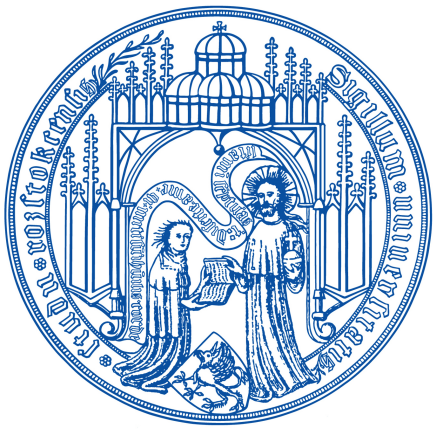
Eignung von funktionalen Abhängigkeiten und Inklusionsabhängigkeiten zur Reformulierung von Anfragen unter Datenschutzaspekten

Bachelorarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik



vorgelegt von: Max Tilman Kaseler
Matrikelnummer: 216204297
geboren am: 19. November 1996 in Lübeck
Erstgutachter: Prof. Dr. rer. nat. habil. Andreas Heuer
Zweitgutachter: Dr.-Ing. Holger Meyer
Betreuer: M.Sc. Hannes Grunert

Abgabedatum: 3. April 2021

Inhaltsverzeichnis

Kapitel 1

Einleitung

Viele smarte Geräte sammeln personenbezogene Daten, um diese später auszuwerten. Die Daten werden hierzu meist vollständig in die Cloud übertragen und erst in dieser ausgewertet. Dieses Vorgehen führt dazu, dass mehr Daten als benötigt übertragen werden. Die zusätzlichen Daten bieten die Möglichkeit, Auswertungen durchzuführen, welche bei der Erstellung des Systems nicht beabsichtigt wurden. Um dem entgegen zu wirken, kann ein System möglichst datensparsam aufgebaut werden. In einem datensparsamen System werden nur die Daten übertragen, welche für die vorhergesehenen Auswertungen benötigt werden. Hierdurch sind weiterhin alle gewollten Auswertungen möglich, aber die möglichen ungewollte Auswertungen werden eingeschränkt.
Die wenigsten Daten werden übertragen, wenn die vollständige Auswertung der Daten bereits bei dem Benutzer durchgeführt wird. Dies kann aus mehreren Gründen nicht umsetzbar sein, z.B. weil beim Nutzer noch nicht alle Daten vorliegen oder weil die Hardware des Nutzers die Auswertung nicht oder nur sehr langsam durchführen kann. Auch wenn nicht die vollständige Anfrage ausgeführt werden kann, ist es möglich, einen Teil der Anfrage bereits auf den Geräten des Nutzers durchzuführen. Durch das Reformulieren von Anfragen ist es möglich, die Umsetzung der Anfrage zu beeinflussen, ohne das Ergebnis zu verändern. Bei der Reformulierung werden meist algebraische Äquivalenzen, welche kein Wissen über den Datensatz voraus setzen, verwendet.
Ziel dieser Arbeit ist es, Umformungen vorzustellen, welche die in einem Datensatz vorhanden funktionalen und Inklusionsabhängigkeiten ausnutzen, um weitere Umformungen zu ermöglichen. Die vorgestellten Umformungen sollen danach, in einem im PArADISE-Projekt entstandenen Rewriter, implementiert werden.

Die Einleitung dieser Arbeit beinhaltet den Abschnitt 1.1, in welchem das PArADISE-Projekt vorgestellt wird. Außerdem werden im Abschnitt 1.2 die in dieser Arbeit verwendeten Begriffe erklärt. In dem Abschnitt 1.3 wird der weitere Aufbau der Arbeit beschrieben.

1.1 Das PArADISE-Projekt

Das PArADISE-Projekt[1] ist ein Datenschutz-Framework, welches an der Universität Rostock entwickelt wird. Da diese Arbeit das Framework erweitert, wird dieses kurz vorgestellt. PArADISE ist eine Abkürzung für **P**rivacy **A**ware **A**ssistive **D**istributed **I**nformation **S**ystem **E**nvironment. Das Framework soll helfen, smarte oder assistive Umgebungen datenschutzfreundlich und datensparsam zu gestalten. In diesen Umgebungen werden Daten meist ungefiltert auf einen Server geladen und dort ausgewertet. Dies

[1] `https://dbis.informatik.uni-rostock.de/forschung/aktuelle-projekte/paradise/`, zuletzt aufgerufen am 1.4.2021 um 16:20

führt dazu, dass für den Anwendungszweck unwichtige, personenbezogene Daten übertragen und gesammelt werden. Dies widerspricht der EU-Verordnung [Cou16], welche festlegt das personenbezogene Daten nur zweckgebunden und im notwendigen Maß erhoben werden dürfen.
Das PArADISE-Framework realisiert unter anderem diese Anforderung durch eine Anfragemodifizierung. Mit Hilfe dieser wird die Anfrage so verteilt, dass diese möglichst nahe am Sensor einer smarten Umgebung ausgeführt werden kann. Welche Operationen ausführbar sind, hängt hierbei von den vorhandenen Daten und der Hardware ab. In der Abbildung 1.1 wird dies beispielhaft dargestellt.
Diese Arbeit beschäftigt sich mit der Anfragemodifizierung im PArADISE-Projekt. Das Ziel ist, diese um weitere mögliche Modifikationen zu erweitern, wozu funktionale und Inklusionsabhängigkeiten verwendet werden.

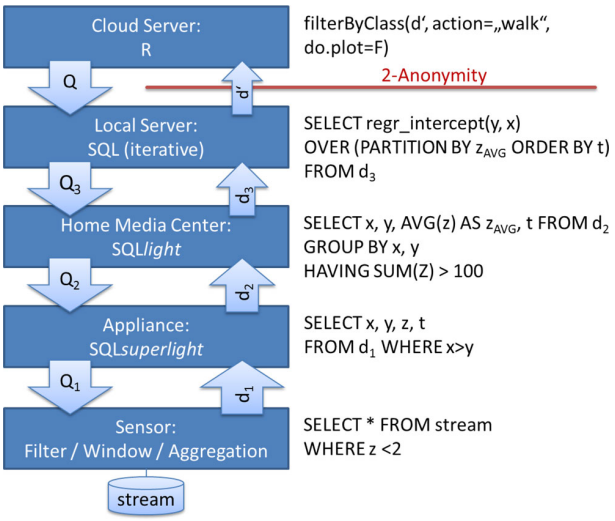


Abbildung 1.1: Beispiel für eine Anfrageumschreibung in PArADISE [GH16]

1.2 Erläuterungen

In diesem Abschnitt werden Begriffe erklärt, welche in dieser Arbeit verwendet werden.

Relationenschema und Relation nach [HSS18] Sei U eine endliche Menge, welche alle Attribute umfasst, so ist das Relationenschema R durch den Ausdruck $R \subseteq U$ definiert. Ein Relationenschema hat die Form $R = \{A_1, ...A_n\}$ und beschreibt, welche Attribute sich später in der Relation befinden. Die in Tabelle 1.1 (a) dargestellte Relation hat das Schema $a = \{A, B\}$.
Eine Relation r über R wird mit $r(R)$ bezeichnet und ist eine endliche Menge von Tupeln. Pro Tupel ist jedem Attribut ein Attributwert zugewiesen.

Funktionale Abhängigkeiten nach [Tür99] Eine funktionale Abhängigkeit (FD, eng. functional dependency) über eine Menge von Attributen U hat die Form $X \rightarrow Y$, wobei $X, Y \subseteq U$ gilt. Sie beschreibt also eine Abhängigkeit zwischen zwei Attributen oder Attributmengen X und Y, wobei sich bei einer funktionalen Abhängigkeit vom X-Wert auf den Y-Wert schließen lässt. Das Schlüsselattribut einer Tabelle bildet einen Spezialfall. Wenn X Schlüsselattribut ist, müssen die X-Werte eindeutig sein und alle in der Tabelle enthaltenen Attribute sind funktional von X abhängig. In Tabelle 1.1 (a) gilt $A \rightarrow B$, also dass B von A funktional abhängig ist. $B \rightarrow A$ gilt hingegen nicht, da, wenn B den Wert b

hat, A sowohl den Wert 2 oder 3 annehmen kann.

Inklusionsabhängigkeiten nach [HSS18] Sollen alle X-Werte in der Relation $r_1(R_1)$ als Y-Werte in der Relation $r_2(R_2)$ vorkommen, kann dies durch eine Inklusionsabhängigkeit (IND, eng. inclusion dependency) festgelegt werden. Die formale Definition einer IND lautet:
Sei d eine Datenbank mit dem Schema S, die Relationen $r_1(R_1), r_2(R_2) \in d(S)$, $X \subseteq R_1$ sowie $Y \subseteq R_2$, so kann eine Inklusionsabhängigkeit durch $R_1[X] \subseteq R_2[Y]$ dargestellt werden. In Tabelle 1.1 ist als Beispiel die IND zwischen der Form $C \subseteq B$ erfüllt.

A	B
1	a
2	b
3	b
4	c

(a)

C	D
a	c
b	d

(b)

Tabelle 1.1: Tabellen mit der FD $A \rightarrow B$ in der Tabelle (a) und einer IND der Form $C \subseteq B$.

1.3 Struktur der Arbeit

Im Kapitel 2 werden bekannte Probleme und Methoden bei der Reformulierung von Anfragen vorgestellt, sowie auf das Finden von funktionalen und Inklusionsabhängigkeiten eingegangen. Im Kapitel 3 werden eigene Umformungen vorgestellt und mit einem Beispiel motiviert. Die Implementation dieser Regeln wird im Kapitel 4 vorgestellt und im Kapitel 5 werden Tests für die Implementierung vorgestellt. Im Kapitel 6 wird die Arbeit kurz zusammengefasst und einen Ausblick auf weitere mögliche Fragestellungen gegeben. Im A werden Hinweise zu der digitalen Version dieser Arbeit gegeben.

Kapitel 2

Stand der Technik

Das Problem des QueryRewriting ist aus dem Problem des *Answering Queries using Views (AQuV)* bekannt,welches unter anderem beim Ausführen von Anfragen über verteilte Datenbestände zur Anwendung kommt. Bei diesen können die Datenbestände als materialisierte Sichten über einem globalen Schema gesehen werden. Um die Anfragen zu beantworten, muss diese also so umgeschrieben werden, dass nur die Sichten genutzt werden [LMSS95]. Das AQuV und das *Answering Queries using Operators (AQuO)* Problem werden im Folgenden kurz vorgestellt. Danach werden Lösungsansätze betrachtet, welche beim Rewriting funktionale und Inklusionsabhängigkeiten mit einbeziehen.
In diesem Kapitel werden als erstes die Probleme des AQuV und des AQuO vorgestellt. Danach wird ein kurzer Überblick über algebraische Optimierungsregeln und bestehende Techniken der Anfragetransformation mit FDs und INDs gegeben. Der Stand der Technik wird durch zwei Abschnitte abgeschlossen, in denen Algorithmen zum Finden von funktionalen und Inklusionsabhängigkeiten vorgestellt und verglichen werden.

2.1 Answering Queries using Views

Beim Problem, mit Sichten Datenbankanfragen zu beantworten (kurz AQuV, von engl. **A**nswering **Q**ueries **u**sing **V**iews), soll eine Anfrage neu formuliert werden, sodass im Ergebnis möglichst viele Tupel aus der Antwort der Originalanfrage enthalten sind. In der Definition 2.1 wird dieses Problem formell dargestellt.

Definition 2.1. *Seien D ein Datenbankschema, Q eine Anfrage und V eine Menge von Sichten über D, so wird eine Anfrage Q_1 gesucht, welche nur Sichten aus V verwendet, sodass:*

$$Q_1(D) \sqsubseteq Q(D) \Leftrightarrow \forall d(D) : Q_1(d) \subseteq Q(d)$$

Die Anfrage Q_1 liefert ein maximal containt set wenn: $\nexists Q' : Q_1(D) \sqsubset Q'(D) \sqsubseteq Q(D)$. Das bedeutet, dass kein Rewriting mehr Tupel von $Q(D)$ berechnen kann. Im Idealfall werden alle Ergebnistupel gefunden und $Q_1(D) \equiv Q(D)$ gilt [AC19].

Es gibt verschiedene Gründe, Anfragen über Sichten auszuführen. Wenn einem Nutzer nur Anfragen über Sichten erlaubt sind, kann über die Definition der Sichten festgelegt werden, auf welche Daten der Nutzer zugreifen darf. Eine weitere Möglichkeit, warum Sichten genutzt werden sollten, ist, dass diese als materialisierte Sichten vorliegen. Da diese vorberechnet sind, können Anfragen über diese schneller sein.

2.2 Answering Queries using Operators

Die Problemstellung des Beantwortens von Anfragen mit einer Einschränkung von verwendbaren Operatoren (kurz AQuO, engl. für **A**nswering **Q**ueries **u**sing **O**perators) taucht im PArADISE-Projekt auf und ist eine Abwandlung des AQuV-Problems. Es stellt dar, dass auf manchen Schichten, wie z.B. der Sensorebene, nicht alle Operatoren verwendet werden können. Gesucht wird hierbei eine Anfrage, welche die erlaubten Operatoren verwendet und nur minimal mehr oder im Idealfall gleich viele Ergebnisse liefert als die ursprüngliche Anfrage. Das AQuO-Problem wird durch die Definition 2.2 formal dargestellt.

Definition 2.2. *(nach [GH17]) Seien D ein Datenbankschema, Q eine Anfrage und L eine Menge von Ebenen, sowie pro Ebene $L_i \in L$ und O_i eine Menge von Operatoren. Gesucht wird eine Anfrage Q_1, welche nur Operatoren aus O_1 verwendet, sodass gilt:*

$$Q_1(D) \sqsupseteq Q(D) \Leftrightarrow \forall d(D) : Q_1(d) \supseteq Q(d)$$

Das Rewriting liefert also eine minimal größere Ergebnismenge. Diese ist im Idealfall genau so groß wie die der ursprünglichen Anfrage. Die mindestens genau so große Ergebnismenge ist wichtig, da sich mit Hilfe dieser bei weiteren Anfragen alle Ergebnisse berechnen lassen. Wenn das Rewriting eine minimal kleinere Ergebnismenge liefern würde, wäre dies nicht mehr möglich [GH17].

2.3 Algebraische Optimierung von Anfragen

Die algebraische Optimierung ist ein Teil der logischen Optimierung von Anfragen. Das Ziel der Optimierung ist das schnellere Ausführen von Anfragen. Da bei der algebraischen Optimierung Äquivalenzregeln verwendet werden, wird diese manchmal auch als regelbasierte Optimierung bezeichnet. Die Anwendung der Regeln ermöglicht das Umformen einer Anfrage und das Entfernen von redundanten Operationen. Da in dieser Arbeit der Regelsatz um Regeln, welche FDs und INDs als Bedingungen haben, erweitert wird, werden erst benötigte Variablen und danach kurz einige der Optimierungsregeln nach [SSH11] vorgestellt. Die Relationen werden in den Regeln durch R_1, R_2 und R_3 dargestellt. X und Y sind Attributmengen. Prädikate werden mit F bezeichnet und die Funktion $attr(F)$ gibt die Attribute, über welche F definiert ist, aus.

1. **KommJoin:** Die KommJoin-Regel beschreibt, dass der Verbundoperator $\bowtie$ kommutativ ist.

$$R_1 \bowtie R_2 \equiv R_2 \bowtie R_1$$

2. **AssozJoin:** Diese Regel beschreibt, dass der Verbundoperator $\bowtie$ assoziativ ist.

$$(R_1 \bowtie R_2) \bowtie R_3 \equiv R_1 \bowtie (R_2 \bowtie R_3)$$

3. **ProjProj:** Bei der Projektion dominiert der äußere Operator den inneren. Hierbei muss $X \subseteq Y$ gelten.

$$\pi_X(\pi_Y(R_1)) \equiv \pi_X(R_1)$$

4. **SelSel:** Die geschachtelten Selektionen können vertauscht oder in einer Selektion kombiniert werden.

$$\sigma_{F_1}(\sigma_{F_2}(R_1)) \equiv \sigma_{F_1 \wedge F_2}(R_1) \equiv \sigma_{F_2}(\sigma_{F_1}(R_1))$$

5. **SelProj:** Die Selektion und die Projektion verhalten sich zueinander kommutativ, falls die Selektion über Attribute der Projektion definiert ist.

$$\sigma_F(\pi_X(R_1)) \equiv \pi_X(\sigma_F(R_1)) \qquad \text{falls } attr(F) \supseteq X$$

Wenn dies nicht der Fall ist, kann die Projektion um die notwendigen Attribute erweitert werden. Diese können nach der Vertauschung wegprojiziert werden.

$$\pi_{X_1}(\sigma_F(\pi_{X_1,X_2}(R_1))) \equiv \pi_{X_1}(\sigma_F(R_1)) \qquad \text{falls } attr(F) \subseteq X$$

6. **SelJoin:** Wenn alle Selektionsattribute aus einer Relation stammen, kann die Selektion vor dem Verbund angewendet werden.

$$\sigma_F(R_1 \bowtie R_2) \equiv \sigma_F(R_1) \bowtie R_2 \qquad \text{falls } attr(F) \subseteq R_1$$

Falls das Selektionsprädikat in F_1 und F_2 aufgespalten werden kann, sodass $F = F_1 \wedge F_2$ gilt, ist folgende Umformung möglich:

$$\sigma_F(R_1 \bowtie R_2) \equiv \sigma_{F_1}(R_1) \bowtie \sigma_{F_2}(R_2) \qquad \text{falls } attr(F_1) \subseteq R_1 \text{ und } attr(F_2) \subseteq R_2$$

Eine weitere Möglichkeit ist, ein F_1 mit Attributen aus R_1 abzuspalten, sodass die Prädikate aus F_2 sowohl R_1 als auch R_2 betreffen.

$$\sigma_F(R_1 \bowtie R_2) \equiv \sigma_{F_2}(\sigma_{F_1}(R_1) \bowtie R_2) \qquad \text{falls } attr(F_1) \subseteq R_1$$

7. **ProjJoin:** Eine Projektion kann in den Verbund hineingezogen werden, wenn die Verbundattribute erhalten bleiben. Dies wird durch die Berechnung der Y_i sichergestellt. Die Verbundattribute können erst nach dem Verbund wegprojiziert werden.

$$\pi_X(R_1 \bowtie R_2) \equiv \pi_X(\pi_{Y_1}(R_1) \bowtie \pi_{Y_2}(R_2))$$

mit

$$Y_1 = (X \cap R_1) \cup (R_1 \cap R_2)$$

und

$$Y_2 = (X \cap R_2) \cup (R_1 \cap R_2)$$

2.4 Query-Rewriting mit Betrachtung von FDs und INDs

Wenn beim Query-Rewriting Abhängigkeiten beachtet werden, können zusätzliche Rewritings möglich sein. Die so gefundenen Rewritings gelten nur, solange die Abhängigkeiten gelten. Ein Algorithmus zum Finden von solchen Rewritings für das AQuV-Problem wird in [Gry99] vorgestellt.
Um Query-Containment unter Abhängigkeiten darzustellen, wird in [Gry99] ein neuer Operator eingeführt. In der Quelle wird $\subseteq_\Delta$ als neuer Operator verwendet. Um mit den Operatoren konsistent zu bleiben, wird in dieser Arbeit $\sqsubseteq_\Delta$ verwendet. Sei Δ eine Menge von Abhängigkeiten, so bedeutet der Ausdruck $Q_1(D) \sqsubseteq_\Delta Q(D)$, dass der Ausdruck $Q_1(D) \sqsubseteq Q(D)$ gilt, solange die Abhängigkeiten Δ gelten. Eine Äquivalenz von den Anfragen Q_1 und Q_2 unter den Abhängigkeiten Δ lässt sich durch $Q_1 \equiv_\Delta Q_2$ darstellen.

Rewriting Beispiel mit INDs Ist eine Datenbank mit den Relationen $R_1(a, b)$, $R_2(b, c)$ und $R_3(b, d)$ mit der IND $R_2[b] \subseteq R_3[b]$ gegeben, ist die Anfrage $Q_1 = \pi_a(R_1 \bowtie R_2)$ in der Anfrage $Q_2 = \pi_a(R_1 \bowtie R_3)$ enthalten. Im AQuV-Problem ist also Q_1 ein Rewriting für Q_2. Im AQuO-Problem wäre Q_2 ein gültiges Rewriting für Q_1, da $Q_2 \subseteq_\Delta Q_1$ gilt.

Rewriting Beispiel mit FDs In einer Datenbank existiert die Relation $R_4(abc)$ mit den funktionalen Abhängigkeiten $a \rightarrow b$ und $a \rightarrow c$ sowie die Sichten $V_1 = \pi_{a,b}(R_4)$ und $V_2 = \pi_{a,c}(R_4)$. Somit ist $Q' = V_1 \bowtie V_2$ ein Rewriting für die Anfrage $Q = R_4$.

2.5 Erkennen von funktionalen Abhängigkeiten

Um funktionale Abhängigkeiten für das Query Rewriting verwenden zu können, müssen diese erst im Datensatz erkannt werden. Da dies nicht der Kern der Arbeit ist, soll auf einen bereits implementierten Algorithmus zurückgegriffen werden. In [PEM+15] vergleichen die Autoren laut ihrer Aussage die wichtigsten und meist zitierten Algorithmen zum Finden von FDs. Hierzu werden diese in Java implementiert und auf verschiedenen Datensätzen ausgeführt. Die Implementationen stehen auf der Webseite des Hasso-Plattner-Instituts [1] zum Download zur Verfügung.
Die Algorithmen lassen sich in drei Kategorien einteilen. Tane, Fun, Fd_Mine und DFD arbeiten mit Gittertraversierung. Dep-Miner und FastFDs arbeiten mit sogenannten *agree* und *differse sets*. Die letzte Kategorie sind Algorithmen, welche über Induktion funktionale Abhängigkeiten herleiten. Nur der Algorithmus Fdep fällt unter diese Kategorie. Im Folgenden werden die Algorithmen kurz vorgestellt und die in [PEM+15] erwähnten Probleme und Änderungen genannt. Für eine genauere Beschreibung der Algorithmen wird auf das jeweilige Vorstellungspaper der einzelnen Algorithmen oder [PEM+15] verwiesen.

2.5.1 Vorstellung der Algorithmen

TANE Der Algorithmus TANE wurde in [HKPT99] vorgestellt und arbeitet mit Gittertraversierung. Das Gitter wird von unten nach oben durchlaufen und hierzu in Ebenen eingeteilt. Die Gitterebene L_i besteht aus Attributkombinationen der Größe i. Auf jeder Ebene werden alle Attributkombinationen $X \subset L_i$ auf die funktionale Abhängigkeit $X \setminus A \rightarrow A, \forall A \in X$ getestet. Um den Suchraum einzugrenzen, werden drei *pruning*-Regeln (pruning, engl. für reduzieren) verwendet. Diese machen sich zum Vorteil, dass, um zu einem vollständigen Ergebnis zu kommen, nur minimale FDs gefunden werden müssen. Zusätzlich wird für jede Attributkombination eine Menge von Kandidaten für die rechte Seite der Abhängigkeit gespeichert. Die drei pruning-Regeln lauten *minimal*, *right-hand-side* und *key pruning*. Da das in [HKPT99] vorgestellte *key pruning* Fehler verursachte, wurde dieses in der in [PEM+15] vorgestellten Implementation überarbeitet.

Fun Fun arbeitet wie TANE mit Gittertraversierung und durchläuft das Gitter auch von unten nach oben. Der Suchraum ist aber durch eine restriktivere Kanidatengenerierung kleiner. Um den Suchraum weiter einzugrenzen, verwendet auch Fun die in TANE umgesetzten *pruning*-Regeln. Anders als bei TANE werden bei Fun hierzu sogenannte *free* und *non-free sets* verwendet. In einem *free set* ist keines der Attribute von einer Untermenge des *free sets* funktional abhängig. Dies ermöglicht ein effizientere Einschränkung des Suchraumes [PEM+15]. Der Fun-Algorithmus wurde in [NC01] vorgestellt.

Fd_Mine Wie TANE und Fun arbeitet auch Fd_Mine mit Gittertraversierung, verwendet aber eine zusätzliche *pruning*-Regel. Diese basiert auf der Äquivalenz der Klasse von zwei Attributen. Die Äquivalenz ist gegeben, wenn zwei Attribute gegenseitig voneinander abhängig sind. Fd_Mine liefert jedoch nicht immer minimale FDs [PEM+15]. In [YHB02] wurde Fd_Mine vorgestellt.

DFD Auch der DFD-Algorithmus (kurz für Discovery of Functional Dependencies) arbeitet mit Gittertraversierung. Im Gegensatz zu TANE, Fun und Fd_Mine durchläuft DFD das Gitter aber in einer randomisierten Tiefensuche. Im Gegensatz zu TANE wird hier der Suchraum durch mehrere Gitter dargestellt. Ein Gitter beschreibt hierbei alle möglichen rechten Seiten der Abhängigkeit für eine linke Seite. Auch

[1] `https://hpi.de/naumann/projects/repeatability/data-profiling/fds.html`, zuletzt besucht am 19.02.21 um 11:40 Uhr

DFD verwendet *pruning*-Regeln um seinen Suchraum einzugrenzen [PEM+15]. Der DFD-Algorithmus wurde in [ASN14] vorgestellt.

Dep-Miner Der in [LPL00] vorgestellte Dep-Miner-Algorithmus arbeitet mit *agree sets*. Ein *agree set* beinhaltet alle Attribute, welche für zwei Tupel gleich sind. Aus den *agree sets* werden für die Attribute sogenannte *maximal sets* erstellt. Die *maximal sets* beschreiben maximale Nicht-FDs. Mit Hilfe ihrer Inversen werden die minimalen FDs berechnet [PEM+15].

FastFDS FastFDS soll eine Verbesserung des Dep-Miner-Algorithmus darstellen und wurde in [WGR01] vorgestellt. Statt mit *maximal sets* werden bei FastFDS *difference sets* erstellt und über diese die minimalen FDs berechnet [PEM+15].

Fdep In [FS99] werden drei Versionen des Fdep-Algorithmus vorgestellt, welcher die funktionalen Abhängigkeiten durch paarweises Vergleichen aller Tupel findet. In [PEM+15] wird die Bottom-Up-Variante implementiert. Diese berechnet erst alle Nicht-FDs und dann aus diesen alle minimalen FDs.

2.5.2 Vergleich der Algorithmen

Um die implementierten Algorithmen zu vergleichen, wurden alle auf einem Rechner mit folgender Hardware ausgeführt: zwei Intel Xeon E5-2650 (2.00 GHz, Octa-Core) Prozessoren, 128 GB DDR3-1600 RAM und 4 TB raid5 Speicher. Als Java-Umgebung wurde OpenJDK 64-Bit Server VM 1.7.0 25 verwendet. Zusätzlich zu bemerken ist, dass alle Algorithmen nur einen Thread verwenden, sowie, dass der nutzbare RAM durch die JVM auf 100GB beschränkt wurde. Als maximale Laufzeit wurden zwei Stunden festgelegt [PEM+15].
Die Algorithmen wurden auf mehreren Datensätzen ausgeführt. In der Abbildung 2.1 werden die Ergebnisgrafiken für die Tests zur Reihen- und Spaltenskalierbarkeit dargestellt. Für die Spaltenskalierbarkeit wurden die Datensätze auf 1000 Tupel gekürzt. Die für die Tests verwendeten Datensätze sind in [PEM+15] kurz beschrieben.

Auswahl Als Orientierung für die Dimensionen des später verwendeten Datensatzes wurde der TPC-H-Benchmark mit dem Skalierfaktor 1 angegeben. In diesem ist die größte Spaltenanzahl 16 und die größte Anzahl von Tupeln in einer Tabelle sechs Millionen. Die Ergebnisse des Tests aus [PEM+15] legen nahe, dass für diesen Anwendungsfall der DFD-Algorithmus am besten geeignet zu sein scheint.

2.6 Erkennen von Inklusionsabhängigkeiten

Auch die Inklusionsabhängigkeiten müssen vor dem Query Rewriting erkannt werden. Zur Auswahl des Algorithmus wird auf das vergleichende Paper [DSW+19] zurückgegriffen. In diesem werden 13 Algorithmen verglichen. Die Implementationen der Algorithmen sind öffentlich verfügbar [2].
Bei den Algorithmen gibt es zwei große Kategorien. Die erste Kategorie findet unäre Inklusionsabhängigkeiten. Das heißt, dass pro Abhängigkeit ein Attribut von einem anderen abhängig ist. Unäre Inklusionsabhängigkeiten haben also die Form $R_1[a] \subseteq R_2[a]$. Bei der zweiten Art von Algorithmen findet man Abhängigkeiten, bei denen n Attribute von n anderen Attributen inklusionsabhängig sind. Diese Algorithmen arbeiten mit unären Inklusionsabhängigkeiten als Ausgangspunkt oder berechnen diese am Anfang selbst [DSW+19].
Im Folgenden wird sich auf die Algorithmen zum Finden von unären Abhängigkeiten beschränkt. Von

[2] https://github.com/HPI-Information-Systems/inclusion-dependency-algorithms, zuletzt besucht am 19.02.21 um 11:40

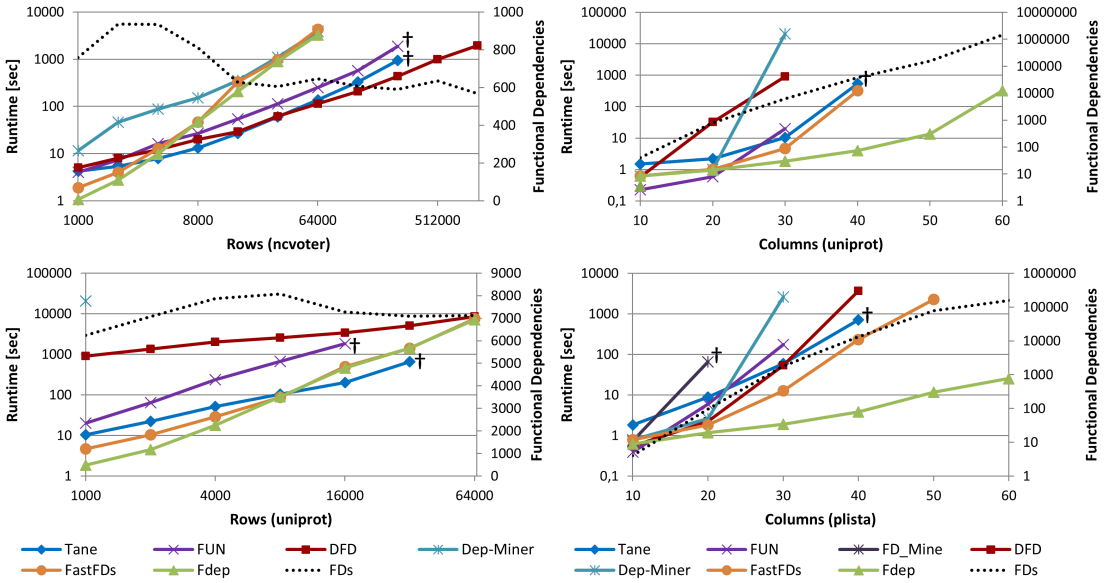


Abbildung 2.1: Die Laufzeiten der Algorithmen zum Finden von FDs mit steigender Reihen- und Spaltenanzahl. Wenn der Arbeitsspeicher nicht mehr ausreicht, ist dies durch ein Kreuz markiert. Die Grafik wurde aus dem Artikel [PEM+15] entnommen.

den acht Algorithmen werden drei Algorithmen kurz vorgestellt. Danach werden die Testumgebung und die Ergebnisse aus [DSW+19] dargestellt.

2.6.1 Vorstellung der Algorithmen

DeMarchi Der DeMarchi-Algorithmus wurde in [DMLP02] vorgestellt. Als erstes werden die Attribute nach ihren Datentypen gruppiert. In den so erstellten Gruppen wird jedes Attribut mit jedem anderen Attribut kombiniert, um Kandidaten für die INDs zu erstellen. Mit Hilfe eines inversen Indexes werden aus den Kandidaten alle Nicht-INDs entfernt. Übrig bleiben unäre INDs [DSW+19].

Faida Faida nutzt zur Bestimmung der INDs Approximationen, um eine schnellere Berechnung zu ermöglichen. Der Nachteil hierbei ist, dass die Ergebnisse nicht immer korrekt sind. In Faidas Fall heißt dies, dass zwar alle INDs im Ergebnis enthalten sind, aber auch Nicht-INDs mit ausgegeben werden. Für die Approximation nutzt Faida *hashing*, *sampling* und *sketching* [DSW+19]. Der Faida-Algorithmus wurde in [KPD+17] vorgestellt.

Many Der Many-Algorithmus ist darauf ausgelegt, mit vielen Attributen, aber verhältnismäßig wenig Tupeln zu arbeiten und wurde in [TPN17] vorgestellt. Um eine gute Performance zu bieten, wird hier besonders auf die Kandidatengenerierung geachtet. Zur Kandidatengenerierung wird pro Spalte ein sogenannter *BloomFilter* verwendet. Durch den Vergleich von *BloomFiltern* werden nun Kandidaten erstellt. Da erwartet wird, dass relativ wenige Zeilen vorliegen, können diese schnell getestet werden [DSW+19].

2.6.2 Vergleich der Algorithmen

Der Test im Paper [DSW+19] wird auf einem Dell PowerEdge R620 mit CentOS 6.10, zwei Intel Xeon E5-2650 (2.00 GHz, Octa-Core) Prozessoren und 128 GB DDR3-1600 RAM durchgeführt. Als Umgebung wird Oracles JDK 64-Bit Server VM 1.8.0_151 verwendet und die Daten werden aus einer PostgreSQL

Datenbank (Version 9.3.23) gelesen. Bis auf den hier nicht vorgestellten Algorithmus Sindy nutzen alle Algorithmen nur einen Kern [DSW+19].

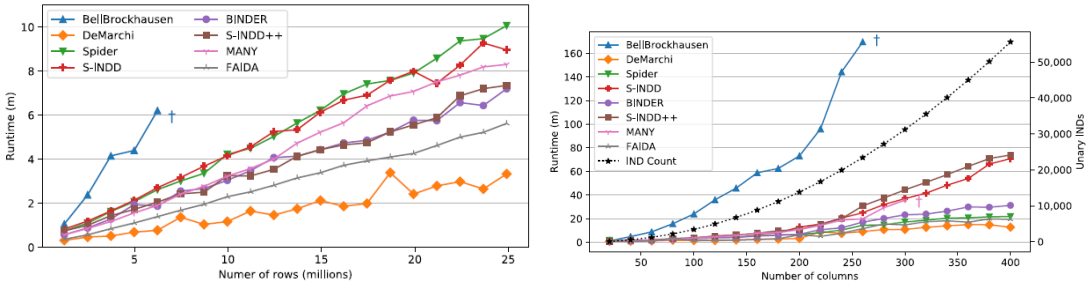


Abbildung 2.2: Die Laufzeiten der Algorithmen zum Finden von INDs mit steigender Reihen und Spaltenanzahl. Die Grafik wurde aus dem Artikel [DSW+19] entnommen.

Auswahl Als Datensatz wird wieder der TPC-H-Benchmark verwendet. Dieser hat insgesamt 61 Spalten und in der größten Tabelle sechs Millionen Tupel. In der aus [DSW+19] entnommenen Abbildung 2.2 lässt sich erkennen, dass sich die Laufzeit der Algorithmen bei dieser Spaltenanzahl nicht nennenswert unterscheidet und nur der Algorithmus BellBrockhausen eine deutlich längere Laufzeit hat. Bei der Tupelskalierbarkeit zeigt sich ein deutlicherer Unterschied. Bei sechs Millionen Tupeln ist DeMarchi schneller als die übrigen Algorithmen. Faida ist der zweitschnellste Algorithmus [DSW+19]. Da Faida auch falsche Ergebnisse liefert, wird DeMarchi zum Finden von IND ausgewählt.

Kapitel 3

Konzept

Um die Umformung von Anfragen umzusetzen, wird in dieser Arbeit mit der relationalen Algebra gearbeitet. In der relationalen Algebra existieren bereits Regeln zu Äquivalenzen zwischen verschiedenen Operatoren. Diese können dazu genutzt werden, eine Anfrage so umzuformulieren, dass diese effizienter ausgeführt werden kann. Durch funktionale und Inklusionsabhängigkeiten lassen sich weitere Regeln formulieren, welche durch Beachtung der Abhängigkeiten zusätzliche Umformungen ermöglichen sollen.

3.1 Eigene Umformungsregeln

Die bekannten Umformungsregeln lassen sich anwenden, ohne Wissen über die vorliegende Datenbank zu haben. Die eigenen Umformungsregeln haben funktionale oder Inklusionsabhängigkeiten als zusätzliche Randbedingungen. Schlüsselbedingungen, ein Spezialfall von FDs, und Fremdschlüsselbedingungen, ein Spezialfall von INDs, können von dem Datenbankmanagementsystem kontrolliert und durchgesetzt werden. Weitere gefundene Abhängigkeiten in einem Datensatz können durch Änderungen im Datensatz ungültig werden.
Für die Regeln werden die in Kapitel 2.3 vorgestellten Notationen für Variablen verwendet. Die Funktion *attr* wird zusätzlich erweitert, damit diese auch auf eine Relation angewandt werden kann, sodass $attr(R)$ die Attribute der Relation R ausgibt.

3.1.1 Umformungen bei der Gruppierung

Eine Gruppierung wird durch den Operator $\gamma_{F(X),Y}(R)$ dargestellt. X und Y sind Teilmengen von Attributen aus R. $F(X)$ steht für eine Menge von (verschiedenen) Aggregatfunktionen über Attributen aus X. Y beinhaltet die Attribute, nach welchen gruppiert wird.

Entfernung von Gruppierungsattributen

Sei $A, B \subseteq Y$ und gelten die FD $A \rightarrow B$, so gilt:

$$\gamma_{F(X),Y}(R) \equiv_{\Delta} \gamma_{F(X),Y \setminus B}(R) \bowtie \pi_{A \cup B}(R)$$

Wenn bei der Gruppierung über A und B gruppiert wird und die Abhängigkeit $A \rightarrow B$ gilt, ist A ausschlaggebend für die Gruppierung. Da B von A abhängig ist, sind, wenn bei Tupeln die Werte der Attribute in A gleich sind, auch die Werte der Attribute in B gleich. Durch die zusätzliche Gruppierung über B kann die Gruppierung also nicht verfeinert werden. Da die Möglichkeit besteht, dass auf das

aus der Gruppierung entfernte Attribut projiziert wird, muss dieses durch einen Verbund hinzugefügt werden.[1]

Ersetzung des Gruppierungsattributs

Sei $A \subseteq Y$, $B \subseteq R$ und gelten die FDs $A \rightarrow B$ und $B \rightarrow A$, so gilt:

$$\gamma_{F(X),Y}(R) \equiv_{\Delta} \gamma_{F(X),(Y \setminus A) \cup B}(R) \bowtie \pi_{A \cup B}(R)$$

Eine FD beschreibt, dass zwischen den Attributmengen eine n:1-Beziehung gilt. Das heißt, dass sich n Attributwertkombinationen, welche pro Attribut einen Wert enthalten, auf eine Attributwertkombination der abhängigen Menge abbilden lassen. Durch die beiden geforderten FDs wird also eine 1:1-Abbildung zwischen den Attributmengen beschrieben, weshalb eine Ersetzung der einen durch die andere möglich ist.
Da auch hier die Möglichkeit besteht, dass auf das aus der Gruppierung entfernte Attribut projiziert wird, muss dieses durch einen Verbund hinzugefügt werden.

3.1.2 Umformung bei der Verbundoperation

Beim Verbundoperator $\bowtie$ werden zwei Relationen miteinander kombiniert. Um beim Verbund als Ergebnis ein kartesisches Produkt zu vermeiden, können zwei Attribute, jeweils eins pro Relation, gleichgesetzt werden.

Entfernen eines Verbundes mit IND

Existieren die Relationen R_1, R_2, die Attributmenge A, sodass $A \equiv R_1 \cap R_2$ gilt, und die IND $R_1[A] \subseteq R_2[A]$, so gilt die Aussage:

$$\pi_{attr(R_1)}(R_1 \bowtie R_2) \equiv_{\Delta} R_1$$

Nach einem Verbund sind nur noch die Attributwerte, welche in beiden Attributen vorhanden sind, enthalten. Durch die Inklusionsabhängigkeit sind hier für die Verbundattribute A alle Werte aus R_1 enthalten, wodurch der Verbund mit R_2 entfernt werden kann.

Beispiel: Sei beispielsweise die Anfrage $\pi_{a,b}(R_1 \bowtie R_2)$ über die beiden Relationen $R_1 = \{a, b\}$ und $R_2 = \{b, c\}$ gegeben. Wenn zwischen den Relationen die IND $R_1[b] \subseteq R_2[b]$ besteht, lässt sich bei der Anfrage der Verbund entfernen. Die neue Anfrage hat dann die Form $\pi_{a,b}(R_1)$. Der Anfragebaum der ursprünglichen Anfrage und der Anfrage nach der Vereinfachung ist in der Abbildung 3.1 dargestellt.

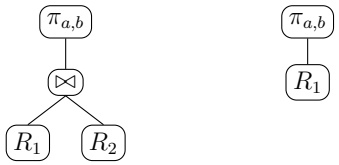


Abbildung 3.1: Darstellung des Anfragebaumes vor und nach dem Entfernen des Verbundpartners mit Hilfe der IND $R_1[b] \subseteq R_2[b]$.

[1] In einigen Datenbankmanagementsystemen können auch die funktional abhängigen Attribute, über welche nicht gruppiert wurde, in der SELECT-Klausel stehen. Dies ist im Feature T301 vom SQL-Standard beschrieben. (persönliche Mitteilung von Hannes Grunert am 31.03.2021)

Tausch des Verbundpartners

Seien R_1, R_2, R_3 und die Attributmenge A, sodass $A \equiv R_1 \cap R_2 \equiv R_1 \cap R_3$ gilt. Existiert außerdem noch die IND $R_2[A] \subseteq R_3[A]$ gilt die Aussage:

$$\pi_{attr(R_1)}(R_1 \bowtie R_2) \subseteq_\Delta \pi_{attr(R_1)}(R_1 \bowtie R_3)$$

Da alle Attributwerte aus A aus R_2 auch in R_3 vorkommen, liefert ein Verbund über ein Attribut A mit R_3 mindestens genauso viele Tupel wie ein Verbund mit R_2. Da es sich bei dieser Regel um keine Äquivalenz handelt, kann nur in eine Richtung ersetzt werden. Für das AQuO-Problem kann R_2 durch R_3 ersetzt werden. Wenn das Ergebnis der Anfrage äquivalent sein soll, kann mit der Relation R_3 nur vor selektiert werden und der Verbund mit R_2 muss bestehen bleiben. Hierdurch wird die Bedingung an die Projektion überflüssig und statt A wird B mit $B \equiv R_1 \cap R_3$ verwendet. Wenn die IND $R_2[B] \subseteq R_3[B]$ besteht, so gilt die Äquivalenz:

$$R_1 \bowtie R_2 \equiv_\Delta \pi_{attr(R_1)}(R_1 \bowtie R_3) \bowtie R_2$$

Beispiel: Seien beispielsweise die drei Relationen $R_1 = \{a, b\}$, $R_2 = \{b, c\}$ und $R_3 = \{b, d\}$ gegeben. Zwischen den Relationen R_2 und R_3 existiere außerdem die IND $R_2[b] \subseteq R_3[b]$. So lässt sich die Anfrage $\pi_{a,b}(R_1 \bowtie R_2)$ im Sinne des AQuO in die Anfrage $\pi_{a,b}(R_1 \bowtie R_3)$ umformen. Die Änderung im Anfragebaum ist in der Abbildung 3.2 dargestellt.

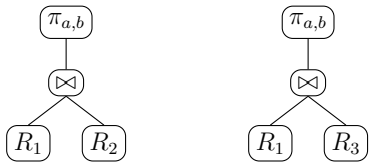


Abbildung 3.2: Darstellung des Anfragebaumes vor und nachdem der Verbundpartner mit Hilfe der IND $R_2[b] \subseteq R_3[b]$ getauscht wurde.

Entfernen eines Verbundes mit FDs

Sei die Relation R_1, die Attributmengen $A, B \subseteq R_1$ mit der Eigenschaft$a \subseteq A \cap Ba$ und die FDs $a \rightarrow A \setminus a$ und $a \rightarrow B \setminus a$ gegeben, so gilt:

$$\pi_A(R_1) \bowtie \pi_B(R_1) \equiv_\Delta \pi_{A \cup B}(R_1)$$

Da sowohl $A \setminus a$ als auch $B \setminus a$ von a funktional abhängig sind, lässt sich mit einem Verbund über a für alle Werte in A und B die originale Relation wiederherstellen. Wenn die Vereinigung von A und B alle Attribute der Relation enthält, kann bei der rechten Seite der Umformung die Projektion entfernt werden.

Beispiel: Die Beispielanfrage sei $\pi_{a,b}(R_4) \bowtie \pi_{a,c}(R_4)$. Die Relation R_4 hat die Attribute a, b und c. Außerdem sind die funktionalen Abhängigkeiten $a \rightarrow b$ und $a \rightarrow c$ in R_4 vorhanden. Mit dem Wissen über die FDs lässt sich die Beispielanfrage auch durch $\pi_{a,b,c}(R_1)$ beantworten. In diesem Beispiel könnte sogar noch die Projektion entfernt werden.

3.2 Beispiel

Als Beispiel für die Anwendung der eigenen Umformungsregeln wird eine Anfrage über den TCP-H-Datensatz verwendet. Im Folgenden werden kurz die verwendeten Relationen und Abhängigkeiten dargestellt.

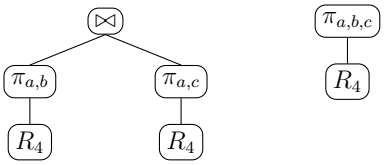


Abbildung 3.3: Darstellung des Anfragebaumes vor und nach dem Entfernen des Verbundpartners mit Hilfe der FDs $a \rightarrow b$ und $a \rightarrow c$.

Relationen:
Die Anfrage wird über vier Relationen vom TPC-H Datensatz gestellt. Im Folgenden ist eine kurze Auflistung der Relationen mit ihren Attributen zu finden:

- $Partsupp = \{Partkey, Suppkey, Avilqty, Supplycost, Comment\}$
- $Supplier = \{Suppkey, Name, Address, Nationkey, Phone, Acctbal, Comment\}$
- $Nation = \{Nationkey, Name, Regionkey, Comment\}$

Abhängigkeiten:
Für die Umformungen wird ein Teil der in dem TPC-H Datensatz[2] vorhandenen Abhängigkeiten verwendet. Die verwendeten funktionalen Abhängigkeiten bestehen alle in der Relation *Supplier* und lauten:

- $Name \rightarrow Suppkey$
- $Suppkey \rightarrow Name$
- $Suppkey \rightarrow Address$.

Außerdem wird für eine Umformung die Inklusionsabhängigkeit $Supplier[Nationkey] \subseteq Nation[Nationkey]$ benötigt.

3.2.1 Anfrage

Die Beispielanfrage ist Umgangssprachlich formuliert: Gib die Anzahl der Teile, welche ein Lieferant aus seinem jeweiligen Standort liefert, sowie den dazugehörigen Firmennahmen und Standort aus. Im Folgenden ist die Anfrage in SQL und der relationalen Algebra formuliert.

SQL:

```
SELECT s.Name, s.Address, n.Name, count(Partkey)
FROM Partsupp ps, Supplier s, Nation n
WHERE ps.Suppkey = s.Suppkey
AND s.Nationkey = n.Nationkey
GROUP BY n.Name, s.Name, s.Address
```

Relationale Algebra:

$$\pi_{Supplier.Name,Supplier.Address,Nation.Name,count(Partkey)}(\gamma_{count(Partkey);Nation.Name,Supplier.Name,Supplier.Address}(Partsupp \bowtie Supplier \bowtie Nation))$$

[2] https://dbis.informatik.uni-rostock.de/vldb2018/, zuletzt aufgerufen am 02.04.2021 um 18:50

3.2.2 Umformungen

Als erste Umformung wird in der Gruppierung das Attribut $Name$ durch $Suppkey$ ersetzt. Dies ist durch die beiden FDs $Name \rightarrow Suppkey$ und $Suppkey \rightarrow Name$ möglich. Als zweite Umformung wird in der Gruppierung das Attribut $Address$ entfernt, da dieses durch die FD $Suppkey \rightarrow Address$ nicht zur Verfeinerung der Gruppierung beiträgt. Durch das Verändern der Gruppierungsattribute wurden zwei neue Verbunde eingefügt. Die neue Anfrage hat nun die Form:

$$
\begin{aligned}
&\pi_{Supplier.Name,Supplier.Address,Nation.Name,count(Partkey)}(\\
&\quad \gamma_{count(Partkey);Nation.Name,Supplier.Suppkey}(\\
&\qquad Partsupp \bowtie Supplier \bowtie Nation) \\
&\quad \bowtie \pi_{Supplier.Suppkey,Supplier.Address}(\\
&\qquad Partsupp \bowtie Supplier \bowtie Nation)) \\
&\quad \bowtie \pi_{Supplier.Suppkey,Supplier.Name}(\\
&\qquad Partsupp \bowtie Supplier \bowtie Nation)) \\
&)
\end{aligned}
$$

Die hinzugefügten Verbunde lassen sich zusammenführen, da alle Attribute der Projektion von dem Attribut $Supplier.Suppkey$ funktional abhängig sind. Außerdem kann die Relation $Nation$ aus dem Verbund unter der Projektion entfernt werden, da auf kein Attribut der Relation projiziert wird und die IND $Supplier[Nationkey] \subseteq Nation[Nationkey]$ hinterlegt ist. Durch diese Umformung erweitern wir den Anfragebaum, zu dem, in der Abbildung 3.4 dargestellten, finalen Anfragebaum.

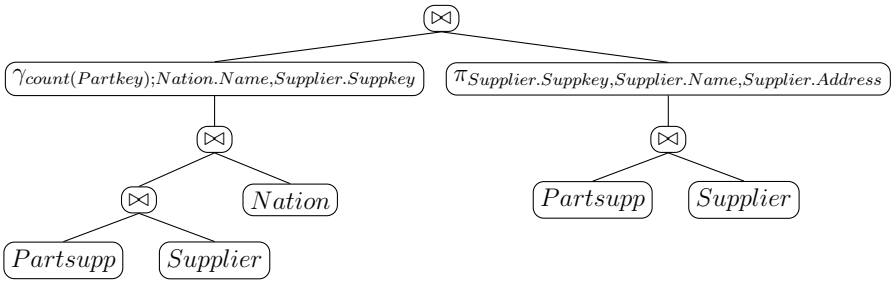


Abbildung 3.4: Der Anfragebaum der Beispielanfrage nach Anwendung der Umformungen.

Kapitel 4

Implementierung

Die in Kapitel 3 vorgestellten Äquivalenzen sollen als Umformungen implementiert werden. Da die Umformungen in einen bestehenden Rewriter hinzugefügt werden, wird dieser im Abschnitt 4.1 vorgestellt. Im Abschnitt 4.2 wird dargestellt, wie die funktionalen und Inklusionsabhängigkeiten abgelegt und eingelesen werden. Die genauen Umformungen mit einer Beschreibung des Ablaufes werden im Abschnitt 4.3 vorgestellt. Die konkrete Implementierung der Umformungen wird im Abschnitt 4.4 beispielhaft an der Umsetzung der Umformung D01 dargestellt. In Abschnitt 4.5 wird ein Beispiel für die Umformungen an einer komplexeren Anfrage gegeben.

4.1 Rewriter

Die im Konzept vorgestellten Umformungsregeln sollen in einen bereits bestehenden Rewriter eingearbeitet werden. Dieser ist in einem in Eclipse erstellten Java-Projekt umgesetzt[1]. Die elementaren Operatoren der relationalen Algebra werden durch Java-Klassen dargestellt. Diese können durch Vererbung und abstrakte Klassen geschachtelt werden. Im Rewriter können so auch komplexe Ausdrücke der relationalen Algebra dargestellt und umgeformt werden. Die relational-algebraischen Ausdrücke werden in XML-Dateien serialisiert. Der XML-Baum einer serialisierten Anfrage gleicht dem Anfragebaum in der relationalen Algebra. Wenn die Ausdrücke in XML dargestellt werden, können diese mit Hilfe der formalen Abfragesprache XPath durchsucht und bearbeitet werden.
Da die Operatoren der relationalen Algebra alle eine Relation als Ergebnis liefern, werden diese als eine Unterklasse der abstrakten Klasse `Relation` dargestellt. Die Basisrelationen werden durch eine gleichnamige Klasse dargestellt, welche auch eine Unterklasse der Klasse `Relation` ist. In der Abbildung 4.1 stellt ein Klassendiagramm alle Klassen dar, welche für die hier vorgestellten Umformungen benötigt werden. Dadurch, dass die Operatoren und die Basisrelationen die gleiche Klasse erweitern, können komplizierte und geschachtelte Ausdrücke der relationalen Algebra einfacher dargestellt werden.
Im Rewriter wird eine SQL-Anfrage als erstes in ein XML-Dokument überführt. Hierzu wird die Anfrage als erstes in eine Calcite[2]-Notation gebracht, um diese im folgenden Schritt in eine POJO-Notation zu überführen. Die Umwandlung in ein XML-Dokument wird durch die Funktion `generateDomFromPojo` umgesetzt, in welcher die Klasse `Serializer` aus dem Simple-Framework [3] und die Java-Klasse `DocumentBuilder` genutzt werden.[4]

[1] Die aktuelle Version des Rewriters ist unter `https://git.informatik.uni-rostock.de/hg/patch-queryrewriter` verfügbar.
[2] Notation für Apache Calcite `https://calcite.apache.org/` zuletzt besucht am 25.03.2021 um 9:30
[3] `http://www.simpleframework.org/`, zuletzt aufgerufen am 2.04.2021 um 10:20
[4] Hannes Grunert, persönliche Mitteilung vom 17.03.2021

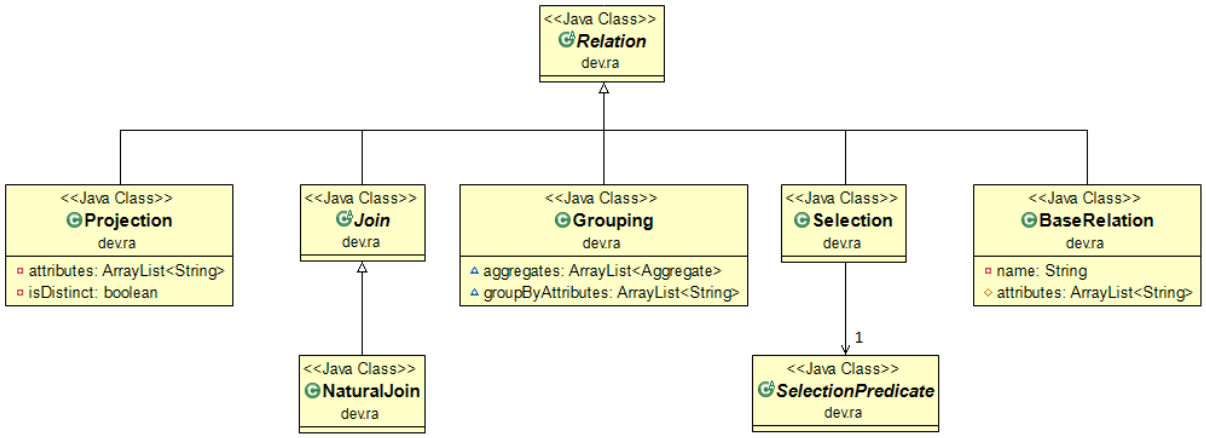


Abbildung 4.1: UML-Diagramm der genutzten Klassen zur Darstellung der relationalen Algebra aus dem Rewriter.

4.2 Hinterlegung der FDs und INDs

Für die in den Abschnitten 2.5 und 2.6 vorgestellten Algorithmen zum Finden der funktionalen und Inklusionsabhängigkeiten sind Implementationen vom HPI [5] vorhanden. Die Ergebnisse der Algorithmen werden durch Java-Klassen dargestellt. Diese implementieren unter anderem auch eine Serialisierung, sodass Abhängigkeiten vorher berechnet, gespeichert und später geladen werden können. Die Abhängigkeiten werden nach der Ausführung des Algorithmus abgelegt. Dies führt dazu, dass alle FDs einer Relation oder alle INDs des Schemas in einer Datei gespeichert werden. Die Abhängigkeiten können mit der Funktion `ResultReader.readResultsFromFile(pfad, typ)` eingelesen werden. Diese gibt eine Liste mit allen in der Datei gespeicherten Abhängigkeiten zurück. Der Parameter `pfad` ist ein String und gibt den Ort der einzulesenden Datei an. Auch der Parameter `typ` ist ein String und gibt den Typ der Abhängigkeit an. Für FDs wird der String `Functional Dependency` und für INDs der String `Inclusion Dependency` verwendet.
Um die Abhängigkeiten einzulesen, wurde die Klasse `DependencyReader` erstellt. Die Funktion `getFds()` ist in dem Quellcodeausschnitt 4.1 dargestellt. Diese wird genutzt, um die abgelegten FDs einzulesen. In der Zeile 4 wird festgelegt, in welchem Ordner die Abhängigkeiten hinterlegt sind. Durch die Schleife in den Zeilen 6 bis 11 werden alle Dokumente, welche auf `_fds` enden, mit der Funktion `readResultsFromFile` der Klasse `ResultReader` eingelesen. Da die Funktion zum Einlesen von Abhängigkeiten diese als Klasse `Result` einliest, wird in den Zeilen 17-19 ein Casting auf die Klasse `FunctionalDependency` durchgeführt. Für das Einlesen der INDs besteht eine ähnliche Funktion, bei welcher nur Dateien mit der Endung `_inds` eingelesen werden und das Casting auf die Klasse `InclusionDependency` durchgeführt wird.

[5] `https://hpi.de/naumann/projects/data-profiling-and-analytics/metanome-data-profiling/algorithms.html`, zuletzt besucht am 18.02.21 um 17:24

```
1  public static List<FunctionalDependency> getFds(){
2    List<Result> fds = new ArrayList<Result>();
3    try {
4      File files = new File("res/dependencies");
5      File[] filesList = files.listFiles();
6      for(int i=0; i < filesList.length; i++) {
7        if(filesList[i].getName().endsWith("_fds") && filesList[i].isFile()){
8          fds.addAll(ResultReader.readResultsFromFile(filesList[i].getPath(),
9            "Functional Dependency"));
10       }
11     }
12   } catch (Exception e){
13     System.out.println("Fehler");
14     e.printStackTrace();
15   }
16   List<FunctionalDependency> res = new ArrayList<FunctionalDependency>();
17   for(int i=0; i<fds.size(); i++) {
18     res.add((FunctionalDependency) fds.get(i));
19   }
20   return res;
21 }
```

Quellcodeausschnitt 4.1: Die Funktion *getFds()*, welche genutzt wird um alle abgelegten funktionalen Abhängigkeiten einzulesen.

4.3 Umformungen

Im Folgenden sollen die Umformungen dargestellt werden. In der Tabelle 4.3 sind die konkreten Umformungen mit ihrer jeweiligen Kennzeichnung gelistet. Die Kennzeichnung wird in der Implementation für den Funktionsnamen verwendet. Die bereits implementierten Regeln sind mit einem oder zwei Großbuchstaben sowie einer zweistelligen Nummerierung versehen. Die Kennzeichnung der hier implementierten Umformungen folgt diesem Schema. Als Großbuchstabe wurde das D für *dependency* gewählt, da alle Umformungen mit Abhängigkeiten arbeiten.
Die Regeln zu den Umformungen D01 und D02 sind in dem Abschnitt 3.1.1 und die Regeln D03 bis D05 sind in dem Kapitel 3.1.2 vorgestellt worden.

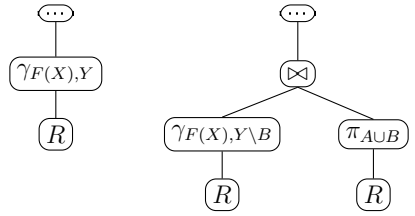
4.3.1 Umformung D01

Von der ersten Umformung ist nur der Gruppierungsoperator betroffen. Dieser kann mit dem in der Abbildung 4.2 dargestellten XPath-Ausdruck gefunden werden. Auch die Darstellungen eines Anfragebaums vor und nach der Umformung sind in der Abbildung zu finden.
Bei einem Durchlauf der Funktion soll als erstes getestet werden, ob die Gruppierung über mindestens zwei Attribute durchgeführt wird. Ist dies der Fall, kann überprüft werden, ob Abhängigkeiten zwischen den Attributen bestehen. Bestehen keine Abhängigkeiten zwischen den Attributen oder wird nur über ein Attribut gruppiert, ist keine Umformung möglich und die Funktion wird beendet. Ist dies nicht der Fall, werden alle abhängigen Attribute der ersten FD entfernt. Als letzter Schritt wird durch einen Verbund über die bestimmenden Attribute das entfernte Attribut wieder angefügt.
Das Anfügen der entfernten Attribute mit einem Verbund bietet den Vorteil, dass nicht getestet werden muss, ob diese in der weiteren Anfrage benötigt werden und das sich die Umformung in mehr Fällen

Kennzeichnung	Umformung	Randbedingungen
D01	$\gamma_{F(X),Y}(R) \Rightarrow \gamma_{F(X),Y \setminus B}(R) \bowtie \pi_{A \cup B}(R)$	FD: $A \rightarrow B$ $A, B \subseteq Y$
D02	$\gamma_{F(X),Y}(R) \Rightarrow \gamma_{F(X),(Y \setminus A) \cup B}(R) \bowtie \pi_{A \cup B}(R)$	FDs: $A \rightarrow B$,$B \rightarrow A$ $B \subseteq R$, $A \subseteq Y$
D03	$\pi_{attr(R_1)}(R_1 \bowtie R_2) \Rightarrow R_1$	IND: $R_1[A] \subseteq R_2[A]$ $A \equiv R_1 \cap R_2$
D04	$R_1 \bowtie R_2 \Rightarrow \pi_{attr(R_1)}(R_1 \bowtie R_3) \bowtie R_2$	IND: $R_2[A] \subseteq R_3[A]$ $A \equiv R_1 \cap R_3$
D05	$\pi_A(R_1) \bowtie \pi_B(R_1) \Rightarrow \pi_{A \cup B}(R_1)$	FDs: $X \rightarrow A$,$X \rightarrow B$ $A \cap B \supseteq X$

Tabelle 4.1: Auflistung der Umformungen mit ihren Randbedingungen sowie einer kurzen Kennzeichnung. Diese wird in der Implementation für den Namen der Funktion genutzt.

anwenden lässt. Um zu testen, ob weitere Attribute aus der Gruppierung entfernt werden können, kann die Funktion mehrfach angewandt werden.

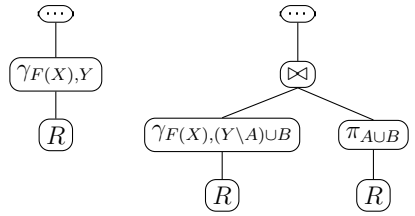


Knoten	XPath-Ausdruck
γ	//grouping
Y	//grouping/groupByAttributes
R	//grouping/[1]

Abbildung 4.2: Ausschnitte von einem Anfragebaum vor und nach der Umformung D01. In der Tabelle sind die XPath-Ausdrücke, mit welchen die entsprechenden Variablen gefunden werden können, dargestellt.

4.3.2 Umformung D02

Auch bei der zweiten Umformung ist nur der Gruppierungsoperator betroffen. In der Abbildung 4.3 ist der Anfragebaum vor und nach der Umformung, sowie die benötigten XPath-Ausdrücke dargestellt.
In einem Durchlauf der Funktion wird als erstes getestet, ob eine Teilmenge der Gruppierungsattribute die Determinante in einer funktionalen Abhängigkeit ist. Wenn dies gegeben ist, wird getestet, ob die abhängigen Attribute auch die Teilmenge bestimmt. Wenn beide Bedingungen gegeben sind, können die Attribute getauscht werden und die aus der Gruppierung entfernten Attribute mit einem Verbund wieder hinzugefügt werden.

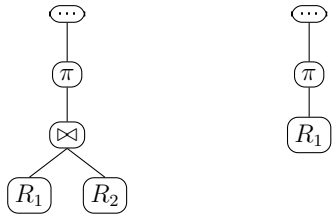


Knoten	XPath-Ausdruck
γ	//grouping
Y	//grouping/groupByAttributes
R	//grouping/[1]

Abbildung 4.3: Ausschnitte von einem Anfragebaum vor und nach der Umformung D02. Die Variablen können mit den in der Tabelle dargestellten XPath-Ausdrücken gefunden werden.

4.3.3 Umformung D03

Für die in der Umformung D03 vorgenommene Entfernung eines Verbundpartners ist es wichtig, dass nur die Attribute aus einer Relation benötigt werden. Dies kann der Fall sein, wenn eine Projektion nach dem Verbund ausgeführt wird. In der Abbildung 4.4 sind die für die Umformung benötigten Knoten sowie deren XPath-Ausdruck dargestellt.
Als erste Bedingung wird getestet, ob zwischen den Relationen eine Inklusionsabhängigkeit über die Schnittmenge der Attribute besteht. Der Spezialfall, dass die Mengen der Attributwerte in beiden Relationen gleich sind, die IND also in beide Richtungen besteht, muss hier beachtet werden. Danach wird die zweite Bedingung, dass auf keine Attribute der zu entfernenden Relation projiziert wird, getestet. Besteht die IND in beide Richtungen, muss der Test für beide Relationen durchgeführt werden. Sind beide Bedingungen erfüllt, kann die Relation entfernt werden. In der Umsetzung wird die verbleibende Relation kopiert und der Verbund samt Kinderknoten durch diese ersetzt.

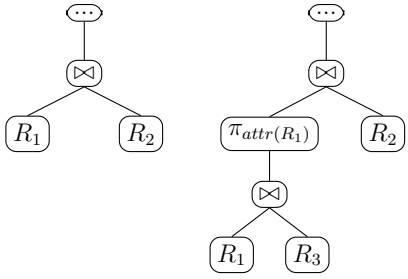


Knoten	XPath-Ausdruck
π	//projection[naturaljoin]
$\bowtie$	//projection/naturaljoin
R_1	//projection/naturaljoin/[1]
R_2	//projection/naturaljoin/[2]

Abbildung 4.4: Ausschnitte eines Anfragebaums vor und nach der Anwendung der Umformung D03 und die XPath-Ausdrücke der Knoten vor der Umformung.

4.3.4 Umformung D04

Bei der Umformung D04 wird ein Verbund in der Anfrage hinzugefügt. Die XPath-Ausdrücke der Knoten sowie der Anfragebaum vor und nach der Umformung sind in der Abbildung 4.5 dargestellt. Als erstes muss festgestellt werden, über welche Attribute der Verbund durchgeführt wird, also die Schnittmenge der Attribute der Relationen. Als zweites muss geprüft werden, ob eine Inklusionsabhängigkeit über die Verbundattribute besteht. Besteht keine IND über den Attributen oder nur zwischen den bereits verbundenen Relationen, kann diese Umformung nicht ausgeführt werden.



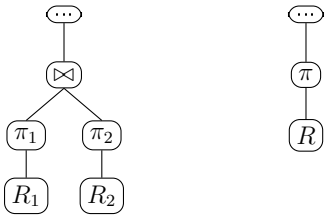
Knoten	XPath-Ausdruck
$\bowtie$	//naturaljoin
R_1	//naturaljoin/[1]
R_2	//naturaljoin/[2]

Abbildung 4.5: Abbildung von einem Anfragebaum vor und nach der Umformung D04. In der Tabelle sind sie XPath-Ausdrücke der Elemente vor der Umformung dargestellt.

4.3.5 Umformung D05

Die benötigten XPath-Ausdrücke für die Umformung zum Entfernen eines Verbundes mit funktionalen Abhängigkeiten sind in der Abbildung 4.6 dargestellt. Als erste Bedingung wird getestet, ob die Knoten

R_1 und R_2 die gleichen Relationen darstellen. Als zweiter Schritt muss festgestellt werden, über welches Attribut verbunden wird und ob die restlichen Attribute der Projektionen von diesem abhängig sind. Ist eine der beiden Bedingungen nicht erfüllt, kann die Transformation nicht durchgeführt werden. Wenn die Bedingungen erfüllt sind, wird der Verbund durch eine Projektion auf die Attribute aus π_1 und π_2 über der Relation R_1 ersetzt.



Knoten	XPath-Ausdruck
$\bowtie$	//naturaljoin[projection]
π_1	//naturaljoin/projection[1]
π_2	//naturaljoin/projection[2]
R_1	//naturaljoin/projection[1]/baseRelation
R_2	//naturaljoin/projection[2]/baseRelation

Abbildung 4.6: Ein Anfragebaum vor und nach der Anwendung der Umformungsregel D05 sowie die XPath-Ausdrücke zum Finden der Knoten vor der Umformung.

4.4 Beispiel für die Implementation

Als Beispiel für die Umsetzung der Umformungen in Java wird hier die Implementation der Regel D01 dargestellt. Diese wurde im Rahmen dieser Arbeit als Funktion `rewriteD01` in der Klasse `Rewriter` hinzugefügt. Zur Darstellung der Implementation wird ein Quellcodeausschnitt eingefügt und dieser dann kurz erklärt.

```
1 public Document rewriteD01(Document input, String prefix) throws Exception{
2   String xPath = "//grouping";
3   if(!(prefix == null || prefix.equals(""))) {
4     xPath = prefix + xPath;
5   }
6   NodeList possibleNodes = findOperatorV2(input, xPath);
7
8   for(int i=0; i<possibleNodes.getLength(); i++) {
9     Node group = possibleNodes.item(i);
```

Quellcodeausschnitt 4.2: Ausschnitt 1 der Funktion rewriteD01.

Im Quellcodeausschnitt 4.2 ist der Anfang der Funktion rewriteD01 zu sehen. In der Zeile 2 wird der XPath-Ausdruck festgelegt und so bestimmt, welcher der Ausgangsknoten für die Umformung ist. Mit dem Aufruf der Funktion `findOperatorV2` wird in dem übergebenen Dokument nach Knoten gesucht, welche den XPath-Ausdruck erfüllen. In diesem Fall werden alle Gruppierungsknoten zurückgegeben. In Zeile 8 wird eine Schleife begonnen, welche alle gefundenen Knoten durchläuft. Der zu bearbeitende Knoten wird in Zeile 9 der variablen `group` zugewiesen.

```
10      //Erstellen einer Liste mit allen Gruppierungsattributen
11      NodeList groAttList = findOperatorV2List(group,
12        "groupByAttributes/attribute");
13      ArrayList<String> groList = new ArrayList<>();
14      for(int j=0; j<groAttList.getLength(); j++) {
15        groList.add(groAttList.item(j).getTextContent());
16      }
```

Quellcodeausschnitt 4.3: Ausschnitt 2 der Funktion rewriteD01.

Als erster Bearbeitungsschritt werden die Gruppierungsattribute ausgelesen. Das Vorgehen hierzu wird in dem Quellcodeausschnitt 4.3 dargestellt. Hierzu wird in Zeile 11 eine Liste mit den Knoten, welche die Gruppierungsattribute beinhalten, erstellt. In den Zeilen 14 und 15 wird der Name der Gruppierungsattribute ausgelesen und in der Liste `groList` gesammelt.

```
17      //wenn über weniger als 2 Attribute gruppiert wird, ist keine
        Entfernung möglich
18      if (groList.size()<2) {
19        break;
20      }
21
22      //testen ob zwischen den Attributen eine FD besteht
23      List<Pair<List<String>, String>> fds =
        DependencyReader.fdTest(groList);
24      if(fds.isEmpty()) {
25        break;
26      }
```

Quellcodeausschnitt 4.4: Ausschnitt 3 der Funktion rewriteD01.

In dem Quellcodeausschnitt 4.4 ist die Überprüfung von zwei Umformungsbedingungen dargestellt. In den Zeilen 18 bis 20 wird überprüft, ob über mehr als zwei Attribute gruppiert wird. Wenn über weniger Attribute gruppiert wird, wird die Umformung abgebrochen. In der Zeile 23 wird die Funktion `fdTest` aufgerufen. Diese liefert alle funktionalen Abhängigkeiten, welche in den übergebenen Attributen vorhanden sind. In den Zeilen 24 bis 26 wird überprüft, ob mindestens eine FD vorliegt. Ist dies nicht der Fall, wird die Umformung für diesen Knoten beendet.

```
27      Node coppyDep = findOperatorV2(group,
        "groupByAttributes/attribute[text()='" + fds.get(0).right +
        "']").cloneNode(true);
28
29      //abhängige Attribute entfernen
30      Node enf = findOperatorV2(group,
        "groupByAttributes/attribute[text()='" + fds.get(0).right + "']");
31      enf.getParentNode().removeChild(enf);
```

Quellcodeausschnitt 4.5: Ausschnitt 4 der Funktion rewriteD01.

In der Zeile 27 des Quellcodeausschnitts 4.5 wird das abhängige Attribut der ersten FD kopiert. In den Zeilen 30 und 31 wird das abhängige Attribut aus der Gruppierung entfernt.

```
32      //Projection mit gestrichenen Attributen erstellen
33      Element newProj = input.createElement("projection");
34      Node attributes = input.createElement("attributes");
35      attributes.appendChild(coppyDep);
36      for(int ii=0; ii<fds.get(0).left.size(); ii++) {
37        attributes.appendChild(findOperatorV2(group,
        "groupByAttributes/attribute[text()='"+fds.get(0).left.get(ii)+"']")
38          .cloneNode(true));
39      }
40      newProj.appendChild(attributes);
41      newProj.setAttribute("isDistinct", "false");
42      newProj.appendChild(group.getChildNodes().item(1).cloneNode(true));
```

Quellcodeausschnitt 4.6: Ausschnitt 5 der Funktion rewriteD01.

In dem Quellcodeausschnitt 4.6 wird eine Projektion auf die bestimmenden Attribute und das abhängige Attribut erzeugt. In der Zeile 33 wird der Projektionsknoten und in der Zeile 34 der Knoten `attributes`, welcher die Projektionsattribute beinhaltet, erzeugt. In den Zeilen 35 bis 39 werden die Projektionsattribute an diesen angefügt. In den Zeilen 40 bis 42 werden die Knoten an die Projektion angehängt.

```
43      //join erstellen und einfügen
44      Element newJoin = input.createElement("naturaljoin");
45      newJoin.appendChild(newProj);
46      newJoin.appendChild(group.cloneNode(true));
47      group.getParentNode().replaceChild(newJoin, group);
48    }
49    return cleanUp(input);
50  }
```

Quellcodeausschnitt 4.7: Ausschnitt 6 der Funktion rewriteD01.

In dem Quellcodeausschnitt 4.7 wird in den Zeilen 44 bis 46 ein neuer Verbund erstellt, welcher die Projektion und die Gruppierung ohne das abhängige Attribut enthält. In der Zeile 47 wird die Gruppierung im Dokument durch den Verbund ersetzt. In der Zeile 49 wird schließlich das veränderte Dokument zurückgegeben.
In der Implementation der Umformung D01 wird die Funktion `fdTest` der Klasse `DependencyReader` aufgerufen. Diese Funktion testet, ob in der übergebenen Menge an Attributen eine funktionale Abhängigkeit besteht. In den folgenden Quellcodeausschnitten wird die Implementation der Funktion dargestellt.

```
1   public static List<Pair<List<String>,String>> fdTest(List<String>
    attributes){
2     List<FunctionalDependency> fds = getFds();
3     List<Pair<List<String>,String>> enthalteneFDs = new
      ArrayList<Pair<List<String>, String>>();
4
5     for(int i=0; i<fds.size(); i++){
6       String depend ="";
7       List<String> determList = new ArrayList<String>();
8       boolean exDep = false;
9       boolean exDet = true;
```

```
10
11      //erzeugen einer Liste mit allen "bestimmenden" Attributen
12      List<ColumnIdentifier> help =
        List.copyOf(fds.get(i).getDeterminant().getColumnIdentifiers());
13      List<String> deter = new ArrayList<String>();
14      for(int ii=0; ii<help.size(); ii++) {
15        deter.add(help.get(ii).toString());
16      }
```

Quellcodeausschnitt 4.8: Ausschnitt 1 der Implementation der Funktion fdTest.

Am Beginn der Funktion `fdTest`, welcher in dem Quellcodeausschnitt 4.8 dargestellt ist, werden in Zeile 2 alle abgelegten FDs mit der Funktion `getFds` eingelesen. In der Zeile 3 wird die Liste `enthalteneFDs` erstellt, in welche die zurückzugebenden FDs abgelegt werden sollen. In der Zeile 5 wird eine Schleife gestartet, welche alle eingelesenen Abhängigkeiten durchläuft. In den Zeilen 12 bis 16 wird eine Liste erstellt, welche alle bestimmenden Attribute der FD als String beinhaltet.

```
17      //testen, ob das abhängige Attribut in der Menge der übergebenen
        Attribute ist
18      for(int ii=0; ii<attributes.size(); ii++){
19        if(fds.get(i).getDependant().getColumnIdentifier()
          .endsWith(attributes.get(ii))){
20          exDep=true;
21          depend = attributes.get(ii);
22        }
23      }
24
25      //testen, ob die bestimmenden Attribute in der Attributmenge sind
26      if(exDep&&(deter.size()<attributes.size())) {
27        for(int ii=0; ii<deter.size();ii++){
28          boolean h2 = false;
29          for(int iii=0; iii<attributes.size(); iii++){
30            if(deter.get(ii).contains(attributes.get(iii))) {
31              h2= true;
32              determList.add(attributes.get(iii));
33            }
34          }
35          exDet = exDet&h2;
36        }
37        if(exDet) {
38          enthalteneFDs.add(new Pair<List<String>, String>(determList,
            depend));
39        }
40      }
41    }
42    return enthalteneFDs;
43  }
```

Quellcodeausschnitt 4.9: Ausschnitt 2 der Implementation der Funktion fdTest.

In dem Quellcodeausschnitt 4.9 wird in den Zeilen 18 bis 23 getestet, ob das abhängige Attribut der

FD in der Menge der übergebenen Attribute ist. Wenn dies der Fall ist, wird die Variable `exDep` mit `true` belegt und das abhängige Attribut in der Variable `depend` hinterlegt. In den Zeilen 26 bis 36 wird getestet, ob alle bestimmenden Attribute in der Menge der übergebenen Attribute sind und diese in der Variable `determList` abgelegt. Ist dies der Fall wird in den Zeilen 37 bis 39 die FD der List `enthalteneFDs` hinzugefügt. Wenn alle eingelesenen FDs durchlaufen sind, werden mit der Zeile 42 alle enthaltenen FDs zurückgegeben.

4.5 Beispiel der Umformungen auf XML-Ebene

In diesem Ausschnitt wird die aus dem Abschnitt 3.2 bekannte Beispielanfrage aufgegriffen. In diesem wurden die im Konzept vorgestellten Äquivalenzen genutzt, um die Anfrage auf Ebene der relationalen Algebra zu verändern. Im Folgendem wird gezeigt, wie die Anfrage vor und nach den Umformungen in der XML-Darstellung aussieht. Die für das Beispiel genutzte Anfrage hat in SQL die Form:

```
SELECT s.Name, s.Address, n.Name, count(Partkey)
FROM Partsupp ps, Supplier s, Nation n
WHERE ps.Suppkey = s.Suppkey
AND s.Nationkey = n.Nationkey
GROUP BY n.Name, s.Suppkey, s.Address
```

Die Anfrage wird vor dem Anwenden der Umformungen in eine XML-Darstellung gebracht. Diese sieht wie folgt aus:

```
1  <projection isDistinct="false">
2    <selection>
3      <grouping>
4        <naturaljoin>
5          <baseRelation name="Partsupp">
6            <attributes>
7              <attribute>Partkey</attribute>
8              <attribute>Name</attribute>
9              <attribute>MFGR</attribute>
10             <attribute>Brand</attribute>
11             <attribute>Type</attribute>
12             <attribute>Size</attribute>
13             <attribute>Container</attribute>
14             <attribute>Retailprice</attribute>
15             <attribute>Comment</attribute>
16           </attributes>
17         </baseRelation>
18         <naturaljoin>
19           <baseRelation name="Supplier">
20             <attributes>
21               <attribute>Suppkey</attribute>
22               <attribute>Name</attribute>
23               <attribute>Address</attribute>
24               <attribute>Nationkey</attribute>
25               <attribute>Phone</attribute>
26               <attribute>Acctbal</attribute>
```

```
27                <attribute>Comment</attribute>
28              </attributes>
29            </baseRelation>
30            <baseRelation name="Nation">
31              <attributes>
32                <attribute>Nationkey</attribute>
33                <attribute>Name</attribute>
34                <attribute>Regionkey</attribute>
35                <attribute>Comment</attribute>
36              </attributes>
37            </baseRelation>
38          </naturaljoin>
39        </naturaljoin>
40        <aggregates>
41          <aggregate name="count" attribute="Partkey"/>
42        </aggregates>
43        <groupByAttributes>
44          <attribute>Name</attribute>
45          <attribute>Suppkey</attribute>
46          <attribute>Address</attribute>
47        </groupByAttributes>
48      </grouping>
49    </selection>
50      <attributes>
51        <attribute>SName</attribute>
52        <attribute>Address</attribute>
53        <attribute>NName</attribute>
54        <attribute>count(Partkey)</attribute>
55      </attributes>
56  </projection>
```

In den Zeilen 5 bis 17 wird die Basisrelation *Partsupp* dargestellt. Wenn Basisrelationen dargestellt werden, werden alle Attribute, welche in der Relation vorhanden sind, aufgelistet. In der Darstellung der Basisrelation *Partsupp*, sind das die Zeilen 7 bis 15. Für eine Gruppierung wird das Element `grouping` verwendet. Dieses hat drei Unterelemente: die Relation, über welche die Gruppierung ausgeführt wird, das Element `aggregates`, welches die Aggregatsfunktionen beinhaltet und das Element `groupByAttributes` beinhaltet die Attribute, über welchen gruppiert wird.
Bei der Umformung der Anfrage wird als erstes die Umformung D02 ausgeführt, danach die Umformung D01. Die, durch diese Umformungen entstandenen Verbünde, werden durch die Anwendung der Umformung D05 zusammengeführt. Als letztes wird die Umformung D03 ausgeführt. Dies führt zu der folgenden XML-Darstellung:

```
1  <projection isDistinct="false">
2    <naturaljoin>
3      <grouping>
4        <naturaljoin>
5          <baseRelation name="Partsupp">
6            <attributes>
7              <attribute>Partkey</attribute>
8              <attribute>Name</attribute>
```

```
9             <attribute>MFGR</attribute>
10            <attribute>Brand</attribute>
11            <attribute>Type</attribute>
12            <attribute>Size</attribute>
13            <attribute>Container</attribute>
14            <attribute>Retailprice</attribute>
15            <attribute>Comment</attribute>
16          </attributes>
17        </baseRelation>
18        <naturaljoin>
19          <baseRelation name="Supplier">
20            <attributes>
21              <attribute>Suppkey</attribute>
22              <attribute>Name</attribute>
23              <attribute>Address</attribute>
24              <attribute>Nationkey</attribute>
25              <attribute>Phone</attribute>
26              <attribute>Acctbal</attribute>
27              <attribute>Comment</attribute>
28            </attributes>
29          </baseRelation>
30          <baseRelation name="Nation">
31            <attributes>
32              <attribute>Nationkey</attribute>
33              <attribute>Name</attribute>
34              <attribute>Regionkey</attribute>
35              <attribute>Comment</attribute>
36            </attributes>
37          </baseRelation>
38        </naturaljoin>
39      </naturaljoin>
40      <aggregates>
41        <aggregate name="count" attribute="Partkey"/>
42      </aggregates>
43      <groupByAttributes>
44        <attribute>Name</attribute>
45        <attribute>Suppkey</attribute>
46        <attribute>Address</attribute>
47      </groupByAttributes>
48    </grouping>
49    <projection>
50      <naturaljoin>
51        <baseRelation name="Supplier">
52          <attributes>
53            <attribute>Suppkey</attribute>
54            <attribute>Name</attribute>
55            <attribute>Address</attribute>
56            <attribute>Nationkey</attribute>
57            <attribute>Phone</attribute>
58            <attribute>Acctbal</attribute>
59            <attribute>Comment</attribute>
```

```
60            </attributes>
61          </baseRelation>
62          <baseRelation name="Partsupp">
63            <attributes>
64              <attribute>Partkey</attribute>
65              <attribute>Name</attribute>
66              <attribute>MFGR</attribute>
67              <attribute>Brand</attribute>
68              <attribute>Type</attribute>
69              <attribute>Size</attribute>
70              <attribute>Container</attribute>
71              <attribute>Retailprice</attribute>
72              <attribute>Comment</attribute>
73            </attributes>
74          </baseRelation>
75        </naturaljoin>
76        <attributes>
77          <attribute>Suppkey</attribute>
78          <attribute>Name</attribute>
79          <attribute>Address</attribute>
80        </attributes>
81      </projection>
82    </naturaljoin>
83    <attributes>
84      <attribute>SName</attribute>
85      <attribute>Address</attribute>
86      <attribute>NName</attribute>
87      <attribute>count(Partkey)</attribute>
88    </attributes>
89  </projection>
```

Kapitel 5

Evaluation der Implementation

Um die Implementation der Umformungen zu testen, wurden JUnit-Tests in der Klasse `Test_D` implementiert. In diesen wird eine Anfrage umgeschrieben und getestet, ob das erwartete Ergebnis zurückgegeben wird. In dem ersten Abschnitt wird der Ablauf eines Tests beispielhaft an dem Test `testD01` dargestellt. Im Abschnitt 5.2 werden alle Ausgangsanfragen und die erwarteten Ergebnisse der Umformungen dargestellt.

5.1 Aufbau der JUnit-Tests

Der Aufbau der JUnit-Tests wird am Beispiel des Tests `testD01` dargestellt. Die Tests bestehen immer aus drei Schritten: erzeugen der Ausgangsanfrage, erzeugen der Zielanfrage und Vergleich der Ausgangsanfrage nach dem Ausführen der Umformung mit der Zielanfrage.

```
1  public void testD01() throws Exception {
2
3    //Ausgangsanfrage erzeugen:
4    BaseRelation tracks = createTrackRelation();
5    ArrayList<Aggregate> aggregates = new ArrayList<Aggregate>();
6    Aggregate agg = new Aggregate("min", "length");
7    aggregates.add(agg);
8    ArrayList<String> groupByAttributes = new ArrayList<String>();
9    groupByAttributes.add("album_id");
10   groupByAttributes.add("track_id");
11   Grouping group = new Grouping(tracks, aggregates, groupByAttributes);
12   Document doc = generateDomFromPojo(group);
13
14   //Rewirte durchführen:
15   doc = rewriter.rewriteD01(doc, null);
```

Quellcodeausschnitt 5.1: Erster Abschnitt des JUnit-Tests zum Testen der Funktion rewriteD01, welche die Umformung D01 implementiert.

In dem Quellcodeausschnitt 5.1 wird der erste Schritt des JUnit-Test `testD01` dargestellt. In den Zeilen 4 bis 11 wird die Ausgangsfrage erzeugt. In Zeile 12 wird mit der Funktion `generateDomFromPojo` ein XML-Dokument erzeugt, auf welchen in der Zeile 15 die Umformung durchgeführt wird.

```
16    //Zielanfrage erzeugen:
17    ArrayList<String> groupByAttributes2 = new ArrayList<String>();
18    groupByAttributes2.add("track_id");
19    Grouping group2 = new Grouping(tracks, aggregates, groupByAttributes2);
20    ArrayList<String> prList = new ArrayList<String>();
21    prList.add("album_id");
22    prList.add("track_id");
23    Projection pr = new Projection(tracks, prList, false);
24    NaturalJoin join = new NaturalJoin(pr, group2);
25    Document doc2 = generateDomFromPojo(join);
```

Quellcodeausschnitt 5.2: Abschnitt 2 von dem JUnit-Test zum Testen der Funktion rewriteD01, welche die Umformung D01 implementiert.

In dem Quellcodeausschnitt 5.2 wird die Erzeugung der Zielanfrage dargestellt, um diese später mit der Ausgangsanfrage nach der Anwendung der Umformung zu vergleichen.

```
26    // Teil 5 - Ausgaben vergleichen (immer gleich)
27    // leider ist Rewriting auf XML-Ebene nicht ordnungserhaltend, daher
      einmal hin- und zurückkonvertieren...
28    Relation test = generatePojoFromXml(doc);
29    Document docTest = generateDomFromPojo(test);
30
31    String original = this.document2String(docTest);
32    String rewritten = this.document2String(doc2);
33    assertEquals(original, rewritten);
34  }
```

Quellcodeausschnitt 5.3: Abschnitt 3 von dem JUnit-Test zum Testen der Funktion rewriteD01, welche die Umformung D01 implementiert.

In dem dritten Quellcodeausschnitt 5.3 wird der letzte Schritt des Tests durchgeführt. Da die Umformungen mit dem simple-Framework nicht ordnungserhaltend sind, wird in den Zeilen 28 und 29 aus dem XML-Dokument eine POJO-Darstellung und aus dieser wieder ein XML-Dokument erstellt. In den Zeilen 31 bis 33 werden die Anfragen verglichen, wofür sie mit der Funktion `document2String` in einen String verwandelt werden.

5.2 Testfälle

Da die einzelnen Tests sich nur in der Ausgangs- und Zielanfrage unterscheiden, werden im Folgenden nur diese für die verschiedenen Anfragen aufgelistet. Für die Testfälle wurden Relationen aus dem Amarok-Schema[1] verwendet, da dieses für bereits implementierte Tests genutzt wurde. Die verwendeten Relationen mit ihren Attributen lauten:

- $Album = \{album\_id, album\_name, artist\_id, image\}$
- $Artist = \{artist\_id, artist\_name\}$
- $Composer = \{composer\_id, composer\_name\}$

[1] wird in dem Musikplayer Amarok genutzt `https://amarok.kde.org/de`, zuletzt aufgerufen am 02.04.2021 um 18:46

- $Tracks = \{track\_id, artist\_id, album\_id, genre\_id, composer\_id, year\_id, title, url, comment, tracknumber, discnumber, bitrate, length, samplerate, filesize, filetype, bpm, createdate, modifydate, albumgain, albumpeakgain, trackgain, trackpeakgain\}$

Damit die Tests funktionieren, wurden folgende funktionale Abhängigkeiten für die Relationen *Tracks* und *Composer* hinterlegt:

- $track\_id \rightarrow album\_id$
- $track\_id \rightarrow title$
- $track\_id \rightarrow comment$
- $track\_id \rightarrow length$
- $track\_id \rightarrow bpm$
- $composer\_id \rightarrow composer\_name$
- $composer\_name \rightarrow composer\_id$

Für die Tests der Umformungen D03 und D04 wurden die beiden Inklusionsabhängigkeiten $Tracks[composer\_id] \subseteq Composer[composer\_id]$ und $Tracks[artist\_id] \subseteq Artists[atrist\_id]$ hinterlegt.

testD01 Als Test für die Implementation der Umformung D01 wird als Ausgangsanfrage eine Gruppierung über der Relation `Tracks` verwendet. Für die Umformung wurde außerdem die FD $track_id \rightarrow album_id$ hinterlegt. Der Anfragebaum vor und nach der Anfrage wurde in der Abbildung 5.1 dargestellt.

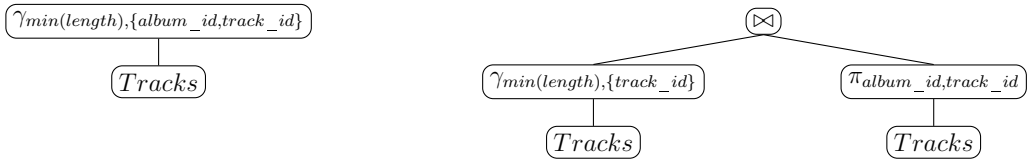


Abbildung 5.1: Links ist die verwendete Ausgangsanfrage und rechts die erwartete Anfrage nach der Umformung für den Test der Funktion *testD01*.

testD02 Für den Test der Implementation der Umformung D02 wird als Ausgangsanfrage eine Gruppierung über der Relation `Composers` verwendet. Für die Umformung wurden außerdem die FDs $composer\_id \rightarrow composer\_name$ und $composer\_name \rightarrow composer\_id$ hinterlegt. Der Anfragebaum 5.2 vor und nach der Anfrage wurde in der Abbildung dargestellt.

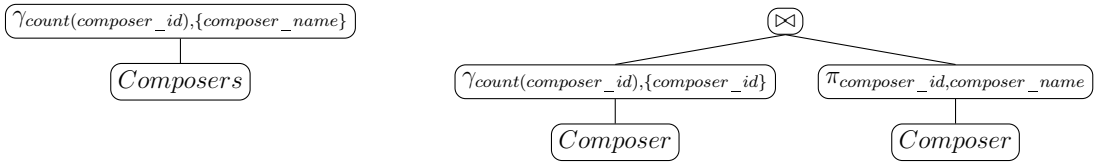


Abbildung 5.2: Links ist die verwendete Ausgangsanfrage und rechts die erwartete Anfrage nach der Umformung für den Test der Funktion *testD02*.

testD03 Für den Test der Funktion `testD03` wird eine Ausgangsanfrage mit einen Verbund über die beiden Relationen `Composer` und `Tracks` verwendet. Außerdem wurde die IND $Tracks[composer\_id] \subseteq Composer[composer\_id]$ hinterlegt. Die als Ergebnis zu erwartende Anfrage beinhaltet nur noch die Relation `Tracks`. Die Anfragen sind als Bäume in der Abbildung 5.3 dargestellt.

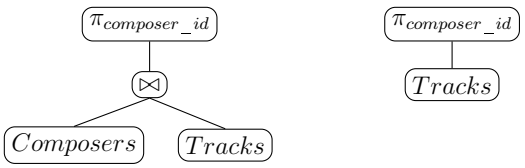


Abbildung 5.3: Links ist die verwendete Ausgangsanfrage und rechts die erwartete Anfrage nach der Umformung für den Test der Funktion *testD03* abgebildet.

testD04 Für den Test der Funktion `testD04` werden die Relationen `Album`, `Tracks` und `Artists` verwendet. Außerdem wurde die IND $Tracks[artist\_id] \subseteq Artists[atrist\_id]$ hinterlegt. Die Anfragen sind als Bäume in der Abbildung 5.4 dargestellt.

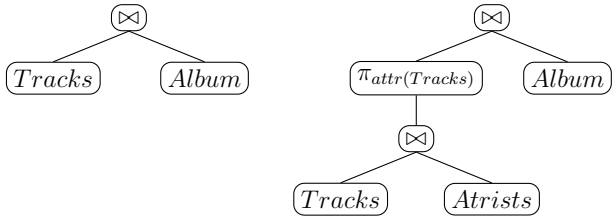


Abbildung 5.4: Links ist die verwendete Ausgangsanfrage und rechts die erwartete Anfrage nach der Umformung für den Test der Funktion *testD04* abgebildet. Um die Darstellung nicht zu überladen, wird die aus dem Abschnitten 3 bekannte Funktion *attr()* verwendet.

testD05 Für den Test der Funktion `testD05` wird nur die Relation `Tracks` und die FDs: $track_id \rightarrow title$, $track_id \rightarrow comment$, $track_id \rightarrow length$ und $track_id \rightarrow bpm$ verwendet. In der Abbildung 5.5 sind die Anfragebäume von der Ausgangsanfrage und dem erwarteten Ergebnis dargestellt.

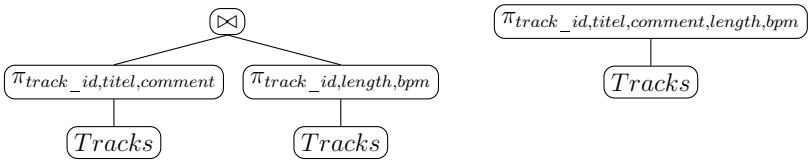


Abbildung 5.5: Links ist die verwendete Ausgangsanfrage und rechts die erwartete Anfrage nach der Umformung für den Test der Funktion *testD05* abgebildet.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Arbeit wurden funktionale und Inklusionsabhängigkeiten und deren Nutzbarkeit für das Umformen von Anfragen betrachtet. Hierzu wurde sich als erstes mit dem Finden von Abhängigkeiten in einem bestehenden Datensatz beschäftigt. Sowohl für die FDs als auch die INDs wurden Algorithmen, welche einen Datensatz nach Abhängigkeiten durchsuchen, vorgestellt und verglichen. Die Problemstellungen des AQuV und des AQuO wurden vorgestellt. Für das AQuV-Problem wurden bereits bestehende Methoden zur Reformulierung von Anfragen mit Beachtung von FDs und INDs vorgestellt.
Im weiteren wurden fünf Äquivalenzen in der relationalen Algebra erarbeitet, welche unter bestimmten FDs und INDs gelten. Zwei der Äquivalenzen sind hauptsächlich über den Gruppierungsoperator definiert. Die anderen drei Äquivalenzen sind über den Verbundoperator definiert. Ausgehend von den Äquivalenzen wurden Umformungsregeln erarbeitet, welche für die Anwendung im AQuO-Problem genutzt werden sollen. In den ersten beiden Umformungen werden Gruppierungsattribute ersetzt oder entfernt. Damit das Ergebnis der Gruppierung gleich bleibt, werden funktionale Abhängigkeiten vorausgesetzt. Mit der dritten Umformung kann ein Verbund aus einer Anfrage entfernt werden, wenn die entsprechende Inklusionsabhängigkeit besteht. Die vierte Umformung ermöglicht das Hinzufügen von einem Verbund, um Tupel vorzusortieren. Mit der letzten Umformung lässt sich wie bei der dritten ein Verbund entfernen, nur dass hier funktionale und keine Inklusionsabhängigkeiten benötigt werden. Die vorgestellten Umformungen wurden außerdem zur Anwendung einem bestehenden Rewriter hinzugefügt und die Implementation mittels JUnit-Tests getestet.

Neben den in dieser Arbeit als Vorbedingungen genutzten funktionalen und Inklusionsabhängigkeiten gibt es weitere Klassen von Abhängigkeiten, wie mehrwertige und transitive Abhängigkeiten, welche auf einen Nutzen für Umformungen getestet werden könnten. In der Arbeit [Tür99] werden sogenannte "linear arithmetic constraints" beschrieben, welche auch betrachtet werden könnten. Außerdem gäbe es die Möglichkeit, nicht nur Abhängigkeiten einer Klasse als Vorbedingung zu nutzen, sondern diese mit weiteren Vorbedingungen zu kombinieren.

Literaturverzeichnis

[AC19] Afrati, Foto N. und Rada Chirkova: *Answering Queries Using Views, Second Edition.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.

[ASN14] Abedjan, Ziawasch, Patrick Schulze und Felix Naumann: *DFD: Efficient Functional Dependency Discovery.* In: *CIKM*, Seiten 949–958. ACM, 2014.

[Cou16] Council of European Union: *VERORDNUNG (EU) 2016/679 DES EUROPÄISCHEN PARLAMENTS UND DES RATES vom 27. April 2016 zum Schutz natürlicher Personen bei der Verarbeitung personenbezogener Daten, zum freien Datenverkehr und zur Aufhebung der Richtlinie 95/46/EG*, 2016.

[DMLP02] De Marchi, Fabien, Stéphane Lopes und Jean-Marc Petit: *Efficient Algorithms for Mining Inclusion Dependencies.* In: Jensen, Christian S., Simonas Šaltenis, Keith G. Jeffery, Jaroslav Pokorny, Elisa Bertino, Klemens Böhn und Matthias Jarke (Herausgeber): *Advances in Database Technology — EDBT 2002*, Seiten 464–476, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[DSW+19] Dürsch, Falco, Axel Stebner, Fabian Windheuser, Maxi Fischer, Tim Friedrich, Nils Strelow, Tobias Bleifuss, Hazar Harmouch, Lan Jiang, Thorsten Papenbrock und Felix Naumann: *Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen Algorithms.* In: *CIKM*, Seiten 219–228. ACM, 2019.

[FS99] Flach, Peter A. und Iztok Savnik: *Database Dependency Discovery: A Machine Learning Approach.* AI Commun., 12(3):139–160, 1999.

[GH16] Grunert, Hannes und Andreas Heuer: *Datenschutz im PArADISE.* Datenbank-Spektrum, 16(2):107–117, 2016.

[GH17] Grunert, Hannes und Andreas Heuer: *Rewriting Complex Queries from Cloud to Fog under Capability Constraints to Protect the Users' Privacy.* Open J. Internet Things, 3(1):31–45, 2017.

[Gry99] Gryz, Jarek: *Query Rewriting Using Views in the Presence of Functional and Inclusion Dependencies.* Inf. Syst., 24(7):597–612, 1999.

[HKPT99] Huhtala, Ykä, Juha Kärkkäinen, P. Porkka und Hannu Toivonen: *TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies.* Comput. J., 42:100–111, 1999.

[HSS18] Heuer, Andreas, Gunter Saake und Kai-Uwe Sattler: *Datenbanken - Konzepte und Sprachen, 6. Auflage.* MITP, 2018.

[KPD+17] Kruse, Sebastian, Thorsten Papenbrock, Christian Dullweber, Moritz Finke, Manuel Hegner, Martin Zabel, Christian Zöllner und Felix Naumann: *Fast Approximate Discovery of Inclusion Dependencies.* In: *Proceedings of the conference on Database Systems for Business, Technology, and Web (BTW)*, Seiten 207–226, 2017.

[LMSS95] Levy, Alon Y., Alberto O. Mendelzon, Yehoshua Sagiv und Divesh Srivastava: *Answering Queries Using Views.* In: Yannakakis, Mihalis und Serge Abiteboul (Herausgeber): *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California, USA*, Seiten 95–104. ACM Press, 1995. LNSS95.

[LPL00] Lopes, Stéphane, Jean-Marc Petit und Lotfi Lakhal: *Efficient Discovery of Functional Dependencies and Armstrong Relations.* In: *EDBT*, Band 1777 der Reihe *Lecture Notes in Computer Science*, Seiten 350–364. Springer, 2000.

[NC01] Novelli, Noël und Rosine Cicchetti: *FUN: An Efficient Algorithm for Mining Functional and Embedded Dependencies.* In: *ICDT*, Band 1973 der Reihe *Lecture Notes in Computer Science*, Seiten 189–203. Springer, 2001.

[PEM+15] Papenbrock, Thorsten, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener und Felix Naumann: *Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms.* Proc. VLDB Endow., 8(10):1082–1093, 2015.

[SSH11] Saake, Gunter, Kai-Uwe Sattler und Andreas Heuer: *Datenbanken - Implementierungstechniken, 3. Auflage.* MITP, 2011.

[TPN17] Tschirschnitz, Fabian, Thorsten Papenbrock und Felix Naumann: *Detecting Inclusion Dependencies on Very Many Tables.* ACM Trans. Database Syst., 42(3), Juli 2017.

[Tür99] Türker, Can: *Semantic Integrity Constraints in Federated Database Schemata*, Band 63 der Reihe *DISDBIS*. Infix Verlag, St. Augustin, Germany, 1999.

[WGR01] Wyss, Catharine, Chris Giannella und Edward Robertson: *FastFDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances Extended Abstract.* In: Kambayashi, Yahiko, Werner Winiwarter und Masatoshi Arikawa (Herausgeber): *Data Warehousing and Knowledge Discovery*, Seiten 101–110, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[YHB02] Yao, Hong, Howard J. Hamilton und Cory J. Butz: *FD_Mine: Discovering Functional Dependencies in a Database Using Equivalences.* In: *ICDM*, Seiten 729–732. IEEE Computer Society, 2002.

Anhang A

Hinweise zur digitalen Version

In der digitalen Version im StudIp befinden sich zusätzliche Materialien zu dieser Arbeit. Der Aufbau und Inhalt dieser Version soll hier kurz dargestellt werden.

- **Bachelorarbeit_Max_Kaseler** ist diese Bachelorarbeit als eine PDF-Datei.
- **Quellen** Beinhaltet verwendeten Quellen als PDF-Datei. Internetseiten auf welche verwiesen wurde, sind in dem Unterordner *Websites* aufzufinden.
- **Rewriter** beinhaltet den zur Abgabe aktuellen Stand des Rewriters. Der Aktuelle Stand ist im Git-Reposetory `https://git.informatik.uni-rostock.de/hg/patch-queryrewriter` verfügbar.

Selbständigkeitserklärung

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

Rostock, den 3. April 2021