Projektdokumentation sql2sttgd

Ivo Kavisanczki Tobias Rudolph Tom Siegl Marian Zuska

30. September 2021

Zusammenfassung

sql2sttgd ist eine Software zur Generierung von Eingabedaten für ChaTEAU. Dazu werden Schema und Instanz aus einer PostgreSQL-Datenbank ausgelesen und eine SQL-Anfrage in s-t tgds umgewandelt. Die berechneten Daten werden dann im XML-Format für ChaTEAU ausgegeben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einführung in den Chase und s-t tgds

Tobias Rudolph

In diesem ersten Abschnitt werden die theoretischen Grundlagen, die zum Verständnis der Funktionsweise unseres Projekts erforderlich sind, kurz erklärt. Diese Erläuterungen sind der Seminararbeit „Der CHASE-Algorithmus: Ein Überblick“ (Rostock, 2021) von Nic Scharlau sinngemäß entnommen.

1.1. Der Chase-Algorithmus

Der Chase ist ein Algorithmus, der u.a. funktionale Abhängigkeiten in Datenbanksystemen darstellen und Anfragen optimieren kann – im Grunde wird eine Menge von Parametern $\star$ in ein Objekt $\bigcirc$ eingearbeitet, sodass diese nun implizit im Objekt enthalten sind, wobei das Objekt bspw. eine Datenbankinstanz oder eine Anfrage darstellt:

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

Der Chase kann demnach sowohl auf Instanzen als auch auf Anfragen arbeiten, wobei wir in unserem Projekt aber nur die Anwendung auf Instanzen betrachten. Das Objekt ist in unserem Fall also immer eine Datenbankinstanz.

1.2. Source-to-Target Tuple Generating Dependencies (s-t tgds)

Zur Definition von s-t tgds sind zunächst die Definitionen von *eingebetteten Abhängigkeiten* und *tuple-generating dependencies* (tgds) nötig.

Eine **eingebettete Abhängigkeit** ist ein prädikatenlogischer Ausdruck 1. Stufe der Form

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

In der eingebetteten Abhängigkeit sind x, y, z Tupel aus Variablen und $\varphi(x, y)$ sowie $\varphi(x, z)$ Konjunktionen über atomare Formeln.

Eine **tgd** ist eine eingebettete Abhängigkeit der Form

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

F ist jeweils eine Konjunktion von atomaren Formeln mit Variablen aus x (F_1) bzw. x, y (F_2).

In einer **s-t tgd** ist F_1 nun eine Konjunktion über ein Quellschema S (source) und F_2 eine Konjunktion über ein Zielschema T (target), sie können also dazu verwendet werden, um ein Quellschema in ein Zielschema zu überführen.

2. Aufgabenstellung & Motivation

Tobias Rudolph

ChaTEAU ist eine Implementation des **Chase**-Algorithmus in Java, welche zum Projekt ProSA gehört. Die Software nimmt Eingaben in einem eigenen XML-Format an, in dem neben einem Datenbankschema und zugehöriger Instanz insbesondere Abhängigkeiten als **s-t tgds** dargestellt werden.

Das Projekt ProSA möchte die Arbeit mit großen Datenmengen in der Forschung durch **Data Provenance** unterstützen. Es verwendet zusätzlich den Chase, um gemeinsam mit Provenance-Informationen eine minimale Teildatenbank eines ursprünglichen Datensatzes zu erstellen. Diese Informationen lassen sich in drei Typen kategorisieren:

1. **where**: Woher kommen die Daten?
2. **why**: Warum dieses Ergebnis?
3. **how**: Wie wurde das Ergebnis berechnet?

Bisher ist die Erzeugung von Eingaben nur umständlich von Hand möglich. Es müssen also manuell Datenbankschema und -instanz abgetippt sowie die Umwandlung einer SQL-Anfrage in s-t tgds durchgeführt werden. Zudem muss der Autor selbst die XML-Datei mit korrektem Layout schreiben. Dadurch entstehen viele Gelegenheiten für menschliche Fehler und es wird ein **Medienbruch** in den Workflow für die Nutzung von ProSA / ChaTEAU eingeführt.

Um diese Probleme zu beheben, haben wir sql2sttgd entwickelt. Das Tool ermöglicht eine vollständige Automatisierung der genannten Abläufe. Dazu muss nur eine Datenbankverbindung zu einer PostgreSQL-DB und eine SQL-Anfrage bereitgestellt werden. Die Software erzeugt daraus dann Input im XML-Format, der bereit für die Nutzung in ChaTEAU ist.

3. Konzept

Tobias Rudolph

sql2sttgd ist in mehrere logische Module aufgeteilt, welche jeweils bestimmte Aufgaben bei der Generierung der finalen Ausgabe übernehmen. Abbildung 1 stellt die Module und deren Wechselwirkungen als Flowchart dar. In den weiteren Abschnitten werden die Module einzeln erklärt.

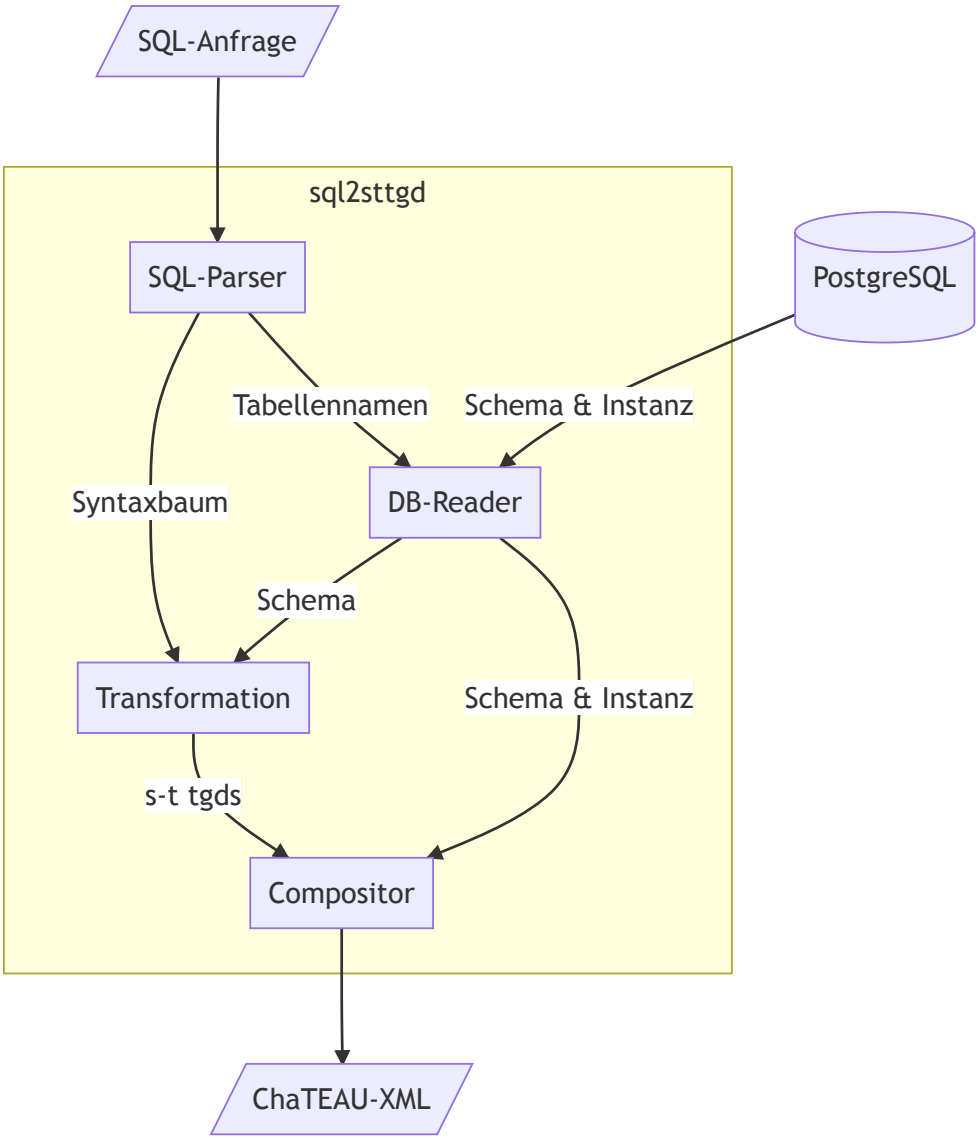


Abbildung 1: Flowchart für den Programmablauf von sql2sttgd

3.1. SQL-Parser

Der SQL-Parser nimmt die SQL-Anfrage als String entgegen. Es wird die syntaktische Korrektheit des Strings geprüft und außerdem getestet, ob es sich um eine Anfrage und nicht etwa um einen Befehl zum Einfügen von Daten handelt.

Die Anfrage wird danach in Form eines Syntaxbaums für die anderen Module bereitgestellt. Die Darstellung als Syntaxbaum vereinfacht die Arbeit mit der Anfrage deutlich.

3.2. DB-Reader

Der DB-Reader verwaltet die vom Nutzer bereitgestellte Datenbankverbindung und liest aus einem Schema der zugehörigen Datenbank alle Tabellen aus, die in der SQL-Anfrage verwendet wurden. Dazu erhält der DB-Reader eine Liste der verwendeten Tabellen vom SQL-Parser.

Für diese Tabellen werden außerdem die Tupel, die in den Tabellen vorhanden sind, ausgelesen und als Instanz gespeichert.

3.3. Transformation

Das Transformationsmodul arbeitet mit der geparsten SQL-Anfrage und dem ausgelesenen Schema, um die Umwandlung der Anfrage in eine oder mehrere s-t tgds durchzuführen. Die Transformation selbst besteht wiederrum aus mehreren Schritten, die anhand eines Beispiels im Abschnitt 4.5 genauer erklärt werden.

3.4. Compositor

Der Compositor ist dafür verantwortlich, dass die Teilergebnisse der vorgestellten Module zusammengeführt und im Anschluss in das Ausgabeformat für ChaTEAU serialisiert werden. Er steht damit am Ende des Kontrollflusses und finalisiert die vom Programm generierten Daten.

4. Implementation

4.1. Verwendete Third-Party-Software

Tobias Rudolph

sql2sttgd wurde mit AdoptOpenJDK Version 11 entwickelt. Unsere Software wird dabei aber von mehreren Open-Source-Bibliotheken unterstützt. Diese übernehmen Aufgaben, bei denen es nicht sinnvoll gewesen wäre, sie selbst zu implementieren. Dies erhöht die Qualität des Programms und senkt gleichzeitig den Wartungsaufwand, da bereits gelöste Teilprobleme durch robuste und getestete Bibliotheken übernommen werden.

In Tabelle 1 wird die verwendete Drittanbietersoftware vorgestellt.

Bibliothek	Aufgabe	Lizenz
Jakarta XML Binding (JAXB) + XJC	Erzeugung von Java-Objekten aus einer XML Schema Definition (XSD), Serialisierung von Java-Objekten in XML	EDL v1.0
JSQLParser	Parsen von SQL, Überführung in einen Syntaxbaum	Apache License v2.0
PostgreSQL JDBC Driver	Kommunikation mit PostgreSQL-Datenbanken über das JDBC-Interface	BSD 2-Clause "Simplified"
Apache Commons Lang	Sammlung verschiedener häufig verwendeter Hilfsfunktionen für Java	Apache License v2.0
Google Guava	Sammlung verschiedener häufig verwendeter Hilfsfunktionen (ähnlich wie Apache Commons Lang), für uns insbesondere interessant wegen einer Graphen-Implementation	Apache License v2.0
Lombok	Reduzierung von Boilerplate-Code wie z.B. triviale Getter / Setter oder NonNull-Checks	MIT License

Tabelle 1: Verwendete Bibliotheken in sql2sttgd

4.2. SQL-Parser

Ivo Kavisanczki

Den SQL-Parser zu implementieren, fiel einfacher als ursprünglich angenommen. Wie der Name JSQLParser bereits vermuten lässt, dient diese Java-Library dazu, SQL-Statements zu parsen. Hierbei wird auch die in Abschnitt 3.1 beschriebene Syntax-Überprüfung durchgeführt.

Da JSQLParser aber beliebige SQL-Statements entgegennimmt und parst, solange sie syntaktisch korrekt sind, war es zusätzlich noch nötig, die Funktion auf Anfragen einzuschränken. Schließlich arbeitet sql2sttgd nur auf Anfragen, aber z.B. nicht auf Update- oder Einfüge-Operationen. Falls der gegebene String keine Anfrage ist, wird eine Exception geworfen. Die Library liefert dann die geparste Anfrage als Syntaxbaum in Form einer Java-Klassenhierarchie zurück, mit der dann in den anderen Modulen komfortabel gearbeitet werden kann.

4.3. DB-Reader

Tobias Rudolph

Der DB-Reader ist in zwei Methoden implementiert: `readSchema()` und `readInstance()`. Das `DatabaseReader`-Objekt nimmt in seinem Konstruktor eine existierende JDBC-Verbindung für eine PostgreSQL-Datenbank entgegen. Die Verbindung wird bspw. bereits in ProSA erzeugt. Beide Methoden verwenden sowohl einen Schemanamen (Standard: `public`) als auch eine Liste von Tabellen, die aus der SQL-Query extrahiert wurden.

Die `readSchema()`-Methode liest über entsprechende JDBC-Funktionen die Namen und Datentypen der Spalten der Tabellen aus. Mithilfe einer Map werden dabei die JDBC-Datentypen den entsprechenden ChaTEAU-Datentypen zugeordnet. Die `readInstance()`-Methode liest dann die Inhalte der Tabellen zeilenweise aus und überprüft dabei auch auf Nullwerte. Beide Methoden speichern ihre Ergebnisse in JAXB-Objekten und geben diese zurück. Die Methode `readSchema()` kapselt die `Relation`-Objekte von JAXB zusätzlich in einer eigenen Klasse `DBSchema`, die das Suchen von Relationen im weiteren Ablauf des Programms einfacher macht.

4.4. Interne Datenrepräsentation

Tobias Rudolph

Die interne Datenrepräsentation erfolgte zunächst exklusiv durch JAXB-Objekte, welche XML-Elemente darstellen. Dadurch sollte die Komplexität des Programms reduziert werden, indem die Daten schon in der Verarbeitung im final benötigten Format vorliegen.

Allerdings wird mittlerweile zusätzlich eine interne Zwischenrepräsentation verwendet, um s-t tgds darzustellen. In Java wurden die entsprechenden Klassen durch den Präfix `Internal` gekennzeichnet. Dies hat den Vorteil, dass wir flexibler auf die relevanten Informationen der Abhängigkeiten zugreifen können und unabhängiger von Formatänderungen des XML-Schemas sind. Z.B. können Variablen so neben dem Namen und dem Index auch Informationen über den Datentypen enthalten.

4.5. Transformation

Marian Zuska

Schauen wir uns nun einmal genauer an, wie das Transformationsmodul von der SQL-Anfrage und dem Schema zu der gewollten Menge an s-t tgds kommt. Hierfür das folgende Beispiel:

Wir haben eine Datenbank mit Studierenden und Noten und möchten Informationen über alle Studierenden, die im Fach *Datenbanken 2* eine Note bekommen haben.

Datenbank-Schema

$Studierende(MatNr, Name, Studiengang)$

$Noten(MatNr, Fach, Note)$

SQL-Anfrage

```
SELECT MatNr, Name, Fach, Note
FROM Studierende NATURAL JOIN Noten
WHERE Fach = 'DB2'
```

Das Transformationsmodul bekommt nun dieses Schema und die SQL-Anfrage als Eingabe und durchläuft eine Reihe von Schritten. Durch diese wird eine initiale s-t tgd erstellt und mit den Informationen aus der SQL-Anfrage Schritt für Schritt erweitert und verändert.

Zuerst brechen wir die Anfrage auf ihre atomaren Unteranfragen herunter. Dort müssen wir vor allem aufpassen, falls sich ein `UNION`-Operator in der Anfrage findet. In diesem Fall haben wir eine verschachtelte Anfrage, die rekursiv in ihre Unteranfragen aufgeteilt wird. Danach erstellen wir für alle diese atomaren Anfragen jeweils eine s-t tgd, wobei die verschiedenen s-t tgds am Ende – sofern nötig – noch angeglichen werden. Für jede der betrachteten Anfragen werden dabei die folgenden Schritte durchlaufen.

Anfangs erstellen wir mit der `buildBody()`-Methode den Rumpf der ersten s-t tgd. Dafür suchen wir uns alle Relationen, die im `FROM`-Teil der Anfrage verwendet werden. Für jede Relation wird dann ein Atom in der s-t tgd erstellt und mit den Attributen der Relation gefüllt. Damit noch keine Attribute in Verbindung zueinander stehen, werden hier noch die Indizes der Attribute unterschiedlich gesetzt. Hier müssen wir von den benötigten Relationen die richtigen Attribute aus der Datenbank auslesen und einfügen. Dafür ist nötig, dass wir mit gegebenen Informationen des SQL-Parsers den DB-Reader anfragen können. Dann werden die Ergebnisse des DB-Readers ausgewertet und in Form eines s-t-tgd-Rumpfes beschrieben. Als Ergebnis ergibt sich die folgende Klausel:

$$Studierende(matnr_1, name_1, studiengang_1) \wedge Noten(matnr_2, fach_2, note_2)$$

Als Nächstes müssen die verschiedenen Relationen in Beziehung zueinander gebracht werden. Das Modul führt die `equalizeVariables()`-Funktion aus. Diese sucht in der Anfrage nach Verwendungen von Joins. Spezifisch wird hier nach natürlichen Verbunden, nach inneren Verbunden mit `ON`-Expression oder nach Kreuzprodukten gesucht. Wird ein passendes Join-Konstrukt gefunden, so werden die Variablen der Join-Attribute der beiden Relationen gleichgesetzt. Im Falle des expliziten Verbunds müssen außerdem noch die Verbundattribute berechnet werden.

Dieser Teil der Implementation hat sich als einer der Schwersten entpuppt. Dies liegt unter anderem daran, dass

1. es unterschiedliche Wege gibt, einen Join zu beschreiben,
2. beim Natural Join noch die Join-Attribute berechnet werden müssen
3. das Ergebnisschema verschiedener Joins unterschiedlich aussieht und
4. sichergestellt werden muss, dass der Join wirklich erlaubt ist.

Haben zwei Attribute zwar den gleichen Namen, aber unterschiedliche Datentypen, so muss dies erkannt werden. Der Join wird in dem Fall zurückgewiesen oder nicht über diese Attribute realisiert. Die Umsetzung und Lösung dieser Probleme werden im weiteren Verlauf der Dokumentation noch beschrieben (siehe Abschnitt 4.7).

In unserem Beispiel findet sich ein Join über die Matrikelnummer. Also werden dort beide Indizes auf den Wert 1 gesetzt.

$$Studierende(matnr_1, name_1, studiengang_1) \wedge Noten(matnr_1, fach_2, note_2)$$

Nun haben wir ausreichend Informationen, um den Kopf der s-t tgd zu initialisieren. Mit der Methode `buildHead()` sucht das Transformationsmodul die auszugebenden Attribute im `SELECT`-Teil der Anfrage und erstellt eine neue Relation *Result*.

Hier dürfen des Weiteren nicht einfach neue Variablen mit eindeutigen Indizes erstellt werden, sondern die Variablen im schon vorhandenen Rumpf der s-t tgd müssen wiederverwendet werden. Somit entsteht keine neue Variable $matnr_3$, sondern $matnr_1$ wird wiederverwendet. Hätten wir hier nicht gejoint, gäbe es außerdem noch zwei $matnr$-Variablen ($matnr_1$ und $matnr_2$) und wir hätten anhand der Tabellenangabe die richtige Variable aussuchen müssen.

$$\begin{aligned} &Studierende(matnr_1, name_1, studiengang_1) \wedge Noten(matnr_1, fach_2, note_2) \\ &\qquad \rightarrow Result(matnr_1, name_1, fach_2, note_2) \end{aligned}$$

Der `SELECT`- und `FROM`-Teil der SQL-Anfrage sind nun vollständig abgearbeitet. Nur im `WHERE`-Teil sind noch entscheidende Informationen. Hier sucht das Modul nach Begrenzungen der Attributwerte, in unserem Beispiel also `Fach = 'DB2'`. Alle Vorkommnisse der zugehörigen Variable werden nun gemäß der Zuweisung mit dem Attributwert ersetzt.

In diesem Schritt müssen also komplexe und unterschiedliche logische Formeln von Einschränkungen eingelesen werden. Dabei muss bspw. bei `Note = 1.3 AND 'DB2' = Fach` rekursiv in die beiden Teilausdrücke gegangen und korrekt erkannt werden, dass der Wert einmal rechts und einmal links steht. Des Weiteren wird sichergestellt, dass es sich bei den Attributwerten um von ChaTEAU unterstützte Datentypen handelt. Diese sind im Moment Integers, Doubles und Strings. Es werden allerdings bspw. die beiden SQL-Datentypen SMALLINT und INTEGER beide zu einem Integer umgewandelt und damit beide unterstützt. Gleiches gilt für REAL und DOUBLE sowie CHAR und VARCHAR. Es entsteht die folgende, finale s-t tgd:

$$\begin{aligned} &Studierende(matnr_1, name_1, studiengang_1) \wedge Noten(matnr_1, \text{‘DB2’}, note_2) \\ &\qquad \rightarrow Result(matnr_1, name_1, \text{‘DB2’}, note_2) \end{aligned} \tag{1}$$

Nun haben wir die atomaren SQL-Anfragen in s-t tgds transformiert. Manche dieser Anfragen waren aber möglicherweise mit dem `UNION`-Operator verknüpft und müssen deswegen angeglichen werden. Möchten wir die Ergebnisse zweier Anfragen vereinigen, so dürfen die Result-Relationen keine Unterschiede aufweisen. Falls Join-Attribute mit unterschiedlichem Namen und/oder Typen existieren, so darf die Vereinigung nicht ausgeführt werden und ein Fehler wird geworfen.

Ist auch dieser letzte Schritt abgeschlossen, so wurde die s-t tgd aus der SQL-Anfrage und dem Schema extrahiert. Im Transformationsmodul sind die s-t tgds als JAXB-Objekt gespeichert. Diese werden dann an den Compositor übergeben und so in die XML eingesetzt.

4.6. Ausgabe mehrerer s-t tgds

Marian Zuska

Wie schon im vorherigen Kapitel angeschnitten, verfügt das Transformationsmodul über die Möglichkeit mehr als eine s-t tgd aus der Query zu extrahieren. Dies ist in den folgenden zwei Fällen von Nöten:

1. Im `WHERE`-Teil der Anfrage steht eine Disjunktion von Zuweisungen. Ein Beispiel dafür wäre die Anfrage:

   ```
   SELECT MatNr, Name, Fach, Note
   FROM Studierende NATURAL JOIN Noten
   WHERE Fach = 'DB2' OR Name = 'Felix Felicis'
   ```

   In diesem Fall werden in der Ergebnistabelle der Anfrage gleichermaßen Informationen über Studierende des Fachs `'DB2'` als auch über Studierende mit Namen `'Felix Felicis'` sein. Würde man versuchen, dies mit einer einzigen s-t tgd zu beschreiben, könnte sie folgendermaßen aussehen:

   $$\begin{aligned} &\mathit{Studierende}(\mathit{matnr}_1, \text{‘Felix Felicis’}, \mathit{studiengang}_1) \wedge \mathit{Noten}(\mathit{matnr}_1, \text{‘DB2’}, \mathit{note}_2) \\ &\qquad \rightarrow \mathit{Result}(\mathit{matnr}_1, \text{‘Felix Felicis’}, \text{‘DB2’}, \mathit{note}_2) \end{aligned}$$

   Diese s-t tgd enthält allerdings die Information, dass alle Tupel im Ergebnis sowohl den gleichen Namen, als auch das gleiche Fach haben. Die Regel gibt den Anschein, als hätte in der Anfrage ein `AND` statt einem `OR` gestanden. Die Lösung sind somit zwei verschiedene s-t tgds. Eine für die Zuweisung `Fach = 'DB2'` und eine für den Namen. Das Ergebnis müsste also das Folgende sein:

   $$\begin{aligned} &\mathit{Studierende}(\mathit{matnr}_1, \mathit{name}_1, \mathit{studiengang}_1) \wedge \mathit{Noten}(\mathit{matnr}_1, \text{‘DB2’}, \mathit{note}_2) \\ &\qquad \rightarrow \mathit{Result}(\mathit{matnr}_1, \mathit{name}_1, \text{‘DB2’}, \mathit{note}_2) \end{aligned}$$

   $$\begin{aligned} &\mathit{Studierende}(\mathit{matnr}_1, \text{‘Felix Felicis’}, \mathit{studiengang}_1) \wedge \mathit{Noten}(\mathit{matnr}_1, \mathit{fach}_2, \mathit{note}_2) \\ &\qquad \rightarrow \mathit{Result}(\mathit{matnr}_1, \text{‘Felix Felicis’}, \mathit{fach}_2, \mathit{note}_2) \end{aligned}$$

2. Der zweite Fall ist, dass die Anfrage eine Vereinigung von zwei oder mehr Anfragen ist. Ein Beispiel mit zwei Unteranfragen wäre dafür die folgende Anfrage:

   ```
   SELECT MatNr, Name, Fach, Note
   FROM Studierende NATURAL JOIN Noten
   WHERE Fach = 'DB2'
   UNION
   (SELECT MatNr, Name, Fach, Note
   FROM Studierende NATURAL JOIN Noten
   WHERE Fach = 'DB')
   ```

Hier müssen s-t tgds für jegliche Unteranfragen der Vereinigung erstellt werden. Unser Modul erzielt das folgende Ergebnis:

$$Studierende(matnr_1, name_1, studiengang_1) \wedge Noten(matnr_1, \text{'DB2'}, note_2) \\ \rightarrow Result(matnr_1, name_1, \text{'DB2'}, note_2)$$

$$Studierende(matnr_1, name_1, studiengang_1) \wedge Noten(matnr_1, \text{'DB'}, note_2) \\ \rightarrow Result(matnr_1, name_1, \text{'DB'}, note_2)$$

Der zweite Fall mit `UNION` wurde in sql2sttgd implementiert, den ersten mit `OR` haben wir allerdings aus zeitlichen Gründen nicht umsetzen können. In Abschnitt 7 wird aber ein Ansatz vorgestellt, mit dem sich auch der erste Fall umsetzen ließe.

4.7. Herausforderungen des Joins

Tom Siegl

Die Behandlung von Joins ist eine große Teilaufgabe der Transformation mit besonderen Herausforderungen, die zusammen mit ihren Lösungen im Folgenden genauer erläutert werden.

4.7.1. Gleichsetzung von Variablen

Ein Join bringt das Problem mit sich, dass herausgefunden werden muss, welche Attribute als Verbundattribute genutzt werden sollen. Diese paarweisen Zuordnungen von Attributen zueinander resultieren in Gleichsetzungen von Variablen in dem Body der s-t tgd.

Dieses Problem wird durch einen graphenbasierten Ansatz gelöst. Dabei werden die Variablen aus dem Body vor der Gleichsetzung der Variablen als Knoten dargestellt. Paare von Variablen, die gleichgesetzt werden müssen, werden als Kanten dargestellt. Die Lösung des Problems basiert auf zusammenhängenden Teilgraphen des resultierenden Graphen, welche im Folgenden als *Zusammenhangskomponenten* bezeichnet werden.

Im Fall des Natural Joins werden die Verbundattribute durch die Schnittmenge der Attributmengen beider Verbundrelationen gewonnen. Dabei wird auf Gleichheit des Namens und des Typs der Attribute getestet.

Falls ein Ausdruck der Form `a JOIN b ON (...)` vorliegt, können die Verbundattribute direkt aus den Gleichheitsbedingungen in dem `ON`-Ausdruck entnommen werden. Andere Bedingungen werden nicht zugelassen, die Bearbeitung einer solchen Anfrage würde abgelehnt werden.

Auch das Kreuzprodukt wird mit in den Graphen aufgenommen. Es entstehen dabei jedoch nur Knoten und keine neuen Kanten. Dieses Vorgehen ist dann nützlich, wenn das Ergebnis von dem Kreuzprodukt wieder eine Eingabe für einen weiteren Verbund ist.

Ist der Graph vollständig aufgebaut, so lässt sich die Lösung für das Problem der Gleichsetzungen von Variablen aus den Zusammenhangskomponenten des Graphen ablesen. Diese enthalten jeweils eine Menge von Knoten (Variablen). Wenn nur ein Knoten in einer Zusammenhangskomponente enthalten ist, bedeutet das, dass dieser Knoten in dem betrachteten Join in keiner Weise als Verbundattribut genutzt wird. Andererseits enthalten Zusammenhangskomponenten genau dann mehrere Knoten, wenn diese Knoten untereinander in Paaren als Verbundattribute genutzt werden [1]. Das ist durch die Definition der Kanten in dem Graphen begründet. Da eine Kante somit für die Gleichsetzung der verbundenen Variablen steht und Gleichheit transitiv gilt, bedeutet eine Zusammenhangskomponente Gleichheit zwischen allen enthaltenen Variablen.

Nachdem die Zusammenhangskomponenten bestimmt sind, werden die Gleichheitsbedingungen in die s-t tgd eingearbeitet. Dazu wird aus jeder Zusammenhangskomponente ein Repräsentant ausgewählt, der in der s-t tgd für alle Elemente der Zusammenhangskomponente eingesetzt wird.

4.7.2. Aufnahme von Verbundattributen in das Ergebnisschema

Da hier nur Joins mit Gleichsetzungen von Verbundattributen betrachtet werden, lassen sich die Verbundattribute als Menge von paarweisen Gleichsetzungen von Attributen darstellen. Ein Problem bei der Bestimmung des Ergebnisschemas ist, dass nicht alle Arten von Joins beide Attribute dieser Paare in das Ergebnis übernehmen. Diese Eigenschaft nimmt Einfluss auf das Schema der Relation, die in dem Head der s-t tgd steht.

Dieses Problem wird durch den inkrementellen Aufbau einer Liste von Variablen gelöst. Je nach Art des Verbundes muss entschieden werden, ob die Verbundattribute beider verbundener Relationen im Ergebnis auftauchen oder nicht. Dafür werden die verbundenen Relationen eines (möglicherweise verketteten) Joins von links nach rechts durchlaufen und für die Attribute dieser Relationen je nach Art des betrachteten (Teil-)Joins über die Aufnahme in das Ergebnis entschieden. Das Ergebnis dieser Entscheidung bestimmt, ob die entsprechende Variable, die dem betrachteten Attribut zugeordnet ist, in die Liste aufgenommen wird oder nicht.

Verbundattribute des Natural Joins treten im Ergebnisschema jeweils nur einfach auf. Das liegt daran, dass beim Natural Join immer klar ist, dass zwischen den Verbundattributen eine Gleichsetzung stattfindet. Daher würden die Werte der Verbundattribute bei doppelter Aufnahme der Verbundattribute ins Ergebnisschema immer gleich sein, es entstünde also Redundanz. Es werden also nur diejenigen Variablen, die den Attributen der rechten Relation im Join entsprechen, der Liste hinzugefügt, die nach dem Hinzufügen der linken Relation keinen Verbundpartner in der Liste haben.

Bei Joins der Form `a JOIN b ON (...)` werden immer die Verbundattribute aus beiden Ausgangsrelationen in das Ergebnisschema übernommen. Da der `ON`-Ausdruck nach dem SQL-Standard mehr

[1] Genau genommen können Knoten keine Verbundattribute sein, weil Knoten Variablen sind. Vor der Gleichsetzung der Variablen im Body der s-t tgd lassen sich Variablen und Attribute aber bijektiv aufeinander abbilden, da jede Variable genau ein Attribut vertritt. Mit den Verbundattributen sind daher die Attribute gemeint, die den Variablen in den jeweiligen Knoten zugeordnet sind.

als nur Gleichheitsbedingungen enthalten darf, ist nicht immer klar, ob durch die Aufnahme der Verbundattribute von beiden Seiten Redundanz entsteht. Daher müssen die Verbundattribute aus beiden Relationen im Ergebnis enthalten sein, obwohl die Gleichheitsbedingungen in dem hier betrachteten Fall immer zu Redundanzen führen. Es werden also alle Variablen, die Attributen der rechten Relation im Join entsprechen, der Liste hinzugefügt.

Da es bei Kreuzprodukten keine Verbundattribute gibt, werden dabei immer alle Variablen beider Seiten in das Ergebnisschema aufgenommen.

Die letztendlich resultierte Liste enthält genau die Variablen, deren zugeordnete Attribute in der Ergebnisrelation enthalten sind. Die Liste gibt daher den Head der s-t tgd vor der Gleichsetzung der Variablen an. In späteren Arbeitsschritten kann der Head, durch die Anwendung der Projektion, auch noch weiteren Änderungen unterzogen werden.

4.7.3. Beispiel

Betrachten wir das folgende Beispiel, um die Funktionsweise der beschriebenen Methode zu verdeutlichen:

Datenbank-Schema

$Studierende(MatNr, Name, Studiengang)$

$Noten(MatNr, FachNr, Note)$

$Faecher(FachNr, Name)$

$Raeume(RaumNr, Adresse)$

SQL-Anfrage

```
SELECT *
FROM Studierende
    NATURAL JOIN Noten
    JOIN Faecher ON
        (Noten.FachNr = Faecher.FachNr)
    CROSS JOIN Raeume
```

s-t tgd Body vor Gleichsetzung der Variablen

$$Studierende(matnr_1, name_1, studiengang_1) \wedge Noten(matnr_2, fachnr_2, note_2) \\ \wedge Faecher(fachnr_3, name_3) \wedge Raeume(raumnr_4, adresse_4)$$

Die Anfrage müssen wir nicht in ihrer Gesamtheit betrachten, da der `SELECT`-Teil die Ergebnisrelation der Verbunde nicht verändert. Diesen Teil werden wir daher im Folgenden nicht weiter betrachten und unsere Konzentration lediglich auf die Verbunde richten.

Dazu werden wir die Verarbeitung in vier Schritte unterteilen. Diese behandeln in der gegebenen Reihenfolge die Initialisierung, den `NATURAL JOIN`, den (`INNER`) `JOIN` und das Kreuzprodukt.

Die gezeigte Liste sowie der gezeigte Graph in allen Schritten entsprechen den beschriebenen Datenstrukturen aus den voranstehenden Texten in diesem Unterkapitel. Beide Datenstrukturen sind in der Implementierung in der Klasse `JoinRelation` zu finden.

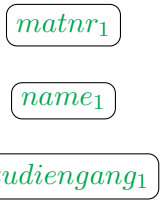
Die hinzukommenden Variablen werden in den Darstellungen immer grün eingefärbt.

1. Begonnen wird mit der `Studierende`-Relation, welche die initiale Knotenmenge im Graphen sowie ein initiales Schema als Startpunkt für das Ergebnisschema liefert.

   Variablen im Ergebnisschema

   $[matnr_1, name_1, studiengang_1]$

   Graph

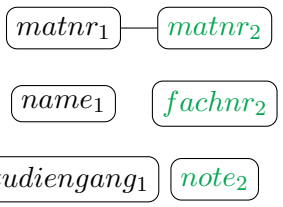
   

2. In der Verarbeitung des `NATURAL JOIN` kommt die Relation `Noten` hinzu. Zuerst müssen wir die Verbundattribute bestimmen. Diese bestehen in diesem Fall aus dem Paar `Studierende.MatNr` und `Noten.MatNr`. Zwischen diesen Attributen wird also eine Gleichheitsbeziehung herrschen, welche in dem Graphen durch eine Kante zwischen den entsprechenden Variablen widergespiegelt wird. Durch die oben beschriebene Eigenschaft des `NATURAL JOIN` werden diese beiden Attribute im Ergebnisschema zusammengefasst. In der Liste, die dafür erstellt wird, wird daher nicht die Variable aufgenommen, die dem Verbundattribut der rechten Relation im Join entspricht ($matnr_2$). Es werden dennoch alle Variablen von `Noten` in die Knotenmenge des Graphen aufgenommen, um die Gleichheitsbeziehung darstellen zu können.

   Variablen im Ergebnisschema

   $[matnr_1, name_1, studiengang_1,$
   $fachnr_2, note_2]$

   Graph

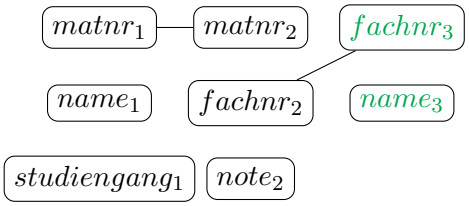
   

3. Das bisherige Ergebnis wird mittels eines `INNER JOIN` mit `ON`-Ausdruck mit der Relation `Faecher` verbunden. Der `ON`-Ausdruck verlangt eine Gleichheitsbeziehung zwischen den Attributen `Noten.FachNr` und `Faecher.FachNr`. Dieses Paar von Attributen können wir also als ein Paar von Verbundattributen betrachten. Für die Veränderung am Graphen bedeutet das, dass eine Kante zwischen den entsprechenden Variablen eingefügt werden muss. Vorher müssen jedoch, unabhängig von den Verbundattributen, alle Attribute aus der Relation `Faecher` in Form von Variablen als Knoten in den Graphen hinzugefügt werden. Es werden auch alle Variablen, die den Attributen der Relation `Faecher` entsprechen, in das Ergebnisschema hinzugefügt.

Variablen im Ergebnisschema

$[matnr_1, name_1, studiengang_1,$
$fachnr_2, note_2,$
$fachnr_3, name_3]$

Graph

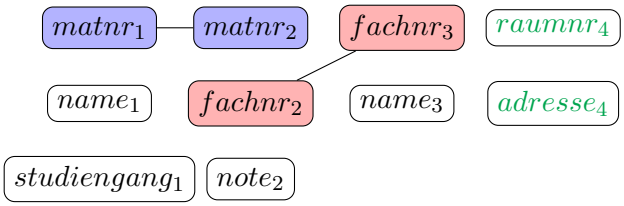


4. Im letzten Schritt müssen wir mittels Kreuzprodukt zu dem bisherigen Ergebnis noch die Relation `Raeume` hinzufügen. Durch das Kreuzprodukt gibt es keine neuen Verbundattribute beziehungsweise Gleichheitsbedingungen zwischen Attributen. Alle Variablen, die Attributen der Relation `Raeume` entsprechen, werden sowohl zum Ergebnisschema als auch, wie gewohnt in Form von Knoten, zum Graphen hinzugefügt.

Variablen im Ergebnisschema

$[matnr_1, name_1, studiengang_1,$
$fachnr_2, note_2,$
$fachnr_3, name_3,$
$raumnr_4, adresse_4]$

Graph



Erinnern wir uns an den Body der s-t tgd bevor wir im nächsten Schritt die Gleichsetzung der Variablen darin vornehmen.

s-t tgd Body vor Gleichsetzung der Variablen

$$Studierende(matnr_1, name_1, studiengang_1) \wedge Noten(matnr_2, fachnr_2, note_2) \\ \wedge Faecher(fachnr_3, name_3) \wedge Raeume(raumnr_4, adresse_4)$$

Da der Graph aufgebaut ist, können wir die Zusammenhangskomponenten bestimmen. Für diese Aufgabe wird eine Breitensuche für jede Variable im Ergebnisschema gestartet. Das Vorgehen der Breitensuche soll hier nicht genauer beleuchtet werden. Daher sind alle Zusammenhangskomponenten mit mehr als einem Knoten in dem Graphen aus Schritt vier bereits farbig markiert. Die daraus resultierenden Gleichsetzungen können wir nun auf den Body anwenden und erhalten den folgenden Body:

s-t tgd Body nach Gleichsetzung der Variablen

$$Studierende(matnr_1, name_1, studiengang_1) \wedge Noten(matnr_1, fachnr_2, note_2) \\ \wedge Faecher(fachnr_2, name_3) \wedge Raeume(raumnr_4, adresse_4)$$

Des Weiteren können die gefundenen Zusammenhangskomponenten dazu genutzt werden, die Variablen im Ergebnisschema gleichzusetzen.

Variablen im Ergebnisschema nach Gleichsetzung

$$
\begin{aligned}
[&matnr_1, name_1, studiengang_1,\\
&fachnr_2, note_2,\\
&fachnr_2, name_3,\\
&raumnr_4, adresse_4]
\end{aligned}
$$

An dieser Stelle wird die Projektion angewendet, sodass nur die Variablen im Ergebnisschema übrig bleiben, die den Attributen aus der Projektion entsprechen. Da die Anfrage in diesem Beispiel bei der Projektion alle Attribute übernimmt, können wir die Liste der Variablen im Ergebnisschema direkt in den Head der s-t tgd überführen. Danach erhalten wir eine vorläufige s-t tgd:

vorläufige s-t tgd

$$
\begin{aligned}
&Studierende(matnr_1, name_1, studiengang_1) \wedge Noten(matnr_1, fachnr_2, note_2)\\
&\wedge Faecher(fachnr_2, name_3) \wedge Raeume(raumnr_4, adresse_4)\\
\rightarrow\ &Result(matnr_1, name_1, studiengang_1, fachnr_2, note_2, fachnr_2, name_3, raumnr_4, adresse_4)
\end{aligned}
$$

Anschließend können Konstanten aus der Anfrage in die s-t tgd eingesetzt werden. Das wird in diesem Beispiel aber nicht vorgeführt.

4.8. Compositor

Marian Zuska

Im Compositor werden die Teilergebnisse der anderen Module in eine XML-Datei zusammengesetzt. Da wir die Open-Source-Bibliothek JAXB (*Jakarta XML Binding*) verwenden, war dieses Modul gut umsetzbar. Die Ausgaben der Transformation und des DB-Readers sind schon in JAXB-Objekten. Somit können wir mit der von JAXB generierten Object-Factory die XML-Root Elemente erstellen und die Ausgaben der anderen Module einfügen.

Das entstandene JAXB-Objekt wird am Ende noch in einen Stream serialisiert, mit dem die XML-Datei erstellt werden kann.

4.9. Continuous Integration

Marian Zuska

Während der Implementation haben wir von unterschiedlichen Rechnern und Orten aus gearbeitet. Manche von uns haben dabei das Betriebssystem Linux verwendet und manche Windows. Dabei können Integrationsprobleme auftreten und es kann passieren, dass eine Codeänderung zwar bei einer Person fehlerlos durchläuft, dann aber bei jemand anderem einen Fehler wirft. Da dies die Implementationsphase unnötig verkomplizieren kann, haben wir uns für eine Continuous Integration (CI) Pipeline entschieden.

Bei jeglichen Commits und Code-Änderungen wird ein CI-Runner gestartet, der das Projekt baut und alle bestehenden Tests auf einem unabhängigen System laufen lässt.

Das Projekt wird dabei immer in dem gleichen Docker-Image ausgeführt. Somit hatten wir eine konsistente Umgebung und alle Code-Änderungen wurden auf dem gleichen System getestet. Der Ausgang der Tests war also nicht mehr von der unterschiedlichen Umgebung abhängig. Dies ist von Vorteil, da wir, wie schon beschrieben, auf verschiedenen Systemen arbeiten und da sich auf diesen Systemen auch Dateien außerhalb des Projektes verändern können. Durch die CI konnten wir alle Tests jederzeit reproduzieren und uns sicher sein, dass ein Test nie durch die Umgebung fehlschlug.

5. Umgesetzte Konzepte

Ivo Kavisanczki

Kurzgefasst konnten diese sprachlichen Konzepte einer SQL-Anfrage in sql2sttgd umgesetzt werden:

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

Die einzelnen Konzepte und deren Umsetzung werden im Folgenden näher erläutert.

5.1. Projektion

Abschließende Projektionen, wie sie in SQL im `SELECT`-Teil einer Anfrage vorkommen, werden als Atom im Kopf einer s-t tgd mit den entsprechenden Variablen dargestellt. Folgende Anfrage z.B.

```
SELECT MatNr, Name
FROM Studierende
```

würde als s-t tgd so aussehen:

$$Studierende(matnr_1, name_1, studiengang_1) \rightarrow Result(matnr_1, name_1)$$

Man sieht, dass die Attribute `MatNr` und `Name` aus der Projektion in der Anfrage als die Variablen $matnr_1$ und $name_1$ im Result-Atom im Kopf der s-t tgd auftauchen. In sql2sttgd werden dafür die entsprechenden Attribute zuerst aus dem `SELECT`-Teil der Anfrage ausgelesen, dann überprüft (sie dürfen z.B. nicht mehrdeutig sein) und zuletzt in das Result-Atom geschrieben. Für den letzten Schritt müssen hierbei die richtigen Variablen des Rumpfs verwendet werden. Dies passiert in der Methode `buildHead()`.

Ein Sonderfall besteht, wenn man eine Anfrage der Form `SELECT * FROM ...` hat. In diesem Fall findet keine Projektion am Ende der Anfrage statt, d.h. alle Attribute der Ergebnisrelation sollen ausgegeben werden. Hierfür muss das Schema der Ergebnisrelation berechnet werden, was insbesondere bei verschiedenen Verbunden nicht mehr trivial ist. In `equalizeVariables()` wird hierfür neben den Verbundattribute gleichzeitig auch das Ergebnisschema iterativ ermittelt. Siehe auch Abschnitte 4.7.2 und 5.2 für genauere Erklärungen der Problemstellung und der Lösung in sql2sttgd. Sobald das Ergebnisschema bekannt ist, kann dann (in gleicher Weise wie bei einer Projektion mit Attributliste) aus den richtigen Variablen des Rumpfs der Kopf der s-t tgd entsprechend des Schemas konstruiert werden.

5.2. Verbund

Ein Verbund wird grundsätzlich über Verwendung gleicher Variablen in verschiedenen Atomen der s-t tgd realisiert. Siehe Abschnitt 4.5 für ein Beispiel. Mit `buildBody()` werden zuerst allerdings nur eindeutige Variablen im Rumpf der s-t tgd erstellt, um ungewollte Konflikte bei identischen Namen zu vermeiden. Mit `equalizeVariables()` werden dann die Verbunde in die noch unfertige s-t tgd eingearbeitet. Wie der Name bereits andeutet, werden dafür Variablen im Rumpf, die vorher unterschiedlich waren, gleichgesetzt. Zum Ermitteln der gleichzusetzenden Verbundattribute wird in sql2sttgd dafür ein graphbasiertes Verfahren genutzt. Die Variablen stellen hierbei Knoten dar und Gleichheitsbeziehungen zwischen zwei Variablen (weil über diese ein Verbund stattfindet) werden als Kanten dargestellt. Der Graph wird iterativ über alle Verbunde im `FROM`-Teil der Anfrage aufgebaut. Mit jedem weiteren Verbund werden weitere Knoten und möglicherweise auch Kanten im Graphen hinzugefügt. Sobald dieser fertig konstruiert ist, werden alle Knoten (d.h. Variablen) innerhalb einer Zusammenhangskomponente des Graphen in der s-t tgd durch einen Repräsentanten der Komponente ersetzt und damit auch gleichgesetzt. In Abschnitt 4.7.1 wird dieses Verfahren im Detail erläutert.

Bei einem natürlichen Verbund werden Verbundattribute nicht explizit angegeben, sondern es findet ein Verbund über jene Attribute statt, die den gleichen Namen und den gleichen Typ (z.B. Integer, String, etc.) haben. Bei einem Verbund `A NATURAL JOIN B` werden also Kanten zwischen den Variablen im Graphen eingefügt, die sowohl in der Relation `A` als auch in `B` vorkommen.

Zusätzlich zu natürlichen Verbunden unterstützt unser Tool aber auch innere Verbunde, bei denen die Verbundattribute explizit über ein Verbundprädikat (mittels `ON`) angegeben werden können. Dies kann z.B. dann nützlich sein, wenn über Spalten mit verschiedenen Namen gejoint werden soll, bspw. für das Datenbankschema

$$\begin{aligned} &\mathit{Studierende}(\mathit{MatNr}, \mathit{Name}, \mathit{Studiengang}),\\ &\mathit{Noten}(\mathit{MatrikelNr}, \mathit{Fach}, \mathit{Note}) \end{aligned}$$

mit entsprechender Anfrage

```
SELECT *
FROM Studierende JOIN Noten ON (Studierende.MatNr = Noten.MatrikelNr)
```

Als Verbundprädikate sind hierbei nur Konjunktionen von Attribut-Gleichheiten erlaubt, d.h. der Form `a = b AND c = d AND ...`. Andere Prädikate, wie z.B. Disjunktionen (`a = b OR c = d`) oder Ungleichungen (`a < b`), lassen sich nicht ohne Weiteres als s-t tgd darstellen. Anstatt wie beim natürlichen Verbund gemeinsame Attribute zu suchen, werden bei einem solchen inneren Verbund die Gleichheitsbedingungen (die dann als Kanten im Graphen dargestellt werden) aus der Expression hinter `ON` gelesen.

Bei einem Kreuzprodukt werden in einer s-t tgd keine Variablen gleichgesetzt, sondern sie müssen zwischen den verbundenen Relationen unterschiedlich sein. Da `buildBody()` aber, wie bereits vorher angemerkt, sowieso eindeutige Variablen konstruiert, muss hier nichts weiter getan werden. Im Graphen werden also bei einem Kreuzprodukt nur die Knoten hinzugefügt, aber keine neuen Kanten.

Ein weiteres Problem bei der Berechnung der Verbunde ist neben der Ermittlung der Verbundattribute auch die Bestimmung des richtigen Ergebnisschemas, da unterschiedliche Verbundarten auch verschiedene Ergebnisschemata erzeugen. Und zwar erhält man bei einem natürlichen Verbund die Vereinigung der Relationenschemata, während sie bei einem inneren Verbund oder einem Kreuzprodukt konkateniert werden. In Tabelle 2 werden die von sql2sttgd unterstützten Verbundarten an einem Beispiel gegenübergestellt (unter der Annahme, dass keine Projektion erfolgt, d.h. die Anfrage mit `SELECT * FROM ...` beginnt).

Verbundart	Ausdruck in SQL	Darstellung als s-t tgd
Natürlicher Verbund	`S NATURAL JOIN N`	$S(mnr_1, nam_1, st_1) \wedge N(mnr_1, fa_1, no_1)$ $\rightarrow R(no_1, nam_1, st_1, fa_1, no_1)$
Innerer Verbund	`S JOIN N ON (S.MNr = N.MNr)`	$S(mnr_1, nam_1, st_1) \wedge N(mnr_1, fa_1, no_1)$ $\rightarrow R(mnr_1, nam_1, st_1, \mathit{mnr_1}, fa_1, no_1)$
Kreuz-produkt	`S CROSS JOIN N`	$S(mnr_1, nam_1, st_1) \wedge N(\mathit{mnr_2}, fa_1, no_1)$ $\rightarrow R(mnr_1, nam_1, st_1, \mathit{mnr_2}, fa_1, no_1)$

Tabelle 2: Darstellung verschiedener Verbundarten als s-t tgd (Namen wurden der Übersichtlichkeit halber abgekürzt und Unterschiede in den s-t tgds zur Variante mit dem natürlichen Verbund farblich hervorgehoben)

Um diese Unterschiede zwischen den Verbunden korrekt abzubilden, wird neben dem Graphen, der die Gleichheit zwischen Variablen modelliert, auch eine Liste von Variablen aufgebaut, die das Schema der Ergebnisrelation darstellen soll. Je nach Verbund werden dieser Liste unterschiedliche Variablen angehängt (also z.B. bei einem natürlichen Verbund nur neue, bei einem Kreuzprodukt alle Variablen). In Abschnitt 4.7.2 wird dieses Verfahren näher beschrieben.

5.3. Selektion

Eine Teilmenge möglicher Selektionen lässt sich sehr natürlich als s-t tgd darstellen, nämlich jene Selektionen, bei denen Attribute mit Konstanten gleich gesetzt werden. Dafür werden in der s-t tgd einfach die entsprechenden Variablen durch die jeweiligen Konstanten ersetzt, z.B. sieht die Anfrage

```
SELECT MatNr
FROM Studierende
WHERE Studiengang = 'Informatik'
    AND Name = 'Felix Felicis'
```

als s-t tgd einfach folgendermaßen aus:

$$\mathit{Studierende}(\mathit{matnr}_1, \text{‘Felix Felicis’}, \text{‘Informatik’}) \rightarrow \mathit{Result}(\mathit{matnr}_1)$$

Hierfür wird die Selektionsbedingung im `WHERE`-Teil analysiert und allen Variablen, die einer Konstante gleichgesetzt werden, wird der Wert der Konstante zugewiesen. Dabei ist die Reihenfolge auch unerheblich, d.h. `'Informatik' = Studiengang` wird genauso erkannt wie `Studiengang = 'Informatik'`. Dies geschieht in der Methode `insertConstants()`, welche gegen Ende der Transformation ausgeführt wird.

Ähnlich wie bei den Verbundprädikaten wurden in sql2sttgd allerdings nur eben solche Konjunktionen von Gleichheitsatomen umgesetzt. Selektionen nach Ungleichheit (also Bereichsanfragen) wie `MatNr < 12` sind für unser Tool nicht zulässig. Disjunktionen (`OR`) lassen sich zwar ähnlich wie Vereinigungen durch mehrere s-t tgds abbilden (siehe Abschnitt 4.6), wurden in unserer Implementation aber

nicht berücksichtigt. Auch komplexere Selektionsbedingungen (z.B. mit Negationen, verschachtelten Anfragen, Quantoren) wurden nicht betrachtet.

5.4. Vereinigung

Die Vereinigung von mehreren Anfrageergebnissen lässt sich mittels einer Menge von s-t tgds realisieren. Dabei entspricht jede s-t tgd in der Menge einer der vereinigten Unteranfragen. In Abschnitt 4.6 wird dies an einem Beispiel näher ausgeführt. Um das zu erreichen werden zuerst die Unteranfragen aus der gesamten Anfrage ausgelesen, dann separat ausgewertet und jeweils in eine s-t tgd transformiert. Diese werden dann in einer Liste zusammengefasst. Im XML-Dokument hat dann das `<dependencies>`-Element mehrere `<sttgd>`-Kindelemente.

6. Ergebnisse bei Eingabe in ChaTEAU

Ivo Kavisanczki

Um zu überprüfen, ob unser Tool in Kombination mit ChaTEAU auch tatsächlich richtige Ergebnisse erzielt, haben wir probehalber einige SQL-Anfragen (plus Verbindung zu einer entsprechend gefüllten Datenbank) als Eingabe für sql2sttgd genommen und das generierte XML-Dokument wiederum als Eingabe für ChaTEAU benutzt. Dabei haben wir eine Version von ChaTEAU genutzt, die aus dem Zeitraum stammt, als wir mit der Implementation des Projekts begonnen haben (Commit `6ce94828` vom 01. Juni 2021).

Alle s-t tgds und Tupel der Datenbank wurden dabei korrekt von ChaTEAU eingelesen. Auch die Ausführung des Chase lieferte die erwarteten Ergebnisse. Nur in den Fällen, wo Konstanten in den s-t tgds vorhanden waren, gab es Konflikte. Die s-t tgds wurden zwar auch hier korrekt eingelesen und in den entsprechenden Boxen der ChaTEAU-GUI richtig angezeigt, aber bei der Ausführung der Tests wurde eine `ConcurrentModificationException` geworfen. Außerdem führt der Chase in diesem Fall stets zu einer leeren Instanz als Ergebnis. Soweit wir es feststellen konnten, lag dieses Verhalten aber nicht an einem Fehler in der XML, sondern an einem Bug in ChaTEAU (der möglicherweise bereits behoben wurde).

6.1. Beispiel

Tobias Rudolph

Im Anhang A haben wir ein Beispiel angefügt, welches ein Datenbankschema und eine SQL-Anfrage sowie die daraus generierte XML-Datei zeigt. Die Instanz wurde der Übersichtlichkeit halber gekürzt. Zusätzlich sind in Anhang B Screenshots zu sehen, welche einen Durchlauf von ChaTEAU mit der generierten XML-Datei zeigen.

7. Ausblick

Ivo Kavisanczki

Einige weiterführende Konzepte aus SQL haben wir aus zeitlichen Gründen nicht implementieren können. Die folgenden Verbesserungen könnten aber in Zukunft umgesetzt werden, um sql2sttgd zu erweitern:

Innerer Verbund mit `USING` Verbundattribute können nicht nur mit einer `ON`-Expression angegeben werden, sondern bei gleichem Namen auch mit dem `USING`-Keyword (z.B. `Studierende JOIN Noten USING (MatNr)`). Dies ist nicht ganz so flexibel, wie die Version mit `ON`, da nicht beliebige Ausdrücke verwendet werden können, sondern nur die Verbundattribute aufgezählt werden, kann dafür aber kürzer sein. Ein Join mit `USING` ließe sich ähnlich implementieren, wie es bei dem natürlichen Verbund und bei einem Verbund mit `ON` bereits der Fall ist. Es muss überprüft werden, dass die Verbundattribute auch tatsächlich in beiden Relationen vorhanden sind, dann muss eine Kante zwischen den entsprechenden Variablen aus der linken und der rechten Relation im Graphen eingefügt werden (jener Graph, der die Äquivalenzen zwischen Variablen modelliert). Dabei muss darauf geachtet werden, dass bei `USING` (im Gegensatz zu `ON`) das Ergebnisschema wie bei einem natürlichen Verbund gebildet wird, eben nur auf den explizit angegeben Verbundattributen. Diese tauchen im Ergebnisschema nur einmal auf, während bei `ON` alle Attribute der linken und der rechten Relation enthalten sind.

Verbundprädikat im `WHERE`-Teil In der aktuellen Version von sql2sttgd sind Verbundprädikate nur in einer `ON`-Expression zugelassen. Sie können aber auch im `WHERE`-Teil auftauchen, wie z.B. bei der „alten“ Schreibweise für Verbunde:

```
SELECT *
FROM Studierende, Noten
WHERE Studierende.MatNr = Noten.MatNr
```

Beim Einarbeiten der Verbunde in `equalizeVariables()` müssten hierfür zusätzlich die für den Verbund relevanten Atome im Prädikat hinter `WHERE` betrachtet werden.

Verbunde mittels UFDS Die Klasse `JoinRelation`, die für die Berechnung der Verbunde zuständig ist, arbeitet mit einem graphbasiertem Verfahren. Möglicherweise lässt sich die Effizienz von `JoinRelation` verbessern, wenn das Verfahren mit Union-Find-Disjoint-Sets (UFDS) statt mit Graphen arbeitet. Jede Zusammenhangskomponente im Graphen würde dabei einem Set im UFDS entsprechen. Die Variablen einer Zusammenhangskomponente sind trivialerweise disjunkt zu den Variablen anderer Komponenten. Wenn eine neue Kante zwischen zwei Komponenten eingefügt wird, entspräche das einer Vereinigung der Mengen. UFDS würden vermutlich das Finden von Variablen beschleunigen.

Bei Verbunden im Graphen nach bestehenden Variablen suchen Aktuell kann bei der Berechnung von Verbunden in Randfällen ein Fehler entstehen. Dieser wird an einem einfachen Beispiel am

deutlichsten. Angenommen, wir haben drei Relationen, welche alle das gleiche Attribut besitzen: $R_1(A), R_2(A), R_3(A)$. Dann würde folgende Anfrage nicht korrekt verarbeitet werden:

```
SELECT *
FROM R1
    NATURAL JOIN R2
    JOIN R3 ON (R2.A = R3.A)
```

Der Body der s-t tgd vor `equalizeVariables()` würde dabei so aussehen: $R_1(a_1) \wedge R_2(a_2) \wedge R_3(a_3)$. Der Fehler entsteht beim Berechnen des dritten Verbundes mit R_3. Das Ergebnisschema enthält davor nur a_1 und der Graph enthält a_1 und a_2 als Knoten mit einer Kante zwischen den beiden (wie zu erwarten bei einem natürlichen Verbund zwischen R_1 und R_2). Wenn allerdings R_3 hinzugefügt wird, kommt es zu einem Fehlverhalten, da die richtige Variable zu `R2.A` (a_2) nicht gefunden wird. Das liegt daran, dass in unserer Implementation nur im Ergebnisschema nach vorhandenen Variablen gesucht wird und nicht auch im Graphen. Um a_2 zu finden, müsste auch im Graphen nach bereits gefundenen Variablen gesucht werden.

Aggregationen Im `SELECT`-Teil der Anfrage sind bisher nur „normale“ Projektionen zugelassen, d.h. eine Projektion auf eine Liste von Attributen oder gar keine Projektion mittels `SELECT *`. Wenn der Parser zusätzlich Aggregatfunktionen unterstützten würde, dann würde eine wichtige Klasse von Anfragen hinzukommen. Die Schwierigkeit dabei ist, dass jedes Tupel einer Relation nur einmal durch die Aggregatfunktionen ausgewertet werden darf, da das Ergebnis sonst möglicherweise fehlerbehaftet ist. Eng in Verbindung mit Aggregationen stehen auch die SQL-Sprachelemente `GROUP BY` und `HAVING`. All diese Konzepte erfordern vermutlich, dass zusätzliche Relationen erzeugt werden, um Zwischenergebnisse zu speichern.

Logisches Oder Wie bereits in Abschnitt 4.6 erklärt, lassen sich auch Disjunktionen bei den Selektionsbedingungen als Menge von s-t tgds darstellen. Konjunktionen werden bereits von unserem Tool unterstützt. Ein möglicher Ansatz, um auch beliebige Kombinationen von `AND` und `OR` zu ermöglichen, wäre, die Selektionsbedingung zuerst in Disjunktive Normalform (DNF) zu bringen. Aus jeder $\wedge$-Klausel könnte dann eine neue Anfrage konstruiert werden, die nur diese Klausel als Selektionsbedingung enthält. Dadurch werden die Disjunktionen eliminiert und für die erhaltene Anfrage kann dann wie gewohnt eine korrespondierende s-t tgd erzeugt werden. All diese könnten dann wie bei `UNION` zu einer Liste zusammengefasst werden.

Normierung der Variablenindizes Um Verbunde in die s-t tgd einzuarbeiten werden vorher eindeutige Variablen gleichgesetzt. Die Eindeutigkeit wird durch unterschiedliche Indizes erreicht, die während `buildBody()` vergeben werden. Bei `equalizeVariables()` werden dann Verbundattribute gleichgesetzt. Dabei wird eine Variable in der Zusammenhangskomponente als Repräsentant ausgewählt und alle anderen Variablen der Komponente werden durch erstere ersetzt. Vonseiten der Semantik ist dabei gleich, welche Variable als Repräsentant gewählt wird, da ohnehin alle gleichgesetzt werden. Durch dieses Vorgehen kann es aber passieren, dass die Indizes einiger Variable größer sind, als sie sein müssten. Bei der s-t tgd 1 sieht man zum Beispiel, dass es eine Variable

$note_2$ gibt, obwohl kein $note_1$ vorhanden ist. Das kann durch eine „schlechte“ Wahl des Repräsentanten passieren oder, wie in diesem Beispiel, weil Variablen sicherheitshalber mit dem Index eindeutig gemacht werden, wo das nicht nötig wäre. Um solche „Schönheitsfehler“ zu beheben, könnten gegen Ende der Transformation die Indizes der Variablen herunter gesetzt werden, wo dies möglich ist.

Aliasse Um SQL-Anfragen kompakter zu machen, können Aliasse sehr nützlich sein, z.B.

```
SELECT *
FROM Studierende S JOIN Noten N ON (S.MatNr = N.MatNr)
WHERE S.Name = 'Felix Felicis'
```

Um solche Aliasse zu unterstützten, wäre ein zusätzlicher Mechanismus nötig, der alle Aliasse aus der Anfrage ausliest und ihnen die entsprechenden Relationen zuordnet. Dies könnte in einer Map gespeichert werden. Wenn dann ein Alias beim Verarbeiten der Anfrage auftaucht (z.B. beim Lesen eines Verbundprädikates oder der Selektionsbedingung), muss die richtige Relation durch Suchen in der Map gefunden werden. Mit dieser sollte dann wie gewohnt weiter verfahren werden können.

8. Fazit

Ivo Kavisanczki

Mit dem Projekt konnten wir zeigen, dass es möglich ist, einfache SQL-Anfragen als s-t tgds darzustellen. Diese werden dann zusammen mit dem gegebenen Datenbank-Schema und der Datenbank-Instanz als XML ausgegeben. Dieses XML-Dokument kann dann wiederum von ChaTEAU weiterverwendet werden.

Damit muss die XML, die als Eingabe für ChaTEAU dient, nun nicht mehr händisch erstellt werden, sondern es genügt, wenn der Nutzer von ProSA eine SQL-Anfrage und eine Verbindung zu einer Datenbank zur Verfügung stellt. Das macht die Benutzung von ProSA komfortabler und robuster, da das manuelle Erstellen von solchen XML-Dokumenten fehleranfällig ist.

Unser Tool unterstützt wichtige Operatoren der Relationenalgebra, wie Projektion, Verbund, Selektion und Vereinigung (mit gewissen Einschränkungen). Mögliche Erweiterungen in der Zukunft wären außerdem die Unterstützung von Aggregationen und komplexeren Selektionsbedingungen.

A. Beispiel für erzeugte XML-Dateien

Datenbank-Schema

$studierende(matnr, name, studiengang)$

$noten(matnr, fachnr, note)$

$faecher(fachnr, name)$

$raeume(raumnr, adresse)$

SQL-Anfrage

```
SELECT studierende.name, note
FROM noten NATURAL JOIN studierende
JOIN faecher ON
    (noten.fachnr = faecher.fachnr)
```

XML-Output

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<input>
    <schema>
        <relations>
            <relation name="faecher" tag="S">
                <attribute name="fachnr" type="int"/>
                <attribute name="name" type="string"/>
            </relation>
            <relation name="studierende" tag="S">
                <attribute name="matnr" type="int"/>
                <attribute name="name" type="string"/>
                <attribute name="studiengang" type="string"/>
            </relation>
            <relation name="noten" tag="S">
                <attribute name="matnr" type="int"/>
                <attribute name="fachnr" type="int"/>
                <attribute name="note" type="double"/>
            </relation>
            <relation name="Result" tag="T">
                <attribute name="name" type="string"/>
                <attribute name="note" type="double"/>
            </relation>
        </relations>
        <dependencies>
            <sttgd>
                <body>
                    <atom name="noten">
                        <variable name="matnr" type="V" index="1"/>
                        <variable name="fachnr" type="V" index="1"/>
                        <variable name="note" type="V" index="1"/>
                    </atom>
                    <atom name="studierende">
                        <variable name="matnr" type="V" index="1"/>
```

```
                    <variable name="name" type="V" index="2"/>
                    <variable name="studiengang" type="V" index="2"/>
                </atom>
                <atom name="faecher">
                    <variable name="fachnr" type="V" index="1"/>
                    <variable name="name" type="V" index="3"/>
                </atom>
            </body>
            <head>
                <atom name="Result">
                    <variable name="name" type="V" index="2"/>
                    <variable name="note" type="V" index="1"/>
                </atom>
            </head>
        </sttgd>
    </dependencies>
</schema>
<instance>
    <relationalAtom name="noten">
        <!-- ... -->
        <tuple id="noten_2">
            <constant name="862327568"/>
            <constant name="1"/>
            <constant name="1.3"/>
        </tuple>
        <!-- ... -->
    </relationalAtom>
    <relationalAtom name="studierende">
        <!-- ... -->
        <tuple id="studierende_2">
            <constant name="862327568"/>
            <constant name="Sophia Vogel"/>
            <constant name="Informatik"/>
        </tuple>
        <!-- ... -->
    </relationalAtom>
    <relationalAtom name="faecher">
        <tuple id="faecher_1">
            <constant name="1"/>
            <constant name="Datenbanken 1"/>
        </tuple>
        <!-- ... -->
    </relationalAtom>
</instance>
</input>
```

B. Verwendung der XML-Datei in ChaTEAU

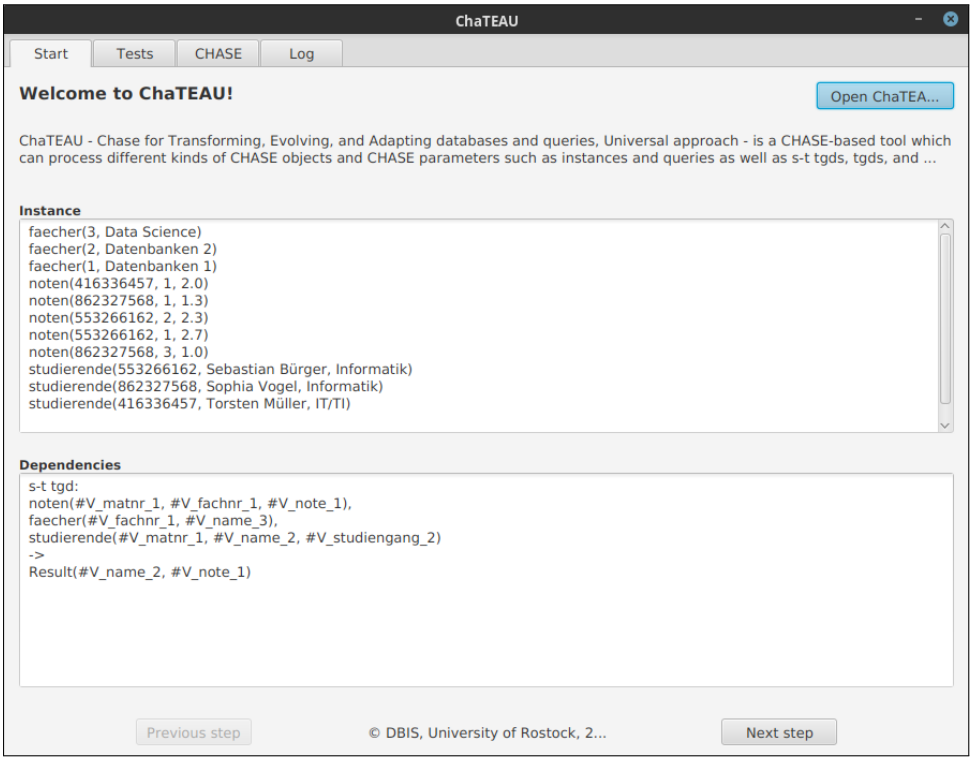


Abbildung 2: Öffnen der XML-Datei in ChaTEAU

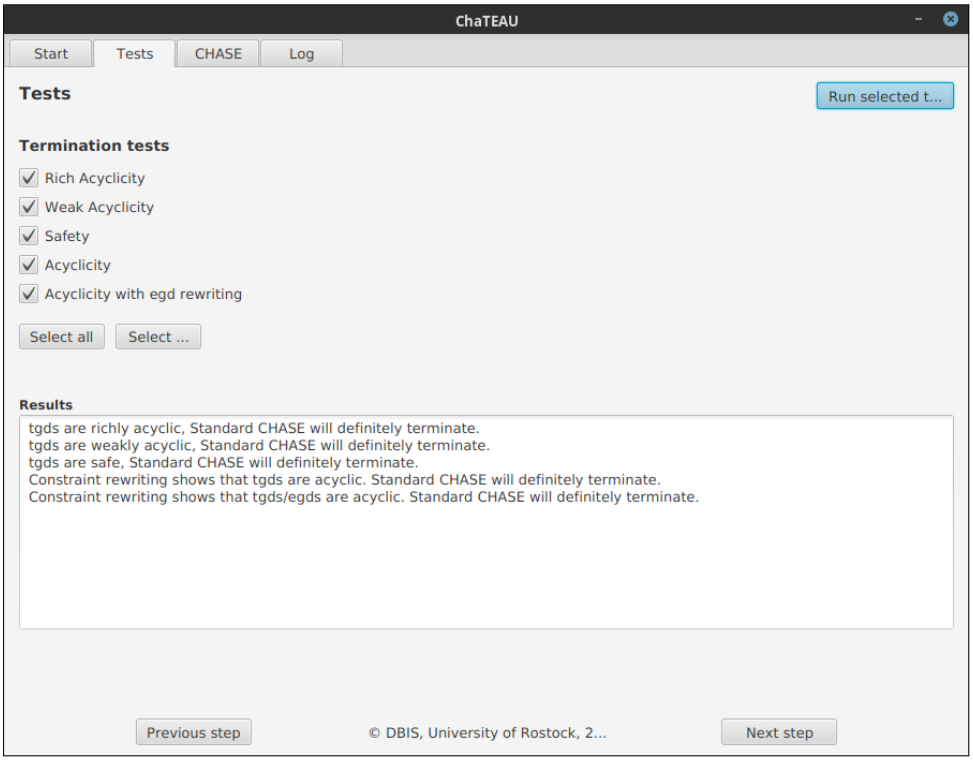


Abbildung 3: Ausführen der Tests

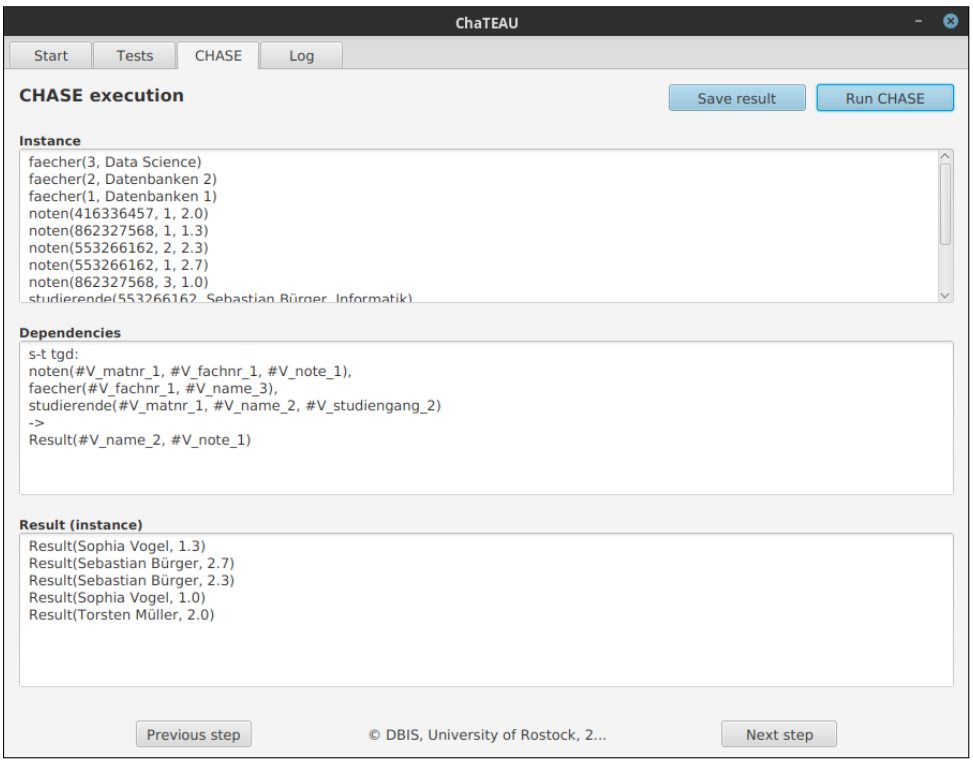


Abbildung 4: Ausführen des Chase

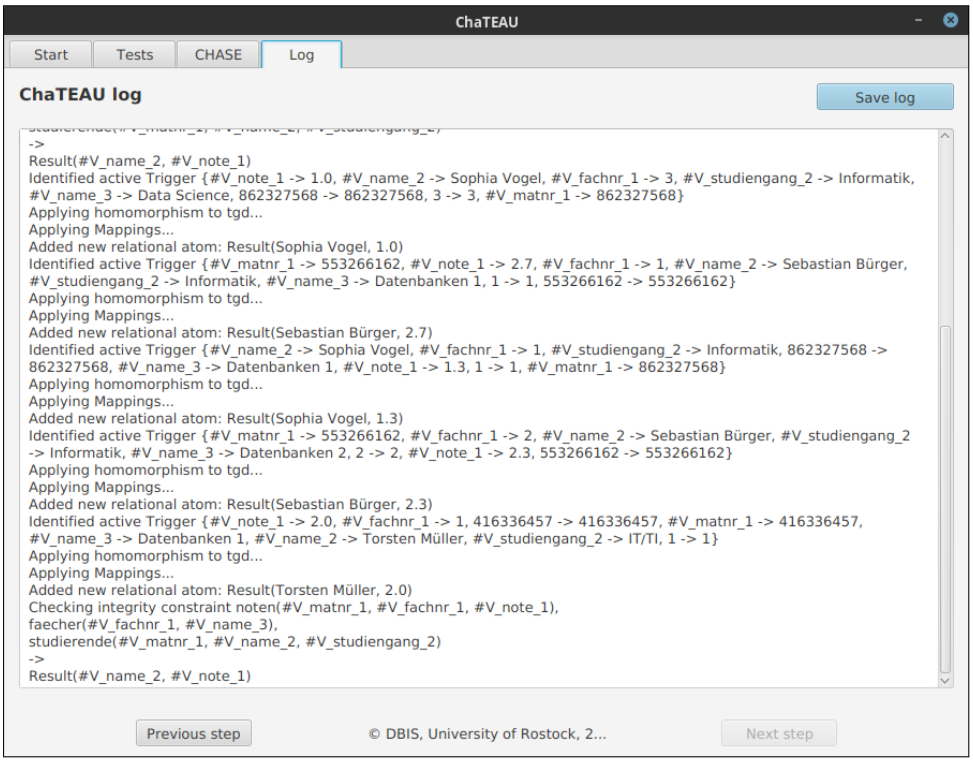


Abbildung 5: Von ChaTEAU erzeugter Log