600 Jahre Universität Rostock

Traditio et Innovatio

Bachelorarbeit zum Thema

Erweiterung des ProSA-Parsers um Aggregatfunktionen

Studiengang:	Informatik
Vorgelegt von:	Ivo Kavisanczki
Matrikelnummer:	218 203 244
Bearbeitungszeitraum:	12. Oktober 2021 – 01. März 2022
Betreuer:	M.Sc. Tanja Auge
Erstgutachter:	M.Sc. Tanja Auge
Zweitgutachter:	Dr.-Ing. Holger Meyer

Lehrstuhl für Datenbank- und Informationssysteme
Institut für Informatik
Fakultät für Informatik und Elektrotechnik
Universität Rostock

Stand: 1. März 2022

Inhalt

Kurzzusammenfassung

Das Tool ProSA dient dem Forschungsdatenmanagement, dabei wird insbesondere eine minimale Teildatenbank einer ursprünglich größeren Forschungsdatenbank berechnet. Dadurch kann die zu veröffentlichende Datenmenge reduziert werden. Gleichzeitig werden aber auch genau die Teile der Datenbank ermittelt, die notwendig sind, um das Forschungsergebnis zu reproduzieren. ProSA nutzt für diese Berechnung u.a. den CHASE-Algorithmus, ein Universalwerkzeug der Datenbanktheorie. Der CHASE arbeitet allerdings mit sog. Abhängigkeiten (egds, tgds, s-t tgds) und nicht mit Anfragen, die in SQL geschrieben werden, was sonst meistens genutzt wird, um Anfragen an Datenbanken zu formulieren. Der sog. Parser in ProSA (auch genannt sql2sttgd) löst dieses Problem bereits ansatzweise: einfache SQL-Anfragen können automatisch in Abhängigkeiten umgewandelt werden. In dieser Arbeit wird die Funktionalität des ProSA-Parsers erweitert, nämlich um Aggregatfunktionen. Dafür wird eine Übersicht gegeben, zu welchen Operatoren aus SQL theoretische Konzepte bestehen, um sie als Abhängigkeiten darzustellen. Es wird gezeigt, welche dieser Konzepte bereits implementiert wurden. Ein bestehendes Konzept zu Aggregatfunktionen wird vereinfacht und um Gruppierungen erweitert. Darüber hinaus wird ein Ansatz vorgestellt, mit dem auch Anfragen behandelt werden können, die Aggregatfunktionen zusammen mit Verbunden und Selektionen enthalten. Für einfache Aggregatfunktionen wird das angepasste Konzept im ProSA-Parser implementiert.

Abstract

The tool ProSA is used for research data management, in particular a minimal subdatabase of an originally larger research database is calculated. Thereby, the amount of data to be published can be reduced. At the same time, exactly those parts of the database are determined which are necessary to reproduce the research result. ProSA uses the CHASE-algorithm for this calculation, a universal tool of database theory. However, the CHASE works with so-called dependencies (egds, tgds, s-t tgds) and not with queries written in SQL, which is usually used to formulate queries on databases. The so-called Parser in ProSA (also called sql2sttgd) already solves this problem to some extend: simple SQL queries can be automatically converted into dependencies. This work extends the functionality of the Parser in ProSA, namely by aggregate functions. For this purpose an overview is given, for which SQL operators theoretical concepts exist, in order to represent them as dependencies. It is shown, which of these concepts have already been implemented. An existing concept for aggregate functions is simplified and extended by groupings. In addition, an approach is presented for handling queries that contain aggregate functions together with joins and selections. For simple aggregate functions the concept is implemented in the Parser of ProSA.

Kapitel 1

Einleitung

In diesem Kapitel

Das Tool ChaTEAU ist eine Implementation des CHASE-Algorithmus (s. Abschnitt 3.2.1 und [AH19]). Der CHASE-Algorithmus ist hierbei ein Werkzeug der Datenbanktheorie (s. Abschnitt 2.3.1 und [GMS12]). Mit dem CHASE können u.a. Anfragen auf Datenbankinstanzen simuliert werden. Allerdings arbeitet der CHASE mit sog. Abhängigkeiten (egds, tgds und s-t tgds, s. Abschnitt 2.3.2) als Eingabe, während Anfragen meistens als SQL-Anfrage oder als Relationenalgebra-Term formuliert werden. Mit dem Tool sql2sttgd können solche SQL-Anfragen bereits geparst und automatisch in s-t tgds transformiert werden. Diese werden dann in einem XML-Format ausgegeben, welches für ChaTEAU nutzbar ist. Dadurch müssen Anfragen nicht mehr manuell in Abhängigkeiten umgeschrieben werden.

Bisher unterstützt sql2sttgd allerdings nur eine Teilklasse möglicher SQL-Anfragen, nämlich Anfragen mit Projektion, Vereinigung und bestimmten Fällen von Verbunden und Selektionen. Die vorliegende Bachelorarbeit hat nun zum Ziel, sql2sttgd um weitere Operationen zu erweitern. Dafür werden wir vor allem Aggregatfunktionen (wie z.B. `MAX`, `MIN`, `COUNT`) betrachten.

1.1 Aufgabenstellung

Das Tool sql2sttgd erlaubt in der `SELECT`-Klausel einer SQL-Anfrage bisher nur Projektionen, allerdings noch keine Aggregatfunktionen. Diese sollen nun in der Bachelorarbeit umgesetzt werden. Dafür ist ein bestehendes Konzept aus [Aug] zu überprüfen, wenn nötig anzupassen und schließlich zu implementieren. Besondere Herausforderung bei der Formulierung von Ag-

gregatfunktionen als s-t tgd ist, dass der CHASE reihenfolgeunabhängig arbeitet, während bei der Berechnung von Aggregatfunktionen Reihenfolgen betrachtet werden müssen (da garantiert werden muss, dass jedes Tupel nur einmal berücksichtigt wird). Das Konzept sieht hierfür die Erzeugung von Hilfstabellen mit zusätzlichen Tupel-IDs vor.

Die Operationen `SELECT *` und das kartesische Produkt konnten bereits im KSWS-Projekt abgeschlossen werden. Neben den beiden genannten Schwerpunkten sollen nach Möglichkeit auch die folgenden Operationen implementiert werden:

- `OR` im `WHERE`-Teil (bisher wird nur `AND` unterstützt)
- Selektion auf Ungleichheit (bisher nur Selektion auf Gleichheit)
- Negation im Sinne der Differenz (bisher nicht unterstützt)

1.2 Durchgängiges Beispiel

In diesem Abschnitt wird eine kleine Datenbank vorgestellt, welche im Rest der Arbeit genutzt wird, um beispielhaft Konzepte zu veranschaulichen. Sie besteht aus drei Relationen *Students*, *Modules* und *Grades*, die an eine Studierenden-Datenbank angelehnt sind. Teile der Relationen sind hier aufgeführt (s. die Tabellen unter 1.1), die vollständigen Tabellen sind im Anhang D zu finden.

Im nächsten Kapitel beginnen wir mit einer kurzen Einführung, wie Anfragen in SQL oder mit der Relationenalgebra formuliert werden können. Außerdem wird der CHASE-Algorithmus kurz vorgestellt und die eben erwähnten Abhängigkeiten werden definiert.

student_id	surname	forename	course	age
1	Miller	Mark	Electrical Engineering	18
2	Lewis	Linda	Computer Science	20
3	Smith	Steven	Mathematics	21
		⋮		

(a) *Students*

module_id	module_name	points
1	Embedded Systems	6
2	Functional Programming	3
3	Linear Algebra	9
	⋮	

(b) *Modules*

module_id	student_id	semester	grade
1	1	5	2.0
2	2	6	1.3
3	3	7	2.7
	⋮		

(c) *Grades*

Tabelle 1.1: Die ersten Einträge der Relationen *Students*, *Modules* und *Grades*

Kapitel 2

Grundlagen

In diesem Kapitel

In diesem Kapitel sollen Grundlagen erklärt werden, die hilfreich sind, um die vorliegende Arbeit verstehen zu können. Dafür werden wir uns zuerst anschauen, wie Anfragen an Datenbanken mittels SQL bzw. Relationenalgebra formuliert werden können. Die wichtigsten Schlüsselwörter und Operatoren werden hierbei beschrieben. Dann werden wir eine kurze Einführung zum CHASE-Algorithmus geben, mit besonderem Fokus darauf, wie der CHASE auf Instanzen arbeitet. Hierfür werden auch die Abhängigkeiten (egds, tgds, s-t tgds) definiert, die in diesem Fall eine Eingabe des CHASE sind. Es wird auch kurz konzeptuell erklärt, wie Anfragen (z.B. aus SQL) als Abhängigkeiten dargestellt werden können. Letzteres wird dann im weiteren Verlauf der Arbeit weiter vertieft, v.a. an verschiedenen Stellen im Kapitel 3.

2.1 SQL-Anfragen

SQL ist der De-Facto-Standard, um Anfragen an Datenbanken zu stellen. Wir wollen einen groben Überblick geben, welche Elemente in einer solchen SQL-Anfrage enthalten sein können. So können wir später verstehen, wie diese Elemente auf Abhängigkeiten abgebildet werden können, wenn Anfragen mit dem CHASE gestellt werden sollen. Für den Abschnitt orientieren wir uns am Lehrbuch [HSS18]. Um einen Überblick über Operationen in SQL zu haben, wurde außerdem die Dokumentation des Datenbank-Managementsystems PostgreSQL genutzt [Pos21].

SQL-Klausel	Bedeutung
`SELECT`	Ergebnisschema (durch Projektionsliste), arithmetische Funktionen, Aggregatfunktionen
`FROM`	verwendete Relationen und Verknüpfungen zwischen diesen durch verschiedene Joins, Umbenennungen
`WHERE`	Selektions- und Verbundbedingungen, verschachtelte Anfragen
`GROUP BY`	Gruppierung nach gleichen Attributwerten
`HAVING`	Selektion von Gruppen, die durch `GROUP BY` erzeugt wurden (ähnlich wie `WHERE` einzelne Tupel selektiert)
`ORDER BY`	Sortierung der Ergebnisrelation

Tabelle 2.1: Bedeutung verschiedener Klauseln in einer SQL-Anfrage

2.1.1 Aufbau einer SQL-Anfrage

Zuallererst schauen wir uns dafür den strukturellen Aufbau einer SQL-Anfrage an: sie besteht aus einer Folge von Klauseln, welche jeweils durch ein Keyword eingeleitet werden, worauf dann die Argumente der Klausel folgen. Notwendig sind dabei die `SELECT`- und die `FROM`-Klausel; weitere Klauseln sind optional (z.B. `WHERE`, `GROUP BY`, `ORDER BY`). In Tabelle 2.1 wird eine Übersicht über einige Funktionen der einzelnen Klauseln gegeben. Neben diesen Klauseln einer elementaren Anfrage sind für diese Arbeit außerdem die Schlüsselworte `UNION`, `EXCEPT` und `INTERSECT` wichtig, mit denen die Vereinigung, die Differenz oder der Durchschnitt zweier Anfragen dargestellt werden kann.

Beispiel 1. Wir betrachten die Datenbank, die in Abschnitt 1.2 vorgestellt wurde (vollständig zu sehen in Anhang D). Eine einfache Anfrage nach den Nachnamen aller Mathematik-Studierenden könnte so aussehen (mit dem Ergebnis der Anfrage rechts):

```
SELECT surname
FROM Students
WHERE course = 'Mathematics'
```

surname
Smith
Smith
Jackson

Hierbei fällt auf, dass im Ergebnis ein Duplikat vorhanden ist: der Nachnahme „Smith“ taucht doppelt auf. In SQL bleiben solche Duplikate erhalten, wenn nicht durch ein `DISTINCT` in der `SELECT`-Klausel anders gekennzeichnet. An Beispiel 4 werden wir später sehen, dass Duplikate in der Relationenalgebra dagegen eliminiert werden. ■

2.1.2 Verbundarten

In der `FROM`-Klausel werden die verwendeten Relationen angegeben, d.h. jene Relationen, auf denen die Anfrage ausgeführt wird, und auch, wie diese untereinander durch Verbunde verknüpft werden. In Tabelle 2.2 werden mögliche Formen solcher Verbunden aufgeführt. Für äußere Ver-

Verbund in SQL	Verbundart
`R NATURAL JOIN S`	natürlicher Verbund
`R CROSS JOIN S`	Kreuzprodukt, alternativ auch kurz als `R, S`
`R [INNER] JOIN S ON (R.a = S.a)`	innerer Verbund über `a`, hinter `ON` kann eine beliebige Verbundbedingung stehen
`R [INNER] JOIN S USING a`	innerer Verbund über `a`, hinter `USING` werden die Verbundattribute aufgelistet
`R NATURAL FULL [OUTER] JOIN S`	natürlicher äußerer Verbund, bei dem sog. „Dangling Tuples“ (also Tupel ohne Verbundpartner) aus `R` und `S` mit Nullwerten aufgefüllt ins Ergebnis aufgenommen werden

Tabelle 2.2: Ausgewählte Verbundarten in SQL (eckige Klammern [] geben optionale Keywords an)

Operator	Bedeutung
<,<=,=,>=,>,<>	Vergleiche eines Attributs mit einer Konstante, mit einem anderen Attribut oder auch mit einer Unteranfrage (wenn in Verbindung mit einem Quantor, also `ANY` oder `ALL`)
`AND`	logisches Und
`OR`	logisches Oder
`NOT`	logische Negation
`EXISTS`	prüft, ob das Ergebnis einer Unteranfrage nicht leer ist
`IN`	prüft, ob ein Tupel im Ergebnis einer Unteranfrage enthalten ist
`ANY`/`SOME`	prüft, ob mindestens ein Tupel einer Unteranfrage eine Bedingung erfüllt
`ALL`	prüft, ob alle Tupel einer Unteranfrage eine Bedingung erfüllen

Tabelle 2.3: Ausgewählte Operatoren in der Selektionsbedingung

bunde wird hierbei repräsentativ nur der `NATURAL FULL OUTER JOIN` aufgeführt. Statt eines natürlichen Verbundes gibt es auch hier Varianten mit `ON` und `USING` (ebenso wie beim inneren Verbund), wo explizit Verbundprädikate/-attribute angegeben werden können. Anstelle des Keywords `FULL` kann auch `LEFT` oder `RIGHT` verwendet werden. Ins Ergebnis werden dann entsprechend nur noch Dangling Tuples aus der linken oder rechten Relation aufgenommen.

2.1.3 Selektionsbedingungen

SQL erlaubt außerdem komplexe Selektionsbedingungen in der `WHERE`-Klausel. In Tabelle 2.3 werden einige wichtige Operatoren und Keywords aufgelistet, die in einer solchen Selektionsbedingung verwendet werden können. Dabei wurde unterteilt in Vergleichsoperatoren wie `<=`, logische Operatoren wie `AND` und Operatoren für geschachtelte Anfragen wie `IN`.

Aggregation	Bedeutung
MIN	Minimum
MAX	Maximum
AVG	arithmetisches Mittel
SUM	Summe
COUNT	Anzahl der Tupel

Tabelle 2.4: Ausgewählte Aggregationsfunktionen in SQL

Beispiel 2. Angenommen, wir wollen die Anfrage aus Beispiel 1 noch weiter konkretisieren, weil bspw. nicht alle Mathematik-Studierenden gesucht werden, sondern nur jene, die auch in einem bestimmten Fach eine Prüfung abgelegt haben (hier das Fach mit der ID 3). Die um entsprechende Selektionen und Verbunde erweiterte Anfrage mit Ergebnis sähe dann so aus:

```
SELECT surname
FROM Students NATURAL JOIN Grades
WHERE course = 'Mathematics'
    AND module_id = 3
```

surname
Smith
Smith

■

2.1.4 Aggregationen

Gegenstand dieser Arbeit werden insbesondere Aggregationsfunktionen sein. Das ist eine Gruppe von Funktionen, die gemein haben, dass sie eine Menge von Werten/Tupeln auf einen einzelnen Wert abbilden, d.h. sie aggregieren Werte. In SQL werden Aggregationsfunktionen in die SELECT-Klausel geschrieben. Als Argument dient meistens das Attribut, auf welches die Funktion angewendet werden soll. Neben einfachen Aggregationen, wie sie in Tabelle 2.4 beschrieben werden, gibt es u.a. auch noch Funktionen auf boolschen Werten, auf XML- oder JSON-Values und Funktionen zur statistischen Analyse [Pos21]. Diese werden hier aber nicht betrachtet. Mit dem Keyword AS können Spalten im Ergebnis auch umbenannt werden, um es besser lesbar zu machen (ansonsten wird die Spalte nach der Funktion benannt).

Beispiel 3. Eine Anfrage nach der Durchschnittsalter aller Studierenden könnte dann so aussehen (Anfrageergebnis rechts):

```
SELECT AVG(age)
FROM Students
```

avg
21.5

■

Aggregationen werden häufig in Kombination mit den Klauseln GROUP BY und HAVING verwendet. Hinter GROUP BY wird eine Liste von Attributen angegeben, nach denen gruppiert werden soll. In Gruppen werden Tupel mit gleichen Werten für die Gruppierungsattribute zusammengefasst. Wenn gruppiert wird, dann wird die Aggregation nicht über die gesamte Tabelle berechnet, sondern je Gruppe.

Die HAVING-Klausel dient, ähnlich wie die WHERE-Klausel, der Selektion. In der HAVING-Klausel können prinzipiell auch die gleichen Expressions verwendet werden, wie bei WHERE. Der Unterschied ist, dass WHERE *einzelne Tupel* vor der Gruppierung selektiert, während HAVING *ganze Gruppen* filtert (also notwendigerweise nach der Gruppierung). Sinnvoll sind in der HAVING-Klausel deshalb vor allem Bedingungen, die auch Aggregationen enthalten.

Beispiel 4. Die Anfrage nach dem Durchschnittsalter je Studiengang lässt sich mit folgender Anfrage berechnen:

```
SELECT course, AVG(age)
FROM Students
GROUP BY course
```

course	avg_age
Electrical Engineering	18
Computer Science	21
Mathematics	23

Die Aggregatfunktion AVG(age) wird in diesem Fall nicht über alle Tupel gebildet, sondern separat je Studiengang, da nach course gruppiert wird. In der betrachteten Datenbank (s. Anhang D) existieren Einträge für drei Studiengänge (drei unterschiedliche Attributwerte): „Electrical Engineering“, „Computer Science“ und „Mathematics“. Deshalb werden drei Gruppen für das Ergebnis gebildet. ■

Beispiel 5. Wie eben in Beispiel 4 suchen wir wieder das Durchschnittsalter je Studiengang. Diesmal stellen wir aber die zusätzliche Bedingung, dass nur jene Studiengänge gezeigt werden sollen, in denen das Durchschnittsalter über 20 ist. Anfrage und Ergebnis lauten:

```
SELECT course, AVG(age)
FROM Students
GROUP BY course
HAVING AVG(age) > 20
```

course	avg_age
Computer Science	21
Mathematics	23

Wir sehen, dass im Vergleich zu Beispiel 4 der Eintrag zu Electrical Engineering fehlt, da das Durchschnittsalter hier 18 ist. ■

Es wurde eine Einführung zu Anfragen an Datenbanken mittels SQL gegeben. SQL wird vor allem in der Praxis genutzt, um Anfragen zu stellen. In der Theorie spielt die sog. Relationenalgebra eine bedeutende Rolle. Diese soll im nächsten Abschnitt vorgestellt werden.

2.2 Anfragen in Relationenalgebra

Die Relationenalgebra ist das gängige theoretische Modell, um Anfragen auf Datenbanken zu formulieren. Sie definiert verschiedene Operationen, die Relationen als Argumente erhalten, um Anfragen zu beschreiben. In Tabelle 2.5 werden einige dieser Operationen vorgestellt. Der obere Teil der Tabelle enthält dabei die Basis-Operationen der Relationenalgebra, während im unteren

Symbol	Bezeichnung	Bedeutung
σ	Selektion	Streichen von Tupeln
π	Projektion	Streichen von Attributen
$\bowtie$	natürlicher Verbund	Verknüpfen von Relationen über gleichnamige Attribute
$\cup$	Vereinigung	Zusammenfassen der Tupel zweier Relationen mit dem gleichen Schema
$-$	Differenz	Entfernen der Tupel einer Relation, die in einer anderen vorkommen (auch hier gleiches Schema notwendig)
β	Umbenennung	Umbenennung von Attributen (u.a. wichtig um Schemata mit unterschiedlichen Namen für $\bowtie, \cup, -$ anzugleichen)
$\cap$	Durchschnitt	Verbleiben der Tupel, die in zwei Relationen vorkommen (gleiches Schema notwendig); keine Basis-Operation, da mit Differenz darstellbar ($r_1 \cap r_2 = r_1 - (r_1 - r_2)$)
$\times$	Kreuzprodukt	Verknüpfen von Relationen, bei dem jedes Tupel der einen Relatione mit jedem Tupel der anderen kombiniert wird; keine Basis-Operation, da mit $\bowtie$ und β darstellbar
γ	Gruppierung	Gruppierung von Tupeln mit gleichen Attributwerten; zusätzlich Funktionen als Parameter, die je Gruppe angewandt werden (um `GROUP BY` und Aggregationen aus SQL darzustellen)

Tabelle 2.5: Operationen der Relationenalgebra (Basis-Operationen oben, Erweiterungen unten), nach [HSS18]

Teil Operationen aufgelistet werden, die sich mit auch mit Basis-Operationen darstellen lassen ($\cap, \times$) oder die nötig sind, um die Mächtigkeit zu erweitern und somit weitere SQL-Anfragen darstellen zu können (γ). Selektion, Projektion, Umbenennung und Gruppierung sind dabei unäre Operationen, während die restlichen binär sind. Parameter wie Selektionsbedingungen oder Projektionslisten werden in dieser Arbeit in Indexschreibweise notiert.

Ein wichtiger Unterschied der Relationenalgebra im Vergleich zu SQL ist, dass die Relationenalgebra auf *Mengen* von Tupeln arbeitet, d.h. die Reihenfolge spielt keine Rolle und es gibt auch keine Duplikate. Somit lassen sich Operationen wie `ORDER BY` (Sortierung) und `DISTINCT` (Duplikateliminierung) nicht ohne Weiteres abbilden.

Beispiel 6. Die SQL-Anfrage aus Beispiel 2 lässt sich auf verschiedene Weisen in Relationenalgebra darstellen; hier eine Möglichkeit (man beachte v.a. auch, dass hier das Tupel „Smith“ nur einmal im Ergebnis enthalten ist):

$$\pi_{\text{surname}}(\sigma_{\text{course}=\text{„Mathematics“}}(\text{Students}) \bowtie \sigma_{\text{module\_id}=3}(\text{Grades}))$$

surname
Smith

■

Beispiel 7. Wir wollen auch die SQL-Anfrage aus Beispiel 4 in Relationenalgebra darstellen. Aufgrund der Aggregation geht dies nur unter Hinzunahme des γ-Operators. Bei der Gruppierung schreiben wir als Parameter zuerst eine Liste von Funktionen und dann nach dem Semikolon die Liste von Gruppierungsattributen. In diesem Fall ist also „avg(age)“ die anzuwendende Funktion

und „course“ das Gruppierungsattribut. Als Name für das Attribut, welches durch die Aggregation entstanden ist, wird einfach der Funktionsname verwendet (also „avg“). Das ist dann im nächsten Schritt wichtig für die Projektion.

$\pi_{\text{course, avg}}(\gamma_{\text{avg(age); course}}(\text{Students}))$

course	avg
Electrical Engineering	18
Computer Science	21
Mathematics	23

■

Wir haben damit zwei gängige Sprachen betrachtet, mit denen Anfragen an Datenbanken formuliert werden: SQL und die Relationenalgebra. Als nächstes wird der CHASE-Algorithmus vorgestellt, der u.a. auch gebraucht werden kann, um Anfragen an Datenbanken zu beantworten.

2.3 Rund um den CHASE

Der CHASE-Algorithmus ist ein Universalwerkzeug der Datenbanktheorie, das in vielfältigen Anwendungsgebieten eingesetzt wird. So kann damit u.a. ein Datenbankschema optimiert oder Data Cleaning betrieben werden [AH19]. In diesem Abschnitt wollen wir eine kurze Einführung zum CHASE geben und die sog. Abhängigkeiten definieren, die als Parameter dienen. Außerdem wird in Abschnitt 2.3.3 kurz gezeigt, wie man mit solchen Abhängigkeiten auch Anfragen mit dem CHASE stellen kann.

2.3.1 Der CHASE-Algorithmus

Nach [AH19] arbeitet der CHASE einfach gesagt eine Menge von Abhängigkeiten $\star$ in ein Objekt $\bigcirc$ ein, sodass $\star$ implizit in $\bigcirc$ enthalten ist, also:

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

Das Objekt $\bigcirc$ kann dabei z.B. eine Datenbank, ein Datenbankschema oder eine Anfrage sein. Der Parameter $\star$ wird als Menge von Abhängigkeiten gegeben. Das sind logische Formeln von bestimmter Form; sie werden im folgenden Abschnitt 2.3.2 definiert. Sie generalisieren von Funktionalen Abhängigkeiten (engl. *functional dependencies*, kurz FDs) und Verbundabhängigkeiten (engl. *join dependencies*, kurz JDs). In dieser Arbeit soll vor allem der Fall betrachtet werden, wo mittels des CHASE Anfragen an Datenbanken gestellt werden. Das Objekt $\bigcirc$ ist dann entsprechend eine Datenbank, der Parameter $\star$ sind Abhängigkeiten, die in diesem Fall eine Anfrage darstellen. Die Rückgabe $\circledast$ ist das Anfrageergebnis.

2.3.2 Abhängigkeiten

Wie eben angedeutet, erhält der CHASE den Parameter $\star$ in Form von bestimmten prädikatenlogischen Formeln, die allgemein als (eingebettete) Abhängigkeiten bezeichnet werden [GMS12]. Im Folgenden werden wir zwei Formen solcher Abhängigkeiten definieren, nämlich egds (Def. 1) und tgds (Def. 2). Die egds stellen dabei eine Verallgemeinerung der FDs dar, weil damit ausgedrückt werden kann, dass Attributwerte unter bestimmten Voraussetzungen gleich sein müssen. Die tgds auf der anderen Seite sind eine Verallgemeinerung der JDs, da sie beschreiben können, dass Tupel, die für einen Verbund notwendig sind, existieren müssen. Zur Darstellung von Anfragen wird hauptsächlich eine besondere Form von tgds verwendet, nämlich sogenannte s-t tgds (Def. 3). Im Laufe dieser Arbeit werden wir später aber auch „normale“ tgds (also tgds, die keine s-t tgds sind) hinzunehmen, um Anfragen darstellen zu können.

Definition 1 (equality generating dependency (egd)). Nach [GMS12; Ben+17; Fag+11]. Eine egd ist ein prädikatenlogischer Ausdruck erster Stufe der Form

$$\forall \mathbf{x} : (\varphi(\mathbf{x}) \rightarrow x_i = x_j).$$

Dabei ist $\mathbf{x}$ ein Vektor von Variablen mit $x_i, x_j \in \mathbf{x}$ und $\varphi(\mathbf{x})$ ist eine Konjunktion von Atomen mit Variablen aus $\mathbf{x}$. ■

Definition 2 (tuple generating dependency (tgd)). Nach [GMS12; Ben+17; Fag+11]. Eine tgd ist ein prädikatenlogischer Ausdruck erster Stufe der Form

$$\forall \mathbf{x} : (\varphi(\mathbf{x}) \rightarrow \exists \mathbf{y} : \psi(\mathbf{x}, \mathbf{y})).$$

Dabei sind $\mathbf{x}$ und $\mathbf{y}$ Vektoren von Variablen; $\varphi(\mathbf{x})$ und $\psi(\mathbf{x}, \mathbf{y})$ sind Konjunktionen von Atomen mit Variablen aus $\mathbf{x}$ bzw. $\mathbf{x} \cup \mathbf{y}$. ■

Definition 3 (source-to-target tgd (s-t tgd)). Nach [Fag+11; Sch21]. Eine s-t tgd ist eine spezielle tgd, wobei $\varphi(\mathbf{x})$ eine Konjunktion von Atomen über einem Quellschema S (source) ist, während $\psi(\mathbf{x}, \mathbf{y})$ eine Konjunktion von Atomen über einem Zielschema T (target) ist. ■

Die in den Definitionen genannten Atome werden dabei hauptsächlich relationale Atome sein. Nach unseren Definitionen sind aber auch weitere prädikatenlogische Atome zugelassen, wie etwa Gleichheitsatome ($x_i = x_j$), Ungleichheitsatome ($x_i \, \theta \, x_j$ mit $\theta \in \{<, \leq, \neq, \geq, >\}$) und auch Negationen von Atomen. Es ist anzumerken, dass damit stückweise von den Definitionen in [GMS12] und [Ben+17] abgewichen wird; in dieser Hinsicht orientieren wir uns eher an [Fag+11]. Das ist notwendig, um weitere Klassen von Anfragen als Abhängigkeiten darstellen zu können (z.B. Selektionen nach Ungleichheit), was sonst nicht möglich wäre.

Nun noch kurz einige Anmerkungen zu Bezeichnungen und Notationen:

- Die linke Seite einer egd oder tgd (d.h. $\varphi(\mathbf{x})$) bezeichnen wir als Rumpf (engl. *body*) und die rechte Seite (d.h. $x_i = x_j$ bzw. $\exists \mathbf{y} : \psi(\mathbf{x}, \mathbf{y})$) als Kopf (engl. *head*).
- Um die Notation von Abhängigkeiten zu verkürzen, werden die Quantoren häufig weggelassen. Existenzquantifizierte Variablen werden dann zur Unterscheidung groß geschrieben.
- Wenn eine Variable x_i einer anderen Variable x_j gleichgesetzt wird, dann ersetzen wir mitunter alle Vorkommen von x_j durch x_i (oder andersherum). Gleiches gilt für Konstanten, also wird $R(\dots, x_i, \dots) \wedge x_i = \mathrm{c}$ zu $R(\dots, \mathrm{c}, \dots)$.

2.3.3 Darstellung von Anfragen als Abhängigkeiten

Wir haben bisher viel davon gesprochen, dass wir Anfragen aus SQL oder der Relationenalgebra auch als Parameter für den CHASE darstellen wollen. Aber wie genau funktioniert das überhaupt? Das wollen wir uns nun in diesem Abschnitt näher anschauen.

Eine Anfrage wird dem CHASE als Menge von Abhängigkeiten als Parameter übergeben. In den meisten Fällen reicht dabei eine s-t tgd, um eine Anfrage darzustellen. Das Quellschema der s-t tgds ist dabei das Datenbankschema; das Zielschema ist das Schema der Ergebnisrelation. Demzufolge tauchen die Relationen, an welche die Anfrage gestellt wird, im Rumpf der s-t tgd auf. Im Kopf steht dann nur ein Atom, nämlich für die Ergebnisrelation. Dieses wird im Folgenden *Result* oder kurz *Res* genannt.

Wir wollen kurz erklären, welche Korrespondenzen zwischen Operationen in SQL/Relationenalgebra und Anfragen als s-t tgds bestehen. Der Zugriff auf eine Relation entspricht einem relationalen Atom im Rumpf der s-t tgd (welches mit weiteren Atomen per Konjunktion verknüpft ist). Eine Selektion der Form $\sigma_{A=\langle Konstante \rangle}$ erhält man durch Hinzufügen eines Gleichheitsatoms im Rumpf, in welchem die Variable für A der Konstante gleich gesetzt wird (bzw. durch Ersetzen der Variable durch die Konstante). Ein Verbund der Form $R_1 \bowtie_A R_2$ wird realisiert, indem in den beiden Atomen für R_1 und R_2 die gleiche Variable für das Attribut A verwendet wird. Analog funktioniert auch ein Verbund über mehrere Attribute. Für eine Projektion auf eine Attributmenge (π_X) muss das Result-Atom verändert werden, da es das Schema der Ergebnisrelation angibt. Das Result-Atom soll dann nur noch Variablen enthalten, die Attributen in X entsprechen.

Beispiel 8. Die eben genannten Korrespondenzen verstehen sich vermutlich am besten anhand eines Beispiels (angelehnt an [Sch21]). Dafür sollen die Anfragen aus den vorangegangenen Beispielen 2 und 6 (dieselbe Anfrage jeweils in SQL und Relationenalgebra) nun als s-t tgd formuliert werden. Die Anfragen aus den genannten Beispielen waren:

```
SELECT surname
FROM Students NATURAL JOIN Grades
WHERE course = 'Mathematics'
    AND module_id = 3
```

$$\pi_{\text{surname}}(\sigma_{\text{course}=\text{„Mathematics“}}(\text{Students}) \bowtie \sigma_{\text{module\_id}=3}(\text{Grades}))$$

Die entsprechende s-t tgd dazu lautet:

$$\begin{aligned} &\mathit{Students}(\mathit{student\_id}_1, \mathit{surname}_1, \mathit{forename}_1, \mathit{course}_1, \mathit{age}_1) \wedge \mathit{course}_1 = \text{Mathematics} \wedge \\ &\mathit{Grades}(\mathit{module\_id}_1, \mathit{student\_id}_1, \mathit{semester}_1, \mathit{grade}_1) \wedge \mathit{module\_id}_1 = 3 \\ &\rightarrow \mathit{Result}(\mathit{surname}_1) \end{aligned}$$

bzw. in verkürzter Notation (durch Einsetzen der Konstanten in die relationalen Atome):

$$\begin{aligned} &\mathit{Students}(\mathit{student\_id}_1, \mathit{surname}_1, \mathit{forename}_1, \text{Mathematics}, \mathit{age}_1) \wedge \\ &\mathit{Grades}(3, \mathit{student\_id}_1, \mathit{semester}_1, \mathit{grade}_1) \\ &\rightarrow \mathit{Result}(\mathit{surname}_1) \end{aligned}$$

Die Korrespondenzen zwischen den Anfragen wurden zum besseren Verständnis farblich hervorgehoben. Die Projektion auf das Attribut `surname` erhält man hier, indem das Result-Atom nur aus einer Variable $\mathit{surname}_1$ besteht. Diese kommt im Rumpf aus dem Atom *Students*. Der Verbund zwischen `Students` und `Grades` findet über das Attribut `student_id` statt. Er wird dargestellt, indem in den beiden Atomen *Students* und *Grades* im Rumpf die gleiche Variable $\mathit{student\_id}_1$ verwendet wird. Das zeigt an, dass der Wert für `student_id` in zwei Tupeln dieser beiden Relationen gleich sein muss, damit auf diesen Tupeln ein Verbund stattfindet. Für die Selektionen wurden die beiden Gleichheitsatome $\mathit{course}_1 = \text{Mathematics}$ und $\mathit{module\_id}_1 = 3$ hinzugefügt. ■

Nun haben wir gesehen, wie sich einfache Anfragen als s-t tgds darstellen lassen. Im nächsten Kapitel werden wir sehen, was der aktuelle Stand der Forschung ist, um auch weitere Anfragen umzuwandeln. Außerdem werden bestehende Tools vorgestellt, die diese Umwandlung implementieren oder damit im Zusammenhang stehen.

Kapitel 3

State of the Art

In diesem Kapitel

Nachdem im vorigen Kapitel notwendige Grundlagen-Kenntnisse beschrieben wurden, soll nun eine Übersicht zum Stand der Technik gegeben werden. In Abschnitt 3.1 soll dabei vor allem bestehende Theorie beschrieben und evaluiert werden. Wir betrachten, welche Konzepte bereits bestehen, um SQL-Anfragen als Abhängigkeiten für den CHASE darzustellen. Dabei untersuchen wir auch, welche Operationen aus SQL damit abgedeckt werden und welche nicht. In Abschnitt 3.2 wird dann mit sql2sttgd ein Tool vorgestellt, das ebendiese Umwandlung von SQL-Anfragen zu Abhängigkeiten implementiert. Auch die Tools ProSA und ChaTEAU werden kurz vorgestellt, da sql2sttgd stark in Zusammenhang mit ihnen steht. Abschließend wird aufgelistet, welche Operationen aus SQL bereits in sql2sttgd unterstützt werden und wo noch Lücken bestehen.

3.1 Stand der Forschung

Wir wollen nun eine Übersicht geben, welche SQL-Anfragen in der Theorie als Abhängigkeiten darstellbar sind. Wir wollen zuerst betrachten, welche Operationen „nativ“ unterstützt werden, sich also sehr natürlich in einer s-t tgd formulieren lassen. In Abschnitt 3.1.2 wird dann eine Erweiterung um Aggregatfunktionen erklärt, da diese den Schwerpunkt dieser Arbeit bilden. In Abschnitt 3.1.3 werden wir aufführen, was bereits an Theorie besteht, um auch weitere Operationen umzusetzen. Zuletzt werden noch Operationen aufgelistet, die unter Umständen umsetzbar sind, zu denen aber nach unserer Kenntnis bisher keine Theorie entwickelt wurde.

3.1.1 Kernoperationen

In Abschnitt 2.3.3 wurde knapp beschrieben, wie einige wichtige Operationen aus SQL bzw. der Relationenalgebra in s-t tgds umgesetzt werden können. Wir wollen kurz wiederholen, wie sich (einfache Fälle für) Projektion, Selektion und Verbund darstellen lassen. Diese drei Operationen bezeichnen wir im Folgenden auch als „Kernoperationen“. Viele grundlegende Anfragen aus SQL lassen sich damit bereits auf s-t tgds abbilden.

Die Projektion aus der `SELECT`-Klausel lässt sich durch Anpassen des Result-Atoms im Kopf darstellen. Ausgenommen davon sind vorerst Umbenennungen und Aggregationen, die in SQL zwar in der `SELECT`-Klausel notiert werden, aber keine Projektionen sind. Diese sind zwar darstellbar, erfordern aber jeweils weitere Arbeit. Genau genommen beinhaltet die Projektion auf Seite der s-t tgds auch immer eine Duplikateliminierung, da hier auf Mengen von Tupeln gearbeitet wird, ähnlich wie in der Relationenalgebra. D.h. das `DISTINCT` der `SELECT`-Klausel wird auch immer implizit umgesetzt.

Wenn die `WHERE`-Klausel betrachtet wird, lassen sich einige Fälle der Selektion durch Hinzufügen von Gleichheitsatomen (bzw. durch Ersetzen von Variablen) darstellen, nämlich Selektionen der Form $\langle Attribut \rangle = \langle Konstante \rangle$. Es können auch mehrere solcher Selektionen durch ein `AND` verknüpft sein, dann werden einfach ebenso die Gleichheitsatome in der s-t tgd durch ein logisches Und verknüpft. Auch Selektionen der Form $\langle Attribut \rangle = \langle Attribut \rangle$ lassen sich in gleicher Weise darstellen.

Da zuletzt genannte Selektionen meistens als Verbundbedingungen genutzt werden, kann dasselbe Prinzip auch für Verbunde genutzt werden. Ein Verbund, bei dem Attributwerte gleich sein müssen, lässt sich somit auch durch Gleichheitsatome darstellen. Umgekehrt lässt sich ein Kreuzprodukt darstellen, indem zwischen den Atomen keine gleichen Variablen benutzt werden. Beliebige Verbundprädikate (wie sie in einer `ON`-Expression stehen können) werden mit dieser Methode aber noch nicht abgedeckt und auch nicht die äußeren Verbunde, d.h. es werden keine Dangling Tuples ins Ergebnis aufgenommen.

3.1.2 Erweiterung um Aggregationen

Wir haben gesehen, wie die Operationen Projektion, Selektion und Verbund dargestellt werden. Eine wichtige weitere Klasse von Operationen sind die Aggregatfunktionen. Sie bilden eine Menge von Eingabewerten auf einen Ergebniswert ab, z.B. die Summe über die gesamte Spalte einer Tabelle. Das hier beschriebene Konzept stammt aus [Aug]. Dort wird ausführlich beschrieben, wie die Aggregatfunktionen `MIN`, `MAX`, `SUM`, `AVG` und `COUNT` für einfache Fälle als (s-t) tgds dargestellt werden können. Die Herausforderung bei den Aggregatfunktionen allgemein ist vor allem, dass sichergestellt werden muss, dass jedes Tupel genau einmal in die Berechnung eingeht. Das ist deshalb schwierig, weil der CHASE mengenorientiert arbeitet. Die Idee des Ansatzes in [Aug]

$$
\begin{aligned}
R(a_1,b_1,t_1) \wedge \neg\tilde{R}(x_2,a_2,b_2,t_2) &\rightarrow S'(t_1) \wedge \tilde{R}(1,a_1,b_1,t_1) \\
R(a_1,b_1,t_1) \wedge \neg S'(t_1) \wedge \tilde{R}(x_2,a_2,b_2,t_2) \wedge \neg S(t_2) &\rightarrow S'(t_1) \wedge \tilde{R}(x_2+1,a_1,b_1,t_1) \wedge S(t_2) \\
\tilde{R}(1,a_1,b_1,t_1) &\rightarrow S''(t_1) \wedge H(b_1,1) \\
\tilde{R}(x_1,a_1,b_1,t_1) \wedge \neg S''(t_1) \wedge H(c_1,x_1-1) &\rightarrow S''(t_1) \wedge H(\max\{b_1,c_1\},x_1) \\
H(c_1,x_1) \wedge \neg H(c_2,x_1+1) &\rightarrow Res(c_1)
\end{aligned}
$$

(a) `MAX(b)` / `MIN(b)` (mit $\min\{b,c_2\}$ anstelle von $\max\{b,c_2\}$)

$$
\begin{aligned}
R(a_1,b_1,t_1) \wedge \neg\tilde{R}(x_2,a_2,b_2,t_2) &\rightarrow S'(t_1) \wedge \tilde{R}(1,a_1,b_1,t_1) \\
R(a_1,b_1,t_1) \wedge \neg S'(t_1) \wedge \tilde{R}(x_2,a_2,b_2,t_2) \wedge \neg S(t_2) &\rightarrow S'(t_1) \wedge \tilde{R}(x_2+1,a_1,b_1,t_1) \wedge S(t_2) \\
\tilde{R}(1,a_1,b_1,t_1) &\rightarrow S''(t_1) \wedge H(b_1,1) \\
\tilde{R}(x_1,a_1,b_1,t_1) \wedge \neg S''(t_1) \wedge H(c_1,x_1-1) &\rightarrow S''(t_1) \wedge H(b_1+c_1,x_1) \\
H(c_1,x_1) \wedge \neg H(c_2,x_1+1) &\rightarrow Res(c_1)
\end{aligned}
$$

(b) `SUM(b)`

$$
\begin{aligned}
R(a_1,b_1,t_1) \wedge \neg\tilde{R}(x_2,a_2,b_2,t_2) &\rightarrow S'(t_1) \wedge \tilde{R}(1,a_1,b_1,t_1) \\
R(a_1,b_1,t_1) \wedge \neg S'(t_1) \wedge \tilde{R}(x_2,a_2,b_2,t_2) \wedge \neg S(t_2) &\rightarrow S'(t_1) \wedge \tilde{R}(x_2+1,a_1,b_1,t_1) \wedge S(t_2) \\
\tilde{R}(1,a_1,b_1,t_1) &\rightarrow S''(t_1) \wedge H(b_1,1) \\
\tilde{R}(x_1,a_1,b_1,t_1) \wedge \neg S''(t_1) \wedge H(c_1,x_1-1) &\rightarrow S''(t_1) \wedge H(b_1+c_1,x_1) \\
H(c_1,x_1) \wedge \neg H(c_2,x_1+1) &\rightarrow Res(c_1/x_1)
\end{aligned}
$$

(c) `AVG(b)`

$$
\begin{aligned}
R(a_1,b_1,t_1) \wedge \neg\tilde{R}(x_2,a_2,b_2,t_2) &\rightarrow S'(t_1) \wedge \tilde{R}(1,a_1,b_1,t_1) \\
R(a_1,b_1,t_1) \wedge \neg S'(t_1) \wedge \tilde{R}(x_2,a_2,b_2,t_2) \wedge \neg S(t_2) &\rightarrow S'(t_1) \wedge \tilde{R}(x_2+1,a_1,b_1,t_1) \wedge S(t_2) \\
\tilde{R}(x_1,a_1,b_1) \wedge \neg\tilde{R}(x_1+1,a_2,b_2) &\rightarrow Res(x_1)
\end{aligned}
$$

(d) `COUNT(*)`

Tabelle 3.1: Aggregatfunktionen als Mengen von (s-t) tgds nach [Aug]

ist es, neue Tupel-IDs hinzuzufügen und dann in Hilfstabellen bereits bearbeitete Tupel zu vermerken. Das Ergebnis wird dabei induktiv über die Tupelmenge berechnet. Das ist nur deshalb möglich, weil jede der oben genannten Funktionen induktiv berechenbar ist. Bei der Summe z.B. wird die gesamte Summe ermittelt, indem bei jedem Schritt nur zwei Werte addiert werden. Der erste Wert ist die bereits berechnete Summe einer Teilmenge aller Tupel. Der zweite ist der Attributwert eines einzelnen Tupels, das noch nicht bearbeitet wurde. Sobald alle Tupel bearbeitet wurden, wird der akkumulierte Wert in der Ergebnisrelation ausgegeben. `COUNT` kann als Spezialfall von `SUM` betrachtet werden; hier wird immer nur 1 addiert. Auch Minimum und Maximum lassen sich auf ähnliche Weise wie die Summe umsetzen. Beim arithmetischen Mittel ist ein weiterer Schritt notwendig. Hier kann nicht einfach der Durchschnitt eines Zwischenergebnisses und eines neuen Attributwertes berechnet werden. Dieses Ergebnis stimmt im Allgemeinen nicht mit dem richtigen Durchschnitt überein. Um das korrekte Ergebnis zu erhalten, werden sowohl die Summe über alle Attributwerte als auch die Tupelanzahl ermittelt. Das Ergebnis ergibt sich dann zum Schluss als Quotient aus Summe durch Anzahl.

Verwendete Relationen

In Tabelle 3.1 werden die tgds und s-t tgds für die genannten Aggregatfunktionen dargestellt. Sie wurden aus [Aug] übernommen, aber die Notation wurde leicht modifiziert (R_{sorted} wurde zu $\tilde{R}$ umbenannt; $1, 2, \ldots$ werden statt i und j als Indizes genutzt; alle Funktionsanwendungen wie $f(b,c) = b + c$ wurden direkt in die Atome eingesetzt). Die Aggregation soll auf einer Relation $R(a,b,t)$ angewandt werden. Dabei ist das Attribut b das Argument der Aggregatfunktion, a sei ein beliebiges anderes Attribut (es könnten analog auch mehrere sein). Das Attribut t sei Schlüssel für die Relation.
$\tilde{R}(x,a,b,t)$ ist eine modifizierte Variante der Relation R. Sie wird nur intern zur Berechnung genutzt und ist zu Beginn leer. Sie enthält die gleichen Attribute wie R (nämlich a, b und t) und ein zusätzliches Attribut x. Mit den ersten zwei tgds (die für alle aufgeführten Aggregate gleich sind) wird $\tilde{R}$ mit Tupeln gefüllt. Dabei wird jedes Tupel aus R auf genau ein Tupel in $\tilde{R}$ abgebildet. Die Attributwerte bleiben gleich, nur der Wert für x wird ergänzt. Dabei ist x (wie t) auch Schlüssel für $\tilde{R}$. Zusätzlich gilt aber die Eigenschaft, dass die Werte von 1 beginnend durchgängig vergeben werden (bei n Tupeln also $1, 2, \ldots, n$). So erhält man eine Ordnung auf der Tupelmenge, womit sich später leichter sicherstellen lässt, dass jedes Tupel genau einmal in das Ergebnis eingeht.
Weiter sind die Relationen $H(c,x)$, $S(t)$, $S'(t)$ und $S''(t)$ intern angelegte Hilfstabellen (später auch als *Sidetables* bezeichnet). Auch sie sind zu Beginn leer. H speichert die Zwischenergebnisse der Aggregation für jeden Schritt. Für x werden nacheinander die Tupel-IDs aus $\tilde{R}$ eingetragen während c das jeweilige Zwischenergebnis darstellt. In den drei restlichen Tabellen S, S' und S'' wird vermerkt, welche Tupel in verschiedenen Arbeitsschritten bereits betrachtet wurden. Hier wird immer der Schlüssel t verwendet. S und S' werden beim Füllen der Relation $\tilde{R}$ genutzt. Bei der Berechnung des Aggregats werden danach in S'' alle Tupel gespeichert, die bereits in die Berechnung eingegangen sind.

Funktionsweise der Regeln

Der Mechanismus hinter den Regelmengen soll nun kurz erklärt werden. Für alle behandelten Aggregatfunktionen sind die ersten zwei tgds identisch. Mit ihnen wird $\tilde{R}$ gefüllt, um eine Reihenfolge auf den Tupeln zu erzeugen, damit sie danach mit den restlichen Regeln in dieser Reihenfolge bearbeitet werden können. Für `MIN`, `MAX`, `SUM` und `AVG` wird das Ergebnis der Aggregation mit zwei weiteren tgds berechnet und zum Schluss mit einer s-t tgd auf die Result-Relation abgebildet. Für `COUNT` sind die zwei „mittleren“ tgds nicht nötig. Hier kann das Ergebnis direkt mittels der generierten IDs in einer s-t tgd ausgelesen werden.
Die erste tgd wird genau einmal ausgeführt. Das zweite Atom $\neg\tilde{R}(x_2, a_2, b_2, t_2)$ stellt dabei sicher, dass noch kein Tupel in $\tilde{R}$ vorhanden ist. Das ist nur am Anfang der Fall, da im Kopf ein neues Tupel in $\tilde{R}$ mit $x = 1$ erzeugt wird. Die ID des Tupels wird auch in S' eingetragen. Für jedes verbleibende Tupel in R wird dann die zweite tgd angewandt. Die Idee hinter den

Hilfstabellen ist hierbei, dass in S' alle Tupel vermerkt werden, die bereits in $\tilde{R}$ eingetragen wurden, während S dazu dient, zu erkennen, welches Tupel zuletzt in $\tilde{R}$ eingetragen wurde. Damit kann der nächste Wert für x ermittelt werden. S enthält dabei die gleichen Tupel wie S', allerdings immer genau eines weniger (außer am Anfang gilt also $|S' - S| = 1$). Das eine Tupel in der Differenz vermerkt jenes Tupel mit dem höchsten Wert für x. Der nächste Eintrag in $\tilde{R}$ erhält dementsprechend die ID $x+1$. Sobald alle Tupel nach $\tilde{R}$ übertragen wurden, wird die eigentliche Aggregation berechnet.

Wir wollen die Regeln zur Berechnung exemplarisch an der Summe erklären. Für die anderen Funktionen ist die Arbeitsweise ähnlich. Die Tupel aus $\tilde{R}$ werden iterativ bearbeitet, beginnend bei dem Tupel mit $x = 1$ bis zum letzten (es sei $x = n$). An dieser Stelle ist die Voraussetzung der durchgängigen IDs wichtig, da der aktuelle Wert für x immer inkrementiert wird.
Die dritte Regel bei der Summe wird einmal ausgeführt, da der Body $\tilde{R}(1, a_1, b_1, t_1)$ nur auf das Tupel mit $x = 1$ passt. In H wird dann ein neues Tupel erstellt: für x wird der Wert 1 eingetragen, da das erste Tupel in $\tilde{R}$ bearbeitet wurde; der aggregierte Wert ist bisher nur der Attributwert für b. Auch in S'' wird der Schlüssel t_1 eingetragen, um zu speichern, dass dieses Tupel bereits bearbeitet wurde.
Nun wird die vorletzte Regel $n-1$ mal ausgeführt ($x = 2, \ldots, n$). Das Atom $\neg S''(t_1)$ stellt dabei sicher, dass das Tupel aus $\tilde{R}$ noch nicht bearbeitet wurde (sonst würde bereits ein Eintrag in S'' existieren). $H(c_1, x_1 - 1)$ sagt aus, dass das Tupel aus $\tilde{R}$ auch tatsächlich das als nächstes zu betrachtende Tupel ist. Der letzte Eintrag in H war mit $x_1 - 1$, dann ist der nächste mit x_1. Der Wert für b wird auf das letzte Zwischenergebnis c_1 addiert. Für jedes Tupel (c, i) in H (mit $i = 1, \ldots, n$) gilt dabei: c entspricht der Summe aller b-Werte von $x = 1$ bis $x = i$ aus $\tilde{R}$.
Sobald alle Tupel bearbeitet wurden (die zweite Regel kann nicht mehr angewandt werden), wird die letzte Regel (eine s-t tgd) genau einmal ausgeführt. Sie dient dazu, das Ergebnis auch in die *Result*-Tabelle einzutragen. Das Atom $\neg H(c_2, x_1 + 1)$ stellt hier sicher, dass auch tatsächlich der letzte Eintrag aus H dafür genutzt wird (es existiert kein Tupel mit größerem Wert für x_1).

Beispiel 9. Hier soll die Generierung der durchgängigen IDs mittels der ersten beiden tgds an einem Beispiel gezeigt werden. Dabei wollen wir die tgds für die Anfrage aus Beispiel 3 anwenden:

```
SELECT AVG(age)
FROM Students
```

Im folgenden Beispiel 10 wird dann die Aggregation berechnet. Aus der Anfrage können wir folgende Entsprechungen mit den Regeln aus Tabelle 3.1 schließen: R entspricht der Relation *Students*, da an diese die Anfrage gestellt wird; t entspricht dem Attribut *student_id*, da es Schlüssel ist; b entspricht dem Attribut *age*, da es Argument für die Aggregation ist; die restlichen Attribute entsprechen a. Die neu erzeugte Relation (um die IDs ergänzt) notieren wir hier mit $\widetilde{St}$ (entspricht $\tilde{R}$). Zuerst wird die erste tgd einmal für ein beliebiges Tupel ausgeführt:

$$St(1, \text{Miller}, \text{Mark}, \text{Electrical Engineering}, 18) \wedge \neg\widetilde{St}(x_2, sid_2, sn_2, fn_2, co_2, a_2)$$
$$\rightarrow S'(1) \wedge \widetilde{St}(1, 1, \text{Miller}, \text{Mark}, \text{Electrical Engineering}, 18)$$

Im Anschluss wird die zweite tgd für die restlichen Tupel in *Students* ausgeführt, für unsere Instanz also 5 mal. Hier zeigen wir die ersten zwei Anwendungen davon, die restlichen geschehen analog:

$$St(2, \text{Lewis}, \text{Linda}, \text{Computer Science}, 20) \wedge \neg S'(2) \wedge$$
$$\widetilde{St}(1, 1, \text{Miller}, \text{Mark}, \text{Electrical Engineering}, 18) \wedge \neg S(1)$$
$$\rightarrow S'(2) \wedge \widetilde{St}(1+1, 2, \text{Lewis}, \text{Linda}, \text{Computer Science}, 20) \wedge S(1)$$

$$St(3, \text{Smith}, \text{Steven}, \text{Mathematics}, 21) \wedge \neg S'(3) \wedge$$
$$\widetilde{St}(2, 2, \text{Lewis}, \text{Linda}, \text{Computer Science}, 20) \wedge \neg S(2)$$
$$\rightarrow S'(3) \wedge \widetilde{St}(2+1, 3, \text{Smith}, \text{Steven}, \text{Mathematics}, 21) \wedge S(2)$$

Nachdem alle Tupel verarbeitet wurden und keine der beiden tgds mehr angewandt werden kann, haben die erzeugten Relationen folgende Einträge:

$\widetilde{St}$:

x	sid	surname	...	age
1	1	Miller	...	18
2	2	Lewis	...	20
3	3	Smith	...	21
4	4	Smith	...	23
5	5	Brown	...	22
6	7	Nelson	...	25

S':

sid
1
2
3
4
5
7

S:

sid
1
2
3
4
5

Am letzten Tupel aus $\widetilde{St}$ kann man gut den Unterschied zwischen dem bereits vorhandenen Schlüssel *student_id* und dem neu generierten Schlüssel x erkennen: bei *student_id* gibt es eine Lücke zwischen den Werten (keine 6), bei x gibt es diese nicht. Solche Lücken können z.B. entstehen, wenn Tupel gelöscht wurden. Auch der „Versatz“ zwischen S' und S zur Bestimmung des letzten Tupels ist an den Instanzen gut erkennbar. ■

Beispiel 10. Nachdem im vorigen Beispiel die IDs generiert wurden, soll nun die Berechnung des Durchschnitts für die Anfrage aus Beispiel 3 durchgeführt werden. Wir gehen davon aus, dass die IDs in der Relation $\widetilde{St}$ bereits vollständig erzeugt wurden, wie es eben in Beispiel 9 zu sehen

war. Dies geschah durch die ersten zwei Regeln aus Tabelle 3.1. Die dritte Regel wird danach zur Initialisierung einmal auf dem ersten Tupel aus $\widetilde{St}$ ausgeführt:

$$\widetilde{St}(1,1,\text{Miller},\text{Mark},\text{Electrical Engineering},18) \quad \rightarrow \quad S''(1) \wedge H(18,1)$$

Dann wird die zweite tgd auf allen verbleibenden Tupel aus $\widetilde{St}$ jeweils einmal ausgeführt. Hier wieder die ersten zwei Anwendungen:

$$\begin{aligned} \widetilde{St}(2,2,\text{Lewis},\text{Linda},\text{Computer Science},20) \wedge \neg S''(2) \wedge H(18,1) \quad &\rightarrow \quad S''(2) \wedge H(18+20,2) \\ \widetilde{St}(3,3,\text{Smith},\text{Steven},\text{Mathematics},21) \wedge \neg S''(3) \wedge H(38,2) \quad &\rightarrow \quad S''(3) \wedge H(38+21,3) \end{aligned}$$

Nachdem die beiden tgds für alle Tupel angewandt wurden, sind H und S'' mit folgenden Tupeln gefüllt:

H:

c	x
18	1
38	2
59	3
82	4
104	5
129	6

S'':

sid
1
2
3
4
5
6

Abschließend wird die s-t tgd (die letzte Regel) ausgeführt:

$$H(129,6) \wedge \neg H(c_2, 6+1) \quad \rightarrow \quad Res(129/6)$$

Dabei wird die eben berechnete Summe durch die Tupelanzahl geteilt. So erhält man das arithmetische Mittel über alle Werte für das Alter. Das Ergebnis wird in der Result-Relation eingetragen. In diesem Fall lautet es $129/6 = 21.5$ und stimmt damit mit dem Ergebnis der SQL-Anfrage überein. ■

3.1.3 Sonstige Erweiterungen

Wir haben gerade betrachtet, wie ausgewählte grundlegende Operationen und auch Aggregationen als Abhängigkeiten dargestellt werden. SQL ist natürlich wesentlich umfangreicher, als die bisher behandelten Operationen (Projektion, Selektion, Verbund, Aggregation). Es gibt sowohl weitere Varianten (z.B. Selektionen mit Ungleichheitsatomen) als auch komplett andere Operationen (z.B. Vereinigung). Deshalb sollen nun einige Erweiterungen dazu beschrieben werden.

Selektionen mit Ungleichheitsatomen

Zuerst sollen weitere Selektionen betrachtet werden. Zu den Kernoperationen wurden im vorigen Abschnitt solche Selektionen gezählt, die nur Gleichheitsatome enthalten. Bei mehreren Gleichheitsatomen können diese nur durch ein logisches Und verbunden sein. Einige weitere Atome zu unterstützen, fällt in der Theorie recht einfach. Atome der Form $\langle Attribut \rangle\ \theta\ \langle Konstante \rangle$ oder $\langle Attribut \rangle\ \theta\ \langle Attribut \rangle$ mit $\theta \in \{<, \leq, \neq, \geq, >\}$ bezeichnen wir als Ungleichheitsatome. Nach [Aug] lassen sich solche Atome in s-t tgds umsetzen, indem (wie bereits bei den Gleichheitsatomen) ein entsprechendes Atom in der s-t tgd mit Und verknüpft hinzugefügt wird. Obwohl das nicht umständlicher ist als bei den Gleichheitsatomen, werden Selektionen mit Ungleichheitsatomen hier trotzdem unter den Erweiterungen aufgeführt, da nur einige Definitionen für s-t tgds solche Atome zulassen. Beispielsweise erlaubt [Ben+17] nur relationale Atome und [GMS12] nur relationale und Gleichheitsatome. In [Fag+11] dagegen werden beliebige Atome zugelassen. Um eben auch weitere s-t tgds zu gestatten, ist die Definition in dieser Arbeit (s. Def. 3) deshalb auch hauptsächlich an [Fag+11] angelehnt. Außerdem werden solche Formeln auch noch nicht im Tool ChaTEAU unterstützt, was für die Implementation wichtig wäre. Später in Abschnitt 4.5.2 wird aber ein Vorschlag gemacht, wie Vergleichsatome zumindest zum XML-Eingabeformat von ChaTEAU hinzugefügt werden können.

Selektionen mit Disjunktionen

Um weitere Einschränkungen bei den Selektionen aufzuheben, können neben den Atomen auch andere logische Operatoren unterstützt werden, also auch Disjunktionen (`OR`) statt nur Konjunktionen (`AND`). Die Hürde besteht nun darin, dass s-t tgds nur Konjunktionen von Atomen zulassen, aber keine Disjunktionen. In [Kav+21] wird vorgeschlagen, wie das Problem gelöst werden kann: statt einer s-t tgd können mehrere verwendet werden. Bei zwei Atomen im Selektionsprädikat, die durch ein `OR` verbunden sind, entspräche das zwei s-t tgds (in diesem Fall eine je Atom). Falls sowohl `AND` als auch `OR` in einem Selektionsprädikat vorkommen, könnte das Prädikat zuerst in die Disjunktive Normalform (DNF) umgeformt werden. Für jede `AND`-Klausel könnte dann eine neue Anfrage generiert werden, die nur noch diese `AND`-Klausel als Selektionsprädikat enthält. Jede der generierten Anfragen wiederum kann dann separat in Abhängigkeiten umgewandelt werden. Zuletzt müssen nur noch alle Abhängigkeiten zusammengefasst werden, dann hat man die Menge an Abhängigkeiten, welche die ursprüngliche Anfrage repräsentieren.

Beispiel 11. Für dieses Beispiel soll die Anfrage aus dem Beispiel 8 modifiziert werden, damit sie auch ein `OR` enthält: nun werden nicht nur Mathematik-Studierende gesucht, sondern auch Studierende der Informatik.

```
SELECT surname
FROM Students NATURAL JOIN Grades
WHERE (course = 'Mathematics' OR course = 'Computer Science')
    AND module_id = 3
```

Allerdings ist das Selektionsprädikat noch nicht in DNF. Die gleiche Anfrage mit dem Selektionsprädikat in DNF sähe so aus:

```
SELECT surname
FROM Students NATURAL JOIN Grades
WHERE (course = 'Mathematics' AND module_id = 3)
    OR (course = 'Computer Science' AND module_id = 3)
```

Nun wird je `AND`-Klausel eine neue Anfrage betrachtet. Hier also zwei (jeweils ohne `OR`):

```
SELECT surname
FROM Students NATURAL JOIN Grades
WHERE course = 'Mathematics'
    AND module_id = 3
```

```
SELECT surname
FROM Students NATURAL JOIN Grades
WHERE course = 'Computer Science'
    AND module_id = 3
```

Die beiden Anfragen werden dann separat in s-t tgds umgewandelt. In diesem Fall resultiert jede Anfrage nur in einer s-t tgd. Die Menge aus beiden s-t tgds stellt dann die ursprüngliche SQL-Anfrage dar. Für die Variablen und Atome wurden Abkürzungen benutzt, wie sie in Anhang D angegeben sind.

$$St(sid_1, sn_1, fn_1, \text{Mathematics}, a_1) \land Gr(3, sid_1, sem_1, gr_1) \land \rightarrow Res(sn_1)$$

$$St(sid_1, sn_1, fn_1, \text{Computer Science}, a_1) \land Gr(3, sid_1, sem_1, gr_1) \land \rightarrow Res(sn_1)$$

■

Mengenoperationen

Mit den Mengenoperationen lassen sich im Datenbankkontext zwei (Ergebnis-)Relationen auf bestimmte Art und Weise verknüpfen. In SQL heißen die drei Mengenoperationen `UNION` (Vereinigung), `INTERSECT` (Schnitt) und `EXCEPT` (Differenz). Sie entsprechen den drei Operationen der Relationenalgebra $\cup$, $\cap$ und $-$ (siehe auch Tabelle 2.5). In [Aug] wird vorgeschlagen, wie diese Operationen als s-t tgds umgesetzt werden können. Dabei ist zu bedenken, dass hier immer von den Varianten *mit* Duplikateliminierung ausgegangen wird (ähnlich wie schon bei der Projektion mit `DISTINCT`).

Zuerst wollen wir die Vereinigung betrachten. Sie hat ähnliche Eigenschaften, wie das `OR`, welches wir im vorigen Abschnitt untersucht haben. Auch für Vereinigungen sind mehrere Abhängigkeiten nötig, da Disjunktionen in Abhängigkeiten per Definition nicht zugelassen sind (vgl. Abschnitt 2.3.2). In [Kav+21] wird ein Beispiel gegeben, welches die Ähnlichkeit zwischen `OR` und `UNION` verdeutlicht. Um das Prinzip allgemeiner zu erklären, nehmen wir an, wir haben eine Anfrage mit Vereinigung der Form Q_1 `UNION` Q_2. Q_1 und Q_2 sind wieder Anfragen. Wichtig ist, dass sie „union compatible“ sind: im Ergebnis muss also die gleiche Anzahl an Attributen stehen,

jeweils mit denselben Datentypen [Pos21]. Diese Eigenschaft muss ebenso auch für die anderen Mengenoperationen gelten. Q_1 und Q_2 werden dann einzeln in Abhängigkeiten umgewandelt. Sei D_i die Menge der Abhängigkeiten, die Q_i entspricht (mit $i \in \{1,2\}$), dann entspricht $D_1 \cup D_2$ der Anfrage Q_1 `UNION` Q_2.

Der Schnitt wiederum hat Ähnlichkeiten mit dem inneren Verbund. Bei einer Anfrage Q_1 `INTERSECT` Q_2 müssen auch hier wieder Q_1 und Q_2 separat in Abhängigkeiten umgewandelt werden. Das Konzept aus [Aug] funktioniert allerdings bisher nur für den Fall, dass Q_1 und Q_2 jeweils nur in eine s-t tgd abgebildet werden (was in den meisten Fällen genügt). Die Bodies der beiden s-t tgds werden durch $\wedge$ in einen Body zusammengefügt und die Variablen, auf die projiziert wird (aus der `SELECT`-Klausel) werden gleichgesetzt. Genau diese Variablen werden dann auch im Head übernommen. Durch die Gleichsetzung wird sichergestellt, dass ein Tupel auch in beiden Ergebnisrelationen der Unteranfragen vorkommt, um ins Ergebnis des Schnitts aufgenommen zu werden (was genau der Semantik des Schnitts entspricht).

Die Differenz der Form Q_1 `EXCEPT` Q_2 lässt sich beinahe auf die gleiche Weise umsetzen, wie bereits das `INTERSECT`. Der entscheidende Unterschied ist folgender: beim Zusammenführen der Bodies müssen alle Atome negiert werden, die aus der s-t tgd von Q_2 kommen. Damit wird ausgesagt, dass nur Tupel des Ergebnisses von Q_1 übernommen werden, für die kein gleiches Tupel im Ergebnis von Q_2 existiert.

Spaltenumbenennung

SQL unterstützt auch Umbenennungen von Spalten (ähnlich wie die Operation β in der Relationenalgebra). Die Umbenennung erfolgt dabei in der `SELECT`-Klausel. Um z.B. dem Attribut „forename“ das Label „name“ zu geben, würde die `SELECT`-Klausel folgendermaßen aussehen: `SELECT forename AS name`. Das `AS` ist dabei in den meisten Fällen optional [Pos21]. Nach [Aug] lässt sich eine Umbenennung a `AS` b umsetzen, indem eine weitere Variable b eingeführt wird. Die Variable a besteht bereits und wird im Body benutzt. Im Kopf soll dann b an der Stelle benutzt werden, wo eigentlich a stehen müsste. Um dann die Verbindung zwischen a und b herzustellen, wird noch das Gleichheitsatom $a = b$ im Body hinzugefügt. Ein Problem mit diesem Vorgehen ist allerdings, dass der Variablenname in s-t tgd eigentlich nicht die Spalte im Ergebnis bezeichnet. In diesem Sinne lässt sich eine Spaltenumbenennung also nicht rein auf Ebene der s-t tgds darstellen; dafür müsste das Schema modifiziert werden. Es ist außerdem anzumerken, dass damit auch nur Umbenennungen von Spalten umgesetzt werden, noch keine Aliases für ganze Relationen, wie sie auch in SQL mit dem Keyword `AS` möglich sind (z.B. in der `FROM`-Klausel).

Geschachtelte Anfragen mit IN

Auch für geschachtelte Anfragen besteht ein Konzept in [Aug]; es werden geschachtelte Anfragen mit dem Keyword IN betrachtet. Vergleichbar zum Schnitt, der oben betrachtet wurde, besteht auch bei geschachtelten Anfragen Ähnlichkeit zum Verbund. Das Prinzip ist auch hier, das IN durch Gleichsetzung von Variablen darzustellen. Äußere und innere Anfrage werden zuerst separat in s-t tgds umgewandelt, dann wird der Body der s-t tgd für die innere Anfrage durch Konjunktion mit dem Body der äußeren verknüpft. Dabei werden für das Attribut, welches beim IN betrachtet wird, die beiden Variablen aus innerer und äußerer Anfrage gleichgesetzt.

Beispiel 12. Auch dieser Fall soll für das bessere Verständnis an einem Beispiel betrachtet werden. Die folgende geschachtelte Anfrage soll als s-t tgd dargestellt werden (vgl. Beispiel 8):

```
SELECT surname
FROM Students
WHERE course = 'Mathematics'
    AND student_id IN (
        SELECT student_id
        FROM Grades
        WHERE module_id = 3)
```

Zuerst betrachten wir die s-t tgds für äußere und innere Anfrage separat:

$$\begin{aligned} \text{innere:} \quad & \mathit{Students}(\mathit{sid}_1, \mathit{sn}_1, \mathit{fn}_1, \text{Mathematics}, a_1) \rightarrow \mathit{Res}(\mathit{sn}_1) \\ \text{äußere:} \quad & \mathit{Grades}(3, \mathit{sid}_2, \mathit{sem}_2, \mathit{gr}_2) \rightarrow \mathit{Res}(\mathit{sid}_2) \end{aligned}$$

Dann werden die beiden Bodies mit Und verknüpft. Als Head wird der Head der äußeren Anfrage übernommen. Außerdem müssen die Variablen sid_1 und sid_2 gleichgesetzt werden (hier wird sid_2 durch sid_1 ersetzt). Die vollständige s-t tgd sieht dann folgendermaßen aus:

$$\mathit{Students}(\mathit{sid}_1, \mathit{sn}_1, \mathit{fn}_1, \text{Mathematics}, a_1) \wedge \mathit{Grades}(3, \mathit{sid}_1, \mathit{sem}_2, \mathit{gr}_2) \rightarrow \mathit{Res}(\mathit{sid}_2) \qquad \blacksquare$$

Wie auch bereits beim Schnitt ist dieses Vorgehen bisher nur für Anfragen konzipiert, die für äußere und innere Anfrage je nur einer s-t tgd entsprechen. Außerdem sind nicht alle geschachtelten Anfragen umgesetzt: weitere Operatoren wie EXISTS oder NOT IN sind damit noch nicht abgebildet.

3.1.4 Noch nicht abgedeckte Operationen

Wir haben eben gesehen, dass bereits einige Erweiterungen bestehen, um weitere Operationen aus SQL auch als Abhängigkeiten darstellen zu können (z.B. weitere Selektionen, Mengenopera-

tionen oder Aggregationen). Allerdings ist SQL eine sehr umfangreiche Sprache, deshalb sollen nun einige Operationen aufgelistet werden, für die nach meiner Kenntnis bisher keine Konzepte für die Umwandlung ausgearbeitet wurden.

Weitere geschachtelte Anfragen Unter den Erweiterungen wurde auch geklärt, wie geschachtelte Anfragen mit dem Keyword `IN` umgesetzt werden können. SQL unterstützt allerdings noch weitere Arten von geschachtelten Anfragen, nämlich auch mit den Keywords `EXISTS`, `NOT IN`, `ANY` und `ALL` (die letzten beiden in Verbindung mit einem Vergleichsoperator wie `<=`). Außerdem müssen die Keywords auch nicht zwangsweise mit einer Unteranfrage formuliert werden, sondern es kann auch ein Array mit festen Werten angegeben werden.

Äußere Verbunde In Abschnitt 3.1.1 wurde beschrieben, wie innere Verbunde mit gleichen Attributwerten über Gleichsetzung von Variablen dargestellt werden. Die Umsetzung von äußeren Verbunden ist damit noch nicht geklärt, ebenso wie Verbunde mit beliebigen Verbundbedingungen (z.B. der Art `R JOIN S ON (R.a < S.b)`).

Selektionen mit Negation Es wurden bisher vielfältige Selektionsbedingungen beschrieben: verschiedene Vergleichsoperatoren (z.B. $=, \neq, \leq$) und verschiedene Konnektoren ($\wedge, \vee$), aber noch keine Negationen (`NOT` in SQL).

Gruppierung In SQL lassen sich mit `GROUP BY` Tupel nach Attributwerten gruppieren. Mit `HAVING` können diese dann selektiert werden. Sie werden häufig in Verbindung mit Aggregationen verwendet. Auch hier erfordert eine mengenbasierte Umsetzung vermutlich weitere Tupel-IDs.

Aliases für Tabellen Als Erweiterung wurden auch Spaltenumbenennungen aufgeführt. Neben der Umbenennung von Attributen lassen sich mit gleicher Syntax auch Aliases für Relationen vergeben. Das wird u.a. genutzt um eine Query lesbarer zu machen, für Self-Joins oder um Zwischenergebnisse referenzieren zu können (z.B. bei Joins oder Subqueries).

Weitere Aggregationen Als letzte Erweiterung wurden die Aggregationen `MIN`, `MAX`, `COUNT`, `SUM` und `AVG` beschrieben. In SQL werden aber noch weitere Aggregatfunktionen unterstützt, z.B. `BOOL_AND` und `BOOL_OR` auf booleschen Werten oder auch statistische Funktionen wie `VARIANCE`, `STDDEV` oder `CORR`.

Skalare Ausdrücke In SQL sind in der `SELECT`-Klausel auch sogenannte skalare Ausdrücke zugelassen. Das sind Funktionen, die tupelweise auf ein Attribut angewandt werden. Bspw. könnte in einer Relation eine Temperatur in Grad Celsius gespeichert sein (`temp`). Wenn sie aber in Kelvin ausgegeben werden soll, kann die Umrechnung in der `SELECT`-Klausel einfach so aussehen: `SELECT temp + 273.15`. Genauso können skalare Ausdrücke auch auf mehrere Attribute angewandt werden. Ein Gesamtpreis je Produkt bei gespeicherter Stückzahl (`quantity`) und Stückpreis (`price`) ließe sich so umsetzen: `SELECT quantity * price`.

Nun haben wir gesehen, welche Operationen aus SQL inwieweit in der Theorie als Abhängigkeiten darstellbar sind. Es wurden auch einige Operationen genannt, zu denen bisher keine uns

bekannte Theorie besteht. Einige davon werden später in den Abschnitten 4.4 und 4.6 aufgegriffen und es werden neue Konzepte zur Umsetzung vorgestellt. Als nächstes wollen wir uns nun aber anschauen, wie die bestehende Theorie bisher praktisch umgesetzt wurde.

3.2 Existierende Implementierungen

Im vorigen Abschnitt wurde erklärt, welches der Stand der Forschung ist, was die Darstellung von SQL-Anfragen als Abhängigkeiten betrifft. Jetzt wollen wir sehen, welche Teile der vorgestellten Theorie bisher implementiert wurden. Dabei werden wir vor allem das Projekt sql2sttgd betrachten, da es nach meiner Kenntnis das bisher umfangreichste Projekt für dieses Problem ist und da es Ziel dieser Arbeit ist, sql2sttgd zu erweitern. Bevor aber sql2sttgd betrachtet wird, sollen zwei weitere Projekte vorgestellt werden: ChaTEAU und ProSA. Für beide gilt zwar, dass sie keine Implementierung der eben genannten Konzepte sind, da sie jeweils andere Aufgaben haben. Sie sollen trotzdem hier aufgeführt werden, da sie nötig sind, um die Arbeitsweise von sql2sttgd vollständig verstehen zu können.

3.2.1 ChaTEAU

Das Tool ChaTEAU (**Cha**se for **T**ransforming, **E**volving, and **A**dapting databases and queries, **U**niversal approach) ist eine Implementation des CHASE-Algorithmus (s. Abschnitt 2.3.1). Im Gegensatz zu anderen Implementationen ist die Besonderheit an ChaTEAU, dass es nicht nur einen Anwendungsfall des CHASE unterstützt, sondern mehrere [AH19] (für eine Übersicht der Anwendungsfälle des CHASE siehe dort Tabelle 1). ChaTEAU erhält das CHASE-Objekt $\bigcirc$ und den CHASE-Parameter $\star$ in einer XML-File und gibt das Ergebnis $\circledast$ auch wieder als XML zurück. Über diese Schnittstelle wird ChaTEAU auch in ProSA aufgerufen.

3.2.2 ProSA

ProSA (**Pro**venance Management using **S**chema mappings with **A**nnotations) ist ein Tool zum Management von Forschungsdaten. In der Forschung können große Datenmengen anfallen. Um nicht alle Daten speichern zu müssen und auch aus Gründen des Datenschutzes wird versucht, nur notwendige Daten dauerhaft zu speichern. Gleichzeitig sollte Forschung aber auch reproduzierbar sein. Dafür sollten so viele Daten wie möglich behalten werden, damit Forschungsergebnisse gut von anderen nachvollzogen und reproduziert werden können. ProSA versucht diesen Konflikt zu lösen, indem vom großen Originaldatensatz eine sog. minimale Teildatenbank bestimmt wird. Außerdem soll berücksichtigt werden, dass der Originaldatensatz nicht unverändert bleibt, sondern möglicherweise Schema- und Daten-Evolution unterliegt [AH19; Pro]. Zur Berechnung der minimalen Teildatenbank wird Data Provenance mit der Anwendung des CHASE kombiniert. Dafür wird zunächst das Anfrageergebnis berechnet und anschließend über eine inverse

Anfrage verarbeitet. Es entsteht eine Teil-Datenbank, mit deren Hilfe das Forschungsergebnis verifiziert werden kann. Für die Ausführung des CHASE auf Instanzen nutzt ProSA dabei das eben vorgestellte Tool ChaTEAU.

3.2.3 sql2sttgd

Die Software sql2sttgd ist ein Bestandteil des größeren Forschungsprojektes ProSA. Ein weiterer Bestandteil von ProSA ist auch das Tool ChaTEAU (s. Abschnitt 3.2.1), welches eine Implementation des CHASE-Algorithmus ist. sql2sttgd hat das Ziel, eine Lücke zu füllen, die „vor“ ChaTEAU entstanden war. Eine Eingabe für ProSA ist die SQL-Anfrage, die an den Originaldatensatz gestellt wurde, um das Forschungsergebnis zu erhalten. Um aber die minimale Teildatenbank zu berechnen, wird mit ChaTEAU der CHASE ausgeführt. ChaTEAU erhält dabei CHASE-Objekt und -Parameter in Form eines XML-Files und keine SQL-Anfrage. Mit sql2sttgd ist es nun möglich, diese XML-Eingabedatei automatisch zu generieren. Wie bereits oben diskutiert, muss dafür die SQL-Anfrage in Abhängigkeiten umgewandelt werden, mit denen der CHASE arbeiten kann. Das ist die Hauptaufgabe von sql2sttgd. – Aus diesem Grund, wird sql2sttgd auch als „Parser“ bezeichnet, da es eine SQL-Anfrage parst und anschließend umwandelt. – Damit das XML-File vollständig ist, werden auch Schema und Instanz der betrachteten Datenbank ausgelesen und in die Datei geschrieben (als CHASE-Objekt).

Architektur

Als nächstes soll die Architektur von sql2sttgd kurz beschrieben werden. Weitere Details können in [Kav+21] nachgelesen werden. In Abbildung 3.1 werden die Ein- und Ausgaben der einzelnen Module schematisch dargestellt. Das gesamte Programm erhält zwei Inputs: die SQL-Anfrage, die umgewandelt werden soll und Informationen für eine Datenbank-Anbindung. Im Folgenden sollen die Teilaufgaben der Module erklärt werden.

SQL-Parser Der SQL-Parser erhält als Eingabe die SQL-Anfrage in Form eines Strings und parst die Anfrage. Sie steht dann als Syntaxbaum im Rest des Programms zur Verfügung und kann so wesentlich einfacher verarbeitet werden. Für diese Aufgabe wird die Library JSQLParser[1] verwendet. Sie stellt den Syntaxbaum als Java-Klassenhierarchie dar. Beim Parsen wird die Anfrage auch syntaktisch geprüft. Außerdem werden auch alle Tabellennamen, die in der Anfrage vorkommen, ausgelesen und an den DB-Reader weitergegeben.

DB-Reader Die Aufgabe des DB-Readers ist es, alle notwendigen Informationen aus der gegebenen Datenbank auszulesen. Dafür erhält er eine Anbindung an eine Datenbank (PostgreSQL) und die relevanten Tabellennamen vom SQL-Parser-Modul. Für jede dieser Tabellen werden Schema und Instanz ausgelesen. Tabellen, die nicht für die Anfrage notwendig

[1] `https://github.com/JSQLParser/JSqlParser`

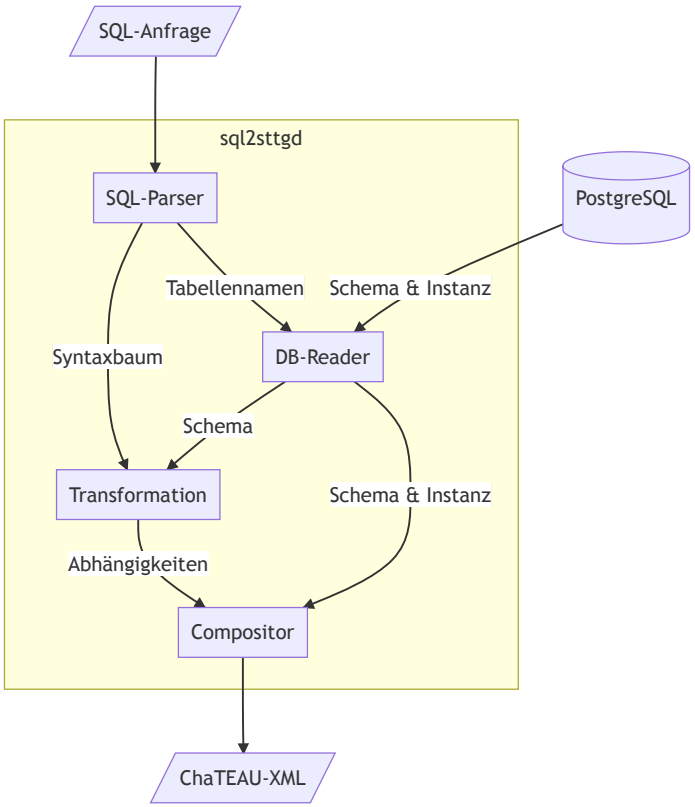

Abbildung 3.1: Flowchart für den Programmablauf von sql2sttgd

sind, werden nicht ausgelesen. Schema und Instanz werden dann in einer Darstellung zurückgegeben, die sich später leicht in die entsprechenden XML-Elemente umformen lassen (hauptsächlich JAXB-Objekte, siehe Compositor).

Transformation Das Transformationsmodul erledigt die Hauptaufgabe von sql2sttgd: die Umwandlung der SQL-Anfrage zu Abhängigkeiten. – Der Ablauf wird auch später noch in Abschnitt 4.2.1 als Pseudocode formuliert. – Wenn Vereinigungen in der Anfrage vorkommen, werden zuerst rekursiv die atomaren Anfragen bestimmt (atomar meint hier einfach einen Select-From-Where-Block). Jede atomare Anfrage wird dann separat transformiert (für die theoretischen Überlegungen dazu siehe auch den Absatz zu Vereinigungen im Abschnitt 3.1.3).

Für jede atomare Anfrage wird anfangs der Rumpf der s-t tgd initialisiert. Hierfür werden die Schema-Informationen vom DB-Reader genutzt. Für jede relevante Tabelle wird ein Atom im Rumpf mit den entsprechenden Variablen für die Spalten erstellt. Die Variablen sind zunächst eindeutig, um mögliche Konflikte zu vermeiden.

Im nächsten Schritt werden dann die Verbunde aus der `FROM`-Klausel in die s-t tgd eingearbeitet. Dafür werden Variablen gleichgesetzt, über die ein Verbund stattfinden soll. Allerdings haben unterschiedliche Arten von Verbunden verschiedene Arten, wie Verbundattribute bestimmt werden. Auch können sich unterschiedliche Schemata der Verbundergebnisse ergeben. Um beides korrekt zu ermitteln, wird ein graphenbasiertes Verfahren

eingesetzt, das gleichzeitig das Ergebnisschema bestimmt. Im Graphen werden Gleichheitsbeziehungen zwischen Variablen gespeichert, die sich aus den Verbunden ergeben. So lassen sich später Gruppen von Variablen ermitteln, die in der s-t tgd gleichgesetzt werden müssen. Für die Arbeit auf Graphen wird dabei auf eine Graphen-Implementation der Library Google Guava[2] zurückgegriffen.

Mit dem berechneten Verbund-Ergebnisschema aus dem vorigen Schritt wird dann der Kopf der s-t tgd erstellt. Falls keine Projektion stattfindet (die Anfrage beginnt also mit `SELECT *`), wird das gesamte Ergebnisschema für den Kopf übernommen. Andernfalls werden nur diejenigen Variablen verwendet, für die auch die entsprechenden Attribute in der `SELECT`-Klausel auftauchen. Damit wird die Projektion umgesetzt. Falls eine Vereinigung vorkommt, wird noch überprüft, ob die Schemata der vereinigten Relationen auch tatsächlich union compatible sind (d.h. sie stimmen in Anzahl und Datentyp der Attribute überein).

Zuletzt werden die Bedingungen aus der `WHERE`-Klausel eingearbeitet. Aktuell werden in sql2sttgd nur Selektionen nach Konstanten unterstützt. Dafür werden Gleichsetzungen von Attributen zu Konstanten aus der `WHERE`-Klausel eingelesen. Die entsprechenden Variablen in der s-t tgd werden dann durch diese Konstanten ersetzt. Im gesamten Verlauf der Transformation lagen die s-t tgds in einer internen Repräsentation vor, die effizientes Arbeiten mit ihnen ermöglicht. Abschließend wird diese Repräsentation nun auf JAXB-Objekte abgebildet, damit diese dann im Compositor leicht in XML serialisiert werden können.

Compositor Der Compositor schließt den Ablauf von sql2sttgd ab, indem er die Ergebnisse der vorigen Module zusammenfasst und als XML ausgibt. Für diese Aufgabe wird die Bibliothek JAXB (Java Architecture for XML Binding)[3] genutzt. Dafür wurde das Schema des XML-Outputs mittels einer XML Schema Definition (XSD) formal beschrieben. JAXB kann aus dieser XSD Java-Klassen generieren, die dem XML-Schema entsprechen (wir bezeichnen diese einfach als JAXB-Klassen bzw. -Objekte). Wenn dann Instanzen dieser generierten Klassen vorliegen, können sie mit JAXB als XML serialisiert werden. Der Compositor erhält also Schema und Instanz vom DB-Reader und die Abhängigkeiten vom Transformationsmodul in JAXB-Objekten. Diese werden dann in einem großen JAXB-Objekt zusammengefügt, als XML serialisiert und am Ende ausgegeben.

Umgesetzte Operationen

In Abschnitt 3.1 wurden Konzepte vorgestellt, wie sich einige Operationen aus SQL als Abhängigkeiten darstellen lassen. Wir wollen nun betrachten, welche dieser Operationen bisher in sql2sttgd umgesetzt werden konnten. Dafür soll hier eine knappe Übersicht gegeben werden – weitere Details sind wieder in [Kav+21] zu finden.

[2] `https://github.com/google/guava`
[3] `https://www.oracle.com/technical-resources/articles/javase/jaxb.html`

- Projektion
  - immer mit Duplikateliminierung
  - auf Attributliste (z.B. `SELECT surname, forename`)
  - ohne Projektion (also `SELECT *`)
- Verbund
  - Kreuzprodukt
  - Innerer Natürlicher Verbund
  - Innerer Verbund mit `ON`-Expression (nur Verbundbedingungen der Form $\langle Attribut \rangle = \langle Attribut \rangle$, verknüpft mit `AND`)
- Selektion
  - Bedingungen der Form $\langle Attribut \rangle = \langle Konstante \rangle$
  - Verknüpfungen mit `AND` möglich
- Mengenoperationen
  - Vereinigung

Es ist zu sehen, dass einige einfache Operationen bereits umgesetzt wurden: hauptsächlich jene, die wir in Abschnitt 3.1.1 als „Kernoperationen“ bezeichnet haben. Gleichzeitig wurde aber ein Großteil der Erweiterungen noch nicht umgesetzt – ausgenommen die Vereinigung. Im nächsten Kapitel sollen deshalb Ansätze vorgestellt werden, wie sich weitere Operationen in sql2sttgd umsetzen lassen. Dabei werden wir wieder einen besonderen Fokus auf Aggregatfunktionen richten.

Kapitel 4

Konzept

In diesem Kapitel

Im letzten Kapitel wurde der aktuelle Stand der Forschung zur Umwandlung von SQL-Anfragen zu Abhängigkeiten betrachtet. Es wurde gezeigt, welche Operationen aus SQL bis dato untersucht wurden und welche nicht. Außerdem wurde die bestehende Implementation dieser Konzepte im Tool sql2sttgd beleuchtet.
In diesem Kapitel sollen nun Beiträge dieser Arbeit zur existierenden Theorie beschrieben werden. Dafür werden zuerst Änderungen am bestehenden Konzept für Aggregationen vorgestellt. Danach wird der Algorithmus, der bisher bei der Transformation in sql2sttgd abläuft, grob formalisiert. Damit können wir dann leichter verstehen, an welchen Stellen Änderungen nötig sind, um auch Aggregationen zum Tool hinzuzufügen. Ein entsprechend angepasster Algorithmus wird in Abschnitt 4.2.2 vorgeschlagen. Danach werden neue Konzepte im Zusammenhang mit Aggregationen erklärt: wir untersuchen, wie Anfragen behandelt werden müssen, wenn eine Aggregation nicht nur isoliert vorkommt, sondern die Anfrage noch andere Operationen enthält. Dafür beschreiben wir sowohl ein Verfahren für Verbunde und Selektionen (Abschnitt 4.3) als auch für die Gruppierung (Abschnitt 4.4). Neben sql2sttgd, sind auch Änderungen am Tool ChaTEAU notwendig. In Abschnitt 4.5 werden dafür Vorschläge gemacht, wie das XML-Eingabeformat von

ChaTEAU erweitert werden kann, um auch Aggregationen zu unterstützen. Zum Schluss werden in Abschnitt 4.6 neue theoretische Konzepte erklärt, v.a. für äußere Verbunde und geschachtelte Anfragen. Nach unserer Kenntnis wurden diese Operationen für die betrachteten Fälle bisher nicht untersucht. Damit wird gezeigt, dass auch diese Fälle als Abhängigkeiten umsetzbar sind. Es wird auch auf eine Gruppe von Tabellen verwiesen, die übersichtlich darstellen sollen, wie verschiedene SQL-Operationen als Abhängigkeiten dargestellt werden können.

4.1 Anpassungen zu Aggregationen

Zuerst betrachten wir die Anpassungen am Konzept für Aggregationen, da diese das Hauptaugenmerk der vorliegenden Arbeit bilden. In [Aug] wird ein Konzept vorgeschlagen, wie einige der Aggregationen aus SQL als Mengen von (s-t) tgds dargestellt werden können. Dieses Konzept wurde bereits im Abschnitt 3.1.2 beschrieben (s. vor allem Tabelle 3.1). Die vorhandenen Regeln wurden im Rahmen dieser Arbeit an einigen Stellen angepasst und vereinfacht. Die neuen Regeln sind in Tabelle 4.1 zu sehen. Die Änderungen sollen hier vorgestellt werden.

4.1.1 Generierung durchgängiger IDs

Jede der fünf Untertabellen beschreibt die Regeln für eine der Aggregatfunktionen. Für die Funktionen `SUM`, `COUNT` und `AVG` werden dabei anfangs auch jeweils zwei Regeln aufgeführt, um durchgängige IDs zu erstellen. Sie entsprechen den zwei tgds, die auch bei [Aug] jeweils am Anfang stehen. Mit diesen zwei tgds lässt sich aus einer Eingaberelation $R(a,b,t)$ eine neue Relation $\tilde{R}(x,a,b,t)$ erzeugen (in [Aug] wird die neue Relation R_{sorted} genannt). Das Attribut t sei hierbei Schlüssel in R; a und b seien weitere Attribute. In den Regeln wird die Aggregation o.B.d.A. immer auf b angewandt (ausgenommen `COUNT(*)`). Die zwei tgds bilden dann jedes Tupel aus R auf ein Tupel in $\tilde{R}$ ab. Die neue Relation $\tilde{R}$ enthält dabei alle Attribute von R und zusätzlich ein weiteres Attribut x, welches auch Schlüssel ist. Allerdings gilt für x die zusätzliche Eigenschaft, dass die Werte von 1 beginnend durchgängig vergeben werden. Bei n Tupeln in R hat $\tilde{R}$ dann also ebenso n Tupel und x nimmt genau die Werte $1,2,\ldots,n$ an. Da der CHASE reihenfolgeunabhängig arbeitet, wird so eine Reihenfolge konstruiert, damit garantiert werden kann, dass jedes Tupel genau einmal ins Ergebnis eingeht und damit die richtigen Zwischenergebnisse verwendet werden.

Um die IDs zu generieren, wird jedes Tupel aus R genau einmal auf ein Tupel in $\tilde{R}$ abgebildet. Dabei wird die ID immer inkrementiert. Die zwei tgds wurden angepasst, sodass keine weiteren Sidetables S und S' notwendig sind, wie es in [Aug] der Fall ist. Das erleichtert die Implementierung, da so neben $\tilde{R}$ keine weiteren Tabellen dem Schema hinzugefügt werden müssen und da die Anzahl der Atome reduziert wurde. Die erste Regel wird initial genau einmal ausgeführt. Das erste Atom $R(a_1,b_1,t_1)$ passt zu jedem Tupel in R, es wird also ein beliebiges Tupel ausgewählt. Das zweite Atom $\neg\tilde{R}(x_2,a_2,b_2,t_2)$ beschreibt, dass noch keine Tupel in $\tilde{R}$ vorhanden sind. Da

$$R(a_1,b_1,t_1) \wedge \neg R(a_2,b_2,t_2) \wedge b_1 < b_2 \quad \rightarrow \quad Res(b_1)$$

(a) `MAX(b)`

$$R(a_1,b_1,t_1) \wedge \neg R(a_2,b_2,t_2) \wedge b_1 > b_2 \quad \rightarrow \quad Res(b_1)$$

(b) `MIN(b)`

$$\begin{aligned}
R(a_1,b_1,t_1) \wedge \neg \tilde{R}(x_2,a_2,b_2,t_2) &\rightarrow \tilde{R}(1,a_1,b_1,t_1) \\
R(a_1,b_1,t_1) \wedge \neg \tilde{R}(x_1,a_1,b_1,t_1) \wedge & \\
\tilde{R}(x_2,a_2,b_2,t_2) \wedge \neg \tilde{R}(x_2+1,a_3,b_3,t_3) &\rightarrow \tilde{R}(x_2+1,a_1,b_1,t_1) \\
\tilde{R}(1,a_1,b_1,t_1) &\rightarrow H(b_1,1) \\
\tilde{R}(x_1,a_1,b_1,t_1) \wedge \neg H(c_1,x_1) \wedge H(c_2,x_1-1) &\rightarrow H(b_1+c_2,x_1) \\
H(c_1,x_1) \wedge \neg H(c_2,x_1+1) &\rightarrow Res(c_1)
\end{aligned}$$

(c) `SUM(b)`

$$\begin{aligned}
R(a_1,b_1,t_1) \wedge \neg \tilde{R}(x_2,a_2,b_2,t_2) &\rightarrow \tilde{R}(1,a_1,b_1,t_1) \\
R(a_1,b_1,t_1) \wedge \neg \tilde{R}(x_1,a_1,b_1,t_1) \wedge & \\
\tilde{R}(x_2,a_2,b_2,t_2) \wedge \neg \tilde{R}(x_2+1,a_3,b_3,t_3) &\rightarrow \tilde{R}(x_2+1,a_1,b_1,t_1) \\
\tilde{R}(1,a_1,b_1,t_1) &\rightarrow H(b_1,1) \\
\tilde{R}(x_1,a_1,b_1,t_1) \wedge \neg H(c_1,x_1) \wedge H(c_2,x_1-1) &\rightarrow H(b_1+c_2,x_1) \\
H(c_1,x_1) \wedge \neg H(c_2,x_1+1) &\rightarrow Res(c_1/x_1)
\end{aligned}$$

(d) `AVG(b)`

$$\begin{aligned}
R(a_1,b_1,t_1) \wedge \neg \tilde{R}(x_2,a_2,b_2,t_2) &\rightarrow \tilde{R}(1,a_1,b_1,t_1) \\
R(a_1,b_1,t_1) \wedge \neg \tilde{R}(x_1,a_1,b_1,t_1) \wedge & \\
\tilde{R}(x_2,a_2,b_2,t_2) \wedge \neg \tilde{R}(x_2+1,a_3,b_3,t_3) &\rightarrow \tilde{R}(x_2+1,a_1,b_1,t_1) \\
\tilde{R}(x_1,a_1,b_1,t_1) \wedge \neg \tilde{R}(x_1+1,a_2,b_2,t_2) &\rightarrow Res(x_1)
\end{aligned}$$

(e) `COUNT(*)`

Tabelle 4.1: Aggregatfunktionen als Menge von (s-t) tgds, überarbeitet

im Kopf dann mit $\tilde{R}(1, a_1, b_1, t_1)$ ein neues Tupel in $\tilde{R}$ erzeugt wird, bleibt der Body nach einmaliger Anwendung dieser Regel falsch. Deshalb wird die erste tgd danach nicht mehr ausgeführt. Die zweite tgd wird dann für alle verbleibenden Tupel angewandt. Das erste Atom wählt, wie schon bei der ersten tgd, ein vorerst beliebiges Tupel aus R. Das zweite Atom $\neg\tilde{R}(x_1, a_1, b_1, t_1)$ konkretisiert, dass für dieses Tupel noch kein entsprechender Eintrag in $\tilde{R}$ existieren darf. Es wird also ein Tupel gewählt, das bisher noch nicht abgebildet wurde. Die beiden verbleibenden Atome im Body sagen aus, dass das „letzte“ Tupel aus $\tilde{R}$ gewählt wird. Das ist jenes Tupel, das zuletzt durch eine Regel in $\tilde{R}$ eingefügt wurde und damit auch die höchste ID hat. Im Body wird das ausgedrückt, indem zur ID x_2 bisher kein anderes Tupel mit einer größeren ID $x_2 + 1$ existiert. Deshalb wird an das Tupel, welches im Kopf neu erzeugt wird, die nächstgrößere ID $x_2 + 1$ vergeben. Sobald alle Tupel aus R abgebildet wurden, matcht auch die zweite tgd nicht mehr und $\tilde{R}$ ist vollständig.
Die Generierung der IDs wollen wir als Nächstes an einem Beispiel erklären.

Beispiel 13. Ähnlich wie bereits in Beispiel 9 aus dem vorigen Kapitel, soll nun die Generierung der durchgängigen IDs mittels der neuen Regeln an einem Beispiel gezeigt werden. Wir nutzen wieder dieselbe Anfrage, also:

```
SELECT AVG(age)
FROM Students
```

Die Entsprechungen zwischen den Regeln aus Tabelle 4.1 und den Relationen und Attributen haben sich nicht verändert. Das bedeutet R entspricht *Students* (abgekürzt *St*); t entspricht *student_id*; b (das Argument der Aggregation) ist *age* und die restlichen Attribute werden a zugeordnet. Die neu erzeugte Relation wird wieder mit $\widetilde{St}$ notiert. Und auch für diese Regeln gilt wieder, dass die erste tgd einmal mit einem beliebigen Tupel aus *Students* ausgeführt wird, z.B.:

$$St(1, \text{Miller}, \text{Mark}, \text{Electrical Engineering}, 18) \wedge \neg\widetilde{St}(x_2, sid_2, sn_2, fn_2, co_2, a_2)$$
$$\rightarrow \widetilde{St}(1, 1, \text{Miller}, \text{Mark}, \text{Electrical Engineering}, 18)$$

Danach wird die zweite tgd jeweils einmal für jedes der verbleibenden Tupel angewandt. Auch hier wieder die ersten zwei Belegungen dafür, um das Prinzip nachvollziehen zu können:

$St(2, \text{Lewis}, \text{Linda}, \text{Computer Science}, 20) \wedge \neg\widetilde{St}(x_1, 2, \text{Lewis}, \text{Linda}, \text{Computer Science}, 20) \wedge$
$\widetilde{St}(1, 1, \text{Miller}, \text{Mark}, \text{Electrical Engineering}, 18) \wedge \neg\widetilde{St}(1+1, sid_2, sn_2, fn_2, co_2, a_2)$
$\rightarrow \widetilde{St}(1+1, 2, \text{Lewis}, \text{Linda}, \text{Computer Science}, 20)$

$St(3, \text{Smith}, \text{Steven}, \text{Mathematics}, 21) \wedge \neg\widetilde{St}(x_1, 3, \text{Smith}, \text{Steven}, \text{Mathematics}, 21) \wedge$
$\widetilde{St}(2, 2, \text{Lewis}, \text{Linda}, \text{Computer Science}, 20) \wedge \neg\widetilde{St}(2+1, sid_2, sn_2, fn_2, co_2, a_2)$
$\rightarrow \widetilde{St}(2+1, 3, \text{Smith}, \text{Steven}, \text{Mathematics}, 21)$

Nach vollständiger Ausführung erhalten wir für $\widetilde{St}$ wie geplant dasselbe Ergebnis wie zuvor in Beispiel 9:

x	sid	surname	...	age
1	1	Miller	...	18
2	2	Lewis	...	20
3	3	Smith	...	21
4	4	Smith	...	23
5	5	Brown	...	22
6	7	Nelson	...	25

Die beiden anderen Sidetables S und S' sind bei den angepassten Regeln dafür nicht mehr nötig. ■

4.1.2 Minimum und Maximum

Auch die Regeln für Minimum und Maximum konnten vereinfacht werden. In [Aug] wurde hier bisher ein iterativer Ansatz zur Berechnung verfolgt, ähnlich wie auch bei den restlichen Aggregatfunktionen. Diese beiden Fälle lassen sich allerdings auch wesentlich knapper logisch beschreiben. Die neuen Regeln sind in den Untertabellen 4.1a und 4.1b zu sehen. Wenn das Maximum für ein Attribut b gesucht wird, dann ist jener Wert b_1 maximal, zu dem kein anderer Wert b_2 existiert, sodass b_2 größer wäre als b_1 (also $b_1 < b_2$). Für das Minimum darf entsprechend kein Wert existieren, der kleiner ist.

Mit dieser Methode sind deutlich weniger Regeln nötig: das gesamte Verhalten lässt sich in einer s-t tgd beschreiben. Außerdem funktioniert dieses Verfahren auch unabhängig von den IDs. Die Voraussetzung der durchgängigen IDs, die bei den restlichen Aggregaten gefordert wird, ist hier nicht notwendig. Möglicherweise ist der in [Aug] beschriebe Ansatz allerdings effizienter bei der

eigentlichen Berechnung, d.h. wenn der CHASE ausgeführt wird. Bei kleinen Tupelmengen, wie sie in ProSA betrachtet werden, dürfte der Unterschied aber vermutlich nicht signifikant sein.

4.1.3 Reduzierung der Sidetables

Auch für die Aggregatfunktionen `SUM` und `AVG` wurden Anpassungen im Vergleich zu dem Konzept aus [Aug] vorgenommen. Hier wurde die Anzahl der Sidetables verringert, ähnlich wie schon bei den Regeln zur ID-Generierung. In [Aug] wurden bei der Berechnung dieser Aggregate zwei Sidetables genutzt: S'' um zu merken, welche Tupel bereits bearbeitet wurden und H um Zwischenergebnisse zu speichern. In S'' wurde die bestehende Tupel-ID t gespeichert, in H der akkumulierte Wert b und die (generierte) ID x des zuletzt bearbeiteten Tupels. Allerdings werden in H schon alle Informationen gespeichert, die nötig sind, um zu wissen, welche Tupel betrachtet wurden. Deshalb kann S'' auch weggelassen werden und es wird stattdessen nur noch H genutzt. Bei der dritten und vierten Regel konnten die Atome zu S'' im Kopf vollständig weggelassen werden, da hier schon ein Tupel in H erzeugt wird. Das Atom $\neg S''(t)$ im Body der vierten Regel wurde durch ein entsprechendes Atom $\neg H(c_1, x)$ ersetzt. Es sagt nun aus, dass noch kein Tupel mit der ID x bearbeitet wurde.
Durch die Reduzierung der Sidetables kommt es auch zu Unterschieden bei der Berechnung des Aggregats. In Beispiel 13 hatten wir bereits gesehen, wie die Generierung der IDs mittels der neuen Regeln erfolgt. Jetzt wollen wir uns an einem weiteren Beispiel anschauen, wie sich auch die Berechnung des Aggregats verändert hat.

Beispiel 14. Nun wollen wir die eigentlichen Aggregationen berechnen; auch hier wieder das Durchschnittsalter der Studierenden wie im Beispiel 10. Wir gehen davon aus, dass die Tabelle $\widetilde{St}$ besteht und mit durchgängigen IDs gefüllt wurde, wie im Beispiel 13 zu sehen. Die Anwendung der dritten tgd wäre dann folgende:

$$\widetilde{St}(1, 1, \text{Miller}, \text{Mark}, \text{Electrical Engineering}, 18) \quad \rightarrow \quad H(18, 1)$$

Anschließend wird durch mehrfache Anwendung der vierten tgd die Summe iterativ berechnet. Hier zeigen wir die ersten zwei Belegungen:

$$\begin{aligned} \widetilde{St}(2, 2, \text{Lewis}, \text{Linda}, \text{Computer Science}, 20) \wedge \neg H(c_1, 2) \wedge H(18, 1) \quad &\rightarrow \quad H(18 + 20, 2) \\ \widetilde{St}(3, 3, \text{Smith}, \text{Steven}, \text{Mathematics}, 21) \wedge \neg H(c_1, 3) \wedge H(38, 2) \quad &\rightarrow \quad H(38 + 21, 3) \end{aligned}$$

Nach vollständiger Abarbeitung aller Tupel hat die Relation H dann dieselben Einträge wie zuvor:

c	x
18	1
38	2
59	3
82	4
104	5
129	6

Aber auch hier gilt wieder, dass das gleiche Ergebnis erzielt werden konnte, ohne die Sidetable S'' zu benötigen. Abschließend wird die letzte s-t tgd ausgeführt:

$$H(129,6) \wedge \neg H(c_2, 6+1) \quad \rightarrow \quad Res(129/6)$$

Hier gibt es keine Unterschiede zu den Regeln aus [Aug]. Die Summe der Attributwerte (129) wird durch die Tupelanzahl (6) geteilt und das Ergebnis (21.5) wird in die Result-Relation eingetragen. ■

Nachdem wir nun gesehen haben, welche Änderungen an den Regeln vorgenommen wurden und wie diese sich auswirken, wollen wir als nächstes Überlegungen anstellen, wie diese im Tool sql2sttgd integriert werden können.

4.2 Formaler Algorithmus der Transformation in sql2sttgd

Bisher wurde das Verhalten von sql2sttgd nur schriftlich und an Beispielen beschrieben. Jetzt soll der Algorithmus auch knapp als Pseudocode beschrieben werden. So kann das Vorgehen beim Erzeugen der (s-t) tgds übersichtlich dargestellt werden. Wenn die Transformation in Abschnitt 4.2.2 danach um Aggregationen erweitert wird, können leicht Unterschiede und Gemeinsamkeiten festgestellt werden.

4.2.1 Bisheriger Algorithmus

Zuerst soll der bisherige Stand der Transformation in sql2sttgd beschrieben werden. In Abschnitt 3.2.3 wurde das Tool sql2sttgd vorgestellt. In der Projektdokumentation [Kav+21] wird es ausführlich beschrieben, mit besonderem Fokus auf das Transformationsmodul. In diesem Modul geschieht die Hauptaufgabe des Tools: die Transformation der SQL-Anfrage in eine Menge

von s-t tgds. In der Dokumentation wird der Ablauf der Transformation zwar erklärt und auch mit Beispielen veranschaulicht, allerdings wird der eigentliche Algorithmus nicht direkt beschrieben. Wir wollen basierend auf den Beschreibungen in [Kav+21] hier einen formalen Algorithmus angeben, zu sehen in Algorithmus 1. Im Folgenden sollen die einzelnen Schritte des Algorithmus nun näher erläutert werden.

Algorithmus 1 Die Transformation in sql2sttgd

Input:
- Q geparste SQL-Anfrage
- S Datenbankschema

Output:
- D Menge von Abhängigkeiten
- S um Result-Relation erweitertes Datenbankschema

```
1:  method TRANSFORM(Q, S)
2:      D := {}
3:      Q_flat := {q_a | q_a ist atomare Anfrage in Q}
4:      for all q_a ∈ Q_flat do
5:          φ := BUILDBODY(S, q_a)
6:          φ, R_⋈ := EQUALIZEVARIABLES(φ, q_a)
7:          ψ := BUILDHEAD(φ, q_a, R_⋈)
8:          φ, ψ := INSERTCONSTANTS(φ, ψ, q_a)
9:          Res := BUILDRESULTRELATION(ψ)
10:         if S enthält noch keine Result-Relation then
11:             S := S ∪ {Res}
12:         else
13:             überprüfe, dass union compatible mit Res
14:         end if
15:         D := D ∪ {φ → ψ}
16:     end for
17:     return R, S
18: end method
```

Input In Übereinstimmung mit der Architektur in Abbildung 3.1, sind die Eingaben für die Transformation einerseits die geparste SQL-Anfrage Q (für Query) und das Datenbankschema S. Die SQL-Anfrage kommt aus dem Modul SQL-Parser, dort wird die SQL-Anfrage – die ursprünglich als String vorliegt – mittels der Library JSqlParser geparst. Die Anfrage steht nun als Syntaxbaum (in Form einer Java-Klassenhierarchie) zur Verfügung. Das Datenbankschema wird vom Modul DB-Reader aus einer Datenbankanbindung ausgelesen.

Output Die Ausgabe der Transformation ist eine Menge D von Abhängigkeiten (für Dependencies). In dem Stand, der in Algorithmus 1 beschrieben wird, enthält D tatsächlich aber nur s-t tgds und keine egds oder normale tgds. Neben D wird auch das Schema S ausgegeben, da es um die Ergebnisrelation erweitert wird. Im eigentlichen Java-Code wird dem Objekt für das Schema nur die Relation hinzugefügt (es ist aber keine Ausgabe der Methode). Die Ergebnisrelation beschreibt das Schema des Anfrageergebnisses,

z.B. für die Anfrage `SELECT module_name, points FROM Modules` wäre die Ergebnisrelation {`module_name`, `points`}, während sie für `SELECT * FROM Modules` folgende wäre: {`module_id`, `module_name`, `points`}.

Vereinigungen Der Kern der Transformation besteht aus dem Rumpf der Schleife von Zeile 5 bis 15. Diese Befehle werden für jede atomare Anfrage q_a aus der gesamten Anfrage Q ausgeführt. Eine atomare Anfrage ist dabei ein einfacher Select-From-Where-Block (SFW-Block) ohne Unteranfragen. Wenn die gesamte Anfrage Q nur aus einem SFW-Block besteht, dann gilt $Q = q_a$. In sql2sttgd werden aber auch Vereinigungen (`UNION`) unterstützt. Für eine Anfrage q_1 `UNION` q_2, in der zwei atomare Anfrage q_1 und q_2 vereinigt werden, wird der Rumpf der Schleife jeweils für q_1 und q_2 angewandt. In Zeile 3 wird dafür zuerst Q durchwandert und alle atomaren Anfragen q_a werden in eine Menge geschrieben. Für das letzte Beispiel ergäbe das einfach $\{q_1, q_2\}$. In der Implementation geschieht dieses sog. Flattening rekursiv. Der Rumpf der Schleife erzeugt aktuell für jede atomare Anfrage genau eine s-t tgd. Das ist allerdings nur der Fall, weil bisher keine Operationen unterstützt wurden, bei denen eine atomare Anfrage in mehreren s-t tgds resultiert. Für Anfragen mit Aggregationen oder Disjunktionen gilt diese 1:1-Beziehung nicht mehr. Die erzeugte s-t tgd $\varphi \rightarrow \psi$ wird am Ende des Rumpfs in Zeile 15 zur Menge der Abhängigkeiten D hinzugefügt.

Body initialisieren In Zeile 5 wird mit der Methode BuildBody der Body φ der s-t tgd initialisiert (für die Notation vergleiche auch Abschnitt 2.3.2). Dabei wird für jede Relation, die in der Anfrage vorkommt, ein entsprechendes Atom im Body φ erzeugt. Für jedes Attribut der Relation wird eine entsprechende Variable im Atom erstellt. Vorerst sind alle Variablen eindeutig, es gibt keine Duplikate im Body. So werden Konflikte bei gleichnamigen Spalten unterbunden – bei gleichen Variablen würden sonst ungewollt Verbunde formuliert werden. Jedem Atom wird dabei ein Index zugeordnet, den auch die Variablen erhalten. Für zwei Relationen $R_1(a, b)$ und $R_2(b, c)$ würden z.B. zwei Atome $R_1(a_1, b_1)$ und $R_2(b_2, c_2)$ erstellt werden. So können die Variablen b_1 und b_2 unterschieden werden.

Verbunde Im nächsten Schritt werden mit EqualizeVariables die Verbunde in den Body eingearbeitet. Dafür werden einige der Variablen, die bisher ja noch eindeutig sind, gleichgesetzt, um so Verbunde über die entsprechenden Attribute auszudrücken. Wenn bspw. mit den Relationen von oben ein Verbund $R_1 \bowtie R_2$ über das Attribut b stattfindet, müssen in der s-t tgd b_1 und b_2 gleichgesetzt werden. Dafür können alle Vorkommen von b_2 durch b_1 ersetzt werden (oder andersherum). Der Body wäre dann $R_1(a_1, b_1) \wedge R_2(b_1, c_2)$. Der neue Body mit den gleichgesetzten Variablen wird zurückgegeben (in der Implementation geschehen die Änderungen am Objekt selbst). Neben der Gleichsetzung wird außerdem auch das Schema der Verbundrelation $R_{\bowtie}$ ermittelt und zurückgegeben. Das ist notwendig, da sich unterschiedliche Arten von Joins unterschiedlich auf das Schema der resultierenden Relation auswirken. Ein natürlicher Verbund $R_1 \bowtie R_2$ ergibt das Schema $\{a, b, c\}$, während das Kreuzprodukt $R_1 \times R_2$ oder ein Verbund R_1 `JOIN` R_2 `ON` (`=`) beide das Schema $\{a, R_1.b, R_2.b, c\}$ ergeben.

In EqualizeVariables wird intern ein Graphenverfahren verwendet, um die gleichzusetzenden Variablen zu ermitteln. Dabei sind Variablen Knoten im Graphen, Kanten stellen Gleichheitsbeziehungen durch Verbunde dar. Im Beispiel wäre also eine Kante zwischen b_1 und b_2, während a_1 und c_2 Knoten ohne Kanten wären. In der s-t tgd werden dann alle Variablen einer Zusammenhangskomponente des Graphen gleichgesetzt. Gleichzeitig wird die Verbundrelation $R_{\bowtie}$ als Liste von Variablen mitgeführt.

Projektion Als Nächstes wird mit der Methode BuildHead der Kopf ψ der s-t tgd konstruiert. Dadurch wird die Projektion aus der `SELECT`-Klausel umgesetzt. Wenn eine Projektionsliste in der `SELECT`-Klausel angegeben ist, werden die entsprechenden Variablen aus dem Body im Head verwendet. Bei `SELECT` a`,` $R_1.b$ ergibt das z.B. $Res(a_1, b_1)$ als Kopf. Wenn alle Attribute gewählt werden, also `SELECT *`, wird die eben erstellte Verbundrelation $R_{\bowtie}$ genutzt, um den korrekten Head zu erhalten. Bei einem natürlichen Verbund wäre das $Res(a_1, b_1, c_2)$, wie eben kurz diskutiert.

Selektion Mit InsertConstants werden die Selektionen aus der Anfrage in die s-t tgd eingearbeitet. Dafür wird die `WHERE`-Klausel betrachtet. Dort werden Selektionen der Form $\langle Attribut \rangle = \langle Konstante \rangle$ in der s-t tgd umgesetzt, indem die Variable, die dem Attribut entspricht, durch die Konstante ersetzt wird. Bei einer Selektion `WHERE` $a = 3$ wird die Variable a_1 durch die Konstante 3 ersetzt, also z.B. $R_1(3, b_1)$ im Body. Die Ersetzung geschieht sowohl im Body als auch im Head, deshalb werden auch φ und ψ zurückgegeben (in der Implementation wieder nur Änderungen auf den Objekten).

Schema erweitern Um die Transformation der atomaren Anfrage nach Einsetzen der Konstanten abzuschließen, muss noch die Result-Relation zum Schema hinzugefügt werden. Dafür wird die Result-Relation *Res* mithilfe der Methode BuildResultRelation erstellt. Dabei wird aus dem Kopf ψ der generierten s-t tgd das Schema der Ergebnisrelation ermittelt. In ähnlicher Weise wie anfangs die Atome im Body aus den Relationen im Schema erzeugt wurden, wird nun umgekehrt aus dem Result-Atom im Kopf die Ergebnisrelation erzeugt. Z.B. ergibt das Atom $Res(a_1, b_1, c_2)$ die Relation $Res(a, b, c)$. Auch die Datentypen (z.B. String oder Integer) werden dabei berücksichtigt.

Danach wird das Datenbankschema um *Res* erweitert. Wenn Vereinigungen in der Anfrage Q vorhanden sind, wird in der Implementation die Ergebnisrelation nicht mehrfach hinzugefügt, aber es wird überprüft, ob die atomaren Anfragen union compatible sind (d.h. die Schemata der atomaren Anfragen passen zueinander).

Zum Schluss wird die s-t tgd $\varphi \rightarrow \psi$ noch zur Menge aller Abhängigkeiten D hinzugefügt. Wenn alle atomaren Anfragen verarbeitet wurden, werden D und das erweiterte Schema S von der Transformation zurückgegeben.

Nachdem ein Überblick gewonnen wurde, wie bisher die Transformation in sql2sttgd ablief, können wir nun leichter verstehen, wie sie um Aggregationen erweitert werden kann. Das soll im nächsten Abschnitt geschehen.

4.2.2 Erweiterung des Algorithmus um alleinstehende Aggregationen

Es sollen nun Änderungen am Algorithmus der Transformation vorgeschlagen werden, um auch die Aggregationen `MIN`, `MAX`, `SUM`, `COUNT` und `AVG` zu unterstützen. Dafür wollen wir vorerst nur die grundlegendsten Anfragen hinzufügen, nämlich alleinstehende Aggregationen. Damit sind hier Anfragen gemeint, die folgende Form haben:

`SELECT` $f(b)$
`FROM` R

Dabei sei f eine der eben genannten Aggregatfunktionen und b ein Attribut aus der Relation R (für `COUNT` gehen wir von `COUNT(*)` aus). Die Anfrage besteht also nur aus einer Aggregation, die auf ein Attribut aus genau einer Relation angewandt wird. Damit soll vorerst der einfachste Fall für Aggregationen umgesetzt werden. Einige weitere Fälle, z.B. mit Selektionen oder Gruppierung, werden in den folgenden Abschnitten ergänzt. Die Änderungen, die im Transformationsmodul nötig sind, um alleinstehende Aggregationen umzusetzen, sind im Algorithmus 2 zu sehen.

Der ursprüngliche Algorithmus wurde dabei nur um eine Verzweigung ergänzt. Der Ablauf für die Transformation einer Anfrage, wie sie bisher betrachtet wurde, wurde dabei nicht verändert. In Zeile 5 wird überprüft, ob die Anfrage aus einem alleinstehenden Aggregat besteht, wie es eben definiert wurde. Wenn das der Fall ist, werden im Block von Zeile 6 bis 14 die Regeln für Aggregate erzeugt. Ansonsten wird die Anfrage behandelt wie zuvor.
Die Kernaufgabe soll dabei die Methode GenerateRules übernehmen. Je nachdem, welche Aggregatfunktion in der Anfrage auftaucht, sollen hier die entsprechenden Abhängigkeiten erzeugt werden, wie sie in Tabelle 4.1 aufgelistet werden. Die Regeln müssen dabei an die konkrete Relation und das konkrete Aggregationsattribut angepasst werden. Die erzeugten Regeln werden dann in der Menge $D_{\text{Aggregate}}$ zurückgegeben. Für Minimum und Maximum besteht $D_{\text{Aggregate}}$ also nur aus einer s-t tgd, für Summe und Durchschnitt aus vier tgds und einer s-t tgd und für `COUNT` aus zwei tgds und einer s-t tgd. Die neu erzeugten Abhängigkeiten werden dann zur Menge aller Abhängigkeiten D hinzugefügt.
Wie auch schon bei den Anfragen ohne Aggregationen, muss auch hier die Result-Relation *Res* zum Datenbankschema hinzugefügt werden. Zusätzlich erfordern die Regeln für Aggregationen aber auch weitere neue Relationen (ausgenommen Minimum und Maximum). Das sind die Relationen $\tilde{R}$, die um durchgängige IDs erweitert wurde und für Summe und Durchschnitt die Sidetable H_{q_a}, um Zwischenergebnisse zu speichern. Auch diese müssen zum Schema hinzugefügt werden. Die Relation H_{q_a} ist dabei abhängig von der atomaren Anfrage q_a, da sie für jedes Aggregat neu erstellt werden muss. Wenn die gleiche Sidetable H für mehrere Aggregate benutzt werden würde, dann würde das zu Konflikten führen, da bei den Abhängigkeiten für Aggregationen die Voraussetzung ist, dass H zu Beginn leer ist. Sie müssen also für verschiedene Aggregate unterscheidbar sein.

Algorithmus 2 Transformation mit Erweiterung um alleinstehende Aggregationen

Input:
- Q geparste SQL-Anfrage
- S Datenbankschema

Output:
- D Menge von Abhängigkeiten
- S erweitertes Datenbankschema

1: **method** TRANSFORM(Q, S)
2: $D := \{\}$
3: $Q_{\text{flat}} := \{q_a \mid q_a \text{ ist atomare Anfrage in } Q\}$
4: **for all** $q_a \in Q_{\text{flat}}$ **do**
5: **if** q_a hat Form `SELECT` $f(b)$ `FROM` R **then**
6: $D_{\text{Aggregate}} :=$ GENERATERULES(S, q_a)
7: neue Relationen seien $\tilde{R}, H_{q_a}$ und Res
8: **if** S enthält noch keine Result-Relation **then**
9: $S := S \cup \{\tilde{R}, H_{q_a}, Res\}$
10: **else**
11: überprüfe, dass union compatible mit Res
12: $S := S \cup \{\tilde{R}, H_{q_a}\}$
13: **end if**
14: $D := D \cup D_{\text{Aggregate}}$
15: **else**
16: $\varphi :=$ BUILDBODY(S, q_a)
17: $\varphi, R_{\bowtie} :=$ EQUALIZEVARIABLES(φ, q_a)
18: $\psi :=$ BUILDHEAD($\varphi, q_a, R_{\bowtie}$)
19: $\varphi, \psi :=$ INSERTCONSTANTS(φ, ψ, q_a)
20: $Res :=$ BUILDRESULTRELATION(ψ)
21: **if** S enthält noch keine Result-Relation **then**
22: $S := S \cup \{Res\}$
23: **else**
24: überprüfe, dass union compatible mit Res
25: **end if**
26: $D := D \cup \{\varphi \to \psi\}$
27: **end if**
28: **end for**
29: **return** R, S
30: **end method**

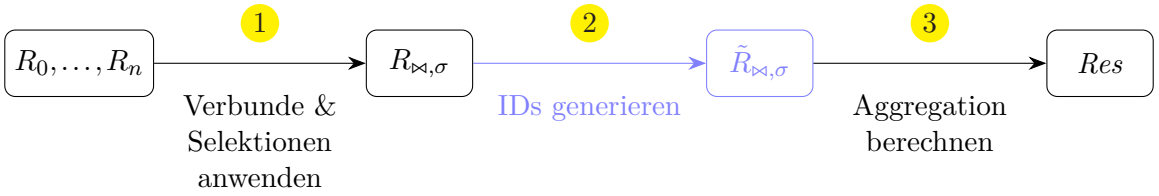

Abbildung 4.1: Aggregationen mit Verbunden und Selektionen: Ablauf des Verfahrens

Wir haben gesehen, wie der einfachste Fall für die Anwendung einer Aggregatfunktion in sql2sttgd integriert werden kann, nämlich alleinstehende Aggregationen. In den nächsten Abschnitten werden Verfahren vorgestellt, mit denen weitere Fälle ergänzt werden können.

4.3 Aggregationen mit Selektionen und Verbunden

Im vorigen Abschnitt wurde beschrieben, wie alleinstehende Aggregationen in die Transformation von sql2sttgd integriert werden können. Jetzt sollen die Aggregationen nicht mehr nur isoliert betrachtet werden, sondern auch in Kombination mit Verbunden und Selektionen. Es werden jetzt also Anfragen folgender Form untersucht:

`SELECT` $f(b)$
`FROM` $R_1 \circ_1 \cdots \circ_{n-1} R_n$
`WHERE` P

Dabei ist f wieder eine Aggregatfunktion und b das Attribut, auf welches die Funktion angewandt wird. $R_1, \ldots, R_n$ sind Relationen, die durch verschiedene Verbunde $\circ_i$ verknüpft sind. In sql2sttgd werden aktuell der natürliche Verbund, das Kreuzprodukt und der innere Verbund mit `ON`-Expression unterstützt. P sei ein Selektionsprädikat. Hier sind in sql2sttgd aktuell Prädikate der Form $a_1 = \text{const}_1$ `AND` $\ldots$ `AND` $a_m = \text{const}_m$ erlaubt, wobei a_i ein Attribut und const_i eine Konstante ist. Um auch die Verbunde und die Selektion umzusetzen, wird eine weitere Zwischentabelle $R_{\bowtie,\sigma}$ eingeführt. Diese stellt das Zwischenergebnis dar, welches man nach Anwendung der Verbunde und der Selektion auf die Relationen $R_1, \ldots, R_n$ erhält. Das Aggregat wird dann danach auf $R_{\bowtie,\sigma}$ berechnet. In Abbildung 4.1 wird schematisch dargestellt, in welcher Reihenfolge die Relationen bearbeitet werden.

In Schritt 1 werden die Abhängigkeiten erzeugt, welche die Verbunde und die Selektion darstellen. Das Ergebnis wird in die Relation $R_{\bowtie,\sigma}$ eingetragen. Für die Verbunde werden wie gewohnt Variablen gleichgesetzt, für die Selektion werden Konstanten eingesetzt bzw. mit Variablen verglichen. Nach Ausführung dieser Regeln sollten die Tupel in $R_{\bowtie,\sigma}$ dem Ergebnis folgender Anfrage entsprechen:

`SELECT` *
`FROM` $R_1 \circ_1 \cdots \circ_{n-1} R_n$
`WHERE` P

Danach wird auf $R_{\bowtie,\sigma}$ das Aggregat berechnet. Für die Funktionen `SUM`, `AVG` und `COUNT` müssen dafür zuerst in Schritt (2) die durchgängigen IDs generiert werden. Die neue Zwischenrelation mit den hinzugefügten IDs bezeichnen wir als $\tilde{R}_{\bowtie,\sigma}$. Für Minimum und Maximum kann dieser Schritt übersprungen werden, da hier keine IDs benötigt werden. Aus diesem Grund wurde der zweite Schritt in Abbildung 4.1 in Blau dargestellt.
Zum Schluss geschieht in Schritt (3) dann die eigentliche Berechnung des Aggregats mit den bekannten Regeln (s. Tabelle 4.1). Das Ergebnis wird in die Result-Relation eingetragen.
Das Verfahren soll nun an einem Beispiel beschrieben werden.

Beispiel 15. Damit die Regeln verhältnismäßig kurz bleiben, nutzen wir für dieses Beispiel nicht die Studierenden-Datenbank aus Anhang D, sondern betrachten zwei simple Relationen $R_1(a,b)$ und $R_2(b,c)$. Die Anfrage, die in Abhängigkeiten umgewandelt werden soll, lautet:

`SELECT` `SUM`(b)
`FROM` R_1 `NATURAL JOIN` R_2
`WHERE` $c = 3$

Die entsprechenden Abhängigkeiten zu dieser Anfrage sind folgende:

$$
\begin{aligned}
R_1(a_1,b_1) \wedge R_2(b_1,3) &\rightarrow R_{\bowtie,\sigma}(a_1,b_1,3) && \} \ (1)\\
R_{\bowtie,\sigma}(a_1,b_1,c_1) \wedge \neg \tilde{R}_{\bowtie,\sigma}(x_2,a_2,b_2,c_2) &\rightarrow \tilde{R}_{\bowtie,\sigma}(1,a_1,b_1,c_1) && \Big\} \ (2)\\
R_{\bowtie,\sigma}(a_1,b_1,c_1) \wedge \neg \tilde{R}_{\bowtie,\sigma}(x_1,a_1,b_1,c_1) \wedge & && \\
\tilde{R}_{\bowtie,\sigma}(x_2,a_2,b_2,c_2) \wedge \neg \tilde{R}_{\bowtie,\sigma}(x_2+1,a_3,b_3,c_3) &\rightarrow \tilde{R}_{\bowtie,\sigma}(x_2+1,a_1,b_1,c_1) && \\
\tilde{R}_{\bowtie,\sigma}(1,a_1,b_1,c_1) &\rightarrow H(b_1,1) && \Big\} \ (3)\\
\tilde{R}_{\bowtie,\sigma}(x_1,a_1,b_1,c_1) \wedge \neg H(acc_1,x_1) \wedge H(acc_2,x_1-1) &\rightarrow H(b_1+acc_2,x_1) && \\
H(acc_1,x_1) \wedge \neg H(acc_2,x_1+1) &\rightarrow Res(acc_1) &&
\end{aligned}
$$

An dieser Stelle wurde das Akkumulator-Attribut aus der Sidetable H von c zu acc umbenannt, um Verwechslungen mit dem Attribut c aus Relation R_2 zu vermeiden. Die erste Regel, eine tgd, setzt den natürlichen Verbund und die Selektion für Schritt (1) um. Das Ergebnis wird auf die Relation $R_{\bowtie,\sigma}$ abgebildet. Auf dieser werden dann mit den zwei tgds unter (2) die durchgängigen IDs erzeugt. Auf dem Ergebnis $\tilde{R}_{\bowtie,\sigma}$ wird mit den Regeln unter (3) wie gewohnt die Summe berechnet. Das Ergebnis der Aggregation befindet sich dann in der Result-Relation. ■

Nachdem wir Verbunde und Selektionen hinzugefügt haben, wird im nächsten Abschnitt ein Verfahren vorgeschlagen, mit dem auch die Gruppierung ergänzt werden kann.

$$R(g_1,a_1,b_1,t_1) \wedge \neg R(g_1,a_2,b_2,t_2) \wedge b_1 < b_2 \quad \rightarrow \quad Res(g_1,b_1)$$

(a) `MAX(b)`

$$R(g_1,a_1,b_1,t_1) \wedge \neg R(g_1,a_2,b_2,t_2) \wedge b_1 > b_2 \quad \rightarrow \quad Res(g_1,b_1)$$

(b) `MIN(b)`

$$\begin{aligned}
R(g_1,a_1,b_1,t_1) \wedge \neg \tilde{R}_g(x_2,g_1,a_2,b_2,t_2) &\rightarrow \tilde{R}_g(1,g_1,a_1,b_1,t_1) \\
R(g_1,a_1,b_1,t_1) \wedge \neg \tilde{R}_g(x_1,g_1,a_1,b_1,t_1) \wedge & \\
\tilde{R}_g(x_2,g_1,a_2,b_2,t_2) \wedge \neg \tilde{R}_g(x_2+1,g_1,a_3,b_3,t_3) &\rightarrow \tilde{R}_g(x_2+1,g_1,a_1,b_1,t_1) \\
\tilde{R}_g(1,g_1,a_1,b_1,t_1) &\rightarrow H(b_1,1,g_1) \\
\tilde{R}_g(x_1,g_1,a_1,b_1,t_1) \wedge \neg H(c_1,x_1,g_1) \wedge H(c_2,x_1-1,g_1) &\rightarrow H(b_1+c_2,x_1,g_1) \\
H(c_1,x_1,g_1) \wedge \neg H(c_2,x_1+1,g_1) &\rightarrow Res(g_1,c_1)
\end{aligned}$$

(c) `SUM(b)`

$$\begin{aligned}
R(g_1,a_1,b_1,t_1) \wedge \neg \tilde{R}_g(x_2,g_1,a_2,b_2,t_2) &\rightarrow \tilde{R}_g(1,g_1,a_1,b_1,t_1) \\
R(g_1,a_1,b_1,t_1) \wedge \neg \tilde{R}_g(x_1,g_1,a_1,b_1,t_1) \wedge & \\
\tilde{R}_g(x_2,g_1,a_2,b_2,t_2) \wedge \neg \tilde{R}_g(x_2+1,g_1,a_3,b_3,t_3) &\rightarrow \tilde{R}_g(x_2+1,g_1,a_1,b_1,t_1) \\
\tilde{R}_g(1,g_1,a_1,b_1,t_1) &\rightarrow H(b_1,1,g_1) \\
\tilde{R}_g(x_1,g_1,a_1,b_1,t_1) \wedge \neg H(c_1,x_1,g_1) \wedge H(c_2,x_1-1,g_1) &\rightarrow H(b_1+c_2,x_1,g_1) \\
H(c_1,x_1,g_1) \wedge \neg H(c_2,x_1+1,g_1) &\rightarrow Res(g_1,c_1/x_1)
\end{aligned}$$

(d) `AVG(b)`

$$\begin{aligned}
R(g_1,a_1,b_1,t_1) \wedge \neg \tilde{R}_g(x_2,g_1,a_2,b_2,t_2) &\rightarrow \tilde{R}_g(1,g_1,a_1,b_1,t_1) \\
R(g_1,a_1,b_1,t_1) \wedge \neg \tilde{R}_g(x_1,g_1,a_1,b_1,t_1) \wedge & \\
\tilde{R}_g(x_2,g_1,a_2,b_2,t_2) \wedge \neg \tilde{R}_g(x_2+1,g_1,a_3,b_3,t_3) &\rightarrow \tilde{R}_g(x_2+1,g_1,a_1,b_1,t_1) \\
\tilde{R}_g(x_1,g_1,a_1,b_1,t_1) \wedge \neg \tilde{R}_g(x_1+1,g_1,a_2,b_2,t_2) &\rightarrow Res(g_1,x_1)
\end{aligned}$$

(e) `COUNT(*)`

Tabelle 4.2: Aggregatfunktionen mit Gruppierung als Mengen von (s-t) tgds

4.4 Gruppierung

Wir haben betrachtet, wie sich die Operationen Verbund und Selektion bei Vorkommen von Aggregatfunktionen behandeln lassen. Jetzt wollen wir eine weitere Operation ergänzen, die häufig zusammen mit Aggregatfunktionen angewandt wird: die Gruppierung. In SQL erfolgt die Gruppierung in der `GROUP BY`-Klausel mit einer Liste von Gruppierungsattributen (s. Abschnitt 2.1.4). Wir betrachten an dieser Stelle nur den Fall, dass ein Gruppierungsattribut angegeben wird. Mehrere Gruppierungsattribute lassen sich aber analog umsetzen. Im Folgenden untersuchen wir also Anfragen der Form:

```
SELECT g, f(b)
FROM R
GROUP BY g
```

Dabei sei das Schema von R in Anlehnung an Tabelle 4.1 jetzt $R(g,a,b,t)$. Das Gruppierungsattribut ist g, die Aggregatfunktion wird wieder auf b angewandt, t sei Schlüssel und a steht stellvertretend für die restlichen Attribute. Um die Gruppierung zu den Aggregatfunktionen zu ergänzen, ist es ausreichend die Regeln aus Tabelle 4.1 anzupassen. Die entsprechend erweiterten Regeln sind nun in Tabelle 4.2 zu finden.

Bei einer Gruppierung wird die Aggregatfunktion nicht auf die gesamte Relation angewandt, sondern je Gruppe. Wir wollen im Folgenden kurz beschreiben, welche Unterschiede das in den Abhängigkeiten hervorruft. Für die Funktion Maximum (bzw. Minimum) beschreibt die Regeln aus Tabelle 4.1 (also ohne Gruppierung), dass jener Attributwert b_1 gesucht wird, zu dem kein anderer Wert b_2 existiert, der größer (bzw. kleiner) ist. Wenn die Gruppierung hinzukommt, beschreibt die Regel beinahe dasselbe: für die Gruppe g_1 wird jener Wert b_1 gesucht, zu dem kein anderer Wert b_2 der *gleichen Gruppe* existiert, der größer (bzw. kleiner) ist.

Für Maximum und Minimum bedeutet die Gruppierung also nur einen kleinen Unterschied in der jeweiligen s-t tgd. Bei den verbleibenden Funktionen `SUM`, `AVG` und `COUNT` sind mehr Änderungen nötig. Die grundsätzliche Idee ist hier, dass die durchgängigen IDs nicht für die gesamte Tabelle fortlaufend vergeben werden, sondern je Gruppe neu bei 1 beginnen. In Beispiel 16 ist das später gut an einer Tabelle zu sehen. Die neue Relation mit diesem ergänzten Attribut x notieren wir für den Fall der Gruppierung nun nicht mehr nur mit $\tilde{R}$, sondern mit $\tilde{R}_g$. Damit kennzeichnen wir, dass die IDs nur je Gruppe für das Attribut g eindeutig sind. In diesem Sinne ist x allein dann auch nicht mehr identifizierend für $\tilde{R}_g$, sondern Schlüssel ist nun $\{x,g\}$. Im Folgenden werden wir x trotzdem vereinfacht als ID bezeichnen.

Schauen wir uns nun die veränderten Regeln zur Erzeugung dieser IDs im Detail an. Die erste tgd dient auch mit Gruppierung wieder der Initialisierung: in $\tilde{R}_g$ wird ein Tupel mit der ID 1 erstellt. Die Regel wird jetzt aber nicht mehr nur einmal am Anfang ausgeführt, sondern pro Gruppe einmal. Dafür wird mit dem ersten Atom $R(g_1,a_1,b_1,t_1)$ ein Tupel aus R gewählt, zu

dessen Gruppe g_1 noch keine Einträge in $\tilde{R}_g$ existieren. Letzteres wird durch das zweite Atom $\neg\tilde{R}_g(x_2, g_1, a_2, b_2, t_2)$ ausgedrückt. Die Gruppe g_1 wurde also noch nicht behandelt. Das Tupel wird dann in $\tilde{R}_g$ durch das Atom $\tilde{R}_g(1, g_1, a_1, b_1, t_1)$ im Head eingetragen.
Die zweite tgd erstellt wieder die fortlaufenden IDs für die verbleibenden Tupel. Es wird wieder ein Tupel aus R gewählt (1. Atom im Body: $R(g_1, a_1, b_1, t_1)$), das noch nicht auf $\tilde{R}_g$ abgebildet wurde (2. Atom: $\neg\tilde{R}_g(x_1, g_1, a_1, b_1, t_1)$). Dann wird ein Tupel aus $\tilde{R}_g$ mit der gleichen Gruppe g_1 gewählt (3. Atom: $\tilde{R}_g(x_2, g_1, a_2, b_2, t_2)$), zu dem kein Tupel mit einer größeren ID existiert (4. Atom: $\neg\tilde{R}_g(x_2{+}1, g_1, a_3, b_3, t_3)$). Es wird also das Tupel mit der höchsten ID x_2 gesucht. Das noch nicht auf $\tilde{R}_g$ abgebildete Tupel wird dann mit der nächsten ID $x_2{+}1$ in diese Relation eingetragen (Head: $\tilde{R}_g(x_2 + 1, g_1, a_1, b_1, t_1)$). Auch hier sind die Atome von der Struktur wieder sehr ähnlich zu den vorigen aus Tabelle 4.1. Es muss nur sichergestellt werden, dass die entsprechenden Tupel innerhalb einer Gruppe gesucht werden. Dies geschieht, indem wir an den benötigten Stellen die gleiche Variable für das Gruppierungsattribut verwenden.

Auch die verbleibenden Abhängigkeiten, die der Berechnung des Aggregats dienen, sind strukturell sehr ähnlich geblieben. Wir erklären das Verhalten hier exemplarisch an der Summe. Nachdem die IDs erzeugt wurden, wird mit der dritten tgd für jede Gruppe das erste Tupel verarbeitet (Body: $\tilde{R}_g(1, g_1, a_1, b_1, t_1)$). Mit dem Head $H(b_1, 1, g_1)$ wird dann der Wert von b initial in die Sidetable H eingetragen. Als Schlüssel wird (wie oben diskutiert) nicht nur x verwendet, sondern x und g sind zusammen Schlüssel.
Mit der vorletzten Regel werden dann die restlichen Werte der Gruppe akkumuliert. Es wird ein Tupel aus $\tilde{R}_g$ gewählt (1. Atom im Body: $\tilde{R}_g(x_1, g_1, a_1, b_1, t_1)$), das noch nicht behandelt wurde (2. Atom: $\neg H(c_1, x_1, g_1)$). Das dritte Atom $H(c_2, x_1 - 1, g_1)$ beschreibt, dass das Tupel auch jenes ist, welches als nächstes zu bearbeiten ist (der letzte Eintrag für die Gruppe war mit $x_1 - 1$). Im Head $H(b_1 + c_2, x_1, g_1)$ wird der neue Attributwert b_1 zur bisherigen Zwischensumme c_2 der Gruppe addiert und in H eingetragen.
Mit der letzten Regeln, einer s-t tgd, wird dann für jede Gruppe die Summe auf die Result-Relation abgebildet. Diese erhält man aus dem letzten Eintrag je Gruppe in H, was durch das zweite Atom $\neg H(c_2, x_1 + 1, g_1)$ beschrieben wird. In unserem Fall wird auch das Gruppierungsattribut mit in der Result-Relation übernommen, das hängt aber davon ab, ob es auch in der `SELECT`-Klausel der SQL-Anfrage steht.

Mit den Abhängigkeiten in Tabelle 4.2 konnten wir also zeigen, dass nur die Regeln leicht modifiziert werden müssen, um auch Gruppierungen mit den Aggregationen zusammen als Abhängigkeiten darzustellen. Es sind keine Änderungen nötig, die das gesamte Verfahren der Umwandlung betreffen, wie es in Abschnitt 4.3 vorgestellt wurde. Aus diesem Grund ist es auch kompatibel mit dem dort vorgestellten Ablauf. Wenn eine Aggregation zusammen mit Verbunden, Selektion und Gruppierung in einer Anfrage vorkommt, kann der Ablauf also derselbe bleiben: in Schritt 1 werden die Verbunde und die Selektion umgesetzt, in Schritt 2 werden die IDs je Gruppe generiert (falls benötigt) und in Schritt 3 werden die Aggregate je Gruppe mit den Regeln aus Tabelle 4.2 berechnet. Die Gruppierung lässt sich also einfach durch Austausch der

Regeln in das Verfahren integrieren. Wir haben die Regeln bisher nur abstrakt beschrieben, nun wollen wir sie abschließend auch an einem Beispiel untersuchen.

Beispiel 16. Wir greifen die Anfrage aus Beispiel 4 auf: es wird das Durchschnittsalter der Studierenden je Studiengang gesucht:

```
SELECT course, AVG(age)
FROM Students
GROUP BY course
```

D.h. das Gruppierungsattribut ist *course* (co) und die Aggregatfunktion wird, wie bei den vorigen Beispielen, auf *age* (a) angewandt. Die zwei tgds, mit denen die IDs generiert werden, sind folgende:

$$St(sid_1, sn_1, fn_1, co_1, a_1) \wedge \neg \widetilde{St}_{co}(x_2, sid_2, sn_2, fn_2, co_1, a_2) \\ \rightarrow \widetilde{St}_{co}(1, sid_1, sn_1, fn_1, co_1, a_1)$$

$$St(sid_1, sn_1, fn_1, co_1, a_1) \wedge \neg \widetilde{St}_{co}(x_1, sid_1, sn_1, fn_1, co_1, a_1) \wedge \\ \widetilde{St}_{co}(x_2, sid_2, sn_2, fn_2, co_1, a_2) \wedge \neg \widetilde{St}_{co}(x_2 + 1, sid_3, sn_3, fn_3, co_1, a_3) \\ \rightarrow \widetilde{St}_{co}(x_2 + 1, sid_1, sn_1, fn_1, co_1, a_1)$$

Nachdem diese beiden Regeln vollständig ausgeführt wurden, wurden in der Relation $\widetilde{St}_{co}$ die IDs pro Gruppe für alle Tupel generiert. Die Relation sieht dann folgendermaßen aus (die Gruppen wurden dabei farblich hervorgehoben):

x	sid	surname	forename	course	age
1	1	Miller	Mark	Electrical Engineering	18
1	2	Lewis	Linda	Computer Science	20
2	5	Brown	Bob	Computer Science	22
1	3	Smith	Steven	Mathematics	21
2	4	Smith	Sarah	Mathematics	23
3	7	Nelson	Nancy	Mathematics	25

Mit den zwei tgds zur Berechnung der Summe auf der linken Seite erhalten wir dann die Sidetable H auf der rechten Seite:

$$\widetilde{St}_{co}(1, sid_1, sn_1, fn_1, co_1, a_1) \rightarrow H(a_1, 1, co_1)$$

$$\widetilde{St}_{co}(x_1, sid_1, sn_1, fn_1, co_1, a_1) \wedge \neg H(c_1, x_1, co_1) \wedge H(c_2, x_1 - 1, co_1) \rightarrow H(a_1 + c_2, x_1, co_1)$$

c	x	course
18	1	Electrical Engineering
20	1	Computer Science
42	2	Computer Science
21	1	Mathematics
44	2	Mathematics
69	3	Mathematics

Abschließend wird die s-t tgd für jede Gruppe einmal ausgeführt, um mit der berechneten Summe und der Gruppengröße die Durchschnitte auf die Result-Relation abzubilden:

$$H(c_1, x_1, co_1) \wedge \neg H(c_2, x_1 + 1, co_1) \rightarrow Res(co_1, c_1/x_1)$$

course	avg
Electrical Engineering	18
Computer Science	21
Mathematics	23

■

Wir haben bisher verschiedene Erweiterungen bzgl. der Aggregatfunktionen untersucht. Als nächstes werden weitere Änderungen vorgeschlagen, die notwendig sind, um die Aggregatfunktionen auch praktisch umsetzen zu können. Hierfür schauen wir uns an, wie das XML-Eingabeformat von ChaTEAU erweitert werden muss. Die entsprechenden Änderungen werden im nächsten Abschnitt erklärt.

4.5 Erweiterung des XML-Formats

Das Tool ChaTEAU erhält die Inputs in Form einer XML-Datei (s. Abschnitt 3.2.1). Um die Erweiterungen umsetzen zu können, die in den Abschnitten 3.1 und 4.1 vorgeschlagen wurden, sind einige Änderungen am bisherigen XML-Eingabeformat notwendig. Diese sollen hier vorgestellt und diskutiert werden.

4.5.1 Bisheriger Stand

Das bisherige XML-Format enthält Schema und Instanz der Datenbank und die Abhängigkeiten für den CHASE. Schema und Instanz werden für die Fälle, die in dieser Arbeit betrachtet werden, bereits gut dargestellt. Hier sind keine weiteren Anpassungen nötig. Für den Teil der Abhängigkeiten sind allerdings Erweiterungen notwendig, damit weitere Operationen implementiert werden können. Beispiel 17 zeigt das bestehende Format für Abhängigkeiten exemplarisch.

Beispiel 17. Angenommen, folgende simple s-t tgd soll als XML in ChaTEAU eingegeben werden (vgl. Beispiel 1):

$$Students(student\_id_1, surname_1, forename_1, \text{Mathematics}, age_1) \rightarrow Result(surname_1)$$

Der Abschnitt im XML (neben Schema, Instanz und möglicherweise weiteren Abhängigkeiten), der diese s-t tgd enthält, sähe so aus:

```
<sttgd>
    <body>
        <atom name="Students">
            <variable name="student_id" type="V" index="1"/>
            <variable name="surname" type="V" index="1"/>
            <variable name="forename" type="V" index="1"/>
            <constant name="course" value="Mathematics"/>
            <variable name="age" type="V" index="1"/>
        </atom>
    </body>
    <head>
        <atom name="Result">
            <variable name="surname" type="V" index="1"/>
        </atom>
    </head>
</sttgd>
```

■

Die gesamte s-t tgd taucht unter dem Element `sttgd` auf. Die s-t tgd besteht aus `head` und `body`. Hier können jeweils eine Liste von `atom`-Elementen stehen (im Beispiel ist jeweils nur eines). Ein `atom`-Element entspricht einem relationalen Atom in der s-t tgd. Dementsprechend besteht es aus einem Attribut `name` (für den Namen der Relation) und einer Liste von `variable`- bzw. `constant`-Elementen. Hier könnten auch Nullwerte stehen. Atome lassen sich negieren, indem das Attribut `negation` den Wert `true` erhält (im Beispiel ausgelassen, da der Default `false` ist). Anzahl und Namen der Variablen und Konstanten müssen mit dem Schema übereinstimmen. Variablen und Konstanten benötigen beide ein Attribut `name`, damit sie der richtigen Spalte zugeordnet werden können. Variablen benötigen außerdem eine Angabe über den Quantor (`type="V"` für allquantifiziert, `type="E"` für existenzquantifiziert) und einen Index. Konstanten enthalten noch den eigentlichen Wert der Konstante. Die Semantik der Konstanten ist ähnlich, wie sie gegen Ende von Abschnitt 2.3.2 als Notation definiert wurde: wenn eine Konstante an Stelle einer Variable steht, dann bedeutet es, dass diese Variable den Wert der Konstanten haben soll. Im XML erfolgt die Zuordnung allerdings nicht über die Position (wie in unserer Definition), sondern über die Namen.

4.5.2 Vergleichsatome

Die einzige Form von Atomen, die im XML unter den Abhängigkeiten bisher dargestellt werden können, sind relationale Atome. Diese sind zwar auch die wichtigsten (da sie notwendigerweise

in jeder sinnvollen Abhängigkeit auftauchen müssen), um aber weitere Operationen (z.B. Aggregatfunktionen) umsetzen zu können, sind auch weitere Formen von Atomen nötig. Wir fassen die hier neu hinzugefügten Atome unter der Bezeichnung „Vergleichsatome“ zusammen. Damit sind alle Atome gemeint, die einen Operator $\theta \in \{<, \leq, \neq, \geq, >, =\}$ beinhalten, z.B. $points_1 > 3$ (s. gleich Bsp. 18). Durch diese Vergleichsatome lassen sich vor allem weitere Selektionen umsetzen, nämlich jene, in denen Attribute nicht einer Konstante gleich gesetzt werden, sondern mit einem θ verglichen werden (also Bereichsanfragen oder Ungleichheit). Für solche Selektionen werden dann neben den relationalen Atomen weitere Vergleichsatome im Rumpf durch Konjunktion hinzugefügt (s. den ersten Abschnitt unter 3.1.3). Außerdem werden sie auch für die Aggregatfunktionen `MIN` und `MAX` verwendet (s. Tabelle 4.1). Vergleichsatome werden wir auch nutzen, um später Funktionen unterzubringen.
Es ist anzumerken, dass der Operator $=$ bereits implizit darstellbar ist: bei Gleichsetzung von zwei Variablen $v_1 = v_2$ können alle Vorkommnisse von v_2 durch v_1 ersetzt werden (oder andersherum). Wenn eine Variable einer Konstante gleichgesetzt wird ($v_1 = c_1$), werden alle Vorkommnisse von v_1 durch die Konstante c_1 ersetzt. Die Erweiterung um Vergleichsatome wollen wir uns zunächst an einem Beispiel ansehen.

Beispiel 18. Die Anfrage auf der linken Seite mit einem Vergleich in der Selektion, würde nach der Theorie als s-t tgd so umgesetzt werden:

```
SELECT module_name
FROM Modules
WHERE points > 3
```

$$Modules(mid_1, mn_1, po_1) \wedge po_1 > 3 \rightarrow Res(mn_1)$$

Der Teil im XML, der diese s-t tgd darstellt, würde dann folgender sein (das Vergleichsatom wurde grün hervorgehoben):

```
<sttgd>
    <body>
        <atom name="Modules">
            <variable name="module_id" type="V" index="1"/>
            <variable name="modulename" type="V" index="1"/>
            <variable name="points" type="V" index="1"/>
        </atom>
    </body>
    <body-comparisons>
        <comparison operator="greater">
            <left>
                <variable name="points" type="V" index="1"/>
            </left>
            <right>
                <math-constant value="3"/>
            </right>
        </comparison>
    </body-comparisons>
    <head>
        <atom name="Result">
            <variable name="modulename" type="V" index="1"/>
        </atom>
    </head>
</sttgd>
```

■

Um Vergleiche im XML darstellen zu können, wurden auf Ebene von `body` und `head` zwei neue Elemente eingeführt: `body-comparisons` und `head-comparisons`. In gleicher Weise, wie `body` und `head` Listen von relationalen Atomen enthalten (`atom`-Elemente), tauchen dort Listen von Vergleichsatomen auf (`comparison`-Elemente). Die Vergleichsatome für den Rumpf der s-t tgd stehen unter `body-comparisons` und jene für den Kopf unter `head-comparisons`. Im Beispiel taucht genau ein Vergleich im Rumpf auf, deswegen enthält `body-comparisons` auch genau ein `comparison`-Element für diesen Vergleich. Im Kopf der s-t tgd sind keine Vergleichsatome, deshalb ist `head-comparisons` leer.
Ein `comparison`-Element muss nun immer ein Attribut `operator` haben. Zulässige Werte sind: `less`, `less-equal`, `not-equal`, `greater-equal`, `greater`, `equal` für die oben genannten Operatoren. `comparison` hat dann zwei Kindelemente `left` und `right` (für den linken bzw. rechten Teil des Vergleichs). Die Elemente `left` und `right` sind nötig, um die Richtung des Operators unterscheiden zu können. Für die Fälle $=$ und $\neq$ wären sie zwar nicht notwendig, da diese symmetrisch sind. Um die Konsistenz zu wahren, müssen `left` und `right` aber auch hier auftauchen. Unter `left` muss dann eine Variable vorhanden sein, andere Elemente sind nicht zugelassen. Unter `right` steht genau ein Element, allerdings sind hier verschiedene Typen zugelassen: `variable`, `math-constant`, `null` oder `function`. Wie wir gleich sehen werden, lassen sich über `function` verschachtelte Ausdrücke bilden. Die linke Seite schränken wir auf Variablen ein, da so sichergestellt werden kann, dass sich ein derartiger Vergleich immer auf mindestens eine Variable bezieht. Das ist in unserem Fall allerdings keine wirkliche Einschränkung, da jeder sinnvolle Vergleich sowieso immer mindestens eine Variable enthält und diese auch alleine auf die linke Seite gestellt werden kann. Das Element `math-constant` unterscheidet sich leicht vom Element `constant`, welches im `atom` genutzt wird. Und zwar benötigt `constant` ein Attribut `name`, mit dem die Konstante einer Spalte zugeordnet werden kann. Bei `math-constant` ist dieses Attribut nicht notwendig und auch nicht sinnvoll. Sie werden lediglich in Vergleichen und Berechnungen genutzt und können daher nicht immer einer Spalte zugeordnet werden.

4.5.3 Funktionen

Der zweite Teil der hier vorgeschlagenen XML-Erweiterungen sind Funktionen. Im Rahmen dieser Arbeit wurden dafür zunächst die Funktionen $+, -, \cdot, /, \min, \max$ hinzugefügt. Funktionen ließen sich bisher gar nicht im XML darstellen. Nach der Erweiterung können solche Funktionen nun im rechten Teil von Vergleichen eingesetzt werden. Die Funktionen sind vor allem für die Umsetzung der Aggregationen erforderlich, da hier natürlich mit Funktionen gearbeitet wird. Wir wollen auch hier wieder zuerst ein Beispiel betrachten.

Beispiel 19. Um die Änderungen an einem einigermaßen übersichtlichen Beispiel zu zeigen, betrachten wir eine s-t tgd, wie sie bei einer Aggregatfunktion `SUM` entstehen könnte (vgl. letzte Regel in Untertabelle 4.1c):

$$H(c_1, x_1) \wedge \neg H(c_2, x_2) \wedge x_2 = x_1 + 1 \rightarrow Res(c_1)$$

Im XML lassen wir Funktionen nur als rechte Seite eines Vergleichs zu und nicht direkt in einem relationalen Atom (wie es bei den Regeln in Tabelle 4.1 der Fall ist, um eine kürzere Notation zu haben). Aus diesem Grund müssen wir eine neue Variable x_2 einführen, die dem gewünschten Ausdruck $x_1 + 1$ gleichgesetzt wird, um die s-t tgd im XML darstellen zu können. Das Ergebnis wäre dann folgendes (die Funktion farblich hervorgehoben):

```
<sttgd>
    <body>
        <atom name="H">
            <variable name="c" type="V" index="1"/>
            <variable name="x" type="V" index="1"/>
        </atom>
        <atom name="H" negation="true">
            <variable name="c" type="V" index="2"/>
            <variable name="x" type="V" index="2"/>
        </atom>
    </body>
    <body-comparisons>
        <comparison operator="equal">
            <left>
                <variable name="x" type="V" index="2"/>
            </left>
            <right>
                <function operator="addition">
                    <left>
                        <variable name="x" type="V" index="1"/>
                    </left>
                    <right>
                        <math-constant value="1"/>
                    </right>
                </function>
            </right>
        </comparison>
    </body-comparisons>
    <head>
        <atom name="Result" negation="false">
            <variable name="c" type="V" index="1"/>
        </atom>
    </head>
    <head-comparisons/>
</sttgd>
```

■

Die eigentliche Funktion aus dem Beispiel (nämlich $x_1 + 1$) wird im Element `function` dargestellt. Man sieht hier wieder eine ähnliche Struktur, wie bereits bei `comparison`. Auch `function` benötigt ein Attribut `operator`. Mögliche Werte sind hier diesmal `addition`, `subtraction`, `multiplication`, `division`, `minumum` und `maximum`. Und auch hier sind wieder die beiden Kind-

elemente `left` und `right` nötig. Unter `left` und `right` können auch hier wieder beliebige Expressions stehen, also Variablen, Konstanten (`math-constant`), Nullwerte oder weitere Funktionen. Die Einschränkung, dass im linken Teil nur Variablen zugelassen sind, entfällt hier allerdings. Obwohl die meisten der unterstützten Funktionen kommutativ sind (nämlich alle bis auf Subtraktion und Division), sind auch hier in allen Fällen `left` und `right` erforderlich, damit überall die gleiche Struktur verwendet wird. So lässt sich das XML leichter in ChaTEAU einlesen. Damit schränken wir uns zwar auf binäre Funktionen ein, sodass sich bspw. eine mehrfache Summe $v_1 + v_2 + v_3$ nicht direkt darstellen lässt. Allerdings sind alle betroffenen Funktionen (Summe, Produkt, Minimum, Maximum) auch assoziativ und kommutativ, sie lassen sich also auch mit beliebiger Klammerung verschachtelt darstellen, also z.B. $v_1 + (v_2 + v_3)$.
Funktionen sind unter `comparison`-Elementen zugelassen, nicht aber anstelle von Variablen in einem `atom`. So besteht eine größere Trennung zwischen den bisher verwendeten Atomen und Variablen einerseits und den Vergleichen und Funktionen andererseits. Letztere lassen sich dadurch leichter in ChaTEAU ergänzen.

Nachdem wir uns vor allem auf Aggregationen konzentriert haben, sollen im letzten Abschnitt dieses Kapitels auch weitere Operationen behandelt werden. Es werden Konzepte zur Umsetzung bisher nicht untersuchter Operationen vorgestellt und es wird ein Überblick zur Umsetzbarkeit verschiedener Operationen gegeben.

4.6 Weitere neu entwickelte Theorie

In Abschnitt 3.1.4 wurden Operationen und sprachliche Elemente aus SQL aufgelistet, die bisher bei der Umwandlung von Anfragen nicht betrachtet wurden. Im Rahmen dieser Arbeit wurden zu zwei dieser Operationen auch Konzepte erstellt, wie sich auch diese umsetzen lassen, nämlich zu äußeren Verbunden und zu geschachtelten Anfragen. Da diese aber nicht den Schwerpunkt dieser Arbeit bilden, werden sie hier nur knapp vorgestellt.

4.6.1 Äußere Verbunde

Bei einem äußeren Verbund zwischen zwei Relationen werden auch Dangling Tuples aus einer oder zwei der beteiligten Relationen mit ins Verbundergebnis aufgenommen. Dangling Tuples sind dabei jene Tupel, für die kein Verbundpartner in der anderen Relation existiert. Um äußere Verbunde mittels s-t tgds umzusetzen, sind mehrere s-t tgds für eine Anfrage nötig. Angenommen es soll ein Verbund R_1 `FULL OUTER JOIN` R_2 zwischen zwei Relationen $R_1(a, b)$ und $R_2(b, c)$ stattfinden, dann ließe sich dieser mittels folgender drei s-t tgds darstellen:

$$
\begin{aligned}
R_1(a_1,b_1) \wedge R_2(b_1,c_1) &\rightarrow Res(a_1,b_1,c_1) \\
R_1(a_1,b_1) \wedge \neg R_2(b_1,c_1) &\rightarrow \exists C : Res(a_1,b_1,C) \\
\neg R_1(a_1,b_1) \wedge R_2(b_1,c_1) &\rightarrow \exists A : Res(A,b_1,c_1)
\end{aligned}
$$

Die erste s-t tgd alleine würde auch bei einem natürlichen Verbund verwendet werden. Sie bildet alle Tupel auf das Ergebnis ab, für die ein Verbundpartner existiert. Mit der zweiten s-t tgd werden Dangling Tuples aus R_1 ins Ergebnis übernommen. Das negierte Atom im Rumpf beschreibt dabei, dass kein Tupel in R_2 mit dem gleichen Wert für b existiert. Im Ergebnis wird dieses Tupel mit einer existenzquantifizierten Variable eingetragen. Nach Ausführung des CHASE wird an dieser Stelle also ein Nullwert eingetragen. Die dritte s-t tgd bildet analog die Dangling Tuples aus R_2 ab. Wenn statt eines Full Outer Joins ein Left Outer Join dargestellt werden soll, dann würde man nur die erste und zweite s-t tgd verwenden. Bei einem Right Outer Join wären es entsprechend nur die erste und dritte.

4.6.2 Geschachtelte Anfragen

Für geschachtelte Anfragen mit `IN` liegt bereits ein Konzept in [Aug] vor. Diese lassen sich ähnlich wie Verbunde auch über gleichnamige Variablen umsetzen (s. dazu auch den letzten Abschnitt unter 3.1.3). Geschachtelte Anfragen können aber auch mit den Keywords `EXISTS`, `NOT IN`, `ANY` oder `ALL` formuliert werden. `ANY` und `ALL` stehen hierbei immer zusammen mit einem Vergleichsoperator $\theta \in \{<, \leq, \neq, =, \geq, >\}$. Wir haben Konzepte erarbeitet, wie diese Fälle auch mit s-t tgds umgesetzt werden können. Diese sollen im Folgenden kurz erklärt werden.

Eine geschachtelte Anfrage mit `EXISTS` wird ähnlich konstruiert, wie bei `IN`. Die Atome für die innere Anfrage werden durch Konjunktion zum Body der s-t tgd hinzugefügt. Allerdings werden hier keine Variablen gleichgesetzt. Dadurch wird dem Body die Bedingung hinzugefügt, dass das Ergebnis der inneren Anfrage nicht leer sein darf.

Der Quantor `ANY` kann ähnlich behandelt werden wie bereits das `IN`. Dabei ist anzumerken, dass der Fall `= ANY` äquivalent zu `IN` ist. Bei `ANY` werden die Atome der inneren Anfrage zum Body hinzugefügt. Zusätzlich ist noch ein Vergleichsatom notwendig. Hier wird die Variable, die zur äußeren Anfrage gehört, über den Vergleichsoperator θ mit der entsprechenden Variablen der inneren Anfragen verglichen. Wenn wir, wie bereits bei den Outer Joins, von zwei simplen Relationen $R_1(a,b)$ und $R_2(b,c)$ ausgehen, sehen wir rechts die s-t tgd, die der Anfrage links entspricht:

```
SELECT * FROM R1
WHERE b θ ANY (
    SELECT b FROM R2)
```

$$R_1(a_1,b_1) \wedge R_2(b_2,c_2) \wedge b_1 \,\theta\, b_2 \rightarrow Res(a_1,b_1)$$

Auch beim Quantor `ALL` benötigen wir ein Vergleichsatom im Body. Im Unterschied zu `ANY` werden aber die Atome der inneren Anfrage und auch das Vergleichsatom negiert. Betrachten wir die gleiche Anfrage wie eben, nur dass das `ANY` durch ein `ALL` ersetzt wird, also:

```
SELECT * FROM R1
WHERE b θ ANY (
    SELECT b FROM R2)
```

$$R_1(a_1, b_1) \wedge \neg R_2(b_2, c_2) \wedge b_1 \,\bar{\theta}\, b_2 \rightarrow Res(a_1, b_1)$$

Dabei sei $\bar{\theta}$ der negierte Vergleichsoperator zu θ, also $<$ entspricht $\geq$, $>$ entspricht $\leq$ und $=$ entspricht $\neq$. Es ist wieder anzumerken, dass der Fall `<> ALL` äquivalent zu `NOT IN` ist. Bei der Regel fällt auch auf, dass sie eine große Ähnlichkeit zu den s-t tgds für Minimum und Maximum bei den Aggregationen hat (s. Tabelle 4.1). Das lässt sich folgendermaßen begründen: wenn b z.B. kleiner sein soll, als *alle* Werte von b aus R_2, dann ist das gleichbedeutend dazu, dass b kleiner als das *Minimum* aller b-Werte von R_2 ist.

Wir haben nun gesehen, wie sich im Prinzip einfache geschachtelte Anfragen umsetzen lassen. Hier sollte allerdings noch weiter untersucht werden, wie sich unterschiedliche Arten von inneren Anfragen auf die Regeln für die gesamte Anfrage auswirken, bspw. wenn bereits mehrere s-t tgds für die innere Anfrage nötig sind. Das kann z.B. bei Disjunktionen in Selektionsprädikaten oder bei Aggregationen der Fall sein. Auch mehrfach geschachtelte Anfragen sollten dabei bedacht werden.

4.6.3 Überblick zur Darstellbarkeit von SQL-Operationen

In Zusammenarbeit mit Dennis Spolwind (vgl. [Spo]) wurde begonnen, eine Übersicht zu erarbeiten, die möglichst viele SQL-Operationen und -Keywords auflistet und zeigt, wie diese als Abhängigkeiten dargestellt werden können. In [Spo] wird diese Übersicht dann mit dem Standard von PostgreSQL verglichen. Mit den Tabellen haben wir versucht, eine knappe, aber präzise Übersicht zu geben, wie sich welche Operationen aus SQL darstellen lassen. Dabei wurden Klassen von Operationen betrachtet (z.B. Verbunde, Mengenoperationen) und jeweils eine Tabelle aufgestellt, welche die einzelnen Fälle unterscheidet (für Mengenoperationen z.B. `UNION`, `EXCEPT` und `INTERSECT`). Da sie viel Platz beanspruchen, werden die Tabellen im Anhang E aufgeführt. Exemplarisch werden aber an dieser Stelle in Tabelle 4.3 die Verbunde aufgeführt. Wir sehen z.B., dass ein natürlicher Verbund über das Attribut d in einer s-t tgd dargestellt wird, indem dieselben Variablen zwischen zwei Atomen verwendet werden. Bei einem Kreuzprodukt dagegen dürfen keine Variablen zwischen den zwei Atomen identisch sein.

In diesem Kapitel wurden vor allem verschiedene Konzepte vorgestellt, wie Aggregationen als Abhängigkeiten dargestellt werden können. Das bestehende Konzept dazu aus [Aug] konnte verbessert werden. Wir haben einen Ansatz erarbeitet, wie Aggregationen in das Tool sql2sttgd integriert werden können. Es wurden weitere Fälle von Anfragen mit Aggregationen betrachtet,

Operation	**Anfrage in SQL**	**Abhängigkeiten**
natürlicher Verb.	`SELECT * FROM` R `NATURAL JOIN` S	$R(a_1,\ldots,a_n,d) \wedge S(b_1,\ldots,b_m,d)$ $\rightarrow Res(a_1,\ldots,a_n,b_1,\ldots,b_m,d)$
Kreuzprodukt	`SELECT * FROM` R `CROSS JOIN` S	$R(a_1,\ldots,a_n) \wedge S(b_1,\ldots,b_m)$ $\rightarrow Res(a_1,\ldots,a_n,b_1,\ldots,b_m)$
`JOIN ON`	`SELECT * FROM` R `JOIN` S `ON` $R.d = S.e$	$R(a_1,\ldots,a_n,d) \wedge S(b_1,\ldots,b_m,e)$ $\wedge d = e \rightarrow Res(a_1,\ldots,a_n,b_1,\ldots,b_m,e)$
`USING`	`SELECT * FROM` R `JOIN` S `USING` (d_1,d_2)	$R(a_1,\ldots,a_n,d_1,d_2) \wedge S(b_1,\ldots,b_m,d_1,d_2)$ $\rightarrow Res(a_1,\ldots,a_n,b_1,\ldots,b_m,d_1,d_2)$
`LEFT JOIN`	`SELECT * FROM` R `LEFT JOIN` S `ON` $R.d = S.e$	$R(a_1,\ldots,a_n,d) \wedge S(b_1,\ldots,b_m,e)$ $\wedge d = e \rightarrow Res(a_1,\ldots,a_n,b_1,\ldots,b_m,e)$, $R(a_1,\ldots,a_n,d) \wedge \neg S(b_1,\ldots,b_m,d)$ $\rightarrow \exists B_1,\ldots,B_n : Res(a_1,\ldots,a_n,B_1,\ldots,B_m,d)$
`RIGHT JOIN`	`SELECT * FROM` R `RIGHT JOIN` S `ON` $R.d = S.e$	$R(a_1,\ldots,a_n,d) \wedge S(b_1,\ldots,b_m,e)$ $\wedge d = e \rightarrow Res(a_1,\ldots,a_n,b_1,\ldots,b_m,e)$, $S(b_1,\ldots,b_m,e) \wedge \neg R(a_1,\ldots,a_n,e)$ $\rightarrow \exists A_1,\ldots,A_n : Res(A_1,\ldots,A_n,b_1,\ldots,b_m,e)$
`FULL OUTER JOIN`	`SELECT * FROM` R `FULL OUTER JOIN` S `ON` $R.d = S.e$	$R(a_1,\ldots,a_n,d) \wedge S(b_1,\ldots,b_m,e)$ $\wedge d = e \rightarrow Res(a_1,\ldots,a_n,b_1,\ldots,b_m,e)$, $R(a_1,\ldots,a_n,d) \wedge \neg S(b_1,\ldots,b_m,d)$ $\rightarrow \exists B_1,\ldots,B_n : Res(a_1,\ldots,a_n,B_1,\ldots,B_m,d)$, $S(b_1,\ldots,b_n,e) \wedge \neg R(a_1,\ldots,a_n,e)$ $\rightarrow \exists A_1,\ldots,A_n : Res(A_1,\ldots,A_n,b_1,\ldots,b_m,e)$

Tabelle 4.3: Darstellung von Verbunden als Mengen von Abhängigkeiten

nämlich Anfragen, bei denen Aggregationen gleichzeitig mit weiteren Operationen vorkommen (Verbunde, Selektion, Gruppierung). Auch das XML-Eingabeformat von ChaTEAU wurde erweitert. Im nächsten Kapitel wollen wir nun die Implementation der hier genannten Konzepte beschreiben.

Kapitel 5

Implementierung

In diesem Kapitel

Im letzten Kapitel haben wir Konzepte vorgestellt, wie Aggregationen in SQL-Anfragen allgemein als Abhängigkeiten darstellbar sind. Wir haben erklärt, wie sie sich im Tool sql2sttgd integrieren lassen und welche Änderungen dafür nötig wären. Jetzt wollen wir betrachten, wie die Umsetzung für alleinstehende Aggregationen geschehen ist (vgl. Abschnitte 4.1 und 4.2.2). Zuerst wird beschrieben, wie die interne Datenrepräsentation angepasst werden musste. Hier geht es darum, in welcher Form z.B. eine tgd oder das Datenbankschema intern dargestellt wird. Dies bildet dann die Grundlage, um die theoretisch beschriebenen Regeln aus Tabelle 4.1 auch im Programm umzusetzen. Wie diese Regeln in sql2sttgd erzeugt werden, wird in Abschnitt 5.2 erklärt. Dann untersuchen wir, wie die Generierung der Regeln in den gesamten Ablauf von sql2sttgd eingepasst wird. Zuletzt wird ein Beispiel gezeigt, das Ein- und Ausgabe für eine Anfrage mit Aggregation zeigt.

5.1 Änderungen an der internen Datenrepräsentation

In sql2sttgd wird eine Reihe von Klassen genutzt, um intern relevante Daten darzustellen. Dazu gehören zum Beispiel das Datenbankschema oder die erzeugten Abhängigkeiten. Die meisten davon werden im Package `sql2sttgd.data` gespeichert. Die Objekte sollen alle notwendigen

Informationen beinhalten und einfach zugreifbar machen. Beim Datenbankschema (in der Klasse `DBSchema`) werden die Relationen z.B. in einer HashMap mit dem Namen als Key gespeichert. So kann leicht auf die gesamte Relation zugegriffen werden, wenn nur der Name bekannt ist, was ein häufiger Anwendungsfall ist.
Diese Klassen zur Datenrepräsentation setzen zwar noch keine eigentliche Funktionalität um, sind aber essentiell, da sie beinahe im gesamten Ablauf der Transformation genutzt werden, um mit den behandelten Daten zu arbeiten. Damit bilden sie quasi das Fundament, auf dem dann die Funktionalität gebaut werden kann. Hier sollen die Änderungen beschrieben werden, die an den Daten-Klassen vorgenommen wurden, um auch Aggregationen darstellen zu können.

In sql2sttgd wird das XML-Ausgabeformat mit einer XSD beschrieben (vgl. Abschnitt zum Compositor unter 3.2.3 und [Kav+21]). Aus der XSD werden mittels JAXB entsprechende Java-Klassen generiert. Instanzen dieser JAXB-Klassen können wiederum mit JAXB als XML serialisiert werden. Da im Output-XML u.a. auch das Datenbankschema und die Abhängigkeiten vorhanden sind, könnten die JAXB-Klassen auch einfach für die interne Repräsentation genutzt werden. Für die Datenrepräsentation werden im Transformationsmodul allerdings andere bzw. erweiterte Klassen genutzt. Diese werden meistens mit dem Präfix `Internal` gekennzeichnet. Die internen Klassen werden dann zum Ende der Transformation mit einem Mapper in die JAXB-Klassen umgewandelt. So ist das gesamte Tool weniger anfällig gegenüber Änderungen an der XSD und die Klassen können besser an den Nutzen angepasst werden (wie z.B. beim Datenbankschema eben).

Um auch Aggregationen zu unterstützen, mussten einige Änderungen an den `Internal`-Klassen erfolgen und neue hinzugefügt werden. Vorher bestand der Head einer s-t tgd nur aus einem Atom, da bei Anfragen, die vorher betrachtet wurden, nur ein Result-Atom nötig war. Bei den Regeln für Aggregationen sind mitunter aber mehrere Atome im Kopf notwendig. Dafür wurde die Klasse `InternalHead` angepasst. Es wurde eine neue Klasse `InternalResultAtom` geschrieben, die einen großen Teil der vorigen Funktionalität übernimmt.

Auch der Body wurde erweitert. Vorher bestand dieser nur aus relationalen Atomen. Jetzt haben wir weitere Formen von Atomen für Zwischentabellen, Sidetables und Vergleichsatome hinzugefügt. Als Zwischentabelle bezeichnen wir hier eine intern angelegte Tabelle, die eine bereits bestehende Tabelle ergänzt. Bei den Aggregationen ist das die Relation $\tilde{R}$ (s. Tabelle 4.1). Sie enthält alle Attribute, die auch in R vorhanden sind und zusätzlich das Attribut x. Beim Datenbankschema wurde dafür die Klasse `IntermediateRelation` ergänzt und auf Ebene der Atome die Klasse `InternalIntermediateTableAtom`. So lässt sich feststellen, von welcher Tabelle die Zwischentabelle ursprünglich stammt und welche Attribute hinzugefügt wurden.
Für Sidetables wurde die Klasse `InternalSideTableAtom` ergänzt und `DBSchema` angepasst. Letztere wurde auch vorher schon benutzt, um das Datenbankschema zu speichern. Wie die Zwischentabellen sind auch Sidetables intern angelegte Tabellen, die vorher nicht in der Datenbank vorhanden waren. Sie dienen aber nicht der Ergänzung bestehender Relationen, sondern sind nur eine Berechnungshilfe. Bei den Aggregationen ist die Relation H eine Sidetable. Beide

eben genannten Arten von Atomen werden im XML auf ein `atom`-Element abgebildet. Die Unterscheidung zwischen Sidetable und Zwischentabelle dient damit nur der einfacheren Bearbeitung innerhalb des Parsers.
Alle relationalen Atome können nun außerdem negiert werden, was auch für die Aggregationen an mehreren Stellen notwendig ist. Ein entsprechendes Flag wurde in der abstrakten Klasse `InternalAtom` hinzugefügt. Von dieser Klasse erben alle Klassen für relationale Atome (so auch `InternalIntermediateTableAtom` und `InternalSideTableAtom`). So lässt sich z.B. auch ein Atom $\neg R(a_2, b_2, t_2)$ darstellen (s. Maximum in Untertabelle 4.1a).

Neben den weiteren Arten von relationalen Atomen wurden auch Vergleichsatome neu eingeführt. Die Klasse `InternalComparison` stellt Vergleiche dar, ähnlich wie sie in Abschnitt 4.5.2 für das XML beschrieben wurden. So lassen sich z.B. auch Bedingungen wie $a_1 < a_2$ umsetzen. Implementiert haben wir die Operatoren $<, \leq, \neq, \geq, >$ und $=$. Bei den Aggregatfunktionen werden solche Vergleiche u.a. für die Berechnung von Minimum und Maximum benötigt. Die Vergleichsatome bilden auch die Grundlagen, um Ungleichheiten in Selektionsprädikaten darstellen zu können (vgl. Abschnitt 3.1.3).
Wie bereits in Abschnitt 4.5.2 diskutiert, kann auf der linken Seite eines Vergleichs eine Variable stehen und auf der rechten Seite ein beliebiger Ausdruck. Ein Ausdruck kann dabei eine Konstante, eine Variable oder eine Funktion sein. Über die Funktionen lassen sich so auch verschachtelte Ausdrücke bilden, bspw. $a_1 \cdot (a_2 + 1)$ (hier sind a_1 und a_2 Variablen, 1 ist eine Konstante, $+$ und $\cdot$ sind Funktionen). Für Ausdrücke wurde das Interface `InternalExpression` geschrieben. Dieses wird dann von den Klassen `InternalMathConstant`, `InternalVariable` und `InternalFunction` implementiert, um die drei Fälle abzudecken.
Die Funktionen entsprechen hierbei den Vorschlägen zur XML-Erweiterung aus Abschnitt 4.5.3. Es wurden die Fälle $+, -, \cdot, /, \min, \max$ implementiert. Eine Funktion besteht links und rechts wieder jeweils aus einer `InternalExpression`. Wenn diese Expressions wieder Funktionen sind, können – wie eben am Beispiel – auch verschachtelte Ausdrücke formuliert werden.

Zusätzlich zu den neuen Atomen kann nun intern auch zwischen tgds und s-t tgds unterschieden werden. Bisher war diese Unterscheidung schlichtweg nicht nötig, da nur s-t tgds zur Darstellung von Anfragen genutzt wurden. Die Information wird nun als Enum-Feld in der Klasse `InternalRule` gespeichert. Um alle eben beschriebenen Änderungen abzuschließen, wurde die XSD angepasst. Sie beschreibt nun auch die Erweiterung um Vergleiche und Funktionen, die in Abschnitt 4.5 vorgestellt wurden. Damit können nun auch die aktualisierten JAXB-Klassen generiert werden. Die gesamte XSD inklusive der Änderungen ist in Anhang G zu finden. Hier z.B. ein Auszug, der die Definitionen für Expressions und Funktionen zeigt:

```
<xs:complexType name="expression">
    <xs:choice>
        <xs:element name="variable" type="variable"/>
        <xs:element name="null" type="null"/>
        <xs:element name="math-constant" type="math-constant"/>
        <xs:element name="function" type="function"/>
    </xs:choice>
</xs:complexType>
```

```
<xs:complexType name="function">
    <xs:all>
        <xs:element name="left" type="expression"/>
        <xs:element name="right" type="expression"/>
    </xs:all>
    <xs:attribute name="operator" type="function-operators" use="required"/>
</xs:complexType>
```

Zuletzt haben wir auch die Mapper angepasst. Diese befinden sich im Package `sql2sttgd.mapper` und sind verantwortlich für das Mapping der internen Daten-Klassen auf die entsprechenden JAXB-Klassen. So können intern die eigenen Daten-Klassen verwendet werden, was sie – wie oben beschrieben – flexibler für die jeweiligen Anforderungen macht. Gleichzeitig werden für den Output die JAXB-Klassen verwendet, die sich leicht als XML serialisieren lassen. Die erweiterten Daten-Klassen bieten somit eine gutes Fundament, um anschließend die eigentliche Funktionalität für die Aggregationen zu implementieren, was im nächsten Abschnitt beschrieben wird.

5.2 Erzeugung der Regeln für Aggregate

Nachdem wir gesehen haben, wie die Daten in sql2sttgd intern repräsentiert werden, widmen wir uns nun der Hauptaufgabe bei der Umsetzung der Aggregationen: die Generierung der entsprechenden Abhängigkeiten (wie in Tabelle 4.1 zu sehen). Gelöst wird diese Aufgabe größtenteils von einer Gruppe von Klassen, die wir als RuleGenerator bezeichnen werden. Sie befinden sich im Package `sql2sttgd.transformation.aggregate.rulegen` und haben die Aufgabe, die konkreten tgds und s-t tgds für die jeweilige Anfrage zu erzeugen. Von außen genutzt wird dabei eigentlich nur eine Methode, nämlich `generateRules()` in der Klasse `AggregateRuleGenerator`. Diese Methode wird in der Transformation aufgerufen, wenn die Anfrage ein alleinstehendes Aggregat enthält (vgl. Abschnitt 4.2.2). In der Methode selbst wird überprüft, welche Aggregatfunktion in der Anfrage vorkommt und abhängig davon wird ein anderer RuleGenerator aufgerufen. Identische oder ähnliche Regeln werden dabei durch die gleichen Methoden erzeugt. In Tabelle 5.1 werden nochmals die Regeln aus Tabelle 4.1 aufgegriffen, dabei wurde farblich hervorgehoben, welche RuleGenerator-Klasse für welche Regeln zuständig ist. Im Folgenden wollen wir kurz erklären, wie die Ähnlichkeiten bei der Implementation der RuleGenerators genutzt werden.

Die beiden s-t tgds für Minimum und Maximum sind beinahe identisch. Nur der Vergleichsoperator unterscheidet sich hier. Deshalb werden beide vom `MinMaxRuleGenerator` erstellt. Der übereinstimmende Part wird für beide Fälle erzeugt und je nachdem, ob Minimum oder Maximum in der Anfrage vorkommt, wird der richtige Vergleichsoperator eingesetzt.
Die zwei tgds zur Erzeugung der durchgängigen IDs bei den Aggregationen `SUM`, `AVG` und `COUNT` sind identisch, da sie nur von der betrachteten Relation R abhängen, aber nicht von der Aggregation. Aus diesem Grund werden sie in der Klasse `IdRuleGenerator` erzeugt. Bei der Behandlung der drei eben genannten Aggregate wird die zuständige Methode `generateIdRules()` zu Beginn

$$R(a_1,b_1,t_1) \wedge \neg R(a_2,b_2,t_2) \wedge b_1 < b_2 \quad \rightarrow \quad Res(b_1)$$

(a) `MAX(b)`

$$R(a_1,b_1,t_1) \wedge \neg R(a_2,b_2,t_2) \wedge b_1 > b_2 \quad \rightarrow \quad Res(b_1)$$

(b) `MIN(b)`

$$\begin{aligned}
R(a_1,b_1,t_1) \wedge \neg \tilde{R}(x_2,a_2,b_2,t_2) &\rightarrow \tilde{R}(1,a_1,b_1,t_1) \\
R(a_1,b_1,t_1) \wedge \neg \tilde{R}(x_1,a_1,b_1,t_1) \wedge & \\
\tilde{R}(x_2,a_2,b_2,t_2) \wedge \neg \tilde{R}(x_3,a_3,b_3,t_3) \wedge x_3 = x_2 + 1 &\rightarrow \tilde{R}(x_3,a_1,b_1,t_1) \\
\tilde{R}(1,a_1,b_1,t_1) &\rightarrow H(b_1,1) \\
\tilde{R}(x_1,a_1,b_1,t_1) \wedge \neg H(c_1,x_1) \wedge H(c_2,x_2) \wedge x_2 = x_1 - 1 &\rightarrow \exists C_3 : H(C_3,x_1) \wedge C_3 = b_1 + c_2 \\
H(c_1,x_1) \wedge \neg H(c_2,x_2) \wedge x_2 = x_1 + 1 &\rightarrow Res(c_1)
\end{aligned}$$

(c) `SUM(b)`

$$\begin{aligned}
R(a_1,b_1,t_1) \wedge \neg \tilde{R}(x_2,a_2,b_2,t_2) &\rightarrow \tilde{R}(1,a_1,b_1,t_1) \\
R(a_1,b_1,t_1) \wedge \neg \tilde{R}(x_1,a_1,b_1,t_1) \wedge & \\
\tilde{R}(x_2,a_2,b_2,t_2) \wedge \neg \tilde{R}(x_3,a_3,b_3,t_3) \wedge x_3 = x_2 + 1 &\rightarrow \tilde{R}(x_3,a_1,b_1,t_1) \\
\tilde{R}(1,a_1,b_1,t_1) &\rightarrow H(b_1,1) \\
\tilde{R}(x_1,a_1,b_1,t_1) \wedge \neg H(c_1,x_1) \wedge H(c_2,x_2) \wedge x_2 = x_1 - 1 &\rightarrow \exists C_3 : H(C_3,x_1) \wedge C_3 = b_1 + c_2 \\
H(c_1,x_1) \wedge \neg H(c_2,x_2) \wedge x_2 = x_1 + 1 &\rightarrow \exists C_3 : Res(C_3) \wedge C_3 = c_1 / x_1
\end{aligned}$$

(d) `AVG(b)`

$$\begin{aligned}
R(a_1,b_1,t_1) \wedge \neg \tilde{R}(x_2,a_2,b_2,t_2) &\rightarrow \tilde{R}(1,a_1,b_1,t_1) \\
R(a_1,b_1,t_1) \wedge \neg \tilde{R}(x_1,a_1,b_1,t_1) \wedge & \\
\tilde{R}(x_2,a_2,b_2,t_2) \wedge \neg \tilde{R}(x_3,a_3,b_3,t_3) \wedge x_3 = x_2 + 1 &\rightarrow \tilde{R}(x_3,a_1,b_1,t_1) \\
\tilde{R}(x_1,a_1,b_1,t_1) \wedge \neg \tilde{R}(x_2,a_2,b_2,t_2) \wedge x_2 = x_1 + 1 &\rightarrow Res(x_1)
\end{aligned}$$

(e) `COUNT(*)`

`MinMaxRuleGenerator` `IdRuleGenerator`
`SumAvgRuleGenerator` `CountRuleGenerator`

(f) Legende für zuständige Klassen

Tabelle 5.1: Aggregatfunktionen als Menge von (s-t) tgds mit zuständigen Klassen

einmal aufgerufen. Danach werden dann die Abhängigkeiten für die jeweilige Aggregatfunktion generiert.
Auch hier sind wiederum die Regeln für Summe und Durchschnitt sehr ähnlich. Nur der Kopf der abschließenden s-t tgd unterscheidet sich. Das liegt daran, dass beim Durchschnitt auch zuerst die Summe über alle Attributwerte berechnet wird und zum Schluss durch die Anzahl geteilt wird, um den Durchschnitt zu erhalten. Die Regeln werden darum gemeinsam in der Klasse `SumAvgRuleGenerator` behandelt. Es werden die Regeln erstellt und je nach Aggregation wird der entsprechende Kopf verwendet.
Damit verbleibt nur noch die letzte s-t tgd bei `COUNT`. Diese wird vom `CountRuleGenerator` generiert.

Neben den farblichen Markierungen unterscheidet sich Tabelle 5.1 von Tabelle 4.1 aus dem vorigen Kapitel auch durch einen weiteren Punkt: die Regeln wurden leicht umformuliert, damit Funktionsanwendungen eher dem entsprechen, wie sie in der Implementierung dargestellt werden. Bisher wurden Funktionsanwendungen einfach mit in die Atome geschrieben, um die Lesbarkeit zu erhöhen, z.B. $H(c_2, x_1 - 1)$ (im Body der vorletzten Regel zur Summe aus Untertabelle 4.1c). Im XML werden allerdings solche Funktionen nicht direkt in die Atome geschrieben, sondern sind nur in `comparison`-Elementen erlaubt. Die Änderungen lassen sich so einfacher in ChaTEAU ergänzen (vgl. auch Abschnitt 4.5.3). Um Funktionen in einer Abhängigkeit darstellen zu können, wird eine neue Variable an der Stelle eingeführt, wo wir vorher vereinfacht die Funktionen stehen hatten. Dann wird ein Gleichheitsatom hinzugefügt, in welchem die neue Variable der Funktion gleich gesetzt wird. Beim Beispiel von eben also $H(c_3, x_1) \wedge c_3 = b_1 + c_2$. Wenn die neue Variable dabei im Body steht, kann sie allquantifiziert werden, wie es eben bei c_3 der Fall war. Wenn die neue Variable allerdings nur im Kopf auftaucht, dann muss sie existenzquantifiziert werden. Das ist u.a. im Head der vorletzten Regel für die Summe der Fall: dort haben wir vorher verkürzt $H(b_1 + c_2, x_1)$ geschrieben; mit der neuen Variable lautet der Kopf nun $\exists C_3 : H(C_3, x_1) \wedge C_3 = b_1 + c_2$ (s. Untertabelle 5.1c).

Um die Regeln für die Aggregationen nicht durch zu lange Namen aufzublähen, wurde in dieser Arbeit eine eher mathematische Notation mit kurzen Namen für Atome und Variablen gewählt. In der Implementation ist diese Einschränkung nicht so relevant. Hier wurden Namen gewählt, die versuchen, den Zweck des jeweiligen Objekts zu beschreiben. Die Entsprechungen sollen nun kurz aufgelistet werden. In der Relation $\tilde{R}$ werden die Tupel aus R um eine durchgängige ID ergänzt. In der Implementation wird $\tilde{R}$ durch den Suffix `_sorted` gekennzeichnet. Wenn die Aggregation also auf der Tabelle `students` stattfindet, lautet der Name der Zwischentabelle `students_sorted`. Das Attribut x für die durchgängige ID bezeichnen wir als `sequential_id`. Die Tabelle H, in der Zwischenergebnisse der Berechnung gespeichert werden, wird einfach `side_table` genannt. Das Attribut c in dieser Sidetable hat in der Implementation den Namen `accumulator`, da es den bisher akkumulierten Wert der Berechnung darstellt.

Wir haben nun gesehen, wie die Abhängigkeiten für die Aggregationen erzeugt werden. Jetzt wollen wir betrachten, wie dieses Vorgehen in die gesamte Transformation eingebunden wird.

5.3 Änderungen am Algorithmus der Transformation

Im vorigen Kapitel wurde in Algorithmus 2 ein Vorschlag gegeben, wie Aggregationen in die Transformation von sql2sttgd integriert werden können. Dafür sollen jetzt die entsprechenden Anpassungen an der Klasse `Transformation` beschrieben werden. Diese Klasse ist zuständig für die Umwandlung der SQL-Anfrage in Abhängigkeiten. Die Funktionalität wurde dabei auf zwei Methoden aufgeteilt. Eine davon ist `transformAtomicQuery()`, welche nur eine atomare Anfrage (q_a) transformiert. Im Pseudocode entspricht sie grob dem Rumpf der for-Schleife von Zeile 5 bis 27. Diese Methode wird dann von `transform()` für jede atomare Anfrage in der gesamten Anfrage (Q) aufgerufen. In `transform()` werden v.a. die Vereinigungen behandelt: wenn nötig wird zuerst die Anfrage Q in atomare Anfragen aufgeteilt, jede atomare Anfrage wird transformiert, die resultierenden Abhängigkeiten werden zusammengeführt und die Result-Relationen werden verglichen. Abschließend wird die Menge aller Regeln ausgegeben. Die Methode `transformAtomicQuery()` hat dabei zwei Ausgaben: die Abhängigkeiten für die atomare Anfrage und die Ergebnisrelation für diese atomare Anfrage. Die Ergebnisrelation wird benötigt, um sie zum Datenbankschema hinzuzufügen (nur einmal für die gesamte Anfrage). Wenn die Ergebnisrelationen aller atomaren Anfragen verglichen werden, kann damit außerdem überprüft werden, ob sie überhaupt union compatible sind.

Um die Aggregatfunktionen in `transformAtomicQuery()` zu behandeln, wird – wie in Algorithmus 2 vorgeschlagen – eine zusätzliche Verzweigung eingefügt. Wenn eine atomare Anfrage eine Aggregatfunktion enthält, werden die entsprechenden Abhängigkeiten generiert und das Schema wird erweitert. Ansonsten wird die atomare Anfrage behandelt wie zuvor (Body initialisieren, Verbunde, Projektion und Selektionen einarbeiten, etc.). Wenn eine Aggregatfunktion vorhanden ist, wird auch überprüft, dass keine anderen Operationen in der atomaren Anfrage vorkommen, da vorerst nur alleinstehende Aggregatfunktionen implementiert wurden. Die Aggregatfunktion wird aus der Anfrage ausgelesen und dann werden die entsprechenden Abhängigkeiten aus Tabelle 5.1 mit den zuständigen RuleGenerators erzeugt, wie im letzten Abschnitt erklärt. Dabei werden auch die zusätzlich benötigten Relationen erstellt, damit sie zum Datenbankschema hinzugefügt werden können. Wenn die Relation $\tilde{R}$ bereits im Schema vorhanden ist, dann muss sie nicht mehr zum Schema hinzugefügt werden. Das ist dann der Fall, wenn sie bereits für eine vorige atomare Anfrage erstellt wurde, die eine Aggregatfunktion auf derselben Tabelle enthält. Auch die Regeln für die Generierung der durchgängigen IDs müssen dann nicht ein zweites Mal erstellt werden. Die Einträge aus $\tilde{R}$ können nämlich auch für weitere Aggregationen genutzt werden, da nur IDs ergänzt werden. Im Gegensatz dazu wird die Sidetable H für jede atomare Anfrage q_a neu erzeugt und dem Schema hinzugefügt, falls eine solche Sidetable benötigt wird (für Summe oder Durchschnitt). In der Transformation hat dafür jede atomare Anfrage einen eindeutigen Index, der als Suffix an den Namen der Sidetable angehängt wird, um auch sie eindeutig zu kennzeichnen. Zu einer atomaren Anfrage mit Index 1 lautet der Name der Sidetable dann `side_table_1`. So werden mögliche Konflikte bei der Berechnung der Aggregate

unterbunden. Die Ergebnisrelation wird nicht direkt in der Methode `transformAtomicQuery()` in das Schema eingefügt, wie es bei $\tilde{R}$ und H der Fall ist, sondern, wie oben beschrieben, an `transform()` zurückgegeben. Dort kann dann überprüft werden, ob die Anfragen union compatible sind.

Abschließend soll nun ein Beispiel gegeben werden, das eine Eingabe und die entsprechende Ausgabe für eine Anfrage mit Aggregat zeigt.

Beispiel 20. Wir betrachten die Anfrage aus Beispiel 13, die das Durchschnittsalter der Studierenden berechnen soll: `SELECT AVG(age) FROM Students`. Als Datenbank benutzen wir auch hier jene, die in Anhang D dargestellt wird. Das vollständige XML, das von sql2sttgd zu diesen Eingaben generiert wird, ist in Anhang F zu finden. Als Auszug wird an dieser Stelle nur die erste tgd aus dem XML gezeigt, sie lautet:

```
<tgd>
    <body>
        <atom name="students" negation="false">
            <variable name="student_id" type="V" index="1" />
            <variable name="surname" type="V" index="1" />
            <variable name="forename" type="V" index="1" />
            <variable name="course" type="V" index="1" />
            <variable name="age" type="V" index="1" />
        </atom>
        <atom name="students_sorted" negation="true">
            <variable name="sequential_id" type="V" index="2" />
            <variable name="student_id" type="V" index="2" />
            <variable name="surname" type="V" index="2" />
            <variable name="forename" type="V" index="2" />
            <variable name="course" type="V" index="2" />
            <variable name="age" type="V" index="2" />
        </atom>
    </body>
    <body-comparisons />
    <head>
        <atom name="students_sorted" negation="false">
            <variable name="student_id" type="V" index="1" />
            <variable name="surname" type="V" index="1" />
            <variable name="forename" type="V" index="1" />
            <variable name="course" type="V" index="1" />
            <variable name="age" type="V" index="1" />
            <constant name="sequential_id" value="1" />
        </atom>
    </head>
    <head-comparisons />
</tgd>
```

Das erste Atom im `body`-Element sagt aus, das ein beliebiges Tupel aus der Relation `students` gewählt wird. Das zweite Atom beschreibt, dass noch kein Tupel in der Relation `students_sorted` existieren darf, da das Atom negiert ist. Die Relation `students` entspricht hierbei R und `students_sorted` entspricht $\tilde{R}$ aus Tabelle 5.1. Das erste Tupel aus `students` wird dann in `students_sorted` eingetragen. Für die durchgängige ID (hier `sequential_id`, entspricht x) wird als Konstante der Wert 1 eingetragen. Das Element `body-comparisons` ist ebenso wie `head-comparisons` in dieser tgd leer, da keine Vergleiche benötigt werden. ■

In diesem Kapitel wurde betrachtet, an welchen Stellen das Tool sql2sttgd angepasst und erweitert werden musste, um auch Aggregationen zu unterstützen. Dabei wurden bisher nur Anfragen mit alleinstehenden Aggregatfunktionen umgesetzt. Es ist wird auch unterstützt, solche atomaren Anfragen durch Vereinigungen zu verknüpfen. Die Konzepte für Aggregatfunktionen in Verbindung mit weiteren Operationen (Verbunde, Selektion, Gruppierung) aus den Abschnitten 4.3 und 4.4 wurden nicht implementiert.

Kapitel 6

Zusammenfassung

In [Aug] wird ein Konzept vorgeschlagen, wie einfache Aggregationsfunktionen in SQL-Anfragen als Mengen von Abhängigkeiten dargestellt werden können. In der vorliegenden Bachelorarbeit konnte dieses Konzept verbessert und anschließend umgesetzt werden. Dafür wurde in Kapitel 3 ein Überblick gegeben, zu welchen Operationen bereits Konzepte bestehen, um sie als Abhängigkeiten zu formulieren. In Abschnitt 3.1.2 wurde dabei auch das Konzept aus [Aug] beschrieben. Später wurden die Regeln vereinfacht und die Anzahl der benötigten Sidetables wurde reduziert (s. Tabelle 4.1). Die angepassten Regeln wurden dann im Tool sql2sttgd implementiert, welches SQL-Anfragen in entsprechende Mengen von tgds und s-t tgds umwandelt. In sql2sttgd werden nun auch Anfragen der Form `SELECT` $f(b)$ `FROM` R unterstützt, wobei f eine der folgenden Aggregatfunktionen ist: `MAX`, `MIN`, `SUM`, `AVG` oder `COUNT` (hier der Fall `COUNT(*)`). Dabei wird auch unterstützt, atomare Anfragen dieser Form durch Vereinigungen zu verknüpfen. Die erzeugten Abhängigkeiten werden zusammen mit dem Schema und der Instanz einer gegebenen Datenbank in einer XML-Datei ausgegeben. Diese kann dann als Eingabe für das Tool ChaTEAU genutzt werden (s. Abschnitt 3.2.1). ChaTEAU führt den CHASE-Algorithmus auf der Instanz mit den erzeugten Abhängigkeiten aus, um das Anfrageergebnis zu erhalten (zum CHASE s. Abschnitt 2.3.1). Die Regeln werden dabei wie erwartet von sql2sttgd generiert. Zur Umsetzung der Aggregatfunktionen war aber eine Erweiterung des XML-Formats um Vergleiche und Funktionen notwendig (s. Abschnitt 4.5). Die entsprechenden Änderungen in ChaTEAU konnten bis zum Abgabezeitpunkt dieser Arbeit nicht fertiggestellt werden. Aus diesem Grund konnten die Ergebnisse noch nicht in der Anwendung mit ChaTEAU validiert werden.
In sql2sttgd wurde nur der eben beschriebene Fall mit alleinstehenden Aggregatfunktionen implementiert. Darüber hinaus wurde aber ein theoretisches Verfahren entwickelt, das es erlaubt, auch weitere Anfragen in Abhängigkeiten umzuwandeln: in einer Anfrage mit einer Aggregatfunktion sind damit auch Verbunde und Selektionen zugelassen (s. Abschnitt 4.3). Das Verfahren arbeitet dabei schrittweise mit weiteren Zwischentabellen: zuerst wird das Ergebnis der Verbunde und Selektionen in einer neuen Tabelle gespeichert. Für diese werden durchgängige IDs generiert,

falls sie benötigt werden. Zuletzt wird darauf dann das Aggregat berechnet. In Abschnitt 4.4 haben wir außerdem die bisher verwendeten Regeln für Aggregatfunktionen so angepasst, dass damit auch Gruppierungen als (s-t) tgds dargestellt werden können.

Neben den Aggregatfunktionen wurden auch weitere Operationen aus SQL behandelt (vgl. mit der Aufgabenstellung in Abschnitt 1.1): für äußere Verbunde und geschachtelte Anfragen wurden neue Konzepte zur Umsetzung als Abhängigkeiten entwickelt (s. Abschnitt 4.6). Für Selektionsprädikate mit Disjunktionen und Ungleichheiten wurden jeweils die Konzepte näher beschrieben (s. Abschnitt 3.1.3). Mit der Erweiterung des XML-Formats um Vergleiche aus Abschnitt 4.5.2 wurde außerdem eine Grundlage geschaffen, um die Selektionen auf Ungleichheit auch in sql2sttgd und ChaTEAU implementieren zu können. Die in der Ausschreibung genannten Operationen `SELECT *` und das kartesische Produkt wurden bereits vor Beginn dieser Arbeit in einem KSWS-Projekt implementiert [Kav+21]. Die Negation wurde aus zeitlichen Gründen nicht tiefer untersucht.

Zusammengefasst konnte also das Konzept zu Aggregatfunktionen verbessert und in sql2sttgd implementiert werden. Darüber hinaus wurden neue Konzepte beschrieben: einerseits für weitere Operationen aus SQL, andererseits auch um Aggregatfunktionen nicht nur isoliert zu betrachten, sondern auch Vorkommen in Verbindung mit Verbunden, Selektionen und Gruppierung zu erlauben.

Kapitel 7

Ausblick

Abschließend wollen wir einige potentielle weitergehende Forschungsvorhaben aufführen, welche sich an die vorliegende Bachelorarbeit anschließen könnten:

Implementation evaluieren Wie oben beschrieben wird das XML für Anfragen mit alleinstehenden Aggregatfunktionen erwartungsgemäß von sql2sttgd erzeugt. Diese XML-Dateien konnten bisher allerdings noch nicht als Eingabe für ChaTEAU getestet werden, da hier noch Änderungen ausstehen. Sobald diese abgeschlossen sind, sollte validiert werden, dass auch die korrekten Ergebnisse berechnet werden.

Außerdem wurde in Abschnitt 4.1.2 beschrieben, dass die Regeln für Minimum und Maximum vereinfacht werden konnten. Gleichzeitig wurde auch schon angedeutet, dass damit ein Verlust an Effizienz einhergehen könnte. Die Auswirkungen können noch vergleichend evaluiert werden.

Aggregationen mit Verbunden und Selektionen In Abschnitt 4.3 wurde ein Konzept vorgestellt, mit dem nicht nur alleinstehende Aggregationen zu sql2sttgd hinzugefügt werden können, sondern auch Anfragen, in denen Aggregationen in Verbindung mit Verbunden und Selektionen vorkommen. Das Konzept wurde zwar ausgearbeitet, die Implementation steht aber noch aus.

Gruppierung Wir haben auch ein Konzept ausgearbeitet, mit dem Aggregatfunktionen gruppiert angewandt werden können (s. Abschnitt 4.4). Wie eben bei den Aggregatfunktionen mit Verbunden und Selektionen aus Abschnitt 4.3, gilt auch hier, dass nur das Konzept erarbeitet, aber noch nicht implementiert wurde.

Auch die `HAVING`-Klausel aus SQL hängt eng mit der Gruppierung zusammen. Diese wird als Selektion nach Berechnung der Aggregate durchgeführt. Dafür könnte noch weiter untersucht werden, wie die `HAVING`-Klausel zu den Regeln für die Gruppierung hinzugefügt werden kann.

Weitere Fälle bei Aggregationen Bisher wurden nur Aggregatfunktionen vom Typ `MAX`, `MIN`, `SUM`, `AVG` und `COUNT` betrachtet. Hier können noch weitere Fälle untersucht werden, u.a. weitere Varianten der bisher unterstützten Funktionen (z.B. `COUNT(b)` für ein Attribut `b`, statt nur `COUNT(*)`) oder komplett neue Funktionen (z.B. boolesche oder statistische Aggregatfunktionen). Des Weiteren könnten statt einer Funktion in der `SELECT`-Klausel auch mehrere betrachtet werden, also eine Klausel der Form `SELECT` $f_1(b_1), \ldots, f_n(b_n)$. Im letzten Fall muss beachtet werden, dass keine Konflikte zwischen den verwendeten Sidetables entstehen, ähnlich wie bereits bei den Vereinigungen von Anfragen mit Aggregationen.

Äußere Verbunde Für äußere Verbunde wurde in Abschnitt 4.6.1 ein theoretisches Konzept entwickelt. Dieses könnte noch implementiert und getestet werden.

Geschachtelte Anfragen Für geschachtelte Anfragen wurde in Abschnitt 4.6.2 ebenso ein Konzept vorgeschlagen. Auch hier steht die Implementation noch aus. Hier sollte noch näher untersucht werden, wie sich innere Anfragen auswirken, für die mehrere (s-t) tgds benötigt werden (z.B. bei Selektionen mit `OR`). Eine Herausforderung könnte vor allem dann bestehen, wenn einige Atome negiert werden müssen (bei `NOT IN` und `ALL`).

Negation Auch Selektionsprädikate mit Negationen sollten noch weiter untersucht werden. Selektionen werden vermutlich vor allem dann kompliziert in der Umsetzung, wenn Konjunktionen, Disjunktionen *und* Negationen in der `WHERE`-Klausel vorkommen.

Der einfachste Fall für Aggregationen konnte in der vorliegenden Arbeit implementiert werden, nämlich Anfragen mit alleinstehenden Aggregatfunktionen. Die erzeugten Ergebnisse sollten aber noch mit dem Tool ChaTEAU validiert werden. Es wurden außerdem auch Ansätze beschrieben, wie auch weitere Fälle für die Aggregationen in (s-t) tgds dargestellt werden können, nämlich mit Verbunden, Selektionen oder Gruppierungen. Diese Konzepte könnten noch weiter verfeinert und anschließend implementiert werden.

Anhang A

Abkürzungen

ChaTEAU	**Cha**se for **T**ransforming, **E**volving, and **A**dapting databases and queries, **U**niversal approach
ProSA	**Pro**venance Management using **S**chema mappings with **A**nnotations
FD	**F**unctional **D**ependency (Funktionale Abhängigkeit)
JD	**J**oin **D**ependency (Verbundabhängigkeit)
egd	**e**quality **g**enerating **d**ependency
tgd	**t**uple **g**enerating **d**ependency
s-t tgd	**s**ource-to-**t**arget tgd

Anhang B

Verzeichnisse

Abbildungen

Tabellen

Literatur

[AH19] Tanja Auge und Andreas Heuer. „ProSA - Using the CHASE for Provenance Management“. In: *Advances in Databases and Information Systems - 23rd European Conference, ADBIS 2019, Bled, Slovenia, September 8-11, 2019, Proceedings*. Hrsg. von Tatjana Welzer, Johann Eder, Vili Podgorelec und Aida Kamisalic Latific. Bd. 11695. Lecture Notes in Computer Science. Springer, 2019, S. 357–372. DOI: 10.1007/978-3-030-28730-6_22. URL: https://doi.org/10.1007/978-3-030-28730-6%5C_22 (siehe S. 1, 10, 26).

[Aug] Tanja Auge. Notizen aus bisher nicht veröffentlichter Dissertation (siehe S. 1, 15–17, 21–24, 32, 35–37, 55, 56, 67, VI).

[Ben+17] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro und Efthymia Tsamoura. „Benchmarking the Chase“. In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*. Hrsg. von Emanuel Sallinger, Jan Van den Bussche und Floris Geerts. ACM, 2017, S. 37–52. DOI: 10.1145/3034786.3034796. URL: https://doi.org/10.1145/3034786.3034796 (siehe S. 11, 21).

[Fag+11] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa und Wang Chiew Tan. „Schema Mapping Evolution Through Composition and Inversion“. In: *Schema Matching and Mapping*. Hrsg. von Zohra Bellahsene, Angela Bonifati und Erhard Rahm. Data-Centric Systems and Applications. Springer, 2011, S. 191–222. DOI: 10.1007/978-3-642-16518-4_7. URL: https://doi.org/10.1007/978-3-642-16518-4%5C_7 (siehe S. 11, 21).

[GMS12] Sergio Greco, Cristian Molinaro und Francesca Spezzano. *Incomplete Data and Data Dependencies in Relational Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012. DOI: 10.2200/S00435ED1V01Y201207DTM029. URL: https://doi.org/10.2200/S00435ED1V01Y201207DTM029 (siehe S. 1, 11, 21).

[HSS18] Andreas Heuer, Gunter Saake und Kai-Uwe Sattler. *Datenbanken - Konzepte und Sprachen*. 6. Aufl. MITP, 2018. ISBN: 978-3-9584577-6-8. URL: https://mitp.de/IT-WEB/Datenbanken/Datenbanken-Konzepte-und-Sprachen-oxid.html (siehe S. 4, 9).

[Kav+21] Ivo Kavisanczki, Tobias Rudolph, Tom Siegl und Marian Zuska. *Projektdokumentation sql2sttgd*. 30. Sep. 2021. URL: http://eprints.dbis.informatik.uni-rostock.de/1051/ (besucht am 15. 12. 2021) (siehe S. 21, 22, 27, 29, 37, 38, 59, 68, VI).

[Pos21] *PostgreSQL documentation*. Technische Dokumentation. Version 14.1. Nov. 2021. URL: https://www.postgresql.org/docs/14/index.html (besucht am 26. 11. 2021) (siehe S. 4, 7, 23, VI).

[Pro] *ProSA: Provenance Management durch Schema-Abbildungen und Annotationen*. Projektbeschreibung auf der Webseite des Lehrstuhls. URL: https://dbis.informatik.uni-rostock.de/forschung/aktuelle-projekte/prosa/ (besucht am 30. 12. 2021) (siehe S. 26).

[Sch21] Nic Scharlau. *Der CHASE-Algorithmus: Ein Überblick*. Gebietsseminar Informationssysteme. 6. Apr. 2021. URL: http://eprints.dbis.informatik.uni-rostock.de/id/eprint/1056 (besucht am 06. 12. 2021) (siehe S. 11, 13).

[Spo] Dennis Spolwind. „Inverse Anfragen in ProSA“. Notizen aus bisher nicht veröffentlichter Masterarbeit (siehe S. 56).

Anhang C

„Datenträger“

Zugriff auf Literatur

Im Stud.IP wurde die Literatur in der Veranstaltung „BA: Erweiterung des ProSA-Parsers“ im Ordner „Literatur“ bereitgestellt. Dazu gehört der Stand der PostgreSQL-Dokumentation [Pos21] als PDF (wurde für diese Arbeit auf `www.postgresql.org/docs` abgerufen) ebenso wie die Auszüge aus [Aug], auf welche wir häufig verwiesen haben.

Zugriff zur Implementation

Die Implementation wurde im GitLab-Projekt zu ProSA verwaltet. Dort kann auch auf die Implementation zugegriffen werden. Die Erweiterung des XML-Formats für ChaTEAU geschah dabei auf dem Branch `ivo_xml-functions-and-comparisons`. Die Änderungen sind gut im entsprechenden Merge Request unter der Ansicht „Changes“ zu sehen. Die restliche Implementierung zu den alleinstehenden Aggregatfunktionen erfolgte auf dem Branch `ivo_singleton-aggregates`. Auch hier sind die Änderungen wieder gut im dazugehörigen Merge Request zu sehen.

Verwendete Libraries

Für die Implementation wurden verschiedene Java-Libraries verwendet, die auch bereits vorher in sql2sttgd benutzt wurden. In Tabelle C.1 werden alle Libraries aufgelistet, die in sql2sttgd verwendet werden. Die Tabelle wurde aus [Kav+21] übernommen und um Versionen ergänzt.

Bibliothek	**Aufgabe**	**Lizenz**	**Version**
Jakarta XML Binding (JAXB) + XJC	Erzeugung von Java-Objekten aus einer XML Schema Definition (XSD), Serialisierung von Java-Objekten in XML	EDL v1.0	3.0.1
JSQLParser	Parsen von SQL, Überführung in einen Syntaxbaum	Apache License v2.0	4.0
PostgreSQL JDBC Driver	Kommunikation mit PostgreSQL-Datenbanken über das JDBC-Interface	BSD 2-Clause “Simplified”	42.2.19
Apache Commons Lang	Sammlung verschiedener häufig verwendeter Hilfsfunktionen für Java	Apache License v2.0	3.12.0
Google Guava	Sammlung verschiedener häufig verwendeter Hilfsfunktionen (ähnlich wie Apache Commons Lang), für uns insbesondere interessant wegen einer Graphen-Implementation	Apache License v2.0	30.1.1-jre
Lombok	Reduzierung von Boilerplate-Code wie z.B. triviale Getter / Setter oder Non-Null-Checks	MIT License	1.18.20

Tabelle C.1: Verwendete Bibliotheken in sql2sttgd

Anhang D

Beispiel-Datenbank

In den folgenden Tabellen sind die Relationen der Datenbank zu sehen, die für einige Beispiele gebraucht wurde. Die Ausdrücke in Klammern geben dabei die Abkürzung der Namen an.

student_id (sid)	surname (sn)	forename (fn)	course (co)	age (a)
1	Miller	Mark	Electrical Engineering	18
2	Lewis	Linda	Computer Science	20
3	Smith	Steven	Mathematics	21
4	Smith	Sarah	Mathematics	23
5	Brown	Bob	Computer Science	22
7	Nelson	Nancy	Mathematics	25

Tabelle D.1: Die vollständige Relation *Students* (*St*)

module_id (mid)	module_name (mn)	points (po)
1	Embedded Systems	6
2	Functional Programming	3
3	Linear Algebra	9
4	Databases 1	6
5	Cryptography	3

Tabelle D.2: Die vollständige Relation *Modules* (*Mo*)

module_id (mid)	student_id (sid)	semester (sem)	grade (gr)
1	1	5	2.0
2	2	6	1.3
3	3	7	2.7
3	4	7	3.7
4	1	5	4.0
4	5	3	1.0

Tabelle D.3: Die vollständige Relation *Grades* (*Gr*)

Anhang E

Darstellbarkeit von SQL-Operationen

Die folgenden Tabellen geben einen Überblick, wie verschiedene SQL-Operationen als Mengen von Abhängigkeiten dargestellt werden können (vgl. Abschnitt 4.6.3).

Operation	**Anfrage in SQL**	**Abhängigkeiten**
Proj. allgemein	`SELECT` $a_{i_1},\ldots,a_{i_k}$ `FROM` R	$R(a_1,\ldots,a_n) \rightarrow Res(a_{i_1},\ldots,a_{i_k})$
`SELECT *`	`SELECT * FROM` R	$R(a_1,\ldots,a_n) \rightarrow Res(a_1,\ldots,a_n)$
Umbenennung	`SELECT` $a_1,\ldots,a_n,d$ `AS` x `FROM` R	$R(a_1,\ldots,a_n,d) \wedge d = x$ $\rightarrow Res(a_1,\ldots,a_n,x)$

Tabelle E.1: Darstellung von Projektion und Umbenennung als Mengen von Abhängigkeiten

Operation	**Anfrage in SQL**	**Abhängigkeiten**
natürlicher Verb.	`SELECT * FROM` R `NATURAL JOIN` S	$R(a_1,\ldots,a_n,d) \wedge S(b_1,\ldots,b_m,d)$ $\rightarrow Res(a_1,\ldots,a_n,b_1,\ldots,b_m,d)$
Kreuzprodukt	`SELECT * FROM` R `CROSS JOIN` S	$R(a_1,\ldots,a_n) \wedge S(b_1,\ldots,b_m)$ $\rightarrow Res(a_1,\ldots,a_n,b_1,\ldots,b_m)$
`JOIN ON`	`SELECT * FROM` R `JOIN` S `ON` $R.d = S.e$	$R(a_1,\ldots,a_n,d) \wedge S(b_1,\ldots,b_m,e)$ $\wedge d = e \rightarrow Res(a_1,\ldots,a_n,b_1,\ldots,b_m,e)$
`USING`	`SELECT * FROM` R `JOIN` S `USING` (d_1,d_2)	$R(a_1,\ldots,a_n,d_1,d_2) \wedge S(b_1,\ldots,b_m,d_1,d_2)$ $\rightarrow Res(a_1,\ldots,a_n,b_1,\ldots,b_m,d_1,d_2)$
`LEFT JOIN`	`SELECT * FROM` R `LEFT JOIN` S `ON` $R.d = S.e$	$R(a_1,\ldots,a_n,d) \wedge S(b_1,\ldots,b_m,e)$ $\wedge d = e \rightarrow Res(a_1,\ldots,a_n,b_1,\ldots,b_m,e)$, $R(a_1,\ldots,a_n,d) \wedge \neg S(b_1,\ldots,b_m,d)$ $\rightarrow \exists B_1,\ldots,B_n : Res(a_1,\ldots,a_n,B_1,\ldots,B_m,d)$
`RIGHT JOIN`	`SELECT * FROM` R `RIGHT JOIN` S `ON` $R.d = S.e$	$R(a_1,\ldots,a_n,d) \wedge S(b_1,\ldots,b_m,e)$ $\wedge d = e \rightarrow Res(a_1,\ldots,a_n,b_1,\ldots,b_m,e)$, $S(b_1,\ldots,b_m,e) \wedge \neg R(a_1,\ldots,a_n,e)$ $\rightarrow \exists A_1,\ldots,A_n : Res(A_1,\ldots,A_n,b_1,\ldots,b_m,e)$
`FULL OUTER JOIN`	`SELECT * FROM` R `FULL OUTER JOIN` S `ON` $R.d = S.e$	$R(a_1,\ldots,a_n,d) \wedge S(b_1,\ldots,b_m,e)$ $\wedge d = e \rightarrow Res(a_1,\ldots,a_n,b_1,\ldots,b_m,e)$, $R(a_1,\ldots,a_n,d) \wedge \neg S(b_1,\ldots,b_m,d)$ $\rightarrow \exists B_1,\ldots,B_n : Res(a_1,\ldots,a_n,B_1,\ldots,B_m,d)$, $S(b_1,\ldots,b_n,e) \wedge \neg R(a_1,\ldots,a_n,e)$ $\rightarrow \exists A_1,\ldots,A_n : Res(A_1,\ldots,A_n,b_1,\ldots,b_m,e)$

Tabelle E.2: Darstellung von Verbunden als Mengen von Abhängigkeiten

Operation	**Anfrage in SQL**	**Abhängigkeiten**
Vgl. mit Konstante	SELECT * FROM R WHERE $a_i\,\theta\,c$	$R(a_1,\ldots,a_n) \wedge a_i\,\theta\,c$ $\rightarrow Res(a_1,\ldots,a_n)$
Vgl. mit Attribut	SELECT $a_{i_1},\ldots,a_{i_k}$ FROM R,S WHERE $a_i\theta b_j$	$R(a_1,\ldots,a_n) \wedge S(b_1,\ldots,b_n) \wedge a_i\,\theta\,b_j$ $\rightarrow Res(a_{i_1},\ldots,a_{i_k})$
AND	SELECT * FROM R WHERE $a_i\theta c_1$ AND $a_j\theta c_2$	$R(a_1,\ldots,a_n) \wedge a_i\,\theta\,c_1 \wedge a_j\,\theta\,c_2$ $\rightarrow Res(a_1,\ldots,a_n)$
OR	SELECT * FROM R WHERE $a_i\theta c_1$ OR $a_j\theta c_2$	$R(a_1,\ldots,a_n) \wedge a_i\,\theta\,c_1$ $\rightarrow Res(a_1,\ldots,a_n)$, $R(a_1,\ldots,a_n) \wedge a_j\,\theta\,c_2$ $\rightarrow Res(a_1,\ldots,a_n)$
NOT	SELECT * FROM R WHERE NOT $a_i\,\theta\,c$	$R(a_1,\ldots,a_n) \wedge \neg(a_i\,\theta\,c)$ $\rightarrow Res(a_1,\ldots,a_n)$
BETWEEN	SELECT * FROM R WHERE a_i BETWEEN c_1 AND c_2	$R(a_1,\ldots,a_n) \wedge a_i \geq c_1 \wedge a_i \leq c_2$ $\rightarrow Res(a_1,\ldots,a_n)$

Tabelle E.3: Darstellung von Selektionen als Mengen von Abhängigkeiten

Operation	**Anfrage in SQL**	**Abhängigkeiten**
IN	SELECT * FROM R WHERE a_i IN (SELECT b_j FROM S WHERE $b_k\,\theta\,c$)	$S(b_1,\ldots,b_n) \wedge b_k\,\theta\,c \rightarrow Res'(b_j)$, $R(a_1,\ldots,a_n) \wedge Res'(b_j) \wedge a_i = b_j$ $\rightarrow Res(a_1,\ldots,a_n)$
ANY	SELECT * FROM R WHERE $a_i\,\theta$ ANY (SELECT b_j FROM S WHERE $b_k\,\theta\,c$)	$S(b_1,\ldots,b_n) \wedge b_k\,\theta\,c \rightarrow Res'(b_j)$, $R(a_1,\ldots,a_n) \wedge Res'(b_j) \wedge a_i\,\theta\,b_j$ $\rightarrow Res(a_1,\ldots,a_n)$
NOT IN	SELECT * FROM R WHERE a_i NOT IN (SELECT b_j FROM S WHERE $b_k\,\theta\,c$)	$S(b_1,\ldots,b_n) \wedge b_k\,\theta\,c \rightarrow Res'(b_j)$, $R(a_1,\ldots,a_n) \wedge \neg Res'(a_i)$ $\rightarrow Res(a_1,\ldots,a_n)$
EXISTS	SELECT * FROM R WHERE EXISTS (SELECT * FROM S WHERE $R.a_i\,\theta\,S.b_k$)	$R(a_1,...,a_n) \wedge S(b_1,...,b_n)$ $\wedge a_i\,\theta\,b_k \rightarrow Res(a_1,...,a_n)$

Tabelle E.4: Darstellung von geschachtelten Anfragen als Mengen von Abhängigkeiten

Operation	Anfrage in SQL	Abhängigkeiten
IN	SELECT * FROM R WHERE a_i IN $(x_1,...,x_n)$	$R(a_1,...,a_n) \wedge X(a_i)$ $\rightarrow Res(a_1,...,a_n)$
ANY	SELECT * FROM R WHERE $a_i\theta$ ANY $(x_1,...,x_n)$	$R(a_1,...,a_n) \wedge X(x_i) \wedge a_i\theta x_i$ $\rightarrow Res(a_1,...,a_n)$
NOT IN	SELECT * FROM R WHERE a_i NOT IN $(x_1,...,x_n)$	$R(a_1,...,a_n) \wedge \neg X(a_i)$ $\rightarrow Res(a_1,...,a_n)$

Tabelle E.5: Darstellung von Selektionen mit Arrays als Mengen von Abhängigkeiten

Operation	Anfrage in SQL	Abhängigkeiten
UNION	SELECT $a_1,\ldots,a_n$ FROM R UNION SELECT $a_1,\ldots,a_n$ FROM S	$R(a_1,\ldots,a_n) \rightarrow Res(a_1,\ldots,a_n)$, $S(a_1,\ldots,a_n) \rightarrow Res(a_1,\ldots,a_n)$
EXCEPT	SELECT $a_1,\ldots,a_n$ FROM R EXCEPT SELECT $a_1,\ldots,a_n$ FROM S	$R(a_1,\ldots,a_n) \wedge \neg S(a_1,\ldots,a_n)$ $\rightarrow Res(a_1,\ldots,a_n)$
INTERSECT	SELECT $a_1,\ldots,a_n$ FROM R INTERSECT SELECT $a_1,\ldots,a_n$ FROM S	$R(a_1,\ldots,a_n) \wedge S(a_1,\ldots,a_n)$ $\rightarrow Res(a_1,\ldots,a_n)$

Tabelle E.6: Darstellung von Mengenoperationen als Mengen von Abhängigkeiten

Anhang F

Beispiel-Output

Der gesamte Inhalt der Ausgabe-XML zur Anfrage aus Beispiel 20:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<input>
    <schema>
        <relations>
            <relation name="students" tag="S">
                <attribute name="student_id" type="int" />
                <attribute name="surname" type="string" />
                <attribute name="forename" type="string" />
                <attribute name="course" type="string" />
                <attribute name="age" type="int" />
            </relation>
            <relation name="students_sorted" tag="S">
                <attribute name="sequential_id" type="int" />
                <attribute name="student_id" type="int" />
                <attribute name="surname" type="string" />
                <attribute name="forename" type="string" />
                <attribute name="course" type="string" />
                <attribute name="age" type="int" />
            </relation>
            <relation name="side_table_0" tag="S">
                <attribute name="accumulator" type="int" />
                <attribute name="sequential_id" type="int" />
            </relation>
            <relation name="Result" tag="T">
                <attribute name="avg" type="double" />
            </relation>
        </relations>
        <dependencies>
            <tgd>
                <body>
                    <atom name="students" negation="false">
                        <variable name="student_id" type="V" index="1" />
                        <variable name="surname" type="V" index="1" />
                        <variable name="forename" type="V" index="1" />
                        <variable name="course" type="V" index="1" />
                        <variable name="age" type="V" index="1" />
                    </atom>
                    <atom name="students_sorted" negation="true">
                        <variable name="sequential_id" type="V" index="2" />
                        <variable name="student_id" type="V" index="2" />
                        <variable name="surname" type="V" index="2" />
                        <variable name="forename" type="V" index="2" />
                        <variable name="course" type="V" index="2" />
                        <variable name="age" type="V" index="2" />
                    </atom>
                </body>
```

```
        <body-comparisons />
        <head>
            <atom name="students_sorted" negation="false">
                <variable name="student_id" type="V" index="1" />
                <variable name="surname" type="V" index="1" />
                <variable name="forename" type="V" index="1" />
                <variable name="course" type="V" index="1" />
                <variable name="age" type="V" index="1" />
                <constant name="sequential_id" value="1" />
            </atom>
        </head>
        <head-comparisons />
    </tgd>
    <tgd>
        <body>
            <atom name="students" negation="false">
                <variable name="student_id" type="V" index="1" />
                <variable name="surname" type="V" index="1" />
                <variable name="forename" type="V" index="1" />
                <variable name="course" type="V" index="1" />
                <variable name="age" type="V" index="1" />
            </atom>
            <atom name="students_sorted" negation="true">
                <variable name="sequential_id" type="V" index="1" />
                <variable name="student_id" type="V" index="1" />
                <variable name="surname" type="V" index="1" />
                <variable name="forename" type="V" index="1" />
                <variable name="course" type="V" index="1" />
                <variable name="age" type="V" index="1" />
            </atom>
            <atom name="students_sorted" negation="false">
                <variable name="sequential_id" type="V" index="2" />
                <variable name="student_id" type="V" index="2" />
                <variable name="surname" type="V" index="2" />
                <variable name="forename" type="V" index="2" />
                <variable name="course" type="V" index="2" />
                <variable name="age" type="V" index="2" />
            </atom>
            <atom name="students_sorted" negation="true">
                <variable name="sequential_id" type="V" index="3" />
                <variable name="student_id" type="V" index="3" />
                <variable name="surname" type="V" index="3" />
                <variable name="forename" type="V" index="3" />
                <variable name="course" type="V" index="3" />
                <variable name="age" type="V" index="3" />
            </atom>
        </body>
        <body-comparisons>
            <comparison operator="equal">
                <left>
                    <variable name="sequential_id" type="V" index="3" />
                </left>
                <right>
                    <function operator="addition">
                        <left>
                            <variable name="sequential_id" type="V" index="2" />
                        </left>
                        <right>
                            <math-constant value="1" />
                        </right>
                    </function>
                </right>
            </comparison>
        </body-comparisons>
        <head>
            <atom name="students_sorted" negation="false">
                <variable name="sequential_id" type="V" index="3" />
                <variable name="student_id" type="V" index="1" />
                <variable name="surname" type="V" index="1" />
                <variable name="forename" type="V" index="1" />
                <variable name="course" type="V" index="1" />
                <variable name="age" type="V" index="1" />
            </atom>
        </head>
```

```
    <head-comparisons />
</tgd>
<tgd>
    <body>
        <atom name="students_sorted" negation="false">
            <variable name="student_id" type="V" index="1" />
            <variable name="surname" type="V" index="1" />
            <variable name="forename" type="V" index="1" />
            <variable name="course" type="V" index="1" />
            <variable name="age" type="V" index="1" />
            <constant name="sequential_id" value="1" />
        </atom>
    </body>
    <body-comparisons />
    <head>
        <atom name="side_table_0" negation="false">
            <variable name="age" type="V" index="1" />
            <constant name="sequential_id" value="1" />
        </atom>
    </head>
    <head-comparisons />
</tgd>
<tgd>
    <body>
        <atom name="students_sorted" negation="false">
            <variable name="sequential_id" type="V" index="1" />
            <variable name="student_id" type="V" index="1" />
            <variable name="surname" type="V" index="1" />
            <variable name="forename" type="V" index="1" />
            <variable name="course" type="V" index="1" />
            <variable name="age" type="V" index="1" />
        </atom>
        <atom name="side_table_0" negation="true">
            <variable name="accumulator" type="V" index="1" />
            <variable name="sequential_id" type="V" index="1" />
        </atom>
        <atom name="side_table_0" negation="false">
            <variable name="accumulator" type="V" index="2" />
            <variable name="sequential_id" type="V" index="2" />
        </atom>
    </body>
    <body-comparisons>
        <comparison operator="equal">
            <left>
                <variable name="sequential_id" type="V" index="2" />
            </left>
            <right>
                <function operator="subtraction">
                    <left>
                        <variable name="sequential_id" type="V" index="1" />
                    </left>
                    <right>
                        <math-constant value="1" />
                    </right>
                </function>
            </right>
        </comparison>
    </body-comparisons>
    <head>
        <atom name="side_table_0" negation="false">
            <variable name="accumulator" type="V" index="3" />
            <variable name="sequential_id" type="V" index="1" />
        </atom>
    </head>
    <head-comparisons>
        <comparison operator="equal">
            <left>
                <variable name="accumulator" type="V" index="3" />
            </left>
            <right>
                <function operator="addition">
                    <left>
                        <variable name="age" type="V" index="1" />
                    </left>
```

```
                        <right>
                            <variable name="accumulator" type="V" index="2" />
                        </right>
                    </function>
                </right>
            </comparison>
        </head-comparisons>
    </tgd>
    <sttgd>
        <body>
            <atom name="side_table_0" negation="false">
                <variable name="accumulator" type="V" index="1" />
                <variable name="sequential_id" type="V" index="1" />
            </atom>
            <atom name="side_table_0" negation="true">
                <variable name="accumulator" type="V" index="2" />
                <variable name="sequential_id" type="V" index="2" />
            </atom>
        </body>
        <body-comparisons>
            <comparison operator="equal">
                <left>
                    <variable name="sequential_id" type="V" index="2" />
                </left>
                <right>
                    <function operator="addition">
                        <left>
                            <variable name="sequential_id" type="V" index="1" />
                        </left>
                        <right>
                            <math-constant value="1" />
                        </right>
                    </function>
                </right>
            </comparison>
        </body-comparisons>
        <head>
            <atom name="Result" negation="false">
                <variable name="avg" type="V" index="1" />
            </atom>
        </head>
        <head-comparisons>
            <comparison operator="equal">
                <left>
                    <variable name="avg" type="V" index="1" />
                </left>
                <right>
                    <function operator="division">
                        <left>
                            <variable name="accumulator" type="V" index="1" />
                        </left>
                        <right>
                            <variable name="sequential_id" type="V" index="1" />
                        </right>
                    </function>
                </right>
            </comparison>
        </head-comparisons>
    </sttgd>
</dependencies>
</schema>
<instance>
    <atom name="students">
        <constant value="5" />
        <constant value="Brown" />
        <constant value="Bob" />
        <constant value="Computer Science" />
        <constant value="22" />
    </atom>
    <atom name="students">
        <constant value="1" />
        <constant value="Miller" />
        <constant value="Mark" />
        <constant value="Electrical Engineering" />
```

```
            <constant value="18" />
        </atom>
        <atom name="students">
            <constant value="2" />
            <constant value="Lewis" />
            <constant value="Linda" />
            <constant value="Computer Science" />
            <constant value="20" />
        </atom>
        <atom name="students">
            <constant value="6" />
            <constant value="Nelson" />
            <constant value="Nancy" />
            <constant value="Mathematics" />
            <constant value="25" />
        </atom>
        <atom name="students">
            <constant value="3" />
            <constant value="Smith" />
            <constant value="Steven" />
            <constant value="Mathematics" />
            <constant value="21" />
        </atom>
        <atom name="students">
            <constant value="4" />
            <constant value="Smith" />
            <constant value="Sarah" />
            <constant value="Mathematics" />
            <constant value="23" />
        </atom>
    </instance>
</input>
```

Anhang G

XSD für ChaTEAU

Die gesamte XML Schema Definition (XSD), welche das Eingabeformat für ChaTEAU definiert:

```
<?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>
        <xs:element name="egd" type="rule" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="tgd" type="rule" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="sttgd" type="rule" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="rule">
    <xs:sequence>
        <xs:element name="body" type="atoms"/>
        <xs:element name="body-comparisons" type="comparisons"/>
        <xs:element name="head" type="atoms"/>
        <xs:element name="head-comparisons" type="comparisons"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="atoms">
    <xs:sequence>
        <xs:element name="atom" type="atom" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="atom">
    <xs:sequence>
        <xs:element name="variable" type="variable" maxOccurs="unbounded" minOccurs="0"/>
        <xs:element name="constant" type="constant" maxOccurs="unbounded" minOccurs="0"/>
        <xs:element name="null" type="null" maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" default=""/>
    <xs:attribute name="negation" type="xs:boolean" default="false"/>
</xs:complexType>

<xs:complexType name="variable">
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="type" type="quantifier"/>
    <xs:attribute name="index" type="xs:int" default="0"/>
</xs:complexType>

<xs:simpleType name="quantifier">
    <xs:restriction base="xs:string">
        <xs:enumeration value="V"/> <!-- all -->
        <xs:enumeration value="E"/> <!-- exists -->
    </xs:restriction>
</xs:simpleType>

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

<xs:complexType name="null">
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="index" type="xs:int" default="0"/>
</xs:complexType>

<xs:complexType name="comparisons">
    <xs:sequence>
        <xs:element name="comparison" type="comparison" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="comparison">
    <xs:all>
        <xs:element name="left">
            <xs:complexType>
```

```
                <xs:all>
                    <xs:element name="variable" type="variable"/>
                </xs:all>
            </xs:complexType>
        </xs:element>
        <xs:element name="right" type="expression"/>
    </xs:all>
    <xs:attribute name="operator" type="comparison-operators" use="required"/>
</xs:complexType>

<xs:simpleType name="comparison-operators">
    <xs:restriction base="xs:string">
        <xs:enumeration value="greater"/>
        <xs:enumeration value="less"/>
        <xs:enumeration value="greater-equal"/>
        <xs:enumeration value="less-equal"/>
        <xs:enumeration value="not-equal"/>
        <xs:enumeration value="equal"/>
    </xs:restriction>
</xs:simpleType>

<xs:complexType name="expression">
    <xs:choice>
        <xs:element name="variable" type="variable"/>
        <xs:element name="null" type="null"/>
        <xs:element name="math-constant" type="math-constant"/>
        <xs:element name="function" type="function"/>
    </xs:choice>
</xs:complexType>

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

<xs:complexType name="function">
    <xs:all>
        <xs:element name="left" type="expression"/>
        <xs:element name="right" type="expression"/>
    </xs:all>
    <xs:attribute name="operator" type="function-operators" use="required"/>
</xs:complexType>

<xs:simpleType name="function-operators">
    <xs:restriction base="xs:string">
        <xs:enumeration value="addition"/>
        <xs:enumeration value="subtraction"/>
        <xs:enumeration value="multiplication"/>
        <xs:enumeration value="division"/>
        <xs:enumeration value="minimum"/>
        <xs:enumeration value="maximum"/>
    </xs:restriction>
</xs:simpleType>

<xs:complexType name="instance">
    <xs:sequence>
        <xs:element name="atom" type="instance-atom" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="instance-atom">
    <xs:sequence>
        <xs:element name="constant" type="constant" maxOccurs="unbounded" minOccurs="0"/>
        <xs:element name="null" type="null" maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" default=""/>
</xs:complexType>

</xs:schema>
```

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Die Arbeit ist noch nicht veröffentlicht und ist in ähnlicher oder gleicher Weise noch nicht als Prüfungsleistung zur Anerkennung oder Bewertung vorgelegt worden.

Rostock, den 01. März 2022