Bachelorarbeit

Ein Framework für ProSA

eingereicht von: Moritz Hanzig

eingereicht am: 08.03.2022

Gutachter:innen: M. Sc. Tanja Auge
Dr.-Ing Holger Meyer

Zusammenfassung

ProSA ist ein Werkzeug zum Forschungsdatenmanagement, das größtenteils auf dem Papier, aber noch nicht in einer praktischen Implementierung vorliegt. Das Werkzeug soll Forschungsergebnisse reproduzierbar machen, indem es die minimale Teildatenbank berechnet, die mit dem Forschungsergebnis veröffentlicht, und mit dem dieses Ergebnis dann reproduziert werden kann. Neben der Reduzierung des Speicherbedarfs durch die Minimierung wird von ProSA auch eine Anonymisierung vorgenommen, um dem Datenschutz gerecht zu werden und sensible Daten zu maskieren. Die einzelnen Ideen und Komponenten für ProSA wurden im Laufe der letzten Jahre in verschiedenen Projektgruppen und Abschlussarbeiten relativ unabhängig voneinander entwickelt. Ziel dieser Arbeit ist nun die Zusammensetzung dieser Komponenten zu einem vollständigen Gesamtgerüst, sodass die vorgesehene Pipeline von der Eingabe einer SQL-Anfrage bis hin zur letztendlichen Ausgabe einer minimalen, anonymisierten Teildatenbank durchlaufen werden kann. Des Weiteren betrifft eine größere Teilaufgabe den „Provenancer“, eine Komponente zur Behebung von verfahrensbedingtem Informationsverlust in der minimalen Teildatenbank, welcher durch Einarbeitung von sogenannten Provenance-Informationen ausgeglichen werden kann. Vom Provenancer gab es schon eine eigenständige, aber inzwischen veraltete und im ProSA-Kontext inkompatible Version, die in dieser Arbeit neu konzipiert und implementiert wird.
Weiterhin werden in Zusammenarbeit mit den Entwicklern an anderen Komponenten die Menge von unterstützten Anfragen um solche mit Aggregatfunktionen erweitert. Zusätzliche Qualitätsanforderungen an diese Arbeit sind dem Bereich der Softwaretechnik anzuordnen: das entwickelte Rahmengerüst soll natürlich leicht erweiterbar, modular, verständlich, effizient und gut dokumentiert sein.

Abstract

ProSA is a tool for managing research data, whose underlying ideas exist mainly in theory, but does not have a practical implementation yet. This tool's purpose is to make published research results reproducible by computing a minimal subdatabase that can be published alongside with the research result and can be then used to reproduce that result. Additionally to reducing the database's size, ProSA also performs an anonymization of the data to account for privacy and obliterate sensitive data. The individual ideas and components of ProSA have been developed over the course of the past years in separate study groups and theses quite independently from each other. Therefor the purpose of this thesis is to combine all those components into a working framework, so that the intended pipeline does run from start (entering an SQL query) to finish (returning a minimized and anozymized subdatabase). Another substantial part of this thesis is the development of the „Provenancer“ component, which componsates process-related information loss by incorporating so-called provenance information into the subdatabase. There already is a stand-alone implementation of the Provenancer, but it is deprecated an not usable in the context of ProSA, so we will provide a new concept and implementation for that in this thesis.
Furthermore, in collaboration with the developers of other components, we will extend the set of supported SQL queries for ProSA by queries containing aggregate functions. Additional quality requirements for this thesis come from a software engineering point of view: the developed framework should obviously be efficient, easily extendable, unterstandable, modular and well-documented.

Inhaltsverzeichnis

1 Einleitung

Die vorliegende Arbeit beschäftigt sich mit der Implementierung eines Frameworks für das Projekt *ProSA*[1]. Zunächst wollen wir einleitend klären, welches Problem ProSA überhaupt lösen soll und stellen dann die genaue Zielsetzung für diese Bachelorarbeit sowie deren Aufbau vor. Den Abschluss der Einleitung bildet die Vorstellung der Beispieldatenbank, die wir für diverse Beispiele verwenden wollen.

1.1 Problemstellung

Eine elementare Anforderung an moderne Forschung ist die Nachvollziehbarkeit der Ergebnisse. Deshalb sollten die veröffentlichten Forschungsdaten nicht nur das Ergebnis selbst, sondern auch die zugrunde liegenden Daten enthalten. Oftmals ist die Erforschung von Daten, die in einer relationalen Datenbank D gespeichert sind, lediglich eine (komplexe) SQL-Anfrage Q, deren Ergebnis wir R nennen:

$$Q(D) = R$$

Für die Nachvollziehbarkeit von R müsste man eigentlich alle Originaldaten D mitveröffentlichen. Dies ist aber zum einen aus Gründen des Speicherplatzes, aber auch wegen der Privatsphäre meist nicht praktikabel. Der Grundgedanke von ProSA zur Lösung dieses Problems ist in Abbildung 1.1 dargestellt: Wir berechnen eine minimale Teildatenbank D_{min}, die nur den Teil von D enthält, der nötig ist, um das Ergebnis R rekonstruieren zu können:

$$Q(D_{min}) = R$$

Dazu formen wir die ursprüngliche SQL-Anfrage Q in sogenannte *embedded dependencies* (EDs) um, welche wir dann im ersten Schritt, der *Chase-Phase*, mit dem *Chase-Algorithmus*[2] in D einarbeiten können. Mit den theoretischen Grundlagen des Chase befassen wir uns in Kapitel 2.1 genauer. Für die Nutzung des Chase in ProSA greifen wir auf das Werkzeug ChaTEAU[3] zurück, welches diesen Algorithmus praktisch implementiert.
Das Ergebnis der Chase-Phase ist unser Anfrageergebnis R. Die minimale Teildatenbank D_{min} erhalten wir, indem wir im zweiten Schritt, der sogenannten *Backchase-Phase*, die Anfrage Q zu Q^{-1} invertieren und mit ChaTEAU auf unserem Ergebnis R ausführen:

$$Q^{-1}(R) = D_{min}$$

Mit D_{min} haben wir jetzt unsere unsere minimale Teildatenbank, mit der zwar das Ergebnis R prinzipiell komplett reproduziert werden kann, aber oft zu ungenau ist, um die *Provenance* von R im Detail

[1] [AH19], ProSA ist ein Projekt des Lehrstuhls für Datenbank- und Informationssysteme der Universität Rostock. Weitere Informationen finden sich auch auf dessen Webseite `https://dbis.informatik.uni-rostock.de/forschung/aktuelle-projekte/prosa/`.
[2] [GMS12]
[3] u.a. [Jur18]

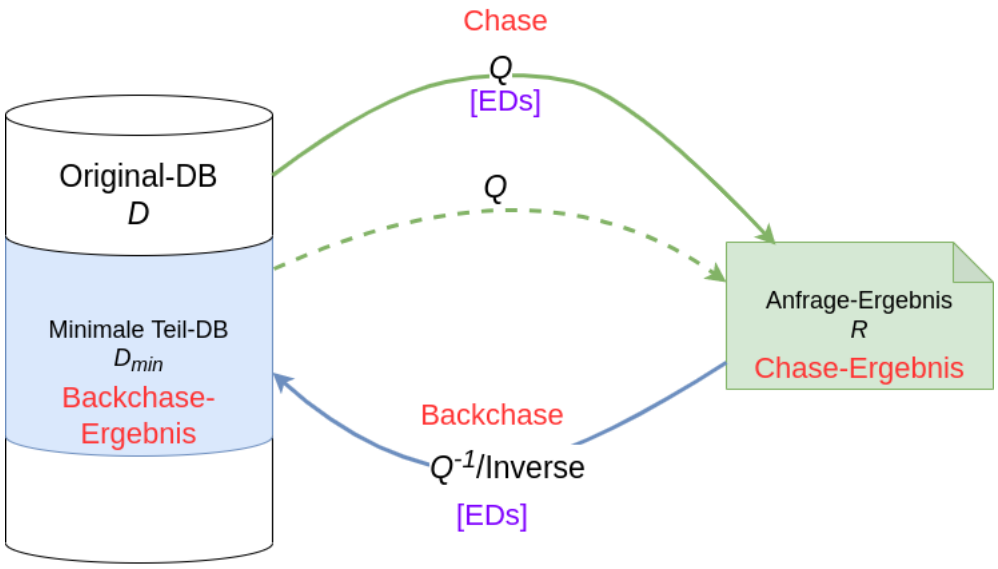


Abbildung 1.1: Die Grundidee hinter ProSA, bei der in den zwei Phasen Chase und Backchase eine minimale Teildatenbank D_{min} berechnet wird, sodass die Ausführung der Anfrage Q auf D_{min} (gestrichelte Linie) immer noch dasselbe Ergebnis liefert.

nachvollziehen zu können. (Data) Provenance[4] ist – simplifiziert – die Antwort auf die Frage, *wie* das Ergebnis einer Datenanalyse (d.h. eine komplexe SQL-Anfrage) zustande kommt. Daher sammelt ProSA bereits während der Chase-Phase sogenannte Provenance-Informationen, mit denen D_{min} zu D_{min_prov} erweitert werden kann, damit sie unserem Anspruch an Nachvollziehbarkeit genügt. Insbesondere die *how*- und *why*-Provenance[5] wollen wir dabei verwenden. Die Grundlagen der Provenance werden noch genauer in Abschnitt 2.2 erklärt. Abschließend soll ProSA D_{min_prov} noch anonymisieren, um sensible Daten zu maskieren.
Nachdem wir jetzt die grobe Idee hinter ProSA verstanden haben, erschließt sich auch der Name des Projektes: er steht für „***Pro**venance Management durch **S**chema-Abbildungen und **A**nnotationen*“. Nun ist noch zu klären, welchen Beitrag diese Bachelorarbeit am Projekt ProSA leisten soll, d.h. was die konkrete Zielsetzung ist.

1.2 Zielsetzung und Aufbau der Arbeit

Die einzelnen Komponenten von ProSA wurden im Laufe der letzten Jahre (Stand: Wintersemester 2021) von unterschiedlichen Leuten entwickelt[6] und müssen nun zusammengesetzt werden. Dies ist das Hauptziel dieser Arbeit. Derzeit kann die GUI beispielsweise noch gar nicht das Ergebnis des Parsers entgegennehmen (siehe Abschnitt 3). Wir setzen uns das explizite Ziel, *genau eine* konkrete Beispielanfrage zu formulieren, für welche die Pipeline von Anfang (Eingabe der SQL-Anfrage) bis Ende (Ausgabe der berechneten minimalen Teildatenbank) durchlaufen soll.
Dabei haben wir die Anfrage so gewählt, dass sie als genau eine *s-t tgd* (eine spezielle ED) formuliert werden kann. Diese Einschränkung hat den Grund, das komplette Framework erst einmal für diesen simplen Fall implementieren zu können, ohne dass die Komplexität der EDs dabei Probleme machen. Sobald dieser Durchlauf der Pipeline funktioniert, kann ProSA immer noch so erweitert werden, dass es mit komplexeren Anfragen umgehen kann. Als zweites Kriterium soll unsere gewählte Anfrage jeweils ohne bzw. mit Berücksichtigung von Provenanceinformationen unterschiedliche Ergebnisse liefern. Das komplette

[4] Wenn wir von Provenance im Allgemeinen reden, meinen wir eigentlich immer Data Provenance. Workflow Provenance ist im Kontext von ProSA irrelevant.
[5] [GT17], [BKT01], [CCT09]
[6] GUI: [FHO+21], Parser: [KRSZ21], ChaTEAU: u.a. [Jur18], [Gör20], [Zim20], [Ros20], [Ren19]

durchlaufende Beispiel mit allen Zwischenschritten ist in Kapitel 4.10 zu finden. Wie die Erweiterung der minimalen Teildatenbank D_{min} um Provenanceinformationen *genau* durchgeführt wird, steht noch nicht fest – hierfür wollen wir verschiedene Ansätze entwickeln und evaluieren.
Fest steht, dass das Framework am Ende zwei unterschiedliche Ergebnisse D_{min} und D_{min_prov} berechnen und ausgeben können soll. Aus dieser Zielsetzung ergeben sich mehrere Unteraufgaben, welche den hier beschriebenen Aufbau für diese Arbeit zur Folge haben:

Kapitel 2: Die für das Verständnis von ProSA notwendigen **theoretischen Grundlagen** erklären, insbesondere den Chase-Algorithmus und Provenance.

Kapitel 3: Den **Ist-Zustand der Komponenten zu Beginn der Arbeit** im Oktober 2021 darlegen.

Kapitel 4: **Konzept und Umsetzung** erarbeiten, d.h. die Architektur des Frameworks planen, Designentscheidungen erläutern, verworfene und umgesetzte Ideen auflisten und anhand eines konkreten Beispiels alle Zwischenschritte der konzipierten Pipeline durchgehen.

Kapitel 5: Einen **Ausblick** über weitere Verbesserungen und Erweiterungen für ProSA geben und ein **Fazit** ziehen.

1.3 Beispieldatenbank

Sowohl zur Erläuterung der theoretischen Grundlagen, als auch für das bei der Implementierung verwendete Beispiel verwenden wir eine simple Studierendendatenbank mit zwei Relationen `students` und `participants` (Abbildung 1.2). Das Attribut für die Matrikelnummer `matno` ist Primärschlüssel in der Relation `students` und Fremdschlüssel in der Relation `participants`. Die roten Tupel-IDs `students_id` und `participants_id` sind **nicht** Teil der originalen Eingaben, sondern werden später vom Framework hinzugefügt. Sie sind hier aber bereits mit dargestellt, damit wir die einzelnen Tupel in Beispielen eindeutig referenzieren können. In Anhang 6.10 ist das SQL-Skript abgebildet, mit dem diese Datenbank erstellt werden kann.

students_id	firstname	matno
students_1	Stefan	1
students_2	Stefan	2
students_3	Mia	3
students_4	Anna	4

(a) Relation students(firstname, matno)

participants_id	modulename	matno
participants_1	Maths	2
participants_2	Maths	3
participants_3	Info	4
participants_4	Logic	1
participants_5	Logic	2

(b) Relation participants(modulename, matno)

Abbildung 1.2: Die in dieser Arbeit verwendete Beispieldatenbank

2 Theoretische Grundlagen

Um zu verstehen, wie die Komponenten von ProSA zusammenarbeiten, müssen wir zunächst die zugrunde liegende Theorie kennenlernen. Im Einzelnen sind das der Chase-Algorithmus[1] und (Data-)Provenance, die wir uns in diesem Kapitel genauer anschauen wollen.

2.1 Der Chase-Algorithmus

Dreh- und Angelpunkt von ProSA ist das Tool ChaTEAU[2], das genau wie ProSA über die Jahre am Rostocker Datenbankenlehrstuhl entwickelt wurde und auch derzeit noch weiterhin aktiv weiterentwickelt wird. ChaTEAU steht für ***Cha**se for **T**ransforming, **E**volving, and **A**dapting databases and queries, **U**niversal approach* und implementiert den Chase-Algorithmus.
Der Chase-Algorithmus ist ein grundlegendes Werkzeug in der Datenbankforschung, mit dem Abhängigkeiten $\star$ in ein Objekt $\bigcirc$ eingearbeitet werden können, sodass diese danach im Objekt selbst implizit enthalten sind:

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

Das **Objekt** $\bigcirc$ kann eine Datenbankinstanz (d.h. formal eine Menge von Tupeln) oder eine Anfrage sein. Im Kontext von ProSA ist das Objekt aber *immer* eine Datenbankinstanz, daher können wir uns auf den *Chase auf Instanzen* beschränken. Diese Instanz nennen wir allgemein I.
Abhängigkeiten $\star$ können konkret Integritätsbedingungen oder eine SQL-Anfrage (wie bei ProSA) sein. Dabei muss diese SQL-Anfrage aber in eine bestimmte Form überführt werden – in sogenannte *eingebettete Abhängigkeiten*, bzw. bestimmte Varianten davon: *egds*, *tgds* und *s-t tgds*. Schauen wir uns an, was das genau bedeutet, wobei wir uns an den Definitionen in [Sch21], bzw. der dort verwendeten Primärquelle [GMS12] orientieren:

Definition 2.1 (Eingebettete Abhängigkeit/embedded dependency (ED), nach [GMS12])**.** Eine *eingebettete Abhängigkeit* ist ein prädikatenlogischer Ausdruck erster Stufe der Form

$$\forall x, y : \varphi(x, y) \rightarrow \exists z : \psi(x, z).$$

Dabei sind x,y und z Variablenvektoren (d.h. Tupel $(x_1, ..., x_n)$, $(y_1, ..., y_m)$ und $(z_1, ..., z_k)$). $\varphi(x, y)$ und $\psi(x, z)$ sind Konjunktionen über atomare Formeln. Den Teil links des Implikationspfeils, also $\varphi(x, y)$, nennt man *Rumpf* der Abhängigkeit. Den Teil rechts, also $\psi(x, z)$, nennt man *Kopf*.

Egds, tgds und s-t tgds sind also spezielle EDs, die wir nun im Einzelnen definieren. Beginnen wir mit den egds:

[1] [GMS12]
[2] [AH19], [AH], [Jur18], [Gör20], [Zim20], [Ros20], [Ren19]

Definition 2.2 (equality-generating dependency (egd), nach [GMS12])**.** Eine gleichheitserzeugende Abhängigkeit *egd* ist eine spezielle ED der Form:

$$\forall x : F_1(x) \rightarrow x_1 = x_2.$$

Dabei besteht der Kopf der ED aus einem einzigen Gleichheitsatom, denn es gibt keine existenzquantifizierten Variablen in einer egd [Sch21].

Durch die Einarbeitung einer egd können wir also Variablen in einer Instanz gleichsetzen. Das können wir später in dieser Arbeit zur Duplikateliminierung benutzen. Betrachten wir beispielsweise zwei Tupel `students('Bob', '23')` und `students`(η, `'23'`), die dieselbe Matrikelnummer `'23'` haben, aber das zweite Tupel hat einen Nullwert η für den Vornamen. In folgender egd können wir formulieren, dass die Matrikelnummer den Vornamen funktional bestimmt:

$$\forall firstname_1, firstname_2, matno : \texttt{students}(firstname_1, matno) \land \texttt{students}(firstname_2, matno) \\ \rightarrow firstname_1 = firstname_2.$$

Wenn wir mit dem Chase diese egd einarbeiten, wird der Nullwert η durch `'Bob'` ersetzt und dadurch, dass der Chase komplett mengenorientiert arbeitet, fällt das neue Duplikat `students('Bob', '23')` weg. Damit haben wir die Integritätsbedingung erfüllt und unsere Datenbank bereinigt.
Wenn in einem weiteren Beispiel allerdings das zweite Tupel anstatt eines Nullwerts η auch einen konkreten Wert `'Alice'` für den Vornamen hätte, wüsste der Chase nicht, welcher der beiden Vornamen `'Bob'` und `'Alice'` der richtige ist und würde **abbrechen**. Damit wird die nicht (automatisiert) reparierbare **Verletzung von Integritätsbedingungen** signalisiert. Neben „*Konstante vs. Nullwert*“ und „*Konstante vs. Konstante*“ gibt es auch noch andere Kombinationen, wie die zwei Terme des Gleichheitsatoms zusammengesetzt sein können. Wie der Chase auf diese einzelnen Fälle reagiert, ist im Pseudocode des Chase (Listing 2.1) ab Zeile 5 beschrieben.

Duplikateliminierung ist ein besonderer Anwendungsfall des Chase und kommt in ProSA nur am Rande zum Tragen, eigentlich wollen wir ja eine SQL-Anfrage als ED formulieren. Dazu verwenden wir tgds und s-t tgds. Schauen wir uns an, worum es sich dabei handelt:

Definition 2.3 (tuple-generating dependency (tgd))**.** Eine tgd ist eine als prädikatenlogischer Ausdruck erster Stufe dargestellte Abhängigkeit der Form:

$$\forall x : F_1(x) \rightarrow \exists y : F_2(x, y).$$

F_1 und F_2 sind Konjunktionen von atomaren Formeln über Variablen aus x bzw. x und y. Sind Variablen aus dem Kopf existenzquantifiziert, ist die tgd eingebettet, andernfalls voll[Sch21].

Tgds erzeugen neue Tupel; die (neu erzeugten,) existenzquantifizierten Variablen erhalten dann nummerierte Nullwerte η_i. Als anschauliches Beispiel für eine sinnvolle Anwendung einer tgd für unsere Datenbank können wir eine Integritätsbedingung formulieren, welche für jeden Eintrag in der `participants`-Relation, ein Tupel in einer neuen Relation `modules` = {*modulename*, *teacher*} erwartet. Wenn dieses `modules`-Tupel nicht existiert, wird es vom Chase neu angelegt und darin für das existenzquantifizierte Attribut für *teacher* einen nummerierter Nullwert η_i eingetragen. Als tgd können wir das so formulieren:

$$\forall modulename, matno : \texttt{participants}(modulename, matno) \\ \rightarrow \exists teacher : \texttt{modules}(modulename, teacher).$$

Jetzt können wir mit egds Tupel gleichsetzen, mit tgds komplett neue Tupel erzeugen, aber noch nicht Tupel „verschieben“, was für die Darstellung einer SQL-Anfrage eigentlich am wichtigsten ist. Der Grund dafür ist, dass wir bei der Verarbeitung einer SQL-Anfrage eigentlich immer eine Quell-Instanz mit den Originaltupeln nach gewissen Kriterien filtern oder modifizieren und dann in einer Ergebnis- oder Ziel-Relation ausgeben wollen. Für diese „Verschiebung von Tupeln“ können wir s-t tgds benutzen, die wir nun kennenlernen:

Definition 2.4 (source-to-target tgd (s-t tgd))**.** Eine s-t tgd ist ein Spezialfall der tgd, bei dem F_1 eine Konjunktion über einem Quellschema S und F_2 eine Konjunktion über einem Zielschema T ist [GMS12] (daher kommt auch der Name **s-t** tgd).

Anders als tgds erzeugen s-t tgds die neuen Tupel nicht *zusätzlich*, sondern überführen sie vom Quell- ins Zielschema, sodass die Tupel im Quellschema sozusagen „gelöscht“ werden. Betrachten wir beispielsweise unsere gesamte Beispieldatenbank als Quellschema und die Relation `Result` $= \{firstname\}$ als Zielschema, so ist

$$\forall firstname, matno : \texttt{students}(firstname, matno) \wedge matno < 3 \\ \rightarrow \texttt{Result}(firstname)$$

eine s-t tgd, die alle Namen von Studierenden mit einer Matrikelnummer kleiner als 3 ausgibt.

Zuletzt müssen wir zum Verständnis des Chase noch *Homomorphismen* und *Trigger* definieren:

Definition 2.5 (Homomorphismus für Instanzen, [Sch21], leicht modifiziert)**.** Seien I_1 und I_2 Instanzen über einem Schema A, $Const_i$ die Menge aller Konstanten und Var_i die Menge aller Variablen der jeweiligen Instanz I_i. Eine Funktion $h : Const_1 \cup Var_1 \rightarrow Const_2 \cup Var_2$ heißt Homomorphismus genau dann, wenn:

- $\forall c \in Const_1 : h(c) = c$
- $\forall R \in A, \forall (a_1, ..., a_n) \in R^{I_1} : (h(a_1), ..., h(a_n)) \in R^{I_2}$
  (R^I : Instanz I über der Relation R)

Einen Homomorphismus vom I_1 nach I_2 schreibt man als $I_1 \rightarrow I_2$.

Definition 2.6 (Trigger)**.** Ein Trigger für eine Abhängigkeit (ED) existiert, wenn mithilfe eines Homomorphismus der Rumpf der Abhängigkeit auf einen Teil des Objektes abgebildet wird, beziehungsweise dass das Muster des Rumpfes im Objekt vorkommt.

Ein geläufigerer Begriff für das „Finden von Triggern mittels Homomorphismen“ ist wahrscheinlich das *Pattern Matching*. Es wird also lediglich geguckt, ob es in der Instanz ein Tupel gibt, dessen Muster auf den Rumpf der ED „matcht“.
Ein Trigger ist *aktiv*, wenn durch Anwendung des Kopfes der Abhängigkeit unter dem gleichen Homomorphismus neue Werte oder Tupel entstehen, also die Abhängigkeit im Objekt noch nicht erfüllt ist [BKM+17].

Den Chase auf Instanzen I sehen wir in Listing 2.1:

```
1  FOREACH Trigger t für eine Abhängigkeit g ∈ ⋆ DO
2      IF t ist aktiv THEN
3          IF g hat Form ∀x : F1(x) → ∃y : F2(x, y) (TGD) THEN
4              F2 anwenden, neue Tupel in I aufnehmen
5          ELSE IF g hat Form ∀x : F1(x) → x1 = x2 (EGD) THEN
6              IF x1, x2 sind Konstanten und x1 ≠ x2 THEN
7                  CHASE schlägt fehl
8              ELSE IF x1 ist Konstante THEN
9                  x2 ← x1
10             ELSE IF x2 ist Konstante ∨ x1, x2 sind Nullwerte THEN
11                 x1 ← x2
```

Listing 2.1: Pseudocode des Chase auf Instanzen, übernommen aus [Ros20], S. 13.

Diese Definitionen sind natürlich sehr abstrakt, daher betrachten wir als Beispiel die SQL-Anfrage `SELECT firstname FROM students WHERE matno='3'`, die wir an unsere Datenbankinstanz I stellen wollen.
Der Chase kann natürlich nicht mit SQL-Anfragen arbeiten, sondern wie schon kurz angemerkt nur mit egds, tgds und s-t tgds, daher formen wir die SQL-Anfrage in eine s-t tgd um:

$$\forall firstname : \texttt{students}(firstname, \texttt{'3'}) \\ \rightarrow \texttt{Result}(firstname)$$

Das bedeutet, für jedes Tupel aus der Relation `students`, dessen Attribut $firstname$ beliebig sein kann ("$\forall$") und dessen Attribut $matno$ = `'3'` ist, ein neues Tupel in der `Result`-Relation angelegt werden soll mit dem Attribut $firstname$ und demselben Attributwert wie im Rumpf.
Diese neue `Result`-Relation gab es vorher noch nicht. Damit der Chase korrekt funktioniert, müssen wir das Zielschema `Result` = $\{firstname\}$ für diese Relation also neu definieren und zur Eingabe hinzufügen.
Bekommt der Chase nun die komplette Eingabe, welche aus den nötigen Quell- und Zielschemata, der als s-t tgd formulierten Anfrage und den Instanztupeln zusammengesetzt wurde, so iteriert er über alle Abhängigkeiten (bei uns nur eine einzige, nämlich die s-t tgd). Pro Abhängigkeit sucht er dann zwischen deren Rumpf (dem Teil links des Implikationspfeils) und den Instanz-Tupeln nach Homomorphismen. Wenn er einen solchen Homomorphismus gefunden hat, wendet er den Kopf der Abhängigkeit entsprechend an. Konkret heißt das hier:

- Gibt es ein Tupel der Relation `students`, bei dem `matno='3'`?
- Ja, Homomorphismus gefunden bei (`'Mia'`, `'3'`).
- Ist dieser Trigger aktiv, d.h. würde sich durch Einfügen des Tupels die Relation im Kopf der Abhängigkeit ändern? → Ja.
- Füge das erzeugte Tupel (`'Mia'`) in die noch leere Relation `Result` ein.
- Gibt es sonst einen Homomorphismus für diese Abhängigkeit? → Ja, und zwar denselben.
- Ist dieser Trigger aktiv, d.h. würde sich durch Einfügen des Tupels die Relation im Kopf der Abhängigkeit ändern? → Nein, die Ergebnisrelation bleibt gleich, da sie eine Menge ist und Einfügen eines Duplikats keine Wirkung hat.
- Gibt es sonst einen Homomorphismus für diese Abhängigkeit? → Nein, da alle anderen Matrikelnummern ≠ `'3'` sind.
- Gibt es sonst noch Abhängigkeiten? → Nein.
- Terminierung des Chase.

Nach Ausführung des Chase liegt also das Anfrageergebnis in der Ergebnisrelation `Result` vor, ganz genau so, als hätten wir die SQL-Anfrage mit einem DBMS ausgeführt.

2.2 Provenance

In der Datenbankforschung wird unter anderem zwischen *Data Provenance* und *Workflow Provenance* unterschieden.[3] Beide fragen nach dem Ursprung oder der Herkunft eines Ergebnisses. Workflow Provenance beschäftigt sich damit, wie ein Ergebnis verschiedene Arbeitsschritte in der Datenverarbeitung durchläuft und wie es dabei verändert wird. Ein Beispiel für eine solche Veränderung ist der Medienbruch, der durch die Digitalisierung antiker Schriftrollen zustande kommt. Workflow Provenance würde hier danach fragen, inwiefern Informationen durch diese Digitalisierung verloren gehen. Workflow Provenance ist für ProSA irrelevant, da es darin keinen harten Medienbruch gibt und es allein mit Data Provenance arbeitet.
Laut der Arbeit von Cheney et. al. in *Provenance in Databases: Why, How, and Where. Foundations and Trends in Databases*[4] gibt es drei Unterarten von Data Provenance, die die Herkunft von Tupeln in einem Ergebnis R einer (SQL-)Anfrage Q auf eine Datenbank D begründen sollen. Nach [CCT09] sind dies sind die *where*-, *why*- und *how*-Provenance:

- **where**-Provenance beantwortet, aus welchen Relationen die Ergebnistupel stammen
  - Antwort als *Zeugenliste*
- **why**-Provenance beantwortet zusätzlich, weshalb ein Tupel im Ergebnis zustande kommt
  - Antwort als *minimale Zeugenbasis*
- **how**-Provenance beantwortet zusätzlich, wie genau ein Ergebnistupel berechnet wurde.
  - Antwort als *Provenance-Polynom* [Aug17]

Streng genommen gibt es auch noch die **why not**-Provenance („Wieso sind gewisse Tupel im Ergebnis *nicht* enthalten?“); diese wird in ProSA allerdings nicht verwendet. Wichtig für das Verständnis der Funktionsweise von ProSA mitzunehmen ist, dass die Provenance-Arten eine Implikationshierarchie bilden: **how**-Provenance ist am aussagekräftigsten und liefert implizit die **why**- und **where**-Provenance. **why**-Provenance ist am zweitaussagekräftigsten und liefert implizit die where-Provenance. Es reicht also, die **how**-Provenance zu berechnen - die anderen beiden können aus dem Provenance-Polynom abgeleitet werden[5]:

$$\textbf{how} \geq \textbf{why} \geq \textbf{where}$$

Daher interessiert uns bei ProSA für die Einarbeitung von Provenanceinformationen insbesondere die **how**-Provenance, da mit ihr am wenigsten Informationen bei der Generierung der minimalen Teildatenbank verloren gehen.

Definition 2.7. (Provenance-Polynom, nach [Aug17]) Ein Provenance-Polynom gibt die Berechnungsvorschrift für ein Ergebnis an. Es ist auf dem kommutativen Halbring $\mathbb{N}[X] = (\mathbb{N}[X], +, \cdot, 0, 1)$ definiert, wobei X eine Menge von Tupel-IDs einer Datenbankinstanz ist. Hierbei steht der Operator „+“ für Projektion oder Vereinigung und Operator „·“ für Verbunde, Selektionen und Schnittmengenbildung zwischen je zwei Tupeln.

[3] [HDB17]
[4] [CCT09]
[5] [Aug17]

Zum Beispiel könnte ein Ergebnistupel mit dem Provenance-Polynom $(t_1 \cdot t_2) + (t_1 \cdot t_3)$ entstanden sein, indem ein Tupel mit der ID t_1 jeweils mit t_2 und t_3 gejoint wurde, und deren Ergebnisse dann vereinigt wurden, wodurch ein Duplikat eliminiert wurde.[6]

Ohne dem Konzeptkapitel 3 vorzugreifen, sollten wir kurz einordnen, welche Implikationen die Arbeit mit Data Provenance in ProSA hat:
Weil Data Provenance mit eindeutigen Tupeln arbeitet, müssen diese IDs bei der Entwicklung des Frameworks irgendwo eingefügt werden. Außerdem kann ChaTEAU, wie wir im nächsten Kapitel 3 noch lernen werden, bereits während der Chase-Phase die **where-**, **why-** und **how**-Provenance berechnen. Die vorläufige, grobe Idee ist also, diese gesammelten Provenance-Informationen später nach der Backchase-Phase mit der berechneten minimalen Teildatenbank D_{min} auf *irgendeine Art und Weise* zu D_{min_prov} zu verknüpfen. Verschiedene Ansätze dafür beschreiben wir in Kapitel 4.3.4. Letztendlich haben wir uns aber stattdessen dazu entschieden, eine eigene Art der Provenance für ProSA, die sogenannte *ProSA-Provenance*, zu verwenden, die uns zumindest für unser konkretes Beispiel sogar noch genauere Provenance-Informationen als die **how**-Provenance liefern kann. Die Konzipierung und Entwicklung dieser ProSA-Provenance ist aber – wie gesagt – erst im Laufe der Arbeit entstanden, deshalb ist sie nicht hier im Grundlagenkapitel, sondern später im Konzeptkapitel 4.3 beschrieben.

In diesem Kapitel haben wir die theoretischen Grundlagen für ProSA kennen gelernt, im Einzelnen den Chase-Algorithmus, mit dem sowohl das Anfrageergebnis, also auch die minimale Teildatenbank berechnet werden kann, als auch grob das Thema Provenance behandelt. Nun schauen wir uns an, inwiefern diese Theorie in den bereits vorhandenen Komponenten bereits umgesetzt wird.

[6] Eine ausführlichere Definition der anderen Provenance-Antwortarten *Zeugenliste*, *minimale Zeugenbasis* mitsamt erklärendem Beispiel kann in [Sch20], Kapitel 2.1.1 (S.13 ff.) nachgelesen werden. Alternativ sei auch hier auf die Originalliteratur [CCT09] verwiesen

3 Ist-Zustand der Komponenten zu Beginn der Arbeit

Nun beschreiben wir den Jetzt-Zustand der Komponenten zum Beginn dieser Arbeit im Oktober 2021. Zur Veranschaulichung des geplanten Gesamtaufbaus von ProSA soll die Abbildung 3.1 dienen. Im Folgenden gehen wir für die einzelnen Komponenten jeweils durch, welche Funktionen sie damals schon erfüllt haben und welche weiteren Funktionen für sie geplant waren. Dabei ist allerdings zu erwähnen, dass im Lauf dieser Arbeit zu diesen ursprünglich geplanten Änderungen noch weitere hinzukamen, die im finalen Konzept (Kapitel 4) umgesetzt wurden.

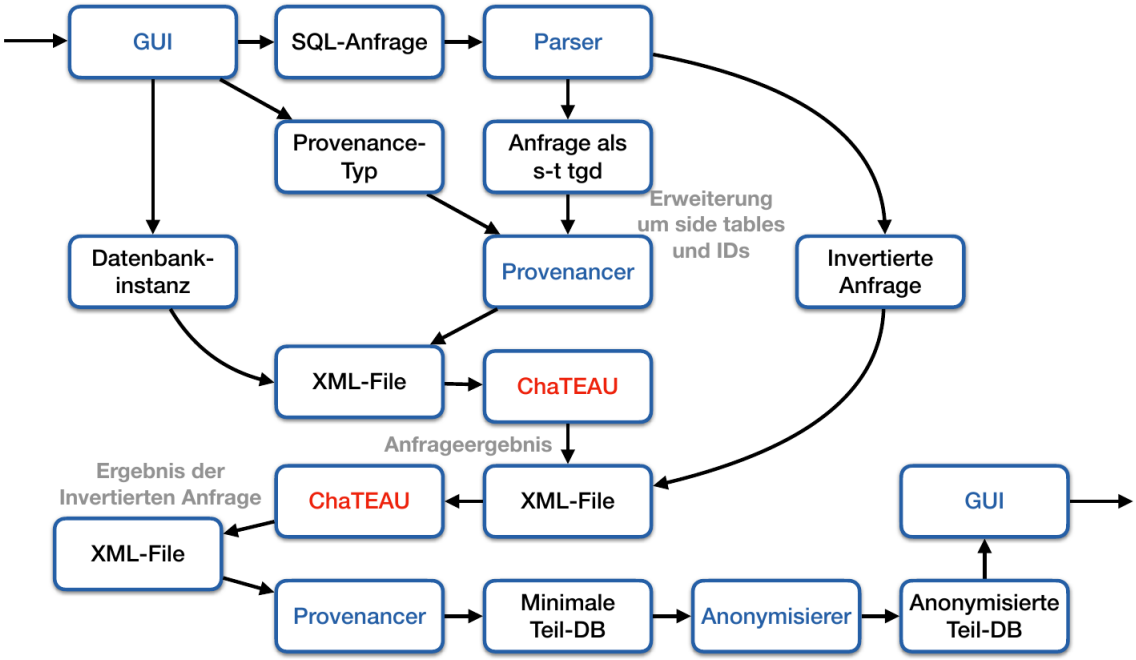


Abbildung 3.1: Überblick über die Komponenten und den geplanten Ablauf von ProSA im Oktober 2021. Grafik aus [Aug22] (leicht modifiziert).

3.1 GUI

Die GUI ist die zentrale Schnittstelle zwischen Benutzer:in und ProSA. Sie ist im Rahmen der Seminararbeit „Komplexe Softwaresysteme“ im Sommersemester 2021 entstanden, deren Abschlussbericht unter [FHO+21] abrufbar ist. Die GUI ist aus vier Tabs aufgebaut, die in den Abbildungen 3.2 bis 3.5 zu sehen sind:

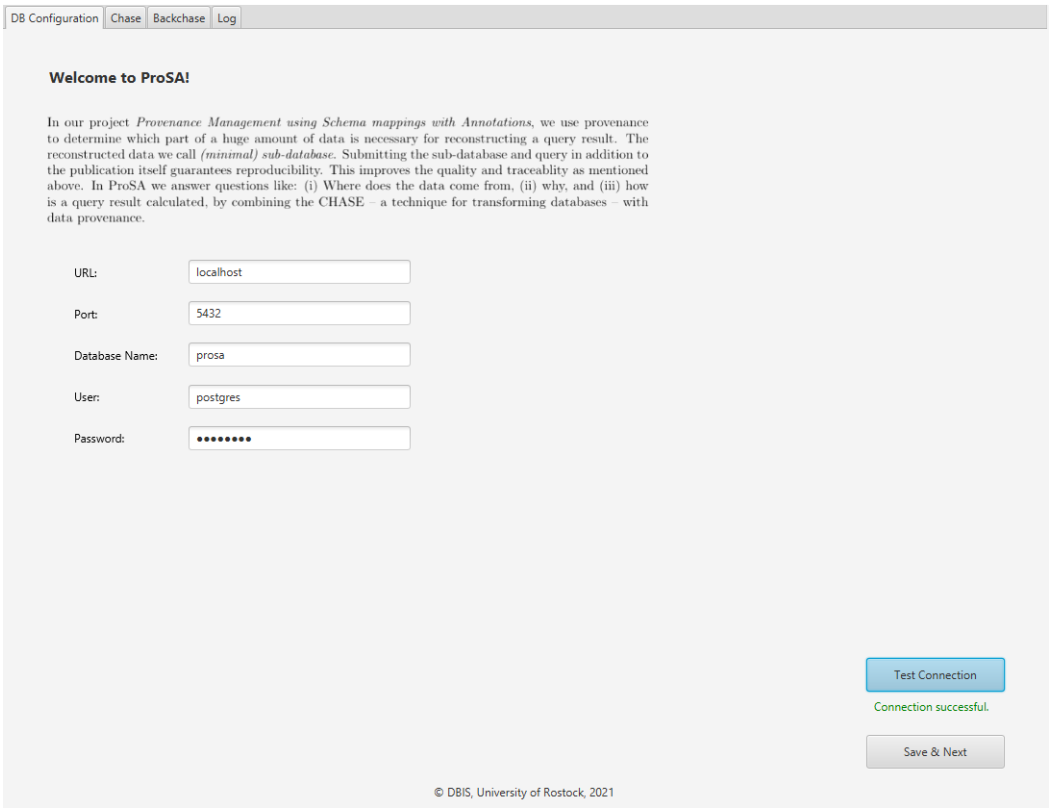


Abbildung 3.2: Der DB Configuration-Tab in der vorliegenden Version, Grafik von [FHO+21], Abb. 6

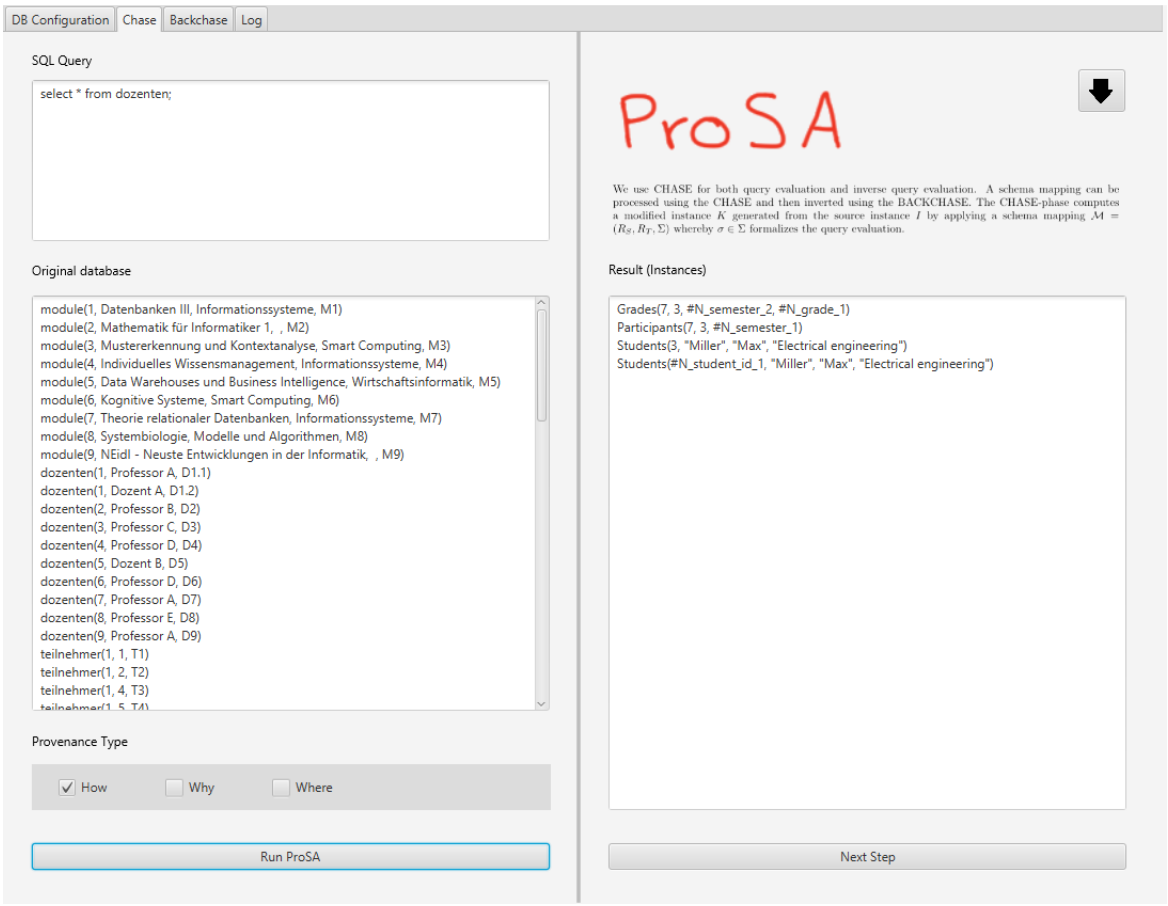


Abbildung 3.3: Der Chase-Tab in der vorliegenden Version, Grafik von [FHO+21], Abb. 7

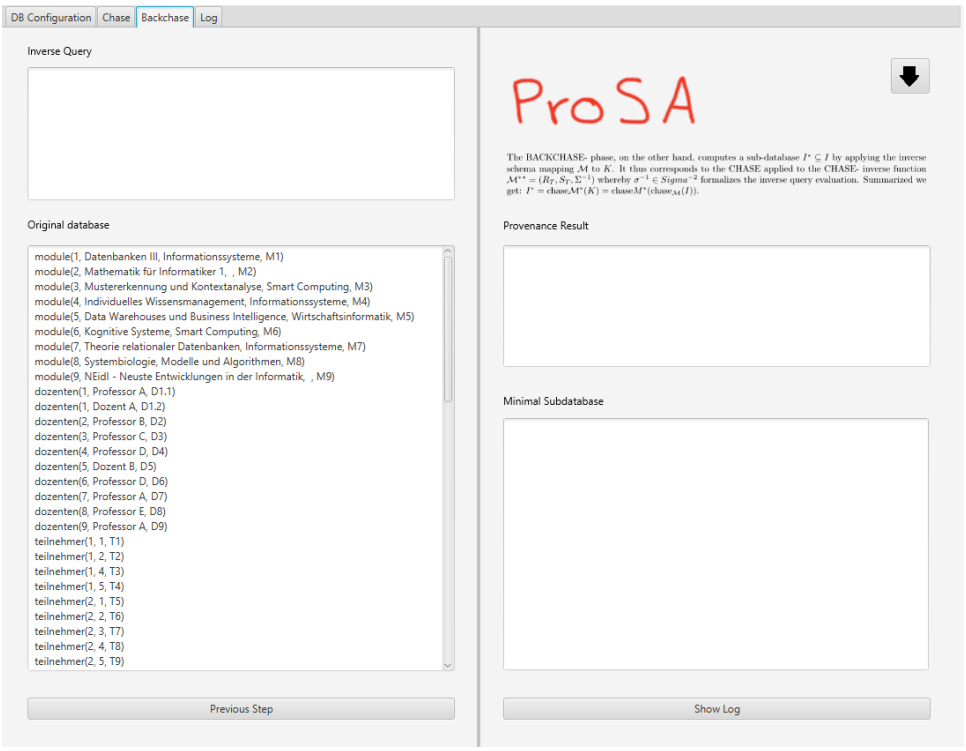


Abbildung 3.4: Der Backchase-Tab in der vorliegenden Version, Grafik von [FHO+21], Abb. 8

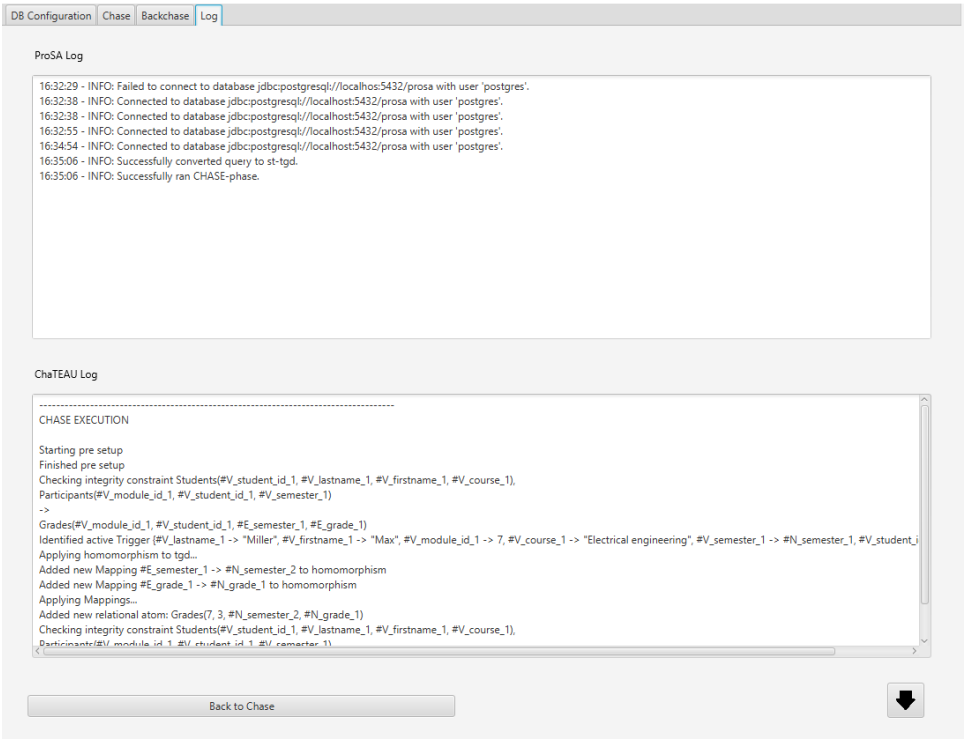


Abbildung 3.5: Der Log-Tab in der vorliegenden Version, Grafik von [FHO+21], Abb. 9

Gehen wir nun für die einzelnen Tabs durch, was deren Ausgangszustand war und was für Änderungen im Oktober 2021 geplant waren. Dabei erwähnen wir teilweise auch schon andere Komponenten, wie beispielsweise den Parser oder Invertierer, weil diese mit der GUI zusammenarbeiten. Im Detail erklärt werden diese aber später nochmal, nachdem wir die GUI kennengelernt haben. Wir geben bei der Betrachtung der GUI teilweise schon einen Ausblick, was zu den damals geplanten Änderungen später noch dazu kam. Die kompletten Änderungen in der finalen GUI können in Kapitel 4.2 nachgelesen werden. Aber zunächst konzentrieren wir uns auf die Pläne zu Beginn der Arbeit:

- DB Configuration (siehe Abb. 3.2)
  - **Derzeitige Funktionen** Hier können die nötigen Informationen eingetragen werden, um eine Datenbankverbindung aufzubauen. Eine Auswahl eines eigenen Datenbankschemas, dessen Relationen die Eingabe für ProSA sein sollen, ist derzeit noch nicht möglich. Dahinter steckt eine JSON-Konfigurationsdatei, die auch manuell angepasst werden kann, ohne die GUI zu benutzen.
  - **Geplante Funktionen** Keine weiteren. Auch die Auswahl eines eigenen Datenbankschemas hat erstmal keine Priorität.
- Chase (siehe Abb. 3.3)
  - **Derzeitige Funktionen** Im Textfeld „Original database“ werden alle Tupel der Originaldatenbank angezeigt. In das „SQL-Query“-Feld kann die Nutzer:in zwar prinzipiell die Anfrage an die Datenbank eingeben, sie wird aber von ProSA noch nicht entgegengenommen oder verarbeitet. Ebenso gibt es die Kontrollkästchen dafür, welcher „Provenance-Type“ in der Chase-Phase berechnet werden soll, aber die Funktion dahinter ist auch noch nicht implementiert.
  - **Geplante Funktionen** Sämtliche Textfelder mit Ausnahme von „Original database“ haben noch keine Funktion, d.h. hier müssen Parser und ChaTEAU so integriert werden, dass die Eingabe-XML für die Chase-Phase aus der „Original database“ als `instance` und der vom Parser generierten s-t tgd als `dependencies` kombiniert wird. Dann soll ChaTEAU mit dieser Eingabe ausgeführt und das Anfrageergebnis im „Result (Instances)“-Textfeld ausgegeben werden. Später kam noch zu diesen anfangs geplanten Änderungen hinzu, im Textfeld „Original database“ die Tupel bereits mit ihren zugefügten IDs anzuzeigen sowie andere GUI-Elemente, um bspw. die vom Parser berechnete s-t tgd auszugeben (vgl. Kapitel 4.2: „*Vorstellung der finalen GUI*“).
- Backchase (siehe Abb. 3.4)
  - **Derzeitige Funktionen** Hier werden genau wie im Chase-Tab die Tupel der Originaldatenbank angezeigt. Alle anderen Textfelder sind noch ohne Funktion.
  - **Geplante Funktionen** Die noch leeren Textfelder müssen um ihre entsprechende Funktion erweitert werden: Die s-t tgd des Invertierers muss eingelesen und im Textfeld „Inverse Query“ angezeigt werden. Die von der Nutzer:in mit den Kontrollkästchen auf dem Chase-Tab ausgewählte Provenance muss in eine textuelle Repräsentation gebracht und im Textfeld „Provenance Result“ dargestellt werden. Schließlich wollen wir im Textfeld „Minimal Subdatabase“ die in der Backchase-Phase berechnete minimale Teildatenbank eintragen. Auch beim Backchase-Tab sind im Laufe der Arbeit mehr Änderungen als anfangs geplant dazugekommen, die in Kapitel 4.2 aufgelistet sind.
- Log (siehe Abb. 3.5)
  - **Derzeitige Funktionen** Hier werden die zwei Logs von ProSA und ChaTEAU getrennt in zwei Textfeldern ausgegeben. Davon wird bis jetzt nur das Log von ProSA überhaupt richtig angezeigt, das Log von ChaTEAU im Status Quo noch gar nicht. In Abbildung 3.5 sind hier zwar Einträge zu sehen, diese wurden dort aber manuell eingefügt.

– **Geplante Funktionen** Zuerst muss das Log von ChaTEAU überhaupt korrekt angezeigt werden. Außerdem ist eine denkbare Verbesserung, die beiden Logs in einem *gemeinsamen* Textfeld anzuzeigen und bspw. farblich oder durch einen Präfix „`ChaTEAU:`“ bzw. „`ProSA:`“ voneinander zu unterscheiden, da dies die zeitliche Abfolge der Logeinträge übersichtlicher darstellt. Das hat aber geringe Priorität.

Die GUI ist mit JavaFX umgesetzt, was eine vielfältige Plattformunterstützung mitbringt, allerdings etwas umständlich für die Entwicklung ist, da nur in der FXML-Konfigurationsdatei beschrieben ist, welche Listener-Methode für die jeweiligen GUI-Elemente registriert ist. Dadurch kann nur umständlich ohne Nutzung der „Call Hierarchy“ einer IDE nachvollzogen werden, welche Methoden wann aufgerufen werden. Dies zu beheben hat allerdings auch erstmal keine Priorität.
Zu guter Letzt hat der Anonymisierer noch gar keine Repräsentation in der GUI. Dafür wollen wir die GUI um einen weiteren Tab in Absprache mit der parallelen Arbeit am Anonymisierer [Sch22] erweitern. Wie dieser Tab aussehen soll, steht zu Beginn der Arbeit noch nicht fest. Die groben Ideen am Anonymisierer selbst, die zu Beginn der Arbeit bekannt waren, sind näher in Abschnitt 3.6 beschrieben.
Nun haben wir den Zustand der GUI und die initialen geplanten Änderungen daran kennengelernt. Gehen wir nun weiter durch die Pipeline und widmen uns als nächstes dem Parser.

3.2 Parser

Der Parser hat die Hauptaufgabe, die eingegebene SQL-Anfrage in EDs – in unserem Fall eine s-t tgd – zu übersetzen. Außerdem erfüllt er, anders als die Bezeichnung „Parser“ suggeriert, noch weitere Aufgaben wie beispielsweise das Auslesen der Original-Datenbank.

Derzeitige Funktionen Der Parser (in seiner Version zum Beginn dieser Arbeit im Oktober 2021) ist ebenfalls im Rahmen der Veranstaltung „Komplexe Softwaresysteme“ entstanden, deren Abschlussbericht unter [KRSZ21] zu finden ist. Er wird zur Zeit dieser Arbeit ebenfalls in einer anderen Bachelorarbeit parallel weiterentwickelt [Kav22]. Diese Komponente nimmt die SQL-Anfrage der Nutzer:in entgegen, generiert daraus einen Syntaxbaum und formt diesen in eine s-t tgd um. Ebenfalls nimmt er alle für die Beantwortung der Anfrage benötigten Tupel aus der Postgres-Datenbank und bildet daraus die Eingabe-Instanz für ChaTEAU. Damit geht diese Komponente eigentlich sprachlich schon weit über die Funktion eines Parsers hinaus. Der Parser kann zu Beginn dieser Arbeit im Oktober 2021 laut Eigenaussage der Entwickler[1] nur bestimmte Arten von SQL-Anfragen in s-t tgds umformen, dazu gehören:

- Projektion auf Attributliste
- Keine abschließende Projektion (SELECT *)
- Natürlicher Verbund
- Innerer Verbund mit eingeschränktem Verbundprädikat
- Kreuzprodukt
- Selektion nach Konstanten
- Vereinigung

[1] [KRSZ21], S.18

Zum Start dieser Arbeit war der Code des Parsers noch gar nicht in ProSA integriert. Dies ist also folglich die erste konkrete Aufgabe dieser Arbeit (zur Umsetzung siehe [4.8.1]).

Geplante Funktionen Zusätzlich sollte der Parser die für die Provenance benötigten Tupel-IDs in die XML einarbeiten. Außerdem wird er in der parallel laufenden Abschlussarbeit [Kav22] noch um weitere SQL-Anfragetypen erweitert, allen voran Aggregatfunktionen und eventuell auch andere Selektionsbedingungen als „Gleichheit“. Für die Unterstützung von Aggregatfunktionen muss die Eingabe für die Chase-Phase um sogenannte *side-tables* erweitert werden, was ebenfalls als Funktionalität des Parsers hinzukommen soll.

3.3 Provenancer

Der Provenancer hat die Aufgabe, die Eingabe für die Chase-Phase so vorzubereiten, dass dabei Provenance-Informationen gesammelt werden können und diese am Ende nach der Backchase-Phase in D_{min} einzuarbeiten. Wie diese Vorbereitung und Einarbeitung konkret ablaufen soll, stand zu Beginn der Arbeit noch nicht fest. Daher haben wir verschiedene Ansätze dafür entwickelt und evaluiert (siehe Abschnitt 4.3.4).

Derzeitige Funktionen Es gibt zu Beginn dieser Arbeit einen Provenancer, der zwar noch lauffähig ist, aber nicht in das Konzept von ProSA passt und daher leider nicht verwendet werden kann. Er wurde zu einem Zeitpunkt entwickelt, als es ChaTEAU noch gar nicht gab und arbeitet über das Umschreiben der SQL-Anfrage selbst, statt mit s-t tgds [AWB16]. Der derzeitige Zustand im Oktober 2021 ist also, dass es für ProSA keinen nutzbaren Provenancer gibt.

Geplante Funktionen Ursprünglich war geplant, mit dem Provenancer direkt vor der Chase-Phase (siehe Abb. 3.1) Tupel-IDs für die Berechnung der Provenanceinformationen hinzuzufügen. Im Laufe der Arbeit ist diese Funktion aber wegen der leichteren Implementierbarkeit in den Parser gewandert (4.8.1). Des Weiteren war ein Plan für den Provenancer, nach der Backchase-Phase in der Pipeline dem Ergebnis-XML der Backchase-Phase und den gesammelten Provenance-Informationen die bezüglich Provenance fehlenden Tupel ausfindig zu machen und hinzuzufügen. Außerdem war hier anfangs geplant, für eine Verarbeitung von Aggregatfunktionen die Chase-Eingabe um sogenannte *side-tables* zu erweitern. Im Laufe der Entwicklung ist diese Funktion aber in den Parser gewandert.
Ob der Provenancer über das Umschreiben der Inversen und doppelte Ausführung des Chase sowie Backchase arbeiten soll oder nur einmal die (Back-)Chase-Phase ausführt und das Ergebnis weiterverarbeitet, war zu Beginn unklar, sondern es wurden wie gesagt verschiedene Verfahren ausprobiert (siehe Kapitel 4.3.4).

3.4 Invertierer

Der Invertierer ist dazu da, die als s-t tgd (oder allgemeiner: EDs) formulierte Anfrage Q so zu Q^{-1} umzuformen, dass mit ihr in der Backchase-Phase die minimale Teildatenbank D_{min} berechnet werden kann.

Derzeitige Funktionen Zurzeit liegt noch keine konkrete Implementierung vor. Der Invertierer wird derzeit in einer parallelen Abschlussarbeit [Spo22] entwickelt.

Geplante Funktionen Der Invertierer soll, entweder als Erweiterung *in* den Parser integriert oder als eigene Komponente *hinter* den Parser geschaltet, die ursprüngliche SQL-Anfrage, die nun durch den Parser als s-t tgd vorliegt, invertieren, damit die Backchase-Phase ausgeführt werden kann. Aus Sicht des Gesamtframeworks interessieren uns vor allem die Schnittstellen und Datenformate, die zwischen den Komponenten ausgetauscht werden. Für die invertierte Anfrage gibt es verschiedene denkbare Ausgabeformate:

- als XML-Datei, die dem Eingabeformat für ChaTEAU entspricht. Diese XML-Datei würde eine leere Instanz, die invertierte Anfrage als s-t tgd und die an dieser s-t tgd beteiligten Relationen im Schema enthalten,
- als Java-Objekt `STTgd`, mit dem die Eingabe für den Backchase direkt im Arbeitsspeicher zusammengesetzt werden kann, ohne den Umweg über die XML-Dateien zu gehen.

Derzeit ist die zweite Variante geplant, um die Zugriffslücke durch das Speichern in einer XML-Datei zu vermeiden, bzw. auch den Aufwand für die (De-)Serialisierung zu umgehen. Im späteren Verlauf der Arbeit [Spo22] stellte sich dann noch heraus, dass die Inverse nicht im Allgemeinen aus genau einer s-t tgd, sondern aus einer Menge von s-t tgds und tgds besteht. Daher haben wir die Schnittstelle entsprechend angepasst (vgl. Abschnitt 4.1).

3.5 ChaTEAU

ChaTEAU[2] ist das Herzstück von ProSA: Mit dieser Komponente ist der Chase-Algorithmus, den wir im Grundlagenkapitel 2.1 kennengelernt haben, implementiert. Konkret wird ChaTEAU in ProSA zweimal verwendet:

1. Beim ersten Aufruf, der *Chase*-Phase, ist die Eingabe die vom Parser ausgelesene Datenbankinstanz, die als s-t tgd umgeformte SQL-Anfrage und die für diese Datenbankinstanz sowie die Ergebnistupel benötigten Relationenschemata.
2. Beim zweiten Aufruf, der *Backchase*-Phase, ist die Eingabe die vom Invertierer invertierte s-t tgd und die durch Provenance-Informationen (d.h. Tupel-IDs und side-tables) angereicherte Ergebnis-XML des ersten ChaTEAU-Durchlaufs.

Derzeitige Funktionen ChaTEAU nimmt Eingaben in einem speziellen XML-Format[3] entgegen, das aus Schema, Abhängigkeiten und Instanz besteht. Dann wird der Chase, wie in Abschnitt [2.1] beschrieben, ausgeführt. ChaTEAU verfügt über unterschiedliche Ein- und Ausgabeschnittstellen: Die Eingabe kann sowohl eine XML-Datei, als auch die Java-Objekte `Instance` als Instanz I und `LinkedHashSet<IntegrityConstraint>` als Abhängigkeiten $\star$. ChaTEAU hat mehrere Zusatzfunktionen, die teilweise über die GUI und vollständig über Kommandozeilenparameter benutzt werden können. Die GUI interessiert uns im Kontext dieser Arbeit ohnehin nicht, da ProSA seine eigene GUI hat und ChaTEAU nur über eine API aufgerufen wird. Die für uns relevanten Zusatzfunktionen sind:

- Terminierungstests: Sie können mit dem Parameter `-t` ausgeführt werden, um wenigstens in einigen Fällen[4] vorhersagen zu können, ob der Chase mit dieser Eingabe terminiert.

[2] [Jur18]
[3] Dafür ist eine XSD definiert, die in Anhang 6.9 zu finden ist.
[4] ChaTEAU kann Terminierungstests u.a. mit den Kriterien *Weak Acyclicity*, *Rich Acyclicity*, und *Safety* durchführen. Auf die Einzelheiten, unter welchen Bedingungen diese Tests (nicht) anschlagen, wollen wir nicht näher eingehen, es sei aber auf [GMS12] verwiesen.

- Provenance-Berechnung kann mit dem Schalter `-p` aktiviert werden. Dies ist für ProSA essentiell. Diese Provenanceinformationen werden derzeit nur im Ausgabe-XML dargestellt.
- Quellschemata können mit dem Schalter `-s` erhalten werden. Normalerweise enthält die Ausgabe-XML von ChaTEAU nur die Schemata, von denen Tupel in der Ergebnisinstanz enthalten sind (d.h. bei uns nur die Relation `Result`). Wir brauchen allerdings für die Backchase-Phase auch die Quellschemata aus der Chase-Phase, welche die neuen Zielschemata im Backchase werden. Einfacher gesagt: Kopf und Rumpf der s-t tgd werden vertauscht, daher müssen auch Quell- und Zielschemata „tauschen“. Damit uns diese Informationen über die ursprünglichen Quellschemata aus der Chase-Phase nicht verloren gehen, können wir diese Funktion benutzen. Bei der Backchase-Phase müssen wir den Schalter `-s` hingegen nicht setzen, weil das Quellschema des Backchase identisch mit dem das Zielschema „`Result`“ des Chase ist, und wir diese Information sowieso schon aus der Parser-Ausgabe kennen.

ChaTEAU unterstützt bereits die Bedienung über eine GUI sowie über die Kommandozeile. Erst kürzlich begann im Rahmen von Hiwi-Arbeiten die Entwicklung einer API für die Benutzung von ChaTEAU über externe Software (sprich: ProSA). Übrigens ist ChaTEAU, anders als der Parser, komplett als eigenes Maven-Projekt gekapselt und somit sauber von ProSA getrennt.

Geplante Funktionen Eigentlich war nicht geplant, ChaTEAU im Rahmen dieser Arbeit noch zu verändern, da es als ausgereift für seine Aufgabe innerhalb ProSAs angesehen wurde. Wir sind allerdings bei der Arbeit schnell auf Änderungswünsche, kleine Fehlfunktionen und weitere Probleme gestoßen:

- Es wäre hilfreich, die Terminierungstests auch über die API ausführen zu können.
- Außerdem sollte auch der Parameter `-s` und `-p` über die API gesetzt werden können.
- Es stellte sich heraus, dass ChaTEAU andere Ergebnisse liefert, wenn die Eingabe die Java-Objekte `Instance` und `LinkedHashSet<IntegrityConstraint>` sind, als wenn die Eingabe eine XML-Datei mit identischem Inhalt ist. Das deutet darauf hin, dass bei der Implementierung des Chase die Homomorphismen über die Referenzen („Pointer“) statt über die Inhalte der Tupel gebildet werden. Diese Referenzen werden vermutlich beim Einlesen im InputReader erzeugt. ChaTEAU sollte also so umgebaut werden, dass es Homomorphismen anhand der Inhalte findet anstatt anhand der Referenzen. Dieser Verbesserungsvorschlag wurde bis heute noch nicht umgesetzt und findet sich daher im Ausblickskapitel 5.1.2 wieder.

Insbesondere ist wichtig, die konzeptuelle Trennung von ProSA und ChaTEAU aufrechtzuerhalten, damit keine Abhängigkeiten zwischen den Komponenten entstehen.

3.6 Anonymisierer

Der Anonymisierer steht am Ende der Pipeline und soll sensible Attribute (oder Attributkombinationen) identifizieren und maskieren.

Derzeitige Funktionen Der Anonymisierer wird während dieser Arbeit parallel in [Sch22] erstmalig entwickelt. Daher liegt uns zu Beginn dieser Arbeit noch kein Konzept dafür oder gar eine Referenzimplementierung vor.

Geplante Funktionen Die Überlegungen der parallelen Arbeit am Anonymisierer sind, dass der Zeitpunkt der Anonymisierung in der Gesamtpipeline für die Korrektheit des Endergebnisses eigentlich egal ist. Aber durch eine Anonymisierung am Anfang wäre der Rechenaufwand größer, weil dann die gesamte Originaldatenbank anonymisiert werden müsste, von der am Ende aber nur ein Teil benötigt wird. Daher wird die in [Sch22] entwickelte Anonymisierung aller Voraussicht nach am Ende der Pipeline stattfinden. Für den Anonymisierungsschritt muss angegeben werden, welche Attribute als sensibel gelten (oder welcher Attribute Kombination, die man *Quasi-Identifikatoren* nennt). Daher wollen wir in dieser Arbeit die GUI um einen weiteren Tab erweitern, in dem diese Quasi-Identifikatoren ausgewählt, der Anonymisierungsschritt gestartet und die anonymisierte minimale Teildatenbank als Ergebnis angezeigt werden können. Die finale Version des Anonymisierer-Tabs ist in Kapitel 4.2 beschrieben, bei der noch weitere Eingabeparameter dazu kamen, wie *l-Diversität*, *k-Anonymität* und *distinct ratio*.

In diesem Kapitel haben wir den Zustand der einzelnen Komponenten zu Beginn dieser Arbeit kennengelernt, sowie sowohl die eigenen vorläufigen Änderungsvorhaben im Oktober 2021 als auch die Pläne anderer Studierender an ihren Komponenten. Natürlich konnte keine Eins-zu-eins-Umsetzung der damals geplanten Änderungen erfolgen, sondern das finale Konzept wurde iterativ im Laufe der Zeit entwickelt und wird im nächsten Kapitel vorgestellt.

4 Konzept und Umsetzung

In diesem Kapitel stellen wir die konzipierte Architektur des Frameworks und deren praktische Umsetzung vor. Natürlich stand das Konzept nicht von Beginn an in Stein gemeißelt, sondern ist über die Zeit in mehreren Iterationen entstanden.
Wir befassen uns zunächst mit der generellen Architektur in der *finalen* Version (im Gegensatz zu Kapitel 3, in dem die *ursprünglich* geplante Architektur vorgestellt wurde). Dann gehen wir, ebenfalls ähnlich wie im letzten Kapitel, erst alle Komponenten grob durch, um dann auf einzelne Aspekte des Frameworks näher einzugehen (Provenancer, Kombinierer und ProSA-Service). Schließlich listen wir noch weitere kleine Änderungen im Rahmen dieser Arbeit auf. Zur Entwicklung wurde ein Beispiel verwendet, dessen Zwischenschritte ein Unit-Test überprüft, den wir danach vorstellen wollen. Dieses Beispiel sehen wir uns dann inhaltlich am Ende dieses Kapitels in Abschnitt 4.10 mit all seinen Zwischenschritten genau an.

4.1 Architektur

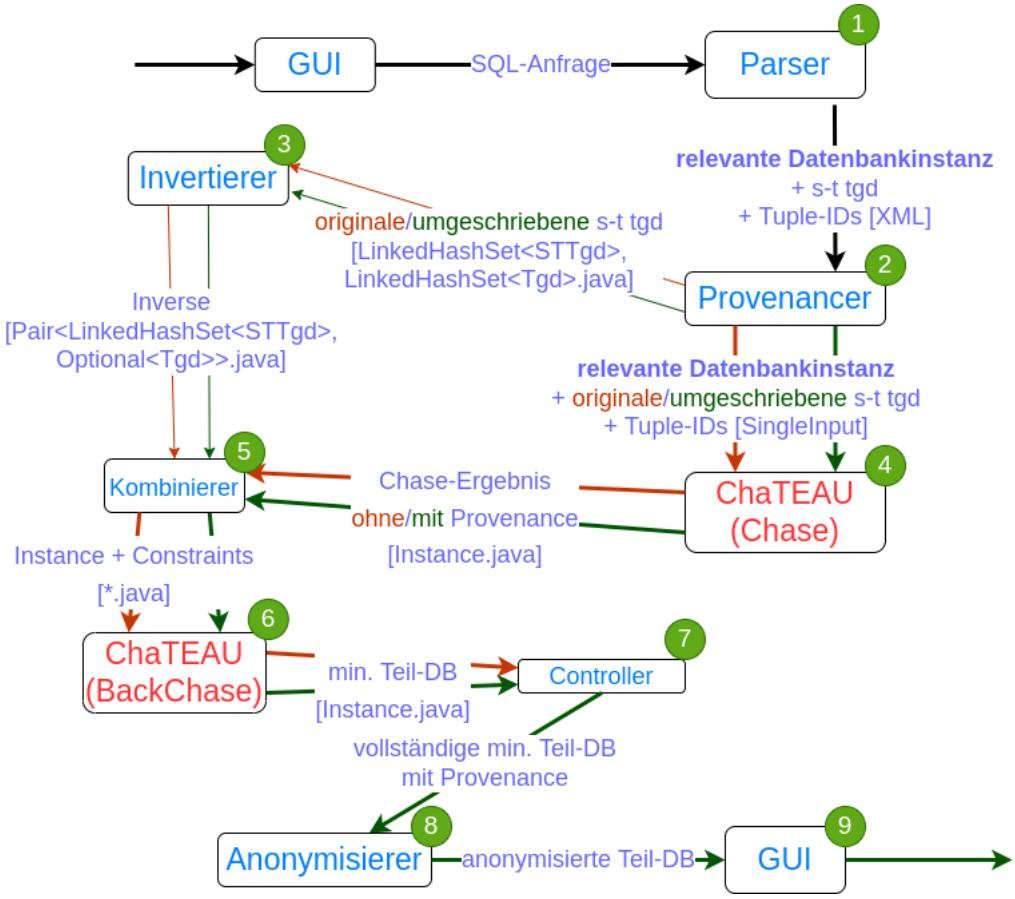


Abbildung 4.1: Architektur von ProSA

In Abbildung 4.1 sehen wir die Architektur von ProSA. Die einzelnen Komponenten sind in blau (bis auf ChaTEAU), die dazwischen übertragenen Daten (d.h. deren Datenstruktur und -typ) in violett dargestellt. Die relevanten Zwischenschritte sind mit grünen Punkten annotiert und werden im Folgenden erläutert. Zuerst aber fällt an der Architektur auf, dass es ab dem Provenancer zwei parallele Stränge in der Pipeline gibt, die jeweils durch die grünen und orangefarbenen Pfeile gekennzeichnet sind. Die „orangefarbene Pipeline“ zeichnet aus, dass sie komplett ohne Einarbeitung von Provenance-Informationen läuft, die grüne Pipeline schreibt die Eingabe so um, dass bei der Verarbeitung keine Provenance-Informationen verloren gehen bzw. zusätzlich erhoben werden (siehe Abschnitt 4.3, in dem diese Umschreibung genauer beschrieben wird). Wer Abbildung 4.1 ganz genau betrachtet, sieht außerdem, dass die Pfeile beim Invertierer etwas dünner sind als die anderen. Damit soll grafisch die „Haupt-Pipeline“ von dem „kleinen Umweg“ über den Invertierer getrennt werden. Der Aufruf des Invertierers ist zwar auch wichtig, aber er ist nur ein kleiner Baustein in der Gesamt-Pipeline, während die Chase-Phase ganz zentral ist.

Sonderfall Aggregatfunktionen Es gibt eine Ausnahme, bei der es keine zusätzliche Ausführung *mit* Provenance gibt: Wenn die eingegebene SQL-Anfrage Aggregatfunktionen enthält (`AVG`, `COUNT`, `SUM`, `MIN` oder `MAX`), so arbeitet der Parser von vornherein die Sammlung von Provenance-Informationen in die XML mit ein. Wir kommen also insgesamt auf **drei Betreibungsmodi** von ProSA:

- SQL-Anfrage enthält keine Aggregate:
  - ***Modus 1***: ohne Provenance
  - ***Modus 2***: mit hinzugefügter Provenance
- SQL-Anfrage enthält Aggregate:
  - ***Modus 3***: immer mit Provenance

Zu den Datentypen, die zwischen den Komponenten übertragen werden, sei angemerkt, dass in der ursprünglichen Version der Architektur (Abb. 3.1) immer XML-Dateien zwischen den Komponenten herumgereicht wurden. Das ist aber sehr ineffizient, da ständig vom Arbeitsspeicher auf die Platte geschrieben und wieder davon gelesen werden müsste (Stichwort „Zugriffslücke“). Stattdessen werden die Daten zwischen den Komponenten jetzt direkt als Java-Objekte übertragen, was außerdem (theoretisch)[1] den Serialisierungsaufwand für die Umformung in XML spart.

Gehen wir nun die Komponenten und ihre Aufgaben im Einzelnen durch. Dabei unterscheiden wir jeweils, ob die eingegebene SQL-Anfrage Aggregatfunktionen enthält oder nicht.

1 Parser Der Parser hat in seiner finalen Version die Aufgabe, SQL-Anfragen in EDs zu übersetzen und die Datenbank auszulesen. Außerdem vergibt er den Tupeln IDs. Sein Ausgabeformat ist ein `ByteArrayOutputStream`, in welchem er eine XML-Datei codiert.

[1] Leider müssen Aufgrund fehlerhafter Funktionen in ChaTEAU, die Objekte in der Praxis trotzdem zum Kopieren serialisiert werden (siehe 4.8.2).

Ohne Aggregatfunktionen Im ersten Schritt verbindet sich der Parser mit der Datenbank und liest alle für die Beantwortung der jeweiligen Anfrage relevanten Tupel aus.[2] Zudem ist seine wohl wichtigste Aufgabe die Umformung der eingegebenen SQL-Anfrage in eine s-t tgd. Um die Einarbeitung von Provenance-Informationen für Modus 2 zu ermöglichen, werden die ausgelesenen Tupel um den Identifikator `tuple_id` ergänzt, wobei der Bezeichner für diesen Identifikator aus dem ursprünglichen Namen der Relation und dem Suffix "`_id`" gebildet wird. Den Wert dieses Identifikators bilden wir genau wie seinen Bezeichner, nur müssen wir "`id`" durch einen fortlaufenden Index ersetzen. Zum Beispiel bekommt das dritte Tupel der Relation `students` die ID mit dem Bezeichner `students_id` und dem Wert `students_3`. Diese ID wird nicht als zusätzliches Attribut in die jeweilige Relation eingefügt, sondern als separates Datum pro Tupel, sodass es bei der Ausführung des Chase keinen Einfluss hat. Die Ausgabe des Parsers ist eine XML-Datei, die der XSD für ChaTEAU-Eingaben (Anhang 6.9) entspricht und direkt ausgeführt werden kann.
Wie solch eine komplette XML-Datei aufgebaut ist, kann anhand der Beispiel-Datei in Listing 6.1 im Anhang nachvollzogen werden: Sie ist grundsätzlich aufgeteilt in zwei Elemente `<instance>` und `<schema>`, wobei `<instance>` die Datenbanktupel als `<atom>`-Elemente enthält. Das `<schema>` wiederum besteht aus den `<relations>` mit den Relationenschemata und den `<dependencies>`, wo eine s-t tgd definiert ist.

Mit Aggregatfunktion Durch die zeitlich parallele Studienarbeit [Kav22] kann der Parser nun auch mit SQL-Anfragen umgehen, die Aggregatfunktionen enthalten. In diesem Fall sind die Abhängigkeiten in der Ausgabe-XML nicht eine einzelne s-t tgd, sondern eine Menge von s-t tgds und eine Menge von tgds.

2 Provenancer Zunächst sieht sich der Provenancer die eingegebene SQL-Anfrage an und erkennt, ob sie Aggregatfunktionen enthält. Dann führt er auf Grundlage dessen die Pipeline in Modus 1 und Modus 2 bzw. in Modus 3 aus:

Ohne Aggregatfunktionen Hier teilt sich die Pipeline in zwei Stränge auf: Der Provenancer liest die vom Parser ausgegebene XML als `SingleInput` ein reicht diesen jeweils zweimal an ChaTEAU und den Invertierer weiter:

- im Originalzustand (Modus 1): der `SingleInput` wird direkt in ChaTEAU eingegeben. Zusätzlich werden die Abhängigkeiten aus dem XML in zwei Mengen von s-t tgds und tgds extrahiert und dem Invertierer übergeben.
- umgeschrieben für *ProSA-Provenance* (Modus 2): hier werden die Tupel-IDs, die vorher als separate Eigenschaft der Tupel definiert waren, als reguläres Attribut in die Tupel integriert: Wenn

  $$R_i : \mathtt{R}(a_1, ..., a_n)$$

  ein Tupel der Relation `R` mit der konkreten ID R_i und den Attributwerten $a_1, ...a_n$ ist, wird daraus

  $$R_i : \mathtt{R}(a_1, ..., a_n, R_i).$$

  Außerdem muss dieses Attribut auch dem Schema hinzugefügt werden und die Abhängigkeiten umgeschrieben werden, sodass die Ergebnistupel die Provenance-Informationen enthalten. Welchen Sinn diese Umformung hat, und inwiefern dadurch Provenance-Informationen gesammelt werden

[2] Es werden nur die Relationen ausgelesen, die auch in der Anfrage vorkommen. In Abbildung 4.1 ist dies deshalb als **relevante** Datenbankinstanz bezeichnet.

können, müssen wir zu diesem Zeitpunkt noch nicht verstehen. Wie ProSA-Provenance genau funktioniert, beschreiben wir erst später in Abschnitt 4.3.

Wichtig ist zu diesem Zeitpunkt, dass wir bei bei Anfragen ohne Aggregate ab hier im Framework **zwei parallele Pipelines** haben, bei denen die eine Pipeline regulär und ohne die Berücksichtigung von Provenance-Informationen für D_{min} arbeitet und die andere mit. In Abbildung 4.1 ist dies durch jeweils zwei Pfeile zwischen den Komponenten dargestellt.

Mit Aggregatfunktionen In Modus 3 wird die Ausgabe des Parsers wie im Fall ohne Aggregate als `SingleInput` eingelesen. Eine Umschreibung der Abhängigkeiten o.Ä. findet hier aber nicht statt. Es werden genau wie im Betrieb ohne Aggregate die Abhängigkeiten (Menge von s-t tgd und Menge von tgds) aus dem XML extrahiert und an den Invertierer überreicht, sowie die XML als `SingleInput` eingelesen und an ChaTEAU für die Chase-Phase übergeben.

3 Invertierer Der Invertierer hat die Aufgabe, die Anfrage Q, welche ihm in Form von s-t tgds und tgds übergeben wird, zu Q^{-1} zu invertieren. Q^{-1} bildet dann im weiteren Verlauf die Abhängigkeit $\star$ für die Backchase-Phase.

Ohne Aggregatfunktionen Der Invertierer bekommt zwei s-t tgds als Eingabe: die originale Version aus dem Parser für Modus 1 und eine modifizierte Version aus dem Provenancer für Modus 2, deren Ausführung zusätzliche Provenance-Informationen liefert. Das für den Invertierer benötigte Eingabeformat ist aber kein XML, bzw. `ByteArrayOutputStream`, wie der Parser liefert, sondern `(LinkedHashSet<STTgd>, LinkedHashSet<Tgd>)`[3]. Daher muss die originale s-t tgd aus dem XML in dieses Format extrahiert werden und die zweite Version ebenfalls in diesem Format vom Provenancer an den Invertierer übergeben werden.

Mit Aggregatfunktionen Hier wird der Invertierer nur einmal aufgerufen, und zwar mit den Abhängigkeiten, wie sie der Parser direkt liefert. Natürlich müssen auch hier wieder die Mengen von s-t tgds und tgds aus dem `ByteArrayOutputStream` extrahiert werden, die der Parser ausgibt. Der Invertierer kann grob auf zwei Arten mit Aggregaten umgehen: er könnte für ein aggregiertes Chase-Ergebnis als Inverse einfach die Identitätsfunktion liefern, was formal immer noch korrekt ist. Dies hätte aber keinen praktischen Nutzen, weil D_{min} mit dieser Inverse nur aus einem einzigen Tupel bestehen würde (dem Ergebnistupel mit dem jeweiligen aggregierten Wert). Stattdessen arbeitet der Invertierer so, wie er in [Spo22] entwickelt wird, in Modus 3 bei uns immer direkt schon Provenanceinformationen ein.

4 ChaTEAU (Chase-Phase) Die Chase-Phase dient dem Zweck, die als Mengen von s-t tgds und tgds dargestellte Abhängigkeit in die Instanz einzuarbeiten, was einer Ausführung der eingegebenen SQL-Anfrage entspricht. Es ist zu erwähnen, dass wir ChaTEAU mit dem „`-s`“-Parameter aufrufen, damit wir die Quellschemata im Ergebnis wiederfinden und später für die Backchase-Eingabe verwendet können (vgl. Abschnitt 3.5).

[3] Aus softwaretechnischer Sicht sollten herumgereichte Datentypen immer so spezifisch wie möglich und so allgemein wie nötig sein, um die Theorie noch korrekt abzubilden. Demzufolge wäre `java.util.Set` also eigentlich der richtige Datentyp, denn der Invertierer gibt zwei ungeordnete Mengen von tgds bzw. s-t tgd zurück und auch der Chase-Algorithmus gibt keine feste Reihenfolge vor, in der die Abhängigkeiten eingearbeitet werden Ebenso legt `Set` keine Iterierungsreihenfolge fest, deshalb würden hier Datenstruktur in Theorie und Praxis perfekt zusammenpassen. Durch die nicht-deterministische Iterierungsreihenfolge über die EDs ist der Chase-Algorithmus aber auch in der reinen Theorie nicht deterministisch, weil durch unterschiedliche Abarbeitungsreihenfolge der Abhängigkeiten unterschiedliche Ergebnisse herauskommen können. Damit dies in der Praxis nicht passiert, sondern man ein deterministisches Ergebnis hat, das getestet werden kann, verwenden wir `LinkedHashSet`, welches immer dieselbe Reihenfolge für Iterierung garantiert.

Ohne Aggregatfunktionen Diese Phase wird zweimal ausgeführt, einmal mit eingearbeiteter ProSA-Provenance, einmal ohne. In Modus 1 (ohne ProSA-Provenance) lassen wir von ChaTEAU direkt die **where-**, **why-** und **how**-Provenance mitberechnen, damit wir diese später in der GUI ausgeben können. In Modus 2 brauchen wir diese drei Provenance-Typen *eigentlich* nicht, weil sie nirgends ausgegeben werden (stattdessen haben wir in diesem Modus ja die ProSA-Provenance). Faktisch lassen wir sie trotzdem zu Debuggingzwecken mitberechnen. Die technische Umsetzung der Chase-Phase von Modus 1 und 2 unterscheidet sich also nicht, sondern nur die Eingabe ist eine andere. Die Ausgabe erfolgt als `Instance`.

Mit Aggregatfunktionen Hier wird ChaTEAU natürlich nur einmal aufgerufen. Der derzeitige Stand ist aber, dass ChaTEAU noch nicht mit EDs umgehen kann, die Funktionen und Vergleiche enthalten, was bei Aggregatfunktionen der Fall ist[4]. Hieran scheitert die Pipeline in Modus 3 also derzeit noch.

5 Kombinierer Egal in welchem Modus wir uns befinden, hier muss jeweils die Inverse aus dem Invertierer mit der passenden Ergebnisinstanz aus der Chase-Phase zu einer gültigen ChaTEAU-Eingabe kombiniert werden.

6 ChaTEAU (Backchase-Phase) In der Backchase-Phase führen wir ChaTEAU mit dem Chase-Ergebnis als Instanz I und der Inversen als Abhängigkeit $\star$ aus. Die technische Ausführung der Backchase-Phase unterscheidet sich von der Chase-Phase nicht, nur die Eingabe ist eine andere. Der einzige Unterschied ist vielleicht, dass wir in der Backchase-Phase weder **where-**, noch **why-** oder **how**-Provenance berechnen und auch keine Quellschemata mit ins Ergebnis übernehmen.

Ohne Aggregatfunktionen In der Backchase-Phase wird (jeweils) die Inverse in die Ergebnisinstanz der Chase-Phase eingearbeitet, sodass wir als Ergebnis die minimale Teildatenbank D_{min} (Modus 1) bzw. D_{min_prov} erhalten (Modus 2). Bei der Variante mit Provenance (Modus 2) enthält D_{min_prov} dann auch die Tupel-IDs, die der Provenancer in Schritt 2 hinzugefügt hat. Die konkrete Umsetzung der Backchase-Phase mitsamt Kombinierer schauen wir uns noch genauer in Abschnitt 4.4 an.

Mit Aggregatfunktionen Genau wie in der Chase-Phase kann ChaTEAU hier nicht mit den eingegebenen Abhängigkeiten umgehen und daher funktioniert dieser Schritt in Modus 3 noch nicht.

7 Controller Obwohl der Controller nach dem Backchase keine „richtig eigenständige Komponente“ ist, erfüllt er ein elementare Aufgabe: Er überprüft, ob die jeweils berechnete minimale Teildatenbank D_{min} oder D_{min_prov} korrekt bezüglich Reproduzierbarkeit des Anfrageergebnisses R ist. Das ist wichtig, weil nur dann die berechnete Teildatenbank überhaupt den Anforderungen an Nachvollziehbarkeit genügt und für das Forschungsergebnis R mitveröffentlicht werden kann. Dazu berechnet der Controller $Q(D_{min})$ und vergleicht das Ergebnis mit dem Ergebnis R aus der Chase-Phase. Wenn dieser Check fehlschlägt, wird eine Fehlermeldung sowohl geloggt als auch in das Textfeld für das Backchase-Ergebnis geschrieben. Diese Überprüfung wird in allen drei Modi durchgeführt. Die Umsetzung des Checks ist näher in Abschnitt 4.6 beschrieben. Außerdem erfüllt der Controller noch diverse Verwaltungsaufgaben, wie etwa die Ausgabe von Ergebnissen in der GUI.

[4] [Aug22], [Kav22]

Ohne Aggregatfunktionen Hier werden lediglich die zwei berechneten Versionen der minimalen Teildatenbank D_{min} und D_{min_prov} in der GUI im Backchase-Tab (siehe Abschnitt 4.2) ausgegeben. Dann wird nur D_{min_prov} an den Anonymisierer zur Weiterverarbeitung weitergereicht.

Ohne Aggregatfunktionen In Modus 3 wird im Backchase-Tab bei beiden Textfeldern für das Backchase-Ergebnis mit/ohne Provenance dasselbe eingetragen.

8 Anonymisierer Der Anonymisierer soll sensible Attributwerte aus D_{min_prov} aus Gründen der Privatsphäre oder beispielsweise wegen finanzieller Interessen, sofern es sich bei D um Firmendaten handelt, maskieren.

Ohne Aggregatfunktionen Hier sollen nur die minimale Teildatenbank mit Provenance, sowie gewisse Anonymisierungsparameter, wie *k-Anonymität*, *l-Diversität*, eine *distinct ratio* und *Quasi-Identifikatoren* als Eingaben für den Anonymisierungsalgorithmus übergeben werden, der parallel in [Sch22] entwickelt wird. In der vorliegenden Arbeit haben wir nur die GUI entwickelt, die diese Parameter entgegennimmt (siehe Abschnitt 4.2).

Mit Aggregatfunktionen Hier gibt es ohnehin nur eine einzige minimale Teildatenbank, die der Anonymisierer verarbeiten kann, ansonsten ist das Vorgehen identisch zu Modus 1 und Modus 2

9 GUI Sowohl in Modus 1, 2 und 3 werden alle durch die Ausführung der Pipeline(s) erhaltenen Daten in der GUI dargestellt.

Der aktuellen Version der GUI widmen wir uns im folgenden Abschnitt im Detail.

4.2 Vorstellung der finalen GUI

Betrachten wir nun die aktuelle Version der GUI und stellen Unterschiede zu der alten Version dar, die in Kapitel 3 präsentiert wurde:

DB-Configuration-Tab

Der Tab für die Datenbank-Konfiguration hat sich gegenüber der Ausgangsversion nicht geändert.

Chase-Tab

In Abbildung 4.2 ist der aktuelle Chase-Tab zu sehen. Gegenüber der alten Version sehen wir, dass die Standard-Buttons („`Run ProSA`“) deutlicher in blau hervorgehoben sind, was die Nutzer:in anleiten soll, wo der nächste Schritt ist, um dem Haupt-Usecase zu folgen. Die SQL-Query kann nach wie vor in dem Textfeld links oben eingegeben werden, aber von der „`Original database`“ werden jetzt das Schema und die eigentlichen Tupel in unterschiedlichen Sub-Tabs ausgegeben und außerdem die vom Parser hinzugefügten Tupel-IDs als Suffix angefügt.
Die Kontrollkästchen für die Auswahl, welche Provenancetypen in der Chase-Phase berechnet werden sollen, wurden entfernt, denn wir haben uns dazu entschieden, standardmäßig alle Typen in Modus 1 sowie zusätzlich die ProSA-Provenance[5] in Modus 2 zu berechnen und im Backchase-Tab auszugeben. Das in [FHO+21] erarbeitete Konzept, die Nutzereingaben und Ausgaben des Programms bei allen Tabs ordentlich auf der linken und rechten Seite zu trennen, haben wir durchgehend. So findet sich auf der rechten Seite wie vom Parser ausgegebene s-t tgd (bzw. noch weitere Abhängigkeiten, im Fall von Aggregatfunktionen) und die Ausgabeinstanz der Chase-Phase wird von beiden Pipelines in das rechte untere Textfeld ausgegeben. Die Ausgabe der Modus-1-Pipeline kommt in das Feld „`Without Provenance`“ und das Ergebnis der Modus-2-Pipeline das Feld „`With ProSA Provenance`“.

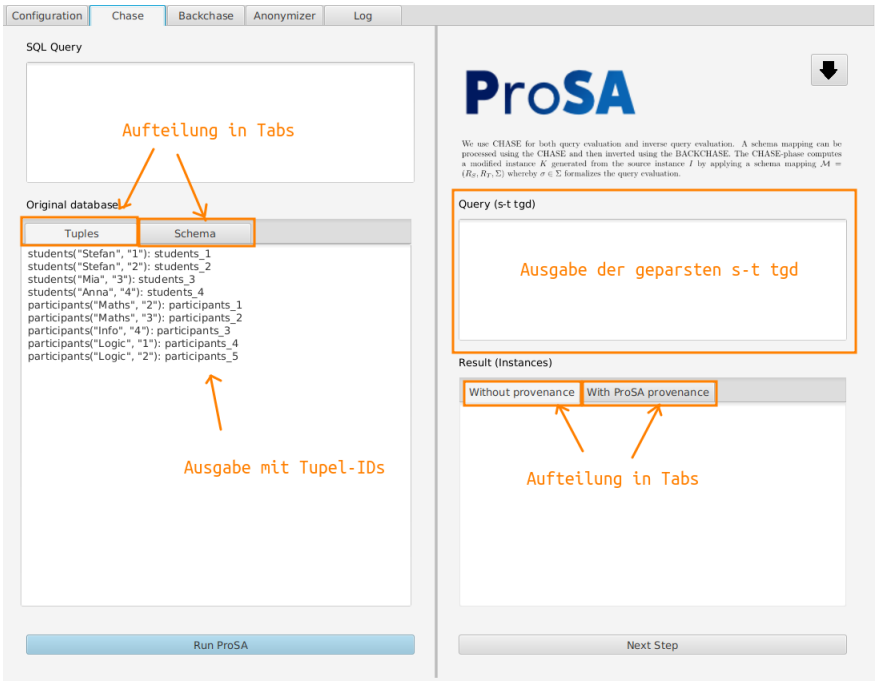


Abbildung 4.2: Der aktuelle Chase-Tab. Änderungen sind farblich vorgehoben.

[5] Wie ProSA-Provenance funktioniert, wird noch in Abschnitt 4.3 beschrieben.

Backchase-Tab

In Abbildung 4.3 sehen wir die aktuelle Version des Backchase-Tabs. Gegenüber der alten Version ist ein Label für den Inversentyp hinzugekommen, der im Rahmen dieser Arbeit zwar noch nicht berechnet wird, aber für die Zukunft geplant ist. Die „`Original database`“ wird genau wie auf dem Chase-Tab auch getrennt nach Schema und Tupeln ausgegeben. Das Provenance-Ergebnis wird ebenfalls in verschiedenen Sub-Tabs ausgegeben, anstatt alle untereinander im selben Textfeld. Auch für die minimale Teildatenbank gibt es zwei Tabs: einen für das Backchase-Ergebnis von Modus 1 („`Without Provenance`“) und einen für das Backchase-Ergebnis von Modus 2 („`With ProSA Provenance`“).

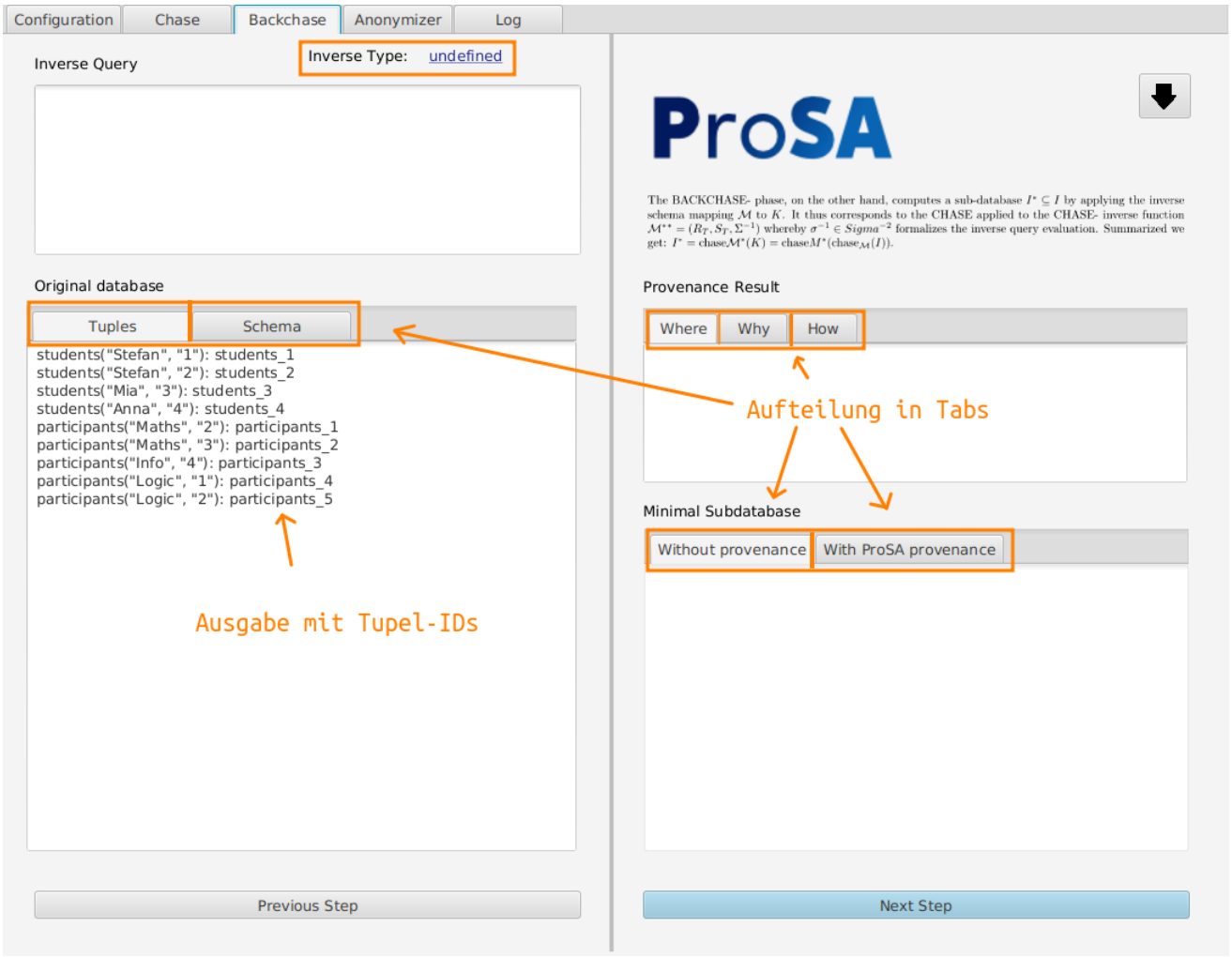


Abbildung 4.3: Der aktuelle Backchase-Tab. Änderungen sind farblich hervorgehoben.

Anonymisierer-Tab

Der Tab für den Anonymisierer haben wir komplett neu hinzugefügt. Auf der linken Seite werden zunächst noch einmal die originale Datenbank und die minimale Teildatenbank (mit ProSA-Provenance) dargestellt. Die GUI wurde in Absprache mit dem Autor der Arbeit [Sch22] entwickelt, besteht aber derzeit nur aus den leeren GUI-Elementen ohne Funktion. Die GUI soll in Zukunft folgendermaßen genutzt werden:
Unten links können die Eingabeparameter für den Anonymisierungsschritt eingegeben werden: dies sind die positiven Ganzzahlen für die k-Anonymität und l-Diversität, sowie eine Zahl zwischen 0 und 1 für die Distinct-Ratio. Als Quasi-Identifikatoren bezeichnet man eine Menge von Attributen, die nicht einzeln, aber in Kombination ein Tupel eindeutig identifizierbar machen. Beispielsweise wäre der Nachname nicht eindeutig, aber die Kombination aus Nachname und Geburtsdatum schon. Die Quasi-Identifikatoren können in einer Konfigurationsdatei definiert sein, die man mit dem Button „`Select Configuration File`“ auswählen kann. In dieser Datei kann zusätzlich noch eine Gewichtung definiert sein, die beschreibt, welche Attribute im Anonymisierungsalgorithmus zuerst anonymisiert werden sollen, und welche erst später. Alternativ können die Quasi-Identifikatoren auch automatisch generiert werden, oder manuell in der Liste links unter Halten der STRG- bzw. Command-Taste mehrere Attribute ausgewählt und dann durch den Button „`Add > >`“ hinzugefügt werden. Schließlich wird der Anonymisierungsalgorithmus durch Drücken des Knopfes „`Run Anonymizer`“ gestartet und das Ergebnis im Textfeld rechts ausgegeben.

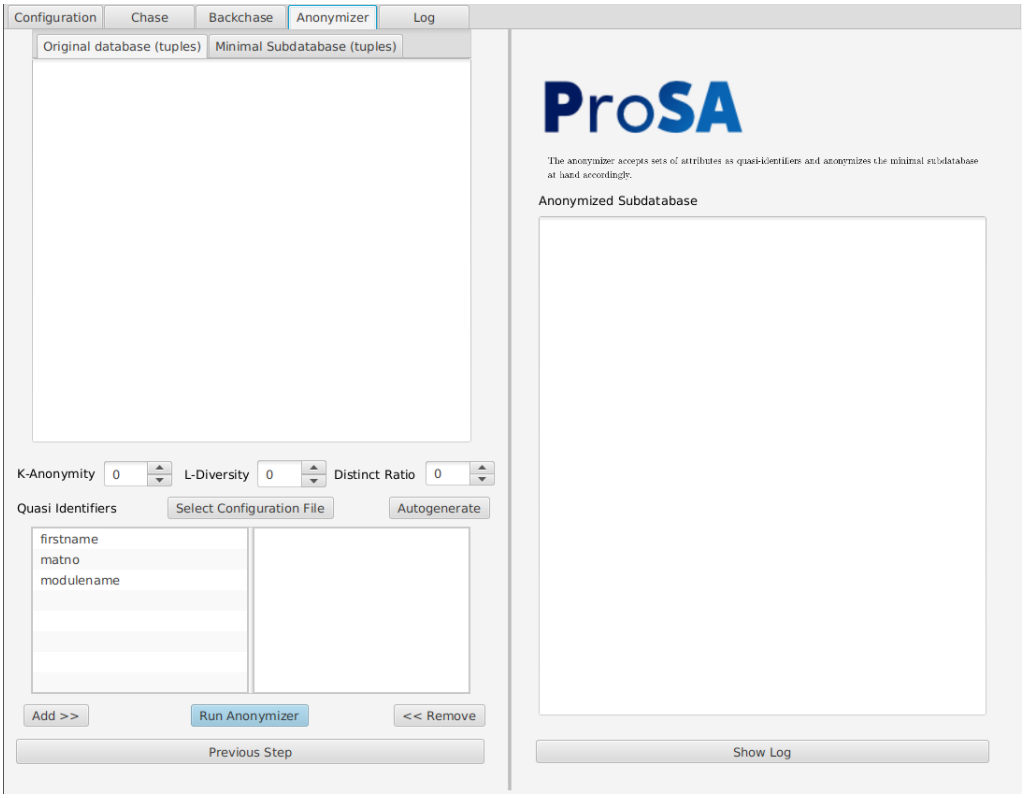


Abbildung 4.4: Der aktuelle Anonymizer-Tab

Log-Tab

Genau wie beim DB-Configuration-Tab hat sich das Layout des Log-Tabs ebenfalls nicht gegenüber der Ursprungs-Version geändert. Es ist allerdings die Funktionalität dazugekommen, dass die Ausgabe des ChaTEAU-Loggers jetzt korrekt im unteren Textfeld angezeigt wird, was vorher nicht ging.

4.3 Provenancer und ProSA-Provenance

Beschäftigen wir uns nun damit, wie genau wir die Einarbeitung von Provenance-Informationen in ProSA konzipiert und umgesetzt haben: Wir wissen bereits, dass bei der Ausführung von SQL-Anfragen Informationen über die Herkunft der Daten verloren gehen. Ein Lösungsansatz sind die bekannten drei Provenancetypen **where**, **why** und **how**[6], die ChaTEAU direkt mitberechnen kann. Aber sogar bei der aussagekräftigsten how-Provenance gehen Informationen darüber verloren, welches Tupel auf *welchen Teil* des s-t tgd-Rumpfes „matcht“ (vgl. Abschnitt 4.3.4). Stattdessen haben wir ein eigene Form der Provenance entwickelt, die wir *ProSA-Provenance* nennen und nun erst formal und dann an einem Beispiel erläutern.

4.3.1 Formales Vorgehen

Zum Erfassen von ProSA-Provenance nehmen wir die Tupel-IDs aus der Instanz, die vorher als separate Eigenschaft der Tupel definiert waren, und fügen sie in diese Tupel als reguläres zusätzliches Attribut ein: Wenn

$$R\_id_i : \mathtt{R}(a_1, ..., a_n)$$

ein konkretes Tupel der Relation `R` mit der konkreten ID R_id_i und den Attributwerten $a_1, ..., a_n$ ist, wird daraus

$$R\_id_i : \mathtt{R}(a_1, ..., a_n, R\_id_i)$$

Außerdem muss dieses Attribut `R_id` analog auch dem Quellschema der Relation `R` hinzugefügt werden. Zum Schluss modifizieren wir die s-t tgd so, dass das Ergebnistupel alle IDs der Tupel, die im Rumpf „gematcht“ haben, als hinten angefügte Attribute enthält:

Nehmen wir eine allgemeine s-t tgd, deren Rumpf Atome aus j Relationen $\mathtt{R}_1 ... \mathtt{R}_j$ mit jeweils j_k Attributen $a_{j_1}, ..., a_{j_k}$ enthält, und deren Kopf ein Tupel der Relation $\mathtt{Result}(r_1, ..., r_m)$ erzeugt:

$$\mathtt{R}_1(a_{1_1}, a_{1_2}, ..., a_{1_k}), ..., \mathtt{R}_j(a_{j_1}, a_{j_2}, ..., a_{j_k}) \\ \rightarrow \mathtt{Result}(r_1, ..., r_m)$$

Bevor wir uns der Umschreibung der s-t tgd widmen, sollten wir uns folgenden Unterschied zwischen dem Attribut R_id und dem Attibut*wert* R_id_i Um alle an der Entstehung des Ergebnistupels beteiligten Tupel-IDs mitzuspeichern, fügen wir die Attribute für die IDs $R\_id_1, ..., R\_id_j$ wie bereits bei der Instanz *jeweils* den zugehörigen Atomen im Rumpf hinzu und *alle* IDs dem `Result`-Atom im Kopf:

$$\mathtt{R}_1(a_{1_1}, a_{1_2}, ..., a_{1_k}, R\_id_1), ..., \mathtt{R}_j(a_{j_1}, a_{j_2}, ..., a_{j_k}, R\_id_j) \\ \rightarrow \mathtt{Result}(r_1, ..., r_m, R\_id_1, ..., R\_id_j)$$

Diese Darstellung, Provenance-Informationen quasi als Suffix in den Ergebnistupeln in Form zusätzlicher Attribute anzugeben, erinnert an die Darstellung, die auch im Provenance-Tool *Perm* verwendet wird (vgl. [SRH+20]). Zu guter Letzt passt die Signatur des Atoms im Kopf jetzt natürlich nicht mehr auf das Zielschema von `Result`, da dort die zusätzlichen Attribute für die IDs fehlen. Daher fügen wir diese Attribute auch im Schema hinzu. Nun wissen wir für jedes Tupel im Chase-Ergebnis, welche Originaltupel jeweils auf welches Atom im Rumpf „gematcht“ haben. Der Unterschied zu klassischer why- und auch how-Provenance ist, wie wie auch im folgenden Beispiel noch sehen werden, dass bei der why- und how-Provenance für ein Ergebnistupel die Informationen über die Reihenfolge verloren gehen, in der die Quelltupel auf die einzelnen Atome des Rumpfes gematcht haben. Aber wenden wir die ProSA-Provenance erst einmal an einem Beispiel an, um sie noch besser zu verstehen.

[6] vgl. [CCT09]

4.3.2 Einarbeitung von ProSA-Provenance am Beispiel

Betrachten als Beispiel für dieses Verfahren folgende SQL-Anfrage auf unsere Beispieldatenbank:

```
SELECT firstname
FROM students NATURAL JOIN participants
WHERE modulename = 'Logic';
```

Die zugehörige s-t tgd mit eingearbeiteter ProSA-Provenance sieht so aus:

$$\forall firstname, matno, students\_id, participants\_id :$$
$$\texttt{students}(firstname, matno, students\_id) \wedge \texttt{participants}(\texttt{'Logic'}, matno, participants\_id)$$
$$\rightarrow \texttt{Result}(firstname, students\_id, participants\_id)$$

Würden wir diese s-t tgd ausführen, sind die Ergebnistupel

```
Result('Stefan', 'students_1', 'participants_4')
Result('Stefan', 'students_2', 'participants_5').
```

Wir haben nun ein Verfahren kennengelernt, mit dem wir die Ausgabe des Parsers so umschreiben können, dass durch die Ausführung der Chase-Phase keine Provenance-Informationen verloren gehen. Schauen wir uns an, wie wir dieses Verfahren implementiert haben.

4.3.3 Implementierung des Provenancers

Betrachten wir die relevanten Code-Ausschnitte um nachzuvollziehen, wie der Provenancer konkret implementiert ist. Zunächst sehen wir in Listing 4.1, dass durch erneutes Einlesen des Parser-Output-Streams (Zeile 7) ein kopiertes `SingleInput`-Objekt angelegt wird, mit dem die zweite Pipeline nebenläufig und unabhängig von der ersten ausgeführt werden kann.

Listing 4.1: Aufruf des Provenancers mit einer kopierten Eingabe. (Klasse `ProSAService`)

```
1 public WithRewrittenInput rewriteInputForProvenance() {
2    /* get a SingleInput copy by reading the byte stream again, since
3    * the constructor {@link #new(Instance instance)} does not create
4    * an actual copy, but keeps references between copies.
5    */
6    InputReader reader = new InputReader();
7    SingleInput inputCopy = reader.readStream(this.parsedStream);
8    SingleInput rewrittenInput = Provenancer.provenanceChaseInput(inputCopy);
9    ...
```

Dann rufen wir in Zeile 8 die Methode `provenanceChaseInput(...)` in der Klasse `Provenancer` auf, die wir uns nun in Listing 4.2 genauer ansehen: Zunächst werden in Zeile 9 und 10 die Attribute für die Tupel-IDs dem Schema sowie die eigentlichen Tupel-IDs als Attributwerte den Instanz-Tupeln hinzugefügt, wie in Abschnitt 4.3.1 beschrieben.
Dann werden die Abhängigkeiten in der Methode `provenanceConstraints(...)` ab Zeile 46 umgeschrieben. Das Verfahren arbeitet nur korrekt, wenn die Abhängigkeiten aus exakt einer s-t tgd bestehen, weil man bei mehreren s-t tgds oder tgds nicht sicher sein kann, dass sie nicht untereinander in Abhängigkeit stehen (beispielsweise, wenn die neu erzeugten Tupel einer tgd auf den Rumpf einer zweiten s-t tgd matchen sollen). Außerdem hatten wir uns als Zielsetzung die Einschränkung gegeben, erst einmal nur eine Anfrage mit genau einer s-t tgd verarbeiten zu können. Im weiteren Arbeiten kann darauf aufgebaut

und das Konzept der ProSA-Provenance erweitert werden. Wegen dieser Einschränkung wird zunächst in Zeile 48-61 sichergestellt, dass die `constraints` genau in diesem richtigen Format sind. Dann gehen wir einzeln die Atome des Rumpfes durch, und fügen dort die Tupel-ID als allquantifizierte Variable als letzten Term hinzu (Zeile 68) und dieselbe Variable auch bei den Termen des `Result`-Atoms (Zeile 69). Dabei setzen wir als Index dieser Variable jedes Mal einen eindeutigen Wert `currentID`, der jedes Mal, wenn ein Atom derselben Relation im Rumpf auftaucht, hochgezählt wird. Der Grund dafür ist, dass im Rumpf auch mehrere Atome derselben Relation auftauchen können, deren Tupel-IDs aber unabhängig voneinander sein sollen.

Listing 4.2: Umschreiben „in-place“. (Klasse `Provenancer`)

```
1  /**
2   * Rewrites the given ChaTEAU input to collect ProSA provenance information.
3   * Does NOT create a copy of given {@link #SingleInput}, but modifies the actual
        given object.
4   * @param inputCopy
5   * @return
6   */
7  public static SingleInput provenanceChaseInput(SingleInput inputCopy) {
8
9      addTupleIDAttributeToSchema(inputCopy.getInstance().getSchema());
10     addTupleIDsToInstanceTuples(inputCopy.getInstance());
11     provenanceConstraints(inputCopy.getConstraints());
12
13     return inputCopy;
14 }
15
16 private static void addTupleIDAttributeToSchema(Map<String, LinkedHashMap<String,
       String>> schema) {
17
18     schema.forEach((relationName, attributeToTypeMap) -> {
19         attributeToTypeMap.put(relationName + "_id", "string");
20     });
21
22 }
23
24
25 private static void addTupleIDsToInstanceTuples(Instance instance) {
26     instance.getRelationalAtoms().forEach(atom -> {
27         atom.getTerms().add(Constant.fromString(atom.getRelationName() + "_id", atom.
               getProvInfo().getId()));
28     });
29 }
30
31 /**
32  * Rewrites the sttgd query so that for each result tuple, all source tuple ids
33  * will be stored, that have contributed in creating that result tuple. Unlike
34  * how-provenance, this provenance is conscious of the order in which tuples have
35  * matched on the source atoms, e.g. for the sttgd
36  * "Rel('Mary'), Rel('Freddy') -> Result('Found Mary and Freddy')" it is stored,
37  * which source tuple id matched on the left atom (i.e. whose attribute value
38  * was 'Mary') and which source tuple id matched on the right atom (i.e. its
39  * attribute value was 'Freddy'). In order to do so, we add the tuple id of
40  * the source tuples as normal attributes and save them for the result tuple:
41  *  "Rel('Mary', #V_Rel_id_1), Rel('Freddy', #V_Rel_id_2)
42  * -> Result('Found Mary and Freddy', #V_Rel_id_1, #V_Rel_id_2)"
43  *
44  * @param constraints the constraints to add provenance to
45  */
46 private static void provenanceConstraints(LinkedHashSet<IntegrityConstraint>
```

```
        constraints) {
47
48      if (constraints.size() != 1) {
49          throw new RuntimeException("More than one constraint.");
50      }
51      IntegrityConstraint dep = constraints.iterator().next();
52      if (!(dep instanceof STTgd)) {
53          throw new RuntimeException("Constraint must be an s-t tgd.");
54      }
55      STTgd stTgd = (STTgd) dep;
56      final Map<String, Integer> idCounter = new HashMap<>();
57
58      if (stTgd.getHead().size() != 1) {
59          throw new RuntimeException("STTGD head must have exactly one atom!");
60
61      }
62      RelationalAtom resultAtom = stTgd.getHead().iterator().next();
63      stTgd.getBody().forEach(atom -> {
64
65          int currentID = idCounter.getOrDefault(atom.getName(), 0);
66          Variable id = new Variable(VariableType.FOR_ALL, atom.getName() + "_id",
67          ++currentID);
68          atom.getTerms().add(id);
69          resultAtom.getTerms().add(id.copy());
70          idCounter.put(atom.getName(), currentID);
71      });
72  }
```

Nun haben wir dargestellt, wie der Provenancer in der finalen Version konzipiert und umgesetzt wurde. Es gab aber im Verlauf dieser Arbeit noch andere Ideen dafür, die sich nicht durchgesetzt haben und die wir im folgenden Abschnitt diskutieren wollen.

4.3.4 Evolution des Provenancers

Ursprünglich war die Idee, die durch die Chase-Phase gesammelten Informationen über die where-, why- und how-Provenance der Ergebnistupel nach der Backchase-Phase in die minimale Teildatenbank *im Nachhinein* wieder einzuarbeiten (vgl. Abb. 4.1 und Abschnitt 3.3).

Einarbeitung von where-Provenance In einer ersten Implementierung[7], von der Ausschnitte in Listing 4.3 zu sehen ist, haben wir zunächst nur die where-Provenance betrachtet. Diese haben wir eingearbeitet, indem wir zu jedem Ergebnistupel der Backchase-Phase die Zeugenliste von Tupel-IDs extrahiert haben (Zeile 11). Zu diesen Tupel-IDs haben wir die zugehörigen *kompletten* Originaltupel aus der Originaldatenbank herausgefiltert (Zeile 14-17). Diese Menge von Tupeln genügt also der where-Provenance des Chase-Ergebnisses und wir könnten dieses Ergebnis direkt ausgeben, dann hätten wir aber keine Information darüber, welche zusätzlichen Informationen diese Einarbeitung der Provenance gegenüber der ursprünglichen minimalen Teildatenbank geliefert hat. Um diesen Unterschied bzw. diesen Informationsgewinn darzustellen, erfassen wir die Differenz dieser zwei Mengen. Dies geschieht in der Implementierung durch das Erfassen und Löschen des Schnitts dieser Mengen im Aufruf der Methode `removeMatchingTuple(...)` in Zeile 20. Wir iterieren also über die Tupel des Backchase-Ergebnisses und suchen in den insgesamt erforderlichen `requiredTuples` ein Tupel mit einem Homomorphismus. Wenn wir eines finden, wird dieses entfernt. Wir können nicht `.equals(...)` o.Ä. für diesen Abgleich verwenden,

[7] Vorgestellt in der Zwischenpräsentation dieser Arbeit [Han22], implementiert mit Commit `431fa1561ef29da99e700e87f4e2c15ab3d52356`

sondern müssen Homomorphismen dafür benutzen, weil die Tupel im Backchase-Ergebnis meist Nullwerte enthalten, während die Tupel aus der Original-Datenbank dort konkrete Werte haben. So bleibt am Ende von den für die where-Provenance benötigten `requiredTuples` nur noch die Differenz übrig, die wir noch nicht im Backchase-Ergebnis haben und wir können diese ausgeben.

Listing 4.3: Erste Version des Einarbeitens von Provenance-Informationen unter Nutzung der where-Provenance. (Klasse `MainGUIController`, Commit: `431fa1561ef29da99e700e87f4e2c15ab3d52356`)

```
1  private Pair<LinkedHashSet<RelationalAtom>, LinkedHashSet<RelationalAtom>>
       combineWithProvenance(
2          LinkedHashMap<String, RelationalAtom> originalDB, Instance backchaseResult,
               Instance instance) {
3      LinkedHashSet<RelationalAtom> tuplesFromBackChase = new LinkedHashSet<>();
4      backchaseResult.getRelationalAtoms().forEach(atom -> {
5          // collect all tuples we have in backChase result:
6          tuplesFromBackChase.add(atom.copy());
7      });
8      LinkedHashSet<String> requiredTupleIDs = new LinkedHashSet<>();
9      instance.getRelationalAtomsBySchema("Result").forEach(atom -> {
10         // collect all tuple IDs mentioned in where tuple provenance:
11         requiredTupleIDs.addAll(atom.getProvInfo().getWhereTupleProvenance());
12     });
13     LinkedHashSet<RelationalAtom> requiredTuples = new LinkedHashSet<>();
14     requiredTupleIDs.forEach(id -> {
15         // collect the actual required tuples:
16         requiredTuples.add(originalDB.get(id).copy());
17     });
18
19     tuplesFromBackChase.forEach(atom -> {
20         removeMatchingTuple(atom, requiredTuples);
21     });
22     logger.log(Level.INFO, "Remaining unmatched tuples: ");
23     requiredTuples.forEach(atom -> logger.log(Level.INFO, atom));
24
25     return Pair.of(tuplesFromBackChase, requiredTuples);
26 }
27
28 private void removeMatchingTuple(RelationalAtom actualAtom, LinkedHashSet<
       RelationalAtom> requiredTuples) {
29     for (RelationalAtom requiredAtom : requiredTuples) {
30         if (actualAtom.hasHomomorphismTo(requiredAtom)) {
31             logger.log(Level.INFO, "Match between actual [" + actualAtom + "] and
                   required [" + requiredAtom + "]");
32             requiredTuples.remove(requiredAtom);
33             break;
34         }
35     }
```

Dieses Verfahren weist den Mangel auf, dass es nur die where-Provenance einarbeitet, die von allen Provenance-Typen die am wenigsten aussagekräftige ist. Demzufolge begannen wir mit der Arbeit, dieses Verfahren zunächst auf die why- und dann auch auf die how-Provenance auszuweiten, stellten dann aber fest, dass sogar eine Einarbeitung der how-Provenance im Nachhinein mit Informationsverlust einhergeht:

How-Provenance reicht nicht Obwohl die how-Provenance von allen Provenance-Typen nach [Aug17] die aussagekräftigste ist, geht auch damit Information über die Reihenfolge verloren, in die Tupel in die Be-

rechnung der Anfrage eingegangen sind. Betrachten wir zur Veranschaulichung die Anfrage

$$\forall matno_1, matno_2 : \texttt{students}(\texttt{'Anna'}, matno_1) \wedge \texttt{students}(\texttt{'Mia'}, matno_2) \\ \rightarrow \texttt{Result}(\texttt{'Anna und Mia gefunden.'}).$$

Die how-Provenance dieser Anfrage liefert das Polynom mit den Tupel-IDs (`students_3` + `students_4`), anhand dessen wir wissen, dass die Tupel `students_3` und `students_4` für dieses Ergebnistupel verantwortlich sind. Aber wegen der Kommutativität (sowohl für „+“ als auch für „·“) können wir nicht herleiten, welches Tupel dieser beiden letztendlich auf das linke Atom $\texttt{students}(\texttt{'Anna'}, matno_1)$, und welches auf das rechte Atom $\texttt{students}(\texttt{'Mia'}, matno_2)$ gematcht hat.[8] Die how-Provenance ist also für unsere Zwecke nicht gut geeignet.

Position des Provenancers in der Pipeline Die bisher vorgestellten Ansätze haben gemeinsam, die Einarbeitung der in der Chase-Phase gesammelten where-, why- oder how-Provenance *erst im Nachhinein* nach der Backchase-Phase vorzunehmen. Angesichts des Nachteils, dass diese Ansätze sowieso Informationsverlust mitbringen (s.o.), haben wir uns dafür entschieden, die Anfragen stattdessen bereits *vor der Chase-Phase* so umzuschreiben, dass auch die Information über die Reihenfolge der für ein Ergebnistupel „verantwortlichen“ Quelltupel erhalten bleibt. Diese Art der Provenance nennen wir *ProSA-Provenance* und das Verfahren wurde bereits in Abschnitt 4.3.1 erläutert. Die where-, why- und how-Provenance wird natürlich nach wie vor während der Chase-Phase in Modus 1 gesammelt und in der GUI ausgegeben, aber zur Berechnung von D_{min_prov} verwenden wir sie nicht.

Reduzieren von Provenanceinformationen nach der Chase-Phase oder zwei separate Pipelines? Zunächst sind wir also so verfahren, die Anfrage vor der Chase-Phase für die ProSA-Provenance umzuschreiben (4.3) und dann nach der Chase-Phase die zusätzlichen Attribute für die ProSA-Provenance wieder zu entfernen. Zur Erinnerung: diese Informationen liegen als Suffix eines Ergebnis-Tupels vor. Dieser Suffix besteht aus einer geordneten Auflistung jeder Tupel-IDs, die auf den Rumpf der s-t tgd gematcht haben. Wenn wir uns das Beispiel aus Abschnitt 4.3.2 noch einmal anschauen, stellen wir fest, dass das Ergebnis gegenüber den Original-Ergebnis eigentlich Duplikate enthält:

```
Result('Stefan', 'students_1', 'participants_4')
Result('Stefan', 'students_2', 'participants_5').
```

Um ProSA-Provenance zwar zu sammeln, aber Duplikate nicht am Ende im Ergebnis zu haben, müssen wir die zusätzlichen Attribute für die Tupel-IDs nach der Chase-Phase wieder „wegprojizieren“. Die Implementierung dafür ist in Listing 4.4 zu sehen. Darin werden zunächst die Instanz und das Schema aus dem Chase-Ergebnis kopiert und dann in den Instanztupeln alle Attribute gelöscht, die auf „`_id`“ enden (ausgelagerte Methode Zeile 29). Außerdem werden natürlich im kopierten Schema die Attribute für die Tupel-IDs auf dieselbe weise entfernt (ausgelagerte Methode Zeile 23). Nun haben die Tupel und das Schema zwar keine Tupel-IDs mehr, es gibt aber immer noch potenzielle Duplikate. Wie auch im Kommentar (Zeile 7-10) angemerkt, *sollte* die Datenstruktur `Set` bzw. natürlich auch `LinkedHashSet` inhärent Duplikate vermeiden, sodass man sich gar nicht extra um die Duplikateliminierung kümmern muss. In der Praxis funktioniert das aber nicht, sondern es bleiben Duplikate im `Set` enthalten. Das ist auch korrekt so, denn die offizielle Java-Dokumentation zu `java.util.Set` definiert explizit kein definiertes Verhalten, wenn die Elemente eines Sets zur Laufzeit bezüglich ihrer Identität modifiziert werden:

[8] In diesem konkreten Beispiel kann man sich natürlich einfach die zu den IDs zugehörigen Originaltupel ansehen und durch Abgleich der Konstanten `'Anna'` und `'Mia'` sofort erkennen, was wo gematcht hat. Bei komplexeren Anfragen ist das aber nicht mehr möglich.

> *Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set* [Ora22].

Daher wird ab Zeile 13 die Duplikateliminierung manuell vorgenommen.
Die Reduzierung der Provenance *nach* der Chase-Phase ist, dass es unnötige Komplexität in das Framework bringt, die Provenance *erst* in die Anfrage einzuarbeiten, dann einerseits mit diesem Ergebnis direkt weiterzuarbeiten, aber dann doch später aus dem Ergebnis wieder zu entfernen und mit diesem zweiten reduzierten Ergebnis die einen zweiten Backchase-Schritt durchzuführen.
Daher haben wir uns dazu entschieden, bereits die Chase-Phase sowohl mit der originalen, als auch mit der umgeschriebenen Eingabe auszuführen und ab da *konsequent zwei parallele Pipelineverläufe bis zum Ende* zu haben, die aber komplett unabhängig voneinander sind. So ist auch die vorgestellte Architektur in Abbildung 4.1.
Wir müssen bedenken, dass der Chase-Algorithmus exponentielle Laufzeit hat und ein doppeltes Ausführen der Chase-Phase dementsprechend vermeintlich schlechte Auswirkung auf die Effizienz des Frameworks hat. Allerdings gerade da die beiden Pipelines jetzt komplett unabhängig voneinander sind, können sie in unserer Implementierung in eigenen Threads nebenläufig voneinander ausgeführt werden. Dadurch ist die Zeitkomplexität in der Praxis kaum beeinträchtigt und wir haben eine saubere konzeptuelle Kapselung der beiden Pipelines *mit* und *ohne* ProSA-Provenance.

Listing 4.4: Methoden zum Reduzieren des Chase-Ergebnisses um die ProSA-Provenance. (Klasse `Provenancer`)

```
1  public static Instance removeProvenance(Instance chaseResult) {
2      Instance instanceCopy = new Instance(chaseResult);
3      HashMap<String, LinkedHashMap<String, String>> schemaCopy = instanceCopy.
           getSchema();
4      removeTupleIDAttributeFromSchema(schemaCopy);
5      removeTupleIDsFromInstanceTuples(instanceCopy);
6
7      // eventual duplicates should be eliminated automatically since the relational
8      // atoms are stored in a set.
9      // TODO does not work yet because the hashCode() function for the atoms is not
10     // implemented to reflect their content.
11
12     // So for now eliminate duplicates manually:
13     LinkedHashSet<RelationalAtom> duplicateFreeAtoms = new LinkedHashSet<>();
14
15     for (RelationalAtom atom : instanceCopy.getRelationalAtoms()) {
16         if (!duplicateFreeAtoms.contains(atom)) {
17             duplicateFreeAtoms.add(atom);
18         }
19     }
20     return new Instance(duplicateFreeAtoms, schemaCopy, OriginTag.INSTANCE);
21 }
22
23 public static void removeTupleIDAttributeFromSchema(Map<String, LinkedHashMap<String,
       String>> schema) {
24     schema.forEach((relationName, attributeToTypeMap) -> {
25         attributeToTypeMap.remove(relationName + "_id");
26     });
27 }
28
29 public static void removeTupleIDsFromInstanceTuples(Instance instance) {
30     instance.getRelationalAtoms().forEach(atom -> {
31         // remove if termName ends with '_id':
32         atom.getTerms().removeIf(term -> (term.isConstant() && (term.getName().
               endsWith("_id"))));
```

```
33      });
34  }
```

Problem mit dem Kopieren von `SingleInput`-Objekten Wir haben uns also für eine Nutzung von zwei parallelen Pipelines entschieden. Damit wir diese unabhängig voneinander betreiben können, brauchen wir eine Kopie des ursprünglichen Parser-Outputs. Den ersten Ansatz zum Anlegen dieser Kopie sehen wir im Aufruf `copy(input)` in Listing 4.5. Dabei offenbarten sich zwei Probleme:

- diverse Methoden zum Kopieren von ChaTEAU-Datentypen sind fehlerhaft: zum Beispiel erzeugt der Konstruktor `new Instance(instance)`, der in Zeile 12 aufgerufen wird, keine komplett neues Objekt, sondern weist der „kopierten“`Instance` dieselben Referenzen wie dem Original zu. Ein anderes Beispiel ist die Methode `copy` der Klasse `STTgd`, die sich hinter dem Aufruf `constraint.copy()` in Zeile 15 verbergen kann, erzeugt ein Objekt vom Typ `Tgd`[9].

- ChaTEAU kann einen `SingleInput` nicht korrekt verarbeiten, wenn er nicht mit dem `InputReader` aus einer XML-Datei ausgelesen wurden, sondern die Komponenten im Konstruktor manuell zusammengesetzt sind, wie in Zeile 17.

Aus den beiden genannten Gründen verwenden wir in dieser Arbeit durchgehend einen Workaround, um Ein- und Ausgabeobjekte zu kopieren: Wir serialisieren die Objekte zu Kopieren in eine temporäre Datei und lesen sie dann wieder aus (siehe 4.8.2 und 4.10).

Listing 4.5: Verworfenes Vorgehen zum Kopieren der Parser-Ausgaben vor der Chase-Phase. (Klasse `Provenancer`)

```
1  public static SingleInput provencanceChaseInput(SingleInput input) {
2      SingleInput copy = copy(input);
3
4      addTupleIDAttributeToSchema(copy.getInstance().getSchema());
5      addTupleIDsToInstanceTuples(copy.getInstance());
6      provenanceConstraints(copy.getConstraints();
7
8      return copy;
9  }
10
11 private static SingleInput copy(SingleInput input) {
12     Instance instanceCopy = new Instance(input.getInstance());
13     LinkedHashSet<IntegrityConstraint> constraintsCopy = new LinkedHashSet<>();
14     input.getConstraints().forEach(constraint -> {
15         constraintsCopy.add(constraint.copy());
16     });
17     return new SingleInput(instanceCopy, constraintsCopy);
18 }
```

Nun haben wir kennengelernt, wie wir in die Originaleingabe ProSA-Provenance einarbeiten können und die Pipeline so aufteilen können, dass wir zwei parallele Chase-Phase durchführen können. Nach der Chase-Phase stellt stellt sich die Frage, wie wir das Chase-Ergebnis mit der Inverse für die Backchase-Phase kombinieren können. Dies schauen wir uns im nächsten Abschnitt genauer an.

[9] im Rahmen dieser Arbeit behoben (siehe 4.8.2)

4.4 Kombinierer und Backchase

Für die Backchase-Phase müssen das ChaTEAU-Ergebnis aus der Chase-Phase mit der Inversen aus dem Invertierer wieder in eine gültige ChaTEAU-Eingabe-XML umgeformt werden. Wir befinden uns also in der Pipeline (Abbildung 4.1) an der Stelle des Kombinierers (5). Den Code für die die gesamte Backchase-Phase inklusive Aufruf des Invertierers und des Kombinierers sehen wir in Listing 4.6 und Listing 4.7: Zunächst sortieren wir die Abhängigkeiten aus der Eingabe für die Chase-Phase nach s-t tgds und tgds (Zeile 2-14), da der Invertierer sie in dieser Form getrennt voneinander erwartet. Dann rufen wir den Invertierer auf (Zeile 16). Das Ergebnis ist ein Paar aus einer Menge von s-t tgds und einer optionalen tgd. Diese getrennten Abhängigkeiten fügen wir für den Kombinierer wieder zu einer Gesamtmenge hinzu (Zeile 17-25). Der `else`-Zweig (Zeile 22-25) wird übrigens nur dann ausgeführt, wenn die ursprüngliche SQL-Anfragen Aggregatfunktionen enthielt – daher ist die tgd in der Ausgabe des Invertierers auch optional. Schließlich rufen wir den Kombinierer mit der Ergebnisinstanz der Chase-Phase und den inversen Abhängigkeiten auf. Dieser ist in eine statische Methode in der Klasse `Combiner` ausgelagert (Listing 4.4) und macht nichts weiter, als die Schema-Tags `source` und `target` im Schema der Chase-Ergebnisinstanz zu vertauschen, weil jetzt natürlich `Result` unser Quellschema ist und die ursprünglichen Quellschema aus der Chase-Phase jetzt Zielschema sind. Außerdem arbeiten wir natürlich auf Kopien, damit wir nicht das Original-Java-Objekt für das Chase-Ergebnis verändern. Nachdem wir die neue Eingabe für die Backchase-Phase als `SingleInput` korrekt zusammengesetzt haben, speichern wir diese Eingabe als XML im Verzeichnis, in dem für jeden Durchlauf der Pipeline die Zwischenschritte als XML abgespeichert werden (`src/main/resources/pipelineSteps/actual/`).

Listing 4.6: Die Methode `backchase()`. (Klasse `Chased`)

```
1  public BackChased backChase() {
2      LinkedHashSet<STTgd> sttgds = new LinkedHashSet<>();
3      sttgds.addAll(this.chaseInput.getConstraints()
4              .stream()
5              .filter(con -> (con instanceof STTgd))
6              .map(sttgd -> (STTgd) sttgd)
7              .collect(Collectors.toSet()));
8
9      LinkedHashSet<Tgd> tgds = new LinkedHashSet<>();
10     tgds.addAll(this.chaseInput.getConstraints()
11             .stream()
12             .filter(con -> (con instanceof Tgd))
13             .map(tgd -> (Tgd) tgd)
14             .collect(Collectors.toSet()));
15
16     Pair<LinkedHashSet<STTgd>, Optional<Tgd>> inverse = InvertingTechniques.invert(
           sttgds, tgds);
17     LinkedHashSet<IntegrityConstraint> constraints = new LinkedHashSet<>();
18     constraints.addAll(inverse.getLeft());
19     Optional<Tgd> tgd = inverse.getRight();
20     if (tgd.isEmpty()) {
21         logger.log(Level.FINE, "Inverse does not contain a tgd, skipping...");
22     } else {
23         logger.log(Level.FINE, "Adding tgd to inverse...");
24         constraints.add(tgd.get());
25     }
26     SingleInput backChaseInput = Combiner.combine(this.chaseResult, constraints,
           logger);
27
28     IOUtils.saveSingleInput(backChaseInput, Constants.ACTUALS.
           BACKCHASE_INPUT_WITHOUT_PROV.toString(),
29             "Could not save backchase input to XML.", logger);
30
```

```
31      return backChase(backChaseInput);
32  }
```

Listing 4.7: Die Methode `combine(...)`. (Klasse `Combiner`)

```
1  public static SingleInput combine(Instance chaseResult,
2          LinkedHashSet<IntegrityConstraint> inverse, Logger logger) {
3
4      // switch source and target tags:
5      SingleInput copyInput = IOUtils.reReadInput(new SingleInput(chaseResult, inverse)
           , logger);
6      System.out.println(copyInput.getInstance());
7      Instance copyChaseResult = copyInput.getInstance();
8      HashMap<String, SchemaTag> switchedSchemaTags = new HashMap<>();
9      copyChaseResult.getSchemaTags().forEach((relation, schemaTag) -> {
10         switch (schemaTag) {
11         case SOURCE:
12             switchedSchemaTags.put(relation, SchemaTag.TARGET);
13             break;
14         case TARGET:
15             switchedSchemaTags.put(relation, SchemaTag.SOURCE);
16             break;
17         default:
18             break;
19         }
20
21     });
22     Instance result = new Instance(
23             copyChaseResult.getRelationalAtoms(),
24             copyChaseResult.getSchema(),
25             copyChaseResult.getOriginTag(),
26             switchedSchemaTags);
27
28     return new SingleInput(result, inverse);
29 }
```

Wir haben jetzt genauer die Konzepte und Umsetzungen für den Provenancer und Backchase-Schritt kennengelernt. Wir wissen also, wie *einzelne* Schritte der Pipeline genau durchgeführt werden, aber noch nicht, wie diese Zwischenschritte untereinander zusammenhängen und wie wir unterschiedliche Zustände von zwei nebenläufigen Pipelines gleichzeitig speichern und verwalten können. Um dieses Problem zu lösen, haben wir den `ProSAService` entwickelt, den wir im nächsten Kapitel genauer beleuchten.

4.5 ProSAService

Die Implementierung für das Framework begann in den Controller-Klassen der GUI (hauptsächlich `ChaTEAUController` und `MainGuiController`), wo erst die komplette Logik platziert wurde. Erst später fingen wir an, die Anwendungslogik in die Klasse `ProSAService` auszulagern, welche nun in der GUI aufgerufen wird, aber auch komplett ohne GUI alleinstehend nutzbar ist. In dem Unit-Test (Kapitel 4.9), in dem das durchgehende Beispiel (Abschnitt 4.10) abgetestet wird, verwenden wir ebenfalls den `ProSAService`.
Ein signifikanter Lerneffekt dieser Arbeit ist, die Implementierung besser direkt von Anfang an in einer eigenständigen Klasse zu machen, weil das nachträgliche Extrahieren der Logik aus den Controllern der GUI sehr aufwendig war. Schauen wir uns an, wie der `ProSAService` genau aufgebaut ist:

Wie im Code-Ausschnitt 4.8 zu sehen ist, akzeptiert die Klasse `ProSAService` im Konstruktor einen Logger und eine Datenbankverbindung zu einer PostgreSQL-Datenbank. Sie verfügt über drei optionale Attribute `withAggregates`, `plainPipeLine` und `provenancePipeLine`.
`withAggregates` ist eine Umgebungsvariable, in der nach dem Parser-Schritt gespeichert wird, ob die eingegebene SQL-Anfrage Aggregatfunktionen enthält. In den anderen beiden Attributen ist jeweils der Zustand der beiden Pipelines gespeichert. Die Klassen für diese Attribute sind `Parsed`, bzw. `WithRewrittenInput`. `Parsed` repräsentiert beispielsweise den Zustand 2 in der Pipeline (Abb. 4.1), während ein Objekt vom Typ `WithRewrittenInput` darstellt, dass die Anfrage schon für die Sammlung von ProSA-Provenance in der Chase-Phase umgeschrieben wurde.

Listing 4.8: Der ProSA-Service mit den beiden Pipelines als optionale Attribute

```
1  public class ProSAService {
2
3      protected Logger logger;
4      protected Connection connection;
5      protected Optional<Boolean> withAggregates = Optional.empty();
6      protected Optional<Parsed> plainPipeLine = Optional.empty();
7      protected Optional<WithRewrittenInput> provenancePipeLine = Optional.empty();
8
9      /**
10      * Constructor
11      *
12      * @param logger Logger to use.
13      * @param con PostgreSQL DB connection to use.
14      */
15     public ProSAService(Logger logger, Connection con) {
16         this.logger = logger;
17         this.connection = con;
18     }
19     //...
```

In Abbildung 4.5 sehen wir das Klassendiagramm des ProSAService. Hier sind die Zustände so implementiert, dass ein Zustand immer von seinem Vorgänger in der Pipeline erbt. Dadurch erhält zum Beispiel der Zustand `BackChased` automatisch alle Eigenschaften von `Chased`, wie das Chase-Ergebnis, aber *zusätzlich* dazu auch ein Backchase-Ergebnis und die Inverse. Wir sehen außerdem, dass `WithRewrittenInput` von Parsed erbt, ohne zusätzliche Attribute oder Methoden zu haben. Das liegt daran, dass die beiden Zustände aus technischer Hinsicht komplett identisch sind, außer dass eben die Eingabe für die Chase-Phase umgeschrieben wurde. So können wir anhand der Klasse erkennen, in welcher Pipeline wir uns befinden und außerdem sämtliche Funktionen der „normalen Pipeline“ in der Pipeline mit Provenance wiederverwenden. Der `ProSAService` verfügt sowohl über Methoden, die Zwischenschritte der Pipeline einzeln durchzuführen (wie „`parse()`“ als auch Methoden, die mehrere Zwischenschritte zusammenfassen. Beispielsweise fasst die Methode `runPlain()` alle Zwischenschritte der Pipeline für Modus 1 „`parse(...)`“, „`chase(...)`“ und den Backchase-Schritt zusammen.
Jedes Mal, wenn ein Schritt irgendeiner der beiden Pipelines durchgeführt wird, passieren neben dem eigentlichen Zwischenschritt selbst zwei Dinge: erstens wird das Attribut in `ProSAService` für den Zustand der Pipeline (`plainPipeline` oder `provenancePipeLine`) jeweils mit seinem „Nachfolger“ überschrieben und zweitens wird das Ergebnis dieses Zwischenschritt im dafür vorgesehenen Verzeichnis[10] als XML-Datei abgespeichert. Zuletzt betrachten wir den Sonderfall, dass die SQL-Anfrage Aggregatfunktionen enthält (Modus 3): Jedes Mal, wenn `parse(...)` aufgerufen wird, erkennt der `ProSAService`, ob es Aggregate in der Anfrage gibt: wenn ja, werden beide Attribute `plainPipeLine` und `provenancePipeLine` auf dasselbe Objekt gesetzt. So haben wir faktisch nur noch eine Pipeline. Der `ProSAService` ist nun ganz simpel so in die GUI eingebunden: Der `MainController` hat den `ProSAService` als Klassenattribut,

[10] `src/main/resources/pipelineSteps/actual/`

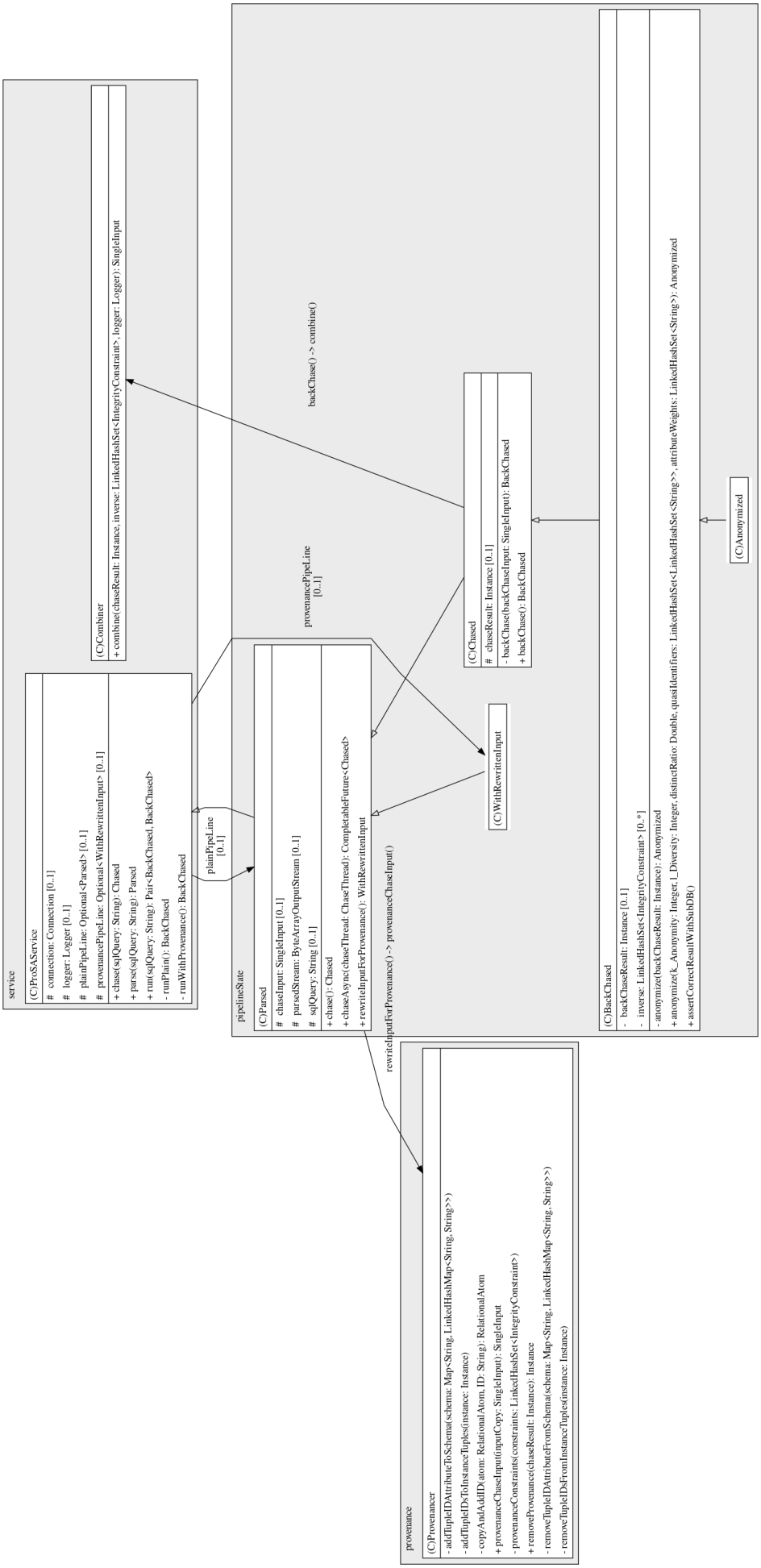


Abbildung 4.5: Klassendiagramm des `ProSAService`

und in den jeweiligen Listener-Methoden der GUI werden die passenden Methoden des `ProSAService` aufgerufen. Die Ergebnisse werden dann per Getter ausgelesen und in die entsprechenden Textfelder der GUI ausgegeben.

4.6 Test auf Reproduzierbarkeit des Originalergebnisses aus der Chase-Phase

Eine minimierte Datenbank kann natürlich nur dann veröffentlicht werden, wenn mit ihr dasselbe Ergebnis für die SQL-Anfrage herauskommt, wie mit der Originaldatenbank. Listing 4.9 zeigt den Code in der Klasse `BackChased`, der dies überprüft. Zunächst müssen wir aus der Instanz des Backchase-Ergebnisses und dem Schema der Eingabe für die Chase-Phase wieder einen gültigen `SingleInput` zusammensetzen (Zeile 5). Dann müssen wir diesen zusammengesetzten SingleInput einmal serialisieren und wieder einlesen (Zeile 8), weil ChaTEAU nicht korrekt mit manuell zusammengesetzten Eingaben arbeitet (vgl. 4.3.4). Nun führen wir den ChaTEAU aus und und vergleichen das Ergebnis mit dem Originalergebnis (Zeile 9-10). Wenn dieser Vergleich fehlschlägt, wird eine Exception vom Typ `ReproducibilityFailure` geworfen. Die `Message` dieser Exception enthält die beiden unterschiedlichen Ergebnisinstanzen, die `StringBuilder` ein wenig grafisch aufbereitet werden (Zeile 12-17). Hinter den dabei verwendeten Konstanten `SLS` und `LINE` verbergen sich das Zeichen für `newline` (`System.lineSeparator()`) und eine horizontale Trennlinie.

Listing 4.9: Methode zum Überprüfen, ob das Originalergebnis der Anfrage mit der berechneten minimalen Teildatenbank reproduziert werden kann. (Klasse: `BackChased`)

```
1  public void assertCorrectResultWithSubDB() throws ReproducibilityFailure {
2      // use schema from chase input for re-building the input with
3      // backChaseResult(which does not have correct schema for running chase):
4      HashMap<String, LinkedHashMap<String, String>> schema = chaseInput.getInstance().
           getSchema();
5      Instance instanceWithCorrectSchema = new Instance(backChaseResult.
           getRelationalAtoms(), schema,
6              backChaseResult.getOriginTag(), chaseResult.getSchemaTags());
7
8      SingleInput reReadInput = IOUtils.reReadInput(new SingleInput(
           instanceWithCorrectSchema, this.getConstraints()), logger);
9      Instance subDBResult = ChaseService.runChase(reReadInput.getInstance(), this.
           getConstraints());
10     if (!subDBResult.equals(this.getChaseResult())) {
11
12         StringBuilder message = new StringBuilder("Result with minimal sub-DB is
               different than result with original DB.");
13         message.append(SLS + LINE + SLS);
14         message.append("Result with orignal DB:");
15         message.append(SLS + LINE + SLS).append(this.getChaseResult()).append(SLS +
               LINE + SLS);
16         message.append("Result with minimal sub-DB:");
17         message.append(SLS + LINE + SLS).append(subDBResult).append(SLS + LINE + SLS)
               ;
18
19         throw new ReproducibilityFailure(message.toString());
20     }
```

4.7 Die Utility-Klasse `IOUtils`

Die Klasse `IOUtils` im Package `util` enthält neben einer Methode, die JSON-Datei mit der Datenbank-Konfiguration zur Herstellung einer Datenbankverbindung einzulesen auch diverse nützliche Methoden, um Zwischenschritte der Pipeline, die als `SingleInput`, `Instance` oder `LinkedHashSet<IntegrityConstraint>` vorliegen, als XML-Datei im dafür vorgesehenen Verzeichnis `src/main/resources/pipelineSteps/actual/`

abzuspeichern. Außerdem befindet sich hier die Methode, um `SingleInput`-Objekte zu kopieren (Zeile 41). Dies ist ein Workaround, um die fehlerhaften `copy`-Methoden und Kopier-Konstruktoren (wie beispielsweise `new Instance(instance)`) zu umgehen. Wir serialisieren den `SingleInput` in eine temporäre Datei und lesen sie wieder ein. So haben wir definitiv ein komplett neues Objekt und keine gemeinsamen Referenzen mehr mit dem Original-`SingleInput`.

Listing 4.10: Utilities zum Schreiben und Kopieren von `SingleInput`-Objekten. (Klasse `IOUtils`

```
1  //...
2  public static void saveSingleInput(SingleInput input, String path, String
       errorMessage, Logger logger) {
3      saveToXML(input.getInstance(), input.getConstraints(), path, false, errorMessage,
           logger);
4  }
5
6  public static void saveInstanceToXML(Instance instance, String path, String
       errorMessage, Logger logger) {
7      saveToXML(instance, null, path, true, errorMessage, logger);
8  }
9
10 protected static void saveToXML(Instance instance, LinkedHashSet<IntegrityConstraint>
        constraints, String path,
11         boolean includeProvenance, String errorMessage, Logger logger) {
12     saveToXML(instance, constraints, new File(path), includeProvenance, errorMessage,
           logger);
13
14 }
15
16 private static void saveToXML(Instance instance, LinkedHashSet<IntegrityConstraint>
       constraints, File file,
17         boolean includeProvenance, String errorMessage, Logger logger) {
18     OutputWriter ow = new OutputWriter();
19     ow.outputProvenance = includeProvenance;
20     ow.preserveSourceSchema = true;
21     try {
22         ow.writeXML(instance, null, constraints, file);
23     } catch (IOException e) {
24         if (logger != null) {
25             logger.log(Level.SEVERE, errorMessage, e);
26         } else {
27             System.out.println(errorMessage);
28             e.printStackTrace();
29         }
30     }
31 }
32
33 //...
34
35 /**
36  * Serves as copy method.
37  * @param input the SingleInput to copy
38  * @param logger
39  * @return the copy
40  */
41 public static SingleInput reReadInput(SingleInput input, Logger logger) {
42
43     try {
44         // Thread safety: use different names for different threads:
45         File file = File.createTempFile(Thread.currentThread().toString(), "xml");
46         file.deleteOnExit();
47         saveToXML(input.getInstance(), input.getConstraints(), file, false, "Could
```

```
                not re-read SingleInput", logger);
48          InputReader reader = new InputReader();
49          return reader.readFile(file);
50      } catch (Exception e) {
51          String msg = "Could not read temporary file";
52          if (logger != null) {
53              logger.log(Level.SEVERE, msg, e);
54          } else {
55              System.out.println(msg);
56              e.printStackTrace();
57          }
58          return null;
59      }
60  }
```

4.8 Weitere Arbeiten

Bis jetzt haben wir die wichtigsten erarbeiteten Konzepte und Umsetzungen dargelegt, die den eigentlichen Kern dieser Abschlussarbeit darstellen. Im Lauf der Implementierung ergaben sich aber auch diverse kleinere Änderungen, die wir uns in diesem Abschnitt ansehen wollen.

4.8.1 Änderungen am Parser

Integration der ersten Parser-Version in ProSA Zu Beginn dieser Arbeit lag der Parser in einem eigenen Java-Projekt und war noch gar nicht in das ProSA-Projekt integriert. Um den Parser zu integrieren, wurde im Rahmen dieser Arbeit daran gearbeitet, mit dem Ansatz, den Parser als eigenes Projekt gekapselt zu lassen und nur die Abhängigkeit von Maven zu importieren. Dies führte allerdings mit mehreren externen Bibliotheken zu Schwierigkeiten, im Einzelnen `Lombok` und `Google Guava`.
Parallel hat eine studentische Hilfskraft den Code des Parsers direkt in das ProSA-Projekt übernommen, was hinsichtlich der Austauschbarkeit nachteilig ist, aber diese Lösung war ganz einfach schneller fertig und wurde daher genommen. Die etwas saubere Lösung mit der Kapselung wird allerdings im Kapitel 5.1.2 als Ausblick für weitere Verbesserungen vermerkt, die nicht so hoch priorisiert sind.

Erweiterung der Tupel-IDs im Parser Die erste konkrete Änderung am Code selbst galt der Erweiterung der Tupel im XML-Output des Parser um IDs. Es gab mehrere Ideen, wie diese konkret eingebaut werden könnten:

- direkt im Parser: dafür spricht, dass die Implementierung an dieser Stelle technisch sehr einfach ist. Dagegen spricht aber, dass eine Erweiterung, d.h. eine konkrete Änderung der Daten aus der Datenbank konzeptuell nicht in eine Komponente namens „Parser“ gehört.
- in ChaTEAU: man könnte
  - entweder einen dritten Aufruf des Chase-Algorithmus machen, in dem das Hinzufügen der IDs als Abhängigkeit definiert. Das wäre als Konzept theoretisch sehr sauber und würde die Mächtigkeit von ChaTEAU als Universalwerkzeug unterstreichen, die Vorbereitung des Inputs für den Chase selbst mit dem Chase durchzuführen.
  - oder im `InputReader` (eine Komponente von ChaTEAU), da dieser genau wie der Parser ohnehin alle Tupel “anfasst”. Dagegen spricht aber dann, genau wie bei der Idee mit dem Parser, dass der InputReader dann nicht mehr nur Eingaben liest, sondern modifiziert.

- eine separate statische Funktion, mit dem einzigen Zweck, Tupel-IDs hinzuzufügen. Das wäre auch konzeptuell sauber, aber sehr aufwendig.

Letztendlich fiel die Wahl auf die Implementierung im Parser, da es dort leicht umzusetzen ist und die Funktionen des Parsers sowieso schon über das hinausgehen, was man sich unter der Bezeichnung "Parser" vorstellt.

4.8.2 Änderungen an ChaTEAU

Im Verlauf der Implementierung fielen diverse Probleme mit ChaTEAU auf, die folgende im Rahmen dieser Arbeit durchgeführte Änderungen nach sich zogen:

Nicht-deterministische Reihenfolge der Provenance-Elemente im ChaTEAU-Ausgabe-XML Ein Problem war, dass die von ChaTEAU unter Nutzung des „`-p`"-Flags ausgegebene where-, why- und how-Provenance bei identischer Eingabe nach jedem Neustart der JVM in verschiedener Reihenfolge ausgegeben wurden. Das bedeutet im einzelnen, dass die Einträge in den Zeugenlisten, Zeugenbasen und Provenance-Polynomen so permutiert waren, dass das Ergebnis wegen der Kommutativität zwar immer noch formal korrekt war, aber anders als beim vorherigen Durchlauf. Zum Testen der Pipeline in einem Unit-Test ist dies natürlich ein großes Problem. Nach einigem Debugging stellte sich heraus, dass die Ursache für dieses Verhalten die verwendete Datenstruktur `HashSet` ist, welche keine Iterierungsreihenfolge vorgibt, sondern die jeweils beim Start der JVM für die jeweiligen Klassen, die im `HashSet` enthalten sind, neu initialisiert wird. Wir haben dieses Fehlverhalten durch die Nutzung der alternativen `Set`-Implementierung `LinkedHashSet` behoben. Zu finden ist diese Implementierung in Mergerequest `!21`[11].

Erweiterung der Klasse `OutputWriter`

- **Funktion, um SingleInputs zu schreiben**: Wie bereits in Abschnitt 4.10 erwähnt, benutzen wir den `OutputWriter`, um Objekte der Klasse `SingleInput` mittels Serialisierung und Deserialisierung zu kopieren. Der `OutputWriter` ist aber eigentlich nur für das Schreiben von Chase-Ergebnissen konzipiert, die ein Schema, sowie wahlweise eine Instanz oder eine s-t tgd (wenn der Chase auf Anfragen ausgeführt wurde) enthalten.
  S-t tgds, tgds und egds, die nicht das Objekt ○, sondern die Abhängigkeiten ★ sind, konnte der `OutputWriter` noch nicht serialisieren, und daher auch mit einem kompletten `SingleInput` als Eingabe nicht umgehen. Diese Funktionalität haben wir im ChaTEAU-Repository durch Mergerequest `!24`[12] hinzugefügt.

- **Anpassung an neues XML-Format**: ChaTEAU kann zwar das neue XML-Format (siehe Abschnitt 4.8.3), korrekt einlesen, gibt aber durch den OutputWriter immer noch das alte Format aus. Dies haben wir in Mergerequest `!23`[13] behoben.

Korrektur der `copy()`-Methoden für `Constant` und `STTgd` Die Methoden zum Kopieren von Konstanten und s-t tgds waren fehlerhaft, zum Beispiel hat eine kopierte `STTgd` ein Objekt vom Typ `Tgd` zurückgegeben, was ebenfalls in Mergerequest `!23` behoben wurde.

[11] `https://git.informatik.uni-rostock.de/ta093/chateau/-/merge_requests/21`
[12] `https://git.informatik.uni-rostock.de/ta093/chateau/-/merge_requests/24`
[13] `https://git.informatik.uni-rostock.de/ta093/chateau/-/merge_requests/23`

Hinzufügen von Terminierungstests in der ChaTEAU-API ChaTEAU verfügte bereits über eine API in der Klasse `ChaseService`, die aber noch keine Terminierungstests anbot. Diese wurden in Mergerequest `!22`[14] hinzugefügt und der Code so umstrukturiert, dass die Tests jetzt entweder direkt im `ChaseService` mit `ChaseService.runTests(instance, constraints)`, oder auch in der separaten neuen Klasse `TesterService` mit `TesterService.runTests(singleInput)` aufgerufen werden, die bei einem fehlgeschlagenen Test eine Exception vom Typ `TerminationTestFailure` werfen, auf die dann entsprechend reagiert werden kann.

4.8.3 Weitere Änderungen

Hier sind Änderungen aufgelistet, die nicht konkrete Komponenten der Pipeline betreffen, sondern z.B. Datenformate oder das Build-Tool Maven.

Abgleich mit der aktuellen Version des Parsers aus [Kav22] und des Invertierers aus [Spo22] Der Parser wurde natürlich in der parallelen Abschlussarbeit [Kav22] ständig weiterentwickelt. Die dortigen Änderungen haben wir über den gesamten Zeitraum in diese Arbeit „hineingemergt“, sodass das ProSA-Framework zum Abschluss dieser Arbeit bereits den aktuellen Parser aus [Kav22] beinhält.
Ebenso wurde der Invertierer in der Arbeit [Spo22] erst relativ spät fertig, aber durch Absprache der Schnittstellen konnten wir ProSA schon entsprechend mit Platzhaltern vorbereiten und den fertigen Invertierer dann ziemlich problemlos integrieren, sodass ProSA hier auch schon die aktuelle Version hat.

Änderung des XML-Formats für ChaTEAU-Eingaben Bei der Integrierung des Parsers stellte sich heraus, dass dieser in seiner Ausgabe zwischen Attributen des Schemas und den Atomen in der Instanz und in den Abhängigkeiten nicht immer dieselbe Reihenfolge beibehält. Das liegt daran, dass der Parser zur Serialisierung das `Jakarta XML Binding` („`JAXB`“) der externen Bibliothek `Jakarta` benutzt. Diese Bibliothek hält sich beim Serialisieren der XML an die vorgegebene XSD (Anhang 6.9), aber in dieser XSD ist es nicht möglich, eine Reihenfolge für die Attribute des Schemas und Attribute der Atome zu definieren, also hält sich die Bibliothek auch an keine Reihenfolge.
ChaTEAU ordnet die Terme eines Atoms in der Instanz oder in in den Abhängigkeiten aber auf Grundlage ihrer Reihenfolge zu, d.h. der zweite Attributwert eines Atoms gehört zum zweiten im Schema definierten Attribut in der zugehörigen Relation. Daher konnte ChaTEAU so nicht mit dem Ausgabe-XML-Format des Parsers umgehen. Die Lösung dafür war eine Anpassung dieses Formats, sodass nun nicht nur bei Variablen und Nullwerten der Name des zugehörigen Attributs enthalten ist, sondern auch bei Konstanten (vgl. Listing 4.11 und Listing 4.12).

Listing 4.11: Altes XML-Format: Weil im Parser-Output die Reihenfolge der Attribute `matno` und `firstname` in Schema und Atom unterschiedlich sind, kann ChaTEAU sie nicht korrekt zuordnen

```
1 <!-- Schema-Definition-->
2 <relation name="students" tag="S">
3    <attribute name="matno" type="int"/>
4    <attribute name="firstname" type="string"/>
5 </relation>
6
7 <!-- ... -->
8
9 <!-- Atom in Dependencies oder Instanz:-->
```

[14] https://git.informatik.uni-rostock.de/ta093/chateau/-/merge_requests/22

```
<atom name="students">
    <constant value="Stefan"/>
    <variable name="matno" type="V" index="1"/>
</atom>
```

Listing 4.12: Im neuen XML-Format enthalten die Elemente für Konstanten zusätzlich den zugehörigen Attributnamen und sind dadurch trotz unterschiedlicher Reihenfolge zuordenbar.

```
<!-- Schema-Definition -->
<relation name="students" tag="S">
    <attribute name="firstname" type="string"/>
    <attribute name="matno" type="int"/>
</relation>

<!-- ... -->

<!-- Atom in Dependencies oder Instanz:-->
<atom name="students">
    <variable name="matno" type="V" index="1"/>
    <constant name="firstname" value="Stefan"/>
</atom>
```

Natürlich mussten daher im Rahmen dieser Arbeit alle Komponenten so umgebaut werden, dass sie mit diesem neuen XML-Format umgehen können. Die Änderungen für die *Eingabe und Verarbeitung* in ChaTEAU hat ein am ProSA-Projekt mitwirkender Hiwi erledigt, die Änderungen am Parser sind in der parallelen Arbeit [Kav22] entstanden, aber die Änderungen für die *Ausgabe* von ChaTEAU (siehe 4.8.2) und sämtliche Zwischenverarbeitungsschritte haben wir in dieser Arbeit implementiert.

Anpassung der Maven-Abhängigkeiten Es gab das Problem, dass die Unit-Tests nicht vom dafür vorgesehen Maven-Goal `test` ausgeführt wurden, sondern nur manuell. Die Ursache lag in einem Bug der Abhängigkeit `maven-surefire-plugin`, die in der Version nicht korrekt mit `JUnit 5` zusammenarbeiten konnte. Dies wurde in Mergerequest `!1`[15] behoben, sodass jetzt bei jedem Build auch gleich die Tests mit durchgeführt werden.

Wir kennen jetzt alle Implementierungsdetails und Designentscheidungen des Frameworks. Zur Qualitätssicherung haben wir mit einem durchgehenden Beispiel gearbeitet, für das ein Unit-Test geschrieben wurde. Diesen wollen wir uns nun genauer anschauen.

4.9 Unit-Test für die gesamte Pipeline

Für die Arbeit mit dem durchgehenden Beispiel (4.10) haben wir einen JUnit-Test entwickelt. Er ist in der Klasse `PipeLineTest` implementiert, von dem der relevante Code in Listing 4.13 zu sehen ist:
Darin wird zunächst die Test-Datenbank in Zeile 3 vorbereitet. Die Methode `TestUtils.prepareDB();` ruft das SQL-Skript `setUpTestDB.sql`[16] auf, welches die Datenbank – falls vorhanden – löscht und neu anlegt. Dann erzeugen wir einen neuen `ProSAService` und rufen den Parser mit der vorbereiteten SQL-Anfrage

```
SELECT firstname, modulename
FROM students NATURAL JOIN participants
WHERE firstname = 'Stefan';
```

[15] `https://git.informatik.uni-rostock.de/ta093/prosa/-/merge_requests/1`
[16] Das Skript ist in Anhang 6.10 zu finden.

auf (Zeile 9). Da der `ProSAService` bei allen Zwischenschritten der Pipeline die Ergebnisse im dafür vorgesehenen Verzeichnis speichert (vgl. Abschnitt 4.5), können wir diese XML-Dateien zum Testen einfach einlesen und mit der erwarteten XML-Datei direkt Byte für Byte vergleichen.[17] Nachdem die Anfrage geparst ist, führen wir jeweils die Pipeline ohne Provenance (Zeile 14-27), bzw. mit Provenance (Zeile 30-46) aus und vergleichen die Zwischenschritte analog. Im Einzelnen sind diese Zwischenergebnisse bei beiden Pipelines jeweils:

- das Chase-Ergebnis
- die kombinierte Backchase-Eingabe, die bereits die Inverse aus dem Invertierer enthält (daher muss die nicht extra getestet werden)
- das Backchase-Ergebnis.

Bei der Pipeline mit Provenance kommt noch ein weiterer Test hinzu: Die umgeschriebene Chase-Eingabe wird in Zeile 32 überprüft. Bei der Pipeline ohne Provenance entfällt dies, weil die Eingabe für den Chase ja identisch mit der Parser-Ausgabe ist, die schon getestet wurde.

Zu guter Letzt wird bei beiden Pipelines noch überprüft, ob das Originalergebnis mit der minimalen Teildatenbank reproduzierbar ist (Zeilen 27 & 46). Im Fehlerfall wirft dieser Check eine `ReproducibilityFailure` (vgl. Abschnitt 4.6). Der Anonymisierer ist noch nicht implementiert und daher auch nicht in ProSA integriert, deshalb findet dieser Schritt im Unit-Test keine Anwendung.

Listing 4.13: Der Unit-Test für die gesamte Pipeline. (Klasse: `PipeLineTest`)

```
1  @Test
2  void testMain() {
3      TestUtils.prepareDB();
4      try {
5          Logger log = Logger.getLogger("PipelineLogger");
6          ProSAService ps = new ProSAService(log, TestUtils.createDBConnection());
7
8          //Parse step:
9          Parsed parsed = ps.parse(Constants.SQL_QUERY);
10         assertArrayEquals(readActualParserOutput(), readExpectedParserOutput(),
11                 "Parser output is incorrect.");
12
13         // Run without provenance:
14         // Chase step:
15         Chased chasedWithoutProvenance = parsed.chase();
16         assertArrayEquals(readChaseResultWithoutProvActual(),
17                 readChaseResultWithoutProvExpected(),
18                 "Chase result without provenance incorrect.");
19         // Backchase step:
20         BackChased backChasedWithoutProv = chasedWithoutProvenance.backChase();
21         assertArrayEquals(readBackChaseInputWithoutProvActual(),
22                 readBackChaseInputWithoutProvExpected(),
23                 "Backchase input without provenance incorrect.");
24         assertArrayEquals(readBackChaseResultWithoutProvActual(),
25                 readBackChaseResultWithoutProvExpected(),
26                 "Backchase result without provenance incorrect.");
27         backChasedWithoutProv.assertCorrectResultWithSubDB();
28
29         // Run with provenance:
30         // Rewrite chase input:
31         WithRewrittenInput withRewrittenQuery = parsed.rewriteInputForProvenance();
```

[17] Die für diesen Test erwarteten korrekten XML-Dateien befinden sich im Verzeichnis `src/test/resources/xml/pipelineSteps_expected/`.

```
32          assertArrayEquals(readChaseInputWithProvActual(),
                readChaseInputWithProvExpected(),
33                  "Chase input with provenance incorrect.");
34          // Chase step:
35          Chased chasedWithProv = withRewrittenQuery.chase();
36          assertArrayEquals(readChaseResultWithProvActual(),
                readChaseResultWithProvExpected(),
37                  "Chase result with provenance incorrect.");
38          // Backchase step:
39          BackChased backChasedWithProv = chasedWithProv.backChase();
40          assertArrayEquals(readBackChaseInputWithProvActual(),
41                  readBackChaseInputWithProvExpected(),
42                  "Backchase input with provenance incorrect.");
43          assertArrayEquals(readBackChaseResultWithProvActual(),
44                  readBackChaseResultWithProvExpected(),
45                  "Backchase result with provenance incorrect.");
46          backChasedWithProv.assertCorrectResultWithSubDB();
47
48      } catch (TransformationException | SQLException | JAXBException |
49              ParsingException | NoColumnsException | InvalidConstraintsFormat |
50              IOException | ParseException | JDOMException |
51              TerminationTestFailure | ReproducibilityFailure e) {
52          fail(e);
53      }
54  }
```

Nun wissen wir, wie die Zwischenschritte des Beispiels aus technischer Hinsicht getestet werden, wir haben uns aber noch gar nicht *inhaltlich* mit diesem Beispiel auseinandergesetzt. Diesen Aspekt betrachten wir im nächsten Abschnitt genauer.

4.10 Durchlaufendes Beispiel

Zur Unterstützung der Pipeline-Implementierung wurde ein durchlaufendes Beispiel verwendet. Darin stellen wir eine simple `SELECT-FROM-WHERE`-Anfrage ohne Aggregatfunktionen an die Datenbank und führen beide Pipelines für Modus 1 und Modus 2 aus. Die XML-Dateien mit allen Zwischenschritten befinden sich in Anhang 6.1. Wir verwenden für dieses Beispiel die Datenbank aus Abschnitt 1.3, die hier noch einmal in Abbildung 4.6 zu sehen ist:

students_id	firstname	matno
students_1	Stefan	1
students_2	Stefan	2
students_3	Mia	3
students_4	Anna	4

(a) Relation students(firstname, matno)

participants_id	modulename	matno
participants_1	Maths	2
participants_2	Maths	3
participants_3	Info	4
participants_4	Logic	1
participants_5	Logic	2

(b) Relation participants(modulename, matno)

Abbildung 4.6: Die vom Parser ausgelesenen Relationen des durchlaufenden Beispiels, bereits erweitert um die roten Tupel-IDs

An diese Datenbank stellen wir folgende Anfrage:

„**An welchen Modulen hat Stefan teilgenommen?**“[18]

In SQL sieht diese Anfrage folgendermaßen aus:

```
SELECT firstname, modulename
FROM students NATURAL JOIN participants
WHERE firstname = 'Stefan';
```

Das erwartete SQL-Ergebnis mitsamt **where-**, **why-** und **how-**Provenance der beiden Ergebnistupel Anfrage sehen wir in Tabelle 4.1:

firstname	modulename	**HOW**	**WHY**	**WHERE**
Stefan	Maths	$(s_2 * p_1)$	$\{\{s_2, p_1\}\}$	Relationen: {students, participants} Tupel: $\{s_2, p_1\}$
Stefan	Logic	$(s_1 * p_4)$ + $(s_2 * p_5)$	$\{\{s_1, p_4\}, \{s_2, p_5\}\}$	Relationen: {students, participants} Tupel: $\{s_1, s_2, p_4, p_5\}$

Tabelle 4.1: Ergebnisrelation der gewählten SQL-Anfrage: `Result`($firstname$, $modulename$). **Aufgrund der Lesbarkeit sind die Tupel-IDs abgekürzt.**

Die Beispielanfrage haben wir nicht zufällig so gewählt, sondern mit zwei Kriterien im Hinterkopf:

- Zum einen soll sie so simpel sein, dass sich damit eine erste Implementierung des Frameworks gut umsetzen lässt. Wir wollten vermeiden, dass etwa Verschachtelungen in der Anfrage oder „exotische Operatoren“ für unnötig viele komplizierte Abhängigkeiten in der Chase-Eingabe sorgen.[19]. Die gewählte Anfrage ergibt genau eine s-t tgd, was für unseren Fall ideal ist.

[18] Stefans Name soll mit ausgegeben werden (z.B. für Weiterverarbeitung des Ergebnisses).
[19] Siehe [Kav22] und [KRSZ21], in denen erarbeitet wurde, wie verschiedenste Formen von SQL-Anfragen als EDs umformuliert werden können.

- Zum anderen soll im Ergebnis der SQL-Anfrage mindestens ein Duplikat eliminiert worden sein,[20] sodass wir durch die Berücksichtigung von Provenance (where-, why-, how- oder ProSA-Provenance) ein anderes Ergebnis für die minimale Teildatenbank bekämen. Für unsere gewählte Anfrage besteht das Ergebnis aus zwei Tupeln `Result('Stefan', 'Maths')` und `Result('Stefan', 'Logic')`, wobei das zweite Tupel `Result('Stefan', 'Logic')` eigentlich durch **jeweils** `students_1` und `participants_4`, sowie `students_2` und `participants_5` im Ergebnis enthalten wäre. Hierdurch macht die Beachtung von Provenance also einen Unterschied, was wir bei der Entwicklung unserer Pipeline brauchten, um das Ergebnis D_{min} überhaupt von D_{min_prov} unterscheiden zu können.

Im ersten Schritt des Frameworks wird die Original-Datenbank ausgelesen und dargestellt:

Auslesen der Original-Relation und Hinzufügen von Tupel-IDs Der Parser verbindet sich mit Datenbank, liest alle Relationen aus und weist den Tupeln eindeutige IDs zu. Diese IDs sind aber kein richtiges Attribut, sondern ein separates Datum (vgl. Abschnitt 4.1). Die tabellarische Darstellung unserer gewählten Beispieldatenbank mitsamt Tupel-IDs haben wir bereits in Abbildung 4.6 gesehen. Als Nächstes werden diese ausgelesenen Original-Relationen mitsamt hinzugefügter Tupel-IDs im Chase-Tab, Backchase-Tab und Anonymisierer-Tab (Abb. 4.7, 4.8a und 4.9) jeweils im Textfeld „*Original database*“ ausgegeben.
Sobald die Nutzer:in durch Drücken des Knopfes „*Run ProSA*“ die Pipeline(s) startet, beginnt der Parser mit seiner Hauptfunktion: der Umformung der eingegebenen SQL-Anfrage in EDs (in unserem Beispiel ist das genau eine s-t tgd). Wie *genau* diese Umformung innerhalb des Parsers vonstattengeht, ist eigentlich nicht Bestandteil dieser Arbeit. Schauen wir uns trotzdem einmal exemplarisch für unser konkretes Beispiel an, wie der Parser unsere Anfrage in eine s-t tgd übersetzen kann:

Übersetzung der SQL-Anfrage in eine s-t tgd Wir zerteilen die SQL-Anfrage in ihre Operationen für die Selektion, natürlichen Verbund und Projektion und bilden daraus drei einzelne verkettete s-t tgds, die später wieder zusammengesetzt werden:

1. `JOIN` (natürlicher Verbund auf `matno`)
   $\texttt{students}(firstname, matno) \wedge \texttt{participants}(modulename, matno)$
   $\rightarrow \texttt{ParticipantNames}(firstname, modulename, matno)$

2. `WHERE` (Selektion “`firstname='Stefan'`”)
   $\texttt{ParticipantNames}(firstname, modulename, matno) \wedge firstname = \texttt{'Stefan'}$
   $\rightarrow \texttt{StefanModuleMatNo}(\texttt{'Stefan'}, modulename, matno)$

3. `SELECT` (Projektion) auf `firstname` und `modulename`
   $\texttt{StefanModuleMatNo}(\texttt{'Stefan'}, modulename, matno)$
   $\rightarrow \texttt{StefanModule}(\texttt{'Stefan'}, modulename)$

Die zusammengesetzte s-t tgd ergibt sich dann aus dem Rumpf der ersten und dem Kopf der dritten Teil-s-t-tgd. Außerdem müssen sich alle Selektionen (`firstname='Stefan'`) im Rumpf wiederfinden:
$\texttt{students}(\texttt{'Stefan'}, matno) \wedge \texttt{participants}(modulename, matno)$
$\rightarrow \texttt{StefanModule}(\texttt{'Stefan'}, modulename)$

Zu guter Letzt kann der Parser natürlich in der Praxis nicht wissen, wie die neue Ergebnisrelation “`StefanModule`” semantisch sinnvoll benannt werden soll, daher werden standardmäßig alle neuen Ergebnisrelationen von s-t tgds als “`Result`” benannt:

[20] Der Chase arbeitet **immer** mengenbasiert, daher müssen wir uns streng genommen für alle SQL-Anfragen einen zusätzlichen impliziten „`DISTINCT`“-Operator dazudenken.

$$\texttt{students}(\texttt{'Stefan'}, matno) \wedge \texttt{participants}(modulename, matno)$$
$$\rightarrow \texttt{Result}(\texttt{'Stefan'}, modulename)$$

Die Gesamt-Eingabe-XML für die Chase-Phase mit den ausgelesenen Instanz-Tupeln, den eingefügten Tupel-IDs und der erstellten s-t tgd sehen wir in Listing 6.1.

Nach dem Aufruf des Parsers werden die berechneten Abhängigkeiten (in unserem Fall die s-t tgd) in das Textfeld rechts oben im Chase-Tab (Abb. 4.7), sowie die aus der Originaldatenbank ausgelesenen Tupel mitsamt hinzugefügter Tupel-IDs im Textfeld unten links ausgegeben.

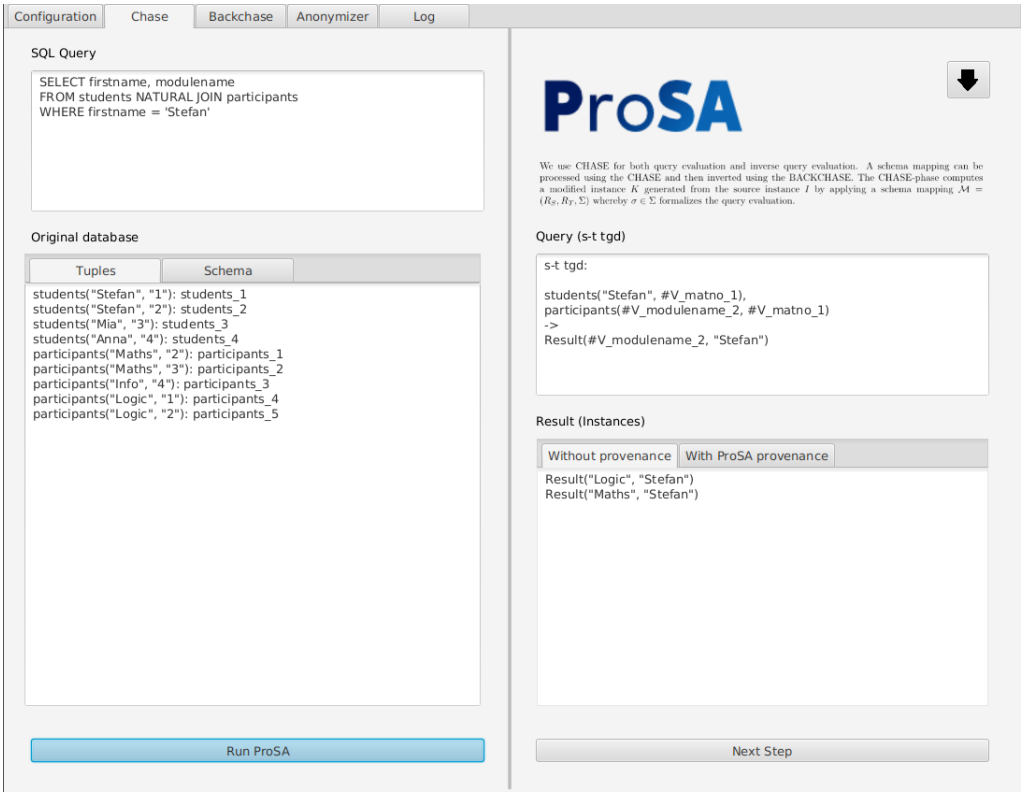


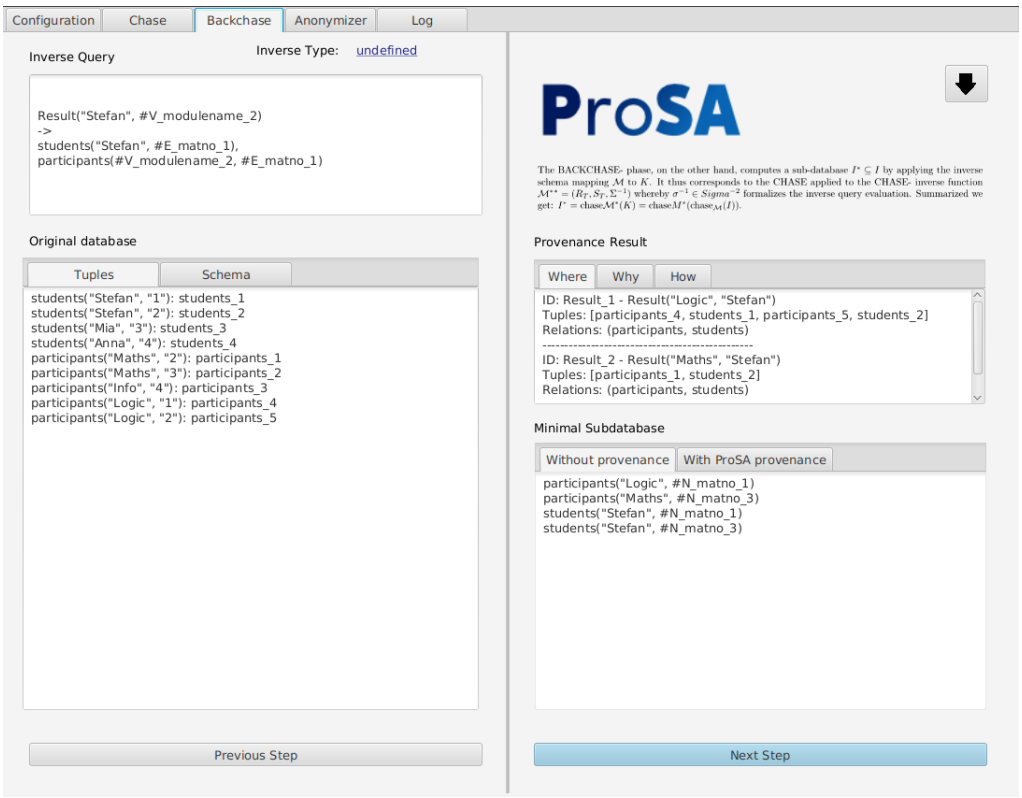
Abbildung 4.7: Der Chase-Tab, nachdem auf „Run ProSA“ geklickt wurde, um das durchlaufende Beispiel auszuführen

Ab hier teilt sich die Pipeline in Modus 1 und Modus 2 auf. Betrachten wir zunächst Modus 1 ohne Provenance:

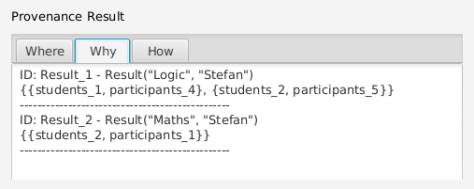
4.10.1 Modus 1 (ohne Provenance)

In diesem Modus in der nächste Schritt der Pipeline direkt die Ausführung der Chase-Phase mit der Ausgabe des Parsers.

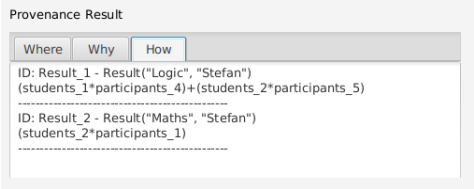
Ergebnis der Chase-Phase: Durch die Chase-Phase erhalten wir sowohl die Ergebnistupel der SQL-Anfrage, als auch deren Provenance-Informationen (where, why und how). Diese Ergebnisinstanz wird im Textfeld „*Without Provenance*“ des Chase-Tabs (Abb. 4.7) ausgegeben und die Provenance-Informationen jeweils in die dafür vorgesehenen Textfelder rechts oben im Backchase-Tab (Abb. 4.8). Das Chase-Ergebnis als XML sehen wir in Listing 6.2.



(a) Backchase-Tab mit where-provenance.



(b) Die why-Provenance des Beispiels



(c) Die how-Provenance des Beispiels

Abbildung 4.8: Der Backchase-Tab, nachdem die Beispielanfrage ausgeführt wurde. Standardmäßig wird oben rechts die where-Provenance angezeigt und unten rechts die minimale Teildatenbank von Modus 1.

Invertierung der s-t tgd Von der s-t tgd, welche die ursprüngliche SQL-Anfrage darstellt, bilden wir die Inverse, indem wir die Implikationsrichtung umdrehen und aus den Allquantoren Existenzquantoren machen:
$\forall modulename_2 : \texttt{Result}(\texttt{'Stefan'}, modulename_2)$
$\rightarrow \exists matno_1 : \texttt{students}(\texttt{'Stefan'}, matno) \wedge \texttt{participants}(modulename_2, matno_1)$.
Diese Inverse wird natürlich auch im Backchase-Tab (Abb. 4.8) links oben ausgegeben.

Eingabe-XML für die Backchase-Phase Nun kombinieren wir das Chase-Ergebnis mit der Inversen, wie in Abschnitt 4.4 beschrieben: wir vertauschen die Schema-Tags und setzen die Inverse als Abhängigkeit. Die fertige Backchase-Eingabe-XML sehen wir in Listing 6.3.

Ergebnis der Backchase-Phase: Wir wissen bereits, dass die Backchase-Phase von der Funktionsweise her exakt dasselbe ist wie die Chase-Phase, nur der Input ist unterschiedlich. Wir führen also ChaTEAU mit den zusammengesetzten Eingabe (Listing 6.3) aus. Die Ergebnisinstanz der Backchase-Phase wird im Textfeld unten rechts im Backchase-Tab (Abb. 4.8) als „*Minimal Subdatabase*“ ausgegeben. Hier sehen wir auch, dass die Variable `matno`, die in der Inversen existenzquantifiziert wurde,

hier mit Nullwerten `#N_matno_i` auftritt. Das Ergebnis des Backchase im XML-Format sehen wir in Listing 6.4.

Im Anonymizer-Tab (Abb. 4.9) wird lediglich die im Parser-Schritt ausgelesene Originaldatenbank oben links angezeigt. Der Tab daneben „*Minimal Subdatabase (tuples)*“ wird nur in Modus 2 befüllt, den wir uns nun genauer anschauen.

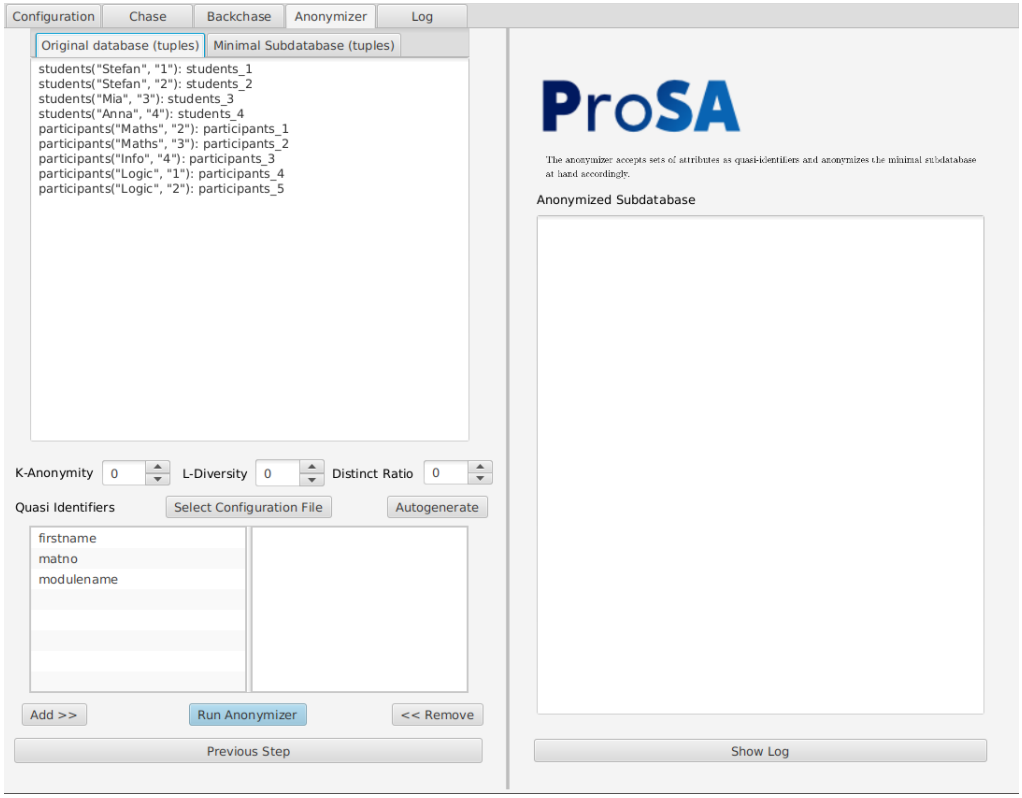


Abbildung 4.9: Der Anonymizer-Tab mit der vom Parser ausgelesenen Originaldatenbank.

4.10.2 Modus 2 (mit ProSA-Provenance)

Wir starten hier in der Pipeline, wie auch eben für Modus 1, nach dem Parser-Schritt. Als Erstes müssen wir also die Eingabe für die Chase-Phase so umschreiben, damit bei diesem Schritt ProSA-Provenance gesammelt werden kann.

Einarbeiten von ProSA-Provenance in die Chase-Eingabe Das Verfahren, um die Chase-Eingabe umzuschreiben, wurde schon in Abschnitt 4.3.1 beschrieben. Wie nehmen also die originale s-t tgd
$\forall matno, modulename : \texttt{students}(\texttt{'Stefan'}, matno) \wedge \texttt{participants}(modulename, matno)$
$\rightarrow \texttt{Result}(\texttt{'Stefan'}, modulename)$
und schreiben sie folgendermaßen um:
$\forall matno, modulename, students\_id, participants\_id :$
$\texttt{students}(\texttt{'Stefan'}, matno, students\_id) \wedge \texttt{participants}(modulename, matno, participants\_id)$
$\rightarrow \texttt{Result}(\texttt{'Stefan'}, modulename, students\_id, participants\_id).$

Nachdem die Tupel-IDs auch in Schema und Instanz eingearbeitet wurden, erhalten wir unsere Eingabe-XML für den Chase-Schritt (Listing 6.5). Mit dieser Eingabe wird nun eigentlich identisch zu Modus 1 die restliche Pipeline ausgeführt. Der einzige Unterschied ist, dass die Ergebnisse natürlich an anderen Stellen, bzw. gar nicht in der GUI ausgegeben werden.

Ergebnis der Chase-Phase: Das Vorgehen hier ist gleich zu Modus 1, nur werden die where-, why- und how-Provenance zwar auch mitberechnet, aber nicht in der GUI ausgegeben. Außerdem wird die Ergebnisinstanz dieses Mal natürlich im anderen Textfeld „*With ProSA provenance*“ des Chase-Tabs ausgeben (Abb. 4.10). Im Vergleich zu Modus 1 sehen wir hier ein zusätzliches Tupel im Ergebnis, weil `Result('Logic', 'Stefan')` jetzt durch die zusätzlichen Attribute `students_id` und `participants_id` vom Chase-Algorithmus nicht mehr als Duplikat gewertet wird.

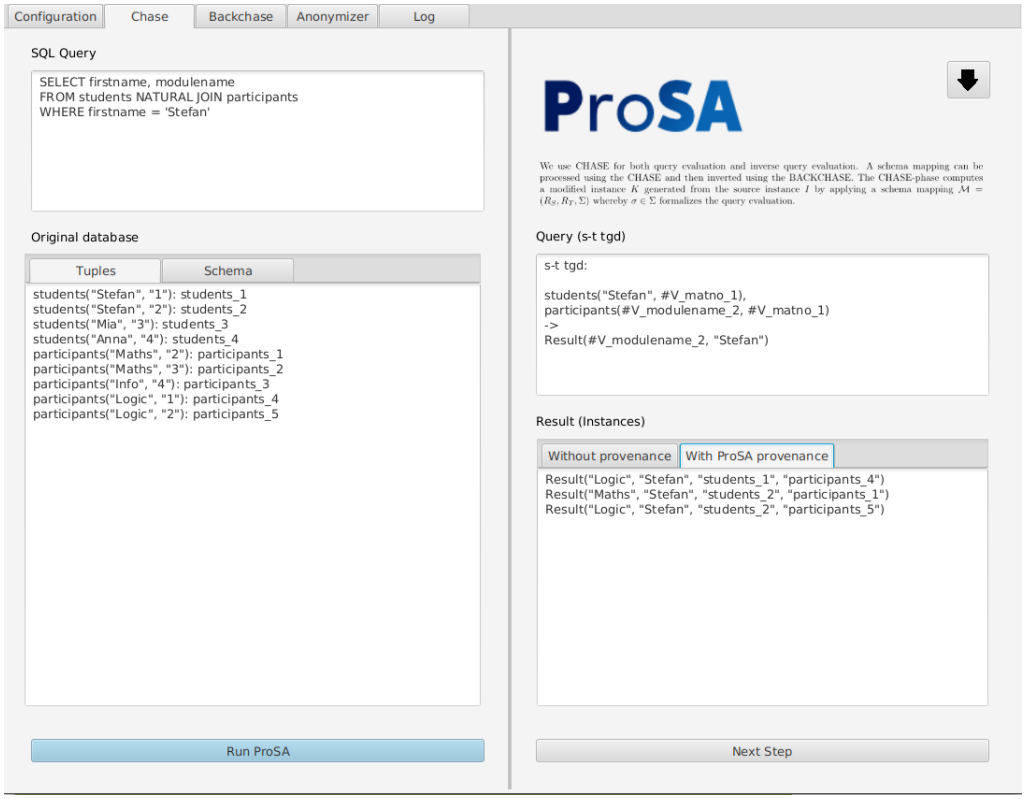


Abbildung 4.10: Die Ausgabe des Chase-Ergebnisses von Modus 2 im Chase-Tab.

Die zugehörige XML-Datei zu diesem Chase-Ergebnis sehen wir in Listing 6.6, in dem die where-, why- und how-Provenance enthalten sind, obwohl diese gar nicht in der GUI ausgegeben werden. Dies dient u.a. dem Zweck des Debuggings und einer zusätzlichen Absicherung des Unit-Tests.[21]

Invertierung der s-t tgd Vom Invertierer erhalten wir auf dieselbe Weise wie in Modus 1 folgende Inverse:
$\forall modulename_2, students\_id_1, participants\_id_1$:
$\texttt{Result('Stefan'}, modulename_2, students\_id_1, participants\_id_1)$
$\rightarrow \exists matno_1 : \texttt{students('Stefan'}, matno_1, students\_id_1)$
$\wedge\, \texttt{participants}(modulename_2, matno_1, participants\_id_1)$.
Anders als bei Modus 1 wird diese Inverse aber in der GUI *nicht* ausgegeben.

Eingabe-XML für die Backchase-Phase Die Kombinierung der Eingabe für den Backchase erfolgt auf dieselbe Weise wie in Modus 1. Die kombinierte XML-Datei befindet sich in Listing 6.7.

Ergebnis der Backchase-Phase: Im Gegensatz zu Modus 1 wird die Ergebnisinstanz der Backchase-Phase hier in der GUI an zwei Stellen ausgegeben: Zum Einen im Backchase-Tab im Textfeld „*With ProSA*

[21] Insbesondere bei der Ausgabe der Provenance im XML gab es in der Vergangenheit ja bereits Probleme, weil die Reihenfolge hier nicht determininistisch war (vgl. 4.8.2).

Provenance“ (Abb. 4.11) und zum Anderen im Anonymisierer-Tab oben links (Abb. 4.12). Wie stellen hier folgendes Problem fest: die berechnete Teildatenbank enthält ein Duplikat des `students_2`-Tupels mit einem unterschiedlichen Nullwert. Das darf eigentlich nicht sein, weil die Tupel-IDs Schlüsseleigenschaft haben. Der Grund dafür ist, dass die Inverse nicht ganz korrekt formuliert wurde. Die Tupel-ID `students_id` wird nämlich als normales Attribut behandelt, obwohl es eine Schlüsseleigenschaft hat. In dieser Form ist die minimale Teildatenbank mit ProSA-Provenance also nicht zu gebrauchen. Ein Lösungsansatz für dieses Problem wird im Ausblickskapitel 5.1.1 vorgestellt. Das Backchase-Ergebnis im XML-Format ist in Listing 6.8 zu sehen.

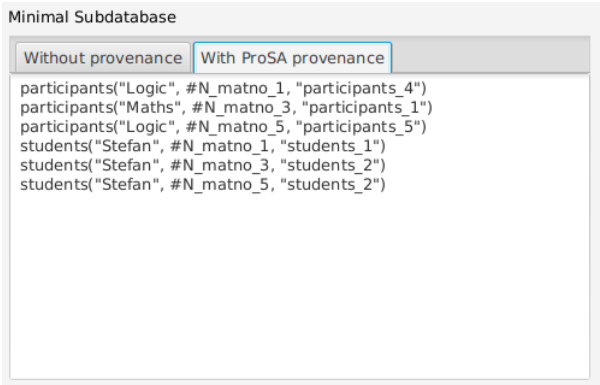


Abbildung 4.11: Die Ausgabe der minimalen Teildatenbank von Modus 2 im Backchase-Tab.

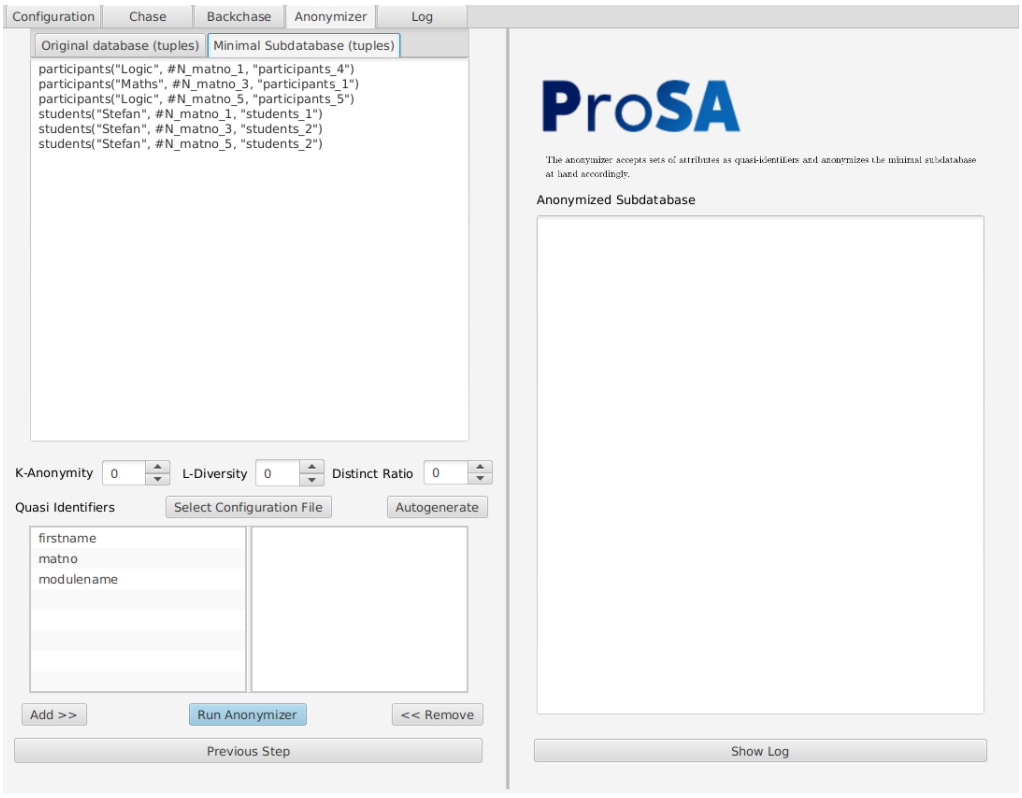


Abbildung 4.12: Die Ausgabe des der minimalen Teildatenbank von Modus 2 im Anonymisierer-Tab.

Damit sind wir am Ende der Pipeline für Modus 2, denn der Aufruf des Anonymisierers ist nicht mehr Teil des Beispiels. Wir haben uns ausführlich mit den Zwischenschritten für beide Pipelines befasst und schließen damit auch das Gesamtkapitel für Konzept und Umsetzung des Frameworks ab.
Darin haben wir gelernt, wie die Architektur des fertigen Frameworks aufgebaut ist, und deren Implementierung in diversen Code-Beispiele Schritt für Schritt erklärt. Außerdem haben wir die ProSA-Provenance als neue Art von Data-Provenance vorgestellt und weitere im Laufe der Arbeit angefallene Änderungen aufgelistet. Abschließend haben wir uns das durchgehende Beispiel mitsamt Unit-Test angesehen. Im nächsten Kapitel wollen wir einen Ausblick auf künftige Arbeiten werfen, die für ProSA noch anstehen und ein abschließendes Fazit ziehen.

5 Ausblick und Fazit

In diesem Kapitel beschäftigen wir uns zunächst im Ausblick mit den offenen Baustellen, die im Rahmen dieser Arbeit noch nicht gelöst werden konnten. Dann ziehen wir ein Fazit darüber, welche unserer Vorhaben wir umsetzen konnten, welche Schwierigkeiten es dabei gab und was wir dabei gelernt haben.

5.1 Ausblick

5.1.1 Mögliche Verbesserungen an ProSA

Vermeidung von Duplikaten bei der ProSA-Provenance

Wie wir im Unit-Test des durchlaufenden Beispiels festgestellt haben, liefert der Invertierer in der aktuellen Version für die umgeschriebene s-t tgd eine Inverse, die Duplikate in der minimalen Teildatenbank erzeugt. Der Grund dafür ist, dass der Invertierer nicht weiß, dass die als reguläre Attribute hinzugefügten Tupel-IDs Schlüsseleigenschaften besitzen. Betrachten wir wieder unsere umgeschriebene s-t tgd des durchgehenden Beispiels

$$\forall matno, modulename, students\_id, participants\_id: \\ \texttt{students}(\texttt{'Stefan'}, matno, students\_id) \wedge \texttt{participants}(modulename, matno, participants\_id) \\ \rightarrow \texttt{Result}(\texttt{'Stefan'}, modulename, students\_id, participants\_id)$$

so bietet sich ad hoc folgender Lösungsansatz an:

Definition der Schlüsseleigenschaft als zusätzliche egds Wir können für jedes Atom aus dem Kopf der s-t tgd definieren, dass die Tupel-IDs eindeutig sein müssen, indem wir pro Atom eine egd definieren. In unserem Beispiel wären das jeweils eine zusätzliche egd für die `students`-Relation und auch eine für die `participants`-Relation:

- $\forall firstname_1, firstname_2, matno_1, matno_2, students\_id:$
  $\texttt{students}(firstname_1, matno_1, students\_id) \wedge \texttt{students}(firstname_2, matno_2, students\_id)$
  $\rightarrow firstname_1 = firstname_2, matno_1 = matno_2$
- $\forall modulename_1, modulename_2, matno_1, matno_2, participants\_id:$
  $\texttt{participants}(modulename_1, matno_1, participants\_id)$
  $\wedge \texttt{participants}(modulename_2, matno_2, participants\_id)$
  $\rightarrow modulename_1 = modulename_2, matno_1 = matno_2$

Dadurch wird die Schlüsseleigenschaft der Tupel-ID-Attribute repräsentiert. Wenn wir diese egds der Inversen hinzufügen, werden dadurch die Duplikate eliminiert. Wir benutzen dieses Verfahren hier nur für unsere selbst hinzugefügten Schlüsselattribute (die Tupel-IDs), prinzipiell ist es aber auch erweiterbar auf „normale Schlüsselattribute“, für die der Parser gleich beim Auslesen aus der Postgres-Datenbank die passenden egds definieren kann.

Erweiterung der ProSA-Provenance für komplexere Anfragen

Der Parser kann bereits komplexere SQL-Anfragen in EDs umformen, beispielsweise geschachtelte Anfragen. Dann bestehen die Abhängigkeiten aber nicht mehr nur aus einer einzigen s-t tgd, sondern aus mindestens einer s-t tgd und einer tgd. Dann ist die ProSA-Provenance aber nicht mehr anwendbar, weil sie zum aktuellen Stand nur mit einer einzigen s-t tgd umgehen kann. Das Verfahren sollte also in dieser Hinsicht erweitert werden.

Integrierung des Anonymisierers

Der in [Sch22] entwickelte Anonymisierer muss in das Framework eingebaut werden. Hierzu haben wir bereits eine leere Methode in der Klasse `BackChased` angelegt, die mit den in der GUI eingegebenen Parametern aufgerufen wird (Listing 5.1).

Listing 5.1: Methodenvorlage für die Anonymisierung (Klasse `BackChased`)

```
1 private Anonymized anonymize(Integer k_Anonymity, Integer l_Diversity,
2         Double distinctRatio, LinkedHashSet<LinkedHashSet<String>> quasiIdentifiers,
3         LinkedHashSet<String> attributeWeights) {
4
5     System.out.println("Running empty anonymizer...");
6     // TODO Implement anonymizer here.
7 }
```

Berechnung der Inversen-Typs

Eine weitere geplante Erweiterung ist die Berechnung des Typs der Inverse. Das Textfeld auf dem Backchase-Tab gibt es schon (vgl. 4.3), wird aber noch konstant auf `undefined` gesetzt. Die Logik hierfür kann in der Methodenvorlage `detectInverseType(String sqlQuery)` implementiert werden (Listing 5.2).

Listing 5.2: Methodenvorlage für die Berechnung des Inversentyps (Klasse `MainGuiController`)

```
1 private String detectInverseType(String sqlQuery) {
2     String inverseType = "undefined";
3     //TODO implement inverse type detection here...
4
5     return inverseType;
6 }
```

Niedrigpriorisierte Verbesserungen

Saubere Kapselung des Parsers Der Parser lag ursprünglich als eigenes Maven-Projekt vor, konnte aber wegen Problemen mit seinen Bibliotheken nicht als solche externe Abhängigkeit eingebunden werden (siehe Abschnitt 4.8.1). Stattdessen wurde der Code des Parsers vorerst *direkt* in ProSA übernommen. Hier sollten in Folgearbeiten also entsprechend der Parser und ProSA entkoppelt werden, was der Austauschbarkeit entgegenkommt.

5.1.2 Mögliche Verbesserungen an ChaTEAU

Kopieren von `Instance` und `SingleInput`

Da der Konstruktor zum Kopieren einer `Instance` kaputt ist, und nicht wirklich ein unabhängiges, neues Objekt erzeugt, sollte dieser angepasst werden. Dasselbe gibt für die Klasse `SingleInput`, die noch nicht korrekt kopiert werden kann. Als Workaround wurde in dieser Arbeit Serialisierung und Deserialisierung dieser Objekte zum Kopieren benutzt (siehe Abschnitt 4.10).

Fehlerhaftes Verhalten bei zusammengesetzten `SingleInput`-Objekten

ChaTEAU arbeitet nicht korrekt, wenn das eingegebene Objekt von Typ `SingleInput` nicht vom `InputReader` aus einer Datei eingelesen wird, sondern „manuell“ aus `Instance` und `constraints` zusammengesetzt wird. Das Problem liegt vermutlich daran, dass ChaTEAU die Homomorphismen nicht auf Grundlage des Inhalts, sondern der Referenzen der Objekte findet, die nur beim Einlesen mit dem `InputReader` korrekt gesetzt werden. Dies sollte behoben werden, da (De-)Serialisierung sehr ineffizient ist.

5.1.3 Fazit

In dieser Arbeit haben wir die vorliegenden Komponenten für ProSA zu einem funktionierenden Gesamt-Framework zusammengesetzt. Dabei waren die häufige Änderung von Datenformaten und Konzepten, sowie der Abgleich mit den parallel laufenden Arbeiten am Parser[1], Invertierer[2] und Anonymisierer[3] eine große Herausforderung, die wir aber durch regelmäßige Treffen und Absprachen bewältigen konnten. Ein Lerneffekt der Arbeit ist definitiv, mit der Implementierung von Anwendungslogik nicht direkt im Code der GUI zu starten, weil man sie dort im Nachhinein nur unter großem Aufwand wieder „rauskriegt“.
Mit der stark an Perm[4] angelegten ProSA-Provenance konnten wir ein Konzept entwickeln, welches die Herkunft von Ergebnissen noch ein bisschen genauer als how-Provenance nachvollziehen kann. Dieses Konzept ist aber in ProSA noch nicht vollständig implementiert (siehe 5.1.1).
Wir konnten unser ursprüngliches Ziel erreichen und für simple `SELECT-FROM-WHERE`-Anfragen die minimale Teildatenbank berechnen. Durch das Design der Architektur sind die Anforderungen an Effizienz erreicht, da beide Pipelines nebenläufig ausgeführt werden können. Die erwünschte Modularität ist u.a. dadurch gewährleistet, dass die genutzten Klassen `Provenancer` und `Combiner` nur statische Methoden anbieten, und daher keine Abhängigkeiten haben. Außerdem begünstigt der `ProSAService` die Modularität, weil er auch komplett ohne GUI genutzt werden kann.

[1] [Kav22]
[2] [Spo22]
[3] [Sch22]
[4] [SRH+20]

6 Anhang

In diesem Kapitel finden sich die XML-Dateien mit den Zwischenschritten des durchgehenden Beispiels und weitere Dokumente, die im Rahmen dieser Arbeit Erwähnung fanden. Die gesamte Implementierung, auf die sich auch die in diesem Dokument genannten Codeausschnitte beziehen, ist im Repository für ProSA im Branch `moritz_framework_master` zu finden: `https://git.informatik.uni-rostock.de/ta093/prosa/-/tree/moritz_framework_master`.

6.1 Zwischenschritte des durchgehenden Beispiels

Listing 6.1: Die Ausgabe des Parsers. Zu finden in `src/test/resources/xml/pipelineSteps_expected/parser_output.xml`

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<input>
    <schema>
        <relations>
            <relation name="students" tag="S">
                <attribute name="firstname" type="string"/>
                <attribute name="matno" type="int"/>
            </relation>
            <relation name="participants" tag="S">
                <attribute name="modulename" type="string"/>
                <attribute name="matno" type="int"/>
            </relation>
            <relation name="Result" tag="T">
                <attribute name="firstname" type="string"/>
                <attribute name="modulename" type="string"/>
            </relation>
        </relations>
        <dependencies>
            <sttgd>
                <body>
                    <atom name="students" negation="false">
                        <variable name="matno" type="V" index="1"/>
                        <constant name="firstname" value="Stefan"/>
                    </atom>
                    <atom name="participants" negation="false">
                        <variable name="modulename" type="V" index="2"/>
                        <variable name="matno" type="V" index="1"/>
                    </atom>
                </body>
                <head>
                    <atom name="Result" negation="false">
                        <variable name="modulename" type="V" index="2"/>
                        <constant name="firstname" value="Stefan"/>
                    </atom>
                </head>
```

```
            </sttgd>
        </dependencies>
    </schema>
    <instance>
        <atom name="students" id="students_1">
            <constant name="firstname" value="Stefan"/>
            <constant name="matno" value="1"/>
        </atom>
        <atom name="students" id="students_2">
            <constant name="firstname" value="Stefan"/>
            <constant name="matno" value="2"/>
        </atom>
        <atom name="students" id="students_3">
            <constant name="firstname" value="Mia"/>
            <constant name="matno" value="3"/>
        </atom>
        <atom name="students" id="students_4">
            <constant name="firstname" value="Anna"/>
            <constant name="matno" value="4"/>
        </atom>
        <atom name="participants" id="participants_1">
            <constant name="modulename" value="Maths"/>
            <constant name="matno" value="2"/>
        </atom>
        <atom name="participants" id="participants_2">
            <constant name="modulename" value="Maths"/>
            <constant name="matno" value="3"/>
        </atom>
        <atom name="participants" id="participants_3">
            <constant name="modulename" value="Info"/>
            <constant name="matno" value="4"/>
        </atom>
        <atom name="participants" id="participants_4">
            <constant name="modulename" value="Logic"/>
            <constant name="matno" value="1"/>
        </atom>
        <atom name="participants" id="participants_5">
            <constant name="modulename" value="Logic"/>
            <constant name="matno" value="2"/>
        </atom>
    </instance>
</input>
```

Listing 6.4: Die Ausgabe-XML der Backchase-Phase ohne ProSA-Provenance. Zu finden in `src/test/resources/xml/pipelineSteps_expected/backchase_result_without_provenance.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
  <schema>
    <relations>
      <relation name="students" tag="T">
        <attribute name="firstname" type="string" />
        <attribute name="matno" type="int" />
      </relation>
      <relation name="participants" tag="T">
        <attribute name="modulename" type="string" />
        <attribute name="matno" type="int" />
      </relation>
      <relation name="Result" tag="S">
        <attribute name="firstname" type="string" />
        <attribute name="modulename" type="string" />
```

Listing 6.2: Die Ausgabe-XML der Chase-Phase ohne ProSA-Provenance. Die where-, why- und how-Provenance wurden berechnet und mit ausgegeben. Außerden enthält diese XML auch die Quellschemata der Chase-Eingabe. Zu finden in `src/test/resources/xml/pipelineSteps_expected/chase_result_without_provenance.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
  <schema>
    <relations>
      <relation name="students" tag="S">
        <attribute name="firstname" type="string" />
        <attribute name="matno" type="int" />
      </relation>
      <relation name="participants" tag="S">
        <attribute name="modulename" type="string" />
        <attribute name="matno" type="int" />
      </relation>
      <relation name="Result" tag="T">
        <attribute name="firstname" type="string" />
        <attribute name="modulename" type="string" />
      </relation>
    </relations>
    <dependencies />
  </schema>
  <instance>
    <atom name="Result" id="Result_1">
      <constant name="modulename" value="Logic" />
      <constant name="firstname" value="Stefan" />
    </atom>
    <atom name="Result" id="Result_2">
      <constant name="modulename" value="Maths" />
      <constant name="firstname" value="Stefan" />
    </atom>
  </instance>
  <provenance>
    <atom name="Result" id="Result_1">
      <where>
        <tuples>
          <tuple id="participants_4" />
          <tuple id="students_1" />
          <tuple id="participants_5" />
          <tuple id="students_2" />
        </tuples>
        <relations>
          <relation name="participants" />
          <relation name="students" />
        </relations>
      </where>
      <why>
        <witnesses>
          <witness id="students_1" />
          <witness id="participants_4" />
        </witnesses>
        <witnesses>
          <witness id="students_2" />
          <witness id="participants_5" />
        </witnesses>
      </why>
      <how polynom="(students_1*participants_4)+(students_2*participants_5)" />
    </atom>
    <atom name="Result" id="Result_2">
      <where>
        <tuples>
          <tuple id="participants_1" />
          <tuple id="students_2" />
        </tuples>
        <relations>
          <relation name="participants" />
          <relation name="students" />
        </relations>
      </where>
      <why>
        <witnesses>
          <witness id="students_2" />
          <witness id="participants_1" />
        </witnesses>
      </why>
      <how polynom="(students_2*participants_1)" />
    </atom>
  </provenance>
</input>
```

Listing 6.3: Die Eingabe-XML für die Backchase-Phase ohne ProSA-Provenance. Zu finden in src/test/resources/xml/pipelineSteps_expected/backchase_input_without_provenance.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<input>
  <schema>
    <relations>
      <relation name="students" tag="T">
        <attribute name="firstname" type="string" />
        <attribute name="matno" type="int" />
      </relation>
      <relation name="participants" tag="T">
        <attribute name="modulename" type="string" />
        <attribute name="matno" type="int" />
      </relation>
      <relation name="Result" tag="S">
        <attribute name="firstname" type="string" />
        <attribute name="modulename" type="string" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="Result">
            <constant name="firstname" value="Stefan" />
            <variable name="modulename" type="V" index="2" />
          </atom>
        </body>
        <head>
          <atom name="students">
            <constant name="firstname" value="Stefan" />
            <variable name="matno" type="E" index="1" />
          </atom>
          <atom name="participants">
            <variable name="modulename" type="V" index="2" />
            <variable name="matno" type="E" index="1" />
          </atom>
        </head>
      </sttgd>
    </dependencies>
  </schema>
  <instance>
    <atom name="Result" id="Result_1">
      <constant name="firstname" value="Stefan" />
      <constant name="modulename" value="Logic" />
    </atom>
    <atom name="Result" id="Result_2">
      <constant name="firstname" value="Stefan" />
      <constant name="modulename" value="Maths" />
    </atom>
  </instance>
</input>
```

```
      </relation>
    </relations>
    <dependencies />
  </schema>
  <instance>
    <atom name="students">
      <constant name="firstname" value="Stefan" />
      <null name="matno" index="1" />
    </atom>
    <atom name="students">
      <constant name="firstname" value="Stefan" />
      <null name="matno" index="3" />
    </atom>
    <atom name="participants">
      <constant name="modulename" value="Logic" />
      <null name="matno" index="1" />
    </atom>
    <atom name="participants">
      <constant name="modulename" value="Maths" />
      <null name="matno" index="3" />
    </atom>
  </instance>
</input>
```

Listing 6.5: Die umgeschriebene Eingabe-XML für die Chase-Phase mit ProSA-Provenance. Zu finden in `src/test/resources/xml/pipelineSteps_expected/chase_input_with_provenance.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
  <schema>
    <relations>
      <relation name="students" tag="S">
        <attribute name="firstname" type="string" />
        <attribute name="matno" type="int" />
        <attribute name="students_id" type="string" />
      </relation>
      <relation name="participants" tag="S">
        <attribute name="modulename" type="string" />
        <attribute name="matno" type="int" />
        <attribute name="participants_id" type="string" />
      </relation>
      <relation name="Result" tag="T">
        <attribute name="firstname" type="string" />
        <attribute name="modulename" type="string" />
        <attribute name="Result_id" type="string" />
        <attribute name="students_id" type="string" />
        <attribute name="participants_id" type="string" />
      </relation>
    </relations>
    <dependencies>
      <sttgd>
        <body>
          <atom name="students">
            <constant name="firstname" value="Stefan" />
            <variable name="matno" type="V" index="1" />
            <variable name="students_id" type="V" index="1" />
          </atom>
          <atom name="participants">
            <variable name="modulename" type="V" index="2" />
            <variable name="matno" type="V" index="1" />
            <variable name="participants_id" type="V" index="1" />
```

```
35            </atom>
36          </body>
37          <head>
38            <atom name="Result">
39              <variable name="modulename" type="V" index="2" />
40              <constant name="firstname" value="Stefan" />
41              <variable name="students_id" type="V" index="1" />
42              <variable name="participants_id" type="V" index="1" />
43            </atom>
44          </head>
45        </sttgd>
46      </dependencies>
47    </schema>
48    <instance>
49      <atom name="students" id="students_1">
50        <constant name="firstname" value="Stefan" />
51        <constant name="matno" value="1" />
52        <constant name="students_id" value="students_1" />
53      </atom>
54      <atom name="students" id="students_2">
55        <constant name="firstname" value="Stefan" />
56        <constant name="matno" value="2" />
57        <constant name="students_id" value="students_2" />
58      </atom>
59      <atom name="students" id="students_3">
60        <constant name="firstname" value="Mia" />
61        <constant name="matno" value="3" />
62        <constant name="students_id" value="students_3" />
63      </atom>
64      <atom name="students" id="students_4">
65        <constant name="firstname" value="Anna" />
66        <constant name="matno" value="4" />
67        <constant name="students_id" value="students_4" />
68      </atom>
69      <atom name="participants" id="participants_1">
70        <constant name="modulename" value="Maths" />
71        <constant name="matno" value="2" />
72        <constant name="participants_id" value="participants_1" />
73      </atom>
74      <atom name="participants" id="participants_2">
75        <constant name="modulename" value="Maths" />
76        <constant name="matno" value="3" />
77        <constant name="participants_id" value="participants_2" />
78      </atom>
79      <atom name="participants" id="participants_3">
80        <constant name="modulename" value="Info" />
81        <constant name="matno" value="4" />
82        <constant name="participants_id" value="participants_3" />
83      </atom>
84      <atom name="participants" id="participants_4">
85        <constant name="modulename" value="Logic" />
86        <constant name="matno" value="1" />
87        <constant name="participants_id" value="participants_4" />
88      </atom>
89      <atom name="participants" id="participants_5">
90        <constant name="modulename" value="Logic" />
91        <constant name="matno" value="2" />
92        <constant name="participants_id" value="participants_5" />
93      </atom>
94    </instance>
95  </input>
```

Listing 6.6: Die Ausgabe-XML der Chase-Phase mit ProSA-Provenance mitsamt where-, why- und how-Provenance, sowie den Quellschemata der Chase-Phase. Zu finden in `src/test/resources/xml/pipelineSteps_expected/chase_result_with_provenance.xml`

```xml
<?xml version="1.0" encoding="UTF-8"?>
<input>
  <schema>
    <relations>
      <relation name="students" tag="S">
        <attribute name="firstname" type="string" />
        <attribute name="matno" type="int" />
        <attribute name="students_id" type="string" />
      </relation>
      <relation name="participants" tag="S">
        <attribute name="modulename" type="string" />
        <attribute name="matno" type="int" />
        <attribute name="participants_id" type="string" />
      </relation>
      <relation name="Result" tag="T">
        <attribute name="firstname" type="string" />
        <attribute name="modulename" type="string" />
        <attribute name="Result_id" type="string" />
        <attribute name="students_id" type="string" />
        <attribute name="participants_id" type="string" />
      </relation>
    </relations>
    <dependencies />
  </schema>
  <instance>
    <atom name="Result" id="Result_1">
      <constant name="modulename" value="Logic" />
      <constant name="firstname" value="Stefan" />
      <constant name="students_id" value="students_1" />
      <constant name="participants_id" value="participants_4" />
    </atom>
    <atom name="Result" id="Result_2">
      <constant name="modulename" value="Maths" />
      <constant name="firstname" value="Stefan" />
      <constant name="students_id" value="students_2" />
      <constant name="participants_id" value="participants_1" />
    </atom>
    <atom name="Result" id="Result_3">
      <constant name="modulename" value="Logic" />
      <constant name="firstname" value="Stefan" />
      <constant name="students_id" value="students_2" />
      <constant name="participants_id" value="participants_5" />
    </atom>
  </instance>
  <provenance>
    <atom name="Result" id="Result_1">
      <where>
        <tuples>
          <tuple id="participants_4" />
          <tuple id="students_1" />
        </tuples>
        <relations>
          <relation name="participants" />
          <relation name="students" />
        </relations>
      </where>
      <why>
```

```
58          <witnesses>
59            <witness id="students_1" />
60            <witness id="participants_4" />
61          </witnesses>
62        </why>
63        <how polynom="(students_1*participants_4)" />
64      </atom>
65      <atom name="Result" id="Result_2">
66        <where>
67          <tuples>
68            <tuple id="participants_1" />
69            <tuple id="students_2" />
70          </tuples>
71          <relations>
72            <relation name="participants" />
73            <relation name="students" />
74          </relations>
75        </where>
76        <why>
77          <witnesses>
78            <witness id="students_2" />
79            <witness id="participants_1" />
80          </witnesses>
81        </why>
82        <how polynom="(students_2*participants_1)" />
83      </atom>
84      <atom name="Result" id="Result_3">
85        <where>
86          <tuples>
87            <tuple id="participants_5" />
88            <tuple id="students_2" />
89          </tuples>
90          <relations>
91            <relation name="participants" />
92            <relation name="students" />
93          </relations>
94        </where>
95        <why>
96          <witnesses>
97            <witness id="students_2" />
98            <witness id="participants_5" />
99          </witnesses>
100       </why>
101       <how polynom="(students_2*participants_5)" />
102     </atom>
103   </provenance>
104 </input>
```

Listing 6.7: Die Eingabe-XML für die Backchase-Phase mit ProSA-Provenance. Zu finden in src/test/resources/xml/pipelineSteps_expected/backchase_input_with_provenance.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <input>
3    <schema>
4      <relations>
5        <relation name="students" tag="T">
6          <attribute name="firstname" type="string" />
7          <attribute name="matno" type="int" />
8          <attribute name="students_id" type="string" />
9        </relation>
10       <relation name="participants" tag="T">
```

```
11          <attribute name="modulename" type="string" />
12          <attribute name="matno" type="int" />
13          <attribute name="participants_id" type="string" />
14        </relation>
15        <relation name="Result" tag="S">
16          <attribute name="firstname" type="string" />
17          <attribute name="modulename" type="string" />
18          <attribute name="Result_id" type="string" />
19          <attribute name="students_id" type="string" />
20          <attribute name="participants_id" type="string" />
21        </relation>
22      </relations>
23      <dependencies>
24        <sttgd>
25          <body>
26            <atom name="Result">
27              <constant name="firstname" value="Stefan" />
28              <variable name="modulename" type="V" index="2" />
29              <variable name="students_id" type="V" index="1" />
30              <variable name="participants_id" type="V" index="1" />
31            </atom>
32          </body>
33          <head>
34            <atom name="students">
35              <constant name="firstname" value="Stefan" />
36              <variable name="matno" type="E" index="1" />
37              <variable name="students_id" type="V" index="1" />
38            </atom>
39            <atom name="participants">
40              <variable name="modulename" type="V" index="2" />
41              <variable name="matno" type="E" index="1" />
42              <variable name="participants_id" type="V" index="1" />
43            </atom>
44          </head>
45        </sttgd>
46      </dependencies>
47    </schema>
48    <instance>
49      <atom name="Result" id="Result_1">
50        <constant name="firstname" value="Stefan" />
51        <constant name="modulename" value="Logic" />
52        <constant name="students_id" value="students_1" />
53        <constant name="participants_id" value="participants_4" />
54      </atom>
55      <atom name="Result" id="Result_2">
56        <constant name="firstname" value="Stefan" />
57        <constant name="modulename" value="Maths" />
58        <constant name="students_id" value="students_2" />
59        <constant name="participants_id" value="participants_1" />
60      </atom>
61      <atom name="Result" id="Result_3">
62        <constant name="firstname" value="Stefan" />
63        <constant name="modulename" value="Logic" />
64        <constant name="students_id" value="students_2" />
65        <constant name="participants_id" value="participants_5" />
66      </atom>
67    </instance>
68  </input>
```

Listing 6.8: Die Ausgabe-XML der Backchase-Phase mit ProSA-Provenance. Zu finden in src/test/resources/xml/pipelineSteps_expected/backchase_result_with_provenance.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<input>
  <schema>
    <relations>
      <relation name="students" tag="T">
        <attribute name="firstname" type="string" />
        <attribute name="matno" type="int" />
        <attribute name="students_id" type="string" />
      </relation>
      <relation name="participants" tag="T">
        <attribute name="modulename" type="string" />
        <attribute name="matno" type="int" />
        <attribute name="participants_id" type="string" />
      </relation>
      <relation name="Result" tag="S">
        <attribute name="firstname" type="string" />
        <attribute name="modulename" type="string" />
        <attribute name="Result_id" type="string" />
        <attribute name="students_id" type="string" />
        <attribute name="participants_id" type="string" />
      </relation>
    </relations>
    <dependencies />
  </schema>
  <instance>
    <atom name="students">
      <constant name="firstname" value="Stefan" />
      <null name="matno" index="1" />
      <constant name="students_id" value="students_1" />
    </atom>
    <atom name="students">
      <constant name="firstname" value="Stefan" />
      <null name="matno" index="3" />
      <constant name="students_id" value="students_2" />
    </atom>
    <atom name="students">
      <constant name="firstname" value="Stefan" />
      <null name="matno" index="5" />
      <constant name="students_id" value="students_2" />
    </atom>
    <atom name="participants">
      <constant name="modulename" value="Logic" />
      <null name="matno" index="1" />
      <constant name="participants_id" value="participants_4" />
    </atom>
    <atom name="participants">
      <constant name="modulename" value="Maths" />
      <null name="matno" index="3" />
      <constant name="participants_id" value="participants_1" />
    </atom>
    <atom name="participants">
      <constant name="modulename" value="Logic" />
      <null name="matno" index="5" />
      <constant name="participants_id" value="participants_5" />
    </atom>
  </instance>
</input>
```

6.2 Sonstiges

Listing 6.9: Die XSD, die das gültige XML-Eingabeformat für ChaTEAU definiert. Zu finden in `src/main/resources/xsd/input.xsd`

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="input">
        <xs:complexType>
            <xs:all>
                <xs:element name="schema" type="schema" minOccurs="0"/>
                <xs:element name="instance" type="instance" minOccurs="0"/>
                <xs:element name="query" type="rule" minOccurs="0"/>
            </xs:all>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="schema">
        <xs:all>
            <xs:element name="relations" type="relations"/>
            <xs:element name="dependencies" type="dependencies"/>
        </xs:all>
    </xs:complexType>

    <xs:complexType name="relations">
        <xs:sequence>
            <xs:element name="relation" type="relation" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="relation">
        <xs:sequence>
            <xs:element name="attribute" type="attribute" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="tag" type="tag"/>
    </xs:complexType>

    <xs:simpleType name="tag">
        <xs:restriction base="xs:string">
            <xs:enumeration value="S"/> <!-- source -->
            <xs:enumeration value="T"/> <!-- target -->
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="attribute">
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="type" type="attribute-type" use="required"/>
    </xs:complexType>

    <xs:simpleType name="attribute-type">
        <xs:restriction base="xs:string">
            <xs:enumeration value="string"/>
            <xs:enumeration value="int"/>
            <xs:enumeration value="double"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="dependencies">
        <xs:sequence>
```

```
57              <xs:element name="egd" type="rule" minOccurs="0" maxOccurs="unbounded"/>
58              <xs:element name="tgd" type="rule" minOccurs="0" maxOccurs="unbounded"/>
59              <xs:element name="sttgd" type="rule" minOccurs="0" maxOccurs="unbounded"
                    />
60          </xs:sequence>
61      </xs:complexType>
62
63      <xs:complexType name="rule">
64          <xs:sequence>
65              <xs:element name="body" type="atoms"/>
66              <xs:element name="body-comparisons" type="comparisons"/>
67              <xs:element name="head" type="atoms"/>
68              <xs:element name="head-comparisons" type="comparisons"/>
69          </xs:sequence>
70      </xs:complexType>
71
72      <xs:complexType name="atoms">
73          <xs:sequence>
74              <xs:element name="atom" type="atom" maxOccurs="unbounded"/>
75          </xs:sequence>
76      </xs:complexType>
77
78      <xs:complexType name="atom">
79          <xs:sequence>
80              <xs:element name="variable" type="variable" maxOccurs="unbounded"
                    minOccurs="0"/>
81              <xs:element name="constant" type="constant" maxOccurs="unbounded"
                    minOccurs="0"/>
82              <xs:element name="null" type="null" maxOccurs="unbounded" minOccurs="0"/>
83          </xs:sequence>
84          <xs:attribute name="name" type="xs:string" default=""/>
85          <xs:attribute name="negation" type="xs:boolean" default="false"/>
86      </xs:complexType>
87
88      <xs:complexType name="variable">
89          <xs:attribute name="name" type="xs:string" use="required"/>
90          <xs:attribute name="type" type="quantifier"/>
91          <xs:attribute name="index" type="xs:int" default="0"/>
92      </xs:complexType>
93
94      <xs:simpleType name="quantifier">
95          <xs:restriction base="xs:string">
96              <xs:enumeration value="V"/> <!-- all -->
97              <xs:enumeration value="E"/> <!-- exists -->
98          </xs:restriction>
99      </xs:simpleType>
100
101     <xs:complexType name="constant">
102         <xs:attribute name="name" type="xs:string" use="required"/>
103         <xs:attribute name="value" type="xs:string" use="required"/>
104     </xs:complexType>
105
106     <xs:complexType name="null">
107         <xs:attribute name="name" type="xs:string" use="required"/>
108         <xs:attribute name="index" type="xs:int" default="0"/>
109     </xs:complexType>
110
111     <xs:complexType name="comparisons">
112         <xs:sequence>
113             <xs:element name="comparison" type="comparison" maxOccurs="unbounded"/>
114         </xs:sequence>
```

```
115     </xs:complexType>
116
117     <xs:complexType name="comparison">
118         <xs:all>
119             <xs:element name="left">
120                 <xs:complexType>
121                     <xs:all>
122                         <xs:element name="variable" type="variable"/>
123                     </xs:all>
124                 </xs:complexType>
125             </xs:element>
126             <xs:element name="right" type="expression"/>
127         </xs:all>
128         <xs:attribute name="operator" type="comparison-operators" use="required"/>
129     </xs:complexType>
130
131     <xs:simpleType name="comparison-operators">
132         <xs:restriction base="xs:string">
133             <xs:enumeration value="greater"/>
134             <xs:enumeration value="less"/>
135             <xs:enumeration value="greater-equal"/>
136             <xs:enumeration value="less-equal"/>
137             <xs:enumeration value="not-equal"/>
138             <xs:enumeration value="equal"/>
139         </xs:restriction>
140     </xs:simpleType>
141
142     <xs:complexType name="expression">
143         <xs:choice>
144             <xs:element name="variable" type="variable"/>
145             <xs:element name="null" type="null"/>
146             <xs:element name="math-constant" type="math-constant"/>
147             <xs:element name="function" type="function"/>
148         </xs:choice>
149     </xs:complexType>
150
151     <xs:complexType name="math-constant">
152         <xs:attribute name="value" type="xs:string" use="required"/>
153     </xs:complexType>
154
155     <xs:complexType name="function">
156         <xs:all>
157             <xs:element name="left" type="expression"/>
158             <xs:element name="right" type="expression"/>
159         </xs:all>
160         <xs:attribute name="operator" type="function-operators" use="required"/>
161     </xs:complexType>
162
163     <xs:simpleType name="function-operators">
164         <xs:restriction base="xs:string">
165             <xs:enumeration value="addition"/>
166             <xs:enumeration value="subtraction"/>
167             <xs:enumeration value="multiplication"/>
168             <xs:enumeration value="division"/>
169             <xs:enumeration value="minimum"/>
170             <xs:enumeration value="maximum"/>
171         </xs:restriction>
172     </xs:simpleType>
173
174     <xs:complexType name="instance">
175         <xs:sequence>
```

```
176             <xs:element name="atom" type="instance-atom" maxOccurs="unbounded"/>
177         </xs:sequence>
178     </xs:complexType>
179
180     <xs:complexType name="instance-atom">
181         <xs:sequence>
182             <xs:element name="constant" type="constant" maxOccurs="unbounded"
                    minOccurs="0"/>
183             <xs:element name="null" type="null" maxOccurs="unbounded" minOccurs="0"/>
184         </xs:sequence>
185         <xs:attribute name="name" type="xs:string" default=""/>
186     </xs:complexType>
187
188 </xs:schema>
```

Listing 6.10: Das SQL-Skript, um die Datenbank für das durchgehende Beispiel zu erstellen. Zu finden in `src/test/resources/sql/setUpTestDB.sql`

```
1  -- Very small database with two tables "students" and "participants" to test a full
       pipeline run.
2  -- Note: When using ‘psql‘, first connect to the correct database ’pipelinedb’ before
        executing this script: ‘\c pipelinedb‘
3  -- or directly call ‘psql pipelinedb‘.
4
5
6  ALTER TABLE IF EXISTS ONLY "participants"
7      DROP CONSTRAINT IF EXISTS participants_students_matNo_fk;
8  ALTER TABLE IF EXISTS ONLY "students"
9      DROP CONSTRAINT IF EXISTS students_pk;
10 DROP TABLE IF EXISTS "students";
11 DROP TABLE IF EXISTS "participants";
12
13 CREATE TABLE "students"
14 (
15     firstname text    NOT NULL,
16     matno     integer NOT NULL
17 );
18
19 CREATE TABLE "participants"
20 (
21     modulename text    NOT NULL,
22     matno      integer NOT NULL
23 );
24
25 INSERT INTO "students" VALUES
26             ('Stefan', 1),
27             ('Stefan', 2),
28             ('Mia'   , 3),
29             ('Anna'  , 4);
30
31 INSERT INTO "participants" VALUES
32             ('Maths', 2),
33             ('Maths', 3),
34             ('Info' , 4),
35             ('Logic', 1),
36             ('Logic', 2);
37
38
39 ALTER TABLE ONLY "students"
40     ADD CONSTRAINT students_pk PRIMARY KEY (matno);
41
```

```
42 ALTER TABLE ONLY "participants"
43     ADD CONSTRAINT participants_students_matNo_fk FOREIGN KEY (matno)
44         REFERENCES "students" (matno);
```

Literaturverzeichnis

[AH] Auge, Tanja ; Heuer, Andreas: *Interne Notizen zu ChaTEAU*. Unveröffentlicht,

[AH19] Auge, Tanja ; Heuer, Andreas: ProSA - Using the CHASE for Provenance Management. In: *ADBIS 2019*, 2019, 357–372

[Aug17] Auge, Tanja: *Umsetzung von Provenance-Anfragen in Big-Data-Analytics-Umgebungen*, Universität Rostock, Diplomarbeit, September 2017. http://eprints.dbis.informatik.uni-rostock.de/880/

[Aug22] Auge, Tanja: *Dissertation zu Provenance Management*. Unveröffentlicht, 2022

[AWB16] Auge, Tanja ; Wilsdorf, Pia ; Brossmann, Sabrina: *Neueste Entwicklungen der Informatik - Abschlusspräsentation Cluster Provenance*. September 2016. – NEidI-Projekt an der Universität Rostock

[BKM+17] Benedikt, Michael ; Konstantinidis, George ; Mecca, Giansalvatore ; Motik, Boris ; Papotti, Paolo ; Santoro, Donatello ; Tsamoura, Efthymia: Benchmarking the Chase. In: Sallinger, Emanuel (Hrsg.) ; Bussche, Jan V. (Hrsg.) ; Geerts, Floris (Hrsg.): *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, ACM, 2017, 37–52

[BKT01] Buneman, Peter ; Khanna, Sanjeev ; Tan, Wang C.: Why and Where: A Characterization of Data Provenance. In: Bussche, Jan V. (Hrsg.) ; Vianu, Victor (Hrsg.): *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings* Bd. 1973, Springer, 2001 (Lecture Notes in Computer Science), 316–330

[CCT09] Cheney, James ; Chiticariu, Laura ; Tan, Wang C.: Provenance in Databases: Why, How, and Where. In: *Found. Trends Databases* 1 (2009), Nr. 4, 379–474. http://dx.doi.org/10.1561/1900000006. – DOI 10.1561/1900000006

[FHO+21] Förster, Leonie ; Heuser, Melinda ; Overat, Judith-Henrike ; Waterstradt, Anne-Sophie ; Wolpers, Anja: *Data Provenance*. http://eprints.dbis.informatik.uni-rostock.de/1052/. Version: September 2021

[GMS12] Greco, Sergio ; Molinaro, Cristian ; Spezzano, Francesca: *Incomplete Data and Data Dependencies in Relational Databases*. Bd. 4. 2012. http://dx.doi.org/10.2200/S00435ED1V01Y201207DTM029. http://dx.doi.org/10.2200/S00435ED1V01Y201207DTM029

[Gör20] Görres, Andreas: *Erweiterung des CHASE-Werkzeugs ChaTEAU um ein Terminierungskriterium*, Universität Rostock, Diplomarbeit, March 2020. http://eprints.dbis.informatik.uni-rostock.de/1009/

[GT17] Green, Todd J. ; Tannen, Val: The Semiring Framework for Database Provenance. In: Sallinger, Emanuel (Hrsg.) ; Bussche, Jan V. (Hrsg.) ; Geerts, Floris (Hrsg.): *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, ACM, 2017, 93–99

[Han22] Hanzig, Moritz: *Ein Framework für ProSA - Zwischenbericht.* January 2022. – Zwischenpräsentation im Rahmen des Forschungsseminars des Datenbankenlehrstuhls der Universität Rostock

[HDB17] Herschel, Melanie ; Diestelkämper, Ralf ; Ben Lahmar, Houssem: A survey on provenance: What for? What form? What from? In: *VLDB J.* 26 (2017), Nr. 6, 881–906. http://dx.doi.org/10.1007/s00778-017-0486-1. – DOI 10.1007/s00778–017–0486–1

[Jur18] Jurklies, Martin: *CHASE und BACKCHASE: Entwicklung eines Universal-Werkzeugs für eine Basistechnik der Datenbankforschung*, Universität Rostock, Diplomarbeit, October 2018. http://eprints.dbis.informatik.uni-rostock.de/968/

[Kav22] Kavisanczki, Ivo: *Erweiterung des ProSA-Parsers um Aggregatfunktionen.* Unveröffentlicht, 2022

[KRSZ21] Kavisanczki, Ivo ; Rudolph, Tobias ; Siegl, Tom ; Zuska, Marian: *sql2sttgd.* http://eprints.dbis.informatik.uni-rostock.de/1051/. Version: 2021

[Ora22] Oracle: *Java Interface Set<E> Documentation.* https://docs.oracle.com/javase/7/docs/api/java/util/Set.html, 2022. – Abgerufen: 2022-02-28

[Ren19] Renn, Fabian: *Erweiterung des CHASE-Werkzeugs ChaTEAU um Anfragetransformationen*, Universität Rostock, Diss., June 2019. http://eprints.dbis.informatik.uni-rostock.de/993/

[Ros20] Rose, Florian: *Erweiterung des CHASE-Werkzeugs ChaETAU um eine BACKCHASE-Phase*, Universität Rostock, Diplomarbeit, 2020. http://eprints.dbis.informatik.uni-rostock.de/1032/

[Sch20] Scharlau, Nic: *Provenance und Privacy in ProSA*, Universität Rostock, Diss., February 2020. http://eprints.dbis.informatik.uni-rostock.de/1010/

[Sch21] Scharlau, Nic: Der CHASE-Algorithmus: Ein Überblick. (2021)

[Sch22] Scharlau, Nic: *Anonymisierung von Data Provenance in ProSA.* Unveröffentlicht, 2022

[Spo22] Spolwind, Dennis: *Inverse Anfragen in ProSA.* Unveröffentlicht, 2022

[SRH+20] Strelnikov, Artur ; Röhrs, Chris ; Hermann, Leon ; Kaseler, Max ; Flach, Rocco: *Provenance-Tools - Technischer Bericht.* 2020

[Zim20] Zimmer, Jakob: *Vereinheitlichung des CHASE auf Instanzen und Anfragen am Beispiel ChaTEAU*, Universität Rostock, Diss., March 2020. http://eprints.dbis.informatik.uni-rostock.de/1008/

Anfragenverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommen Stellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Rostock, den 8. März 2022