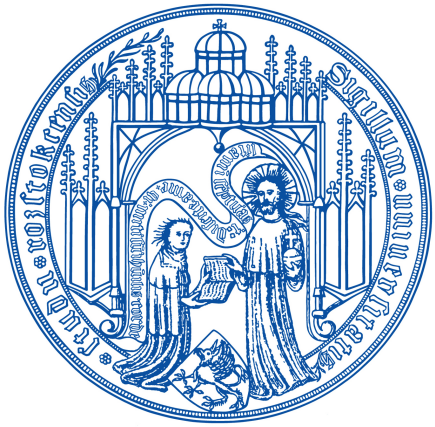
Nutzung von ChaTEAU für das Answering-Queries-using-Views-Problem (AQuV)

Bachelorarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik



vorgelegt von: Nico Trebbin
Matrikelnummer: 218204402
geboren am: 10.01.2000 in Malchin
Erstgutachter: Prof. Dr. rer. nat. habil. Andreas Heuer
Zweitgutachter: Dr.-Ing. Holger Meyer
Betreuer: Florian Rose
Abgabedatum: 10.02.2022

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit dem Ziel, Anfragen auf Datenbanken nur über Sichten zu beantworten. Das Finden von äquivalenten Anfragen auf Sichten ist nicht von einfacher Natur. Es gibt eine handvoll Algorithmen, mit denen diese Anfragen umgeschrieben werden können. Ein Verfahren ist der CHASE&BACKCHASE. Diese Arbeit beschäftigt sich mit diesem Verfahren und seinen Optimierungen. Das Ziel dieser Arbeit ist es, den CHASE&BACKCHASE und den Provenance-Directed CHASE&BACKCHASE in ChaTEAU zu implementieren. Das Hauptaugenmerk liegt hierbei auf der provenance-basierten Variante.

Abstract

This thesis deals with the question of how queries on databases can only be answered using views. Finding equivalent queries is not a simple matter. There exists a handful of algorithms that can rewrite queries. One of them is the CHASE&BACKCHASE. This thesis deals with the previously mentioned procedure and his optimizations. The aim of the thesis is the implementation of the CHASE&BACKCHASE and the provenance-directed CHASE&BACKCHASE in ChaTEAU. Great attention is paid here to the provenance-directed algorithm.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Ziel der Arbeit . . . 1
1.2 Beispieldatensatz . . . 1
1.3 Gliederung der Arbeit . . . 2

2 Grundlagen **5**
2.1 Relationenmodell . . . 5
2.1.1 Schemata und Instanzen . . . 5
2.1.2 Integritätsbedingungen . . . 6
2.2 Relationenalgebra . . . 8
2.3 Answering-Queries-using-Views . . . 9
2.3.1 Query Rewriting . . . 10
2.3.2 Containment-Mappings . . . 11
2.4 CHASE . . . 11
2.5 Provenance . . . 12

3 Stand der Forschung und Technik **13**
3.1 Stand der Forschung . . . 13
3.1.1 Bucket-Algorithmus . . . 14
3.1.2 MiniCon-Algorithmus . . . 15
3.1.3 CHASE&BACKCHASE . . . 17
3.1.4 Vergleich der Algorithmen . . . 21
3.2 Stand der Technik . . . 22
3.2.1 Bucket-Algorithmus . . . 22
3.2.2 MiniCon-Algorithmus . . . 22
3.2.3 CHASE&BACKCHASE . . . 22
3.2.4 ChaTEAU . . . 23

4 Konzept **25**
4.1 CHASE-Phase . . . 25
4.2 BACKCHASE-Phase . . . 25
4.2.1 Invertierung der Sichtbedingungen . . . 25
4.2.2 Standard-BACKCHASE . . . 26
4.2.3 Provenance-Directed CHASE&BACKCHASE . . . 26

5 Implementierung **31**
5.1 Grundgerüst . . . 31
5.2 BACKCHASE . . . 32
5.3 BACKCHASE-Pruning . . . 34

5.4 Provenance-Directed CHASE&BACKCHASE 34
5.4.1 Provenance-bewusster CHASE 35
5.4.2 Berechnung der Why-Provenance der Originalanfrage 40
5.4.3 Berechnung der Rewritings 42

6 Fazit & Ausblick **45**
6.1 Fazit 45
6.2 Evaluation der Implementierungen 46
6.2.1 Aufbau und Ablauf des Experiments 46
6.2.2 Vergleich der Laufzeit 46
6.3 Ausblick 48

Literaturverzeichnis **51**

Abkürzungsverzeichnis **53**

A Beispiele **I**
A.1 CHASE&BACKCHASE I
A.2 CHASE&BACKCHASE – Pruning II

B Quellcode **V**
B.1 Grundgerüst V
B.2 BACKCHASE XIII
B.3 BACKCHASE-Pruning XVI
B.4 Provenance-Directed CHASE&BACKCHASE XVII
B.5 Experiment XXXIII

C Datenträger **XLI**

Kapitel 1

Einleitung

In der heutigen Zeit spielen Anfragen auf Sichten eine immer größere Rolle. Durch die Verwendung von materialisierten Sichten kann die Berechnungszeit einer Anfrage um einen nicht geringen Wert verringert werden. Auch Anfragen auf virtuelle Datenbanken, zum Beispiel bei der Datenintegration, werden somit bewerkstelligt [DHI12; HL00]. Durch das Benutzen von Sichten kann zudem der Zugriff auf sicherheitskritische und sensible Daten eingeschränkt werden. Das Stellen von Anfragen auf Sichten besitzt demnach viele Vorteile. In vielen Fällen lohnt es sich deshalb, Anfrage auf Basisrelationen über Sichten zu beantworten. Beim Umformen der Anfragen muss jedoch darauf geachtet werden, dass sich das Ergebnis der Anfrage nicht verändert. Eine Möglichkeit Anfragen umzuschreiben, bietet der CHASE&BACKCHASE. Ein Themenschwerpunkt der Arbeit ist hierbei der Provenance-Directed CHASE&BACKCHASE (PD-CHASE&BACKCHASE), eine Optimierung des vorher genannten Verfahrens.

1.1 Ziel der Arbeit

In dieser Arbeit soll der CHASE&BACKCHASE als auch der Provenance-Directed CHASE&BACKCHASE in ChaTEAU umgesetzt werden. Unter Anwendung dieser Verfahren können Anfragen umgeformt werden. Die Anfragesprache ist, wie von ChaTEAU vorgegeben, auf konjunktive Anfragen begrenzt. Die implementierten Prototypen sollen hierbei unter Verwendung von TGDs funktionieren.

1.2 Beispieldatensatz

Im Verlauf dieser Arbeit werden Beispiele anhand einer Datenbank gezeigt. Diese Datenbank liegt einer Social-Cataloging-Website zugrunde. Jegliche Leser können Bücher rezensieren und bewerten. Insgesamt gibt es vier Relationen, die `BÜCHER`-, die `LESER`-, die `AUTOREN`- und die `REZENSIONEN`-Relation.

Abbildung 1.1: Relationenschemata des Beispieldatensatzes
`AUTOREN` = {AutorID, Vorname, Nachname}
`BÜCHER` = {BuchID, AutorID, Titel, Verlag, Veröffentlichung, Preis}
`LESER` = {LeserID, Benutzername, Passwort, E-Mail}
`REZENSIONEN` = {LeserID, BuchID, Bewertung, Inhalt}

Abbildung 1.2: Datenbankschema des Beispieldatensatzes
S := {`AUTOREN`, `BÜCHER`, `LESER`, `REZENSIONEN`}

Jedes in Abbildung 1.1 auftretende Attribut benötigt einen Wertebereich. Die hier angegebenen Wertebereiche sind in der Sprache `PL/pgSQL` des Datenbankmanagementsystems PostgreSQL formuliert.

Abbildung 1.3: Wertebereiche der Attribute
dom(AutorID) := INT CHECK(MOD(AutorID, 10) = 1 AND AutorID >= 10)
dom(Vorname) := VARCHAR(50)
dom(Nachname) := VARCHAR(50)
dom(Bewertung) := DECIMAL(4, 2)(CHECK(Bewertung BETWEEN 0 AND 10))
dom(BuchID) := INT CHECK(MOD(BuchID, 10) = 2 AND BuchID >= 10)
dom(Titel) := VARCHAR(50)
dom(Verlag) := VARCHAR(50)
dom(Veröffentlichung) := DATE
dom(Preis) := DECIMAL(10, 2)
dom(LeserID) := INT CHECK(MOD(LeserID, 10) = 3 AND LeserID >= 10)
dom(Benutzername) := VARCHAR(50)
dom(Passwort) := VARCHAR(64)
dom(E-Mail) := VARCHAR(50)
dom(Inhalt) := CLOB [1]

Datenbanken enthalten oftmals Integritätsbedingungen. Einige Bedingungen wurden schon in Abbildung 1.3 angegeben.

Abbildung 1.4: Integritätsbedingungen des Beispieldatensatzes
`AUTOR`(AutorID) → `AUTOR`(Vorname, Nachname)
`BÜCHER`(BuchID) → `BÜCHER`(AutorID, Titel, Verlag, Veröffentlichung, Preis, Bewertung)
`LESER`(LeserID) → `LESER`(Benutzername, Passwort, E-Mail)
`REZENSIONEN`(LeserID, BuchID) → `REZENSIONEN`(Bewertung, Inhalt)

Hierbei ist {AutorID} der Primärschlüssel von `AUTOREN`, {BuchID} der Primärschlüssel von `BÜCHER`, {LeserID} der Primärschlüssel von `LESER` und {LeserID, BuchID} der Primärschlüssel von `REZENSIONEN`.

Zudem sind folgende Fremdschlüssel definiert:
`BÜCHER`(AutorID) → `AUTOREN`(AutorID)
`REZENSIONEN`(LeserID) → `LESER`(LeserID)
`REZENSIONEN`(BuchID) → `BÜCHER`(BuchID)

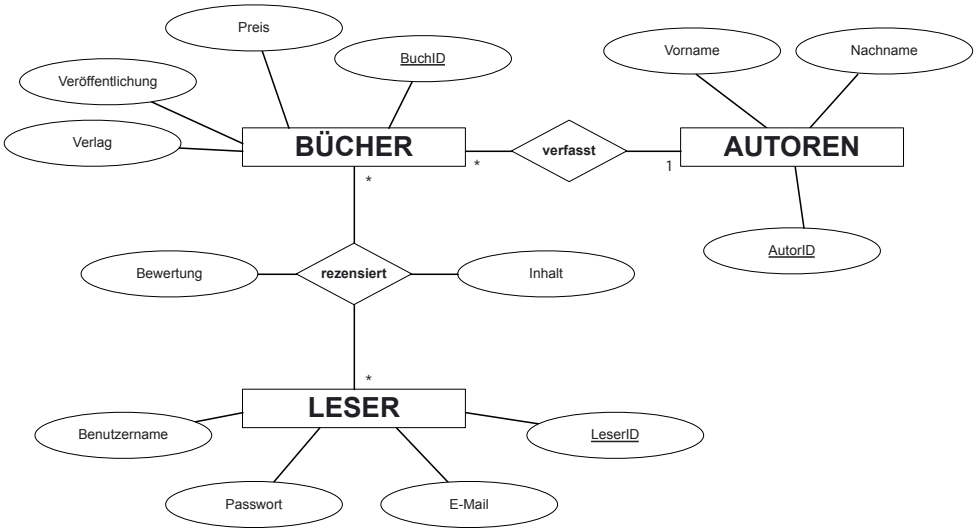
Zum besseren Verständnis, wie die Beispieldatenbank aufgebaut ist, findet sich in Abbildung 1.5 das Entity-Relationship-Modell der Datenbank. Die Schlüsselattribute wurden in dieser Abbildung unterstrichen. Als Notationsform wurde hier die CHEN-Notation verwendet.

1.3 Gliederung der Arbeit

In diesem Kapitel wurde in das Thema des Anfragen-Umschreibens eingeführt. Es wurde das Ziel der vorliegenden Arbeit beschrieben. Zudem wurde ein Beispieldatensatz vorgegeben, auf den im Verlauf der Arbeit öfters referenziert wird. Im nächsten Kapitel werden grundlegende Begrifflichkeiten vorgestellt, die für den weiteren Verlauf der Arbeit von Wichtigkeit sind. Das dritte Kapitel gibt den aktuellen Stand der Technik und der Forschung wieder. Hierbei wird auf theoretische Algorithmen zum Beantworten von Anfragen über Sichten eingegangen. Zudem werden einige Systeme vorgestellt, die diese Algorithmen implementiert haben. Im vierten Kapitel wird der CHASE&BACKCHASE als auch seine Optimierungen detaillierter angeschaut. Im fünften Kapitel wird gezeigt, wie die vorgestellten Konzepte in ChaTEAU umgesetzt wurden. Das letzte Kapitel fasst die Kernthesen dieser Arbeit zusammen. Dieses Kapitel erläutert

[1] Character-Large-Object (CLOB) ist ein Datentyp, der für das Speichern sehr langer Zeichenketten verwendet wird.

Abbildung 1.5: ER-Modell



kurz, was in der Arbeit erreicht wurde und welche Erweiterungen und Optimierungen noch implementiert werden können. Zudem wurde die Laufzeit der Prototypen analysiert und verglichen.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen vermittelt, die für das Verständnis dieser Arbeit notwendig sind. In den ersten beiden Abschnitten werden die relationalen Konzepte näher erläutert. Dabei wird genauer auf das Relationenmodell und die Relationenalgebra eingegangen. Im dritten Abschnitt wird das Answering-Queries-using-Views-Problem (AQuV) behandelt. Der darauffolgende Abschnitt beschreibt den Standard-CHASE. Im letzten Abschnitt werden Begriffe der Provenance definiert. Einzelne Themenbereiche, darunter die mathematische Logik und Algebra, werden als bekannt vorausgesetzt.

2.1 Relationenmodell

Das Relationenmodell bildet das Grundgerüst relationaler Datenbanksysteme. Mit ihm ist es möglich, Daten effizient zu speichern und zu verwalten. In diesem Abschnitt werden grundlegende Begriffe des Relationenmodells definiert.

2.1.1 Schemata und Instanzen

Das Universum einer Datenbank enthält alle Attribute, die innerhalb der Datenbank vorkommen. Die Domänen der Attribute des Beispieldatensatzes wurde bereits in Abbildung 1.3 verdeutlicht.

Definition 2.1: Universum, Domäne nach [SSH18]
Das Universum $\mathcal{U} := \{A_1, \ldots, A_n\}$ ist eine nichtleere, endliche Menge von Attributen. Eine Domäne dom ist eine Abbildung $\text{dom} : \mathcal{U} \to \mathcal{D}$, die jedes Attribut aus $\mathcal{U}$ auf einen Wertebereich $D \in \mathcal{D}$ abbildet. Der Wertebereich $\text{dom}(A)$ eines Attributs $A \in \mathcal{U}$ gibt hierbei die erlaubten Werte für A an.

Die Schemata geben die Struktur einer Relation oder einer Datenbank an. Die Struktur einer Relation besteht nur aus den enthaltenen Attributen. In Abbildung 1.1 wurden die Relationenschemata des Beispieldatensatzes definiert.

Definition 2.2: Relationenschema nach [SSH18]
Sei $\mathcal{U}$ das Universum. Die Menge $R \subseteq \mathcal{U}$ heißt Relationenschema.

Eine Relation ist eine Menge von Datensätzen. Der Aufbau dieser Datensätze erfolgt nach dem zugrundeliegenden Relationenschema.

Definition 2.3: Relation, Tupel nach [SSH18]
Sei $\mathcal{D}$ eine Menge von Domänen. Eine Relation r über einem Relationenschema R, kurz $r(R)$, ist

eine endliche Menge von Abbildungen t mit:

$$t : R \to \bigcup_{D \in \mathcal{D}} D$$

Hierbei gilt $\forall A \in R$, dass $t(A) \in \mathrm{dom}(A)$ enthalten ist.

Datenbankschemata geben die Struktur von Datenbanken an. In Abbildung 1.2 wurde das Schema der Beispieldatenbank definiert.

Definition 2.4: Datenbankschema nach [SSH18]
Sei $R \subseteq \mathcal{U}$ ein Relationenschema. Die Menge von Relationenschemata $S := \{R_1, \ldots, R_n\}$ wird als Datenbankschema bezeichnet.

Definition 2.5: Datenbank nach [SSH18]
Eine Datenbank $d := \{r_1, \ldots, r_n\}$ über dem Datenbankschema $S := \{R_1, \ldots, R_n\}$ ist eine Menge von Relationen, wobei $\forall i \in \{1, \ldots, n\} : r_i\,(R_i)$.

Definition 2.6: Terme nach [Ben+17]
Seien Konstanten, Null-Werte und Variablen paarweise disjunkte Mengen. Konstanten geben festgelegte Werte und Null-Werte geben unbekannte Werte an. Ein Term t ist dann eine Konstante, ein Null-Wert oder eine Variable.

Definition 2.7: relationales Atom / Atom nach [Ben+17]
Ein relationales Atom über einem Datenbankschema S ist ein Ausdruck der Form $R\left(\vec{t}\right)$. Hierbei ist $R \in S$ ein Relationenschema mit Stelligkeit $n \in \mathbb{N}$ und $\vec{t}$ ein Tupel mit n Termen. Ein Gleichheitsatom (engl. equality atom) ist ein spezielles Atom der Form $t_1 = t_2$, wobei t_1 und t_2 Terme sind.
Ein Atom heißt null-frei, falls es keine Null-Werte enthält.

2.1.2 Integritätsbedingungen

Integritätsbedingungen sind Bedingungen, die eine Datenbank erfüllen muss. Man unterscheidet zwischen lokalen und globalen Integritätsbedingungen. Während lokale Integritätsbedingungen nur eine Relation einschränken, können sich globale Integritätsbedingungen über die gesamte Datenbank auswirken. In Abbildung 1.3 und Abbildung 1.4 wurden die Integritätsbedingungen des Beispieldatensatzes angegeben.

Definition 2.8: lokale Integritätsbedingungen nach [SSH18]
Sei $\mathcal{B}$ eine Menge von lokalen Integritätsbedingungen. Lokale Integritätsbedingungen für ein Relationenschema R und einer Relation $r(R)$ sind Abbildungen $b \in \mathcal{B}$ mit:

$$b : \{r | r(R)\} \to \{\mathrm{true}, \mathrm{false}\}$$

Definition 2.9: globale Integritätsbedingungen nach [SSH18]
Eine Menge globaler Integritätsbedingungen Γ besteht aus Abbildungen $\gamma : \{d | d(S)\} \to \{\mathrm{true}, \mathrm{false}\}$. Diese Abbildungen überprüfen, ob die Datenbank d die Integritätsbedingung $\gamma \in \Gamma$ erfüllt.

Abhängigkeiten sind spezielle Integritätsbedingungen, sie geben Beziehungen zwischen Attributen an.

Definition 2.10: Equality-Generating-Dependency (EGD) nach [Ben+17]
Sei $\vec{x}$ ein Vektor von Variablen. Sei F eine Konjunktion von null-freien Atomen. Eine Equality-Generating-Dependency (EGD) ist eine Abhängigkeit der folgenden Form:

$$\forall \vec{x} : F(\vec{x}) \rightarrow x_i = x_j \text{ mit } x_i, x_j \in \vec{x}$$

Durch die Verwendung von EGDs können Beziehungen zwischen Attributen dargestellt werden.

Definition 2.11: Functional-Dependency, Funktionale Abhängigkeit (FD) nach [SSH18]
Eine funktionale Abhängigkeit (FD) für eine Relation $r(R)$ und den Attributmengen $X, Y \subseteq R$ hat die Form $X \rightarrow Y$. Die Relation r genügt der FD genau dann, wenn für alle Tupel $t_1, t_2 \in r$ gilt: $t_1(X) = t_2(X) \Rightarrow t_1(Y) = t_2(Y)$. Dies ist genau dann der Fall, wenn die Attributmenge X die Attributmenge Y eindeutig bestimmt. Eine FD ist eine spezielle EGD.

Beispiel 2.1: EGD
In der Beispieldatenbank gibt es nach Abbildung 1.4 die Schlüsselabhängigkeit *AutorID* → *Vorname, Nachname* über die Autorenrelation. Diese Abhängigkeit ist eine FD, also eine spezielle EGD. Sie lässt sich in mehrere Teilabhängigkeiten aufteilen, eine davon ist *AutorID* → *Vorname*. Die EGD dieser Abhängigkeit sieht wie folgt aus:

$$\forall aid, vn_1, nn_1, vn_2, nn_2 : r_{\texttt{AUTOREN}}(aid, vn_1, nn_1) \wedge r_{\texttt{AUTOREN}}(aid, vn_2, nn_2) \rightarrow vn_1 = vn_2$$

Wenn die AutorID (*aid*) zweier Autoren gleich ist, so sind auch deren Vornamen (vn_1, vn_2) gleich. Der inverse Fall ist im Allgemeinen nicht gültig. Die AutorID zweier Autoren ist also in der Regel nicht äquivalent, selbst wenn deren Vornamen identisch sind.
Die gewählten Variablen repräsentieren hierbei die korrespondierenden Attribute aus dem Relationenschema `AUTOREN`.

Definition 2.12: Tuple-Generating-Dependency (TGD) nach [Ben+17]
Seien $\vec{x}$ und $\vec{y}$ Vektoren von Variablen. Seien F und G Konjunktionen von null-freien Atomen. Eine Tuple-Generating-Dependency (TGD) ist eine Abhängigkeit der folgenden Form:

$$\forall \vec{x} : F(\vec{x}) \rightarrow \exists \vec{y} : G(\vec{x}, \vec{y})$$

Durch die Verwendung von TGDs können neue Tupel generiert werden.

Definition 2.13: Join-Dependency, Verbundabhängigkeit (JD) nach [SSH18]
Eine Verbundabhängigkeit für ein Datenbankschema $S = \{R_1, \ldots, R_n\}$ über ein Universum $\mathcal{U}$ hat die Form: $\bowtie [R_1, \ldots, R_n]$. Eine Relation $r(\mathcal{U})$ erfüllt eine Verbundabhängigkeit genau dann, wenn

$$r = \pi_{R_1}(r) \bowtie \cdots \bowtie \pi_{R_n}(r)$$

gilt. Eine JD ist eine spezielle TGD.

Definition 2.14: Source-to-Target Tuple-Generating-Dependency (s-t-TGD) nach [Ben+17]
Eine s-t-TGD ist eine TGD, deren Rumpf (linke Seite der Abhängigkeit) nur aus Relationen des

Quellschemas und deren Kopf (rechte Seite der Abhängigkeit) nur aus Relationen des Zielschemas besteht.

Beispiel 2.2: TGD
Mithilfe von TGDs kann man Sichtdefinitionen beschreiben. Als Beispiel wird die Sicht `LIEST` dargestellt. Diese Sicht ist über die Attribute LeserID und BuchID definiert.

$$\forall lid, bid, bew, inh : r_{\texttt{REZENSIONEN}}(lid, bid, bew, inh) \rightarrow v_{\texttt{LIEST}}(lid, bid)$$

Die Sicht $v_{\texttt{LIEST}}$ enthält alle Leser-IDs und Buch-IDs aus der `REZENSIONEN`-Relation. Die gewählten Variablen repräsentieren hierbei die korrespondierenden Attribute aus den Relationenschemata `LESER` und `REZENSIONEN`.

2.2 Relationenalgebra

Um Anfragen auf Datenbanken zu ermöglichen, werden Operatoren benötigt. Diese Menge an Operatoren definiert die Relationenalgebra. Mithilfe der Relationenalgebra können Relationen gefiltert, verbunden oder allgemein modifiziert werden.

Definition 2.15: minimale Relationenalgebra nach [SSH18]
Die minimale Relationenalgebra besteht aus den Operatoren Projektion π, Selektion σ, Verbund $\bowtie$, Vereinigung $\cup$, Differenz $-$ und Umbenennung β.

Definition 2.16: Projektion nach [SSH18]
Sei R ein Relationenschema und sei $X \subseteq R$ eine ausgewählte Attributmenge in R. Die Projektion π auf einer Relation $r(R)$ bildet dann die Werte der Tupel $t \in r$ auf die Attribute aus X ab.

$$\pi_X(r) := \{t(X) | t \in r\}$$

Definition 2.17: Selektion nach [SSH18]
Sei $R = \{A_1, \ldots, A_n\}$ ein Relationenschema und sei F eine Formel. Die Selektion σ auf einer Relation $r(R)$ wählt dann alle Tupel t aus, die die Formel $F(t)$ erfüllen.

$$\sigma_F(r) := \{t | t \in r \wedge F(t) = true\}$$

Seien $A_i, A_j \in R$ mit $A_i \neq A_j$, sei c eine Konstante und sei $\theta \in \{\leq, <, =, \neq, >, \geq\}$ ein Vergleichssymbol. Die Formel F kann dann folgende Formen annehmen:

- Konstantenselektion: $A_i \, \theta \, c$
- Attributselektion: $A_i \, \theta \, A_j$
- logische Verknüpfungen von Formeln F

Definition 2.18: natürlicher Verbund nach [SSH18]
Der natürliche Verbund zweier Relationen $r_1(R_1)$ und $r_2(R_2)$ wird wie folgt definiert:

$$r_1 \bowtie r_2 := \{t | t(R_1 \cup R_2) \wedge \exists t_1 \in r_1 : t_1 = t(R_1) \wedge \exists t_2 \in r_2 : t_2 = t(R_2)\}$$

Neben dem natürlichen Verbund gibt es auch andere Verbundarten, wie das Kreuzprodukt und den äußeren Verbund. Zur besseren Verständlichkeit von Anfragen können zudem weitere Operatoren außerhalb der minimalen Relationenalgebra verwendet werden, namentlich der Schnitt und die Division zweier Relationen, als auch die eben genannten Verbundarten.

2.3 Answering-Queries-using-Views

In diesem Abschnitt werden die Begrifflichkeiten definiert, die man zum Beantworten von Anfragen über Sichten benötigt. Das Answering-Queries-using-Views-Problem (AQuV) vereinigt eine Vielzahl an Problemen. In dieser Arbeit wird sich mit der Frage beschäftigt, ob eine Anfrage über ein Datenbankschema nur über die gegebenen Sichten beantwortet werden kann. Ein weiteres Beispiel des AQuV-Problems ist die Frage, wie viele Tupel maximal nur durch das Verwenden von Sichten hergeleitet werden können [HL00].

Definition 2.19: Konjunktive Anfrage, Teilziel nach [HL00]
Sei q das Anfrageprädikat und seien $r_1, \ldots, r_n$ Relationen einer Datenbank. Seien $X, X_1, \ldots X_n$ nullfreie Tupel. Eine konjunktive Anfrage ist eine Anfrage der Form

$$q(X) :- r_1(X_1), \ldots, r_n(X_n)$$

Diese Anfrage heißt sicher, wenn $X \subseteq X_1 \cup \cdots \cup X_n$ gilt. Die Atome $r_1(X_1), \ldots, r_n(X_n)$ sind die Teilziele im Rumpf der Anfrage. Die Teilziele können relationale Atome und Gleichheitsatome sein.

Definition 2.20: Sicht, materialisierte Sicht nach [DHI12]
Eine Sicht ist eine Anfrage, dessen Kopf (linke Seite der Anfrage) eine neue Relation definiert. Wird das Ergebnis der Anfrage zwischengespeichert, so bezeichnet man dies als materialisierte Sicht.

Definition 2.21: Ausgezeichnete Variablen, Kopfvariablen [DHI12]
Ausgezeichnete Variablen, auch Kopfvariablen genannt, sind die Variablen im Kopf einer Anfrage. Eine Variable heißt nichtausgezeichnet, falls sie nur im Rumpf der Anfrage auftaucht.

Beispiel 2.3: Relationenalgebra und konjunktive Anfragen
Wie die Relationenalgebra, unterstützen auch konjunktive Anfragen die Projektion, die Selektion und den Verbund. Mithilfe von Gleichheitsatomen können Variablen durch den Operator = näher eingegrenzt werden. Falls eine Variable durch ein Gleichheitszeichen eingegrenzt wird, so kann das Gleichheitsatom entfernt und jedes Vorkommen der eingegrenzten Variable mit dem Term auf der anderen Seite des Gleichheitszeichens ersetzt werden.
Durch die Nutzung von Variablen können Beziehungen zwischen Atomen beschrieben werden [Ben+17]. Falls mehrere Atome in der Anfrage enthalten sind, so wird ein Verbund ausgeführt. Dieser Verbund artet zum kartesischen Verbund aus, falls keine Verbundattribute vorhanden sind.
Mithilfe der ausgezeichneten Variablen einer Anfrage kann das Ergebnis dieser Anfrage auf bestimmte Variablen projiziert werden.

Beispiel 2.4: Konjunktive Anfragen
Die Sicht $V_{\texttt{LIEST}}$ über den Attributen LeserID und BuchID ist eine Anfrage der folgenden Form:

$$V_{\texttt{LIEST}}(lid, bid) :- \texttt{REZENSIONEN}\,(lid, bid, bew, inh)$$

Die gewählten Variablen repräsentieren hierbei die korrespondierenden Attribute aus dem Relationenschema `REZENSIONEN`. Im Rahmen der Relationenalgebra entspricht dies der Anfrage $\pi_{\text{LeserID, BuchID}}(r_{\texttt{REZENSIONEN}})$

Definition 2.22: Äquivalenz von Anfragen nach [LMS95]
Zwei Anfragen heißen äquivalent, falls sie dasselbe Ergebnis über jeder Datenbank, die die Integritätsbedingungen erfüllen, ausgeben.

Im weiteren Verlauf der Arbeit wird die Sprache der Anfragen und Sichten auf konjunktive Anfragen beschränkt. Zudem werden die Teilziele einer Anfrage auf relationale Atome begrenzt.

2.3.1 Query Rewriting

Query Rewriting (das Umschreiben von Anfragen) ist ein prominentes Problem aus dem Datenbankmilieu. Im Falle des AQuV-Problems versucht man Relationen innerhalb der Anfrage mit Sichten auszutauschen.

Definition 2.23: Query Rewriting nach [LMS95]
Sei Q eine Anfrage über den Sichten $\mathcal{V} = \{V_1, \ldots, V_m\}$. Eine Anfrage Q' ist ein Rewriting von Q, falls folgendes gilt:

- Q und Q' sind äquivalent, und
- Q' enthält mindestens ein Atom aus $\mathcal{V}$

Definition 2.24: globale und lokale minimale Rewritings nach [LMS95]
Sei Q' ein Rewriting von der Anfrage Q. Q' ist lokal minimal, falls das Entfernen eines beliebigen Teilziels die Äquivalenz zu Q verletzt. Q' ist global minimal, falls kein lokal minimales Rewriting mit weniger Teilzielen existiert.

Definition 2.25: Komplettes Rewriting nach [LMS95]
Sei Q' ein Rewriting von der Anfrage Q über den Sichten $\mathcal{V} = \{V_1, \ldots, V_n\}$. Q' ist ein komplettes Rewriting, falls es nur Atome aus $\mathcal{V}$ und die Vergleichsoperatoren $<, \leq, >, \geq, =$ und $\neq$ verwendet.

Die Teilziele einer Anfrage bestehen nur aus relationalen Atomen und Gleichheitsatomen, die Menge der verwendbaren Vergleichsoperatoren wird demnach auf die Gleichheit beschränkt.

Beispiel 2.5: Rewritings
Angenommen, für die Anfrage Q gibt es eine äquivalente Anfrage Q'. Die Anfrage Q' ist nach

$$Q(bid) :- \texttt{REZENSIONEN}(lid, bid, bew, inh), \texttt{BÜCHER}(bid_1, aid, titel, verl, ver, pr), bid = bid_1$$
$$Q'(bid) :- V_{\texttt{LIEST}}(lid, bid), \texttt{BÜCHER}(bid_1, aid, titel, verl, ver, pr), bid = bid_1$$
$$Q''(bid) :- V_{\texttt{LIEST}}(lid, bid), \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$$

Definition 2.23 ein Rewriting, da beide Bedingungen erfüllt sind. Durch die Verwendung der `LIEST`-Sicht enthält Q' mindestens ein Atom aus $\mathcal{V}$. Dieses Rewriting ist jedoch nicht minimal, das Gleichheitsatom kann aufgelöst werden. Auch Anfrage Q'' ist ein Rewriting von Q. Dieses Rewriting ist zudem lokal minimal. Wenn die Menge der Sichten $\mathcal{V}$ auf die `LIEST`-Sicht beschränkt wird, so ist sie sogar global minimal. Kein Teilziel von Q'' kann weggelassen werden. Durch das Weglassen eines Teilziels ist der Verbund zwischen `REZENSIONEN` und `BÜCHER` nicht mehr gewährleistet.

Die eben genannten Rewritings sind keine kompletten Rewritings. Jede Anfrage enthält das BÜCHER-Atom der Originalanfrage Q. Ein komplettes Rewriting darf nur aus Gleichheitsatomen und Sichten bestehen.

2.3.2 Containment-Mappings

Mithilfe von Containment-Mappings versucht man die Äquivalenz zweier Anfragen zu zeigen. Dieses Problem ist sehr stark mit dem Query Rewriting verwandt.

Definition 2.26: Containment-Mapping nach [LMS95]
Seien Q_1, Q_2 Anfragen über eine Datenbank. Das Containment-Mapping von Q_1 zu Q_2 ist eine Abbildung von den Variablen von Q_1 auf die Variablen von Q_2. Alle Atome aus Q_1 werden also auf ein Atom aus Q_2 abgebildet.

Definition 2.27: Containment nach [LMS95; CLL06]
Seien Q_1, Q_2 Anfragen über eine Datenbank. Falls es ein Containment-Mapping von Q_1 zu Q_2 gibt, so ist Q_2 in Q_1 enthalten, kurz $Q_2 \sqsubseteq Q_1$. Falls $Q_2 \sqsubseteq Q_1$ und $Q_1 \sqsubseteq Q_2$ gilt, dann sind Q_1 und Q_2 äquivalent.

2.4 CHASE

Der CHASE-Algorithmus ist einer der prominentesten Algorithmen aus der Datenbankforschung. Mit ihm kann eine Vielzahl von Problemen gelöst werden.

Definition 2.28: Termabbildung nach [Ben+17]
Eine Termabbildung $\sigma := \{t_1 \to s_1, \ldots, t_n \to s_n\}$ ist eine partielle Abbildung von Termen auf Termen.

Definition 2.29: Trigger, aktiver Trigger nach [Ben+17]
Sei τ eine Abhängigkeit. Ein Trigger für τ im Objekt $\mathcal{O}$ ist eine Abbildung, die den Rumpf von τ auf einen Teil von $\mathcal{O}$ abbildet.
Ein aktiver Trigger h für τ in $\mathcal{O}$ ist wie folgt definiert:

- Falls τ eine TGD ist, dann gibt es keine Erweiterung von h, mit der man den Kopf von τ auf einen Teil von $\mathcal{O}$ abbilden kann.
- Falls τ eine EGD ist, dann ist $h(x_i) \neq h(x_j)$, wobei x_i, x_j Variablen aus dem Kopf von τ sind.

Im CHASE werden Abhängigkeiten $\star$ in ein CHASE-Objekt $\bigcirc$ eingearbeitet. CHASE-Objekte können hierbei zum Beispiel Instanzen und Anfragen sein [AH19].

$$\text{chase}_\star(\bigcirc) = \circledast$$

Im Quellcodeausschnitt 2.1 ist der Ablauf des CHASE näher beschrieben. Der CHASE reicht zum Umschreiben von Anfragen allein nicht aus. Damit das AQuV-Problem auch mit dem CHASE lösbar ist, muss eine BACKCHASE-Phase eingeführt werden. Die Funktionsweise und Anwendung des CHASE&BACKCHASE wird im Abschnitt 3.1.3 näher erläutert.

Quellcodeausschnitt 2.1 Standard-CHASE $(\Sigma, \mathcal{O})$ nach [Ben+17]

```
1  i := 0; CHASE-Objekt O_i := O;
2  WHILE O_i ≠ ∅ do:
3    FOREACH Abhängigkeit τ ∈ Σ DO:
4      FOREACH aktiven Trigger h für τ aus O DO:
5        IF τ ist eine TGD der Form ∀x⃗ : F(x⃗) → ∃y⃗ : G(x⃗, y⃗) THEN:
6          Erweitere h um die Abbildung von y⃗ auf ein Vektor freier
           Nullwerte
7          Füge Tupel zum Objekt O_i hinzu
8        ELSE IF τ ist eine EGD der Form ∀x⃗ : F(x⃗) → x_i = x_j mit i ≠ j, x_i, x_j ∈ x⃗
9          IF x_i, x_j sind Konstanten und h(x_i) ≠ h(x_j) THEN:
10           CHASE schlägt fehl
11         ELSE IF x_i ist eine Konstante THEN:
12           Bilde x_j auf x_i ab
13         ELSE:
14           Bilde x_i auf x_j ab
15   erweitere O um die neu generierten Tupel und ändere die
     Variablenabbildungen, falls EGDs verwendet wurden
16 return O
```

2.5 Provenance

Provenance beschreibt den Ursprung und die Geschichte von Daten. Sie wird in einer Vielzahl von verschiedenen Umgebungen eingesetzt, darunter auch in Datenbankmanagementsystemen. Mithilfe der Provenance wird erkenntlich, von wo (*where*) die Ursprungsdaten kommen, warum (*why*) und wie (*how*) gewisse Ergebnisdaten erzeugt wurden. Dadurch soll die Seriosität als auch die Reproduzierbarkeit von Ergebnissen sichergestellt werden.

Definition 2.30: Why-Provenance nach [CCT09]
Sei t ein Tupel aus dem Ergebnis der Anfrage Q. Die Why-Provenance eines Tupels t enthält alle Informationen, auch Zeugen genannt, mit denen das Tupel t produziert wurde. Diese Menge an Informationen nennt man auch Zeugenbasis.

Why-, How- und Where-Provenance stehen in Beziehung zueinander. Die Where-Provenance ist enthalten in der Why-Provenance. Wiederum kann die Why-Provenance von der How-Provenance abgeleitet werden [CCT09].

Beispiel 2.6: Provenance
Die Why-Provenance wird im späteren Verlauf der Arbeit von großer Wichtigkeit sein. Hierbei interessiert man sich jedoch nicht für die tupelspezifische Why-Provenance, sondern für die relationenspezifische Provenance. Man betrachte zuerst die in Beispiel 2.2 angegebene TGD. Diese TGD sagt aus, dass von der `REZENSIONEN`-Relation die `LIEST`-Sicht hergeleitet werden kann. Die Why-Provenance der `LIEST`-Sicht ist demnach $\{\{$`REZENSIONEN`$\}\}$. Hierbei gibt die `REZENSIONEN`-Relation an, warum die Sicht hergeleitet wurde.

Kapitel 3

Stand der Forschung und Technik

Dieses Kapitel beschäftigt sich mit den Erkenntnissen der Forschung und den praktischen Systemen, die das AQuV-Problem behandeln. Im ersten Abschnitt werden bestimmte Algorithmen der aktuellen Forschung näher beschrieben. Im darauffolgenden Abschnitt werden einige Systeme vorgestellt, in denen diese oder ähnliche Algorithmen implementiert sind.

3.1 Stand der Forschung

In diesem Abschnitt werden die theoretischen Lösungsansätze des AQuV-Problems diskutiert. Hierbei werden verschiedene Algorithmen vorgestellt, mit denen das AQuV-Problem gelöst werden kann. Zum Schluss wird zudem auf das CHASE&BACKCHASE-Verfahren mitsamt seinen Optimierungen eingegangen. Zur Illustration der behandelten Algorithmen werden diese anhand eines allgemeinen Beispiels beschrieben.

Beispiel 3.1: Answering-Queries-using-Views
Eine Website möchte den Titel und den Autor aller Bücher angeben, die mindestens eine Bewertung haben.

$$Q(vn, nn, titel) :- \overbrace{\texttt{REZENSIONEN}(lid, bid, bew, inh)}^{(1)}, \overbrace{\texttt{AUTOREN}(aid, vn, nn)}^{(2)}, \underbrace{\texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)}_{(3)}$$

Aus Gründen der Effizienz sollen hierbei nur die angegebenen materialisierten Sichten verwendet werden.

$$\begin{aligned}
&V_{\texttt{LIEST}}(lid_1, bid_1) :- \texttt{REZENSIONEN}(lid_1, bid_1, bew_1, inh_1)\\
&V_{\texttt{LIEST_REZ}}(lid_2, bid_2, bew_2, titel_2) :- \texttt{REZENSIONEN}(lid_2, bid_2, bew_2, inh_2),\\
&\qquad\qquad \texttt{BÜCHER}(bid_2, aid_2, titel_2, verl_2, ver_2, pr_2)\\
&V_{\texttt{BÜCHERLISTE}}(bid_3, aid_3, titel_3) :- \texttt{BÜCHER}(bid_3, aid_3, titel_3, verl_3, ver_3, pr_3)\\
&V_{\texttt{AUTORENLISTE}}(aid_4, vn_4, nn_4) :- \texttt{AUTOREN}(aid_4, vn_4, nn_4)\\
&V_{\texttt{LESERLISTE}}(lid_5, ben_5) :- \texttt{LESER}(lid_5, ben_5, pas_5, email_5)
\end{aligned}$$

3.1.1 Bucket-Algorithmus

Der Bucket-Algorithmus ist einer der älteren Algorithmen zum Beantworten von Anfragen über Sichten. Seinen Ursprung findet der Algorithmus im Artikel [LRO96]. Das Beantworten von Anfragen über Sichten erfolgt innerhalb von zwei Schritten:

1. Generierung von Buckets für jedes Teilziel der Anfrage
2. Generierung ausführbarer, äquivalenter Anfragen

In den folgenden Abschnitten werden beide Schritte genauer betrachtet.

Generierung der Buckets

Der Bucket-Algorithmus erstellt für jedes Teilziel der Anfrage einen Bucket. Ein Bucket für ein Teilziel R enthält alle Sichten, in denen mindestens ein Teilziel S der Sicht auf R abgebildet werden kann. Das Teilziel R ist also in den Sichten enthalten. Jede ausgezeichnete Variable des Teilziels S wird auf die korrespondierende Variable aus R abgebildet. Falls dies nicht möglich ist, so wird sie auf eine freie Variable abgebildet [LRO96].

Generierung von Anfragen

Durch das unkomplizierte Aneinanderbinden einer Sicht aus jedem Bucket können mögliche Rewritings der ursprünglichen Anfragen Q erstellt werden. Diese potentiellen Rewritings müssen nicht notwendigerweise äquivalent zu Q sein. Zum Überprüfen der Äquivalenz muss das Enthaltensein des Rewritings in Q beachtet werden [LRO96; Hal01].

Beispiel 3.2: Bucket-Algorithmus
Wie oben erwähnt, besteht der Bucket-Algorithmus aus zwei Phasen. Zuerst werden die Buckets erstellt, danach wird ein Containment-Mapping für jedes mögliche Rewriting der Anfrage durchgeführt.

Generierung der Buckets
Die Anfrage $Q(vn, nn, titel)$ aus Beispiel 3.1 besteht aus drei Teilzielen. Diese Teilziele wurden in der Anfrage mit den Ziffern (1), (2) und (3) gekennzeichnet. Der Bucket-Algorithmus generiert für jedes Teilziel einen Bucket, es werden demnach drei Buckets gebildet.

$\texttt{REZENSIONEN}(lid, bid, bew, inh)$
$V_{\texttt{LIEST}}\,(lid, bid)$
$V_{\texttt{LIEST_REZ}}\,(lid, bid, bew, t'_1)$

$\texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$
$V_{\texttt{BÜCHERLISTE}}\,(bid, aid, titel)$
$V_{\texttt{LIEST_REZ}}\,(t'_2, bid, t'_3, titel)$

$\texttt{AUTOREN}(aid, vn, nn)$
$V_{\texttt{AUTORENLISTE}}\,(aid, vn, nn)$

Der Bucket des Teilziels (1) enthält zwei Sichten. Man betrachte zuerst die Sicht $V_{\texttt{LIEST}}$. $V_{\texttt{LIEST}}$ enthält zwei ausgezeichnete Variablen, namentlich lid_1 und bid_1 (Beispiel 3.1). Man versucht nun, das entsprechende Teilziel der Sicht ($\texttt{REZENSIONEN}\,(lid_1, bid_1, bew_1, inh_1)$) auf das Teilziel der Anfrage abzubilden. Hierbei werden nicht ausgezeichnete Variablen auf neue und freie Variablen abgebildet. In diesem Beispiel ist das Teilziel der Anfrage in der Sicht enthalten. Die ausgezeichneten Sichtvariablen lid_1 und bid_1 werden respektiv auf die Variablen lid und bid abgebildet.
Die Sicht $V_{\texttt{LIEST_REZ}}$ ist etwas spannender. Die ausgezeichnete Variable $titel_2$ der Sicht wird von (1) nicht eingeschränkt. Es gibt demnach keine Abbildung von einem Attribut des Teilziels auf die Sicht selbst. Das Attribut $titel_2$ wird deshalb auf eine freie Variable t'_1 abgebildet.

Die Sicht $V_{\texttt{LESERLISTE}}$ ist in keinem Bucket enthalten. Die `LESER`-Relation kann auf kein Teilziel der Anfrage abgebildet werden.
Angenommen, es gäbe noch die Sicht $V_{\texttt{AUTORENLISTE}_2}(aid_6) :- \texttt{AUTOREN}(aid_6, vn_6, nn_6)$. Das Teilziel dieser Sicht kann nicht auf (2) abgebildet werden. Die nichtausgezeichneten Variablen vn_6 und nn_6 müssten dafür auf die ausgezeichneten Anfragevariablen vn und nn abgebildet werden. Folglich ist diese Sicht in keinem Bucket enthalten.

Query-Containment

Durch das Verbinden einer Sicht aus jedem Bucket können mögliche Rewritings gefunden werden. Die Anzahl an potentiellen Rewritings wird durch die Mächtigkeit des kartesischen Produkts der Buckets bestimmt. In diesem Beispiel gibt es $2 \cdot 2 \cdot 1 = 4$ mögliche Rewritings, namentlich:

1. $V_{\texttt{LIEST}}, V_{\texttt{AUTORENLISTE}}, V_{\texttt{BÜCHERLISTE}}$
2. $V_{\texttt{LIEST}}, V_{\texttt{AUTORENLISTE}}, V_{\texttt{LIEST_REZ}}$
3. $V_{\texttt{LIEST_REZ}}, V_{\texttt{AUTORENLISTE}}, V_{\texttt{BÜCHERLISTE}}$
4. $V_{\texttt{LIEST_REZ}}, V_{\texttt{AUTORENLISTE}}, V_{\texttt{LIEST_REZ}}$

Diese potentiellen Rewritings müssen nur noch auf das Enthaltensein in der Originalanfrage getestet werden.

Bewertung

Der Bucket-Algorithmus erstellt für jedes Teilziel in der Anfrage einen Bucket. Dadurch wird der Aufwand in der Kombinationsphase drastisch erhöht. Zudem werden Beziehungen zwischen Teilzielen, darunter Verbundattribute, nicht beachtet. Dies führt dazu, dass nicht relevante Sichten in die Buckets eingefügt werden. In der Kombinationsphase wird das kartesische Produkt der Buckets gebildet und jegliche potentiellen Rewritings werden auf das Enthaltensein in der Originalanfrage getestet. Je mehr Teilziele die Originalanfrage enthält, desto langwieriger ist diese Phase. Gerade für komplexe Anfragen mit einer Vielzahl von Teilzielen ist diese Methode nur bedingt geeignet.

3.1.2 MiniCon-Algorithmus

Der MiniCon-Algorithmus ähnelt dem Bucket-Algorithmus. Anders als beim Bucket-Algorithmus, werden hier zu jeder Sicht Buckets erstellt. Diese Buckets heißen MiniCon-Description (MCD) und enthalten mehr Informationen, als die Buckets vom Bucket-Algorithmus. Die enthaltenen Informationen der Buckets des Bucket-Algorithmus und der MCDs unterscheiden sich voneinander. Der MiniCon-Algorithmus besteht aus zwei Phasen: Zuerst werden die MCDs generiert, dann werden sie kombiniert.

Generierung von MCDs

Definition 3.1: MiniCon-Description (MCD) nach [PH01]
Sei Q eine Anfrage und $V(\vec{Z})$ eine Sicht mit den ausgezeichneten Variablen $\vec{Z}$. Eine MCD C für Q über V ist ein Quadrupel der Form $\left(h_C, V(\vec{Y})_C, \varphi_C, G_C\right)$, für das gilt:

- h_C ist ein Homomorphismus, der die Variablen aus V auf sich selbst abbildet. Nichtausgezeichnete Variablen werden dabei auf die identische Variable abgebildet

- $V(\vec{Y})_C = V(h_C(\vec{Z}))_C$ ist die Sicht V, in denen die ausgezeichneten Variablen $\vec{Z}$ mit den korrespondierenden Variablen aus h_C ausgetauscht wurden
- φ_C ist eine partielle Abbildung von den Variablen aus Q auf die Variablen aus V unter Anwendung von h_c
- G_C ist eine Menge von Teilzielen der Anfrage Q, die auf ein Teilziel aus V abgebildet werden können

Laut [PH01] ist eine MCD C für eine Anfrage Q über eine Sicht V nur von Interesse, wenn folgende Bedingungen gelten:

C1. Für jede ausgezeichnete Variable x aus Q, die durch φ_C abgebildet wird, ist $\varphi_C(x)$ eine ausgezeichnete Variable in $h_C(V)$.

C2. Falls $\varphi_C(x)$ eine nichtausgezeichneten Variable aus $h_C(V)$ ist, dann gilt für jedes Teilziel g aus Q, das x enthält, folgendes:

(a) alle Variablen aus g werden durch φ_C abgebildet

(b) $\varphi_C(g) \in h_C(V)$

Entgegen dem Bucket-Algorithmus, gilt für eine MCD zusätzlich die Klausel C2. Falls eine Variable x ein Verbundattribut aus der Anfrage Q ist, so muss die eingesetzte Sicht entweder das Verbundattribut ausgeben oder den Verbund selbst durchführen [PH01; DHI12].

Kombinierung von MCDs

Zum Erstellen von Rewritings müssen nur die Sichten aus den MCDs betrachtet werden. Ein Rewriting heißt redundanzfrei, falls die Menge von Teilzielen G der gewählten MCDs paarweise disjunkt sind und deren Vereinigung alle Teilziele der Anfrage enthält.

Beispiel 3.3: MiniCon-Algorithmus
Der MiniCon-Algorithmus besteht aus 2 Schritten: Zuerst werden die MCDs erstellt, dann werden die Sichten der MCDs kombiniert.

Generierung von MCDs

Das Beispiel 3.1 enthält 5 Sichten, demnach können 5 potentielle MCDs existieren. Wie schon beim Bucket-Algorithmus, wird auch hier die Sicht $V_{\texttt{LESERLISTE}}$ ignoriert, da kein Teilziel der Anfrage auf ein Teilziel der Sicht abgebildet werden kann.

MCDs

$V(\vec{Y})$	h	φ	G
$V_{\texttt{LIEST}}\,(lid_1, bid_1)$	$lid_1 \to lid_1, bid_1 \to bid_1$	$lid \to lid_1, bid \to bid_1$ $bew \to bew_1, inh \to inh_1$	(1)
$V_{\texttt{BÜCHERLISTE}}\,(bid_3, aid_3, titel_3)$	$bid_3 \to bid_3, aid_3 \to aid_3$ $titel_3 \to titel_3$	$bid \to bid_3, aid \to aid_3$ $titel \to titel_3, pr \to pr_3$ $verl \to verl_3, ver \to ver_3$	(3)
$V_{\texttt{AUTORENLISTE}}\,(aid_4, vn_4, nn_4)$	$aid_4 \to aid_4, vn_4 \to vn_4$ $nn_4 \to nn_4$	$aid \to aid_4, vn \to vn_4$ $nn \to nn_4$	(2)

Innerhalb der Anfrage gibt es mehrere Verbundattribute, namentlich *bid* und *aid*. Anders als beim Bucket-Algorithmus, müssen die Sichten beim MiniCon-Algorithmus auch die Eigenschaft

C2 erfüllen. Die LIEST_REZ-Sicht enthält sowohl das Verbundattribut bid_2 als auch aid_2. Diese Sicht realisiert den Verbund von (1) und (3), der Verbund zwischen (2) und (3) wird jedoch nicht abgedeckt. Folglich muss aid_2 eine ausgezeichnete Variable sein, aber auch dies ist nicht der Fall. Für die Sicht $V_{\texttt{LIEST_REZ}}$ gibt es demnach keine MCD von Interesse.
Man betrachte nun die Sicht $V_{\texttt{LIEST}}$. Sie enthält das Verbundattribut bid_1, nicht aber den Verbund selbst. Da die Variable bid_1 jedoch ausgezeichnet ist, sind alle Eigenschaften erfüllt und es existiert eine MCD für diese Sicht.

Kombinierung von MCDs

Unter Betrachtung der generierten MCDs ist nur $V_{\texttt{LIEST}}, V_{\texttt{BÜCHERLISTE}}, V_{\texttt{AUTORENLISTE}}$ ein mögliches Rewriting. Für die ausgewählten MCDs gilt:

$$G_{\texttt{LIEST}} \cap G_{\texttt{BÜCHERLISTE}} = G_{\texttt{LIEST}} \cap G_{\texttt{AUTORENLISTE}} = G_{\texttt{BÜCHERLISTE}} \cap G_{\texttt{AUTORENLISTE}} = \varnothing$$
$$G_{\texttt{LIEST}} \cup G_{\texttt{BÜCHERLISTE}} \cup G_{\texttt{AUTORENLISTE}} = \{(1), (2), (3)\}$$

Folglich ist $Q' :- V_{\texttt{LIEST}}, V_{\texttt{BÜCHERLISTE}}, V_{\texttt{AUTORENLISTE}}$ ein redundanzfreies Rewriting.

Bewertung

Der MiniCon-Algorithmus gibt für jede Anfrage die zugehörigen redundanzfreien Rewritings wieder. Dabei wird für jede Sicht eine MCD erstellt. MCDs enthalten mehr Informationen, als die Buckets vom Bucket-Algorithmus. Der Aufwand in der Kombinierungsphase wird dadurch drastisch reduziert. Es sind nur die MCDs von Interesse, die die erwähnten Eigenschaften C1 und C2 erfüllen. Gerade durch Klausel C2 werden die Beziehungen von Teilzielen durch Verbundattribute beachtet. Eben diese Restriktionen können zu einer Reduktion der berechneten Rewritings führen. Dabei können mögliche Rewritings verloren gehen.

3.1.3 CHASE&BACKCHASE

Der CHASE&BACKCHASE-Algorithmus erweitert den ursprünglichen CHASE-Algorithmus um eine BACKCHASE-Phase. Wie die vorigen Algorithmen, ermöglicht auch dieser Algorithmus das Beantworten von Anfragen unter Verwendung von Sichten. Der Algorithmus wurde ursprünglich im Kapitel [DH13] vorgestellt. Das Berechnen äquivalenter Rewritings einer Anfrage wird dabei in zwei Phasen bewerkstelligt, die CHASE- und die BACKCHASE-Phase.

CHASE

Beim AQuV-Problem gibt es zwei Arten von Anforderungen: Integritäts- und Sichtbedingungen. Wie in Abschnitt 2.1.2 schon erläutert, geben Integritätsbedingungen Anforderungen an, die eine Datenbank erfüllen muss. Sichtbedingungen geben die Definition der Sichten an. Wie beim CHASE-Algorithmus üblich, haben die Anforderungen die Form einer EGD oder TGD. Sichtbedingungen sind spezielle TGDs, sogenannte s-t-TGDs. Der CHASE-Algorithmus chaset die Anfrage mit den eben erwähnten Anforderungen [DH13].

Definition 3.2: Universalplan nach [DH13]
Sei Q eine Anfrage. Zudem seien Sichtabhängigkeiten $\mathcal{V}$ und Integritätsabhängigkeiten $\mathcal{I}$ gegeben. Sei Q' die mit $\mathcal{V} \cup \mathcal{I}$ gechasete Anfrage Q. Der Universalplan $\mathcal{U}$ ist eine Anfrage, die nur die Sichten der Anfrage Q' verwendet.

BACKCHASE

Der BACKCHASE ist eine Erweiterung des CHASE. Neben der CHASE-Phase, in der Abhängigkeiten in ein CHASE-Objekt eingearbeitet werden, versucht man in der BACKCHASE-Phase durch Anwendung des CHASE, das CHASE-Ergebnis zu optimieren und zu validieren. Eine Anwendung für die BACKCHASE-Phase ist das AQuV- oder das Datenintegrationsproblem [AH19]. Bei Anfragen auf Sichten versucht man zuerst, mit dem CHASE-Algorithmus die Anfrage zu chasen. Danach wird der Universalplan des CHASE-Ergebnis generiert. Dieser Universalplan enthält sehr viele möglicherweise redundante Sichten. Darüber hinaus kann es sogar möglich sein, dass die Sichten nicht ausreichen, um die Anfrage zu beantworten. In der BACKCHASE-Phase wird jede Teilanfrage des Universalplans mit den inversen Sichtbedingungen und den Integritätsbedingungen gechaset. Dadurch werden alle möglichen Rewritings gefunden. Mithilfe dieser Technik kann die minimale Sichtmenge, die zum Beantworten der Anfrage benötigt wird, berechnet werden [DH13]. Eine detailliertere Beschreibung des CHASE&BACKCHASE findet sich im Konzept-Kapitel wieder.

Beispiel 3.4: CHASE&BACKCHASE

CHASE

Im ersten Schritt müssen die Sichten aus Beispiel 3.1 als s-t-TGDs dargestellt werden. Da fünf Sichten existieren, besteht die Menge $\mathcal{V}$ aus fünf s-t-TGDs.

$$C_{V_{\texttt{LIEST}}} : \forall lid, bid, bew, inh : \texttt{REZENSIONEN}\,(lid, bid, bew, inh) \rightarrow V_{\texttt{LIEST}}\,(lid, bid)$$

$$C_{V_{\texttt{LIEST_REZ}}} : \forall lid, bid, bew, inh, aid, titel, verl, ver, pr : \texttt{REZENSIONEN}\,(lid, bid, bew, inh)\,, \texttt{BÜCHER}\,(bid, aid, titel, verl, ver, pr) \rightarrow V_{\texttt{LIEST_REZ}}\,(lid, bid, bew, titel)$$

$$C_{V_{\texttt{BÜCHERLISTE}}} : \forall bid, aid, titel, verl, ver, pr : \texttt{BÜCHER}\,(bid, aid, titel, verl, ver, pr) \rightarrow V_{\texttt{BÜCHERLISTE}}\,(bid, aid, titel)$$

$$C_{V_{\texttt{AUTORENLISTE}}} : \forall aid, vn, nn : \texttt{AUTOREN}\,(aid, vn, nn) \rightarrow V_{\texttt{AUTORENLISTE}}\,(aid, vn, nn)$$

$$C_{V_{\texttt{LESERLISTE}}} : \forall lid, ben, pas, email : \texttt{LESER}\,(lid, ben, pas, email) \rightarrow V_{\texttt{LESERLISTE}}\,(lid, ben)$$

Für die Sichtbedingung $C_{V_{\texttt{LESERLISTE}}}$ gibt es keinen aktiven Trigger. Grund dafür ist, dass die LESER-Relation in der Anfrage als auch in den Integritätsbedingungen nicht vorkommt. Dafür gibt es für die restlichen Anforderungen genau einen aktiven Trigger. Der Rumpf der übrigen Sichtbedingungen lässt sich auf mindestens ein Teilziel der Anfrage Q abbilden. Insgesamt existieren demnach vier aktive Trigger für die Sichtbedingungen $C_{V_{\texttt{LIEST}}}, C_{V_{\texttt{LIEST_REZ}}}, C_{V_{\texttt{BÜCHERLISTE}}}$ und $C_{V_{\texttt{AUTORENLISTE}}}$.

$$\texttt{REZENSIONEN}\,(lid, bid, bew, inh)\,, \texttt{AUTOREN}\,(aid, vn, nn)\,, \texttt{BÜCHER}\,(bid, aid, titel, verl, ver, pr)$$
$$\Downarrow C_{V_{\texttt{LIEST}}}$$
$$\texttt{REZENSIONEN}\,(lid, bid, bew, inh)\,, \texttt{AUTOREN}\,(aid, vn, nn)\,, \texttt{BÜCHER}\,(bid, aid, titel, verl, ver, pr)$$
$$V_{\texttt{LIEST}}\,(lid, bid)$$
$$\Downarrow C_{V_{\texttt{LIEST_REZ}}}$$
$$\texttt{REZENSIONEN}\,(lid, bid, bew, inh)\,, \texttt{AUTOREN}\,(aid, vn, nn)\,, \texttt{BÜCHER}\,(bid, aid, titel, verl, ver, pr)$$
$$V_{\texttt{LIEST}}\,(lid, bid)\,, V_{\texttt{LIEST_REZ}}\,(lid, bid, bew, titel)$$
$$\Downarrow C_{V_{\texttt{BÜCHERLISTE}}}$$
$$\texttt{REZENSIONEN}\,(lid, bid, bew, inh)\,, \texttt{AUTOREN}\,(aid, vn, nn)\,, \texttt{BÜCHER}\,(bid, aid, titel, verl, ver, pr)\,,$$
$$V_{\texttt{LIEST}}\,(lid, bid)\,, V_{\texttt{LIEST_REZ}}\,(lid, bid, bew, titel)\,, V_{\texttt{BÜCHERLISTE}}\,(bid, aid, titel)$$

$$\texttt{REZENSIONEN}(lid, bid, bew, inh), \texttt{AUTOREN}(aid, vn, nn), \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr),$$
$$V_{\texttt{LIEST}}(lid, bid), V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel)$$
$$\Downarrow C_{V_{\texttt{AUTORENLISTE}}}$$
$$\texttt{REZENSIONEN}(lid, bid, bew, inh), \texttt{AUTOREN}(aid, vn, nn), \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr),$$
$$V_{\texttt{LIEST}}(lid, bid), V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel),$$
$$V_{\texttt{AUTORENLISTE}}(aid, vn, nn)$$

Nach der Anwendung der aktiven Trigger erhält man die gechasete Anfrage:

$$\begin{aligned} Q' = \text{chase}_{\mathcal{V}}(Q) :- & \texttt{REZENSIONEN}(lid, bid, bew, inh), \texttt{AUTOREN}(aid, vn, nn) \\ & \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr), V_{\texttt{LIEST}}(lid, bid) \\ & V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel) \\ & V_{\texttt{AUTORENLISTE}}(aid, vn, nn) \end{aligned}$$

Der Universalplan dieser Anfrage sieht dann wie folgt aus:

$$\begin{aligned} \mathcal{U} = & V_{\texttt{LIEST}}(lid, bid), V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel), \\ & V_{\texttt{AUTORENLISTE}}(aid, vn, nn) \end{aligned}$$

BACKCHASE
Anhand dieses Universalplans können $2^4 - 1 = 15$ verschiedene Teilpläne gestellt werden. Voraussetzung für die Anwendung des BACKCHASE ist das Invertieren der Sichtbedingungen. Da die Sicht $V_{\texttt{LESERLISTE}}$ im Universalplan nicht auftaucht und auch nicht hergeleitet werden kann, muss die Inverse dieser Sichtbedingung nicht beachtet werden.

$$B_{V_{\texttt{LIEST}}} : \forall lid, bid : V_{\texttt{LIEST}}(lid, bid) \rightarrow \exists bew, inh : \texttt{REZENSIONEN}(lid, bid, bew, inh)$$
$$\begin{aligned} B_{V_{\texttt{LIEST_REZ}}} : \forall lid, bid, bew, titel : V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel) \rightarrow & \exists inh, aid, verl, ver, pr : \\ & \texttt{REZENSIONEN}(lid, bid, bew, inh), \\ & \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr) \end{aligned}$$
$$\begin{aligned} B_{V_{\texttt{BÜCHERLISTE}}} : \forall bid, aid, titel : V_{\texttt{BÜCHERLISTE}}(bid, aid, titel) \rightarrow & \exists verl, ver, pr : \\ & \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr) \end{aligned}$$
$$B_{V_{\texttt{AUTORENLISTE}}} : \forall aid, vn, nn : V_{\texttt{AUTORENLISTE}}(aid, vn, nn) \rightarrow \texttt{AUTOREN}(aid, vn, nn)$$

Mithilfe der eben erstellten inversen Sichtbedingungen $\mathcal{V}^{-1}$ kann der CHASE-Algorithmus nun auf die 15 Teilpläne des Universalplans ausgeführt werden.
Man betrachte zuerst den Teilplan $V_{\texttt{LIEST_REZ}}, V_{\texttt{AUTORENLISTE}}$.

$$\begin{aligned} & \texttt{chase}_{\mathcal{V}^{-1}}(V_{\texttt{LIEST_REZ}}, V_{\texttt{AUTORENLISTE}}) = \texttt{REZENSIONEN}(lid, bid, bew, inh_1), \\ & \texttt{BÜCHER}(bid, aid_2, titel, verl_2, ver_2, pr_2), \texttt{AUTOREN}(aid, vn, nn) \end{aligned}$$

Wie der MiniCon-Algorithmus schon gezeigt hat, ist diese Teilanfrage kein Rewriting. Auch der CHASE-Algorithmus kommt auf diesen Entschluss. Die berechnete Ergebnisanfrage ist nicht äquivalent zur Ursprungsanfrage. Der Verbund zwischen der BÜCHER- und AUTOREN-Relation ist nicht enthalten.

Nun betrachtet man den Teilplan $V_{\texttt{LIEST}}, V_{\texttt{BÜCHERLISTE}}, V_{\texttt{AUTORENLISTE}}$.

$$\text{chase}_{\mathcal{V}^{-1}}(V_{\texttt{LIEST}}, V_{\texttt{BÜCHERLISTE}}, V_{\texttt{AUTORENLISTE}}) = \texttt{REZENSIONEN}(lid, bid, bew_1, inh_1),$$
$$\texttt{BÜCHER}(bid, aid, titel, verl_2, ver_2, pr_2), \texttt{AUTOREN}(aid, vn, nn)$$

Es ist deutlich erkennbar, dass die Ergebnisanfrage ein Rewriting der Ursprungsanfrage ist. Es gibt weitere Rewritings, nämlich $V_{\texttt{BÜCHERLISTE}}, V_{\texttt{AUTORENLISTE}}, V_{\texttt{LIEST_REZ}}$ und $V_{\texttt{LIEST}}, V_{\texttt{BÜCHERLISTE}}, V_{\texttt{AUTORENLISTE}}, V_{\texttt{LIEST_REZ}}$. Da letzteres eine Oberanfrage der eben berechneten Ergebnisanfrage ist, muss es auch ein Rewriting sein.
Die weiteren Berechnungen sind im Anhang A.1 näher beschrieben.

Pruning

In der BACKCHASE-Phase werden eine Vielzahl an Teilanfragen gechaset. Um diese Menge an Anfragen zu reduzieren, wurde das Pruning eingeführt. Beim chasen der Teilpläne müssen nicht zwingend alle Pläne von Relevanz sein. Falls eine Teilanfrage schon ein Rewriting der Originalanfrage enthält, so ist sie auch ein Rewriting. Das Auswerten dieser Teilanfrage bietet somit keinen Mehrwert. Es dürfen nicht alle Teilanfragen ignoriert werden, die nicht alle Relationen der Anfrage enthalten. Falls neben den Sichtbedingungen auch TGDs im CHASE betrachtet werden, so kann ein Rewriting auch weniger Relationen als die Originalanfrage abdecken [DH13]. Gerade dieses Problem wird im folgenden Beispiel verdeutlicht.

Beispiel 3.5: Pruning
Die Anfrage $Q(bid) :- \texttt{REZENSIONEN}(lid, bid, bew, inh), \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$ soll nur durch Sichten beantwortet werden. Hierbei sind folgende Sicht- und Integritätsbedingungen gegeben:

$$C_{V_{\texttt{REZ}}} : \forall lid, bid, bew, inh : \texttt{REZENSIONEN}(lid, bid, bew, inh) \rightarrow V_{\texttt{REZ}}(bid)$$
$$C_{V_{\texttt{BUCH}}} : \forall bid, aid, titel, verl, ver, pr : \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr) \rightarrow V_{\texttt{BUCH}}(bid)$$
$$C_I : \forall lid, bid, bew, inh : \texttt{REZENSIONEN}(lid, bid, bew, inh) \rightarrow \exists aid, titel, verl, ver, pr : \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$$

Es ist direkt erkennbar, dass der Universalplan aus den Sichten $V_{\texttt{REZ}}$ und $V_{\texttt{BUCH}}$ besteht. Ohne Beachtung der Integritätsbedingung wäre nur die Anfrage $Q' :- V_{\texttt{BUCH}}, V_{\texttt{REZ}}$ ein Rewriting der Originalanfrage. Mithilfe der Fremdschlüsselbeziehungen C_I kann aber allein aus der Sicht $V_{\texttt{REZ}}$ die BÜCHER-Relation abgeleitet werden. Folglich reicht $V_{\texttt{REZ}}$ aus, um die Originalanfrage zu bilden.
Die weiteren Berechnungen sind im Anhang A.2 näher beschrieben.

Provenance-Directed CHASE&BACKCHASE

Der CHASE&BACKCHSAE-Algorithmus verarbeitet eine Vielzahl an Anfragen. Selbst mithilfe des Prunings bleibt die Anzahl an Vergleichen enorm. Abhilfe schafft hier der Provenance-Directed CHASE&BACKCHASE-Algorithmus (PD-CHASE&BACKCHASE). Der Unterschied zum CHASE&BACKCHASE-Algorithmus liegt in der BACKCHASE-Phase. Anstelle der isolierenden Betrachtung aller Teilanfragen wird hier der gesamte Universalplan gechaset. Dabei wird die Why-Provenance jedes Teilziels notiert.

Beispiel 3.6: Provenance-Directed CHASE&BACKCHASE
Man betrachte den Universalplan

$$\mathcal{U} = V_{\texttt{LIEST}}(lid, bid), V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel), V_{\texttt{AUTORENLISTE}}(aid, vn, nn)$$

aus Beispiel 3.4. Mittels der bereits bekannten inversen Sichtbedingungen führt man den modifizierten CHASE-Algorithmus auf $\mathcal{U}$ aus.

$$\begin{aligned}\mathcal{U}' = chase_{\mathcal{V}^{-1}}(\mathcal{U}) = {} & \overbrace{V_{\texttt{LIEST}}(lid, bid)}^{V_{\texttt{LIEST}}}, \overbrace{V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel)}^{V_{\texttt{LIEST_REZ}}}, \overbrace{V_{\texttt{BÜCHERLISTE}}(bid, aid, titel)}^{V_{\texttt{BÜCHERLISTE}}} \\ & \overbrace{V_{\texttt{AUTORENLISTE}}(aid, vn, nn)}^{V_{\texttt{AUTORENLISTE}}}, \overbrace{\texttt{REZENSIONEN}(lid, bid, bew, inh_1)}^{V_{\texttt{LIEST}} \vee V_{\texttt{LIEST_REZ}}}, \\ & \overbrace{\texttt{BÜCHER}(bid, aid_3, titel, verl_3, ver_3, pr_3)}^{V_{\texttt{LIEST_REZ}}}, \overbrace{\texttt{BÜCHER}(bid, aid, titel, verl_4, ver_4, pr_4)}^{V_{\texttt{BÜCHERLISTE}}}, \\ & \overbrace{\texttt{AUTOREN}(aid, vn, nn)}^{V_{\texttt{AUTORENLISTE}}}\end{aligned}$$

Nun werden Homomorphismen gesucht, mit denen die Anfrage Q aus Beispiel 3.1 auf $\mathcal{U}'$ abgebildet werden kann. Die Provenance der verwendeten Atomen aus $\mathcal{U}'$ gibt dann die Rewritings von Q an. Ein mögliches Rewriting wäre beispielsweise $V_{\texttt{LIEST}}, V_{\texttt{BÜCHERLISTE}}, V_{\texttt{AUTORENLISTE}}$ mit dem Homomorphismus

$$\{lid \to lid, bid \to bid, bew \to bew, inh \to inh_1, aid \to aid, vn \to vn, nn \to nn, aid \to aid, titel \to titel, verl \to verl_4, ver \to ver_4, pr \to pr_4\}$$

Bewertung

Der CHASE&BACKCHASE-Algorithmus kann gerade durch seine Optimierungen sehr viel Zeit in der BACKCHASE-Phase einsparen. Falls Fremdschlüsselbeziehungen vorhanden sind, so können sogar Rewritings gefunden werden, die weniger Relationen als die Originalanfrage verwenden. Falls der CHASE terminiert und die Integritätsbedingungen nur auf TGDs beschränkt sind, so ist der Provenance-Directed CHASE&BACKCHASE korrekt und vollständig [DH13].

3.1.4 Vergleich der Algorithmen

Alle erwähnten Algorithmen beschränken sich auf konjunktive Anfragen. Der Bucket-Algorithmus ist vom Aufbau her der einfachste Algorithmus, dies spiegelt sich auch in der Laufzeit wider. Der MiniCon-Algorithmus ähnelt in der Anfangsphase dem Bucket-Algorithmus. Aufgrund des Aufbaus der MCDs und den Eigenschaften, die sie erfüllen müssen, ist der MiniCon-Algorithmus sehr effizient. Wegen den Eigenschaften von MCDs können mögliche Rewritings nicht gefunden werden. Der einfache CHASE&BACKCHASE-Algorithmus ist weit weniger performant, als die eben erwähnten Algorithmen. In der BACKCHASE-Phase müssen alle Teilanfragen des Universalplans gechaset werden, dies sind exponentiell viele CHASE-Schritte. Abhilfe schafft hier der CHASE&BACKCHASE-Algorithmus unter Beachtung der Provenance. Die Laufzeit wird hierbei drastisch reduziert. Der CHASE&BACKCHASE-Algorithmus hat noch einen weiteren Vorteil: unter Beachtung von speziellen Abhängigkeiten, darunter auch Fremdschlüsselbeziehungen, müssen die Sichten eines Rewritings nicht alle Teilziele der Anfrage enthalten. Weder der Bucket-Algorithmus noch der MiniCon-Algorithmus können solche Rewritings generieren. Hierbei sei noch zu erwähnen, dass außer der CHASE&BACKCHASE-Algorithmus, keiner der erwähnten Algorithmen Integritätsbedingungen bei dem Erstellen von Rewritings beachtet.

3.2 Stand der Technik

In diesem Abschnitt werden einige Systeme vorgestellt, die sich mit dem AQuV-Problem oder verwandten Problemen auseinandersetzen.

3.2.1 Bucket-Algorithmus

Der Bucket-Algorithmus aus [LRO96] wurde als Prototyp in das *Information-Manifold*-System integriert. Mithilfe dieses Systems können Anfragen an verschiedenen Informationsquellen, darunter auch relationale Datenbanken, gestellt werden. Die Funktionsweise dieses Systems wurde in einigen Artikeln sorgfältig beschrieben. Leider konnte keine öffentlich zugängliche Implementierung dieses Systems gefunden werden.

3.2.2 MiniCon-Algorithmus

Zur Veranschaulichung des MiniCon-Algorithmus wurden im Artikel [PH01] mehrere Prototypen programmiert und getestet. Die Prototypen basierten dabei auf dem MiniCon-Algorithmus oder einer Erweiterung des MiniCon-Algorithmus (MiniCon-IP). Der MiniCon-IP-Algorithmus erlaubt, im starken Kontrast zum ursprünglichen Algorithmus, auch die Vergleichsoperatoren $\leq, <, >, \geq$. Die Laufzeit der Algorithmen unterscheidet sich dabei kaum. Auch wenn es nicht sonderlich beeindruckend ist, finden die Prototypen des MiniCon-Algorithmus in fast jeder Anfragen schneller Rewritings als der Bucket-Algorithmus [PH01]. Die Verwendung des Algorithmus in praktischen Systemen ist derzeit nicht bekannt. Da viele Unternehmen keine Auskunft über die verwendeten Algorithmen in ihren Systemen geben, ist es aber nicht ausgeschlossen, dass der MiniCon-Algorithmus im Datenintegrationsbereich verwendet wird. Wie beim vorigen System, konnte auch hier keine öffentlich zugängliche Implementierung gefunden werden.

3.2.3 CHASE&BACKCHASE

In diesem Abschnitt werden einige Systeme vorgestellt, die auf Grundlage des CHASE-Algorithmus Anfragen umschreiben können. Im Studentenprojekt [Aug+20] der Universität Rostock wurden die einzelnen Anwendungen detaillierter betrachtet und verglichen.

ProvCB

Der Prototyp ProvCB[1] aus dem Kapitel [Ile+14] ist eine Java-Anwendung, mit der Sichten in Anfragen eingearbeitet werden können. Der $\text{Prov}_{C\&B}$-Algorithmus berechnet hierbei zuerst den Universalplan der Anfrage. Danach wird der Universalplan gechaset und die Provenance der Atome notiert. Mithilfe von Containment-Mappings wird dann die Provenance der Anfrage bestimmt. Da nur minimale Rewritings von Interesse sind, wird die berechnete Provenance der Anfrage minimalisiert. Jedes Konjunkt dieser Provenance ist ein Rewriting der ursprünglichen Anfrage. ProvCB unterstützt sowohl EGDs als auch TGDs.

PDQ

Das System PDQ[2] (Proof-Driven Querying) ist eine Java-Anwendung zur semantischen Optimierung von Anfragen. Unter Angabe von Selektions-Projektions-Verbund-Anfragen und Schemata werden Anfragepläne berechnet. Der Plan mit den geringsten Kosten wird dann ausgegeben. Das übergebene Schemata enthält die genutzten Relationen und die Integritätsbedingungen. Die Integritätsbedingungen können hierbei die Form einer EGD oder einer TGD annehmen.

[1] `https://github.com/IoanaIleana/ProvCB`
[2] `https://github.com/ProofDrivenQuerying/pdq`

3.2.4 ChaTEAU

ChaTEAU (Chase for Transforming, Evolving, and Adapting databases and queries, Universal approach) ist eine Java-Anwendung, die mithilfe des CHASE-Algorithmus TGDs und EGDs in Anfragen und Instanzen einarbeiten kann. Das Ziel von ChaTEAU ist es, möglichst viele Anwendungsfälle des CHASE-Algorithmus abzudecken. Die Grundlagen dieses Systems wurden hierbei in der Masterarbeit [Mar18] gelegt. Die Struktur von ChaTEAU orientiert sich am ProvCB-System. Im Zuge verschiedener studentischer Arbeiten wurde dieses Projekt erweitert. Die Bachelorarbeit [Ren19] erweiterte ChaTEAU um eine dynamische Ein- und Ausgabe mittels XML-Dateien. Falls der CHASE auf Instanzen angewendet wird, so wird zudem eine CSV-Datei mit der Ergebnisinstanz generiert. Jegliche Arbeitsschritte der CHASE-Phase von ChaTEAU werden in einer Logdatei protokolliert. Neben der dynamischen Ein- und Ausgabe wurde auch der CHASE auf Anfragen teilweise umgesetzt. Die Bachelorarbeit [Zim19] vereinheitlichte den CHASE auf Instanzen und Anfragen. Neben der Optimierung des CHASE auf Anfragen wurden zudem s-t-TGDs eingeführt, die gerade für Sichten von Interesse sind. Die Masterarbeit [Gör20] erweiterte die CHASE-Phase von ChaTEAU um ein Terminierungskriterium. Gerade im Forschungsbereich ist die Reproduzierbarkeit der Ergebnisse einer Anfrage maßgeblich. Mithilfe der Masterarbeit [Ros20] konnte die Provenance einer Anfrage, also die Tupel, die diese Anfrage nutzte, ermittelt und gespeichert werden. Die Bestimmung dieser Provenance wird durch den CHASE&BACKCHASE-Algorithmus erreicht. In der Masterarbeit [Zim21] wurde ChaTEAU um das GaLVE-Verfahren (Global-as-Local-View-Extension) erweitert. Mithilfe von GaLVE können Informationen des globalen Schemas auf die lokalen Schemata abgebildet werden. Die lokalen Schemata werden demnach durch das globale Schema erweitert. Dies wird häufig in der Datenintegration verwendet.

Kapitel 4

Konzept

In diesem Kapitel wird das Konzept der Arbeit vorgestellt, mit dem Anfragen nur unter Verwendung von Sichten beantwortet werden können. Die Lösung des AQuV-Problems wird hierbei in zwei Phasen unterteilt, die CHASE- und die BACKCHASE-Phase.

4.1 CHASE-Phase

Bevor eine Anfrage unter Verwendung von Sichten beantwortet werden kann, müssen zuerst die relevanten Sichten bestimmt werden. Mithilfe des CHASES werden diese Sichten, die als s-t-TGDs spezifiziert wurden, in die Anfrage eingearbeitet. Die gechasete Anfrage enthält verschiedene relationale Atome. Für den weiteren Verlauf des CHASE&BACKCHASE sind die Atome, die von einer s-t-TGD hergeleitet wurden, von hoher Bedeutung. Hierfür wird eine neue Anfrage generiert, der Universalplan. Diese Anfrage enthält alle Sichten aus der gechaseten Anfrage.

4.2 BACKCHASE-Phase

Die BACKCHASE-Phase besteht aus mehreren Schritten. Im ersten Schritt werden die inversen Sichtbedingungen berechnet. Im zweiten Schritt müssen die potentiellen Rewritings bestimmt werden. Mithilfe der potentiellen Rewritings und dem CHASE-Algorithmus können dann die Rewritings berechnet werden.

4.2.1 Invertierung der Sichtbedingungen

Damit eine Anfrage auf Basisrelationen über Sichten beantwortet werden kann, müssen die Sichtbedingungen zuerst invertiert werden [DH13]. Bei der Invertierung von Sichtbedingungen wird der Kopf und der Rumpf dieser Bedingungen vertauscht. Wie bei einer TGD üblich, müssen alle Variablen im Kopf der invertierten Sichtbedingung allquantifiziert werden. Jede nicht gebundene Variable im Rumpf der Bedingung wird zudem existenzquantifiziert. Neben Variablen können Sichtbedingungen zudem auch Konstanten enthalten. Hierbei ist zu beachten, dass der Wert einer Konstante schon festgelegt ist. Die Quantifizierung einer Konstante ist deswegen nicht zulässig.

Beispiel 4.1: Invertierung
Sei `BUCH_TOP` eine Sicht, die alle BuchIDs enthält, die mindestens einmal die höchste Bewertung erhalten haben. In der folgenden Sichtbedingung sind die Variablen $lid, bid, inh, aid, titel, verl, ver$ und pr enthalten. Zudem wurde eine Konstante 10.0 definiert, die die Bewertung des rezensierten

Buches angibt.

$$C_{V_{\texttt{BUCH_TOP}}}: \forall lid, bid, inh, aid, titel, verl, ver, pr : \texttt{REZENSIONEN}(lid, bid, 10.0, inh), \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr) \rightarrow V_{\texttt{BUCH_TOP}}(bid)$$

Beim Invertieren dieser Sichtbedingung wird der Rumpf und der Kopf vertauscht. Der Rumpf der invertierten s-t-TGD enthält nun die `BUCH_TOP`-Sicht. Wie bei einer TGD üblich, wird die Variable *bid* allquantifiziert. Im Kopf der Sichtbedingung werden alle Variablen außer *bid* existenzquantifiziert. Die Konstante 10.0 wird hierbei nicht geändert.

$$B_{V_{\texttt{BUCH_TOP}}}: \forall bid: V_{\texttt{BUCH_TOP}}(bid) \rightarrow \exists lid, inh, aid, titel, verl, ver, pr : \texttt{REZENSIONEN}(lid, bid, 10.0, inh), \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$$

4.2.2 Standard-BACKCHASE

Der Standard-BACKCHASE chaset alle Teilpläne des Universalplans mit den inversen Sichtbedingungen und den Integritätsbedingungen. Falls ein Homomorphismus von der Originalanfrage zum Ergebnis des BACKCHASE existiert, so wurde ein Rewriting gefunden. Der Aufwand des Standard-BACKCHASE ist exponentiell. Die Anzahl an gechaseten Anfragen ist äquivalent zu der Anzahl aller Teilpläne des Universalplans. Wenn der Universalplan aus x Sichten besteht, so müssten $2^x - 1$ Teilanfragen betrachtet werden. Die leere Anfrage wird dabei nicht betrachtet. Dieser Vorgang ist sehr aufwendig. Mithilfe von Optimierungen kann die Effizienz des BACKCHASE erhöht werden. Um zu überprüfen, ob Rewritings existieren, müssen nicht alle Teilpläne des Universalplans gechaset werden. Aufgrund der Vollständigkeit des CHASE&BACKCHASE-Algorithmus muss nur überprüft werden, ob der Universalplan ein Rewriting der Originalanfrage ist. Falls dies nicht der Fall ist, so können auch die Teilpläne des Universalplans keine Rewritings sein. Der Standard-BACKCHASE kann somit relativ effizient überprüfen, ob die Anfrage Rewritings besitzt [DH13].

4.2.3 Provenance-Directed CHASE&BACKCHASE

Die BACKCHASE-Phase des Provenance-Directed CHASE&BACKCHASE-Algorithmus unterscheidet sich grundlegend von den vorigen BACKCHASE-Variationen. Mithilfe der Why-Provenance der Atome des gechaseten Universalplans wird versucht, alle Rewritings zu bestimmen. Anders als in den vorigen Phasen ist die Anzahl an gechaseten Plänen nicht mehr exponentiell. Die BACKCHASE-Phase des Provenance-Directed CHASE&BACKCHASE besteht aus zwei Schritten. Im ersten Schritt wird der Universalplan provenance-bewusst gechaset (provenance-aware chase). Im zweiten Schritt wird die Why-Provenance der Originalanfrage bestimmt.
Der provenance-bewusste CHASE berechnet die Why-Provenance jedes Atoms des gechaseten Universalplans. Damit der BACKCHASE auch unter Verwendung von TGDs funktioniert, muss der Test auf aktive Trigger erweitert werden.
In den nachfolgenden Abschnitten werden TGDs und s-t-TGDs gleichbedeutend betrachtet.

Provenance-bewusste aktive Trigger

Mithilfe von aktiven Triggern wird entschieden, welche TGD im CHASE angewendet wird. Der Test auf provenance-bewusste aktive Trigger (provenance-aware active trigger) erweitert den ursprünglichen Test auf aktive Trigger. Sei ein Homomorphismus für eine TGD ein Trigger, so gibt es einen Homomorphismus vom Rumpf der TGD in das CHASE-Objekt. Nun wird überprüft, ob dieser Trigger aktiv ist. Falls keine Erweiterung des Homomorphismus existiert, mit dem der Kopf der TGD in das CHASE-Objekt abgebildet werden kann, so ist der Homomorphismus ein provenance-bewusster aktiver Trigger. Falls es

eine Erweiterung gibt, so wird die Why-Provenance der betroffenen Atome betrachtet. Falls die Why-Provenance des Rumpfes der TGD unter Anwendung des Homomorphismus die Why-Provenance des Kopfes unter Anwendung des erweiterten Homomorphismus nicht impliziert, so ist der Trigger provenance-bewusst aktiv. Im anderen Fall existiert kein provenance-bewusster aktiver Trigger [DH13].
Die Why-Provenance $P_2 = \bigvee_{i=1}^{m} C_i^2$ der Rumpf-Atome impliziert die Why-Provenance $P_1 = \bigvee_{i=1}^{n} C_i^1$ der Kopf-Atome, falls P_1 der Provenance P_2 untergeordnet ist. Dies ist genau dann der Fall, wenn $\forall i \in \{1, \ldots, m\}\, \exists j \in \{1, \ldots, n\} : C_j^1 \subseteq C_i^2$. Für jedes Provenance-Atom C_i^2 der Why-Provenance des Rumpfes existiert mindestens ein Provenance-Atom des Kopfes, welches in C_i^2 enthalten ist [Ile14].
Der erweiterte Test auf aktive Trigger gilt nur für TGDs und s-t-TGDs. Die Verwendung von EGDs ist in der Doktorarbeit [Ile14] näher beschrieben.

Berechnung der Why-Provenance der gechaseten Atome

Bevor der provenance-bewusste CHASE durchgeführt wird, wird die Why-Provenance jedes Atoms des Universalplans gesetzt. Die Provenance dieser Atome sind die Atome selbst. Jegliche Atome, die durch eine TGD abgeleitet wurden, bekommen die Provenance des Rumpfes der TGD unter Anwendung des provenance-bewussten aktiven Triggers. Falls ein hinzugefügtes Atom schon in der Anfrage enthalten ist, so wird die Why-Provenance mit dem logischen Oder verknüpft und die Atome werden verschmolzen [DH13].

Berechnung der Why-Provenance der Originalanfrage

Nachdem der Universalplan provenance-bewusst gechaset wurde, kann die Why-Provenance der Originalanfrage bestimmt werden. Hierbei werden die möglichen Homomorphismen von der Originalanfrage auf den gechaseten Universalplan gebildet. Die Why-Provenance eines Homomorphismus entspricht der Why-Provenance eines oder mehrerer Rewritings der Originalanfrage. Die Why-Provenance jedes Homomorphismus wird dann mit dem logischen Oder verknüpft. Die daraus resultierende Why-Provenance ist die Why-Provenance der Originalanfrage [DH13].
Die Rewritings der Anfrage sind jedoch noch nicht direkt ablesbar. Die Why-Provenance der Originalanfrage muss zuvor noch in die disjunktive Normalform transformiert werden. Jeder Konjunktionsterm entspricht hierbei einem Rewriting.

Beispiel 4.2: Provenance-Directed CHASE&BACKCHASE
Die Anfrage $Q(bid) :- \texttt{REZENSIONEN}(lid, bid, 10.0, inh), \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$ soll nur über folgende Sichten beantwortet werden:

$$C_{V_{\texttt{REZ}}} : \forall lid, bid, bew, inh : \texttt{REZENSIONEN}(lid, bid, bew, inh) \rightarrow V_{\texttt{REZ}}(bid)$$
$$C_{V_{\texttt{BUCH}}} : \forall bid, aid, titel, verl, ver, pr : \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr) \rightarrow V_{\texttt{BUCH}}(bid)$$
$$C_{V_{\texttt{BUCH_TOP}}} : \forall lid, bid, inh, aid, titel, verl, ver, pr : \texttt{REZENSIONEN}(lid, bid, 10.0, inh), \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr) \rightarrow V_{\texttt{BUCH_TOP}}(bid)$$

Es sei anzumerken, dass eine Fremdschlüsselbeziehung zwischen REZENSIONEN und BÜCHER besteht.

$$C_I : \forall lid, bid, bew, inh : \texttt{REZENSIONEN}(lid, bid, bew, inh) \rightarrow \exists aid, titel, verl, ver, pr : \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$$

Nach der Ausführung des CHASE erhält man die gechasete Anfrage Q' und den Universalplan $\mathcal{U}$.

$$Q'(bid) :- \texttt{REZENSIONEN}(lid, bid, 10.0, inh), \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr), V_{\texttt{REZ}}(bid), V_{\texttt{BUCH}}(bid), V_{\texttt{BUCH_TOP}}(bid)$$
$$\mathcal{U} = V_{\texttt{REZ}}(bid), V_{\texttt{BUCH}}(bid), V_{\texttt{BUCH_TOP}}(bid)$$

Wie im vorigen Beispiel schon verdeutlicht, werden die Sichtbedingungen invertiert.

$$B_{V_{\texttt{REZ}}} : \forall bid : V_{\texttt{REZ}}(bid) \rightarrow \exists lid, bew, inh : \texttt{REZENSIONEN}(lid, bid, bew, inh)$$
$$B_{V_{\texttt{BUCH}}} : \forall bid : V_{\texttt{BUCH}}(bid) \rightarrow \exists aid, titel, verl, ver, pr : \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$$
$$B_{V_{\texttt{BUCH_TOP}}} : \forall bid : V_{\texttt{BUCH_TOP}}(bid) \rightarrow \exists lid, inh, aid, titel, verl, ver, pr : \texttt{REZENSIONEN}(lid, bid, 10.0, inh), \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$$

Die Integritätsbedingung C_I wird nicht invertiert und es gilt: $C_I = B_I$. Nach dem Invertieren der Sichtbedingungen wird jedem Atom des Universalplans eine Why-Provenance gegeben.

$$\mathcal{U} = \overbrace{V_{\texttt{REZ}}(bid)}^{V_{\texttt{REZ}}}, \overbrace{V_{\texttt{BUCH}}(bid)}^{V_{\texttt{BUCH}}}, \overbrace{V_{\texttt{BUCH_TOP}}(bid)}^{V_{\texttt{BUCH_TOP}}}$$

Der provenance-bewusste CHASE durchsucht die Integritätsbedingungen dann nach provenance-bewussten aktiven Triggern. Ein aktiver Trigger für die Sichtbedingung $B_{V_{\texttt{REZ}}}$ ist der Homomorphismus $\{bid \rightarrow bid\}$. Der Kopf dieser Sichtbedingung ist nicht im CHASE-Objekt enthalten, da das `REZENSIONEN`-Atom nicht im CHASE-Objekt vorkommt. Das hinzugefügte Atom erhält die Provenance des Atoms $V_{\texttt{REZ}}(bid)$.

$$\overbrace{V_{\texttt{REZ}}(bid)}^{V_{\texttt{REZ}}}, \overbrace{V_{\texttt{BUCH}}(bid)}^{V_{\texttt{BUCH}}}, \overbrace{V_{\texttt{BUCH_TOP}}(bid)}^{V_{\texttt{BUCH_TOP}}}$$
$$\Downarrow B_{V_{\texttt{REZ}}}$$
$$\overbrace{V_{\texttt{REZ}}(bid)}^{V_{\texttt{REZ}}}, \overbrace{V_{\texttt{BUCH}}(bid)}^{V_{\texttt{BUCH}}}, \overbrace{V_{\texttt{BUCH_TOP}}(bid)}^{V_{\texttt{BUCH_TOP}}}, \overbrace{\texttt{REZENSIONEN}(lid_1, bid, bew_1, inh_1)}^{V_{\texttt{REZ}}}$$

Auch zu der Sichtbedingung $B_{V_{\texttt{BUCH}}}$ existiert ein provenance-bewusster aktiver Trigger. Mithilfe des Homomorphismus $\{bid \rightarrow bid\}$ kann der Rumpf der Bedingung auf das CHASE-Objekt abgebildet werden. Wie bei der vorigen Sichtbedingung, existiert auch hier kein erweiterter Homomorphismus, mit dem der Kopf der Sichtbedingung auf das CHASE-Objekt abgebildet werden kann. Die hinzugefügten Atome erhalten die Provenance des Atoms $V_{\texttt{BUCH}}(bid)$.

$$\overbrace{V_{\texttt{REZ}}(bid)}^{V_{\texttt{REZ}}}, \overbrace{V_{\texttt{BUCH}}(bid)}^{V_{\texttt{BUCH}}}, \overbrace{V_{\texttt{BUCH_TOP}}(bid)}^{V_{\texttt{BUCH_TOP}}}, \overbrace{\texttt{REZENSIONEN}(lid_1, bid, bew_1, inh_1)}^{V_{\texttt{REZ}}}$$
$$\Downarrow B_{V_{\texttt{BUCH}}}$$
$$\overbrace{V_{\texttt{REZ}}(bid)}^{V_{\texttt{REZ}}}, \overbrace{V_{\texttt{BUCH}}(bid)}^{V_{\texttt{BUCH}}}, \overbrace{V_{\texttt{BUCH_TOP}}(bid)}^{V_{\texttt{BUCH_TOP}}}, \overbrace{\texttt{REZENSIONEN}(lid_1, bid, bew_1, inh_1)}^{V_{\texttt{REZ}}}, \overbrace{\texttt{BÜCHER}(bid, aid_2, titel_2, verl_2, ver_2, pr_2)}^{V_{\texttt{BUCH}}}$$

Die letzte Sichtbedingung $B_{V_{\texttt{BUCH_TOP}}}$ ist etwas interessanter. Der Rumpf der Sichtbedingung ist mithilfe des Homomorphismus $\{bid \rightarrow bid\}$ im CHASE-Objekt enthalten. Nun wird der Kopf der Bedingung

betrachtet. Das Teilziel BÜCHER lässt sich auf das korrespondierende Teilziel des CHASE-Objekts abbilden. Gleiches gilt aber nicht für das Teilziel REZENSIONEN. Die Konstante 10.0 lässt sich nicht auf eine Variable abbilden, da Konstanten ihren Wert nicht ändern können. Folglich gibt es einen provenance-bewussten aktiven Trigger für die Sichtbedingung $B_{V_{\texttt{BUCH_TOP}}}$. Beim Einfügen der Atome muss jedoch darauf geachtet werden, dass die BÜCHER-Atome miteinander verschmolzen werden.

$$\overbrace{V_{\texttt{REZ}}(bid)}^{V_{\texttt{REZ}}}, \overbrace{V_{\texttt{BUCH}}(bid)}^{V_{\texttt{BUCH}}}, \overbrace{V_{\texttt{BUCH_TOP}}(bid)}^{V_{\texttt{BUCH_TOP}}}, \overbrace{\texttt{REZENSIONEN}(lid_1, bid, bew_1, inh_1)}^{V_{\texttt{REZ}}},$$
$$\overbrace{\texttt{BÜCHER}(bid, aid_2, titel_2, verl_2, ver_2, pr_2)}^{V_{\texttt{BUCH}}}$$
$$\Downarrow B_{V_{\texttt{BUCH_TOP}}}$$
$$\overbrace{V_{\texttt{REZ}}(bid)}^{V_{\texttt{REZ}}}, \overbrace{V_{\texttt{BUCH}}(bid)}^{V_{\texttt{BUCH}}}, \overbrace{V_{\texttt{BUCH_TOP}}(bid)}^{V_{\texttt{BUCH_TOP}}}, \overbrace{\texttt{REZENSIONEN}(lid_1, bid, bew_1, inh_1)}^{V_{\texttt{REZ}}},$$
$$\overbrace{\texttt{BÜCHER}(bid, aid_2, titel_2, verl_2, ver_2, pr_2)}^{V_{\texttt{BUCH_TOP}} \vee V_{\texttt{BUCH}}}, \overbrace{\texttt{REZENSIONEN}(lid_3, bid, 10.0, inh_3)}^{V_{\texttt{BUCH_TOP}}}$$

Neben den Sichtbedingungen sind auch die Integritätsbedingung von Interesse. Man betrachte die Bedingung B_I. Während alle anderen Bedingungen genau einen Homomorphismus besaßen, besitzt B_I gleich zwei. Man betrachte zuerst den Homomorphismus $H = \{lid \to lid_1, bid \to bid, bew \to bew_1, inh \to inh_1\}$. Dieser Homomorphismus bildet den Rumpf der Bedingung auf einen Teil des CHASE-Objekts ab. Bei Anwendung eines erweiterten Homomorphismus $H' = H \cup \{aid \to aid_2, titel \to titel_2, verl \to verl_2, ver \to ver_2, pr \to pr_2\}$ ist jedoch auch der Kopf der Bedingung im CHASE-Objekt enthalten. Nun interessiert man sich für die Why-Provenance der Atome im Rumpf und im Kopf der Integritätsbedingung. Das Atom im Rumpf der Integritätsbedingung unter Anwendung von H hat die Provenance $V_{\texttt{REZ}}$. Der Rumpf des Atoms bekommt also gerade die Provenance des korrespondierenden Atoms im CHASE-Objekt. Ähnlich sieht es im Kopf der Integritätsbedingung aus. Die Why-Provenance unter Anwendung von H' entspricht der Provenance $V_{\texttt{BUCH_TOP}} \vee V_{\texttt{BUCH}}$. Da $V_{\texttt{REZ}}$ die Provenance $V_{\texttt{BUCH_TOP}} \vee V_{\texttt{BUCH}}$ nicht impliziert, ist der Homomorphismus ein provenance-bewusster aktiver Trigger. Wie bei der vorigen Sichtbedingung, werden auch hier die BÜCHER-Atome miteinander verschmolzen.

$$\overbrace{V_{\texttt{REZ}}(bid)}^{V_{\texttt{REZ}}}, \overbrace{V_{\texttt{BUCH}}(bid)}^{V_{\texttt{BUCH}}}, \overbrace{V_{\texttt{BUCH_TOP}}(bid)}^{V_{\texttt{BUCH_TOP}}}, \overbrace{\texttt{REZENSIONEN}(lid_1, bid, bew_1, inh_1)}^{V_{\texttt{REZ}}},$$
$$\overbrace{\texttt{BÜCHER}(bid, aid_2, titel_2, verl_2, ver_2, pr_2)}^{V_{\texttt{BUCH_TOP}} \vee V_{\texttt{BUCH}}}, \overbrace{\texttt{REZENSIONEN}(lid_3, bid, 10.0, inh_3)}^{V_{\texttt{BUCH_TOP}}}$$
$$\Downarrow B_I$$
$$\overbrace{V_{\texttt{REZ}}(bid)}^{V_{\texttt{REZ}}}, \overbrace{V_{\texttt{BUCH}}(bid)}^{V_{\texttt{BUCH}}}, \overbrace{V_{\texttt{BUCH_TOP}}(bid)}^{V_{\texttt{BUCH_TOP}}}, \overbrace{\texttt{REZENSIONEN}(lid_1, bid, bew_1, inh_1)}^{V_{\texttt{REZ}}},$$
$$\overbrace{\texttt{BÜCHER}(bid, aid_2, titel_2, verl_2, ver_2, pr_2)}^{V_{\texttt{BUCH_TOP}} \vee V_{\texttt{BUCH}} \vee V_{\texttt{REZ}}}, \overbrace{\texttt{REZENSIONEN}(lid_3, bid, 10.0, inh_3)}^{V_{\texttt{BUCH_TOP}}}$$

Man betrachte nun den zweiten Trigger $H = \{lid \to lid_3, bid \to bid, bew \to 10.0, inh \to inh_3\}$. Unter Betrachtung des erweiterten Homomorphismus H' lässt sich der Kopf der Bedingung auf ein Teilziel des CHASE-Objekts abbilden.

$$H' = H \cup \{aid \to aid_2, titel \to titel_2, verl \to verl_2, ver \to ver_2, pr \to pr_2\}$$

Man betrachtet jetzt die Why-Provenance des Kopfes und des Rumpfes unter Anwendung des erweiterten Homomorphismus. Der Kopf der TGD besitzt die Why-Provenance $P_1 = V_{\texttt{BUCH_TOP}} \vee V_{\texttt{BUCH}} \vee V_{\texttt{REZ}}$. Es ist ersichtlich, dass die Why-Provenance $P_2 = V_{\texttt{BUCH_TOP}}$ des Rumpfes in P_1 enthalten ist. $V_{\texttt{BUCH_TOP}}$ ist zwar ungleich $V_{\texttt{BUCH}}$ und $V_{\texttt{REZ}}$, jedoch enthält P_1 das Atom $V_{\texttt{BUCH_TOP}}$. Nun müssen noch die anderen Atome aus P_2 betrachtet werden. Da es keine weiteren Atome gibt und die Provenance des Rumpfes die Provenance des Kopfes enthält, ist der Trigger nicht provenance-bewusst aktiv.
Sei $Q'(bid)$ der provenance-bewusst gechasete Universalplan. Nun werden alle Homomorphismen bestimmt, die es von der Originalanfrage in den gechaseten Univesalplan gibt.

$$Q'(bid) :- \overbrace{V_{\texttt{REZ}}(bid)}^{V_{\texttt{REZ}}}, \overbrace{V_{\texttt{BUCH}}(bid)}^{V_{\texttt{BUCH}}}, \overbrace{V_{\texttt{BUCH_TOP}}(bid)}^{V_{\texttt{BUCH_TOP}}}, \overbrace{\texttt{REZENSIONEN}(lid_1, bid, bew_1, inh_1)}^{V_{\texttt{REZ}}},$$
$$\overbrace{\texttt{BÜCHER}(bid, aid_2, titel_2, verl_2, ver_2, pr_2)}^{V_{\texttt{BUCH_TOP}} \vee V_{\texttt{BUCH}} \vee V_{\texttt{REZ}}}, \overbrace{\texttt{REZENSIONEN}(lid_3, bid, 10.0, inh_3)}^{V_{\texttt{BUCH_TOP}}}$$

Ein solcher Homomorphismus wäre H_1. Weitere Homomorphismen existieren nicht. Durch die Selektion im `REZENSIONEN`-Atom der Originalanfrage lässt sich das `REZENSIONEN`-Atom der `REZ`-Sicht nicht verwenden.

$$H_1 = \{lid \to lid_3, bid \to bid, 10.0 \to 10.0, inh \to inh_3\} \cup$$
$$\{aid \to aid_2, titel \to titel_2, verl \to verl_2, ver \to ver_2, pr \to pr_2\}$$

Nun wird die Why-Provenance der verwendeten Atome zusammengefasst. Das `BÜCHER`-Atom stammt von der `BUCH_TOP`-, `BUCH`-, als auch indirekt von der `REZ`-Sicht. Das `REZENSIONEN`-Atom stammt von der `BUCH_TOP`-Sicht. Da beide Atome für das Rewriting benötigt werden, wird die Why-Provenance mit dem logischen Und verknüpft. Die Why-Provenance der Originalanfrage lautet dann: $(V_{\texttt{BUCH_TOP}} \vee V_{\texttt{BUCH}} \vee V_{\texttt{REZ}}) \wedge V_{\texttt{BUCH_TOP}}$. Diese Formel wird nun in die disjunktive Normalform gebracht. Die einzelnen Rewritings sind dann direkt ablesbar.

$$\text{Provenance der Anfrage}: V_{\texttt{BUCH_TOP}} \vee (V_{\texttt{BUCH_TOP}} \wedge V_{\texttt{BUCH}}) \vee (V_{\texttt{BUCH_TOP}} \wedge V_{\texttt{REZ}})$$

Das minimale Rewriting der Originalanfrage wäre $Q(bid) :- V_{\texttt{BUCH_TOP}}(bid)$.

Kapitel 5

Implementierung

Dieses Kapitel behandelt die Implementierung der vorgestellten Konzepte in ChaTEAU. Die Grundlage hierfür bietet das in Abschnitt 5.1 vorgestellte Grundgerüst. In Abschnitt 5.2 wird die Implementierung des Standard-BACKCHASE beschrieben. Im darauffolgenden Abschnitt 5.3 wird der Standard-BACKCHASE um eine Pruningphase erweitert. Hauptaugenmerk liegt auf den Abschnitt 5.4, in dem die Implementierung des Provenance-Directed CHASE&BACKCHASE detaillierter beschrieben wird.

5.1 Grundgerüst

In diesem Abschnitt wird das Grundgerüst des CHASE&BACKCHASE-Algorithmus beschrieben. Bevor die BACKCHASE-Phase ausgeführt werden darf, wird die Originalanfrage gechaset. Aus der daraus resultierenden Anfrage kann nachfolgend der Universalplan gebildet werden. Da die im Konzept vorgestellten BACKCHASE-Varianten im Ablauf große Ähnlichkeiten aufwiesen, kann die BACKCHASE-Phase abstrahiert werden.

Quellcodeausschnitt 5.1 Instance.java - Methode zur Bildung des Universalplans

```
1  public Instance calculateUniversalPlan() {
2    if(this.getOriginTag() == OriginTag.INSTANCE) {
3      return null;
4    }
5    LinkedHashSet<RelationalAtom> universalplan = new
6      LinkedHashSet<>();
7    this.relationalAtoms.forEach(relationalAtom -> {
8      if (this.getSchemaTags().get(relationalAtom.getName()) ==
9        SchemaTag.TARGET)
10       universalplan.add(relationalAtom);
11   });
12   return new Instance(universalplan, this.schema, this.originTag,
13     this.schemaTags, this.headAtoms);
14 }
```

Im Quellcodeausschnitt 5.1 wird die `calculateUniversalPlan`-Methode der `Instance`-Klasse beschrieben. Diese Methode ermöglicht die Berechnung des Universalplans einer Anfrage. Falls der Universalplan einer Instanz berechnet werden soll, so wird der Standardwert `null` zurückgegeben. Die Anfrage besteht aus mehreren Atomen des Typs `RelationalAtom`. Jedes Atom besitzt einen Schema-Tag. Dieser Tag gibt an, ob das Atom zum Quell- oder Zielschema gehört. Alle Atome der Originalanfrage besitzen den Schema-Tag `SOURCE`. Falls ein Atom aus einer s-t-TGD hergeleitet wurde, so bekommt sie den Tag `TARGET`. Der Algorithmus überprüft für jedes Atom der Anfrage, welches Schema-Tag es besitzt. Falls das Atom zum

Zielschema gehört, so wird es zum Universalplan hinzugefügt. Die `calculateUniversalPlan`-Methode gibt den generierten Universalplan zurück. Diese Anfrage unterscheidet sich von der gechaseten Anfrage nur von den genutzten Atomen. Während die gechasete Anfrage sowohl Atome mit dem Schema-Tag `SOURCE` als auch `TARGET` enthält, besitzt die zurückgegebene Anfrage nur Atome aus dem Zielschema.

Quellcodeausschnitt 5.2 AbstractBackchase.java - Klassenvariablen

```
1 private Instance universalPlan;
2 private LinkedHashSet<IntegrityConstraint> integrityConstraints;
3 private Instance originalQuery;
```

Als Grundgerüst des BACKCHASE dient die `AbstractBackchase`-Klasse. Sie enthält drei Klassenvariablen: `universalPlan`, `integrityConstraints` und `originalQuery`. Das `Instance`-Objekt `universalPlan` enthält den Universalplan der Anfrage, `originalQuery` enthält die Originalanfrage. Das Objekt `integrityConstraints` umfasst neben den Integritätsbedingungen der Anfrage auch die inversen Sichtbedingungen.

Quellcodeausschnitt 5.3 AbstractBackchase.java - Initialisierung

```
1  public AbstractBackchase(Instance chasedInstance,
2    LinkedHashSet<IntegrityConstraint> integrityConstraints,
3    Instance originalQuery) {
4    this.originalQuery = originalQuery;
5    this.universalPlan = chasedInstance.calculateUniversalPlan();
6    this.integrityConstraints = new Provenance(chasedInstance,
7      integrityConstraints)
8        .invert()
9        .getSigma();
10 }
```

Der Quellcodeausschnitt 5.3 zeigt den Konstruktor der `AbstractBackchase`-Klasse. Dieser Konstruktor setzt die Klassenvariablen. Während der Initialisierung werden zudem die Integritätsbedingungen invertiert. Für die Invertierung der Integritätsbedingungen wird die `Provenance`-Klasse benutzt. Die `Provenance::invert`-Methode berechnet für jede Sichtbedingung aus `integrityConstraints` ihre Inverse. Mithilfe der `Provenance::getSigma`-Methode werden die Integritäts- und Sichtbedingungen zurückgegeben. Damit die Variablen der invertierten s-t-TGDs korrekt gebunden werden, wurden einige Änderungen an der `STTgd`- und der `Variable`-Klasse vorgenommen. Nähere Details zur Implementierung sind im Anhang B.1.4 und B.1.5 zu finden.

Quellcodeausschnitt 5.4 AbstractBackchase.java - `backchase`- und `getQueries`-Methode

```
1 public abstract Set<Instance> backchase();
2 protected abstract List<Instance> getQueries();
```

Neben dem Konstruktor enthält die `AbstractBackchase`-Klasse noch weitere Methoden. Die zwei wichtigsten Methoden sind im Quellcodeausschnitt 5.4 angegeben. Die `getQueries`-Methode berechnet die zu chasenden Teilanfragen. Mithilfe der `backchase`-Methode werden diese Teilanfragen gechaset und nach Rewritings durchsucht. Weitere Implementierungsdetails werden in den später beschriebenen BACKCHASE-Klassen bekanntgegeben.

5.2 BACKCHASE

Die `Backchase`-Klasse implementiert den Standard-BACKCHASE. Sie nutzt den gleichen Konstruktor wie die Oberklasse `AbstractBackchase`.

Quellcodeausschnitt 5.5 Backchase.java - `BACKCHASE`-Methode des Standard-BACKCHASE

```
1  public Set<Instance> backchase() {
2    List<Instance> subqueries = this.getQueries();
3    Set<Instance> rewritings = new HashSet<>();
4    for (Instance subquery : subqueries) {
5      Instance chasedQuery = Chase.chase(subquery,
6        this.getIntegrityConstraints());
7      if(this.getOriginalQuery()
8        .isHomomorphicallyContainedInTest(chasedQuery))
9        rewritings.add(subquery);
10   }
11   return rewritings;
12 }
```

Der Quellcodeausschnitt 5.5 stellt die `backchase`-Methode des Standard-BACKCHASE vor. Im ersten Schritt bildet diese Methode alle Teilpläne des Universalpans. Zudem wird auch eine Ergebnisliste (`rewritings`) generiert, die die Rewritings enthält. Im zweiten Schritt wird jeder Teilplan (`subquery`) mit den Integritätsbedingungen und inversen Sichtbedingungen gechaset. Falls ein Homomorphismus von der Originalanfrage in die gechasete Anfrage existiert, so wurde ein Rewriting gefunden und der Teilplan wird zu `rewritings` hinzugefügt. Zum Schluss gibt die `backchase`-Methode die Ergebnisliste, die alle gefundenen Rewritings enthält, zurück.

Quellcodeausschnitt 5.6 Backchase.java - `getQueries`-Methode des Standard-BACKCHASE

```
1  public List<Instance> getQueries() {
2    List<Instance> queries = new ArrayList<>();
3    Set<RelationalAtom> relationalAtoms = new
4      LinkedHashSet<>(this.getUniversalPlan()
5        .getRelationalAtoms());
6    List<Set<RelationalAtom>> subqueries = new ArrayList<>();
7    powerSet(subqueries, relationalAtoms);
8    subqueries.sort(Comparator.comparing(Set::size));
9    for (Iterator<Set<RelationalAtom>> it =
10     subqueries.iterator();it.hasNext();) {
11     Set<RelationalAtom> subquery = it.next();
12     queries.add(new Instance(
13       (LinkedHashSet<RelationalAtom>) subquery,
14       this.getUniversalPlan().getSchema(),
15       this.getUniversalPlan().getOriginTag(),
16       this.getUniversalPlan().getSchemaTags()
17     ));
18   }
19   return queries;
20 }
```

Quellcodeausschnitt 5.6 beschreibt die `getQueries`-Methode des Standard-BACKCHASE mit und ohne Pruning. Diese Methode gibt alle Teilpläne des Universalplans zurück. Im ersten Schritt werden die relationalen Atome aus dem Universalplan extrahiert. Zudem wird die Ergebnisliste `queries` erstellt. Im zweiten Schritt wird die Potenzmenge der gerade extrahierten Menge von Atomen generiert. Diese Potenzmenge wird nach der Größe aufsteigend sortiert. Da Anfragen keine lose Ansammlung von Atomen sind, wird jedes Element der Potenzmenge in eine Instanz eingearbeitet und zur Ergebnisliste hinzugefügt. Die generierten Instanzen unterscheiden sich vom Universalplan nur von den genutzten relationalen Atomen. Die Ergebnisliste, die die gerade erstellten Instanzen enthält, wird dann zurückgegeben.

5.3 BACKCHASE-Pruning

Die `BackchasePruning`-Klasse erweitert die `Backchase`-Klasse um einen Pruning-Schritt. Sie nutzt dabei den gleichen Konstruktor wie die Oberklasse `Backchase`.

Quellcodeausschnitt 5.7 BackchasePruning.java - `getQueries`-Methode des `BackchasePruning`

```
1  public Set<Instance> backchase() {
2    List<Instance> subqueries = this.getQueries();
3    Set<Instance> rewritings = new HashSet<>();
4    for (Instance subquery : subqueries) {
5      boolean contained = false;
6      for (Instance rewriting: rewritings) {
7        if (rewriting.isHomomorphicallyContainedInTest(
8          subquery)) {
9          contained = true;
10         break;
11       }
12     }
13     if (contained) continue;
14     Instance chasedQuery = Chase.chase(subquery,
15       this.getIntegrityConstraints());
16     if(this.getOriginalQuery()
17       .isHomomorphicallyContainedInTest(chasedQuery))
18       rewritings.add(subquery);
19   }
20   return rewritings;
21 }
```

Quellcodeausschnitt 5.7 beschreibt die `backchase`-Methode des BACKCHASE mit Pruning. Der Ablauf der BACKCHASE-Phase ähnelt dem Standard-BACKCHASE. Im ersten Schritt wird die Ergebnisliste erstellt und die Teilanfragen des Universalplans berechnet. Im zweiten Schritt wird für jede Teilanfrage überprüft, ob sie ein Rewriting enthält. Hierbei wird getestet, ob ein Homomorphismus von einem Rewriting zu der Teilanfrage definiert ist. Falls dies der Fall ist, so wird die nächste Teilanfrage behandelt. Wenn die Teilanfrage kein Rewriting enthält, so wird sie gechaset. Falls die Originalanfrage in der gechaseten Teilanfrage enthalten ist, so ist die Teilanfrage ein Rewriting und wird zur Ergebnisliste hinzugefügt. Zum Schluss gibt die `backchase`-Methode die Ergebnisliste zurück.

5.4 Provenance-Directed CHASE&BACKCHASE

Die BACKCHASE-Phase des Provenance-Directed CHASE&BACKCHASE unterscheidet sich grundlegend von den anderen BACKCHASE-Varianten. Bei der Instanziierung eines `BackchasePD`-Objekts wird der Konstruktor seiner Oberklasse `AbstractBackchase` aufgerufen. Zudem wird auch eine Map namens `provID` erstellt, die zu jeder Provenance-ID einer Sicht das zugehörige relationale Atom speichert.

Quellcodeausschnitt 5.8 BackchasePD.java - `backchase`-Methode des `BackchasePD`

```
1  public HashSet<Instance> backchase() {
2    setViewProvenance();
3    Chase chase = new Chase(this.getUniversalPlan(),
4      this.getIntegrityConstraints());
5    chase.setAQUV(true);
6    Instance chasedQuery = chase.chase();
7    Set<RelationalAtom> relationalAtomsWithoutViews =
```

```
 8       getRelationalAtomsWithoutViews(chasedQuery);
 9     Map<String, Set<RelationalAtom>> keys =
10       this.getRelationKeys();
11     Map<RelationalAtom, Set<RelationalAtom>> keyRelationMap =
12       this.getKeyRelationMap(keys, relationalAtomsWithoutViews);
13     Set<List<RelationalAtom>> cartesianProduct =
14       com.google.common.collect.Sets.cartesianProduct(
15         new ArrayList<>(keyRelationMap.values()));
16     Set<Set<String>> queryWhyProvenance =
17       this.getQueryWhyProvenance(cartesianProduct);
18     Set<Instance> rewritings =
19       this.getRewritings(queryWhyProvenance);
20     return rewritings;
21   }
```

Die im Quellcodeausschnitt 5.8 vorgestellte Backchase-Phase kann in vier Bereiche untergliedert werden. Im ersten Bereich (Zeile 2) wird die Why-Provenance der Atome des Universalplans generiert. Im zweiten Bereich (Zeile 3 bis 6) wird der provenance-bewusste CHASE ausgeführt. Nach dem CHASE wird im dritten Bereich (Zeile 7 bis 17) die Why-Provenance der Originalanfrage bestimmt. Im letzten Bereich (Zeile 18 bis 20) werden anhand der vorher bestimmten Why-Provenance die möglichen Rewritings berechnet.

5.4.1 Provenance-bewusster CHASE

Bevor der provenance-bewusste CHASE ausgeführt wird, muss die Why-Provenance der Atome aus dem Universalplan generiert werden.

Quellcodeausschnitt 5.9 BackchasePD.java - `setViewProvenance`-Methode des `BackchasePD`

```
 1  private void setViewProvenance() {
 2    for(RelationalAtom relationalAtom :
 3      this.getUniversalPlan().getRelationalAtoms()) {
 4      relationalAtom.getProvInfo().setId(relationalAtom
 5        .generateProvenanceId(relationalAtom
 6        .getRelationName()));
 7      List whyProvenance = new ArrayList<>();
 8      whyProvenance.add(relationalAtom.getProvInfo().getId());
 9      relationalAtom.getProvInfo().getWhyProvenance()
10        .add((ArrayList<String>)whyProvenance);
11      provID.put(relationalAtom.getProvInfo().getId(),
12        relationalAtom);
13    }
14  }
```

Die `setViewProvenance`-Methode aus Quellcodeausschnitt 5.9 generiert für jedes Atom des Universalplans die Why-Provenance. Als Why-Provenance kommt hier die Provenance-ID des Atoms in Betracht. Damit im späteren Verlauf anhand der IDs effizient auf die Atome zugegriffen werden kann, enthält die `BackchasePD`-Klasse die `provID`-Map. Sie speichert zu jeder Provenance-ID das aktuelle Atom. Nachdem die Why-Provenance der Atome des Universalpans bestimmt wurde, kann der provenance-bewusste CHASE ausgeführt werden.

Quellcodeausschnitt 5.10 Chase.java - `processTrigger`-Methode des `BackchasePD`

```
 1  private boolean processTrigger(IntegrityConstraint constraint,
 2    Homomorphism homomorphism) {
 3    var success = true;
```

```
4    if (this.aquv && constraint instanceof Tgd) {
5      if (homomorphism.isProvenanceAwareActiveTriggerFor(
6        instanceWithNewFacts, (Tgd) constraint)) {
7        var tgd = (Tgd) constraint;
8        applyTgd(tgd, homomorphism);
9      }
10   } ...
11   return success;
12 }
```

Die Methode `processTrigger` der `Chase`-Klasse arbeitet Integritäts- und Sichtbedingung in die Anfrage ein. Mithilfe von `Homomorphism::isProvenanceAwareActiveTriggerFor` wird überprüft, ob die Trigger der TGDs und s-t-TGDs provenance-bewusst aktiv sind. Falls ein Trigger provenance-bewusst aktiv ist, so wird die Bedingung mithilfe der `applyTgd`-Methode eingearbeitet.

Quellcodeausschnitt 5.11 Homomorphism.java - Why-Provenance des Rumpfes

```
1  public boolean isProvenanceAwareActiveTriggerFor(
2    Instance instance, Tgd integrityConstraint) {
3    boolean isActiveTrigger = false;
4    Set<Set<Set<String>>> bodyAtomsWhyProvenance = new HashSet<>();
5    Set<Set<Set<String>>> headAtomsWhyProvenance = new HashSet<>();
6    this.getTermMappings().forEach(termMapping -> {
7      Set<Set<String>> prov = new HashSet<>();
8      termMapping.getTargetTermRelationPosition().getKey()
9        .getProvInfo().getWhyProvenance().forEach(whyProv ->
10         prov.add(new HashSet(whyProv)));
11     bodyAtomsWhyProvenance.add(prov);
12   });
13   ...
14 }
```

Quellcodeausschnitt 5.11 verdeutlicht den Aufbau der `isProvenanceAwareActiveTriggerFor`-Methode der `Homomorphism`-Klasse. Damit überprüft werden kann, ob ein Trigger provenance-bewusst aktiv ist, muss zuerst die Why-Provenance des Rumpfes und des Kopfes generiert werden. Die Why-Provenance des Rumpfes der Bedingung kann durch den Homomorphismus schon abgelesen werden. Die Why-Provenance der Atome des Rumpfes werden mit dem logischen Und verknüpft.

Quellcodeausschnitt 5.12 Homomorphism.java - Why-Provenance des Kopfes

```
1  public boolean isProvenanceAwareActiveTriggerFor(
2    Instance instance, Tgd integrityConstraint) {
3    ...
4    for (RelationalAtom constraintAtom :
5      integrityConstraint.getHead()) {
6      RelationalAtom mappedHeadAtom = getMappedAtom(instance,
7        this.applyMappingsTo(constraintAtom));
8      if (mappedHeadAtom == null) {
9        isActiveTrigger = true;
10       break;
11     }
12     Set<Set<String>> prov = new HashSet<>();
13     mappedHeadAtom.getProvInfo().getWhyProvenance()
14       .forEach(whyProv -> prov.add(new HashSet(whyProv)));
15     headAtomsWhyProvenance.add(prov);
```

```
16    }
17    if (isActiveTrigger)
18      return true;
19    ...
20  }
```

Die Berechnung der Why-Provenance des Kopfes ist etwas komplexer und wird in Quellcodeausschnitt 5.12 näher beschrieben. Zuerst muss überprüft werden, ob ein erweiterter Homomorphismus von den Kopfatomen zu den Atomen der gechaseten Anfrage existiert. Falls mindestens ein Atom nicht auf ein Teilziel der Anfrage abgebildet werden kann, so ist der Trigger aktiv und es wird `true` zurückgegeben. Dieses Prinzip ist schon vom originalen CHASE bekannt. Die Why-Provenance der abgebildeten Teilziele der Anfrage wird während dieser Phase gespeichert. Diese Provenance entspricht der Why-Provenance des Kopfes. Wie auch beim Rumpf, wird auch hier die Why-Provenance jedes Atoms mit dem logischen Und verknüpft.

Quellcodeausschnitt 5.13 Homomorphism.java - Provenance-bewusste aktive Trigger

```
1  public boolean isProvenanceAwareActiveTriggerFor(
2    Instance instance, Tgd integrityConstraint) {
3    ...
4    Set<Set<String>> bodyWhyProvenance =
5      ProvenanceUtils.transformWhyProvInDNF(bodyAtomsWhyProvenance);
6    Set<Set<String>> headWhyProvenance =
7      ProvenanceUtils.transformWhyProvInDNF(headAtomsWhyProvenance);
8    ProvenanceInformation bodyProvInfo =
9      new ProvenanceInformation(null);
10   bodyProvInfo.setWhyProvenance(new ArrayList(bodyWhyProvenance));
11   isActiveTrigger = !bodyProvInfo.whyProvenanceImplies(
12     new ArrayList(headWhyProvenance));
13   return isActiveTrigger;
14 }
```

Quellcodeausschnitt 5.13 zeigt die weitere Verwertung der generierten Why-Provenance-Ausdrücke. Beim Test auf provenance-bewusste aktive Trigger müssen die Why-Provenance-Formeln in disjunktiver Normalform (DNF) vorliegen. Die Why-Provenance des Kopfes und Rumpfes ist jedoch nicht in DNF. Diese Form wird durch Anwenden des Distributivgesetzes erreicht. Zum Schluss wird überprüft, ob die Why-Provenance des Rumpfes die Why-Provenance des Kopfes impliziert. Dieser Test wird durch die Methode `ProvenanceInformation::whyProvenanceImplies` ausgeführt. Falls die Provenance nicht impliziert ist, so ist der Trigger aktiv und es wird `true` zurückgegeben, ansonsten wird `false` zurückgegeben.

Quellcodeausschnitt 5.14 ProvenanceInformation.java - `whyProvenanceImplies`-Methode

```
1  private ArrayList<ArrayList<String>> whyProvenance;
2  ...
3  public boolean whyProvenanceImplies(
4    ArrayList<ArrayList<String>> whyProvenance) {
5    boolean isImplied = true;
6    for (Collection provenanceBody : this.getWhyProvenance()) {
7      boolean contained = false;
8      for (Collection provenanceHead : whyProvenance) {
9        if (provenanceBody.containsAll(provenanceHead)) {
10         contained = true;
11         break;
12       }
13     }
```

```
14      if (!contained) {
15        isImplied = false;
16        break;
17      } else isImplied = true;
18    }
19    return isImplied;
20  }
```

Die vorliegenden Why-Provenance-Formeln der `whyProvenanceImplies`-Methode aus Quellcodeausschnitt 5.14 sind in DNF. Dadurch ist jedes Element der Why-Provenance ein Konjunktionsterm. Die einzelnen Terme sind mit dem logischen Oder verknüpft. Der Algorithmus überprüft für jeden Konjunktionsterm der Why-Provenance des Rumpfes der Bedingung, ob mindestens ein Konjunktionsterm der Provenance des Kopfes der Bedingung enthalten ist. Falls dies der Fall ist, so wird mit dem nächsten Element der Why-Provenance des Rumpfes fortgeführt. Falls kein Konjunktionsterm der Provenance des Kopfes impliziert wurde, so ist der Trigger aktiv und es wird `true` zurückgegeben. Falls jedes Element der Provenance des Rumpfes mindestens ein Konjunktionsterm der Provenance des Kopfes enthält, so ist die Provenance des Kopfes impliziert und es wird `false` zurückgegeben.
Nachdem überprüft wurde, dass der Trigger provenance-bewusst aktiv ist, wird die Integritäts- oder Sichtbedingung mit der `applyTgd`-Methode der `CHASE`-Klasse eingearbeitet. Beim Einarbeiten der Bedingung wird die Why-Provenance mithilfe der `calculateProvenance`-Methode berechnet.

Quellcodeausschnitt 5.15 Chase.java - `calculateProvenance`-Methode

```
1  private void calculateProvenance(Homomorphism tempHomomorphism,
2    RelationalAtom relationalHeadAtom, RelationalAtom newAtom) {
3    ...
4    var whyProvenance = new ArrayList<ArrayList<String>>();
5    var howProvenanceHelper =
6      new LinkedHashSet<Pair<RelationalAtom, RelationalAtom>>();
7    tempHomomorphism.getTermMappings().stream().
8        filter(mapping -> mapping.getTargetTermRelationPosition()
9          != null)
10       .forEach(mapping -> howProvenanceHelper
11           .add(new Pair<RelationalAtom, RelationalAtom>(mapping
12               .getSourceTermRelationPosition().getKey(),
13               mapping.getTargetTermRelationPosition().getKey())));
14   Set<Set<Set<String>>> unnormalizedWhyProvenance =
15     new HashSet<>();
16   howProvenanceHelper.stream().forEach(pair -> {
17     Set<Set<String>> provenance = new HashSet<>();
18     pair.getValue().getProvInfo().getWhyProvenance().forEach(
19       provenanceAtom -> provenance.add(
20         new HashSet<>(provenanceAtom)));
21     unnormalizedWhyProvenance.add(provenance);
22   });
23   ...
24 }
```

Die `calculateProvenance`-Methode aus Quellcodeausschnitt 5.15 berechnet die Provenance jedes eingechaseten Kopfatoms einzeln. ChaTEAU unterstützte schon in vorigen Versionen die Berechnung der Why-Provenance. Die generierte Why-Provenance enthält die Why-Provenance aller Atome des Rumpfes der Bedingung. Die verwendeten Teilziele der gechaseten Anfrage sind im angewendeten Homomorphismus gespeichert. Beim Berechnen der Why-Provenance eines Atoms muss beachtet werden, dass sie in DNF vorliegen muss. Wie in Quellcodeausschnitt 5.13 schon gesehen, wird auch hier die berechnete

Why-Provenance in DNF gebracht. Die Liste `whyProvenance` enthält die normalisierte Provenance des Rumpfes der Bedingung.

Quellcodeausschnitt 5.16 Chase.java - Verschmelzen von Atomen

```
1  private void calculateProvenance(Homomorphism tempHomomorphism,
2    RelationalAtom relationalHeadAtom, RelationalAtom newAtom) {
3    ...
4    if (newInstance.getRelationalAtoms().contains(newAtom)) {
5      ...
6    } else {
7      AtomicReference<Boolean> contained =
8        new AtomicReference<>(false);
9      newInstance.getRelationalAtoms().stream().filter(atom ->
10       newAtom.hasHomomorphismTo(atom) &&
11       atom.hasHomomorphismTo(newAtom)).forEach(atom -> {
12         for (RelationalAtom relationalAtom :
13           newInstance.getRelationalAtoms()) {
14           if (relationalAtom == atom) continue;
15           if (atom.getTerms().stream().anyMatch(term -> {
16             if (term.isVariable() && ((Variable) term)
17               .getVariableType().equals(VariableType.FOR_ALL))
18               return false;
19             return relationalAtom.getTerms().contains(term);
20           })) {
21             return;
22           }
23         }
24         whyProvenance.forEach(whyProvAtom -> {
25           if (!atom.getProvInfo().getWhyProvenance()
26             .contains(whyProvAtom))
27             atom.getProvInfo().getWhyProvenance()
28               .add(whyProvAtom);
29         });
30         contained.set(true);
31       });
32     ...
33     if (contained.get())
34       return;
35   }
36 }
```

Falls das Kopfatom schon in der gechaseten Anfrage enthalten ist, so werden die Atome miteinander verschmolzen. Die Vorgehensweise ist im Quellcodeausschnitt 5.16 verdeutlicht. Die Instanz `newInstance` enthält alle gechaseten Atome aus dem jetzigen CHASE-Schritt. Im ersten Schritt wird überprüft, ob ein Homomorphismus von dem Kopfatom zu einem Atom aus `newInstance` hin und zurück existiert (Zeile 9-11). Falls das Kopfatom schon in der Anfrage enthalten ist, so wird überprüft, ob das korrespondierende Atom aus `newInstance` ein Verbundattribut enthält (Zeile 15 bis 20). Falls kein Verbundattribut enthalten ist, so können die Atome miteinander verschmolzen werden. Falls ein Verbundattribut existiert und das Attribut nicht allquantifiziert ist, dürfen die Atome nicht verschmolzen werden (Zeile 16 bis 18). Falls das Kopfatom mit einem Atom aus `newInstance` verschmolzen werden darf, so wird die Why-Provenance des Kopfatoms zur Why-Provenance des Atoms aus `newInstance` hinzugefügt (Zeile 24 bis 29). Neben der Instanz `newInstance` gibt es auch eine Instanz `instanceWithNewFacts`. Da das Kopfatom auch

mit Atomen aus den vorigen CHASE-Schritten verschmolzen werden kann, wird der eben vorgestellten Quellcodeausschnitt auch für die Instanz `instanceWithNewFacts` durchgeführt.

Quellcodeausschnitt 5.17 Chase.java - Hinzufügen neuer Atome

```
1  private void calculateProvenance(Homomorphism tempHomomorphism,
2    RelationalAtom relationalHeadAtom, RelationalAtom newAtom) {
3    ...
4    if (newInstance.getRelationalAtoms().contains(newAtom)) {
5      ...
6    } else {
7      ...
8      var provInfoWhyProv = new ArrayList<ArrayList<String>>();
9      provInfoWhyProv.addAll(whyProvenance);
10     var provInfo = new ProvenanceInformation(provenanceId,
11       whereTupleProvenance, whereRelationsProvenance,
12       provInfoWhyProv);
13     newAtom.setProvInfo(provInfo);
14     newInstance.addRelationalAtom(newAtom);
15   }
16 }
```

Falls das Kopfatom nicht mit einem anderen Atom verschmolzen werden kann, so wird, wie im Quellcodeausschnitt 5.17 gezeigt, das Atom selbst zur Instanz `newInstance` hinzugefügt. Zuvor muss jedoch noch die Provenance des Atoms gesetzt werden. Dafür wird zuerst ein neues `ProvenanceInformation`-Objekt erstellt. Dieses Objekt enthält die Provenance-Informationen des Atoms, darunter auch die Why-Provenance des Rumpfes der Bedingung. Nachdem die Provenance-Informationen gesetzt und dem Kopfatom zugeordnet wurden, wird das Atom zu `newInstance` hinzugefügt. Nachdem die Provenance berechnet wurde, wird mit dem nächsten Kopfatom der Bedingung fortgeführt.

5.4.2 Berechnung der Why-Provenance der Originalanfrage

Nach dem Anwenden des provenance-bewussten CHASE auf den Universalplan, kann die Why-Provenance der Originalanfrage berechnet werden. Dafür müssen zuerst die relationalen Atome des Quellschemas aus der gechaseten Anfrage extrahiert werden.

Quellcodeausschnitt 5.18 BackchasePD.java - Extraktion der Atome des Quellschemas

```
1  protected Set<RelationalAtom>
2    getRelationalAtomsWithoutViews(Instance chasedQuery) {
3    Set<RelationalAtom> relationalAtomsWithoutViews =
4      new HashSet<>();
5    chasedQuery.getRelationalAtoms().forEach(relationalAtom -> {
6      if (chasedQuery.getSchemaTags().get(relationalAtom.getName())
7        == SchemaTag.SOURCE)
8        relationalAtomsWithoutViews.add(relationalAtom);
9    });
10   return relationalAtomsWithoutViews;
11 }
```

Die `getRelationalAtomsWithoutViews`-Methode aus dem Quellcodeausschnitt 5.18 gibt alle Atome des Quellschemas aus der gegebenen Instanz aus. Es wird für jedes Atom geprüft, ob es den Schema-Tag `SOURCE` hat. Falls dies der Fall ist, so wird das Atom zur Ergebnismenge hinzugefügt. Nachdem alle Atome überprüft wurden, wird die Ergebnismenge zurückgegeben.

Nachdem die Atome des Quellschemas extrahiert worden sind, wird zur effizienteren Berechnung der potentiellen Rewritings eine Relationname-Relationen-Map generiert.

Quellcodeausschnitt 5.19 BackchasePD.java - Generierung der Relationname-Relationen-Map

```
1  private Map<String, Set<RelationalAtom>> getRelationKeys() {
2    Map<String, Set<RelationalAtom>> keys = new HashMap();
3    this.getOriginalQuery().getRelationalAtoms()
4      .forEach(relationalAtom -> {
5      if (this.getOriginalQuery().getSchemaTags()
6        .get(relationalAtom.getName()) == SchemaTag.SOURCE) {
7        if (!keys.containsKey(relationalAtom
8          .getRelationName())) {
9          Set<RelationalAtom> container = new HashSet<>();
10         container.add(relationalAtom);
11         keys.put(relationalAtom.getRelationName(), container);
12       } else keys.get(relationalAtom.getRelationName())
13           .add(relationalAtom);
14     }
15   });
16   return keys;
17 }
```

Die im Quellcodeausschnitt 5.19 beschriebene `getRelationKeys`-Methode berechnet eine Map, die für jeden Relationnamen die Teilziele aus der Originalanfrage mit den gleichen Relationnamen speichert. Diese Map wird dann zurückgegeben.
Nachdem die Map erstellt wurde, wird mithilfe der `getKeyRelationMap`-Methode jedem Atom der Originalanfrage die interessanten Atome der gechaseten Anfrage zugeordnet.

Quellcodeausschnitt 5.20 BackchasePD.java - `getKeyRelationMap`-Methode

```
1  private Map<RelationalAtom, Set<RelationalAtom>>
2    getKeyRelationMap(
3      Map<String, Set<RelationalAtom>> relationNames,
4      Set<RelationalAtom> query) {
5    Map<RelationalAtom, Set<RelationalAtom>> keyRelationMap =
6      new HashMap<>();
7    relationNames.values().forEach(keys -> keys.forEach(
8        key -> keyRelationMap.put(key, new HashSet<>())));
9    query.forEach(relation -> {
10     Set<RelationalAtom> atomContainer =
11       relationNames.get(relation.getRelationName());
12     if (atomContainer != null)
13       atomContainer.forEach(atom ->
14         keyRelationMap.get(atom).add(relation));
15   });
16   return keyRelationMap;
17 }
```

Die `getKeyRelationMap`-Methode erhält als Parameter die provenance-bewusst gechasete Anfrage und die im vorigen Quellcodeausschnitt generierte Map. Die `keyRelationMap`-Map enthält für jedes Atom aus der übergebenden Map eine Menge an Atomen aus der gechaseten Anfrage. Hierbei muss der Name dieser Atome gleich sein. Die generierte Map wird danach zurückgegeben.
Nachdem die `keyRelationMap` generiert wurde, können die potentiellen Rewritings berechnet werden. Durch das Bilden des kartesischen Produkts von den Mengen der Werte von `keyRelationMap`

kann die Why-Provenance der potentiellen Rewritings bestimmt werden. Zum Berechnen der Why-Provenance sind jedoch nicht die potentiellen, sondern die wirklichen Rewritings von Interesse. Die Methode `getQueryWhyProvenance` berechnet die Why-Provenance der Originalanfrage.

Quellcodeausschnitt 5.21 BackchasePD.java - `getQueryWhyProvenance`-Methode

```
1  private Set<Set<String>>
2    getQueryWhyProvenance(
3      Set<List<RelationalAtom>> potentialRewritings) {
4    Set<Set<String>> queryWhyProvenance = new HashSet<>();
5    for (List<RelationalAtom> subqueryAtoms : potentialRewritings) {
6      Instance subquery = new Instance(
7        new LinkedHashSet(subqueryAtoms),
8        this.getUniversalPlan().getSchema(),
9        this.getUniversalPlan().getOriginTag(),
10       this.getUniversalPlan().getSchemaTags(),
11       this.getUniversalPlan().getHeadAtoms());
12     if (this.getOriginalQuery()
13       .isHomomorphicallyContainedInTest(subquery)) {
14       Set<Set<Set<String>>> unnormalizedWhyProvenance =
15         new HashSet<>();
16       for (RelationalAtom atom : subqueryAtoms) {
17         Set<Set<String>> prov = new HashSet<>();
18         atom.getProvInfo().getWhyProvenance().forEach(
19           whyProv -> prov.add(new HashSet(whyProv)));
20         unnormalizedWhyProvenance.add(prov);
21       }
22       queryWhyProvenance.addAll(ProvenanceUtils
23         .transformWhyProvInDNF(
24           unnormalizedWhyProvenance));
25     }
26   }
27   return queryWhyProvenance;
28 }
```

Die `getQueryWhyProvenance`-Methode berechnet anhand der potentiellen Rewritings die Why-Provenance der Originalanfrage. Für jedes potentielle Rewriting (`subqueryAtoms`) wird überprüft, ob ein Homomorphismus von der Originalanfrage in das potentielle Rewriting existiert. Falls dies der Fall ist, so wurde ein Rewriting gefunden. Die Why-Provenance des Rewritings wird dann in DNF transformiert und zur Why-Provenance der Originalanfrage hinzugefügt. Die berechnete Provenance der Originalanfrage `queryWhyProvenance` wird zum Schluss zurückgegeben.

5.4.3 Berechnung der Rewritings

Nachdem die Why-Provenance der Originalanfrage berechnet wurde, können die Rewritings dieser Anfrage mithilfe der Methode `getRewritings` bestimmt werden.

Quellcodeausschnitt 5.22 BackchasePD.java - `getRewritings`-Methode

```
1  private Set<Instance> getRewritings(
2    Set<Set<String>> queryWhyProvenance) {
3    Set<Instance> rewritings = new HashSet<>();
4    queryWhyProvenance.forEach(whyProvenance -> {
5      Set<RelationalAtom> views = new LinkedHashSet<>();
6      whyProvenance.forEach(provenanceID ->
```

```
 7         views.add(provID.get(provenanceID)));
 8       rewritings.add(new Instance(
 9         (LinkedHashSet<RelationalAtom>) views,
10         this.getUniversalPlan().getSchema(),
11         this.getUniversalPlan().getOriginTag(),
12         this.getUniversalPlan().getSchemaTags(),
13         this.getUniversalPlan().getHeadAtoms()
14       ));
15     });
16     return rewritings;
17   }
```

Der Quellcodeausschnitt 5.22 beschreibt, wie der Prototyp die Rewritings mithilfe der `getRewritings`-Methode generiert. Als Parameter wird dieser Methode die Why-Provenance der Anfrage übergeben. Jedes Element dieser Provenance ist ein Konjunktionsterm und entspricht einem Rewriting. Da die Konjunktionsterme nur die Provenance-IDs der Sichten, nicht aber die Sichten selbst enthalten, müssen diese noch bestimmt werden. In der `setViewProvenance`-Methode wurde hierfür die Map `provID` angelegt. Die jeweilige Sicht kann unter Angabe der Provenance-ID aufgerufen werden. Die `queryWhyProvenance`-Methode gibt eine Menge von Instanzen zurück, folglich müssen die Sichten der Rewritings noch in eine Instanz eingearbeitet werden. Die Menge an Rewritings wird zum Schluss zurückgegeben.

Beispiel 5.1: BackchasePD

Man betrachte noch einmal das Beispiel 4.2 mit der Anfrage $Q(bid)$. Mithilfe des provenancebewussten CHASE wurde die Anfrage $Q'(bid)$ hergeleitet.

$$Q(bid) :- \texttt{REZENSIONEN}(lid, bid, 10.0, inh), \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$$

$$Q'(bid) :- \overbrace{V_{\texttt{REZ}}(bid)}^{V_{\texttt{REZ}}}, \overbrace{V_{\texttt{BUCH}}(bid)}^{V_{\texttt{BUCH}}}, \overbrace{V_{\texttt{BUCH_TOP}}(bid)}^{V_{\texttt{BUCH_TOP}}}, \overbrace{\texttt{REZENSIONEN}(lid_1, bid, bew_1, inh_1)}^{V_{\texttt{REZ}}},$$
$$\overbrace{\texttt{BÜCHER}(bid, aid_2, titel_2, verl_2, ver_2, pr_2)}^{V_{\texttt{BUCH_TOP}} \vee V_{\texttt{BUCH}} \vee V_{\texttt{REZ}}}, \overbrace{\texttt{REZENSIONEN}(lid_3, bid, 10.0, inh_3)}^{V_{\texttt{BUCH_TOP}}}$$

Der `BackchasePD` extrahiert nun aus der gechaseten Anfrage die Atome aus dem Quellschema. Hierfür wird die `getRelationalAtomsWithoutViews`-Methode genutzt. Mithilfe der `getRelationKeys`-Methode wird eine Map erstellt, die jedes Atom der Originalanfrage unter ihrem Namen speichert.

Schlüssel	**Wert**
REZENSIONEN	$\{\texttt{REZENSIONEN}(lid, bid, 10.0, inh)\}$
BÜCHER	$\{\texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)\}$

In der Originalanfrage muss nicht unbedingt jeder Relationenname nur einmal vorkommen. Ein Beispiel dafür wären Selbstverbunde zwischen Atomen. Damit auch solche Anfragen behandelt werden können, bestehen die Werte dieser Map aus einer Menge von Atomen. Die `getKeyRelationMap`-Methode weist nun jedem Atom der Originalanfrage Atome der gechaseten Anfrage zu. Hierbei wird jedes Atom hinzugefügt, das den gleichen Relationennamen besitzt. Die Methode gibt folgende Map zurück:

Schlüssel	**Wert**
$\texttt{REZENSIONEN}\,(lid, bid, 10.0, inh)$	$\{\texttt{REZENSIONEN}\,(lid_1, bid, bew_1, inh_1)\,,$ $\texttt{REZENSIONEN}\,(lid_3, bid, 10.0, inh_3)\}$
$\texttt{BÜCHER}\,(bid, aid, titel, verl, ver, pr)$	$\{\texttt{BÜCHER}\,(bid, aid_2, titel_2, verl_2, ver_2, pr_2)\}$

Mithilfe dieser Map können die potentiellen Rewritings bestimmt werden. Da für ein Rewriting jedes Teilziel der Originalanfrage abgedeckt sein muss, wird das kartesische Produkt über alle Werte der Map gebildet.

$$\{\{\texttt{REZENSIONEN}\,(lid_1, bid, bew_1, inh_1)\,, \texttt{BÜCHER}\,(bid, aid_2, titel_2, verl_2, ver_2, pr_2)\}\} \cup$$
$$\{\{\texttt{REZENSIONEN}\,(lid_3, bid, 10.0, inh_3)\,, \texttt{BÜCHER}\,(bid, aid_2, titel_2, verl_2, ver_2, pr_2)\}\}$$

Die Methode `getQueryWhyProvenance` berechnet mithilfe der potentiellen Rewritings die Why-Provenance der Originalanfrage. Man betrachte zuerst das potentielle Rewriting $\texttt{REZENSIONEN}\,(lid_1, bid, bew_1, inh_1)\,, \texttt{BÜCHER}\,(bid, aid_2, titel_2, verl_2, ver_2, pr_2)$. Im ersten Schritt muss überprüft werden, ob ein Homomorphismus von der Originalanfrage in das potentielle Rewriting existiert. In diesem Beispiel ist es nicht der Fall, da die Konstante 10.0 aus dem `REZENSIONEN`-Atom der Originalanfrage nicht auf die Variable bew_1 aus dem potentiellen Rewriting abgebildet werden kann. Die Why-Provenance des potentiellen Rewritings wird demnach nicht zur Why-Provenance der Originalanfrage hinzugefügt. Anders sieht es beim potentiellen Rewriting $\texttt{REZENSIONEN}\,(lid_3, bid, 10.0, inh_3)\,, \texttt{BÜCHER}\,(bid, aid_2, titel_2, verl_2, ver_2, pr_2)$ aus. Hier existiert ein Homomorphismus von der Originalanfrage in das potentielle Rewriting. Dadurch wurde ein Rewriting gefunden und die Why-Provenance dieses Rewritings wird zur Why-Provenance der Originalanfrage hinzugefügt. Die Why-Provenance der Originalanfrage ist demnach $(V_{\texttt{BUCH_TOP}} \vee V_{\texttt{BUCH}} \vee V_{\texttt{REZ}}) \wedge V_{\texttt{BUCH_TOP}}$. Die Methode `getRewritings` berechnet mithilfe der gebildeten Why-Provenance die möglichen Rewritings der Originalanfrage. Hierfür wird die Why-Provenance zuerst in die disjunkte Normalform gebracht. Die Why-Provenance der Originalanfrage wird demnach in folgende Form transformiert: $V_{\texttt{BUCH_TOP}} \vee (V_{\texttt{BUCH_TOP}} \wedge V_{\texttt{BUCH}}) \vee (V_{\texttt{BUCH_TOP}} \wedge V_{\texttt{REZ}})$. Mithilfe der `provID`-Map können die Informationen der Why-Provenance in Sichten umgewandelt werden. Die Rewritings sind dann von der Provenance ablesbar.

Kapitel 6

Fazit & Ausblick

Dieses Kapitel beschreibt und vergleicht die implementierten Prototypen. In Abschnitt 6.1 werden die Herausforderungen und Probleme der Arbeit verdeutlicht. Abschnitt 6.2 vergleicht zwei der implementierten Algorithmen aus Kapitel 5. Der letzte Abschnitt gibt einen Ausblick darauf, welche Erweiterungen auf Basis des CHASE&BACKCHASE möglich sein könnten.

6.1 Fazit

Diese Arbeit beschäftigte sich mit dem AQuV-Problem. Das Ziel dieser Arbeit war es, den CHASE&BACKCHASE und den Provenance-Directed CHASE&BACKCHASE aus [DH13] in ChaTEAU zu implementieren. Es wurden zudem andere Algorithmen und Systeme zur Lösung des AQuV-Problems, wie in etwa der MiniCon-Algorithmus, vorgestellt. Die theoretischen Anteile des normalen und des provenance-basierten CHASE&BACKCHASE (Provenance-Directed CHASE&BACKCHASE) wurden im Kapitel 3 und Kapitel 4 übermittelt. Die Implementierung des CHASE&BACKCHASE mit einem und ohne einen Pruningschritt war dem theoretischen Modell sehr ähnlich. Die Implementierung des Provenance-Directed CHASE&BACKCHASE war etwas komplexer. Während der provenance-bewusste CHASE größtenteils originalgetreu blieb, wich die Generierung der Why-Provenance der Originalanfrage vom Originalalgorithmus ab. Es wurde nicht direkt ein Homomorphismus von der Originalanfrage auf den gechaseten Universalplan gebildet. Zuvor wurde der gechasete Universalplan in Teilanfragen untergliedert. Mithilfe dieser Teilanfragen wurde dann überprüft, ob ein Homomorphismus von der Originalanfrage existiert. Falls dies der Fall war, so wurden die Sichten aus der Why-Provenance der Teilanfrage extrahiert und es wurde mindestens ein Rewriting gefunden.
Die Komplexität des implementierten CHASE&BACKCHASE mit und ohne Pruningschritt ist exponentiell. In beiden Fällen werden im worst-case-Szenario alle Teilpläne des Universalplans gechaset. Dies wirkt sich gerade bei Universalplänen mit vielen Sichten negativ aus. Der implementierte Provenance-Directed CHASE&BACHCHASE hat dieses Problem nicht. Hier wird nur der Universalplan selbst provenance-bewusst gechaset. Dies bedeutet jedoch nicht, dass diese Variante immer effizienter ist. Der Test auf provenance-bewusste Trigger und die Bildung der Rewritings benötigen die Why-Provenance der beteiligten Teilanfragen in DNF. Die Why-Provenance selbst, sofern die Teilanfrage nicht nur ein Atom enthält, liegt jedoch nicht in DNF vor. Demnach muss die Why-Provenance bei jedem Test auf provenance-bewusste aktive Trigger die Provenance des Kopfes und Rumpfes der Bedingung mit polynomiellen Aufwand in DNF überführen.
Dieser Performance-Unterschied ist auch in Abschnitt Abschnitt 6.2 ersichtlich. Die Laufzeit des Standard-BACKCHASE ist, außer bei der Verwendung weniger Sichten, mehrere Größenordnungen höher als die des Provenance-Directed CHASE&BACKCHASE.

6.2 Evaluation der Implementierungen

In diesem Abschnitt wird die Performance des `Backchase`- und des `BackchasePD`-Prototyps verglichen. Wie im Implementierungs-Kapitel schon verdeutlicht, implementiert der `Backchase`-Prototyp den CHASE&BACKCHASE-Algorithmus und der `BackchasePD`-Prototyp den Provenance-Directed CHASE&BACKCHASE-Algorithmus. Der BACKCHASE mit einer Pruning-Phase wird nicht betrachtet.

6.2.1 Aufbau und Ablauf des Experiments

Zum Vergleich der Performance wurden eine Vielzahl an Anfragen mit einer variablen Anzahl an Sichten im Universalplan und einer variablen Anzahl an Attributen der Sichten an ChaTEAU gestellt. Das erste und letzte Atom der Anfrage enthält zwei Verbundattribute. Alle übrigen Atome enthalten nur ein Verbundattribut. Zu jedem Atom der Anfrage existiert genau eine Sicht. Diese Sicht umfasst nur das Atom der Anfrage selbst und gibt alle Attribute aus. Neben den Sichtbedingungen wurden keine weiteren Integritätsbedingungen mit betrachtet. Um mögliche Schwankungen zu vermeiden, wurde jede Anfrage 100 mal nacheinander ausgeführt und die Zeit jedes CHASE&BACKCHASE-Durchgangs notiert. Hierbei wurde nur die Zeit der BACKCHASE-Phase selbst notiert. Das Berechnen der inversen Sichtbedingungen oder das Berechnen des Universalplans wurde nicht mit einberechnet. Der Quellcode dieses Experiments findet sich im Anhang B im Quellcodeausschnitt B.5.1 wieder.

6.2.2 Vergleich der Laufzeit

Abbildung 6.1 spiegelt die Performance des implementierten CHASE&BACKCHASE- und Provenance-Directed CHASE&BACKCHASE-Algorithmus wider. Es wurden Universalpläne mit zwei bis zehn Atomen getestet. Die Zeitdauer ist in Millisekunden angegeben. Zum besseren Vergleich beider Algorithmen wurde die Zeit logarithmisch angegeben.

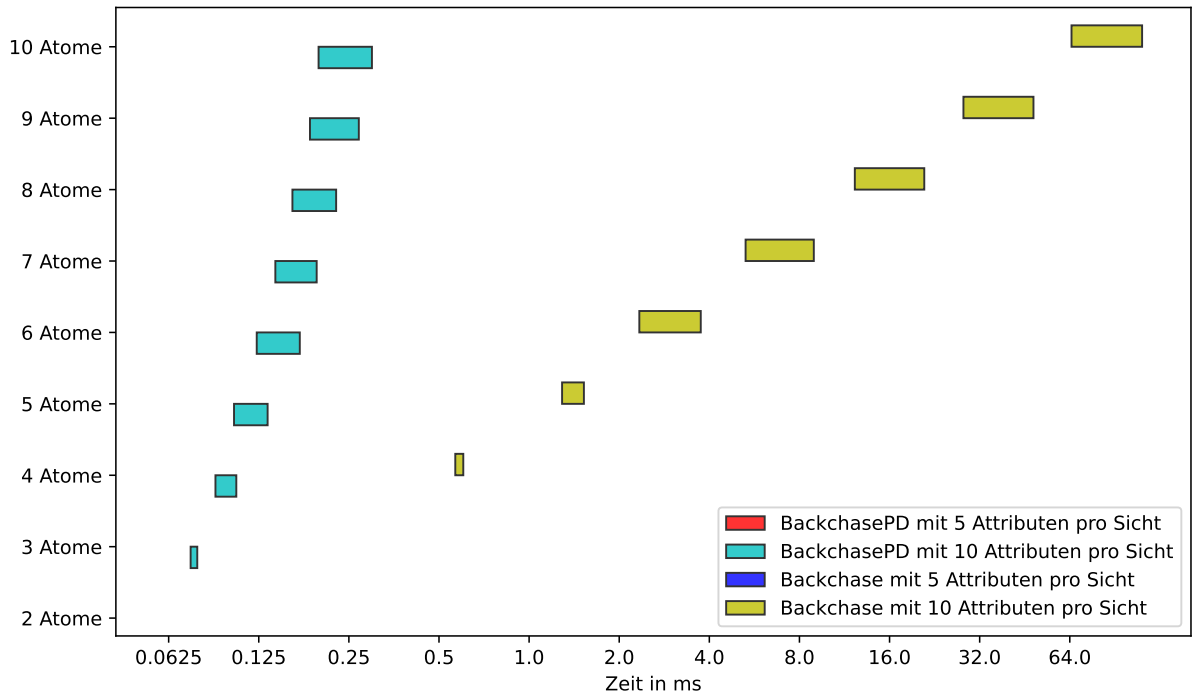


Abbildung 6.1 Performance des `Backchase`- und des `BackchasePD`-Prototyps

Die Laufzeit des CHASE&BACKCHASE-Algorithmus erhöht sich, wie zu erwarten war, exponentiell mit der Anzahl an Atomen im Universalplan. Mit jeder Sicht im Universalplan erhöht sich die Laufzeit ca. um den Faktor zwei. Dies liegt daran, dass jede Anfrage der Potenzmenge des Universalplans gechaset werden muss. Während die BACKCHASE-Phase des CHASE&BACKCHASE bei Universalplänen mit zwei Atomen sich weitestgehend der Null annähert, schlägt die BACKCHASE-Phase bei zehn Sichten schon mit über 66 Millisekunden zu Buche. Die Performance des CHASE&BACKCHASE mit fünf Attributen und zehn Attributen pro Sicht unterscheidet sich vor allem bei Universalplänen mit mehreren Atomen. Bei der Verwendung von zehn Sichten mit je zehn Attributen ist die Laufzeit fast um das Doppelte höher als bei Sichten mit fünf Attributen.
Der Provenance-Directed CHASE&BACKCHASE arbeitet effizienter als der `Backchase`-Prototyp. Der Unterschied bei Universalplänen mit zwei Atomen ist vernachlässigbar klein. Bei der Verwendung von Universalplänen mit mehreren Atomen erhöht sich die Laufzeit jedoch drastisch. Während der `Backchase`-Prototyp bei zehn Atomen im Universalplan ca. 65 bzw. 112 Millisekunden benötigt, benötigt der `BackchasePD`-Prototyp nur ca. $0,2$ bzw. $0,3$ Millisekunden. Dies liegt hauptsächlich an der Anzahl an CHASE-Vorgängen. Während der provenance-basierte Prototyp nur einmal den CHASE durchführt, durchläuft der `Backchase`-Prototyp ganze $2^{10} - 1 = 1023$ CHASE-Vorgänge.

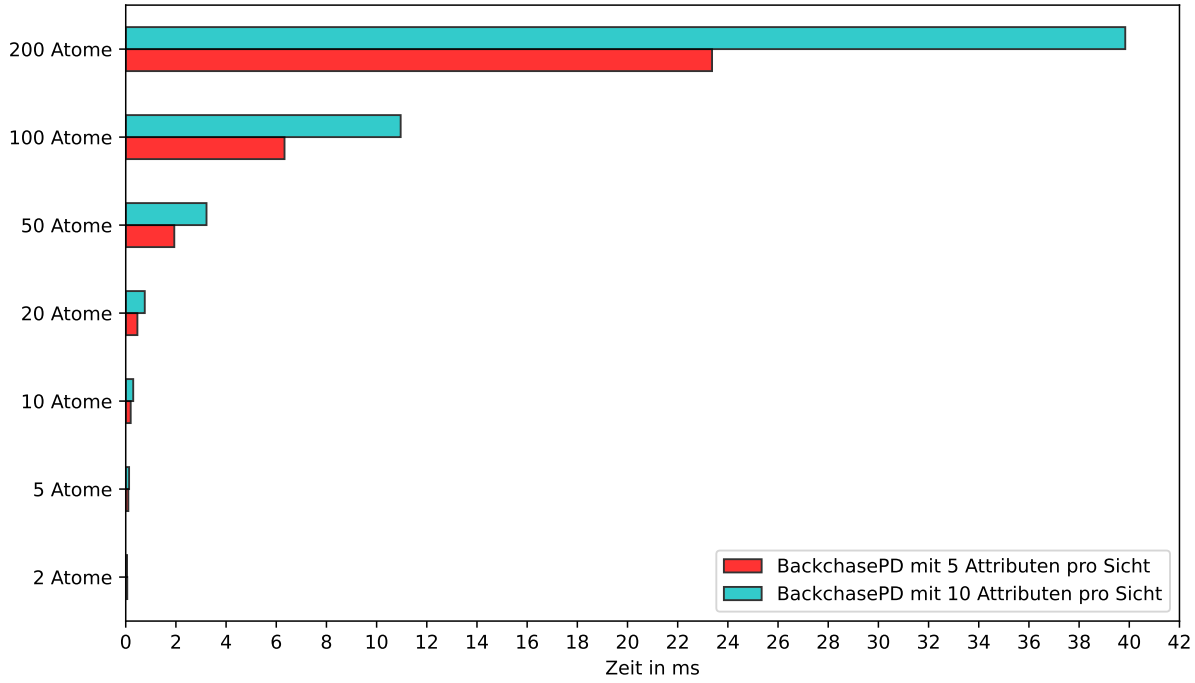


Abbildung 6.2 Performance des `BackchasePD`-Prototyps

In Abbildung 6.2 wird die Performance des implementierten Provenance-Directed CHASE&BACKCHASE gezeigt. Diese Abbildung gibt die Laufzeit für Universalpläne mit zwei bis 200 Atomen an. Die Anzahl an Atomen erhöht sich exponentiell. Die Zeitachse gibt die Zeit in Millisekunden an.
Der provenance-basierte Prototyp besitzt keine exponentielle Laufzeit. Der scheinbar exponentielle Anstieg der Laufzeit in Abbildung 6.2 ist dem Umstand geschuldet, dass auch die Anzahl an Atomen exponentiell zunimmt. Wie beim `Backchase`-Prototyp, wird auch hier ein Unterschied bei der Verwendung von fünf oder zehn Attributen pro Sicht ersichtlich. Die Laufzeit bei Sichten mit zehn Attributen ist um einen Faktor von fast zwei höher.

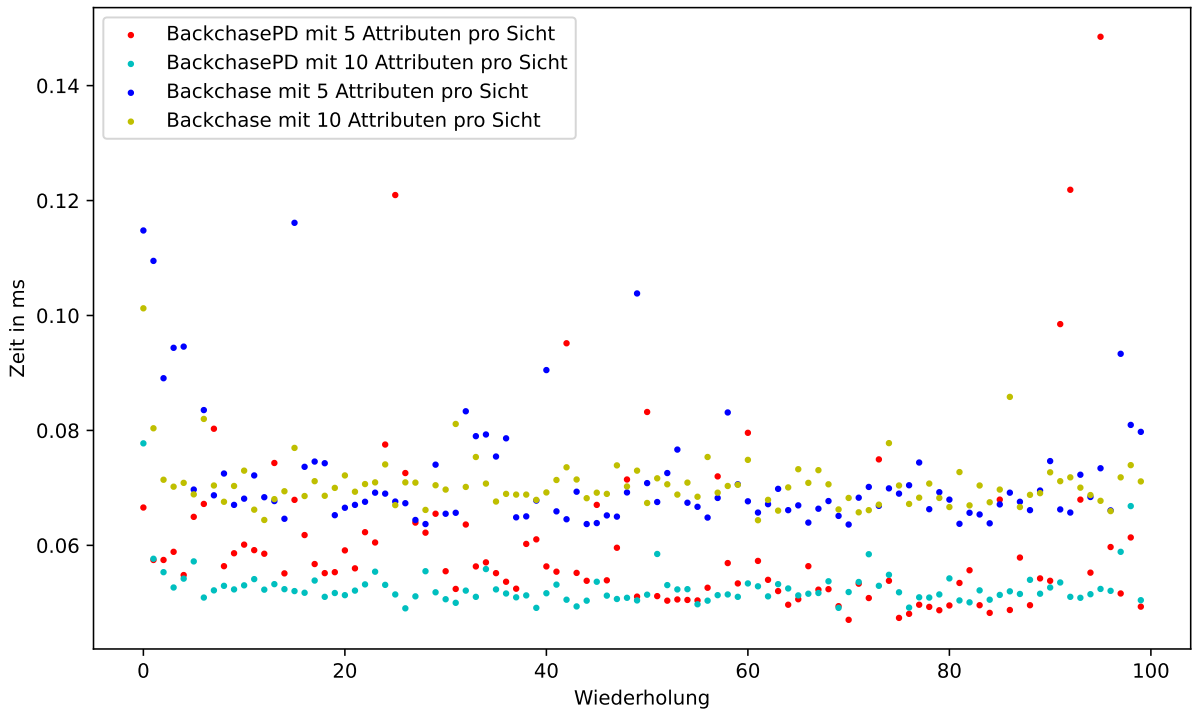


Abbildung 6.3 Performance der implementierten Algorithmen bei zwei Sichten

Abbildung 6.3 vergleicht den `Backchase`- und den `BackchasePD`-Prototyp bei Universalplänen mit nur zwei Atomen. Es ist direkt erkennbar, dass die benötigte Zeit der Prototypen zum Teil inkonsistent erscheint. Diese Inkonsistenz kann viele Gründe haben. Die Ausführungszeit von Prozessen in realen Systemen unterliegt Schwankungen. Die durchschnittliche Ausführungszeit der BACKCHASE-Phase des provenance-basierten Prototyps liegt bei beiden Fällen bei ca. $0,06$ Millisekunden. Der CHASE&BACKCHASE-Prototyp mit fünf bzw. zehn Attributen pro Sicht liegt hierbei mit einer Laufzeit von ca. $0,075$ Millisekunden nur geringfügig höher.
Der Performance-Unterschied zwischen dem CHASE&BACKCHASE- und dem Provenance-Directed CHASE&BACKCHASE-Prototypen ist gerade bei Universalplänen mit wenigen Sichten vernachlässigbar klein. Die Laufzeit ist jedoch schon bei Universalplänen mit fünf Atomen je fünf Attributen um einen Faktor von fast elf höher als die Laufzeit der provenance-basierten Variante. Hierbei muss jedoch darauf geachtet werden, dass die getesteten Anfragen von einfacher Natur waren. Die Integritätsbedingungen enthielten nur Sichtbedingungen. Zudem umfassten die Sichten nur ein Atom. Trotz der Einschränkungen zeigt dieser Test grob, wo die Grenzen des CHASE&BACKCHASE-Prototyps liegen.
Der provenance-basierte Prototyp konnte in den Tests eine relativ gute Laufzeit erzielen. Diese Laufzeit muss bei komplexeren Anfragen nicht notwendigerweise identisch sein. Anders als beim vorigen Prototyp ist die Laufzeit hier nicht exponentiell, sondern höchstens polynomiell. Zudem lässt sich dieser Prototyp auch für Universalpläne mit mehreren Atomen verwenden. Der `BackchasePD`-Prototyp ist für alltägliche Anfragen demnach besser geeignet, als der `Backchase`-Prototyp.

6.3 Ausblick

Im Rahmen dieser Arbeit wurde der CHASE&BACKCHASE und der Provenance-Directed CHASE&BACKCHASE von [DH13] implementiert. Die unterstützten Anfragen waren hierbei auf TGDs be-

schränkt. Mithilfe dieser Bedingungen war es möglich, Beziehungen zwischen Atomen näher zu beschreiben. Es konnten dadurch neue Atome aus bereits bestehenden Atomen generiert werden. Der Provenance-Directed CHASE&BACKCHASE funktioniert nach [Ile14] auch unter Anwendung von EGDs. Durch die Verwendung von EGDs können beispielsweise Schlüsselbedingungen in ein Atom eingebaut werden. Eine Optimierung des Provenance-Directed CHASE&BACKCHASE wäre, wie in [Ile14] vorgestellt, ein kostenbasiertes Verfahren. Die Effizienz des provenance-basierten Prototyps kann dadurch erhöht werden.
Durch die Anbindung von ChaTEAU an ein Datenbankmanagementsystem kann das AQuV- und das Datenintegrationsproblem auch an realen Systemen betrachtet werden. Diese Anbindung unterliegt jedoch ein paar Problemen. Die implementierten Prototypen unterstützen derzeit nur konjunktive Anfragen. SQL-Anfragen müssen demnach zuerst in konjunktive Anfragen umgewandelt werden. Im Studentenprojekt [Kav+21] der Universität Rostock wurden erste Ansätze erbracht, mit denen SQL-Anfragen in konjunktive Anfragen umgewandelt werden können. Der Umfang der unterstützten Anfrageoperationen ist jedoch beschränkt. Der programmierte Prototyp aus dem Studentenprojekt erstellt XML-Dateien, auf denen ChaTEAU den CHASE ausführen kann. Die generierten XML-Dateien unterstützen derzeit nur Instanzen. Die Anwendung des CHASE&BACKCHASE und des Provenance-Directed CHASE&BACKCHASE wird demnach nicht unterstützt.
Die in dieser Arbeit implementierten Prototypen beschränken die Menge an Vergleichsoperatoren in konjunktiven Anfragen auf das Gleichheitszeichen. Diese Beschränkungen verhindern es, komplexe Anfragen von realen Systemen zu umschreiben. Das Stellen von Bereichsanfragen ist derzeit nicht möglich. Hierfür muss zuerst die Menge an Vergleichsoperatoren erweitert werden. Es ist zudem nicht möglich, Mengenoperationen auf Anfragen auszuführen. Auch SQL-spezifische Operationen, darunter Skalarfunktionen, Aggregatfunktionen und Gruppierungen, können derzeit nicht erfasst werden.
Aufgrund dieser Einschränkungen können nur sehr einfache SQL-Anfragen unterstützt werden.

Literaturverzeichnis

[AH19] Tanja Auge und Andreas Heuer. "ProSA—Using the CHASE for Provenance Management". In: *Advances in Databases and Information Systems*. Hrsg. von Tatjana Welzer, Johann Eder, Vili Podgorelec und Aida Kamišalić Latifić. Cham: Springer International Publishing, 2019, S. 357–372. ISBN: 978-3-030-28730-6.

[Aug+20] Tanja Auge, Jasper Roloff, Andreas Heuer, Andreas Görres, Ruven Kronenberg, Willi Brekenfelder, Jakob Zimmer, Maximilian Lamster und Erik Manthey. "Testing CHASE-based Tools for Data Exchange and Query Rewriting". Technischer Bericht. 2020. 23 S.

[Ben+17] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro und Efthymia Tsamoura. "Benchmarking the Chase". In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '17. event-place: Chicago, Illinois, USA. New York, NY, USA: Association for Computing Machinery, 2017, S. 37–52. ISBN: 978-1-4503-4198-1. DOI: 10.1145/3034786.3034796.

[CCT09] James Cheney, Laura Chiticariu und Wang-chiew Tan. "Provenance in Databases: Why, How, and Where". In: *Foundations and Trends in Databases* 1 (1. Jan. 2009), S. 379–474. DOI: 10.1561/1900000006.

[CLL06] Rada Chirkova, Chen Li und Jia Li. "Answering queries using materialized views with minimum size". In: *The VLDB Journal* 15.3 (2006). Publisher: Springer, S. 191–210.

[DH13] Alin Deutsch und Richard Hull. "Provenance-Directed Chase&Backchase". In: *In Search of Elegance in the Theory and Practice of Computation: Essays Dedicated to Peter Buneman*. Hrsg. von Val Tannen, Limsoon Wong, Leonid Libkin, Wenfei Fan, Wang-Chiew Tan und Michael Fourman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 227–236. ISBN: 978-3-642-41660-6. DOI: 10.1007/978-3-642-41660-6_11.

[DHI12] AnHai Doan, Alon Halevy und Zachary Ives. "2 - Manipulating Query Expressions". In: *Principles of Data Integration*. Hrsg. von AnHai Doan, Alon Halevy und Zachary Ives. Boston: Morgan Kaufmann, 1. Jan. 2012, S. 21–63. ISBN: 978-0-12-416044-6. DOI: 10.1016/B978-0-12-416044-6.00002-8.

[Gör20] Andreas Görres. "Erweiterung des CHASE-Werkzeugs ChaTEAU um ein Terminierungskriterium". Masterarbeit. Universität Rostock, 2020. 175 S.

[Hal01] Alon Y. Halevy. "Answering queries using views: A survey". In: *The VLDB Journal* 10.4 (1. Dez. 2001), S. 270–294. DOI: 10.1007/s007780100054.

[HL00] A.Y. Halevy und L. Libkin. "Theory of answering queries using views". In: *SIGMOD Record (ACM Special Interest Group on Management of Data)* 29 (1. Dez. 2000), S. 40–47.

[Ile+14] Ioana Ileana, Bogdan Cautis, Alin Deutsch und Yannis Katsis. "Complete yet Practical Search for Minimal Query Reformulations under Constraints". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. event-place: Snowbird, Utah, USA. New York, NY, USA: Association for Computing Machinery, 2014, S. 1015–1026. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2593683.

[Ile14] Ioana Ileana. “Query rewriting using views : a theoretical and practical perspective”. Issue: 2014ENST0062. Theses. Télécom ParisTech, Okt. 2014.

[Kav+21] Ivo Kavisanczki, Tobias Rudolph, Tom Siegl und Marian Zuska. *Projektdokumentation sql2sttgd*. 2021.

[LMS95] Alon Y Levy, Alberto O Mendelzon und Yehoshua Sagiv. “Answering queries using views”. In: *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 1995, S. 95–104.

[LRO96] Alon Y. Levy, Anand Rajaraman und Joann J. Ordille. “Querying Heterogeneous Information Sources Using Source Descriptions”. In: *Proceedings of the 22th International Conference on Very Large Data Bases*. VLDB ’96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, S. 251–262. ISBN: 1-55860-382-4.

[Mar18] Jurklies Martin. “CHASE und BACKCHASE: Entwicklung eines Universal-Werkzeugs für eine Basistechnik der Datenbankforschung”. Masterarbeit. Universität Rostock, 2018. 81 S.

[PH01] Rachel Pottinger und Alon Halevy. “MiniCon: A scalable algorithm for answering queries using views”. In: *The VLDB Journal* 10.2 (1. Sep. 2001), S. 182–198. DOI: 10.1007/s007780100048.

[Ren19] Fabian Renn. “Erweiterung des CHASE-Werkzeugs ChaTEAU um Anfragetransformationen”. Bachelorarbeit. Universität Rostock, 2019. 72 S.

[Ros20] Florian Rose. “Erweiterung des CHASE-Werkzeugs ChaTEAU um eine BACKCHASE-Phase”. Masterarbeit. Universität Rostock, 2020. 88 S.

[SSH18] Gunter Saake, Kai-Uwe Sattler und Andreas Heuer. *Datenbanken – Konzepte und Sprachen*. Frechen, GERMANY: mitp, 2018. ISBN: 978-3-95845-777-5.

[Zim19] Jakob Zimmer. “Vereinheitlichung des CHASE auf Instanzen und Anfragen am Beispiel ChaTEAU”. Bachelorarbeit. Universität Rostock, 2019. 89 S.

[Zim21] Jakob Zimmer. “Datenintegration durch inverse Schemaabbildungen: Erweiterung der Rostocker GaLVE-Technik”. Masterarbeit. Universität Rostock, 2021. 197 S.

Abkürzungsverzeichnis

AQuV Answering-Queries-using-Views.

ChaTEAU Chase for Transforming, Evolving, and Adapting databases and queries, Universal approach.

CLOB Character-Large-Object.

CSV Comma-separated Values.

DNF Disjunktive Normalform.

EGD Equality-Generating-Dependency.

FD Functional-Dependency, Funktionale Abhängigkeit.

GaLVE Global-as-Local-View-Extension.

JD Join-Dependency, Verbundabhängigkeit.

MCD MiniCon-Description.

PD Provenance-Directed.

s-t-TGD Source-to-Target Tuple-Generating-Dependency.

TGD Tuple-Generating-Dependency.

XML Extensible-Markup-Language.

Anhang A

Beispiele

A.1 CHASE&BACKCHASE

Abbildung A.1: CHASE&BACKCHASE

$$Q(vn, nn, titel) :- \texttt{REZENSIONEN}(lid, bid, bew, inh), \texttt{AUTOREN}(aid, vn, nn), \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$$

$$\mathcal{U} = V_{\texttt{LIEST}}(lid, bid), V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel), V_{\texttt{AUTORENLISTE}}(aid, vn, nn)$$

$$B_{V_{\texttt{LIEST}}} : \forall lid, bid : V_{\texttt{LIEST}}(lid, bid) \rightarrow \exists bew, inh : \texttt{REZENSIONEN}(lid, bid, bew, inh)$$

$$B_{V_{\texttt{LIEST_REZ}}} : \forall lid, bid, bew, titel : V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel) \rightarrow \exists inh, aid, verl, ver, pr : \texttt{REZENSIONEN}(lid, bid, bew, inh), \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$$

$$B_{V_{\texttt{BÜCHERLISTE}}} : \forall bid, aid, titel : V_{\texttt{BÜCHERLISTE}}(bid, aid, titel) \rightarrow \exists verl, ver, pr : \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$$

$$B_{V_{\texttt{AUTORENLISTE}}} : \forall aid, vn, nn : V_{\texttt{AUTORENLISTE}}(aid, vn, nn) \rightarrow \texttt{AUTOREN}(aid, vn, nn)$$

Man betrachte zuerst den Teilplan $Q'(vn, nn, titel) :- V_{\texttt{LIEST}}(lid, bid), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel), V_{\texttt{AUTORENLISTE}}(aid, vn, nn)$.

$$V_{\texttt{LIEST}}(lid, bid), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel), V_{\texttt{AUTORENLISTE}}(aid, vn, nn)$$
$$\Downarrow B_{V_{\texttt{LIEST}}}$$
$$V_{\texttt{LIEST}}(lid, bid), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel), V_{\texttt{AUTORENLISTE}}(aid, vn, nn), \texttt{REZENSIONEN}(lid, bid, bew_1, inh_1)$$
$$\Downarrow B_{V_{\texttt{BÜCHERLISTE}}}$$
$$V_{\texttt{LIEST}}(lid, bid), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel), V_{\texttt{AUTORENLISTE}}(aid, vn, nn), \texttt{REZENSIONEN}(lid, bid, bew_1, inh_1), \texttt{BÜCHER}(bid, aid, titel, verl_2, ver_2, pr_2)$$
$$\Downarrow B_{V_{\texttt{ATUORENLISTE}}}$$

$$V_{\texttt{LIEST}}(lid, bid), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel), V_{\texttt{AUTORENLISTE}}(aid, vn, nn),$$
$$\texttt{REZENSIONEN}(lid, bid, bew_1, inh_1), \texttt{BÜCHER}(bid, aid, titel, verl_2, ver_2, pr_2),$$
$$\texttt{AUTOREN}(aid, vn, nn)$$

Unter Verwendung des Homomorphismus H' wird erkenntlich, dass der Teilplan Q' ein Rewriting der Originalanfrage ist.

$$H' = \{lid \to lid, bid \to bid, bew \to bew_1, inh \to inh_1, aid \to aid, vn \to vn, nn \to nn, titel \to titel,$$
$$verl \to verl_2, ver \to ver_2, pr \to pr_2\}$$

Nun wird der Teilplan $Q''(vn, nn, titel) :- V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel), V_{\texttt{AUTORENLISTE}}(aid, vn, nn)$ betrachtet.

$$V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel), V_{\texttt{AUTORENLISTE}}(aid, vn, nn)$$
$$\Downarrow B_{V_{\texttt{LIEST_REZ}}}$$
$$V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel), V_{\texttt{AUTORENLISTE}}(aid, vn, nn)$$
$$\texttt{REZENSIONEN}(lid, bid, bew, inh_1), \texttt{BÜCHER}(bid, aid_2, titel, verl_2, ver_2, pr_2)$$
$$\Downarrow B_{V_{\texttt{BÜCHERLISTE}}}$$
$$V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel), V_{\texttt{AUTORENLISTE}}(aid, vn, nn)$$
$$\texttt{REZENSIONEN}(lid, bid, bew, inh_1), \texttt{BÜCHER}(bid, aid_2, titel, verl_2, ver_2, pr_2)$$
$$\texttt{BÜCHER}(bid, aid, titel, verl_3, ver_3, pr_3)$$
$$\Downarrow B_{V_{\texttt{AUTORENLISTE}}}$$
$$V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel), V_{\texttt{AUTORENLISTE}}(aid, vn, nn)$$
$$\texttt{REZENSIONEN}(lid, bid, bew, inh_1), \texttt{BÜCHER}(bid, aid_2, titel, verl_2, ver_2, pr_2)$$
$$\texttt{BÜCHER}(bid, aid, titel, verl_3, ver_3, pr_3), \texttt{AUTOREN}(aid, vn, nn)$$

Mithilfe des Homomorphismus H'' kann die Originalanfrage auf die gechasete Anfrage abgebildet werden. Der Teilplan Q'' ist demnach ein Rewriting der Originalanfrage.

$$H'' = \{lid \to lid, bid \to bid, bew \to bew, inh \to inh_1, aid \to aid, vn \to vn, nn \to nn, titel \to titel,$$
$$verl \to verl_3, ver \to ver_3, pr \to pr_3\}$$

Auch der Teilplan $Q'''(vn, nn, titel) :- V_{\texttt{LIEST_REZ}}(lid, bid, bew, titel), V_{\texttt{BÜCHERLISTE}}(bid, aid, titel), V_{\texttt{AUTORENLISTE}}(aid, vn, nn), V_{\texttt{LIEST}}(lid, bid)$ ist ein Rewriting der Originalanfrage. Sowohl der Homomorphismus H' als auch H'' kann verwendet werden, um die Originalanfrage auf die gechasete Anfrage abzubilden.

A.2 CHASE&BACKCHASE – Pruning

Abbildung A.2: CHASE&BACKCHASE – Pruning
$Q(bid) :- \texttt{REZENSIONEN}(lid, bid, bew, inh), \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$
$\mathcal{U} = V_{\texttt{REZ}}(bid), V_{\texttt{BUCH}}(bid)$

$B_{V_{\texttt{REZ}}}: \forall bid: V_{\texttt{REZ}}(bid) \rightarrow \exists lid, bew, inh: \texttt{REZENSIONEN}(lid, bid, bew, inh)$
$B_{V_{\texttt{BUCH}}}: \forall bid: V_{\texttt{BUCH}}(bid) \rightarrow \exists aid, titel, verl, ver, pr: \texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$
$B_I: \forall lid, bid, bew, inh: \texttt{REZENSIONEN}(lid, bid, bew, inh) \rightarrow \exists aid, titel, verl, ver, pr:$
$\texttt{BÜCHER}(bid, aid, titel, verl, ver, pr)$

Zuerst wird der Teilplan $Q'(bid) :- V_{\texttt{REZ}}(bid)$ betrachtet.

$$V_{\texttt{REZ}}(bid)$$
$$\Downarrow B_{V_{\texttt{REZ}}}$$
$$V_{\texttt{REZ}}(bid), \texttt{REZENSIONEN}(lid_1, bid, bew_1, inh_1)$$
$$\Downarrow B_I$$
$$V_{\texttt{REZ}}(bid), \texttt{REZENSIONEN}(lid_1, bid, bew_1, inh_1), \texttt{BÜCHER}(bid, aid_2, titel_2, verl_2, ver_2, pr_2)$$

Nun wird überprüft, ob ein Homomorphismus von der Originalanfrage in die gechasete Anfrage existiert. In diesem Fall existiert ein Homomorphismus H. Der Teilplan ist demnach ein Rewriting der Originalanfrage.

$H = \{lid \rightarrow lid_1, bid \rightarrow bid, bew \rightarrow bew_1, inh \rightarrow inh_1, aid \rightarrow aid_2, titel \rightarrow titel_2, verl \rightarrow verl_2, ver \rightarrow ver_2,$
$pr \rightarrow pr_2\}$

Nun wird der Teilplan $Q''(bid) :- V_{\texttt{BUCH}}(bid)$ betrachtet.

$$V_{\texttt{BUCH}}(bid)$$
$$\Downarrow B_{V_{\texttt{BUCH}}}$$
$$V_{\texttt{BUCH}}(bid), \texttt{BÜCHER}(bid, aid_1, titel_1, verl_1, ver_1, pr_1)$$

Nun wird überprüft, ob ein Homomorphismus von der Originalanfrage in die gechasete Anfrage existiert. In diesem Fall existiert kein Homomorphismus. Die gechasete Anfrage enthält nicht das REZENSIONEN-Atom. Der Teilplan ist demnach kein Rewriting der Originalanfrage.
Der Teilplan $Q'''(bid)\, V_{\texttt{REZ}}(bid), V_{\texttt{BUCH}}(bid)$ muss nicht weiter betrachtet werden, da es den Teilplan Q' enthält. Q''' ist demnach ein Rewriting der Originalanfrage.

Anhang B

Quellcode

Die Methoden und Klassen der hier angegebenen Quellcodeausschnitte wurden, sofern nicht anders angegeben, eigenständig erstellt. Bei Methoden, die erweitert wurden, sind die eigenständig erstellten Codebereiche vor dem Quellcodeausschnitt angegeben. In einigen Fällen kann es dazu kommen, dass die abgebildete Methode einer anderen Methode von der Funktionsweise ähnelt. Bei den betreffenden Ausschnitten wird dies zuvor erwähnt.

B.1 Grundgerüst

Quellcodeausschnitt B.1.1 AbstractBackchase.java

```
1  public abstract class AbstractBackchase {
2
3    // the chased query restricted only to atoms with schema tag
     TARGET
4    private Instance universalPlan;
5
6    // the inverted integrity constraints
7    private LinkedHashSet<IntegrityConstraint> integrityConstraints;
8
9    // the original (not chased) query
10   private Instance originalQuery;
11
12   /**
13    * Calculate possible rewritings of the {@link #originalQuery}
      instance
14    * @return Set of rewritings
15    * @see Instance
16    */
17   public abstract Set<Instance> backchase();
18
19   /**
20    * Get the {@link #originalQuery}
21    * @return {@link #originalQuery}
22    * @see Instance
23    */
24   public Instance getOriginalQuery() {
```

```
25      return originalQuery;
26    }
27
28    /**
29     * Set the {@link #originalQuery}
30     * @param originalQuery the original (not chased) query
31     */
32    public void setOriginalQuery(Instance originalQuery) {
33      this.originalQuery = originalQuery;
34    }
35
36    /**
37     * Get the {@link #universalPlan}
38     * @return {@link #universalPlan}
39     * @see Instance
40     */
41    public Instance getUniversalPlan() {
42      return universalPlan;
43    }
44
45    /**
46     * Set the {@link #universalPlan}
47     * @param universalPlan the chased query restricted only to
       atoms with schema tag target
48     * @see SchemaTag#TARGET
49     */
50    public void setUniversalPlan(Instance universalPlan) {
51      this.universalPlan = universalPlan;
52    }
53
54    /**
55     * Get the {@link #integrityConstraints}
56     * @return Set of integrity constraints
57     * @see IntegrityConstraint
58     */
59    public LinkedHashSet<IntegrityConstraint>
      getIntegrityConstraints() {
60      return integrityConstraints;
61    }
62
63    /**
64     * Set the {@link #integrityConstraints}
65     * @param integrityConstraints the inverted sttgds and
       non-inverted tgds
66     * @see IntegrityConstraint
67     * @see Provenance#invert()
68     */
69    public void
      setIntegrityConstraints(LinkedHashSet<IntegrityConstraint>
      integrityConstraints) {
70      this.integrityConstraints = integrityConstraints;
```

```
71    }
72
73    /**
74     * Class constructor. Calculates the universal plan and
      inverted constraints
75     * @param chasedInstance the chased instance
76     * @param integrityConstraints the not inverted integrity
      constraints
77     * @param originalQuery the original (not chased) query
78     * @see Instance
79     * @see IntegrityConstraint
80     */
81    public AbstractBackchase(Instance chasedInstance,
      LinkedHashSet<IntegrityConstraint> integrityConstraints,
      Instance originalQuery) {
82      this.originalQuery = originalQuery;
83
84      //calculate the universal plan
85      this.universalPlan = chasedInstance.calculateUniversalPlan();
86
87      //invert the integrity constraints
88      this.integrityConstraints = new Provenance(chasedInstance,
        integrityConstraints)
89          .invert()
90          .getSigma();
91    }
92
93    /**
94     * Get the to be chased instances
95     * @return a list of instances
96     * @see Instance
97     */
98    protected abstract List<Instance> getQueries();
99
100   /**
101    * Calculates a string of rewritings.
102    * @param rewritings a hash set of rewritings
103    * @return Returns a string of rewritings.
104    */
105   protected String getRewritingString(Set<Instance> rewritings) {
106     String rewritingsString = "";
107     for (Instance rewriting : rewritings) {
108       rewritingsString +=
109           "Rewriting\n" +
110           "---------\n";
111       for (RelationalAtom relationalAtom :
          rewriting.getRelationalAtoms()) {
112         rewritingsString += "" +
113             "Relational Atom: " + relationalAtom + "\n";
114       }
115       rewritingsString += "\n";
```

```
116       }
117       return rewritingsString;
118     }
119   }
```

Quellcodeausschnitt B.1.2 Instance.java - calculateUniversalPlan-Methode

```
1   public class Instance {
2     ...
3     /**
4      * Calculates the universal plan of the instance
5      * @return An instance where the relational atoms are
       restricted to atoms with schema tag TARGET (e.g. views)
6      * @see Instance
7      * @see SchemaTag
8      */
9     public Instance calculateUniversalPlan() {
10      if(this.getOriginTag() == OriginTag.INSTANCE) {
11        ChateauLogger.log(Level.SEVERE, "cannot calculate universal
          plan of an instance");
12        return null;
13      }
14      ChateauLogger.log(Level.INFO, "calculating universal
        plan...");
15      LinkedHashSet<RelationalAtom> universalplan = new
        LinkedHashSet<>();
16
17      // for each relational atom of this.relationalAtoms ...
18      this.relationalAtoms.forEach(relationalAtom -> {
19        // if the relational atom has the schema tag TARGET than
          add the relational atom to the universal plan
20        if (this.getSchemaTags().get(relationalAtom.getName()) ==
          SchemaTag.TARGET)
21          universalplan.add(relationalAtom);
22      });
23
24      return new Instance(universalplan, this.schema,
        this.originTag, this.schemaTags, this.headAtoms);
25    }
26    ...
27  }
```

Die `isHomomorphicallyContainedInTest`-Methode der Instance-Klasse ähnelt vom Aufbau der `isHomomorphicallyContainedIn`-Methode. Sie behebt einen Fehler der ursprünglichen Methode.

Quellcodeausschnitt B.1.3 Instance.java - isHomomorphicallyContainedInTest-Methode

```
1   public Instance {
2     ...
3     /**
4      * Checks if this instance is homomorphically contained in
       another given
5      * instance.
6      *
```

```
7     * @param other the other instance to where a homomorphism
      should exist
8     * @return {@code true} if there is a homomorphism from this
      instance to the
9     *         other, {@code false} otherwise
10    */
11   public boolean isHomomorphicallyContainedInTest(Instance other)
     {
12     return buildHomomorphism(this.getRelationalAtoms(),
       other.getRelationalAtoms(),new Homomorphism());
13   }
14
15   /**
16    * Checks if this instance is homomorphically contained in
      another given
17    * instance.
18    *
19    * @param source the source instance
20    * @param target the target instance to where a homomorphism
      should exist
21    * @param homomorphism a homomorphism from source to target
22    * @return {@code true} if there is a homomorphism from this
      instance to the
23    *         other, {@code false} otherwise
24    */
25   private static boolean
     buildHomomorphism(HashSet<RelationalAtom> source,
     HashSet<RelationalAtom> target, Homomorphism homomorphism) {
26     //create a new copy from the source instance
27     HashSet<RelationalAtom> sourceCopy = new HashSet<>(source);
28
29     for (RelationalAtom atomThis : source) {
30       boolean containmentFound = false;
31
32       //check containment from the current source atom to each
         target atom
33       for (RelationalAtom atomOther : target) {
34         //check containment
35         containmentFound = containmentFound ||
           atomThis.hasHomomorphismTo(atomOther);
36
37         if (containmentFound) {
38           //add term bindings to homomorphism
39           Homomorphism tmpHomomorphism = new
             Homomorphism(homomorphism);
40           for (int i = 0; i < atomOther.getTerms().size(); i++) {
41             Term sourceTerm = atomThis.getTerms().get(i);
42             Term targetTerm = atomOther.getTerms().get(i);
43             TermMapping newMapping = new TermMapping(sourceTerm,
               targetTerm);
```

```
44              containmentFound =
                tmpHomomorphism.addMapping(newMapping);
45
46
47              //check if mapping was successful
48              if (!containmentFound) {
49                boolean contained = false;
50                for (TermMapping termMapping :
                  tmpHomomorphism.getTermMappings()) {
51                  if (termMapping.getMappingSource().
52                    equals(sourceTerm)) {
53                    if (targetTerm.isVariable()) {
54                      //create new helper homomorphism
55                      Homomorphism newHomomorphism = new
                        Homomorphism(tmpHomomorphism);
56
57                      //add failed term mapping to newHomomorphism
58                      newHomomorphism.getTermMappings()
59                      .remove(termMapping);
60                      newHomomorphism.addMapping(new
                        TermMapping(sourceTerm, targetTerm));
61
62                      HashSet<RelationalAtom> targetCopy = new
                        HashSet<>(target);
63                      HashSet<Term> remapRelations = new
                        HashSet<>();
64
65                      //for each relation atom of target instance...
66                      for (RelationalAtom relationalAtom : target) {
67                        //... check if the atom contains the to be
                          mapped term
68                        if (relationalAtom.getTerms()
69                          .contains(termMapping
70                          .getMappingTarget())) {
71                          //remove the atom
72                          targetCopy.remove(relationalAtom);
73
74                          //add all irrelevant terms
75                          remapRelations.addAll(
76                            relationalAtom.getTerms());
77                          }
78                        }
79
80                        // add all source atoms that need to be
                          remapped
81                        for (RelationalAtom relationalAtom :
                          source) {
82                          if (relationalAtom.getTerms().contains(
83                            termMapping.getMappingSource())) {
84                            sourceCopy.add(relationalAtom);
85                          }
```

```
86                          }
87
88                          //delete all impossible remappings
89                          for (Term remapRelation : remapRelations) {
90                            for (TermMapping irrelevantMapping :
                              homomorphism.getTermMappings()) {
91                              if (irrelevantMapping.getMappingTarget()
92                                .equals(remapRelation))
93                                newHomomorphism.getTermMappings()
94                                  .remove(irrelevantMapping);
95                            }
96                          }
97
98                          //check if new homomorphism is contained
99                          contained = buildHomomorphism(sourceCopy,
                            targetCopy, newHomomorphism);
100
101                         //if it is than return true
102                         if (contained)
103                           return true;
104
105                         //rollback tmpHomomorphism
106                         tmpHomomorphism = homomorphism;
107                         break;
108                       }
109                       //no containment
110                       contained = false;
111                       break;
112                   }
113
114                 }
115                 if (!contained)
116                   break;
117               }
118             }
119             //add new term mappings to homomorphism
120             homomorphism = tmpHomomorphism;
121             break;
122           }
123         }
124         //if no containment was found return false
125         if (!containmentFound) {
126           return false;
127         }
128       }
129       //containment was found
130       return true;
131     }
132     ...
133   }
```

Quellcodeausschnitt B.1.4 Variable.java - invert-Methode

```
1  public class Variable extends Term {
2    private VariableType variableType;
3    private String indexName;
4    private int index;
5    ...
6    @Override
7    public Term invert() {
8      Variable newVar = this.copy();
9      if (variableType == VariableType.EXISTS)
10       newVar.variableType = VariableType.FOR_ALL;
11     else
12       newVar.variableType = VariableType.EXISTS;
13     return newVar;
14   }
15   ...
16 }
```

In der vorliegenden Methode wurden nur die Zeilen 12-18 und einzelne Zeilen ab Zeile 27 hinzugefügt beziehungsweise geändert.

Quellcodeausschnitt B.1.5 STTgd.java - invert-Methode

```
1  public class STTgd extends Tgd {
2    ...
3    @Override
4    public STTgd invert() {
5      // Construct new Body from Head
6      LinkedHashSet<RelationalAtom> newBody = new
       LinkedHashSet<RelationalAtom>();
7      for (RelationalAtom a : this.getHead()) {
8        ArrayList<Term> terms = a.getTerms();
9        ArrayList<Term> newTerms = new ArrayList<Term>();
10       for (Term t : terms) {
11         Term newTerm;
12         // if the term is a variable ...
13         if (t.isVariable()) {
14           // ... and the variable type is existential ...
15           if (((Variable) t).getVariableType() ==
             VariableType.EXISTS)
16             newTerm = t.invert(); // ... than invert the term
17           else newTerm = t.copy(); // otherwise copy it
18         } else newTerm = t.copy(); // if the term is not a
           variable than copy the term
19
20         newTerms.add(newTerm);
21       }
22       newBody.add(new RelationalAtom(a.getName(), newTerms));
23     }
24     // Construct new Head from Body
25     LinkedHashSet<RelationalAtom> newHead = new
       LinkedHashSet<RelationalAtom>();
26     LinkedHashSet<RelationalAtom> oldHead = this.getHead();
```

```
27
28      for (RelationalAtom a : this.getBody()) {
29        ArrayList<Term> terms = a.getTerms();
30        ArrayList<Term> newTerms = new ArrayList<Term>();
31        for (Term t : terms) {
32          Term newTerm = t.copy();
33          for (RelationalAtom atom : oldHead) {
34            if (!(atom.getTerms().contains(t)) && t.isVariable()) {
35              int countOccurrences = 0;
36              // for each relational atom in oldHead ...
37              for (RelationalAtom b : oldHead) {
38                // ... if it contains the term t
39                if (b.getTerms().contains(t)) {
40                  // increment countOccurences
41                  countOccurrences++;
42                  break;
43                }
44              }
45              if (countOccurrences > 0) {
46                // if the atom is contained in the old head than
                  continue
47                // because the term newTerm has already the
                  variable type FOR_ALL
48                continue;
49
50              }
51
52              // invert newTerm
53              newTerm = t.invert();
54
55            }
56          }
57          newTerms.add(newTerm);
58        }
59
60        newHead.add(new RelationalAtom(a.getName(), newTerms));
61      }
62      return new STTgd(newBody, newHead);
63    }
64  }
```

B.2 BACKCHASE

Quellcodeausschnitt B.2.1 Backchase.java

```
1  public class Backchase extends AbstractBackchase {
2    /**
3     * @see AbstractBackchase#AbstractBackchase(Instance,
      LinkedHashSet, Instance)
4     */
```

```
5   public Backchase(Instance instance,
    LinkedHashSet<IntegrityConstraint>
    integrityConstraints,Instance originalQuery) {
6     super(instance, integrityConstraints, originalQuery);
7   }
8
9   @Override
10  public Set<Instance> backchase() {
11    // calculate subqueries
12    ChateauLogger.log(Level.INFO, "Start backchase phase");
13    List<Instance> subqueries = this.getQueries();
14
15    Set<Instance> rewritings = new HashSet<>();
16
17    ChateauLogger.log(Level.INFO, "Start chasing the subqueries");
18    // for each subquery of subqueries ...
19    for (Instance subquery : subqueries) {
20      // ... chase the subquery
21      ChateauLogger.log(Level.INFO, "Chase subquery: " +
        subquery);
22      Instance chasedQuery = Chase.chase(subquery,
        this.getIntegrityConstraints());
23      ChateauLogger.log(Level.INFO, "Chased subquery: " +
        chasedQuery);
24
25      // if there exists a homomorphism from originalQuery to
        chasedQuery
26      // than the subquery is a rewriting
27      if(this.getOriginalQuery()
28        .isHomomorphicallyContainedInTest(chasedQuery)) {
29        rewritings.add(subquery);
30        ChateauLogger.log(Level.INFO, "Added  " + subquery + " to
          rewriting list");
31      }
32    }
33    ChateauLogger.log(Level.INFO, "End backchase phase");
34    ChateauLogger.log(Level.INFO, "Found rewritings: \n" +
      getRewritingString(rewritings));
35    return rewritings;
36  }
37
38  @Override
39  public List<Instance> getQueries() {
40    ChateauLogger.log(Level.INFO, "Calculate subqueries...");
41    List<Instance> queries = new ArrayList<>();
42
43    ChateauLogger.log(Level.INFO, "Get all atoms of universal
      plan...");
44    // get all relational atoms from the universal plan
45    Set<RelationalAtom> relationalAtoms = new LinkedHashSet
46        (this.getUniversalPlan().getRelationalAtoms());
```

```
47      List<Set<RelationalAtom>> subqueries = new ArrayList<>();
48
49      ChateauLogger.log(Level.INFO, "Calculate power set of the
        universal plan...");
50      // create the power set from relationalAtoms
51      powerSet(subqueries, relationalAtoms);
52      subqueries.sort(Comparator.comparing(Set::size));
53
54      ChateauLogger.log(Level.INFO, "Generate instances for
        subqueries ");
55      //add the subqueries to the queries array
56      for (Iterator<Set<RelationalAtom>> it =
        subqueries.iterator();it.hasNext();) {
57        Set<RelationalAtom> subquery = it.next();
58        //create instance with subquery as relational atoms
59        queries.add(new Instance (
60            (LinkedHashSet<RelationalAtom>) subquery,
61            this.getUniversalPlan().getSchema(),
62            this.getUniversalPlan().getOriginTag(),
63            this.getUniversalPlan().getSchemaTags()
64        ));
65      }
66      ChateauLogger.log(Level.INFO, "Subqueries were calculated");
67      return queries;
68    }
69
70
71    /**
72     * Create the power set of given set oldList
73     * @param newList the power set, initially empty
74     * @param oldList the relational atoms
75     */
76    private static void powerSet(List newList, Set oldList) {
77      Iterator oldIT = oldList.iterator();
78
79      // cannot add elements to list while iterating
80      Iterator<Set> newIT = new LinkedHashSet(newList).iterator();
81
82      Object first;
83      if (oldList.isEmpty())
84        return; // nothing to do, empty set is not necessary
85      else
86        first = oldIT.next(); // declare the first entry of oldList
87
88      // for each entry of newList
89      while(newIT.hasNext()) {
90        //select current subset of newList
91        Set currentSet = newIT.next();
92        Set modifiedSet = new LinkedHashSet(currentSet);
93
94        //add first to modified set
```

```
95          modifiedSet.add(first);
96
97          //add subquery to newList
98          newList.add(modifiedSet);
99        }
100       // remove used element
101       oldList.remove(first);
102
103
104       HashSet tmp = new LinkedHashSet();
105       tmp.add(first);
106       newList.add(tmp);
107
108       // if oldList is not empty build the powerSet with the
          changed newList and oldList
109       if (oldIT.hasNext())
110         powerSet(newList,oldList);
111
112     }
113   }
```

B.3 BACKCHASE-Pruning

Quellcodeausschnitt B.3.1 BackchasePruning.java

```
1   public class BackchasePruning extends Backchase {
2     /**
3      * @see AbstractBackchase#AbstractBackchase(Instance,
       LinkedHashSet, Instance)
4      */
5     public BackchasePruning(Instance instance,
      LinkedHashSet<IntegrityConstraint> integrityConstraints,
      Instance originalQuery) {
6       super(instance, integrityConstraints, originalQuery);
7     }
8
9     @Override
10    public Set<Instance> backchase() {
11      ChateauLogger.log(Level.INFO, "Start backchase phase with
        pruning");
12      List<Instance> subqueries = this.getQueries();
13      Set<Instance> rewritings = new HashSet<>();
14
15      ChateauLogger.log(Level.INFO, "Start chasing the subqueries");
16      // for each subquery ...
17      ChateauLogger.log(Level.INFO, "Check if subquery contains a
        rewriting");
18      for (Instance subquery : subqueries) {
19        boolean contained = false;
20        for (Instance rewriting: rewritings) {
```

```
21          // ... check if the subquery is already contained in
            rewritings
22          if (rewriting.isHomomorphicallyContainedInTest(subquery))
            {
23            ChateauLogger.log(Level.INFO, "Subquery is already a
              rewriting");
24            // if subquery is equal to rewriting than break the loop
25            contained = true;
26            break;
27          }
28        }
29
30        // if the subquery is already contained in rewritings than
          check the next subquery
31        if (contained) {
32          ChateauLogger.log(Level.INFO, "Check next subquery");
33          continue;
34        }
35
36        ChateauLogger.log(Level.INFO, "Chase subquery: " +
          subquery);
37        // if it is not contained than chase the subquery
38        Instance chasedQuery = Chase.chase(subquery,
          this.getIntegrityConstraints());
39        ChateauLogger.log(Level.INFO, "Chased subquery: " +
          chasedQuery);
40
41        // if there exists a homomorphism from the original query
          to the chased query
42        // than the subquery is a rewriting
43        if(this.getOriginalQuery()
44          .isHomomorphicallyContainedInTest(chasedQuery)) {
45          ChateauLogger.log(Level.INFO, "Added  " + subquery + " to
            rewriting list");
46          rewritings.add(subquery);
47        }
48      }
49      ChateauLogger.log(Level.INFO, "End backchase phase");
50      ChateauLogger.log(Level.INFO, "Found rewritings: \n" +
        getRewritingString(rewritings));
51      return rewritings;
52    }
53  }
```

B.4 Provenance-Directed CHASE&BACKCHASE

Quellcodeausschnitt B.4.1 BackchasePD.java

```
1  public class BackchasePD extends AbstractBackchase {
2    Map<String, RelationalAtom> provID;
```

```
3   /**
4    * @see AbstractBackchase#AbstractBackchase(Instance,
     LinkedHashSet, Instance)
5    */
6   public BackchasePD(Instance chasedInstance,
    LinkedHashSet<IntegrityConstraint> integrityConstraints,
    Instance originalQuery) {
7     super(chasedInstance, integrityConstraints, originalQuery);
8     provID = new HashMap();
9   }
10
11  @Override
12  public Set<Instance> backchase() {
13    ChateauLogger.log(Level.INFO, "Start provenance-directed
      backchase phase");
14
15    //set why provenance for view atoms
16    setViewProvenance();
17
18    ChateauLogger.log(Level.INFO, "Set chase algorithm to
      provenance-aware chase");
19    Chase chase = new Chase(this.getUniversalPlan(),
      this.getIntegrityConstraints());
20
21    // configure chase algorithm to perform a provenance aware
      chase
22    chase.setAQUV(true);
23
24    ChateauLogger.log(Level.INFO, "Provenance-aware chase the
      universal plan");
25    // provenance-aware chase the query
26    Instance chasedQuery = chase.chase();
27
28    ChateauLogger.log(Level.INFO, "Get all atoms from the chased
      query with schema tag SOURCE");
29    // get relational atoms with schema tag SOURCE
30    Set<RelationalAtom> relationalAtomsWithoutViews =
      getRelationalAtomsWithoutViews(chasedQuery);
31
32    ChateauLogger.log(Level.INFO, "Calculate possible
      rewritings");
33    // mapping of chased atoms to relational names of the
      original query
34    Map<String, Set<RelationalAtom>> keys =
      this.getRelationKeys();
35    Map<RelationalAtom, Set<RelationalAtom>> keyRelationMap =
      this.getKeyRelationMap(keys, relationalAtomsWithoutViews);
36
37    // calculate the possible rewritings
```

```
38      Set<List<RelationalAtom>> cartesianProduct =
        com.google.common.collect.Sets.cartesianProduct(new
        ArrayList<>(keyRelationMap.values()));
39
40      // calculate the query's why-provenance
41      Set<Set<String>> queryWhyProvenance =
        this.getQueryWhyProvenance(cartesianProduct);
42
43      //return rewritings
44      Set<Instance> rewritings =
        this.getRewritings(queryWhyProvenance);
45
46      ChateauLogger.log(Level.INFO, "End provenance-directed
        backchase phase");
47
48
49      ChateauLogger.log(Level.INFO, "Found rewritings: \n" +
        getRewritingString(rewritings));
50      return rewritings;
51    }
52
53    /**
54     * Set the Why-Provenance of the atoms with schema tag TARGET
55     */
56    private void setViewProvenance() {
57      // for each view atom ...
58      for(RelationalAtom relationalAtom :
        this.getUniversalPlan().getRelationalAtoms()) {
59        // ... add the view provenance id to the why provenance
60        relationalAtom.getProvInfo().setId(relationalAtom
61          .generateProvenanceId(relationalAtom.getRelationName()));
62        List whyProvenance = new ArrayList<>();
63        whyProvenance.add(relationalAtom.getProvInfo().getId());
64        relationalAtom.getProvInfo().getWhyProvenance().add(
65          (ArrayList<String>) whyProvenance);
66
67        // add provenance id to provID
68        provID.put(relationalAtom.getProvInfo().getId(),
          relationalAtom);
69      }
70    }
71
72    /**
73     * Get the relational atom's names with schema tag SOURCE
74     * @return the relational atom's names with the corresponding
       atoms
75     * @see SchemaTag#SOURCE
76     */
77    private Map<String, Set<RelationalAtom>> getRelationKeys() {
78      Map<String, Set<RelationalAtom>> keys = new HashMap();
79
```

```
80      ChateauLogger.log(Level.INFO, "Save for each relation name
        the corresponding atoms " +
81          "of the original query");
82      // for each relational atom ...
83      this.getOriginalQuery().getRelationalAtoms().forEach(
84        relationalAtom -> {
85          // ... check if the atom's schema tag is source
86          if (this.getOriginalQuery().getSchemaTags().get(
87            relationalAtom.getName()) == SchemaTag.SOURCE) {
88            // add the atom's name and the atom to keys
89            if
              (!keys.containsKey(relationalAtom.getRelationName())) {
90              Set<RelationalAtom> container = new HashSet<>();
91              container.add(relationalAtom);
92              keys.put(relationalAtom.getRelationName(), container);
93            } else {
94              keys.get(relationalAtom.getRelationName())
95                .add(relationalAtom);
96            }
97          }
98      });
99
100     //return the names
101     return keys;
102   }
103
104   /**
105    * Get the rewritings from the query's why-provenance
106    * @param queryWhyProvenance the query's why-provenance
107    * @return the rewritings
108    * @see Instance
109    * @see ProvenanceInformation#getWhyProvenance()
110    */
111   private Set<Instance> getRewritings(Set<Set<String>>
      queryWhyProvenance) {
112     ChateauLogger.log(Level.INFO, "Generate rewritings from
        original query's why-provenance");
113
114     ChateauLogger.log(Level.INFO, "Transform the original query's
        why-provenance in " +
115         "disjunctive normal form");
116
117     Set<Instance> rewritings = new HashSet<>();
118
119     // for each rewriting's why-provenance ...
120     queryWhyProvenance.forEach(whyProvenance -> {
121       Set<RelationalAtom> views = new LinkedHashSet<>();
122
123       ChateauLogger.log(Level.INFO, "Generate rewriting of
          why-provenance " + whyProvenance);
```

```
124         // ... and for each provenance id from atom with schema tag
            TARGET ...
125         whyProvenance.forEach(provenanceID -> {
126           // ... add the relational atoms with the same provenance
              id
127           views.add(provID.get(provenanceID));
128         });
129
130         ChateauLogger.log(Level.INFO, "Add " + views + " to
            rewritings list");
131         // add the rewriting
132         rewritings.add(new Instance(
133           (LinkedHashSet<RelationalAtom>) views,
134           this.getUniversalPlan().getSchema(),
135           this.getUniversalPlan().getOriginTag(),
              this.getUniversalPlan().getSchemaTags(),
136           this.getUniversalPlan().getHeadAtoms()
137         ));
138       });
139
140       ChateauLogger.log(Level.INFO, "Rewritings have been
          calculated");
141       return rewritings;
142     }
143
144     /**
145      * Get the query's why-provenance
146      * @param potentialRewritings the potential rewritings
147      * @return the query's why-provenance
148      */
149     private Set<Set<String>>
        getQueryWhyProvenance(Set<List<RelationalAtom>>
        potentialRewritings) {
150       Set<Set<String>> queryWhyProvenance = new HashSet<>();
151
152       ChateauLogger.log(Level.INFO, "Calculate the original query's
          why-provenance");
153       // for each subquery atoms ...
154       for (List<RelationalAtom> subqueryAtoms :
          potentialRewritings) {
155         ChateauLogger.log(Level.INFO, "Generate instance for the
            subquery" + subqueryAtoms);
156         // build the subquery
157         Instance subquery = new Instance(new
            LinkedHashSet(subqueryAtoms),
            this.getUniversalPlan().getSchema(),
158             this.getUniversalPlan().getOriginTag(),
                this.getUniversalPlan().getSchemaTags(),
159             this.getUniversalPlan().getHeadAtoms());
160
```

```
161         ChateauLogger.log(Level.INFO, "Check if original query is
            contained in subquery");
162         // if there exists an homomorphism from the original query
            to the subquery ...
163         if
            (this.getOriginalQuery().isHomomorphicallyContainedInTest(
164           subquery)) {
165           Set<Set<Set<String>>> unnormalizedWhyProvenance = new
              HashSet<>();
166
167           ChateauLogger.log(Level.INFO, "Get the subquery's
              why-provenance");
168           // ... than for each atom
169
170           for (RelationalAtom atom : subqueryAtoms) {
171             Set<Set<String>> prov = new HashSet<>();
172
173
174             // get the why-provenance
175             atom.getProvInfo().getWhyProvenance().forEach(whyProv ->
176                 prov.add(new HashSet(whyProv)));
177             unnormalizedWhyProvenance.add(prov);
178           }
179           ChateauLogger.log(Level.INFO, "Add the subquery's
              why-provenance to the " +
180               "original query's why-provenance");
181           // add the why-provenance to the query's why-provenance
182           queryWhyProvenance.addAll(ProvenanceUtils
183             .transformWhyProvInDNF(unnormalizedWhyProvenance));
184         }
185       }
186
187       ChateauLogger.log(Level.INFO, "The original query's
          why-provenance has been calculated");
188       return queryWhyProvenance;
189     }
190
191     /**
192      * Get the relationName-setOfRelations-map for given
          relationNames and the pa-chased query. For each relation name
          in relationNames
193      * add the Relational Atom with the same relation name.
194      * @param relationNames the relational atom's names of the
          original query
195      * @param query the provenance-aware chased query
196      * @return the relationalName-setOfRelations-map
197      * @see RelationalAtom
198      */
199     private Map<RelationalAtom, Set<RelationalAtom>>
        getKeyRelationMap(Map<String, Set<RelationalAtom>>
        relationNames, Set<RelationalAtom> query) {
```

```
200     Map<RelationalAtom, Set<RelationalAtom>> keyRelationMap = new
        HashMap<>();
201
202     ChateauLogger.log(Level.INFO, "Save for each atom of the
        original query all " +
203         "atoms with the same relation name");
204
205     // for each relation name of relationNames add the relation
        atom and a new HashSet to the map
206     relationNames.values().forEach(keys -> keys.forEach(
207         key -> keyRelationMap.put(key, new HashSet<>())));
208
209     // for each relational atom add the atom where the atom's
        relation name is equal the key
210     query.forEach(relation -> {
211       Set<RelationalAtom> atomContainer =
        relationNames.get(relation.getRelationName());
212       if (atomContainer != null)
213         atomContainer.forEach(atom ->
214           keyRelationMap.get(atom).add(relation));
215         }
216
217     );
218     return keyRelationMap;
219   }
220
221   @Override
222   public List<Instance> getQueries() {
223     List<Instance> queries = new ArrayList<>();
224
225     // add universal plan to queries
226     queries.add(this.getUniversalPlan());
227
228     return queries;
229   }
230
231   /**
232    * Get the relational atoms with schema tag SOURCE of an
       instance
233    * @param chasedQuery the chased query
234    * @return set of relational atoms with schema tag SOURCE
235    * @see Instance
236    * @see RelationalAtom
237    * @see SchemaTag#SOURCE
238    */
239   protected Set<RelationalAtom>
      getRelationalAtomsWithoutViews(Instance chasedQuery) {
240     Set<RelationalAtom> relationalAtomsWithoutViews = new
        HashSet<>();
241     //for each relational atom ...
242     chasedQuery.getRelationalAtoms().forEach(relationalAtom -> {
```

```
243         //... check if it has provenance information
244         if
            (chasedQuery.getSchemaTags().get(relationalAtom.getName())
            == SchemaTag.SOURCE)
245           relationalAtomsWithoutViews.add(relationalAtom);
246       });
247       return relationalAtomsWithoutViews;
248     }
249
250     /**
251      * Prints the provenance of the given atoms.
252      * @param atoms a hash set of atoms
253      */
254     public void printProvenanceOfAtoms(Set<RelationalAtom> atoms) {
255       atoms.forEach(atom -> {
256         System.out.println("atom: " + atom);
257         System.out.println("provenance: " +
            atom.getProvInfo().getWhyProvenance() + "\n---\n");
258       });
259     }
260
261 }
```

Quellcodeausschnitt B.4.2 ProvenanceUtils.java

```
 1 public class ProvenanceUtils {
 2   /**
 3    * Transforms the given why provenance in disjunctive normal
      form.
 4    * @param whyProvenance the formula to be transformed
 5    * @return the why-provenance in disjunctive normal form
 6    */
 7   public static Set<Set<String>>
     transformWhyProvInDNF(Set<Set<Set<String>>> whyProvenance) {
 8     Set<Set<String>> result = new HashSet<>();
 9     if (whyProvenance.isEmpty())
10       return result;
11     var r = com.google.common.collect.Sets.cartesianProduct(new
       ArrayList<>(whyProvenance));
12     r.forEach(whyProv -> {
13       Set<String> prov = new HashSet<>();
14       for (Set<String> provAtom : whyProv)
15         prov.addAll(new HashSet<String>(provAtom));
16       result.add(prov);
17     });
18     return result;
19   }
20 }
```

Die Chase-Klasse musste im Zuge des provenance-bewussten chasens erweitert werden. Hierbei wurden Zeile 20 - 30 und Zeile 40 - 55 hinzugefügt. Die `calculateProvenance`-Methode konnte die nicht normalisierte Why-Provenance berechnen. Das Berechnen der normalisierten Why-Provenance wurde hinzuge-

fügt. Zeile 100 - 194 dienen zur Verschmelzung der Atome. Die schon implementierte Verschmelzung der Atome war diesbezüglich nicht ausreichend.

Quellcodeausschnitt B.4.3 Chase.java

```
1  public class Chase {
2
3    // fields accessed by the chase method - must be initialized
     beforehand or
4    // passed as parameters
5    private Instance instance;
6    private LinkedHashSet<IntegrityConstraint> integrityConstraints;
7    ...
8
9    // cache the new instance during each run of the chaseMainLoop
     - initialized
10   // with the empty instance
11   private Instance newInstance;
12
13   // cache the entire new instance during a run of the
     chaseMainLoop - initialized
14   // with the whole instance
15   private Instance instanceWithNewFacts;
16   ...
17   // variable to calculate provenance informations
18   public boolean calculateProvenance;
19
20   // variable to activate provenance-aware chase
21   private boolean aquv;
22
23   /**
24    * Activate the provenance-aware chase
25    * @param aquv a boolean to activate the provenance-aware chase
26    */
27   public void setAQUV(boolean aquv) {
28     this.aquv = aquv;
29     this.calculateProvenance = aquv;
30   }
31   ...
32
33   /**
34    * process one of the triggers for an integrity constraint
35    *
36    * @param constraint
37    * @param homomorphism
38    * @return
39    */
40   private boolean processTrigger(IntegrityConstraint constraint,
     Homomorphism homomorphism) {
41     var success = true;
42
```

```
43      // if provenance-aware chase is configured and the constraint
        is a Tgd ...
44      if (this.aquv && constraint instanceof Tgd) {
45        ChateauLogger.log(Level.INFO, "Check if " + homomorphism +
          " is a provenance-aware " +
46            "active trigger");
47        // check if the homomorphism is a provenance-aware active
          trigger for instanceWithNewFacts and the constraint
48        if (homomorphism.isProvenanceAwareActiveTriggerFor(
49            instanceWithNewFacts, (Tgd) constraint)) {
50          ChateauLogger.log(Level.INFO, "Identified
            provenance-aware active Trigger " +
51              homomorphism);
52          // apply the tgd
53          var tgd = (Tgd) constraint;
54          applyTgd(tgd, homomorphism);
55        } ...
56      }
57      return success;
58    }
59    ...
60
61    /**
62     * for each mapping, get the provenance id of the atom for
       which the mapping was
63     * created and save it as why provenance. if the atom is a
       duplicate, then add
64     * the why provenance to the existing why provenance of the
       existing duplicate.
65     *
66     * @param tempHomomorphism
67     * @param relationalHeadAtom
68     * @param newAtom
69     */
70    private void calculateProvenance(Homomorphism tempHomomorphism,
      RelationalAtom relationalHeadAtom,
71        RelationalAtom newAtom) {
72
73      var provenanceId =
        newAtom.generateProvenanceId(newAtom.getRelationName());
74      ...
75      LinkedHashSet<String> whereTupleProvenance = new
        LinkedHashSet<String>();
76      LinkedHashSet<String> whereRelationsProvenance = new
        LinkedHashSet<String>();
77      ...
78
79      var whyProvenance = new ArrayList<ArrayList<String>>();
80      // ensures that no id's are added to the why provenance for
        identical
81      // combinations of mapping source and target
```

```
82      var howProvenanceHelper = new
        LinkedHashSet<Pair<RelationalAtom, RelationalAtom>>();
83      tempHomomorphism.getTermMappings().stream().
84      // null values do not have an origin atom
85          filter(mapping -> mapping.getTargetTermRelationPosition()
        != null)
86          .forEach(mapping -> howProvenanceHelper
87              .add(new Pair<RelationalAtom, RelationalAtom> (mapping
88                .getSourceTermRelationPosition().getKey(),
89                mapping.getTargetTermRelationPosition()
90                  .getKey())));
91
92      Set<Set<Set<String>>> unnormalizedWhyProvenance = new
        HashSet<>();
93      // for each pair of relational atoms add the why-provenance
        of the mapped atom
94      howProvenanceHelper.stream().forEach(pair -> {
95        Set<Set<String>> provenance = new HashSet<>();
96        pair.getValue().getProvInfo().getWhyProvenance().forEach(
97          provenanceAtom -> provenance.add(
98            new HashSet<>(provenanceAtom)));
99        unnormalizedWhyProvenance.add(provenance);
100     });
101     Set<Set<String>> normalizedWhyProvenance =
        ProvenanceUtils.transformWhyProvInDNF(
102       unnormalizedWhyProvenance);
103     normalizedWhyProvenance.forEach(provenanceAtom -> {
104       whyProvenance.add(new ArrayList(provenanceAtom));
105     });
106
107     if (newInstance.getRelationalAtoms().contains(newAtom)) {
108       ...
109     } else {
110       AtomicReference<Boolean> contained = new
          AtomicReference<>(false);
111       ChateauLogger.log(Level.INFO, "Check if new Atom is
          homomorphically equivalent " +
112           "to an atom of newInstance");
113       // check if new atom has a homomorphism to an newInstance's
          atom
114       newInstance.getRelationalAtoms().stream().filter(atom ->
          newAtom.hasHomomorphismTo(atom) &&
          atom.hasHomomorphismTo(newAtom)).forEach(atom -> {
115         // check if the atoms are equal
116         for (RelationalAtom relationalAtom :
            newInstance.getRelationalAtoms()) {
117           // if the atoms has the same reference then continue
118           if (relationalAtom == atom)
119             continue;
120           // check if a variable is a join variable
121           if (atom.getTerms().stream().anyMatch(term -> {
```

```
122             if (term.isVariable() && ((Variable)
                term).getVariableType().equals(VariableType.FOR_ALL))
123               return false;
124             return relationalAtom.getTerms().contains(term);
125           })) {
126             // there exists a join variable thus can't unify atoms
127             // continue with the next atom
128             return;
129           }
130         }
131
132         ChateauLogger.log(Level.INFO, "new Atom " + newAtom + "
            is homomorphically equivalent " +
133             "to atom " + atom + ". Unify atoms and add the new
                atom's why-provenance to atom.");
134         // if the atoms can be unified than
135         // add the why-provenance to the atom
136         whyProvenance.forEach(whyProvAtom -> {
137           if (!atom.getProvInfo().getWhyProvenance()
138             .contains(whyProvAtom))
139             atom.getProvInfo().getWhyProvenance()
140               .add(whyProvAtom);
141         });
142         ChateauLogger.log(Level.INFO, atom + " now has
            why-provenance "
143             + atom.getProvInfo().getWhyProvenance());
144         // atom is contained
145         contained.set(true);
146
147       });
148
149       ChateauLogger.log(Level.INFO, "Check if new Atom is
          homomorphically equivalent " +
150           "to an atom of instanceWithNewFacts");
151       // check if new atom has a homomorphism to an instance's
          atom
152       instanceWithNewFacts.getRelationalAtoms().stream().filter(
153           atom -> newAtom.hasHomomorphismTo(atom) &&
              atom.hasHomomorphismTo(newAtom)).forEach(atom -> {
154         boolean joinVar = false;
155         // check if the atoms are the same
156         for (RelationalAtom relationalAtom :
            instanceWithNewFacts.getRelationalAtoms()) {
157           // if atoms has the same reference then continue
158           if (relationalAtom == atom)
159             continue;
160
161           // check if a variable is a join variable
162           if (atom.getTerms().stream().anyMatch(term -> {
163             if (term.isVariable() && ((Variable)
                term).getVariableType().equals(VariableType.FOR_ALL))
```

```
164                 return false;
165               return relationalAtom.getTerms().contains(term);
166             })) {
167               // there exists a join variable thus can't unify atoms
168               // continue with the next atom
169               return;
170             }
171           }
172           ChateauLogger.log(Level.INFO, "new Atom " + newAtom + "
              is homomorphically equivalent " +
173               "to atom " + atom + ". Unify atoms and add the new
                  atom's why-provenance to atom.");
174
175           // if the atoms can be unified than
176           // add the why-provenance to the atom
177           whyProvenance.forEach(whyProvAtom -> {
178             if (!atom.getProvInfo().getWhyProvenance()
179               .contains(whyProvAtom))
180               atom.getProvInfo().getWhyProvenance()
181                 .add(whyProvAtom);
182           });
183           ChateauLogger.log(Level.INFO, atom + " now has
              why-provenance "
184               + atom.getProvInfo().getWhyProvenance());
185           contained.set(true);
186         });
187
188         // if newAtom was contained than
189         // there is nothing more to do
190         if (contained.get()) {
191           ChateauLogger.log(Level.INFO, newAtom + " is already
              contained. Generate why-provenance " +
192               "of next atom");
193           return;
194         }
195
196         ChateauLogger.log(Level.INFO, "add " + newAtom + " to
            newInstance.");
197         // set provenance info and
198         // add newAtom to newInstance
199         var provInfoWhyProv = new ArrayList<ArrayList<String>>();
200         provInfoWhyProv.addAll(whyProvenance);
201         var provInfo = new ProvenanceInformation(provenanceId,
            whereTupleProvenance, whereRelationsProvenance,
202             provInfoWhyProv);
203         newAtom.setProvInfo(provInfo);
204         newInstance.addRelationalAtom(newAtom);
205       }
206     }
207     ...
208 }
```

Die `getMappedAtom`-Methode ist eine Kopie der `isContained`-Methode der `Homomorphism`-Klasse. Durch diese Methode wird, wenn ein Kopfatom auf ein Atom der Anfrage abgebildet werden kann, das abgebildete Atom selbst zurückgegeben.

Quellcodeausschnitt B.4.4 Homomorphism.java

```
1  /**
2   * The class represents a homomorphism and consists of a set of
3   * {@link homomorphism.TermMapping} objects.
4   *
5   * @author Martin Jurklies
6   */
7  public class Homomorphism {
8    private LinkedHashSet<TermMapping> mappings;
9    ...
10   /**
11    * Check if there exists a provenance-aware active trigger-
12    * @param instance the instance
13    * @param integrityConstraint the tgd
14    * @return {@code true} if there exists a pa active trigger,
      otherwise {@code false}
15    * @see Instance
16    * @see Tgd
17    */
18   public boolean isProvenanceAwareActiveTriggerFor(Instance
      instance, Tgd integrityConstraint) {
19     boolean isActiveTrigger = false;
20
21     Set<Set<Set<String>>> bodyAtomsWhyProvenance = new
       HashSet<>();
22     Set<Set<Set<String>>> headAtomsWhyProvenance = new
       HashSet<>();
23
24     ChateauLogger.log(Level.INFO, "Generate the body's
       why-provenance");
25     // get the why-provenance of the tgd's body
26     this.getTermMappings().forEach(termMapping -> {
27           Set<Set<String>> prov = new HashSet<>();
28           termMapping.getTargetTermRelationPosition().getKey()
29             .getProvInfo().getWhyProvenance().forEach(
30               whyProv -> prov.add(new HashSet(whyProv))
31           );
32           bodyAtomsWhyProvenance.add(prov);
33         }
34     );
35
36     ChateauLogger.log(Level.INFO, "Generate the head's
       why-provenance.");
37     // get the why-provenance of the tgd's head
38     for (RelationalAtom constraintAtom :
       integrityConstraint.getHead()) {
```

```
39        RelationalAtom mappedHeadAtom = getMappedAtom(instance,
          this.applyMappingsTo(constraintAtom));
40        // if current head atom isn't implied in the instance than
41        // the trigger is active
42        if (mappedHeadAtom == null) {
43          isActiveTrigger = true;
44          break;
45        }
46
47        Set<Set<String>> prov = new HashSet<>();
48        mappedHeadAtom.getProvInfo().getWhyProvenance().forEach(
49          whyProv -> prov.add(new HashSet(whyProv))
50        );
51        headAtomsWhyProvenance.add(prov);
52      }
53
54      // if trigger is active return true
55      if (isActiveTrigger)
56        return true;
57
58      // transform the body's and head's why-provenance in
          disjunctive normal form
59      Set<Set<String>> bodyWhyProvenance =
          ProvenanceUtils.transformWhyProvInDNF(bodyAtomsWhyProvenance);
60      Set<Set<String>> headWhyProvenance =
          ProvenanceUtils.transformWhyProvInDNF(headAtomsWhyProvenance);
61
62      ProvenanceInformation bodyProvInfo = new
          ProvenanceInformation(null);
63      bodyProvInfo.setWhyProvenance(new
          ArrayList(bodyWhyProvenance));
64
65      ChateauLogger.log(Level.INFO, "Check if body's why-provenance
          implies the heads's why-provenance \n" +
66              bodyWhyProvenance + "\n" +
67              "->" +
68              headWhyProvenance);
69      isActiveTrigger = !bodyProvInfo.whyProvenanceImplies(new
          ArrayList(headWhyProvenance));
70
71      if (isActiveTrigger)
72        ChateauLogger.log(Level.INFO, "Why-provenance is not
          implied thus the trigger is active" );
73      else
74        ChateauLogger.log(Level.INFO, "Why-provenance is implied
          thus the trigger is inactive" );
75
76      return isActiveTrigger;
77    }
78    ...
79    /**
```

```
80     * @return the mapped relational atom
81     * @see #isContained
82     * @see RelationalAtom
83     */
84    private RelationalAtom getMappedAtom(Instance instance,
      RelationalAtom headAtom) {
85      boolean isContained = false;
86      RelationalAtom mappedAtom;
87
88      for (RelationalAtom instanceAtom :
        instance.getRelationalAtomsBySchema(headAtom.getName())) {
89        for (int i = headAtom.getTerms().size() - 1; i >= 0; i--) {
90          isContained = true;
91          var headAtomTerm = headAtom.getTerms().get(i);
92          if (!instanceAtom.getTerms().get(i).equals(headAtomTerm))
            {
93            // move on if head is a variable
94            if (headAtomTerm.isVariable()) {
95              var headAtomVariable = (Variable) headAtomTerm;
96              var variableType = headAtomVariable.getVariableType();
97              //
98              if (variableType == VariableType.EXISTS) {
99                // variable is existence quantified and can be
                  mapped to any instance term
100               continue;
101             } else if (variableType == VariableType.FOR_ALL) {
102               if (instanceAtom.getTerms().get(i).isConstant()) {
103                 // variable is all-quantified and can be mapped
                    to constants
104                 continue;
105               }
106             }
107           }
108           isContained = false;
109           break;
110         }
111       }
112       if (isContained) {
113         return instanceAtom;
114       }
115     }
116
117     return null;
118   }
119   ...
120 }
```

Quellcodeausschnitt B.4.5 ProvenanceInformation.java

```
1  public class ProvenanceInformation {
2    ...
3    private ArrayList<ArrayList<String>> whyProvenance;
```

```
4     ...
5     /**
6      * Check if this.whyProvenance implies whyProvenance
7      * @param whyProvenance the implied why-provenance
8      * @return {@code true} if this.whyProvenance implies
       whyProvenance, otherwise {@code false}
9      */
10    public boolean whyProvenanceImplies(
11      ArrayList<ArrayList<String>> whyProvenance) {
12      boolean isImplied = true;
13      // for each provenance in this.whyProvenance ...
14      for (Collection provenanceBody : this.getWhyProvenance()) {
15        boolean contained = false;
16        // ... and for each provenance in whyProvenance ...
17        for (Collection provenanceHead : whyProvenance) {
18          // ... check if provenanceBody contains at least one
            provenance atom in this.whyProvenance
19          if (provenanceBody.containsAll(provenanceHead)) {
20            contained = true;
21            break;
22          }
23        }
24        // if the provenance is not contained than return false
25        if (!contained) {
26          isImplied = false;
27          break;
28        } else {
29          // otherwise check the next provenance atom
30          isImplied = true;
31        }
32      }
33      return isImplied;
34    }
35  }
```

B.5 Experiment

Quellcodeausschnitt B.5.1 GeneratorMain.java

```
1   public class GeneratorMain {
2
3     public GeneratorMain() { }
4
5     public Set<RelationalAtom> generateAtoms(int num, int vars) {
6       Set<RelationalAtom> result = new LinkedHashSet<>();
7
8       Term joinVar = null;
9       int currentTermIndex = 0;
10      for (int i = 1; i <= num; i++) {
```

```
11        ArrayList<Term> atomTerms =generateTerms(vars,
          currentTermIndex,joinVar);
12        RelationalAtom atom = new RelationalAtom(
13            "R_"+i,
14            atomTerms
15        );
16        joinVar = atomTerms.get(vars-1);
17        currentTermIndex += vars;
18
19        result.add(atom);
20      }
21
22      return result;
23    }
24
25    public ArrayList<Term> generateTerms(int num, int lastIndex,
      Term joinVar) {
26      ArrayList<Term> result = new ArrayList<>();
27
28      if (joinVar != null) {
29        result.add(joinVar);
30        num--;
31      }
32
33      for (int i = 1; i<=num; i++) {
34        Variable variable = new Variable(VariableType.EXISTS,"VAR",
          i + lastIndex);
35        result.add(variable);
36      }
37
38      return result;
39    }
40
41    public HashMap generateSchemaTags(Set<RelationalAtom> atoms,
      Set<RelationalAtom> views) {
42      HashMap<String, SchemaTag> result = new HashMap<>();
43
44      atoms.forEach(atom -> result.put(atom.getRelationName(),
        SchemaTag.SOURCE));
45      views.forEach(atom -> result.put(atom.getRelationName(),
        SchemaTag.TARGET));
46
47      return result;
48    }
49
50    public Instance generateQuery(Set<RelationalAtom>
      relationalAtoms, HashMap schema,HashMap schemaTags) {
51      Instance result = new Instance(
52          (LinkedHashSet<RelationalAtom>) relationalAtoms,
53          schema,
54          OriginTag.QUERY,
```

```
55          schemaTags
56      );
57      return result;
58
59    }
60
61    public HashMap generateSchema(Set<RelationalAtom> atoms,
      Set<RelationalAtom> views) {
62      HashMap<String, LinkedHashMap<String, String>> schema = new
        HashMap();
63
64      for (RelationalAtom atom : atoms) {
65        LinkedHashMap<String, String> schem = new LinkedHashMap();
66        atom.getTerms().forEach(term -> schem.put(term.toString(),
          "string"));
67        schema.put(atom.getRelationName(), schem);
68      }
69
70      for (RelationalAtom view : views) {
71        LinkedHashMap<String, String> schem = new LinkedHashMap();
72        view.getTerms().forEach(term -> schem.put(term.toString(),
          "string"));
73        schema.put(view.getRelationName(), schem);
74      }
75
76      return schema;
77    }
78
79    public Set<RelationalAtom> generateViews(Set<RelationalAtom>
      atoms) {
80      Set<RelationalAtom> views = new LinkedHashSet<>();
81
82      for (RelationalAtom atom : atoms) {
83        RelationalAtom view = new RelationalAtom(
84            "V_" + atom.getRelationName(), new
              ArrayList(atom.getTerms())
85        );
86        views.add(view);
87      }
88
89      return views;
90    }
91
92    public LinkedHashSet<IntegrityConstraint>
      generateConstraints(Set<RelationalAtom> atoms) {
93      LinkedHashSet<IntegrityConstraint> result = new
        LinkedHashSet<>();
94      for (RelationalAtom atom : atoms) {
95        ArrayList<Term> terms = new ArrayList();
96        for (Term term : atom.getTerms())
97          terms.add(term.invert());
```

```
98          RelationalAtom body_atom = new RelationalAtom(
99              atom.getRelationName(),
100             terms
101         );
102
103         RelationalAtom view = new RelationalAtom(
104             "V_" + body_atom.getRelationName(), new
                ArrayList(body_atom.getTerms())
105         );
106         Set<RelationalAtom> body = new LinkedHashSet<>();
107         Set<RelationalAtom> head = new LinkedHashSet<>();
108
109         body.add(body_atom);
110         head.add(view);
111
112         IntegrityConstraint sttgd = new STTgd(
113             (LinkedHashSet<RelationalAtom>) body,
114             (LinkedHashSet<RelationalAtom>) head
115         );
116
117         result.add(sttgd);
118       }
119       return result;
120     }
121
122     private static long generateResult(Constructor BackchaseClass,
        Instance instance, LinkedHashSet<IntegrityConstraint>
        integrityConstraints, Instance chased)
123         throws InvocationTargetException, InstantiationException,
            IllegalAccessException {
124       AbstractBackchase ab = (AbstractBackchase)
          BackchaseClass.newInstance
125           (chased, integrityConstraints, instance);
126
127       long start = System.nanoTime();
128       ab.backchase();
129       long end = System.nanoTime();
130
131
132       return end - start;
133     }
134
135     private static void writeToCSV(String filename, ArrayList<Long>
        time) throws IOException {
136       FileWriter fw = new FileWriter("../timestamps/" + filename +
          ".csv");
137       fw.append("Time");
138       fw.append("\n");
139
140       for (long t : time) {
141         fw.append("" + t);
```

```
142         fw.append("\n");
143       }
144       fw.flush();
145       fw.close();
146     }
147
148
149
150     public static void main(String[] args) throws
        NoSuchMethodException, InvocationTargetException,
        InstantiationException, IllegalAccessException, IOException,
        InterruptedException {
151       test1();
152       test2();
153     }
154
155     private static void test2() throws IOException,
        NoSuchMethodException, InvocationTargetException,
        InstantiationException, IllegalAccessException {
156       final int[] atomNumbers = {2,3,4,5,6,7,8,9,10,20,50,100,200};
157       final int[] termNumbers = {5,10};
158       final int repetition = 100;
159
160       GeneratorMain gm = new GeneratorMain();
161
162       //warm up of the chase
163       for (int i = 0; i < 20; i++) {
164         Set<RelationalAtom> atoms =gm.generateAtoms(5,5);
165         Set<RelationalAtom> views = gm.generateViews(atoms);
166         Instance instance =
          gm.generateQuery(atoms,gm.generateSchema(atoms, views),
          gm.generateSchemaTags(atoms, views));
167         LinkedHashSet<IntegrityConstraint> integrityConstraints =
          gm.generateConstraints(atoms);
168         var d = Chase.chase(instance, integrityConstraints);
169         new Backchase(d, integrityConstraints, instance
          ).backchase();
170         new BackchasePD(d, integrityConstraints,
          instance).backchase();
171       }
172
173       for (int termNumber : termNumbers) {
174         for (int atomNumber : atomNumbers) {
175           LogManager.getLogManager().reset();
176           ArrayList<Long> backchasePDTime = new ArrayList();
177
178           for (int i = 0; i < repetition; i++) {
179             LogManager.getLogManager().reset();
180             Set<RelationalAtom> atoms =
              gm.generateAtoms(atomNumber,termNumber);
181             Set<RelationalAtom> views = gm.generateViews(atoms);
```

```
182             LinkedHashSet<IntegrityConstraint> integrityConstraints
                = gm.generateConstraints(atoms);
183             Instance instance =
                gm.generateQuery(atoms,gm.generateSchema(atoms, views),
                gm.generateSchemaTags(atoms, views));
184             Instance chased = Chase.chase(instance,
                integrityConstraints);
185
186             backchasePDTime.add(generateResult(
187                 BackchasePD.class.getConstructor(Instance.class,
                    LinkedHashSet.class,Instance.class),
188                 new Instance(new
                    LinkedHashSet(instance.getRelationalAtoms()), new
                    HashMap(instance.getSchema()),
                    instance.getOriginTag(), new
                    HashMap(instance.getSchemaTags())),
189                 new LinkedHashSet(integrityConstraints),
190                 new Instance(new
                    LinkedHashSet(chased.getRelationalAtoms()), new
                    HashMap(chased.getSchema()), chased.getOriginTag(),
                    new HashMap(chased.getSchemaTags()))
191             ));
192
193           }
194           writeToCSV("backchasePD_" + atomNumber + "_" +
              termNumber, backchasePDTime);
195         }
196       }
197     }
198     private static void test1() throws IOException,
      NoSuchMethodException, InvocationTargetException,
      InstantiationException, IllegalAccessException {
199       GeneratorMain gm = new GeneratorMain();
200
201       //warm up of the chase
202       for (int i = 0; i < 20; i++) {
203         Set<RelationalAtom> atoms =gm.generateAtoms(5,5);
204         Set<RelationalAtom> views = gm.generateViews(atoms);
205         Instance instance =
          gm.generateQuery(atoms,gm.generateSchema(atoms, views),
          gm.generateSchemaTags(atoms, views));
206         LinkedHashSet<IntegrityConstraint> integrityConstraints =
          gm.generateConstraints(atoms);
207         var d = Chase.chase(instance, integrityConstraints);
208         new Backchase(d, integrityConstraints, instance
          ).backchase();
209         new BackchasePD(d, integrityConstraints,
          instance).backchase();
210       }
211
212       final int[] atomNumbers = {2,3,4,5,6,7,8,9,10};
```

```
213       final int[] termNumbers = {5,10};
214       final int repetition = 100;
215
216       for (int termNumber : termNumbers) {
217         for (int atomNumber : atomNumbers) {
218           ArrayList<Long> backchaseTime = new ArrayList();
219           for (int i = 0; i < repetition; i++) {
220             LogManager.getLogManager().reset();
221             Set<RelationalAtom> atoms =
              gm.generateAtoms(atomNumber,termNumber);
222             Set<RelationalAtom> views = gm.generateViews(atoms);
223             Instance instance =
              gm.generateQuery(atoms,gm.generateSchema(atoms, views),
              gm.generateSchemaTags(atoms, views));
224             LinkedHashSet<IntegrityConstraint> integrityConstraints
              = gm.generateConstraints(atoms);
225
226             Instance chased = Chase.chase(instance,
              integrityConstraints);
227
228             backchaseTime.add(generateResult(
229                 Backchase.class.getConstructor(Instance.class,
                  LinkedHashSet.class,Instance.class),
230                 new Instance(instance),
231                 new LinkedHashSet(integrityConstraints),
232                 chased
233             ));
234           }
235           writeToCSV("backchase_" + atomNumber + "_" + termNumber,
              backchaseTime);
236         }
237       }
238     }
239   }
```

Anhang C

Datenträger

Aufgrund der digitalen Abgabe dieser Arbeit wird der Programmcode und die verwendete Literatur online zur Verfügung gestellt.

Arbeit

Jegliche Dateien zur Arbeit werden auf StudIP im `Dateien`-Bereich zur Verfügung gestellt.
Veranstaltungsname *Forschungskolloquium: Bachelorarbeit Nico Trebbin: Nutzung von ChaTEAU für das Answering-Queries-using-Views-Problem (AQuV)*
Semester Wintersemester 2021/22

- TEX-Dateien der Arbeit: `Arbeit/tex`
- Literatur: `Arbeit/Literatur`
- Abbildungen: `Arbeit/tex/chapters/6.zusatzlicheDateien/etc`

Die Dateinamen der Literatur-Ordner entsprechen dem Kürzel im Literaturverzeichnis. Jeder Ordner enthält die PDF und die zugehörige `.bib`-Datei. **Die PDF-Dateien der Literatur dienen der Nachvollziehbarkeit. Sie dürfen nicht öffentlich bereitgestellt werden.**

Quellcode

Der Quellcode der implementierten Prototypen befindet sich im Git des Instituts der Informatik.
Branch *Bachelorarbeit-Nico-Trebbin*
Link `https://git.informatik.uni-rostock.de/ta093/chateau.git`

Selbständigkeitserklärung

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

Rostock, den 2022-02-10