THM
TECHNISCHE HOCHSCHULE MITTELHESSEN

Projektbericht

Projektphase bei INFORM GmbH - GB 70

Oktober 2022 - Dezember 2022

Madelaine Hestermann
madelaine.hestermann@mni.thm.de

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

23. Dezember 2022
Technische Hochschule Mittelhessen, Gießen

Inhaltsverzeichnis

1 Einleitung **1**

2 Kurzvorstellung des Projektpartners **1**

3 Projekt & weitere Aufgaben **1**
3.1 WorkforcePlus . . . 1
3.1.1 Clients . . . 2
3.1.2 Back-end . . . 2
3.1.3 Roxx . . . 2
3.2 Theoretischer Hintergrund von Roxx . . . 3
3.2.1 Deduktive Datenbanken . . . 3
3.2.2 Prädikate in Datalog & Roxx . . . 3
3.2.3 Das Partition Predicate . . . 4
3.2.4 Interpretation von Roxx . . . 7
3.3 Aufgabenstellung . . . 7
3.3.1 Anwendungsfälle . . . 7
3.3.2 Ausgangssituation . . . 8
3.3.3 Motivation . . . 11
3.4 Herangehensweise . . . 11
3.4.1 Mehrfachsortierung . . . 11
3.4.2 Datenstruktur . . . 12
3.5 Umsetzung . . . 13
3.5.1 Frameworks im Back-end . . . 13
3.5.2 Architektur des Partition Predicates im Back-end . . . 13
3.5.3 Implementierung . . . 15
3.6 Weitere Aufgaben . . . 17
3.6.1 Ersetzen des Makros . . . 17
3.6.2 Unique-Ranking . . . 17

4 Ausblick **21**

Anhang **I**

Abbildungsverzeichnis **II**

Tabellenverzeichnis **III**

Literatur **IV**

1 Einleitung

Dieser Bericht handelt von dem Projekt "Multiple Rank Orderings in Partition Predicate", das im Rahmen der Projektphase erstellt wurde.
Der Gegenstand des Projekts ist die Erweiterung des Partition Predicates (dt.: Partitions-Prädikat) der Modellierungssprache Roxx um die Möglichkeit die Sortier-Operation des Partition Predicates nach beliebig vielen Attributen auszuführen.

Das Projekt wurde bei der INFORM GmbH durchgeführt und von Dimitri Bohlender betreut. Die Projektphase startete am 01.10.2022 und endete am 31.12.2022.

2 Kurzvorstellung des Projektpartners

Die INFORM GmbH wurde 1969 gegründet und beschäftigt heute mehr als 900 Mitarbeitende weltweit.
Der Hauptstandort der INFORM GmbH befindet sich im nordrhein-westfälischen Aachen. Weitere Standorte befinden sich in Köln, Düsseldorf, Frankfurt, Amsterdam und Brüssel. Außerdem hat die INFORM GmbH weltweite Partnergesellschaften in den USA, Chile, Brasilien, Portugal, Singapur, sowie in Australien.
Kunden der INFORM GmbH sind mittelständische Unternehmen, sowie Großkonzerne, welche von der INFORM GmbH Softwarelösungen für Unternehmensprozesse wie Absatzplanung, Produktionsplanung, Personaleinsatz, Logistik und Transport, Organisation von Lagerbeständen und Supply Chain Management[1], sowie Betrugsabwehr bei Versicherungen, Telekommunikation und im Zahlungsverkehr beziehen[2].

Der Geschäftsbereich, in welchem dieses Projekt entstanden ist, GB 70, entwickelt Software im Bereich der Personaleinsatzplanung. Die Softwarelösungen bieten dem Kunden mit der Applikation WorkforcePlus Unterstützung in ihrem Workforce Management[2] und verhelfen dem Kunden damit zu einem optimierten und bedarfsorientierten Personaleinsatz.

3 Projekt & weitere Aufgaben

In diesem Abschnitt werden die Applikation, die dem Projekt zugrunde liegt, einige Grundlagen, sowie die Aufgabenstellung erläutert, um ein theoretisches Fundament für das Projekt zu liefern. Anschließend wird die konkrete Umsetzung des Projekts und schließlich weitere Aufgaben, die sich zusätzlich ergaben, vorgestellt.

3.1 WorkforcePlus

Die Software, welche den Kern des Projekt darstellt, ist WorkforcePlus. Wie eingehend schon beschrieben, ist WorkforcePlus eine Applikation zur optimierten und automatisierten Personaleinsatzplanung.

[1] Management eines Netzwerks miteinander verbundener Betriebe, die an der Bereitstellung fertiger Produkte und Dienstleistungen an den Endverbraucher beteiligt sind[1, S. 63 ff.]
[2] Einteilung des richtigen Personals, mit der richtigen Qualifikation, zum richtigen Zeitpunkt, am richtigen Ort[3]

Die Architektur von WorkforcePlus lässt sich, wie in der folgenden Abbildung visualisiert, in drei Komponenten aufteilen. Sie besteht aus einem Client, dem Back-end und einem entsprechenden Roxx Modell.

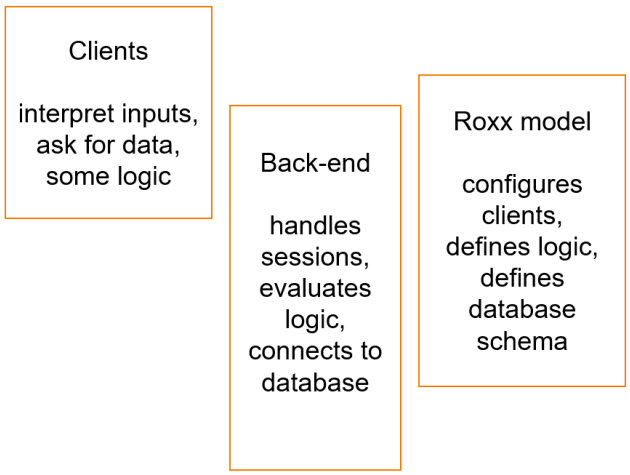

Abbildung 1: Komponenten von WorkforcePlus[4]

3.1.1 Clients

WorkforcePlus bietet mit einer Applikation für Windows Betriebssysteme (Abbildung 9), dem webbasierten WorkforcePlus Base Data Editor (Abbildung 10) und dem ebenfalls webbasierten WorkforcePlus Employee Portal (Abbildung 11) drei unterschiedliche Clients.

Die Clients bieten eine direkte Schnittstelle zwischen dem Benutzer und WorkforcePlus, indem sie auf Benutzeranfragen reagieren, Daten aus dem Back-end lesen, interpretieren und dann auf einer grafischen Benutzeroberfläche darstellen.

3.1.2 Back-end

Das WorkforcePlus Back-end ist ein überwiegend in der Programmiersprache Java geschriebener Anwendungs-Webserver, der über HTTP Requests und Responses mit den Komponenten von WorkforcePlus kommuniziert.
Die Implementierung der HTTP Kommunikation des Back-ends erfolgt durch den Open-Source-Webserver Apache Tomcat.

Des Weiteren implementiert das Back-end die Geschäftslogik von WorkforcePlus, organisiert seine Benutzerumgebung in Sessions und handelt diese und verwaltet die Daten der WorkforcePlus Applikation durch das Schreiben und Lesen von und in externe Datenbanken wie PostgreSQL und Oracle Database[4].

3.1.3 Roxx

Roxx Modelle bilden die Grundlage der WorkforcePlus Applikation und definieren sowohl Daten, die in den Datenbanken gespeichert werden, als auch die vom Back-end berechnete Logik und bestimmen damit die konkreten Konfigurationen der WorkforcePlus Clients[4].

Roxx ist eine von GB 70 entwickelte DSL[3] und ein Dialekt der deklarativ logischen Programmiersprache Datalog[5].
Sie erlaubt daher eine deklarative Sichtweise auf die zu modellierenden Daten, erinnert syntaktisch aber an imperative Programmiersprachen wie beispielsweise Java oder C.

Die folgende Abbildung stellt ein Beispiel für die Roxx Syntax dar.

```
intensionalPredicate {
        plannedInterval('empl', 'date', 'interval') {
                plannedShifts('empl', 'date', 'shift')
                shiftIntervals('shift', 'interval')
        }
}
```

Abbildung 2: Example Roxx Syntax[5]

3.2 Theoretischer Hintergrund von Roxx

Roxx gilt als Dialekt der Programmiersprache Datalog und gehört ebenfalls dem deklarativ logischen Sprachparadigma an und auch ihr liegt ein deduktives Datenbanksystem als Kern zugrunde[5].

3.2.1 Deduktive Datenbanken

Deduktive Datenbanken sind im Allgemeinen dazu in der Lage etwa relationale Datenbanken durch regelbasierte Ausdrücke zu erweitern, sodass es möglich ist durch einige solcher Regeln eine große Faktenmenge abzuleiten und darzustellen[6, S. 1].

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

3.2.2 Prädikate in Datalog & Roxx

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

Sowohl Datalog als auch Roxx verfügen über zwei unterschiedlich zu betrachtende Arten von Prädikaten: extensionale und intensionale Prädikate.
Extensionale Prädikate entsprechen den Inhalten der Tabellen der Quelldatenbank (Beispielsweise eine relationale Datenbank, die als Grundlage verwendet wird).
Intensionale Prädikate hingegen entsprechen hergeleiteten Daten, die sich aus errechneten Regeln ergeben[8, S. 114].

[3] Abk.: Domain Specific Language

3.2.3 Das Partition Predicate

Das Partition Predicate bildet den Kern des Projektes und stellt ein fortgeschrittenes Feature der Sprache Roxx dar[5].
Es ist eine Art intensionales Prädikat, das sich syntaktisch und semantisch aber von anderen intensionalen Prädikaten abhebt.

Die folgende Abbildung zeigt die Syntax eines beispielhaften Partition Predicates.

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

Abbildung 3: Beispiel Partition Predicate[5]

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

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

Dabei gibt die Zeile

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

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

Die Zeile

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

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

Zuletzt gibt es eine Operation, die ein Partition Predicate ausführen kann.
Über die *Rank()* Operation ist es dem Modellierer möglich innerhalb der einzelnen Partitionen nach einem Attribut zu sortieren. Jedes Element, erhält daraufhin eine von 0 beginnend aufsteigende Nummerierung, die dann dem *Rank*, also einem Rang entspricht.
Hierzu wählt der Modellierer einen beliebigen Bezeichner und weist diesem dann das Ergebnis

der Rank Operation zu.
Im oben aufgeführtem Beispiel entspricht das folgender Zeile:

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

Die Sortierung nach dem gewählten Attribut kann mit der Funktion *OrderAsc()* sowohl aufsteigend als auch absteigend durch die Funktion *OrderDesc()* ausgewertet werden[5].

Zur Veranschaulichung der Funktionalität des Partition Predicates soll das in Abbildung 3 aufgeführte Partition Predicate im Folgenden anhand von Beispieldaten evaluiert werden.

Das Prädikat, das als Grundlage für das Partition Predicate verwendet wurde, *employeesBirthday*, enthält drei Attribute: die Gruppe von Arbeitnehmern *emplGroup*, der entsprechende Mitarbeiter *employee*, sowie der Geburtstag des Mitarbeiters *birthday*.
Daraus ergeben sich für das Prädikat *employeesBirthday* Beispieldaten in einer Tabelle wie in Abbildung 4 aufgeführt.

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

Abbildung 4: Beispieldaten [5]

Wie in Abbildung 5 erkennbar ist, wurden die Beispieldaten nun nach dem Attribut *emplGroup* partitioniert.
Um die einzelnen Partitionen in der Abbildung besser erkennen zu können, wurden die zu einer Partition zugehörigen Zeilen und Spalten außerdem in einer einheitlichen Farbe markiert.

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

Abbildung 5: Beispieldaten nach der Partitionierung [5]

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

Wichtig zu beachten ist allerdings auch der Fall, bei welchem derselbe Wert des Sortierattributs für mehrere Spalten, beziehungsweise Mitarbeiter, auftritt.
Dieser ist in der unten aufgeführten Abbildung in den Spalten der beiden Mitarbeiter *Ron* und *Tina* zu beobachten.
Beide Mitarbeiter besitzen den Wert 03.04.1986 für das Attribut *birthday*. Die Konsequenz daraus ist, dass beide nach dem sogenannten *Dense-Ranking* denselben Rang, den Startwert 0, erhalten und die nächste unterscheidbare Zeile den um eins erhöhten Rang erhält.
Zu beachten ist also auch, dass beim *Dense-Ranking* nach der mehrfachen Vergabe gleicher Ränge kein Rang übersprungen wird.

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

Abbildung 6: Beispieldaten nach Evaluierung der *Rank*-Operation[5]

3.2.4 Interpretation von Roxx

Technisch betrachtet steckt hinter Roxx eine Logik, die im Back-end implementiert wird.
Dabei ist zu beachten, dass Roxx Code nicht von einem Parser gelesen und verarbeitet wird, um dann in einem weiteren Schritt kompiliert zu werden.
Viel mehr wird Roxx Code als Groovy Code interpretiert und ausgewertet.
Dies geschieht indem das Back-end einige Groovy Methoden definiert, die den erlaubten Roxx Konstrukten, wie beispielsweise dem Partition Predicate, syntaktisch entsprechen und deren Semantik implementieren.
Letztlich ist valider Roxx Code also ebenfalls valider Groovy Code.

3.3 Aufgabenstellung

Die Aufgabenstellung für das Projekt ergibt sich aus konkreten Anwendungsfällen, in welchen das Partition Predicate eingesetzt wird um bestimmte Modelle von WorkforcePlus für den Kunden generieren zu können. Wesentlich dabei ist, dass diese Anwendungsfälle fordern, dass die Sortierung der *Rank()* Operation nicht nur nach einem einzigen Attribut geschieht sondern nach beliebig vielen.

3.3.1 Anwendungsfälle

Im Rahmen der WorkforcePlus Applikation gibt es einige Anwendungsfälle, die das Sortieren nach mehreren Attributen fordern. Im Allgemeinen ist der praktische Nutzen einer solchen Erweiterung hoch.
Ein konkreter Anwendungsfall etwa sind Requests der Mitarbeiter im WorkforcePlus Employee Portal, wie sie in Abbildung 7 zu sehen sind.

Die Abbildung zeigt einen kleinen Ausschnitt, das Wesentliche für den zu erläuternden Anwendungsfall, der grafischen Benutzeroberfläche des WorkforcePlus Employee Portals.
Abgebildet sind die Requests, also Anfragen auf beispielsweise Schicht Wechsel (Shift Swap) mit anderen Mitarbeitern, die ein Mitarbeiter innerhalb der Applikation sehen und organisieren kann.
Hinter diesen Requests steckt im Roxx Modell das Partition Predicate.
Um die Requests in der Benutzeroberfläche sinnvoll anzeigen zu können, werden sie nach zwei Attributen sortiert: zunächst nach dem Status, der auf der Oberfläche bildlich durch das Haken Icon, sowie die Fragezeichen Icons ausgedrückt wird, und dann nach dem Bearbeitungszeitpunkt, der in der Abbildung als unterste Notiz jedes Requests aufgeführt wird.
Diese mehrfache Sortierung ist einzig mit dem Partition Predicate, so wie es bislang erläutert wurde, nicht möglich, da es nicht in der Lage ist mehr als ein Attribut zur Sortierung entgegen zu nehmen und zu verarbeiten.

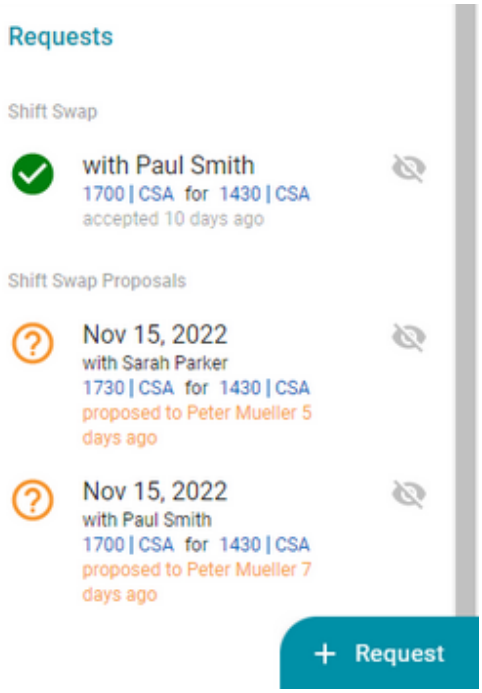

Abbildung 7: Mitarbeiter Requests der Applikation WorkforcePlus Employee Portal[9]

Tatsächlich wurden Modelle mit Sortierung nach mehreren Attributen allerdings schon vor der Umsetzung des Projektes erstellt. Wie genau dies gelingen konnte, wird im kommenden Abschnitt erläutert.

3.3.2 Ausgangssituation

Vor Beginn der Bearbeitung des Projekts konnte das Partition Predicate das Sortieren nach mehreren Attributen durch ein sogenanntes Makro realisieren.

Makros sind Konstrukte, die hauptsächlich dazu dienen Roxx Code zu modularisieren.
Makros beinhalten Roxx Code in einer eigens für sie erstellten Datei und können durch einen Aufruf in einer anderen Roxx Datei verwendet werden.
So lässt sich insbesondere die Duplikation von Roxx Code, der in vielen unterschiedlichen Roxx Modellen vorkommt, vermeiden[5].

Technisch betrachtet bestehen Makros aus Groovy Closures, also aus Closures, welche in derselben Sprache geschrieben sind wie auch Roxx selbst.
Daher kann man Makros grundsätzlich aus beliebigen Anteilen von sowohl Roxx als auch Groovy Code zusammenbauen.

Diese Eigenschaften hat man genutzt, um das Sortieren nach mehreren Attributen beim Partition Predicate zu simulieren.

Das sogenannte *multiRankPredicate* Makro besteht aus einer Closure, die als Argumente zunächst alle wesentlichen Bestandteile des Partition Predicates entgegennimmt: den Namen des Prädikats, der Name des Grundprädikats, auf welchem die Partitionierung basieren soll, eine Liste aller Attribute des Grundprädikats, eine Liste derjenigen Attribute, die dann für die Partitionierung

genutzt wird und zuletzt eine n-stellige Map aller Attribute, die als Sortierattribute genutzt werden sollen, gemappt mit der Information, ob die Sortierung nach dem jeweiligen Attribut auf- oder absteigend angewandt wird.

Anschließend wird der kombinierte Rang aus der Ausführung von n Partition Predicates, die jeweils mit einem einzigen Sortierattribut ausgeführt werden, zusammengesetzt.
Dies geschieht durch die Berechnung der Linearkombination

$$\sum_{i=0}^{n} rank_i(\prod_{j=i}^{n-i}(maxRank_{j+1}+1)).$$

Die Variablen $rank_i$ sind dabei das Ergebnis der n Partition Predicates.
Für die Berechnung der Linearkombination ebenfalls relevant sind die Variablen $maxRank_{j+1}$, die jeweils den höchsten errechneten Rang je Sortierattribut darstellen.

Das Partition Predicate selbst verfügt über keine Operation, die es erlaubt einen maximalen Wert zu errechnen, da es vor Beginn des Projekts einzig über die *Rank()* Operation verfügte. Roxx besitzt aber ein anderes High-Level Prädikat, das hierzu in der Lage ist: das Aggregation Predicate.

Mit dem Aggregation Predicate ist es möglich nach gegebenen Attributen zu aggregieren, Elemente also in Gruppen zusammenzufügen.
Außerdem besitzt es eine Reihe von Operationen, die es dann auf den Elemente der einzelnen Gruppen ausführen kann, unter anderem eine *Max()* Operation, mit welcher Maximalwerte errechnet werden können[5].

In dem Makro *multiRankPredicate* werden also zunächst mithilfe einer Groovy Schleife n Partition Predicates und n Aggregation Predicates durchlaufen, jeweils eines für jedes Sortierattribut. Mit Verlassen dieser Schleife liegen alle benötigten Variablen $rank_i$ und $maxRank_{j+1}$ vor und es kann zum eigentlichen Berechnen der Linearkombination übergegangen werden.
Hierzu wird in dem Makro ein intensionales Prädikat erstellt.
Dieses bindet zunächst die in dem Aggregation Predicate errechneten Werte, in n Variablen $maxRank_{j+1}$ und erzeugt mit dem Inkrementieren dieser um den Wert 1 die einzelnen Faktoren $maxRank_{j+1}+1$ zur Berechnung des Produkts der Linearkombination.
Anschließend wird aus diesen einzelnen Faktoren der gesamte Koeffizient zusammengesetzt, womit das Produkt komplett ist.
Zuletzt fügt es die Koeffizienten dann mit der entsprechenden $rank_i$ Variable zusammen, um den Summanden $Summand_i$ zu erstellen und berechnet zuletzt die Summe aus allen Summanden.
Um das Ergebnis aus der Linearkombination bei Anwendung des intensionalen Prädikats verwenden zu können, wird es dann noch mit dem Attribut des Prädikats, welches dem kombinierten Rang entspricht, gleichgesetzt und daran gebunden.

Aus Gründen der Stringenz wird dieser kombinierte Rang zuletzt noch normalisiert, da die Berechnung mithilfe der Linearkombination zwar zu einer korrekten Sortierung führt, aber gleichzeitig auch zwei Eigenschaften verletzen kann, die der Rang erfüllen soll.
Gefordert wird, dass der Rang stets bei 0 beginnt und alle Ränge zusammenhängend, also ohne Lücken, nummeriert sind.
Die Normalisierung geschieht durch ein einfaches Partition Predicate, dass das intensionale

Prädikat, welches den kombinierten Rang errechnet, als Grundlage nimmt, nach denselben Attributen wie dieses partitioniert und dann aufsteigend nach dem kombinierten Rang sortiert, sodass die Reihenfolge unverändert erhalten bleibt, aber die geforderten Eigenschaften, die die Anwendung eines Partition Predicates grundsätzlich garantiert, wieder hergestellt sind.

Um die Funktionsweise der Linearkombination bei der Berechnung des kombinierten Rangs besser nachvollziehen zu können, soll sie im Folgenden für ein Beispiel berechnet werden.
Das Beispiel bezieht sich auf die Daten in Abbildung 5.
Die Annahme ist nun jedoch, dass nicht mehr nur nach dem Attribut *birthday* sortiert werden soll, sondern zusätzlich nach dem Attribut *employee*.

Zu beachten ist hierbei, dass sich für die Arbeitnehmergruppen *EmplGr3* sowie *EmplGr1* bereits aus der Sortierung nach dem Attribut *birthday* allein jeweils ein einzigartiger Rang für jedes Element ergibt.
Da die darauffolgende Sortierung nach dem Attribut *employee* in diesen Gruppen ohnehin zu keinem anderen Ergebnis führt, befasst sich die folgende Beispielrechnung also nur mit den Daten der Arbeitnehmergruppe *EmplGr2*.

Die folgende Tabelle zeigt die Ausgangsdaten der Mitarbeiter, die für die Berechnung genutzt werden sollen.
Sie besteht aus dem Namen des jeweiligen Mitarbeiters, des Geburtstages und der bereits errechneten Ränge, die entsprechend aus der Anwendung zweier Partition Predicates mit dem bereits erläuterten *Dense Ranking* resultieren.
Zuletzt ist der zu erwartende Gesamtrang aufgeführt, also der Rang, der sich ergibt, wenn zunächst nach dem Attribut *birthday* und anschließend für alle Elemente mit bis dann gleichem Rang, nach dem Attribut *employee* sortiert wird.
Wie zu erkennen ist, ist die Konsequenz dieser zweiten Sortierung die, dass die Ränge von Ron und Tina abweichende Werte enthalten, da nach der Sortierung nach dem zweiten Attribut Ron einen niedrigeren Rang erhält, als Tina.

Tabelle 1: Mitarbeiterdaten der Arbeitnehmergruppe *EmplGr2* (Eigene Tabelle)

employee	**birthday**	$\boldsymbol{rank_0}$	$\boldsymbol{rank_1}$	**zu erwartender Gesamtrang**
Ron	03.04.1986	0	1	0
Tina	03.04.1986	0	2	1
Haley	05.09.1988	1	0	2

Angewandt auf zwei Sortierattribute ergibt sich aus der Formel der Linearkombination folgende Gleichung:

$$combinedRank = rank_0 * (maxRank_1 + 1) + rank_1 .$$

Die Variable $maxRank_1$ ist dabei für alle Mitarbeiter gleich gesetzt als der höchste aller $rank_1$ und hat den Wert 2.
Die anderen beiden Variablen $rank_0$ und $rank_1$ können für jeden Mitarbeiter aus der jeweiligen Zeile in der Tabelle gelesen werden, sodass sich dann folgende Rechnungen ergeben:

Ron: $0 * (2 + 1) + 1 = 1$ Tina: $0 * (2 + 1) + 2 = 2$ Haley: $1 * (2 + 1) + 0 = 3$

Als Ergebnis der Berechnung der Linearkombination erhält Ron also den Wert 1, Tina den Wert 2 und Haley den Wert 3.

Dies weicht von unseren zu erwartenden Ergebnis nur in einem Punkt ab: die Zählung beginnt bei 1, sodass jeder Rang um den Wert 1 zu hoch ist.
Wie bereits erläutert, kann dies durch die Normalisierung des Rangs behoben werden, sodass die Linearkombination einzig die richtige Reihenfolge errechnen muss, was in diesem Beispiel offenbar geschieht.

3.3.3 Motivation

Die Motivation dafür das Sortieren nach einer beliebigen Anzahl von Attributen als Grundfunktionalität für das Partition Predicate zu implementieren, ergibt sich aus der Komplexität und aus der Ineffizienz des *multiRankPredicate* Makro.

Mithilfe des Makros ist es notwendig für die Berechnung eines kombinierten Rangs bei n Attributen $n + 1$ Partition Predicates, n Aggregation Predicates und ein intensionales Prädikat, das wiederum einige Berechnungen durchführt, zu durchlaufen.

Ziel des Projektes ist es den Aufwand auf das Ausführen eines einzigen Partition Predicates zu beschränken.
Wie genau dies umgesetzt wurde, wird im folgenden Abschnitt besprochen.

3.4 Herangehensweise

Die Funktionsweise des Partition Predicates ist im Back-end in der Programmiersprache Java in einer Vielzahl an Klassen implementiert, welche die Struktur und die Funktionalität des Partition Predicates definieren.

Zu Beginn des Projektes mussten zunächst zwei Implementierungsdetails für die wesentlichen Änderungen, die am Partition Predicate vorzunehmen waren, beschlossen werden.
Dies war zum einen die konkrete Implementierung der mehrfachen Sortierung und zum anderen die Datenstruktur, die nun nicht mehr nur ein einzelnes Sortierattribut definieren sollte, sondern eine unbestimmte Menge davon.

3.4.1 Mehrfachsortierung

Vor Beginn des Projekts war die Sortierung mithilfe des funktionalen Java Interfaces *Comparator* implementiert und basierte hauptsächlich auf dessen Methoden *comparing()* und *reversed()*, wobei *comparing()* einen *Comparator* erzeugt, der nach einer natürlichen Ordnung sortiert und *reversed()* dazu in der Lage ist, diese natürliche Ordnung umzudrehen, sodass dadurch das absteigende Sortieren realisiert werden kann[10].

Das Interface *Comparator* bietet insgesamt eine ganze Reihe an nützlichen Methoden, welche eine Vielzahl an unterschiedlichen Möglichkeiten zur Sortierung bietet.

So konnte mithilfe der offiziellen Java Dokumentation von *Oracle* sehr schnell eine passende

Methode gefunden werden, die genau das umsetzt, was benötigt wurde um die Mehrfachsortierung umzusetzen.

Mit der Methode *thenComparing* ist es möglich mehrere *Comparator* zu sequenzieren, wobei mit jedem Aufruf der Methode ein *Comparator* mit dem entsprechenden Sortierattribut als Argument übergeben und zusätzlich der Methodenaufruf der Methode *reversed()* angehängt werden kann, je nachdem wie die gewünschte Sortierrichtung ist.
Laut Dokumentation erfolgt die tatsächliche Sortierung nach dem *Comparator* der *thenComparing* Methode nur dann, wenn der *Comparator*, auf welchen die Methode angewendet wurde, zwei zu vergleichende Elemente auf Grundlage seines angegebenen Sortierattributs als gleich erachtet.
Andernfalls kommt ein weiterer Sortierdurchgang nicht in Frage[10].

Durch diese Funktionalität überzeugte die Methode gegenüber des zuvor verwendeten Makros, das unabhängig von der eigentlichen Notwendigkeit immer sämtliche Rechenschritte vollzog.

3.4.2 Datenstruktur

Die Wahl der am besten geeigneten Datenstruktur erfolgte hingegen nicht so eindeutig.

In einem ersten Durchgang wurden sowohl das eigentliche Sortierattribut, als auch die Information darüber ob die Sortierung auf- oder absteigend erfolgen soll, in zwei voneinander getrennten *ArrayLists* implementiert.

Die Trennung dieser beiden eigentlich zusammengehörigen Elemente führt aber zu zwei Schwierigkeiten.
Einerseits muss akribisch darauf geachtet werden, dass zusammengehörige Attribute und Sortierrichtungen auch zusammenbleiben. Bei Bearbeitung der Listen, darf es also nie zu Wechseln oder Tauschen der Indizes kommen.
Außerdem muss ebenfalls darauf geachtet werden, dass beide Listen simultan gleich lang sind. Zwei unterschiedlich lange Listen bedeuten in jedem Fall, dass sie inkonsistent sind und keine korrekte Zuordnung von Sortierrichtung zu Attribut gewährleistet werden kann.

Aufgrund dieser Probleme fiel die Entscheidung daraufhin auf die Datenstruktur des *Pair*.

Die Java Klasse *Pair* besteht aus einem Paar zweier Dinge, die in der offiziellen Java Dokumentation von *Oracle* als *key* und *value* bezeichnet werden.
Diese beiden Elemente können unterschiedliche oder auch denselben Datentypen besitzen und werden vom Konstruktor eines *Pair* Objektes schon bei der Instanziierung dessen gefordert.
Des Weiteren besitzt ein *Pair* mit den beiden Methoden *getKey()* und *getValue()* Möglichkeiten auf die einzelnen Bestandteile des im *Pair* gespeicherten Paares zuzugreifen und dient daher sehr gut als Datenstruktur zweier zusammengehöriger Elemente.

Ein einzelnes *Pair* hält aber nur genau zwei Dinge und ist in dieser Form noch nicht dazu in der Lage eine beliebige Anzahl an Paaren zu verwalten.

Bereits für andere Anwendungsfälle wurde im Back-end aber eine Datenstruktur entwickelt, die einer *ArrayList* an *Pair* Objekten entspricht: die *Pairs*. Die Klasse *Pairs* erfüllt damit

alle Anforderungen, die sich für die Datenstruktur ergeben. Sie garantiert eine konstante Zusammengehörigkeit zwischen Sortierattribut und Sortierrichtung und erlaubt eine dynamische Anzahl an Elementen.

Zuletzt sind einige Überlegungen aufgekommen, ob an einzelnen komplexeren Stellen eine alternative Datenstruktur statt der *Pairs* nicht auch in Frage kommt, da die Methoden *getKey()* und *getValue()* sehr allgemein sind, keine sonderlich sprechenden Namen haben und es dem Programmierer somit nicht direkt ersichtlich ist, welche konkreten Werte die Methoden zurückgeben.
Allerdings bestünde diese Alternative wieder aus Listen, sodass man sich mit den ausgangs beschriebenen Problemen konfrontiert sah und außerdem einige praktische Funktionalität der *Pairs* Klasse nicht mehr hätte nutzen können, sodass diese Überlegungen schließlich verworfen wurden.

3.5 Umsetzung

Im folgenden Abschnitt wird die konkrete Umsetzung des Projekts erläutert.
Dabei werden die im Back-end verwendeten Frameworks, die Architektur des Back-ends sowie die angewandte Test Strategie vorgestellt.

3.5.1 Frameworks im Back-end

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

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

3.5.2 Architektur des Partition Predicates im Back-end

Die Architektur des Partition Predicate im Back-end umfasst alle in der Abbildung 8 aufgeführten Klassen.

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

Grundsätzlich ist das Back-end in einer Schichtenarchitektur organisiert und besitzt vier Komponenten.
Diese sind die Ebene der DSL, auf welcher die genaue Syntax der Sprache Roxx definiert wird.
Auf der DSL basiert die Ebene der *Runtime*, der Laufzeit, welche der Semantik des Roxx Programms entspricht.
Hier werden logische Regeln organisiert, die etwa gültigen Datalog Regeln entsprechen können.

Basierend auf diesen Regeln folgt darauf die Ebene des Compilers, der zwar kein richtiger Compiler im herkömmlichen Sinne ist, da dieser keinen Maschinencode erzeugt, allerdings ist diese Ebene dafür zuständig, dass Elemente wie Regeln oder sogenannte *Commands* durch entsprechende Klassen gebaut werden.
Die letzte dieser vier Komponenten ist die Ebene Der Datenbank, *Database*, auf welcher ein Prädikat nur noch aus Zeilen und Spalten besteht, welche die im Prädikat enthaltenen Tupel darstellen.
Anhand dieser Zeilen und Spalten wird hier die Sortierlogik implementiert und der Roxx Code somit interpretiert.

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

Dazu zählen zwei Klassen der *Runtime* Ebene.
Eine der beiden, *ToDatalogWithIdsVisitor* generiert aus Roxx Regeln äquivalente Datalog Regeln.
Die andere der beiden, *RuleUtils* stellt eine Hilfsklasse dar, die Hilfsmethoden für Regeln über verschiedene Arten von Prädikaten hinweg bereitstellt.

Des Weiteren sind für die Änderungen am Partition Predicates ebenfalls die sogenannten *Constraints* und ein vom Back-end erzeugter Graph relevant.

Constraints meint sowohl *Completeness Constraints* als auch *Uniqueness Constraints*.
Diese sind Einschränkungen der Anzahl einzelner Elemente eines Prädikats, die Kardinalitäten darstellen indem sie angeben wie viele Tupel gültiger Elemente mindestens oder maximal bei Auswertung eines Prädikats vorkommen dürfen.
Completeness Constraints dienen hierbei dazu eine Untergrenze der Anzahl zu definieren, wohingegen *Uniqueness Constraints* dazu dienen eine Obergrenze der Anzahl zu definieren[5].

In einem späteren Abschnitt werden *Uniqueness Constraints* in Bezug auf das Partition Predicate näher erläutert. Die *Completeness Constraints* werden mangels Relevanz im Weiteren vernachlässigt.

Der oben genannte Graph wiederum ist ein Graph, welcher in der Lage ist Abhängigkeiten von Prädikaten zueinander zu modellieren und darzustellen.

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

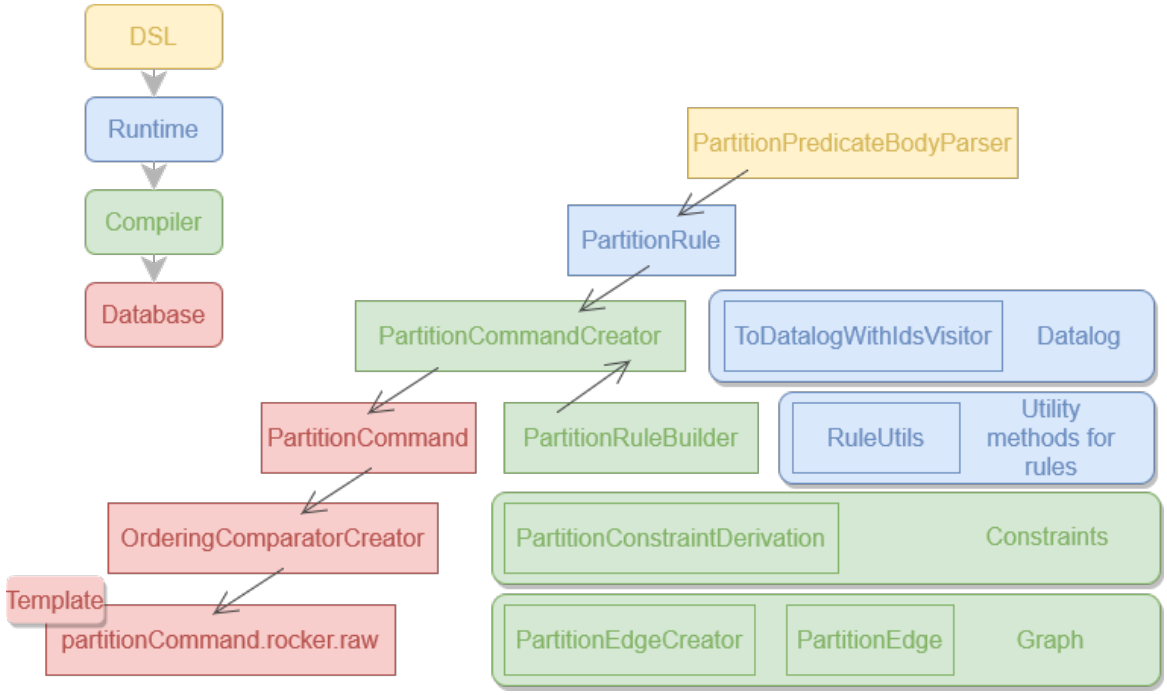

Abbildung 8: Architektur Partition Predicate (Eigene Abbildung)

3.5.3 Implementierung

Bei der Implementierung der geplanten Änderungen wurde systematisch vom Kern des Backends ausgehend gearbeitet.
Begonnen wurde also mit den Anpassungen auf der Ebene der *Database*, anschließend wurden die entsprechenden Klassen auf der Ebene des Compilers bearbeitet, daraufhin die *Runtime* Ebene und zuletzt die Klasse welche die Syntax auf DSL Ebene definiert.
Dank dieser systematischen Vorgehensweise konnte die Arbeit am Projekt sehr dynamisch gestaltet werden und mit jedem Abschluss einer Ebene wäre es möglich gewesen das Projekt zu unterbrechen oder unvollständig zu beenden, falls unerwartete zeitliche oder technische Schwierigkeiten aufgetreten wären.

Database Der größte Implementierungsaufwand im Projekt ergab sich im Bereich der *Database*.
Hier wurde die neue Sortierlogik mithilfe der *thenComparing* Methode des *Comparator* Interfaces, wie bereits erläutert, implementiert.

Vor Beginn des Projekts befand sich sämtliche Logik zur Sortierung in dem Template auf unterster Ebene.
Da das Sortieren mit Einführung des mehrfachen Sortierens allerdings an Komplexität gewann, wurde beschlossen die Logik in eine neu erstellte Klasse, *OrderingComparatorCreator*, auszulagern.
Diese Klasse enthält nun die Logik, welche mit Aufruf einer statischen Methode im Template nur noch genutzt wird.

Des Weiteren wurden hier die Felder für das einzelne Sortierattribut und die Sortierrichtung im *Command* angepasst und zur Datenstruktur der *Pairs* umgewandelt.

Compiler & Runtime Vereinfacht betrachtet musste auf Ebene von Compiler und *Runtime* ausschließlich jeweils die Datenstruktur angepasst werden, da beide Ebenen nur unterschiedliche Abstraktionsformen eines Prädikats darstellen und keine eigene Logik implementieren.

DSL Auf dieser Ebene wurde die neue Syntax für das Partition Predicate definiert.
Die Definition der Syntax gelingt praktisch indem in der Klasse *PartitionPredicateBodyParser* Groovy Methoden genau so bezeichnet werden, wie es die Syntax von Roxx vorgibt.
Beim Starten eines Roxx Modells werden im Back-end diese Methoden dann schlicht aufgerufen und ausgeführt.

Für die Entscheidung wie die Syntax konkret aussieht, gab es zwei Anforderungen.
Einerseits sollte sie möglichst intuitiv und einfach sein und andererseits war gefordert, dass die Syntax außerdem abwärtskompatibel war, damit nicht sämtliche Anwendungen des Partition Predicate mit einem einzigen Sortierattribut in bereits bestehenden Roxx Modellen angepasst werden mussten.
Um die Einfachheit der Syntax zu garantieren wurde ein Vorschlag seitens des Roxx Teams, welches im GB 70 für das Programmieren von Roxx Code zuständig ist, als Grundlage gewählt, welches außerdem bereits abwärtskompatibel war, sodass insofern nichts mehr daran geändert werden musste.
Die folgende *Rank()* Operation ist ein Beispiel für die finale Syntax, bei welcher zwei Sortierattribute gewählt wurden:

```
Rank(OrderDesc('birthday'), OrderAsc('employee'))
```

Konkret wurde auf dieser Ebene die *Rank()* Operation also so angepasst, dass eine ganze Liste an Attributen geparset und auch weiterverarbeitet werden kann.

Tests Jede Ebene des Back-ends 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 auf Ebene der *Database* die reine Sortierlogik anhand von Tabellen aus Zeilen und Spalten mit Werten getestet wird.
Auf Ebene des Compilers wird getestet, ob das Bauen von Regeln korrekt funktioniert.
Die Regel selbst und die Äquivalenz zu einer validen Datalog Regel wird auf der *Runtime* Ebene getestet und zuletzt verifizieren Tests auf der Ebene der DSL die korrekte Funktionsweise des Partition Predicates 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 neue hinzugefügt um jeweils die neu implementierte Mehrfachsortierung 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.

3.6 Weitere Aufgaben

Da das eigentliche Projekt sehr schnell abgeschlossen werden konnte, war es möglich zusätzlich zwei zusätzlichen Aufgaben zu definieren.

Eine dieser Aufgaben war es das *multiRankPredicate* Makro zu ersetzen.
Die andere Aufgabe bestand darin eine weitere Sortieroperation neben der bereits bestehenden *Rank()* Operation zu implementieren: die *UniqueRank()* Operation.

3.6.1 Ersetzen des Makros

Das *multiRankPredicate* Makro fand in je drei weiteren Makros Anwendung, die ihrerseits in Konfigurationsmodellen bei Kunden genutzt werden.

Um zu gewährleisten, dass das neue Partition Predicate auch im operativen Kontext wie gewünscht funktioniert, wurde zunächst das Makro so angepasst, dass die Mehrfachsortierung nicht mehr durch die Berechnung der Linearkombination realisiert wurde, sondern durch das neue Partition Predicate.
Der Code, welcher die Berechnungen vollführte, wurde also schlicht gelöscht und ersetzt.
Daraufhin wurde ein für das ursprüngliche Makro geschriebener Unit Test auf das Makro, welches dann das Partition Predicate enthielt, angewendet.
Nachdem das Makro auch mit dem neuen Partition Predicate die gewünschten Ergebnisse errechnete, konnte garantiert werden, dass es auch in den Kundenmodellen ohne Probleme funktioniert, sodass die Nutzung des *multiRankPredicate* Makros an den entsprechenden Stellen gelöscht und durch ein äquivalentes Partition Predicate mit Mehrfachsortierung ersetzt wurde.
Anschließend wurde das gesamte *multiRankPredicate* Makro gelöscht, da es keine Verwendung mehr findet.

3.6.2 Unique-Ranking

Nachdem das *multiRankPredicate* Makro erfolgreich ersetzt war, konnte die zweite Zusatzaufgabe gelöst werden, bei welcher es darum ging eine neue Operation zu implementieren, welche als Alternative zum *Dense-Ranking* bereitstellen sollte.

UniqueRank() Operation Wie im theoretischen Teil zum Partition Predicate bereits erläutert, zeichnet sich das *Dense-Ranking* dadurch aus, dass es bei der Evaluierung mehrerer gleicher Werte mehrfach denselben Rang vergibt.

Einige Anwendungsfälle erfordern es jedoch, dass jedes Element ausnahmslos einzigartige Ränge erhält und kein Rang mehrfach vergeben wird.

Um das realisieren zu können, wurde die *UniqueRank()* Operation implementiert.
Äquivalent zur Umsetzung der Mehrfachsortierung des Partition Predicates wurde auch hier wieder abschnittsweise jede Ebene im Back-end einzeln bearbeitet und mit jedem fertigen Abschnitt wurden Tests hinzugefügt, um die neue Funktionalität ausreichend zu testen und zu validieren.

Auf der Ebene der *Database* musste konkret die Logik zur Vergabe der Ränge je Element erweitert werden.

Die mehrfache Vergabe desselben Rangs wurde beim *Dense-Ranking* realisiert indem für jedes Element in einer Kontrollstruktur überprüft wurde, ob es denselben Wert hat wie sein

Vorgängerelement und der Rang nur dann erhöht, wenn dies nicht der Fall war.
Um das *Unique-Ranking* zu implementieren war es nötig diese Bedingung auszulassen.
Nachdem das allererste Element den Rang 0 erhalten hat, wird für jedes weitere Element der Rang um den Wert 1 erhöht. Unabhängig davon, ob Elemente denselben Wert haben, oder nicht.
Zu beachten ist, dass die Vergabe der Ränge für Elemente mit demselben Wert beim *Unique-Ranking* nichtdeterministisch, also zufällig, geschieht und Ergebnisse bei jeder Berechnung unterschiedlich ausfallen können.

Dies hat zur Konsequenz, dass beim Neustarten eines Client Programms Reihenfolgen von Elementen anders sein können als die Applikation sie vor dem Neustart dargestellt hat.
Dies gilt grundsätzlich als nicht gewünschtes Verhalten.

Um die Berechnung ein wenig mehr deterministisch zu gestalten und diesem Problem etwas entgegenzuwirken, wurde die Entscheidung getroffen, dass beim *Unique-Ranking* zuletzt immer noch ein weiteres mal nach den Hash-Codes der Elemente sortiert wird.
Da die Hash-Codes der Elemente sich nicht ändern, solange die Elemente selbst sich nicht ändern, ist es damit möglich eine deterministische Berechnung zur Sortierung hinzuzufügen, sodass die Vergabe der Ränge dahingehend nicht zufällig geschehen kann[11, S. 722].
Das einzige Problem, das trotz der zusätzlichen Sortierung noch besteht ist, dass es keine Garantie gibt, dass nicht zufällig zwei Elemente denselben Hash-Code erhalten, da die Anzahl an möglichen Elementen in Modellen nahezu unbegrenzt ist, die Anzahl an freien Hash-Codes allerdings nicht.
Die Wahrscheinlichkeit, dass genau zwei Elemente den gleichen Wert haben und insofern als gleich gelten und zusätzlich denselben Hash-Code besitzen ist allerdings äußerst gering.

Man kann also sagen, dass die Implementierung verhältnismäßig sicher vor unerwartetem Verhalten schützt, aber nicht gänzlich garantieren kann, dass es nicht doch auftritt.

Aufgrund dessen werden Roxx Entwickler künftig dazu angehalten werden die *UniqueRank()* Operation nur an vertretbaren Stellen zu nutzen, die den Endverbraucher nicht irritieren, wie beispielsweise ein Pop-up Fenster in der Benutzeroberfläche, das aber ohnehin nicht dauerhafter Teil der grafischen Oberfläche ist.

Um grundsätzlich zwischen dem *Unique-Ranking* und dem *Dense-Ranking* unterscheiden zu können, wurde außerdem ein boolesches Feld definiert, das in den Klassen auf den Ebenen der *Database*, des Compilers und der *Runtime* spezifiziert ob die Sortierung nach dem *Unique-Ranking* oder dem *Dense-Ranking* erfolgen soll.
Auf der Ebene der DSL entscheidet eine Enumeration, welche die beiden Operationen *RANK* und *UNIQUE_RANK* enthält, darüber.

Außerdem wurde auf dieser Ebene wieder eine entsprechende Syntax für das *Unique-Ranking* definiert.
Ein Beispiel für die Syntax stellt die folgende Operation dar:

```
UniqueRank(OrderAsc('name'))
```

Uniqueness Constraints Der zweite Teil der zweiten Zusatzaufgabe bestand aus der Verfeinerung der *Uniqueness Constraints* eines Partition Predicates bei Anwendung der *UniqueRank()*-Operation.

Wie bereits erläutert, definieren *Uniqueness Constraints* eine Obergrenze der möglich vorkommenden Tupel in einem Prädikat.
Bei extensionalen Prädikaten werden *Uniqueness Constraints* als Teil des Prädikats mit angegeben[5].
Bei intensionalen Prädikaten, wie dem Partition Predicate, werden die *Uniqueness Constraints* aus dem Grundprädikat des intensionalen Prädikats und der angewandten Regel logisch geschlossen und abgeleitet.
Konkret bedeutet dies, dass die *Uniqueness Constraints* intensionaler Prädikate im Back-end berechnet werden.

Als Grundlage für ein Beispiel zu den *Uniqueness Constraints* soll nochmals das Partition Predicate aus Abbildung 3 dienen.

Eine sinnvolle Belegung der *Uniqueness Constraints* des Grundprädikats *employeesBirthdays* wäre hier beispielsweise die Liste an *Uniqueness Constraints* [TTF].
Diese Belegung bedeutet, dass es möglich ist, dass es weitere Tupel mit demselben Mitarbeiter und Geburtstag in anderen Mitarbeitergruppen geben kann.
Dafür steht das erste T.
In anderen Worten: ein und derselbe Mitarbeiter kann in verschiedenen Gruppen sein.
Diese Belegung bedeutet auch, dass es möglich ist, dass es für unterschiedliche Mitarbeiter weitere Tupel mit derselben Mitarbeitergruppe und demselben Geburtstag gibt.
Hierfür steht das zweite T.
In anderen Worten bedeutet dies, dass es mehrere Mitarbeiter geben kann, die zufällig dieselbe Gruppe und denselben Geburtstag teilen, obwohl sie nicht dieselbe Person sind.
Zuletzt bedeutet die Belegung aber auch, dass es kein zweites Tupel mit derselben Mitarbeitergruppe und demselben Mitarbeiter geben kann, der allerdings einen anderen Geburtstag hat.
Dafür steht das F.
Konkret bedeutet dies, dass ein und dieselbe Person keine zwei Geburtstage haben kann.

Neben einigen Randfällen, auf welche mangels Berührungspunkte mit der eigentlichen Aufgabe an dieser Stelle nicht eingegangen werden soll, leitet die *Rank()* Operation beim Partition Predicate stets *Uniqueness Constraints* ab, die aus den *Uniqueness Constraints* des Grundprädikats des Partition Predicates, bestehen, welche mit einem *Uniqueness Constraint* F erweitert werden.
Die *Uniqueness Constraints*, welche sich für das Beispiel ergäben, entsprechen also der Liste [TTFF].
Diese Erweiterung resultiert aus der logischen Folgerung, dass es kein zweites Tupel mit derselben Gruppe, demselben Mitarbeiter und denselben Geburtstag geben kann, das aber einen anderen Rang hat als ein bestimmtes einzigartiges Tupel, denn andernfalls könnte ein und derselbe Mitarbeiter mehrfach in der Sortierung mit unterschiedlichen Rängen auftauchen, was dem Sinn der *Rank()* Operation widerspricht.

Es kann bei der *Rank()* Operation aber keine Aussage darüber getroffen werden, dass die *Uniqueness Constraints*, die sich aus dem Grundprädikat ergeben ebenfalls strenger sind, als sie

in dem Grundprädikat definiert sind, da das *Dense-Ranking* die mehrfache Vergabe desselben Rangs erlaubt. Es kann durchaus sein, dass es mehrere Tupel gibt, die dieselbe Mitarbeitergruppe, denselben Geburtstag und Rang für einen anderen Mitarbeiter definieren.
Entsprechendes gilt für den *Uniqueness Constraint* der Mitarbeitergruppe.

Dadurch, dass das *Unique-Ranking* aber gerade keine mehrfache Vergabe von Rängen erlaubt, können bei dieser Art der Sortierung strengere *Uniqueness Constraints* abgeleitet werden, die sich zwangsweise aus der Logik des *Unique-Rankings* ergeben.

Zusätzlich zur Tatsache, dass ein Tupel auch beim *Unique-Ranking* eindeutig über den jeweiligen Rang bestimmt werden können, kann auch die Aussage getroffen werden, dass jedes Attribut, welches weder dem Rang entspricht, noch eines der Attribute ist, welche die Partition definieren, ebenfalls ein F *Uniqueness Constraint* erhalten müssen.

Diese Aussagen auf das Beispiel angewandt, würde zu den zusätzlichen *Uniqueness Constraints* [TFFT] führen.
Das bedeutet also, dass für jeden Mitarbeiter ein einziges Tupel mit derselben Mitarbeitergruppe und demselben Geburtstag und Rang, sowie für jeden Geburtstag ein einziges Tupel mit derselben Mitarbeitergruppe und demselben Mitarbeiter und Rang existieren kann, wohingegen für jede Mitarbeitergruppe mehr als nur ein Tupel desselben Mitarbeiters, Geburtstags und Rang existieren kann.
Auch für den Rang selbst ist es möglich, dass für unterschiedliche Ränge mehrere Tupel derselben Mitarbeitergruppe und desselben Mitarbeiters und Geburtstags existieren.
Der Grund hierfür ist der, dass nun mal der Rang beim *Unique-Ranking* innerhalb jeder Partition einzigartig sein muss und damit das gesamte Tupel, welches den Rang beinhaltet, ebenfalls einzigartig sein muss.
Dies gilt aber nur für jede Partition für sich betrachtet.
Über verschiedene Partitionen hinweg können sich Ränge offensichtlich wiederholen, sodass sich daraus ein *Uniqueness Constraint* T für alle Attribute ergibt, welche eine Partition definieren oder eben selbst der Rang sind.

Für das Partition Predicate des Beispiels würden insgesamt also die *Uniqueness Constraints* [TTFF] und [TFFT] errechnet werden.

Sobald die Logik für die *Uniqueness Constraints* herausgearbeitet wurde, konnte sie in der Klasse *PartitionConstraintDerivation* implementiert und der dort bereits vorhandenen Logik hinzugefügt werden.

Anschließend wurden auch hierfür noch eine Reihe an Unit Tests geschrieben, die alle möglichen Kombinationen an unterschiedlicher Belegung der ursprünglichen *Uniqueness Constraints* des Grundprädikats und der Attribute zur Partitionierung für ein Partition Predicate mit drei Ausgangsattributen abdecken, um die implementierte Logik möglichst umfangreich überprüfen zu können.

4 Ausblick

Für die Bachelor Thesis, welche das hier vorgestellte Projekt als Grundlage behandeln soll, ergibt sich die Möglichkeit auf die theoretischen Inhalte von Roxx tiefer einzugehen.

Denkbar ist dabei beispielsweise die Funktionsweise anderer High-Level Prädikate ebenfalls zu erörtern und noch spezifischer auf die Theorie hinter den Regeln einzugehen und zu erläutern, wie genau diese Regeln im Kontext von Datalog funktionieren und worin sich Roxx von dieser Funktionsweise unterscheidet.

Außerdem interessant wäre eine präzise Untersuchung welche Schwachstellen Roxx aufweist und wo seine Grenzen sind, insbesondere um die Frage diskutieren zu können, ob der Einsatz von Roxx als Modellierungssprache des logisch deklarativen Programmierparadigmas gegenüber imperativen Programmiersprachen wie zum Beispiel Java wesentliche Vorteile bringen und ob sich dieser Einsatz lohnt.

Anhang

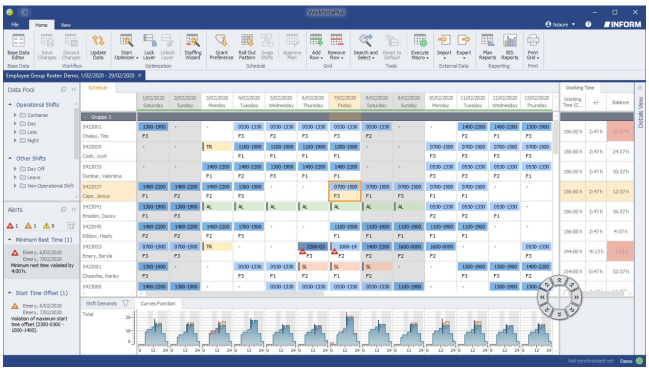

Abbildung 9: WorkforcePlus Windows Application Client[12]

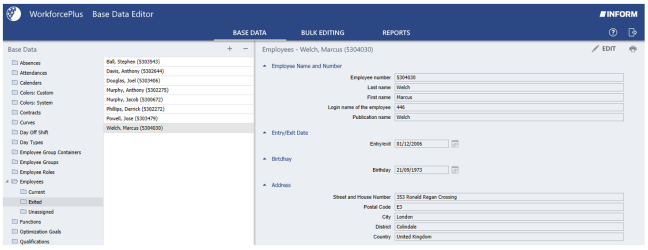

Abbildung 10: WorkforcePlus Base Data Editor[12]

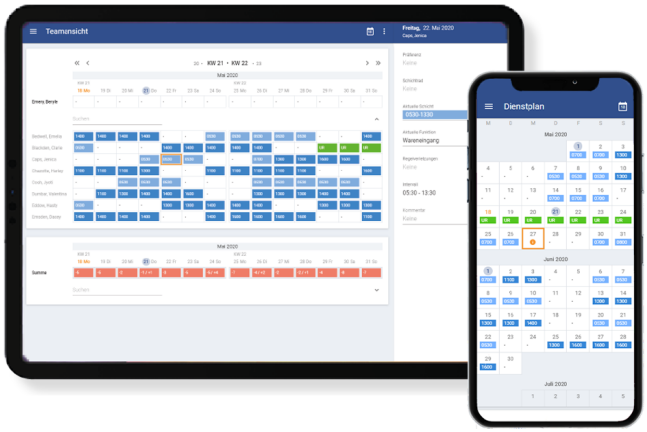

Abbildung 11: WorkforcePlus Employee Portal[12]

Abbildungsverzeichnis

1 Komponenten von WorkforcePlus 2
2 Example Roxx Syntax 3
3 Beispiel Partition Predicate 4
4 Beispieldaten 5
5 Beispieldaten nach der Partitionierung 6
6 Beispieldaten nach Evaluierung der *Rank*-Operation 6
7 Mitarbeiter Requests der Applikation WorkforcePlus Employee Portal 8
8 Architektur Partition Predicate (Eigene Abbildung) 15
9 WorkforcePlus Windows Application Client I
10 WorkforcePlus Base Data Editor I
11 WorkforcePlus Employee Portal I

Tabellenverzeichnis

1 Mitarbeiterdaten der Arbeitnehmergruppe *EmplGr2* (Eigene Tabelle) 10

Literatur

[1] C. M. Harland, "Supply Chain Management: Relationships, Chains and Networks," *British Journal of Management*, vol. 7, no. s1, 1996.

[2] "Über INFORM," zuletzt geprüft am 22.12.2022. [Online]. Available: https://www.inform-software.de/unternehmen/ueber-inform

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

[4] INFORM, "Overview Back-end," Jul. 2020.

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

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

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

[8] 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.

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

[10] *Interface Comparator*, zuletzt geprüft am 23.12.2022. [Online]. Available: https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Comparator.html

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

[12] "Workforceplus - Clients," zuletzt geprüft am 23.12.2022. [Online]. Available: https://confluence.inform-software.com/display/WFM/WorkforcePlus+-+Clients