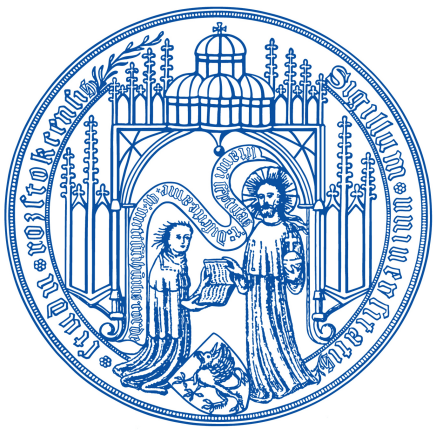
Vertikale Verteilung von SQL-Anfragen auf DBMS-Servern abnehmender Leistungsfähigkeit

Masterarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik



vorgelegt von:	Christian Langmacher
Matrikelnummer:	8200742
geboren am:	18. August 1989 in Rostock
Erstgutachter:	Prof. Dr. rer.nat.habil. Andreas Heuer
Zweitgutachter:	Prof. Dr. rer.nat.habil. Karsten Wolf
Betreuer:	Hannes Grunert
Abgabedatum:	04. September 2017

Inhaltsverzeichnis

Kapitel 1

Einleitung

In der heutigen Zeit kommt der Verarbeitung von großen Datenmengen eine immer größere Bedeutung zu. Um dies auch performant zu halten, muss die Arbeit auf mehrere Rechnerknoten verteilt werden.
Speziell im zum Anlass genommenen Projekt PArADISE[1] müssen sehr große (Strom-)Datenmengen schnell verarbeitet werden. Hierbei werden komplexe Anfragen auf mehreren Ebenen auf mehrere Teilsysteme mit weniger Leistungsfähigkeit — hiermit sind sowohl Rechenleistung als auch Funktionalität, wie bestimmte Operatoren, gemeint — verteilt, wodurch eine horizontale und eine vertikale Verteilung dieser Anfragen nötig wird.
In diesem Projekt werden zur Unterstützung hilfsbedürftiger Menschen Sensordaten von bspw. Beschleunigungs- und Berührungssensoren verarbeitet, um zu erkennen, welches Ziel das Subjekt verfolgt und ob es bei der Erreichung dieses eventuell Hilfe benötigt. Dies resultiert in sehr großen Datenmengen und der Notwendigkeit, schnell Ergebnisse zu liefern, was auch die Aufteilung der Anfragen an diese Daten auf mehrere Rechnerknoten nötig macht, die auch jeweils unterschiedliche Fähigkeiten haben. Auch sind die erhobenen Daten personengebunden, was aus Datenschutzgründen eine Filterung und Aggregation der Daten möglichst nah am Sensor nötig macht.

Hierbei gibt es mehrere Probleme: Zunächst sind die Anfragen hochkomplexe SQL-Anfragen, für die kein Algorithmus zur Aufteilung existiert. Zudem sind die Teilcomputer nicht immer in der Lage, sämtliche SQL-Operationen auszuführen. Damit müssen also einige Operationen *ausgelagert* und beim Zusammenführen der Teilresultate ausgeführt werden. Alternativ dazu kann es möglich sein, dass komplexe, nicht unterstützte Operatoren in einfachere, unterstützte Operatoren transformiert werden können, um so möglichst wenig Daten übertragen zu müssen.

1.1 Aufteilung von Anfragen

Für die Aufteilung einer SQL-Anfrage auf verschiedene Rechnerknoten gibt es mehrere verschiedene Möglichkeiten, die im Wesentlichen von der Auftrennung der Datenbasis in die jeweiligen Datenbankfragmente abhängig sind. Der Vorgang ist jedoch immer der gleiche: Aus einer eingehenden SQL-Anfrage müssen mehrere valide SQL-Anfragen generiert werden sowie eine Funktion f, die angibt, wie aus den ankommenden Ergebnismengen die zusammengefügte Ergebnismenge generiert werden muss.
Der Typ der Auftrennungsfunktion g muss also der Folgende sein:
$g : Query \to [Query] \times ([ResultSet] \to ResultSet)$
Hierbei ist ein Query eine Zeichenkette, also ein Array von Charactern: $Query = [Char]$, ein ResultSet ist

[1] Privacy AwaRe Assistive Distributed Information System Environment — für weiterführende Informationen siehe [GH16]

eine Liste von Ergebnistupeln (*Rows*), die jeweils Objektlisten sind: $ResultSet = [[Object]]$. Die zweite Ausgabe ist damit also die Funktion f, die eine Menge von ResultSets in ein ResultSet kombiniert.
Dann wird also zunächst die Anfrage analysiert, dann Teilanfragen für die einzelnen Rechnerknoten berechnet, die dann ausgeführt werden. Die zurückkehrenden Fragmente werden mit der Funktion f zusammengeführt und das Ergebnis zurückgegeben. Wie die jeweiligen Fragmente aussehen, hängt im Grunde von der Verteilung der Universaldatenbank ab. Für die Auftrennung der Datenbank auf mehrere Rechnerknoten (nicht zu verwechseln mit der Verteilung der **Anfrage** auf mehrere Rechnerknoten) gibt es die folgenden Möglichkeiten, die hier betrachtet werden sollen:

- Horizontale Verteilung
- Vertikale Verteilung
- Hybride (horizontale + vertikale) Verteilung
- Redundante Speicherung / Spiegelung

Im vorliegenden Projekt ist von einer hybriden Verteilung auszugehen — mehrere gleichartige Sensoren (bspw. Temperatursensoren) sind mit einer horizontalen Verteilung gleichzusetzen (wobei der Ort des Sensors als zusätzliche Annotation oder ein weiteres Attribut der Tupel angesehen werden kann), während heterogene Sensoren (bspw. Temperatur- und Beschleunigungssensoren) mit einer vertikalen Verteilung gleichzusetzen sind. Ähnliche Äquivalenzen sind auch auf den anderen Verteilungsebenen zu finden.

1.2 Randbedingungen

Das vorliegende Problem berührt die bereits untersuchten Probleme des *Query Rewriting* (Umschreiben einer Anfrage in eine äquivalenten Anfrage) sowie des *Query Containment* (Untersuchen, ob die Resultatmengen zweier gegebener Anfragen bei jeder Datenbasis in einer Teilmengenbeziehung stehen, ohne diese zu kennen), die beide noch nicht vollständig gelöst sind. Lediglich bestimmte Arten von Anfragen sind bei den bekannten Herangehensweisen unterstützt. Allerdings gibt es einige Randbedingungen, die die Analyse einfacher machen.
Die Anfragen an den Datenbestand, die im Grunde genommen alle Matrixoperationen sind, werden in der Programmiersprache R geschrieben und von dort in äquivalente Anfragen (unter anderem auch SQL-Anfragen) transformiert. Wie dies funktioniert, wird in [Weu16] erläutert. Dies bedeutet allerdings auch, dass nur Anfragen aufgeteilt werden müssen, die auch tatsächlich von einer Anfrage in R erzeugt werden können. Da allerdings Anfragen prinzipiell von überall herkommen können, kann dies nicht verwendet werden — insbesondere da nach derzeitigem Erkenntnisstand nicht alle Anfragen in R in äquivalente SQL-Anfragen transformiert werden können.
Eine weitere Bedingung ist eine, die die Analyse nur bedingt vereinfacht: Da natürlich keine Bewegungs- oder Positionsdaten von reellen Personen nach außen, und sei es auch nur zur automatischen Analyse, gelangen dürfen, werden die Ergebnismengen *anonymisiert*. Dies betrifft sowohl Eingaben als auch Ausgaben der zusammenführenden Anfrage: Zunächst muss davon ausgegangen werden, dass alle oder einige Ergebnismengen, die von den Teilrechnerknoten zurückgeliefert werden, bereits anonymisiert sind. Dies hat im Grunde genommen zwei Auswirkungen:

- Typen anonymisierter Datenwerte müssen, wie beispielsweise bei Generalisierung, nicht zwangsläufig mit dem Typen des Original-Datenwerts übereinstimmen. Dies hat zur Folge, dass der Datentyp in der Datenbank nicht zur Analyse verwendet werden darf, da derjenige in der Ergebnismenge ein anderer sein könnte. Teilweise lassen sich allerdings aus generalisierten Werten, wie beispielsweise Wertebereichen, wieder Repräsentantenwerte erzeugen (siehe auch [Müll6]).

- Da identifizierende (Schlüssel) oder quasi-identifizierende Attributkombinationen die Anonymität verletzen, werden in anonymisierten Ergebnismengen niemals identifizierende Attribute in der ursprünglichen Form auftauchen. Allerdings sind bei vertikaler Verteilung die Schlüsselattribute gerade die Joinattribute, über die die Fragmente wieder verbunden werden. Dies ist ein inhärenter Widerspruch, weshalb im Rahmen dieser Arbeit davon ausgegangen wird, dass die Ergebnismengen nicht anonymisiert werden, eine horizontale Verteilung vorliegt, der letzte Verbund vor der ersten Anonymisierung oder die Anonymisierung nach der Transformation stattfindet.

Zudem muss auch dafür Sorge getragen werden, dass, wenn die Teilresultate die Anonymitätsanforderungen erfüllen, auch das Gesamtresultat die Anonymitätsanforderungen erfüllt (es sei denn, es selbst wird noch einmal anonymisiert). Hierfür existieren allerdings bereits die Ansätze der *Distributed k-anonymity* [NJTF15] u. Ä., weshalb dies nicht gesondert betrachtet werden soll.

Im Grunde existieren zwei unterschiedliche Herangehensweisen für das Problem: Da es mehrere unterliegende (evtl. heterogene) Systeme gibt, kann die ankommende Anfrage zunächst auf die externen Systeme aufgeteilt und dann jede einzelne Teilanfrage dahingehend transformiert werden, dass sie auf dem jeweiligen System ausgeführt werden kann. Alternativ dazu kann allerdings auch die Originalanfrage zunächst transformiert werden (hierbei ist davon auszugehen, dass die unterliegenden Systeme trotz Heterogenität gleiche Fähigkeiten haben) und dann die transformierte, im Allgemeinen einfachere (weil weniger komplexe und auch weniger Operationen verwendet werden) Anfrage an die Rechnerknoten verteilt werden. Da sich diese Arbeit mit der **vertikalen** Verteilung von SQL-Anfragen auseinander setzt und sich herausstellen wird, dass Ansätze auch auf mehrere Fähigkeiten erweiterbar sind, soll das Hauptaugenmerk auf der Transformation von Anfragen zur Ausführung mithilfe **einer** Menge von zulässigen Operationen liegen.

Im Folgenden sollen zunächst die relevante Theorie sowie einige Grundtechniken (siehe Kapitel 2), die für die gestellte Aufgabe von Wichtigkeit sein können, vorgestellt werden, wobei auch die Unzulänglichkeiten bezüglich der Komplexität der ankommenden Anfragen betrachtet werden müssen. Daraufhin werden gegebenenfalls bereits existierende Applikationen zur Berechnung der Lösungen der in den Grundtechniken gegebenen Probleme vorgestellt, worauf eigene Analysen (siehe Kapitel 3) und Implementationen (siehe Kapitel 4) zur Anwendung bei der Aufteilung von Anfragen folgen werden. Hierbei ist von Bedeutung, dass beliebig komplexe Anfragen bearbeitet werden können müssen — auch die Performance ist natürlich von Bedeutung. Zum Schluss folgt dann eine Zusammenfassung, was erreicht werden konnte und was nicht, sowie ein Ausblick, was die Zukunft in diesem Bereich bergen mag (siehe Kapitel 5).
Nicht Teil dieser Arbeit werden die Aufteilung von Anonymisierungstechniken, die horizontale Verteilung von Anfragen und andere Ansätze zur Sicherung von Performance und Privatheit sein. Auch die Verteilung von Nicht-SQL-Anfragen, wie beispielsweise in Datalog, wird nur beiläufig Erwähnung finden.

Kapitel 2

Grundlagen

In diesem Kapitel sollen einige grundlegende Techniken erläutert werden, die zur Transformation von Anfragen verwendet werden können. Hierbei sollen insbesondere das *Capabilities-based Query Rewriting* sowie das *Answering Queries Using Limited External Query Processors* betrachtet werden, die dieses Ziel verfolgen.

2.1 Capabilities-based Query Rewriting

Das *Capabilities-based Query Rewriting* (etwa: Fähigkeiten-basierte Anfrageumschreibung) befasst sich mit der Anfragetransformation für die Ausführung auf DBMS-Servern geringerer Funktionalität in Mediator-basierten Systemen. Kernstück dieser Technik, die 1998 von IBM entwickelt wurde, ist der sogenannte *Capabilities-Based Rewriter* (CBR). Dieser erzeugt aus einer Beschreibung der Fähigkeiten der unterliegenden Ebene und einer an sie gerichteten Anfrage möglichst große Teilanfragen, die auf der unterliegenden Ebene ausführbar sind (*Component SubQuerys, CSQs*), sowie einen *Plan*, der die Teilresultate zusammenführen kann. Hierbei sind jedoch nur konjunktive Anfragen unterstützt [PGH98]. Für die Darstellung der Fähigkeiten des jeweiligen Servers wird eine neue Sprache eingeführt, die *Relational Query Description Language* (RQDL).

2.1.1 Relational Query Description Language

Die RQDL beschreibt nicht direkt die Fähigkeiten der jeweiligen Ebene, sondern vielmehr die unterstützten Anfragen (bzw. Anfrageoperationen) mithilfe einer Grammatik. Diese sogenannten *Query Templates* sind im Grunde Grammatiken für Anfragen, die parametrisierte Sichten beschreiben, wie beispielsweise

$$answer(R) :- Table(A), Cond(A), subset(R, A) \tag{2.1}$$

Dieses Template beschreibt lediglich, dass die Antwort einer Anfrage R ist, wenn die Tabelle A angefragt wird, die Bedingung *Cond(A)* zutrifft und R eine Teilmenge von A ist. Durch dieses Template sind demnach recht genau die *FROM*- und *WHERE*-Klauseln einer SQL-Anfrage abgedeckt. Die Arten der Vergleiche (Attribut-Attribut, Attribut-Konstante; Vergleich auf Gleichheit, kleiner als o.Ä.) werden hierbei durch die Spezifikation von Cond(A) definiert.
Im Beispiel sei diese Spezifikation durch die folgenden beiden zusätzlichen Regeln gegeben:

$$Cond(A) :- substring(X, \$S), Cond(A) \tag{2.2}$$

$$Cond(A) :- \tag{2.3}$$

wobei Variablen, die mit „$“ beginnen, nur mit Terminalen unifiziert werden dürfen. Hierbei können aus $Cond(A)$ mithilfe von 2.2 immer mehr *substring()*-Bedingungen eingefügt werden und, sobald genügend

vorhanden sind, die Kette mithilfe der Regel 2.3 abgebrochen werden. Damit lässt diese Grammatik eine unbegrenzte Anzahl an Substring–Bedingungen auf der jeweiligen Tabelle A zu und nichts Weiteres — selbst einfache Attribut-Wert-Vergleiche sind verboten.

Hierdurch müssen für die Transformation lediglich solche Anfragen betrachtet werden, die durch die Template-Grammatik erzeugt werden können. Damit ist also bekannt, welche Anfragen möglich sind, die tatsächliche Transformation steht allerdings noch aus, was in einem Modul des Capabilities-Based Rewriter geschieht.

2.1.2 Capabilities-Based Rewriter

Der Capabilities-Based Rewriter besteht aus drei Modulen, wobei die Gesamtheit dafür sorgen soll, dass eine ankommende Anfrage Q dahingehend transformiert wird, dass sie von der bearbeitenden Ebene ausgeführt werden kann, d. h. von ihrer beschreibenden RQDL erzeugt werden kann. Die drei Module werden nacheinander ausgeführt und sind im Folgenden beschrieben:

- Das CSQ Discovery-Modul findet Anfragen, die von der RQDL erzeugt werden können und eines oder mehrere Teilziele von Q beinhalten. Hierbei werden lediglich solche Anfragen zurückgegeben, die eine maximale Anzahl von Selektionen und Joins, aber keine Projektion beinhalten. Dies verhindert eine Explosion des Suchraums.
- Das Plan Construction-Modul kombiniert die gefundenen CSQs und erzeugt daraus Pläne für die Zusammenführung, sodass wieder Q berechnet wird.
- Das Plan Refinement-Modul verfeinert den oder die berechneten Pläne, indem so viele Projektionen wie möglich hinzugefügt werden.

2.1.3 CSQ Discovery

Das CSQ Discovery-Modul findet Teilanfragen, indem es das Template (das RQDL-Regelsystem) als Entwicklungsregeln auffasst und die Teilziele als Grundterme. Dann werden im Bottom-Up-Verfahren immer alle Regeln angewandt, die zu einem oder mehreren Teilzielen entwickelt werden können. Weitergegeben werden hierbei allerdings nur die *maximalen* CSQs, also die Teilanfragen mit der Maximalzahl an Selektionen und Joins, ohne Projektionen, die von keinem anderen CSQ subsumiert werden. Dies garantiert, dass so wenige CSQs wie möglich generiert werden, die allerdings so viele Teilziele wie möglich erfüllen. Die Anfrage

$$SELECT\ *\ FROM\ emp\ WHERE\ salary > 2000\ AND\ lastname\ LIKE\ `\%amp\%' \tag{2.4}$$

entspricht den folgenden Prädikaten:

$$Table(emp), subset(R, emp), substring(lastname, ' amp'), salary > 2000 \tag{2.5}$$

Zu der gegebenen Grammatik ist lediglich ein CSQ zu finden:

$$Table(emp), subset(R, emp), substring(lastname, ' amp'), \tag{2.6}$$

da dieses mithilfe von Anwendung 2.3, 2.2 und anschließend 2.1 zu answer(R) entwickelt werden kann, salary>2000 jedoch keine Regelanwendung zulässt. Eine Grammatik, die >-Vergleiche zulässt, würde ebenfalls eine entsprechende salary-Teilanfrage entdecken bzw. eine, die gar beide Bedingungen erfüllt.

2.1.4 Plan Construction

Das Plan Construction-Modul kombiniert die maximalen CSQs in der Form, sodass alle Teilziele erfüllt werden. Hierfür werden bei jedem CSQ die bereits erreichten Teilziele ermittelt und mit den CSQs kombiniert, die die restlichen Teilziele erfüllen. Der Plan ergibt sich dann über die Ausführung der CSQs und daraufhin der Verbund der Resultatmengen, was sich aus der folgenden Gleichung ergibt:

$$\sigma_X(R) \bowtie \sigma_Y(S) = \sigma_{X \wedge Y}(R \bowtie S) \tag{2.7}$$

Dies ist allerdings nur bei relativ simplen Prädikaten X und Y wahr, wie sie häufig in der WHERE-Klausel von Anfragen vorkommen. Nicht immer gültig ist die Aussage, falls die Verbundattribute nicht exportiert werden (dies ist allerdings bei maximalen CSQs immer der Fall), Prädikate aus der WHERE- und der HAVING-Klausel kombiniert werden oder bei Attribut-Attribut-Vergleichen zwischen zwei Relationen. Auch hier ist die Einschränkung dieses Algorithmus auf konjunktive Anfragen mit Selektion, Projektion und Join sichtbar.
Im Beispiel würden beide entstehenden CSQs zurückgegeben und die Ergebnisse verknüpft, sollte kein CSQ alle Anforderungen erfüllen. Da beide CSQ maximal sind, ist hierbei sichergestellt, dass die minimale Anzahl an Joins benötigt wird.

2.1.5 Plan Refinement

Das Plan Refinement-Modul hat zwei große Aufgabenbereiche: Zunächst sind die maximalen CSQs nur solche, die tatsächlich maximal sind und damit insbesondere keine Projektion aufweisen. Das Plan Refinement-Modul muss also alle überflüssigen Attribute durch Projektion eliminieren, wobei die Join-Attribute belassen werden müssen. Außerdem ist die Kombination von maximalen CSQs auch mit dieser Projektion nicht notwendigerweise algebraisch optimal; die Plananfragen müssen dann also noch optimiert werden.

2.1.6 Zusammenfassung

Das *Capabilities-based Query Rewriting* hat, wenn auch mit einem anderen Startpunkt (mediator-basierte Datenbankföderation), das gleiche Ziel verfolgt. Die erzeugten Anfragen sind zumindest unter den erwähnten Einschränkungen optimal bezüglich Minimalität, allerdings sind nur recht simple Anfragen unterstützt, was die Nutzbarkeit für unsere komplexen Anfragen einschränkt. Zu beachten ist auch, dass die Beschreibungssprachen explizit schema-unabhängig sind und aus diesem Grund die entstehenden Pläne wie auch die CSQs nur mit einem geeigneten Mapping auf äquivalente SQL-Anfragen abbildbar sind. Die Hauptkomplexität in diesem Ansatz liegt in der Berechnung der Pläne, da hier relevante CSQ-Kombinationen geprüft werden müssen, was diesen Schritt zu einem mit einer exponentiellen Laufzeit (in Anzahl der CSQs) macht. Die weiteren Techniken basieren insbesondere auf einer Technik namens *Answering Queries using Views* (AQuV), die im folgenden Abschnitt vorgestellt werden soll.

2.2 Answering Queries using Views

Sichten (engl. Views) erlauben es, häufig vorkommende Anfragen vorzudefinieren sowie evtl. deren Ergebnisse bereits vorzuhalten (materialisierte Sichten). Da diese Sichten im Normalfall schneller ausführbar sind als andere Anfragen, macht es Sinn, für ankommende Anfragen so viel wie möglich vordefinierte Sichten zu verwenden — eine Technik, die unter dem Namen *Answering Queries using Views* bekannt ist.
Für die Aufteilung und Transformation von Anfragen können die einzelnen Server als vorgefertigte Sicht betrachtet werden, wobei die globale Anfrage mithilfe der Technik optimal auf die Teilsichten (-Server) verteilt werden kann. Für die Lösung gibt es mehrere Ansätze, von denen ein einfacher im Folgenden vorgestellt werden soll.

Bucket-Algorithmus

Der Bucket-Algorithmus findet speziell Anwendung in der Informationsintegration, wobei die lokalen Datenbanken als Sicht der globalen Datenbank angesehen werden (*Local as View* — LaV). Er ist allerdings auch für das „traditionelle“ AQuV anwendbar. Hierfür werden die in der Anfrage nötigen Relationen in *Buckets* aufgeteilt, wobei initial die einzelnen Prädikate in jeweils einem eigenen Bucket liegen und daraufhin den Buckets die Views, die für die jeweilige Relation nutzbar sind, hinzugefügt werden. Daraufhin werden Kombinationen aus den Views der Buckets gebildet und für jedes wird geprüft, welche Teilziele abbildbar sind. Dies verkleinert den Suchraum um einen großen Faktor [Vas09, LN07].
Beispielsweise sei die Sichtenmenge V, bestehend aus V_1, V_2 und V_3 gegeben, mit (in Datalog-Notation)

$V_1(a)$:- `zitiert(a,b), zitiert(b,a)`
$V_2(c,d)$:- `gleichesGebiet(c,d)`
$V_3(f,h)$:- `zitiert(f,g), zitiert(g,h),gleichesGebiet(f,g)`

sowie die Anfrage Q_1:

$Q_1(x)$:- `zitiert(x.y), zitiert(y,x), gleichesGebiet(x,y)`

Aus den einzelnen Prädikaten werden nun Buckets erzeugt und dann mit den passenden Sichten gefüllt. Die Tabelle 2.1 zeigt die Buckets, wobei eine Zeile für ein Bucket steht. Nun werden die Kombinationen

Tabelle 2.1: Die gefundenen Buckets der Anfrage Q_1

zitiert(x,y)	**zitiert(y,x)**	**gleichesGebiet(x,y)**
$V_1(x)$	$V_1(x)$	$V_2(x,y)$
$V_3(x,y)$	$V_3(x,y)$	$V_3(x,y)$

der drei Buckets gebildet (dies resultiert in einem kartesischen Produkt und damit in diesem Fall acht verschiedenen Plänen) und für jede Kombination geprüft, welche Teilziele abbildbar sind — die einzigen Kombinationen, bei denen die Query Containment-Überprüfung — die faktisch durch Containment Mappings realisiert wird — erfolgreich ist, sind

1. $V_3(x,y)$, $V_2(x,y)$
2. $V_3(x,y)$

Die Sicht $V_3(x,y)$ liefert ein äquivalentes Ergebnis zur originalen Anfrage und ist im Gegensatz zu $V_3(x,y)$, $V_2(x,y)$ minimal, womit die Umschreibung $Q_1'(x)$ mit $Q_1'(x) :- V_3(x,x)$ gefunden wird [Klu17].

Weitere Verfahren

Es existieren noch andere Verfahren für das Lösen des AQuV-Problems, wie der sogenannte MiniCon und der Inverse-Regeln-Algorithmus, beide vorgestellt sowohl in [HDI12] als auch in [Klu17]. Eine andere Möglichkeit, die *Answering Queries using Views*-Problematik zu lösen, besteht in der Anwendung des Chase-Algorithmus. Da dieser Algorithmus zusammen mit dem Backchase eine elegante Lösung für das Problem bietet, soll er in dem gesonderten Abschnitt Provenance-Directed Chase&Backchase vorgestellt werden. Der Chase-Algorithmus ist ursprünglich eine Technik, um zusätzliche Informationen, wie funktionale oder Verbund-Abhängigkeiten, in Datenbankschemata oder Anfragen einzuarbeiten. Die besondere Unterart, die für die Transformation von Anfragen wie auch das *Answering Queries using Views*-Problem verwendet werden kann, fügt Sichtinformationen in Datenbankanfragen ein. Für die Ermittlung der transformierten bzw. optimierten Anfrage ist zusätzlich der Backchase-Algorithmus nötig, der allerdings ein Backtracking-Algorithmus ist und aus diesem Grund eine erhebliche Performance-Belastung darstellt. Für die Lösung dieses Problems wurde der *Provenance-Directed Chase&Backchase* entwickelt.

2.3 Provenance-Directed Chase&Backchase

Der Chase-Algorithmus ist ein iterativer Algorithmus, der für den AQuV-Fall Sichtinformationen in eine Anfrage in mehreren Schritten einbaut. Diese Anzahl an Chase-Schritten ist zwar beschränkt, kann aber dennoch eine hohe sein. Die Anzahl der durchzuführenden Chases für den Backchase ist exponentiell, was die Komplexität dieses Algorithmus explodieren lässt. Der *Provenance-Directed Chase&Backchase* [DH13] reduziert die Anzahl der zu untersuchenden Möglichkeiten mithilfe der Provenance-Informationen (*why*[1]), die während des Chase mit aufgenommen werden können. An dieser Stelle sei erwähnt, dass der Chase, wie er für diesen Ansatz verwendet wird, lediglich konjunktive Anfragen unterstützt.

2.3.1 Bisherige Vorgehensweise

Der normale Chase&Backchase (C&B)-Algorithmus geht in zwei Schritten vor:

1. Die Anfrage Q wird mit den Sichtbedingungen V und den Integritätsbedingungen I gechaset, das Ergebnis ist $Q^{V \cup I}$. Dann wird die Teilanfrage U, der *universelle Plan*, erzeugt, indem $Q^{V \cup I}$ nur mithilfe der Sichten ausgedrückt wird.

2. Die Teilanfragen von U werden auf Äquivalenz zu Q geprüft, und alle minimalen äquivalenten Teilanfragen werden zurückgegeben. Hierfür wird für jede Teilanfrage SQ überprüft, ob ein Containment Mapping von Q auf $SQ^{V \cup I}$ existiert.

Hierbei ist der zweite Schritt der sogenannte Backchase, der eine exponentielle Komplexität (in der Anzahl der Teilanfragen) mit sich bringt. Es gibt Ansätze, die zum Beispiel zunächst alle atomaren Teilanfragen betrachten, dann die aus zwei Prädikaten usw., doch sind es auch mit dieser Taktik im schlechtesten Fall exponentiell viele Untersuchungen. Um die Performance zu steigern, müssen also Teilziele von U, die nicht Teil von Q sein können, abgeschnitten und nicht weiter betrachtet werden. Welche Teilziele dies jedoch im Einzelnen sind, ist nicht einfach herauszufinden (insbesondere durch die Präsenz von tupelgenerierenden Abhängigkeiten — TGDs) und das Prüfen jeder Möglichkeit ist dadurch, dass jedes Mal gechaset werden muss, nicht billig.

2.3.2 Neue Vorgehensweise

Um solche unnötigen Prädikate früh zu entfernen, führt der *Provenance-Directed Chase&Backchase* für jedes Atom a in $U^{V \cup I}$ *Provenance-Informationen* ein, die angeben, welche Atome aus U für die Erzeugung von a beim Chasen verantwortlich waren. Hierfür müssen für den Backchase (d.h. die Containment Mappings) nur diejenigen Prädikate betrachtet werden, die in den Provenance-Sets von $U^{V \cup I}$ auftauchen (da andere nicht notwendig sind). Diese Menge ist deutlich kleiner, weshalb viel weniger Möglichkeiten überprüft werden müssen. Da beim Chasen bekannt ist, auf welche Zeilen die Schritte angewandt werden, können dabei die Zeilen des Chase-Ergebnisses auch annotiert werden, um so den Backchase zu beschleunigen.

2.4 Answering Queries Using Limited External Query Processors

Dieser Ansatz liefert eine Möglichkeit, AQuV (also auch den gut verstandenen und nun auch performanten Provenance-Directed Chase&Backchase) zur Transformation von Anfragen zu verwenden. Grundlage hierfür ist die Erkenntnis, dass (operatorbasierte) Fähigkeiten von Systemen durch Mengen von Sichten darstellbar sind. Das Paper [LRU99] beschreibt eine Möglichkeit, diese Erkenntnis algorithmisch zur Transformation von Anfragen zu verwenden.

[1] Die *why*-Provenance gibt an, warum ein bestimmtes Tupel im Ergebnis auftaucht — in diesem Fall, welche Tupel/Teilziele für die Erzeugung eines Tupels im Ergebnis verantwortlich waren.

2.4.1 Aufteilung in Sichtäquivalenzklassen

Für die Beschreibung von Fähigkeiten sind, wie bereits erwähnt, Mengen von Sichten möglich, wie in [PGGMU95] gezeigt wird. Diese Mengen sind aber im Normalfall unendliche Mengen von Sichten, wie beispielsweise parametrisierte Sichten. Da diese Mengen schlecht algorithmisch handhabbar sind, müssen sie in Äquivalenzklassen aufgeteilt werden. Wenn zwei Sichten äquivalent sind, können sie in der gleichen Art und Weise in Anfragen verwendet werden — wenn bisher eine Sicht verwendet wurde, kann sie ohne Verlust durch eine andere ersetzt werden. Hierfür ist ein Äquivalenzbegriff nötig, der über die *Signaturen* definiert wird.

Signatur

Es gibt zwei Arten von Signaturen in Bezug auf diese Technik: Sicht- und Anfragesignaturen, die im Folgenden definiert sein sollen. Sei V eine Sicht mit Stelligkeit m, den Kopfvariablen $Z_1, \ldots, Z_m$ und sei Q eine Anfrage. Sei ψ ein vollständiges Variablenmapping von den Variablen von V auf die Variablen von Q. Dann ist die Anfragesignatur von ψ, $sig_Q(\psi)$, definiert als $sig_Q(\psi) :=< \psi(Z_1), \ldots, \psi(Z_m) >$.
Sei Q eine Anfrage und V eine Sicht sowie ϕ ein partielles Variablenmapping von den Variablen von Q auf die Variablen von V. Dann ist die Sichtsignatur von ϕ, $sig_V(\phi)$, definiert als $sig_V(\phi) := (\phi^{head}, atoms(\phi))$, wobei:

- ϕ^{head} sind die Teile von ϕ, die auf Kopfvariablen von V mappen,
- $atoms(\phi)$ sind die Teile von ϕ, die auf Teilziele von V mappen.

Nach dieser Definition sind zwei Sichten V_1 und V_2 genau dann äquivalent, wenn die Sicht- und Anfragesignaturen, projiziert auf die Variablen von V_1 bzw. V_2, gleich sind. Damit ist es möglich, unendlich viele Sichten in eine endliche Anzahl von Sichtäquivalenzklassen (weil es nur endlich viele Signaturen gibt) umzuformen. Bei diesen Äquivalenzklassen ist dann allerdings mindestens eine noch unendlich groß (im Normalfall sogar alle), sodass wir für eine algorithmische Handhabbarkeit eine endliche Repräsentation für unendliche Sichtenmengen benötigen. Hierfür wiederum werden Datalog-Programme verwendet.

2.4.2 Repräsentation von Sichtenmengen

Sichtenmengen werden typischerweise mithilfe von Datalog-Programmen beschrieben. Hierbei gibt es grundsätzlich zwei Arten von Prädikaten:

- EDB-Prädikate (*extensional database-Prädikate*) sind die Prädikate, die tatsächliche Relationen der Datenbank beschreiben. Sie können nur auf der rechten Seite von Regeln auftauchen.
- IDB-Prädikate (*intensional database-Prädikate*) sind die Prädikate, die Sichten beschreiben. Sie können auf beiden Seiten der Regeln auftauchen.

Eine *endliche Ableitung* ist eine endlich lange Abfolge der Anwendung von Regeln. Eine Datalog-Regel definiert auf diese Weise gleichzeitig eine Menge von Sichten: die Menge von Sichten, die durch endliche Ableitungen aus den IDB-Prädikaten entstehen können. Um diese Mengen mit Signaturen zu versehen, wurde das Konzept der *Adornments* entwickelt.

2.4.3 Adornments

Die Adornments, oder Verzierungen, sind ein Tupel von Werten, mit denen die einzelnen Regeln des Programmes *dekoriert* werden. Das Adornment (S_1, S_2) besteht aus einer Menge von Anfragesignaturen S_1 sowie einer Menge von Sichtsignaturen S_2. Dies ist bei den EDB-Prädikaten recht simpel, da auch Relationen (simplifizierte) Sichten darstellen, sie bilden den *Anker* der Verzierungen / Signaturen. Die

Verzierungen der linken Seiten von Regeln müssen berechnet werden, indem alle Kombinationen der Signaturen der rechten Seiten betrachtet werden und inkonsistente Signaturen (d.h. Signaturen, deren Mappings sich widersprechen) eliminiert werden.

2.4.4 Vorgehensweise und Zusammenfassung

Der Ansatz wurde bisher in Top-Down-Manier beschrieben, weshalb hier noch einmal die algorithmische Sicht dargestellt werden soll. Denn algorithmisch sind die Sichtenmengen, die die Fähigkeiten beschreiben und für den AQuV-Prozess nötig sind, gar nicht vorhanden — sie müssen berechnet werden.
Als Basis für einen Algorithmus, der diese Technik benutzt, wird ein Datalog-Programm P benötigt. Für die einzelnen EDB-Prädikate in den Regeln von P werden dann die Sicht- und Anfragesignaturen berechnet und daraus die Adornments.
Aus diesen Adornments der EDB-Prädikate können dann die Adornments der IDB-Prädikate berechnet werden. Die endlichen Ableitungen des IDB-Prädikates x sowie aller Prädikate, die das gleiche Adornment wie x haben, definieren dann die Sichtäquivalenzklassen. Aus diesen Klassen müssen dann Repräsentanten gewählt werden — wie ein Repräsentant gewählt werden soll, ist dabei nicht explizit festgelegt. Es soll ein möglichst kurzer sein, wobei ein tatsächliches Optimum schwer oder aufgrund der unendlichen Mächtigkeit der zu überprüfenden Mengen gar unmöglich zu bestimmen ist, doch da die Sichten *rewriting-äquivalent* sind, kann prinzipiell irgendeine der Ableitungen gewählt werden.
Diese Repräsentanten werden dann mithilfe des Chase, wie beispielsweise des *Provenance-Directed Chase&Backchase*, in die Anfrage eingearbeitet. Aus dem Ergebnis wird dann mithilfe des Backchase eine transformierte Anfrage generiert, die die Sichten und damit auch die Fähigkeiten des Systems nicht mehr verletzen kann.

Die *Answering Queries Using Limited External Query Processors* Technik ist theoretisch fundiert, ist allerdings nur für einige konjunktive Anfragen zu verwenden — lediglich die Prädikate $=$, $\leq$ sowie $<$ sind an eingebauten Prädikaten unterstützt; ebenso müssen die verwendeten Prädikate *lokal* sein, das heißt sie dürfen sich nur auf eine Relation beziehen. Zudem ist der entstehende Algorithmus nicht sehr performant; auch wenn prinzipiell nur eine endliche Ableitung pro Äquivalenzklasse der Sichtprädikate berechnet werden muss, müssen dennoch die Adornments für alle Prädikate berechnet werden. Die Berechnung der Adornments für ein Prädikat ist, aufgrund der exponentiellen Anzahl der möglichen Signaturen (in Anzahl der Prädikate), in NP und damit (vermutlich) nicht effizient berechenbar.

2.5 Schlussfolgerungen

Die beiden vorgestellten Ansätze *Capabilities-based Query Rewriting* sowie *Answering Queries Using Limited External Query Processors* haben tatsächlich mehr gemein, als auf den ersten Blick zu vermuten wäre. So basiert das *Capabilities-based Query Rewriting* auf Grammatiken für Anfragen, in die die ankommende Anfrage so gut es geht eingepasst wird — dies resultiert in den meisten Fällen in einer Destruktion nicht passender Teilziele. Im Gegensatz dazu basiert das *Answering Queries Using Limited External Query Processors* primär auf Datalog-Programmen, die Sichten definieren. Hierbei ist zu beachten, dass alle Sichten, die durch valide Datalog-Programme beschreibbar sind, sich also im Rahmen der Einschränkungen der Technik bewegen, beim Chasen zusätzliche Bedingungen in die Anfragen einbaut, die den gleichen Einschränkungen wie die sichtenerzeugenden Programme unterliegen — so sind nur lokale Bedingungen darstellbar und solche, die nur die erlaubten Operatoren verwenden. Solche zusätzlichen Bedingungen lassen sich aber auch immer in die Template-Grammatiken des *Capabilities-based Query Rewriting* hinzufügen. Tatsächlich lässt sich für jede Anfrage Q und jedes Datalog-Programm P, das den Einschränkungen unterliegt, eine Grammatik finden, die die durch das Chasen der repräsentativen Sichtinformationen in Q und darauf folgenden Backchase transformierte Anfrage Q' auch erzeugen kann. So sind die Sichten-Äquivalenzklassen gewissermaßen mit aus den CSQs generierten Plänen vergleichbar

(bis auf die durchgeführten Chase- und Backchase- Schritte und den inhärenten Anfrageinformationen in den Äquivalenzklassen). Auf die gleiche Weise vergleichbar sind die Terminologien *maximaler CSQ* und *repräsentative Sicht* sowie *Datalog-Programm* und *Anfragegrammatik*. Zur Illustration soll das folgende Beispiel dienen.

Gegeben sei die Anfrage 2.5 sowie die repräsentative Sicht

$$v(R) :- Table(emp), subset(R, emp), substring(lastname,' amp'). \tag{2.8}$$

Dass diese Sicht durch ein geeignetes Programm auf endliche Weise beschrieben werden kann und damit auch repräsentative Sicht sein kann, ist trivial und soll daher hier nicht beschrieben werden. Es ist zu beachten, dass diese Sicht zwar an 2.6 erinnert, aber nicht dasselbe bedeutet — sie ist eine Sicht auf die Datenbank, keine Anfrage. Der Chase findet dennoch genau diesen als *universellen Plan* (nach Anwendung keiner Chase-Schritte, da die Anfrage die Sicht gar nicht verletzt hat). Der Backchase findet dann ein Containment Mapping auf Q mit dem identischen Mapping und gibt ganz U und damit die gleiche Anfrage 2.6, die auch beim *Capabilities-based Query Rewriting* gefunden wurde, zurück.
Der Ansatz des *Answering Queries Using Limited External Query Processors* ist theoretisch fundiert, die Eigenschaften sind beweisbar, was diesen Ansatz exakter macht als das *Capabilities-based Query Rewriting*; er hat aber auch Nachteile. Es ist zu beachten, dass die oben angedeutete Implikation tatsächlich nur in dieser Richtung gilt — wenn es eine Grammatik gibt, die einen bestimmten Plan erzeugen kann, bedeutet dies **nicht** zwangsläufig, dass auch ein Datalog-Programm existiert, deren repräsentative Sichten die Anfrage in den Plan transformieren. Dies folgt unmittelbar aus den geringen Anforderungen an die Grammatik, in die die CSQs eingepasst werden — alle Prädikate sind zugelassen, auch beispielsweise komplexe und nicht lokale. Das bedeutet zwangsläufig, dass das *Capabilities-based Query Rewriting* eine größere Menge von Fähigkeiten unterstützt. Auch eine größere Menge von Anfragen wird vom *Capabilities-based Query Rewriting* unterstützt, da es nicht darauf angewiesen ist, dass die ankommende Anfrage mit dem AQuV-Algorithmus kompatibel ist. Zudem ist die Performance des *Capabilities-based query rewriting* besser, da der exponentielle Faktor von der Anzahl der maximalen CSQs abhängt und nicht von der Anzahl der Prädikate wie im *Answering Queries Using Limited External Query Processors*-Ansatz — im Allgemeinen gibt es mehr Prädikate als maximale CSQs. Des Weiteren sind als Eingabe die Beschreibungen der tatsächlich erlaubten Anfragen vonnöten anstelle der Beschreibungen der Sichtenmengen, was direkter ist und damit die Konfiguration einfacher macht. Zudem sind valide Anfragen typischerweise in den Dokumentationen der Standards beschrieben und es müssen keine Beschreibungen von Sichten entwickelt werden. Ebenso ist eine Erweiterung der Grammatik von Anfragen, wie beispielsweise in Hinblick auf Aggregationen, einfacher als die Erweiterung der möglichen Programme, Sichtenmengen und Chase&Backchase, was beim Ansatz des *Answering Queries Using Limited External Query Processors* nötig wäre. Aus diesen Gründen wird im Folgenden vom *Capabilities-based Query Rewriting* als Grundtechnik ausgegangen.

Tabelle 2.2: Techniken im Vergleich

Kriterium	**Answering Queries Using Limited External Query Processors**	**Capabilities-based Query Rewriting**
Komplexität	exponentiell in Anzahl Prädikate	exponentiell in Anzahl maximale CSQs
Anzahl nötiger Anpassungen zur Erweiterung	4+	1
Mögliche Prädikate	lokale $<$,$\leq$, $=$	ganz RQDL
Konfiguration	Datalog-Programme für beschreibende Sichtenmengen	Grammatiken für Anfragen in RQDL
Unterstützte Anfragen	konjunktive Anfragen mit den Prädikaten $<$, $\leq$, $=$	alle konjunktiven Anfragen

Kapitel 3

Konzept

Bevor nun eigene Konzepte vorgestellt werden, soll nun zunächst ein weiterer Ansatz erläutert werden — es ist tatsächlich so, dass das Projekt PArADISE eigene Ansätze verfolgt, diese aber noch nicht implementiert hat. Tatsächlich gibt es eine Arbeit zu dem Thema, die im Folgenden kurz vorgestellt werden soll.

3.1 Answering Queries using Operators

Answering Queries using Operators (AQuO) im Gegensatz zu *Answering Queries using Views* ist der Name, der dem vorliegenden Problem gegeben wurde. Auch hier gibt es also eine Originalanfrage Q, eine Menge von zulässigen Operatoren auf der unterliegenden Ebene O_i sowie eine Datenbank D. Weiterhin sei Q_i die transformierte Anfrage Q auf der i-ten Ebene.
Gesucht ist dann also ein Rewriting r, sodass gilt:

- $r(Q) = Q_i$
- $Q(D) \sqsubseteq Q_i(D) \Leftrightarrow \forall d \in D : Q(d) \subseteq Q_i(d)$
- Q_i enthält nur Operationen aus O_i
- $\nexists Q' : Q(D) \sqsubseteq Q'(D) \sqsubset Q_i(D)$

Die ersten drei Punkte beschreiben hierbei lediglich, dass die Anfrage Q_i durch Transformation mithilfe von r aus Q entsteht, nur eine Teilmenge der erlaubten Operatoren O verwendet und mindestens die benötigten Tupel (eventuell mehr) aus jeder möglichen Datenbankinstanz zurückliefert. Der vierte Punkt fordert die *Minimalität* der gefundenen Anfrage: es soll keine andere Anfrage existieren, die ebenfalls alle nötigen Tupel zurückliefert und datensparsamer als die gefundene Anfrage Q_i ist. Dies ist die stärkste Anforderung an r — ob sie erfüllt werden kann, muss später gesondert geprüft werden.

3.1.1 Vorbereitung

Um die Anfrage zu verarbeiten, werden sowohl die WHERE- als auch die HAVING-Klauseln der Anfrage (Selektionen auf Tupel- und Aggregationsebene) in eine konjunktive Normalform überführt. Hierdurch ergibt sich schon von vornherein, dass jedes Teilziel (= Klausel der Formel) hochkomplex sein kann — sie wird aber immer eine Disjunktion sein. Die Teilterme können Aggregationen, verschachtelte Anfragen und alle möglichen Arten von Vergleichen beinhalten, doch vereinfacht diese Vorgehensweise den Algorithmus — es ist zu beachten, dass auch das *Capabilities-based Query Rewriting* nur Anfragen mit WHERE-Klauseln in KNF unterstützt und darauf den ganzen Algorithmus anpassen musste, um eine Lösung

finden zu können. Auch dort sind sowohl ursprüngliche als auch transformierte Anfrage in konjunktiver Normalform, wobei die transformierte Anfrage einzelne Teilziele nicht darstellen kann und darum nicht ausführt. Im Allgemeinen wird es nicht immer so sein, dass die Anzahl an Klauseln in der Formel abnimmt — sie könnte in der transformierten Anfrage auch größer sein. So kann der BETWEEN-Operator, sollte er nicht darstellbar sein, mithilfe von jeweils einem $>$- und einem $<$-Operator ausgedrückt werden, womit sich die Anzahl an Klauseln, zumindest für dieses Teilziel, erhöht.

3.1.2 Vorgehensweise

Wenn für jedes Teilziel ein äquivalentes Mapping gefunden werden kann, das die gleichen Ergebnisse zurückliefert, so ist sogar eine äquivalente Anfrage gefunden. Andernfalls muss ein Mapping gefunden werden, dass die Ergebnismenge minimal vergrößert — im schlimmsten Falle der Wahrheitswert **true** — es ist allerdings zu beachten, dass ein solches Mapping immer existiert, auch wenn im schlimmsten Falle alle Daten übertragen werden müssen.
Damit wird die Anfrage also immer in drei Arten von Zielen transformiert:

$$Q := \bigwedge_{G_X} \wedge \bigwedge_{G_Y} \wedge \bigwedge_{G_Z} . \tag{3.1}$$

Hierbei ist G_X die Menge der äquivalent gemapten Teilziele, G_Y die Menge der Obermenge-liefernden Teilziele und G_Z die Menge der nicht gemapten (d. h. auf **true** gemapten) Teilziele.

3.1.3 Verfahren bei nicht äquivalent mapbaren Teilzielen

Gesucht ist dann also eine möglichst ergebnisreduzierende Transformation der Teilziele $G_Y \cup G_Z$ (bzw. bei G_Z, da bei G_Y schon ein Mapping gefunden wurde). Jedoch ist hierzu zu sagen, dass solche Mappings erst gefunden werden müssen — und diese können dann Teilziele aus allen drei Teilen ergeben. Diese Transformation wird im *Answering Queries using Operators*-Ansatz unter anderem mithilfe einer Reduktion der Operatorbäume gefunden. Hierbei werden die einzelnen Teilbäume der Operatorbäume der Teilziele durchmustert, wobei jeweils der höchste Teilbaum, der nur zulässige Operatoren verwendet, auf der unterliegenden Ebene ausgeführt wird. Die Restoperation, die auf der oberen Ebene auszuführen ist, ergibt sich dann aus dem restlichen Operatorbaum für jedes Teilziel [GH17].

3.1.4 Zusammenfassung

Auch wenn es nicht offensichtlich sein mag, so ist dieser Ansatz — wenn auch ohne Implementation — eine Erweiterung des *Capabilities-based Query Rewriting*. Denn die Grammatik, die dort als Basis dient, ist eine Beschreibung möglicher ausführbarer Anfragen. Die Rückwärtsanwendungen der Regeln (bzw. genauer: die Unifikationen, die auf diesen Wegen passieren) sind Transformationen der Teilziele. Hierbei werden durch die Unifikationen diejenigen Teilziele bestimmt, die äquivalent auf ausführbare Teilziele *gemapt* werden können (auch wenn diese Mappings recht trivial sind). Die restlichen Teilziele werden dann im Normalfall weggelassen, also auf **true** gemapt. Damit werden dort äquivalent transformierbare Teilziele gefunden, allerdings nicht alle — ist G_C die Menge aller im gefundenen Plan verbundenen Teilziele, so gilt immer:

$$G_C \subseteq G_X \tag{3.2}$$

Dies gilt, da jedes gefundene Teilziel, wie bereit erläutert, im Grunde genommen durch ein Mapping gefunden wurde, aber andere, eventuell komplexere Mappings — als Beispiel sei nur der BETWEEN-Operator genannt — die in G_X möglich sind, vom *Capabilities-based Query Rewriting* nicht gefunden werden können. Der Ansatz, nicht äquivalent mapbare Teilziele mittels der Auftrennung der Operatorbäume zu transformieren, betrifft $G_Y \cup G_Z$ und ist damit eine weitere echte Erweiterung. Die Implementationen, die im *Capabilities-based Query Rewriting* existieren, können damit auch für diesen Ansatz genutzt werden

(mit Zusatzbehandlung der Teilziele $(G_X \backslash G_C) \cup G_Y \cup G_Z$). Hierbei ist zu beachten, dass die Auftrennung mithilfe der Operatorbäume zwar mehr Teilziele auslagert, aber noch immer nicht genug.
Damit ist auch hier eine weitere Verfeinerung wünschenswert, die im Folgenden gesucht werden soll. Zunächst soll aber die Grundlage gelegt werden: die *Relational Query Description Language* (RQDL), die im *Capabilities-based Query Rewriting* benutzt wird, hat einige in unserem Ansatz unnütze Prädikate (wie beispielsweise das *in*-Prädikat), es fehlen allerdings auch Konstrukte für die Darstellung komplexer Anfragen, wie beispielsweise von Aggregationen. Aus diesem Grund soll nun zunächst eine Abwandlung der RQDL, die *Relational Query Description Language for SQL* (RQDL4SQL), konzipiert werden.

3.2 Relational Query Description Language for SQL

Primäres Ziel der *Relational Query Description Language for SQL* ist die Beschreibung von komplexen SQL-Anfragen mithilfe von konjunktiven Prädikaten. Auf diese Weise können die Ansätze des *Capabilities-based Query Rewriting* und des *Answering Queries using Operators* kombiniert werden. Vordefinierte Prädikate dieser Beschreibungssprache sollen im Folgenden eingeführt und erläutert werden.
Zunächst soll aber das Ziel und die Vorgehensweise bei der Konzeption erläutert werden: die Prämisse der Sprache RQDL4SQL ist die Darstellung möglichst komplexer Anfragen (primär und auch auf SQL basierend) in Prädikatenlogik konjunktiver Form. Dies bedeutet auch, dass für möglichst viele Schlüsselworte und Funktionalitäten in SQL prädikatenlogische Ausdrücke gefunden werden, die die Funktionalität möglichst gut konjunktiv wiedergibt. Eine besondere Bedeutung kommt auch der Kombination der Funktionalitäten in SQL zu (wie beispielsweise SELECT und GROUP BY), denn auch wenn sich die Schlüsselworte auch wechselseitig beeinflussen, kann eine SQL-Anfrage in sieben Phasen, die sequentiell abgearbeitet werden, aufgeteilt werden.

1. Auswahl der Tabellen, angegeben in der FROM-Klausel, gegebenenfalls auch berechnete Tabellen
2. Selektion der Tupel, angegeben in der WHERE-Klausel
3. Gruppierung der Tupel nach Attributmengen, angegeben in der GROUP BY-Klausel
4. Selektion der Gruppen, angegeben in der HAVING-Klausel
5. Umbenennung von Spaltennamen
6. Projektion auf Attributmengen, dabei auch ggf. Angabe der zu verwendenden Aggregatfunktionen für die Attribute in den Gruppen, angegeben in der SELECT-Klausel
7. Sortierung der Ergebnistupel nach Attributwerten

Hierbei sind 2, 3, 4, 5 und 7 optional, alle Punkte können auch mehrfach vorkommen, die Reihenfolge bleibt dennoch die gleiche. Aus diesem Grunde werden auch die Anfrageprädikate von RQDL4SQL in sieben Partitionen aufgeteilt, die in der angegebenen Reihenfolge vorkommen sollen — dies vereinfacht die Auffindung und damit auch die Transformation der Prädikate der verschiedenen Partitionen, die auch unterschiedlich adressiert werden müssen. Durch die Möglichkeit der berechneten Tabellen in der FROM-Klausel kann dort wieder eine Anfrage auftauchen, womit nicht notwendigerweise Prädikate gleichen Typs in einem zusammenhängenden Abschnitt der Prädikatmenge aufzufinden sind — allerdings kann dadurch bei komplexeren Anfragen auch in der Prädikatmenge eine Schachtelung abgelesen werden. Verbunde werden hierbei durch die Angabe mehrerer Basisrelationen (kartesisches Produkt) und darauffolgende Selektion nach Gleichheit der Verbundattribute dargestellt. Dies resultiert auch für die Struktur der Arbeit in einer natürlichen Aufteilung der Anfrage. Hierzu ist zu bemerken, dass Rename-Prädikate für die Transformation eher unwichtig sind, da es keinen SQL-Standard gibt, der das Schlüsselwort AS nicht unterstützt. Die Umbenennungen müssen allerdings in der transformierten oder Restanfrage wieder auftauchen, sodass ein Prädikattyp dafür Sinn macht.

Noch zwei Punkte sollen vor der Einführung der einzelnen Prädikate analysiert werden, die eine solche Serialisierung der Anfrage gefährden: der erste Punkt betrifft die Verschachtelung von Anfragen. So können in der FROM-Klausel einer SQL-Anfrage auch eine durch eine Unteranfrage berechnete Tabelle angegeben werden, die ihrerseits wieder SQL-Anfragen sind. Für die Transformation ist dies allerdings nicht kritisch: Müssen Selektionen von Unteranfragen transformiert werden, so sind mehr Tupel in der Basisrelation vorhanden. Neben einigen Vorkehrungen (wie Erweiterung der Projektion bzw. Gruppierung auf die Selektionsattribute), die aber immer bei der Transformation vorkommen müssen, können dennoch alle Operationen weiterhin ausgeführt werden und die fehlenden Selektionen nachfolgend ausgeführt werden. Müssen Gruppierungen von Unteranfragen transformiert werden, so sind statt der Gruppen die Basistupelmengen im Ergebnis vorhanden. Neben einigen Vorkehrungen (wie Entfernung der Aggregationsfunktionen aus den Projektionsprädikaten) können alle Operationen (außer Gruppierungen auf den Gruppen, die aber bei der Transformation dann auch entfernt werden) weiterhin durchgeführt werden. Die *Rekursionstiefe* bei einer solchen Verschachtelung müsste in einer Repräsentation natürlich mit aufgenommen werden — für die Transformation, für die RQDL4SQL konzipiert wird, ist dies nicht nötig, es können lediglich Abbildungen der Teilziele berechnet und diese dann an der jeweiligen Stelle der Anfrage eingefügt werden.
Der zweite Punkt betrifft die Schachtelung von Prädikaten. Nach der Umwandlung in die konjunktive Normalform, die hier zuerst durchgeführt werden muss, bestehen solche aus Disjunktionen von Prädikaten bzw. deren Negation. Solche Disjunktionen lassen sich leider nicht in Konjunktionen umformen — zwar geben die De Morganschen Regeln[1] solche Umformungen, allerdings kommen hier als Ergebnis Negationen von Mengen von Konjunktionen vor, die für die Anwendung von Grammatikregeln, die eventuell nur für einen Teil der Prädikate Mappings finden können, denkbar ungeeignet sind. Diese Prädikate müssen demnach in der Transformation in RQDL4SQL tatsächlich als Schachtelung von Prädikaten Eingang finden.
Das Startsymbol einer jeden Grammatik G in RQDL4SQL ist *Query*. Auch hier dürfen Nichtterminale, die mit *$* beginnen, nur mit Terminalen unifiziert werden. Eine jede Grammatik hat aufgrund der vorher erwähnten Partitionierung nur eine Produktionsregel mit *Query* auf der linken Seite:

```
Query:- From, Where, Group, Having, Rename, Select, Order.
```

Die Konfiguration der Fähigkeiten einer Sprache erfolgt dann über die Angabe der Ableitungsregeln der Prädikate From, Where, Group, Having, Select und Order (und evtl. Rename), wobei es möglich sein muss, diese Prädikate in endlich vielen Schritten zu einer Menge von Teilzielprädikaten (die hier Terminale sind) zu entwickeln. Erlaubte Teilzielprädikate werden in den folgenden Abschnitten definiert.

3.2.1 From-Prädikate

Das wichtigste Prädikat der From-Partition ist das einstellige *Table(T)*-Prädikat. T bezeichnet hierbei die erlaubten (bzw. bei der Beschreibung einer Anfrage die angefragte) Tabellennamen. Sind die unterstützten Basisrelationen, wie durch eine vertikale Datenbankfragmentierung, in irgendeiner Weise eingeschränkt, so wird hier ausdrücklich die einstufige Regelkonzeption empfohlen, wie zum Beispiel

```
From:- Table('emp').
```

Zwar erlaubt auch die zweistufige Regelkonzeption

```
From:- Table(TableName).
TableName:-'emp'.
```

[1] hier besonders: $a \vee b \Leftrightarrow \neg(\neg a \wedge \neg b)$

die gleichen Anfragen, doch ist dadurch die Rückwärtsanwendung der Regeln für die spätere Transformation unnötig aufwendig. Für die Unterstützung von berechneten Basisrelationen (auch in der Where-Klausel) ist auch

```
From:- Select, Where, Group, Having, Rename, Order, From.
```

erlaubt. Die Änderung der Reihenfolge erfolgt hier zur Vermeidung von direkter und indirekter Linksrekursion, die vielen Parsern Probleme bereitet (vgl. [Par13]). Denkbare Ableitungsmengen für die From-Klausel wären beispielsweise

```
From:- Table($T).
```

oder

```
From:- From, From.
From:- Select, Where, Group, Having, Rename, Order, From.
From:- Table('emp').
From:- Table('dep').
```

Hierbei beschreibt die erste Regelmenge die From-Partition einer Sprache, die nur eine Basisrelation zulässt (d.h. insbesondere keine Verbunde), sonst aber keine Einschränkungen an die Wahl dieser Tabellen vorgibt. Die zweite Regelmenge lässt Joins (durch die Anwendung der ersten Regel lassen sich beliebig viele Table-Prädikate erzeugen) und verschachtelte Anfragen zu, jedoch nur auf den Tabellen 'emp' und 'dep'.

3.2.2 Where-Prädikate

Die Where-Partition beschreibt die Selektionsprädikate der Anfragen, die im Allgemeinen sehr unterschiedlich ausgeprägt sein können. Aus diesem Grunde sind die hier vorgestellten Prädikate zunächst nur als Auswahl zu sehen. An dieser Stelle sei ebenso erwähnt, dass ein nachträgliches Hinzufügen von neuen Prädikaten nicht nur die Anpassung der Grammatik vonnöten macht, sondern auch die Anpassung der Konvertierung SQL in RQDL4SQL und umgekehrt sowie die Entwicklung der später eingeführten Transformationsregeln für diese Prädikate. Bevor die Prädikate definiert werden, muss noch Erwähnung finden, dass die Where-Prädikate häufig Vergleiche sind, die zwei Formen annehmen können: Attribut-Attribut- oder Attribut-Wert-Vergleiche. Ein Spezialfall sind Wert-Wert-Vergleiche wie beispielsweise auch bei Vergleichen von Ergebnissen von Unteranfragen, die allerdings auch als Attribut-Wert-Vergleiche mit konstantem Attribut aufgefasst werden können. Aus diesem Grund müssen hier die Vergleichsprädikate unterschiedliche Ausprägungen für diese Formen zulassen, da nicht immer alle zugelassen sind. Aus diesem Grunde wird bereits an dieser Stelle das nullstellige Prädikat *CompType* definiert, mit den Terminalsymbolen *av* (attribute-value-comparison) und *aa* (attribute-attribute-comparison) sowie den Ableitungsregeln

```
CompType:- av.
CompType:- aa.
```

Der Typ des Parameters lässt sich aus der geparsten Anfrage ermitteln (beispielsweise ob es sich um einen konstanten Integerwert handelt oder um einen Attributnamen). Auf diese Weise lässt sich auch beispielsweise zwischen $Attribut > Wert$ und $Attribut < Wert$ unterscheiden, da beide das gleiche Prädikat verwenden, aber mit Wertparameter an unterschiedlicher Stelle. Nun folgen zunächst die Vergleichsoperatoren. Dies sind Operatoren, die zwei Werte (meist des gleichen Typs) als Parameter haben und einen Wahrheitswert zurückgeben.

Das eq-Prädikat

Der erste der Prädikate, die solche Operatoren darstellen, ist *eq*(*Value1*, *Value2*, *CompType*). Es steht für den '='- (oder 'NOT !='-, die bei der Transformation ebenfalls in eq-Prädikate übersetzt werden sollen) Operator in SQL und ist genau dann wahr, wenn *Value1* und *Value2* gleich sind. Beispiele für die Nutzung von *eq* sind

```
Where:- eq(Value1,Value2,CompType).
```

sowie

```
Where:- Comparison,Where.
Where:- .
Comparison:- eq(Value1,Value2,av).
```

Hierbei gibt die erste Ableitungsregelmenge eine Sprache an, die eine Gleichheitsoperation zulässt, die sowohl Attribut-Attribut als auch Attribut-Wert sein kann. Die zweite Menge lässt eine uneingeschränkte Anzahl von Gleichheitsoperatoren zu, die jedoch alle Attribut-Wert (oder Wert-Wert) sein müssen.

Das gt-Prädikat

Das Prädikat *gt*(*Value1*, *Value2*, *CompType*) steht für die *Value1* > *Value2*-Operation in SQL und ist genau dann wahr, wenn die vergleichbaren Werte / Attribute $Value1$ und $Value2$ nicht $NULL$ sind und der Wert von $Value1$ echt größer als der Wert von $Value2$ ist. Auf die gleiche Weise kann auch der <-Operator dargestellt werden. Um Ambiguität zu vermeiden, werden auch $\leq$ und $\geq$ mithilfe dieses Prädikates dargestellt.

Das bw-Prädikat

Das Prädikat *bw*(*CompValue*, *$Value1*, *$Value2*) steht für den $BETWEEN$-Operator in SQL und wird dann wahr, wenn *Value1* $\leq$ *CompValue* $\leq$ *Value2* gilt, was demnach dem Where-Prädikat *'CompValue BETWEEN Value1 AND Value2'* entspricht. Es ist zu erwähnen, dass manche DBMS (beispielsweise PostgreSQL) eine *symmetrische* $BETWEEN$-Variante anbieten, die auch Tupel liefert, für die *Value2* $\leq$ *CompValue* $\leq$ *Value1* gilt. Dies ist allerdings nicht Standard und es lässt sich in zwei *bw*-Prädikate übersetzen, sodass dies an dieser Stelle nicht weiter betrachtet wird. Hierbei ist die Differenzierung zwischen Attribut-Wert- und Attribut-Attribut-Vergleichen nicht notwendig, da *CompValue* immer den Namen eines Attributes enthält und *Value1* sowie *Value2* immer Konstanten sind. Beispiel für die Nutzung des *bw*-Prädikates ist beispielsweise

```
Where:- bw(CompValue,$Value1,$Value2).
```

Diese Regel gibt an, dass die Where-Klausel aus einem $BETWEEN$-Operator (d.h. einem *bw*-Prädikat) bestehen muss. Andere Prädikate sind, falls nur diese Ableitungsregel für das Where-Prädikat existiert, nicht erlaubt. Zu erwähnen ist, dass hier Einschränkungen von *CompValue* von der Grammatikseite her für die Einschränkung der erlaubten Attribute, auf denen $BETWEEN$-Prädikate angewandt werden können, benutzt werden können, während die weitere Einschränkung von *$Value1* bzw. *$Value2* nur bei einer horizontalen Verteilung der Datenbank für die Transformation der Anfragen sinnvoll ist. Diese Parameter sind allerdings nicht vom verwendeten SQL-Dialekt abhängig, sondern von der Datenbankfragmentierung, weshalb hier zusätzliche Metainformationen (wie schon in den Ableitungsregeln der From-Partition angedeutet) in die Grammatik mit einfließen können und sollen.

Das in-Prädikat

Das Prädikat *in* gab es bereits in RQDL, wird hier aber für einen anderen Wahrheitswert verwendet. Es modelliert den *IN*-Operator in SQL und ist damit der einzige Vergleichsoperator, dessen zwei Argumente nicht den gleichen Typ haben. Beim Prädikat *in*(*Value*, *ValueList*) ist *Value* ein Attributname mit Attributwerten vom Typ T und *ValueList* eine Menge von Werten vom Typen T und modelliert damit den SQL-Ausdruck *Value IN (ValueList[0],ValueList[1],...)*.

Prädikate arithmetischer Operationen

Die bisherigen Prädikate, die Vergleichsoperationen in SQL modellieren, benötigen ein oder mehrere Wert- oder Attributparameter. Diese Parameter müssen aber nicht notwendigerweise Literale sein, sondern können auch durch arithmetische Operationen entstehen, weshalb nun die Prädikate für die arithmetischen Operatoren definiert werden sollen. Hierzu sei auch gesagt, dass diese Parameter nicht nur in den Vergleichsoperationsprädikaten vorkommen können, sondern beispielsweise auch in den später folgenden Prädikaten, die *SELECT* modellieren. Allerdings liefern arithmetische Operationen immer einen nominalen Wert zurück und können daher nicht direkt in der KNF-Form der *WHERE*- oder *HAVING*- Klauseln auftauchen (und damit auch nicht direkt auf der rechten Seite der Ableitungsregeln der *Where*- oder *Having*-Nichtterminale, sondern typischerweise auf der rechten Seite der Ableitungsregeln der *Value*-Nichtterminale).
Zunächst seien hier die binären arithmetischen Operatoren in RQDL4SQL definiert — sie verbinden zwei nominale Werte (oder Attributwerte) zu einem neuen nominalen Wert und kommen typischerweise nicht in den *SELECT*-Klauseln vor (auch wenn dies zulässig ist, ist es eher untypisch). Der erste dieser Operatoren an dieser Stelle ist der +-Operator, in RQDL4SQL durch das +(*Value1*, *Value2*)-Prädikat definiert. Beispielsweise wäre eine Regelmenge wie

```
Value:- $Value.
Value:- AttributeName.
Value:- +(Value,Value).
```

denkbar. Äquivalent sind auch die Prädikate −(*Value1*, *Value2*), ∗(*Value1*, *Value2*), /(*Value1*, *Value2*) sowie %(*Value1*, *Value2*) (% ist der Modulo-Operator) definiert. Allerdings existieren auch n-stellige arithmetische Operationen, die wiederum ein anderer Typus sind (darum sind in den meisten Fällen zwei differenzierte Value-Nichtterminale zu empfehlen). N-stellige arithmetische Operationen (bzw. die Operatoren und in RQDL4SQL die Prädikate, die solche Operatoren darstellen) können nicht in der *WHERE*-Klausel einer SQL-Anfrage auftauchen (es sei denn, es handelt sich um die *WHERE*-Klausel einer äußeren Anfrage, wobei die innere Anfrage eine Gruppierung beinhaltet und aggregierte Attributnamen nicht umbenannt hat), sondern nur in der *SELECT*- und *HAVING*-Klausel. Sie treten im Normalfall (aber nicht notwendigerweise) in Kombination mit Gruppierung auf, wogegen Gruppierung nur mit Aggregation auftauchen kann.
Die n-stelligen arithmetischen Prädikate, die hier definiert werden, aggregieren n numerische Werte zu einem Wert. Hierzu zählen die Prädikate *sum*(*Values*), *avg*(*Values*), *min*(*Values*), *max*(*Values*) und *count*(*Values*), die die Summation, Durchschnittbildung, Minimal- und Maximalbildung sowie die Anzahl der aggregierten Werte beschreiben (wie die gleichnamigen SQL-Funktionale).

Boolesche Prädikate

Neben den arithmetischen Operationen, die die Parameterwerte der Vergleichsoperatoren transformieren, gibt es auch boolesche Operatoren, die die ausgehenden Wahrheitswerte der Vergleichsoperatoren transformieren — nur so sind viele der in SQL möglichen Vergleichsoperatoren in RQDL4SQL darstellbar. Hier gibt es nur drei boolesche Prädikate, zwei davon zweistellig, eines einstellig. Diese sind

and(*Value1*, *Value2*), *or*(*Value1*, *Value2*) sowie *not*(*Value*), die das logische Und, das logische Oder sowie die Negation beschreiben.

3.2.3 Group-Prädikate

Die Group-Prädikate beschreiben lediglich die Art der Gruppierung, weshalb es auch nur eine Art von Prädikat gibt: das *group*(*AttributeName*)-Prädikat. Dieses Prädikat beschreibt lediglich, dass nach dem Attribut *AttributeName* gruppiert wird, also dass alle anderen Attribute in der *SELECT*-Klausel aggregiert sein müssen. Eine typische Group-Ableitungsmenge ist

```
Group:- group(AttributeName),Group.
Group:- .
```

Diese Regelmenge erlaubt die Gruppierung nach beliebig vielen Attributen. Weitere Group-Prädikate existieren nicht; Prädikate für Aggregatfunktionen, die bei Gruppierungen auftauchen, werden im Abschnitt Prädikate arithmetischer Operationen vorgestellt. Data Warehouse-Gruppierungsmöglichkeiten können durch mehrere group-Prädikate dargestellt werden (und sollten sie auch — neue Prädikate für sie würden für Ambiguität sorgen).

3.2.4 Having-Prädikate

Die Prädikate der *Having*-Klausel sind Vergleichsoperatoren, die bereits im Abschnitt Where-Prädikate vorgestellt wurden, mit einer Anpassung: Die beiden Parameter des Prädikates sind nicht mehr nur Attributnamen oder Konstanten, sondern können auch aggregierte Werte sein. Da diese Prädikate aber bereits eingeführt wurden, ist hier kein neues Prädikat erforderlich. Es sollen allerdings Beispielableitungsregeln für die Differenzierung der *Where*- und *Having*- Prädikate gegeben werden.

```
Where:- eq(Value,Value),Where.
Where:- .
Value:- AttributeName.
Value:- $Value.
Value:- +(Value,Value).
Having:- eq(HValue,HValue),Having.
Having:- .
HValue:- Value.
HValue:- sum(AttributeName).
```

3.2.5 Rename-Prädikate

Die Rename-Prädikate beschreiben die zulässigen Umbenennungen in SQL. Das einzige Rename-Prädikat ist *rename*(*Attribute*, *Name*), das den SQL-Ausdruck *Attribute AS Name* modelliert — diese Ausdrücke kommen typischerweise in der SELECT-Klausel vor, sind in RQDL4SQL aber voneinander getrennt. Warum Umbenennungen trotz der relativen Unwichtigkeit für Transformationen, für die RQDL4SQL konzipiert wurde, relevant sind, wurde in der Einleitung 3.2 erläutert.

3.2.6 Select-Prädikate

Das einzige Prädikat, das die *SELECT*-Klausel einer SQL-Anfrage beschreibt, ist das *select*(*Name*)-Prädikat. Es ist das einzige Projektionsprädikat, benötigt aber dennoch nur ein weiteres Prädikat: es werden einfach alle selektierten Attributnamen und Werte (oder arithmetische Operationen von Attributnamen und Werten) mit jeweils einem *select*-Prädikat aufgezählt. Das fehlende Prädikat ist

das *top*($*Number*)-Prädikat, das das *TOP* $*Number* (bzw. *LIMIT* / *ROWNUM*; je nach DBMS)-Schlüsselwort in SQL modelliert. Es existiert auch das $*$-Terminal, das für alle Attribute der erzeugten Relation steht. Ist keine Projektion zugelassen, ist die einzige Ableitungsregel der Select-Partition

```
Select:- select(*).
```

3.2.7 Order-Prädikate

Die Order-Prädikate modellieren die *ORDER BY* und *ORDER BY DESCENDING*-Schlüsselworte in SQL, die eine aufsteigende bzw. absteigende Sortierung der Tupel nach einem Attributwert zulassen. Die Schlüsselworte werden durch die beiden Prädikate *order*(*Name*) bzw. *orderdesc*(*Name*) modelliert, wobei eine Grammatik natürlich mehrere solcher Prädikate erzeugen kann. Hierzu ist zu bemerken, dass diese Partition die einzige ist, in der die Reihenfolge der Prädikate von Bedeutung ist und damit die Kommutativität der Konjunktion verletzt.
Alle vorgestellten Prädikate zusammen decken natürlich nur einen Teil des SQL-Standards ab: Von den relationalen Sprachanteilen der Standards SQL-92 bis SQL:2003 sind mit diesen Prädikaten komplexe Aggregate, Rekursion, Fensterfunktionen, das *DISTINCT*-Schlüsselwort und OLAP-Erweiterungen nicht unterstützt. Ebenso können *user defined functions* (bzw. *stored procedures*, je nach Sprachgebrauch) nicht unterstützt werden — es steht dem Nutzer allerdings frei, zusätzliche Prädikate mit geeigneten Transformationsregeln zu entwerfen. Eine Beispielgrammatik ist in Anhang A.1 gegeben — sie stellt die möglichen Anfragen des TinyDB-Systems dar.

3.2.8 Zur Transformation in RQDL4SQL

Hier sollen lediglich einige Bemerkungen bezüglich der Transformation von SQL in RQDL4SQL und umgekehrt gegeben werden. So ist zunächst zu bemerken, dass RQDL4SQL nicht den ganzen SQL-Standard darstellen kann — sie beschreibt lediglich die Struktur der Konstrukte, die im jeweiligen SQL-Standard zugelassen sind. Auch wenn zusätzliche Prädikate prinzipiell unproblematisch ergänzt werden können, so wird sie doch immer weniger mächtig als SQL sein. Dies macht die Transformation einer geparsten SQL-Anfrage in RQDL4SQL prinzipiell recht simpel — häufig müssen lediglich die Schlüsselworte gesucht und gegebenenfalls Infix-Operatornotationen in das passende Prädikat umgeformt werden. Allerdings bedeutet es auch, dass eine Anfrage, die in RQDL4SQL geschrieben ist, nicht immer ohne zusätzliche Informationen wieder in eine valide SQL-Anfrage transformiert werden kann — aus diesem Grund sollten die geparsten Originalanfragen immer vorgehalten werden, um die Rücktransformation zu unterstützen. An dieser Stelle sei allerdings auch erwähnt, dass prinzipiell nichts eine Anwendung der RQDL4SQL-Sprache für Nicht-SQL-Anfragen verhindert, sofern geeignete Matchings zwischen Ausdrücken der Anfragesprache und den RQDL4SQL-Prädikaten gefunden werden können.
Nun erlaubt RQDL4SQL die Darstellung der grundlegenden Funktionalität des jeweiligen DBMS-Servers als auch die Darstellung einer ggf. sehr komplexen Anfrage mithilfe von Prädikaten. Dies erlaubt die Anwendung des *Capabilities-based Query Rewriting*-Ansatzes auf komplexere Anfragen, allerdings wird immer noch nur der Anteil G_C gefunden. Der folgende Abschnitt behandelt die Transformation von Prädikaten und damit die Erzeugung von $(G_X \backslash G_C) \cup G_Y \cup G_Z$.

3.3 Transformation von RQDL4SQL-Prädikaten

Da nicht alle Teilziele durch Containment Mappings in Ziele überführt werden können, die auf der unteren Ebene darstellbar sind, müssen die noch nicht darstellbaren Teilziele durch *Transformationen* so (ggf. nicht äquivalent oder gar auf **true**) transformiert werden, dass sie ausgeführt werden können. Jede solche Transformation wird im Grunde durch vier Eigenschaften identifiziert:

1. Die mindestens nötigen Operatoren, um das Teilziel ausführen zu können
2. Die mindestens nötigen Operatoren, um das transformierte Teilziel ausführen zu können
3. Eine Beschreibung des Originalteilziels
4. Eine Beschreibung des transformierten Teilziels

Das Originalteilziel kann dabei zum Beispiel durch reguläre Ausdrücke oder durch Operatorbäume mit Platzhaltern dargestellt werden. Zur Beschreibung des transformierten Teilziels sind zunächst einige grundlegende Dinge zu sagen.
Zunächst hängt die auszuführende Transformation nur vom jeweiligen Teilziel (und den gegebenen Operatoren auf der unterliegenden Ebene) ab, ist also auf der linken Seite **lokal** — die Transformation selbst kann aber auch andere Prädikate betreffen. Als Beispiel sei eine fehlende Gruppierungsoperation zu nennen — ohne sie werden die Having-Teilziele, die nicht in gleichbedeutende Where-Teilziele transformiert werden können (also diejenigen mit Aggregation) und die aggregierten Select-Teilziele nicht darstellbar sein und müssen ebenfalls transformiert werden. Es ist allerdings zu beachten, dass diejenigen SQL-Standards, die keine Verbunde unterstützen, auch nur wenig andere Schlüsselworte und damit nur recht simple Anfragen unterstützen [Sch16]. Aus diesem Grund können Verbunde und einfache Selektionen als eine Art *Vorstufe* der komplexeren Anfrageverarbeitung gesehen werden und damit nur **eine** transformierte Anfrage mit **einer** Menge von Restoperationen betrachtet werden. Die Beschreibung des Transformationsergebnisses, in der vorherigen Aufzählung im Punkt 4 subsumiert, zerfällt damit in drei Teile:

1. Prädikate des transformierten Teilziels
2. Prädikate der Restanfrage (d.h. derjenige Teil der Anfrage, der nicht auf der unteren Ebene durchführbar ist und ausgelagert werden muss)
3. Menge von durchzuführenden Resttransformationen auf anderen Prädikaten

Die angesprochenen Resttransformationen sind hierbei als Unterart der normalen Prädikattransformation zu sehen — sie müssen allerdings nicht notwendigerweise über Beschreibungen der vor- und nachliegenden Operatormengen verfügen, falls diese bereits durch die Elterntransformation gegeben ist. Allerdings **kann** eine solche Kindtransformation auch von den verfügbaren Operatormengen abhängen, was die einfachere Beschreibung der Transformation durch das 2-Tupel (Menge zu entfernender Prädikate, Menge hinzuzufügender Prädikate) verhindert. Solche Kindtransformationen sind allerdings **keine** normalen Transformationen mit optionalen Feldern — die dort beschriebenen Transformationen können sich grundlegend von vergleichbaren Elterntransformationen unterscheiden und müssen somit immer gesondert betrachtet werden, um nicht versehentlich eine Folgetransformation bei der Suche nach einer geeigneten Transformation eines Teilzieles zu finden. Eine Transformation des Teilzieles P, bezeichnet als $\Theta(P)$ wird damit durch das folgende 4-Tupel bestimmt:

$$\Theta(P) \;:=\; (O_0, O_1, P, (r(P), Q_{0_\delta}, [\Theta^*(P)])) \tag{3.3}$$

Der Wert einer solchen Transformationsbeschreibung sei die Menge der Prädikate der transformierten Anfrage $(r(P) \cup \bigcup_i \Theta_i^*(P))$. Hierbei müssen die folgenden Bedingungen gelten:

- $\Theta(P)$ enthält nur Operationen aus O_1
- Ist $\Theta_x = (O_{0_x}, O_{1_x}, P_x, (r(P_x), Q_{0_{x_\delta}}, \{\}))$ und
  $\Theta = (O_0, O_1, P, (r(P), Q_{0_\delta}, \{\Theta_1^*, \ldots, \Theta_n^*\}))$, mit
  $Q_{\delta_{\Theta_x}} = Q_{0_{x_\delta}}$ und
  $Q_{\delta_\Theta} = Q_{\delta_{\Theta_1^*}} \circ \cdots \circ Q_{\delta_{\Theta_n^*}} \circ Q_{0_\delta}$.
  Dann: $Q_{\delta_\Theta}(\Theta(P)) = P$

Hierbei ist die erste Bedingung die schon aus AQuO bekannte Anforderungsspezifikation, dass das transformierte Prädikat (und alle betroffenen Teilziele) nur die verfügbaren Operatoren verwenden. Die zweite Bedingung sagt aus, dass die Transformation (in ihrer Gesamtheit) das Ergebnis nicht verändert. Die Transformation der Anfrage Q ergibt sich dann aus der Transformation aller Teilziele:

$$\Theta(Q) = \Theta(\bigcup_{P \in Q} P) = \bigcup_{P \in Q} \Theta(P) \tag{3.4}$$

Die folgenden Abschnitte befassen sich mit dem Finden passender Transformationsregeln der verfügbaren Teilziele. Hierbei ist zu beachten, dass diese Transformationsregeln nur mit der Kenntnis *semantischer* Eigenschaften der modellierten Operatoren erzeugbar sind. Eine automatische Erzeugung einer solchen Transformationsregelmenge wäre also nur mit semantischen Hilfsmitteln wie beispielsweise Ontologien möglich, die allerdings nicht in adäquater Menge und Tiefe verfügbar sind. Aus diesem Grund wird in dieser Arbeit die Wissensbasis manuell hergeleitet.
Hierbei gibt es zu jedem Prädikat zwei Standard-Transformationsregeln, die Ausnahmefälle modellieren. Die erste Regel beschreibt die Transformation eines **darstellbaren** Teilziels — es muss also nicht transformiert werden:

$$\Theta(P) \;=\; (O_0, O_0, P, (P, \{\}, \{\})) \tag{3.5}$$

Die zweite Transformationsregel ist die Fallback-Transformation für den Negativfall. Ist keine andere passende Transformationsregel vorhanden, aber eine Transformation dennoch notwendig, so muss das Prädikat vollständig entfernt und in den Restanfrageteil verschoben werden:

$$\Theta_0(P) \;=\; (O_0, \{\}, P, (\{\}, P, [\Theta_0^*(P)])) \tag{3.6}$$

Hierzu ist zu beachten, dass O_0 und $[\Theta_0^*(P)]$ an dieser Stelle nicht gegeben werden können. Sie sind von der jeweiligen Prädikatausprägung abhängig und werden daher in jedem der folgenden Abschnitte gesondert definiert.

3.3.1 From-Prädikate

Das einzige Prädikat der From-Partition, das an dieser Stelle relevant ist, ist das $Table(TableName)$-Prädikat. Falls es nicht erlaubt ist (d. h. transformiert werden muss), gibt es hierfür zwei Gründe, die unterschiedliche Transformationen benötigen.

Nicht erlaubter Relationenname

Die erste Möglichkeit, die dazu führt, dass ein Table-Prädikat nicht erlaubt ist, ist, dass es sich um eine Relation handelt, die nicht im Fragment vorliegt — d.h. es gibt diese Relation nicht und es muss alles, was mit dieser Relation zu tun hat, aus der transformierten Anfrage entfernt und in den Restteil verschoben werden. Probleme bereiten Select- und Where-Prädikate, die sich auf Attribute der nicht unterstützten Relation beziehen — vom Attributnamen kann ohne Metainformationen nicht auf die beinhaltende Relation geschlossen werden. Aus diesem Grund müssen entweder alle Anfragen, bei denen dies ein Problem sein kann, mit der Syntax *TableName.ColumnName* arbeiten oder die erlaubten Attributnamen in der Grammatik spezifizieren (hierfür muss die **URSA**, die *Universal Relation Scheme Assumption*[2], gelten). Im Folgenden wird die Syntax *TableName.ColumnName* angenommen. Des Weiteren müssen die Verbundattribute in die Projektionsliste mit aufgenommen werden (da die Projektionsliste nie leer ist, ist dies niemals mit Informationsverlust verbunden) und Projektionen auf beiden Teilen müssen ebenfalls in Q_δ übernommen werden. Es ergibt sich die folgende (vereinfachte) Transformationsregel:

[2] Die *Universal Relation Scheme Assumption* sagt aus, dass Attributnamen global eindeutig sein müssen oder zumindest in allen vorkommenden Relationen die gleiche Bedeutung haben.

```
(Table($Table1), {}, Table($Table1), ({}, {}, {
  ({},{},eq($Table1.$JoinAttribute1,$Table2.$JoinAttribute2),
    ({select($Table2.$JoinAttribute2)},{eq($Table1.$JoinAttribute1,$Table2.
    $JoinAttribute2)},
  {
    ({},{},select($Table2.$Attribute),({select($Table2.$Attribute)},{select($Table2.
    $Attribute)},{}))
  })),
  ({}, {}, $pred($Table1.$Attribute,$CompValue,$CompType),
    ({},{$pred($Table1.$Attribute,$CompValue,$CompType)},{})),
  ({},{},select($Table1.$Attribute),({select($Table1.$Attribute)},{select($Table1.
  $Attribute)},{})),
  [...]
})
```

Diese Transformationsregel ist stark verkürzt, da die $\$Table1$-Attribute auch an der zweiten Stelle der jeweiligen Prädikate auftauchen können und das $\$pred$-Prädikat für jedes der *Where*-, *Having*-, *Order*-, *Rename*- und *Select*-Prädikate (in beliebiger Stelligkeit) stehen kann. Ebenso müssen für die *Rename*-Prädikate, die berührte Attributnamen betreffen, die gleichen Transformationsschritte auch für die Umbenennungen durchgeführt werden. Da allerdings Rename-Ketten möglich sind und die Transformationsregeln sich formell immer wieder selbst aufrufen müssten (für jedes *rename*-Prädikat müssen auch die Prädikate mit der betroffenen Bezeichnung durchmustert werden), resultiert dies in Rekursion und wird darum implementatorisch gelöst. Diese Transformation liefert eine Obermenge, da alle Prädikate, die Attribute der nicht unterstützten und einer anderen Tabelle beinhalten (wie u. A. die Join-Bedingung), nicht ausgeführt werden können. Da diese Transformationsregeln sehr technisch und lang sind, werden sie in der Folge lediglich verbal beschrieben.

Beispiel

Diese Transformationsregel, angewandt auf die Anfrage

$$\begin{aligned} &Table('emp'), Table('dep'), eq(dep.EmployeeId, emp.Id), \\ &gt(dep.Value, 2000), select(dep.Description), select(emp.Name) \end{aligned} \tag{3.7}$$

die der SQL-Anfrage

```
SELECT dep.Description, emp.Name
FROM emp, dep
WHERE dep.EmployeeId=emp.Id
AND dep.Value>2000
```

entspricht, ergibt die transformierte Anfrage (angenommen die Tabelle 'dep' wird nicht unterstützt)

$$R = Table('emp'), select(emp.Name), select(emp.Id) \tag{3.8}$$

sowie die Restanfrage

$$eq(dep.EmployeeId, R.emp.Id), gt(dep.Value, 2000), select(dep.Description), select(emp.Name) \tag{3.9}$$

die sich aus der Vereinigung aller Q_δ-Mengen ($= Q_{\delta_\Theta}$) ergibt. Dies ist das erwartete Ergebnis, da die dep-Tabelle bei nicht unterstützter Relation aus einem anderen Fragment kommen muss und die Restanfrage nach Verbund wieder ausführbar wird.

Nicht erlaubter Verbund

Der zweite Grund, dass ein *Table*-Prädikat nicht unterstützt wird, ist ein nicht ausführbarer Verbund (in diesem Fall wird jedes außer das erste *Table*-Prädikat nicht in die Grammatik einpassbar sein). Die notwendige Transformation ist ähnlich zu der, die bei nicht erlaubter Relation durchzuführen ist — es muss allerdings nur der Verbund auf der oberen Ebene durchgeführt werden ($eq(\$Table1.\$JoinAttribute1, \$Table2.\$JoinAttribute2)$) sowie eventuell Vergleiche zwischen Attributen unterschiedlicher Relationen, die restlichen Prädikate können auf der unteren Ebene ausgeführt werden. Hier kommt zum tragen, was bereits bei 3.3 angedeutet wurde: Da der Verbund nicht unterstützt wird, stehen auch nur begrenzt viele weitere Operatoren zur Verfügung — die transformierte Anfrage ist recht simpel (im Normalfall nur einfache Selektionen) und die auszuführenden Anfragen (da keine Verbunde zur Verfügung stehen, sind dies mehrere) ergeben sich durch Aufteilung der Prädikate auf involvierte Relation. Eine entsprechende, verkürzte Transformationsregel ist im Anhang A.2 zu finden. Zu beachten ist, dass das *Table*-Prädikat wieder auftauchen muss (da die entsprechende Relation erlaubt ist, aber nicht in Kombination mit einer anderen) und daher folgende Untersuchungen, die die Durchführbarkeit der transformierten Anfrage sicherstellen sollen, auf den einzelnen Teilanfragen-Prädikatmengen durchgeführt werden müssen und nicht auf der gesamten Prädikatmenge.

Beispiel

Die bereits bekannte Beispielanfrage

$$\begin{aligned} &Table('emp'), Table('dep'), eq(dep.EmployeeId, emp.Id), \\ &gt(dep.Value, 2000), select(dep.Description), select(emp.Name) \end{aligned} \tag{3.10}$$

mit dieser Transformation ergibt die beiden Teilanfragen

$$R_1 = Table('emp), select(emp.Name), select(emp.Id) \tag{3.11}$$

und

$$R_2 = Table('dep'), select(dep.EmployeeId), gt(dep.Value, 2000), select(dep.Description) \tag{3.12}$$

sowie der Restanfrage

$$eq(R_2.dep.EmployeeId, R_1.emp.Id), select(R_2.dep.Description), select(R_1.emp.Name), \tag{3.13}$$

was auch das zu erwartende Ergebnis ist.

3.3.2 Where-Prädikate

An dieser Stelle werden lediglich die Vergleichsoperatoren betrachtet, die booleschen Prädikate sowie die 2-stelligen arithmetischen Operatoren folgen in den darauffolgenden Abschnitten, während die n-stelligen Prädikate im Rahmen der *Having*-Prädikate betrachtet werden sollen. Die Transformationen der Vergleichsoperatoren sind hierbei untereinander gleichartig aufgebaut.

eq-Prädikat

Falls ein eq-Prädikat nicht in die Grammatik eingepasst werden kann, gibt es hierfür vier Möglichkeiten, warum (dies beschreibt den O_0-Teil der Transformationsregel): Zunächst ist es möglich, dass die angegebene Sprache **überhaupt keine** Gleichheitsoperationen zulässt. Für die Transformation gibt es dann vier Möglichkeiten, die grundlegend von der Fähigkeit der unterliegenden Ebene abhängen (also den O_1-Teil der Transformationsregel beschreiben).
Ist kein *eq*-Prädikat erlaubt, aber das *in*-Prädikat (da *IN* in SQL intern auf Gleichheit überprüft,

ist dies sehr unwahrscheinlich, die Möglichkeit soll dennoch gegeben werden), so ist es äquivalent in ein solches zu überführen: $eq(x, y) \Leftrightarrow in(x, (y))$. Ist das *in*-Prädikat hingegen nicht erlaubt, aber das *gt*- und das not-Prädikat, so kann es äquivalent in die $\leq \wedge \geq$-Operatoren überführt werden: $eq(x, y) \Leftrightarrow not(gt(x, y)), not(gt(y, x))$. Sind allerdings keine zwei Prädikate erlaubt, so lässt sich das Prädikat nicht mehr äquivalent transformieren. Sind sowohl $not()$ als auch $gt()$ erlaubt, so lässt es mithilfe dieser darstellen: $eq(x, y) \Rightarrow not(gt(x, y))$. Hierbei können die Parameter natürlich auch getauscht werden, ob dies allerdings besser wäre, lässt sich nur mit Kenntnis der Attributwertverteilung herausfinden, die an dieser Stelle nicht gegeben ist — *sollte* die Attributwertverteilung bekannt sein, ist $not(gt(x, y))$ immer dann performanter (d. h. seltener wahr), wenn y kleiner als der Median des Attributes ist. Da hiervon allerdings nicht auszugehen ist, wird dies nicht weiter behandelt. Ist auch dies nicht möglich, muss das Prädikat vollständig entfernt werden. Bei den letzten beiden Transformationsregeln (denen, die keine Äquivalenz erlauben), muss das Prädikat zudem in den Q_δ-Teil verschoben werden.
Der zweite Grund besteht darin, dass zwar Gleichheitsoperationen möglich sind, aber nicht genügend viele. Hier ist tatsächlich keine Transformation notwendig: der *Capabilities-based Query Rewriting*-Ansatz wird hier lediglich zwei (oder mehr) CSQs finden, die dann über einen Verbund kombiniert werden können — der resultierende Plan wird sämtliche Gleichheitsbedingungen erfüllen. Der dritte Grund besteht darin, dass das *eq*-Prädikat vom Typ *aa* ist, aber nur *eq*-Prädikate vom Typ *av* unterstützt werden. Hier ist tatsächlich keine Transformation möglich, da keine Informationen über Integritätsbedingungen o.Ä. vorliegen, das Prädikat muss vollständig in Q_δ verschoben werden.
Der vierte mögliche Grund ist, dass genügend viele *eq*-Prädikate des notwendigen Typs erzeugt werden können, aber die einzelnen Parameter nicht in der Grammatik darstellbar sind, wie beispielsweise durch fehlenden Additionsoperator. Dann müssen zur Transformation des Prädikates die beiden Parameter transformiert werden: $\Theta(eq(x, y)) = eq(\Theta(x), \Theta(y))$. Hierbei ist zu beachten, dass dies nur gilt, wenn für beide Parameter äquivalente Umschreibungen gefunden werden können. Ist dies für ein Teilziel oder beide nicht der Fall, wird $eq(x, y)$ vollständig in Q_δ verschoben (ohne Hinzufügungen im $r(P)$-Teil). Aus diesem Grund werden später Transformationen von Wahrheits- und anderen Werten gegeben, die zwar nicht notwendigerweise die Bezeichnung *Teilziel* verdienen, aber dennoch transformiert werden müssen.

gt-Prädikat

Ist ein *gt*-Prädikat nicht unterstützt, sind die Gründe hierfür die gleichen wie beim *eq*-Prädikat, auch viele Transformationen lassen sich übernehmen. Sind keine *gt*-Prädikate unterstützt, kann es nicht äquivalent transformiert werden. Tatsächlich gibt es hier nur zwei mögliche Transformationen:
Sind sowohl das *eq*- als auch das *not*-Prädikat unterstützt, kann der $>$-Operator in einen $!=$-Operator transformiert werden: $gt(x, y) \Rightarrow not(eq(x, y))$. Ist dies nicht möglich, so muss es ersatzlos in den Q_δ-Teil verschoben werden.
Die Transformation bei einer Nicht-Unterstützung der benötigten Anzahl von Where- oder gt-Prädikaten ist identisch mit der, die schon bei den eq-Prädikaten gegeben ist: keine Transformation ist nötig, die Prädikate können in mehrere CSQs aufgeteilt werden. Auch die Transformation bei fehlender $gt(x, y, aa)$-Operation ist identisch mit der für das *eq*-Prädikat vorgestellten - das Prädikat muss vollständig in Q_δ verschoben werden.
Ist hingegen ein Parameter nicht darstellbar, so ist hier mehr Freiraum gegeben — es müssen zwar auch hier die beiden Parameter durch Transformationen dargestellt werden, aber nicht notwendigerweise äquivalent: Beim Prädikat $gt(x, y)$ kann x ggf. auch vergrößert werden, während y verkleinert werden darf. Dies zeigt, dass Werttransformationen in drei Ausprägungen vorkommen müssen: Eine äquivalente Θ-Transformation, wie sie bereits bei der Transformation der eq-Parameter Verwendung findet, eine Θ_+-Transformation, die (auch) größere Werte zurückliefern darf, sowie eine Θ_--Transformation, die kleinere Werte erlaubt. Damit ergibt sich für die Transformation in diesem Falle $\Theta(gt(x, y)) = gt(\Theta_+(x), \Theta_-(y))$ (mit $gt(x, y)$ in den Q_δ-Teil, falls ein Parameter nicht äquivalent transformiert werden kann).

bw-Prädikat

Bei der Nicht-Unterstützung eines bw-Prädikates gibt es im Grunde vier Gründe, die unterschiedliche Transformationen nötig machen. Die ersten zwei dieser Gründe sind bereits bekannt, weshalb auch die dazugehörigen Transformationsvorschriften sich nur wenig unterscheiden: Sind nicht genügend bw-Operatoren in die Grammatik einfügbar, so werden mehrere CSQs gefunden, der Verbund dieser wird alle Prädikate erfüllen. Sind keine bw-Prädikate in der Sprache erlaubt, so lässt sich dieser durch die $\leq$ und $\geq$-Operatoren ausdrücken:
$bw(CompValue, \$Value1, \$Value2) \Leftrightarrow and(not(gt(\$Value1, CompValue)), not(gt(CompValue, \$Value2)))$.
Die restlichen Fälle betreffen die Parameter: es kann entweder *CompValue* oder einer der *Value*-Parameter nicht unterstützt sein. Ist *CompValue* nicht unterstützt (im Normalfall durch eine vertikale Verteilung auf Attributebene), muss das Prädikat vollständig in Q_δ verschoben werden, wobei zusätzlich Selektionen nach einer identifizierenden Attributmenge nötig werden (um die Fragmente ggf. wieder verbinden zu können). Sind solche Transformationen gewünscht, ist also von vornherein die Angabe einer solchen Attributmenge vonnöten. Ist hingegen einer der Werte nicht unterstützt, so ist er außerhalb der Grenzen (beispielsweise durch horizontale Fragmentierung), die Anfrage ist allerdings dennoch ohne Transformation ausführbar.

in-Prädikat

Ist ein *in*-Prädikat nicht in die Grammatik einpassbar, ist dies auf zwei Gründe zurückzuführen: Zunächst ist es möglich, dass das Attribut nicht im betrachteten Fragment vorliegt. In diesem Fall muss auch hier das Prädikat in Q_δ verschoben werden. Die andere Möglichkeit besteht darin, dass der *in*-Operator tatsächlich nicht unterstützt wird. Dieser kann dann auf zwei Arten transformiert werden: Sind das *eq*- und das *or*-Prädikat in beliebiger Anzahl erzeugbar, kann das *in*-Prädikat (das aus einer endlichen und insbesondere festen Anzahl von Werten besteht) in ebenso viele *eq*-Prädikate transformiert werden: $in(Attribute, (ValueList[0], \ldots, ValueList[n\text{-}1])) \Leftrightarrow or(or(\ldots or(eq(ValueList[0], Attribute), eq(ValueList[1], Attribute)\ldots), eq(ValueList[n\text{-}1], Attribute))$.
Dieser Baum kann gegebenenfalls balanciert werden, sodass bei einer darauffolgenden Transformation möglichst viele Teilbäume abgespaltet werden können; im ersten Prototyp wird allerdings die lineare Variante realisiert. Ist dies nicht möglich, so ist die nicht äquivalente Umformung in das *bw*-Prädikat möglich: $in(Attribute, ValueList) \Rightarrow bw(Attribute, min(ValueList), max(ValueList))$. Auch wenn das *bw*-Prädikat nicht unterstützt wird, ist diese Transformation zu empfehlen, wenn eine äquivalente Transformation ausgeschlossen ist, da hier nur ein Prädikat durch Folgetransformationen umgeformt werden muss.

3.3.3 Prädikate zweistelliger arithmetischer und boolescher Operatoren

Die zweistelligen arithmetischen Operatoren können (fast) alle mit den gleichen Transformationen versehen werden: Ist die tatsächliche Operation nicht erlaubt (das heißt ist beispielsweise kein $+(Value1,Value2)$ darstellbar), so kann es nur dann äquivalent transformiert werden, wenn beide Parameter Werte sind und das Ergebnis (in diesem Fall die Summe) für das Prädikat eingesetzt wird. Andernfalls muss das Prädikat in den Q_δ-Teil verschoben werden, mit zusätzlichen *select*()-Prädikaten für sowohl die linke als auch die rechte Seite. Ist hingegen einer der Parameter nicht unterstützt, so müssen sie transformiert werden. Diese Transformationsregeln unterscheiden sich je nach Rechenart:

- $\Theta_+(+(x,y)) \Rightarrow +(\Theta_+(x), \Theta_+(y))$
- $\Theta_-(+(x,y)) \Rightarrow +(\Theta_-(x), \Theta_-(y))$
- $\Theta_+(*(x,y)) \Rightarrow *(\Theta_+(x), \Theta_+(y))$
- $\Theta_-(*(x,y)) \Rightarrow *(\Theta_-(x), \Theta_-(y))$
- $\Theta_+(-(x,y)) \Rightarrow -(\Theta_+(x), \Theta_-(y))$

- $\Theta_-(-(x,y)) \Rightarrow -(\Theta_-(x), \Theta_+(y))$
- $\Theta_+(/(x,y)) \Rightarrow /(\Theta_+(x), \Theta_-(y))$
- $\Theta_-(/(x,y)) \Rightarrow /(\Theta_-(x), \Theta_+(y))$

Hierbei mag aufgefallen sein, dass das %-Prädikat hier keine Erwähnung findet. Dies liegt nicht nur daran, dass es keinen solchen Rechenregeln folgt (es muss sogar äquivalent — oder vollständig in Q_δ transformiert werden, da nicht immer berechenbar ist, ob eine gegebene Transformation es verringert oder vergrößert), sondern auch, dass eine weitere Transformation existiert: $x \ mod \ y = x - y * (x \ div \ y)$. *div* ist hier die ganzzahlige Division.
Die booleschen Operatoren hingegen stellen eine der größten Herausforderungen dar, die für die Transformation existieren. Der simpelste ist die Konjunktion, also das *and*-Prädikat. Da RQDL4SQL konjunktiv aufgebaut ist, kann ein *and*-Prädikat nur innerhalb eines anderen Prädikates auftauchen. Hier gibt es (wie bei allen in diesem Abschnitt vorgestellten Prädikaten) zwei Gründe für eine nötige Transformation: Ist das *and*-Prädikat an dieser Stelle nicht erlaubt, so kann es nicht äquivalent transformiert werden. Zwar ist eine Aufteilung in mehrere CSQs, wie an anderer Stelle schon angedeutet, denkbar, dies würde aber evtl. äußere Aggregationen verfälschen. Aus diesem Grund kann dies nur im Where-Teil angewandt werden oder in Grammatiken, die keine Aggregation erlauben. Andernfalls muss tatsächlich ein Teilziel entfernt werden, wobei hierbei das Prädikat mit der geringeren Rekursionstiefe empfohlen wird, da sämtliche verwendeten Attribute im entfernten Teilziel zur Projektionsliste im *Select*-Teil hinzugefügt werden müssen (und das **gesamte** Prädikat zum Q_δ-Teil hinzugefügt werden muss) und dies bei flacheren Bäumen tendenziell weniger sind. Ist hingegen eines der Teilprädikate des *and*-Prädikates nicht darstellbar, so sind diese, wie bereits häufiger gesehen, zu transformieren und die transformierten Teilziele einzusetzen. Das gleiche trifft auch auf die *or*-Prädikate zu, deren Teilprädikate nicht darstellbar sind. Zwar kann das gesamte Prädikat entfernt werden, wenn nur ein Teilziel auf **true** gemapt werden muss, doch muss sich damit die Transformation nicht befassen — da bei einer nicht äquivalenten Transformation das ganze Prädikat in Q_δ verschoben werden muss, kann der Anfrageoptimierer eventuell ausstehende (äquivalente) Resttransformationen vornehmen. Sollte das *or*-Prädikat selbst allerdings nicht darstellbar sein, müsste eine (im mengentheoretischen Bilde) minimale Obermenge zu der Vereinigung zweier Mengen gefunden werden, die bestimmten Bedingungen (den verfügbaren Operatoren) genügt. Dieses Problem ist wenig erforscht und eine Ausarbeitung an dieser Stelle würde den Rahmen dieser Arbeit sprengen, weshalb bei einer Nicht-Darstellbarkeit des *or*-Prädikates das gesamte Prädikat in Q_δ verschoben und somit auf **true** gemapt werden muss.
Auch das *not*-Prädikat ist schwierig zu transformieren, wenn auch in diesem Fall in technischer Weise. So kann bei einer Nicht-Darstellbarkeit des *not*-Prädikates selbst tatsächlich kein anderes Prädikat gefunden werden, das eine ähnliche Funktionalität aufweist. Das ganze Prädikat muss in diesem Fall auf **true** gemapt und in den Q_δ-Teil verschoben werden. Ist hingegen der Parameter nicht darstellbar, so muss er transformiert werden — allerdings nicht mit einer normalen oder einer Kindtransformationsregel. Soll ein $not(x)$-Prädikat eine größere Tupelmenge liefern, so muss das x-Prädikat eine kleinere Tupelmenge liefern. Dies bedeutet, dass es für alle Prädikate, die Parameter des *not*-Prädikates sein können, spezielle Transformationsregeln geben muss, die eine Teilmenge (oder die gleiche Menge) von Tupeln zurückliefern und mit $\Theta_-(x)$ bezeichnet werden sollen. Da diese bisher nicht behandelt wurden, wird sich der folgende Abschnitt mit den Θ_--Transformationsregeln der bereits behandelten Prädikate befassen.

3.3.4 Θ_--Transformationsregeln

Dies sind die Θ_--Transformationsregeln der bisher behandelten Prädikate, die restlichen (wahrheitswertigen) Prädikate werden in den jeweiligen Abschnitten Erläuterungen zu den korrespondierenden Θ_--Transformationen erhalten. Hierbei sind natürlich nur diejenigen Prädikate zu betrachten, die einen tatsächlichen Wahrheitswert für die folgende Anfrageauswertung zurück liefern. Prädikate wie beispielsweise das *Table*-Prädikat, die lediglich aussagen, ob die Teilanfrage ausführbar ist, müssen nicht betrachtet

werden. Damit sind an dieser Stelle nur diejenigen Prädikate, die aus dem Where-Nichtterminal entstehen können, zu betrachten. Dabei sind Θ_--Transformationsregeln genau so aufgebaut wie die üblichen Transformationsregeln Θ. Auch sie bestehen also aus einer Menge von nötigen Operatoren für die Ausführung von P, einer Menge von nötigen Operatoren für die Ausführung von $r(P)$, einer Beschreibung von P sowie einem Tupel für das Transformationsergebnis (der Wert der Transformation), das aus einer Beschreibung von P, einer Restoperationenmenge Q_δ sowie einer Menge von Kindtransformationen $[\Theta^*]$ besteht.
Zunächst ist hier das *eq*-Prädikat zu betrachten: Auch hier gibt es für die Nicht-Einpassbarkeit des Prädikates vier Gründe. Falls die angegebene Sprache überhaupt keine Gleichheitsoperationen zulässt, so sind zunächst die beiden äquivalenten Transformationen in der entsprechenden Θ-Regel 3.3.2 zu verwenden. Allgemein können bei solchen Θ_--Transformationen immer auch die jeweiligen Θ-Transformationen verwendet werden, die eine Äquivalenz darstellen, weshalb diese Fälle im Folgenden nicht mehr betrachtet werden müssen. Sind hingegen die *in*- oder die *not*- und *gt*-Prädikate nicht zur Transformation verfügbar, so ist das *eq*-Prädikat vollständig zu entfernen (da $\{x \in A : x \neq y\} \subseteq \{x \in A\}$) und in den Q_δ-Teil zu verschieben. Dass nicht genügend *eq*-Prädikate erlaubt sind, muss nicht überprüft werden, da Θ_--Transformationen nur im *not*-Prädikat vorkommen und dort nur ein Teilprädikat vorkommen kann. Die restlichen Transformationen unterscheiden sich, da es sich um Teilziele in G_X oder G_Z handelt, nicht von den bereits vorgestellten Θ-Transformationen.
Ist das *gt*-Prädikat nicht darstellbar, fällt die (nicht äquivalente) Umformung in $not(eq(x, y))$ für die Θ_--Transformation weg — tatsächlich gibt es kein Obermenge-lieferndes Mapping dieses Teilzieles (in Θ_-) außer das auf **true**, sodass das ganze Prädikat in Q_δ verschoben werden muss. Ist hingegen einer der Parameter nicht darstellbar, so müssen sie transformiert werden. Da allerdings $\neg(x > y) \Leftrightarrow x \leq y$ gilt, müssen sie mit Toleranz in die jeweils andere Richtung transformiert werden: $\Theta_-(gt(x, y)) = gt(\Theta_-(x), \Theta_+(y))$.
Beim *bw*-Prädikat ist keine Anpassung nötig, auch bei der Transformation des *in*-Prädikates gibt es nur eine Transformationsregel, die angepasst werden muss: Die nicht äquivalente Umformung $in(Attribute, ValueList) \Rightarrow bw(Attribute, min(ValueList), max(ValueList))$ fällt weg, da sie sich informationserhöhend auswirkt. Es ist allerdings die Transformation $\Theta_-(in(Attribute, ValueList)) = eq(Attribute, ValueList[0], av)$ möglich, die allerdings, abhängig von der Länge von *ValueList*, nur eine sehr kleine Teilmenge zurückliefert.

3.3.5 Prädikate n-stelliger arithmetischer Operatoren

Für alle Prädikate, die einen n-stelligen arithmetischen Operator modellieren, gibt es zwei Gründe für eine Nicht-Darstellbarkeit. Entweder ist die Operation nicht erlaubt oder aber die Spalte, auf der die Operation ausgeführt werden muss, ist nicht unterstützt. Ist die Spalte, die angefragt wird, nicht verfügbar, so darf die Gruppierung nicht auf diesem Fragment allein durchgeführt werden (da sich durch das Hinzufügen weiterer Fragmente andere Gruppen ergeben können). Dadurch müssen die Prädikate in Selektionen, die Aggregation beinhalten, durch die jeweiligen Spaltennamen ohne Aggregation ersetzt werden. Zudem müssen alle Gruppierungen entfernt werden und mitsamt sämtlichen weiteren Prädikaten, die Aggregation nutzen, in den Q_δ-Teil verschoben werden. Wie transformiert werden muss, sollte die Operation nicht verfügbar sein, ist vom jeweiligen Prädikat abhängig, wobei hier zu beachten ist, dass die Anzahl der Operanden **beliebig und variabel** ist. Dies bedeutet, dass beispielsweise die *SUM*-Operation nicht durch $n-1$ Additionen ausgedrückt werden kann, da hierfür für die Berechnung der Anfrage eine weitere Datenbankanfrage vonnöten wäre — was allerdings die datensparsame Transformation ad absurdum führt — während die Transformation des *in* in die verschiedenen *eq*-Prädikate nur darum möglich war, da die Anzahl der Operanden **beliebig, aber fest** war. Aus diesem Grund gibt es hier in der Regel keine äquivalenten Transformationen.

sum-Prädikat

Das *sum*-Prädikat bezeichnet die Summation des im Parameter gegebenen Attributwertes. Ist Summation nicht erlaubt, gibt es keine günstige Transformation, die das gleiche Ergebnis zurückliefert. Hierbei gibt es verschiedene Möglichkeiten, wo das *sum*-Prädikat auftaucht: entweder im *select*- oder im *having*-Prädikat. Im *select*-Prädikat kommt es auf die Art der weiteren Selektionen an, wie verfahren werden muss: Sei G eine beliebige Gruppe, die durch ein zusätzliches Gruppierungsattribut in die Gruppen $G_1, \ldots, G_n$ zerfällt. Gibt es für alle Aggregatsfunktionen a im *select*-Teil und alle Gruppen G eine Funktion f_a, sodass

$$a(G) = f_a(\bigcup_{i=1}^{n} a(G_i)) \tag{3.14}$$

gilt, kann das Prädikat $select(sum(A))$ in $group(A)$ mit einer Zusatzberechnung, die die Gruppen mit gleichen Werten auf allen Attributen außer A zusammenfasst, transformiert werden. Die Summation erfüllt diese Bedingung, mit $f_{sum} = (G_1, \ldots, G_n) \rightarrow \sum_{i=1}^{n} sum(G_i)$. Gibt es hingegen eine Aggregatfunktion, die die Bedingung nicht erfüllt, so kann $sum(A)$ mithilfe von $*(count(A), avg(A))$ wieder berechnet werden. Ist dies nicht möglich, so müssen aufgrund der unterschiedlichen Gruppengrößen die Aggregatfunktionen im *select*-Teil entfernt und vollständig in Q_δ verschoben werden. Diese Transformationen sind für alle Aggregatsfunktionen die gleichen, weshalb in Folge nur ggf. die Funktion f_a gegeben wird, womit dann die Aggregatfunktion im *select*-Teil eventuell transformiert werden kann.
Ist die Summation im *having*-Teil, so handelt es sich um einen numerischen Wert und es gibt drei Möglichkeiten, wie die Transformation ausgeführt werden muss: Sie kann

- zwingend äquivalent (Θ),
- möglicherweise vermindernd (Θ_-) oder
- möglicherweise vergrößernd (Θ_+)

sein. Ist sie zwingend äquivalent (dies ist natürlich auch bei den anderen Transformationsarten möglich), ist die einzige Transformationsart in $*(count(A), avg(A))$. Ist dies nicht möglich, muss das Prädikat entfernt werden. Darf es vermindert werden (Θ_-), aber die äquivalente Transformation nicht möglich, so ist eine Abschätzung auf sowohl $avg(A)$, $max(A)$ als auch auf $*(count(A), min(A))$ erlaubt. $min(A)$ ist zwar auch immer kleiner als $sum(A)$, es gibt allerdings keine Sprache die max, aber nicht min erlaubt, und alle vorgestellten Transformationen sind größer (also insbesondere auch $max(A)$). Hierbei gilt typischerweise (d. h. bei zu erwartenden Datenbeständen mit relativ kleiner Standardabweichung — $\sigma \leq count(A)$) $avg(A) \leq max(A) \leq *(count(A), min(A))$ (siehe auch [Tür99]), sodass die Transformationen mit der Priorisierung auf den größeren vorgenommen werden sollten. Ist hingegen eine Vergrößerung möglich, so ist eine Abschätzung mit $*(max(A), count(A))$ möglich.

avg-Prädikat

Die Durchschnittsbildung erfüllt die Bedingung 3.14, sofern die Anzahl der Elemente in den Fragmenten vorliegt. Dies kann durch eine Transformation bewerkstelligt werden, wenn das *select*-Prädikat Parameter der Form $count(\$ColumnName)$ unterstützt. Dann ist $f_{avg} = (G_1, \ldots, G_n) \rightarrow \frac{\sum_{i=1}^{n} (avg(G_i) * count(G_i))}{\sum_{i=1}^{n} count(G_i)}$ die Zusammenführungsfunktion.
Im *having*-Teil gibt es die äquivalente Transformation $avg(A) = /(sum(A), count(A))$. Ist eine Verminderung möglich, so kann $min(A)$ betrachtet werden, bei einer möglichen Vergrößerung $max(A)$.

min-Prädikat

Die Minimumbildung erfüllt die Bedingung 3.14, $f_{min} = min$ ist die Zusammenführungsfunktion. Eine Abschätzung nach oben ist mit $avg(A)$ gegeben bzw. der äquivalenten Umschreibung $/(sum(A), count(A))$. Eine Abschätzung nach unten kann ohne Kenntnis der Verteilung von A nicht gegeben werden, das gesamte Prädikat muss in Q_δ verschoben werden.

max-Prädikat

Die Maximumbildung erfüllt die Bedingung 3.14 mit $f_{max} = (G_1, \ldots, G_n) \rightarrow max(max(G_1), \ldots, max(G_n))$. Im *having*-Teil ist keine äquivalente Transformation möglich, aber eine vermindernde zu $avg(A)$ oder der äquivalenten Umschreibung und auch eine vergrößernde zu $sum(A)$.

count-Prädikat

Die Anzahlbildung erfüllt ebenfalls die Bedingung 3.14 mit $f_{count} = (G_1, \ldots, G_n) \rightarrow \sum_{i=1}^{n} count(G_i)$, womit alle (vorgegebenen) Aggregatfunktionen diese Bedingung erfüllen und eine Umformung, wie in 3.3.5 vorgestellt wurde, immer möglich ist. Tritt das *count*-Prädikat hingegen im *having*-Teil auf, so ist es das einzige der vorgestellten, das immer positiv ganzzahlige Werte liefert und kaum mit anderen Prädikaten vergleichbar ist. Lediglich die äquivalente Umformung in $/(sum(A), avg(A))$ ist möglich, lassen die Prädikate dies nicht zu, ist das Prädikat vollständig in Q_δ zu verschieben.

3.3.6 Group-Prädikate

Es gibt nur ein Prädikat der *Group*-Partition, *group*. Hierbei gibt es drei Möglichkeiten, warum das Prädikat nicht verfügbar ist. Zunächst ist es möglich, dass das Attribut, nach dem gruppiert wird, nicht verfügbar ist. Hierdurch ergeben sich im Allgemeinen größere Gruppen (da mehr Äquivalenzen) sowie ein Nicht-Auftauchen des Attributes im Ergebnis. Beide Symptome zusammen sind problematisch für die Transformation: Zwar würde das Ergebnis in jedem Fall das Attribut nicht exportieren können, weil es dieses im Fragment nicht gibt — es müsste auch beim gegenwärtigen *FetchAll*-Ansatz durch einen Verbund realisiert werden. Bei der Transformation ist ein solcher Verbund aber im Allgemeinen nicht möglich, da auch die anderen Fragmente nicht alle Attribute haben müssen und damit die Originalwerte der in größeren Gruppen berechneten Aggregatwerte nicht mehr verfügbar sind. Da die notwendige Transformation für alle Gründe die gleiche ist, soll sie allerdings erst nach der Erläuterung sämtlicher Gründe für die Nicht-Darstellbarkeit vorgestellt werden.
Bei einer Nicht-Darstellbarkeit genügend vieler *group*-Prädikate ergeben sich die gleichen Symptome, mit den gleichen durchzuführenden Transformationen. Die Nicht-Darstellbarkeit des *group*-Prädikates im Allgemeinen führt ebenfalls dazu, dass die Gruppen größer werden — dann existiert nämlich nur eine, die das gesamte Ergebnis umfasst.
Aus diesem Grund müssen in diesem Fall nicht nur **sämtliche** *group*-Prädikate, sondern auch sämtliche Aggregatfunktionen in den *select*- und *having*-Klauseln entfernt und durch *select*-Klauseln für alle betroffenen Attribute ersetzt werden, während alle entfernten Klauseln in den Q_δ-Teil verschoben werden. An dieser Stelle sei ebenfalls erwähnt, dass hinzugefügte *select*-Prädikate durch beliebige Transformationen bei vorliegender Aggregation in der *Select*-Klausel ebenfalls zu zusätzlichen *group*-Prädikaten des jeweiligen Attributes führen müssen.

3.3.7 Having-Prädikate

Prädikate im *Having*-Teil sind wieder Vergleichsoperatoren wie schon im *Where*-Teil vorgestellt (für die Transformationsregeln siehe 3.3.2) und die verfügbaren Transformationen unterscheiden sich nicht, weshalb auf diese hier nicht erneut eingegangen wird. Allerdings ist zu beachten, dass bei jeglicher

nicht äquivalenter Transformation der Where-Bedingung weitere *select*- und *group*-Prädikate entstehen und damit bestehende Gruppen aufgespaltet werden. Hierdurch werden eventuell vorher zutreffende *Having*-Prädikate falsch gemacht und damit Gruppen aus dem Ergebnis entfernt. Damit dies nicht geschieht, müssen entweder alle Transformationen im *Where*-Teil äquivalent sein oder die Prädikate im *Having*-Teil müssen *sicher* sein.

Definition 3.1. *Das Prädikat P ist* ***sicher****, wenn* ***genau eine*** *der folgenden Bedingungen zutrifft:*

- $P = gt(pluspred(A), \$Value)$ *oder*
- $P = gt(\$Value, minuspred(A))$ *oder*
- $P = not(gt(minuspred(A), \$Value))$ *oder*
- $P = not(gt(\$Value, pluspred(A)))$ *oder*
- $P = and(P_1, P_2) \wedge P_1$ *ist* ***sicher*** $\wedge\, P_2$ *ist* ***sicher*** *oder*
- $P = or(P_1, P_2) \wedge (P_1$ *ist* ***sicher*** $\vee\, P_2$ *ist* ***sicher***$)$ *oder*
- P *enthält keine Aggregationen*

mit

- $minuspred \in \{sum, max, count\}$,
- $pluspred = min$.

Es können auch nicht *sichere* Prädikate unproblematisch sein — die notwendigen Bedingungen dafür würden allerdings an dieser Stelle den Rahmen sprengen. Ist ein *Where*-Prädikat nicht äquivalent transformiert worden und alle *Having*-Prädikate sind sicher, so sind lediglich Kopien der *Having*-Prädikate im Original in Q_δ zu machen. Sind nicht alle *Having*-Prädikate sicher, so müssen die nicht sicheren in den Q_δ-Teil verschoben werden, während von den sicheren Prädikaten Kopien in Q_δ gemacht werden. Zudem müssen bei einer notwendigen Transformation für alle betroffenen Attribute *select*-Prädikate erzeugt werden, um die tatsächliche *Having*-Klausel noch ausführen zu können.

Beispiel

Sei beispielsweise die Anfrage

$$Table('emp'), gt(PLZ, 15000), group(lastname), not(gt(count(*), 20)), select(lastname), select(avg(Age)) \tag{3.15}$$

gegeben, die der SQL-Anfrage

```
SELECT lastname, Avg(Age)
FROM emp
WHERE PLZ>15000
GROUP BY lastname
HAVING Count(*)<=20
```

entspricht. Sei weiterhin $gt(PLZ, 15000)$ nicht darstellbar und vollständig in Q_δ zu verschieben. Es ist zu beachten, dass das *Having*-Prädikat $not(gt(count(*), 20))$ dem Muster $not(gt(minuspred(A), \$Value))$ entspricht und damit *sicher* ist. Die entsprechende transformierte Anfrage ist

$$\begin{aligned} R = Table('emp'), group(lastname), group(PLZ), not(gt(count(*), 20)), \\ select(lastname), select(PLZ), select(avg(Age)), select(count(Age)) \end{aligned} \tag{3.16}$$

oder in SQL-Syntax

```
SELECT lastname, PLZ, Avg(Age), Count(Age)
FROM emp
GROUP BY lastname, PLZ
HAVING Count(Age)<=20
```

die Restanfrage ist

$$gt(R.PLZ, 15000), group(R.lastname), not(gt(count(*), 20)), select(R.lastname), \\ select(Avg(Age)), rename(/(sum(*(R.avg(Age), R.count(Age))), sum(R.count(Age))), Avg(Age)) \tag{3.17}$$

Hierbei entsteht der komplexe Ausdruck im *rename*-Prädikat durch die Zusammenführung der verschiedenen Gruppen bei der *avg*-Aggregatfunktion. Dennoch entsteht das gleiche Ergebnis.

3.3.8 Select-Prädikate

Für die Nicht-Darstellbarkeit einer Projektion gibt es nur zwei Gründe — es ist entweder das projizierte Attribut nicht erlaubt oder aber es sind gar keine Projektionen erlaubt (wie beispielsweise im DBMS *TinyDB*). Die Nicht-Darstellbarkeit einer genügenden Anzahl von Projektionen ist in Standard-DBMS unüblich und darum auf keine Projektionen zurückzuführen. Tatsächlich sind damit beide Möglichkeiten auf die gleiche Transformation zurückzuführen: das Prädikat muss restlos in Q_δ verschoben werden, wenn keines verbleibt wird $select(*)$ eingefügt. Ist das Attribut nicht erlaubt, so kann nicht auf es projiziert werden und es wird aus einem anderen Fragment kommen — eine Projektion ist nicht nur unmöglich, sondern auch unnötig. Ist keine Projektion erlaubt, so werden durch diese Regel sämtliche *select*-Prädikate entfernt und durch $select(*)$ ersetzt; dies führt dazu, dass sämtliche Attribute exportiert werden, durch die *select*-Prädikate in Q_δ werden in der Nachbearbeitung nicht notwendige Attribute wegprojiziert. Zusätzlich müssen nach Entfernung aller *select*-Prädikate (vorher ist das tatsächlich nicht nötig) sämtliche Gruppierungen — die sich in *having*- und *group*-Klauseln ausdrücken — entfernt werden.

3.4 Vorgehensweise

Nun existiert sowohl eine Sprache, um die Anfragen für die Transformation darzustellen, als auch Transformationsregeln für Prädikate dieser Sprache, die eine Transformation in ausführbare Anfragen möglich machen. Es fehlt nun also eine algorithmische Vorgehensweise, die die Vorteile konjunktiver Darstellung und die Möglichkeit der Transformation der einzelnen Literale miteinander verbindet. Dies soll in diesem Abschnitt geschehen und damit die Grundidee für die Implementation liefern.
Zunächst gibt es einige Informationen, die vor dem eigentlichen Ablauf des Algorithmus vorliegen müssen: Zunächst die möglichen Operatoren auf den unterschiedlichen Schichten (auf einem Computer müssen lediglich Informationen über die Rechner der jeweils nächstniedrigeren Schicht vorliegen), die durch Grammatiken angegeben werden. Diese Grammatiken sind im Normalfall für viele der Computer unterschiedlich und sollten darum lokal gespeichert werden, beispielsweise in einer XML-Datei. Zusätzlich sind Informationen über die zugelassenen Prädikate vonnöten — auch wenn hier bereits viele vorgestellt wurden, können sie nicht den ganzen SQL-Standard abdecken. Weitere Prädikate sind also definierbar, die dann aber auch bei der Konvertierung in RQDL4SQL bekannt sein müssen (zusammen mit den Konvertierungsregeln). Zudem sind für alle Prädikate im Falle der Nicht-Darstellbarkeit Transformationsregeln — zumindest die Θ_0-Transformation — anzugeben. Diese Informationen allerdings sind global, sie sollten an einer zentralen Stelle gespeichert werden, sodass sie von überall abrufbar sind. Eine Speicherung in einer zentralen Datenbankrelation ist denkbar, wobei zu beachten ist, dass sechs unterschiedliche Arten von Transformationen zu unterscheiden sind: Die Standard-Θ-Transformation und $\Theta\_$-Transformationen für Prädikate, Θ, Θ_+ und $\Theta\_$-Transformationen für Werte sowie die Folgetransformationen Θ^*.
Im Rahmen der Verarbeitung muss die ankommende Anfrage dann zunächst in RQDL4SQL transformiert werden. Hierfür sollte die Anfrage zunächst geparst und dann konvertiert werden, da die RQDL4SQL

prinzipiell nur eine auf Transformationen spezialisierte Darstellung der Pre-Order-Notation des Operatorbaumes der Anfrage ist. Sämtliche Bezeichnungen, wie beispielsweise Spaltenbezeichnungen und Umbenennungen können hierbei bestehen bleiben — da die Transformation keine Bezeichnungen erzeugt, wird so aus einer validen Anfrage (im höheren SQL-Standard) wieder eine valide Anfrage im geringeren Standard. Es ist zu beachten, dass der Operatorbaum im Original noch benötigt wird und darum während der Konvertierung nicht verändert werden darf.
Daraufhin ist zu ermitteln, ob die Anfrage transformiert werden muss und wenn ja, welche Prädikate betroffen sind. Hierfür wird die Anfrage in die Grammatik eingepasst, wie im *Capabilities-based Query Rewriting*-Ansatz. Hierbei ist zu beachten, dass die maximalen CSQs hier nicht notwendigerweise Teilanfragen ohne Projektionen sind. Tatsächlich sind maximale CSQs genau diejenigen Teilanfragen, die **darstellbar sind** und eine **maximale Anzahl an Prädikaten** besitzen. Aufgrund von im Eingang des Absatzes Relational Query Description Language for SQL erwähnten Besonderheiten wird es bei gefundenen Plänen immer entweder genau einen maximalen CSQ oder aber mehrere CSQs, die einen Plan bilden können, geben (sofern es sich um einen korrekten SQL-Standard handelt). Hieraus folgt, dass es immer genau einen Plan gibt, der aus allen gefundenen maximalen CSQs besteht. Alle Prädikate, die nicht Teil dieses Planes sind, müssen dann also transformiert werden. Hierbei sind in den Transformationsregeln zwar immer die Zielfähigkeiten mit aufgelistet, dennoch kann nicht notwendigerweise sichergestellt werden, dass nach einmaliger Anwendung der Transformationsregeln auf die restlichen Prädikate tatsächlich die ganze Anfrage darstellbar ist. Aus diesem Grunde muss diese Prozedur solange wiederholt werden, bis die Menge der Prädikate, die nicht im Plan sind, leer ist.
In Folge wird die Anfrage optimiert (beispielsweise sind doppelte Negationen oder Wiederholungen von gleichen Prädikaten nach der Transformation nicht ausgeschlossen) und dann wieder zurück in SQL transformiert. Hierfür müssen die Prädikate der Anfrage, die transformiert werden mussten, eine Referenz auf den darstellenden Teilbaum enthalten und an dessen Stelle eingesetzt werden. Im Zuge der Rücktransformation muss hierbei besonders auf die Rekursionstiefe geachtet werden, da RQDL4SQL keine vollständige Anfragesprache ist und diese nicht darstellen kann. Aus diesem Grund muss hier dafür gesorgt werden, dass die Rücktransformationen sämtlicher durch Transformation entstandenen Prädikate auf der richtigen Anfrageebene eingefügt werden.
Daraufhin kann die Anfrage ausgeführt werden. Auf das Ergebnis (bzw. die Ergebnisse bei mehreren angesprochenen Computern) ist dann die Vereinigung der Q_{δ_Θ}-Mengen auszuführen (hierfür müssen natürlich auch Funktionen für die Ausführung der jeweiligen Prädikate zur Verfügung stehen) und das Ergebnis daraufhin zurückzugeben.

3.5 Bemerkungen und Zusammenfassung

In diesem Abschnitt sollen einige abschließende Bemerkungen sowie eine Zusammenfassung zum vorgestellten Konzept gegeben werden. Zunächst ist zu betrachten, dass die gesamte Vorgehensweise grundlegend von der Grammatik und den Transformationsregeln abhängt, womit auch die Qualität des Ergebnisses maßgeblich von diesen Größen abhängig ist. Es kann aus diesem Grund auch als bloße Kombination der Ansätze des *Answering Queries using Operators* und des *Capabilities-based Query Rewriting* betrachtet werden, wobei es grundlegend die algorithmische Handhabbarkeit und geeignete Transformationsregeln für die gegebenen Basisprädikate definiert. Er kann lediglich für die Anfragen verwendet werden, die in RQDL4SQL darstellbar sind und vermag nur solche Operatoren zu verwenden, die in der Grammatik der verwendeten Anfragesprache in RQDL4SQL definiert sind. Hierbei ist jedoch zu beachten, dass diese beiden Aspekte im Grunde beliebig erweitert werden können.
Es ist zu beachten, dass die Transformationsregeln **lokal** sind und damit Wechselwirkungen zwischen Prädikaten und auch Wechselwirkungen zwischen verschiedenen Transformationen nicht modelliert werden können. Zudem können sowohl die Qualität als auch die Performance von der ausgewählten Transformationsreihenfolge abhängen. Die vorgestellten Transformationsregeln erfüllen die Bedingungen 3.3 sowie die Punkte (1) bis (3) von 3.1 — dies folgt aus der Konjunktivität der Sprache und der Gestalt der

Transformationsregeln $\Theta(P)$ und $\Theta_0(P)$. Die *Minimalität* der gefundenen Transformation kann hingegen nicht garantiert werden, da sie von den verwendeten Transformationsregeln abhängt. Bei den gegebenen Transformationsregeln ist durch die Reihenfolge bei gleichwertigen Transformationen eine Priorisierung gegeben, wobei die zuerst genannten Transformationen ein kleineres Ergebnis liefern. Allerdings folgt durch die Konjunktivität aus der Minimalität aller Prädikattransformationen (sollte sie bestehen) auch die Minimalität der transformierten Anfrage.
Im folgenden Kapitel sollen nun einige Gedankengänge zur Implementation des vorgestellten Konzeptes als auch die Implementation einiger Funktionen beispielhaft dargestellt werden. Die gesamte Codebasis kann aufgrund ihrer Größe und der Einbindung in das Projekt PArADISE hier nicht dargestellt werden, sie ist aber im Git des Projektes oder auf der beiliegenden CD vorhanden.

Kapitel 4

Implementierung

In diesem Kapitel sollen einige grundlegende Gedankengänge bei der Implementierung des vorgestellten Konzeptes dargelegt werden, sowie auch einige besondere Codeabschnitte. Hierzu zählen beispielsweise Datenstrukturen für die benötigten Objekte als auch Funktionen für die Transformationsschrittabfolge. Die Implementierung geschieht im Rahmen des Projektes PArADISE, dessen Codebasis insbesondere eine JDBC-ODBC-Bridge für den Datenbankzugriff implementiert und bereitstellt, wo auch die nötigen Pre- und Postprocessing-Schritte für die Anfragetransformation durchgeführt werden können. Die verwendete Programmiersprache ist Java. Die hier gezeigten Programmabschnitte werden sich grundsätzlich an der zeitlichen Abfolge der Programmschritte im Zielprogramm orientieren, die Reihenfolge erhält also eine zeitliche Basis. Ebenso an dieser Stelle zu erwähnen ist, dass das vorgestellte Programm lediglich einen Prototypen darstellt und noch nicht alle im Konzept vorgestellten möglichen Anfragen und Transformationen unterstützt.

4.1 Parsing und Konvertierung

Zunächst muss die ankommende Anfrage geparst werden, um dann in RQDL4SQL konvertiert werden zu können. Der verwendete Parser in PArADISE ist der jsqlparser 0.9.3, der ein *Statement* zurückliefert. Der *PlainSelect*-Anteil dieses *Statements* enthält alle Informationen über verwendete Tabellen, durchgeführte Projektionen und so weiter. So können beispielsweise mithilfe der bereitgestellten Funktionen *getFromItem*() und *getJoins*() bestimmte Objekte analysiert werden, die Aufschluss über die verwendeten Basisrelationen geben, während *getWhere* den gesamten Ausdruck in der $WHERE$-Klausel zurückgibt.

4.1.1 Datenstrukturen für RQDL4SQL-Prädikate

Vor der eigentlichen Konvertierung müssen natürlich Datenstrukturen geschaffen werden, die die jeweiligen Prädikate der RQDL4SQL-Anfrage darstellen können. Sämtliche Prädikate sind vom Typ *PredicateDescription* (siehe 4.1),

```
public abstract class PredicateDescription {
  public PredicateType type;
}
```

Code 4.1: Der Typ PredicateDescription

der allerdings abstrakt ist, da Objekte diesen Typs im Grunde einer von zwei Gruppen angehören: Sie können einen Wert darstellen, wie beispielsweise die *sum*-Funktion oder auch nur einen Attributwert

(*ValuePredicate*), oder einen komplexen Anfrageteil wie beispielsweise das *select*-Prädikat oder die einzelnen *Where*-Prädikate, die keinen annotierten Wert haben und wenn überhaupt einen Wahrheitswert zurückliefern. Der Typ *PredicateType*, von dem das Attribut *type* eines jeden solchen Objektes ist, ist lediglich eine Auflistung der möglichen Prädikattypen:

```
package queryTransformation.types;

public enum PredicateType {
  Table, Eq, Gt, Bw, In, Or, And, Not, Group, Rename, Select, Value
}
```

Hierbei sind mit value sämtliche Wertprädikate gemeint — die Zuweisung eines zusätzlichen Prädikattypen macht die Implementation auf rekursive Art einfacher. In ähnlicher Art und Weise sind auch die Wert-Prädikate unterteilt. Dann gibt es für jeden dieser Prädikattypen einen anderen Typen, da sich der Typ und die Anzahl der Parameter unterscheiden. Als Beispiel sei hier das Objekt für ein *Table*-Prädikat gegeben.

```
package queryTransformation.types;

import java.util.ArrayList;

public class TablePredicate extends PredicateDescription {

  public TablePredicate(String First) {
    type = PredicateType.Table;
    first = First;
  }

  public String first;

  @Override
  public String ToString() {
    return first;
  }

  @Override
  public ArrayList<ValuePredicate> FindAttributes(String TableName) {
    ArrayList<ValuePredicate> res = new ArrayList<ValuePredicate>();
    return res;
  }
}
```

Das Prädikat *Table('emp')* wird dann durch eine *PredicateDescription*, genauer: ein *TablePredicate*, dargestellt. Dieses *TablePredicate* hat dann den Typ *PredicateType.Table* und den *first*-Wert (*first* für erster Parameter des Prädikates) von „emp“.

4.1.2 Konvertierung

Die Konvertierung vom *Statement* des Parsers zu den einzelnen Prädikaten entsteht im Grunde lediglich durch eine Auslesung der verschiedenen Teile. So entstehen die *Table*-Prädikate beispielsweise durch eine Analyse des *From*-Teils, kombiniert mit den einzelnen vorkommenden Verbunden (*Joins*). Von besonde-

rer Bedeutung für die konjunktive Darstellung der Anfragen ist der *Where*-Teil, weshalb die relevanten Anteile der Konvertierung im Anhang B.1 zu finden sind. Die Funktion *TranslateExpression* übernimmt die an vielen Stellen der Konvertierung wichtige Fallunterscheidung, um die korrekten Prädikattypen für den jeweiligen Ausdruck zu erzeugen. Die Funktion *FlattenPredicate* ermöglicht es, den *Where*- (und später auch den *Having*-) Teil mithilfe von mehreren Prädikaten darzustellen. So ist eine *WHERE*-Bedingung aus mehreren Teilzielen für den Parser lediglich ein Und-Ausdruck aus den verschiedenen Teilzielen. Ein Und-Ausdruck ist auch in RQDL4SQL möglich mithilfe des *and*-Prädikates, das allerdings nur zwei Parameter erhält anstatt der beliebigen Anzahl an Teilzielen. Dies bedeutet, dass ein Und-Ausdruck nicht immer in mehrere Prädikate übersetzt werden darf — ist er selbst Parameter eines anderen Prädikates, muss ein *and*-Prädikat herauskommen. Anstatt die erste Ebene gesondert zu betrachten und damit dort auf die Möglichkeit der Rekursion zu verzichten, entfernt die *FlattenPredicate*-Funktion die oberen *and*-Prädikate des normal konvertierten Ausdrucksbaumes und erzeugt daraus die Vereinigung der linken und rechten Prädikatmengen.
Schon mit diesem recht simplen Konverter kann die Anfrage 3.3.1 korrekt transformiert werden. Es entstehen sechs Prädikate; zwei Table-, ein eq-, ein gt- und zwei select-Prädikate mit den korrekten Parametern, wie schon in 3.7 angegeben. Unten ist das UML-Objektdiagramm der entstehenden Prädikatobjekte gegeben.

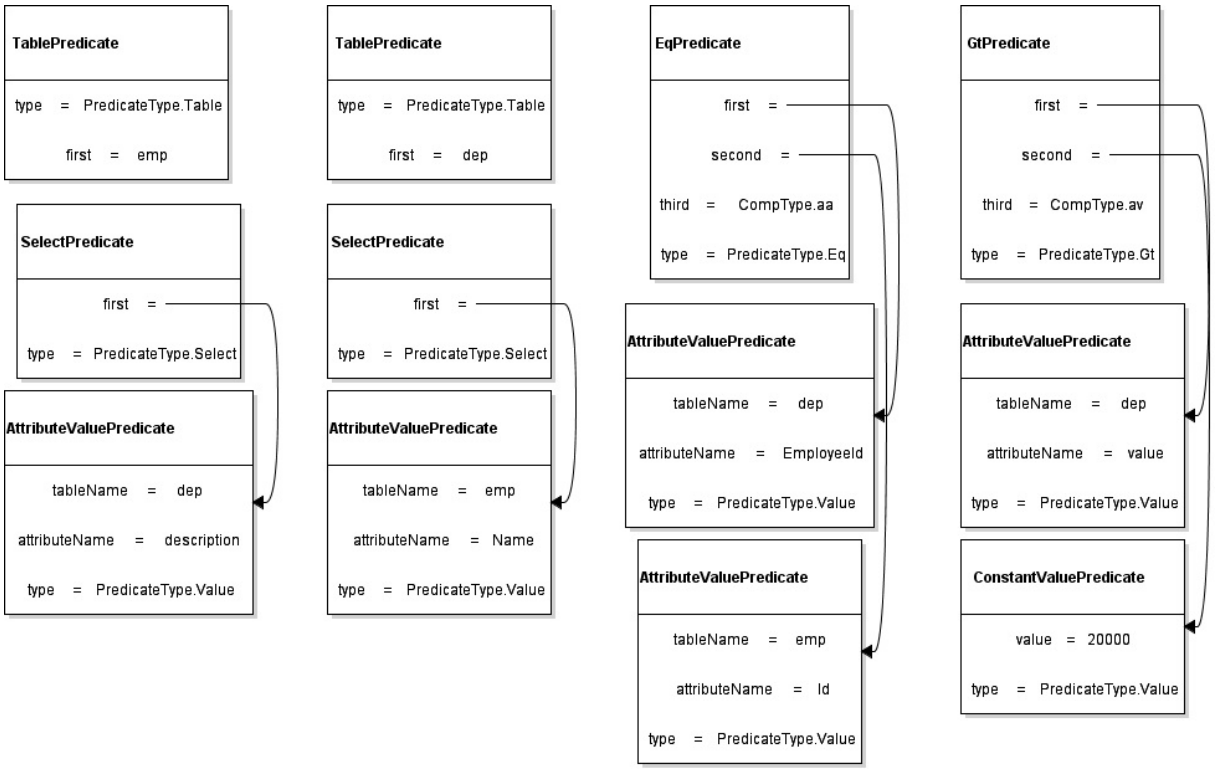


Abbildung 4.1: UML-Objektdiagramm der RQDL4SQL-Prädikatobjekte

4.2 Transformation

Im Kernteil der Implementation, die sich mit der Transformation befasst, werden wieder einige neue Datenstrukturen benötigt. So sind neben Objekten, die Transformationen und Anfrageumschreibungen beschreiben, auch solche vonnöten, die Prädikatpattern und Grammatiken beschreiben.

4.2.1 Datenstrukturen

Die erste Datenstruktur, die an dieser Stelle beschrieben werden soll, ist ein Pattern für Prädikate, das für die Anwendbarkeit einer Transformationsregel unerlässlich ist (P), im Programmcode als *PredicateTemplate* bezeichnet. Die Implementierung eines solchen Typs ist in B.2 gegeben.
Der Typ *PredicateBoolTemplate* implementiert die abstrakte Klasse *PredicateTemplate* und zeigt die Arbeitsweise gut. Die Funktion $FulfillsTemplate$ muss von jedem Template implementiert werden und gibt für das Prädikat im Parameter an, ob es von diesem Pattern unterstützt wird. Trifft ein Pattern auf das Prädikat zu, so gibt die Funktion $FulfillsTemplate$ *true* zurück.
EnabledTypes ist eine Liste von Prädikattypen, die von dem jeweiligen Template unterstützt werden. Auch wenn nur ein Typ je Pattern typisch ist, so erlaubt die Angabe einer Liste von Typen doch die leichtere Angabe von Pattern, die auf mehrere Prädikattypen zutreffen können, wie beispielsweise das Prädikattyppattern *pred* in 3.3.1. Dies bedeutet auch, dass die zugreifenden Methoden unterschiedliche Anzahlen von Parametern unterstützen müssen, insbesondere auch $Fulfillspredicate$.
EnabledFirsts und *EnabledSeconds* geben Templates für die ersten beziehungsweise zweiten Parameter des Prädikates an. *EnabledComparisonTypes* sind die unterstützen Vergleichstypen (Attribut-Attribut — aa oder Attribut-Wert — av) dieses Prädikates, wenn anwendbar — eine Angabe bei einem Pattern für *select*- oder *Table*-Prädikate ist unsinnig. Bei *aa* sollte auch *av* angegeben werden — auch wenn das Eine das Andere bedingt, so ist von der implementatorischen Warte eine solche Angabe sinnvoll. *PredicateTemplate*s sind auch recht gut für die Angabe von Grammatiken geeignet, weshalb im Prototypen die Grammatik mithilfe einer Liste solcher Pattern-Beschreibungen gegeben wird.
Ein weiterer interessanter Typ ist die Beschreibung der Transformation selbst. Auch in der Implementation wird er durch eine Menge von nötigen Operatoren für das nicht transformierte Teilziel, eine Menge von nötigen Operatoren für das transformierte Teilziel, eine Beschreibung des Prädikates und eine Beschreibung des Transformationsergebnisses angegeben. Hierbei ist das Transformationsergebnis als Funktion (von Prädikat und Anfrage auf neue Anfrage) implementiert, da die Generation einer Anweisungsfolge anhand von weiteren Objekten zwar möglich, aber unnötig aufwendig ist. Die Menge der Folgetransformation ist dann Teil dieser Funktion.

```
package queryTransformation.types;

import java.util.ArrayList;
import java.util.function.Function;

public class QueryTransformation {
  public QueryTransformation(ArrayList<PredicateTemplate> operators0, ArrayList<
  PredicateTemplate> operators1,
      PredicateTemplate predicate, Function<PredicateForTransformation, QueryRewriting
      > execute) {
    Operators0 = operators0;
    Operators1 = operators1;
    Predicate = predicate;
    Execute = execute;
  }

  public ArrayList<PredicateTemplate> Operators0;
  public ArrayList<PredicateTemplate> Operators1;
  public PredicateTemplate Predicate;
  public Function<PredicateForTransformation, QueryRewriting> Execute;

  /**
```

```
   * executes the given transformation function of this query transformation on the
   given predicate
   *
   * @param pred the predicate with the current query rewriting
   * @return the new rewriting after executing this transformation
   */
  public QueryRewriting Transform(PredicateForTransformation pred) {
    return Execute.apply(pred);
  }
}
```

4.2.2 Vorgehensweise Transformation

Die Transformation selbst bekommt nun also eine Menge von Prädikaten (*PredicateDescriptions*) und muss diese in zwei Mengen von Prädikaten transformieren — die transformierte / umgeschriebene Anfrage sowie die Restanfrage Q_δ. Dieses Konstrukt wird im Typen *QueryRewriting* gespeichert, das lediglich aus den beiden Prädikatlisten besteht. Initialisiert wird es mit der gesamten Prädikatmenge im Anteil der transformierten Anfrage, während die Restanfrage mit einer leeren Liste initialisiert wird (dies geschieht im Konstruktor des *QueryRewriting*-Objekts).

Vorbedingungen

Bevor die eigentliche Transformation beginnen kann, werden noch einige zusätzliche Objekte benötigt: Zunächst müssen die Fähigkeiten der unteren Schicht bereitstehen, die auch die durchführbaren Transformationen determinieren. Wie im entsprechenden Abschnitt bereits angedeutet, kann eine solche Grammatik recht gut mithilfe einer Liste von *PredicateTemplate*s dargestellt werden. Der derzeitige Prototyp bezieht diese Liste aus einer statischen Klasse — dies muss später natürlich angepasst werden, da die Fähigkeiten der unterliegenden Schicht vom ausführenden Computer und nicht vom Programm abhängt, für eine Demonstration reicht allerdings die statische Liste. Für die bereits gegebene Beispielanfrage 3.3.1 wird ein Datenbanksystem, das nur die Tabelle *emp* unterstützt, aber sonst keine Einschränkungen vorweist, zugrunde gelegt. Die entsprechende Templateliste ist im Anhang B.3 zu finden.
Das zweite benötigte Objekt sind die möglichen Transformationen. Auch diese Liste von *QueryTransformation*-Objekten ist im derzeitigen Prototyp in einer statischen Klasse, doch kann dies so bleiben, sollte keine Möglichkeit gefunden werden, Transformationen automatisch zu generieren, da Transformationen generell nicht vom ausführenden Rechner abhängen, sondern lediglich vom Prädikat. Die entsprechende Klasse mit der nötigen Beispieltransformation ist in B.4 gegeben. Hierbei gibt die *includesTable*-Funktion eine Zahl zurück, die angibt, ob und wenn ja, in welchem Parameter des Prädikates eine andere als die gegebene Tabelle auftaucht. Die Funktion *FindAttributes* findet zu selektierende, da zu der jeweiligen Tabelle gehörende, Teilausdrücke und gibt sie zurück.

Transformation

Die eigentliche Transformation ist nun recht simpel. Zunächst wird die initiale Anfragetransformation erzeugt (durch den Konstruktor von *QueryRewriting*). Dann werden die folgenden Schritte durchgeführt:

1. Finden der nicht unterstützten Prädikate — dies tut die Funktion *GetNotSupportedPredicates*
2. Wenn die Liste der nicht unterstützten Prädikate leer ist: Ende
3. Für jedes Prädikat eine passende Transformation finden und diese ausführen

4. Zurück zu 1.

Hierbei ist zu beachten, wann eine Transformation ausführbar ist. Im Grunde müssen dafür alle der folgenden Bedingungen erfüllt sein:

- Das Prädikat erfüllt das Template P der Transformation
- Das Prädikat wird nicht durch die Grammatik, die *AvailablePredicates*, unterstützt
- Die Operatoren O_0 der Transformation sind in der Grammatik nicht unterstützt
- Die Operatoren O_1 der Transformation sind in der Grammatik unterstützt

Dann kann im Grunde die erste gefundene Transformation angewandt werden, wenn die Transformationen in der im Kapitel Konzept angegebenen Reihenfolge überprüft werden (diese impliziert die Priorisierung). Andernfalls ist diejenige Transformation auszuführen, die ausführbar ist (4.2.2) und die höchste Priorität hat.
Die Implementierung reflektiert die auszuführenden Schritte recht genau, im Folgenden ist die Hauptfunktion dargestellt. Die ganze Klasse mit den Implementationen der verwendeten Funktionen findet sich im Anhang B.5.

```
public static void transform(String OriginalQuery) throws Exception {
    AvailableTransformations.AddTransformations();
    ArrayList<PredicateTemplate> AllowedPredicates = AvailablePredicates.Predicates;
    QueryDescription description = RQDL4SQLConverter.ConvertToRQDL4SQL(OriginalQuery);
    QueryRewriting rewriting = new QueryRewriting(description);

    while (true) {
      ArrayList<PredicateDescription> predicates = GetNotSupportedPredicates(rewriting
      .QueryRewriting,
          AllowedPredicates);

      if (predicates.size() == 0)
        break;

      for (PredicateDescription predicate : predicates) {
        for (QueryTransformation trans : AvailableTransformations.Transformations) {
          if (isApplicable(predicate, trans, AllowedPredicates)) {
            PredicateForTransformation pred = new PredicateForTransformation(
            rewriting, predicate);
            rewriting = trans.Transform(pred);
            // only one transformation per predicate
            break;
          }
        }
      }
    }

    setQueryRewriting(rewriting);
  }
```

Mithilfe der gegebenen Funktion kann die Beispielanfrage 3.3.1 bereits korrekt transformiert werden. Es entstehen die Repräsentanten der Prädikate $Table(emp), select(emp.Name), select(emp.Id)$ im Anteil

der transformierten Anfrage sowie die Repräsentanten der Prädikate $eq(dep.EmployeeId, emp.Id), gt(dep.Value, 2000), select(dep.Description), select(emp.Name)$ im Restanfrageanteil.

4.3 Rückkonvertierung

Nun muss natürlich die entstandene transformierte Anfrage wieder in eine valide SQL-Anfrage zurück konvertiert werden. Dies ist tatsächlich komplexer als die Konvertierung von SQL in RQDL4SQL, da, wie an anderen Stellen schon erwähnt, RQDL4SQL keine vollständige Sprache ist und damit nicht in der Lage ist, SQL-Anfragen beliebiger Komplexität vollständig darzustellen. Besonders bei der Verwendung von Unteranfragen kann aus der Reihenfolge der Prädikate nicht notwendigerweise die Unteranfrage, zu der das Prädikat gehört, erkannt werden, da einige Partitionen vollständig optional sind. Aus diesem Grund wird für die Rückkonvertierung solcher Anfragen mit Unteranfragen nicht nur die Prädikatmenge der transformierten Anfrage benötigt, sondern auch der Operatorbaum der Originalanfrage und Abbildungen von Prädikaten auf ersetzende Prädikatmengen. Dies allerdings ist im derzeitigen Prototypen noch nicht implementiert, sondern es werden lediglich die Prädikate in eine ausführbare SQL-Anfrage konvertiert. Auch die Differenzierung zwischen Prädikaten der *Where*- und der *Having*-Partition wird noch nicht vorgenommen (auch weil *Having*-Prädikate noch nicht bei der Konvertierung entstehen können).

```
public static String ConvertToQuery(ArrayList<PredicateDescription> predicates) {
    List<SelectPredicate> selects = GetSelectPart(predicates);
    List<TablePredicate> froms = GetFromPart(predicates);
    List<PredicateDescription> wheres = GetWherePart(predicates);

    StringBuilder query = new StringBuilder("SELECT ");
    Boolean first = true;

    for (SelectPredicate select : selects) {
      if (!first)
        query.append(", ");
      else
        first = false;
      query.append(select.ToString());
    }

    first = true;
    query.append(" FROM ");

    for (TablePredicate table : froms) {
      if (!first)
        query.append(", ");
      else
        first = false;
      query.append(table.ToString());
    }

    if (wheres.size() != 0) {
      first = true;
      query.append(" WHERE ");
```

```
      for (PredicateDescription where : wheres) {
        if (!first)
          query.append(", ");
        else
          first = false;
        query.append(where.ToString());
      }
    }

    return query.toString();
  }
```

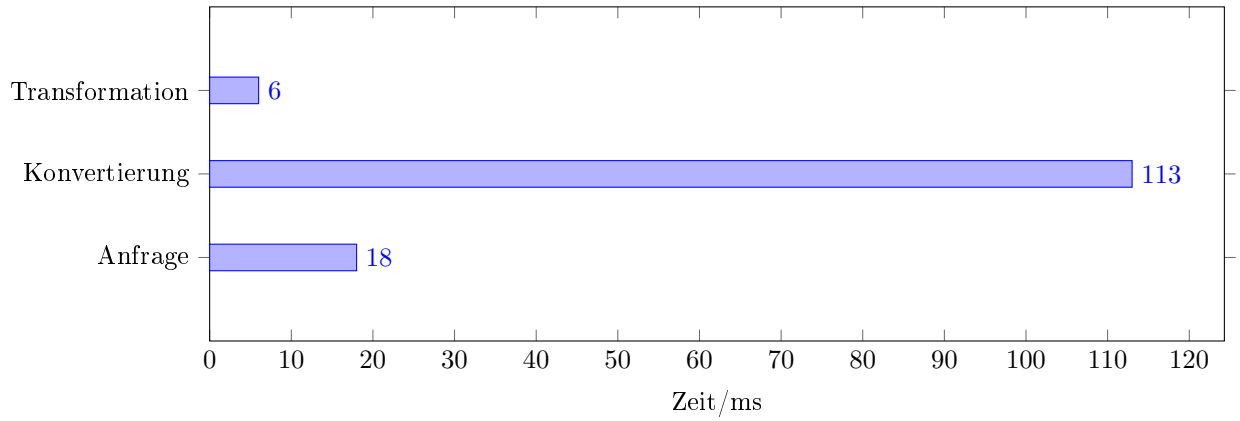
Trotz dieser Unzulänglichkeiten kann die Prädikatmenge der Beispielanfrage korrekt zurück konvertiert werden. So entsteht aus der Prädikatmenge $Table(emp), select(emp.Name), select(emp.Id)$ die korrekte Anfrage *SELECT emp.Name, emp.Id FROM emp.*

4.4 Abschließende Bemerkungen

An dieser Stelle sollen einige abschließende Bemerkungen bezüglich der Implementation Platz finden. Zunächst handelt es sich bei der Implementation in der jetzigen Form, wie bereits häufiger erwähnt, lediglich um einen Prototypen, weshalb einige Unzulänglichkeiten auftreten, die im Folgenden skizziert werden sollen. Beginnend ist hier das Parsing zu erwähnen: im Rahmen dieser Arbeit wurde nicht untersucht, ob und wenn ja, wie geeignet dargestellt der verwendete Parser in der Lage ist, komplexe Anfragen zu parsen. Die Konvertierung und Rückkonvertierung von Anfragen in RQDL4SQL-Prädikate und umgekehrt ist derzeit nur für einfache Anfragen mit einfacher Selektion mit Gleichheits- und Größer als- Vergleichen, Verbünden und Projektion implementiert. Die Fähigkeiten der einzelnen Computer der unterliegenden Schichten sind derzeit nur als abstrakte Klasse und damit nur für eine Schicht nutzbar implementiert. Die Darstellung dieser Fähigkeiten sollte in der Weiterentwicklung in eine Datei auf dem ausführenden Rechner, beispielsweise eine XML-Datei, ausgelagert werden, wobei darauf zu achten ist, dass unterschiedliche Rechner der nächstniedrigeren Ebene unter Umständen unterschiedliche Fähigkeiten besitzen können, sodass mehrere solcher Darstellungen vonnöten sein können. Die Nutzbarkeit für die Transformation schränkt dies allerdings nicht ein: sind mehrere Rechner mit unterschiedlicher Leistungsfähigkeit auf der unterliegenden Ebene, so kann für jeden einzelnen die Anfrage transformiert werden. Die aus diesen transformierten Anfragen entstehenden Ergebnismengen können dann vereinigt oder anderweitig kombiniert werden. Bei der Transformation sind, wie bereits erwähnt, die Überprüfungen der Bedingungen (3) und (4) von 4.2.2 nicht implementiert — für diese ist eine Funktionalität, die die Teilmengenbeziehung oder das anderweitige *enthalten sein* von Prädikatpattern feststellt, vonnöten, die noch nicht implementiert wurde. Auch wurden für die verwendeten Methoden auf den *PredicateDescription*- und *PredicateTemplate*-Objekten teilweise nur Methodenstubs erstellt. Die Unzulänglichkeiten bei der Rückkonvertierung wurden im letzten Abschnitt erläutert und sollen aus diesem Grunde hier nicht noch einmal auftauchen. Ebenfalls nicht implementiert wurde die Anwendung der Restanfrage — diese Anfrage wird nicht auf eine Datenbank angewandt und kann aus diesem Grunde nicht einfach in eine SQL-Anfrage konvertiert und ausgeführt werden. Eher wahrscheinlich ist es, dass eine Funktion implementiert werden muss, die für jede Partition der Restanfrage die Ergebnismenge abhängig der enthaltenen Prädikate weiter transformiert. Doch auch mit diesen recht gravierenden Einschränkungen ist die Implementation in der Lage, Anfragen korrekt zu transformieren. Die Architektur ist erweiterbar angelegt und darum ohne großen Aufwand in der Lage, auch für neue Prädikate angemessen erweitert zu werden, was für komplexe Anfragen nur angemessen sein kann.
Zuletzt soll die Performance angesprochen werden. So wird die Laufzeit für komplexe Anfragen und mehr Transformationen entsprechend wachsen, doch gibt auch das verwendete Beispiel eine grobe Orientie-

rung. Besteht die Anfrage aus n Prädikaten und sind m Transformationsregeln zu überprüfen, so sind die Komplexitäten für die Konvertierung und Rückkonvertierung $O(n)$ und für die Transformation $O(n^3m)$ (worst case). Im folgenden Diagramm sind einige Messwerte für die Laufzeit gegeben. Diese Laufzeiten wurden berechnet mithilfe der $System.currentTimeMillis()$-Funktion, die betrachtete Anfrage ist die Beispielanfrage 3.3.1 mit nur einer Transformationsregel, die an der selben Stelle definiert wurde. Die Testdatenbank ist eine lokale *localdb*-Instanz bestehend aus einer Tabelle und 200 Tupeln. Die erhobenen Zeiten sind über 20 Programmläufe gemittelt und beschreiben die folgenden Zeitspannen:

- Die benötigte Zeit für das Ausführen einer Anfrage (diese ist bei Original- und transformierter Anfrage identisch) (Anfrage)
- Die benötigte Zeit für das Parsen der Anfrage und die Konvertierung in RQDL4SQL (Konvertierung)
- Die benötigte Zeit für die Transformation (Transformation)



Kapitel 5

Zusammenfassung und Ausblick

In diesem Kapitel soll zunächst dargestellt werden, welche der eingangs gestellten Ziele zu welchem Grad erreicht werden konnten. Darauf folgt ein Ausblick, der Inspirationen für zukünftige Arbeiten in diesem Bereich zu geben versucht.
Viele der formalen Anforderungen an die Zielanfrage wurden erst im Kapitel 3 definiert, doch sollen hier auch einige der Punkte in der Einleitung diskutiert werden. So war zunächst gefordert, dass die durch das Rewriting entstehende Ergebnismenge eine Obermenge, wenn möglich auch die gleiche Menge, der bei der Originalanfrage entstehenden Ergebnismenge ist. Diese Eigenschaft folgt aus der Konjunktivität der als Grundlage für die Transformation gewählten Sprache RQDL4SQL. So führt das (transformationsgerechte) Entfernen eines Prädikates dazu, dass weniger Forderungen an Basis- oder Ergebnistupel gestellt werden und somit mehr Tupel im Ergebnis auftauchen. Es gibt allerdings zwei Prädikate, bei denen diesbezüglich weitere Erläuterungen angebracht sind: das *select*- und das *group*-Prädikat. Diese beiden Prädikate wirken atomar informationserhöhend, das Entfernen würde sich dementsprechend informationsvermindernd auswirken, da ein fehlendes *select*-Prädikat die entsprechende Spalte des Attributes wegprojiziert, während ein fehlendes *group*-Prädikat ggf. zur Aggregation des entsprechenden Attributes führt. Nur die korrekte Ausführung der gegebenen Transformationsregeln verhindert hier eine Informationsabnahme: ein *select*-Prädikat wird nur unter einer von zwei Bedingungen transformiert. So kann das entsprechende Attribut nicht im Fragment vorkommen, oder aber es sind keine Projektionen in der Sprache zugelassen. Ist das Attribut nicht Teil des Fragmentes, so kommt es durch das Entfernen des Prädikates **nicht** zum Informationsverlust, da das Fragment dieses Attribut in keinem Fall exportieren kann — die zurückgegebene Tupelmenge ist identisch. Sind keine Projektionen in der Sprache zugelassen, so werden zwingend sämtliche *select*-Prädikate entfernt und durch ein *select*$(*)$-Prädikat ersetzt. Auch hier kommt es zu keinem Informationsverlust, da $SELECT *$ sämtliche Attribute zurückliefert und entgegen der Originalanfrage keine Attribute wegprojiziert werden — so ist hier tatsächlich eine Informationserhöhung zu konstatieren. Beim *group*-Prädikat sind die Gründe für Nicht-Darstellbarkeit identisch und die Folgen ähnlich: ist das entsprechende Attribut nicht unterstützt, könnte es keinesfalls exportiert werden und die Ergebnismenge ist identisch; sind keine *group*-Prädikate unterstützt, so werden notwendigerweise sämtliche *group*-Prädikate entfernt und durch entsprechende *select*-Prädikate ersetzt. Dies führt dazu, dass die Gruppierungsattribute auch in der transformierten Anfrage exportiert werden (durch die entsprechenden *select*-Prädikate), es werden lediglich die restlichen Attribute nicht mehr aggregiert, sondern auch sie werden original exportiert — auch hier ist Informationserhöhung die Folge. Die Korrektheit bei möglichen Unteranfragen, die nicht im Fokus dieser Arbeit stand, liegt auf der Hand — wenn die temporäre Ergebnistabelle der Unteranfrage mehr Tupel enthält, kann auch das Ergebnis der Gesamtanfrage mindestens genau die Tupel der Originalanfrage zurückliefern. Die Minimalität dieser Obermenge kann nicht bewiesen werden — sie liegt allerdings vor, wenn alle Transformationsregeln minimale Obermengen liefern und korrekt konfiguriert sind.
Ebenfalls wurde gefordert, dass nur Operatoren aus der Operatorenmenge der unterliegenden Ebene

verwendet werden. Dies hängt im Grunde von der richtigen Definition der jeweiligen Transformationsregeln ab — so wird eine Transformationsregel nur dann ausgeführt, wenn das unterliegende System die Operatoren aus O_1 der Transformation unterstützt. So werden die Prädikate in $r(P)$ bei korrekter Konfiguration ausführbar sein. Zwar können entstehende Teilziele der Resttransformation (oder ggf. auch in $r(P)$ in einigen seltenen Fällen) die Grammatik verletzen, doch auch diese werden dann transformiert, sodass sie nur die erlaubten Operatoren verwenden. Bei fehlerhafter Konfiguration kann es allerdings zu Deadlocks kommen, falls durch die Transformation immer wieder nicht unterstützte Operatoren entstehen (d.h. insbesondere: solche außerhalb von O_1). Andernfalls werden immer wieder Transformationen ausgeführt werden, die solche Teilziele produzieren, die darstellbar sind (ggf. auch **true** bei der Default-Transformationsregel Θ_0), womit nach maximal so vielen Schleifendurchläufen, wie Prädikate vorhanden sind, sämtliche transformierten Prädikate nur noch die erlaubten Prädikate verwenden. Dies bedeutet auch, dass mit korrekter Konfiguration die Transformation in endlicher Zeit mit einer korrekt transformierten Anfrage terminiert.
Eine weitere Forderung war die Äquivalenz von Ergebnistupeln bei Originalanfrage und transformierter Anfrage mit Q_δ-Teil. Dies hängt ausschließlich von den verwendeten Transformationen ab — gibt es eine gegebene Transformation, die nicht ein äquivalentes Ergebnis liefert, so kann nach der Gesamttransformation der Anfrage nicht garantiert werden, dass die Ergebnismenge die gleiche ist.
Ob eine Transformation äquivalent ist, ist allgemein nur mit Formalisierung der Semantik zu beweisen — teilweise wird $\Theta(P) \cup Q_{\delta_\Theta} = P$ sein, womit die Äquivalenz offensichtlich ist; dies ist aber gerade bei komplexeren Operatoren und Prädikaten nicht der Fall. Allerdings erlaubt genau diese Freiheit bei der Konzeption der Transformationen die Adaption auf evtl. vorliegende besondere Verteilungsformen / mögliche Operatoren.
Das vorgestellte Konzept ist tatsächlich wenig limitiert — es ist lediglich nötig, dass sowohl Anfrage als auch Grammatik in RQDL4SQL-Syntax formuliert werden können (was die Definierbarkeit neuer Prädikate möglich macht) und entsprechende Transformationsregeln existieren. So werden entgegen dem bestehenden *Capabilities-based Query Rewriting*-Ansatz nicht nur auch Teilziele in $(G_X \backslash G_C) \cup G_Y \cup G_Z$ gefunden, sondern auch deutlich mehr Anfragen und Fähigkeitenbeschreibungen unterstützt.
Auch die Performance ist recht gut — die Komplexität liegt bei maximal n^2m Schleifendurchläufen (n Prädikate der Anfrage, m Transformationsregeln), im Mittel etwa $2nm$ Schleifendurchläufe mit $O(n)$ Operatoren pro Durchlauf (im Kern eine Transformation des Prädikates P und eine feste Anzahl von Folgetransformationen Θ^* mit n zu überprüfenden Prädikaten), was der derzeit implementierten Transformation eine kubische Komplexität (in Anzahl Prädikate) verleiht.
Allerdings sollen auch diese Ergebnisse nicht über die Tatsache hinweg täuschen, dass es in diesem Bereich noch genügend Raum für weiterführende Arbeiten gibt. Neben den an anderer Stelle angesprochenen nötigen implementatorischen Erweiterungen seien hier einige weitere genannt: So ist nicht garantiert, dass die gegebenen Transformationen optimal sind, d.h. möglichst minimale Obermengen der Originalergebnismengen liefern. So ist es möglich, dass bessere Transformationsregeln gefunden werden können, die dann auch bessere Gesamttransformationen mit sich bringen. Auch ist eine automatische Generierung von Transformationsregeln eventuell möglich (trotz der eingangs genannten Herausforderungen) — auch bei diesen ist der Fokus auf Minimalität zu legen. Für die Unterstützung von mehr Anfragen und die Darstellbarkeit der Fähigkeiten von leistungsstärkeren DBMS-Servern ist die Entwicklung von weiteren Prädikaten unabdingbar, wobei für die effektive Nutzbarkeit auch Transformationsregeln für jedes möglicherweise nicht darstellbare Prädikat entwickelt werden müssen. Ebenso die Untersuchung bestimmter im Laufe der Arbeit erwähnter Randfälle kann zweckmäßig sein, wie die bessere Transformation von *or*-Prädikate durch mengentheoretische Methoden und die Erweiterung der Definition der *sicheren* Prädikate im Rahmen der *Having*-Partition. Auch die Adaption der gegebenen Prädikate und Transformationsregeln an andere Anfragesprachen als SQL lag nicht im Fokus dieser Arbeit, ist aber relativ unproblematisch umsetzbar. Ebenfalls interessant ist die Untersuchung der Durchführbarkeit einer Anfragetransformation mit dem *Answering Queries Using Limited External Query Processors*-Ansatz 2.4 — trotz der im entsprechenden Absatz 2.5 bemerkten Probleme ist dieser Ansatz theoretisch fundiert und damit auch formalisierbar. Die vorgestellten Transformationsregeln können in diesem Ansatz, verbunden mit dem

Provenance-Directed Chase&Backchase, als zusätzliche Chase- und Backchaseregeln konzipiert werden, womit auch die Minimalität beweisbar ist. Es ist allerdings zu beachten, dass hierfür Erweiterungen in den Chase- und Backchase-Algorithmen und dem die Sichtenmenge erzeugenden Datalog-Programm nötig werden. Ebenso sind die Konzepte der *Signaturen* und *Adornments* zu erweitern und es muss untersucht werden, wie sich Anfragetransformationen auf diese auswirken.

Literaturverzeichnis

[DH13] Deutsch, A. und R. Hull: *Provenance-Directed Chase & Backchase.* In Search of Elegance in the Theory and Practice of Computation, 8000:227–236, 2013.

[GH16] Grunert, Hannes und Andreas Heuer: *Datenschutz im PArADISE.* Datenbank-Spektrum, 16(2):107–117, 2016.

[GH17] Grunert, Hannes und Andreas Heuer: *Rewriting Complex Queries from Cloud to Fog under Capability Constraints to Protect the Users' Privacy.* Open Journal of Internet Of Things (OJIOT), 3(1):31–45, 2017. Special Issue: Proceedings of the International Workshop on Very Large Internet of Things (VLIoT 2017) in conjunction with the VLDB 2017 Conference in Munich, Germany.

[HDI12] Halevy, Alon, AnHai Doan und Zachary Ives: *Manipulating Query Expressions.* In: *Principles of Data Integration.* Elsevier, 2012.

[Klu17] Kluth, Johann: *Eignung von Query Containment-Techniken zur Reformulierung von Anfragen bei der datenschutzfreundlichen Gestaltung von Informationssystemen.* Bachelorarbeit, Lehrstuhl für Datenbank- und Informationssysteme, Universität Rostock, 2017. Zur Zeit der Niederschrift noch nicht veröffentlicht.

[LN07] Leser, U. und F. Naumann: *Informationsintegration.* dpunkt.verlag, 2007.

[LRU99] Levy, Alon Y., Anand Rajaraman und Jeffrey D. Ullman: *Answering Queries Using Limited External Query Processors.* Journal of Computer and System Science, 58, 1999.

[Mül16] Müller, Martin: *Generalisierung von Attributen in Relationalen Datenbanken.* Technischer Bericht, Universität Rostock, Institut für Informatik, 2016.

[NJTF15] Nielsen, J.H., D. Janusz, J. Taeschner und J.-C. Freytag: *D2Pt: Privacy-Aware Multiparty Data Publication.* Proceedings of the 16. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW), Hamburg, Germany, 2015.

[Par13] Parr, Terence: *The definitive ANTLR 4 reference.* Pragmatic Bookshelf, 2013.

[PGGMU95] Papakonstantinou, Y., A. Gupta, H. Garcia-Molina und J. Ullman: *A Query Translation Scheme for Rapid Implementation of Wrappers.* Proceedings of the Conference on Deductive and Object Oriented Databases, 1995.

[PGH98] Papakonstantinou, Y., A. Gupta und L. Haas: *Capabilities-Based Query Rewriting in Mediator Systems.* Distributed and Parallel Databases, 6:73–110, 1998.

[Sch16] Schmude, Maik: *Systematische Untersuchung der Anfragekapazität verschiedener DBMS.* Bachelorarbeit, Lehrstuhl für Datenbank- und Informationssysteme, Universität Rostock, 2016.

[Tür99] Türker, Can: *Semantic Integrity Constraints in Federated Database Schemata.* DISDBIS. Infix Verlag, St. Augustin, Germany, 1999.

[Vas09] Vassalos, V.: *Answering Queries Using Views.* Encyclopedia of Database Systems, Seiten 92–98, 2009.

[Weu16] Weu, D.: *Erkennung, Übersetzung und parallele Auswertung von Operatoren und Funktionen aus R in SQL.* Bachelorarbeit, Lehrstuhl für Datenbank- und Informationssysteme, Universität Rostock, 2016.

Anhang A

Anhang Konzept

A.1 Beispielgrammatik

Hier liegt die RQDL4SQL-Beispielgrammatik für die verfügbaren Anfragen an das TinyDB-DBMS vor. Eine allgemeine Beschreibung findet sich in 3.2, der Verweis auf diesen Anhang in 3.2.7.

```
Query:- From, Where, Group, Having, Rename, Select, Order.

From:- Table($TableName).
From:- Select, Where, Group, Having, Rename, Order, From.

Where:- .
Where:- Where, Where.
Where:- eq(Value,Value,CompType).
Where:- gt(Value,Value,CompType).
Where:- bw(Value,$Value1,$Value2).
Where:- in(AttributeName, ValueList).
Where:- or(Where, Where).
Where:- not(Where).
Where:- and(Where,Where).

CompType:- aa.
CompType:- av.

Value:- $Value.
Value:- AttributeName.
Value:- +(Value,Value).
Value:- -(Value,Value).
Value:- /(Value,Value).
Value:- *(Value,Value).
Value:- %(Value,Value).

Group:- .

Having:- .

Rename:- rename(AttributeName,$Name),Rename.
```

```
Rename:- .

Select:- select(*).

Order:- order(AttributeName).
Order:- orderdesc(AttributeName).
Order:- Order, Order.
Order:- .
```

A.2 Transformationsregel: Verbund

Hier wird die verkürzte Transformation für einen nicht erlaubten Verbund gegeben. Die verbale Beschreibung findet sich im Abschnitt Nicht erlaubter Verbund.

```
(Table(Table), Table(Table), Table($Table1), ({Table($Table1)}, {}, {
  ({},{},pred($Table1.$Attribute1,$Table2.$Attribute2),
    ({select($Table1.$Attribute1),select($Table2.$Attribute2)},{pred($Table1.
    $Attribute1,$Table2.$Attribute2)},
  {
    ({},{},select($Table2.$Attribute),({select($Table2.$Attribute)},{select($Table2.
    $Attribute)},{}))
  })),
  ({},{},select($Table1.$Attribute),({select($Table1.$Attribute)},{select($Table1.
  $Attribute)},{}))
}))
```

Anhang B

Anhang Implementierung

B.1 Konvertierung der Where-Prädikate

Hier sind die relevanten Programmteile für die Konvertierung des $WHERE$-Teiles einer Anfrage zu finden. Von besonderer Bedeutung für die konjunktive Darstellbarkeit ist die Funktion $FlattenPredicate$.

```
private static ArrayList<PredicateDescription> GetWhere(PlainSelect statement)
  {
    ArrayList<PredicateDescription> res=new ArrayList<PredicateDescription>();
    Expression where=statement.getWhere();
    try
    {
      res.addAll(FlattenPredicate(TranslateExpression(where)));
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
    return res;
  }

  private static PredicateDescription TranslateExpression(Expression expression) throws
   Exception
  {
    if(expression instanceof AndExpression)
    {
      return new AndPredicate(TranslateExpression(((AndExpression) expression).
      getLeftExpression()),TranslateExpression(((AndExpression) expression).
      getRightExpression()));
    }
    if(expression instanceof OrExpression)
    {
      return new OrPredicate(TranslateExpression(((OrExpression) expression).
      getLeftExpression()),TranslateExpression(((OrExpression) expression).
      getRightExpression()));
    }
    if(expression instanceof EqualsTo)
```

```
    {
      return new EqPredicate(TranslateValueExpression(((EqualsTo) expression).
      getLeftExpression()),TranslateValueExpression(((EqualsTo) expression).
      getRightExpression()));
    }
    if(expression instanceof GreaterThan)
    {
      return new GtPredicate(TranslateValueExpression(((GreaterThan) expression).
      getLeftExpression()),TranslateValueExpression(((GreaterThan) expression).
      getRightExpression()));
    }
    throw new Exception(expression.toString());
  }

  private static ArrayList<PredicateDescription> FlattenPredicate(PredicateDescription
  des)
  {
    ArrayList<PredicateDescription> res=new ArrayList<PredicateDescription>();

    if(des instanceof AndPredicate)
    {
      res.addAll(FlattenPredicate(((AndPredicate) des).first));
      res.addAll(FlattenPredicate(((AndPredicate) des).second));
    }
    else res.add(des);

    return res;
  }
```

B.2 Der Typ *PredicateTemplate*

An dieser Stelle ist die Implementierung des Typs *PredicateTemplate* gegeben. Wofür dieser Typ gebraucht wird und was die einzelnen Attribute sind, wird im Absatz 4.2.1 erläutert.

```
package queryTransformation.types;

import java.util.ArrayList;

public class PredicateBoolTemplate extends PredicateTemplate {
  public PredicateBoolTemplate(ArrayList<PredicateType> enabledTypes, ArrayList<
  PredicateTemplate> enabledFirsts,
      ArrayList<PredicateTemplate> enabledSeconds, ArrayList<CompType>
      enabledCompTypes) {
    EnabledTypes = enabledTypes;
    EnabledFirsts = enabledFirsts;
    EnabledSeconds = enabledSeconds;
    EnabledComparisonTypes = enabledCompTypes;
  }
```

```
public ArrayList<PredicateType> EnabledTypes;
public ArrayList<PredicateTemplate> EnabledFirsts;
public ArrayList<PredicateTemplate> EnabledSeconds;
public ArrayList<CompType> EnabledComparisonTypes;

@Override
public Boolean FulfillsTemplate(PredicateDescription pred) {
  Boolean found = false;
  if (EnabledTypes != null) {
    for (PredicateType enabledType : EnabledTypes) {
      if (enabledType == pred.type) {
        found = true;
        break;
      }
    }
    if (!found)
      return false;
  }
  if (EnabledFirsts != null) {
    PredicateDescription first = null;

    switch (pred.type) {
    case Table:
      first = pred;
      break;
    case And:
      first = ((AndPredicate) pred).first;
      break;
    case Bw:
      break;
    case Eq:
      first = ((EqPredicate) pred).first;
      break;
    case Group:
      break;
    case Gt:
      first = ((GtPredicate) pred).first;
      break;
    case In:
      break;
    case Not:
      break;
    case Or:
      first = ((OrPredicate) pred).first;
      break;
    case Rename:
      break;
    case Select:
      break;
    case Value:
      first = pred;
```

```
        break;
      default:
        break;

      }

      found = false;
      for (PredicateTemplate enabledValue : EnabledFirsts) {
        if (enabledValue.FulfillsTemplate(first)) {
          found = true;
          break;
        }
      }
      if (!found)
        return false;
    }

    if (EnabledSeconds != null) {
      PredicateDescription second = null;
      // if pred has second parameter: check it
      switch (pred.type) {
      case Table:
        break;
      case And:
        second = ((AndPredicate) pred).second;
        break;
      case Bw:
        break;
      case Eq:
        second = ((EqPredicate) pred).second;
        break;
      case Group:
        break;
      case Gt:
        second = ((GtPredicate) pred).second;
        break;
      case In:
        break;
      case Not:
        break;
      case Or:
        second = ((OrPredicate) pred).second;
        break;
      case Rename:
        break;
      case Select:
        break;
      case Value:
        break;
      default:
        break;
```

```
      }

      found = false;
      for (PredicateTemplate enabledValue : EnabledSeconds) {
        if (enabledValue.FulfillsTemplate(second)) {
          found = true;
          break;
        }
      }
      if (!found)
        return false;
    }

    return true;
  }
}
```

B.3 Die abstrakte Klasse *AvailablePredicates*

Hier sind die erlaubten Anfragetypen der unteren Ebene für die Beispielanfrage 3.3.1 gegeben. Erläuterungen finden sich im Abschnitt 4.2.2.

```
package queryTransformation;

import java.util.ArrayList;

import queryTransformation.types.PredicateBoolTemplate;
import queryTransformation.types.PredicateTemplate;
import queryTransformation.types.PredicateType;
import queryTransformation.types.SpecificPredicateBoolTemplate;

/**
 *
 * provides access to Predicates, the description of the capabilities of the lower
 level DBMS
 *
 * @author Christian
 *
 */
public abstract class AvailablePredicates {
  public static ArrayList<PredicateTemplate> Predicates;

  /**
   * SetUp() initializes the List of available predicates
   */
  public static void SetUp() {
    Predicates = new ArrayList<PredicateTemplate>();
    ArrayList<PredicateType> allowedtypes = new ArrayList<PredicateType>();
```

```
    allowedtypes.add(PredicateType.Table);
    ArrayList<PredicateTemplate> enabledFirsts = new ArrayList<PredicateTemplate>();
    ArrayList<Object> enabledValues = new ArrayList<Object>();
    enabledValues.add("emp");
    SpecificPredicateBoolTemplate values = new SpecificPredicateBoolTemplate(
    enabledValues);
    enabledFirsts.add(values);
    Predicates.add(new PredicateBoolTemplate(allowedtypes, enabledFirsts, null, null))
    ;
    allowedtypes = new ArrayList<PredicateType>();
    allowedtypes.add(PredicateType.Select);
    allowedtypes.add(PredicateType.Eq);
    allowedtypes.add(PredicateType.Gt);
    enabledValues = new ArrayList<Object>();
    enabledValues.add("*");
    values = new SpecificPredicateBoolTemplate(enabledValues);
    enabledFirsts = new ArrayList<PredicateTemplate>();
    enabledFirsts.add(values);
    Predicates.add(new PredicateBoolTemplate(allowedtypes, enabledFirsts, null, null))
    ;
  }
}
```

B.4 Die abstrakte Klasse *AvailableTransformation*

Hier ist die Transformationsregel für die nicht verfügbare Tabelle für das gegebene *Table*-Prädikat gegeben. Nicht beachtet sind Prädikattypen, die noch nicht von der Konvertierung erzeugt werden können sowie mögliche *rename*-Prädikate.

```
package queryTransformation;

import java.util.ArrayList;
import java.util.function.Function;

import queryTransformation.types.AndPredicate;
import queryTransformation.types.AttributeValuePredicate;
import queryTransformation.types.EqPredicate;
import queryTransformation.types.GtPredicate;
import queryTransformation.types.OrPredicate;
import queryTransformation.types.PredicateBoolTemplate;
import queryTransformation.types.PredicateDescription;
import queryTransformation.types.PredicateForTransformation;
import queryTransformation.types.PredicateTemplate;
import queryTransformation.types.PredicateType;
import queryTransformation.types.QueryRewriting;
import queryTransformation.types.QueryTransformation;
import queryTransformation.types.SelectPredicate;
import queryTransformation.types.SpecificPredicateBoolTemplate;
import queryTransformation.types.TablePredicate;
```

```
import queryTransformation.types.ValuePredicate;

/**
 *
 * provides access to Transformations, the List of available query transformations
 *
 * @author Christian
 *
 */
public abstract class AvailableTransformations {
  public static ArrayList<QueryTransformation> Transformations;
  private static PredicateType currentOperation;

  /**
   * sets up the List of available transformations and predicates for the lower level
   * has to be called once before query transformation is possible
   * if it is not called, there will be no changes made to the query
   * unless AvailablePredicates.SetUp() was called, which will result in an infinite
   loop
   */
  public static void AddTransformations() {
    AvailablePredicates.SetUp();
    Transformations = new ArrayList<QueryTransformation>();
    ArrayList<PredicateTemplate> requiredPredicates = new ArrayList<PredicateTemplate
    >();
    ArrayList<PredicateType> allowedtypes = new ArrayList<PredicateType>();
    allowedtypes.add(PredicateType.Table);
    ArrayList<PredicateTemplate> enabledFirsts = new ArrayList<PredicateTemplate>();
    ArrayList<Object> enabledValues = new ArrayList<Object>();
    enabledValues.add("*");
    SpecificPredicateBoolTemplate values = new SpecificPredicateBoolTemplate(
    enabledValues);
    enabledFirsts.add(values);
    PredicateBoolTemplate tablePredicate = new PredicateBoolTemplate(allowedtypes,
    enabledFirsts, null, null);
    requiredPredicates.add(tablePredicate);
    Function<PredicateForTransformation, QueryRewriting> execute = pred -> {
      PredicateDescription toApply = pred.Predicate;
      QueryRewriting res = pred.Rewriting;

      String tableName = ((TablePredicate) toApply).first;

      res.QueryRewriting.remove(toApply);
      res.RemainingQuery.add(toApply);

      PredicateDescription[] currentRewriting = new PredicateDescription[res.
      QueryRewriting.size()];
      currentRewriting = res.QueryRewriting.toArray(currentRewriting);

      for (PredicateDescription thisPredicate : currentRewriting) {
        int includestable = includesTable(thisPredicate, tableName);
```

```
    if (includestable != 0) {
      res.QueryRewriting.remove(thisPredicate);
      res.RemainingQuery.add(thisPredicate);
      if (includestable != 3) {
        ArrayList<ValuePredicate> FoundPredicates = new ArrayList<ValuePredicate
        >();
        switch (currentOperation) {
        case And:
          if (includestable == 1) {
            FoundPredicates = ((AndPredicate) thisPredicate).second.FindAttributes
            (tableName);
          } else
            FoundPredicates = ((AndPredicate) thisPredicate).first.FindAttributes(
            tableName);
          break;
        case Bw:
          break;
        case Eq:
          if (includestable == 1) {
            FoundPredicates = ((EqPredicate) thisPredicate).second.FindAttributes(
            tableName);
          } else
            FoundPredicates = ((EqPredicate) thisPredicate).first.FindAttributes(
            tableName);
          break;
        case Gt:
          if (includestable == 1) {
            FoundPredicates = ((GtPredicate) thisPredicate).second.FindAttributes(
            tableName);
          } else
            FoundPredicates = ((GtPredicate) thisPredicate).first.FindAttributes(
            tableName);
          break;
        case Or:
          if (includestable == 1) {
            FoundPredicates = ((OrPredicate) thisPredicate).second.FindAttributes(
            tableName);
          } else
            FoundPredicates = ((OrPredicate) thisPredicate).first.FindAttributes(
            tableName);
          break;
        default:
          break;
        }
        for (ValuePredicate found : FoundPredicates) {
          res.QueryRewriting.add(new SelectPredicate(found));
        }
      }
    }

  }
```

```
    return res;
  };
  QueryTransformation trans = new QueryTransformation(requiredPredicates, new
  ArrayList<PredicateTemplate>(),
      tablePredicate, execute);
  Transformations.add(trans);
}

/**
 * searches the predicate for occurrences of expressions from another table
 *
 * @param pred Predicate where to look
 * @param tableName tableName to be excluded in the search
 * @return 0, if pred does not contain any other table; 3, if all parameters contain
  other tables; 1, if first contains another table; 2, if second contains
 */
private static int includesTable(PredicateDescription pred, String tableName) {
  switch (pred.type) {
  case Table:
    return 0;
  case And:
    currentOperation = pred.type;
    if (includesTable(((AndPredicate) pred).first, tableName) > 0) {
      if (includesTable(((AndPredicate) pred).second, tableName) > 0)
        return 3;
      return 1;
    }
    if (includesTable(((AndPredicate) pred).second, tableName) > 0)
      return 2;
    return 0;
  case Bw:
    break;
  case Eq:
    currentOperation = pred.type;
    if (includesTable(((EqPredicate) pred).first, tableName) > 0) {
      if (includesTable(((EqPredicate) pred).second, tableName) > 0)
        return 3;
      return 1;
    }
    if (includesTable(((EqPredicate) pred).second, tableName) > 0)
      return 2;
    return 0;
  case Group:
    break;
  case Gt:
    currentOperation = pred.type;
    if (includesTable(((GtPredicate) pred).first, tableName) > 0) {
      if (includesTable(((GtPredicate) pred).second, tableName) > 0)
        return 3;
      return 1;
    }
```

```
      if (includesTable(((GtPredicate) pred).second, tableName) > 0)
        return 2;
      return 0;
    case In:
      break;
    case Not:
      break;
    case Or:
      currentOperation = pred.type;
      if (includesTable(((OrPredicate) pred).first, tableName) > 0) {
        if (includesTable(((OrPredicate) pred).second, tableName) > 0)
          return 3;
        return 1;
      }
      if (includesTable(((OrPredicate) pred).second, tableName) > 0)
        return 2;
      return 0;
    case Rename:
      break;
    case Select:
      return includesTable(((SelectPredicate) pred).first, tableName);
    default:
      // ValuePredicates land here
      if (pred instanceof AttributeValuePredicate) {
        if (((AttributeValuePredicate) pred).tableName.compareTo(tableName) == 0)
          return 3;
      }
    }
    return 0;
  }
}
```

B.5 Die Anfragetransformation

Hier ist die Implementation der Anfragetransformation gegeben. Nicht betrachtet sind die 3. und 4. Bedingung von 4.2.2.

```
package queryTransformation;

import java.util.ArrayList;

import queryTransformation.conversion.RQDL4SQLConverter;
import queryTransformation.types.PredicateDescription;
import queryTransformation.types.PredicateForTransformation;
import queryTransformation.types.PredicateTemplate;
import queryTransformation.types.QueryDescription;
import queryTransformation.types.QueryRewriting;
import queryTransformation.types.QueryTransformation;
```

```
/**
 * main transformation class
 *
 * @author Christian
 *
 */
public abstract class TransformQuery {
  private static String rewrittenQuery;
  private static ArrayList<PredicateDescription> remainingQuery;

  /**
   * sets the current active rewriting to the given query rewriting
   *
   * @param rewriting the to be set rewriting
   */
  private static void setQueryRewriting(QueryRewriting rewriting) {
    rewrittenQuery = RQDL4SQLConverter.ConvertToQuery(rewriting.QueryRewriting);
    remainingQuery = rewriting.RemainingQuery;
  }

  /**
   * gets the transformed query
   *
   * @return the transformed query string of the latest set query transformation
   */
  public static String GetQuery() {
    return rewrittenQuery;
  }

  /**
   * gets the remainder query
   *
   * @return the remaining query predicates of the latest set query transformation
   */
  public static ArrayList<PredicateDescription> GetRemaining() {
    return remainingQuery;
  }

  /**
   * transforms the given query and saves the values
   * retrieve via GetQuery(), GetRemaining()
   *
   * @param OriginalQuery the original query to be transformed
   * @throws Exception if something went wrong during Parsing/Conversion
   */
  public static void transform(String OriginalQuery) throws Exception {
    AvailableTransformations.AddTransformations();
    ArrayList<PredicateTemplate> AllowedPredicates = AvailablePredicates.Predicates;
    QueryDescription description = RQDL4SQLConverter.ConvertToRQDL4SQL(OriginalQuery);
    QueryRewriting rewriting = new QueryRewriting(description);
```

```
  while (true) {
    ArrayList<PredicateDescription> predicates = GetNotSupportedPredicates(rewriting
    .QueryRewriting,
        AllowedPredicates);

    if (predicates.size() == 0)
      break;

    for (PredicateDescription predicate : predicates) {
      for (QueryTransformation trans : AvailableTransformations.Transformations) {
        if (isApplicable(predicate, trans, AllowedPredicates)) {
          PredicateForTransformation pred = new PredicateForTransformation(
          rewriting, predicate);
          rewriting = trans.Transform(pred);
          // only one transformation per predicate
          break;
        }
      }
    }
  }

  setQueryRewriting(rewriting);
}

/**
 * tells if the given transformation is applicable
 *
 * @param pred the to be transformed query predicate
 * @param trans the tested query transformation
 * @param AllowedPredicates the allowed predicates of the lower level
 * @return true, if the transformation can be executed; false, if not
 */
private static Boolean isApplicable(PredicateDescription pred, QueryTransformation
trans,
    ArrayList<PredicateTemplate> AllowedPredicates) {
  return trans.Predicate.FulfillsTemplate(pred) && canBeExecuted(pred,
  AllowedPredicates);
}

/**
 * tells if the predicate is not supported by the lower level server, so a
 transformation is applicable
 *
 * @param pred the tested predicate
 * @param AllowedPredicates the capabilities of the lower level
 * @return false, if pred is supported by the lower level server; true, if not
 */
private static Boolean canBeExecuted(PredicateDescription pred, ArrayList<
PredicateTemplate> AllowedPredicates) {
  // TODO: add containment function between trans.Operators1 and AllowedPredicates
  // to see if it HAS to be executed
```

```
    // at the moment the function just checks if the predicate can be operated
    for (PredicateTemplate temp : AllowedPredicates) {
      if (temp.FulfillsTemplate(pred))
        return false;
    }
    return true;
  }

  /**
   * gets the not supported predicates
   *
   * @param predicates all to be checked predicates
   * @param allowedPredicate the capabilities of the lower level server
   * @return all predicates in predicates that are NOT supported by the lower level (e
   .g. have to be transformed)
   */
  private static ArrayList<PredicateDescription> GetNotSupportedPredicates(ArrayList<
  PredicateDescription> predicates,
      ArrayList<PredicateTemplate> allowedPredicate) {
    ArrayList<PredicateDescription> res = new ArrayList<PredicateDescription>();
    Boolean supported;

    for (PredicateDescription pred : predicates) {
      supported = false;
      for (PredicateTemplate temp : allowedPredicate) {
        if (temp.FulfillsTemplate(pred)) {
          supported = true;
          break;
        }
      }
      if (!supported)
        res.add(pred);
    }

    return res;
  }
}
```

Anhang C

Anhang Datenträger

C.1 Aufbau des Datenträgers

Auf der beigefügten CD befinden sich zwei Ordner: *Quellcode* und *doc*. Im Ordner *Quellcode* befindet sich der Programmcode für die Implementation des Anfragetransformationsalgorithmus. In ihm befinden sich vier *.java*-Dateien und die beiden Ordner *conversion* und *types*. Die vier *.java*-Dateien beinhalten Klassen, die sich spezifisch mit der eigentlichen Transformation von Anfragen befassen. Die Datei *TestTransform-Query.java* beinhaltet Mustercode zum Aufruf und der Nutzung der Transformationsfunktionalität. Die verwendete Beispielanfrage und die gewählte Umgebung ist identisch mit dem in 3.3.1 gegebenen.
Der Ordner *conversion* beinhaltet die Klassen, die sich mit der Konvertierung von unterschiedlichen Typen ineinander befasst, wie der *RQDL4SQLConverter*, der die Konvertierung von SQL-Anfrage (*String*) in RQDL4SQL-Prädikatmenge und umgekehrt durchführt. Der Ordner *types* enthält Klassen, die sich mit der Deklaration der notwendigen Typen befassen.
Der Ordner *doc* auf der anderen Seite enthält die entstehende Javadoc (ein automatisches Dokumentationswerkzeug im JDK), die aus dem Quellcode generiert wird. Die Erläuterungen im Quellcode und damit auch in der entstehenden Javadoc sind in englischer Sprache gehalten und erstrecken sich auf alle erklärungsbedürftigen Member der Klassen, nicht nur öffentliche und damit auch von außen nutzbare.

C.2 Datenträger

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 1. September 2017