zusätzlich spezifiziert werden kann. Die Methode `thenComparing` hat immer dann einen Effekt, wenn die `Comparator` Instanz, auf welche sie angewendet wird, das Ergebnis hält, dass zwei zuvor verglichene Werte gleich sind. In diesem Fall löst die Methode eine weitere Sortierung nach der Sortiervariable aus, welche ihr als Argument übergeben wird [17]. Einen Auszug aus dem Code, welcher die Sortierlogik implementiert kann in Listing 2 betrachtet werden.

Listing 2: Auszug der Sortierlogik

```
1  for (Pair<Integer, Boolean> orderByPair : a_orderByPairs) {

3      Comparator<IRow> nextComparator =
       Comparator.comparing(row -> (Comparable<Object>) row
5      .c(orderByPair.getKey())
       .getValue());
7      if (orderByPair.getValue()) {
           nextComparator = nextComparator.reversed();
9      }
       if (comparator == null) {
11         // there is no preceding comparator
           //   -> set the first comparator
13         comparator = nextComparator;
```

```
    } else {
15      // there already is a comparator
        //   -> combine with the current comparator
17      comparator = comparator.thenComparing(nextComparator);
    }
19 }
```

Des Weiteren musste zur Realisierung der Mehrfachsortierung eine neue Datenstruktur gefunden werden, welche sowohl Information über die Sortiervariable, als auch über die Sortierrichtung hält. Ein naiver Ansatz wäre es hierfür zwei Instanzen der `ArrayList` zu wählen. Die Schwierigkeit liegt hierbei allerdings darin, dass Sortiervariable und Sortierrichtung immer nur als ein zusammengehöriges Paar funktionieren können. Es ist zwar möglich die Zusammengehörigkeit einer Variable und Richtung über die Indizes der Listen zur organisieren, allerdings muss hierbei akribisch darauf geachtet werden, dass diese beiden Elemente niemals unterschiedliche Indizes besetzen, oder dass die Listen unterschiedlich lang werden. Beide Fälle würden auf Inkonsistenzen hinweisen, die bei zwei unterschiedlichen Instanzen von Listen aufgrund ihrer Unabhängigkeit durch Programmierfehler oder Synchronisierungsschwierigkeiten durch Ausführung von Programmteilen in unterschiedlichen Programmthreads schnell auftreten können.

Statt der `ArrayList` ist für die Implementierung daher eine Datenstruktur gewählt, die von der Java Klasse `Pair` abgeleitet ist. Diese Klasse entspricht einer Datenstruktur, welche stets zwei Elemente beinhaltet, die als `key` und `value` bezeichnet werden. Diese beiden Elemente können unterschiedliche oder auch denselben Datentypen besitzen und werden bei der Instanziierung eines `Pair` Objektes direkt an dessen Konstruktor übergeben. Weiterhin besitzt eine Instanz der Klasse `Pair` durch die Methoden `getKey` und `getValue` Möglichkeiten auf die einzelnen Bestandteile des im `Pair` Objekt gespeicherten Paares zuzugreifen und dient daher sehr gut als Datenstruktur zweier zusammengehöriger Elemente [18].

Da ein einzelnes `Pair` Objekt aber noch nicht eine ganze Reihe an Sortiervariablen mit dazugehöriger Sortierrichtung beinhalten kann, wird stattdessen eine Datenstruktur gewählt, welche bereits im Backend der WorkforcePlus Applikation implementiert war, namens `Pairs`. Eine Instanz der Klasse `Pairs` ist eine `ArrayList` von `Pair` Objekten, die entsprechend eine beliebig lange Liste organisieren kann. In dieser Liste kann eine beliebige Anzahl an Sortiervariablen und zugehöriger Richtung in jeweils einem Feld der `ArrayList` gehalten werden , ohne dass hierbei Inkonsistenzen in der Datenstruktur auftreten können.

Zuletzt muss außerdem eine Syntax gefunden werden, welche die Verwendung der Mehrfachsortierung im Partitionsprädikat erlaubt. Hierfür wird das Argument, welches an die Partitionsfunktion übergeben wird zu einer Kommata getrennten Liste erweitert und die Groovy Methode `Rank` in der Klasse `PartitionPredicateBodyParser` dahingehend angepasst. Sollte das Beispiel aus Listing 1 die Mehrfachsortierung nutzen und darum erweitert werden, könnte die entsprechende Syntax folgendermaßen aussehen:

Listing 3: Partitionsprädikat mit Sortierung nach zwei Variablen

```
1 partitionPredicate {
    employeeBirthdayRank('emplGroup','empl','birthday','birthdayRank') {
3       from = employeesBirthdays('emplGroup','empl','birthday')
        partitionBy = ['emplGroup']
5       birthdayRank = Rank(OrderDesc('birthday'), OrderAsc('empl'))
```

```
    }
}
```

Neben der Mehrfachsortierung wurde im Rahmen dieser Arbeit auch eine neue Partitionsfunktion eingeführt: die `uniqueRank` Funktion, welche das eindeutige Ranking implementiert. Hierzu wird im Modul `Database` ein boolescher Wert eingeführt, der spezifiziert ob das eindeutige Ranking oder das lückenlose Ranking angewandt wird. Auf den verbleibenden drei Ebenen wird dies in einer Enumeration realisiert, welche es erlaubt auch zukünftig hinzukommende Partitionsfunktionen hinzuzufügen, ohne die Datenstruktur an dieser Stelle nochmals zu ändern. In der Klasse `SequentialPartitionProcessor` wurde außerdem die Logik der Rangvergabe dahingehend angepasst, dass Tupel mit gleichen Werten der Sortiervariablen zufällig unterschiedliche Ränge zugeteilt bekommen, wenn die `uniqueRank` Partitionsfunktion angewandt wird. Listing 4 zeigt den Code, welcher die Logik der Rangvergabe für beide Partitionsfunktionen implementiert.

Listing 4: Auszug der Logik der Rangvergabe

```
for (List<IRow> part : partition.values()) {
    part.sort(a_rowComparator);
    // rank is zero-based
    int rank = 0;
    IRow rowBefore = null;
    for (IRow row : part) {
        if (rowBefore != null) {
            // Unique ranking
            // Same elements randomly get different ranks.
            // So each rank will always be increased except for the
            // first element's rank that is always rank 0
            if (a_uniqueRanking) {
                rank++;
            }
            // This condition implements so-called "dense ranking" here:
            // Same elements get the same rank.
            // So we only increase the rank if the element before
            // was not equal according to the comparator
            else if (a_rowComparator.compare(rowBefore, row) != 0) {
                rank++;
            }
        }
        result.add(row, IntegerValue.of(rank));
        rowBefore = row;
    }
}
```

Wie bereits in Abschnitt 3.1 erläutert, geschieht die Rangvergabe beim eindeutigen Ranking aufgrund der zufälligen Vergabe unterschiedlicher Ränge bei gleichen Werten nichtdeterministisch. Dies kann in der Applikation allerdings zu unerwünschten Verhalten führen. Wird beispielsweise im Rahmen eines Partitionsprädikats ein eindeutiger Rang für mehrere Tupel berechnet, wobei zwei oder mehrere Tupel denselben Wert der Sortiervariable aufweisen, ergibt sich hieraus die rein zufällige Rangvergabe für diese Tupel. Wenn nach einem Neustart der Applikation dasselbe Prädikat erneut berechnet wird, besteht die Wahrscheinlichkeit, dass

die zufällige Rangvergabe zu diesem Zeitpunkt ein anderes Ergebnis errechnet, als vor dem Neustart. Dies gilt als unerwünschtes Verhalten, da es zu nicht zu erwartenden Effekten führt, sodass die Motivation besteht die Berechnung deterministisch zu gestalten.

Der Ansatz, welcher hierfür gewählt wurde, sieht vor bei Anwendung des eindeutigen Rankings ein weiteres Mal die Methode `thenComparing` mit dem Hash-Code der zu vergleichenden Elemente als Argument aufzurufen. Dies geschieht zum Schluss, nachdem alle im Partitionsprädikat angegebenen Sortiervariablen abgearbeitet wurden, sodass es ein weiteres Sortierkriterium existiert, wenn es zu diesem Zeitpunkt immer noch mehrere Tupel mit gleichen Werten gibt. Hiermit kann die Rangvergabe beim eindeutigen Ranking etwas deterministischer gestaltet werden, wobei sie dennoch nicht als völlig deterministisch gilt. Es ist möglich, dass zwei Objekte denselben Hash-Code erhalten [19, S. 722].

In einem speziellen Fall, in welchem zwei miteinander zu vergleichende Variablen zweier Tupel dieselben Werte enthalten, die Tupel darüber hinaus außerdem denselben Hash-Code besitzen, würde die Rangvergabe beim eindeutigen Ranking weiterhin nichtdeterministisch sein, da hier prinzipiell auch nach Vergleich nach dem Hash-Code weiterhin derselbe Rang vergeben werden müsste, sodass die Rangvergabe dann tatsächlich wieder zufällig geschieht. Offensichtlich ist ein solcher Fall allerdings derart unwahrscheinlich, dass die Implementierung, so wie sie beschrieben ist, bei Anwendung des eindeutigen Rankings im Rahmen der WorkforcePlus Applikation als ausreichend deterministisch gilt.

Schließlich wird auch für das eindeutige Ranking eine neue Roxx Syntax eingeführt. Hierfür wird die Groovy Methode `UniqueRank` in der Klasse `PartitionPredicateBodyParser` hinzugefügt, sodass die Verwendung des eindeutigen Rankings in einem Roxx Partitionsprädikat beispielsweise folgendermaßen aussehen kann:

Listing 5: Partitionsprädikat mit Partitionfunktion UniqueRank

```
partitionPredicate {
    employeeBirthdayRank('emplGroup','empl','birthday','birthdayRank') {
        from = employeesBirthdays('emplGroup','empl','birthday')
        partitionBy = ['emplGroup']
        birthdayRank = UniqueRank(OrderDesc('birthday'))
    }
}
```

Jede Ebene des Backends verfügt außerdem über eine umfangreiche Menge an Tests. Diese sind Unit-Tests des Frameworks *Spock* und decken je Ebene die Funktionsweise in der entsprechenden Abstraktionsform ab.

Konkret bedeutet dies, dass im Modul `Database` die reine Partitionerungs- und Sortierlogik anhand von Tabellen aus Zeilen und Spalten mit Werten getestet wird. Im Modul `Compiler` wird getestet, ob das Bauen von Regeln korrekt funktioniert. Die Regel selbst und die Äquivalenz zu einer validen Datalog-Regel werden im Modul `Runtime` getestet und zuletzt verifizieren Tests im Modul `DSL` die korrekte Funktionsweise des Partitionsprädikats in einem Gesamtkontext der Applikation mitsamt der entsprechenden Syntax.

Mit Abschluss der Implementierung jeder Ebene wurden also vor Beginn des nächsten Abschnitts zu den bereits bestehenden Tests im Backend neue Tests hinzugefügt um jeweils die neu implementierten Erweiterungen, die Mehrfachsortierung sowie die neue Partitionsfunktion `uniqueRank`, auf mögliche Fehler zu überprüfen. Erst wenn die korrekte Funktionsweise durch

die geschriebenen Tests garantiert wurde, galt die Implementierung der Änderungen auf jeder Ebene als vollständig und abgeschlossen. Damit kann gewährleistet werden, dass die implementierten Erweiterungen das zu erwartende Verhalten aufweisen und unerwünschtes Verhalten ausbleibt.

# 6 Zusammenfassung und Ausblick

In dieser Arbeit wurde das Partitionsprädikat im Rahmen der Programmiersprache Datalog und der domänenspezifischen Sprache Roxx definiert und vorgestellt. Dazu wurden außerdem der theoretische Hintergrund von Datalog und insbesondere die Berechnung rekursiver Datalog Programme erläutert. Des Weiteren folgten Definition und Motivation domänenspezifischer Sprachen und anschließend eine Vorstellung von domänenspezifischen Sprachen im Kontext der Programmiersprache Groovy. Zuletzt erfolgte die Beschreibung der Implementierung des Partitionsprädikats und der im Rahmen dieser Arbeit entwickelten Erweiterungen um die Mehrfachsortierung und der Partitionsfunktion des eindeutigen Rankings.

Bezüglich zusätzlicher Erweiterungen des Partitionsprädiakts gibt es im Rahmen der Rangvergabe einige Arten des Rankings, welche je nach Anwendungsfall und Bedarf noch zur WorkforcePlus Applikation hinzugefügt werden könnten. Dazu zählt beispielsweise das Ranking in der Art wie es bei Sportwettkämpfen üblicherweise vollzogen wird. Dabei erhalten Tupel, welche gleiche Werte aufweisen wie beim lückenlosen Ranking denselben Rang, allerdings wird im Anschluss ein Rang übersprungen. Bei drei Tupeln, wobei zwei den gleichen Wert besitzen und das dritte nach diesen beiden Tupeln einzusortieren ist, würde sich hierfür dann die Rangfolge 1,1,3 ergeben. Doch auch abgesehen von Rangfunktionen ist grundsätzlich jegliche Funktion denkbar, welche ausgehend von Werten der Tupeln in einer Partition neue Werte für jedes einzelne der in der Partition befindlichen Tupel errechnet.

Eine weitere nützliche Erweiterung der Sprache Roxx im Allgemeinen, wäre die Implementierung einer Entwicklungsumgebungserweiterung. Diese Erweiterung sollte der Syntaxhervorhebung und der automatischen Codevervollständigung mächtig sein. Dies würde den operativen Gebrauch von Roxx deutlich unterstützen und das Schreiben des Roxx-Codes vereinfachen.

# 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] P. Avgustinov, O. de Moor, M. Peyton Jones, and M. Schäfer, "QL: Object-oriented Queries on Relational Data," 2016.

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

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

[5] M. Bender, S. Berckey, M. Elberfeld, and J. Herbers, "Real-world staff rostering via branch-and-price in a declarative framework," in *Operations Research Proceedings 2018*. Springer International Publishing, pp. 445–451.

[6] INFORM, "Sprint Review - 22 S15 - Declare It!" Nov. 2022.

[7] *Groovy*, zuletzt geprüft am 21.02.2023. [Online]. Available: http://www.groovy-lang.org/

[8] P. J. Layzell, "The History of Macro Processors in Programming Language Extensibility," *The Computer Journal*, vol. 28, no. 1, 1985.

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

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

[11] S. Abiteboul, R. Hull, and V. Victor, *Foundations of Databases*, 1995.

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

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

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

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

[16] INFORM, "Overview Backend," Jul. 2020.

[17] *Interface Comparator*, zuletzt geprüft am 10.02.2023. [Online]. Available: https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Comparator.html

[18] *Class Pair*, zuletzt geprüft am 13.02.2023. [Online]. Available: https://docs.oracle.com/javase/9/docs/api/javafx/util/Pair.html

[19] C. Heinisch, F. Müller-Hofmann, and J. Goll, *Java als erste Programmiersprache*, 2000.