Der CHASE-Algorithmus: Ein Überblick

Gebietsseminar Informationssysteme

Nic Scharlau

06. April 2021

---

1 Einleitung

In dieser Seminararbeit beschäftigen wir uns mit dem *CHASE-Algorithmus*. Dieser wurde ursprünglich von *Maier*, *Mendelzon* und *Sagiv* in den 1980er Jahren entwickelt und wird in der Datenbanktheorie seitdem unter anderem für die Untersuchung von Äquivalenzen, der Darstellung funktionaler Abhängigkeiten in Datenbanken sowie für das Optimieren von Anfragen verwendet. Allgemein gesagt ist es das Ziel des CHASE, eine Menge von Parametern $\star$ in ein Objekt $\bigcirc$ einzuarbeiten, sodass diese am Ende implizit im Objekt enthalten sind:

$$\mathrm{CHASE}_{\star}(\bigcirc) = \circledast.$$

Das Objekt kann dabei bspw. eine Datenbankinstanz oder eine Anfrage, die Parameter bspw. eine Menge von Integritätsbedingungen oder ebenfalls eine Anfrage sein. Zunächst werden wir in diesem Abschnitt ein Beispiel einführen, welches uns durch diese Arbeit begleiten wird (Unterabschnitt 1.1). Anschließend beschäftigen wir uns in Abschnitt 2 mit *Tableaus*, welche die Grundlage des CHASE sind. Da diese in der Praxis jedoch recht unhandlich sind, beschränken wir sie auf diesen Abschnitt. Für das Verstehen des CHASE sind sie jedoch unabdingbar. In Abschnitt 3 beschäftigen wir uns dann mit dem CHASE auf Datenbankinstanzen. Dabei lernen wir *eingebettete Abhängigkeiten*, konkreter *egds*, *tgds* sowie *s-t tgds*, kennen und lernen mit *Data Cleaning* und *Data Exchange* auch zwei Anwendungsbeispiele kennen. In Abschnitt 4 schauen wir uns dann die Anwendung des CHASE auf Anfragen an und gehen insbesondere in Unterabschnitt 4.1 auf das *Answering-Queries-using-Views-Problem* ein, bevor wir uns in Abschnitt 5 den verschiedenen CHASE-Varianten sowie in Abschnitt 6 der Terminierung des CHASE-Algorithmus widmen.

1.1 Laufendes Beispiel

Als Beispiel dient in dieser Arbeit ein fiktiver, stark vereinfachter Datensatz einer Universität, der aus drei Relationen besteht:

- Studierende = {SID, Name, Vorname, Studiengang, Fakultät} mit
  - $\mathbb{D}(\text{SID}) = \mathbb{N}$
  - $\mathbb{D}(\text{Name}) = \{\text{Fieber, Jansen, Müller, Sonnenschein}\}$
  - $\mathbb{D}(\text{Vorname}) = \{\text{Fabian, Jana, Max, Sarah}\}$
  - $\mathbb{D}(\text{Studiengang}) = \{\text{Informatik, Mathematik}\}$
  - $\mathbb{D}(\text{Fakultät}) = \{\text{IEF, MNF, } \perp\}$ mit $\perp$ als Nullwert
- Prüfungen = {PID, Bezeichnung, ECTS} mit
  - $\mathbb{D}(\text{PID}) = \mathbb{N}$
  - $\mathbb{D}(\text{Name}) = \{\text{Mathe I, Mathe II, DB1, Data Science}\}$
  - $\mathbb{D}(\text{ECTS}) = \mathbb{N}$
- Noten = $\{\text{SID}_{\text{Studierende}}, \text{PID}_{\text{Prüfungen}}, \text{Note}\}$ mit
  - $\mathbb{D}(\text{SID}) = \mathbb{N}$
  - $\mathbb{D}(\text{PID}) = \mathbb{N}$
  - $\mathbb{D}(\text{Note}) = \{1.0, 1.3, 1.7, 2.0, 2.3, 2.7, 3.0, 3.3, 3.7, 4.0, 5.0\}$

SID	Name	Vorname	Studiengang	Fakultät
1	Fieber	Fabian	Informatik	IEF
2	Sonnenschein	Sarah	Informatik	IEF
3	Müller	Max	Mathematik	MNF
4	Jansen	Jana	Informatik	$\perp$

Tabelle 1: Studierende (S)

PID	Bezeichnung	ECTS
1	Mathe I	9
2	Mathe II	9
3	DB1	6
4	Data Science	6

Tabelle 2: Prüfungen (P)

SID	PID	Note
1	1	1.7
1	3	1.0
2	1	1.0
2	2	2.3
3	4	1.3
4	3	3.0
5	5	2.7

Tabelle 3: Noten (N)

2 Der CHASE auf Tableaus

Um die verschiedenen Anwendungsbereiche des CHASE verstehen zu können, müssen wir uns zunächst einige theoretische Grundlagen aneignen. Wir beginnen diese Seminararbeit deshalb mit dem CHASE auf Tableaus. Dazu klären wir zunächst, was Tableaus eigentlich sind und wozu sie notwendig sind. Außerdem lernen wir mit der F- und der J-Regel zwei wichtige Regeln kennen, welche die Grundlage für alle weiteren Anwendungsbereiche des CHASE bilden.

2.1 Tableaus

Tableaus sind eine tabellarische Darstellung von Attributen und Schemata einer Datenbank. Alle Attribute stehen dabei in einer Spalte, alle Schemata in einer Zeile. Den genaueren Aufbau sowie die Konstruktion eines Tableaus werden wir im weiteren Verlauf dieses Abschnitts kennenlernen.

Definition 2.1 (Tableau, nach [Mai83])**.** *Sei V eine Menge von Variablen, $R := \{A_1, \ldots, A_m\}$ eine Menge von Attributen, dann heißt T Tableau über R genau dann, wenn:*

1. *T ist endliche Menge von Abbildungen von R nach V*
2. *$V = V_a \cup V_n$ mit $V_a = \{a_1, a_2, \ldots\}, V_n = \{b_1, b_2, \ldots\}, V_a \cap V_n = \emptyset$ und*
   V_a: Menge der ausgezeichneten Variablen,
   V_n: Menge der nichtausgezeichneten Variablen.
3. *Sei $i \neq j$. Dann gilt: $\forall v \in \pi_{A_i}(T) \forall v' \in \pi_{A_j}(T) : v \neq v'$.*
4. *Sei $V_{a,A_i} := \pi_{A_i}(T) \cap V_a$. Dann gilt $|V_{a,A_i}| \leq 1$. O.B.d.A. soll dieses Element dann a_i sein.*

Die Tupel eines Tableaus heißen Zeilen. □

Das Aufstellen eines Tableaus ist dabei recht intuitiv. Stellen wir uns eine Datenbank vor, die aus den Attributen R= $\{A_1, A_2, A_3, A_4\}$ und einem Schema $S = \{\{A_1, A_2, A_3\}, \{A_1, A_4\}, \{A_2, A_3\}\}$ besteht. Wollen wir diese Datenbank als Tableau T_S darstellen, reihen wir zunächst alle Relationen nebeneinander auf:

A_1	A_2	A_3	A_4

Im nächsten Schritt erhält jedes $s \in S$ seine eigene Zeile. Attribute, die im jeweiligen Schema vorhanden sind, werden dabei durch eine Variable a_i (übereinstimmend mit dem jeweiligen A_i) gekennzeichnet. Diese Variablen werden als *ausgezeichnete Variablen* bezeichnet; die Menge aller ausgezeichneten Variablen ist V_a. Für unser Schema S sieht das Tableau T_S anschließend wie folgt aus:

A_1	A_2	A_3	A_4
a_1	a_2	a_3	
a_1			a_4
	a_2	a_3	

In einem letzten Schritt werden die nun vorhandenen Leerstellen („blanks“) mit sogenannten *nichtausgezeichneten Variablen* aufgefüllt, welche durch ein b_i gekennzeichnet werden ($i \geq 1$). Die Menge aller nichtausgezeichneten Variablen wird mit V_b gekennzeichnet. Üblicherweise beginnt man dabei oben links und arbeitet sich aufsteigend nach unten rechts durch. Unser fertiges Tableau T_S sieht dann wie folgt aus:

A_1	A_2	A_3	A_4
a_1	a_2	a_3	b_1
a_1	b_2	b_3	a_4
b_4	a_2	a_3	b_5

T_S ist hierbei ein *spezielles* Tableau, weil es dem Datenbankschema S genügt. In dieses spezielle Tableau können wir nun Abhängigkeiten einarbeiten, um die Konsistenz einer Datenbank zu sichern oder gar inkonsistente/fehlerhafte Datensätze zu reparieren, wie wir später noch detailliert sehen werden. Dadurch erhalten wir ein *allgemeines* Tableau, in welchem pro Spalte auch mehrere nichtausgezeichnete Variablen auftauchen dürfen. Bevor wir nun jedoch den CHASE anwenden können, benötigen wir zunächst noch zwei weitere Definitionen: die der F- und J-Regeln.

2.2 F-Regel und J-Regel

Definition 2.2 (F-Regel, nach [Mai83])**.** *Eine F-Regel ist eine Vorschrift, die aus einem Tableau T ein neues Tableau T' erzeugt: Sei $X \to A$ eine funktionale Abhängigkeit. Seien ferner w_1 und w_2 zwei Zeilen des Tableaus, also $w_1, w_2 \in T$, wobei $w_1(X) = w_2(X)$. Angenommen, es gelten $w_1(A) = v_1$ und $w_2(A) = v_2$ mit $v_1 \neq v_2$, so können wir folgende Regel anwenden:*

1. *Falls $v_1 \in V_a \wedge v_2 \in V_b$, so wird v_2 (und alle weiteren Vorkommen der Variablen) durch v_1 ersetzt.*
2. *Falls $v_1, v_2 \in V_b$, so ersetzt die Variable mit dem kleineren Index diejenige mit dem größeren Index (und alle weiteren Vorkommen).*
3. *Falls $v_1, v_2 \in V_a$, so geschieht nichts.*

□

Grundlage der F-Regel sind die sogenannten funktionalen Abhängigkeiten (functional dependencies; FDs), welche als bekannt vorausgesetzt werden. Angenommen, wir haben eine FD $A_3 \rightarrow A_4$ („A_3 bestimmt A_4“). Wollen wir diese in unser Tableau T_S einarbeiten, so suchen wir in der Spalte A_3 zunächst nach gleichen a_3 bzw. b_i und prüfen anschließend die Attribute der Spalte A_4. Im Beispiel vergleichen wir so die Variablen b_1 und b_5. Da beide Variablen nichtausgezeichnet sind, „gewinnt“ diejenige mit dem kleineren Index, also b_1. Folglich wird b_5 durch b_1 ersetzt:

A_1	A_2	A_3	A_4
a_1	a_2	a_3	b_1
a_1	b_2	b_3	a_4
b_4	a_2	a_3	b_1

Neben den funktionalen Abhängigkeiten existieren außerdem Verbundabhängigkeiten (join dependencies; JDs). Auch diese werden an dieser Stelle als bereits bekannt vorausgesetzt. Auf Grundlage der JDs existiert eine zweite Regel, die sogenannte J-Regel:

Definition 2.3 (J-Regel, nach [Mai83])**.** *Seien $S = \{S_1, S_2, \ldots, S_n\}$ eine Menge von Relationenschemata, $\bowtie[S]$ eine JD über dem Universum U, T ein Tableau und $w_1, w_2, \ldots, w_k \in T$ die Reihen dieses Tableaus, welche über S verbunden werden können und ein Ergebnis w liefern. Die Anwendung der J-Regel auf T liefert uns dann das neue Tableau $T' = T \cup \{w\}$.* □

Gehen wir beispielsweise davon aus, dass wir eine JD $\bowtie [A_1A_2, A_1A_3A_4]$ haben. A_1A_2 und $A_1A_3A_4$ werden nun also über das gemeinsame Element A_1 miteinander verbunden, wodurch wir zwei neue Tupel (a_1, a_2, b_3, a_4) und (a_1, b_2, a_3, b_1) erhalten, welche wir in unser Tableau übernehmen:

A_1	A_2	A_3	A_4
a_1	a_2	a_3	b_1
a_1	b_2	b_3	a_4
b_4	a_2	a_3	b_1
a_1	a_2	b_3	a_4
a_1	b_2	a_3	b_1

Um den *CHASE* auf Tableaus anzuwenden, werden so lange alle F- und J-Regeln angewandt, bis ein finales Tableau entsteht, auf dem die Regeln keine Auswirkungen mehr haben. Für FDs und JDs ist der CHASE dabei sowohl terminierend als auch konfluent (er liefert immer dasselbe Ergebnis, egal in welcher Reihenfolge die FDs und JDs eingearbeitet werden) und somit vollständig. Wir werden später sehen, dass diese Eigenschaften jedoch nicht für den allgemeinen CHASE gelten.

Definition 2.4 (CHASE auf Tableaus, nach [Mai83])**.** *Seien T als Tableau und C als Menge von Abhängigkeiten gegeben. Eine generierende Sequenz ist eine Sequenz $T_1, T_2, \ldots, T_n$ von Tableaus, wobei $T = T_0$ und $T_i \neq T_{i+1}$ gilt und T_{i+1} aus T durch Anwendung von F- und J-Regeln in C erzeugt wird (mit $i \geq 0$). Wenn diese Sequenz ein letztes Element T_n besitzt, welches*

schlussendlich weder durch F-Regeln noch durch J-Regeln weiter verändert werden kann, so ist T_n *eine* CHASE-Lösung *von* T*. Die Menge aller CHASE-Lösungen von* T *wird durch* $CHASE_C(T)$ *beschrieben.* □

2.3 Tableauanfragen

Damit kennen wir nun sowohl Tableaus als auch die Anwendung des CHASE auf solche. Wir werden nun abschließend anhand eines Beispiels noch zwei Erweiterungen der Tableaus kennenlernen, welche Anfragen auf diese ermöglichen: die Tags und die Summary. Sei T unser gegebenes, reduziertes[1] Tableau nach dem „Einchasen“ der FD und JD:

A_1	A_2	A_3	A_4
a_1	a_2	a_3	b_1
a_1	a_2	b_3	a_4

Zunächst lernen wir die *Tags* kennen, welche – vereinfacht gesagt – die Relation beschreiben, für die eine Tableauzeile steht. Bislang gingen wir der Einfachheit halber von der sogenannten *Universal Instance Assumption* aus, der Annahme, dass eine universelle Relation existiert, welche alle Attribute des Schemas umfasst. Von dieser Annahme lösen wir uns jetzt, weshalb es notwendig wird, den Tableauzeilen eine Relation zuzuordnen. Zudem werden alle nichtausgezeichneten Variablen aus dem Tableau entfernt. Gehen wir davon aus, dass die erste Zeile eine Relation r_1 und die zweite Zeile eine Relation r_2 darstellt. Dann sieht die Erweiterung des Tableaus um Tags wie folgt aus:

A_1	A_2	A_3	A_4	
a_1	a_2	a_3		r_1
a_1	a_2		a_4	r_2

Die *Summary* ist eine Zeile, die selbst kein Bestandteil des Tableaus ist, aber dazu verwendet wird, für Anfragen benötigte Variablen zu kennzeichnen. Zudem ist es notwendig, neben den ausgezeichneten und nichtausgezeichneten Variablen zwei weitere Arten von Symbolen einzuführen: *Konstanten* und *Blanks* (Leerstellen). Gehen wir davon aus, dass wir die Anfrage

$$\pi_{a_4}(\sigma_{a_3=5}(r_1) \bowtie_{a_2} r_2)$$

an das Tableau stellen wollen. Auf jeden Fall benötigen wir die Variablen, auf die projiziert wird, in diesem Fall a_4. Außerdem benötigen wir alle Variablen, die für Verbunde erforderlich sind, also auch a_2. Allerdings dürfen nur Variablen ausgezeichnet sein, die für die Projektion benötigt werden, weshalb a_2 durch eine nichtausgezeichnete Variable (in diesem Fall b_2) ersetzt wird. Diese tragen wir in einer neuen Zeile oberhalb der Schemata, die sogenannte Summary, ein:

[1] Die Reduktion eines Tableaus ist ein Verfahren, um die Anzahl der Zeilen zu minimieren. Zeilen, die mittels eines Homomorphismus auf eine Zeile abgebildet werden können, werden durch diese subsumiert. Auf eine ausführliche Erläuterung wird an dieser Stelle jedoch verzichtet. Als Literatur sei auf [Mai83] verwiesen.

A_1	A_2	A_3	A_4	
	b_2		a_4	
a_1	b_2	a_3		r_1
a_1	b_2		a_4	r_2

Anstelle von a_1 und a_3 befinden sich in der Summary *Blanks*, also Leerstellen, da diese beiden Variablen nicht für unsere Anfrage benötigt werden. Was nun noch fehlt, ist die Selektion nach $a_3 = 5$. Dazu ersetzen wir einfach die Variable innerhalb der entsprechenden Zeile(n) durch den konkreten Wert:

A_1	A_2	A_3	A_4	
	b_2		a_4	
a_1	b_2	5		r_1
a_1	b_2		a_4	r_2

Damit haben wir unsere Anfrage mithilfe des erweiterten Tableaus dargestellt. Im weiteren Verlauf der Arbeit werden Tableaus jedoch keine relevante Rolle mehr einnehmen, da sie viel Platz beanspruchen. Heutzutage verwenden wir eine deutlich kompaktere und abstraktere Schreibweise. Nichtsdestotrotz sind sie nach wie vor geeignet, um – auch wegen ihrer Visualität – ein grundlegendes Verständnis des CHASE zu erhalten. In Abschnitt 4 werden wir den CHASE auf Anfragen kennenlernen, welcher genau auf ebendiesen Anfragen basiert. Zuvor beschäftigen wir uns jedoch mit dem CHASE auf Instanzen.

3 Der CHASE auf Instanzen

Bis jetzt haben wir die Tableaus und die Anwendung des CHASE auf ebendiese kennengelernt. Tableaus sind die Grundlage des CHASE, heutzutage verwenden wir jedoch eine andere Schreibweise für den CHASE auf Instanzen, den wir im folgenden Abschnitt kennenlernen werden. Außerdem lernen wir insgesamt drei Arten von Abhängigkeiten kennen, die eine Erweiterung der FDs und JDs darstellen: egds, tgds und s-t tgds.

3.1 Equality-generating- und tuple-generating dependencies

Bevor wir uns den egds und tgds widmen können, brauchen wir zunächst die Definition der eingebetteten Abhängigkeit. Egds, tgds und s-t tgds sind spezielle Varianten von eingebetteten Abhängigkeiten.

Definition 3.1 (Eingebettete Abhängigkeit, nach [GMS12])**.** *Eine* eingebettete Abhängigkeit *ist ein prädikatenlogischer Ausdruck erster Stufe mit der Form* $\forall x \forall y : \varphi(x, y) \rightarrow \exists z : \psi(x, z)$*, wobei* x*,* y *und* z *Tupel aus Variablen (Variablenvektoren) und* $\varphi(x, y)$ *sowie* $\psi(x, z)$ *Konjunktionen über atomare Formeln sind.* $\varphi(x, y)$ *wird hierbei* Rumpf *und* $\psi(x, z)$ Kopf *der Abhängigkeit genannt.* □

Definition 3.2 (equality-generating dependency (egd), nach [GMS12])**.** *Eine* gleichheitserzeugende Abhängigkeit (egd) *ist eine eingebettete Abhängigkeit der Form* $\forall x : F_1(x) \rightarrow x_1 = x_2$*. Sie besitzt keine existenzquantifizierten Variablen und ihr Kopf besteht nur aus einem einzigen Gleichheitsatom.* □

Betrachten wir beispielsweise die Relation `Studierende` (Tabelle 1). Für die Studentin mit der `SID 4` ist der Wert der Fakultät ein Nullwert ($\perp$). Wenn wir jedoch voraussetzen, dass der Studiengang die Fakultät funktional bestimmt, können wir diesen inkonsistenten Datensatz reparieren. Die egd „Studiengang bestimmt Fakultät“ sieht dann wie folgt aus:

$$S(sid_1, na_1, vo_1, st, fak_1) \wedge S(sid_2, na_2, vo_2, st, fak_2) \rightarrow fak_1 = fak_2.$$

Auf die explizite Angabe des Allquantors, in diesem Fall

$$\forall sid_1, sid_2, na_1, na_2, vo_1, vo_2, fak_1, fak_2, st,$$

wird in der Regel verzichtet, um die Abhängigkeiten übersichtlicher zu gestalten. Außerdem kürzen wir die Namen der Relationen und Attribute ab. Wenden wir diese egd nun auf unsere Instanz an, so wird der Nullwert durch den Wert „IEF“ ersetzt, da aus dem Datensatz `Informatik` $\rightarrow$ `IEF` hervorgeht.

Definition 3.3 (tuple-generating dependency (tgd), nach [GMS12])**.** *Eine tupelerzeugende Abhängigkeit (tgd) ist eine eingebettete Abhängigkeit der Form* $\forall x : F_1(x) \rightarrow \exists y : F_2(x, y)$*.* $F_1(x)$ *ist dabei eine Konjunktion von atomaren Formeln mit Variablen aus* x*, während* $F_2(x, y)$ *eine Konjunktion von atomaren Formeln mit Variablen aus* x *und* y *ist. Sie besitzt keine Gleichheitsatome. Besitzt die tgd im Kopf existenzquantifizierte Variablen, so ist sie* eingebettet*, andernfalls ist sie* voll*.* □

Um ein Beispiel für eine tgd zu erhalten, betrachten wir die Relation `Noten` (Tabelle 3). Dort fällt auf, dass das Tupel $(5, 5, 2.7)$ existiert, obwohl es weder in der `Studierende`- noch in der `Prüfungen`-Relation Tupel mit der entsprechenden ID 5 gibt. Da SID und PID als Fremdschlüssel definiert sind, liegt hier eine Verletzung einer Integritätsbedingung vor. Eine tgd für diese Bedingung könnte beispielsweise wie folgt aussehen (auch hier ohne Allquantor):

$$N(sid, pid, no) \rightarrow \exists Na, Vo, St, Fa, Be, Ec : S(sid, Na, Vo, St, Fa) \wedge P(pid, Be, Ec).$$

Vereinfacht gesagt drückt diese tgd aus, dass die Existenz eines Tupels in der `Noten`-Relation die Existenz von Tupeln in den `Studierende`- und `Prüfungen`-Relationen impliziert, wobei die SIDs bzw. PIDs jeweils übereinstimmen. Wird, wie hier, der Existenzquantor weggelassen, werden existenzquantifizierte Variablen üblicherweise großgeschrieben, um sie dennoch schnell erkennen zu können. Wenden wir diese tgd an, so erhalten wir in der `Studierende`-Relation das neue Tupel $(5, \eta_1, \eta_2, \eta_3, \eta_4)$ sowie in der `Prüfungen`-Relation das Tupel $(5, \eta_5, \eta_6)$, wobei η_i nummerierte Nullwerte sind:

SID	Name	Vorname	Studiengang	Fakultät
1	Fieber	Fabian	Informatik	IEF
...	...	...	...	...
4	Jansen	Jana	Informatik	IEF
5	η_1	η_2	η_3	η_4

PID	Bezeichnung	ECTS
1	Mathe I	9
...	...	...
4	Data Science	6
5	η_5	η_6

Definition 3.4 (source-to-target tgd (s-t tgd), nach [GMS12])**.** *Eine source-to-target tgd ist ein Spezialfall der tgd, bei dem F_1 eine Konjunktion über ein Quellschema S (source) und F_2 eine Konjunktion über ein Zielschema T (target) ist.* □

s-t tgds dienen also dazu, ein Quellschema S in ein Zielschema T zu überführen. Auch hier betrachten wir als Beispiel unsere drei Relationen aus Abschnitt 1.1. Angenommen, die gegebenen Relationen wären unser Quellschema, welches wir in ein Zielschema bestehend aus der (einzigen) Relation `Leistungen` = {Vorname, Name, Note} überführen wollen. Eine dazu passende s-t tgd könnte wie folgt aussehen:

$$N(sid, pid, no) \wedge S(sid, na, vo, st, fa) \rightarrow Leistungen(vo, na, no).$$

Wenden wir diese s-t tgd an, so erhalten wir eine neue Relation `Leistungen` mit folgenden Tupeln:

Vorname	Name	Note
Fabian	Fieber	1.7
Fabian	Fieber	1.0
Sarah	Sonnenschein	1.0
Sarah	Sonnenschein	2.3
Max	Müller	1.3
Jana	Jansen	3.0
η_2	η_1	2.7

Um nun die Anwendung des CHASE auf Instanzen definieren zu können, brauchen wir zuvor noch einige grundlegendere Definitionen.

Definition 3.5 (Homomorphismus für Instanzen, nach [FKPT11])**.** *Seien I_1 und I_2 zwei Instanzen über einem Schema S, Const die Menge aller Konstanten und Var die Menge aller Variablen der Instanz. Eine Funktion $h : \mathsf{Const} \cup \mathsf{Var} \rightarrow \mathsf{Const} \cup \mathsf{Var}$ ist ein Homomorphismus von I_1 nach I_2 genau dann, wenn:*

- $\forall c \in \mathsf{Const} : h(c) = c$,
- $\forall R \in S, \forall (a_1, \ldots, a_n) \in R^{I_1} : (h(a_1), \ldots, h(a_n)) \in R^{I_2}$
  (R^I: Instanz I über der Relation R).

Ein Homomorphismus von I_1 nach I_2 wird als $I_1 \rightarrow I_2$ notiert. Existiert zusätzlich ein Homomorphismus $I_2 \rightarrow I_1$, also h^{-1}, so gelten I_1 und I_2 als äquivalent: $I_1 \leftrightarrow I_2$. □

Algorithmus 1: Der Standard-CHASE auf Instanzen, nach [BKM+17]

```
1  ΔI := I
2  while ΔI ≠ ∅ do
3      N := ∅,   μ := ∅
4      foreach τ ∈ Σ mit Rumpf λ(x⃗) do
5          foreach Trigger h für τ in I sodass h(λ(x⃗)) ∩ ΔI ≠ ∅ do
6              if h ist aktiver Trigger für τ in μ(N ∪ I) then
7                  if τ ist tgd then
8                      h' := h ∪ {y⃗ → v⃗}, v⃗ ⊆ Nulls, Werte in v⃗ sind neu
9                      N := N ∪ h'(ρ(x⃗, y⃗))
10                 else if τ ist egd then
11                     if {h(x_i) ≠ h(x_j)} ⊆ Const then
12                         I := ⊥
13                     else
14                         ω := {max(h(x_i), h(x_j)) → min(h(x_i), h(x_j))}
15                         μ := μ ∘ (ω ∘ μ)
16     ΔI := μ(N ∪ I) \ I
17     I := μ(I) ∪ ΔI
```

Wir haben einen solchen Homomorphismus bereits angewandt, als wir die egd aus dem Beispiel anwendeten. Der Nullwert $\perp$ wurde dabei auf die Konstante *„IEF“* abgebildet, während alle anderen Werte der `Studierende`-Relation auf sich selbst abgebildet wurden. Es stellt sich nun jedoch die Frage, wann solche Homomorphismen überhaupt zur Anwendung kommen. Dazu benötigen wir zwei weitere Begriffe, den des *Triggers* sowie den des *aktiven Triggers*.

Definition 3.6 ((Aktiver) Trigger, nach [Jur18])**.** *Sei I eine Instanz und $b \in \mathcal{B}$ eine tgd oder egd. Ein Trigger für b in I ist ein Homomorphismus $h : \varphi(x) \to I$ mit $\varphi(x)$ als Rumpf von b (der Rumpf von b wird auf die Tupel aus den Relationen in I abgebildet).*
Ein aktiver Trigger für b in I ist ein Trigger h mit:

- *Falls b eine tgd ist, existiert keine Erweiterung von h zu einem Homomorphismus von $\psi(x, y) \to I$ (wobei $\psi(x, y)$ eine Konjunktion von relationalen Atomen ist).*
- *Falls b eine egd ist, dann gilt $h(x_i) \neq h(x_j)$ für mindestens ein Gleichheitsatom ($x_i = x_j$) aus dem Kopf der egd.*

□

Ziel des CHASE auf Instanzen ist es, solange Homomorphismen anzuwenden, bis es keine aktiven Trigger mehr gibt. Dazu wird für diese definierte Abhängigkeit geprüft, ob der Rumpf dieser Abhängigkeit mittels eines Homomorphismus auf ein Atom der Instanz abgebildet werden kann. Ist dies der Fall, existiert ein Trigger; erfüllt die Instanz noch nicht den Kopf der Abhängigkeit, ist dieser Trigger aktiv und die Abhängigkeit wird in die Instanz eingearbeitet. Formal ist der CHASE auf Instanzen definiert wie in Algorithmus 1 zu sehen.

3.2 Data Cleaning

Ein Anwendungsfall des CHASE auf Instanzen ist das *Data Cleaning*, dem Ermitteln und Reparieren von unvollständigen, korrupten oder inkonsistenten Datensätzen. Data Cleaning wird relevant, wenn ein gegebener Datensatz fehlerhafte Informationen enthält. Als wir die egds und tgds einführten, kam Data Cleaning bereits zum Einsatz: ein unvollständiger, inkonsistenter Datensatz wurde mittels zweier eingebetteter Abhängigkeiten „repariert“. Allerdings kann die Technik auch verwendet werden, um mehrere Datensätze zu „säubern“, beispielsweise, wenn sie widersprüchliche Informationen beinhalten. In Tabelle 4 sehen wir drei Tools, die entwickelt wurden, um Data Cleaning zu betreiben: *Llunatic*, *ChaseFun* und *ChaseTEQ*. Insbesondere die Funktionsweise von *Llunatic*, aber auch der Prozess des Data Cleaning selbst, kann in der Dissertation von *Donatello Santoro* nachvollzogen werden [San14].

3.3 Data Exchange

Ein weiterer Anwendungsfall des CHASE auf Instanzen ist *Data Exchange*. Hierbei sollen die Daten eines Quellschemas (*source*) in die Instanz eines anderen Zielschemas (*target*) überführt werden. Formal ist das Data-Exchange-Problem wie folgt definiert:

Definition 3.7 (Data Exchange, nach [FKMP05])**.** *Ein Data-Exchange-Setting ist ein Tupel* $(S, T, \Sigma_{st}, \Sigma_t)$ *mit* S *als Quellschema,* T *als Zielschema* ($S \cap T = \emptyset$)*,* Σ_{st} *als Menge von s-t tgds, die den Zusammenhang zwischen Quelle und Ziel beschreiben, und* Σ_t *als Menge von Abhängigkeiten über dem Zielschema* T*. Sei* I *außerdem eine Instanz über* S*. Das Data-Exchange-Problem besteht darin, eine neue Instanz* J *über* T *so zu materialisieren, dass* J *die Abhängigkeiten in* Σ_t *erfüllt und die s-t tgds in* Σ_{st} *sowohl von* I *als auch von* J *erfüllt werden. Eine solche Instanz* J *ist eine Lösung des Problems.* □

Auch diesen Anwendungsfall lernten wir bereits kennen, als wir unser Beispiels für s-t tgds erläuterten. Das Quellschema ist unsere Beispielinstanz, das Zielschema besteht aus der (einzigen) neuen Relation `Leistungen` und der Zusammenhang zwischen beiden wird mittels einer s-t tgd beschrieben. Für Data Exchange gibt es eine Reihe von Anwendungen, die versuchen, dieses Problem zu lösen. Zwei davon, die sich ebenfalls mit Data Cleaning auseinandersetzen, sind in Tabelle 4 aufgelistet. Genaueres kann bei Bedarf in [FKMP05] nachvollzogen werden.

4 Der CHASE auf Anfragen

Bisher haben wir den CHASE auf Tableaus sowie den CHASE auf Instanzen kennengelernt. Der CHASE kann jedoch auch auf Anfragen angewandt werden. Zunächst schauen wir uns dazu an, wie SQL- oder Datalog-Anfragen in s-t tgds umformuliert werden können. Betrachten wir dazu Anfrage 1 auf unser Studierenden-Beispiel.

[2] *ProSA*: Provenance-Management durch Schema-Abbildungen und Annotationen. `https://dbis.informatik.uni-rostock.de/forschung/aktuelle-projekte/prosa/`, zuletzt abgerufen 05.04.2021, 14:25 Uhr

	Parameter ⋆	Objekt ◯	Ergebnis ⊛	Ziel	Tools
0.	Abhängigkeiten	DB-Schema	DB-Schema mit IBs	optimierter DB-Entwurf	
I.	Abhängigkeiten	Anfragen	Anfragen	semantische Optimierung	PDQ, Graal, Pegasus
II.	Sichten	Anfragen	Anfragen auf Sichten	AQuV	$\text{Prov}_{\text{C\&B}}$
III.	s-t tgds, egds	Quell-DB	Ziel-DB	Data Exchange, Data Integration	Llunatic, ChaseFun
IV.	tgds, egds	DB	modifizierte DB	Data Cleaning	Llunatic, ChaseFun, ChaseTEQ
V.	s-t tgds, egds	DB	Anfrageergebnis	gesicherte Antworten	
VI.	s-t tgds, egds, tgds	DB	Anfrageergebnis	invertierbare Auswertung	ProSA^2

Tabelle 4: Anwendungsfälle des CHASE und entsprechende Tools

```
SELECT Name
FROM Studierende S JOIN Noten N ON (S.SID = N.SID)
WHERE Note = 3.0
```

Anfrage 1: Wer (Nachname) erhielt eine 3.0?

Wir verbinden die Relationen `Studierende` und `Noten` über das gemeinsame Attribut *SID*, selektieren nach $Note = 3.0$ und projizieren dann auf das Attribut `Name`. Der (optimierte) relationenalgebraische Ausdruck dazu lautet

$$\pi_{Name}(S \bowtie (\sigma_{Note=3.0}(N))),$$

die dazugehörige Datalog-Notation ist

```
R(name) := S(sid, name, vorname, studiengang, fakultät), N(sid, pid, 3.0),
```

wobei `R` („result“) ein neues Prädikat ist, welches unser Ergebnis enthält. Auf eine ausführliche Erklärung der Datalog-Notation wird an dieser Stelle verzichtet. Das Beispiel ist aber für die Einführung von Anfragen geeignet, da sich der Aufbau dem der s-t tgds ähnelt. Da wir einen Verbund durchführen, müssen wir sicherstellen, dass das gemeinsame Attribut `SID` in beiden Relationen übereinstimmt. Damit haben wir bereits den natürlichen Verbund in die s-t tgd integriert. Um zu projizieren, brauchen wir ein neues Ergebnisprädikat, welches unsere Ergebnisrelation beschreibt. Da wir auf das Attribut `Name` projizieren, muss unser Ergebnisprädikat auch (nur) genau dieses Attribut umfassen. Außerdem müssen wir natürlich sicherstellen, dass der `Name` in der `Studierende`-Relation identisch mit dem in unserem Ergebnisprädikat ist, weshalb dieses Attribut ebenfalls existenzquantifiziert wird. Wir nennen es *Result* und definieren es entsprechend als `R(Name)`. Um nach bestimmten Werten zu selektieren, ersetzen wir die entsprechenden Variablen in den Quellrelationen einfach durch die konkrete Werte, nach denen selektiert werden soll. Die neue s-t tgd, die unsere SQL-Anfrage abbildet, sieht somit wie folgt aus:

$$S(sid, name, vo, st, fa) \wedge N(sid, pid, 3.0) \rightarrow R(name).$$

4.1 Answering Queries using Views

Als besonderer Anwendungsfall des CHASE auf Anfragen steht *Answering Queries using Views (AQuV)*, zu Deutsch „Anfragen mittels Sichten beantworten“, hervor. Beim AQuV-Problem sind eine Anfrage auf eine Datenbank sowie eine Menge von Sichten gegeben. Je nach Auffassung besteht das Ziel darin, bei der Beantwortung dieser Anfrage entweder ausschließlich Sichten oder aber möglichst viele dieser Sichten zu verwenden. Wir beschränken uns in dieser Arbeit auf den ersten Fall.

Die Lösung des AQuV-Problems besteht nicht nur aus einer CHASE-, sondern zusätzlich aus einer BACKCHASE-Phase, in welcher das vom CHASE erzeugte Ergebnis erneut mit anderen Parametern „gechaset“ wird. Daher wird der Algorithmus fortan als *C&B-Algorithmus* bezeichnet. Basierend auf der Arbeit von *Ioana Ileana* haben *Alin Deutsch* und *Richard Hull* 2013 im Paper *„Provenance-directed Chase & Backchase“* ein praktisches Beispiel präsentiert, welches wir uns im Folgenden gemeinsam anschauen werden [Ile14] [DH13]:

Sei $q : R(x, w, y) \wedge S(y, z) \wedge T(z, u) \rightarrow Q(x)$ eine gegebene Anfrage. Seien außerdem

$$\begin{aligned} R(x, w, y) &\rightarrow V_R(x, y) \\ S(y, z) &\rightarrow V_S(y, z) \\ R(x, w, y) \wedge S(y, z) &\rightarrow V_{RS}(x, z) \\ T(z, u) &\rightarrow V_T(z, u) \end{aligned}$$

vier gegebene Sichtdefinitionen (jeweils als (s-t) tgds). In der CHASE-Phase wenden wir diese tgds auf die Anfrage q an und erhalten folgende Prädikate (der Einfachheit halber als Menge notiert):

$$\{R(x, w, y), S(y, z), T(z, u), V_R(x, y), V_S(y, z), V_{RS}(x, z), V_T(z, u)\}\,.$$

Die Anfrage wurde also aufgebläht. Im nächsten Schritt werden alle Basisrelationen aus dieser Anfrage entfernt, sodass nur noch die Sichten übrigbleiben:

$$U = \{V_R(x, y), V_S(y, z), V_{RS}(x, z), V_T(z, u)\}\,.$$

Diese Menge U wird als *Universalplan* bezeichnet. Wieder formuliert als Anfrage sieht dieser nun wie folgt aus:

$$V_R(x, y) \wedge V_S(y, z) \wedge V_{RS}(x, z) \wedge V_T(z, u) \rightarrow U(x).$$

Hätten wir neben den Sichten noch eine Menge von Integritätsbedingungen gegeben, so würden diese an dieser Stelle ebenfalls eingearbeitet werden. Der Einfachheit halber verzichten wir aber darauf. An dieser Stelle ist die CHASE-Phase abgeschlossen und die BACKCHASE-Phase beginnt. Dazu müssen wir zunächst die Sichten invertieren, wobei Variablen, die nicht im Kopf der

Sichten auftauchen, im neuen Rumpf existenzquantifiziert werden müssen:

$$\begin{aligned} V_R(x,y) &\to \exists W : R(x,W,y) \\ V_S(y,z) &\to S(y,z) \\ V_{RS}(x,z) &\to \exists W,Y : R(x,W,Y) \wedge S(Y,z) \\ V_T(z,u) &\to T(z,u) \end{aligned}$$

Nun gilt es zu ermitteln, welche (minimalen) Teilmengen der Sichten genügen, um ein *query containment rewriting* (fortan *Rewriting*) von q zu erfüllen. Dazu gibt es verschiedene Möglichkeiten. Klassischerweise kann die Potenzmenge von $\mathcal{V}$ (die Menge aller Sichten) abzüglich der leeren Menge gebildet und alle Möglichkeiten mittels des CHASE durchprobiert werden, also $\mathcal{P}(\mathcal{V})\backslash\{\emptyset\}$. In unserem Beispiel erfüllen sowohl die Sichtmengen $\{V_R, V_S, V_T\}$ als auch $\{V_{RS}, V_T\}$ ein solches Rewriting, da sie genau die Prädikate $R(x,w,y)$, $S(y,z)$ sowie $T(z,u)$ erzeugen:

$$\begin{aligned} V_R(x,y) \wedge V_S(y,z) \wedge V_T(z,u) &\to R_1(x), \\ V_{RS}(x,z) \wedge V_T(z,u) &\to R_2(x). \end{aligned}$$

Die anderen 13 Kombinationen scheitern oder sind – im Gegensatz zu R_1 und R_2 – nicht minimal. Ein großes Problem dieser Methode ist ihre Komplexität. Es ist einfach zu erkennen, dass die Laufzeitkomplexität $\mathcal{O}(2^{|\mathcal{V}|}-1) \approx \mathcal{O}(2^n)$ beträgt und somit exponentiell wächst. Jedoch existieren Methoden, um den $C\&B$-Algorithmus effizienter zu gestalten. Zum einen kann *Pruning* angewendet werden. Das bedeutet, dass Obermengen einer gefundenen, gültigen Teilmenge nicht mehr berücksichtigt werden (wenn $V \subseteq \mathcal{V}$ eine Lösung ist, dann sind auch alle Obermengen von V Lösungen, jedoch keine minimalen). Zum anderen entwickelte *Ileana* eine deutlich effizientere Methode unter Ausnutzung der *why-Provenance*, welche *Deutsch* und *Hull* ebenfalls präsentierten. Der Kerngedanke dabei ist, dass sich für jedes erzeugte Prädikat in der BACKCHASE-Phase dessen Ursprung, also die invertierte Sichtdefinition, gemerkt wird. Anschließend lässt sich die Quelle der erzeugten Prädikate direkt auf die jeweilige Sicht zurückführen, ohne dass exponentiell viele Möglichkeiten ausprobiert werden müssen. Die genaue Funktionsweise des *provenance-directed C&B* kann bei Bedarf im Paper [DH13] nachvollzogen werden.

AQuV wird beispielsweise in Data Warehouses eingesetzt, um nach einer Informationsintegration möglichst nur noch auf materialisierten Sichten zu arbeiten. Ein weiterer Anwendungsfall ist der Datenschutz. Hier wird das AQuV-Problem relevant, wenn aus Datenschutzgründen der Zugriff aus Basisrelationen nicht gestattet ist. Wird eine Anfrage auf eine Datenbank gestellt, muss ein Rewriting gefunden werden, welches die Anfrage nur mithilfe definierter Sichten beantwortet. Sollte dies nicht möglich sein, muss die Anfrage entsprechend scheitern.

Bis hierher haben wir den CHASE auf Tableaus, Instanzen und Anfragen kennengelernt. Wann immer vom „CHASE“ die Rede war, war jedoch der *Standard-CHASE* gemeint. Darüber hinaus existieren noch weitere Varianten, die wir im folgenden Abschnitt betrachten werden.

5 CHASE-Varianten

In den vorherigen Abschnitten beschäftigten wir uns ausführlich mit dem CHASE-Algorithmus. Genau genommen handelte es sich dabei jedoch stets um eine bestimmte Variante des CHASE, den Standard CHASE. Daneben existieren jedoch noch einige weitere Varianten, die wir in diesem Abschnitt behandeln wollen. Alle CHASE-Varianten haben gemeinsam, dass durch das Umbenennen von Nullwerten sowie dem Erzeugen neuer Tupel eine Datenbankinstanz „repariert“ werden soll. Die einzelnen Schritte, die dabei durchgeführt werden, werden als *CHASE-Schritte* bezeichnet; die Art und Weise, wie diese CHASE-Schritte durchgeführt werden, bestimmt die Art des CHASE [GMS12].

Standard CHASE Der *Standard CHASE* ist der CHASE, wie wir ihn in dieser Arbeit kennengelernt und angewandt haben. Er ist der Nachfolger des Oblivious CHASE und wendet Homomorphismen genau dann an, wenn es (mindestens) einen aktiven Trigger gibt. Das bedeutet, dass es nach Definition 3.5 eine Abbildung vom Rumpf einer Abhängigkeit auf eine Instanz gibt und diese Abhängigkeit noch nicht erfüllt ist.

Oblivious CHASE Der *Oblivious CHASE* unterscheidet sich vor allem dadurch vom Standard CHASE, dass die Überprüfung auf aktive Trigger nicht erfolgt. Stattdessen wird ein Homomorphismus *immer* dann angewendet, wenn gemäß Definition 3.5 eine Abbildung vom Rumpf einer Abhängigkeit zu einer Instanz existiert, auch dann, wenn die Abhängigkeit bereits erfüllt ist. Der Oblivious CHASE existiert zudem in zwei Varianten: Zum einen gibt es den Naive Oblivious CHASE. Dieser erzeugt immer dann Gleichheiten oder neue Tupel, sobald ein Trigger existiert. Zum anderen gibt es den Skolem Oblivious CHASE, welcher eine Skolemfunktion verwendet, um neu erzeugte Nullwerte zu markieren. Auf diese Weise soll verhindert werden, dass bereits erfüllte tgds erneut angewandt werden.
Betrachten wir erneut die tgd aus Abschnitt 3:

$$N(sid, pid, no) \rightarrow \exists Na, Vo, St, Fa, Be, Ec : S(sid, Na, Vo, St, Fa) \wedge P(pid, Be, Ec).$$

Der Standard CHASE erzeugt je ein neues Tupel in `Studierende` sowie `Prüfungen`. Der Naive Oblivious CHASE hingegen würde endlos viele neue Tupel erzeugen, da die Prüfung auf die Aktivität eines Triggers entfällt. Der Skolem Oblivious CHASE würde sich die Nullwerte $\eta_1, \ldots, \eta_6$ hingegen „markieren“ und ab der 2. Ausführung berücksichtigen, weshalb in diesem Beispiel keine neuen Tupel entstehen.

Core CHASE Der *Core CHASE* ist eine parallele Version des CHASE. Wann immer mehrere (aktive) Trigger existieren, wählt der CHASE nichtdeterministisch einen der möglichen CHASE-Schritte aus. *Deutsch et al.* haben dabei in [DNR08] gezeigt, dass der Standard-CHASE kein vollständiger Algorithmus ist, um universelle Lösungen zu suchen, da er je nach der Reihenfolge angewandter Abhängigkeiten zu anderen Resultaten führen kann. Weiterhin ist es möglich, dass

der Standard-CHASE je nach Anwendung der egds und tgds nicht terminiert. Deshalb stellten sie 2008 den Core CHASE vor, welcher aus zwei Schritten besteht: Zunächst werden die CHASE-Schritte parallel durchgeführt. Anschließend wird aus den jeweiligen Ergebnissen ein Kern berechnet, welcher alle Abhängigkeiten erfüllen soll. Der Core CHASE ist damit vollständig hinsichtlich des Findens universeller Lösungen.
Um den Core CHASE zu verstehen, betrachten wir ein Beispiel aus [GMS12]: Seien $D = \{\}$ eine leere Datenbankinstanz und

$$\begin{aligned} r_0: &\quad \{\} &&\rightarrow \quad E(u,v) \wedge E(v,u), \\ r_1: &\quad E(x,y) \wedge E(y,x) &&\rightarrow \quad E(u,u), \\ r_2: &\quad E(x,y) &&\rightarrow \quad E(x,u) \wedge E(u,y) \end{aligned}$$

eine Menge von Abhängigkeiten Σ. Zunächst wenden wir wie gewohnt r_0 an und erhalten

$$D_1' = \{E(\eta_1, \eta_2), E(\eta_2, \eta_1)\}$$

als neue Instanz. Nun berechnen wir den Kern $core(D_1')$ der Instanz. Dabei prüfen wir, ob eine Teilmenge der Prädikate existiert, welche die bisherigen Abhängigkeiten erfüllt. Dies ist jedoch noch nicht der Fall, weshalb der Kern ebenfalls

$$D_1 = core(D_1') = \{E(\eta_1, \eta_2), E(\eta_2, \eta_1)\}$$

lautet. Nun sind die Trigger von r_1 und r_2 aktiv, weshalb beide Abhängigkeiten eingearbeitet werden:

$$D_2' = \{E(\eta_1, \eta_2), E(\eta_2, \eta_1), E(\eta_3, \eta_3), E(\eta_1, \eta_4), E(\eta_4, \eta_2), E(\eta_2, \eta_5), E(\eta_5, \eta_1)\}.$$

Im nächsten Schritt wird erneut der Kern der Instanz berechnet. Dieser besteht ausschließlich aus dem Prädikat $E(\eta_3, \eta_3)$, da dieses mit der Belegung $x := \eta_3, y := \eta_3, u := \eta_3$ sowohl die Abhängigkeiten r_1 als auch r_2 erfüllt (für r_0 existiert kein Trigger mehr):

$$D_2 = core(D_2') = \{E(\eta_3, \eta_3)\} \models \Sigma.$$

An dieser Stelle haben wir eine universelle Lösung D_2 gefunden und der Core CHASE terminiert. Der Core CHASE ist jedoch der einzige CHASE, der vollständig ist, also konfluent ist und in jedem Fall terminiert. Für den Oblivious CHASE gilt dies, wie wir gesehen haben, nicht, doch auch der Standard CHASE ist nicht vollständig. Deshalb werden wir abschließend eine Methode kennenlernen, um den Standard CHASE auf Terminierung zu überprüfen.

6 Terminierung des CHASE

Um den Standard CHASE auf Terminierung zu untersuchen, gibt es eine Reihe von Methoden, darunter die *schwache Azyklizität*, die *starke Azyklizität*, die *C-stratification* und die *Safety*. Wir beschränken uns in diesem Abschnitt auf erstere. Die anderen Methoden können im Buch *„Incomplete Data and Data Dependencies in Relational Databases“* nachgeschlagen werden [GMS12].

Schwache Azyklizität Um eine Menge von tgds auf schwache Azyklizität zu testen, muss zunächst ein Abhängigkeitsgraph aufgebaut werden.

Definition 6.1 (Schwach azyklische Menge von tgds, nach [GMS12])**.** *Sei Σ eine Menge von tgds über einem festen Schema. Ein* Abhängigkeitsgraph *$dep(\Sigma) = (pos(\Sigma), E)$ ist ein Graph mit Knoten in $pos(\Sigma)$, wobei $pos(x)$ die Position eines Terms bestimmt. Die Kantenmenge E wird nun für jede tgd $\varphi(\vec{x}, \vec{z}) \to \exists \vec{Y} : \psi(\vec{x}, \vec{Y})$ in Σ und jedes $x \in \vec{x}$, das in φ an der Position R_i auftaucht, wie folgt gebildet:*

1. *Für jedes Vorkommen von x in ψ an Stelle (Position) S_j wird eine Kante $R_i \to S_j$ von R_i nach S_j erzeugt, sofern sie noch nicht existiert.*

2. *Für jede existenzquantifizierte Variable Y und für jedes Vorkommen von Y in ψ an Stelle T_k wird eine spezielle Kante $R_i \overset{*}{\to} T_k$ erzeugt, sofern sie noch nicht existiert.*

Existiert im gebildeten Abhängigkeitsgraph $dep(\Sigma)$ kein Zyklus durch eine spezielle Kante, so ist Σ schwach azyklisch *und der Standard CHASE terminiert.* □

Erneut betrachten wir dazu ein Beispiel aus [GMS12]: Sei

$$\begin{aligned} r_1: \quad & N(x) && \to \quad \exists Y : E(x, Y), \\ r_2: \quad & S(y) \wedge E(x, y) && \to \quad N(y) \end{aligned}$$

eine Menge von Integritätsbedingungen Σ. Betrachten wir zunächst r_1. Das x an Position 1 in $E(x, Y)$ taucht in $N(x)$ an Position 1 auf. Entsprechend legen wir eine Kante $N_1 \to E_1$ an (Schritt 1). Das existenzquantifizierte Y an Position 2 erhält zudem eine spezielle Kante von x nach Y, also $N_1 \overset{*}{\to} E_2$ (Schritt 2). Nun betrachten wir r_2. Das y in $N(y)$ taucht an Position 1 von $S(y)$ und Position 2 von $E(x, y)$ auf und erzeugt entsprechend zwei neue Kanten $E_2 \to N_1$ und $S_1 \to N_1$ (Schritt 1). Da die tgd keine existenzquantifizierten Variablen besitzt, entfällt Schritt 2. Unser Abhängigkeitsgraph lautet somit

$$dep(\Sigma) = (pos(\Sigma), \{N_1 \to E_1, N_1 \overset{*}{\to} E_2, E_2 \to N_1, S_1 \to N_1\}).$$

Nun ist leicht erkennbar, dass es einen Zyklus $N_1 \overset{*}{\to} E_2 \to N_1$ durch eine spezielle Kante gibt. Σ ist somit nicht schwach azyklisch; der Standard CHASE terminiert wahrscheinlich nicht.

7 Zusammenfassung

In dieser Arbeit haben wir einen grundlegenden Überblick über den *CHASE*-Algorithmus erhalten. In Abschnitt 2 lernten wir die *Tableaus*, *Tableauanfragen* und die Funktionsweise des Algorithmus' auf diese kennen. In Abschnitt 3 lernten wir zunächst *eingebettete Abhängigkeiten* (*egds*, *tgds* und *s-t tgds*) kennen und erfuhren, wie der CHASE diese in Instanzen integrieren kann. Als Anwendungsfälle betrachteten wir dabei *Data Cleaning* und *Data Exchange*. In Abschnitt 4 beschäftigten wir uns mit dem CHASE auf Anfragen und insbesondere mit dem *Answering-Queries-using-Views-Problem (AQuV)*. In Abschnitt 5 erfolgte dann ein Überblick über die verschiedenen Varianten des CHASE. Dabei lernten wir neben dem *Standard CHASE* auch kurz den *Oblivious CHASE* sowie den *Core CHASE* kennen. Zu guter Letzt erfuhren wir in Abschnitt 6 etwas über Terminierungstests für den CHASE, konzentrierten uns dabei jedoch nur auf die *schwache Azyklizität*. Diese Arbeit diente somit als Einstiegspunkt für das Verstehen und Anwenden des Algorithmus'.

Literatur

[BKM+17] Benedikt, Michael ; Konstantinidis, George ; Mecca, Giansalvatore ; Motik, Boris ; Papotti, Paolo ; Santoro, Donatello ; Tsamoura, Efthymia: Benchmarking the Chase. In: *PODS*, ACM, 2017, S. 37–52

[DH13] Deutsch, Alin ; Hull, Richard: Provenance-Directed Chase&Backchase. In: *In Search of Elegance in the Theory and Practice of Computation* Bd. 8000, Springer, 2013 (Lecture Notes in Computer Science), S. 227–236

[DNR08] Deutsch, Alin ; Nash, Alan ; Remmel, Jeffrey B.: The chase revisited. In: *PODS*, ACM, 2008, S. 149–158

[FKMP05] Fagin, Ronald ; Kolaitis, Phokion G. ; Miller, Renée J. ; Popa, Lucian: Data exchange: semantics and query answering. In: *Theor. Comput. Sci.* 336 (2005), Nr. 1, S. 89–124

[FKPT11] Fagin, Ronald ; Kolaitis, Phokion G. ; Popa, Lucian ; Tan, Wang C.: Schema Mapping Evolution Through Composition and Inversion. In: *Schema Matching and Mapping*. Springer, 2011 (Data-Centric Systems and Applications), S. 191–222

[GMS12] Greco, Sergio ; Molinaro, Cristian ; Spezzano, Francesca: *Incomplete Data and Data Dependencies in Relational Databases*. Morgan & Claypool Publishers, 2012 (Synthesis Lectures on Data Management)

[Ile14] Ileana, Ioana: *Query rewriting using views : a theoretical and practical perspective. (Réécriture de requêtes avec des vues : une perspective théorique et pratique)*, Télécom ParisTech, Frankreich, Diss., 2014

[Jur18] Jurklies, Martin: *CHASE und BACKCHASE: Entwicklung eines Universal-Werkzeugs für eine Basistechnik der Datenbankforschung*. Masterarbeit, Universität Rostock, Lehrstuhl für Datenbank- und Informationssysteme, 2018

[Mai83] Maier, David: *The Theory of Relational Databases*. Computer Science Press, 1983

[San14] Santoro, Donatello: *Advanced Techniques for Mapping and Cleaning*, Universität Rom III, Diss., 2014