Masterarbeit

Anonymisierung von Data Provenance in ProSA

Eingereicht von: Nic Scharlau, B. Sc.

Eingereicht am: 15. März 2022

Gutachter/-innen: Tanja Auge, M. Sc.
Dr.-Ing. Thomas Mundt

Betreuerin: Tanja Auge, M. Sc.

Zusammenfassung

Das an der *Universität Rostock* angesiedelte Forschungsprojekt *ProSA*, bestehend aus der gleichnamigen Software, hat das Ziel, ausgehend von einer Forschungsdatenbank eine **minimale Teildatenbank** ebendieser Datenbank zu berechnen, welche in der Lage ist, Forschungsergebnisse langfristig replizieren und rekonstruieren zu können. Dabei wird das Konzept der **Data Provenance** genutzt, um all diejenigen Tupel (sogenannte Zeugen) einer oder mehrerer Relationen bestimmen zu können, welche für das Zustandekommen des Ergebnisses unentbehrlich sind. Allerdings können nach einer solchen Rekonstruktion gewisse Datenschutz-Anforderungen verletzt werden. Dies kann insbesondere auch nach dem Entfernen eindeutig identifizierender Attribute weiterhin der Fall sein, wenn sich im Datensatz sogenannte **Quasi-Identifikatoren** befinden, welche mit externem Wissen Rückschlüsse auf natürliche Personen ermöglichen. Doch auch bei Daten ohne Personenbezug kann eine Anonymisierung sinnvoll sein, wenn diese Daten als besonders schützenswert betrachtet werden. Eine Lösung für *ProSA* gibt es bislang nicht und auch die Forschung zur Kombination von Data Provenance und Datenschutz (**Privacy**) ist vage. Das Ziel der Masterarbeit *„Anonymisierung von Data Provenance in ProSA“* ist es, die verschiedenen Stellen des *ProSA*-Workflows zu untersuchen, an denen eine Anonymisierung der Daten möglich und sinnvoll ist, geeignete Anonymisierungsmethoden sowie Anonymitätsmaße zu bestimmen und *ProSA* um eine praktische Implementierung zu erweitern, welche Privacy-Aspekte gewährleistet. Dazu wurde das Modul *ProSAnon* (*„ProSA anonymizer“*) entwickelt, welches im *ProSA*-Workflow nach der Berechnung der minimalen Teildatenbank eingesetzt wird. Als Anonymisierungsmethode kommt dabei die Generalisierung von Tupeln mittels **Konzepthierarchien**, als Anonymitätsmaß die k-Anonymität zum Einsatz.

Abstract

The research project *ProSA*, settled at the *University of Rostock*, consisting of the same-named software, aims at calculating a **minimal subdatabase** of a research database which is then able to replicate and reconstruct research results in the long term. The concept of **data provenance** is used to determine all those tuples (so-called witnesses) of one or more relations which are indispensable for the occurrence of the result. However, certain data protection requirements may be violated after such a reconstruction. This can still be the case, especially after the removal of uniquely identifying attributes, if there are so-called **quasi-identifiers** in the dataset that allow conclusions to be drawn about natural persons with external knowledge. However, anonymization can also be useful for data without personal references if this data is considered to be particularly worthy of protection. So far, there is no solution for *ProSA* and research on the combination of data provenance and **privacy** is vague. The goal of the master thesis *“Anonymization of Data Provenance in ProSA”* is to investigate the different points in the *ProSA* workflow where anonymization of data is possible and useful, to determine suitable anonymization methods as well as anonymity measures, and to extend *ProSA* with a practical implementation that ensures privacy aspects. For this purpose, the module *ProSAnon* (*“ProSA anonymizer”*) has been developed and is used in the *ProSA* workflow after the calculation of the minimal subdatabase. The anonymization method used is the generalization of tuples with **concept hierarchies**, and the chosen anonymity measure is k-anonymity.

Inhaltsverzeichnis

1. Einleitung

Die vorliegende Masterarbeit „Anonymisierung von Data Provenance in ProSA“ beschäftigt sich mit der datenschutzkonformen Anonymisierung von Data-Provenance-Annotationen im an der *Universität Rostock* angesiedelten Forschungsprojekt *ProSA* („Provenance Management durch Schema-Abbildungen und Annotationen“). Ziel von *ProSA* ist es, mithilfe des aus der Datenbanktheorie vielfach bekannten *CHASE*-Algorithmus' sowie inversen Schemaabbildungen (möglichst) genau die Datenmenge zu berechnen, die für die Rückverfolgbarkeit eines Anfrageergebnisses unbedingt erforderlich ist und umgekehrt all jene Datensätze zu eliminieren, die für das Ergebnis nicht benötigt werden [AH19]. Dies kann in der Praxis sowohl durch Speicherplatzoptimierung als auch durch Datensparsamkeit motiviert sein.

Data Provenance bezeichnet die Rückverfolgung eines Ergebnis-Datensatzes bis zu den originalen Datensätzen. Übertragen auf (relationale) Datenbanken bedeutet dies, dass aus dem Ergebnis einer Datenbankanfrage all jene Tupel rekonstruiert können werden müssen, die zu dem Ergebnis beigetragen haben. Dazu ist es notwendig, zusätzlich zum Ergebnis sogenannte Provenance-Annotationen zu erfassen. Im Falle der Data Provenance handelt es sich dabei um Tupel- und Relationen-Identifikatoren, (minimale) Zeugenmengen und -basen sowie um Provenance-Polynome [GT17]. Ausgehend von der Art der Annotationen können damit drei wesentliche Fragen der Data Provenance beantwortet werden [CCT09]: (1) ***Where***-Provenance: Woher stammen die Daten? (2) ***Why***-Provenance: Warum kam gerade dieses Ergebnis zustande? (3) ***How***-Provenance: Wie genau wurde das Ergebnis berechnet?

Allgemein gilt, dass umso mehr Originaldaten rekonstruiert werden können, je granularer die zusätzlich gespeicherten Informationen sind. So liefert beispielsweise die ***where***-Provenance unter Verwendung von einfachen Tupel-Identifikatoren oder Relationennamen Informationen über die direkt verwendeten Tupel oder Relationen, während die ***how***-Provenance unter Verwendung von Provenance-Polynomen Informationen über alle direkt sowie indirekt genutzten Tupel (beispielsweise Verbundattribute, die selbst nicht im Ergebnis auftauchen) und deren exakten Einfluss auf das Ergebnis liefert.

Dem gegenüber stehen verschiedene Ziele der **Privacy**, darunter vor allem Datenschutz und Datensparsamkeit. Eine umfangreiche oder gar vollständige Rekonstruktion der Originaldaten kann unerwünscht sein, wenn dadurch – bei personenbezogenen Daten – einzelne Personen oder Personengruppen re-identifiziert werden können oder – bei Daten ohne Personenbezug – mehr wertvolle Daten veröffentlicht würden als absolut notwendig. Gerade bei aufwändig erhobenen Forschungsdaten, die oft unter großem monetären und zeitlichen Aufwand erhoben werden, besteht häufig ein Interesse, ungenutzte Datensätze zumindest vorübergehend zurückzuhalten [ASH21b]. Der Privacy-Begriff umfasst in dieser Masterarbeit also sowohl Datensparsamkeit und -schutz als auch den Schutz geistigen Eigentums (engl. *intellectual property*).

Um Datensätze privacy-konform zu transformieren, etablierten sich in den letzten Jahrzehnten einige Anonymisierungstechniken und -maße. Viele davon nutzen die Methode der Generalisierung und Unterdrückung einzelner Zeilen bzw. Tupel. Hierbei wird der Wertebereich einer Spalte verkleinert, indem verschiedene Werte zu einer Kategorie zusammengefasst werden. Die entsprechenden Anonymitätsmaße sind dabei die *k-Anonymität* sowie die *l-Diversität* (siehe [Sam01] und [Swe02b]). Andere Techniken

nutzen bestimmte stochastische Eigenschaften aus, um ein Ergebnis nicht genau, sondern nur näherungsweise zu bestimmen bzw. bereits die Originaldaten hinreichend zu verrauschen (Differential Privacy; siehe [Dwo06]). Besonders im Forschungsdatenmanagement ist diese Methode interessant, da ein damit erfolgreich anonymisierter Datensatz Informationen über eine Gruppe, jedoch nicht über ihre einzelnen Individuen enthält. Doch auch das einfache Permutieren von Spalten oder Zeilen kann bereits eine effektive Datenschutzmaßnahme sein [Sch20].

Das Ziel dieser Masterarbeit ist es, das *ProSA*-Projekt um eine selbst gewählte Anonymisierungsmethode zu erweitern, um oben beschriebene Privacy-Aspekte in der Auswertung von Forschungsdaten zu berücksichtigen und mittels eines geeigneten Anonymisierungsmaßes zu bewerten. Dazu benötigen wir zunächst einen Überblick über verschiedene Techniken, ihre Vor- und Nachteile sowie Anwendungsbereiche. Die entsprechende Vorarbeit dazu ist in der Bachelorarbeit *„Provenance und Privacy in ProSA“* zu finden [Sch20]. Anschließend soll ermittelt werden, an welchen Stellen im Ablauf des *ProSA*-Projekts mit welchen Methoden und Maßen anonymisiert werden kann und wo genau dies überhaupt sinnvoll erscheint. Zu guter Letzt soll eine der gewählten Techniken praktisch implementiert werden.

1.1. Forschungsprojekt ProSA

Gerade aufgrund der immer größer werdenden Datenmengen, die sowohl in der Forschung als auch in der Industrie – beispielsweise in Data Warehouses – anfallen, besteht eine zunehmende Notwendigkeit, die Datenlast zu reduzieren. Gleichzeitig ist es gerade in wissenschaftlichen Auswertungen erforderlich, im Sinne guter wissenschaftlicher Arbeit die Forschungsergebnisse nachvollziehen, reproduzieren und rekonstruieren zu können, was wiederum den Erhalt der Originaldaten voraussetzt. Allerdings reicht es dabei häufig, nur einen Teil des originalen Datensatzes aufzubewahren, nämlich genau den Teil, der zu den Ergebnissen beigetragen hat. Das an der *Universität Rostock* angesiedelte Forschungsprojekt *ProSA*[1] („Provenance Management durch Schema-Abbildungen und Annotationen“) hat deshalb das Ziel, diesen Prozess zu unterstützen, indem zu einer gegebenen Quelldatenbank D, einer Anfrage Q an diese und die resultierende Zieldatenbank $V = Q(D)$ eine minimale Teildatenbank $D' \subseteq D$ berechnet wird, die ausschließlich Tupel enthält, die für die Nachvollziehbarkeit des Ergebnisses V absolut erforderlich sind. Im Umkehrschluss werden all jene Daten eliminiert, die nicht zum Ergebnis beitragen. Dazu werden neben der Anfrage und dem Ergebnis zusätzliche Informationen, sogenannte Provenance-Annotationen, berechnet und gespeichert. Anschließend wird mittels inverser Anfragen, dem *CHASE*-Algorithmus und den Provenance-Annotationen ebendiese minimale Teildatenbank D' berechnet. Doch auch die Annotationen selbst sollen so minimal wie möglich gehalten werden. In einigen Fällen liefert bereits das bloße Invertieren alle nötigen Informationen, in den meisten Fällen trifft dies jedoch nicht zu. Die Annotationen, die dann erfasst werden, sind in der Regel Provenance-Polynome, welche die Zusammenhänge und einzelnen Berechnungsschritte zwischen bzw. mit den beteiligten Tupeln darstellen. Das *ProSA*-Projekt ist Bestandteil der Dissertation *„Provenance Management using Schema Mappings with Annotations (Arbeitstitel)“* von *Tanja Auge*, welche gegenwärtig verfasst wird [Aug22]. Es wurde bereits 2019 im Artikel „ProSA – Using the CHASE for Provenance Management“ vorgestellt [AH19].

[1] `https://dbis.informatik.uni-rostock.de/forschung/aktuelle-projekte/prosa/`, zuletzt abgerufen am 27.10.2021 15:38+0200

1.2. Die Beispieldatenbank

Wir betrachten in dieser Masterarbeit ein fortlaufendes Beispiel, um alle Überlegungen und Theorien zu erläutern. Dazu wird eine fiktive, stark vereinfachte, Datenbank einer Universität aufgestellt, welche aus den drei Relationen

- STUDENTS(`student_id`, `lastname`, `firstname`, `birthday`, `zipcode`, `district`, `course`),
- MODULES(`module_id`, `name`) und
- EXAMS(`student_id`, `module_id`, `semester`, `grade`)

besteht und auf den Seiten 10 und 11 dargestellt sind. Die STUDENTS-Relation, zu sehen in Tabelle 1.1, beinhaltet Informationen über alle immatrikulierten Studierenden und besitzt den Primärschlüssel {`student_id`}. Die MODULES-Relation, dargestellt in Tabelle 1.2, weist jeder Modul-ID einen Modulnamen zu und verfügt über den Primärschlüssel {`module_id`}. In der EXAMS-Relation, zu sehen in Tablle 1.3, befinden sich Informationen darüber, welche Studentin bzw. welcher Student in welchem Modul in welchem Semester welche Note erhielt. Mit `student_id` aus STUDENTS sowie `module_id` aus MODULES verfügt sie über zwei Fremdschlüsselattribute. Ihr zusammengesetzter Primärschlüssel ist {`student_id`, `module_id`, `semester`}.

1.3. Aufbau der Arbeit

Wir beginnen die Masterarbeit in Kapitel 2 mit den benötigten Grundlagen. Dazu betrachten wir in Abschnitt 2.1 die wichtigsten Konzepte und Definitionen relationaler Datenbanken, eingebettete Abhängigkeiten sowie den *CHASE*-Algorithmus. In Abschnitt 2.2 folgen die Grundideen der Data Provenance sowie in Abschnitt 2.3 fundamentale Ziele und Methoden der Privacy. Anschließend sehen wir uns in Kapitel 3 den aktuellen Stand der Forschung und Technik an. Mit diesem gesammelten Wissen entwickeln wir in Kapitel 4 ein Konzept, um *ProSA* um einen Anonymisierer zu erweitern. Dabei betrachten wir in Abschnitt 4.1 zunächst den genauen Ablauf eines *ProSA*-Durchlaufs, bevor wir in Abschnitt 4.2 mögliche Ansätze für eine Anonymisierung, in Abschnitt 4.3 verschiedene Anonymisierungsmaße und in Abschnitt 4.4 mögliche Anonymisierungsmethoden evaluieren. Am Ende jedes Abschnitts wählen wir die für *ProSA* geeignetsten Optionen. In Abschnitt 4.5 entwickeln wir den Algorithmus für den Anonymisierer, bevor wir diesen in Kapitel 5 praktisch in Form des Tools *ProSAnon* implementieren. Zu guter Letzt formulieren wir in Kapitel 6 ein Fazit dieser Masterarbeit und geben einen Ausblick über Ideen und Herausforderungen für eine Weiterentwicklung der Theorie und Praxis.

	student_id	lastname	firstname	birthday	zipcode	district	course
s_1	10001	Fieber	Fabian	08.03.1998	18059	Südstadt	INF
s_2	10002	Sonnenschein	Sarah	21.10.1993	18059	Südstadt	INF
s_3	10003	Müller	Max	22.10.1994	18057	Hansaviertel	WIN
s_4	10004	Deckert	Luisa	22.10.1994	18057	Hansaviertel	WIN
s_5	10005	Gebauer	Luisa	01.02.2000	18106	Evershagen	WIN
s_6	10006	Zimmermann	Jonas	23.07.1999	18106	Evershagen	ET
s_7	10007	Bach	Franziska	08.03.1998	18147	Gehlsdorf	ET
s_8	10008	Kemper	Moritz	27.11.1991	18055	Stadtmitte	INF
s_9	10009	Wolter	Franziska	22.10.1994	18147	Gehlsdorf	WIN
s_{10}	10010	Jansen	Jana	27.11.1991	18059	Biestow	INF
s_{11}	10011	Wegner	Daniel	19.01.1995	18059	Südstadt	INF
s_{12}	10012	Wegner	Laura	13.12.1992	18059	Südstadt	WIN
s_{13}	10013	Scholz	Erich	03.10.1993	18106	Evershagen	ET
s_{14}	10014	Müller	Mira	05.10.1994	18146	Dierkow	INF
s_{15}	10015	Miller	Mia	12.09.1994	18146	Dierkow	INF
s_{16}	10016	Freiberg	Damian	01.02.2000	18106	Evershagen	INF
s_{17}	10017	Schmidt	Hans	26.11.1991	18119	Warnemünde	INF

Tabelle 1.1.: Beispielrelation STUDENTS

	module_id	name
m_1	1	Vortragsseminar
m_2	2	Logik und Berechenbarkeit
m_3	3	Imperative Programmierung
m_4	4	Rechnernetze und Datensicherheit
m_5	5	Theoretische Informatik
m_6	6	Algorithmen und Datenstrukturen
m_7	7	Digitale Systeme
m_8	8	Datenbanken
m_9	9	Softwaretechnik
m_{10}	10	Mathematik für Informatik
m_{11}	11	Computergrafik

Tabelle 1.2.: Beispielrelation MODULES

	student_id	module_id	semester	grade
e_1	10001	10	WS 20/21	1.3
e_2	10002	10	WS 20/21	2.0
e_3	10003	10	WS 20/21	2.7
e_4	10004	10	WS 20/21	1.3
e_5	10005	10	WS 20/21	1.7
e_6	10008	10	WS 20/21	4.0
e_7	10009	10	WS 20/21	3.3
e_8	10010	10	WS 20/21	5.0
e_9	10011	10	WS 20/21	5.0
e_{10}	10012	10	WS 20/21	3.0
e_{11}	10014	10	WS 20/21	1.3
e_{12}	10015	10	WS 20/21	2.3
e_{13}	10016	10	WS 20/21	1.0
e_{14}	10017	10	WS 20/21	3.3
e_{15}	10002	8	SS 21	1.3
e_{16}	10003	8	SS 21	1.3
e_{17}	10005	6	SS 21	1.0
e_{18}	10007	8	SS 21	3.0
e_{19}	10008	7	SS 21	3.7
e_{20}	10011	11	SS 21	1.7
e_{21}	10014	8	SS 21	1.0
e_{22}	10016	3	SS 21	1.0
e_{23}	10002	5	SS 21	3.3
e_{24}	10002	2	SS 21	2.0
e_{25}	10002	4	SS 21	1.0
e_{26}	10003	7	SS 21	1.7
e_{27}	10005	9	SS 21	2.0
e_{28}	10005	1	SS 21	3.0
e_{29}	10003	8	WS 21/22	5.0

Tabelle 1.3.: Beispielrelation EXAMS

2. Grundlagen

Um ein Verständnis für das Thema dieser Arbeit zu erhalten, sehen wir uns in diesem Kapitel zunächst einige Grundlagen an, die für diese Masterarbeit relevant sind. Wir beginnen dabei zunächst in Abschnitt 2.1 mit den wichtigsten Konzepten der relationalen Datenbanktheorie und sehen uns eingebettete Abhängigkeiten, insbesondere die sogenannten egds und (s-t) tgds, und den *CHASE*-Algorithmus an, bevor wir in Abschnitt 2.2 mit der Data Provenance fortfahren. Zu guter Letzt beschäftigen wir uns dann in Abschnitt 2.3 mit den Grundlagen der Privacy und dabei insbesondere einigen wichtigen Anonymisierungstechniken und -maßen.

2.1. Relationale Datenbanken

Das relationale Datenbankmodell ist heute eines der verbreitetsten Modelle der Datenbanktheorie und wird unter anderem von den heute etablierten Datenbankmanagementsystemen (DBMS) wie *PostgreSQL*, *IBM DB2*, *MySQL* bzw. den Fork[1] *MariaDB* sowie *Microsoft SQL Server* verwendet. Die fundamentale Philosophie dahinter ist das Darstellen von Datensätzen als Tabellen, welche über Verbundattribute miteinander verknüpft werden können und das Stellen von Anfragen an die gesamte Datenbank ermöglichen. Eine **Datenbank** besteht dabei aus **Relationen**, welche wiederum aus **Attributen** bestehen, denen ein gewisser Wertebereich (auch **Domäne** genannt) zugeordnet ist. Das Buch *„Datenbanken: Konzepte und Sprachen“* von *Andreas Heuer*, *Kai-Uwe Sattler* und *Gunter Saake* bietet für die interessierte Leserin bzw. den interessierten Leser einen detaillierten Einstieg in das Thema [HSS18]. Nachfolgend schauen wir uns die wichtigsten Definitionen an, die aus dem Buch übernommen wurden.

Definition 2.1 (Attribute und Wertebereiche, nach [HSS18])**.** Sei $\mathcal{U}$ eine nicht-leere, endliche Menge, das **Universum** der Attribute. Ein Element $A \in \mathcal{U}$ heißt **Attribut**. Sei $\mathcal{D} = \{D_1, \ldots, D_m\}$ eine Menge endlicher, nicht-leerer Mengen mit $m \in \mathbb{N}$. Jedes D_i wird **Wertebereich** oder **Domäne** genannt. Es existiert eine total definierte Funktion $\text{dom} : \mathcal{U} \mapsto \mathcal{D}$. $\text{dom}(A)$ heißt dann Wertebereich bzw. Domäne von A. Ein $w \in \text{dom}(A)$ wird **Attributwert** für A genannt. □

Definition 2.2 (Relationen und Relationenschemata, nach [HSS18])**.** Eine **Relation** r über $R = \{A_1, \ldots, A_n\}$ (kurz: $r(R)$) mit $n \in \mathbb{N}$ ist eine endliche Menge von Abbildungen

$$t : R \mapsto \bigcup_{i=1}^{m} D_i,$$

die **Tupel** genannt werden, wobei $t(A) \in \text{dom}(A)$ gilt. Eine Menge $R \subseteq \mathcal{U}$ heißt **Relationenschema**. □

Definition 2.3 (Datenbankschema und Datenbank, nach [HSS18])**.** Eine Menge von Relationenschemata $S := \{R_1, \ldots, R_p\}$ mit $p \in \mathbb{N}$ heißt **Datenbankschema**. Ein Datenbankwert (kurz: **Datenbank**)

[1] eine Open-Source-Abspaltung, welche weitestgehend zu *MySQL* kompatibel ist

über einem Datenbankschema S ist eine Menge von Relationen $d := \{r_1, \ldots, r_p\}$, wobei $r_i(R_i)$ für alle $i \in \{1, \ldots, p\}$ gilt. Eine Datenbank d über S wird mit $d(S)$ bezeichnet, eine Relation $r \in d$ heißt **Basisrelation**. □

Ein fundamentales Prinzip relationaler Datenbanken ist die Existenz von *Schlüsseln*. Jedes Tupel einer Relation muss über eine gewisse Attributmenge eindeutig identifizierbar sein. Diese Menge kann einelementig sein, beispielsweise in Form eines fortlaufenden ID-Attributs, sie kann aber auch aus mehreren Attributen bestehen, die erst in Kombination jedes Tupel eindeutig bestimmen. Eine solche Attributmenge wird als *identifizierende Attributmenge* bezeichnet.

Definition 2.4 (Identifizierende Attributmenge, nach [HSS18])**.** Eine **identifizierende Attributmenge** für ein Relationenschema R ist eine Menge $K := \{B_1, \ldots, B_k\} \subseteq R$, so dass für jede Relation $r(R)$ gilt: $\forall t_1, t_2 \in r : t_1 \neq t_2 \Rightarrow \exists B \in K : t_1(B) \neq t_2(B)$. Ein **Schlüssel** ist eine bezüglich $\subseteq$ minimale identifizierende Attributmenge. Die einzelnen Attribute eines Schlüssels werden **Primattribute** genannt. □

Die Existenz einer solchen identifizierenden Attributmenge je Tupel ist eine *Integritätsbedingung* bzw. eine sogenannte *eingebettete Abhängigkeit*. Daneben kann es diverse weitere Abhängigkeiten geben, welche die Integrität einer Datenbank sichern. Diese sind vielfältig einsetzbar und können beispielsweise auch genutzt werden, um Schema-Evolutionen darzustellen, unvollständige Datensätze zu reparieren (*„Data Cleaning“*) oder Datenbankmigrationen zu ermöglichen (*„Data Exchange“*). Einen vollständigen Überblick über eingebettete Abhängigkeiten bieten die Autoren *Sergio Greco* und *Cristian Molinaro* sowie die Autorin *Francesca Spezzano* in ihrem Buch „Incomplete Data and Data Dependencies in Relational Databases“, aus dem auch nachfolgende Definitionen übernommen wurden [GMS12].

Definition 2.5 (Eingebettete Abhängigkeit, nach [GMS12])**.** Eine **eingebettete Abhängigkeit** ist ein prädikatenlogischer Ausdruck erster Stufe mit der Form $\forall x, y : \varphi(x, y) \rightarrow \exists z : \psi(x, z)$, wobei x, y und z Tupel aus Variablen und $\varphi(x, y)$ sowie $\psi(x, z)$ Konjunktionen über atomare Formeln sind. $\varphi(x, y)$ wird hierbei *Rumpf* und $\psi(x, z)$ *Kopf* der Abhängigkeit genannt. □

Spezielle Varianten der eingebetteten Abhängigkeiten sind die sogenannten **equality-generating dependencies (egds)** sowie die **tuple-generating dependencies (tgds)**. Erstere können zum Beispiel für Data Cleaning eingesetzt werden, während letztere vor allem bei Anfrageoptimierungen und Data Exchange nützlich sind. Die sogenannten *source-to-target tgds* sind wiederum eine spezielle Art von tgds, bei der Tupel von einem Quell- in ein Zielschema überführt werden sollen. Sie können jedoch auch genutzt werden, um eine Teilmenge von SQL-Anfragen darzustellen, da jede SQL-Anfrage für sich im weitesten Sinne eine solche Überführung durchführt. In den Abschnitten ... werden wir sehen, dass *ProSA* vor allem Gebrauch von s-t tgds macht.

Definition 2.6 (egd, nach [GMS12])**.** Eine **equality-generating dependency** (dt. gleichheitserzeugende Abhängigkeit) ist eine eingebettete Abhängigkeit der Form $\forall x : F_1(x) \rightarrow x_1 = x_2$. Sie besitzt keine existenzquantifizierten Variablen und ihr Kopf besteht nur aus einem einzigen Gleichheitsatom. □

Betrachten wir an dieser Stelle ein Beispiel. Gegeben sei eine Relation $R(a, b, c, d)$. Die egd

$$\forall a, b, c, d_1, d_2 : R(a, b, c, d_1) \wedge R(a, b, c, d_2) \rightarrow d_1 = d_2$$

sagt aus, dass für je zwei Tupel in R mit übereinstimmenden Attributwerten für a, b und c auch beide Attributwerte für d identisch sein müssen. (Anmerkung: aus Gründen der Übersichtlichkeit wird der

Allquantor in der Praxis und auch im weiteren Verlauf dieser Arbeit vernachlässigt. Der Vollständigkeit halber wurde er hier aber einmal angegeben.)

Definition 2.7 ((s-t) tgd, nach [GMS12]). Eine **tuple-generating dependency** (dt. tupelerzeugende Abhängigkeit) ist eine eingebettete Abhängigkeit der Form $\forall x : F_1(x) \rightarrow \exists y : F_2(x, y)$. $F_1(x)$ ist dabei eine Konjunktion von atomaren Formeln mit Variablen aus x, während $F_2(x, y)$ eine Konjunktion von atomaren Formeln mit Variablen aus x und y ist. Sie besitzt keine Gleichheitsatome. Eine **source-to-target tgd** ist ein Spezialfall der tgd, bei dem F_1 eine Konjunktion über ein Quellschema S (source) und F_2 eine Konjunktion über ein Zielschema T (target) ist. □

Auch hier wollen wir wieder ein Beispiel betrachten. Wie wir bereits wissen, lassen sich viele SQL-Anfragen auch als s-t tgds darstellen. Wir wollen deshalb nachfolgende Anfrage an unsere Beispieldatenbank als s-t tgd ausdrücken.

```
SELECT firstname, lastname
FROM Students
WHERE district = 'Hansaviertel';
```

Anfrage 2.1: Wie heißen alle Studierenden, die im Stadtteil „Hansaviertel“ wohnen?

Die STUDENTS-Relation hat den vereinfachten Aufbau $S(id, la, fi, bd, zi, di, co)$, wobei S für STUDENTS, id für `student_id`, la für `lastname`, fi für `firstname`, bd für `birthday`, zi für `zipcode`, di für `district` und co für `course` stehen. Wenn wir diese Schreibweise nun als Schablone für alle enthaltenen Tupel betrachten, so müssen wir für die Selektion `district = 'Hansaviertel'` die Variable di durch den konkreten Wert „Hansaviertel“ ersetzen. Um die Projektion auf `firstname` und `lastname` durchzuführen, geben wir diese beiden Attribute in einem neuen, selbst definierten Ergebnisprädikat R an, welches das „Zielschema“ der Anfrage darstellt. Wir erhalten somit eine s-t tgd

$$S(id, la, fi, bd, zi, \text{„Hansaviertel“}, co) \rightarrow R(fi, la),$$

welche unsere SQL-Anfrage 2.1 repräsentiert. Eine ausführlichere Beschreibung von egds, (s-t) tgds sowie der Überführung von SQL-Anfragen bietet die Seminararbeit *„Der CHASE-Algorithmus: Ein Überblick“* [Sch21].

CHASE-Algorithmus Der *CHASE-Algorithmus* ist ein Algorithmus, der in der Datenbanktheorie für eine Reihe von Anwendungsfällen verwendet werden kann, darunter Data Cleaning, Data Exchange, Anfrageoptimierung und Datenintegration. Verallgemeinert beschrieben benötigt er ein *CHASE*-Objekt ◯ sowie einen *CHASE*-Parameter $\star$ als Eingabe und arbeitet den Parameter dann in das Objekt ein, sodass nach dem Anwenden des *CHASE* ein neues Objekt ⊛ entsteht. Am allgemeinsten lässt sich der *CHASE* deshalb mit $\text{CHASE}_\star(◯) = ⊛$ zusammenfassen. Damit dient er in der Datenbanktheorie als Universalwerkzeug. In der Regel handelt es sich bei dem Objekt ◯ um eine Datenbankinstanz I oder eine Anfrage Q und bei dem Parameter $\star$ um eine Menge von Integritätsbedingungen, genauer gesagt um egds und (s-t) tgds. Der CHASE setzt dann so lange Tupel gleich beziehungsweise generiert neue Tupel, bis alle Integritätsbedingungen in der Zielinstanz erfüllt sind. Auch hier betrachten wir ein Beispiel und nutzen dafür die s-t tgd, die wir zuvor anhand der SQL-Anfrage 2.1 aufstellten. Der *CHASE*-Algorithmus prüft schrittweise jedes Tupel der Relation STUDENTS auf einen Homomorphismus. Vereinfacht ausgedrückt existiert ein **Homomorphismus**, wenn sich der linke Teil – also der Rumpf – der s-t tgd auf ein Tupel der Relation abbilden lässt und umgekehrt. Diese Existenz wird dann als **Trigger** für den *CHASE* bezeichnet. Für die Tupel s_1 = (10001, Fieber, Fabian, 08.03.1998, 18059, Südstadt,

Algorithmus 1: Der Standard-CHASE-Algorithmus nach [AH19]

Data: Menge von Integritätsbedingungen Σ, Datenbankinstanz I
Result: modifizierte Datenbankinstanz I'

```
forall Trigger h für eine Integritätsbedingung σ ∈ Σ do
    if h ist ein aktiver Trigger then
        if σ ist eine tgd then
            Füge neue Tupel zu I hinzu
        else if σ ist eine egd then
            if verglichene Werte sind verschiedene Konstanten then
                CHASE schlägt fehl
            else
                Ersetze Nullwerte durch andere Nullwerte oder Konstanten
```

INF) und s_2 = (10002, Sonnenschein, Sarah, 21.10.1993, 18059, Südstadt, INF) existieren keine Trigger, da sich die Werte „Südstadt“ und „Hansaviertel“ nicht aufeinander abbilden lassen. Anders sieht das bei den Tupeln s_3 = (10003, Müller, Max, 22.10.1994, 18057, Hansaviertel, WIN) und s_4 = (10004, Deckert, Luisa, 22.10.1994, 18057, Hansaviertel, WIN) aus. Diese lassen sich aufeinander abbilden und „passen in die Schablone“, weshalb hier jeweils ein Trigger existiert. Da wir diese Tupel noch nicht in die Zielinstanz (das Anfrageergebnis) überführt haben, das Anwenden der s-t tgd also die Zielinstanz modifiziert, ist dieser Trigger **aktiv**. Wenden wir die s-t tgd an, so erhalten wir gemäß ihres Kopfes die neuen Tupel R(Max, Müller) sowie R(Luisa, Deckert).

Für die restlichen Tupel der Relation existieren keine Trigger und der *CHASE* terminiert am Ende der Relation. Eine Beschreibung des *Standard CHASE*[2] ist in Algorithmus 1 zu sehen. Für eine ausführliche Erklärung des Algorithmus' inklusive Beispiele sei ebenfalls auf [Sch21] verwiesen.

2.2. Data Provenance

Provenance (dt. „Provenienz“, allgemeiner: „Herkunft“), bezeichnet die Nachverfolgbarkeit eines Objektes bis hin zu seinen Ursprüngen. Im allgemeinen Kontext kann sich der Begriff auf den Ursprung von beispielsweise Kunstwerken, Büchern oder auch Personen beziehen [ASH21b]; im wissenschaftlichen Kontext ist damit das Rückverfolgen von Ergebnissen bis zu den Ursprüngen (Data Provenance) oder das Rückverfolgen von Arbeitsabläufen (Workflow Provenance) gemeint. Das Ziel der **Data Provenance** (dt. „Datenherkunft“) ist es entsprechend, ausgehend von einem Ergebnisdatensatz all jene Ursprungsdaten zu ermitteln, die zu diesem Ergebnis beigetragen haben. Im Kontext relationaler Datenbanken bedeutet dies, dass genau die Tupel der Quellrelationen ermittelt werden sollen, die für ein bestimmtes Anfrageergebnis (Zielrelation) verantwortlich sind.

Die Forschung an Data Provenance beläuft sich im Wesentlichen auf die letzten zwei Jahrzehnte und gewann währenddessen stetig an Bedeutung. In der Praxis ist es häufig von Interesse und im Sinne guter wissenschaftlicher Arbeit oft auch eine Voraussetzung, Forschungsdaten für einen längeren Zeitraum aufzubewahren. Jedoch fallen dabei zunehmend mehr Daten an, was die Komplexität von archivierten Datensätzen und damit den Speicherplatzbedarf zunehmend erhöht. Außerdem kann gerade bei wertvollen, beispielsweise aufwändig erhobenen, Datensätzen das Interesse bestehen, nur so viele Daten wie absolut nötig herauszugeben. In Kapitel 3 werden wir uns damit noch genauer beschäftigen. Auch die Industrie steht durch **Big Data** vor der Herausforderung, einerseits die Nachvollziehbarkeit,

[2] Neben dem *Standard CHASE* existieren mit dem *Oblivious CHASE* sowie dem *Core CHASE* noch weitere *CHASE*-Varianten, die an dieser Stelle jedoch nicht näher betrachtet werden. Es sei auf [GMS12] als tiefergehende Literatur verwiesen.

Antworttyp	Ergebnis	Antwort auf
extensional	Beteiligte Tupel der Originaldaten	***where***, ***why***
intensional	Beschreibung der Daten	***where***, ***why***, ***how***
anfragebasiert	Selektionsprädikate	***why***, ***how***
modifikationsbasiert	Minimaler Änderungsvorschlag der Auswertung	***why not***

Tabelle 2.1.: Antworttypen auf Provenance-Anfragen und deren Anwendungsbereich (vgl. [HDB17])

Reproduzierbarkeit oder gar Rekonstruierbarkeit von Daten und Ergebnissen zu gewährleisten und andererseits die Speicherplatzauslastung zu minimieren. Data Provenance kann dabei helfen, ebendieses Ziel zu erreichen, indem nur absolut notwendige Rohdaten aufbewahrt und die restlichen verworfen werden. Allgemein stellen sich vier Fragen, die es mithilfe von Data Provenance zu beantworten gilt (vgl. [AH19]):

1. Woher stammen die Daten? (***where***-Provenance)
2. Warum kam dieses Ergebnis zustande? (***why***-Provenance)
3. Wie genau wurde das Ergebnis berechnet? (***how***-Provenance)
4. Warum sind gewisse Daten nicht im Ergebnis enthalten? (***why not***-Provenance)

Um diese vier Fragen zu beantworten, gibt es verschiedene Möglichkeiten (vgl. [HDB17]). Tabelle 2.1 liefert einen kompakten Überblick darüber. Extensionale Antworten liefern die beteiligten Tupel der Originaldaten. Intensionale Antworten hingegen liefern nur eine abstrahierte Beschreibung der Originaldaten, aber keine Rohdaten selbst. Anfragebasierte Antworten liefern Informationen über Selektionsprädikate, die zu einem Ergebnis geführt haben. Modifikationsbasierte Antworten zu guter Letzt empfehlen minimale Anpassungen von Anfragen, um zu begründen, wieso bestimmte Tupel *kein* Teil des Ergebnisses sind. Für diese Arbeit relevant sind vor allem extensionale und untergeordnet intensionale Antworten sowie die ***where***-, ***why***- sowie ***how***-Provenance. Das Ziel der ***why not***-Provenance, die Begründung, wieso bestimmte Daten kein Teil eines Ergebnisses sind, ist für diese Arbeit weniger relevant.

Weitere Provenance-Arten Neben der Data Provenance gibt es noch weitere Formen der Provenance, die jedoch für diese Masterarbeit nicht relevant sind und deshalb nur der Vollständigkeit halber Beachtung finden. Hierzu gehören beispielsweise die Workflow Provenance und die Metadata Provenance.

Die **Workflow Provenance** beschäftigt sich mit der Rückverfolgbarkeit von Arbeitsabläufen, um ein (wissenschaftliches) Ergebnis zu begründen. Dabei werden beispielsweise eingesetzte Anwendungen, deren Ein- und Ausgaben sowie Abläufe zwischen verschiedenen Anwendungen nachverfolgt. Die Frage der Workflow Provenance ist also, welche Arbeitsschritte, Transformationen, Ein- und Ausgaben und Anwendungen zu einem bestimmten Ergebnis geführt haben (siehe hierzu [DF08]).

Das Ziel der **Metadata Provenance** ist es, Provenance-Informationen über ein Objekt in Form von Metadaten zu vermitteln. Dabei kann es sich beispielsweise um Fotografien, Bücher, aber auch um nicht-digitale Objekte wie Pakete oder gar Lebensmittel handeln [HDB17]. Die Metadata Provenance ist die Provenance-Art, die am wenigsten automatisiert werden kann und somit den meisten manuellen Aufwand benötigt.

Je nach Literatur existiert mit der **Information System Provenance** noch eine weitere Variante, welche Datenflüsse in Informationssystemen nachverfolgen soll. Dabei kann es sich beispielsweise um das Empfangen, Senden und Verteilen von Informationen oder um Kommunikationsprozesse handeln [HDB17]. Die

Information System Provenance lässt sich einfacher automatisieren als die Metadata Provenance, aber schwieriger als die Workflow Provenance und der Data Provenance.

Im folgenden Abschnitt sehen wir uns die ***where***-, ***why***- sowie ***how***-Provenance im Detail an. Alle drei Provenance-Arten gehören zum Bereich der Data Provenance und können auf relationale Datenbanken angewandt werden.

2.2.1. Where-Provenance

Wir beginnen mit der sogenannten ***where***-Provenance. Wie der Name schon verrät, ist es ihre Aufgabe, die Frage „Woher kommen die Daten?“ (engl. *Where* does the data come from?) zu beantworten. Dies kann in der Regel auf zwei Arten erfolgen: tupelorientiert oder relationenorientiert [AH19]. Bei der tupelorientierten ***where***-Provenance ist die Antwort der Frage eine Liste von Tupel-Identifikatoren all jener Tupel, deren Daten ein Ergebnis repräsentieren. Bei der relationenorientierten ***where***-Provenance hingegen ist die Antwort eine Liste von Relationen und somit eine abstraktere Beantwortung der Frage. Sie beschreibt also den Zusammenhang zwischen dem Ursprungs- und Zielort eines Datums. Wir betrachten dazu exemplarisch die SQL-Anfrage 2.2, welche die Vornamen aller Studierenden zurückgeben soll, die im Stadtteil Evershagen leben. Das Ergebnis der Anfrage inklusive Provenance-Annotationen ist in Tabelle 2.2 zu sehen.

```
SELECT DISTINCT zipcode, district
FROM Students
WHERE firstname = 'Franziska';
```

Anfrage 2.2: Wie lauten PLZ und Stadtteil aller Studierenden, die Franziska heißen?

zipcode	district	
18147	Gehlsdorf	$\{s_7, s_9\}$

Tabelle 2.2.: Ergebnis der Anfrage 2.2 inklusive ***where***-Provenance

Für das einzige Tupel `(18147, Gehlsdorf)` des Ergebnisses lautet die tupelorientierte ***where***-Provenance $\{s_7, s_9\}$, da die Werte „`18147`“ und „`Gehlsdorf`“ aus ebendiesem *Tupel* übernommen wurde; die relationenorientierte ***where***-Provenance lautet {STUDENTS}, da die Werte aus selbiger *Relation* stammen. $\{s_7, s_9\}$ und {STUDENTS} sind im Kontext des Tupels sogenannte **Zeugenlisten**, während s_7, s_9 sowie STUDENTS für sich stehend sogenannte **Zeugen** sind. Sie bezeugen die Daten, die in dem Tupel auftauchen. Umgekehrt würde eine Elimination aller Zeugen – also der Tupel s_7 und s_9 bzw. der gesamten Relation STUDENTS dazu führen, dass das Tupel aus dem Ergebnis verschwindet und dieses somit leer wäre.

Es sei angemerkt, dass es neben der tupel- sowie relationenorientierten ***where***-Provenance noch weitere Arten gibt, diese zu beschreiben. *Bunemal et al.* definieren die ***where***-Provenance beispielsweise zellen- bzw. attributorientiert [BKT01]. Dabei werden nicht nur die Ursprungsrelation und das Ursprungstupel, sondern auch die Ursprungsspalte in Betracht gezogen. Wir werden uns in dieser Arbeit jedoch auf die tupel- sowie relationenorientierte ***where***-Provenance beschränken.

2.2.2. Why-Provenance

Nach der ***where***-Provenance schauen wir uns nun die nächste Art der Data Provenance, die ***why***-Provenance, an. Diese hat das Ziel, die Frage „Wie kam das Ergebnis zustande?“ (engl. *Why* was the result achieved?) zu beantworten, indem alle Quelltupel angegeben werden, die direkt oder indirekt zu einem Zieltupel beigetragen haben. Anders als die ***where***-Provenance beschreibt sie somit Beziehungen zwischen Tupeln statt Beziehungen zwischen Orten von Daten (vgl. [CCT09]). Auch hier betrachten wir wieder die Beispielanfrage (Anfrage 2.2) und ihr Ergebnis, diesmal allerdings mit der dazugehörigen ***why***-Provenance (Tabelle 2.3).

zipcode	district	
18147	Gehlsdorf	$\{\{s_7\},\{s_9\}\}$

Tabelle 2.3.: Ergebnis der Anfrage 2.2 inklusive ***why***-Provenance

Betrachtet man die Beispielrelation STUDENTS (Tabelle 1.1), so stellt man fest, dass es mit s_7 und s_9 zwei Tupel gibt, die die Anfrage erfüllen. Das Ergebnistupel `(18147, Gehlsdorf)` kommt also aufgrund des Tupels s_7 *oder* des Tupels s_9 zustande bzw. wird von ihnen bezeugt. s_7 und s_9 sind dementsprechend Zeugen; $\{s_7\}$ und $\{s_9\}$ sind die jeweiligen **Zeugenmengen**. Die Menge aller Zeugenmengen, in diesem Beispiel $\{\{s_7\},\{s_9\}\}$, wird **Zeugenbasis** genannt. Auf den ersten Blick scheint es noch keinen direkten Unterschied zur ***where***-Provenance zu geben, denn diese würde ebenfalls $\{s_7, s_9\}$ bzw. {STUDENTS} lauten, da die Daten „`18147`“ sowie „`Gehlsdorf`“ aus ebendiesen Tupeln bzw. der Relation STUDENTS übernommen wurden. Allerdings umfasst die ***why***-Provenance auch Quelltupel, die indirekt an einem Zieltupel beteiligt waren, das heißt keine direkt sichtbaren Informationen liefern. Dazu schauen wir uns eine weitere Anfrage 2.3 an. Das Ergebnis ist in Tabelle 2.4 zu sehen.

```
SELECT Students.firstname, Students.lastname, Modules.name AS module
FROM Students NATURAL JOIN Exams NATURAL JOIN Modules
WHERE grade = 1.7;
```

Anfrage 2.3: Welche Studierenden haben in welchem Modul eine 1,7 geschrieben?

firstname	lastname	module	
Luisa	Gebauer	Mathematik für Informatik	$\{\{s_5, e_5, m_{10}\}\}$
Daniel	Wegner	Computergrafik	$\{\{s_{11}, e_{20}, m_{11}\}\}$
Max	Müller	Digitale Systeme	$\{\{s_3, e_{26}, m_7\}\}$

Tabelle 2.4.: Ergebnis der Anfrage 2.3 inklusive ***why***-Provenance

Zur Veranschaulichung betrachten wir auch hier nur das erste Tupel `(Luisa, Gebauer, Mathematik für Informatik)`. Die Zeugenbasis für dieses Tupel lautet $\{\{s_5, m_{10}, e_5\}\}$, besteht also allein aus der Zeugenmenge $\{s_5, m_{10}, e_5\}$. Das bedeutet, dass die Quelltupel s_5, m_{10} und e_5 *gemeinsam* zu dem Zieltupel beitragen. Keines der drei Tupel kann dieses Ergebnis allein produzieren und auch die neun Zweierkombinationen der drei Tupel können dieses Tupel nicht bezeugen. Die ***where***-Provenance lautet jedoch $\{s_5, m_{10}\}$ bzw. {STUDENTS, MODULES}, da die Werte „`Luisa`“ sowie „`Gebauer`“ aus dem Quelltupel s_5 und der Wert „`Mathematik für Informatik`“ aus dem Quelltupel m_{10} übernommen wurden. Dass dieses Zieltupel jedoch ohne den Verbund über die Relation EXAMS bzw. das Quelltupel e_5 gar nicht zustande kommen kann, wird nicht erfasst. Die ***why***-Provenance ist somit aussagekräftiger als die ***where***-Provenance.

2.2.3. How-Provenance

In den vorherigen Unterabschnitten betrachteten wir bereits die ***where***- sowie die ***why***-Provenance. Erstere beschreibt den Zusammenhang zwischen Orten von Daten, letztere beschreibt Beziehungen zwischen Tupeln. Beide sind jedoch nicht in der Lage, genaue Berechnungsabläufe von Ergebnissen zu beschreiben. Deshalb gibt es mit der ***how***-Provenance eine weitere Variante der Data Provenance, welche genau dies als Ziel hat. Die ***how***-Provenance beantwortet die Frage „Wie wurde das Ergebnis berechnet?“ (engl. *How was the result calculated?*). Dazu werden sogenannte **Provenance-Polynome** genutzt, die auf kommutativen Halbringen basieren. Diese werden an dieser Stelle als bekannt vorausgesetzt. *Green*, *Karvounarakis* und *Val Tannen* stellten erstmals 2007 das **Semiring-Framework** für ***how***-Provenance vor [GKT07]. Der Artikel wurde zehn Jahre später mit dem *Test of Time Award* der *SIGMOD* ausgezeichnet und erneut veröffentlicht [GT17]. Er bietet eine Einführung in sowie eine Übersicht über verschiedene Halbringe, die für die ***how***-Provenance verwendet werden können. Für die ***how***-Provenance von besonderer Bedeutung ist der $\mathbb{N}[X]$-Halbring (Definition 2.8).

Definition 2.8 (Provenance-Halbring, nach [GT17])**.** Der Provenance-Halbring $\mathbb{N}[X] = (\mathbb{N}[X], +, \cdot, 0, 1)$ ist ein Halbring mit Polynomen über X und Koeffizienten aus $\mathbb{N}$. Er wird für Provenance-Polynome verwendet. □

Duplikate – bei Projektionen und Vereinigungen – werden hierbei durch die einfache Addition „+“, Verbunde, Selektionen und Schnittmengen durch die einfache Multiplikation „·“ dargestellt [GT17]. Wir betrachten erneut die Anfrage 2.3 sowie das Ergebnis, diesmal mit der ***how***-Provenance (Tabelle 2.5).

firstname	lastname	module	
Luisa	Gebauer	Mathematik für Informatik	$s_5 \cdot e_5 \cdot m_{10}$
Daniel	Wegner	Computergrafik	$s_{11} \cdot e_{20} \cdot m_{11}$
Max	Müller	Digitale Systeme	$s_3 \cdot e_{26} \cdot m_7$

Tabelle 2.5.: Ergebnis der Anfrage 2.3 inklusive ***how***-Provenance

Die ***how***-Provenance des ersten Zieltupels lautet $s_5 \cdot e_5 \cdot m_{10}$, da die drei Quelltupel s_5, e_5 und m_{10} in Form eines Verbunds zum Zieltupel beitrugen. Wie bereits in Unterabschnitt 2.2.2 dargelegt, sind alle drei Tupel für das Zustandekommen des Ergebnisses obligatorisch und können deshalb nicht von anderen Quelltupeln subsumiert werden. Anders sieht das beim Ergebnis der Anfrage 2.2 aus, welches inklusive ***how***-Provenance in Tabelle 2.6 zu sehen ist.

zipcode	district	
18147	Gehlsdorf	$s_7 + s_9$

Tabelle 2.6.: Ergebnis der Anfrage 2.2 inklusive ***how***-Provenance

Die ***how***-Provenance des Zieltupels lautet $s_7 + s_9$, da *entweder* das Tupel s_7 *oder* das Tupel s_9 benötigt wird, um das Ergebnis zu produzieren. Es ist jedoch theoretisch nicht notwendig, *beide* Tupel zu erhalten, was im Polynom durch die Addition beider Tupel ausgedrückt wird. Außerdem ist die ***how***-Provenance in der Lage, konkrete Werte sowie die genauen Schritte der Berechnung darzustellen. Dazu gibt es bei aggregierten Werten mit dem Tensorprodukt $\odot$ und der direkten Summe $\oplus$ zwei weitere Operatoren, um Provenance-Polynome um konkrete Werte zu erweitern ($\odot$) und miteinander zu verknüpfen ($\oplus$). Auch hier betrachten wir mit Anfrage 2.4 ein Beispiel; das Ergebnis ist inklusive Hilfstabelle in Tabelle 2.7 zu sehen.

	student_id	grade
$s_2 \cdot e_{15}$	10002	1.3
$s_3 \cdot e_{16}$	10003	1.3
$s_7 \cdot e_{18}$	10007	3.0
$s_{14} \cdot e_{21}$	10014	1.0
$s_3 \cdot e_{29}$	10003	5.0

(a) Hilfstabelle

average
2.32

(b) Ergebnis

Tabelle 2.7.: Ergebnis der Anfrage 2.4 inklusive Zwischenschritt

```
SELECT AVG(grade) AS average
FROM Modules NATURAL JOIN Exams
WHERE name = 'Datenbanken';
```

Anfrage 2.4: Was ist die durchschnittliche Note der Prüfung in „Datenbanken“?

Die ***how***-Provenance für das Ergebnis lautet

$$p = \frac{(s_2 \cdot e_{15} \odot 1.3) \oplus (s_3 \cdot e_{16} \odot 1.3) \oplus (s_7 \cdot e_{18} \odot 3.0) \oplus (s_{14} \cdot e_{21} \odot 1.0) \oplus (s_3 \cdot e_{29} \odot 5.0)}{s_2 \cdot e_{15} \oplus s_3 \cdot e_{16} \oplus s_7 \cdot e_{18} \oplus s_{14} \cdot e_{21} \oplus s_3 \cdot e_{29}}$$

und kann in zwei Teile zerlegt werden. Der Zähler stellt die Summe aller fünf Werte dar und kann alleinstehend als `SUM`-Anfrage aufgefasst werden. Der Nenner stellt die Anzahl aller Werte dar und kann auch als `COUNT`-Anfrage interpretiert werden, da der Durchschnitt (genauer: arithmetisches Mittel) den Quotienten aus Summe und Anzahl darstellt (`AVG(grade)`$=\frac{\texttt{SUM(grade)}}{\texttt{COUNT(grade)}}$). Aus dem Polynom p geht nun hervor, dass die Tupel s_2 und e_{15} verbunden wurden und den Wert 1.3 beitrugen, s_3 und e_{16} verbunden wurden und den Wert 1.3 beitrugen und so weiter. Aus den einzelnen Teilpolynomen lassen sich nun zudem die Zeugenmengen $\{s_2, e_{15}\}, \{s_3, e_{16}\}, \ldots, \{s_3, e_{29}\}$ ablesen, sodass die ***why***-Provenance direkt aus der ***how***-Provenance abgeleitet werden kann. Für die drei erläuterten Arten der Data Provenance gilt also ***where*** $\preceq$ ***why*** $\preceq$ ***how***, wobei die Ableitung von ***where*** aus ***why*** wie in Unterabschnitt 2.2.2 gezeigt unter Umständen mehr Zeugen als nötig liefern kann (vgl. [AH19]).

2.3. Privacy und Anonymisierung

Nachdem wir uns im vorherigen Abschnitt ausführlich der Data Provenance gewidmet haben, schauen wir uns nun die Grundlagen der Privacy an. Dabei starten wir mit der Frage, was „Privacy“ überhaupt definiert, welche potenziellen Bedrohungen es gibt und letztlich wie man diesen entgegenwirken kann.

Der Privacy-Begriff Zunächst müssen wir klären, was genau überhaupt unter „Privacy“ verstanden wird. Im klassischen Sinne wird der Begriff aus dem Englischen mit „Datenschutz“ oder „Privatheit“ übersetzt und meint den Schutz meist personenbezogener Daten vor Dritten. Re-Identifizierungen einzelner Personen aus veröffentlichten Datensätzen sollen dabei unbedingt vermieden werden. Dass dies ein Problem ist, zeigte *Latanya Sweeney* bereits 2002 anhand eines Beispiels im US-Bundesstaat Massachusetts [Swe02a]. Dort ist es möglich, vor Wahlen eine Kopie des Wählerverzeichnisses zu erwerben. Zeitgleich bot eine örtliche Krankenversicherung Forscherinnen und Forschern einen aus ihrer Sicht anonymisierten Datensatz an, der Informationen über 135 000 Patientinnen und Patienten erhielt. Zwar wurden eindeutig identifizierende Merkmale wie Namen und Sozialversicherungsnummern entfernt, u.a. die Postleitzahl, das Geburtsdatum sowie das Geschlecht waren jedoch weiterhin enthalten, genau wie

– ein hochsensibles Attribut – die diagnostizierte Krankheit. Auch im Wählerverzeichnis fanden sich die Postleitzahl, das Geschlecht und das Geburtsdatum aller wahlberechtigten Personen. Mithilfe beider Datensätze konnte letztendlich dem Gouverneur eindeutig seine Diagnose zugeordnet werden, da es im gesamten Datensatz der Versicherung nur drei Männer mit seinem Geburtstag und davon nur einen mit seiner Postleitzahl gab. Hier konnte also mithilfe externen Wissens eine Re-Identifikation durchgeführt werden. Bereits zwei Jahre zuvor zeigte *Sweeney*, dass die Kombination aus PLZ, Geburtsdatum und Geschlecht als sogenannter Quasi-Identifikator dient (siehe Definition 2.9); 1990 waren 87% aller US-Amerikanerinnen und -Amerikaner allein durch diese drei Attribute eindeutig bestimmbar [Swe00], 2006 waren es immerhin noch 63,3% [Gol06].

Oft geht das Verständnis des Privacy-Begriffs jedoch über den Schutz personenbezogener Daten hinaus und umfasst auch allgemeine Daten wie beispielsweise wertvolle Forschungsdaten [Sch20]. Wir weiten den Begriff deshalb auf allgemeine Daten aus und betrachten auch solche, die keinen Personenbezug haben, als schützenswert. Genaueres sehen wir uns in Unterabschnitt 3.1.2 an.

Bedrohungen Bevor wir uns mit Lösungen der Privacy-Probleme beschäftigen können, brauchen wir zunächst ein Verständnis für mögliche Angriffsszenarien und -vektoren, die bei Datensätzen erfolgen können. *Gkoulalas-Divanis et al.* nennen in [GLS14] vor allem drei mögliche Bedrohungen durch unzureichende Anonymisierung:

1. **Identity disclosure (Re-Identifizierung)**: Ein Angreifer kann einen Datensatz (ein Tupel) eindeutig einer Person zuordnen. Die Identität wird somit trotz Anonymisierung vollständig aufgedeckt.

   Ein Beispiel dafür sahen wir bereits im Paper von *Latanya Sweeney* [Swe02b]. Durch eine Überlappung zweier Datensätze – einer davon galt als anonymisiert – konnte mindestens eine Person, der Gouverneur von Massachusetts, im anonymisierten Datensatz re-identifiziert werden. Ausschlaggebend war dabei der Quasi-Identifikator bestehend aus PLZ, Geburtstag und Geschlechtseintrag.

2. **Membership disclosure**: Ein Angreifer kann mit hoher Wahrscheinlichkeit sagen, ob eine bestimmte Person in einem anonymisierten Datensatz enthalten ist oder nicht. Das Problem kann auch dann auftreten, wenn der Datensatz vor Re-Identifizierungen geschützt ist und die genaue Identität der Person unbekannt bleibt.

   Beispiel: Ein Unternehmen veröffentlicht eine in Tabelle 2.8 dargestellte Statistik über vergebene Abmahnungen des letzten Geschäftsjahres. In der Abteilung „Logistik“ arbeitet nur ein einziger Mitarbeiter. Die eine Abmahnung, die es in dieser Abteilung gab, ist somit unmittelbar dem Mitarbeiter zuzuordnen. Auch ohne konkreten Rückschluss auf seine Identität wissen wir, dass er in diesem Datensatz enthalten ist.

Abteilung	Abmahnungen
Logistik	1
IT	2
Einkauf	2

Tabelle 2.8.: Beispiel für einen Datensatz, der nicht vor einer membership disclosure geschützt ist

3. **Attribute disclosure**: Ein Angreifer kann eine Person mit einem sensiblen Attribut in Verbindung bringen, auch dann, wenn nicht bekannt ist, welches konkrete Tupel der Person zugehörig ist.

   Beispiel: Ein Unternehmen veröffentlicht den in Tabelle 2.9 dargestellten Datensatz über Mitarbeitende verschiedener Abteilungen und deren Bruttoverdienst pro Monat. In der Abteilung „IT“

arbeiten insgesamt drei Personen. Wir wissen nicht, welches der Tupel zu welcher konkreten Person gehört, da jedoch die Gehälter aller drei Personen übereinstimmen, können wir trotzdem das Gehalt aller drei Personen bestimmen, ohne zu wissen, wer wer ist.

Abteilung	Gehalt (EUR)
Logistik	4.000
IT	5.300
IT	5.300
IT	5.300
Einkauf	3.600
Einkauf	3.700
Einkauf	3.450
Einkauf	3.740

Tabelle 2.9.: Datensatz, der nicht vor einer attribute disclosure geschützt ist

2.3.1. Anonymisierungsmaße

Wir haben nun verschiedene Bedrohungen kennengelernt, die scheinbar anonymisierten Datensätzen einhergehen. In diesem Unterabschnitt schauen wir uns einige Maße an, um auf der einen Seite die Gefahr solcher Angriffe auf Datensätze bzw. auf der anderen Seite die Datenschutzgarantien quantifizieren zu können. Dazu müssen wir zunächst verstehen, wie Datensätze in solchen Szenarien betrachtet werden. Allgemein unterscheiden wir im Anonymisierungsprozess zwischen identifizierenden Attributen, nichtsensiblen Attributen und sensiblen Attributen, in der Regel eines. Ein Beispiel für eine solche Unterteilung sehen wir in Tabelle 2.10.

Identifizierende Attribute ermöglichen einen direkten Schluss auf natürliche Personen, zum Beispiel über eine eindeutige ID (einen Schlüssel) oder persönlichste Attribute wie den Namen. Sie sind in jedem Fall vollständig zu entfernen. **Nichtsensible Attribute** sind Attribute, die weder direkt identifizierend noch sensibel sind. Allerdings sind sie es, die einen indirekten Schluss auf natürliche Personen ermöglichen, wenn sie nicht hinreichend anonymisiert werden. Unter den nichtsensiblen Attributen befinden sich möglicherweise sogenannte **Quasi-Identifikatoren**, die möglicherweise Rückschlüsse auf Identitäten ermöglichen (siehe Definition 2.9). Es handelt sich bei ihnen um Attributkombinationen, die im Datensatz selten genug auftauchen, um für einen Großteil der Datensätze als Schlüssel zu dienen. Wie groß dieser „Großteil“ sein muss, um eine Kombination als Quasi-Identifikator zu qualifizieren, bestimmt beispielsweise der sogenannte **distinct ratio**, der das Verhältnis zwischen der Anzahl *verschiedener* Kombinationen und der Anzahl *aller* Kombinationen einer Menge von Attributen angibt. Die Menge aller Tupel, die den gleichen Quasi-Identifikator besitzen, wird als **Äquivalenzklasse** bezeichnet (siehe Definition 2.10). Das **sensible Attribut** letztendlich ist das Attribut, für das wir uns interessieren, welches jedoch besonders schützenswert ist. In Beispielen in der Fachliteratur handelt es sich dabei häufig um diagnostizierte Erkrankungen, in unserem Beispiel um die ECTS-Note von Studierenden. Da das sensible Attribut von Interesse ist, kann es nicht anonymisiert werden. Theoretisch wären mehrere sensible Attribute möglich, in der Praxis existiert jedoch häufig nur ein solches Attribut.

Definition 2.9 (Quasi-Identifikator, nach [PS17])**.** Ein **Quasi-Identifikator** ist eine Attributkombination, die in Verbindung mit extern verfügbaren Informationen die eindeutige Identifikation einer Person ermöglicht. □

student_id	lastname	firstname	birthday	zipcode	district	grade
10001	Fieber	Fabian	08.03.1998	18059	Südstadt	C
10002	Sonnenschein	Sarah	21.10.1993	18059	Südstadt	B
10003	Müller	Max	22.10.1994	18057	Hansaviertel	A
10004	Deckert	Luisa	22.10.1994	18057	Hansaviertel	A
10005	Gebauer	Luisa	01.02.2000	18106	Evershagen	C
10006	Zimmermann	Jonas	23.07.1999	18106	Evershagen	D

Tabelle 2.10.: Beispiel für eine Unterscheidung zwischen identifizierenden, nichtsensiblen und sensiblen Attributen

Definition 2.10 (Äquivalenzklasse bzgl. Quasi-Identifikatoren, nach [NAC07])**.** Die Äquivalenzklasse QI^* eines Tupels t in einem Datensatz T^* ist die Menge aller Tupel in T^*, die denselben Quasi-Identifikator besitzen. □

k-Anonymität Eine Möglichkeit, einerseits die Güte eines (anonymisierten) Datensatzes zu messen, andererseits Garantien für den Datenschutz zu geben, stellt die **k-Anonymität** dar. Das Ziel ist es, jede Quasi-Identifikator-Äquivalenzklasse (QI^*) soweit zu anonymisieren, dass sie mindestens k identische Werte besitzt. Jede Attributkombination eines QI^*-Blocks setzt also $k-1$ weitere, nicht von diesem Tupel unterscheidbare, Attributkombinationen voraus (siehe Definition 2.11). Eine gut umgesetzte k-Anonymisierung verhindert identity disclosures, da sich die Datensätze hinterher zu stark ähneln, um einzelne Individuen re-identifizieren zu können.

Definition 2.11 (k-Anonymität, nach [PS17])**.** Eine Tabelle (Relation) erfüllt **k-Anonymität**, wenn mindestens k Zeilen (Tupel) in allen zum Quasi-Identifikator gehörenden Spalten (Attributen) identische Werte besitzen. Zu jedem Tupel existieren also mindestens $k-1$ andere, davon auch mit externen Informationen nicht unterscheidbare. □

Unser Beispiel (Tabelle 2.10) weist für den Quasi-Identifikator {`birthday`, `zipcode`, `district`} momentan eine Anonymität von $k = 1$ auf und ist somit 1-anonym, da der kleinste QI^*-Block nur ein Tupel umfasst. In diesem speziellen Fall ist sogar jedes Tupel der QI-Attribute nur einmal vertreten. Im nächsten Unterabschnitt werden wir anhand eines Beispiels lernen, wie wir die k-Anonymität erhöhen können.

l-Diversität Ein weiteres Anonymitätsmaß, welches die k-Anonymität erweitert, ist die sogenannte **l-Diversität**. Dabei wird zusätzlich die Anzahl verschiedener Attributwerte des sensiblen Attributs innerhalb der QI^*-Blöcke gemessen (siehe Definition 2.12). Weist der Wertebereich des sensiblen Attributs bezüglich eines QI^*-Blocks l verschiedene Werte auf, so ist dieser Block l-divers. Für die l-Diversität des gesamten Datensatzes wird – wie auch schon bei der k-Anonymität – das kleinste l gewählt.

Definition 2.12 (l-Diversität, vgl. [MGKV06])**.** Eine k-anonyme Relation erfüllt l-Diversität, wenn der Wertebereich eines sensiblen Attributs pro Äquivalenzklasse mindestens l verschiedene Werte umfasst. □

Für unser Beispiel ergibt sich momentan eine 1-Diversität. In diesem Fall ist dies trivial, da auch nur eine 1-Anonymität vorliegt und jeder QI^*-Block ohnehin nur ein Tupel und somit auch nur einen Attributwert für `grade` umfasst. Im nächsten Unterabschnitt werden wir sehen, wie wir höhere l-Werte herstellen können.

2.3.2. Anonymisierungsmethoden

Nachdem wir im vorherigen Unterabschnitt einige Anonymisierungsmaße kennengelernt haben, schauen wir uns nun verschiedene Anonymisierungsmethoden an.

Generalisieren und Unterdrücken Bei der Methode des **Generalisierens** werden einzelne Attributwerte verallgemeinert, indem (numerische) Werte – sofern sinnvoll – zu Intervallen zusammengefasst oder einzelne Zeichen maskiert werden. Eine solche Generalisierung wird in der Regel attributweise durchgeführt, bis ein gewisses Maß an Anonymität hergestellt wurde. Um diese zu messen, eignet sich das Maß der *k-Anonymität* (siehe Definition 2.11). Eine Generalisierung kann außerdem *global* oder *lokal* erfolgen. Im ersten Fall werden alle Zellen einer Spalte gleichermaßen generalisiert, im zweiten Fall werden unterschiedliche Generalisierungen je nach Notwendigkeit durchgeführt. Ersteres ist einfacher, geht jedoch mit einem höheren Informationsverlust einher, bei letzterer Methode ist es umgekehrt. Beim **Unterdrücken** von Tupeln werden einzelne Zeilen, die zu eindeutig sind, aus der Relation entfernt. Dies ist insbesondere bei Tupeln der Fall, die sich nur sehr schwer einer Äquivalenzklasse zuordnen lassen. Das Löschen einzelner Zeilen kann dann einen geringeren Informationsverlust bedeuten als eine grobe Generalisierung mehrerer Tupel.

Wir wollen dies anhand unseres Beispiels (Tabelle 2.10) ausführlich nachvollziehen. Der Quasi-Identifikator sei die Kombination {`birthday`, `zipcode`, `district`}; das sensible Attribut sei `grade`. Momentan weist die Relation eine 1-Anonymität und damit auch eine 1-Diversität auf, da jedes Tupel nur ein einziges Mal vorkommt. Beispielsweise existiert zu (08.03.1998, 18059, Südstadt) kein äquivalentes Tupel. Da die k-Anonymität das Minimum identischer Tupel innerhalb der QI^*-Klassen angibt, steht die 1-Anonymität bereits jetzt fest, unabhängig davon, ob andere QI^*-Klassen höhere Werte aufweisen, was hier jedoch ohnehin nicht der Fall ist. Unser Ziel soll eine 2-anonyme und 2-diverse Tabelle sein. Um dies zu erreichen, entfernen wir zunächst alle identifizierenden Attribute.

birthday	zipcode	district	grade
08.03.1998	18059	Südstadt	C
21.10.1993	18059	Südstadt	B
22.10.1994	18057	Hansaviertel	A
22.10.1994	18057	Hansaviertel	A
01.02.2000	18106	Evershagen	C
23.07.1999	18106	Evershagen	D

Anschließend müssen wir die Attributwerte des Quasi-Identifikators generalisieren, um die Tupel in größere Äquivalenzklassen zusammenfassen zu können. Hier stehen uns verschiedene Möglichkeiten zur Auswahl. Beispielsweise können wir statt der Geburtstage nur die Jahre oder sogar Jahresintervalle nutzen oder einzelne Stellen der Postzeitzahlen maskieren. Die Herausforderung beim Generalisieren besteht darin, den Informationsverlust so gering wie möglich zu halten, ein Problem, welches NP-schwer ist und deshalb kaum automatisiert werden kann [MW04]. Betrachten wir das Beispiel, stellen wir fest, dass für {`zipcode`, `district`} bereits eine 2-Anonymität besteht (jede Attributkombination ist zweimal im Datensatz enthalten), weshalb wir uns für eine lokale Generalisierung des `birthday`-Attributs entscheiden.

birthday	zipcode	district	grade
1993-1998	18059	Südstadt	C
1993-1998	18059	Südstadt	B
22.10.1994	18057	Hansaviertel	A
22.10.1994	18057	Hansaviertel	A
1999-2000	18106	Evershagen	C
1999-2000	18106	Evershagen	D

Nach dieser lokalen Generalisierung des `birthday`-Attributs ist unser Datensatz 2-anonym, da jeder QI^*-Block (mindestens) zwei Tupel mit dem gleichen Quasi-Identifikator umfasst. Allerdings ist der Datensatz weiterhin nur 1-divers, da der Wertebereich des mittleren Blocks nur den Attributwert „A“ beinhaltet. Da wir 2-Diversität herstellen wollen, müssen wir somit eine weitere Generalisierung durchführen. Dies ist auch deshalb sinnvoll, da wir an dieser Stelle zwar identity disclosures verhindern können, aber unter gewissen Umständen keine membership disclosures und attribute disclosures. Sollte es im gesamten Hansaviertel nur zwei Studierende geben, die am 22.10.1994 geboren wurden, so wissen wir weiterhin, dass beide im Datensatz enthalten sind (membership disclosure). Da zudem die Attributwerte übereinstimmen, kennen wir auch ihre ECTS-Note (attribute disclosure). Eine mögliche Generalisierung, welche alle drei Angriffsszenarien unterbindet, sieht wie folgt aus.

birthday	zipcode	district	grade
1993-1998	1805*	Rostock	C
1993-1998	1805*	Rostock	B
1993-1998	1805*	Rostock	A
1993-1998	1805*	Rostock	A
1999-2000	18106	Evershagen	C
1999-2000	18106	Evershagen	D

In diesem Generalisierungsschritt haben wir die Geburtsdaten noch weiter zusammengefasst, das letzte Zeichen einiger Postleitzahlen maskiert und die Stadtteile durch die übergeordnete Stadt ersetzt. Dadurch erhalten wir zwar einen 2-diversen (und weiterhin 2-anonymen) Datensatz, der Informationsverlust ist jedoch wesentlich größer. Dies ist auch der Natur der l-Diversität geschuldet. Während der k-Parameter der k-Anonymität maximal die Anzahl aller Tupel eines Datensatzes annehmen kann (die ganze Relation wäre dann ein großer QI^*-Block), ist der l-Parameter der l-Diversität auf die Anzahl verschiedener Attributwerte limitiert. Eine Studie von *Justin Brickell* und *Vitaly Shmatikov* aus dem Jahr 2008 kommt zu dem Ergebnis, dass selbst eine k-Anonymität von $k = 1000$ unter Umständen mehr Informationen aufrechterhält als eine l-Diversität von $l = 10$ [BS08].

Differential Privacy Die Methode der Differential Privacy geht vor allem auf die Forschung von *Cynthia Dwork* zurück und wurde erstmals 2006 vorgestellt [Dwo06]. Das Grundprinzip ist es, statistische Datensätze so mit einem zufälligen Rauschen zu versehen, dass zwar Aussagen über Populationen, nicht jedoch über einzelne Individuen getroffen werden können. Ob eine bestimmte Person Teil eines Datensatzes ist oder nicht, soll das Ergebnis nicht (klassische Definition) beziehungsweise nur marginal (ϵ-Differential-Privacy) beeinflussen dürfen. Dazu wird eine Anfrage Q nicht direkt an eine Datenbank D, sondern an eine randomisierte Anfragefunktion $\mathcal{K}$ gestellt (siehe Definition 2.13), welche in einem ersten Schritt die Anfrage $Q(D)$ stellt, in einem zweiten Schritt deren Ergebnis V verrauscht und es erst dann zurückgibt. Die Wahrscheinlichkeit, dass das Entfernen eines einzelnen Tupels das Anfrageergebnis signifikant verändert, muss entsprechend gering sein und wird bei der ϵ-Differential-Privacy durch den Parameter ϵ

repräsentiert. Wie genau dieser Wert zu wählen ist, hängt vom jeweiligen Kontext sowie Anwendungsgebiet ab. Je kleiner ein Datensatz ist, desto höher ist natürlich die Gewichtung eines einzelnen neuen Tupels und desto mehr Rauschen muss hinzugefügt werden, was den Datensatz stärker verfälscht. Analog gilt, dass mit zunehmender Größe des Datensatzes weniger Rauschen benötigt wird, der relative Rausch-Anteil somit sinkt und der Datensatz aussagekräftiger bleibt. Differential Privacy eignet sich somit vor allem für große Datensätze. Weiterhin gilt, dass der Datenschutz umso besser garantiert werden kann, je kleiner das gewählte ϵ ist. Wir können der Definition 2.13 entnehmen, dass ϵ als Exponent der natürlichen Exponentialfunktion genutzt wird, welche wiederum als Koeffizient der Wahrscheinlichkeit „S ist im Anfrageergebnis $\mathcal{K}(D_2)$ enthalten“ dient. Ist $\epsilon = 0$, so ist $e^\epsilon = 1$ und die Wahrscheinlichkeit, dass zwei Datensätze r sowie $r \cup \{t\}$ dasselbe Ergebnis liefern, muss 100% betragen. Je größer der ϵ-Wert gewählt wird, desto unwahrscheinlicher darf dies der Fall sein.

Definition 2.13 (Randomisierte Anfragefunktion, nach [Dwo06])**.** Eine **randomisierte Anfragefunktion** $\mathcal{K}$ bietet ϵ-Differential-Privacy, wenn für alle Datensätze D_1, D_2, die sich in höchstens einem Element unterscheiden, sowie für alle $S \subseteq W(\mathcal{K})$ gilt:

$$\mathbb{P}[\mathcal{K}(D_1) \in S] \leq e^\epsilon \cdot \mathbb{P}[\mathcal{K}(D_2) \in S].$$

□

Neben der randomisierten Anfragefunktion definiert *Dwork* noch den Begriff der Sensitivität, um die inhaltliche Distanz zweier Datensätze zu bestimmen.

Definition 2.14 (Sensitivität, nach [Dwo06])**.** Für $f : \mathcal{D} \to \mathbb{R}^k$ ist die **Sensitivität** von f definiert als

$$\Delta f = \max_{D_1, D_2} ||f(D_1) - f(D_2)||_1$$

für alle D_1, D_2, die sich in höchstens einem Element unterscheiden. □

Im Kontext relationaler Datenbanken kann f als Anzahl aller Tupel einer Relation r aufgefasst werden. Für zwei Relationen r_1, r_2 gilt bei der Betrachtung von Differential Privacy dann $\Delta f = 1$.

Permutationen Ein häufiges Problem bei veröffentlichten Datensätzen ist das Beibehalten der Reihenfolge aller Zeilen/Tupel. Dies kann einen sogenannten **Unsorted-Matching-Angriff** ermöglichen [PS17]. Dabei werden zwei Datensätze mit derselben Sortierung aneinandergelegt und verknüpft, um externe Informationen über die Individuen zu erhalten. Auch hier wollen wir wieder ein Beispiel betrachten. Das Studienbüro veröffentlicht in einem Aushang alle Noten, die in der Prüfung „Datenbanken“ im Sommersemester 2021 erzielt wurden (Anfrage 2.5). Zeitgleich veröffentlicht das Prüfungsbüro eine Liste aller Studierenden, die diese Prüfung absolvierten (Anfrage 2.6). Beide Datensätze liefern alleinstehend wenig Informationen, da jedoch die Sortierung dieselbe ist, lassen sie sich problemlos nebeneinanderlegen und kombinieren. Jeder Person kann problemlos ihre Note zugeordnet werden. Wie bereits in [Sch20] aufgezeigt, kann dieses Problem sogar bei der ***how***-Provenance für Aggregate auftreten. Berechnen wir den Durchschnitt aller vier Noten, erhalten wir den Wert 1,65 und das Provenance-Polynom

$$\frac{(s_2 \odot 1.3) \oplus (s_3 \odot 1.3) \oplus (s_7 \odot 3.0) \oplus (s_{14} \odot 1.0)}{s_2 \oplus s_3 \oplus s_7 \oplus s_{14}},$$

aus welchem direkt die Reihenfolge s_2, s_3, s_7, s_{14} abgeleitet werden kann. Mithilfe dieses Polynoms ist es also zumindest theoretisch möglich, einen Unsorted-Matching-Angriff sogar auf aggregierte Werte durchzuführen.

grade	
1.3	s_2
1.3	s_3
3.0	s_7
1.0	s_{14}

(a) Noten-Aushang mit ergänzter ***how***-Provenance

firstname	lastname
Sarah	Sonnenschein
Max	Müller
Franziska	Bach
Mira	Müller

(b) Teilnehmenden-Aushang

Tabelle 2.11.: Beispiel für einen Unsorted-Matching-Angriff

Ein solcher Angriff setzt natürlich Wissen über die genutzte Sortierung voraus. Da jedoch viele DBMS mit Indizes und Caches arbeiten[3] und die Sortierung zwischen zwei Anfragen häufig automatisch beibehalten wird, sollte dieses Problem nicht unterschätzt werden. Die Lösung dafür ist eine **Permutation** der Tupel (bzw. im Provenance-Fall der Teilpolynome), also ein Variieren der Reihenfolge.

```
SELECT grade
FROM Exams
WHERE module_id = 8 AND semester = 'SS 21'
ORDER BY student_id ASC;
```

Anfrage 2.5: Welche Noten wurden im Sommersemester 2021 in der Prüfung „Datenbanken“ vergeben?

```
SELECT firstname, lastname
FROM Students NATURAL JOIN Exams
WHERE module_id = 8 AND semester = 'SS 21'
ORDER BY student_id ASC;
```

Anfrage 2.6: Wer absolvierte im Sommersemester 2021 die „Datenbanken“-Prüfung?

Intensionale Antworten Bei **intensionalen Provenance-Antworten** werden Provenance-Anfragen nicht wie bei extensionalen Antworten durch das Veröffentlichen eines Teildatensatzes oder Tupel-Identifikatoren beantwortet, sondern durch eine abstraktere, allgemeinere Beschreibung des Ergebnisses. Diese kann beispielsweise textuell erfolgen; wollen wir, wie im vorherigen Absatz, die Durchschnittsnote 1,65 nachvollziehen können, kann dies nicht nur durch die Angabe der beteiligten Originaldaten erfolgen, sondern zum Beispiel auch durch die Aussage „Werte: 1.3, 1.3, 3.0, 1.0“. *Jan Svacina* beschäftigte sich 2016 in seiner Bachelorarbeit *„Intensional Answers for Provenance Queries in Big Data Analytics“* mit genau diesem Thema [Sva16]. Darin wird die Provenance-Software *PERM* verwendet, um statt eines Anfrageergebnisses einer relationalen Datenbank alle Tupel zu erhalten, die die in einer Anfrage formulierten Selektionen erfüllen. In einem weiteren Schritt wird dieses Ergebnis mittels Konzepthierarchien generalisiert und identische Zeilen zusammengefasst. Dies stellt dann eine intensionale Antwort dar. *Maximilian Lamster* entwickelte 2021 in seiner Masterarbeit *„Provenance-unterstütze Datenanalyse in Kombination mit intensionalen Antworten zur Steigerung der Privatsphäre“* eine ähnliche Umsetzung auf Basis von Konzeptrelationen, welche Generalisierungshierarchien durch zusätzliche Relationen darstellen [Lam21]. In beiden Fällen werden die relevanten Originaldaten ermittelt und bestmöglich und mit zeilenweiser Duplikateliminierung generalisiert. Je nach Ansatz wird dabei auch die Anzahl der zusammengeführten Tupel gespeichert, um später feststellen zu können, wie oft ein generalisiertes Tupel im Datensatz auftauchte. Je mehr Tupel zusammengefasst werden können und je geringer somit die finale Menge von Tupeln ist, desto einfacher lässt sich eine intensionale Antwort aufstellen. Diesen Ansatz verfolgten auch schon *Park* und *Yoon* im Artikel *„An Approach to Intensional Query Answering at Multiple Abstraction*

[3] Es sei erneut auf [HSS18] verwiesen.

Levels using Data Mining Approaches“ [PY99]. In ihrem Beispiel interessieren sie sich für Autoverkäufe und stellen die Anfrage, welche Menschen typischerweise Sportwagen kaufen. Im ersten Schritt erhalten sie folgendes Ergebnis.

Name	Typ	Alter	Einkommen
Miller	Sportwagen	28	58000
Johnson	Sportwagen	33	60000
Wallace	Sportwagen	30	65000

Dieses Ergebnis gilt es nun möglichst so zu generalisieren, dass Tupel zusammengefasst werden können. Im ersten Schritt wird das identifizierende Attribut `Name` entfernt. Das Alter wird mittels einer vorab definierten Konzepthierarchie für alle drei Tupel zu „jung“ generalisiert. Zuletzt wird das Einkommen in allen drei Fällen zu „hoch“ generalisiert. Die entsprechende Hierarchie liegt auch hier bereits vor. Nachfolgend sehen wir das Ergebnis nach diesen Generalisierungen.

Typ	Alter	Einkommen
Sportwagen	jung	hoch
Sportwagen	jung	hoch
Sportwagen	jung	hoch

Da die drei Tupel in allen Werten übereinstimmen, können sie jetzt zusammengefasst werden. Je nach konkreter Umsetzung dieses Prinzips merken wir uns zusätzlich die Anzahl der zusammengefassten Tupel, hier dargestellt durch das neue `count`-Attribut. Am Ende erhalten wir ein (intensionales) Ergebnis, welches wie folgt aussieht.

Typ	Alter	Einkommen	Count
Sportwagen	jung	hoch	3

Daraus können wir nun die intensionale Antwort auf die Frage „Welche Menschen kaufen Sportwagen?“ ableiten. Diese lautet „(3) junge Menschen mit hohem Einkommen“. Der Vorteil besteht darin, dass diese Antwort sehr viel kompakter und verständlicher ist als eine extensionale Antwort, der Nachteil besteht jedoch darin, dass auch der Informationsgehalt sehr viel geringer ist und wir die Originaldaten am Ende nicht rekonstruieren können. Die Antwort ist diesem Beispiel ist auch nicht mehr nachvollziehbar, sondern maximal plausibel.

In diesem Kapitel haben wir uns die wichtigsten Grundlagen der Data Provenance sowie Privacy angeeignet, die wir für ein Verständnis dieser Masterarbeit benötigen. Im ersten Teil haben wir mit der ***where***-, ***why***- und ***how***-Provenance drei Provenance-Arten kennengelernt, die extensionale Antworten auf Provenance-Anfragen liefern und dadurch beispielsweise eine Minimierung der Datenbank ermöglichen. Im zweiten Teil haben wir einige Probleme kennengelernt, die mit unzureichend anonymisierten Datensätzen einhergehen können, sowie verschiedene Anonymisierungsmethoden und -maße betrachtet, die diesen Problemen entgegenwirken können. Im nächsten Kapitel beschäftigen wir uns über die Grundlagen hinaus mit dem aktuellen Stand der Forschung und Technik.

3. Stand der Forschung und Technik

3.1. Stand der Forschung

In diesem Abschnitt werfen wir einen Blick auf aktuelle Forschungsergebnisse zu den Themen Provenance und Privacy. Dabei beschäftigen wir uns zunächst mit bestehenden Ansätzen zur Kombination von Provenance und Privacy und anschließend mit dem Privacy-Begriff selbst, bevor wir eine ausführlichen Blick auf Domain-Generalisierungs-Hierarchien werfen und weitere Anonymisierungsmaße kennenlernen, die über die Grundlagen hinausgehen.

3.1.1. Provenance

In diesem Unterabschnitt betrachten wir aktuelle Forschungsansätze, um Provenance mit Privacy-Aspekten zu verbinden. Eine der aktuellsten Veröffentlichungen stellt der Artikel *„On Optimizing the Trade-off between Privacy and Utility in Data Provenance“* von *Deutch et al.* dar [DFGM21a]. Darin wollen die Autoren die Privacy einer Provenance-Auswertung erhöhen, indem die zugrundeliegende Anfrage an die Datenbank verschleiert wird. Dies erfolgt durch eine Generalisierung der ***how***-Provenance mittels Konzepthierarchien. Dieses Verallgemeinern der Provenance-Polynome erfolgt so lange, bis k Anfragen existieren, welche zur selben verallgemeinerten Provenance führen, ähnlich wie bei der k-Anonymität. Dazu wurde eine eigene Software namens *PITA* entwickelt, auf welche wir in Abschnitt 3.2 noch einmal aufgreifen werden.

Auch die studentischen Abschlussarbeiten von *Jan Svacina* und *Maximilian Lamster* sind an dieser Stelle erwähnenswert [Sva16] [Lam21]. Beide legen den Fokus auf intensionale Antworten, welche mittels Teile der Originaldaten erzeugt werden sollen. Auch hier werden diese Daten mithilfe von Konzepthierarchien so lange generalisiert, bis sich möglichst viele Tupel zusammenfassen lassen, um das Ergebnis auf einige wenige Aussagen reduzieren zu können. Konzepthierarchien, auch (Domain-)Generalisierungshierarchien genannt, sind somit ein weit verbreiteter Ansatz, um Provenance-Auswertungen hinsichtlich bestimmter Privacy-Aspekte wie der k-Anonymität zu erweitern. Wir werden diese im Unterabschnitt 3.1.3 deshalb im Detail betrachten.

Allgemein existiert momentan jedoch wenig Forschung auf diesem Gebiet. Zwar wurden in den letzten Jahren einige Artikel veröffentlicht, die Provenance und Privacy kombinieren, diese haben jedoch erheblich andere Anwendungsszenarien oder nutzen eine andere Art der Provenance, beispielsweise die Workflow Provenance. Eine Auswahl dieser Artikel ist nachfolgend aufgelistet.

- Der Artikel *„An FPGA Implementation of Privacy Preserving Data Provenance Model Based on PUF for Secure Internet of Things“* von *Hamadeh* und *Tyagi* beschäftigt sich mit der Data Provenance für sogenannte „Internet of Things“-Geräte (*IoT*) wie Smart-Home-Sensoren [HT21]. Der Fokus liegt hier jedoch nicht auf Anonymisierungsmethoden für diese Daten, sondern auf der Integrität und Sicherheit der Daten, die auf solchen Geräten anfallen und übermittelt werden, beispielsweise an einen zentralen Server, welche diese Informationen dann auswertet. Die Zielstellungen der Autoren liegen vor allem auf der Sicherung der Integrität dieser Daten sowie den Schutz der Daten

gegenüber Dritten. Beides wird mithilfe kryptografischer Verfahren erreicht, indem die Daten verifiziert und verschlüsselt werden. Mit unserem Anwendungsfall von Privacy hat dies jedoch nichts zu tun.

- Der Artikel *„A novel approach to provenance management for privacy preservation“* von *Can* und *Yilmazer* beschäftigt sich mit Privacy-Verletzungen bei der Information System Provenance [CY20]. Auch hier geht das Ziel jedoch in eine andere Richtung als bei uns. Der Fokus liegt auf das Erfassen, Löschen und Abrufen von personenbezogenen Daten in Informationssystemen, beispielsweise einer medizinischen Datenbank. Um den Verlauf dieser Operationen nachvollziehen zu können, werden bei jeder Modifizierung eines Datensatzes Informationen darüber gespeichert, wer die Änderungen durchgeführt hat, welche Operationen auf den Daten ausgeführt wurden, welche Zugriffsberechtigungen für diese Daten gelten und ob dabei gegen Datenschutzbestimmungen verstoßen wurde. Diese Informationen stellen die Information System Provenance dar. Das Ziel des Artikels ist es, letztere Informationen, also mögliche Datenschutzverstöße, automatisch zu identifizieren.

Damit haben wir einen Überblick über den aktuellen Stand der Forschung zur Kombination von Privacy-Aspekten mit Data Provenance erhalten. Im nächsten Unterabschnitt betrachten wir nun aktuelle Forschung zu Privacy-Themen als alleinstehendes Fachgebiet.

3.1.2. Privacy

Wie bereits in Abschnitt 2.3 erwähnt, umfasst das allgemeine Verständnis des Privacy-Begriffs mehr als nur den Schutz personenbezogener Daten. Ferner fallen darunter alle möglichen (Forschungs-)Daten, die als schützenswert betrachtet werden können, beispielsweise weil ein monetärer Aufwand erbracht wurde, um diese Daten zu sammeln. Dies zeigt eine Umfrage aus dem Jahr 2020 [ASH21b]. Diese umfasst drei Abschnitte: im ersten Abschnitt wurden die Teilnehmerinnen und Teilnehmer nach ihrer persönlichen Situation und ihren eigenen Umgang mit Daten befragt. Im zweiten Abschnitt erhielten sie Fragen zu den Themen Provenance und Privacy. Hier wurde bewusst nicht der Datenschutz-Begriff gewählt (engl. data protection), sondern der allgemeinere Privacy-Begriff. Im dritten Teil wurden den Beteiligten schließlich Fragen zum Thema des Forschungsdatenmanagements gestellt. Die 21 Teilnehmer/-innen (ein Interview wurde mit zwei Beteiligten geführt, deren Aussagen zusammengefasst wurden) wurden dabei möglichst vielfältig gewählt. Unter ihnen waren fünf Personen aus dem Wissenschafts- oder Ingenieurwesen, zwei Geisteswissenschaftler/-innen, fünf Personen aus dem Datenmanagement, zwei Softwareentwickler/-innen sowie zwei Personen aus der Lehre. Ferner beschrieben kamen zehn Befragte aus einem wissenschaftlichen Bereich, sieben aus einem nichtwissenschaftlichen Bereich und vier waren Studierende.

70% der Teilnehmer/-innen bringen den Privacy-Begriff zunächst allgemein mit personenbezogenen Daten in Verbindung. Einige assoziieren ihn jedoch bereits hier mit allgemeinen Daten, darunter (zurückgehaltene) Forschungsdaten, und unterscheiden explizit zwischen „persönlichen“ und „personenbezogenen“ Daten. Persönliche Daten müssen somit nicht zwangsläufig personenbezogen sein, was Forschungsdaten jeglicher Art in den Privacy-Begriff einschließt. Weitere Antworten, was unter dem Privacy-Begriff zu verstehen ist, waren das Verhindern von Re-Identifizierungen natürlicher Personen, das Verhindern, dass etwas zu öffentlichem Wissen wird sowie das Recht auf informative Selbstbestimmung. Auch als Anwendungsfälle wurden nicht nur der personenbezogene Datenschutz im klassischen Sinne genannt, sondern auch der Schutz militärischer sowie Regierungsdaten, Forschungsdaten jeglicher Art, Daten aus Kooperationen mit Dritten wie bspw. privatwirtschaftlichen Unternehmen sowie Daten, die möglicherweise patentierbar sind.

zip code	date of birth	gender	diagnosis
18059	06.03.1998	male	influenza
18055	21.09.1995	female	depression
18106	29.02.1994	male	heart attack
...	...	...	...

Tabelle 3.1.: Unzureichend anonymisierter Teildatensatz, der den Befragten präsentiert wurde [ASH21b]

Anschließend wurde den Teilnehmerinnen und Teilnehmern ein Auszug aus einem medizinischen Datensatz gezeigt, der zwar keine direkt identifizierenden Attribute, aber mit der Kombination aus Postleitzahl, Geburtstag und Geschlechtseintrag einen Quasi-Identifikator enthält, zu sehen in Tabelle 3.1. Die Teilnehmer/-innen wurden gefragt, ob dieser Datensatz so veröffentlicht werden könnte. Lediglich eine befragte Person antwortete mit Ja, der Rest und somit 95% der Befragten erkannten die Probleme, die damit einhergehen. Daraufhin wurde die Anschlussfrage gestellt, welche Möglichkeiten es gibt, den Datensatz zu anonymisieren. Hier wurden vor allem das Entfernen gesamter Spalten sowie Generalisierungen als Methoden genannt. Insbesondere das Geburtsdatum sowie die Postleitzahl wurden als kritische Attribute erkannt. Ein Verständnis von Privacy-Problemen, insbesondere identity disclosures, ist bei den Befragten somit allgemein vorhanden. Abschließend lässt sich feststellen, dass der Privacy-Begriff nicht nur den personenbezogenen Datenschutz im klassischen Sinne umfasst, sondern auch schützenswerte Daten jeglicher Art. Dazu gehören neben aufwändig erhobenen Forschungsdaten zur Monetarisierung oder exklusiven Weiterverwendung beispielsweise auch militärische Daten oder Regierungsdaten, deren vollständige Veröffentlichung möglicherweise sogar eine Gefahr für die öffentliche Sicherheit darstellen können. Wir fassen deshalb auch Datensätze ohne Personenbezug als schützenswert auf.

Im nächsten Unterabschnitt beschäftigen wir uns nun mit Generalisierungshierarchien, auch als Konzepthierarchien bekannt, die uns dabei helfen, einen Datensatz für eine Veröffentlichung adäquat zu anonymisieren.

3.1.3. Domain-Generalisierungs-Hierarchien

Domain-Generalisierungs-Hierarchien, auch als **Konzepthierarchien** bekannt, sind ein Prinzip, welches vor allem bei der intensionalen Beantwortung von Provenance-Anfragen eine wichtige Rolle spielt. Dabei werden einzelne Werte schrittweise generalisiert, indem sie semantisch nach sogenannten Konzepten zusammengefasst werden. Anstelle konkreter Werte kann dann ein übergeordneter, allgemeinerer Wert zurückgegeben werden. *Jan Svacina* hat sich 2016 bereits in seiner Bachelorarbeit „Intensional Answers for Provenance Queries in Big Data Analytics“ mit dem Nutzen von Konzepthierarchien bei intensionalen Antworten beschäftigt [Sva16]. Dabei wird eine auf *PostgreSQL* basierende Provenance-Software namens *PERM* mit ebendiesen Hierarchien kombiniert, um gestellte Provenance-Anfragen intensional statt – wie üblich – extensional beantworten zu können. Ebenso stellt er im Ausblick seiner Arbeit fest, dass damit möglicherweise Privacy-Probleme einhergehen und schlägt als Lösungsansatz eine Einschränkung der Granularität sowie das vollständige Sperren sensibler Attribute vor. Definition 3.1 beschreibt seine Definition des Begriffs der Konzepthierarchie.

Definition 3.1 (Konzepthierarchie, nach [Sva16])**.** Eine Konzepthierarchie ist ein Baum, der aus mindestens drei Ebenen besteht. Die erste Ebene bildet die Wurzel des Baums uns enthält den allgemeinsten Begriff. Die Zwischenebenen müssen semantisch in einer verallgemeinernden Beziehung stehen und werden **Konzepte** genannt. In den Blättern befinden sich jeweils die konkreten Werte der Konzepthierarchie. □

Im Artikel „Data-Driven Discovery of Quantitative Rules in Relational Databases“ von *Han et al.* wird das Konzept ebenfalls im Kontext intensionaler Provenance-Antworten vorgestellt [HCC93]. Die Autoren

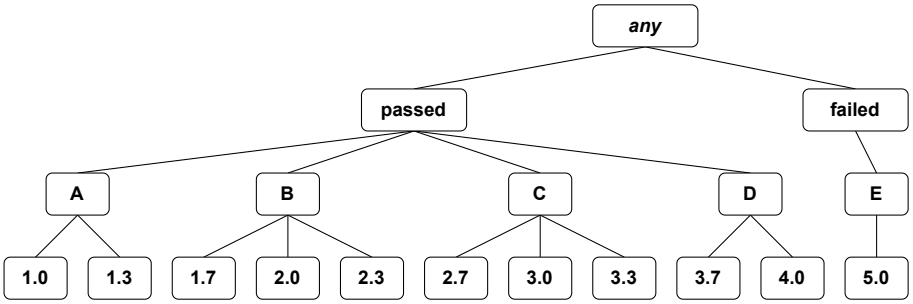

Abbildung 3.1.: Eine mögliche Konzepthierarchie für das Attribut `grades`

betonen, dass eine solche Hierarchie in der Regel von einem **Domain-Experten** manuell erstellt werden muss, um einen optimalen semantische Nutzen zu erzielen. Zwar wäre es theoretisch möglich, diese zumindest bei numerischen Werten auch durch Data-Mining-Methoden wie Clustering oder der Zerlegung in (statistisch sinnvollen) Intervalle aufzustellen, dabei können jedoch Konzepthierarchien entstehen, die semantisch unbrauchbar sind oder nur einen geringen Mehrwert liefern. Weiterhin ist es nicht immer möglich oder sinnvoll, überhaupt eine Konzepthierarchie aufzustellen. Dies kann bei einem Attribut der Fall sein, deren Domäne (Wertebereich) sich nicht sinnvoll zusammenfassen lässt. Eine Generalisierung der Werte von `student_id` in der STUDENTS-Relation aus unserem Beispiel (Tabelle 1.1) ist zwar prinzipiell möglich (beispielsweise können die Werte zu den Intervallen [10000, 10004], [10005, 10009], [10010, 10014] sowie [10015, 10019] generalisiert werden), liefert aber keinen informativen Mehrwert. Andere Werte, wie die Namen der Studierenden (im Beispiel `lastname` bzw. `firstname`), lassen sich hingegen überhaupt nicht semantisch sinnvoll generalisieren.

Aufbauend auf der Arbeit von [HCC93] haben *Park* und *Yoon* 1999 eine Methode entwickelt, um Provenance-Anfragen bei einer gegebenen Hierarchie intensional bestimmen zu können [PY99]. Dieser Ansatz besteht aus drei Phasen: der Vorberechnung, dem Ausführen der Anfrage sowie die Generation der Antwort. Uns interessiert vor allem die erste Phase. In dieser Vorberechnungsphase werden zwei Arten von Hierarchien aufgestellt: zum einen eine Menge der bereits beschriebenen Konzepthierarchien, zum anderen eine Menge von „virtuellen Hierarchien“ für eine globale Darstellung von Beziehungen zwischen verschiedenen Konzepten der verschiedenen Hierarchien. Dabei können verschiedene Konzepte generalisierend zusammengefasst werden, allerdings hierarchieübergreifend. Wie in [HCC93] bereits erwähnt, müssen diese Hierarchien in der Regel durch einen Domain-Experten vorgegeben werden. Die Autoren stellen jedoch fest, dass sich eine Domäne allgemein nicht generalisieren lässt, sobald eine der folgenden zwei Eigenschaften erfüllt ist:

1. Die Domäne besteht aus einer großen Menge verschiedener nicht-numerischer Werte, es existieren jedoch keine allgemeineren Konzepte, die diese beschreiben bzw. es ist nicht möglich, die Werte sinnvoll zu abstrahieren.

2. Die Domäne besteht aus einer großen Menge verschiedener numerischer Werte, doch es existiert, ausgehend von der statistischen Verteilung der Werte, nur eine endliche Menge von Intervallen.

Dies ist häufig insbesondere bei Schlüsselattributen der Fall. Allerdings würde eine Generalisierung hier, wie bereits anhand der `student_id` dargestellt, oft auch keinen informativen Mehrwert liefern. Ein Beispiel für eine sinnvolle Generalisierung stellt das Attribut `grade` der Relation EXAMS dar (Tabelle 1.3). Dies ist in Abbildung 3.1 zu sehen. Die einzelnen Noten können hier zunächst in einem ersten Schritt zu den Stufen A bis E zusammengefasst werden. Ein weiteres Konzept auf der nächsthöheren Ebene ist die Abstraktion in „bestanden“/„passed“ und „nicht bestanden“/„failed“. Schlussendlich stellt die Wurzel des Baums den Wert „any“ für einen beliebigen Wert dar.

3.1.4. Anonymisierungsmaße

Im Grundlagen-Kapitel 2 haben wir mit der k-Anonymität sowie der l-Diversität bereits zwei Anonymisierungsmaße kennengelernt. Darüber hinaus existieren diverse weitere Maße, welche in der Grundlagenliteratur oft keine Beachtung finden, in der Praxis aber durchaus relevant sein können. Deshalb beschäftigen wir uns im folgenden Abschnitt mit einigen weiteren Anonymisierungsmaßen.

k-map Eine Erweiterung der k-Anonymität stellt die sogenannte k-map dar [ED08]. Dabei wird nicht nur sichergestellt, dass jede Äquivalenzklasse hinsichtlich eines Quasi-Identifikators mindestens k nicht voneinander unterscheidbare Tupel enthält, sondern auch, dass es in der Grundgesamtheit aller Individuen – in der Praxis gesamte Städte, Länder oder die gesamte Welt – mindestens k Individuen gibt, die im Datensatz repräsentiert werden. Jeder Quasi-Identifikator muss also im Universum mindestens k mal vertreten sein, damit eine k-map gegeben ist. Auf diese Weise kann eine membership disclosure und unter Umständen auch eine attribute disclosure verhindert werden. Wir betrachten dazu exemplarisch einen anonymisiert veröffentlichten Datensatz sowie einen Auszug aus der dazugehörigen Grundgesamtheit (Tabelle 3.2).

Unter gegebenen $k, l = 2$ ist dieser Datensatz als anonymisiert (2-anonym und 2-divers) zu betrachten. Sollte es jedoch – wie in unserem Beispiel – im PLZ-Bereich 18055 insgesamt nur 2 Männer geben, die 1931 geboren sind, so steht fest, dass beide im Datensatz enthalten und somit gesichert erkrankt sind (membership disclosure). Wären auch noch die Attribute identisch, also nur 1-Diversität gegeben, so könnte man ebenfalls auf die Diagnose schließen, was zusätzlich eine attribute disclosure darstellt. Um dem entgegenzuwirken, wurde das Konzept der k-map entwickelt. In diesem Beispiel wäre $k = 2$, da das Tupel (`18055`, `1931`, `m`) in der Grundgesamtheit nur 2 Mal vertreten ist. Der entscheidende Nachteil besteht jedoch darin, dass dieses Maß universelles Wissen über eine Grundgesamtheit voraussetzt, in diesem Beispiel die Gesamtheit aller Einwohnerinnen und Einwohner des entsprechenden PLZ-Gebietes. In der Praxis scheitert dies oft an realen Bedingungen, insbesondere dann, wenn man das Konzept auf ganze Städte, Länder oder gar die gesamte Welt ausweitet.

delta-presence Zu guter Letzt werfen wir einen kurzen Blick auf die sogenannte δ-presence. Diese stellt den Quotienten aus k-Anonymität und k-map dar und gibt somit das Verhältnis zwischen den Individuen im anonymisierten Datensatz und den Individuen in der Grundgesamtheit an. Bei der Quantifizierung, wann ein Datensatz „gut genug“ ist, um als anonym zu gelten, wird die δ-presence üblicherweise als Tupel $(\delta_{min}, \delta_{max})$ aus einer unteren Grenze δ_{min} und einer oberen Grenze δ_{max} angegeben [NAC07]. δ_{max} gibt dabei an, wie hoch die Wahrscheinlichkeit sein darf, dass eine Person der Grundgesamtheit im anonymisierten Datensatz enthalten ist. Je geringer dieser Wert ist, desto eher kann Anonymität

zipcode	born	sex	disease
18055	1931	m	Rheumatism
18055	1931	m	Tuberculosis
18059	1974	f	Asthma
18059	1974	f	CFS

(a) Veröffentlichter Datensatz ($k, l = 2$)

zipcode	born	sex
18051	1987	m
18055	1931	m
18055	1931	m
18059	1974	f
18059	1974	f
18059	1974	f
18059	1975	m
18059	1975	m
...	...	...

(b) Auszug der Grundgesamtheit

Tabelle 3.2.: Beispiel für einen hinsichtlich k-map unzureichend anonymisierten Datensatz

gewährleistet werden. Die untere Grenze δ_{min} hingegen gibt an, wie wahrscheinlich eine Person nicht im anonymisierten Datensatz enthalten sein darf. Diesem Maß liegt die Annahme zugrunde, dass auch das Nicht-Enthaltensein in einem Datensatz Informationen über eine bestimmte Person offenbaren kann. Denken wir beispielsweise an einen anonymisierten Datensatz, der alle Personen beinhaltet, die vor einer Veranstaltung negativ auf das SARS-CoV-2-Virus getestet wurden (Tabelle 3.3). Wissen wir von einer Freundin aus dem eigenen Bekanntenkreis, die in Rostock-Südstadt (18059) wohnt sowie 1973 geboren wurde und an jener Veranstaltung teilnehmen wollte, stellen wir fest, dass kein passendes Tupel im Datensatz auftaucht. Daraus können wir schlussfolgern, dass diese Person positiv auf SARS-CoV-2 getestet wurde und erhalten somit – wenn auch implizit – eine brisante Information. Bemerkenswert hierbei ist außerdem, dass dazu nicht einmal ein sensibles Attribut im Datensatz vorliegen muss. Es war jedoch nach bestem Wissen und Gewissen nicht möglich, einen praktischen Fall zu finden, in dem das Fehlen einer Person in einem Datensatz tatsächlich zu einem Problem wurde. Häufig wird diese untere Grenze deshalb vernachlässigt. Allgemein lässt sich die δ-presence wie folgt definieren.

Definition 3.2 (δ-presence, nach [NAC07])**.** Sei P ein extern verfügbarer, öffentlicher Datensatz und T ein privater, zu anonymisierender Datensatz. Die δ-presence mit $\delta = (\delta_{min}, \delta_{max})$ gilt für eine Generalisierung T^* genau dann, wenn

$$\forall t \in P : \delta_{min} \leq P(t \in T \,|\, T^*) \leq \delta_{max}.$$

□

Diese Definition gibt sowohl für δ_{min} als auch δ_{max} die Wahrscheinlichkeiten an, dass ein Tupel des generalisierten, privaten Datensatzes T/T^* auch im öffentlichen Datensatz P enthalten ist. Eine eigene, etwas einfachere Definition, welche δ_{min} vernachlässigt, jedoch direkten Bezug auf k-Anonymität und k-map nimmt, kann wie folgt aussehen:

Definition 3.3 (δ-presence)**.** Sei k_{anon} ein gegebener Parameter für k-Anonymität und k_{map} ein gegebener Parameter für k-map. Der Quotient $\delta = \delta_{max}$ stellt das Verhältnis

$$\delta = \frac{k_{anon}}{k_{map}}$$

dar und befindet sich im Intervall $(0, 1]$, wobei der garantierte Datenschutz zunimmt, je kleiner δ ist. □

In unserem Beispiel ergibt sich sowohl für k_{anon} als auch für k_{map} ein Wert von 2 und folglich ein $\delta = \frac{2}{2} = 1$. Die Wahrscheinlichkeit, dass eine Person der Grundgesamtheit im anonymisierten Datensatz enthalten ist, beträgt somit 100% und der Datenschutz kann nicht gewährleistet werden. Eine Möglichkeit, diesen Umstand zu verbessern, besteht darin, Tupel im veröffentlichten Datensatz zu unterdrücken. Entfernen wir eines der zwei (`18055`, `1931`, `m`, `_`)-Tupel, erhalten wir ein $\delta = \frac{1}{2} = 0.5 = 50\%$, verlieren aber eben auch ein Tupel und somit Informationen.

3.2. Stand der Technik

In diesem Abschnitt betrachten wir den aktuellen Stand der Technik. Dazu betrachten wir verschiedene Softwareprojekte, die bereits existieren und frei verfügbar sind (*ARX*) oder sich noch in Entwicklung befinden, aber bereits wichtige Grundfunktionalitäten bieten (*PITA*, *ProSA*, *ChaTEAU*).

zipcode	born	sex
18055	1931	m
18055	1931	m
18059	1974	f
18059	1974	f
18059	1974	f
18059	1975	m
18059	1975	m
...	...	...

Tabelle 3.3.: Beispiel für einen Datensatz, bei dem das Fehlen einer Person ihren Datenschutz verletzen kann

3.2.1. ARX

Die in Java entwickelte *ARX*-Software[1] wurde erstmals 2015 in dem Artikel *„ARX – A Comprehensive Tool for Anonymizing Biomedical Data“* von *Prasser et al.* vorgestellt [PKLK14]. Heute gilt es als de-facto-Standard bei der Anonymisierung von wissenschaftlichen Datensätzen. Zum einen bietet ARX die Möglichkeit, Datensätze aus unterschiedlichsten Quellen – darunter CSV-Dateien und relationale Datenbanken – einzulesen und zu anonymisieren. Dabei werden gegenwärtig k-Anonymität, k-map, δ-presence sowie eine Variante der Differential Privacy – die (ϵ, δ)-differential privacy – unterstützt. Jede Anonymisierungsmethode kommt mit einer Menge von Parametern daher, um beispielsweise besonders informationshaltige Attribute zu priorisieren. Abgesehen von der Differential Privacy nutzt *ARX* vor allem die Methode des Generalisieren und Unterdrückens, um den eingelesenen Datensatz zu anonymisieren. Dabei können über eine grafische Benutzeroberfläche (GUI) für jedes Attribut Generalisierungshierarchien erstellt werden. Je nach Art des Attributs werden dabei Intervalle, Maskierungen und semantische Konzepthierarchien unterstützt. Zum anderen können anonymisierte Datensätze auf ihre Risiken untersucht und Annahmen über den gewährleisteten Datenschutz getroffen werden. Eine ausführliche Beschreibung sowie Auswertung dieser Analysemethoden ist in dem Artikel *„Putting Statistical Disclosure Control into Practice: The ARX Data Anonymization Tool“* nachzulesen [PK15]. Zudem verfügt *ARX* über eine umfangreich dokumentierte[2] API, welche zum einen die Einbindung der Software „als Library“ in andere Java-Projekte ermöglicht, zum anderen aber auch eine Alternative zur GUI darstellt. Mithilfe der API können Daten eingelesen, Anonymisierungsmethoden und -maße festgelegt sowie Konzepthierarchien definiert und aufgestellt werden. Es existiert eine große Menge an Java-Klassen, die die verschiedenen Strukturen und Operationen abbilden und eine intuitive Programmierung ermöglichen.

Einordnung *ARX* unterstützt viele der Anonymisierungsmethoden und -maße, die – wie wir in Kapitel 4 sehen werden – für *ProSA* in Betracht kommen. Der Blick auf die Software ist deshalb von besonderer Relevanz. Doch auch der Blick „hinter die Kulissen“ hilft uns, eine Vorstellung für die praktische Implementierung in *ProSA* zu bekommen. Für jede mögliche Generalisierung existieren Klassen, die diese Generalisierungen implementieren. Eigene Konzepthierarchien können schrittweise mittels einfacher Funktionsaufrufe aufgestellt werden. Dies ermöglicht in Verbindung mit Input- und Output-Services auch das Einlesen von Hierarchien aus Dateien. Insbesondere macht *ARX* bei vielen Methoden von abstrakten, vererbenden Oberklassen und konkreten Unterklassen Gebrauch, um verschiedene Arten von Generalisierungen und Konzepten gleichermaßen behandeln zu können. Diese Konzepte sind für *ProSA* selbst vielversprechend und werden daher, wie wir in Kapitel 4/5 sehen werden, bei der praktischen Implementierung berücksichtigt werden.

[1] `https://arx.deidentifier.org/`, zuletzt abgerufen: 10.03.2022 16:52+0100
[2] `https://arx.deidentifier.org/wp-content/uploads/javadoc/current/api/index.html`, zuletzt abgerufen: 10.03.2022 17:05+0100

3.2.2. PITA

Daniel Deutch, *Ariel Frankenthal*, *Amir Gilad* und *Yuval Moskovitch* entwickeln eine Software namens *PITA*, welche die Privacy bei Provenance-Ergebnissen erhöhen soll, indem der Informationsgehalt hinreichend abgeschwächt wird, um Rückschlüsse auf die zugrundeliegende Anfrage zu vermeiden [DFGM21b]. Der gewählte Ansatz ist somit eine Anonymisierung der Provenance-Polynome, als schützenswert wird die Anfrage betrachtet. Dazu werden Generalisierungshierarchien (von den Autoren Abstraktionsbäume genannt) genutzt, um anstelle konkreter Tupel-Identifikatoren abstraktere Werte anzugeben. Die Privacy steigt dann mit der Anzahl möglicher Anfragen, die zum gleichen Provenance-Ergebnis führen. Allerdings betonen die Autoren, dass das Berechnen einer optimalen Generalisierung ein NP-schweres Problem darstellt. Deshalb wird zusätzlich der Informationsverlust mithilfe eines heuristischen Maßes, der Entropie, bestimmt und eine Generalisierung nur dann ausgeführt, wenn der Informationsverlust geringer als maximal erlaubt ist. PITA benötigt somit drei Eingabeparameter: Anfrageergebnisse und ihre Provenance-Annotationen, eine Generalisierungshierarchie sowie einen Privacy-Grenzwert k, angelehnt an die k-Anonymität [Swe02b]. Dieser Grenzwert sagt aus, dass es nach der Anonymisierung mindestens k Anfragen geben muss, die zu den generalisierten Anfrage- und Provenance-Ergebnissen führen.

Einordnung Die *PITA*-Software ist nach bestem Wissen und Gewissen die einzige Software, welche die polynombasierte ***how***-Provenance nutzt und mit Privacy-Aspekten verbindet. Eine Anonymisierung der Provenance-Polynome kann grundsätzlich auch für diese Masterarbeit in Betracht kommen. Wir werden jedoch in Abschnitt 4.2 sehen, wieso dieser Ansatz für *ProSA* ungeeignet ist. Hinzu kommt, dass *PITA* nicht öffentlich einsehbar ist und deshalb nicht selbst evaluiert werden konnte.

3.2.3. ProSA

Nicht zuletzt beschäftigen wir uns natürlich auch mit den an der *Universität Rostock* selbst entwickelten Tools. Dabei fangen wir mit der Software *ProSA*, benannt nach dem gleichnamigen Forschungsprojekt, an. Ein Screenshot des Tools ist in Abbildung 3.4 zu sehen. Die Software wurde erstmals im Jahr 2016 im Rahmen der Lehrveranstaltung *Neueste Entwicklungen in der Informatik* von *Tanja Auge*, *Pia Wilsdorf* und *Sabrina Brossmann* entwickelt und 2017 in der Masterarbeit *„Umsetzung von Provenance-Anfragen in Big-Data-Analytics-Umgebungen“* von *Tanja Auge* erweitert [AWB16] [Aug17]. Seitdem wurde das Tool grundlegend überarbeitet und unter anderem um einen Parser, welcher SQL-Anfragen in s-t tgds umschreibt, sowie um eine Anbindung von *ChaTEAU* zur Ausführung des *CHASE*-Algorithmus ergänzt. Das Ziel von *ProSA* ist es, ausgehend von einer Datenbankanfrage und einer Datenbank eine minimale Teildatenbank zu berechnen, welche zur Nachvollziehbarkeit, Reproduktion und Rekonstruktion des Anfrageergebnisses obligatorisch ist. Dazu werden alle Tupel mit einem globalen Identifikator versehen und Provenance-Informationen über diese Tupel erfasst. Das Anfrageergebnis wird dabei nicht direkt durch das Stellen der Anfrage an die relationale Datenbank berechnet, sondern durch die integrierte Software *ChaTEAU*, welche den *CHASE*-Algorithmus auf die Datenbankinstanz anwendet. Nachdem die Anfrage ausgeführt und das Ergebnis (die Zieldatenbank) berechnet wurde, wird in einer zweiten Phase mittels invertierter s-t tgds[3] und der Provenance-Annotationen eine Teildatenbank der originalen Datenbank berechnet, welche genau die Tupel umfasst, die direkt zum Ergebnis beigetragen haben. Wir werden uns in Abschnitt 4.1 ausführlich mit dem sogenannten *ProSA*-Workflow beschäftigen und dabei einen detaillierteren Einblick in die Software erhalten.

[3] Das Implementieren dieser Invertierung ist gegenwärtig Bestandteil der Masterarbeit *„Inverse Anfragen in ProSA“* von *Dennis Spolwind*, momentan jedoch noch kein Bestandteil der Software [Spo22].

3.2.4. ChaTEAU

Im vorherigen Unterabschnitt haben wir die *ProSA*-Software betrachtet und dabei gelernt, dass diese den *CHASE*-Algorithmus nutzt. Dieser ist selbst kein Bestandteil des *ProSA*-Tools, sondern wird in einem weiteren Tool namens *ChaTEAU* („***Cha**se* for **T**ransforming, **E**volving, and **A**dapting databases and queries, **U**niversal approach“) implementiert. *ChaTEAU* wird ebenfalls an der *Universität Rostock* entwickelt und hat das Ziel, alle mit dem *CHASE*-Algorithmus abdeckbaren Anwendungsfälle in einer einzigen Anwendung zusammenzufassen. Das Tool erhält dafür als Eingabe ein Datenbankschema, eine Datenbankinstanz sowie eine Menge von Integritätsbedingungen dargestellt als egds, tgds und/oder s-t tgds (siehe Abschnitt 2.1), kodiert in Form einer einzelnen XML-Datei. Dadurch ist es *ChaTEAU* möglich, einfache SPJU-Datenbankanfragen[4] auszuführen (unter Verwendung von s-t tgds), Data Cleaning (unter Verwendung von egds) und Data Exchange (unter Verwendung von s-t tgds) zu betreiben sowie die aus der Datenbanktheorie bekannten Tableaus und Tableauanfragen (unter Verwendung von egds und tgds) darzustellen und auszuführen. Auch das Optimieren und Umschreiben von Anfragen ist möglich, wenn anstelle der Instanz eine Datenbankanfrage als Eingabeobjekt angegeben wird. Weiterhin speichert *ChaTEAU* die Ursprünge einzelner Tupel und ist somit in der Lage, Provenance-Informationen (***where***, ***why*** und ***how***) bereitzustellen. Da der *CHASE*-Algorithmus nicht in jedem Fall terminiert, wurden 2020 eine Reihe von Terminierungstests implementiert, welche die Integritätsbedingungen vor der Ausführung des Algorithmus' unter anderem auf schwache und starke Azyklizität untersuchen und eine Warnung erzeugen, falls ein potenziell unendlicher Kreislauf entdeckt wird. Details hierzu können in der Masterarbeit *„Erweiterung des CHASE-Werkzeugs ChaTEAU um ein Terminierungskriterium“* von *Andreas Görres* nachvollzogen werden [Gör20].

Die Eingabe der benötigten Daten (Instanz/Anfrage, Menge von eingebetteten Abhängigkeiten) erfolgt über ein spezielles XML-Format. Dieses definiert die Schemata aller Relationen inklusive der Zielrelation, die eingebetteten Abhängigkeiten sowie die Instanz als universelle Menge von Tupeln, die jeweils einer Relation zugeordnet sind und ihrem Schema folgen müssen. Ein Beispiel für eine solche XML-Datei ist im Anhang A zu sehen. Ein aktueller Screenshot von *ChaTEAU* in der Version 1.1 ist in Abbildung 3.5 zu finden. Wir sehen dort das Tool nach der Anwendung einer s-t tgd – äquivalent zur SQL-Anfrage `SELECT firstname, lastname FROM Students WHERE district = 'Hansaviertel'` – auf unsere STUDENTS-Relation.

[4] selection, projection, join, union

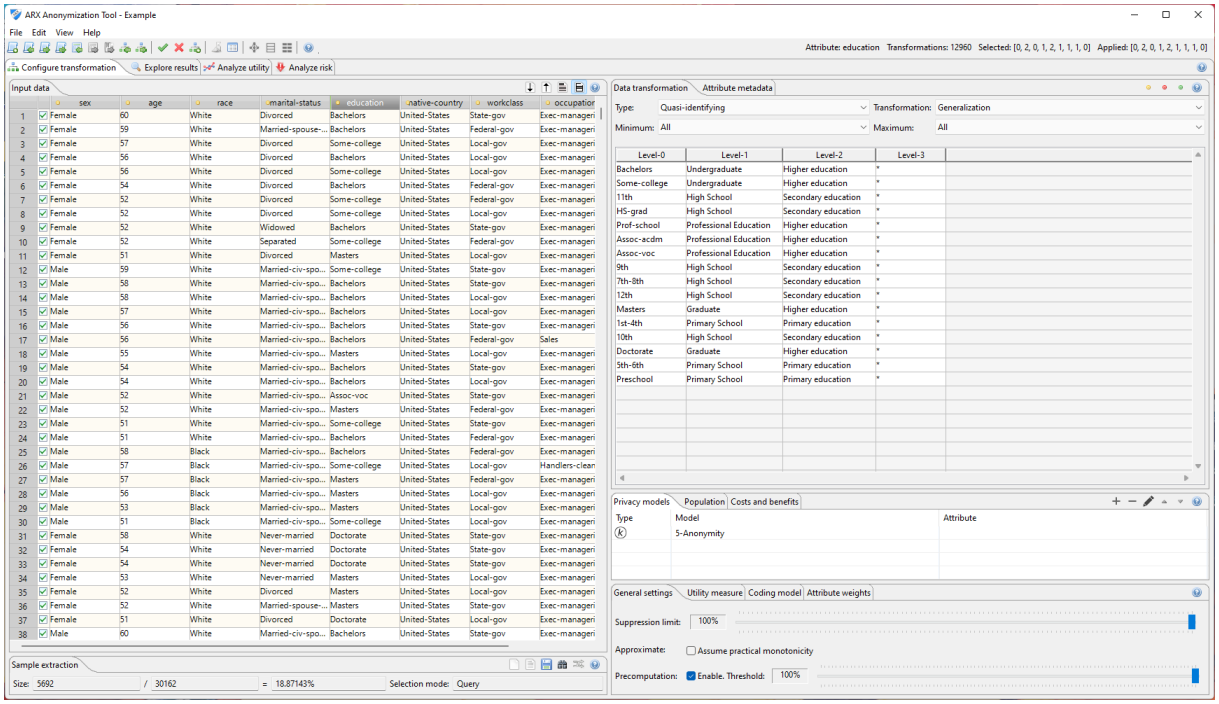

Abbildung 3.2.: *ARX* mit dem offiziellen Beispieldatensatz und einer Generalisierungshierarchie

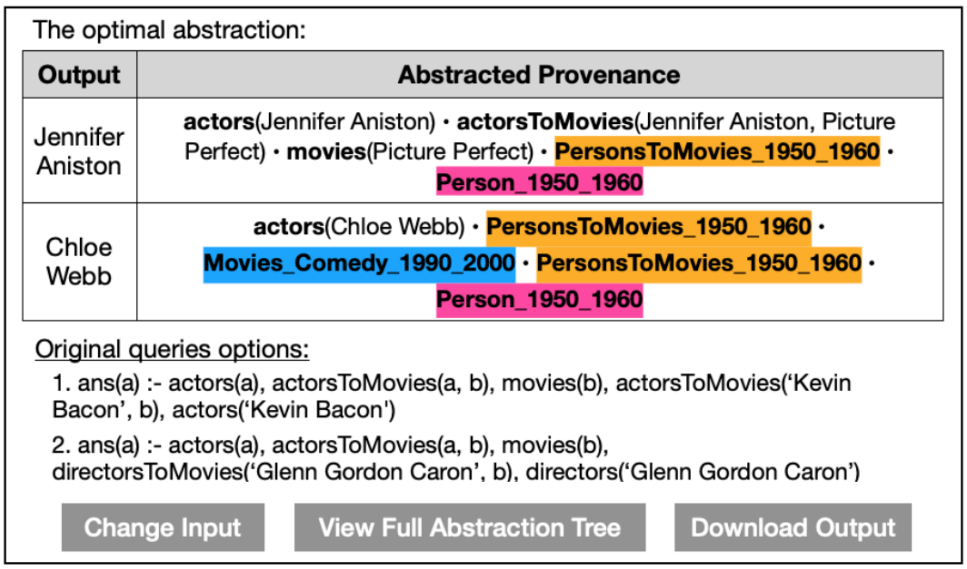

Abbildung 3.3.: Ein Screenshot der Benutzeroberfläche von PITA [DFGM21b]

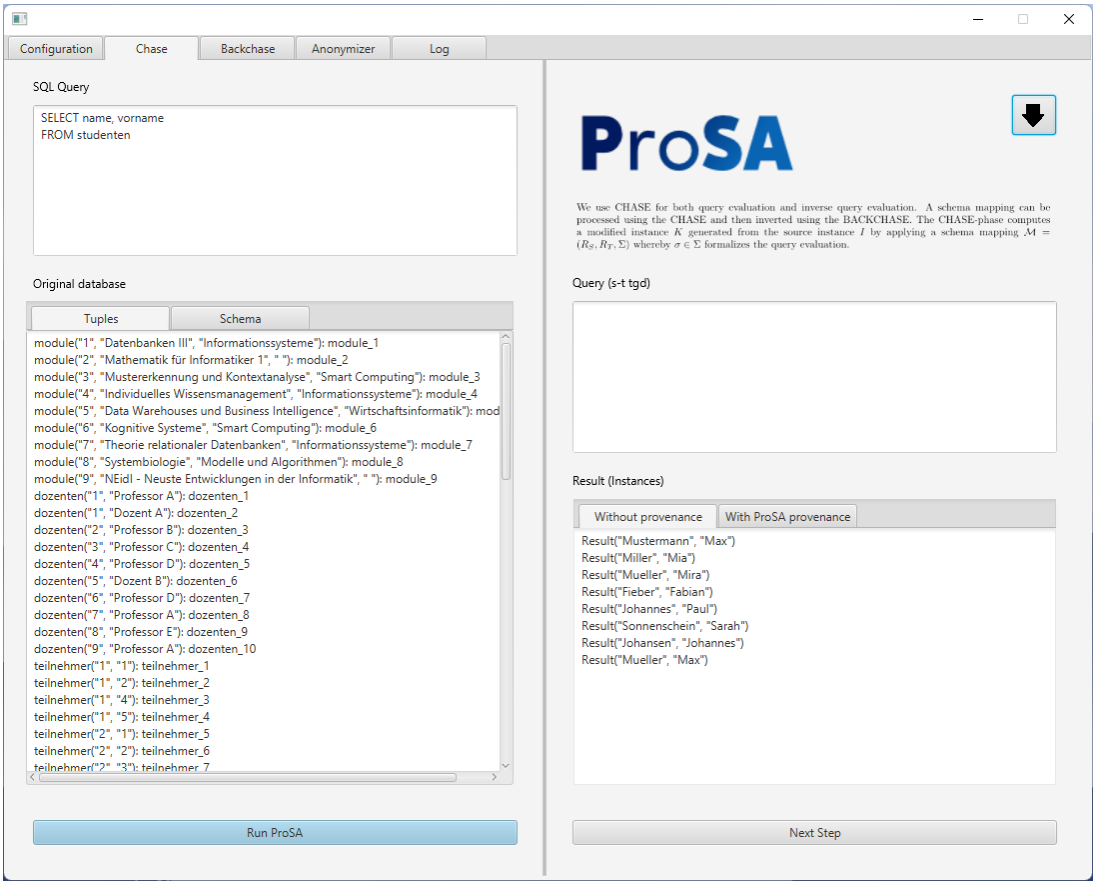

Abbildung 3.4.: Aktueller Screenshot von *ProSA*

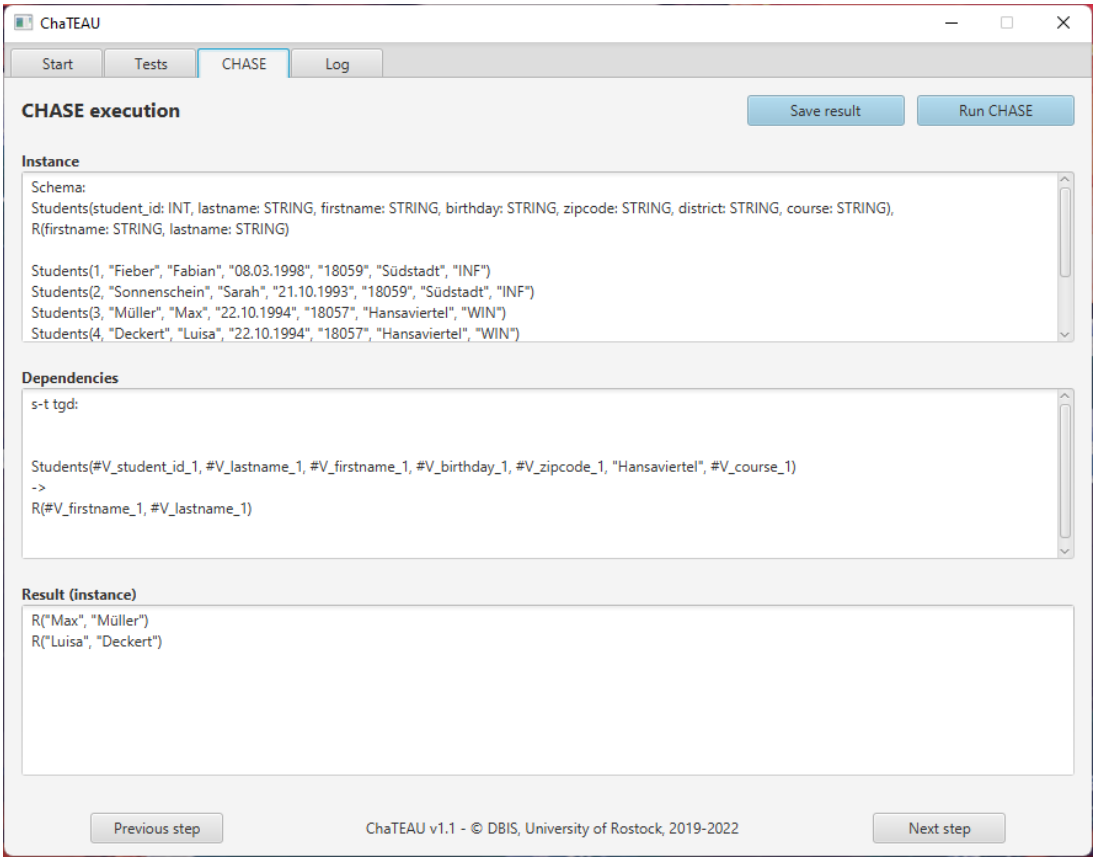

Abbildung 3.5.: *ChaTEAU* nach Einarbeitung einer s-t tgd in eine Instanz

4. Konzept

Nachdem wir uns in den vorherigen Kapiteln die Grundlagen sowie den aktuellen Stand der Forschung und Technik aneigneten, entwickeln wir nun ein Konzept für eine theoretische Integration von Privacy in *ProSA* sowie eine praktische Implementierung. Dabei erläutern wir zunächst, wie das *ProSA*-Tool allgemein funktioniert und welche Zwischenschritte durchlaufen werden, bevor wir den geeignetsten Ansatz für eine Anonymisierung diskutieren und evaluieren und zuletzt geeignete Anonymisierungmethoden und -maße sowie einen Algorithmus für die praktische Implementierung wählen.

4.1. Der ProSA-Workflow

Bereits in Unterabschnitt 3.2.3 wurden das *ProSA*-Projekt sowie die dazugehörige, gleichnamige Software vorgestellt. In diesem Unterabschnitt betrachten wir zunächst den Arbeitsablauf von *ProSA*, nachfolgend als Workflow bezeichnet, um anschließend mögliche Ansatzpunkte für Anonymisierungen zu ermitteln. Eine grafische Darstellung der einzelnen Schritte sehen wir in Abbildung 4.1.

- Einstiegspunkt von *ProSA* ist eine grafische Benutzeroberfläche (GUI; in der Abbildung 4.1 durch die (1) gekennzeichnet). Beim erstmaligen Ausführen der Software muss einmalig eine Datenbankverbindung hinterlegt werden. Anschließend wird die gesamte Instanz ausgelesen und in *ProSA* importiert.

- Im nächsten Schritt kann über die GUI eine SQL-Anfrage eingegeben werden, welche mittels eines Parsers (2) in eine s-t tgd umgeschrieben wird. Die Datenbankinstanz sowie die neu aufgestellte s-t tgd werden dann in einem Kombinationsschritt (3) um globale IDs und side tables erweitert. Der Kombinierer wird gegenwärtig in der Bachelorarbeit *„Ein Framework für ProSA“* von *Moritz Hanzig* implementiert [Han22].

- Diese gebündelten Informationen (Anfrage als s-t tgd, Instanz mit IDs, side tables) werden dann an die *ChaTEAU*-Software übergeben (4), welche für die Ausführung des *CHASE*-Algorithmus zuständig ist und eine Zieldatenbank – das Anfrageergebnis – zurückgibt. Zeitgleich ist ein Invertierer (5) dafür zuständig, die vom Parser aufgestellte s-t tgd zu invertieren und somit eine inverse Anfrage zu formulieren. Dieser ist gegenwärtig noch kein Bestandteil der Software, befindet sich aber in Entwicklung [Spo22].

- Das Anfrageergebnis sowie die invertierte s-t tgd werden dann erneut zusammengeführt (6) und für die sogenannte *BACKCHASE*-Phase[1] erneut an *ChaTEAU* übergeben (7), woraufhin erneut der *CHASE*-Algorithmus ausgeführt und das Ergebnis der invertierten Anfrage zurückgegeben wird.

- Dieses wird dann mittels Provenance-Informationen zu einer minimalen Teildatenbank vervollständigt (8), die für die Begründung des Anfrageergebnisses unentbehrlich ist.

[1] eine erneute Ausführung des *CHASE*-Algorithmus mit invertierten Anfragen

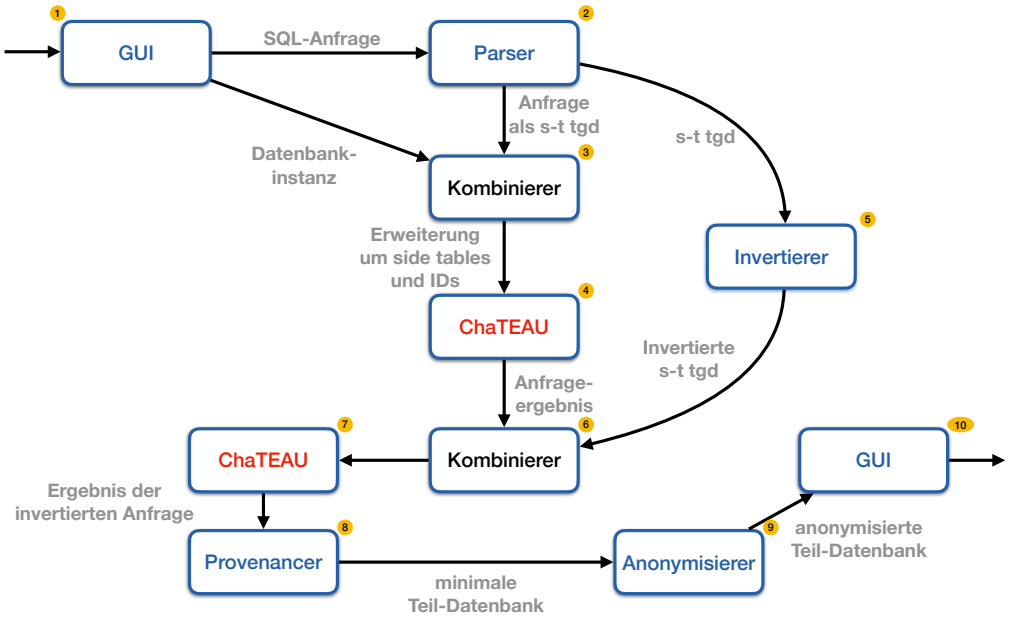

Abbildung 4.1.: Eine grafische Darstellung des ProSA-Workflows [Aug22]

- Die Aufgabe dieser Masterarbeit ist die Entwicklung eines Anonymisierers an einer geeigneten Stelle. In der Konzeptgrafik erfolgt dieser Schritt nach dem Aufstellen ebendieser minimalen Teildatenbank (9), bevor die Ergebnisse anonymisiert wieder an die GUI übergeben werden (10). Ob diese Stelle tatsächlich die geeignetste für die Anonymisierung ist und welche Alternativen in Betracht kommen, werden wir in Abschnitt 4.2 diskutieren.

4.2. Ansätze für Anonymisierungen

Zunächst werden wir uns in diesem Abschnitt mit der Frage beschäftigen, an welcher Stelle bzw. an welchen Stellen eine Anonymisierung in *ProSA* möglich und/oder sinnvoll ist. Wir starten dabei mit einigen Vorüberlegungen aus dem Jahr 2020, bevor wir die verschiedenen Möglichkeiten im Detail diskutieren und uns letztendlich für die geeignetste Stelle entscheiden werden.

```
SELECT id, AVG(grade)
FROM Students
GROUP BY id;
```

Anfrage 4.1: Wie lauten die Durchschnittsnoten der Studierenden?

Vorüberlegungen Bereits im Jahr 2020 sind gemeinsam mit *Tanja Auge* einige Vorüberlegungen in Vorbereitung des Artikels *„Privacy Aspects of Provenance Queries“* für die *Provenance Week 2020* entstanden [AS20] [ASH21a]. Diese Überlegungen umfassten vor allem die Frage, wie sich verschiedene Generalisierungen an den verschiedenen Stellen auf die Rekonstruierbarkeit mittels ***where***-, ***why***- und ***how***-Provenance auswirken. Betrachtet wurden dabei die Quelldatenbank, die Zieldatenbank sowie die mittels Provenance rekonstruierte Teildatenbank anhand einer fiktiven Relation Students(`id`, `module`, `grade`), dargestellt in Tabelle 4.1 (Seite 45). Für ***where*** und ***why*** sind Generalisierungen egal, da diese Provenance-Arten ohnehin keine konkreten Werte beinhalten. Interessant ist somit, wie sich Generalisierungen auf die ***how***-Provenance und die Rekonstruierbarkeit auswirken. Dies betrachten wir im Detail, indem wir die Anfrage

	id	module	grade
s_1	13	Mathe 1	1.0
s_2	27	Mathe 1	2.3
s_3	27	DB1	1.7
s_4	115	DB2	4.0
s_5	115	Mathe 2	5.0
s_6	115	DBAP	1.3

Tabelle 4.1.: Beispielrelation für eine Provenance-Generalisierung [AS20]

id	module	grade
13	η_1	1.0
27	η_2	2.3
27	η_3	1.7
115	η_4	4.0
115	η_5	5.0
115	η_6	1.3

Tabelle 4.2.: Rekonstruierte Relation mittels ***how***-Polynome und dem Anfrageergebnis

4.1 stellen, um die Durchschnittsnoten der Studierenden zu erhalten. Das Ergebnis sehen wir in Tabelle 4.3.

Wollen wir unsere Originalrelation rekonstruieren, benötigen wir die Polynome der ***how***-Provenance für alle drei Ergebnistupel r_1,r_2,r_3. Diese lauten

$$\frac{s_1 \odot 1.0}{s_1} \text{ für } r_1, \qquad \frac{s_2 \odot 2.3 \oplus s_3 \odot 1.7}{s_2 \oplus s_3} \text{ für } r_2, \qquad \frac{s_4 \odot 4.0 \oplus s_5 \odot 5.0 \oplus s_6 \odot 1.3}{s_4 \oplus s_5 \oplus s_6} \text{ für } r_3.$$

Daraus können wir nicht nur die Anzahl jeweils beteiligter Tupel, sondern auch ihre konkreten `grade`-Werte erschließen. Da wir die `id`-Werte aus dem Ergebnis ablesen können, können wir folgende Relation rekonstruieren.

Die η_i-Werte stellen dabei Nullwerte dar, da uns die Attributwerte für `module` fehlen. Nun wollen wir auf diesem dargestellten Weg die `grade`-Werte generalisieren, wobei jede Note durch ihre entsprechende US-amerikanische Note (1 = A, 2 = B, ...) generalisiert wird. Wir können dies sowohl in der Quellrelation, in der Zielrelation als auch in der rekonstruierten Teilrelation durchführen. Unabhängig davon, welche Stelle wir generalisieren, entsteht immer wieder dasselbe Ergebnis. Dies liegt daran, dass wir die Struktur der Polynome sowie die Tupel-Identifikatoren nicht modifiziert haben, sondern lediglich die konkreten Werte, welche im Falle der Quellrelation und dem Polynom auf den jeweils nächsten Schritt übertragen werden. Daraus folgt die Hypothese, dass die Stelle der Anonymisierung nicht entscheidend ist, zumindest für die Quelldatenbank, das Anfrageergebnis (die Zieldatenbank) sowie die rekonstruierte Teildatenbank.

Quelldatenbank Entsprechend des *ProSA*-Workflows beginnen wir mit Überlegungen zur Anonymisierung der Quelldatenbank D. Ein Vorteil besteht darin, dass diese ganz am Anfang des Workflows steht und andere, spätere Stellen nicht betreffen würde. Im weiteren Verlauf wären somit keine weiteren Schritte

	id	AVG(grade)
r_1	13	1.0
r_2	27	2.0
r_3	115	3.3

Tabelle 4.3.: Ergebnis der Anfrage 4.1

notwendig, um die Anonymität konsistent zu gewährleisten. Andererseits müssten in diesem Fall mehr Daten anonymisiert werden als möglicherweise notwendig. Um dies zumindest teilweise zu umgehen, könnte eine Anfrageanalyse erfolgen, um all jene Relationen zu bestimmen, deren Daten Teil des Ergebnisses sein werden. Letztendlich entsprächen diese Relationen der relationenorientierten ***where***-Provenance. Gleichzeitig müssten jedoch Verbundattribute von einer frühzeitigen Anonymisierung ausgenommen werden, um die Integrität der Datenbank zu erhalten. Auch alle Attribute, auf die selektiert wird, müssten davon ausgenommen werden, damit die fehlerfreie Selektion nach wie vor gewährleistet werden kann. Beispielsweise wäre eine Selektion $\sigma_{\text{zipcode}=18055}(\text{STUDENTS})$ nicht mehr möglich, wenn das `zipcode`-Attribut bereits generalisiert vorliegt. Hier kann es zu Problemen kommen, wenn beispielsweise auf einen Quasi-Identifikator oder Attribute, die Bestandteil eines solchen sind, selektiert wird. Eine Anonymisierung könnte dann an dieser Stelle womöglich gar nicht erfolgen. Um dies zu verdeutlichen, betrachten wir ein weiteres Beispiel. Uns interessieren alle Noten, die jemals im Postleitzahl-Bereich 18057 vergeben wurden (Anfrage 4.2).

```
SELECT district, grade
FROM Students NATURAL JOIN Exams
WHERE zipcode = 18057;
```

Anfrage 4.2: Welche Noten wurden an Studierende im PLZ-Bereich 18057 vergeben?

Wir wissen, dass es sich bei {`birthday`, `zipcode`, `district`} um einen Quasi-Identifikator handelt, dessen Attributwerte vor einer Veröffentlichung anonymisiert werden müssen. Dieser Schritt muss bereits vor dem Stellen der Anfrage erfolgt sein. Unsere Anfrage selektiert jedoch auf das `zipcode`-Attribut und somit auf einen Teil des Quasi-Identifikators. Läge die gesamte Relation anonymisiert vor, wäre dies nicht mehr möglich. Um dem entgegenzukommen, wäre es denkbar, nicht die Relationen direkt, sondern Kopien von ihnen zu anonymisieren und Teile der Anfrage so umzuschreiben, dass für die interne Auswertung bei Selektionen weiterhin die originalen Attributwerte verwendet würden. Dies führt zu der Frage, wann und wie oft eine solche anonymisierte Kopie erstellt werden muss. Der technische Overhead, vor jeder Anfrage eine neue Kopie anzulegen und diese hinreichend zu anonymisieren, wäre auf *ProSA* übertragen womöglich unverhältnismäßig, zumal dies für jede Datenbank separat erfolgen würde. Die Alternative wäre eine einmalige Anonymisierung der betreffenden Relationen. Dies kann jedoch nur funktionieren, wenn die Datenbank statisch ist. Jede Transaktion, bei der Tupel hinzugefügt, modifiziert oder gelöscht werden, hätte folglich eine Neuberechnung der Kopien zur Folge. Der Ansatz, die Quelldatenbank zu anonymisieren, wird deshalb aufgrund der Komplexität sowie der überwiegenden Nachteile nicht weiter verfolgt, wäre aber zumindest in der Theorie denkbar.

Anfrage Einen weiteren möglichen Ansatzpunkt für die Anonymisierung in *ProSA* stellt die Anfrage selbst dar. Hier können wir zum einen auf Generalisierungen, zum anderen auf Differential Privacy zurückgreifen, indem die Anfrage dahingehend modifiziert wird, dass sie gröbere bzw. rauschende Ergebnisse liefert. Als Beispiel betrachten wir die Anfrage 4.3.

```
SELECT birthday, zipcode, district
FROM Students NATURAL JOIN Exams
WHERE grade = 3.3;
```

Anfrage 4.3: Wie lauten Geburtstag, PLZ und Stadtteil aller Studierenden, die eine 3.3 erhalten haben?

Der kardinale Wert 3.3 kann ausgeweitet werden, um nicht direkt auf 3.3 zu selektieren, sondern auf ein Intervall um diesen Wert herum, zum Beispiel [3.0, 3.7]. Auf diese Weise entsteht ein verrauschtes Ergebnis. Dies würde jedoch das Anfrageergebnis um Tupel erweitern, die kein Bestandteil des eigentlichen

Ergebnisses sind. Auch durch die Provenance wäre nicht mehr nachvollziehbar, welche Tupel tatsächlich konserviert werden müssen und welche nicht. Dem könnte zwar entgegengewirkt werden, beispielsweise durch das Durchlaufen zweier paralleler Auswertungen – einerseits mit den originalen Werten und andererseits inklusive der „erweiterten“ Werte – und der Berechnung der Differenz beider Ergebnisse, eine solche Methode wäre jedoch kontraproduktiv, da sie die Anonymisierung ad absurdum führen würde. Differential Privacy kann hier ähnlich angewandt werden. Einerseits kann ein ähnlicher Ansatz genutzt werden, um mittels einer Anfragefunktion den konkreten Anfragewert zu verrauschen bzw. auf ein (zufällig bestimmtes) Intervall zu erweitern. Andererseits bestehen dabei dieselben Probleme im Zusammenspiel mit der Provenance.

Zudem stützten sich diese Überlegungen auf kardinale, numerische Werte, die automatisiert und semantisch sinnvoll durch Intervalle ersetzt werden können. Für numerische (ordinale) und nicht-numerische Attribute, die sich nicht automatisiert, aber dennoch semantisch sinnvoll generalisieren lassen, müssen Generalisierungshierarchien auf Datenbankebene hinterlegt sein, um konkrete Werte durch ihre übergeordneten, abstrakteren Werte ersetzen zu können. Für Attribute, die sich nicht semantisch sinnvoll generalisieren lassen, existiert keine sinnvolle Lösung. Eine Anonymisierung der Anfrage hätte also zur Folge, dass nur bestimmte Anwendungsfälle abgedeckt würden. Fraglich ist auch die technische Umsetzung in *ProSA*. Die Eingabe einer SQL-Anfrage ist nur der erste Schritt. Bereits unmittelbar danach wird diese durch einen im Jahr 2021 entwickelten Parser namens *sql2sttgd* in eine s-t tgd umgeschrieben [KRSZ21]. Zum aktuellen Zeitpunkt werden nur einfache Anfragen ohne Funktionen, Vergleiche und Aggregate unterstützt; in der Bachelorarbeit *„Erweiterung des ProSA-Parsers um Aggregatfunktionen“* von *Ivo Kavisanczki* wird der Parser zwar parallel weiterentwickelt, um ihn um ebendiese Funktionalitäten zu erweitern, eine vollständige Umsetzung in *ProSA* liegt zum Zeitpunkt dieser Masterarbeit jedoch noch nicht vor [Kav22]. Für die Darstellung von Intervallen in s-t tgds müsste allerdings auf Vergleiche zurückgegriffen werden ($\texttt{grade} \geq 3.0 \wedge \texttt{grade} \leq 3.7$). Für Generalisierungshierarchien ist der Ansatz von *Maximilian Lamster* interessant, welcher in der Masterarbeit *„Provenance-unterstütze Datenanalyse in Kombination mit intensionalen Antworten zur Steigerung der Privatsphäre“* Generalisierungshierarchien mittels Relationen darstellt [Lam21]. Aufgrund der derzeit bestehenden technischen Inkompatibilität mit *ProSA* sowie der anzunehmenden Inkompatibilität dieses Ansatzes mit den erhobenen Provenance-Annotationen wird jedoch auch dieser Ansatz nicht weiter verfolgt.

Provenance-Annotationen Als nächstes betrachten wir die Anonymisierung der Provenance-Annotationen selbst. Auch hier schauen wir uns zunächst ein Beispiel an. Wir stellen dazu Anfrage 4.4, welche uns den Nachnamen sowie die Prüfungssemester der Studentin mit der Matrikelnummer 10002 liefert, an unsere Beispieldatenbank und erhalten das in Tabelle 4.4 dargestellte Ergebnis, hier inklusive Provenance-Annotationen dargestellt. Es sei angemerkt, dass es sich bei der angegebenen ***where***-Provenance um die tupelorientierte Variante handelt. Die relationenorientierte ***where***-Provenance lautet für alle fünf Tupel {Students, Exams}.

Damit stellt die relationenorientierte ***where***-Provenance im weitesten Sinne bereits eine Form der Generalisierung dar. Genauso lässt sich die relationenorientierte ***where***-Provenance nicht weiter generalisieren, da die nächsthöhere Stufe bereits die komplette Datenbank wäre. Fraglich ist jedoch auch, wie sinnvoll eine Generalisierung überhaupt wäre. Werden als Provenance-Informationen nur die Relationennamen angegeben, so muss die gesamte Relation im Sinne guter wissenschaftlicher Arbeit aufbewahrt bzw. veröffentlicht werden, was aus Datenschutzsicht den worst case darstellt. Die ***where***-Provenance liefert vor allem relationenorientiert so wenig Informationen, dass eine Generalisierung selbst dann vernachlässigt werden könnte, wenn sie sinnvoll wäre.

Die ***why***-Provenance liefert uns Informationen über alle direkt und indirekt beteiligten Tupel der Quelldatenbank, die zu einem Tupel der Zieldatenbank beigetragen haben. Dazu zählen anders als bei der

lastname	semester	*where*	*why*	*how*
Sonnenschein	WS 20/21	s_2, e_2	$\{\{s_2, e_2\}\}$	$s_2 \cdot e_2$
Sonnenschein	SS 21	s_2, e_{15}	$\{\{s_2, e_{15}\}\}$	$s_2 \cdot e_{15}$
Sonnenschein	SS 21	s_2, e_{23}	$\{\{s_2, e_{23}\}\}$	$s_2 \cdot e_{23}$
Sonnenschein	SS 21	s_2, e_{24}	$\{\{s_2, e_{24}\}\}$	$s_2 \cdot e_{24}$
Sonnenschein	SS 21	s_2, e_{25}	$\{\{s_2, e_{25}\}\}$	$s_2 \cdot e_{25}$

Tabelle 4.4.: Ergebnis der Anfrage 4.4 inklusive ***where***-, ***why***- und ***how***-Provenance

where-Provenance auch Tupel, deren Daten selbst kein Teil des Ergebnisses sind, aber beispielsweise bei Verbunden beteiligt waren. Die ***why***-Provenance gibt dazu eine Menge von Zeugenmengen an, die sogenannte Zeugenbasis. Zeugenmengen wiederum bestehen aus einer Menge von Zeugen, welche Tupel repräsentieren. Im *ProSA*-Fall sind dies die global vergebenen Tupel-IDs. Für diese existiert wie bereits bei der ***where***-Provenance keine sinnvolle Generalisierung. Es wäre allerdings denkbar, die Tupel-IDs durch die dazugehörigen Relationennamen zu ersetzen. Aus $\{\{s_2, e_2\}\}$ würde dann beispielsweise {{Students, Exams}}. Bei umfangreicheren Zeugenbasen aus mehreren Zeugenmengen wären Mehrfachnennungen explizit zugelassen. Wir erhalten dadurch eine neue Art der Provenance, deren Informationsgehalt höher als bei der relationenorientierten ***where***-Provenance, aber geringer als bei der ***why***-Provenance ist. Die Anzahl der beteiligten Tupel je Zeugenmenge sowie die Ursprungsrelationen wären weiterhin rekonstruierbar, das Bestimmen konkreter Tupel, also der Zeugen, wäre allerdings nicht mehr möglich. Diese Generalisierung kann somit maximal eine Erweiterung der ***where***-Provenance darstellen, umfasst hinterher jedoch nicht mehr genügend Informationen zur einwandfreien Rekonstruktion der benötigten Teildatenbank und ist somit für den konkreten Anwendungsfall ungeeignet, da auch sie im worst case das Veröffentlichen gesamter Relationen zur Folge hätte.

Bei der ***how***-Provenance gibt es mehrere mögliche Ansätze. Im hier dargestellten einfachen Fall ohne Aggregationen kommt eine Generalisierung der Tupel-IDs der eben erläuterten „erweiterten ***where***-Provenance“ gleich. Dies liegt daran, dass es zwischen der ***why***- und der ***how***-Provenance ohne Aggregate eine starke Verbindung gibt: Produkte stellen die Zeugenmengen, Summen die Zeugenbasen dar. Für einfache Polynome ohne Aggregate bietet sich eine Anonymisierung somit nicht an.

```
SELECT lastname, semester
FROM Students NATURAL JOIN Exams
WHERE student_id = 10002;
```

Anfrage 4.4: Wie lauten Nachname und Prüfungssemester der Studentin mit der Matrikelnummer 10002?

Anders sieht dies bei Aggregatspolynomen aus. Als Beispiel betrachten wir eine weitere Anfrage 4.5, welche die Durchschnittsnote der Studentin mit der Matrikelnummer 10002 im Sommersemester 2021 liefert. Das Ergebnis ist 1,9; das dazugehörige Provenance-Polynom lautet

$$\frac{e_{15} \odot 1.3 \oplus e_{23} \odot 3.3 \oplus e_{24} \odot 2.0 \oplus e_{25} \odot 1.0}{e_{15} \oplus e_{23} \oplus e_{24} \oplus e_{25}}.$$

In einem solchen Polynom können wir nun theoretisch die Tupel-Identifikatoren, die konkreten Werte oder beides anonymisieren. Eine Anonymisierung der Werte führt allerdings zu denselben Problemen, die auch zwischen Data Provenance und der Differential Privacy auftreten: die Rekonstruierbarkeit wäre nicht mehr gegeben, möglicherweise noch nicht einmal die Nachvollziehbarkeit. Eine Generalisierung der Werte im Polynom müsste folglich eine Generalisierung derselben Werte in der Quelldatenbank zur Folge haben. Tatsächlich stellt das Anonymisieren der Identifikatoren hier einen ersten Ansatz dar, der jedoch in Zukunft ausführlicher zu evaluieren ist. Eine Generalisierung der Tupel-Identifikatoren in die korre-

spondierenden Relationennamen sähe dann zunächst wie folgt aus:

$$\frac{\textsc{Exams} \odot 1.3 \oplus \textsc{Exams} \odot 3.3 \oplus \textsc{Exams} \odot 2.0 \oplus \textsc{Exams} \odot 1.0}{\textsc{Exams} \oplus \textsc{Exams} \oplus \textsc{Exams} \oplus \textsc{Exams}}.$$

```
SELECT AVG(grade)
FROM Exams
WHERE student_id = 10002 AND semester = 'SS 21';
```

Anfrage 4.5: Was war die Durchschnittsnote der Studentin mit der Matrikelnummer 10002 im Sommersemester 2021?

Bei der Rekonstruktion einer Teildatenbank anhand dieses Polynoms müssen unter Umständen (weitaus) weniger Tupel aufbewahrt werden als bei der ***where***-Provenance, nämlich nur diejenigen aus der Exams-Relation, deren `grade`-Attributwert mit einem der Werte des Polynoms übereinstimmt (vorausgesetzt wir wissen, dass die Attributwerte aus dem `grade`-Attribut stammen). Allerdings würde auch diese Methode der Data Provenance mehr Daten liefern als erforderlich. Für *ProSA* kommt dieser Ansatz daher nicht in Betracht, allgemein könnte er jedoch eine Alternative zu den etablierten Provenance-Varianten darstellen. Genaue Vor- und Nachteile sowie potenzielle Anwendungsfälle müssten jedoch zunächst erforscht und evaluiert werden. Insgesamt sehen wir, dass es für eine Anonymisierung der Provenance-Annotationen zwar einige Ansätze gibt, allerdings ist die Anwendung auf *ProSA* nicht ohne einen Verlust der Rekonstruierbarkeit oder das Verfälschen von Datensätzen und Ergebnissen möglich. Auch diese Stelle scheidet somit für eine Anonymisierung aus.

Zieldatenbank Der Vorteil einer Anonymisierung der Zieldatenbank ist, dass diese vergleichsweise weit am Ende des *ProSA*-Workflows steht. Zu diesem Zeitpunkt wurde die Anfrage bereits ausgewertet und die Provenance-Annotationen wurden berechnet. Als Möglichkeiten einer Anonymisierung kommen grundsätzlich Differential Privacy sowie eine Generalisierung und Unterdrückung infrage. Allerdings bestehen mit Differential Privacy auch hier alle vorhergegangenen Probleme: es funktioniert nur für statistische Datenbanken und numerische, kardinale Werte. Für nominale oder ordinale Werte bzw. nicht-numerische Werte bestehen dieselben Probleme wie andernorts. Zudem besteht eine Inkompatibilität zwischen Differential Privacy und der Data Provenance, insbesondere der ***how***-Provenance, welche in gewissen Fällen – beispielsweise Aggregaten – konkrete Werte beinhaltet. Werden diese Polynom-Werte verrauscht, werden falsche Daten rekonstruiert, die möglicherweise nicht einmal mehr plausibel und somit erst recht nicht nachvollziehbar, reproduzierbar oder gar rekonstruierbar sind. Ein Verrauschen beider Werte – der Werte der Zieldatenbank und der Polynome – führt zwangsläufig zum selben Problem. Das Polynom wäre zwar in der Lage, das modifizierte Ergebnis zu begründen, bei der Rekonstruktion der minimalen Teildatenbank im nächsten Schritt entsteht jedoch ein falsches Ergebnis. Dies ist der Philosophie der Differential Privacy geschuldet und kann nützlich sein, für den *ProSA*-Ansatz ist diese Methode dennoch nicht geeignet. Werden hingegen die Werte der Zieldatenbank verrauscht, ist das Provenance-Polynom nicht mehr in der Lage, dieses Ergebnis zu begründen. Das Speichern zusätzlicher Informationen, welche das Rauschen nachvollziehbar machen, könnten zu den Originaldaten führen und das Konzept der Differential Privacy aufheben; eine ausführliche Darstellung dieses Problems ist in der Bachelorarbeit *„Provenance und Privacy in ProSA“* aus dem Jahr 2020 zu finden [Sch20].

Eine weitere Option ist das Generalisieren und Unterdrücken von Tupeln der Zieldatenbank. Eine solche Anonymisierung hätte keine Auswirkungen mehr auf frühere Schritte des *ProSA*-Workflows. Da auch die Provenance-Annotationen unverändert bleiben, ist die Rekonstruierbarkeit für den letzten Schritt gesichert. Generalisierte Werte würden dann – sofern möglich – originale Werte der rekonstruierten Teildatenbank ersetzen. Der Ansatz scheitert jedoch, sobald Daten in der Zieldatenbank aggregiert oder anderweitig

kombiniert vorliegen. Allgemeiner gilt dieser Ansatz nur für Tupel, die sowohl in der Quell- als auch in der Zieldatenbank (teilweise) vorhanden sind. Anonymisieren wir beispielsweise die Durchschnittsnoten aller Studierenden, haben wir an dieser Stelle zwar einen (potenziell) hinreichenden Informationsverlust, nach der Rekonstruktion der (minimalen) Teildatenbank liegen die beteiligten Tupel jedoch wieder im Original und somit nicht anonymisiert vor. Eine Anonymisierung der Zieldatenbank kann somit für einfache Anfragen ohne Aggregatfunktionen und Gruppierungen funktionieren. Genau diese einfachen SPJU-Anfragen[2] werden momentan von *ProSA* unterstützt. Da eine Erweiterung um ebendiese Aggregatfunktionen jedoch gegenwärtig in Form der Masterarbeit [Kav22] in Arbeit ist, wäre es nicht zukunftssicher, die Anonymisierung an dieser Stelle vorzunehmen, auch wenn für den aktuellen Stand nichts dagegen spricht. Eine Anonymisierung an dieser Stelle entfällt somit ebenfalls.

Rekonstruierte Teildatenbank Zu guter Letzt betrachten wir die Anonymisierung der rekonstruierten Teildatenbank als verbleibende Option. Der entscheidende Vorteil an dieser Stelle besteht darin, dass alle Berechnungen, die auf Provenance-Annotationen aufbauen, an dieser Stelle bereits erfolgt sind und die Daten nicht mehr weiterverarbeitet werden müssen: die inverse Anfrage wurde gestellt, die Teildatenbank wurde berechnet und mittels Data Provenance vervollständigt. All die Schritte, die an vorherigen Stellen zu Problemen führen können, sind bereits abgeschlossen und die Daten können problemlos anonymisiert werden, ohne dass es zu einem Verlust der Rekonstruierbarkeit kommt. Auch der technische Overhead ist entscheidend gering. Zwar ist es, wie zuvor beschrieben, auch möglich, die Quelldatenbank problemlos zu anonymisieren, der Aufwand ist dort allerdings wesentlich höher und es würden ggf. Daten anonymisiert werden, die gar keine Anonymisierung benötigen, weil sie nicht zum Ergebnis beitragen. Bei der rekonstruierten Teildatenbank steht die Menge der obligatorischen Tupel hingegen fest, sodass eine Anonymisierung keine unbeteiligten Daten umfasst. Zusätzlich erhalten wir durch diesen Ansatz eine einfache Möglichkeit, die rekonstruierte Teildatenbank auch ohne Anonymisierung zu bewahren, beispielsweise für Vergleiche beider Datensätze, indem der Anonymisierer auf einer Kopie der Teildatenbank ausgeführt wird. Aufgrund der anzunehmenden geringen Größe dieser wäre der Overhead dabei verhältnismäßig gering. Wir wählen deshalb die rekonstruierte Teildatenbank als optimale Stelle für die Anonymisierung in *ProSA*, erkennen aber gleichzeitig an, dass auch die Quelldatenbank und im weitesten Sinne auch die Zieldatenbank für eine Anonymisierung infrage kommen.

Zwischenfazit In diesem Unterabschnitt betrachteten wir die verschiedenen Zwischenstationen, die während eines *ProSA*-Durchlaufs absolviert werden. Dazu zählen die Quelldatenbank, die Anfrage, die Ergänzung um Provenance-Annotationen, die Zieldatenbank sowie die rekonstruierte Teildatenbank. Wir sahen, dass eine Anonymisierung der Quelldatenbank theoretisch möglich, der damit verbundene Aufwand jedoch unverhältnismäßig hoch ist. Eine Anonymisierung der Anfrage verletzt in gewissen Fällen die Rekonstruierbarkeit und kann zu Problemen in weiteren Zwischenschritten des *ProSA*-Workflows führen. Das Anonymisieren der Provenance-Annotationen wäre zwar eine Möglichkeit, ist jedoch aufgrund der Natur der Annotationen wenig sinnvoll, insbesondere für die ***where***- sowie ***why***-Provenance. Die ***how***-Provenance umfasst neben der Tupel-Identifikatoren gerade bei Aggregationen auch konkrete Werte der Instanz, diese können jedoch nicht angemessen generalisiert oder anderweitig anonymisiert werden, da dies ebenfalls einen Verlust der Rekonstruierbarkeit in *ProSA* nach sich ziehen würde. Eine Anonymisierung der Zieldatenbank ist ebenfalls theoretisch möglich, kann jedoch in *ProSA* zumindest gegenwärtig nicht verarbeitet werden, da bislang keine Unterstützung für Gruppierungen und Aggregationen vorhanden ist, welche für diese Methode jedoch unter gewissen Bedingungen erforderlich wären. Wir haben uns daher für eine Anonymisierung der rekonstruierten Teildatenbank entschieden, da diese Stelle keine Nachteile mit sich bringt. Im nächsten Unterabschnitt sehen wir uns nun verfügbare Anonymisierungsmaße an und werden uns auch hier für eine Auswahl entscheiden.

[2] selection, projection, join, union

4.3. Wahl der Anonymisierungsmaße

Im vorherigen Abschnitt haben wir verschiedene Ansätze für eine Anonymisierung im *ProSA*-Workflow analysiert und uns letztendlich für die Anonymisierung der Zieldatenbank entschieden. In diesem Abschnitt beschäftigen wir uns nun mit verschiedenen Anonymisierungsmaßen und werden auch hier eine Auswahl treffen, die für *ProSA* am geeignetsten ist.

k-Anonymität und l-Diversität k-Anonymität und l-Diversität lassen sich in Kombination mit geeigneten Generalisierungshierarchien auf jede Art von Datenbanken anwenden. Die erforderlichen Schritte hin zu einer vollständigen Anonymisierung unter gegebenen Parametern sind zwar umfangreicher als bei einer Differential-Privacy-Implementierung und führen möglicherweise zu einem höheren Informationsverlust, aufgrund der universellen Einsetzbarkeit der Anonymisierungsmaße kommen sie allerdings für die praktische Implementierung in Betracht. Dabei ist die k-Anonymität der l-Diversität aufgrund der flexibleren Einsatzmöglichkeiten sowie des geringeren Informationsverlustes zu bevorzugen.

k-map Wie wir in Unterabschnitt 3.1.4 kennengelernt haben, stellt die k-map eine schärfere Erweiterung der k-Anonymität dar, in dem sie versucht, membership disclosures und unter Umständen auch attribute disclosures zu verhindern. Der Kerngedanke dabei ist es, die Grundgesamtheit aller Individuen, beispielsweise ganze Städte oder gar Länder, zu berücksichtigen, indem mindestens k Tupel der Grundgesamtheit im veröffentlichten Datensatz enthalten sein müssen. Allerdings scheitert dieser Ansatz in der Praxis oft an realen Bedingungen, da kein Wissen über solche Grundgesamtheiten vorliegt und auch nicht zur Verfügung steht. Eine Berücksichtigung der k-map als praktisches Anonymisierungsmaß für die *ProSA*-Software scheidet somit aus.

delta-presence Die δ-presence ist ein Anonymisierungsmaß, welches – vereinfacht formuliert – das Verhältnis zwischen k-Anonymität und k-map angibt und sowohl verhindert, dass ein Tupel der Grundgesamtheit gar nicht im Datensatz repräsentiert wird, als auch, dass zu viele Tupel im Datensatz repräsentiert werden und die Gefahr einer membership disclosure somit wächst. Für eine Implementierung in *ProSA* kann der δ_{max}-Wert gemäß Definition 3.3 in Betracht kommen. Es stellt sich jedoch das bereits dargelegte Problem, dass eine Implementierung der k-map in *ProSA* an realen Bedingungen scheitert und deshalb nicht in Betracht kommt. Somit kann konsequenterweise auch die δ-presence nicht umgesetzt werden. Zwar gibt es Variationen und Erweiterungen, die versuchen, dieses Problem zu lösen. Als Beispiel sei die c-confident δ-presence genannt (siehe Definition 4.1). Dabei werden mithilfe von Verteilungsfunktionen stochastische Annahmen über die Grundgesamtheit getroffen, ohne dass diese umfänglich bekannt ist. So nutzt beispielsweise die *ARX*-Software als Parameter für die δ-presence sowohl die Standard-Poissonverteilung als auch die positive Poissonverteilung.

Definition 4.1 (c-confident δ-presence, nach [NC10])**.** Gegeben seien eine öffentlich zugängliche Menge von Verteilungsfunktionen F, eine private Tabelle T, ein Konfidenzniveau $c \in [0, 1]$ und eine generalisierte Tabelle T^* auf Grundlage von T. I_t sei ein Ereignis, bei dem ein Tupel $t \in T$ sowohl im generalisierten Datensatz T^* als auch in der Grundgesamtheit P enthalten ist: $\delta_{min} \leq \mathcal{P}(t \in T \,|\, T^*) \leq \delta_{max}$. δ-presence mit $\delta = (\delta_{min}, \delta_{max})$ ist für T^* erfüllt, wenn gilt:

$$\forall t \in T : \mathcal{P}(I_t \,|\, F) \geq c.$$

□

Dennoch müssen solche Verteilungen individuell auf den jeweiligen Anwendungsfall zugeschnitten sein. Wenn vorab bekannt ist, um welche Grundgesamtheit es sich handelt, können solche Schätzungen als Möglichkeit in Betracht kommen, für den allgemeinen Fall lässt sich jedoch keine entsprechende Verteilung bestimmen. Auch die δ-presence kommt somit für *ProSA* nicht infrage.

Zwischenfazit Wie wir sehen, gibt es verschiedenste Möglichkeiten, die Anonymität eines Datensatzes vorauszusetzen und folglich auch zu messen. Viele der genannten Maße sind jedoch nicht mit der Philosophie von *ProSA* kompatibel (Differential Privacy) oder scheitern an realen Bedingungen (k-map, δ-presence). Die k-Anonymität und l-Diversität hingegen erfüllen sowohl den Anwendungsfall von *ProSA* und sind dort praktisch einsetzbar. Deshalb wählen wir an dieser Stelle ebendiese Maße für die praktische Implementierung.

4.4. Wahl der Anonymisierungsmethoden

Im vorherigen Abschnitt haben wir uns verschiedene Anonymisierungsmaße angesehen und uns letztendlich für die k-Anonymität als geeignetes Maß für *ProSA* entschieden. In diesem Abschnitt werden wir uns analog dazu einige Anonymisierungsmethoden ansehen und uns auch an dieser Stelle für geeignete Methoden entscheiden.

Differential Privacy Differential Privacy gilt inzwischen als de-facto-Standard bei der Anonymisierung von statistischen Datensätzen. Unternehmen wie *Apple Inc.*[3] nutzen Differential Privacy, um Statistiken über das Verhalten ihrer Nutzer zu erhalten, ohne dass einzelne Nutzer dabei re-identifiziert werden können. Wir erinnern uns daran, dass es drei große Angriffsvektoren gibt: die identity disclosure, die membership disclosure und die attribute disclosure. Verschiedene Anonymitätsmaße und -methoden sowie die von ihnen verhinderten Angriffe sind in Tabelle 4.5 dargestellt. Dieser Tabelle können wir entnehmen, dass Differential Privacy alle drei Angriffsszenarien verhindern kann, was ein großer Vorteil gegenüber anderen Methoden und Maßen ist. Allerdings beschränkt sich das Anwendungsgebiet oft auf statistische Datenbanken mit einer zugrundeliegenden Population. *ProSA* hingegen operiert mit jeder denkbaren Art von Datenbanken, im Beispiel dieser Masterarbeit mit einer fiktiven Datenbank einer Universität. Um alle denkbaren Anwendungsfälle und nicht nur Spezialfälle abzudecken, wird das Konzept der Differential Privacy deshalb nicht weiter verfolgt.

Generalisierungen Als nächstes schauen wir uns Generalisierungen an. Diese können, je nach Art des Attributs, unterschiedlich erfolgen. *Lamster* nennt in seiner Masterarbeit vier verschiedene Attributtypen, die in allgemeinen Relationen auftreten können [Lam21]:

1. Typ 1: numerische Attribute mit numerischer Konzepthierarchie
2. Typ 2: beliebige Attribute mit beliebiger Konzepthierarchie
3. Typ 3: beliebige Attribute ohne sinnvolle Hierarchie
4. Typ 4: Attribute ohne Generalisierung

[3] `https://www.apple.com/privacy/docs/Differential_Privacy_Overview.pdf`, zuletzt abgerufen: 01.03.2022 21:50+0100

Es ist davon auszugehen, dass im *ProSA*-Anwendungsfall alle vier Typen vertreten sein werden. Attribute vom Typ 1 lassen sich einfach generalisieren, beispielsweise zu Intervallen. Attribute vom Typ 2 lassen sich mithilfe einer vordefinierten Konzepthierarchie ebenfalls einfach generalisieren. Für Attribute vom Typ 3 gibt es ebenfalls Alternativen zu semantischen Hierarchien, beispielsweise Masking, eine Methode, bei der einige Zeichen maskiert werden. Auch diese Attribute können wir somit anonymisieren. Das einzige Problem stellen Attribute vom Typ 4 dar. Diese müssen entweder direkt maximal generalisiert werden (zu „∗“), was einen maximalen Informationsverlust bedeutet, oder sie müssen von der jeweiligen Relation ausgenommen werden. All diese Schritte sind technisch umsetzbar und zudem mit der k-Anonymität als Anonymitätsmaß kompatibel. Somit entscheiden wir uns für die Methode des Generalisierens.

Unterdrücken von Tupeln Das Unterdrücken von Tupeln geht oft mit dem Generalisieren. Alle Tupel, die sich keiner Äquivalenzklasse hinsichtlich eines Quasi-Identifikators zuordnen lassen, werden dann aus dem Datensatz entfernt. Diese Methode ist mit der k-Anonymität kompatibel und kann theoretisch auch in *ProSA* Anwendung finden. Ein optimaler und effizienter Algorithmus, der zwischen dem Generalisieren und dem Unterdrücken abwägen kann und die Option mit dem geringeren Informationsverlust wählt, ist jedoch, wie wir in Unterabschnitt 2.3.2 erfahren haben, NP-schwer und kann deshalb nicht effektiv umgesetzt werden [MW04]. Auch wenn die technischen Voraussetzungen für Unterdrückungen vorhanden sind, entscheiden wir uns dafür, diese Methode nicht zu verwenden.

Intensionale Antworten Viele existierende Ansätze aus der Forschung zu intensionalen Antworten sind interessant, da sie in der Regel auf Generalisierungshierarchien zurückgreifen. Als Beispiele seien die Arbeiten von *Svacina* und *Lamster* genannt [Sva16] [Lam21]. Letztere kombiniert die intensionale Provenance sogar mit den Privacy-Problemen, die auch in dieser Masterarbeit betrachtet werden. *ProSA* beantwortet Anfragen allerdings extensional. Ansätze, um von einem solchen Ergebnis zurück zu den beteiligten Originaldaten zu gelangen, sind zwar vorhanden, widersprechen aber dem Anwendungsfall von *ProSA*, da die entstehenden intensionalen Antworten nicht ausreichen, um ein Ergebnis nachvollziehen oder reproduzieren zu können. Das Anonymisieren dieser Originaldaten ohne Duplikateliminierung kommt der Anonymisierung der rekonstruierten Teildatenbank gleich. Die Methoden der intensionalen Antwortbestimmung, vor allem Konzepthierarchien, sind somit für unser Konzept relevant, intensionale Antworten selbst führen jedoch am Ziel von *ProSA* vorbei.

Permutationen Zu guter Letzt betrachten wir mit Permutationen eine vergleichsweise einfache Möglichkeit, die Privacy der rekonstruierten Teildatenbank zu erhöhen. Bereits in Unterabschnitt 2.3.2 sowie in der Bachelorarbeit *„Provenance und Privacy in ProSA“* haben wir gesehen, dass diese eine weitere Möglichkeit darstellen, mit verhältnismäßig geringem Aufwand Angriffe auf einen Datensatz zu verhindern, konkret die Möglichkeit eines Unsorted-Matching-Angriffs [PS17] [Sch20]. Werden mehrere Datensätze mit derselben zugrundeliegenden Quelle parallel veröffentlicht und dabei die gleiche Sortierung verwendet – bewusst mittels `ORDER BY`-Statements oder unbewusst durch Indizes und Caches des DBMS –, können diese je nach Aufbau direkt miteinander verknüpft werden, was möglicherweise Re-Identifizierungen zur Folge hat. Um dies zu verhindern, ist es ratsam, Datensätze vor der Veröffentlichung zu permutieren. Permutationen sind zwar nicht in der Lage, Privacy-Probleme alleinstehend zu lösen, sie stellen allerdings eine zusätzliche Sicherheitsmaßnahme in Kombination mit anderen Methoden dar. Wir können diesen Ansatz problemlos auf *ProSA* übertragen, indem unabhängig von anderen, geeigneten Anonymisierungsmethoden die Zeilen der rekonstruierten Teildatenbank zufällig neu geordnet werden.

	Identity disclosure	Membership disclosure	Attribute disclosure
k-**Anonymität**	✓		
k-**Map**		✓	
l-**Diversität**			✓
δ-**Presence**		✓	
Differential Privacy	✓	✓	✓

Tabelle 4.5.: Einige Anonymitätsmaße sowie -methoden und von ihnen verhinderte Angriffe

4.5. Konzept für die praktische Umsetzung

Nachdem wir in den vorherigen Abschnitten mit der rekonstruierten Teildatenbank eine geeignete Stelle für die Anonymisierung sowie mit der k-Anonymität ein geeignetes Anonymisierungsmaß und mit Generalisierungen eine geeignete Methode für eine Implementierung gewählt haben, beschäftigen wir uns in diesem Abschnitt mit einem konkreten Konzept für die Implementierung, an dessen Ende ein Algorithmus für die praktische Anonymisierung der Datensätze mittels k-Anonymität steht.

Eingabe Wir benötigen zunächst die Datensätze und die gewünschte k-Anonymität als Eingabe. Die Quasi-Identifikatoren können wir entweder manuell angeben oder sie mittels eines Distinct Ratios berechnen lassen. Wir entscheiden uns für letzteres, um die Eingabe später simpel zu halten. Neben dem Distinct Ratio als zusätzliche Eingabe benötigen wir jedoch auch die Menge der identifizierenden Attribute, damit diese ignoriert werden können. Außerdem müssen auch die Konzepthierarchien vorher bereits feststehen. Insgesamt haben wir somit fünf Eingaben: die Datensätze, die Konzepthierarchien, die identifizierenden Attribute, den Distinct Ratio und die k-Anonymität.

Quasi-Identifikatoren Die Berechnung der Quasi-Identifikatoren kann über die Potenzmenge aller nichtidentifizierenden Attribute erfolgen, da diese alle möglichen Kombinationen beinhaltet. Die leere Menge kann vernachlässigt werden. Allerdings gibt es für n Attribute somit $2^n - 1$ verschiedene Kombinationen, weshalb dieser Ansatz exponentiell ist. Für jede mögliche Kombination wird dann überprüft, ob der jeweilige Distinct Ratio des (potenziellen) Quasi-Identifikators größer oder gleich dem eingegebenen Distinct Ratio ist. Ist dies der Fall, wird er in eine Liste aller gefundenen Quasi-Identifikatoren aufgenommen. Eine Optimierung stellt das sogenannte Pruning dar. Wenn eine Attributmenge einen Quasi-Identifikator darstellt, sind auch all ihre Obermengen Quasi-Identifikatoren.

k-Anonymität Die k-Anonymität kann in drei Schritten berechnet werden. Als erstes müssen bezüglich eines Quasi-Identifikators alle verschiedenen Attributkombinationen ermittelt werden. Im zweiten Schritt wird die Anzahl ihrer Vorkommen gezählt. Zu guter Letzt wird das Minimum dieser Vorkommen bestimmt. Dieses ist das gesuchte k.

Generalisierungen Jedem Attribut muss vorher eine Generalisierungshierarchie zugeordnet werden, welche aus verschiedenen Generalisierungsschritten besteht. Wir entscheiden uns für eine globale Generalisierung einzelner Spalten. Diese ist zwar im Gegensatz zu lokalen Generalisierungen anfälliger für Informationsverluste, dafür sind lokale Generalisierungen deutlich schwieriger zu implementieren. Ohnehin stellt das Finden einer optimalen Generalisierung ein NP-schweres Problem dar [MW04]. Das Generalisieren kann schrittweise erfolgen, indem nach jedem Schritt die bestehende k-Anonymität gemessen und mit der gewünschten k-Anonymität verglichen wird. Sollte diese erreicht sein, kann der Vorgang

abgebrochen werden, ansonsten wird mit der nächsten Spalte fortgefahren. Dies wird solange wiederholt, bis entweder die gewünschte k-Anonymität erzielt wird oder der Datensatz maximal generalisiert ist.

Algorithmus Aus den oben ausgeführten Einzelschritten können wir nun einen Algorithmus entwickeln, der einen Datensatz, eine Konzepthierarchie, eine Menge von identifizierenden Attributen I, einen dr-Wert für den Distinct Ratio sowie einen k-Wert entgegennimmt und den generalisierten Datensatz zurückgibt. Dabei werden zunächst die Quasi-Identifikatoren ermittelt, indem alle denkbaren Attributkombinationen auf ihren Distinct Ratio geprüft werden. Ist dieser größer gleich dr, nehmen wir die Kombination als neuen Quasi-Identifikator in die Menge aller Quasi-Identifikatoren Q auf. Für jeden Quasi-Identifikator $q \in Q$ generalisieren wir dann so lange spaltenweise und schrittweise die Werte seiner Attribute, bis das gewünschte k erreicht wird oder keine weitere Generalisierung mehr möglich ist. So wird sichergestellt, dass der Algorithmus in jedem Fall terminiert, auch wenn die gewünschte k-Anonymität nicht erreicht wird. Eine formale Beschreibung unseres Algorithmus' ist der dargestellte Algorithmus 2. Die Eingabe der Konzepthierarchien ist hier nicht dargestellt, da diese Eingabe kein essentieller Teil der Anonymisierung ist. Stattdessen wird davon ausgegangen, dass jedes Attribut bereits über eine Konzepthierarchie verfügt.

In Zeile 1 werden die identifizierenden Attribute I sowie das sensible Attribut $last(D)$ entfernt und in Zeile 2 die Menge Q initialisiert. Die Zeilen 3 bis 5 veranschaulichen die Berechnung der Quasi-Identifikatoren mittels der Potenzmenge aller quasi-identifizierenden Attribute und den Vergleich auf den gewünschten Distinct-Ratio-Wert dr. Jeder gefundene Quasi-Identifikator wird zur Menge Q hinzugefügt. Von Zeile 6 bis 12 wird über die Quasi-Identifikatoren iteriert. Diese werden dabei als Liste behandelt, welche in Zeile 8 absteigend nach der Anzahl der verbleibenden Generalisierungsschritte je Attribut sortiert wird. Besitzt in dieser sortierten Liste das erste Attribut keinen weiteren Generalisierungsschritt mehr (und alle anderen somit ebenfalls nicht), kann für diesen Quasi-Identifikator keine weitere Generalisierung erfolgen und er wird nicht weiter behandelt (Zeile 10). Andernfalls wird D' für das Attribut mit den meisten verbleibenden Schritten global generalisiert. Dieser Vorgang wird so lange wiederholt, bis die gewünschte k-Anonymität erreicht wird (Zeile 7) oder keine Generalisierung mehr möglich ist. Am Ende wird der anonymisierte Datensatz zurückgegeben (Zeile 13).

Algorithmus 2: Der Algorithmus zur Anonymisierung

Data: Datensatz D, identifizierende Attributmenge I, Distinct Ratio dr, k-Anonymität k
Result: anonymisierter Datensatz D'

```
1  D' ← D − I − last(D);
2  Q ← {};
3  forall c = {a_1, a_2, ..., a_n} ∈ P(D') − {∅} do
4      if distinctRatio(D', c) ≥ dr then
5          Q ← Q + {c}
6  forall q = [a_1, a_2, ..., a_n] ∈ Q do
7      while kAnonymity(D', q) < k do
8          q ← orderByRemainingStepsDesc(q);
9          if remainingSteps(q(a_1)) = 0 then
10             break;
11         else
12             generalizeColumn(D', q(a_1));
13 return D'
```

In diesem Kapitel haben wir einen detaillierten Blick auf den Ablauf von *ProSA* geworfen, mögliche Ansätze für Anonymisierungen, einige Anonymitätsmaße sowie diverse Anonymisierungsmethoden evaluiert. Als geeignetste Stelle für eine Anonymisierung stellt sich die rekonstruierte Teildatenbank heraus. Eine für *ProSA* geeignete Anonymisierungsmethode ist das Generalisieren; das dazugehörige Anonymitätsmaß ist die k-Anonymität. Zu guter Letzt haben wir einen Algorithmus für die Anonymisierung entwickelt, den es nun im nächsten Kapitel praktisch zu implementieren gilt.

5. Implementierung

In diesem Kapitel geht es um die praktische Implementierung des Konzeptes, welches wir im vorherigen Kapitel erarbeitet hatten. Wir werden uns zunächst im Abschnitt 5.1 einige allgemeine Bemerkungen wie die verwendete Programmierumgebung, das Build-Management-Tool und verwendete Drittanbieter-Software ansehen. Anschließend lernen wir in Abschnitt 5.2, welche Eingabemöglichkeiten wir für die Datensätze haben und setzen dabei einen Schwerpunkt auf das XML-Format der *ChaTEAU*-Software, bevor wir in Abschnitt 5.3 den Aufbau der Hierarchiedateien kennenlernen und eine eigene Datei anlegen, mit der wir *ProSAnon* in Abschnitt 5.4 ausführen und einen vollständigen Durchlauf anhand dieses Beispiels nachvollziehen können. Zu guter Letzt sehen wir uns in Abschnitt 5.5 die wichtigsten Bestandteile der konkreten Implementierung an und ziehen in Abschnitt 5.6 ein Fazit.

5.1. Allgemeines

Eine Aufgabe dieser Masterarbeit ist die Erweiterung der *ProSA*-Software um einen Anonymisierer. Wir haben bereits in Kapitel 4 einen ausführlichen Blick auf *ProSA* und den Aufbau des Tools geworfen. Der verwendete Parser geht aus dem KSWS-Projekt *sql2sttgd* hervor; die Ausführung des *CHASE*-Algorithmus wird von der eigenständigen Software *ChaTEAU* übernommen. Es ist daher naheliegend, auch den Anonymisierer als Modul zu entwickeln, welches bis zu einem gewissen Grad auch alleinstehend funktioniert, dessen eigentlicher Anwendungsfall allerdings die *ProSA*-Software ist. Aus diesem Grund wurde das Tool *ProSAnon* („*ProSA anonymizer*“) prototypisch entwickelt. Nachfolgend werden die wichtigsten Dinge wie die verwendete Programmiersprache und 3rd-party-Dependencies aufgelistet.

- Als Programmiersprache wird Java in der Version 11.0.11 genutzt, da sowohl *ChaTEAU* als auch bestehende Teile von *ProSA* in Java implementiert wurden und eine Einbettung von *ProSAnon* an *ProSA* somit problemlos gewährleistet ist.
- Die Software ist mit dem *Adopt OpenJDK 11*[1] entwickelt und getestet worden. Für maximale Kompatibilität wird diese Version des *Java Development Kits* vorausgesetzt.
- Die Entwicklungs- und Testumgebung ist ein Windows-Laptop mit Windows 11 Pro, 8GB RAM und einem „AMD Ryzen 5 3500U“-Prozessor. Als IDE kam *Visual Studio Code*[2] in der Version 1.65.2 zum Einsatz.
- Das verwendete Build-Management-Tool ist *Apache Maven*[3] in der Version 3.6.5. Alle von Maven bekannten Befehle funktionieren innerhalb dieses Projekts. So kann beispielsweise mit `mvn clean package` eine ausführbare JAR-Datei erzeugt werden.
- Der Code der Methode `fromSet` in der Klasse `anonymizer.util.PowerSet` wurde, wie auch im Code selbst angegeben, von *StackOverflow*[4] übernommen und modifiziert.

[1] `https://adoptopenjdk.net/`, zuletzt abgerufen: 14.03.2022 19:05+0100
[2] `https://code.visualstudio.com`, zuletzt abgerufen: 14.03.2022 19:06+0100
[3] `https://maven.apache.org/`, zuletzt abgerufen: 14.03.2022 19:07+0100
[4] `https://stackoverflow.com/a/14818944`, zuletzt abgerufen 12.03.2022 17:53+0100

- Die Software nutzt mittels *Maven* zwei Dependencies von Drittanbietern. Diese sind:
  - der PostgreSQL JDBC Driver
    Version: 42.2.19
    Maven-ID: `org.postgresql`
    URL: `https://mvnrepository.com/artifact/org.postgresql/postgresql/42.2.19` sowie
  - den XML-Parser jdom2
    Version: 2.0.6
    Maven-ID: org.jdom
    URL: `https://mvnrepository.com/artifact/org.jdom/jdom2/2.0.6`.
- Der Quellcode für *ProSAnon* befindet sich im Git-Repository unter der URL `https://git.informatik.uni-rostock.de/ns382/prosanon`. Die Version zur Abgabe befindet sich im Branch `abgabe`.
- Der Quellcode von *ProSA* befindet sich im Git-Repository unter der URL `https://git.informatik.uni-rostock.de/ta093/prosa`.
- Der Quellcode von *ChaTEAU* befindet sich im Git-Repository unter der URL `https://git.informatik.uni-rostock.de/ta093/chateau`.

Damit wissen wir nun alles, was wir über die Voraussetzungen dieser Software wissen müssen. In den nächsten Abschnitten werden wir erfahren, welche Dateien wir für die Software benötigen und wie wir diese ausführen.

5.2. Eingabe der Quelldaten

Zunächst sehen wir uns die Eingabemöglichkeiten für die Quelldaten für *ProSAnon* an. Die primäre Methode ist das Verwenden einer XML-Datei im sogenannten *ChaTEAU*-Format, welches auch von *ProSA* verwendet wird. Der genaue Aufbau der Beispieldatei ist im Anhang A zu finden. Aufgrund der Komplexität dieser Dateien schauen wir uns nur die wichtigsten Bestandteile sowie den groben Aufbau an. Die Wurzel der XML-Datei wird durch das `input`-Element dargestellt. Innerhalb dieses Elements gibt es genau ein `schema`- sowie entweder ein `instance`- oder ein `query`-Element. Letzteres ist für *ProSA* irrelevant. Unter `schema` befinden sich zum einen die Relationen, repräsentiert durch `relations`, zum anderen die eingebetteten Abhängigkeiten, dargestellt durch `dependencies`. Auch die Abhängigkeiten können für dieses Projekt vernachlässigt werden. Das `relations`-Element beinhaltet eine Menge von `relation`-Elementen mit einem `name`-Tag, welche wiederum aus einer Menge von `attribute`-Elementen mit einem `name`- sowie einem `type`-Tag bestehen. Sie stellen das Datenbankschema dar und sind somit wichtig für *ProSAnon*. Das `instance`-Element besteht aus einer Menge von `atom`-Elementen mit einem `name`-Tag, der den Namen der zugehörigen Relation angibt. Sie repräsentieren die Tupel und bestehen wiederum aus einer Menge von Konstanten, Nullwerten oder Variablen, dargestellt durch `constant`-, `null`- sowie `variable`-Elemente. Letztere treten im Anwendungsfall von *ProSAnon* nicht auf. Das gesamte `instance`-Element beinhaltet somit die eigentlichen Daten und wird für dieses Projekt benötigt. Haben wir eine solche Datei angelegt oder vom *ProSA*-Parser erhalten, können wir sie in Kombination mit einer Hierarchie-Datei in *ProSAnon* verwenden. Wie genau eine solche Datei aufgebaut ist, erfahren wir im nächsten Kapitel. Zuvor wollen wir jedoch noch den Beispieldatendatz betrachten. Es handelt sich dabei um eine leicht modifizierte Version unserer Students-Relation (vgl. Tabelle 1.1). Der vollständige Datensatz ist in Tabelle 5.1 dargestellt.

`lastname` und `firstname` sind identifizierende Attribute, welche vor einer Anonymisierung somit zu entfernen sind. Die quasi-identifizierenden bzw. nicht-sensiblen Attribute stellen `birthyear`, `zipcode`

lastname	firstname	birthyear	zipcode	sex	grade
Fieber	Fabian	1998	18059	m	2.5
Sonnenschein	Sarah	1993	18059	f	1.2
Mueller	Max	1994	18057	m	2.1
Deckert	Luisa	1994	18057	f	2.2
Gebauer	Luisa	2000	18106	f	3.0
Zimmermann	Jonas	1999	18106	m	2.7
Bach	Franziska	1998	18147	f	1.1
Kemper	Moritz	1991	18055	m	1.5

Tabelle 5.1.: Beispieldatensatz für *ProSAnon*

und `sex` dar; `grade` ist unser sensibles Attribut. Unser Datensatz folgt damit dem charakteristischen Aufbau, bei dem die identifizierenden Attribute am Anfang und die quasi-identifizierenden gefolgt vom sensiblen Attribut am Ende stehen. Dieses Schema ist für *ProSAnon* auch zwingend erforderlich. Das `birthyear`-Attribut lässt sich auf verschiedene Weisen generalisieren, beispielsweise zu Intervallen oder aber in Kategorien wie „jung“ und „alt“. Das Attribut `zipcode` besitzt keine semantisch sinnvolle Konzepthierarchie, kann aber beispielsweise mittels Masking trotzdem generalisiert werden. Das `sex`-Attribut hingegen besitzt ebenfalls keine semantisch sinnvolle Hierarchie und kann höchstens zum allgemeinsten „*any*“ generalisiert werden.

Neben der Möglichkeit, Datensätze mittels *ChaTEAU*-XML-Datei einzulesen, unterstützt *ProSAnon* zusätzlich das Auslesen von Relationen aus *PostgreSQL*-Datenbanken, darunter auch Zielrelationen, also Anfrageergebnisse. Eine CLI-Schnittstelle steht dazu zwar nicht zur Verfügung, entsprechende Klassen und Methoden sind jedoch implementiert und können so von anderen Projekten genutzt werden. Dazu steht die `DatabaseReader`-Klasse zur Verfügung, welche wiederum unter anderem eine `getTable(String tableName)`- sowie eine `getResultTable(String query, String resultName)`-Methode besitzt. Erstere importiert eine komplette Relation aus der Datenbank, letztere führt erst eine Anfrage aus und importiert dann das Ergebnis als neue Relation mit `resultName` als Namen. Diese zusätzliche Implementierung erhöht die Flexibilität von *ProSAnon* bei der Wahl der Anonymisierungsstelle, sei aber nur am Rande erwähnt.

Code-Erläuterungen Die Datenstrukturen werden im *ProSAnon*-Projekt hochgradig objektorientiert repräsentiert. So wird jede Relation durch ein `Table`-Objekt dargestellt, welches neben einiger weiterer Felder wie dem Namen vor allem eine Menge von `Attribute`-Objekten und `Row`-Objekten enthält. Ein `Attribute` repräsentiert dabei ein einzelnes Attribut und besteht aus einem Namen, einem Typ und einem `GeneralizationHierarchy`-Objekt, welches eine Generalisierungshierarchie für dieses Attribut beinhaltet und im nächsten Abschnitt vorgestellt wird. Eine `Row` zuletzt besteht aus einer Menge von `Attribute`-Objekten und einer gleich großen Menge von Attributwerten, dargestellt durch eine Menge von `Object`-Objekten, um das Zusammenspiel verschiedener Datentypen je Zeile zu gewährleisten. Dies erfordert wiederum ein Casten oder Konvertieren der jeweiligen Werte während ihrer Verarbeitung (bspw. Generalisierung), was umständlich ist. Dies wird jedoch zugunsten einer simpleren und leichter verständlichen Datenstruktur und somit letztendlich für einfache, interne Schnittstellen in Kauf genommen. Alle hier erläuterten Klassen verfügen über diverse Getter- und Setter- sowie Hilfsmethoden. Genannt seien in `Row` die sogenannten „Fingerprint-Methoden“ und in `Table` vor allem das Transponieren der Relation. Beides sind Konzepte, die in Abschnitt 5.5 ausführlich erläutert werden. Zunächst wollen wir im nächsten Abschnitt allerdings lernen, wie wir Konzepthierarchien verwenden können.

Generalisierung	Syntax	Beispiel
any	`any`	`course: any;`
interval	`interval {<start>-<end> and ...}`	`interval {1-3 and 4-6};`
map	`map: {<old> to <new> and ...}`	`map: {1.0 to A and 1.3 to A};`
masking	`masking: {<chars>}`	`masking: {3};`
round	`round`	`round;`
substring	`substring {<start> to <end>}`	`substring {0 to 4};`

Tabelle 5.2.: Generalisierungen in *ProSAnon* und ihre Syntax

5.3. Aufbau der Hierarchien-Datei

Nachdem wir nun wissen, wie wir Datensätze in *ProSAnon* verwenden können, erfahren wir nun, wie wir Generalisierungshierarchien einlesen können. Die Hierarchien, welche in der Software verarbeitet werden, müssen mithilfe einer Textdatei importiert werden. Dazu wurde ein Eingabeformat entwickelt, welches alle in der Software implementierten Generalisierungen darstellen kann. In diesem werden die Generalisierungshierarchien, angeführt von dem dazugehörigen Attributnamen, zeilenweise dargestellt. Tabelle 5.2 vermittelt einen genaueren Überblick darüber. Nachfolgend wollen wir uns jedoch auf die in *ProSAnon* enthaltenen Generalisierungen fokussieren, bevor wir eine vollständige Eingabedatei für unser Beispiel erstellen.

- Die `MapGeneralization` kann jegliche Werte durch andere Werte ersetzen. Sie erfordert eine String-to-String-Abbildung (`Map<String, String>`) und kann durch Hintereinanderausführungen für mehrstufige Hierarchien genutzt werden. Außerdem nutzt die Intervall-Generalisierung, welche über keine eigene Java-Klasse verfügt, ebenfalls die `MapGeneralization`. Sie eignet sich somit für jede Art von semantischen Generalisierungen.

- Die `MaskingGeneralization` benötigt als Parameter die Anzahl der letzten Zeichen, die in dem Wert zu maskieren sind. Sie kann trivialerweise für Maskierungen genutzt werden und eignet sich für Attribute, die sich nicht anderweitig semantisch sinnvoll generalisieren lassen.

- Die `RoundGeneralization` benötigt keine zusätzlichen Parameter. Sie kann genutzt werden, um reelle Zahlen auf- bzw. abzurunden. Technisch betrachtet konvertiert sie Double- in Integer-Werte, was ebenfalls eine Generalisierung darstellt, und kann für ebendiesen Zweck genutzt werden.

- Die `SubstringGeneralization` benötigt als Parameter einen (ganzzahligen) Start- sowie Endwert. Sie kann, genau wie die `MaskingGeneralization`, für Attribute genutzt werden, die sich nicht anders generalisieren lassen. Dabei gibt sie nur einen Teil des Wertes – als Zeichenkette interpretiert – zurück.

- Die `AnyGeneralization` zu guter Letzt ersetzt jede Art von Wert durch ein „$*$“ und benötigt keine Parameter. Sie stellt die maximale Generalisierung und somit auch den kompletten Verlust der Werte dar und sollte deshalb nur verwendet werden, wenn das Attribut von vergleichsweise geringer Priorität ist.

Wir wollen aufbauend auf unsere XML-Datei im *ChaTEAU*-Format Konzepthierarchien für die Attribute `birthyear`, `zipcode` und `sex` aufstellen. Für das `birthyear`-Attribut ergänzen wir die Datei somit um „`birthyear:`“ und entscheiden uns als ersten Schritt für eine `IntervalGeneralization` in die Bereiche 1991-1994, 1995-1997 und 1998-2000. Dies drücken wir mit `interval {1991-1994 and 1995-1997 and 1998-2000};` aus. Als zweites wollen wir die Werte noch weiter zu „$<$ 1995“ und „$\geq$ 1995“ generalisieren. Dazu nutzen wir eine `MapGeneralization`, um die jeweiligen Intervalle auf die neuen Werte abzubilden. Der Ausdruck dazu lautet `map {1991-1994 to <1995 and 1995-1997 to >=1995 and 1998-2000 to >=1995};`. Da wir mindestens diese Information unbedingt erhalten wollen, verzichten

wir auf eine `AnyGeneralization` und schließen die Zeile somit ab. Bei `zipcode` wollen wir eine dreistufige `MaskingGeneralization` nutzen. Wir ergänzen auch hier zunächst den Attributnamen, gefolgt von drei Masking-Definitionen, je für das letzte, die zwei sowie die drei letzten Zeichen. Die vollständige Zeile lautet `zipcode: masking {1}, masking {2}, masking {3};`. Weitere Generalisierungen soll es auch hier nicht geben. Das `sex`-Attribut zu guter Letzt lässt sich nicht sinnvoll weiter generalisieren. Da wir entscheiden, dass es für die Auswertung nicht von allzu hoher Relevanz ist, versehen wir es mit einer `AnyGeneralization`. Die dritte und letzte Zeile unserer Datei lautet somit `sex: any;`. Der Vollständigkeit halber ist die ganze Datei hier noch einmal dargestellt.

```
birthyear: interval {1991-1994 and 1995-1997 and 1998-2000}; map {1991-1994 to <1995 and
    1995-1997 to >=1995 and 1998-2000 to >=1995};
zipcode: masking {1}; masking {2}; masking {3};
sex: any;
```

Diese Datei können wir nun gemeinsam mit unserer XML-Datei verwenden, um *ProSAnon* auszuführen. Bevor wir uns dies detailliert im nächsten Abschnitt ansehen werden, folgt auch an dieser Stelle eine Erläuterung der wichtigsten Abläufe im Code.

Code-Erläuterungen Bei der Implementierung der Konzepthierarchien wurde zum einen auf eine Modularität und Erweiterbarkeit der verschiedenen Konzepte und zum anderen auf möglichst einfache Schnittstellen „nach innen und nach außen“ geachtet. Alle Generalisierungen sind im Package `generalizations` zu finden. Jede konkrete Generalisierung ist eine Unterklasse der abstrakten Oberklasse `Generalization`. Diese stellt ein minimales Grundgerüst dar und erzwingt die Implementierung der abstrakten Methode `generalize(String value)`. Konkrete Generalisierungen werden während des Parsens der Datei als solche erstellt und zu einem Objekt der Klasse `GeneralizationHierarchy` hinzugefügt, welche danach ihrem `Attribute`-Objekt zugeordnet wird. Durch die Vererbung wird sichergestellt, dass alle Generalisierungen durch ihre jeweilige `generalize`-Methode durchgeführt werden können, unabhängig davon, wie diese konkret aufgebaut ist. Auch eine Erweiterung um alternative Generalisierungen erfordert neben minimaler Anpassungen im `HierarchyReader` lediglich das Implementieren einer solchen, von `Generalization` erbenden, Klasse.

5.4. Ausführen von ProSAnon

Um *ProSAnon* auszuführen, muss zunächst eine ausführbare *JAR*-Datei des Projektes erzeugt werden. Dies geschieht mit dem Befehl `mvn clean package`, welcher einen `target`-Ordner mit der u.a. darin enthaltenen Datei `prosanon-1.0-all.jar` erzeugt. Diese ausführbare Datei ist eine kompilierte Version der Software, welche alle benötigten Dependencies beinhaltet. Wird sie ohne Parameter ausgeführt, erscheint eine Übersicht über alle benötigten Parameter, ihre Syntax sowie ihre Bedeutung. Dies ist auch in Tabelle 5.3 zu sehen.

Parameter	Syntax	Bedeutung
source	`source=<filepath>`	Pfad zur *ChaTEAU*-XML-Datei
hierarchies	`hierarchies=<filepath>`	Pfad zur Textdatei mit den Hierarchien
ids	`ids=<int list>`	kommaseparierte Indizes der identifizierenden Attribute
dr	`dr=<float>`	Distinct Ratio zur Berechnung der Quasi-Identifikatoren
k	`k=<int>`	gewünschte k-Anonymität

Tabelle 5.3.: Parameter für *ProSAnon*

Wir wollen *ProSAnon* exemplarisch mit unserer XML-Datei (siehe Anhang A) und der Hierarchien-Datei ausführen. Die identifizierenden Attribute befinden sich in den ersten zwei Spalten und haben somit die nullbasierten Indizes 0 und 1. Die Quasi-Identifikatoren sollen mit einem Distinct Ratio von 80% berechnet werden, außerdem wollen wir einen 2-anonymen Datensatz erhalten. Führen wir den Befehl

```
java -jar prosanon-1.0-all.jar \
  source=example.xml \
  hierarchies=hierarchies.txt \
  ids=0,1 dr=0.8 k=2
```

aus, erhalten wir folgende Ausgabe, welche die Tabelle(n) vor der Anonymisierung, gefundene Quasi-Identifikatoren, erreichte Anonymisierungen und die Tabelle(n) nach der Anonymisierung umfasst.

```
Table Students:
Students(lastname, firstname, birthyear, zipcode, sex, grade):
Fieber | Fabian | 1998 | 18059 | m | 2.5
Sonnenschein | Sarah | 1993 | 18059 | f | 1.2
Mueller | Max | 1994 | 18057 | m | 2.1
Deckert | Luisa | 1994 | 18057 | f | 2.2
Gebauer | Luisa | 2000 | 18106 | f | 3.0
Zimmermann | Jonas | 1999 | 18106 | m | 2.7
Bach | Franziska | 1998 | 18147 | f | 1.1
Kemper | Moritz | 1991 | 18055 | m | 1.5

*** Found QI: [birthyear, zipcode, sex] ***
*** Achieved 2-anonymity for this QI ***

*** Found QI: [birthyear, zipcode] ***
*** Achieved 4-anonymity for this QI ***

*** Found QI: [birthyear, sex] ***
*** Achieved 2-anonymity for this QI ***

*** Found QI: [zipcode, sex] ***
*** Achieved 4-anonymity for this QI ***

Final table Students:
Students(lastname, firstname, birthyear, zipcode, sex, grade):
[Deckert, Luisa, <1995, 18***, f, 2.2]
[Zimmermann, Jonas, >=1995, 18***, m, 2.7]
[Gebauer, Luisa, >=1995, 18***, f, 3.0]
[Bach, Franziska, >=1995, 18***, f, 1.1]
[Sonnenschein, Sarah, <1995, 18***, f, 1.2]
[Mueller, Max, <1995, 18***, m, 2.1]
[Kemper, Moritz, <1995, 18***, m, 1.5]
[Fieber, Fabian, >=1995, 18***, m, 2.5]
```

Nachdem wir in vorherigen Abschnitten alle Vorbereitungen für *ProSAnon* trafen und in diesem Abschnitt einen vollständigen Durchlauf sahen, wollen wir im nächsten Abschnitt einen Blick hinter die Kulissen werfen und uns mit der konkreten Implementierung ausgewählter Operationen beschäftigen.

5.5. Implementierung des Anonymisierers

Die Hauptmethode `anonymizeTable`, die genutzt wird und gleichzeitig die Schnittstelle für *ProSA* darstellt, ist in der Klasse `Anonymization` im Package `anonymization` zu finden. Sie benötigt ein `Table`-Objekt, einen Integer-Wert für die k-Anonymität, einen Double-Wert für den Distinct Ratio sowie ein Integer-Array mit den Indizes identifizierender Attribute als Parameter. Nachfolgend ist diese Methode vollständig dargestellt, gefolgt von einer Erläuterung der wichtigsten Abschnitte.

```
1  /**
2   * Anonymizes a table.
3   *
4   * @param table         the table to anonymize
5   * @param k             desired k-anonymity
6   * @param distinctRatio minimum distinct ratio for quasi-identifiers
7   * @param ids           indices of (directly) identifying attributes
8   * @return
9   */
10 public static Table anonymizeTable(Table table, int k, double distinctRatio, int[] ids) {
11     // print original table
12     System.out.println(String.format("Table %s:", table.getName()));
13     System.out.println(table);
14
15     int columnCount = table.getAttributes().size();
16
17     // identifying part
18     Table identifyingPart = table.subtable(ids);
19
20     // get "remaining", quasi-identifying + sensitive indices
21     int[] quasiIdentifyingIndices = new int[columnCount - ids.length];
22     int index = ids.length;
23     for (int i = 0; i < quasiIdentifyingIndices.length; i++) {
24         quasiIdentifyingIndices[i] = index++;
25     }
26
27     // quasi-identifying and sensitive part
28     Table qiTable = table.subtable(quasiIdentifyingIndices);
29
30     // "main algorithm"
31     for (QuasiIdentifier qi : qiTable.findQuasiIdentifiers(columnCount - 1, distinctRatio
           , false)) {
32         System.out.println(String.format("*** Found QI: %s ***", qi.getAttributes().
               toString()));
33
34         // generalize until desired k has been reached
35         WhileLoop: while (qiTable.measureKAnonymity(qi) < k) {
36             // compare attributes by number of remaining generalization steps
37             Comparator<Attribute> compareByRemainingSteps = (Attribute a1, Attribute a2)
                   -> Integer
38                     .compare(a2.getRemainingGeneralizationSteps(), a1.
                           getRemainingGeneralizationSteps());
39
40             // order attributes by remaining steps
41             ArrayList<Attribute> orderedAttributes = qi.getAttributes();
42             Collections.sort(orderedAttributes, compareByRemainingSteps);
43
44             try {
45                 Attribute selectedAttribute = orderedAttributes.get(0);
46
47                 // quasi-identifying attributes can't be further generalized
```

```
48                  if (selectedAttribute.getRemainingGeneralizationSteps() == 0) {
49                      break WhileLoop;
50                  }
51
52                  // column generalization
53                  qiTable.generalizeColumn(selectedAttribute);
54              } catch (EndOfHierarchyException | NullPointerException ex) {
55                  // no generalization available or highest generalization reached
56              }
57          }
58
59          // output result
60          int achievedK = qiTable.measureKAnonymity(qi);
61          System.out.println(String.format("*** Achieved %d-anonymity for this QI ***",
                achievedK));
62          System.out.println();
63      }
64
65      // put table back together and shuffle it
66      Table finalTable = Table.combine(identifyingPart, qiTable);
67      finalTable.shuffleRows();
68
69      // return final table
70      return finalTable;
71  }
```

Der eigentliche Algorithmus beginnt in Zeile 18 mit dem Aufteilen der Tabelle in einen identifizierenden Teil (`identifyingPart`) und einen sonstigen Teil (qiTable). Dazu wird die `subtable(int... indices)`-Methode der `Table`-Klasse genutzt, um zwei Teiltabellen zu erhalten. Der identifizierende Teil bleibt bis zum Schluss unberührt, während der sonstige Teil anonymisiert wird. Dies geschieht für jeden Quasi-Identifikator `qi`, der für den angegebenen Distinct-Ratio-Wert in der `qiTable`-Tabelle gefunden wird. Dabei wird die `WhileLoop`-Schleife solange durchlaufen, bis das gewünschte k für `qi` erreicht wurde oder alle Generalisierungshierarchien voll ausgeschöpft wurden (Zeile 48-50). Die Attribute bzw. Spalten des Quasi-Identifikators werden dabei nach jedem Generalisierungsschritt absteigend nach der Anzahl der verbleibenden Generalisierungsschritte sortiert (Zeile 41-42). So wird verhindert, dass Attribute mit wenig Generalisierungsoptionen voreilig generalisiert werden. Ist die gewünschte k-Anonymität hergestellt oder das Ende aller Generalisierungshierarchien erreicht, wird erneut die k-Anonymität der `qiTable`-Relation mithilfe der `measureKAnonymity(QuasiIdentifier qi)` für den entsprechenden Quasi-Identifikator `qi` gemessen und ausgegeben (Zeile 60f.). Nachdem dieser Prozess für alle Quasi-Identifikatoren durchlaufen wurde, wird die finale Tabelle wieder zusammengefügt, also um die zuvor ignorierten identifizierenden Attribute ergänzt. Dies geschieht in Zeile 66 durch die `Table`-Methode `combine(Table t1, Table t2)`. Zu guter Letzt wird die finale Tabelle mittels ihrer `shuffleRows()`-Methode permutiert (Zeile 67) und anschließend in Zeile 70 zurückgegeben. Die Methode ist in Anbetracht ihrer Funktionalität vergleichsweise kurz, weil viele benötigte Funktionalitäten in anderen Klassen, beispielsweise `Table`, stecken. Die wichtigsten Zwischenschritte schauen wir uns nachfolgend im Detail an.

Fingerprints und transponierte Relationen Zuvor benötigen wir jedoch noch ein Verständnis von zwei Konzepten, die in *ProSAnon* eine große Rolle spielen. Dies sind zum einen **Fingerprints** und zum anderen **transponierte Relationen**. Wir beginnen dabei mit den Fingerabdrücken; der englische sowie der deutsche Begriff werden simultan genutzt. Jede Zeile, also jedes Objekt der `Row`-Klasse, verfügt über drei Fingerprint-Methoden zur Berechnung des gesamten, eines partiellen sowie eines selektiven Fingerprints.

- Der **gesamte Fingerabdruck** einer Zeile ist der Hashwert aller konkatenierten, semikolon-separierten Attributwerte der gesamten Zeile. Er kann mit `fingerprint()` berechnet werden und wird als Integer-Wert repräsentiert.
- Ein **partieller Fingerabdruck** einer Zeile funktioniert genau wie der gesamte Fingerabdruck, betrachtet bei der Berechnung des Hashes allerdings nur einen bestimmten, zusammenhängenden Teilbereich der jeweiligen Zeile. Er kann mit `partialFingerprint(int start, int end)` berechnet werden. Die Integer-Werte `start` und `end` geben dabei den nullbasierten Start- bzw. End-Index der Unterzeile an.
- Der **selektive Fingerabdruck** nutzt zur Berechnung des Hashwertes nur bestimmte Indizes, die nicht zwingend zusammenhängen müssen. Er kann mit `selectiveFingerprint(int... indices)` berechnet werden und benötigt eine Menge bzw. ein Array von Integer-Werten, die die Indizes repräsentieren, als Parameter.

Als Hashfunktion wird die Java-eigene `hashCode()`-Methode der `String`-Klasse verwendet. Fingerprints sind nützlich, um die Identität einer Zeile bestimmen zu können: gleiche Zeilen (oder Zeilenabschnitte) besitzen den gleichen Fingerprint. Dies ermöglicht das Erkennen von identischen Zeilen(abschnitten) sowie das Zählen dieser und wird beispielsweise für die Bestimmung der k-Anonymität verwendet.

Das zweite wichtige Konzept sind transponierte Relationen. Dabei handelt es sich um Relationen, deren Spalten und Zeilen gespiegelt werden, analog zu den aus der Mathematik bekannten transponierten Matrizen. Spalten stellen demnach die neuen Zeilen dar und umgekehrt (siehe Definition 5.1). Viele Operationen sind in *ProSAnon* zeilenweise und somit in der `Row`-Klasse implementiert, beispielsweise das Ändern eines Attributwertes mit dem Index `index` (`updateValue(int index, Object value)`) oder das Ermitteln aller verschiedenen Werte analog zum „`SELECT DISTINCT`“-Statement in SQL (`getDistinctValues()`). Solche Operationen können problemlos auch auf Spalten angewandt werden, indem die Relation zunächst transponiert, anschließend die gewünschte Operation auf den neuen Zeilen durchgeführt und die Relation letztendlich erneut transponiert wird, um das Schema wiederherzustellen.

Definition 5.1 (Transponierte Relation)**.** Sei R eine gegebene Relation. Die dazugehörige **transponierte Relation** $tr(R) = R^T$ stellt das Spiegelbild von R dar, in dem die Zeilen von R die neuen Spalten und die Spalten von R die neuen Zeilen sind. Die Operation ist invertierbar, sodass gilt: $tr(tr(R)) = R$. □

Wir werden in den nächsten Absätzen sehen, dass dieses Konzept immer dann zum Einsatz kommt, wenn wir auf Spalten arbeiten wollen, beispielsweise bei der Generalisierung eines Attributs oder dem Erzeugen von Teiltabellen (`subtable(int... indices)`). Das Muster „transponieren, operieren, transponieren“ wird unter anderem dort ausführlich genutzt.

Berechnung der Quasi-Identifikatoren Für eine Relation ohne identifizierende Attribute ist das Bestimmen ihrer Quasi-Identifikatoren als `findQuasiIdentifiers(int sensitiveIndex, double threshold, boolean usePruning)` in der `Table`-Klasse definiert. Der `sensitiveIndex`-Parameter gibt den nullbasierten Index des sensiblen Attributs in der Relation an, der `threshold`-Parameter bestimmt den Distinct Ratio und der boolesche Parameter `usePruning` legt fest, ob eine Pruning-Optimierung genutzt werden soll. Dabei werden Quasi-Identifikatoren, deren Attribute bereits Untermenge eines anderen Quasi-Identifikators sind, ignoriert. Es empfiehlt sich jedoch, diesen Wert auf `false` zu setzen, um später alle ermittelten Quasi-Identifikatoren berücksichtigen zu können. Diese Methode testet jede mögliche Kombination aller quasi-identifizierenden Attribute (ihre Potenzmenge exklusive der leeren Menge). Für jede Attributkombination wird der selektive Fingerabdruck jeder Zeile bestimmt. Der resultierende Hashwert wird gemeinsam mit einem Zähler in einer `<int, int>-Map` gespeichert. Wird derselbe Hashwert im

birthyear	zipcode
1998	18059
1993	18059
1994	18057
1994	18057
2000	18106
1999	18106
1998	18147
1991	18055

Tabelle 5.4.: Beispieldatensatz für *ProSAnon*, reduziert auf `birthday` und `zipcode`

weiteren Verlauf erneut gefunden, wird der Zähler inkrementiert. Wir betrachten zur Veranschaulichung einen Auszug unserer Beispieldatei für die Attributkombination {`birthyear`, `zipcode`}, dargestellt in Tabelle 5.4. Der (selektive) Fingerprint für die erste Zeile lautet `"1998;18059".hashCode()` = `-532288007`. Dies sei nur der Vollständigkeit halber einmalig dargestellt; aus Gründen der Lesbarkeit verwenden wir nachfolgend die Schreibweise `hash(1998, 18059)` und verzichten auf konkrete Hashwerte. Dieser Wert ist noch nicht in unserer `Map` enthalten und wird deshalb hinzugefügt. Analog geschieht dies für die Zeilen 2 und 3. Unsere `Map` sieht inzwischen wie folgt aus.

```
{
  hash(1998, 18059): 1,
  hash(1993, 18059): 1,
  hash(1994, 18057): 1
}
```

Zeile 4 der Tabelle 5.4 liefert denselben Fingerprint wie Zeile 3, da alle Attributwerte übereinstimmen. Wir erhöhen also den Zähler von `hash(1994, 18057)` um 1. Die restlichen Zeilen treten im Datensatz je einmal auf, sodass unsere `Map` am Ende folgendermaßen aussieht.

```
{
  hash(1998, 18059): 1,
  hash(1993, 18059): 1,
  hash(1994, 18057): 2,
  hash(2000, 18106): 1,
  hash(1999, 18106): 1,
  hash(1998, 18147): 1,
  hash(1991, 18055): 1
}
```

Aus der Anzahl der Elemente dieser `Map` ergibt sich der „Distinct Value“ für die jeweiligen Quasi-Identifikator-Attribute. Teilen wir diesen Wert durch die Anzahl aller Zeilen der Relationen, erhalten wir den Distinct Ratio (für unser Beispiel $\frac{7}{8} = 0.875 = 87.5\%$). Ist dieser Wert höher als der angegebene `threshold`, haben wir einen Quasi-Identifikator gefunden und speichern ihn als Objekt der Klasse `QuasiIdentifier`. Diesen Prozess wiederholen wir analog für alle anderen Attributkombinationen. Am Ende erhalten wir eine Liste von Quasi-Identifikatoren, welche wir in einem letzten Schritt absteigend nach der Anzahl der QI-Attribute sortieren, bevor wir sie zurückgeben.

Berechnung der k-Anonymität Als nächstes betrachten wir die Schritte, die zur Berechnung der k-Anonymität erforderlich sind. Allgemein kann die k-Anonymität über drei Fragen beantwortet werden:

1. Welche verschiedenen Attributkombinationen gibt es?
2. Wie oft tauchen diese Attributkombinationen jeweils in der gesamten Relation auf?
3. Wie oft taucht die seltenste Kombination auf?

Die letzte Frage liefert letztendlich unser k. Der relevante Code befindet sich erneut in der `Table`-Klasse und kann mit der überladenen Methode `measureKAnonymity(int... indices)` beziehungsweise `measureKAnonymity(QuasiIdentifier qi)` aufgerufen werden. Im ersten Fall benötigen wir die Indizes der entsprechenden Attribute, im zweiten Fall ein zuvor erstelltes `QuasiIdentifier`-Objekt, welches diese Indizes ebenfalls beinhaltet. Auch hier werden – genau wie bei den Quasi-Identifikatoren – mittels einer `<int, int>-Map` die Vorkommen der selektiven Fingerabdrücke je Zeile für die entsprechenden Attribute gezählt. Anschließend wird die kleinste Anzahl an Vorkommen eines Fingerprints zurückgegeben. In unserem Beispiel für die Attributkombination {`birthyear`, `zipcode`} wäre dies 1, da diverse Wertkombinationen nur einmal vorkommen. Die Relation ist bezüglich des Quasi-Identifikators {`birthyear`, `zipcode`} somit 1-anonym.

Generalisierungen Zu guter Letzt schauen wir uns die Implementierung der Generalisierungen und ihrer Hierarchien an. Bereits in Abschnitt 5.3 haben wir gelernt, dass es im Package `generalizations` eine Menge von konkreten Generalisierungen (`MapGeneralization`, `MaskingGeneralization`, ...) gibt, die Unterklassen der abstrakten Klasse `Generalization` darstellen. Wir verzichten deshalb auf eine erneute Erläuterung der Klassen und ihrer vererbenden Struktur. Stattdessen betrachten wir nachfolgend die überladene Methode `generalizeColumn(int index)` bzw. `generalizeColumn(Attribute attribute)` der `Table`-Klasse, welche die Generalisierung einer bestimmten Spalte – angegeben durch den Attributindex oder das Attribut selbst – vollzieht.

```
1  /**
2  * Generalizes a specific column by one step.
3  *
4  * @param index The zero-based index of the column
5  * @throws EndOfHierarchyException Signals that the attribute cannot further be
6  *                                 generalized
7  */
8  public void generalizeColumn(int index) throws EndOfHierarchyException {
9      transpose(); // column mode
10
11     // get hierarchy of attribute of interest
12     Row row = rows.get(index);
13     GeneralizationHierarchy hierarchy = attributes.get(index).getHierarchy();
14
15     if (hierarchy == null) {
16         transpose();
17         throw new NullPointerException(String.format("Attribute at index %d has no
               hierarchy.", index));
18     }
19
20     // generalize every attribute
21     int i = 0;
22     for (Object value : row.getValues()) {
23         row.updateValue(i++, hierarchy.current().generalize(value.toString()));
24     }
```

```
25
26      // set hierarchy to next step; if end has been reached, a custom exception will be
            thrown
27      try {
28          hierarchy.next();
29      } catch (EndOfHierarchyException ex) {
30          throw ex;
31      } finally {
32          transpose(); // restore row mode
33      }
34  }
```

Wir sehen, dass wir in Zeile 10 zunächst die Relation transponieren, da wir nicht zeilen-, sondern spaltenweise arbeiten wollen. Als nächstes bestimmen wir die Generalisierungshierarchie `hierarchy` des Attributs am angegebenen Index `index` (Zeile 14). Diese wurde während des Parsens der Hierarchien-Datei aufgestellt und an das jeweilige `Attribute` übergeben. Die Zeilen 16 bis 19 sind wichtig, falls das angegebene Attribut keine Generalisierungshierarchie besitzt. In diesem Fall wird die Relation zurücktransponiert und eine `NullPointerException` ausgelöst. Andernfalls fahren wir in den Zeilen 22 bis 25 mit der Generalisierung jeder einzelnen Spalte (welche im transponierten Modus die Zeilen darstellen) an der entsprechenden Stelle fort. Wir übergeben dazu den aktuellen (alten) Wert an die `generalize`-Methode der jeweiligen `Generalization` und ersetzen ihn durch den (neuen) Rückgabewert dieser Methode. Anschließend gehen wir in der Generalisierungshierarchie einen Schritt nach oben (Zeile 30). Sollte das Ende bereits erreicht sein, wird eine eigens implementierte `EndOfHierarchyException` ausgelöst (Zeile 32). In jedem Fall wird die Relation am Ende erneut transponiert (Zeile 35), um die korrekte Darstellung der Zeilen und Spalten wiederherzustellen. Wir erhalten somit am Ende unsere Ausgangsrelation, deren Spalte an der Stelle `index` bzw. für das Attribut `attribute` nun jedoch generalisiert vorliegt. Die `generalizeColumn`-Methode wird im Hauptalgorithmus so lange für jedes Attribut aufgerufen, bis die gewünschte k-Anonymität erreicht wurde oder keine weitere Generalisierung mehr möglich ist.

Bevor wir diesen Abschnitt beenden, wollen wir noch einen Blick auf **Subtables** sowie die Berechnung der l-Diversität werfen, welche in *ProSAnon* zwar implementiert ist, aber keine praktische Anwendung findet, da l-Diversität aufgrund der Natur des Maßes automatisiert schwer bis gar nicht zu erreichen ist. Auch die Software *ARX*, welche als de-facto-Standard für Anonymisierungen von Datensätzen gilt, bietet momentan keine Unterstützung für die l-Diversität an.

Berechnung der l-Diversität Zwischen der Berechnung der k-Anonymität und der Berechnung der l-Diversität bestehen große Parallelen. So werden auch hier die (partiellen) Fingerabdrücke der entsprechenden Zeilen genutzt, um folgende Fragen zu beantworten, die letztendlich zur l-Diversität führen.

1. Welche verschiedenen Attributkombinationen gibt es?
2. Welche verschiedenen sensiblen Werte sind diesen Kombinationen zugeordnet?
3. Wie hoch ist die seltenste Anzahl verschiedener Werte innerhalb eines solchen QI^*-Blocks?

Die `Table`-Methode `measureLDiversity(int sensitiveIndex, int... indices)` nimmt dazu den nullbasierten Index des sensiblen Attributs sowie die Indizes der entsprechenden Quasi-Identifikator-Attribute entgegen. Zunächst legt sie eine Teilrelation der Originalrelation an, eine sogenannte Subtable, welche nur aus den angegebenen Attributen sowie dem sensiblen Attribut besteht. Dazu wird die Originalrelation zunächst transponiert. Anschließend werden die gewünschten Zeilen (eigentlich Spalten) in ein

neues `Table`-Objekt kopiert, welches dann ebenfalls transponiert wird, damit die korrekte Zeilen-Spalten-Darstellung wieder gegeben ist. Ausgehend von dieser neuen Subtable wird für jede Zeile der partielle Fingerabdruck bestimmt und in einer `<int, Set<Object>-Map` gespeichert. Jedem Fingerprint ist somit eine Menge von Attributwerten zugeordnet. Für jede Zeile der Teilrelation wird anschließend der Wert des sensiblen Attributs zu diesem `Set` hinzugefügt, Duplikate werden dabei aufgrund der Mengensemantik automatisch eliminiert. Am Ende erhalten wir eine Abbildung von Fingerprints auf die verschiedenen Attributwerte, die in Zeilen mit ebendiesen Fingerprints auftreten. Zu guter Letzt zählen wir für jeden Fingerprint die Anzahl der Attribute im zugeordneten `Set` und bestimmen auch hier das Minimum, welches unser l darstellt.

Damit haben wir einen detaillierten Einblick in die wichtigsten Klassen, Methoden und Funktionalitäten von *ProSAnon* erhalten. Wir wollen dieses Kapitel deshalb nachfolgend mit einem Fazit und einigen Schlussbemerkungen abschließen.

5.6. Fazit und Schlussbemerkungen

In diesem Kapitel haben wir das in Kapitel 4 entwickelte Konzept praktisch in Form der Software *ProSAnon* implementiert. Diese unterstützt für eingegebene Datensätze im *ChaTEAU*-XML-Format oder von *PostgreSQL*-Datenbanken eine Anonymisierung mittels Generalisierungshierarchien als Anonymisierungsmethode und verwendet dafür die k-Anonymität als Anonymisierungsmaß. Darüber hinaus existieren erste Ansätze für das Unterdrücken von Tupeln sowie zur Berechnung der l-Diversität. Die Quasi-Identifikatoren, die in dem Datensatz vorhanden sind, werden von der Software mithilfe eines Distinct-Ratio-Parameters automatisch ermittelt und müssen somit nicht manuell eingegeben werden. Für die Generalisierungshierarchien haben wir ein eigenes Dateiformat entwickelt, welches alle in *ProSAnon* implementierten Generalisierungen unterstützt und das Ziel verfolgt, möglichst verständlich und lesbar zu sein. Der Hauptalgorithmus bietet Potenzial für zukünftige Optimierungen, beispielsweise durch das Erweitern der Attribute um eine Priorisierung oder Gewichtungen. Zudem ist ein Backtracking-Ansatz zur Minimierung des Informationsverlustes vorstellbar. Es ist jedoch aufgrund der exponentiellen Natur solcher Algorithmen anzunehmen, dass dies negative Auswirkungen auf die Laufzeit der Software hätte.

Bei der Implementierung von *ProSAnon* in Java lag ein großer Fokus auf der Modularität, Erweiterbarkeit und Nachhaltigkeit der Software. Alle Klassen und Methoden sowie ihre Parameter und Rückgabewerte sind mit *JavaDoc*-Kommentaren versehen, Variablen wurden in den allermeisten Fällen sprechend benannt und der Code folgt den Java-typischen Konventionen. Durch die umfangreiche Nutzung von objektorientierten Strukturen lassen sich einzelne Code-Elemente austauschen, ohne dass dazu tiefe Eingriffe in den gesamten Code notwendig sind. Insbesondere die Generalisierungshierarchien lassen sich vergleichsweise einfach erweitern. Andererseits können auch Teile dieses Codes in anderen, ähnlichen Projekten mit ähnlichen Datenstrukturen Anwendung finden.

Zeitgleich zur Entstehung dieser Masterarbeit arbeitet *Moritz Hanzig* in seiner Bachelorarbeit *„Ein Framework für ProSA“* daran, die einzelnen Bestandteile der *ProSA*-Software zu einem gemeinsamen Framework zu kombinieren [Han22]. Darunter fällt auch eine Erweiterung der GUI um die einzelnen Bestandteile, inklusive des Anonymisierers. Der aktuelle Stand dieser Oberfläche ist in Abbildung 5.1 zu sehen. Im Verlauf dieser Masterarbeit haben sich einige Änderungen an der geplanten Funktionsweise ergeben. Beispielsweise war für die Quasi-Identifikatoren eine manuelle Eingabe vorgesehen und auch die l-Diversität wird in der GUI noch berücksichtigt. Alle anderen implementierten Funktionalitäten, darunter die Angabe der gewünschten k-Anonymität und des Distinct Ratios, sind bereits in der GUI repräsentiert.

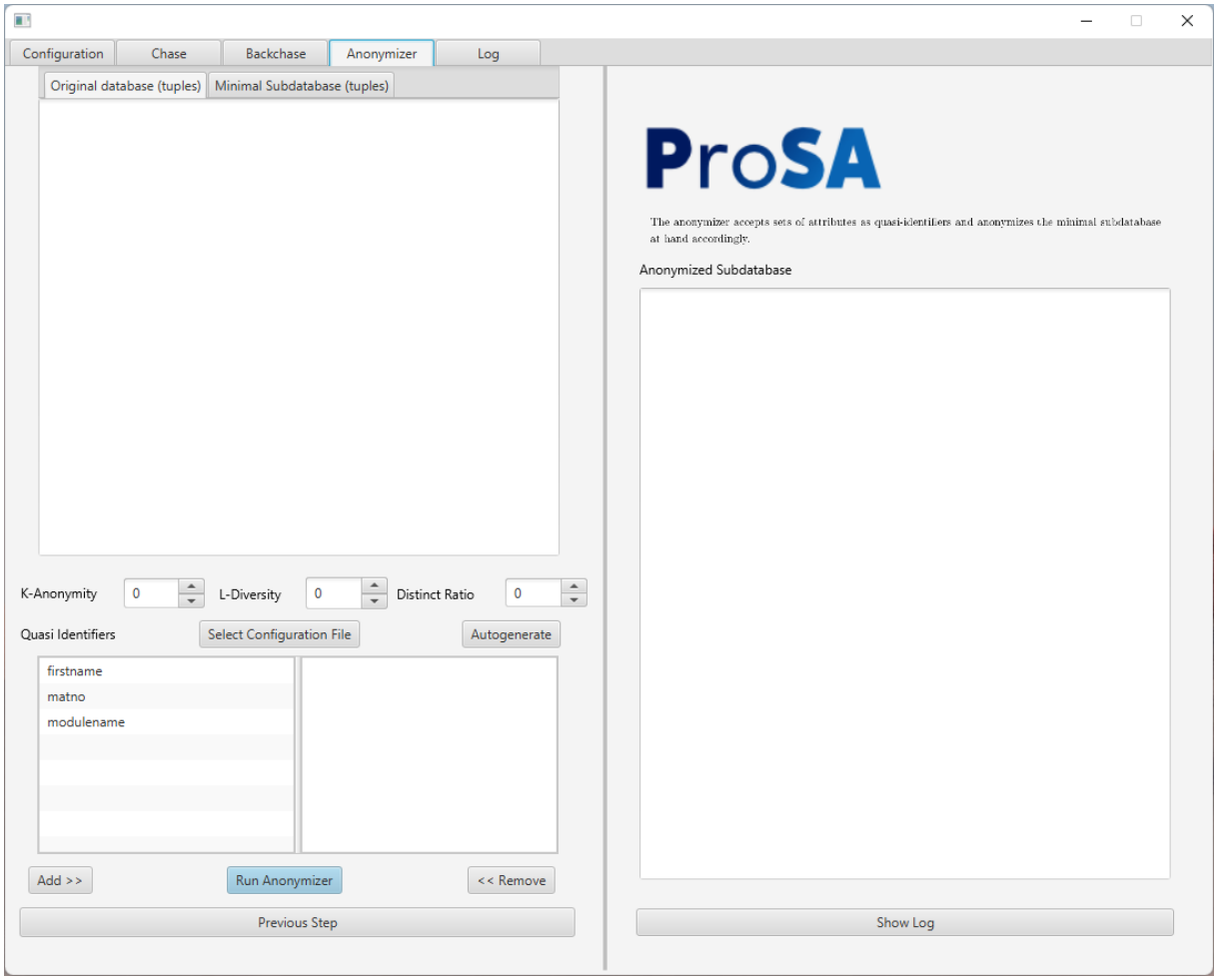

Abbildung 5.1.: Die momentane *ProSA*-GUI für den Anonymisierer

Zum Abschluss dieser Masterarbeit ziehen wir im folgenden und letzten Kapitel ein Resumé über die gestellten Aufgaben sowie die erreichten Ziele und geben einen Ausblick über Ideen für eine Weiterentwicklung von *ProSAnon* und der Anonymisierung von Data Provenance in *ProSA*.

6. Fazit und Ausblick

In diesem letzten Abschnitt betrachten wir die Ergebnisse dieser Masterarbeit und geben einen Ausblick auf zukünftige Aufgaben zur Weiterentwicklung der Theorie sowie der Software *ProSAnon*. Wir beginnen dabei zunächst mit einem Fazit.

6.1. Fazit

Das Ziel dieser Masterarbeit war die theoretische Entwicklung einer Anonymisierung von Data Provenance in *ProSA* sowie eine praktische, prototypische Implementierung dieser. Dabei haben wir infrage kommende Stellen im *ProSA*-Workflow auf ihre Verträglichkeit mit Anonymisierungen sowie die damit verbundenen Vor- und Nachteile untersucht und uns für die rekonstruierte Teildatenbank am Ende des Workflows entschieden. Mit der Generalisierung von Tupeln mittels Konzepthierarchien als Anonymisierungsmethode sowie der k-Anonymität als Anonymitätsmaß stehen uns zwei geeignete Werkzeuge zur Verfügung, um dieses Ziel zu erreichen.

In Kapitel 2 haben wir zunächst wichtige Grundlagen für diese Masterarbeit kennengelernt, darunter die Data-Provenance-Arten ***where***, ***why*** und ***how*** sowie verschiedene Möglichkeiten zur Erhöhung der Privacy einer Auswertung, darunter das Generalisieren und Unterdrücken von Tupeln, Differential Privacy, intensionale Provenance-Antworten und das Permutieren von Teilpolynomen und Zeilen. In Kapitel 3 sahen wir den aktuellen Stand der Forschung und legten dabei einen großen Fokus auf die Kombination von Provenance und Privacy sowie Anonymitätsmaße, die über die Grundlagen hinausgehen. In Kapitel 4 entwickelten wir ein Konzept für den Anonymisierer. Dabei evaluierten wir zunächst den *ProSA*-Workflow und suchten die geeignetste Stelle, die wir mit der rekonstruierten Teildatenbank fanden. Anschließend wählten wir mit den Generalisierungen eine dafür geeignete Anonymisierungsmethode sowie mit der k-Anonymität ein damit kompatibles Anonymitätsmaß. Abschließend entwickelten wir Ideen für einzelne Teilschritte wie die Berechnung der k-Anonymität und der Quasi-Identifikatoren in *ProSA* und kombinierten diese Schritte zu einem Anonymisierungs-Algorithmus, welchen es in Kapitel 5 prototypisch zu implementieren galt. Dazu wurde die Software *ProSAnon* entwickelt, die als Modul von *ProSA* aufgefasst werden kann und soll. Bei der Implementierung lag ein großer Fokus auf der einfachen Anbindung der Software mittels geeigneter Schnittstellen sowie auf einer einfachen Erweiterbarkeit. Wir haben *ProSAnon* deshalb immer wieder von außen wie von innen betrachtet und dabei Aufbau und Funktionsweise wichtiger Klassen und Methoden ausführlich beschrieben. Am Ende erhielten wir eine Software, die das ausgearbeitete Konzept prototypisch umsetzt, dabei jedoch noch Potenzial für eine zukünftige Optimierung und Weiterentwicklung bietet, wie wir im Ausblick sehen werden. Insgesamt wurde die Aufgabenstellung erfüllt und das Ziel dieser Masterarbeit somit erreicht.

6.2. Ausblick

Unsere *ProSAnon*-Software setzt die Aufgabenstellung der Masterarbeit um, stellt jedoch nur eine prototypische Implementierung dar. In zukünftigen Arbeiten kann das Projekt deshalb an verschiedenen

Stellen erweitert und optimiert werden. Auch wenn die l-Diversität kein Bestandteil des Anonymisierungsprozesses ist, existieren in der Software erste Ansätze. Mit der Methode `measureLDiversity(int sensitiveIndex, int... indices)` der `Table`-Klasse kann diese bereits für einen gegebenen Quasi-Identifikator sowie ein sensibles Attribut berechnet werden; eine automatisierte Kombination der k-Anonymität und l-Diversität stellt allerdings eine große Herausforderung dar. Auch die k-map sowie darauf aufbauend die δ-presence können implementiert werden, wenn das Problem der unbekannten Grundgesamtheit gelöst werden kann. Ein möglicher Ansatz wäre die Verwendung der gesamten Originaldatenbank für entsprechend große Datenbanken. Dazu ist jedoch zunächst zu evaluieren, ob und wenn ja, in welchen Fällen eine solche Datenbank als Grundgesamtheit dienen kann. Weiterhin existieren Methoden, um Zeilen zu unterdrücken, die die k-Anonymität nicht erfüllen. Die Kombination dieser Methode mit den Generalisierungen ist jedoch ebenfalls schwierig. Allgemein haben wir gelernt, dass das Finden einer optimalen Anonymisierung mit minimalem Informationsverlust eine Herausforderung darstellt. Unser Algorithmus kann jedoch auch an anderen Stellen erweitert werden. So ist beispielsweise eine Priorisierung oder Gewichtung der Attribute denkbar, welche alternativ zu der Sortierung nach der Anzahl verbleibender Generalisierungsschritte verwendet werden können. Denkbar ist auch die Erweiterung des Algorithmus' um Backtracking. Aufgrund der exponentiellen Natur solcher Algorithmen ist jedoch anzunehmen, dass dies negative Auswirkungen auf die Laufzeit des Programms hätte. Auch die Generalisierungen können dank des modularen Aufbaus dieser erweitert werden, beispielsweise um eine Datums-Generalisierung. Da viele Anwendungsfälle auf Abbildungen und somit auf die implementierte `MapGeneralization` reduzierbar sind – wir sahen das beispielsweise bei Intervallen –, sind an dieser Stelle viele Erweiterungen vorstellbar. Sobald *ProSA* das Invertieren und Rekonstruieren von Aggregatfunktionen unterstützt, ist auch eine erneute Untersuchung der Anonymisierung der Zieldatenbank als Alternative zur rekonstruierten Teildatenbank ratsam. Der flexible Aufbau von *ProSAnon* würde dies bereits jetzt unterstützen. Außerdem kann die Software um Unit Tests erweitert werden, um die Stabilität bei der Weiterentwicklung zu erhöhen. Zu guter Letzt sei erwähnt, dass während der Überlegungen für das Konzept Ideen für Provenance-Alternativen zu ***where***, ***why*** und ***how***entstanden. Eine zu Relationennamen generalisierte ***why***- sowie ***how***-Provenance lässt mehr Rückschlüsse auf die Strukturen der beteiligten Tupel zu als die ***where***-Provenance, besitzt aber einen deutlich geringeren Informationsgehalt als ***why*** und ***how*** ohne Generalisierung. Mögliche Vor- und Nachteile dieser Variante sowie praktische Anwendungsszenarien sind zu erforschen.

Literaturverzeichnis

[AH19] Auge, Tanja ; Heuer, Andreas: ProSA - Using the CHASE for Provenance Management. In: *ADBIS* Bd. 11695, Springer, 2019 (Lecture Notes in Computer Science), S. 357–372

[AS20] Auge, Tanja ; Scharlau, Nic: *Diskussionen zur Vorbereitung eines Artikels für die ProvenanceWeek 2020.* Private Gespräche und Notizen, 2020

[ASH21a] Auge, Tanja ; Scharlau, Nic ; Heuer, Andreas: Privacy Aspects of Provenance Queries. In: *CoRR* abs/2101.04432 (2021)

[ASH21b] Auge, Tanja ; Scharlau, Nic ; Heuer, Andreas: Provenance and Privacy in ProSA - A Guided Interview on Privacy-Aware Provenance. In: *DEXA Workshops* Bd. 1479, Springer, 2021 (Communications in Computer and Information Science), S. 52–62

[Aug17] Auge, Tanja: *Umsetzung von Provenance-Anfragen in Big-Data-Analytics-Umgebungen.* Masterarbeit, Universität Rostock, Lehrstuhl für Datenbank- und Informationssysteme, 2017

[Aug22] Auge, Tanja: *Provenance Management using Schema Mappings with Annotations (Arbeitstitel).* Laufende Dissertation, Universität Rostock, Lehrstuhl für Datenbank- und Informationssysteme, 2022

[AWB16] Auge, Tanja ; Wilsdorf, Pia ; Brossmann, Sabrina: *Neueste Entwicklungen in der Informatik: Abschlusspräsentation Cluster Provenance*, 2016. `http://eprints.dbis.informatik.uni-rostock.de/826/1/Neidi_Komplett.pdf`

[BKT01] Buneman, Peter ; Khanna, Sanjeev ; Tan, Wang C.: Why and Where: A Characterization of Data Provenance. In: *ICDT* Bd. 1973, Springer, 2001 (Lecture Notes in Computer Science), S. 316–330

[BS08] Brickell, Justin ; Shmatikov, Vitaly: The cost of privacy: destruction of data-mining utility in anonymized data publishing. In: *KDD*, ACM, 2008, S. 70–78

[CCT09] Cheney, James ; Chiticariu, Laura ; Tan, Wang C.: Provenance in Databases: Why, How, and Where. In: *Found. Trends Databases* 1 (2009), Nr. 4, S. 379–474

[CY20] Can, Özgü ; Yilmazer, Dilek: A novel approach to provenance management for privacy preservation. In: *J. Inf. Sci.* 46 (2020), Nr. 2

[DF08] Davidson, Susan B. ; Freire, Juliana: Provenance and scientific workflows: challenges and opportunities. In: *SIGMOD Conference*, ACM, 2008, S. 1345–1350

[DFGM21a] Deutch, Daniel ; Frankenthal, Ariel ; Gilad, Amir ; Moskovitch, Yuval: On Optimizing the Trade-off between Privacy and Utility in Data Provenance. In: *SIGMOD Conference*, ACM, 2021, S. 379–391

[DFGM21b] Deutch, Daniel ; Frankenthal, Ariel ; Gilad, Amir ; Moskovitch, Yuval: PITA: Privacy Through Provenance Abstraction. In: *ICDE*, IEEE, 2021, S. 2713–2716

[Dwo06] Dwork, Cynthia: Differential Privacy. In: *ICALP (2)* Bd. 4052, Springer, 2006 (Lecture Notes in Computer Science), S. 1–12

[ED08] Emam, Khaled E. ; Dankar, Fida K.: Research Paper: Protecting Privacy Using k-Anonymity. In: *J. Am. Medical Informatics Assoc.* 15 (2008), Nr. 5, S. 627–637

[GKT07] Green, Todd J. ; Karvounarakis, Gregory ; Tannen, Val: Provenance semirings. In: *PODS*, ACM, 2007, S. 31–40

[GLS14] Gkoulalas-Divanis, Aris ; Loukides, Grigorios ; Sun, Jimeng: Publishing data from electronic health records while preserving privacy: A survey of algorithms. In: *J. Biomed. Informatics* 50 (2014), S. 4–19

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

[Gol06] Golle, Philippe: Revisiting the uniqueness of simple demographics in the US population. In: *WPES*, ACM, 2006, S. 77–80

[Gör20] Görres, Andreas: *Erweiterung des CHASE-Werkzeugs ChaTEAU um ein Terminierungskriterium.* Masterarbeit, Universität Rostock, Lehrstuhl für Datenbank- und Informationssysteme, 2020

[GT17] Green, Todd J. ; Tannen, Val: The Semiring Framework for Database Provenance. In: *PODS*, ACM, 2017, S. 93–99

[Han22] Hanzig, Moritz: *Ein Framework für ProSA.* Bachelorarbeit, Universität Rostock, Lehrstuhl für Datenbank- und Informationssysteme, 2022

[HCC93] Han, Jiawei ; Cai, Yandong ; Cercone, Nick: Data-Driven Discovery of Quantitative Rules in Relational Databases. In: *IEEE Trans. Knowl. Data Eng.* 5 (1993), Nr. 1, S. 29–40

[HDB17] Herschel, Melanie ; Diestelkämper, Ralf ; Ben Lahmar, Houssem: A survey on provenance: What for? What form? What from? In: *VLDB J.* 26 (2017), Nr. 6, S. 881–906

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

[HT21] Hamadeh, Hala ; Tyagi, Akhilesh: An FPGA Implementation of Privacy Preserving Data Provenance Model Based on PUF for Secure Internet of Things. In: *SN Comput. Sci.* 2 (2021), Nr. 1, S. 65

[Kav22] Kavisanczki, Ivo: *Erweiterung des ProSA-Parsers um Aggregatfunktionen.* Bachelorarbeit, Universität Rostock, Lehrstuhl für Datenbank- und Informationssysteme, 2022

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

[Lam21] Lamster, Maximilian: *Provenance-unterstütze Datenanalyse in Kombination mit intensionalen Antworten zur Steigerung der Privatsphäre.* Masterarbeit, Universität Rostock, Lehrstuhl für Datenbank- und Informationssysteme, 2021

[MGKV06] Machanavajjhala, Ashwin ; Gehrke, Johannes ; Kifer, Daniel ; Venkitasubramaniam, Muthuramakrishnan: l-Diversity: Privacy Beyond k-Anonymity. In: *ICDE*, IEEE Computer Society, 2006, S. 24

[MW04] Meyerson, Adam ; Williams, Ryan: On the Complexity of Optimal K-Anonymity. In: *PODS*, ACM, 2004, S. 223–228

[NAC07] Nergiz, Mehmet E. ; Atzori, Maurizio ; Clifton, Chris: Hiding the presence of individuals from shared databases. In: *SIGMOD Conference*, ACM, 2007, S. 665–676

[NC10] Nergiz, Mehmet E. ; Clifton, Christopher W.: d-Presence without Complete World Knowledge. In: *IEEE Trans. Knowl. Data Eng.* 22 (2010), Nr. 6, S. 868–883

[PK15] Prasser, Fabian ; Kohlmayer, Florian: Putting Statistical Disclosure Control into Practice: The ARX Data Anonymization Tool. In: *Medical Data Privacy Handbook*. Springer, 2015, S. 111–148

[PKLK14] Prasser, Fabian ; Kohlmayer, Florian ; Lautenschläger, Ronald R. ; Kuhn, Klaus A.: ARX - A Comprehensive Tool for Anonymizing Biomedical Data. In: *AMIA*, AMIA, 2014

[PS17] Petrlic, Ronald ; Sorge, Christoph: *Datenschutz: Einführung in technischen Datenschutz, Datenschutzrecht und angewandte Kryptographie*. Wiesbaden : Springer Vieweg, 2017. – ISBN 978–3–658–16838–4

[PY99] Park, E. K. ; Yoon, Suk-Chung: An Approach to Intensional Query Answering at Multiple Abstraction Levels using Data Mining Approaches. In: *HICSS*, IEEE Computer Society, 1999

[Sam01] Samarati, Pierangela: Protecting Respondents' Identities in Microdata Release. In: *IEEE Trans. Knowl. Data Eng.* 13 (2001), Nr. 6, S. 1010–1027

[Sch20] Scharlau, Nic: *Provenance und Privacy in ProSA*. Bachelorarbeit, Universität Rostock, Lehrstuhl für Datenbank- und Informationssysteme, 2020

[Sch21] Scharlau, Nic: *Der CHASE-Algorithmus: Ein Überblick*. Seminararbeit, Universität Rostock, Lehrstuhl für Datenbank- und Informationssysteme, 2021

[Spo22] Spolwind, Dennis: *Inverse Anfragen in ProSA*. Masterarbeit, Universität Rostock, Lehrstuhl für Datenbank- und Informationssysteme, 2022

[Sva16] Svacina, Jan: *Intensional Answers for Provenance Queries in Big Data Analytics*. Bachelorarbeit, Universität Rostock, Lehrstuhl für Datenbank- und Informationssysteme, 2016

[Swe00] Sweeney, Latanya: Simple Demographics Often Identify People Uniquely, Carnegie Mellon University, 2000

[Swe02a] Sweeney, Latanya: Achieving k-Anonymity Privacy Protection Using Generalization and Suppression. In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10 (2002), Nr. 5, S. 571–588

[Swe02b] Sweeney, Latanya: k-Anonymity: A Model for Protecting Privacy. In: *Int. J. Uncertain. Fuzziness Knowl. Based Syst.* 10 (2002), Nr. 5, S. 557–570

Tabellenverzeichnis

Anfragenverzeichnis

Abbildungsverzeichnis

A. Verwendete Beispieldateien

Beispiel für eine XML-Datei im ChaTEAU-Format

```
<input>
  <schema>
    <relations>
      <relation name="Students">
        <attribute name="lastname" type="string" />
        <attribute name="firstname" type="string" />
        <attribute name="birthyear" type="int" />
        <attribute name="zipcode" type="string" />
        <attribute name="sex" type="string" />
        <attribute name="grade" type="double" />
      </relation>
    </relations>

    <dependencies />
  </schema>

  <instance>
    <atom name="Students">
      <constant name="lastname" value="Fieber" />
      <constant name="firstname" value="Fabian" />
      <constant name="birthyear" value="1998" />
      <constant name="zipcode" value="18059" />
      <constant name="sex" value="m" />
      <constant name="grade" value="2.5" />
    </atom>

    <atom name="Students">
      <constant name="lastname" value="Sonnenschein" />
      <constant name="firstname" value="Sarah" />
      <constant name="birthyear" value="1993" />
      <constant name="zipcode" value="18059" />
      <constant name="sex" value="f" />
      <constant name="grade" value="1.2" />
    </atom>

    <atom name="Students">
      <constant name="lastname" value="Mueller" />
      <constant name="firstname" value="Max" />
      <constant name="birthyear" value="1994" />
      <constant name="zipcode" value="18057" />
      <constant name="sex" value="m" />
      <constant name="grade" value="2.1" />
    </atom>

    <atom name="Students">
      <constant name="lastname" value="Deckert" />
      <constant name="firstname" value="Luisa" />
      <constant name="birthyear" value="1994" />
```

```
49        <constant name="zipcode" value="18057" />
50        <constant name="sex" value="f" />
51        <constant name="grade" value="2.2" />
52      </atom>
53
54      <atom name="Students">
55        <constant name="lastname" value="Gebauer" />
56        <constant name="firstname" value="Luisa" />
57        <constant name="birthyear" value="2000" />
58        <constant name="zipcode" value="18106" />
59        <constant name="sex" value="f" />
60        <constant name="grade" value="3.0" />
61      </atom>
62
63      <atom name="Students">
64        <constant name="lastname" value="Zimmermann" />
65        <constant name="firstname" value="Jonas" />
66        <constant name="birthyear" value="1999" />
67        <constant name="zipcode" value="18106" />
68        <constant name="sex" value="m" />
69        <constant name="grade" value="2.7" />
70      </atom>
71
72      <atom name="Students">
73        <constant name="lastname" value="Bach" />
74        <constant name="firstname" value="Franziska" />
75        <constant name="birthyear" value="1998" />
76        <constant name="zipcode" value="18147" />
77        <constant name="sex" value="f" />
78        <constant name="grade" value="1.1" />
79      </atom>
80
81      <atom name="Students">
82        <constant name="lastname" value="Kemper" />
83        <constant name="firstname" value="Moritz" />
84        <constant name="birthyear" value="1991" />
85        <constant name="zipcode" value="18055" />
86        <constant name="sex" value="m" />
87        <constant name="grade" value="1.5" />
88      </atom>
89    </instance>
90  </input>
```

Beispiel für eine Hierarchien-Datei

```
1  birthyear: interval {1991-1994 and 1995-1997 and 1998-2000}; map {1991-1994 to <1995 and
       1995-1997 to >=1995 and 1998-2000 to >=1995};
2  zipcode: masking {1}; masking {2}; masking {3};
3  sex: any;
```

B. Auszüge aus dem Quellcode

Anmerkung: *Aufgrund des hohen Umfangs des Codes wurden nachfolgend einige Passagen entfernt, dargestellt durch einen „[...]“-Kommentar. Dazu zählen beispielsweise die Package-Deklaration, Importe, Getter- und Setter-Methoden sowie überschriebene Hilfsmethoden wie* `toString`, `equals` *und* `compareTo`. Der vollständige Code ist in einem Git-Repository zu finden. Näheres ist im Anhang C beschrieben.

anonymizer.anonymization.Anonymization

```
1  // [...]
2
3  /**
4   * This class provides the main anonymization process.
5   */
6  public class Anonymization {
7
8      /**
9       * Anonymizes a table.
10      *
11      * @param table         the table to anonymize
12      * @param k             desired k-anonymity
13      * @param distinctRatio minimum distinct ratio for quasi-identifiers
14      * @param ids           indices of (directly) identifying attributes
15      * @return
16      */
17     public static Table anonymizeTable(Table table, int k, double distinctRatio, int[]
            ids) {
18         System.out.println(String.format("Table %s:", table.getName()));
19         System.out.println(table);
20
21         int columnCount = table.getAttributes().size();
22
23         // identifying part
24         Table identifyingPart = table.subtable(ids);
25
26         // get "remaining" indices
27         int[] quasiIdentifyingIndices = new int[columnCount - ids.length];
28         int index = ids.length;
29         for (int i = 0; i < quasiIdentifyingIndices.length; i++) {
30             quasiIdentifyingIndices[i] = index++;
31         }
32
33         // quasi-identifying and sensitive part
34         Table qiTable = table.subtable(quasiIdentifyingIndices);
35
36         // "main algorithm"
37         for (QuasiIdentifier qi : qiTable.findQuasiIdentifiers(columnCount - 1,
               distinctRatio, false)) {
38             System.out.println(String.format("*** Found QI: %s ***", qi.getAttributes().
                   toString()));
```

```
39
40              // generalize until desired k has been reached
41              WhileLoop: while (qiTable.measureKAnonymity(qi) < k) {
42                  // compare attributes by number of remaining generalization steps
43                  Comparator<Attribute> compareByRemainingSteps = (Attribute a1, Attribute
                        a2) -> Integer
44                            .compare(a2.getRemainingGeneralizationSteps(), a1.
                                  getRemainingGeneralizationSteps());
45
46                  // order attributes by remaining steps
47                  ArrayList<Attribute> orderedAttributes = qi.getAttributes();
48                  Collections.sort(orderedAttributes, compareByRemainingSteps);
49
50                  try {
51                      Attribute selectedAttribute = orderedAttributes.get(0);
52
53                      if (selectedAttribute.getRemainingGeneralizationSteps() == 0) {
54                          // quasi-identifying attributes can't be further generalized
55                          break WhileLoop;
56                      }
57
58                      // column generalization
59                      qiTable.generalizeColumn(selectedAttribute);
60                  } catch (EndOfHierarchyException | NullPointerException ex) {
61                      // no generalization available or highest generalization reached
62                  }
63              }
64
65              // output result
66              int achievedK = qiTable.measureKAnonymity(qi);
67              System.out.println(String.format("*** Achieved %d-anonymity for this QI ***",
                      achievedK));
68              System.out.println();
69          }
70
71          // put table back together and shuffle it
72          Table finalTable = Table.combine(identifyingPart, qiTable);
73          finalTable.shuffleRows();
74
75          // return final table
76          return finalTable;
77      }
78
79  }
```

anonymizer.structures.Table

```
1   // [...]
2
3   /**
4    * Class that represents a table/relation.
5    */
6   public class Table {
7
8       private String tableName;
9       private ArrayList<Attribute> attributes;
10      private ArrayList<Row> rows;
11      private boolean isTransposed;
```

```
12      private TableType tableType;
13
14      /**
15       * Constructor for empty tables (tables that have a name but no attributes yet).
16       *
17       * @param name The table name
18       */
19      public Table(String name) {
20          this.tableName = name;
21          this.attributes = new ArrayList<>();
22          this.rows = new ArrayList<>();
23          this.isTransposed = false;
24          this.tableType = TableType.TARGET;
25      }
26
27      /**
28       * Constructor for tables with a name and list of attributes.
29       *
30       * @param name       The table name
31       * @param attributes The table's attributes, as {@link ArrayList}
32       */
33      public Table(String name, ArrayList<Attribute> attributes) {
34          this(name);
35          this.attributes = attributes;
36      }
37
38      // [...]
39
40      /**
41       * Switches the "transposed" status. Use with caution, it might break the
42       * semantics of transpositions if applied wrong.
43       */
44      private void toggleTransposedState() {
45          isTransposed = !isTransposed;
46      }
47
48      /**
49       * Transposes the table, which means that rows become columns and vice versa. In
50       * other words, the table is handled as a matrix M and transformed into M^T.
51       */
52      public void transpose() {
53          // switch status
54          toggleTransposedState();
55
56          ArrayList<Row> newRows = new ArrayList<>();
57
58          // transpose table by treating it like a matrix
59          for (int i = 0; i < getRow(0).size(); i++) {
60              Row newRow = new Row(this);
61              for (Row row : rows) {
62                  newRow.addValue(row.getValue(i));
63              }
64              newRows.add(newRow);
65          }
66
67          // replace rows with transposed rows
68          rows = newRows;
69      }
70
71      /**
72       * Returns a new table that contains only selected columns.
```

```
73          *
74          * @param indices The columns the new table shall contain, as zero-based indices
75          * @return A subtable of the current table, including only selected columns
76          */
77         public Table subtable(int... indices) {
78             Table subTable = new Table(tableName);
79
80             // transpose table, then select rows (actually columns) to keep
81             transpose();
82
83             for (int index : indices) {
84                 subTable.addAttribute(attributes.get(index));
85                 subTable.addRow(rows.get(index));
86             }
87
88             // transpose original table back
89             transpose();
90
91             // transpose new subtable; set transposed to false
92             subTable.transpose();
93             subTable.toggleTransposedState();
94
95             return subTable;
96         }
97
98         /**
99          * Measures k-anonymity for given columns (as zero-based indices).
100         *
101         * @param indices The zero-based indices of rows in interest
102         * @return The k of k-anonymity
103         */
104        public int measureKAnonymity(int... indices) {
105            // store occurrences of unique rows
106            LinkedHashMap<Integer, Integer> occurrences = new LinkedHashMap<>();
107
108            // measure occurrences. Unique rows are rows that have every selected attribute
109            // in common, which also makes their selective fingerprint identical.
110            for (Row row : rows) {
111                int fingerprint = row.selectiveFingerprint(indices);
112
113                // init
114                occurrences.putIfAbsent(fingerprint, 0);
115
116                occurrences.put(fingerprint, occurrences.get(fingerprint) + 1);
117            }
118
119            // get minimum of unique occurrences
120            int min = Collections.min(occurrences.values());
121            return min;
122        }
123
124        /**
125         * Measures k-anonymity for a given {@link QuasiIdentifier quasi-identifier}.
126         *
127         * @param quasiIdentifier The quasi-identifier of interest
128         * @return The k of k-anonymity
129         */
130        public int measureKAnonymity(QuasiIdentifier quasiIdentifier) {
131            int[] indices = quasiIdentifier.getAttributeIndices().stream().mapToInt(Integer::
                intValue).toArray();
132            return measureKAnonymity(indices);
```

```
133     }
134
135     /**
136      * Measures l-diversity for a given sensitive attribute and given QI* columns
137      * (as zero-based indices).
138      *
139      * @param sensitiveIndex The zero-based index of the sensitive attribute
140      * @param indices        The zero-based indices of the insensitive attributes
141      * @return The l of l-diversity
142      */
143     public int measureLDiversity(int sensitiveIndex, int... indices) {
144         // build array of relevant indices
145         int[] relevantIndices = new int[indices.length + 1];
146
147         for (int i = 0; i < indices.length; i++) {
148             relevantIndices[i] = indices[i];
149         }
150
151         relevantIndices[relevantIndices.length - 1] = sensitiveIndex;
152
153         // create subtable out of relevant indices
154         Table subTable = subtable(relevantIndices);
155         sensitiveIndex = relevantIndices.length - 1;
156
157         LinkedHashMap<Integer, HashSet<Object>> attributeMap = new LinkedHashMap<>();
158
159         for (Row row : subTable.getRows()) {
160             // get fingerprint for non-sensitive attributes
161             int fingerprint = row.partialFingerprint(0, sensitiveIndex - 1);
162
163             // init
164             if (attributeMap.get(fingerprint) == null) {
165                 attributeMap.put(fingerprint, new HashSet<>());
166             }
167
168             // get distinct sensitive values per QI* class
169             HashSet<Object> attributes = attributeMap.get(fingerprint);
170             attributes.add(row.getValue(sensitiveIndex));
171             attributeMap.put(fingerprint, attributes);
172         }
173
174         // reduce value sets to their size
175         ArrayList<Integer> distinctValues = new ArrayList<>();
176         attributeMap.values().forEach(set -> distinctValues.add(set.size()));
177
178         // get minimum value of distinct value occurrences per QI* block
179         int min = Collections.min(distinctValues);
180         return min;
181     }
182
183     /**
184      * Finds all quasi-identifiers that satisfy a given threshold value for their
185      * distinct ratio.
186      *
187      * @param sensitiveIndex The zero-based index of the sensitive attribute
188      * @param threshold      The minimum distinct ratio an attribute combination has
189      *                       to reach in order to count as quasi-identifier
190      * @param usePruning     if true, supersets of already determined
191      *                       quasi-identifiers will be ignored
192      * @return A descending list of quasi-identifiers, from largest to smallest
193      */
```

```
194     public ArrayList<QuasiIdentifier> findQuasiIdentifiers(int sensitiveIndex, double
            threshold,
195             boolean usePruning) {
196         ArrayList<QuasiIdentifier> quasiIdentifiers = new ArrayList<>();
197
198         // build array of attribute indices to consider
199         int[] range = IntStream.range(0, attributes.size() - 1).toArray();
200         LinkedHashSet<Integer> indices = new LinkedHashSet<>();
201
202         for (int i : range) {
203             if (i != sensitiveIndex) {
204                 indices.add(i);
205             }
206         }
207
208         // try every combination ("powerset") of attributes, with regard to pruning for
209         // supersets, if enabled
210         for (LinkedHashSet<Integer> combination : PowerSet.fromSet(indices)) {
211             if (usePruning) {
212                 ArrayList<QuasiIdentifier> identifiersToAdd = new ArrayList<>();
213
214                 for (QuasiIdentifier qi : quasiIdentifiers) {
215                     // if combination is a superset of an already determined combination
                            ...
216                     if (combination.containsAll(qi.getAttributeIndices())) {
217                         // build indices and attributes lists
218                         ArrayList<Attribute> qiAttributes = new ArrayList<>();
219                         ArrayList<Integer> qiIndices = new ArrayList<>();
220
221                         for (int i : combination) {
222                             qiAttributes.add(attributes.get(i));
223                             qiIndices.add(i);
224                         }
225
226                         // add quasi identifier without knowing its distinct ratio
227                         QuasiIdentifier quasiIdentifier = new QuasiIdentifier(this,
                                qiAttributes, qiIndices);
228                         identifiersToAdd.add(quasiIdentifier);
229                     }
230                 }
231
232                 for (QuasiIdentifier identifier : identifiersToAdd) {
233                     if (!quasiIdentifiers.contains(identifier)) {
234                         quasiIdentifiers.add(identifier);
235                     }
236                 }
237             }
238
239             LinkedHashMap<Integer, Integer> occurrences = new LinkedHashMap<>();
240
241             for (Row row : rows) {
242                 int[] qiIndices = combination.stream().mapToInt(Integer::intValue).
                        toArray();
243                 int fingerprint = row.selectiveFingerprint(qiIndices);
244
245                 // init
246                 occurrences.putIfAbsent(fingerprint, 0);
247
248                 occurrences.put(fingerprint, occurrences.get(fingerprint) + 1);
249             }
250
```

```
251             // get distinct ratio
252             int distinctValues = occurrences.keySet().size();
253             int allValues = rows.size();
254             double distinctRatio = (double) distinctValues / (double) allValues;
255
256             // catch all idenfifying combinations that satisfy the threshold
257             if (distinctRatio >= threshold) {
258                 ArrayList<Attribute> qiAttributes = new ArrayList<>();
259
260                 for (int i : combination) {
261                     qiAttributes.add(attributes.get(i));
262                 }
263
264                 ArrayList<Integer> qiIndices = new ArrayList<>(combination);
265                 QuasiIdentifier quasiIdentifier = new QuasiIdentifier(this, qiAttributes,
                        qiIndices, distinctRatio);
266                 quasiIdentifiers.add(quasiIdentifier);
267             }
268         }
269
270         // order descending by size, then return list of QIs
271         Collections.sort(quasiIdentifiers);
272         return quasiIdentifiers;
273     }
274
275     /**
276      * Given a quasi-identifier and a minimum k value, all rows that do not satisfy
277      * k-anonymity will be removed.
278      *
279      * @param qi             The quasi-identifier defining the QI* classes
280      * @param minOccurrences The k of k-anonymity
281      */
282     public void suppressRows(QuasiIdentifier qi, int minOccurrences) {
283         // get qi indices as array
284         LinkedHashMap<Integer, Integer> occurrences = new LinkedHashMap<>();
285         int[] qiIndices = qi.getAttributeIndices().stream().mapToInt(Integer::intValue).
                toArray();
286
287         // count occurrences
288         for (Row row : rows) {
289             int fingerprint = row.selectiveFingerprint(qiIndices);
290
291             // init
292             occurrences.putIfAbsent(fingerprint, 0);
293
294             occurrences.put(fingerprint, occurrences.get(fingerprint) + 1);
295         }
296
297         // get list of row fingerprints
298         ArrayList<Integer> fingerprints = new ArrayList<>(occurrences.keySet());
299
300         // filter by rows that do not satisfy k-anonymity
301         for (int fingerprint : fingerprints) {
302             if (occurrences.get(fingerprint) >= minOccurrences) {
303                 occurrences.remove(fingerprint);
304             }
305         }
306
307         // remove rows that do not satisfy k-anonymity
308         for (int i = 0; i < rows.size(); i++) {
309             if (occurrences.containsKey(rows.get(i).selectiveFingerprint(qiIndices))) {
```

```
310                 deleteRow(i--);
311             }
312         }
313     }
314
315     /**
316      * Generalizes a specific column by one step.
317      *
318      * @param index The zero-based index of the column
319      * @throws EndOfHierarchyException Signals that the attribute cannot further be
320      *                                 generalized
321      */
322     public void generalizeColumn(int index) throws EndOfHierarchyException {
323         // column mode
324         transpose();
325
326         // get hierarchy of attribute of interest
327         Row row = rows.get(index);
328         GeneralizationHierarchy hierarchy = attributes.get(index).getHierarchy();
329
330         if (hierarchy == null) {
331             transpose();
332             throw new NullPointerException(String.format("Attribute at index %d has no
                    hierarchy.", index));
333         }
334
335         // generalize every attribute
336         int i = 0;
337         for (Object value : row.getValues()) {
338             row.updateValue(i++, hierarchy.current().generalize(value.toString()));
339         }
340
341         // set hierarchy to next step; if end has been reached, a custom exception will
342         // be thrown
343         try {
344             hierarchy.next();
345         } catch (EndOfHierarchyException ex) {
346             throw ex;
347         } finally {
348             // restore row mode
349             transpose();
350         }
351     }
352
353     /**
354      * Generalizes a specific column by one step.
355      *
356      * @param attribute The attribute to generalize
357      * @throws EndOfHierarchyException Signals that the attribute cannot be further
358      *                                 generalized
359      */
360     public void generalizeColumn(Attribute attribute) throws EndOfHierarchyException {
361         int index = this.getAttributes().indexOf(attribute);
362         generalizeColumn(index);
363     }
364
365     // [...]
366
367     /**
368      * Permutates the rows.
369      */
```

```
370     public void shuffleRows() {
371         Collections.shuffle(this.rows);
372     }
373
374     // [...]
375
376 }
```

anonymizer.structures.Row

```
1  // [...]
2
3  /**
4   * Class that represents a single row (aka tuple).
5   */
6  public class Row {
7
8      private ArrayList<Attribute> rowAttributes;
9      private ArrayList<Object> rowValues;
10
11     /**
12      * Constructor for an empty row.
13      *
14      * @param table The table that contains the new row
15      */
16     public Row(Table table) {
17         this.rowAttributes = table.getAttributes();
18         this.rowValues = new ArrayList<>();
19     }
20
21     /**
22      * Constructor for a row with values.
23      *
24      * @param table  The table that contains the new row
25      * @param values The values, with any kind of length
26      */
27     public Row(Table table, Object... values) {
28         this(table);
29
30         for (int i = 0; i < values.length; i++) {
31             rowValues.add(values[i]);
32         }
33     }
34
35     // [...]
36
37     /**
38      * Gets all distinct values (row in set semantics). Especially useful for
39      * transposed tables.
40      *
41      * @return The distinct row values as {@link ArrayList} of objects
42      */
43     public ArrayList<Object> getDistinctValues() {
44         // init new set and list
45         HashSet<Object> valueSet = new HashSet<>();
46         ArrayList<Object> valueList = new ArrayList<>();
47
48         // convert list to set
```

```
49          for (Object value : rowValues) {
50              valueSet.add(value);
51          }
52
53          // re-convert set to list
54          for (Object value : valueSet) {
55              valueList.add(value);
56          }
57
58          return valueList;
59      }
60
61      // [...]
62
63      /**
64       * Gets the ratio of distinct values and all values of the row. Useful for QI
65       * calculations.
66       *
67       * @return Ratio, as double-precision value
68       */
69      public double distinctRatio() {
70          return (double) sizeDistinct() / (double) size();
71      }
72
73      /**
74       * Returns the so-called "fingerprint" of a row.
75       * This is a special case of a partial fingerprint, starting at zero, ending at
76       * the maximum value.
77       *
78       * @return The fingerprint
79       */
80      public int fingerprint() {
81          return partialFingerprint(0, rowValues.size() - 1);
82      }
83
84      /**
85       * Returns the so-called "partial fingerprint" of a row segment.
86       * This is a special case of a selective fingerprint, starting at the start
87       * index, ending at the end index, including all values in-between.
88       *
89       * @param start The zero-based start index
90       * @param end   The zero-based end index
91       * @return The "partial fingerprint"
92       */
93      public int partialFingerprint(int start, int end) {
94          IntStream indicesStream = IntStream.range(start, end + 1);
95          int[] indices = indicesStream.toArray();
96          indicesStream.close();
97
98          return selectiveFingerprint(indices);
99      }
100
101     /**
102      * Returns the so-called "selective fingerprint" of a row segment.
103      *
104      * @param indices An array of attribute indices
105      * @return The "selective fingerprint"
106      */
107     public int selectiveFingerprint(int... indices) {
108         StringBuilder compressedString = new StringBuilder();
109
```

```
110         for (int i : indices) {
111             compressedString.append(rowValues.get(i)).append(";");
112         }
113
114         // remove last semicolon
115         compressedString.deleteCharAt(compressedString.length() - 1);
116
117         // return hash code of compressed string ("fingerprint")
118         return compressedString.toString().hashCode();
119     }
120
121     // [...]
122
123 }
```

anonymizer.generalizations.Generalization

```
1  package anonymizer.generalizations;
2
3  /**
4   * This class represents an abstract generalization. Classes inheriting from
5   * this class need to implement the {@link Generalization#generalize(String)}
6   * method.
7   */
8  public abstract class Generalization {
9
10     protected Generalization() {
11
12     }
13
14     public abstract String generalize(String value);
15
16 }
```

anonymizer.generalizations.MapGeneralization

Diese Klasse dient als Beispiel für eine konkrete Implementierung der abstrakten `Generalization`-Klasse. Auf die Angabe weiterer Generalisierungen wird aus Platzgründen verzichtet.

```
1  // [...]
2
3  /**
4   * Generalization that maps values to more specific values. It can be used to
5   * implement concept hierarchies that cannot be expressed mathematically.
6   */
7  public class MapGeneralization extends Generalization {
8
9      private Map<String, String> map;
10
11     public MapGeneralization(Map<String, String> map) {
12         super();
13
14         this.map = map;
15     }
16
```

```
17      @Override
18      public String generalize(String value) {
19          return map.containsKey(value) ? map.get(value) : value;
20      }
21
22  }
```

anonymizer.generalizations.GeneralizationHierarchy

```
1   // [...]
2
3   /**
4    * Class that represents a generalization hierarchy for a certain attribute.
5    */
6   public class GeneralizationHierarchy {
7
8       private ArrayList<Generalization> generalizations;
9       private int step;
10
11      /**
12       * Constructor for a new, empty hierarchy.
13       */
14      public GeneralizationHierarchy() {
15          this.generalizations = new ArrayList<>();
16          this.step = 0;
17      }
18
19      /**
20       * Constructor for an already existing list of {@link Generalization}s.
21       *
22       * @param generalizations A list of generalizations
23       */
24      public GeneralizationHierarchy(ArrayList<Generalization> generalizations) {
25          this();
26          this.generalizations = generalizations;
27      }
28
29      // [...]
30
31      /**
32       * Gets the current generalization.
33       *
34       * @return the current generalization
35       */
36      public Generalization current() {
37          return generalizations.get(step);
38      }
39
40      /**
41       * Gets the next generalization and increases the internal step counter.
42       *
43       * @return the next generalization
44       * @throws EndOfHierarchyException Signals that the end of the hierarchy has
45       *                                 been reached
46       */
47      public Generalization next() throws EndOfHierarchyException {
48          if (step < generalizations.size() - 1) {
49              return generalizations.get(++step);
```

```
50          } else {
51              step++;
52              throw new EndOfHierarchyException();
53          }
54      }
55
56      /**
57       * Gets the amount of remaining generalization steps.
58       *
59       * @return Remaining stepts (integer)
60       */
61      public int getRemainingSteps() {
62          return this.generalizations.size() - step;
63      }
64
65  }
```

C. Aufbau des Datenträgers

Da diese Masterarbeit bedingt durch die COVID-19-Pandemie ausschließlich digital abgegeben wird, existiert kein Datenträger. Alle benötigten Daten werden deshalb online bereitgestellt.

Stud.IP Auf Stud.IP befindet sich eine Kopie dieser Masterarbeit sowie des LaTeX-Projektes. Ebenfalls wird dort sämtliche verwendete Literatur bereitgestellt. Die Veranstaltung heißt *„MA: Anonymisierung von Data Provenance in ProSA“* und ist unter der URL `https://studip.uni-rostock.de/seminar_main.php?auswahl=fb305c24a0865edd4f5a493b5085cb9e` zu finden. Dort wiederum befinden sich alle benötigten Dateien im Ordner „Abgabe“.

GitLab Für die *ProSAnon*-Software existiert ein Git-Repository auf dem *Universität-Rostock*-eigenen GitLab-Server. Die URL lautet `https://git.informatik.uni-rostock.de/ns382/prosanon` und kann mittels `git` geklont werden.

- HTTPS: `https://git.informatik.uni-rostock.de/ns382/prosanon.git`
- SSH: `git@git.informatik.uni-rostock.de:ns382/prosanon.git`

Selbstständigkeitserklärung

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

Rostock, den 15. März 2022