Eignung von Query Containment-Techniken zur Reformulierung von Anfragen bei der datenschutzfreundlichen Gestaltung von Informationssystemen

Bachelorarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik

vorgelegt von: Johann Kluth
Matrikelnummer: 214205878
geboren am: 29.11.1995 in Ludwigslust
Erstgutachter: Prof. Dr. rer. nat. habil. Andreas Heuer
Zweitgutachter: Dr.-Ing. Holger Meyer
Betreuer: Hannes Grunert
Abgabedatum: 07.09.2017

Inhaltsverzeichnis

1 Einleitung **5**

1.1 Motivation . . . 5
1.2 Problemstellung . . . 7
1.3 Erläuterungen . . . 8
1.4 Das PArADISE-Projekt . . . 9
1.4.1 Grundlagen und Ausgangssituation . . . 9
1.4.2 Einordnung dieser Arbeit . . . 10
1.5 Leitfaden zur Arbeit . . . 10

2 Stand der Forschung und Technik **11**

2.1 Stand der Forschung . . . 11
2.1.1 Answering Queries using Views . . . 11
2.1.2 Answering Queries Using Operators . . . 13
2.1.3 Bucket . . . 14
2.1.4 MiniCon . . . 17
2.1.5 Inverse Rules . . . 19
2.2 Stand der Technik . . . 21

3 Konzept **23**

3.1 Vergleich der Algorithmen . . . 23
3.1.1 Allgemein . . . 23
3.1.2 Einsatzmöglichkeiten . . . 24
3.1.3 Laufzeiteffizienz . . . 24
3.1.4 Zusammenfassung . . . 25

4 Implementierung **27**

4.1 Vorbereitung . . . 28
4.2 Die Klassen . . . 28
4.2.1 Klasse: MiniCon . . . 28
4.2.2 Klasse: Query . . . 28
4.2.3 Klasse: View . . . 29

4.2.4 Klasse: MCD 30
4.2.5 Weitere Klassen 30
4.3 Wichtige Funktionen 31
4.3.1 checkFirstMiniConProperty(View) 31
4.3.2 checkSecondMiniConProperty(View) 32
4.3.3 formingMCDs(List<View>) 32
4.3.4 combiningMCD(List<MCD>) 32
4.4 Testfälle 33

5 Zusammenfassung und Ausblick **37**
5.1 Zusammenfassung 37
5.2 Ausblick 38

6 Aufbau des Datenträgers **39**
6.1 Aufbau 39
6.2 Datenträger 40

Literaturverzeichnis **41**

A Anhang **45**
A.1 combiningMCD(List<MCD>) 45

Kapitel 1

Einleitung

Riesige, verteilte Datenbestände in Datenbanken sind heutzutage mehr die Regel als eine Ausnahme. Somit wird es immer wichtiger, effizient auf diese Daten zugreifen zu können. Ein Problem bei diesen Verteilungen ergibt sich dadurch, dass die gesuchten Attribute einer Anfrage auf verschiedenen Servern liegen können. Wenn ein Nutzer nun auf diese Daten zugreifen möchte, sollte er sich aber keine Gedanken darum machen müssen, welche Daten auf welchem Server liegen. Wenn der Nutzer eine Anfrage stellt, soll diese automatisch und möglichst effizient verteilt an die beteiligten Server geleitet werden.
Außerdem können solche Techniken zum Verteilen von Anfragen auch für eine einzelne, nicht verteilte Datenbank genutzt werden, indem die Anfrage auf die dort vorhandenen materialisierten Sichten verteilt wird. Dadurch kann die Geschwindigkeit der Bearbeitung der Anfrage erhöht werden, weil diese Sichten auf der Datenbank bereits vorberechnet im Speicher liegen. Materialisierte Sichten haben den größten Effekt für Anfragen, welche oft abgefragt werden. Sie profitieren dadurch am meisten von der Beschleunigung [Köl10].

1.1 Motivation

Solche Arten von Anfragen werden heutzutage schon an verteilte Datenbanken gestellt, indem die Anfragen nicht direkt an einen Server gerichtet sind, sondern an ein globales Schema, welches alle Server des Clusters abdeckt. Auf den einzelnen Teilknoten der Datenbank liegen verschiedene Relationen, wobei einzelne Knoten die Relationen auch redundant speichern können.
Bei einer Anfrage wird mit der Hilfe des globalen Datenbankschemas entschieden, welche Daten von welchen Servern geholt werden müssen. Im besten Fall kommen die Daten nur von einem einzigen Server, im schlechtesten aber von einer sehr großen Menge an verschiedenen Servern. Um die Anfragen an die Knotenpunkte der Datenbank verteilen zu können, werden effiziente Algorithmen benötigt. Die Algorithmen, welche es für solche Fälle gibt, werde ich im Laufe dieser Arbeit genauer vorstellen. Sie sind darauf ausgerichtet, die Anfrage nur an die Server zu stellen, welche die benötigten Daten speichern.

Diese sogenannten Query-Containment-Techniken haben aber auch noch andere Anwendungsmöglichkeiten. Wenn in smarten Umgebungen, wie dem Smart Appliance Lab an der Universität Rostock

[YNK14], die Umgebungssensoren Aktivitäten der Nutzer aufzeichnen, können die smarten Systeme im Hintergrund die Intentionen der Nutzer erkennen und darauf eigenständig Handlungen ausführen. Dabei könnten aber auch Datenschutzprobleme entstehen, weil die aufgezeichneten Daten personenbezogen sein können. Die Nutzung der personenbezogenen Daten ist laut dem Recht auf informationelle Selbstbestimmung [Deu49, Art.1 + Art.2] nur für die Verwendungszwecke erlaubt, für die der Benutzer im Vorfeld zugestimmt hat.
Um das Gesetz einzuhalten, kann bei der Auswertung von Daten mit Sichtkonzepten, feingranularen Zugriffsrechten und Datenschutz-Algorithmen, wie dem Slicing [LLZM12] oder der k-Anonymisierung [Swe02], gearbeitet werden, wodurch der Zugriff auf die Daten beschränkt wird. Diese Maße anonymisieren die Daten einer Anfrage jedoch nur im Nachhinein. Es wäre aber auch möglich, die Anfrage zu analysieren und dabei partielle Vorberechnungen auf leistungsarmen Rechnern, wie Sensoren und Kleinstrechnern, durchzuführen. Die restlichen Berechnungen können dann mit diesen voraggregierten und vorselektierten Daten auf dem Zielrechner durchgeführt werden. Bei der Berechnung des Zwischenergebnisses dürfen die Daten aber keine Informationen verlieren, welche für die späteren Berechnungen notwendig sind.
Daher müssen die Verfahren bei diesem sogenannten AQuV[1], entweder ein *maximally contained set of rewritings* liefern, oder ein *rewriting supremum*[2] . Welche dieser Mengen benötigt wird, hängt von der jeweiligen Anwendung ab. Beim normalen AQuV wird ein *maximally contained set of rewritings* gesucht, wohingegen in einer abgewandelten Form, dem AQuO[3], ein *rewriting supremum* gesucht wird. Ein *maximally contained set* ist die Menge von Umschreibungen der Teilziele, welche in einer Anfrage zusammengefasst werden können. Die dabei entstehende Anfrage ist zur Originalen so identisch wie nur möglich. Das bedeutet, dass eine maximale Anzahl von Tupeln aus der originalen Anfrage in dieser Menge enthalten ist. Es gibt dementsprechend keine weitere Umschreibung mehr, die näher an die originale Anfrage herankommt. Das *rewriting supremum* hingegen liefert minimal mehr Ergebnisse als noch die originale Anfrage und ist damit die Umschreibung, welche am minimalsten mehr Ergebnisse liefert, im Vergleich zu anderen Umschreibungen. Im Idealfall wären bei einem *maximally contained set* die Ergebnismenge gleich der Ergebnismenge der originalen Anfrage, genau wie auch beim *rewriting supremum* [Gru17]. Zur Veranschaulichung eines *rewriting supremums* dient die Abbildung 1.1.
Die Ergebnismenge des *rewriting supremums* reicht für die Anfrage eigentlich aus, um sie ausreichend zu beantworten. Die heutigen Systeme arbeiten aktuell jedoch eher nach einem anderen Prinzip: *Gib mir alle Daten die du mir liefern kannst und ich suche mir meine notwendigen Daten dort heraus* [GH17]. Genau das ist hier der datenschutzkritische Punkt. Es werden zu viele unnötige personenbezogene Daten an externe Server gesendet und dort verarbeitet, obwohl sie für die ursprüngliche Verwendung nicht benötigt werden. Daher stellt sich die Frage, warum die Endanwender ihre privaten Daten einfach so preisgeben sollen — geht das nicht auch datenschutzfreundlicher?
Wenn wie beim Beantworten von Datenbankanfragen mit Hilfe von Sichten eine Anfrage verteilt wird, werden diese Sichten benutzt, um Umschreibungen zu finden. Wenn im Gegensatz dazu nun eine Umschreibung gefunden werden soll, um den Datenschutz in einer smarten Umgebung zu erhalten, dann sollen

[1] Answering Queries using Views, siehe dazu 2.1.1
[2] Rein technisch ist ein *rewriting supremum* ein *minimally more containing set of rewritings.*
[3] Answering Queries using Operators, siehe dazu 2.1.2

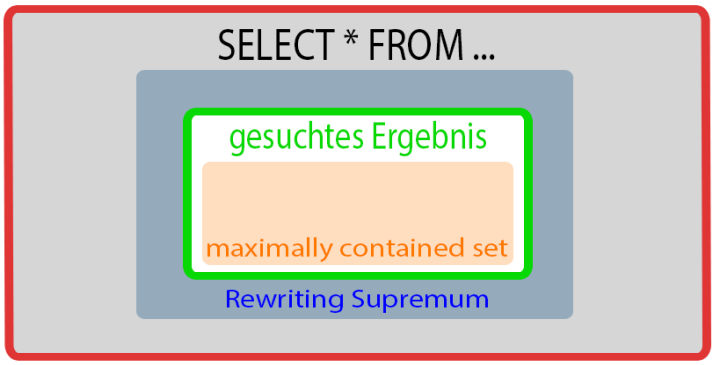

Abbildung 1.1: Die Darstellung der durch die Anfrage gesuchten Ergebnismenge (grüner Kasten) im Vergleich zur gesamt möglichen Ergebnismenge der Datenbank (roter Kasten). Das *rewriting supremum* befindet sich je nach Anfrage im blau-grauen Bereich zwischen der gesuchten und der gesamten Ergebnismenge. Im besten Fall liegt es direkt auf dem grünen Rand, wobei es wirklich nur die benötigten Daten liefert, und im schlechtesten Fall liegt es auf dem roten. Das *maximally contained set* einer AQuV Anfrage liegt in dem orangen Bereich, innerhalb der gesuchten Ergebnismenge. Auch hierbei wäre der grüne Rand das Optimum.

Sichten zusammen mit Operationen verwendet werden, welche jedoch aufgrund der Hardwareeigenschaften von Sensoren und Kleinstrechnern limitiert sind. Dabei können beispielsweise Verbundoperationen verboten werden, sowie Aggregatfunktionen oder in speziellen Fällen sogar Projektionen. Das ist jeweils von der Rechenleistung der anzusprechenden Rechner abhängig, da oftmals nicht genügend Ressourcen für diese Operationen vorhanden sind. Auf diesen leistungsschwachen Rechnern sind nur sehr einfache Operationen möglich, mit denen verhältnismäßig wenig gefiltert werden kann. Mit der Hilfe dieser limitierten Operationen sollen Anfragen an Sensoren in smarten Umgebungen möglich sein, welche dann möglichst nur die Daten liefern sollen, die auch wirklich benötigt werden um bestimmte Situationen zu erkennen. Anstatt einer Menge von Sichten gibt es hier beim AQuO eine Menge von erlaubten Operationen auf den Sensoren und deren Sichten. Weil bei AQuO-Anfragen bedeutende Erkenntnisse gewonnen werden sollen, ist es wichtig, dass bei einer Umschreibung der originalen Anfrage auf die erlaubten Operationen mindestens die Menge der benötigten Ergebnisse zurückgeliefert wird. Wenn ein Teilziel beim AQuV oder AQuO nicht umgeschrieben werden kann, wird es aus der originalen Anfrage übernommen [GH17].

1.2 Problemstellung

Eine Anforderung an die Anfrage, welche mit Query-Containment-Techniken bearbeitet werden soll, ist, dass die Teilberechnungen immer eine Obermenge der gewünschten Ergebnisse zurückliefern. Im schlimmsten Fall wäre so eine Umschreibung gleich einer $SELECT * FROM...$ - Anfrage. Sie ist zwar eine Obermenge, aber in den seltensten Fällen minimal [Gru17].

In dieser Arbeit wird untersucht, welche Verfahren zum Testen von Query Containment und das Finden des Rewriting Supremums für die Probleme im PArADISE-Projekt[4] geeignet sind. Dazu werden bereits

[4] Privacy AwaRe Assistive Distributed Information System Environment

entwickelte Algorithmen untersucht, welche eine Datenbankanfrage auf Sichten umschreiben können (AQuV). In späteren Arbeiten kann auf diese Erkenntnisse zurückgegriffen werden und die Algorithmen so modifiziert werden, dass sie Operationen in den Anfragen erkennen und diese umschreiben können (AQuO). Das beste AQuV-Verfahren soll anschließend in Java implementiert werden und im PArADISE Projekt zum Einsatz kommen.

Die Anfragen an die Datenbanken werden in Structured Query Language (SQL) formuliert. Als Datenbanksystem wird PostgreSQL genutzt. Dementsprechend arbeiten die Algorithmen auch auf diesen SQL-Anfragen und sollen entsprechende Umschreibungen auch wieder als eine SQL-Anfrage formulieren, welche letztendlich an die Datenbank gestellt werden kann.

1.3 Erläuterungen

Für diese Arbeit sind einige Vorkenntnisse nötig. Zum einen arbeiten alle hier untersuchten Algorithmen mit Sichten auf Datenbanken. Diese werden sehr ausführlich im Buch **Datenbanken: Konzepte und Sprachen** [SSH13, 469 ff.] beschrieben. Die hier vorgestellten Algorithmen arbeiten vor allem mit materialisierten Sichten. Dabei handelt es sich um bereits vorberechnete Datenbankanfragen, welche im Zwischenspeicher des Datenbanksystems gespeichert werden, um einen möglichst schnellen Zugriff auf diese Daten zu ermöglichen [Köl10].

Des Weiteren wird öfters die Bezeichnung *„Conjunctive Queries"* (dt. konjunktive Anfragen) fallen. Diese können wie folgt als einzelne, negations- und funktionsfreie Datalog-Regeln [SSH13], die syntaktisch wie Prolog-Regeln aussehen, dargestellt werden:

$$Q(\overline{X}) \leftarrow R_1(\overline{X}_1) \wedge ... \wedge R_n(\overline{X}_n) \text{ [Vas09].}$$

In dieser formalen Anfrage sind $R_1(\overline{X}_1), ..., R_n(\overline{X}_n)$ Teilziele, welche auch als Konjunkte (engl. conjuncts) bezeichnet werden. Die R_i's stehen dabei für Datenbankrelationen mit den Tupeln $\overline{X}_i$ aus Variablen und Konstanten [DHI12]. Die Variablen im Kopf der Anfrage werden als ausgezeichnete Variablen bezeichnet. Eine wichtige Bedingung der konjunktiven Anfrage ist die Sicherheit: Eine Anfrage heißt sicher, wenn jede ausgezeichnete Variable in einem Teilziel $R_n(\overline{X}_n)$ vorkommt [Vas09].

Eine weitere Technik, welche in der Einleitung schon des Öfteren erwähnt wurde, ist das *„Query Containment"* (kurz QC). Dabei geht es um das Testen, ob eine Datenbankanfrage in einer anderen enthalten ist. Das wird benötigt, um die Richtigkeit von Anfrageumschreibungen zu testen [Vas09].
Eine dazu ähnliche Technik ist der Test auf *„Query Equivalence"*. Dabei wird eine Datenbankanfrage auf semantische Äquivalenz zu einer anderen Anfrage getestet (siehe Kapitel 2).

1.4 Das PArADISE-Projekt

Bei dem PArADISE-Projekt handelt es sich um ein Datenschutz-Framework, welches an der Universität Rostock entwickelt wird. Die in dieser Arbeit vorgestellten Techniken werden in dieses Framework mit eingebunden, um die Datensparsamkeit zu fördern, indem nur Anfragen auf freigegebenen Daten möglich sein sollen. Im folgenden Abschnitt wird dieses Projekt genauer vorgestellt.

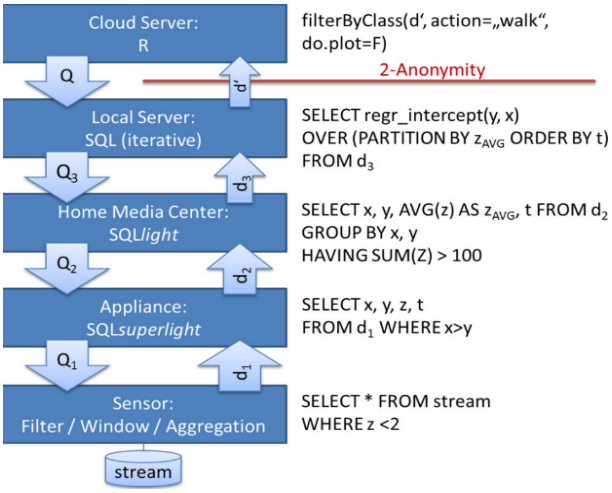

Abbildung 1.2: Stufen der Anfrageumschreibungen [GH16].

1.4.1 Grundlagen und Ausgangssituation

Das Framework wird mit dem Ziel entwickelt, smarte oder auch assistierende Umgebungen datenschutzfreundlicher zu gestalten. Momentan sammeln und verarbeiten viele solcher Assistenzsysteme zu viele personenbezogene Informationen, welche sie nicht alle benötigen um ihren Zweck zu erfüllen. Diese ungefilterten Daten werden von den Betreibern an Server gesendet, wo diese erst verarbeitet werden. Dabei könnten die Daten schon viel früher gefiltert und nur die wichtigen Informationen letztendlich auch weitergeleitet werden. Um die Datenschutzbestimmungen, wie das Recht auf informationelle Selbstbestimmung, welches im Bundesdatenschutzgesetz [Deu17] gesetzlich festgeschrieben ist, durchzusetzen, werden zusätzliche Sicherungen in den aktuellen Systemen benötigt. Das PArADISE-Framework soll Forschern und Entwicklern die Möglichkeit geben, smarte Assistenzsysteme zu erschaffen, welche die Privatsphäre der Nutzer respektieren. Dies wäre unter anderem möglich, indem die Informationen aus den Sensoren auch schon dort, oder auf einer höheren Ebene im System, verarbeitet werden würden (siehe Abbildung 1.2). In den blauen Kästen der Abbildung werden die verschiedenen Stufen der Verarbeitung der Informationen aus den Sensoren (ganz unten) bis zu den externen Cloud Servern (ganz oben) dargestellt. Rechts daneben befindet sich ein Beispiel zu der möglichen SQL-Version, welche in dieser Ebene eingesetzt werden kann um die Anfragen zu filtern. Eine Anfrage wird dabei vom Cloud Server gestellt und durch die einzelnen Ebenen hindurch bis an die Sensoren geleitet. Zur Beantwortung der Anfrage leitet der Sensor sein Ergebnis durch die Ebenen wieder zurück an den Cloud Server [GH16].

1.4.2 Einordnung dieser Arbeit

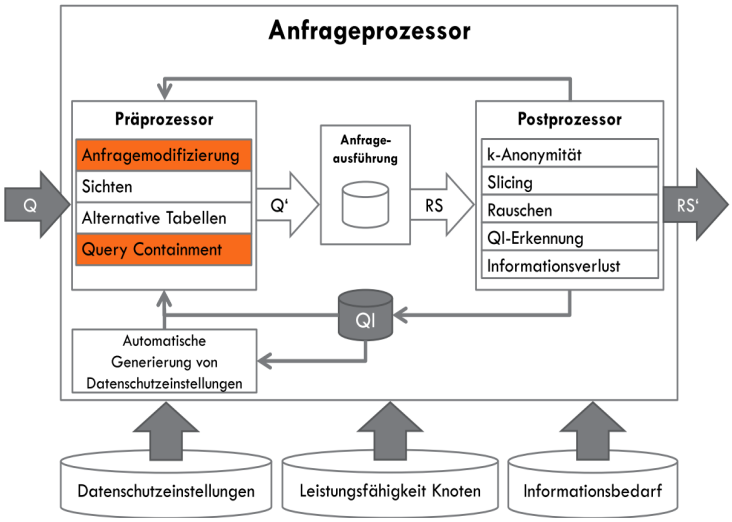

Abbildung 1.3: Die Einordnung von QC im Anfrageprozessor des PArADISE-Projekts [Gru17].

Wie in der Abbildung 1.3 orange markiert ist, findet das Umschreiben von Datenbankanfragen im Präprozessor des Anfrageprozessors statt. Das bedeutet, dass die Anfrage analysiert und umformuliert wird, bevor sie an die Datenbank gestellt wird. Das Gleiche gilt in diesem Fall auch für Anfragen an Assistenzsysteme, nur dass dabei die Anfrage an die verschiedenen Stufen verteilt werden kann, wie es in der Abbildung 1.2 grafisch dargestellt ist. Bei Anfragen an Assistenzsysteme wird zur Anfragemodifizierung die AQuO-Technik verwendet, welche im Kapitel 2 genauer erklärt wird.

1.5 Leitfaden zur Arbeit

Zu Beginn wird der aktuelle Stand der Forschung und Technik bezüglich der hier untersuchten Algorithmen im Kapitel 2 – „Stand der Technik und Forschung" vorgestellt. Im Anschluss daran werden die Algorithmen im Kapitel 3 – „Konzept" verglichen und bewertet. Der Algorithmus, welcher sich dabei als am besten für das PArADISE-Projekt passend herausstellt, wird in Java implementiert und in das Projekt mit eingebunden. Zur Dokumentation dazu dient das Kapitel 4 – Implementierung. Zum Abschluss der Arbeit wird diese noch einmal zusammengefasst und ein Ausblick auf ähnliche Techniken wird im Kapitel 5 – „Zusammenfassung und Ausblick" gegeben.

Kapitel 2

Stand der Forschung und Technik

Die grundlegenden Probleme vom Beantworten von Datenbankanfragen mit der Hilfe von Sichten, stammen aus der Zeit, als die ersten verteilten Datenbanken angelegt wurden. Dort gab es das Problem, dass die verteilten Datenbanken jeweils nur einige Bereiche einer Datenbank gespeichert hatten. Dadurch stellte sich die Frage, wie mit einer Anfrage auf alle Datenbanken zugegriffen werden kann. Gelöst wurde das ganze Problem mit einem globalen Schema, welches über mehrere Datenbanken gelegt ist. Wenn eine Anfrage an die Datenbank gestellt wurde, ging diese an das Schema welches die Anfrage an die einzelnen Knoten der Datenbank verteilte. Am Anfang der 1990er Jahre wurden Sichten auf Datenbanken erstmal zum Beschleunigen von Anfragen verwendet, obwohl diese Sichten in der Anfrage nicht explizit gegeben waren. Einige Jahre später entstanden aus diesen Techniken Lösungen für das jetzige Problem vom Beantworten von Datenbankanfragen mit der Hilfe von materialisierten Sichten [Vas09], welche in dieser Arbeit genauer analysiert werden.

2.1 Stand der Forschung

In diesem Kapitel wird der aktuelle Stand der Forschung zu den in dieser Arbeit relevanten Gebieten präsentiert. Dazu werden zum Beginn die AQuV- und AQuO-Techniken ausführlich erklärt und anschließend Algorithmen dazu vorgestellt, welche Anfragen auf Sichten umschreiben.

2.1.1 Answering Queries using Views

Bei dem Beantworten von Datenbankanfragen mittels Sichten (engl. „Answering Queries using Views" oder auch „Query Rewriting using Views" [Vas09]) handelt es sich eher um eine Gruppe von Problemen als um ein einzelnes Problem an sich. Worum es dabei grundlegend geht, steht formal in der Definition 1 beschrieben.

Definition 1. *Gegeben ist eine Anfrage Q an ein Datenbankschema Σ, in einer Anfragesprache L_Q und eine Menge von Sichten $V = \{V_1, V_2, ..., V_n\}$ über dasselbe Datenbankschema, in der Anfragesprache L_V. Ist es möglich, eine Antwort auf die Anfrage Q zu bekommen, indem (nur) die Sichten aus V genutzt werden [Vas09]?*

Eine äquivalente Formulierung der Definition 1 lautet: *Was ist die maximale Anzahl an Tupeln in der Antwort von Q, welche nur aus den Sichten gewonnen werden können [Vas09]?*
Das Umformulieren der Anfragen auf Sichten hat in der Anwendung mehrere Vorteile. Zum einen kann mit den Sichten der Datenschutz eingehalten werden, indem nur freigegebene Daten in die Sichten mit aufgenommen werden. Die Rechte der Nutzer über den Zugriff auf diese Datenbank können so erteilt werden, dass Anfragen nur auf den Sichten erlaubt sind. Der Zugriff auf die anderen Daten in den Relationen ist dann nicht mehr möglich.
Weiterhin können materialisierte Sichten erstellt werden, welche eine komplette Datenbankanfrage als eine neue temporäre Relation im schnellen Zwischenspeicher eines Systems anlegen. Gestellte Anfragen an diese Sichten können schneller beantwortet werden, weil die Zugriffszeit auf den Zwischenspeicher sehr gering ist. Diese Art der Anwendung von Sichten wird durch die Algorithmen, welche in dieser Arbeit vorgestellt werden, ausgenutzt um eine Datenbankanfrage zu beschleunigen. Um eine Anfrage äquivalent auf die Sichten der Datenbank umzuschreiben, muss diese zuerst einmal in Teilziele zerlegt werden. Damit dies überhaupt möglich ist, muss die Anfrage in der konjunktiven Normalform vorliegen, das heißt, dass sie nur aus Konjunktionen einzelner Teilanfragen zusammengebaut ist. Wenn diese Voraussetzung erfüllt ist, stellt jedes Literal der Konjunktion ein Teilziel dar. Alle Teilziele sollen, nachdem sie auf Sichten abgebildet wurden, zusammen weniger oder gleich viele Ergebnisse zurückliefern als die originale Anfrage (maximally contained set of rewritings). Formal wird beim AQuV ein Rewriting $r(V) = Q_1$ gesucht, sodass:

$$Q_1(D) \sqsubseteq Q(D) \leftrightarrow \forall d \in D : Q_1(d) \subseteq Q(d).$$

Dabei sind die Sichten (V) über einer Datenbank (D) gegeben und $Q(D)$ steht für die originale Anfrage. $r(V) = Q_1$ bedeutet, dass wir ein Rewriting der Anfrage Q suchen, welches abhängig von der Menge von Sichten (V) erstellt wurde und welches hier als Q_1 bezeichnet wird. Im Idealfall liefern die umgeschriebenen Anfragen das gleiche Ergebnis, welches auch durch die originale Anfrage geliefert würde: $Q_1(D) \equiv Q(D)$. Nach dem Test auf Query-Containment wäre das Rewriting der Anfrage in seinem Original enthalten. Da das Rewriting zu seiner originalen Anfrage äquivalent ist, wäre diese originale Anfrage auch wieder im Rewriting enthalten. Wenn keine Äquivalenz vorliegen würde und das Rewriting somit weniger Ergebnisse liefern würde als noch die originale Anfrage, dann ist, bezüglich des Query Containments, nur das Rewriting in seiner originalen Anfrage enthalten [LMSS95].

Es gibt verschiedene Anforderungen an das Finden von Rewritings, welche im Folgenden als Probleme bezeichnet werden. Das erste Problem, welche es beim AQuV gibt, ist ein Rewriting einer Anfrage zu finden, welches materialisierte Sichten benutzt. Für die Lösung dieses Problems werden in den Abschnitten 2.1.3 bis 2.1.5 Algorithmen vorgestellt, welche eine gegebene Datenbankanfrage und eine gegebene Menge von Sichten über diese Datenbank nehmen und daraus eine umgeschriebene Anfrage entwickeln,

welche möglichst komplett auf den Sichten ausgeführt werden kann.
Ein weiteres Problem stellt das Finden eines minimalen Rewritings einer Anfrage da. Dies wird benötigt, damit die gestellte Anfrage möglichst effizient, mit der Hilfe von Sichten, ablaufen kann. Minimal bedeutet hier, dass möglichst wenig verschiedene Sichten benutzt werden müssen und damit weniger Verbundoperationen zwischen den Daten notwendig sind, welche diese Anfrage wieder verlangsamen würden.
Das dritte Problem ist, ein vollständiges Rewriting der Anfrage zu finden. Dieses Rewriting muss mit der originalen Anfrage semantisch äquivalent sein. Die Lösungen dieser drei Probleme sind im Allgemeinen nicht berechenbar. Wenn aber die Anfrage und die Sichten konjunktiv sind und keine eingebauten Vergleichsprädikate haben, dann sind sie nur NP-vollständig. Ein dazu äquivalentes Problem, welches auch die gleichen Bedingungen für die NP-Vollständigkeit besitzt, ist eine Anfrage zu finden, welche möglichst wenige Relationen einer Datenbank verwendet [LMSS95].

Die Komplexität der Probleme kommt aus zwei verschiedenen Quellen. Einerseits von der Anzahl der möglichen „containment mappings", also der Menge der Zuordnungen zwischen Anfrage und Sichten. Im Normalfall ist die Komplexität hier aber noch nicht einmal exponentiell, sondern nur quadratisch. Der Algorithmus ist nur in seltenen, realitätsfernen Fällen exponentiell, nämlich wenn mindestens zwei Quellen (Literale aus der Anfrage) auf mindestens zwei Ziele (Literale aus der Sicht) abgebildet werden müssen. Für unsere spätere Anwendung im PArADISE-Framework ist dies aber nicht relevant, da die von uns genutzte Anfrageverarbeitung immer einen Baum bildet und somit zwei Quellen immer nur auf ein Ziel abgebildet werden[1]. Um die Geschwindigkeit der Anfrage durch die exponentielle Berechnung der Optimierung nicht zu verringern, kann die Optimierung für diese Teile auch weggelassen werden. Dadurch wird die Anfrage zwar nicht schneller, aber da die Optimierung zu kostenaufwändig wäre, kann diese in solch einem Fall auch weggelassen werden, damit keine Nachteile entstehen. Andererseits kommt die Komplexität der Probleme von der Entscheidung, wie viele Literale der Anfrage gelöscht werden können. Dies sind im Normalfall alle Tupel, welche nicht mehr abgebildet werden müssen, weil bereits ein anderes Literal diesen Teil überdeckt. Dass solche Literale gelöscht werden, beschleunigt die Anfrage, weil weniger Rechenoperationen notwendig sind [LMSS95].

Die verschiedenen AQuV-Techniken werden nicht nur genutzt um Datenbankanfragen zu optimieren. Sie werden unter anderem auch zur Datenintegration eingesetzt, sowie in Data Warehouses. Dort werden die Techniken genutzt um die Cubes als materialisierte Sichten zu speichern und die Anfrageoperationen auf ihnen durchzuführen, um diese zu beschleunigen [LMSS95].

2.1.2 Answering Queries Using Operators

Mit dem PArADISE-Framework sollen die Anfragen nicht in erster Linie auf Sichten umgeschrieben werden. Das eigentliche Ziel ist, die Ergebnisse der Anfrage, zur Anonymisierung der Daten, möglichst früh von den Sensoren, oder den Rechnern in der Hierarchie über ihnen (siehe Abbildung 1.2), zu filtern. Aus diesem Grund muss ein Rewriting der Anfrage stattfinden, sodass möglichst nur grundlegende SQL-

[1] Mitteilung von Hannes Grunert vom 18.07.2017

Techniken wie Selektion und Projektion oder einfache Vergleiche genutzt werden. Anstatt einer Menge von Sichten, kann hier eine Menge von erlaubten Operationen zur Verfügung stehen. Dieses Rewriting der Anfrage, um auf einfache SQL-Techniken zu kommen, ist für die leistungsschwachen Sensoren notwendig. Sie können im Normalfall keine komplexen SQL-Anfragen verarbeiten und sind dadurch auf möglichst einfache und damit stark filternde SQL-Techniken angewiesen.
Die formale Definition von AQuO ist der von AQuV sehr ähnlich. Gegeben sind hierbei wieder eine Datenbank D (es kann auch ein Sensor sein, welcher von Stromdaten liest) und eine Anfrage $Q(D)$ an ebendiese, sowie eine Menge von erlaubten Operationen in Q_1. Dazu wird ein Rewriting $r(Q) = Q_1$ gesucht, sodass

$$Q_1(D) \sqsupseteq Q(D) \leftrightarrow \forall d \in D : Q_1(d) \sqsupseteq Q(d).$$

Q_1 enthält dann nur noch erlaubte Operationen. Diese umgeschriebene Anfrage soll, im Gegensatz zu AQuV, eine minimal größere Ergebnismenge zurückliefern, als noch die originale Anfrage, wobei im Idealfall auch hier die Ergebnisse der Anfragen gleich wären: $Q_1(D) \equiv Q(D)$. Diese minimal größere Ergebnismenge wird benötigt, da bei weniger Ergebnissen nicht das gleiche Ergebnis am Ende der Anfrage geliefert werden würde. Der Grund dafür ist, dass auf höheren Schichten der Anfrageumschreibung (siehe Abbildung 1.3) noch weitere Anfragen durchgeführt werden, welche schlussendlich das korrekte Resultat liefern sollen. Diese Operationen sind aber nur korrekt auf einer Obermenge möglich, nicht aber auf einer Teilmenge des Ergebnisses, da dort eventuell schon wichtige Tupel fehlen[2]. Wenn es mehr oder gleich viele Ergebnisse sind, ist das Ergebnis der Anfrage wieder korrekt [GH17].

Diese Vorgehensweise streben wir im PArADISE-Projekt an. Da dies aber den Umfang einer Bachelorarbeit übersteigen würde, liegt der Schwerpunkt dieser Arbeit auf den Vorleistungen, die dafür benötigt werden. Damit sind hier die einfachen AQuV-Algorithmen gemeint. Im Folgenden werden drei Algorithmen vorgestellt, welche die AQuV-Technik nutzen. Diese Algorithmen können gegebene Anfragen umschreiben, indem eine dazu gegebene Menge von Sichten auf eine Datenbank genutzt wird. Die neue umgeschriebene Anfrage soll dann möglichst nur mit den Sichten in der Datenbank arbeiten.

2.1.3 Bucket

Der erste Algorithmus zum Erstellen von Anfrageplänen mit Hilfe von Sichten ist der Bucket-Algorithmus. Als Eingabe nimmt er eine Anfrage und eine Menge von Sichten und liefert damit eine Menge von semantisch korrekten Anfrageplänen, von denen jeder Plan jeweils auf den gegebenen Sichten arbeitet. Der Vorteil von semantisch korrekten Plänen ist, dass die Antworten, die sie liefern, auch die Antworten für die originale Anfrage sind. Die Grundidee vom Algorithmus ist, den Suchraum für das Rewriting einzuschränken, sodass möglichst die komplette Anfrage mit Sichten beantwortet werden kann. Wenn dies nicht vollständig möglich ist, sollte zumindest der größtmögliche Teil der Anfrage mit Sichten abgedeckt werden.
Der Ablauf des Algorithmus erfolgt in zwei Schritten, wobei im **ersten Schritt** [LRO96a] die semantisch korrekten konjunktiven Pläne generiert werden, welche im **zweiten Schritt** [LRO96b] sortiert werden,

[2]Mitteilung von Prof. Dr. Andreas Heuer vom 02.09.2017

um zu garantieren, dass die Pläne auch ausgeführt werden können.
Beim genaueren Betrachten des **ersten Schrittes** kann dieser nochmals in drei Abschnitte unterteilt werden:

1. Zuerst wird die Anfrage in Teilziele zerlegt. Weil die gegebene Anfrage in der konjunktiven Normalform vorliegt, können die einzelnen Konjunkte dafür verwendet werden.
2. Danach wird versucht, für die Teilziele jeweils passende Sichten zu finden.
3. Sind so viele Teilziele wie möglich durch Sichten abgedeckt, werden diese Sichten kombiniert, um die ursprüngliche Anfrage erfüllen zu können.

Wenn ein passendes Rewriting gefunden wurde, muss dann im **zweiten Schritt** durch Query Containment geprüft werden, ob dieses auch tatsächlich semantisch äquivalent zur ursprünglichen Anfrage ist. Abschließend wird der beste Plan noch einmal optimiert, indem redundante Teilziele gelöscht werden, falls solche vorhanden sind.
Beispiel: Zitierzirkel (Bucket) [3]
Anfrage:

$$Q_1(x) \leftarrow zitiert(x,y) \wedge zitiert(y,x) \wedge gleichesGebiet(x,y)$$

Sichten:

$$V_1(a) \leftarrow zitiert(a,b) \wedge zitiert(b,a)$$
$$V_2(c,d) \leftarrow gleichesGebiet(c,d)$$
$$V_3(f,h) \leftarrow zitiert(f,g) \wedge zitiert(g,h) \wedge gleichesGebiet(f,g)$$

Aus der gegebenen Anfrage und den Sichten können nun die Buckets generiert werden. Diese sehen wie folgt aus (jede Spalte steht für ein Bucket):

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)

Tabelle 2.1: Die gefundenen Buckets der Anfrage Q.

Jetzt wird im zweiten Schritt das kartesische Produkt der Buckets gebildet, wobei die Prädikate zu konjunktiven Anfragen zusammengefasst werden. Dadurch kann nun das Query Containment der Buckets bezüglich der Anfrage Q geprüft werden. Wenn dabei eine umgeschriebene Anfrage ein Teil der Ergebnisse der originalen Anfrage liefert, dann ist diese konjunktive Anfrage ein Teil des Ergebnisses. Die dabei gefundenen konjunktiven Anfragen werden im Ergebnis vereinigt.

[3] Beispiele abgeleitet aus [Heu17].

Die gefundenen Kombinationen der Buckets:

1. $V_1(x) \wedge V_1(x) \wedge V_2(x, y)$
2. $V_1(x) \wedge V_1(x) \wedge V_3(x, y)$
3. $V_1(x) \wedge V_3(x, y) \wedge V_2(x, y)$
4. $V_1(x) \wedge V_3(x, y) \wedge V_3(x, y)$
5. $V_3(x, y) \wedge V_1(x) \wedge V_2(x, y)$
6. $V_3(x, y) \wedge V_1(x) \wedge V_3(x, y)$
7. $V_3(x, y) \wedge V_3(x, y) \wedge V_2(x, y)$
8. $V_3(x, y) \wedge V_3(x, y) \wedge V_3(x, y)$

Bei dem Test auf Query Containment schlagen alle Kombinationen mit der Sicht $V_1(x)$ fehl. Dadurch bleiben nur noch die folgenden Kombinationen:

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

Dass V_1 nicht genutzt werden kann, liegt an den Verbundbedingungen über y, welche in V_1 nicht darstellbar sind. Wenn bei den verbleibenden Kombinationen die gleichen Prädikate eliminiert werden, bleiben noch die Folgenden möglich:

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

Durch das Containment Mapping werden dann in der Sicht V_3 die Variablen auf die der Anfrage abgebildet. Dabei gibt es folgende Abbildungen:

Variablen der Anfrage: $x,\ y$
Variablen der Sicht V_3: $f,\ g,\ h$
Abbildung von f: $f \rightarrow x$
Abbildung von g: $g \rightarrow y$
Abbildung von h: $h \rightarrow x$

Mit der Sicht $V_3(x, y)$ wurde durch den Bucket-Algorithmus das Rewriting $Q'_1(x)$ gefunden, welche ein äquivalentes Ergebnis zur originalen Anfrage liefert: $Q'_1(x) \leftarrow V_3(x, x)$ [Heu17].

□

Ein großes Problem, welches der Bucket-Algorithmus hat, ist, dass bei ihm die Teilziele alle einzeln betrachtet werden. Das hat zur Folge, dass dadurch beim erstellen des kartesischen Produktes im **zweiten Schritt** eine sehr große Menge an möglichen Sichten zustande kommen kann, welche dann beim Prüfen der semantischen Äquivalenz alle betrachtet werden müssen, welche von diesen Sichten sich am besten für die optimierte Anfrage eignen. Dadurch wird die Laufzeit stark erhöht und der Algorithmus ineffizient, wenn das Berechnen einer schnellen Anfrage durch Sichten länger dauern würde als die Anfrage ohne eine Umschreibung. Weil der Bucket-Algorithmus dadurch langsam werden kann, wurde auf seiner Grundlage der MiniCon-Algorithmus entwickelt [Hal01].

2.1.4 MiniCon

Der MiniCon-Algorithmus ist eine Verbesserung vom Bucket-Algorithmus [PH01], was sich auch am sehr ähnlichen Aufbau dieser Algorithmen zeigt. Wie beim Bucket-Algorithmus wird auch hier begonnen, passende Sichten für Teilziele einer gegebenen Anfrage zu finden. Im **ersten Schritt** werden, wie auch beim Bucket-Algorithmus, die Anfragen in Teilziele zerlegt, bevor passende Sichten für die Teilziele gesucht werden.
Wenn nun eine passende Sicht für ein Teilziel gefunden wurde, werden die Variablen der Anfrage noch einmal genauer analysiert, indem geschaut wird, ob die gefundene Sicht noch andere Teilziele abdecken kann. Dabei beachtet der Algorithmus auch die Join-Prädikate der Anfrage (diese sind durch das mehrfache Auftreten derselben Variable leicht zu erkennen) und findet zu ihnen die minimal zusätzliche Menge an Teilzielen, welche all die Teilziele der Anfrage beinhaltet, die auf die Teilziele der gefundenen Sicht abgebildet werden müssen, wenn die originale Anfrage auf diese Sicht abgebildet wird. Diese Menge von Teilzielen und Abbildungsinformationen wird *MiniCon Description* (kurz MCD) genannt und kann als eine Verallgemeinerung der Buckets angesehen werden. Formal sind die MiniCon Descriptions in der Definition 2 beschrieben [PH01].

Definition 2. *(MiniCon Description) Ein MCD C für eine Anfrage Q über einer Sicht V ist ein Tupel der Form* $(h_C, V(\bar{Y})_C, \varphi_C, G_C)$ *wobei gilt:*

- h_C *ist der Kopf-Homomorphismus der Sicht V,*
- $V(\bar{Y})_C$ *ist das Ergebnis der Anwendung von* h_C *auf die Sicht V, das bedeutet, dass* $\bar{Y} = h_C(\bar{A})$*, wobei* $\bar{A}$ *die Kopf-Variablen von V sind,*
- φ_C *ist die partielle Abbildung von Vars(Q) auf* $h_C(Vars(V))$
- G_C *ist eine Teilmenge der Teilziele von Q, welche von einigen Teilzielen in* $h_C(V)$ *abgedeckt sind, indem die Abbildung* φ_C *genutzt wird (nicht alle solche Teilziele müssen in* G_C *sein).*

Zusätzlich zu diesen Regeln gibt es noch eine Besonderheit. Die gefundene MCD C kann nur dann für ein nichtredundantes Rewriting der Anfrage Q genutzt werden, wenn folgende zwei Bedingungen gelten:

1. *Für jede Kopf-Variable x aus der Anfrage Q, welche im Wertebereich von* φ_C *liegt, ist* $\varphi_C(x)$ *eine Kopf-Variable in* $h_C(V)$*.*

2. *Wenn $\varphi_C(x)$ eine existenzielle Variable in $h_C(V)$ ist, dann gilt für jedes Zwischenziel G der Anfrage Q, welches x beinhaltet, dass alle Variablen aus G auch im Wertebereich von φ_C liegen und $\varphi_C(G) \in h_C(V)$.*

In anderen Worten: φ_C ist eine Abbildung von Q auf die Spezialisierung der Sicht V, welche durch den Kopf-Homomorphismus h_C erhalten wird. G_C ist eine Menge von Teilzielen aus Q, welche durch die Abbildung von φ_C abgedeckt sind [PH01].

Im **zweiten Schritt** werden diese MCDs kombiniert um konjunktive Rewritings der Anfrage zu generieren. Hierbei ist zu erwähnen, dass durch die Art, wie die MCDs erstellt werden, keine Query Containment-Überprüfungen mehr gemacht werden müssen, wodurch der Algorithmus an diesem Punkt schneller arbeitet als der Bucket-Algorithmus [PH01].
Nach dem Kombinieren der MCDs wird für jede gültige Kombination ein konjunktives Rewriting der Anfrage erstellt. Während des Kombinierens muss der MiniCon-Algorithmus nur die Kombinationen von MCDs betrachten, die paarweise disjunkte Mengen der Teilziele abdecken, wodurch redundante Rewritings der Anfrage vermieden werden. Aus diesem Grund verringert sich der Suchraum des Algorithmus signifikant. Am Ende des **zweiten Schrittes** werden redundante Teilziele aus dem Rewriting entfernt. Dadurch entsteht das endgültige Ergebnis des Algorithmus, welches eine Vereinigung von konjunktiven Anfragen ist [PH01].

Beispiel: Zitierzirkel (MiniCon)
Als Beispielanfrage verwenden wir hier auch die Anfrage und die dazu gegebenen Sichten aus dem Bucket-Beispiel. Die Anfrage Q kann wie folgt in ihre einzelnen Teilziele aufgeteilt werden:

1. $zitiert(x, y)$
2. $zitiert(y, x)$
3. $gleichesGebiet(x, y)$

Angefangen mit dem ersten Teilziel, wird nach einer passenden Sicht gesucht. Eine Sicht wird hier als passend bezeichnet, wenn sie die erste und zweite Bedingung für eine MCD erfüllt.
Zum Erfüllen der ersten MCD-Bedingung muss eine Sicht im Kopf mindestens eine Kopfvariable der Anfrage besitzen. Für die zweite Bedingung muss eine Variable aus einem Verbund der Anfrage entweder auch im Kopf der Sicht stehen, oder die Sicht hat denselben Verbund wie die Anfrage. Da V_2 und V_3 diese Bedingungen erfüllen, können mit ihnen die folgenden MCDs gebildet werden:

$h(v_i)$	h	φ	G
$V_2(c, d)$	$c \to c, d \to d$	$x \to c, y \to d$	3
$V_3(f, f)$	$f \to f, h \to f$	$x \to f, y \to g$	1, 2, 3

Tabelle 2.2: Die gefundenen MCDs der Anfrage Q.

In der ersten Spalte dieser Tabelle stehen die Kopfhomomorphismen. Sie werden in der zweiten Spalte auf die Variablen in der Sicht angewandt. In der dritten Spalte stehen die partiellen Abbildungen von Variablen aus der Anfrage auf die Variablen der Sicht. In der letzten Spalte G stehen die Teilziele, welche von der jeweiligen Sicht abgedeckt werden können.
Diese erstellten MCDs werden dann kombiniert und zu einer Menge von konjunktiven Rewritings vereinigt [Heu17]. Da auch hier das beste Rewriting gesucht wird, würde auch der MiniCon-Algorithmus die Sicht V_3 als optimales Rewriting der Anfrage liefern.

□

2.1.5 Inverse Rules

Der dritte hier vorgestellte Algorithmus besitzt einen ganz anderen Ansatz als die Vorhergehenden. Der „Inverse Rules" Algorithmus (dt. „invertierte Regeln") wurde ursprünglich für Datenintegrationssysteme entwickelt und basiert auf einem logischen Ansatz. Die Idee hinter diesem Algorithmus ist es, eine Menge von Regeln zu erstellen, welche die gegebenen Sichtdefinitionen invertieren. Das heißt, dass die Regeln dann zeigen, wie Tupel der Datenbankrelationen aus den Tupeln der Sichten berechnet werden. Für jedes Teilziel in dem Körper einer Sicht wird eine inverse Regel gebildet. Dazu wird für jede nichtausgezeichnete Variable in der Sicht ein Funktionssymbol generiert, welches im Kopf der inversen Regel eingesetzt wird. Diese neuen Funktionssymbole zeigen, welche Informationen aus den Sichten extrahiert werden können [Vas09]. Für eine gegebene Anfrage und einer gegebenen Menge von Sichten liefert der Inverse-Rules-Algorithmus das maximally-contained Rewriting der Anfrage, indem die Sichten genutzt werden [Hal01].

Beispiel: Zitierzirkel (Inverse Rules) - Teil I
Als Anfrage und Sichten werden hier auch wieder diejenigen aus dem Bucket-Beispiel verwendet. Als erstes werden diese Sichten invertiert:

$$\begin{aligned} V_1(a) &\leftarrow zitiert(a,b) \wedge zitiert(b,a) \\ V_2(c,d) &\leftarrow gleichesGebiet(c,d) \\ V_3(f,h) &\leftarrow zitiert(f,g) \wedge zitiert(g,h) \wedge gleichesGebiet(f,g) \end{aligned}$$

Die invertierten Sichten, wobei f das Funktionssymbol darstellt:

$$\begin{aligned} R_1 &: zitiert(a, f_{V_1}(a)) \leftarrow V_1(a) \\ R_2 &: zitiert(f_{V_1}(a), a) \leftarrow V_1(a) \\ R_3 &: gleichesGebiet(c,d) \leftarrow V_2(c,d) \\ R_4 &: zitiert(f, f_{V_3}(f,h)) \leftarrow V_3(f,h) \\ R_5 &: zitiert(f_{V_3}(f,h), h) \leftarrow V_3(f,h) \\ R_6 &: gleichesGebiet(f, f_{V_3}(f,h)) \leftarrow V_3(f,h) \end{aligned}$$

Zwischen dem Bucket- und dem Inverse-Rules-Algorithmus gibt es einige Gemeinsamkeiten und Unterschiede. Im ersten Schritt berechnet der Bucket-Algorithmus eine Menge von Sichten, welche für die Anfrage relevant sind. Dabei berücksichtigt der Bucket-Algorithmus auch den Kontext, in welchem die Atome der Anfrage vorkommen, der Inverse-Rules-Algorithmus hingegen nicht. Das bedeutet, dass wenn eine Sicht gefunden wird, welche für die Anfrage irrelevant ist, wird sie beim Bucket-Algorithmus nicht weiter berücksichtigt. Beim Inverse-Rules-Algorithmus hingegen, kann es passieren, dass diese Sicht weiter genutzt und in die umgeschriebene Anfrage mit aufgenommen wird. Dafür hat der Inverse-Rules-Algorithmus aber einen anderen Vorteil: Die inversen Regeln können einmal berechnet und dann auf jede passende Anfrage angewendet werden. Um irrelevante Sichten aus den umformulierten Anfragen zu entfernen, welche vom Inverse-Rules-Algorithmus erstellt wurden, muss eine Propagierungsphase angeschlossen werden, wie sie in [LFS97] und [SR92] beschrieben wurde [Hal01].

Wichtige Vorteile des Inverse-Rules-Algorithmus sind seine Einfachheit und die Modularität. Er kann beispielsweise erweitert werden, um funktionale Abhängigkeiten auf den Datenbankschemata auszunutzen oder um rekursive Anfragen beantworten zu können. Diese Modularität wird erreicht, indem zusätzliche Regeln zu den bereits berechneten, inversen Regeln, hinzugefügt werden. Die Komplexität des Algorithmus ist im Normalfall polynomial zu der Länge der Anfrage und der Sichten. Dabei kann der Algorithmus am Ende jedoch nicht garantieren, dass das gefundene *Rewriting supremum* oder das *maximally contained set of rewritings* auch äquivalent zur originalen Anfrage ist. Es gibt nämlich keinen eingebauten Test auf Query Containment. Dementsprechend ist das Problem, ein Rewriting zu finden, welches äquivalent zur Ausgangsanfrage sein soll, NP-vollständig [LMSS95], aber auch nur wenn überhaupt eine Lösung existiert. Das größte Problem dieses Algorithmus, welches ihn in der Praxis unbrauchbar macht, ist, dass eine Lösung für ein Rewriting vorausgesetzt werden muss. Diese Entscheidung ist für rekursive Regeln unentscheidbar. Der Algorithmus würde dementsprechend nie terminieren und kann nicht entscheiden, ob es eine Lösung gibt [DG97a]. Ein weiteres großes Problem, welches der Inverse-Rules-Algorithmus besitzt, ist, dass durch die zu einfache Berechnung des Rewritings die Vorteile der materialisierten Sichten größtenteils verloren gehen [Hal01]. Dementsprechend wäre er für die Beschleunigung von Datenbankanfragen mittels materialisierten Sichten auch nicht zu gebrauchen.

Beispiel: Zitierzirkel (Inverse Rules) - Teil II

Die invertierten Sichten werden der Anfrage hinzugefügt. Die neue Anfrage $Q'_1(x)$ sieht wie folgt aus:

$$\begin{aligned} Q'(x) = \; & zitiert(x,y) \wedge zitiert(y,x) \wedge gleichesGebiet(x,y) \wedge zitiert(a, f_{V_1}(a)) \wedge \\ & zitiert(f_{V_1}(a), a) \wedge gleichesGebiet(c,d) \wedge zitiert(f, f_{V_3}(f,h)) \wedge \\ & zitiert(f_{V_3}(f,h), h) \wedge gleichesGebiet(f, f_{V_3}(f,h)) \end{aligned}$$

Damit ist das Rewriting fertig und diese neue Anfrage wird dem Optimierer des Datenbankmanagementsystems überlassen. Dieser filtert aus der Anfrage die unnötigen Sichten heraus und löst so viele Verbunde wie möglich dabei auf [Heu17].

□

2.2 Stand der Technik

In diesem Unterabschnitt werden existente Softwarelösungen erwähnt, welche die soeben präsentierten Algorithmen verwenden. Wie bereits im Abschnitt Answering Queries using Views erwähnt wurde, stecken diese Techniken in den Grundlagen von bestimmten Systemen zur Datenintegration. Dort wurden diese Algorithmen nach ihrer Vorstellung höchstwahrscheinlich auch eingesetzt, wie der Inverse Rules-Algorithmus, welcher speziell für das Infomaster System entwickelt wurde [DG97b]. Weitere Verwendungen dieser Algorithmen finden sich nur in den Prototypen, welche für die Präsentation der wissenschaftlichen Arbeiten entwickelt wurden: MiniCon [PH01] und Bucket [LRO96b].
Speziell in der Datenintegration können diese AQuV-Algorithmen für die Integration mittels Local-As-View und Global-As-View genutzt werden [FKPT11] [FKMP03]. Weiterhin gibt es aus der Forschung einige Implementationen dieser Algorithmen. Beispielsweise nutzte der Forschungsprototyp "Clio" von IBM diese Techniken für die Datenintegration[4].
Weitere Verwendungen dieser Algorithmen sind nicht leicht zu finden, da sie speziell für den Einsatz in Anwendungen entwickelt wurden, welche ein direktes Ziel besitzen. Alleinstehend haben diese Algorithmen keinen großen Nutzen. Da die Firmen keine Auskunft über die verwendete Technik in ihren Softwaresystemen geben, kann nur vermutet werden, dass Softwarelösungen für die Datenintegration oder Data Warehouses diese Techniken, vielleicht auch in einer angepassten Form, verwenden[5].

[4] Mitteilung von Prof. Dr. Andreas Heuer vom 02.09.2017.
[5] Mündliche Mitteilung von Prof. Dr. Andreas Heuer vom 31.05.2017

Kapitel 3

Konzept

Hier im Konzept werden die im Stand der Forschung und Technik vorgestellten Algorithmen genauer verglichen und anhand dessen wird entschieden, welcher dieser Algorithmen im weiteren Verlauf der Arbeit implementiert werden soll. Wenn der beste oder die besten Algorithmen gefunden wurden, werden noch einige Testfälle entwickelt, mit denen der implementierte Algorithmus auf Funktionalität getestet werden kann.

3.1 Vergleich der Algorithmen

Hier vergleiche ich die Algorithmen Bucket, MiniCon und Inverse Rules nach Funktionsweise und Komplexität sowie nach der möglichen Einsetzbarkeit im PArADISE Projekt. Für die Vergleiche werden bereits durchgeführte Tests aus dem originalen Artikel zum MiniCon-Algorithmus [PH01] verwendet sowie Informationen aus dem Buch "Principles of Data Integration" [DHI12].

3.1.1 Allgemein

Beim Vergleichen des Aufbaus der Algorithmen fällt auf, dass der Bucket- und der MiniCon-Algorithmus auf dem gleichen Ansatz basieren. Sie durchsuchen die gegebenen Sichten einer Datenbank auf ihre Art und finden dabei die Passenden für eine gegebene Anfrage. Bei diesen Algorithmen wird auch das Enthaltensein der Abfrageergebnisse im Rewriting garantiert. Der Inverse-Rules-Algorithmus hingegen hat einen ganz anderen Ansatz, welcher in der Vorstellung dieses Algorithmus bereits erklärt wurde. Die Sichtdefinitionen können unabhängig von einer Anfrage bereits vorbereitet werden und dann einfach in die Anfrage eingesetzt werden. Die eigentliche Erstellung eines Rewritings wird dabei dem Anfrageoptimierer des Datenbankmanagementsystems überlassen.

3.1.2 Einsatzmöglichkeiten

Die drei Algorithmen können alle für die gleichen Gebiete eingesetzt werden, für welche sie auch entwickelt wurden. Das Beantworten von Datenbankabfragen mittels Sichten wird genutzt, um Datenmanagementprobleme zu lösen, Anfragen zu optimieren, Datenbanken zu integrieren und um Data-Warehouse-Anfragen zu beantworten. Dafür kann ein beliebiger Algorithmus eingesetzt werden, um diese Probleme zu lösen [Hal01].

3.1.3 Laufzeiteffizienz

Die theoretische Laufzeit im schlechtesten Fall ist bei allen drei Algorithmen gleich. Sie beträgt:

$$O((n\ m\ M)^n)$$

wobei **n** für die Anzahl der Teilziele in der Anfrage, **m** für die maximale Anzahl von Teilzielen in Sichten und **M** für die Anzahl der Sichten steht. In der Realität zeigen sich aber teils deutliche Unterschiede zwischen den Algorithmen. Im Artikel [PH01] haben die Autoren bereits einige Tests mit den Algorithmen vollzogen. Das System, auf welchem die Tests durchgeführt wurden, ist aus heutiger Sicht schon veraltet, die Aussage hinter den Diagrammen hat sich aber nicht verändert. Alle Bewertungen wurden auf einem Rechner mit dem Betriebssystem Windows NT 4.0 durchgeführt, welcher einen Pentium II Prozessor mit 450 MHz besitzt und 512 MB Hauptspeicher verbaut hat. Die Algorithmen waren alle in Java implementiert und zu einer ausführbaren Datei kompiliert.

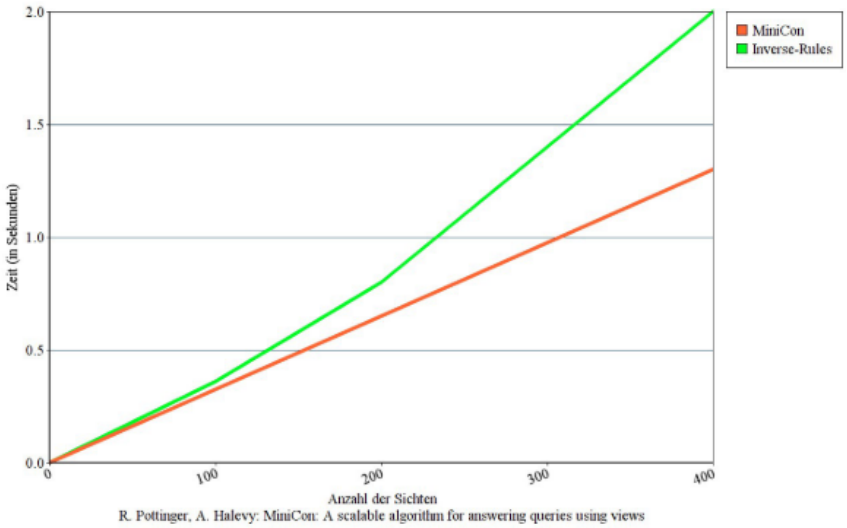

Abbildung 3.1: Anfragen mit zehn Teilzielen und zwei unterschiedlichen Variablen.

In den Abbildungen 3.1 und 3.2 ist zu erkennen, dass der MiniCon-Algorithmus im direkten Vergleich zum Inverse-Rules-Algorithmus der direkte Gewinner ist, sowie dass der MiniCon-Algorithmus besser skaliert als sein Kontrahent. Während der Inverse-Rules-Algorithmus exponentiell mit der Anzahl der Sichten skaliert, steigt die Zeitdauer zum Finden der Rewritings beim MiniCon-Algorithmus nur linear

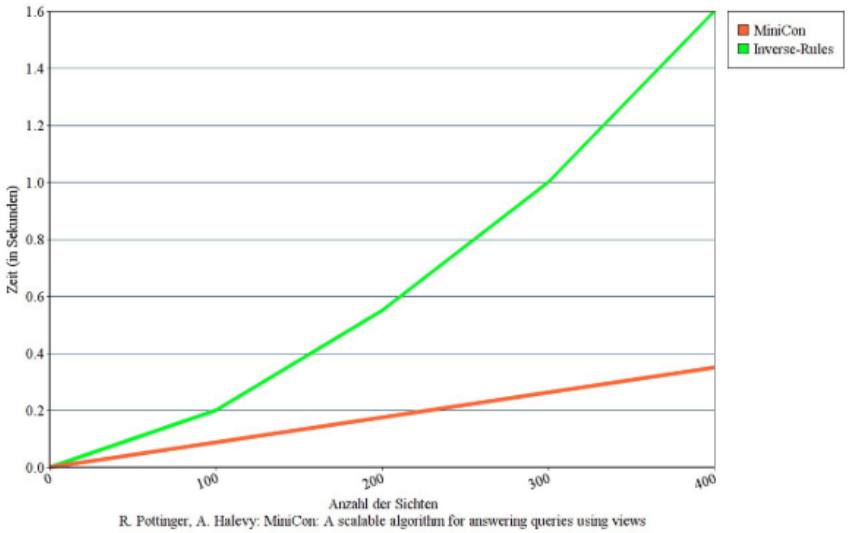

Abbildung 3.2: Anfragen mit zwei unterschiedlichen Variablen und zwölf Teilzielen. Die Sichten besitzen zwischen zwei und vier Teilziele.

an. Dafür muss aber angemerkt werden, dass die gesuchten Sichten in der Abbildung 3.1 identisch zur Anfrage sein müssen um als relevante Sicht in frage zu kommen. In der Abbildung 3.3 ist im direkten Vergleich zum Bucket-Algorithmus zu erkennen, dass die anderen beiden Algorithmen bei Anfragen mit mehreren unterschiedlichen Variablen besser skalieren. Zusätzlich kann auch hier wieder erkannt werden, dass der MiniCon-Algorithmus besser skaliert als der Inverse-Rules-Algorithmus. Laut [PH01] ist der MiniCon-Algorithmus zwischen 10% und 25% schneller als der Inverse-Rules-Algorithmus. Diese größere Differenz entsteht durch die unterschiedliche Anzahl der möglichen Rewritings einer Anfrage.

3.1.4 Zusammenfassung

Das Ergebnis der Benchmark-Tests ist, dass der MiniCon-Algorithmus den anderen beiden Algorithmen in jedem Punkt überlegen ist. Beim genaueren Betrachten der Erklärungen zu diesen Algorithmen (siehe Kapitel 2), hätte die Entscheidung auch ohne die Tests auf den MiniCon-Algorithmus fallen können, da dieser als eine Verbesserung des Bucket-Algorithmus vorgestellt wurde und daher effizienter als dieser arbeitet. Auch der Inverse-Rules-Algorithmus wäre nicht geeignet, da er nur funktioniert, wenn eine Lösung existiert und kann durch den fehlenden Test auf Query Containment auch nicht garantieren, dass das Rewriting äquivalent zur originalen Anfrage ist.
Da in den praktischen Einsätzen nicht garantiert werden kann, dass eine Lösung existiert, wäre der Inverse-Rules-Algorithmus nicht zuverlässig genug. Aus diesem Grund habe ich mich für die Umsetzung des MiniCon-Algorithmus in Java entschieden.

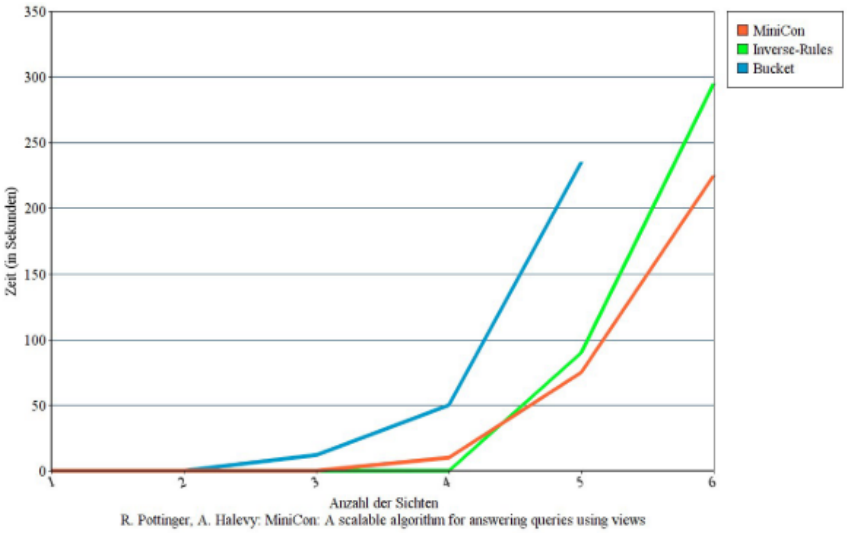

Abbildung 3.3: Anfragen mit zehn Teilzielen und komplett unterschiedlichen Variablen.

Kapitel 4

Implementierung

Wie im Konzept bereits analysiert wurde, wird nun in diesem Kapitel der beste Query-Rewriting-Algorithmus, der MiniCon-Algorithmus, implementiert und die Vorgehensweise dokumentiert. Die verwendete Programmiersprache ist Java in der Version 1.8 mit den Erweiterungen JSqlParser[1] und JDBC zusammen mit dem PostgreSQL-Treiber[2]. Als Abfragesprache wird SQL eingesetzt, womit Anfragen an Datenbanken auf einem PostgreSQL-Server gestellt werden können. Aufgrund der Art der Implementierung kann ohne Veränderungen im Quellcode kein anderes Datenbanksystem verwendet werden, weil dort PostgreSQL spezifische Anfragen gestellt werden, um die vorhandenen Sichten für eine Datenbank zu bekommen (genauere Informationen dazu im Abschnitt 4.2.3).
Die genutzte PostgreSQL-Version ist 9.2.18. Eingesetzt wird sie auf einer Linux-VM mit dem Betriebssystem CentOS in der Version 7.3, welche 4 GB Hauptspeicher besitzt. Der Zugriff auf diesen Server erfolgt von einem externen Rechner über das Informatik-VPN der Universität Rostock.
Zum Testen des Algorithmus und der Beispiele wurde eine Musik-Datenbank gewählt, welche als Sternschema angelegt ist. Eine grafische Darstellung der Datenbank befindet sich in der Abbildung 4.3.
Zusätzlich muss zu dieser Implementierung noch erwähnt werden, dass sie nicht komplett den originalen MiniCon-Algorithmus umsetzt, sondern eine für die Anwendung angepasste Version. Der originale MiniCon-Algorithmus, wie er im Artikel [PH01] vorgestellt wird, erstellt eine Menge von Rewritings zu einer Anfrage und der Menge von Sichten. Da diese Umsetzung jedoch im PArADISE-Framework praktisch eingesetzt werden soll, um ein effizientes Rewriting einer Anfrage zu finden, genügt es ein möglichst gutes Rewriting einer Anfrage zu finden, welches am besten alle Teilziele mit Sichten abdeckt und somit schneller ausgeführt werden kann.

[1] Version 1.1 `https://github.com/JSQLParser/JSqlParser`
[2] Version 42.1.3 `https://jdbc.postgresql.org/`

4.1 Vorbereitung

Damit die hier vorgestellte MiniCon-Implementierung ordnungsgemäß verwendet werden kann, müssen bestimmte Voraussetzungen an die gestellte Anfrage und an die Sichten auf der Datenbank erfüllt sein.

1. Bei den zu selektierenden Attributen muss in einer Anfrage mit Verbunden der Relationsname zu jedem Attribut mit angegeben werden. Beispiel: "SELECT **artists.**id, **albums.**image FROM albums JOIN artists ON **albums.**artist = **artists.**id".

2. Die gestellte Anfrage muss eine einfache SPJ-Anfrage in der SQL-Notation sein, wobei nur Vergleiche auf Äquivalenz im WHERE-Teil erlaubt sind.

3. Wenn als Ergebnis ein Verbund zwischen verschiedenen Relationen gesucht wird, darf bei einem sogenannten CROSS-JOIN keine Verbundbedingung geschrieben werden. Dafür sollte eine einfache JOIN-Klausel verwendet werden, wie es im folgenden Beispiel einmal veranschaulicht wird:

   - Falsch: "SELECT ... FROM a, b WHERE a.x = b.x"
   - Richtig: "SELECT ... FROM a JOIN b ON a.x = b.x"

4. Alle Sichten auf der Datenbank müssen in ihrer Definition komplett klein benannt werden und dürfen keine kompletten Relationennamen enthalten. Beispiel: Auf einer Datenbank existiert eine Relation *rel1*, dann darf der Name der Sicht nicht den gesamten String *"rel1"* enthalten.

5. Als Datenbankmanagementsystem funktioniert ausschließlich PostgreSQL.

6. Zusatz: Die Verwendung von Aliassen im SELECT-Teil der Anfragen ist erlaubt.

Wenn diese Bedingungen erfüllt sind, kann die Anfrage an das Java-Programm übergeben werden. Dort wird dann mit Hilfe dieser Anfrage und einer vorher definierten *JDBC-Connection* ein neues *MiniCon-Objekt* (2.1.4) erzeugt. Dabei wird innerhalb dieses MiniCon Objektes ein neues *Query-Objekt* (4.2.2) erstellt, in welchem die gegebene Anfrage genauer analysiert und für die spätere Bearbeitung vorbereitet wird.

4.2 Die Klassen

4.2.1 Klasse: MiniCon

Die *MiniCon-Klasse* stellt den Rahmen für das gesamte Programm dar. In ihr werden die anderen Objekte erzeugt und das gesamte Rewriting der Anfragen gesteuert. Für die Erzeugung einer neuen Instanz dieser Klasse werden eine *JDBC-Connection* und eine SQL-Anfrage als String benötigt. Mit Hilfe dieser beiden Objekte wird dann ein neues Query-Objekt erzeugt.

4.2.2 Klasse: Query

Die *Query-Klasse* repräsentiert als Objekt eine geparste Version der gegebenen SQL-Anfrage. Sie beinhaltet unter anderem jeweils eine Liste der selektierten Attribute, der beteiligten Relationen sowie Angaben

über die optionalen WHERE- und LIMIT-Klauseln in einer SQL-Anfrage. Weiterhin werden die Verbunde genauer analysiert, indem *Join-Objekte* (4.2.5) für jeden vorhandenen Verbund in der Anfrage angelegt werden. Zusätzlich wird zur Veranschaulichung auch eine Datalog-Notation der Anfrage erstellt, welche während der Erzeugung des Objektes in der Konsole ausgegeben wird, wie in der Abbildung 4.1 kurz vor dem unteren Ende zu sehen ist.

```
-----------------------------------
     Bearbeitung der Query ...
-----------------------------------
Anfrage = SELECT artists.name, albums.name FROM artists JOIN albums ON albums.artist = artists.id LIMIT 10

SELECT-Items:
artists.name
albums.name

FROM-Items:
artists
albums

Die Anfrage enthält keine WHERE-Klausel.
LIMIT = LIMIT 10

Somit gefundene Teilziele:
SELECT name FROM artists
SELECT name FROM albums

In der Anfrage sind folgende Joins enthalten:
artists JOIN albums ON albums.artist = artists.id

DATALOG: Query(artists.name, albums.name) :- artists(name, id), albums(name, artist)

-----------------------------------
Bearbeitung der Query abgeschlossen
-----------------------------------
```

Abbildung 4.1: Ausgabe auf der Konsole nachdem eine Anfrage analysiert wurde.

4.2.3 Klasse: View

Wenn für das Erstellen eines Rewritings in der *MiniCon-Klasse* die Methode *getViews()* aufgerufen wird, werden über die Metadaten der Datenbank die Namen aller Sichten abgefragt und in einem *JDBC-ResultSet* gespeichert. Durch diese Ergebnismenge wird dann in der Methode *getViewDefinition(ResultSet)* iteriert und für jede Sicht die Definition vom Datenbanksystem abgefragt. PostgreSQL besitzt für diese Art der Anfrage eine eigene Funktion *"pg_get_viewdefinition"*, welche an die Datenbank gestellt werden kann und als Ergebnis die Definition der Sicht in der SQL-Notation liefert. Durch die Verwendung dieser Funktion, kann diese Implementierung ohne weitere Veränderungen nur auf einer PostgreSQL-Datenbank arbeiten. Wenn die Sichtdefinition vorliegt, wird sie ähnlich wie eine einfache

Anfrage geparst und auch als ein eigenes Objekt angelegt. Am Ende der Analyse jeder Sicht wird auf der Konsole jeweils eine kleine Zusammenfassung der Informationen ausgegeben, wie es in der Abbildung 4.2 zu sehen ist.

```
Subgoals der View: artnamejoin
------------------
SELECT name FROM artists
+++++++JOINS+++++++
artists JOIN albums ON albums.artist = artists.id

DATALOG: artnamejoin(artists.name) :- artists(name, id), albums(artist)
------------------
```

Abbildung 4.2: Ausgabe auf der Konsole nachdem eine Sicht analysiert wurde.

4.2.4 Klasse: MCD

Diese Klasse repräsentiert die *MiniCon-Descriptions*, wie sie in der Definition 2 formal definiert sind. Der einzige Unterschied ist, dass hierbei auf die internen Abbildungen $V(\bar{Y})_C$ verzichtet wird, weil sie für diese Implementierung nicht benötigt wurden. Die MCD-Objekte werden in der *MiniCon-Klasse* (2.1.4) erstellt, während die verschiedenen Sichten überprüft werden, ob diese auch die MiniCon-Eigenschaften erfüllen.

4.2.5 Weitere Klassen

Klasse: DatalogNotation Die *DatalogNotation-Klasse* dient zur Generierung und Speicherung der Datalog-Notation aus SQL-Anfragen. Als Ausgabe liefert sie die SQL-Anfrage in der Datalog-Notation, jedoch ohne die WHERE-Bedingungen zu betrachten.

Klasse: Join Die *Join-Klasse* stellt Verbunde aus der Anfrage, sowie aus den Sichtdefinitionen als Objekt dar. Durch die Nutzung eines eigenen Objektes für einen Verbund, können die Verbunde aus den Anfragen und den Sichten einfacher miteinander verglichen werden.

Unterklasse: AddedJoin Diese Unterklasse zum *Join* wird bei der Kombination der MCDs (4.3.4) verwendet. Wenn bei dem Iterieren durch die *Mappings* (4.2.5) der MCDs ein Teilziel der Anfrage durch ein Teilziel der Sicht ersetzt werden kann, wodurch eine weitere Relation in die Menge der selektierten Relationen aufgenommen werden muss, und die Relation, aus welcher das originale Teilziel ursprünglich selektiert wurde, noch weitere Teilziele in der Anfrage besitzt, dann wird an die bestehende FROM-Klausel ein neuer Verbund angehängt, bei welchem die neue Sicht mit der originalen Relation verbunden werden soll. Wenn im weiteren Verlauf des Algorithmus eine Relation nicht mehr benötigt wird, kann der Verbund mit dieser Relation einfach wieder gelöscht werden.

Klasse: Mapping Die *Mapping-Klasse* dient als Abbildung von einem Teilziel der Anfrage auf ein Teilziel einer Sicht. Die *Mapping*-Objekte werden nur in der *MCD-Klasse* (4.2.4) verwendet, um die Ab-

bildungen von Teilzielen darzustellen. Diese Klasse besitzt daher nur drei Informationen. Zum einen wird der String des originalen Teilziels gespeichert, sowie der String des Teilziels der Sicht. Um das *Mapping* genauer zuordnen zu können, wird außerdem noch eine Teilziel-ID gespeichert, welche die Nummer des Teilziels aus der Anfrage repräsentiert.

Klasse: SelectItem Ein *SelectItem* ist eine genauer analysierte Version eines selektierten Attributes in einer SQL-Anfrage. In dieser Klasse werden alle Informationen zu den selektierten Attributen gespeichert. Dazu gehören der Attributname, der Relationenname, von welchem das Attribut selektiert wird, sowie eine Angabe über ein eventuell vorhandenen Alias zu dem Attribut.

Klasse: Subgoal Diese Klasse dient zur Repräsentation von Teilzielen einer Anfrage oder von Sichten. Dafür wird ein selektiertes Attribut, mittels *SelectItem* (4.2.5), einer Relation zugeordnet.

Unterklasse: SubgoalSelectList Diese Unterklasse speichert anstatt einem Attribut und einer Relation eine Liste von Attributen, welche aus einer Relation selektiert werden.

4.3 Wichtige Funktionen

Die hier vorgestellten Funktionen gehören zum MiniCon-Algorithmus. Sie sind alle ein Teil der MiniCon-Klasse (2.1.4). Um ein Rewriting der Anfrage mittels des MiniCon-Algorithmus zu finden, wird zuerst die *formingMCDs-Funktion* aufgerufen, um danach die erstellte Liste der MCDs an die Funktion *combiningMCD* zu übergeben, welche als Ergebnis eine fertige Umschreibung der Anfrage auf Sichten zurückgibt. Wenn sich jedoch herausstellt, dass keine MCDs erstellt werden konnten, wird die originale Anfrage zurückgegeben, mit dem Hinweis, dass kein Rewriting dieser gefunden werden konnte.

4.3.1 checkFirstMiniConProperty(View)

Für die Erzeugung einer MiniCon Description (siehe Definition 2) müssen zwei Bedingungen erfüllt sein. Die erste Bedingung bedeutet so viel wie: *Die gefundene Sicht enthält im Kopf mindestens eine Kopfvariable der Anfrage.* Diese Bedingung wird jeweils für eine gegebene Sicht getestet. Der dafür verwendete Algorithmus steht als Pseudocode im Algorithmus 1.

Algorithmus 1 : Prüfung der ersten Bedingung für eine MCD.

```
Input : View v, (Query q)
Output : Boolean, hat diese Sicht mindestens eine gemeinsame Kopfvariable mit der Anfrage
for Kopfvariable k von q do
    if Kopfvariablen von v enthalten k then
        return true;
    end
end
return false;
```

4.3.2 checkSecondMiniConProperty(View)

Wenn die erste Bedingung an eine Sicht zum Erstellen einer MiniCon Description erfüllt ist, wird diese Sicht auch auf die zweite Bedingung getestet. Diese lautet grob: *Wenn eine Variable der Anfrage in einem Verbund als Prädikat vorkommt, wobei dieser Verbund nicht in der Sicht vorhanden ist, dann muss diese Variable im Kopf der Sicht stehen.* Der für diesen Test verwendete Algorithmus steht als Pseudocode in einer verkürzten Version in Algorithmus 2 beschrieben.

Algorithmus 2 : Prüfung der zweiten Bedingung für eine MCD.

```
Input : View v, (Query q)
Output : Boolean, funktionieren die Verbunde der Anfrage mit dieser Sicht weiterhin
if q hat Verbunde then
    if v hat dieselben Verbunde wie q then
        return true;
    end
    → lösche alle Verbunde, die gleichzeitig in q und v vorkommen
    if Variablen der restlichen Verbundbedingungen aus q stehen im Kopf von v then
        return true;
    else
        return false;
    end
else
    if v hat keine Verbunde then
        return true;
    else
        return false;
    end
end
```

4.3.3 formingMCDs(List<View>)

In dieser Funktion werden die MiniCon Descriptions anhand einer gegebenen Menge von Sichten auf einer Datenbank erstellt. Die Bedingungen, welche erfüllt sein müssen um eine MCD zu erstellen, werden in den Funktionen *checkFirstMiniConProperty(View)* und *checkSecondMiniConProperty(View)* getestet. Wie dieser Algorithmus in etwa abläuft steht als Pseudocode in Algorithmus 3 beschrieben.

4.3.4 combiningMCD(List<MCD>)

Am Ende müssen die erstellten MCDs für ein Rewriting der Anfrage noch kombiniert werden. Der dazu verwendete Algorithmus befindet sich im Anhang. Als Ergebnis wird hier ein Rewriting zurück gegeben, bei dem möglichst viele Teilziele durch Sichten ersetzt wurden.

Algorithmus 3 : Erstellung der MCDs anhand einer gegebenen Menge von Sichten.

```
Input : List<View> vList, Query q
Output : List<MCD> listOfMCDs
listOfMCDs = new List<MCD>;
listOfMappings = new List<Mapping>;
listOfSubgoals = new List<Integer>;
for View v ∈ vList do
    if checkFirstMiniConProperty(v) & checkSecondMiniConProperty(v) then
        erstelle neue Objekte der oben erzeugten Listen;
        for Subgoal sq aus q do
            for Subgoal sv aus v do
                if sq == sv then
                    erstelle ein neues Mapping von sq nach sv und füge es zur listOfMappings hinzu;
                    füge die ID des Subgoals in die listOfSubgoals ein;
                end
            end
        end
        erstelle eine MCD aus v, listOfMappings und listOfSubgoals und füge sie der listOfMCDs
         hinzu;
    end
end
return listOfMCDs;
```

4.4 Testfälle

In diesem Abschnitt werden entwickelte Testfälle für den MiniCon-Algorithmus vorgestellt. Die Beispielanfragen und -sichten wurden so gewählt, dass möglichst viele Fälle abgedeckt werden können.

Als Datenbank für diese Testfälle verwenden wir eine Musik-Datenbank, welche von Mitarbeitern des Lehrstuhls für Datenbank- und Informationssysteme der Universität Rostock[3] zusammengestellt wurde. Das Datenbankschema ist in der Abbildung 4.3 dargestellt.

Auf dieser Datenbank wurden folgende Sichten erstellt.

- **albview** - *SELECT albums.id, albums.name, albums.artist, albums.image FROM albums;*
- **aliasv** - *SELECT artists.name AS artname, albums.id, albums.name AS albname, albums.artist FROM artists JOIN albums ON albums.artist = artists.id;*
- **artnamejoin** - *SELECT artists.name FROM artists JOIN albums ON albums.artist = artists.id;*
- **artview** - *SELECT artists.id, artists.name FROM artists;*
- **irrelevantview** - *SELECT images.id, images.path FROM images;*
- **containedset** - *SELECT composers.id, composers.name FROM composers WHERE id < 500;*

[3] `https://dbis.informatik.uni-rostock.de/`

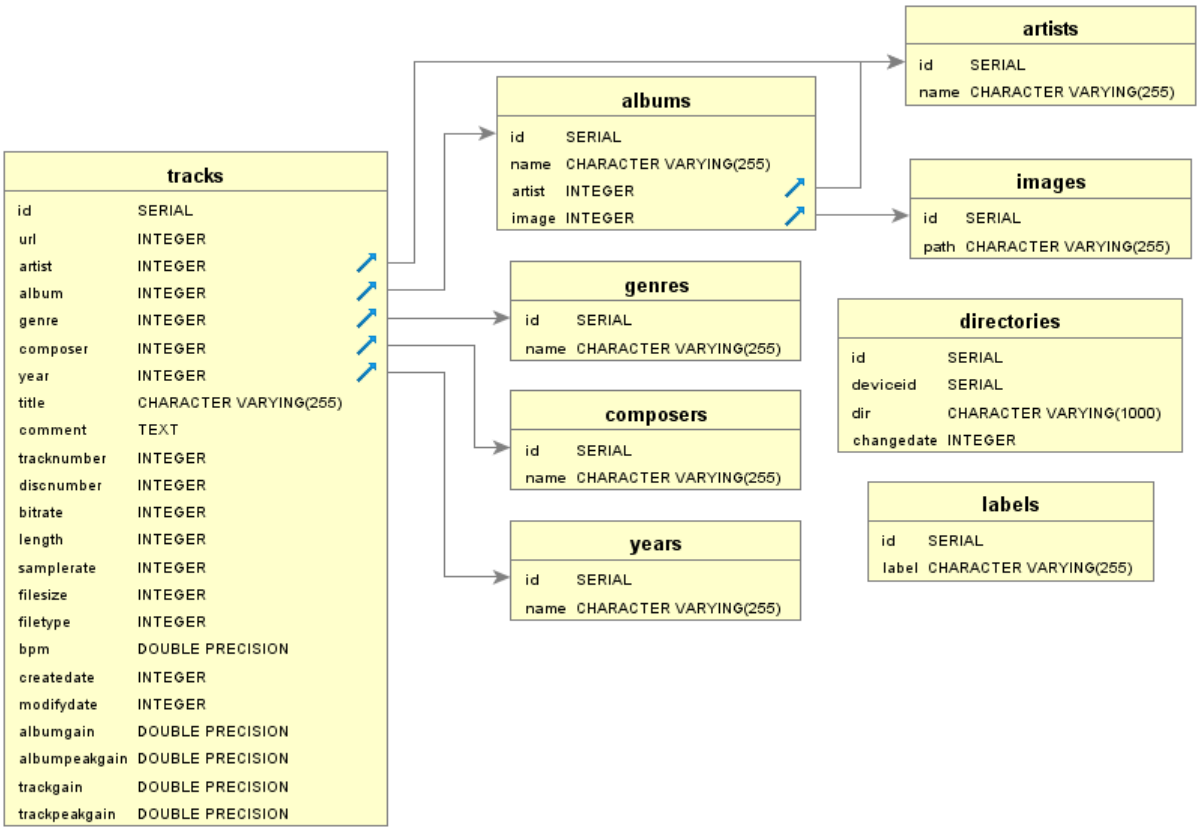

Abbildung 4.3: Grafische Darstellung der verwendeten Beispieldatenbank.

Diese Sichten wurden gewählt, weil damit ziemlich alle Fälle abgedeckt werden können. Mit den Sichten *"albview"* und *"artview"* wurden zwei komplette Relationen als Sichten angelegt. In der Sicht *"aliasv"* wurde in der Sicht ein Attribut umbenannt und es wurde ein Verbund bei der Erzeugung durchgeführt. In der Sicht *"artnamejoin"* wird auch der selbe Verbund verwendet, wie auch in der *"aliasv"*, jedoch wird hier nur ein Attribut selektiert, womit maximal ein Teilziel einer Anfrage abgedeckt werden kann. Zusätzlich wurde die Sicht *"irrelevantview"* erstellt, welche in keiner der weiter unten vorgestellten Beispielanfragen eine Anwendung findet.
Für diese Sichten wurden dann folgende Beispielanfragen entwickelt und mit dem implementierten MiniCon-Algorithmus erfolgreich getestet:

1. *SELECT id, name FROM artists;*
2. *SELECT artists.name AS artName, albums.id, albums.name AS albName, albums.artist, albums.image FROM artists, albums LIMIT 10;*
3. *SELECT artists.name, albums.id AS album, albums.artist AS artistsID FROM artists JOIN albums ON albums.artist=artists.id;*
4. *SELECT * FROM artists JOIN albums ON albums.artist=artists.id;*

5. *SELECT albums.name, albums.id FROM albums WHERE albums.name='Kingdom';*
6. *SELECT * FROM years;*
7. *SELECT * FROM composers;*

Die erste Beispielanfrage ist eine einfache Abfrage aller Attribute der Relation *"artists"*. Solche Arten von Anfragen funktionieren mit dem Algorithmus ohne Probleme. Da passend zu dieser Anfrage die Sicht *"artview"* existiert, kann sie komplett durch eine Anfrage an die Sicht ersetzt werden. Dementsprechend wird das Ergebnis der Anfrage komplett aus der *"artview"*-Sicht selektiert: *"SELECT artview.id, artview.name FROM artview"*.

Die zweite Anfrage besitzt gleich zwei Besonderheiten. Zum einen besitzen zwei selektierten Attribute einen Alias, der so auch im Rewriting wieder zu finden sein soll und zum anderen soll das Ergebnis ein CROSS-Join sein, zwischen den Relationen *"artists"* und *"albums"*. Zusätzlich hat diese Anfrage auch eine LIMIT-Klausel, welche aber nur optional ist und das Rewriting der Anfrage nicht beeinflusst. Damit der Algorithmus besser mit den Verbunden arbeiten kann, wird beim Parsen der Anfrage zuerst das Komma im FROM-Teil durch "CROSS JOIN" ersetzt, was semantisch äquivalent zum Komma ist. Das gefundene Rewriting dieser Anfrage lautet folgendermaßen: *"SELECT aliasv.artname AS artName, albview.id, albview.name AS albName, albview.artist, albview.image FROM aliasv CROSS JOIN albview LIMIT 10"*. Es setzt sich wie die originale Anfrage aus einem CROSS-Join zwischen zwei Relationen zusammen und erhält die Aliasse und die LIMIT-Klausel. Die einzelnen Teilziele werden außerdem alle von Sichten abgedeckt.

Die dritte Anfrage ist, ähnlich wie schon die erste, äquivalent zu einer Sichtdefinition. Das gefundene Rewriting dieser Abfrage sieht wie folgt aus: *"SELECT aliasv.artname, aliasv.id AS album, aliasv.artist AS artistsID FROM aliasv"*. Die Besonderheit bei diesem Rewriting ist, dass die Anfrage einen Verbund besaß, welcher genau so auch in der Sichtdefinition vorkommt. Daher muss beim Rewriting dieser Verbund nicht mehr beachtet werden und es genügt, das Ergebnis von der Sicht abzufragen.

Das vierte Beispiel ist eine Anfrage, in welcher alle Attribute von einem Verbund zwischen den Relationen *"albums"* und *"artists"* abgefragt werden. Damit die Teilziele in dieser Anfrage richtig bestimmt werden können, wird bei der Analyse dieser Anfrage der *' * '* aufgelöst und an seiner Stelle werden alle Attribute der selektierten Relationen eingefügt. Mit diesem neuen String wird dann im weiteren Verlauf gearbeitet. Das gefundene Rewriting dieser Anfrage sieht wie folgt aus: *"SELECT artview.id, artview.name, albview.id, albview.name, albview.artist, albview.image FROM artview JOIN albview ON albview.artist=artview.id"*. Wie zu erkennen ist, konnte diese Anfrage auch wieder komplett auf die Sichten umgeschrieben werden. Außerdem wurde auch der Verbund dementsprechend angepasst.

Die fünfte Anfrage hat als Besonderheit einen Filter nach dem Namen des Albums *"Kingdom"*. Wenn so eine Anfrage gestellt wird, muss auch die Bedingung in der WHERE-Klausel angepasst werden. Das hier gebildete Rewriting lautet: *"SELECT albview.name, albview.id FROM albview WHERE albview.name = 'Kingdom"'*. Dabei wurde die Anfrage komplett auf die *"albview"*-Sicht umgeschrieben und auch die Bedingung daran angepasst.

Für die sechste Anfrage existiert keine passende Sicht. Daher kann diese Anfrage nicht umgeschrieben werden. Als Resultat liefert die MiniCon-Klasse die originale Anfrage zurück und gibt auf der Konsole die Fehlermeldung aus, dass kein Rewriting gefunden werden konnte.

Die letzte Anfrage ist beinahe identisch zu der Sicht "containedset", nur dass hier in der Anfrage nicht gefiltert wird. Als Rewriting liefert der Algorithmus hier: *"SELECT containedset.id, containedset.name FROM containedset"*. Da in der verwendeten Sicht nur id's auftreten die kleiner als 500 sind, liefert das Rewriting dieser Anfrage nur ein "maximally contained set", welches nicht vollständig der originalen Anfrage entspricht.

Kapitel 5

Zusammenfassung und Ausblick

5.1 Zusammenfassung

Das Ziel dieser Arbeit war es, bestehende Verfahren zum Testen des Query Containments zu analysieren und zu vergleichen. Dazu wurden zu Beginn die grundlegenden Techniken dafür vorgestellt, bevor drei Verfahren im Detail, mit jeweils einem mitlaufenden Beispiel, erklärt wurden. Dabei stellte sich heraus, dass der MiniCon-Algorithmus auf dem des Buckets basierte und diesen durch Anpassungen effizienter gemacht hat. Diese beiden Algorithmen sind jedoch auf Anfragen beschränkt, die nur Selektionen, Projektionen und Verbunde verwenden dürfen und in der konjunktiven Normalform vorliegen müssen. Das dritte Verfahren, der Inverse-Rules-Algorithmus, hatte einen anderen Ansatz, wodurch er auch für andere Arten von Anfragen, wie beispielsweise rekursive Anfragen, genutzt werden könnte. Wie aber schon die Entwickler dieses Algorithmus feststellten, kann er auf Anfragen nur eine Antwort liefern, wenn mindestens eine Lösung existiert. Gibt es für eine Anfrage jedoch gar keine Lösung, so würde der Algorithmus nicht terminieren.

Nach der Theorie wurden die drei Algorithmen in der Praxis miteinander verglichen. Dazu wurden bereits durchgeführte Tests aus einer veröffentlichten Arbeit ausgewertet. Sie wurden hier anhand ihrer Funktionsweise, Komplexität und der Möglichkeit, sie im PArADISE-Framework einzusetzen, verglichen. Dabei stellte sich heraus, dass der MiniCon-Algorithmus am effizientesten arbeitet. Dementsprechend wurde er anschließend in Java implementiert und getestet.

Die Implementierung wurde fertiggestellt und auch auf möglichst viele unterschiedliche Fälle hin überprüft. Da die Anfragen in SQL auf verschiedene Art und Weise formuliert werden können, mussten Einschränkungen an die gestellten Anfragen und die Sichten gemacht werden. Zwar wurden dadurch schon sehr viele mögliche Fälle abgedeckt, jedoch kann es immer noch passieren, dass das Programm in speziellen Sonderfällen nicht korrekt arbeitet. Aufgrund der begrenzten Zeit konnte ich wahrscheinlich nicht alle dieser möglichen Sonderfälle identifizieren und abfangen.

5.2 Ausblick

Neben den in dieser Arbeit vorgestellten Techniken für das AQuV-Problem gibt es seit einiger Zeit auch ein CHASE-BACKCHASE-Verfahren für dieses Problem, seit kurzer Zeit auch eine sehr effiziente Version dieses Verfahrens [DH13].

Der CHASE ist ein Universalwerkzeug der Datenbanktheorie, dass bestimmte Informationen über Datenbanken (den CHASE-Parameter) in andere Datenbankstrukturen (das CHASE-Objekt) einarbeiten kann. So gibt es beispielsweise Varianten für Probleme der Datenintegration, der Schemaevolution (allgemein Schemaabbildungen) und des Provenance Managements, bei dem mit dem CHASE Integritätsbedingungen oder Anfragen (als CHASE-Parameter) auf Datenbanken (als CHASE-Objekt) angewendet werden, um eine Zieldatenbank aus Quelldatenbanken zu erstellen. Weitere Varianten des CHASE arbeiten Integritätsbedingungen oder Sichten (als CHASE-Parameter) in Anfragen (als CHASE-Objekt) ein, um eine Zielanfrage zu erhalten, die entweder die Integritätsbedingungen oder die Sichten berücksichtigt. Im Fall der Einarbeitung von Sichten in eine Anfrage können wir eine bestehende Anfrage um anwendbare Sichten anreichern.

Die korrespondierende BACKCHASE-Phase realisiert in allen Fällen eine passende Rückabbildung (oder inverse Abbildung), die wiederum diversen Zwecken dienen kann. In unserem AQuV-Fall berechnet der BACKCHASE aus der um Sichten erweiterten Anfrage eine äquivalente Anfrage, die ausschließlich aus Sichten besteht (falls diese existiert).

Diese Technik wurde in dieser Arbeit nicht weiter berücksichtigt, weil die zugehörige Theorie des CHASE erst im Master-Studium behandelt wird. Außerdem liefen parallel zu dieser Bachelorarbeit zwei Masterarbeiten, welche die CHASE-BACKCHASE-Technik berücksichtigt oder sogar verwendet haben. Tanja Auge [Aug17] hat für den ersten Fall der Schemaabbildungen und des Provenance Management ein CHASE-BACKCHASE-Verfahren ausführlich vorgestellt und weiterentwickelt. Christian Langmacher [Lan17] hat für das hier behandelte AQuV-Problem das CHASE-BACKCHASE-Verfahren im Vergleich mit anderen Verfahren vorgestellt und dann ein konkurrierendes Verfahren für das in dieser Arbeit zu Beginn erwähnte AQuO-Problem weiterentwickelt. Auch aus diesen Gründen haben wir in dieser Arbeit auf eine Vorstellung der CHASE-BACKCHASE-Variante verzichtet[1].

[1] Mitteilung von Prof. Dr. Andreas Heuer vom 04.09.2017

Kapitel 6

Aufbau des Datenträgers

Auf dem beiliegenden Datenträger befinden sich ergänzende Materialien. Die Struktur und der Inhalt werden im Folgenden kurz erläutert. Die Überschriften entsprechen den Ordnern im Wurzelverzeichnis des Datenträgers.

6.1 Aufbau

Quellen Die im Literaturverzeichnis aufgelisteten Quellen, mit Ausnahme digital nicht verfügbarer Werke, werden hier aufgelistet. Es wurden ebenso alle Webseiten, welche innerhalb der Arbeit zitiert wurden, als Portable Dokument Format (*.pdf) exportiert und bereitgestellt. Der Dateiname der Quellen entspricht dem Kürzel im Literaturverzeichnis. Hinweis: Die beigefügte Literatur dient der Nachvollziehbarkeit von Aussagen und darf nicht öffentlich bereitgestellt werden. Dies gilt speziell für Werke, welche durch Lizenzverträge der Universität Rostock durch die Universitätsbibliothek zur Verfügung gestellt wurden.

PArADISE

- Aktueller Stand des Prototypen zum Zeitpunkt des Drucks dieser Arbeit[1]
- Ausführbare Version des Prototypen.
- JavaDoc des Prototypen

[1] aktuelle Fassung unter `https://git.informatik.uni-rostock.de/hg/PArADISE`

6.2 Datenträger

Literaturverzeichnis

[Aug17] Auge, Tanja: *Umsetzung von Provenance-Anfragen in Big-Data-Analytics-Umgebungen.* Masterarbeit, Universität Rostock, 2017. Zur Zeit der Niederschrift dieser Arbeit noch nicht veröffentlicht.

[Deu49] Deutschland, Bundesrepublik: *Grundgesetz der Bundesrepublik Deutschland in der Fassung vom 21. Juli 2010, das zuletzt durch Artikel 1 des Gesetzes vom 23. Dezember 2014 geändert worden ist.* 1949.

[Deu17] Deutschland, Bundesrepublik: *Bundesdatenschutzgesetz in der Fassung der Bekanntmachung vom 14. Januar 2003, das zuletzt durch Artikel 7 des Gesetzes vom 30. Juni 2017 geändert worden ist.* 2017.

[DG97a] Duschka, Oliver M. und Michael R. Genesereth: *Answering Recursive Queries Using Views.* In: Mendelzon, Alberto O. und Z. Meral Özsoyoglu (Herausgeber): *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona, USA*, Seiten 109–116. ACM Press, 1997.

[DG97b] Duschka, Oliver M. und Michael R. Genesereth: *Query Planning in Infomaster.* In: Bryant, Barrett R., Janice H. Carroll, Dave Oppenheim, Jim Hightower und K. M. George (Herausgeber): *Proceedings of the 1997 ACM symposium on Applied Computing, SAC'97, San Jose, CA, USA, February 28 - March 1*, Seiten 109–111. ACM, 1997.

[DH13] Deutsch, Alin und Richard Hull: *Provenance-Directed Chase&Backchase.* In: Tannen, Val, Limsoon Wong, Leonid Libkin, Wenfei Fan, Wang-Chiew Tan und Michael P. Fourman (Herausgeber): *In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman*, Band 8000 der Reihe *Lecture Notes in Computer Science*, Seiten 227–236. Springer, 2013.

[DHI12] Doan, AnHai, Alon Y. Halevy und Zachary G. Ives: *Principles of Data Integration.* Morgan Kaufmann, 2012.

[FKMP03] Fagin, Ronald, Phokion G. Kolaitis, Renée J. Miller und Lucian Popa: *Data Exchange: Semantics and Query Answering.* In: Calvanese, Diego, Maurizio Lenzerini

und Rajeev Motwani (Herausgeber): *ICDT 2003, 9th International Conference on Database Theory, Siena, Italy, January 8-10, 2003, Proceedings*, Band 2572 der Reihe *Lecture Notes in Computer Science*, Seiten 207–224. Springer, 2003.

[FKPT11] Fagin, Ronald, Phokion G. Kolaitis, Lucian Popa und Wang Chiew Tan: *Schema Mapping Evolution Through Composition and Inversion.* In: Bellahsene, Zohra, Angela Bonifati und Erhard Rahm (Herausgeber): *Schema Matching and Mapping*, Data-Centric Systems and Applications, Seiten 191–222. Springer, 2011.

[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.

[Gru17] Grunert, Hannes: *Taschenspielertricks beim Query Containment.* interner Seminarvortrag am Lehrstuhl für Datenbanken- und Informationssysteme Universität Rostock, Januar 2017.

[Hal01] Halevy, Alon Y.: *Answering queries using views: A survey.* VLDB Journal, 10(4):270–294, 2001.

[Heu17] Heuer, Andreas: *Theorie relationaler Datenbanken.* Vorlesungsskript, Sommersemester 2017.

[Köl10] Köln, Fachhochschule: *Datenbanken online Lexikon: Materialisierte Sicht.*, 2010. `http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/Materialisierte-Sicht`, Accessed: 2017-05-22.

[Lan17] Langmacher, Christian: *Vertikale Verteilung von SQL-Anfragen auf DBMS-Servern abnehmender Leistungsfähigkeit.* Masterarbeit, Universität Rostock, 2017. Zur Zeit der Niederschrift dieser Arbeit noch nicht veröffentlicht.

[LFS97] Levy, Alon Y., Richard Fikes und Yehoshua Sagiv: *Speeding up Inferences Using Relevance Reasoning: A Formalism and Algorithms.* Artificial Intelligence, 97(1-2):83–136, 1997.

[LLZM12] Li, Tiancheng, Ninghui Li, Jian Zhang und Ian Molloy: *Slicing: A New Approach for Privacy Preserving Data Publishing.* IEEE Transactions on Knowledge and Data Engineering, 24(3):561–574, 2012.

[LMSS95] Levy, Alon Y., Alberto O. Mendelzon, Yehoshua Sagiv und Divesh Srivastava: *Answering Queries Using Views.* In: Yannakakis, Mihalis (Herausgeber): *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California, USA*, Seiten 95–104. ACM Press, 1995.

[LRO96a] Levy, Alon Y., Anand Rajaraman und Joann J. Ordille: *Query-Answering Algorithms for Information Agents.* In: Clancey, William J. und Daniel S. Weld (Herausgeber): *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, August 4-8, 1996, Volume 1.*, Seiten 40–47. AAAI Press / The MIT Press, 1996.

[LRO96b] Levy, Alon Y., Anand Rajaraman und Joann J. Ordille: *Querying Heterogeneous Information Sources Using Source Descriptions.* In: Vijayaraman, T. M., Alejandro P. Buchmann, C. Mohan und Nandlal L. Sarda (Herausgeber): *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, Seiten 251–262. Morgan Kaufmann, 1996.

[PH01] Pottinger, Rachel und Alon Y. Halevy: *MiniCon: A Scalable Algorithm for Answering Queries Using Views.* VLDB Journal, 10(2-3):182–198, 2001.

[SR92] Srivastava, Divesh und Raghu Ramakrishnan: *Pushing Constraint Selections.* In: Vardi, Moshe Y. und Paris C. Kanellakis (Herausgeber): *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California, USA*, Seiten 301–315. ACM Press, 1992.

[SSH13] Saake, Gunter, Kai-Uwe Sattler und Andreas Heuer: *Datenbanken - Konzepte und Sprachen, 5. Auflage.* MITP, 2013.

[Swe02] Sweeney, Latanya: *k-Anonymity: A Model for Protecting Privacy.* International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 10(5):557–570, 2002.

[Vas09] Vassalos, Vasilis: *Answering Queries Using Views.* In: Liu, Ling und M. Tamer Özsu (Herausgeber): *Encyclopedia of Database Systems*, Seiten 92–98. Springer US, 2009.

[YNK14] Yordanova, Kristina, Martin Nyolt und Thomas Kirste: *Strategies for Reducing the Complexity of Symbolic Models for Activity Recognition.* In: *AIMSA*, Band 8722 der Reihe *Lecture Notes in Computer Science*, Seiten 295–300. Springer, 2014.

Anhang A

Anhang

A.1 combiningMCD(List<MCD>)

```
private String combiningMCD(List<MCD> descriptions) {
    List<AddedJoin> addedJoins;
    List<Integer> numberOfCoveredSubgoals = new ArrayList<>();
    // Die Liste der bereits abgedeckten Subgoals
    int numberOfSubgoalsInQuery = query.getSubgoals().size();
    // Anzahl der Teilziele in der Anfrage (zum Test, ob alle abgedeckt
    wurden)
    String queryNewFrom;
    String rewriting;
    if (query.hasJoins()) rewriting = query.getSelectString() + " " +
    query.getFromString();
    else {
        rewriting = query.getDatalog().toSQLString();
    }
    descriptions = sortMCDs(descriptions); // Funktion im Anschluss
    boolean joinAdded = false;
      /*
      *     Falls die Sicht die Anfrage komplett ersetzen kann:
      */
    for (MCD mcd : descriptions) {
        if (mcd.getG().size() == numberOfSubgoalsInQuery &&
        mcd.gethViews().getFromString().equals(query.getFromString())) {
            String[] rwSplit = rewriting.split(" FROM ");
            String[] vSplit;
            if (mcd.gethViews().getDefinition().contains(" WHERE ")) {
```

```
            String[] dummy = mcd.gethViews().getDefinition().split("
            WHERE ");
            vSplit = dummy[0].split(" FROM ");
        }
        else { vSplit = mcd.gethViews().getDefinition().split("
        FROM "); }
        if (vSplit[1].equals(rwSplit[1])) rewriting = rwSplit[0] +
        " FROM " + mcd.gethViews().getViewName();
        for (Mapping mapDieKarte : mcd.getPhi()) {
            rewriting =
            rewriting.replaceAll(mapDieKarte.getFirst(),
            mapDieKarte.getSecond());
            if (query.hasWhereClause()) {
                String[] firstS =
                mapDieKarte.getFirst().split("\\.");
                String[] secondS =
                mapDieKarte.getSecond().split("\\.");
                rewriteWhere(firstS[0], secondS[0]);
            }
        }
        if (query.hasWhereClause() && !rewriting.contains("
        WHERE")) rewriting += " WHERE " + query.getWhere();
        if (query.hasLimitClause() && !rewriting.contains(" LIMIT
        ")) rewriting += query.getLimit();
        return rewriting;
    }
}

for (MCD mcd : descriptions) { // für jede MCD
    String addedJoin = new String();
    addedJoins = new ArrayList<>();
    if (numberOfCoveredSubgoals.size() != numberOfSubgoalsInQuery)
    {  // solange es noch offene Subgoals gibt ...
        for (Mapping mapDieKarte : mcd.getPhi()) { // hole die IDs
        aus der Menge der vorhandenen SubgoalIDs
            int id = mapDieKarte.getSubgoalID();
            if (!numberOfCoveredSubgoals.contains(id)) {
                numberOfCoveredSubgoals.add(id);  // wenn die
                aktuelle ID noch nicht abgedeckt wurde, markiere
                sie als Abgedeckt und decke sie danach über das
                Mapping ab:
```

```
String mappingFrom = mapDieKarte.getFirst();
String mappingTo = mapDieKarte.getSecond();
rewriting = rewriting.replaceAll(mappingFrom,
mappingTo);  // ersetzt die SELECT-Items mit der
Sicht (Nebeneffekt: ersetzt auch die
Join-Attribute, falls diese äquivalent zu einem
SELECT-Item sind)

String[] mappingQuerySplit =
mappingFrom.split("\\.");
String[] mappingViewSplit = mappingTo.split("\\.");
String[] rewriteSplit = rewriting.split(" FROM ");
queryNewFrom = rewriteSplit[1];
if (query.hasWhereClause())
rewriteWhere(mappingQuerySplit[0],
mappingViewSplit[0]);
if (rewriteSplit[1].contains(" " +
mappingQuerySplit[0] + " ") |
rewriteSplit[1].contains(mappingQuerySplit[0] + "
") | rewriteSplit[1].contains(" " +
mappingQuerySplit[0]) &&
!rewriteSplit[0].contains(mappingQuerySplit[0] +
".")) // Wenn alle Vorkommen von der originalen
Relation im SELECT-Teil der Anfrage ersetzt wurden,
aber im FROM-Teil der Relationenname noch nicht
durch den Sicht-Namen ersetzt wurde, ersetze ihn.
{
if (joinAdded) {
 List<AddedJoin> joinsToDelete = new ArrayList<>();
 for (AddedJoin newJoinInQuery : addedJoins) {
   if (newJoinInQuery.getOriginalFromItem()
   .equals(mappingQuerySplit[0])) {
    queryNewFrom = rewriteSplit[1]
    .replaceAll(newJoinInQuery.getAddedJoin(), "");
    queryNewFrom = queryNewFrom
    .replaceAll(mappingQuerySplit[0],
    mappingViewSplit[0]);
    joinsToDelete.add(newJoinInQuery);
   }
  }
```

```
 addedJoins.removeAll(joinsToDelete);
 // lösche alle ersetzten Joins
 joinAdded = false;
} else queryNewFrom =
rewriteSplit[1].replaceAll(mappingQuerySplit[0],
mappingViewSplit[0]);
} else if (rewriteSplit[1].contains(" " +
mappingQuerySplit[0] + " ") |
rewriteSplit[1].contains(mappingQuerySplit[0] + "
") | rewriteSplit[1].contains(" " +
mappingQuerySplit[0]) &&
rewriteSplit[0].contains(mappingQuerySplit[0] +
"."))  // Wenn im FROM-Teil des aktuellen
Rewritings noch das aktuell zu ersetzende From-Item
der originalen Anfrage steht und dieses noch im
SELECT-Teil des Rewritings vorkommt:
{
    AddedJoin newJoin;
    boolean itBool = false;
    if (rewriteSplit[1].contains(" JOIN ")) {
        addedJoin = " JOIN " + mappingViewSplit[0]
        + " ON " + mappingFrom + " = " + mappingTo;
        newJoin = new AddedJoin(addedJoin,
        mappingQuerySplit[0], mappingViewSplit[0]);
        for (AddedJoin aj : addedJoins) {
            if (aj.equals(newJoin) && !itBool) {
                queryNewFrom = rewriteSplit[1];
                // Wenn dieser Join bereits
                hinzugefügt wurde, ignoriere die
                neue Version
                itBool = true;
            } else {
                queryNewFrom = rewriteSplit[1] +
                addedJoin;
                // Wenn er noch nicht hinzugefügt
                wurde, füge ihn an das Rewriting an.
                itBool = true;
            }
        }
        if (!itBool) queryNewFrom = rewriteSplit[1]
        + addedJoin;
```

```
                        } else {
                            addedJoin = mappingQuerySplit[0] + " JOIN "
                            + mappingViewSplit[0] + " ON " +
                            mappingFrom + " = " + mappingTo;
                            newJoin = new AddedJoin(addedJoin,
                            mappingQuerySplit[0], mappingViewSplit[0]);
                            queryNewFrom = rewriteSplit[1] + addedJoin;
                        }
                        if (!itBool) {
                            addedJoins.add(newJoin);
                            joinAdded = true;
                        }
                    }
                    rewriting = rewriteSplit[0] + " FROM " +
                    queryNewFrom;
                }
            }
        } else break;
    }

    if (query.getFromItems().size() < query.getSelectStrings().size())
    // dieser Test ist notwendig um einen Bug zu vermeiden, wenn es
    mehr Joins als selektierte Attribute gibt.
        rewriting = updateJoins(rewriting);
    if (query.hasWhereClause() && !rewriting.contains(" WHERE"))
    rewriting += " WHERE " + query.getWhere();
    if (query.hasLimitClause() && !rewriting.contains(" LIMIT "))
    rewriting += query.getLimit();

    return rewriting;
}
```

```
private List<MCD> sortMCDs(List<MCD> list) {
    List<MCD> mcdList = new ArrayList<>();
    List<Integer> querySubgoals = new ArrayList<>();
    for (int i = 1; i <= query.getSubgoals().size(); i++)
    querySubgoals.add(i);
    list.sort((o2, o1) -> Integer.compare(o1.getG().size(),
    o2.getG().size()));
  // Am Anfang werden die MCD's nach der Anzahl der von ihnen
  abgedeckten Teilzielen sortiert.
```

```
  // Dadurch sollen möglichst wenig genutze Sichten ermöglicht werden.
    querySubgoals.removeAll(list.get(0).getG());
    // lösche alle abgedeckten Subgoals aus der Liste
    mcdList.add(list.get(0));
    if (querySubgoals.size() > 0)
        for (MCD mcd : list) {
            if (mcd.getG().containsAll(querySubgoals)) {
                mcdList.add(mcd);
                return mcdList;
            }
        }

  for (int i : querySubgoals) {   // einfach zum durchgehen
        List<SubgoalMCDMapping> mapDieListe = new ArrayList<>();
    // Liste, wobei jedes Element eine Anzahl der Teilziele hat, die
    abgedeckt werden können und dazu die MCD
        for (MCD mcd : list) {
            int numberOfCoverredSubgoals = 0;
            for (int g : mcd.getG()) {
                if (querySubgoals.contains(g))
                numberOfCoverredSubgoals++;
            }
            mapDieListe.add(new
            SubgoalMCDMapping(numberOfCoverredSubgoals, mcd));
        }
        mapDieListe.sort((o2, o1) ->
        Integer.compare(o1.getNumberOfSubgoals(),
        o2.getNumberOfSubgoals()));
        querySubgoals.removeAll(mapDieListe.get(0).getMcd().getG());
        mcdList.add(mapDieListe.get(0).getMcd());
        if (querySubgoals.size() == 0) return list;
    }
    return list;
}
```

Selbständigkeitserklärung

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

Rostock, den 6. September 2017