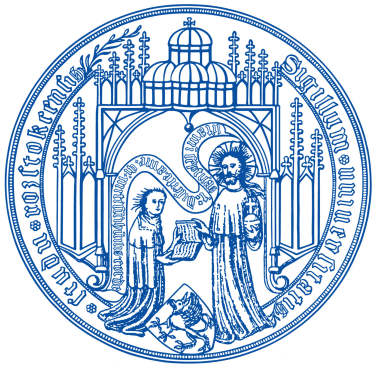
Datenschutzkonforme Auswertung von Stromdaten

Masterarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik



vorgelegt von:	Eric Klein
Matrikelnummer:	215204156
geboren am:	08.02.1997
Betreuer / Erstgutachter:	Florian Rose
Zweitgutachter:	Michael Fellmann
Abgabedatum:	10.12.2024

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Personenbezogene Daten stellen in der heutigen Zeit mehr denn je ein fundamentales Gut dar. Trotz öffentlich eingekehrten Bewusstseins und gesetzlichen Regularien ist und bleibt das Thema wohl stets ein hochaktuelles – die Welt verändert sich schnelllebig, der allgemeine Umgang ist an vielen Stellen nach wie vor leichtfertig und absoluter Schutz lässt sich nur in den seltensten Fällen erreichen. Das von manchen Menschen überaus leichtfertig geäußerte Argument, dass man selbst ja nichts zu verbergen hätte, ist so geläufig geworden, dass es im Jahre 2013, also im Zuge der Snowden-Enthüllungen, einen eigenständigen Wikipedia-Eintrag erhalten hat [Wik13]. Man kann nur erahnen, wie viele Informationen eine Person allein durch alltägliches Agieren im digitalen Raum hinterlässt, geschweige denn gezwungenermaßen preisgeben muss, zumal sich dieser stets und allgegenwärtig ausweitet. Das Digitale ist längst in die Sphären der kleinsten und unscheinbarsten Geräte vorgedrungen. Smarte Geräte erheben zum Beispiel gezielt bis ganz beiläufig kontinuierliche Daten über die Umgebung, die körperliche Verfassung, soziale Interaktionen oder das Konsumverhalten. Üblicherweise sind diese fortlaufenden Datenströme dabei auf Aktualität bedacht und daher Speicher- sowie Zeitbeschränkungen unterworfen. Besonders kritisch wird es allerdings, falls diese Daten obendrein personenbezogen Ursprungs sein sollten.

Personenbezogene Daten befinden sich nämlich in der misslichen Lage, dass sie sich zur Gewinnung allgemeiner Erkenntnisse fernab des Individuums nutzen lassen, aber zugleich hochsensible Informationen darstellen könnten. Aufgrund dessen sind verschiedene Anonymisierungsverfahren entwickelt worden, die gewährleisten sollen, dass auch Außenstehende mit ausgewählten Daten arbeiten dürfen, sofern sie soweit verfremdet wurden, dass eine rückwirkende Identifizierung ausgeschlossen ist.

Die Kernfrage dieser Masterarbeit lautet, inwiefern sich gängige Anonymisierungstechniken datenschutzkonform mit Stromdaten kombinieren lassen. Hierzu bedarf es vorab natürlich einem Konzept zur Operationalisierung von Anonymisierungsverfahren, das bereits praktischerweise von mir selbst in meiner damaligen Bachelorarbeit „Parallele Anonymisierung von großen Datenbeständen“ [Kle20] erarbeitet wurde. Die dort aufgestellten Formalisierungen werden daher die fundamentale Grundlage dieser Masterarbeit bilden. Jene aufgestellten Operationalisierungen werden dementsprechend wiederaufgegriffen und darüber hinaus in erheblichem Maße ausgebaut. Die Erweiterung verfolgt nämlich zwei Ziele zugleich: Zum einen werden elementare neue Parameter zum privacy-konformen Arbeiten mit Stromdaten eingeführt; zum anderen wird nebenbei die Gelegenheit genutzt, um die damals äußerst minimalistisch angelegten Operatoren um weitere Einstellungsmöglichkeiten grundlegend zu verfeinern.

1.2 Aufbau

Zu Beginn dieser Arbeit werden in Kapitel 2 drei der gängigsten Anonymisierungsverfahren vorgestellt: Generalisierung, Slicing und Differential Privacy. Jeder dieser Techniken wird dann in gesonderten Kapiteln als formaler Operator der relationalen Algebra eingeführt – natürlich in deutlich reduzierter Form mit Beschränkung auf wesentliche Kernfunktionen, da sich innerhalb dessen unmöglich alle, geschweige denn besonders komplexe, Spezialtechniken

abdecken lassen. Kapitel 3 Daten führt in das Thema Stromdaten ein. Kapitel 4 bietet veranschaulichend eine selbsterstellte umfangreiche Übersicht zu einer Vielzahl von Fenstertypen. In Kapitel 5 werden spezifische neue Grundprobleme und Lösungsansätze herausgestellt, die sich bei der Kombination von Anonymisierungstechniken mit Fensterverarbeitung ergeben. Kapitel 6 befasst sich dem Differential-Privacy-Operator, Kapitel 7 mit dem Slicing-Operator und Kapitel 8 mit dem Generalisierungs-Operator. Zunächst wird jeweils zum allgemeinen Verständnis der damalige Minimaloperator beschrieben. Anschließend werden dessen Schwächen in Bezug auf verschiedene Fenstertypen ergründet. Zum Schluss wird jeweils der neue erweiterte Operator eingeführt und dessen Funktionsreichtum an etlichen exemplarischen Beispielen verdeutlicht. Kapitel 9 stellt eine praktisch umgesetzte Implementierung vor, die die datenschutzkonforme Arbeitsweise einer Anonymisierungstechnik in einer Stromdatenumgebung zeigt. Hierbei wurde sich für das Slicing-Verfahren entschieden, da es demonstrativ alle elementaren Konzepte aufgreift und sich überdies als exzellente Blaupause für andere Techniken eignet. Kapitel 10 umfasst ein kurzes Schlusswort, in dem zuletzt noch einmal allgemein gewonnene Erkenntnisse und einige zukünftige Forschungsmöglichkeiten herausgestellt werden. Daran anschließend folgt dann das Literaturverzeichnis.

2 Datenschutz

Das Anfangskapitel dieser Arbeit widmet sich der allgemeinen Einführung in das Themengebiet Datenschutz. Es vermittelt grundlegende Techniken und Metriken, die eingesetzt werden, um ebendiesen gewährleisten zu können. Dieses Kapitel entstammt den Einführungskapiteln 2 und 3 der eigenen Bachelorarbeit [Kle20], die abseits einiger Modifikationen weitestgehend wortgetreu übernommen wurden. Die Abschnitte 2.1−2.5 basieren dabei auf Kapitel 2; der Abschnitt 2.6 basiert auf der Einleitung von Kapitel 3 und Abschnitt 2.7 basiert auf Abschnitt 3.2.

2.1 Anonymisierungstechniken

Die Ausgangsdaten, auf die ein Anonymisierungsverfahren angewandt werden soll, können je nach Kontext unterschiedlichen Kategorien angehören. Allgemein lassen sie sich zwischen Identifikatoren (IDs), Quasi-Identifikatoren (QIDs) und sensitiven Daten (SD) unterteilen. Identifikatoren sind die aus relationalen Datenbanken bekannten Schlüsselattribute. Sie ermöglichen die eindeutige Tupelidentifizierung und dürfen auf keinen Fall offengelegt werden, da sich mittels Verknüpfung noch weitere Informationen gewinnen ließen. Sensitive Daten sind jene Attribute, die schützenswerte Informationen darstellen. Quasi-Identifikatoren sind Attributkombinationen mit schlüsselähnlicher Wirkung. Hierbei sei betont, dass nicht gleich alle wie ein Schlüssel wirken müssen, sondern schon ein Großteil genügt [Sam01]. Zur abschließenden Veranschaulichung ein Beispiel, zu sehen in Abbildung 1, das fernab einiger Abwandlungen von Nielsen et al. stammt [NJTF15]:

IdNr	PLZ	Alter	Krankheit
1	47677	29	Herzerkrankung
2	47602	22	Herzerkrankung
3	47678	27	Herzerkrankung
4	47905	43	Grippe
5	47909	52	Herzerkrankung
6	47906	47	Krebs
7	47605	30	Herzerkrankung
8	47673	36	Krebs
9	47607	32	Krebs

Abbildung 1: ursprüngliche Patientendaten

Die abgebildete Tabelle zeigt eine unanonymisierte Relation, die Personendaten von erkrankten Patienten enthält. Als ID wurde eine künstliche Identifikationsnummer namens `IdNr` vergeben. Die schützenswerten Daten beschränken sich auf das sensitive Attribut `Krankheit`. Die Attribute `Alter` und die Postleitzahl `PLZ` sind nicht-sensitiv und als QID-Menge zu betrachten. In diesem Beispiel sind sie nicht erst in Kombination, sondern sogar schon einzeln gefährlich.

2.2 Generalisierung

Bei Generalisierung erfolgt eine Werteersetzung hin zu weniger präzisen Daten. Das Ganze geschieht aber nicht willkürlich, sondern wahrt eine semantische Konsistenz. Numerische

Zahlen, Uhrzeit oder Tag lassen sich beispielsweise als Intervall zusammenfassen. Worte mit geringer Zeichendistanz können verschiedenartige Teile durch ein universelles Token, was quasi ein zensierendes Symbol außerhalb des regulären Zeichensatzes repräsentiert, unterdrücken. Häufig verwendete Unterdrückungszeichen sind dabei das X, ein Unterstich oder das Sternsymbol $*$. Es können auch sprachwissenschaftliche Konzepte wie Taxonomien genutzt werden, was insbesondere bei Werten fernab mathematischer Grundlagen nötig wird. Ganz allgemein ausdrückt werden bei der Generalisierung zusammenfassende Oberbegriffe gefunden [Swe02].

Sweeney definiert Generalisierung als eine Funktion $f : A \to B$, die die Werte eines Attributs A begrenzt oft in neue überführen kann, also nach der Form:

$$A_0 \xrightarrow{f_0} A_1 \xrightarrow{f_1} ... \xrightarrow{f_{n-1}} A_n$$

Die Ausgangsmenge A_0 ist dabei die Wertedomäne einer noch unanonymisierte Attributspalte A. Des Weiteren gibt es stets ein maximales Element A_n, an welchem jedes weitere Generalisieren zum Erliegen kommt. Die gesamte Ordnungsrelation, die sich darauf bilden lässt, wird Generalisierungshierarchie DGH_A genannt [Swe02]. Neben *domain generalization hierarchy* (DGH) ist ebenso der Begriff *value generalization hierarchy* (VGH) gebräuchlich [NJTF15].

Ausgehend von zuvor vorgestellten Patientendaten verdeutlichen Abbildung 2 und 3 mögliche Hierarchien für die QID-Attribute `PLZ` und `Alter`.

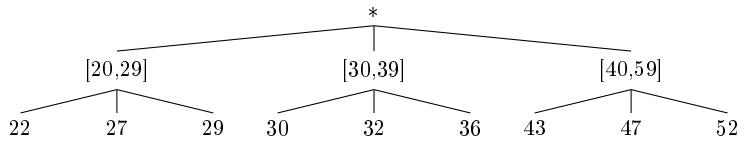


Abbildung 2: DGH_{Alter}

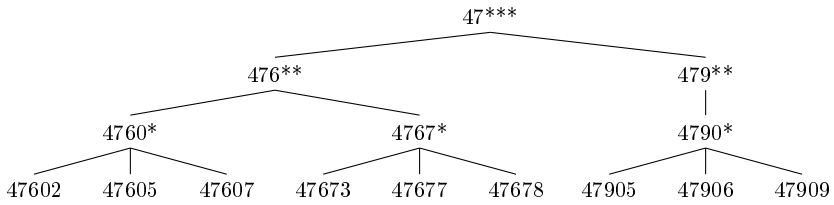


Abbildung 3: DGH_{PLZ}

2.2.1 k-Anonymität

Eine Relation ist k-anonym, falls jedes Tupel bezüglich seiner quasi-identifizierenden Attributmenge von mindestens $k-1$ anderen nicht mehr unterscheidbar ist. Da die wenigsten

Datenbestände so etwas naturgemäß erfüllen, wendet man zur Erreichung dessen eine Generalisierung an [SS98]. Blöcke mit gleicher Belegung der QIDs bilden folglich Äquivalenzklassen (ÄK) aus, die mitunter auch als Q^*-Blöcke bezeichnet werden [MKGV07].

Im Konkreten lässt sich aus unseren Patientendaten (Abb. 1) sowie den beiden Generalisierungshierarchien (Abb. 2 und 3) eine generalisierte Tabelle erzeugen, die 3-Anonymität zusichert, sodass jeder Patient von mindestens zwei anderen nicht mehr zu unterscheiden ist. Dieser Ansatz hat jedoch seine Grenzen, da er sich nur um die Gruppenbildung selbst, nicht aber deren Mitglieder sorgt. Den Patienten mit den ID-Nummern 1 bis 3 nützt es zum Beispiel herzlich wenig, wenn Angreifer mit QID-Kenntnis dennoch von deren Herzkrankheit erfahren, da alle Attributwerte dieses Blocks wertgleich sind. Machanavajjhala et al. [MKGV07] bezeichnen einen solchen Angriff bei Mangel an Diversität eine Homogenitätsattacke (*homogeneity attack*). Außerdem legten sie dar, dass ein Angriff über Hintergrundwissen (*background knowledge attack*) möglich wäre. Durch Zusatzwissen könne ein Angreifer Bestimmtes ausschließen oder aber für wahrscheinlich halten [MKGV07]. Aus diesem Grunde wurde das Konzept der k-Anonymität erweitert.

	Nicht-sensitiv		Sensitiv	
IdNr	PLZ	Alter	Krankheit	
1	476**	[20,29]	Herzerkrankung	1-divers
2	476**	[20,29]	Herzerkrankung	
3	476**	[20,29]	Herzerkrankung	
4	479**	[40,59]	Grippe	3-divers
5	479**	[40,59]	Herzerkrankung	
6	479**	[40,59]	Krebs	
7	476**	[30,39]	Herzerkrankung	2-divers
8	476**	[30,39]	Krebs	
9	476**	[30,39]	Krebs	

Abbildung 4: 3-anonymisierte Fassung

2.2.2 l-Diversität

Allgemein versteht man unter l-Diversität, dass neben k-Anonymität zusätzlich gefordert wird, dass die sensitiven Daten einer jeden Äquivalenzklasse mindestens l verschiedene Werte aufweist [Hau07]. Unbeabsichtigte Eindeutigkeit wird dadurch unterbunden. Die soeben anonymisierte Tabelle mit 3-Anonymität (Tabelle 4) gewährt leider nur 1-Diversität, da der erste Q^*-Block wertetechnisch homogen ist.

Machanavajjhala et al. [MKGV07] haben darüber hinaus sogar noch eine deutlich differenziertere Unterteilung aufgestellt. Dies würde allerdings zu sehr in die Tiefe gehen. Für weitere Techniken wie die Entropie-l-Diversität und andere sei daher auf meine damalige Bachelorarbeit [Kle20] verwiesen, in der noch deutlich speziellere Metriken behandelt wurden.

Nach [Hau07, LLV07] bietet l-Diversität typischerweise Spielraum für Angriffe zweierlei Arten. Die *similarity attack* ist dabei ein Angriff, der sehr einer Homogenitätsattacke gleicht. Zwar mögen konkrete Werte einer Äquivalenzklasse verschieden ausgeprägt sein, doch könnten sie zugunsten des Angreifers dennoch auf eine inhaltliche Gemeinsamkeit hinauslaufen.

Bei einer *skewness attack* kommt ein schief verteilter Sachverhalt zum Tragen, bei dem der eher unwahrscheinliche Anteil eine Gefahr ausstrahlende Assoziation birgt. Wer sich aufgrund von Diversitätsforderung nun einen Q^*-Block mit einigen Gefahrenfällen teilen muss, die dort gewiss nicht unterrepräsentiert sind, könnte aus Vorsicht ebenso wie einer behandelt werden.

Das nun bestehende Problem ist, dass der Diversitätsgedanke zwar innerhalb der Gruppen selbst sichergestellt wird, jedoch nicht die semantische Nähe untereinander [LLV07].

2.2.3 t-Closeness

Der Eigenschaft t-closeness liegt folgendes Prinzip zugrunde: Jede Äquivalenzklasse sowie die gesamte Tabelle selbst soll eine bestimmte Verteilung von sensitiven Attributen besitzen. Wenn die Verteilung einer Gruppe gegenüber der Gesamtverteilung höchstens eine Distanz von t aufweist, dann liegt t-closeness vor. Trifft dies auf alle Äquivalenzklassen zu, so ist auch die gesamte Tabelle t-close [LLV07].

Kurz gesagt ist t also ein Maß dafür, wie sehr sich alle Gruppen unterscheiden. Möchte man die Probleme der l-Diversität kleinhalten, muss man wiederum ähnliche Verteilungen anstreben. Je ähnlicher die Gruppen, desto geringer nämlich der Wissensgewinn für Angreifer [Hau07]. Ab einem gewissen Punkt führt es den Zweck von Äquivalenzklassen aber wohlgemerkt ad absurdum.

Die Messung der Ähnlichkeit kann über verschiedene Distanzmetriken geschehen. Für alle werden als Argumente zwei diskrete Verteilungen $P = (p_1, p_2, ..., p_m)$ und $Q = (q_1, q_2, ..., q_m)$ erwartet. P soll im Folgenden immer die Gesamtverteilung des sensitiven Attributes `Krankheit` repräsentieren und Q dessen Verteilung innerhalb einer der Äquivalenzklassen. Jede deckt definitionsraumseitig die gleichen Attributwerte ab. Im Einzelnen kann eine bestimme Häufigkeit auch den Wert 0 annehmen. Ein gleicher Index repräsentiert auch untereinander gleiche Definitionswerte. Da in diesem Fall ein einheitliches t gewollt ist, sind die Häufigkeiten relativ bzw. die Werte normiert. In der Summe ist jede Verteilung daher 1.

Eine Vorstellung konkreter Distanzmetriken würde an dieser Stelle wohl etwas zu sehr vom Thema abführen. Bei vertiefendem Interesse an Metriken wie der Variationsdistanz, der Kullback-Leibler-Divergenz sowie der Earth-Mover's-Distanz sei daher erneut auf meine damalige Bachelorarbeit[Kle20] verwiesen, in der auch jene mit vorgestellt wurden.

2.3 Unterdrückung

Der Vollständigkeit halber sei noch die Technik der Unterdrückung erwähnt. Bei dieser werden üblicherweise bestimmte Attributwerte ausgeblendet. Darüber hinaus kann Unterdrückung auch auf ganze Attribute oder Tupel angewandt werden. Auf der einen Seite enthält man sensitive Daten zwar vor, auf der anderen müsste man unter Umständen weniger stark generalisieren, da diese Ausreißer nicht abgedeckt werden müssten. Vereinzelte Löschung könnte daher feiner aufgeteilte QIDs ermöglichen und womöglich aussagekräftigere Daten bieten [SS98]. Bezogen auf das bloße Erreichen von k-Anonymität oder l-Diversität hilft einem diese Technik jedoch nicht weiter – sie arbeitet ja prinzipiell dagegen an. Für

bessere t-closeness kann sie aber durchaus von Nutzen sein. Alternativ zur Unterdrückung wäre auch eine Generalisierung der sensitiven Attribute denkbar [LLV07].

2.4 Slicing

Anstelle von Generalisierung setzt die Technik des Slicens auf Vertauschung. Die Grundidee des Ganzen stammt von [LLZM12]. Da mit Generalisierung eine Minderung des Informationsgehaltes einhergeht, wollte man eine alternative Anonymisierungstechnik anbieten, die höheren Datennutzen erlaubt. Dies kommt vornehmlich der Datenanalyse zugute, insbesondere der aggregierenden – den regressiven und korrelierenden Verfahren dagegen nicht. Beim Slicing wird ein Datenbestand n - mal horizontal sowie m - mal vertikal zerlegt. Die $n \cdot m$ Teilrelationen dieses Rasters werden dann zufällig permutiert und anschließend wieder verknüpft. Folglich werden manche Attributverbindungen bewahrt, während andere aufgelöst werden. Zur Sicherung der Privatheit müssen korrelierende Attribute dabei unbedingt zusammenbleiben [LLZM12, GH15]. Im Folgenden wird das Slicing-Prinzip von Li und anderen nach dem Wortlaut von [GH15] erklärt.

Geschlecht	Alter	Krankheit	PLZ
männlich	29	Herzleiden	47677
männlich	22	Herzleiden	47602
weiblich	27	Herzleiden	47678
weiblich	43	Grippe	47905
weiblich	52	Herzleiden	47909
männlich	47	Krebs	47906
weiblich	30	Herzleiden	47605
weiblich	36	Krebs	47673
männlich	32	Krebs	47607

Abbildung 5: Ausgangsdaten

Der erste Schritt ist das horizontale Splitten (bei Li et al. *tuple partition* genannt). Hierbei findet eine Partitionierung in n disjunkte Teile mittels Selektionsbedingungen statt. Im Idealfall ließen sich die zugrunde liegenden Werte damit bestens aufteilen. Ansonsten muss ein zusätzliches Ranking-Attribut eingeführt werden. Man nummeriert also die gesamte Tupelmenge, sodass man sich dann beliebige Intervalle heraussuchen kann. Für die Tabelle in Abb. 5 kam eine Nummerierung mit Gruppierungsgröße 3 zum Einsatz. Der Übersichtlichkeit wegen wurde auf temporäre Attribut verzichtet und stattdessen auf eine farbliche Vorhebung zurückgegriffen.

Der zweite Schritt ist das vertikale Splitten (*attribute partition*). Hierbei wird das Partitionieren mittels Projektion noch weitergetrieben. Alles, was horizontal unterteilt wurde, wird nun auch m - mal vertikal unterteilt. Dies muss aber nicht zwanghaft disjunkt sein, wie es noch bei [LLZM12] der Fall ist. Wichtig ist nur ein Zusammenbleiben stark korrelierender Attribute, da deren Teilung Rückschlüsse erlauben könnte. Im ausgewählten Beispiel bleiben einerseits die Attribute `Geschlecht` und `Alter` und andererseits die Attribute `Krankheit` und `PLZ` zusammen.

Der nächste Schritt ist das Permutieren, der eigentliche Anonymisierungsschritt. Auf jeder

der entstandenen Teilrelationen wird nun eine randomisierte Umsortierung angewandt. Das Ergebnis des Ganzen spiegelt sich bereits jetzt in Abb. 6 wider.

Der vierte und letzte Schritt ist das erneute Zusammenfügen. Alle Teilrelationen werden wieder zu einer ganzen gemacht. Sie ist ebenso groß wie die ursprüngliche Relation, nur dass einiges positionell vertauscht wurde. Die intern eingeführten Hilfsattribute werden selbstverständlich wieder ausgeblendet. Es folgen ein gesamtheitlicher Verbund, der die vertikale Teilung aufhebt, und eine Vereinigung, die Gleiches mit der horizontalen tut.

Geschlecht	Alter	Krankheit	PLZ
männlich	22	Herzleiden	47678
weiblich	27	Herzleiden	47677
männlich	29	Herzleiden	47602
weiblich	43	Herzleiden	47909
männlich	47	Krebs	47906
weiblich	52	Grippe	47905
weiblich	30	Krebs	47673
männlich	32	Krebs	47607
weiblich	36	Herzleiden	47605

Abbildung 6: Geslicte Daten

Abschließend sei noch angemerkt, dass der relationale Beschreibungsstil der Schritte vor allem der Verständlichkeit wegen ausgewählt wurde. Eine gute praktische Implementierung würde diesen Ansatz wohl niemals wortgetreu umsetzen, da er sehr ineffizient laufen würde. Man sollte in der Praxis also davon ausgehen, dass sich Einträge über Felder oder ähnliche Strukturen noch weitaus effizienter ansprechen lassen.

2.5 Differential Privacy

Bei Differential Privacy werden vereinzelte Daten durch zufälliges Rauschen verzerrt. Dieser Ansatz soll eine zuverlässige individuelle Identifizierbarkeit verhindern, ohne dabei die Verteilung von Werten gravierend zu beeinflussen [DKM+06].

Kernstück des Ganzen ist eine randomisierende Funktion: der Zufallsmechanismus κ [Dwo06]. Bei jeder Ausführung auf einen Datenbestand erfolgt eine zufällige Manipulation einiger Attributwerte. Anonymität wird ermöglicht, da man nicht mehr sagen kann, ob ein einzelner Eintrag noch wahrheitsgetreu oder aber verfälscht ist. Natürlich könnte eine willkürliche Veränderung von Daten diese zunehmend unbrauchbar machen. Damit sie in ihrer Gesamtheit dennoch Aussagekraft behalten, existieren formalen Modelle wie ε-Differential-Privacy (kurz ε-DP) oder auch die (ε,δ)-DP, durch welche man dies sicherstellen kann. Da sie die budgetierte Obergrenze an erlaubter Anonymität angeben, spricht man diesbezüglich auch von *privacy budget* [FS10].

Im Folgenden werden diese zwei Metriken sowie die Grundidee von Datenverzerrung veranschaulicht. Erstere gehen zurück auf [Dwo06, Dwo08, DKM+06], wogegen das reine Konzept des Zufälligmachens deutlich weiter zurückreicht und daher schwer zurückzuverfolgen ist.

2.5.1 Randomised-Response-Technik

Das Grundprinzip des Randomisierens kann unter anderem schon bei [War65] gefunden werden. Die dortige Methode veranschaulicht einfach, abstrakt und effektiv, wie man zufälliges Rauschen in seine Daten bringen kann. Auch anderswo ist dieses gängige Beispiel mit leichten Abwandlungen anzutreffen.

Im Grunde wird einer Probandengruppe eine simple Ja-Nein-Frage gestellt und deren Antwort obendrein von einem Münzwurf abhängig gemacht. Man kann sich an dieser Stelle eine beliebige platzhaltende Frage wie „Kennen Sie X?“ oder „Mögen Sie Y?“ zu einer Person, einem Film oder dergleichen ausdenken. Im Übrigen kann die Münze sowohl ideell als auch gezinkt sein. Dies obliegt ganz dem Experimentator. Bevor die Testperson eine Antwort verkündet, muss sie zuvor noch einige Male eine Münze werfen. Zum Fällen einer Entscheidung braucht sie höchstens zwei Würfe. Sollte das Werfen nicht geheim erfolgen, dann wirft sie sie auch noch die verbleibenden Male, sodass niemand imstande ist, Rückschlüsse zu ziehen.

Bei Kopf weiß sie bereits, dass sie wahrheitsgemäß antwortet. Bei Zahl soll stattdessen die Münze für sie antworten. Was sie sagt, hängt ganz vom zweiten Wurf ab. Bei Kopf beispielsweise „ja“, bei Zahl „nein“.

Natürlich ließe sich das Ganze auch komplexer gestalten. Anstelle des zweiten Münzwurfes (hier quasi eine Bernoulli-Verteilung) kann eine ausgedachte Antwort auch auf Basis einer beliebigen anderen Verteilung erzeugt werden. Der Ergebnisraum kann endlich, abzählbar unendlich oder ebenso gut stetig sein.

Geschlecht	Alter	Krankheit	PLZ
männlich	29	Herzleiden	47677
männlich	23	Herzleiden	47602
weiblich	27	Herzleiden	47678
weiblich	43	Krebs	47909
weiblich	52	Herzleiden	47909
männlich	47	Grippe	47906
weiblich	30	Herzleiden	47605
weiblich	36	Krebs	47673
männlich	32	Krebs	47607

Abbildung 7: Tabelle mit Rauschen

Zur weiteren Veranschaulichung wurde auch auf die Daten aus Abb. 5 ein Rauscherzeugungsverfahren angewandt, ersichtlich in Abb. 7. Da dieses Beispiel zunächst einzig Anschauungszwecken dienen soll, wurden die konkreten Verrauschungsmethoden an dieser Stelle ausgelassen. Manipulierte Werte sind rot hervorgehoben worden. Welche Werte beim Verrauschen herauskommen können, mag zufällig sein, aber werteseitig nichtsdestotrotz vordefiniert. Es könnten komplett neue Werte hinzukommen, es könnte aber auch eine reine Häufigkeitsverschiebung der vorliegenden Werte geben. Obwohl der Zufall alles gestattet und seine Notwendigkeit hat, darf dieser nicht ausarten. Die Zahl an verhältnismäßigen Änderungen sollte also eine Grenze besitzen. Ebenso sollte ein Kompromiss zwischen dem Auftreten gänzlich neuer Werte und bloßer Umverteilung (sprich Slicing) gefunden werden.

2.5.2 ε-Differential-Privacy

Die folgende Definition geht zurück auf die Arbeit von [Dwo06]. Seien D_1 und D_2 zwei Datenbestände, die sich höchstens um einen Eintrag unterscheiden. Sei $\mathcal{K} : X \to W$ ein Zufallsmechanismus, dessen Münzwurf auf einen Wahrscheinlichkeitsraum abgebildet wird. Weiterhin ist die Verteilung eines jeden Attributes gemeinhin bekannt, sodass Pr[E] die Eintrittswahrscheinlichkeit eines Ereignisses E angibt. Sei ε ein Maß für die Privatheit mit der Randbedingung $\varepsilon \geq 0$. Für alle D_1, D_2 und $S \subseteq W$ muss gelten:

$$\Pr[\mathcal{K}(D_1) \in S] \leq e^{\varepsilon} \cdot \Pr[\mathcal{K}(D_2) \in S]$$

Durch diese Vorschrift wird sichergestellt, dass die Ausblendung eines beliebigen Eintrages keinen signifikanten Unterschied für die Gesamtheit zur Folge hat. Da die An- oder Abwesenheit eines Einzelnen keine Rolle spielt, kann in jedem Fall für dessen individuellen Schutz gesorgt werden. Absoluten Schutz garantiert dies allerdings nicht [Dwo08].

Wie man sehen kann, wird eine zulässige Grenzabweichung um einen multiplikativen, also relativen Faktor e^{ε} gefordert, der wiederum maßgeblich vom gewählten ε abhängt. Da eine relative Toleranz aber durchaus restriktiv wirken kann, war es naheliegend zur Relativierung zusätzlich ein konstante, sprich absolute Toleranz einzuführen.

2.5.3 (ε, δ)-Differential-Privacy

Wie der Name schon vermuten lässt, handelt es sich hierbei um eine Erweiterung der ε-Differential-Privacy. Zur Definierung von (ε,δ)-DP kann also auf das Vorherige aufgebaut werden. Einzig bei der längeren Ungleichung am Ende bedarf es einer Neudefinition. Neu ist die zweite wählbare Maßzahl δ mit der Randbedingung $0 \leq \varepsilon \leq 1$. Im Grunde bewirkt sie nicht viel mehr, als die frühere Gleichung additiv zu ergänzen. Im Konkreten gilt nun:

$$\Pr[\mathcal{K}(D_1) \in S] \leq e^{\varepsilon} \cdot \Pr[\mathcal{K}(D_2) \in S] + \delta$$

Das neu eingeführte δ erlaubt es, bei den vorangegangenen Anforderungen ein wenig toleranter zu agieren. Ist die Wahrscheinlichkeit eines Datensatzes grundsätzlich groß, würde das linear mitwachsende e^{ε} noch genug Spielraum zur Erfüllung der Ungleichung bieten, als dass der eine Datensatz großartig ins Gewicht fiele. Geringe Wahrscheinlichkeiten sind bei ε-DP daher kritische Probleme. Durch δ gibt es bei (ε, δ)-DP nun eine minimale Schranke, die eine bestimmte Grundtoleranz ermöglichen soll [DKM$^+$06].

2.6 Auswahl eigener Operatoren

Im Verlaufe dieses Kapitels sind etliche grundlegende Anonymisierungstechniken vorgestellt worden. Einige davon werden nachfolgend wie Operatoren der relationalen Algebra formalisiert werden. Da sich aufgrund des Umfangs unmöglich alle berücksichtigen lassen, wird der Fokus nur auf drei grundlegenden Techniken liegen. Im Rahmen der eigenen Bachelorarbeit wurde dafür bereits eine fundamentale Grundlage geschaffen, die nun wieder aufgegriffen und erweitert wird. In der eigenen Bachelorarbeit [Kle20] wurde die Operationalisierung absolut minimalistisch gehalten und hinsichtlich der Vertauschbarkeit mit anderen relationalen Operatoren untersucht. Im Zuge dieser Masterarbeit soll sie nun grundlegend erweitert und

im Besonderen auch streamingtauglich gemacht werden. Die operationalisierte Auswahl ist in Abb. 8 ersichtlich.

Techniken	metrische Unterteilung	eigener Operator
Generalisierung	k-Anonymität	ja
	l-Diversität	nein
	t-closeness	nein
Permutation	Slicing	ja
Differential Privacy	Münzwurf-Randomisierung	ja
	ε-DP	nein

Abbildung 8: Übersicht der eingeführten Operatoren

2.7 Fortlaufendes Beispiel

Zur effektiven Veranschaulichung aller weiteren Beispiele wurde bereits für die eigene damalige Bachelorarbeit [Kle20, Abschnitt 3.2] eine kompakte Wertedomäne geschaffen, mit der sich alle drei Techniken einheitlich visualisieren ließen. Diese Beispielwelt wird auch für die Masterarbeit übernommen. Rein inhaltlich sind die Daten an das Thema Blutspende angelehnt. Der vorrangige Grund geht dabei auf die Darstellbarkeit zurück. Die verschiedenen Wertedomänen heben sich nämlich deutlich voneinander ab und sind überschaubar ausgeprägt. Aufgrund der Vielzahl und grundverschiedenen Ausrichtung an Beispielen sollte die inhaltliche Ebene gar nicht allzu sehr zerdacht werden. Obwohl man das Ganze also recht abstrakt betrachten sollen, sei nichtsdestotrotz kurz die Bedeutung und Werteausprägung aller Attribute aufgeschlüsselt. Alles näheren Informationen sind Abb. 9 zu entnehmen.

Attr.	Bedeutung	Wertebereich
A	Spendetag	01.01. bis 31.12.
B	Blutgruppe (genotypisch)	AA, A0, BB, B0, AB, 00
C	Rhesusfaktor	$+$, $-$
D	Alter der Person	18 bis 73

Abbildung 9: Erklärung der einzelnen Attribute

Zuletzt sei noch angemerkt, dass von den ausgewählten Techniken Slicing und Differential Privacy mit Zufall arbeiten. Ein konkretes Beispiel kann dementsprechend nur eine von vielen möglichen Relationen bebildern, die ausgehend von der Ausgangsrelation bei Gebrauch des Operators resultieren kann.

3 Datenströme

Dieses Kapitel widmet sich der Einführung in das Thema Stromdaten. Strukturell wie inhaltlich nutzt es als ständige Leitlinie das Buch von [ÖV11]. Wenn spezifische Aussagen aus wissenschaftlichen Papern aufgegriffen wurden, sind selbstverständlich auch die entsprechenden Originalverweise mit übernommen worden.

Unter einem Datenstrom wird eine fortlaufend übertragene, geordnete Abfolge von Datensätze verstanden, welche hauptsächlich zur Informationsverarbeitung bestimmt sind. Sie kommen üblicherweise in Echtzeit und einer bestimmten Reihenfolge an, die sich nicht völlig kontrollieren lässt. Damit stets eine flexible und zeitkritische Verarbeitung gewährleistet werden kann, sind die analytischen Möglichkeiten maßgeblich an die Größe, Menge und Ankunftsgeschwindigkeit gebunden [BFO04, GÖ03]. Aufgrund dessen treten auch Aspekte wie die Speicherung weit in den Hintergrund. Eine historische Sicherung ist zumeist zweitrangig und ohnehin selten von Relevanz – und selbst der zeitweilige Aufbau effizienter Suchstrukturen würde kaum etwas nützen, weil es die Verarbeitung zunächst ausbremst und spätere Suchersparnisse aufgrund der schnellen Irrelevanz der Daten wiederum ausbleiben.

Datenströme verhalten sich push-lastig und werden datengetrieben ausgewertet. Sie stehen damit im Gegensatz zu pull-lastigen Modellen wie relationalen Datenbanken, deren Auswertung abfragegesteuert angetrieben wird. Der Kernanspruch bei Datenströmen liegt nicht auf deren Lagerung, sondern Verarbeitung. Neue Stromdaten können laufend eintreffen. Erschwerend kann hinzukommen, dass nur ein Bruchteil der eingehenden Daten überhaupt von Relevanz ist. Um fortlaufend mithalten zu können, muss eine ebenso zügige Verarbeitung erfolgen, was zeit- und speicherlimitierende Echtzeitanforderungen bedeutet. Neue Daten sind dadurch zwangsläufig wichtiger als ältere und haben irgendwann deren Verdrängung zur Folge. Allerdings heißt dies auch, dass nur begrenzte Ausschnitte zur Verfügung stehen. Da der potenziell unendliche Stream nie in Gänze vorliegen kann, können sogar grundsätzlich keine blockierenden Operationen (z. B. ein Verbund) auf den gesamten Strom angewandt werden [ÖV11, S. 724].

Für die praktische Umsetzung ergibt sich daraus: Abfragen müssen sich während ihrer Lebensdauer an variable Bedingungen anpassen und notfalls Einbußen in Kauf nehmen, um performant zu bleiben; die Abfrageplanung muss Puffer, Warteschlangen und die Zeitplanung managen; bei Ausreizung der Speicherkapazität oder Ankunftsrate ist ein Lastabwurf nötig; bei übermäßigem Berechnungsaufwand können wenn überhaupt nur noch Teilergebnisse erzeugt werden oder man entscheidet sich, dass manche Abfragen ausgelassen werden – wahlweise durch Absenkung der Frequenz oder Priorisierung der wichtigeren. An dieser Stelle sei allerdings betont, dass dies kein relevanter Fokus dieser Arbeit ist. Im Folgenden wird stets von gleichbleibenden guten Bedingungen ausgegangen, die Einbußen ausschließen.

Im alltäglichen Sprachgebrauch sind Stromdaten bzw. Datenströme ein deutlich weit gefasserer Begriff, was bereits stark durch die Anwendungsmöglichkeiten bedingt ist. Die fortlaufend eingehenden Datenströme können nämlich verschiedensten Quellen entstammen. Sie reichen beispielsweise von Sensordaten aus mobilen Geräten, wissenschaftlichen Instrumenten, Datenbanken, News-Feeds und Kameras bis hin zu sozialen Netzwerken. Auch wenn die obige wissenschaftliche Definition es nicht vorschreibt, sind im wissenschaftlichen Kontext jedoch im Regelfall fest strukturierte Einträge, die dem Relationenmodell folgen, gemeint

[ÖV11]. Dementsprechend sind etwa Audiodaten, Videodaten sowie analoge Messungen, die sich in schlichter relationaler Form nicht repräsentieren lassen, dort weitgehend marginalisiert, sodass sie vielmehr ein relativ spezifischeres Anwendungsgebiet darstellen. Ähnliches gilt dann auch für semistrukturierte Daten, Graphdaten etc.

Auch wenn es nicht der Regel entspricht, ist ebenso noch die Verarbeitung von aufgezeichneten Datenströmen denkbar. Darunter versteht sich, dass ein endlicher Ausschnitt oder beendeter Strom ebenso als sekundär gespeicherte Aufzeichnung vorliegen könnte. In diesem Fall wären idealisierte Abfragen möglich, die keinen zeit- oder speicherkritischen Einbußen unterliegen und somit auch experimentell reproduzierbar sein können. Im Kern bleiben es dabei die gleichen fensterbeschränkten Auswertungen, nur unter idealisierten Bedingungen. Weitere dadurch ermöglichte Anwendungsszenarien wären beispielsweise: (1) die Sicherung oder Beschaffung elementarer Ein- oder Ausgabeströme für einen selbst oder Dritte; (2) das jederzeit mögliche Nachholen von Analysen; (3) die Verbindung historischer Datenquellen mit Echtzeitdaten; (4) eine Verfeinerung oder fundamentale Veränderung der persistenten Abfrage; (5) eine Gütevergleich zwischen Echtzeit- und Aufzeichnungsergebnissen.

Die Anwendungsmöglichkeiten erstrecken sich über verschiedenste Gebiete. Eine Vielzahl von Stromdaten tritt in transaktionaler Form bei der Protokollierung geräteübergreifender Kommunikation auf. Dazu zählen etwa Kreditkartenkäufe, Telefongespräche und Serverzugriffe auf Webresourcen. Der andere Großteil entsteht in Form von Messungen bei der Zustandsbeobachtung von Einzelgeräten. Hierzu zählen beispielsweise Aktienkurse, Bewegungs- und Positionsdaten, Klimadaten einer Wetterstation, lokale Verkehrsmessungen und weitere sensorisch erfassbaren Phänomene.

3.1 Relationales Datenstrom-Modell

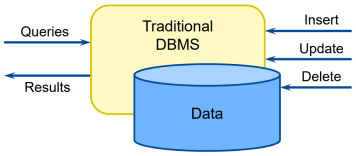
Ein Stream setzt sich aus einer potenziell unendlichen Abfolge von Tupeln zusammen. Die Tupel eines Streams unterliegen wiederum einem festen vordefiniertem Relationenschema. Darüber hinaus soll jedes Tupel notwendigerweise über ein Zeitstempel-Attribut verfügen, was u. a. die Ankunftszeit oder den geplanten Lesezeitpunkt repräsentieren könnte. Diese wesentlichen Eigenschaften bilden das relationale Datenstrom-Modell, das die Stellung relationaler Anfragen über Streams ermöglicht. Für abweichende Datenformen wie semistrukturierte oder multimedialen Inhalte sind dementsprechend alternative Modelle nötig.

Üblicherweise besitzt man keine Kontrolle über die Ankunftsreihenfolge von Einträgen. Die erste Möglichkeit ist daher, die implizite Ordnung durch den Ankunftszeitpunkt hinzunehmen. Dies wäre z. B. bei Messinstrumenten mit direkten Übertragungswegen denkbar. Außerdem könnte der zeitliche Puffer oder die Updaterate ausreichend groß gewählt sein, sodass man schwankenden Übertragungszeiten und instabilen Verbindungen entgegenwirkt – etwa bei viertelstündlich erhobenen Wetterdaten. Nicht zuletzt spielt die Reihenfolge in manchen Szenarien keine entscheidende bis gar keine Rolle.

3.2 Data Base vs. Data Stream Management System

Database Management System (DBMS)

Ein Datenbank-Management-System ist ein Softwaresystem, das die langfristige Speicherung und Verwaltung von Datenbeständen sowie die Ausführung nutzerseitig gestellter Anfragen ermöglicht. Es fungiert als Schnittstelle zwischen Datenbank und Endbenutzern und erlaubt diesen, Daten einzufügen, darauf Abfragen zu stellen, sie zu aktualisieren und zu löschen.



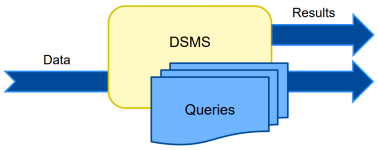
Die Daten eines DBMS sind auf persistente Lagerung angelegt. Sie liegen wohldefiniert in relationaler Form und werden als exakt vorausgesetzt. Ein Datenbestand ist über die Zeit hinweg in der Regel moderaten Änderungsraten ausgesetzt – zwar nicht unbedingt geringfügig, wenn ständige Verwaltung ein Kernbestandteil ist, aber nicht in einem derart häufigen Maße, als würden überall sensorische Messwerte nahezu live abgebildet werden, die millisekündlich Updates erfahren. Im Falle einer Anfrage wird der gesamte Datenbestand als Grundlage genutzt. Zur Erhöhung der punktuellen Effizienz können Indexe erzeugt und mitverwaltet werden, die Direktzugriffe in unterlinearer Zeit ermöglichen.

Eine gewöhnliche Anfrage ist von flüchtiger Dauer. Sie wird einmalig gestellt, ausgeführt und danach vergessen. Als Resultat produziert sie eine einmalige Ergebnismenge in Request/Response-Manier. Für das Endergebnis ist allgemein nur der gegenwärtige Datenbankzustand von Bedeutung. Reihenfolgen sind in einem DBMS im Regelfall nicht garantiert, wenngleich es zumeist so wirkt, da dieselbe Abfolge sehr wahrscheinlich aufgrund sequenziellen Lesens wiederholt zustande kommt. Zudem kann die Abfrageverarbeitung geplant und optimiert werden.

Weiterhin wird an ein DBMS nur geringe bis keine zeitkritischen Anforderungen gestellt. Die Abarbeitung soll selbstverständlich möglichst zeitnah geschehen, aber man nimmt Zeit- und Ressourcenaufwand notfalls billigend in Kauf, solange dadurch die ACID-Regeln gewährleistet bleiben. Außerdem soll stets genügend Sekundärspeicher zur Verfügung stehen – theoretisch unbeschränkt, praktisch ausreichend.

Data Stream Management System (DSMS)

Ein Data Stream Management System ist ein Softwaresystem zur Verwaltung von kontinuierlich eingehenden Datenströmen. Es ermöglicht, wie es bereits bei traditionellen DBMS der Fall ist, die deklarative Formulierung von Abfragen, was über SQL-ähnliche Erweiterungen wie CQL möglich ist. Der eigentliche Umgang mit Daten, Abfragen und weiteren Kernaspekten ist dagegen diametral entgegengesetzt zu DBMS ausgerichtet.



Die Daten eines DSMS sind auf Flüchtigkeit angelegt. Sie liegen in relationaler Form vor, was bereits durch das relationale Datenstrom-Modell vorgeschrieben ist. Stromdaten können mitunter das Risiko bergen, ungenau oder rasch veraltet zu sein. Außerdem kann der Zufluss hohe Datenraten, sprich Aktualisierungen, in den GB/s-Bereich und darüber hinaus annehmen – mit Aktualisierung ist dabei zumeist das Verdrängen alter Daten durch neue gemeint. Aufgrund dessen stellen Indexe weit mehr Aufwand als Nutzen dar, sodass ein schlichter sequentieller Zugriff bevorzugt wird.

Eine Anfrage bleibt solange beständig, bis sie ausdrücklich fallen gelassen wird. Aufgrund ihres Fortbestehens wird sie auch als kontinuierlich bezeichnet. Sie wird einmalig gestellt und dann wiederholt ausgeführt. Als Resultat erzeugt sie fortlaufend neue (Teil)Ergebnismengen. Neben dem eigentlichen Inhalt ist bei der Ergebnisbildung ebenso die schwer kontrollierbare Ankunftsreihenfolge, auch Historie genannt, mitentscheidend. Bei einem DBMS ist die Reihenfolge im Vergleich dazu im Regelfall nicht gesichert, auch wenn es aufgrund sequenziellen Lesens zumeist so erscheint. Darüber hinaus wirken noch selbstbestimmte Variablen wie die Fenstergröße oder das Bewegungsverhalten darauf ein.

Weiterhin wird an ein DSMS Echtzeitanforderungen gestellt. In der Regel sind Anfragen zeitkritisch, sodass alle notwendigen Stromdaten im limitierten Hauptspeicher gehalten werden müssen. Falls es aufgrund schwankender Datenraten zu einem Flaschenhalseffekt kommt oder sich einer abzeichnet, muss adaptiv über Einbußen in Form eines Lastenabwurfs entschieden werden. Die Abstoßung kann dabei veraltete oder neue Daten begünstigen. Falls selektive Eintragsverdrängung die Ergebnisse berechnungsbedingt stark verfälscht oder die Berechnungen grundsätzlich nicht mehr hinterherkommen, kann sogar eher eine großräumige bis komplette Leerung ratsam sein.

Im Regelfall können Stromdaten irgendwann vergessen werden. In Bezug auf den Hauptspeicher ist diese Aussage auch absolut zutreffend. Gleichwohl könnte der Strom aber auch auf einem sekundären Medium als Backup für Nachbetrachtungen aufgezeichnet werden, falls Revidierungen oder erneute Lesevorgänge, solange der Datenstrom oder die ursprüngliche Abfrage noch läuft, ausgeschlossen sind. Im Nachhinein lassen sich dann auch aufgezeichnete Ströme analyisieren, was ohne Echtzeiteinschränkungen bestmöglich und vor allem reproduzierbar getan werden kann.

3.3 Datenmodelle

Das *append-only*-Datenmodell verfolgt die schlichte Aneinanderreihung von Elementen in Bezug auf einen Zeitstempel [GMM+03]. Vorgegebene Reihenfolgen erfordern in diesem Fall ein vorausschauendes Management, das Aspekte wie Vollständigkeit, Korrektheit und eine rechtzeitige Ankunft sicherzustellen versucht. Im trivialsten Fall würde sogar schon die indirekte Reihenfolge akzeptiert werden, in der die Eingabedaten eingetroffen sind. Sie würde

als grundsätzlich korrekt angenommen werden und wäre damit unveränderbar. Das *append-only*-Modell stellt das allgemein präferierte Modell hinsichtlich der Ankunftsreihenfolge dar.

Darüber hinaus stünden aber auch flexiblere Ansätze zur Verfügung. Durch Einführung von Revisionstupels *(revision tuples)* könnten frühere Ausgaben gezielt korrigiert werden, indem sich ausgewählte Eingabedaten für ungültig erklären und alle daraus entstandenen Einträge wieder zurücknehmen ließen [RMCZ06].

Da Einträge auch in dichten Schüben eintreffen könnten, ließe sich ein Datenstrom außerdem als eine Mengen- oder Multimengenabfolge von Einträgen modellieren [TMSF03]. Eine Menge speichert in diesem Ansatz alle Einträge, die zur selben Zeiteinheit eingetroffen sind. Zwischen den Tupeln der gleichen Mengeneinheit würde keine festgelegte Reihenfolge bestehen.

Neben rein relationalen Auswertungen existieren auch die artverwandten Event-Modelle, die zur Verarbeitung von Ereignissen in einem Publish/Subscribe-Umfeld konzipiert sind. Das Auftreten von Ereignissen kann in der globalen Betrachtung nämlich ebenso als Stream aufgefasst werden. Der Vorteil von Ereignisabfolgen besteht in der Vielzahl weiterer Analysemöglichkeiten. So lassen sich zum Beispiel auch Änderungen in den Daten detektieren oder Mustern erkennen [WDR06].

Innerhalb dieser Arbeit werden ideelle Daten angenommen, die nicht revidiert werden müssen. Revisionen würden in Verbindung mit Anonymisierungen nämlich weitere Sicherheitsprobleme verursachen, was kein Schwerpunkt dieser Arbeit sein soll. Bei Relevanz der Reihenfolge ist davon auszugehen, dass zur Annäherung an ideelle Bedingungen eine ausreichende Toleranz mit eingeplant wurde, welche die Korrektheit erwartende Auswertung hinauszögert. Vereinzelte Nachzügler, die sich jenseits der eingeräumten Toleranz verspäten, werden ignoriert oder erhalten einfach einen später datierten Zeitstempel, solange dies keine destruktiven Auswirkungen auf die inhaltliche Integrität der Daten haben sollte.

3.4 Monotonie-Eigenschaft

Aufgrund des nichtflüchtigen Charakters von Anfragen ist es schon aus Echtzeitgründen unabdingbar, zwischen grundverschiedenen Entwicklungsverhalten der Ergebnismenge zu unterscheiden. Eine solche Unterscheidung ist über die Monotonie möglich. Persistente Abfrage können wahlweise monoton oder nichtmonoton ausfallen. Vorab sei angemerkt, dass die beiden Formeln rein zur mengentheoretischen Beschreibung der bis dato entstandenen Gesamtergebnismenge dienen, also nicht als Berechnungsvorschrift einer Abfrage missverstanden werden sollten. Außerdem wurde von der Notation der Lehrbuchformeln von [ÖV11] deutlich abgewichen, um den beabsichtigten Sachverhalt noch verständlicher darzustellen.

Eine monoton wachsende Abfrage ist eine Abfrage, die inkrementell um weitere Ergebnisse anwächst. In formalerer Ausdrucksweise hieße das: Sei $Q^+(t)$ die inkrementelle Gesamtergebnismenge einer persistenten Anfrage Q zum Zeitpunkt t, die sich aus der Vereinigung zurückliegender Einzelergebnismengen $Q(t)$ zusammensetzt. Abfrage Q heißt monoton, falls $Q(t_i) \subseteq Q(t_j)$ für alle $t_i \leq t_j$ gilt. Daraus folgt, dass sofern sich keine Stagnation einstellen sollte, nur Wachstum über Inserts zu erwarten ist. Im Falle einer monotonen Anfrage

wird die Gesamtergebnismenge also rein über die zum Zeitpunkt t_n neu hinzugekommenen Einträge ergänzt:

$$\textit{monoton wachsend:} \quad Q^+(t_n) := \bigcup_{i=1}^{n} (Q(t_i) - Q(t_{i-1})) \quad \text{für } n \geq 1 \text{ mit } Q(t_0)=\varnothing$$

Abfrage $Q(t_0)$ repräsentiert hierbei die initiale leere Menge, noch bevor die erste Auswertung stattgefunden hat. Der Ausdruck $Q(t_i) - Q(t_{i-1})$ beschreibt formal die Differenzmengenbildung zwischen einer Abfrage mit der vorherigen. Aufgrund der Obermengeneigenschaft $Q(t_i) \supseteq Q(t_{i-1})$ bedeutet dies im Grunde einfach, dass die Gesamtergebnismenge um die gänzlich neuen Einträge der aktuellsten Abfrage ergänzt wird. Bei Monotonie gilt auf mengentheoretischer Ebene die Gleichsetzung $Q^+(t) = Q(t)$.

Die Wertebasis eines Append-only-Streams ist grundsätzlich monoton. Ein Fortbestehen auf höherer Ergebnisebene ist wiederum nur mit monotonitätsbewahrenden Operatoren gewährleistet. Dazu zählen etwa Einzeloperatoren wie Selektion oder Verbund. Von [LWZ04] wurde bewiesen, dass eine Abfrage nur dann monoton ist, wenn sie nicht blockiert, d. h. wenn sie nicht bis zum Ende der Eingabedaten warten muss, bis sie Ergebnisse liefern kann. Der Vorteil monotoner Abfragen besteht in der äußerst effektiven Ergebniserweiterung. Folglich ist die Antwort auf eine monotone persistente Abfrage ein kontinuierlicher Strom von Ergebnissen, der sich einfach anhängen und auch ohne sonderlichen Aufwand zeitweilig aufschieben ließe, falls zudem Stapelverarbeitung gewünscht sein sollte.

Eine nichtmonotone Abfrage kann wiederum Ergebnisse liefern, die in späteren Einzelabfragen nicht mehr vorhanden sind. Dies kann sich sowohl im Hinzufügen neuer als auch Ändern oder Löschen bestehender Daten äußern. Folglich muss jede Ergebnismenge von Grund auf neu berechnet werden. Für die bis dato entstandene Gesamtergebnismenge gilt dann formal:

$$\textit{nichtmonton:} \quad Q^+(t_n) := \bigcup_{i=0}^{n} Q(t_i) \quad \text{für } n \geq 1 \text{ mit } Q(t_0)=\varnothing$$

Veranlasste Änderungen revidieren innerhalb dieser Notation keine früheren Abfragen, sondern wirken sich nur auf das nächstfolgende Abfrageergebnis $Q(t_{n+1})$ aus. Auf formaler Ebene entspricht die Nichtmonotonie einer Generalisierung der Monotonie, die diese als unwahrscheinlichen Spezialfall inkludiert. Rein praktisch wird das Wort aber eher zur strikten Trennung benutzt. Man sollte im Regelfall also davon ausgehen, dass sich zwangsläufig eine Änderung ereignet, durch die irgendwann $Q^+(t_n) \neq Q(t_n)$ eintritt. Schon bei einer einmaligen zeitaktuellen Abfrage in einem DBMS sind nichtmonotone Abfragen (wie z. B. eine einfache Differenzmenge) blockierend, da sie den gesamten Datensatz lesen müssen, bevor sie Ergebnisse liefern können. Im Rahmen eines DSMS und persistenten Abfragen verhält sich eine Negation erst recht nichtmonoton, selbst wenn sie bloß über einem reinen Append-Stream ausgeführt werden würde.

3.5 Fenster

Ein Fenster (*window*) begrenzt einen potenziell unbegrenzten Datenstrom zwecks Weiterverarbeitung auf einen ausgewählten endlichen Ausschnitt. Es ist das gängige Konvertierungsmittel zur Umwandlung eines Datenstroms in eine Relation. Fenster lassen sich typischerweise nach folgenden vier Kriterien klassifizieren:

3.5.1 Wertebereiche

Bei *Tupelfenstern*, auch physische Fenster, Instanzfenster oder Zählfenster genannt, wird die Fensterlänge durch eine bestimmte Anzahl an Einträgen definiert. Bei *Zeitfenstern*, auch logische Fenster genannt, wird die Fensterlänge dagegen in Form eines Zeitintervalls vorgeschrieben. Beide Fenstertypen sind an ihren direkten oder indirekten Sortierschlüssel gebunden. Bei Tupelfenstern sind die Eintragsentwicklungen unweigerlich vorherbestimmt, bei Zeitfenstern dagegen ungewiss.

Darüber hinaus lassen sich *partitionierte Fenster* definieren, bei denen der Fensterinhalt in Gruppen unterteilt und jedes Gruppenfenster dann separat ausgewertet wird [ABW06]. Die allgemeinste Fensterform ist das *Prädikatfenster*, bei dem ein auswählbares Prädikat den Inhalt des Fensters bestimmt [GAE06]. Es ist weitgehend mit einer materialisierten Sicht vergleichbar.

3.5.2 Bewegungsverhalten der Endpunkte

Zwei feste Endpunkte bilden ein sogenanntes stehendes Fenster (*fixed window*). Zwei in gleichen Teilen verschobene Endpunkte, wahlweise gleichgerichtet vorwärts oder rückwärts, bilden ein sogenanntes Jumping-Fenster (*jumping window*). Ein fester und ein beweglicher Endpunkt bilden ein Landmark-Fenster (*landmark window*, kurz LM). Die Verschiebung des beweglichen Endes kann wahlweise vorwärts oder rückwärts erfolgen, wobei die Richtung üblicherweise konsequent beibehalten wird. Insgesamt ergeben sich neun Bewegungsmöglichkeiten, da jeder der beiden Endpunkte wahlweise fixiert sein oder aber sich vorwärts oder rückwärts bewegen kann. Falls man den Endpunkten sogar Beschleunigung und Richtungsumkehr gestattet, würden sich noch weitere kombinatorische Möglichkeiten ergeben. Die folgende selbsterstellte Übersicht, die neben den typischen Bewegungsverhalten auch ungewöhnlichere mit abdeckt, soll dazu dienen, um auch für Letztere eigens gewählte, konsistente Bezeichnungen einzuführen.

hop_A	hop_E		(vollständige)	Fenster-Bezeichnung(en)	Streben
< 0	< 0	$hop_A{=}hop_E$	vergangenheitsgerichtetes	Sliding, Jumping, Tumbling, Session	konst.
		$hop_A{>}hop_E$	vergangenheitsgerichtet	relativ wachsend	$\to \infty$
		$hop_A{<}hop_E$	vergangenheitsgerichtet	relativ schrumpfend	$\to 0$ [ace]
< 0	$= 0$		vergangenheitsgerichtet	einseitig wachsendes LM	$\to \infty$
$= 0$	< 0		vergangenheitsgerichtet	einseitig schrumpfendes LM	$\to 0$ [ad]
< 0	> 0		–	beidseitig wachsendes LM	$\to \infty$
$= 0$	$= 0$		–	stehend	
> 0	< 0		–	beidseitig schrumpfendes LM	$\to 0$ [ab]
$= 0$	> 0		zukunftsgerichtet	einseitig wachsendes LM	$\to \infty$
> 0	$= 0$		zukunftsgerichtet	einseitig schrumpfendes LM	$\to 0$ [ad]
> 0	> 0	$hop_A{=}hop_E$	zukunftsgerichtetes	Sliding, Jumping, Tumbling, Session	konst.
		$hop_A{<}hop_E$	zukunftsgerichtet	relativ wachsend	$\to \infty$
		$hop_A{>}hop_E$	zukunftsgerichtet	relativ schrumpfend	$\to 0$ [ace]

Zur Bezeichnung des Richtungsverhaltens eines Hops sind nachfolgend vier synonyme Entsprechungen möglich:

Hop-Wert	Hop-Vorzeichen	Symbol	adjektivische Bezeichnung
$hop < 0$	$\mathrm{sgn}(hop) = -1$	◀	vergangenheitsgerichtet
$hop = 0$	$\mathrm{sgn}(hop) = \ 0$	▽	stehend
$hop > 0$	$\mathrm{sgn}(hop) = +1$	▶	zukunftsgerichtet

Anmerkungen:

[a] Sofern irgendwann einmal Zeitstempel $t_A > t_E$ eintreten sollte, würden standardmäßig leere Ergebnismengen erzeugt. Falls Umstand a verhindert werden soll, könnte man als weitere Sonderform einen Tausch der aktuellen beiden Zeitstempel vornehmen.

[b] Im Falle gegensätzlicher Sprungrichtung ist es dabei gleichgültig, ob die Sprungweite (hop-Größe) mitgetauscht wird oder nicht.

[c] Im Falle gleicher Sprungrichtung muss dagegen auch zwangsweise die Sprungweite getauscht werden, um die Verklemmung aufzulösen. Ohne hop-Tausch bliebe die Ergebnismenge bei gleichgerichteten Sprüngen weiterhin leer, da trotz Endpunkte-Tausch immer wieder sofort der irreguläre Zustand $t_A > t_E$ anschließen würde. Erst durch gleichzeitigen Tausch der hop-Größe kann die anfängliche Schrumpfphase in fortwährendes relatives Wachstum umgekehrt werden.

[d] Im Falle dessen, dass nur ein Ende stehend vorliegt, verhält es sich wie bei Umstand c. Zur Auflösung einer Verklemmung ist es auch hier zwingend, bei einem Aufeinandertreffen die Sprungweite mitzuvertauschen.

[e] Verschiebt man bei c alternativ zum Mittausch der Sprungweiten stattdessen das langsame Ende in besonderem Maße ausweitend vom schnelleren Ende weg, wann immer der Kipppunkt erreicht ist, ließen sich auf diese Weise zyklische Schrumpfphasen hervorrufen.

Im Allgemeinen bietet der schrumpfungsbedingte Kipppunkt eine zusätzliche Möglichkeit, beliebige Verhaltenswechsel vorzunehmen. Der umgekehrte Fall – ein wachstumshemmender Kipppunkt beim Überschreiten einer Maximalgröße – ist folgerichtig natürlich ebenso denkbar. Dadurch eröffnet sich eine Vielzahl weiterer Kombinationen jenseits der oben genannten, die sich der Einordnung in ein derartiges Schema entziehen. Ein simples Beispiel wäre eine jo-jo-ähnliches Fenster mit einem stehenden Ende und einem zweiten, dass sich abwechselnd von ersterem entfernt und wieder annähert.

3.5.3 Fenster innerhalb von Fenstern

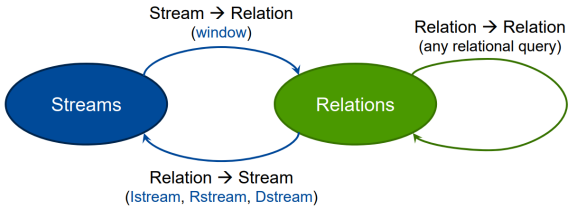
Beim *elastischen Fenstermodell* wird eine maximale Fenstergröße vorgegeben, um dann Abfragen über jedes denkbare kleinere Unterfenster innerhalb der Grenzen des Oberfensters laufen zu lassen [ZS03]. Auch beim *n-von-N-Fenstermodell* wird eine maximale Fenstergröße von N Tupeln oder Zeiteinheiten vorgegeben, um dann Abfragen auf allen kleineren Fenster der Größen $n \leq N$ durchzuführen, allerdings mit der zusätzlichen Einschränkung, dass einer der Unterfensterendpunkte stets mit einem Oberfensterendpunkt übereinstimmen muss [LLXY04].

3.5.4 Aktualisierungsverhalten

Bei der update-bestrebten (schnellen) Aktualisierung wird ein Fenster beim Eintreffen jedes neuen Tupels oder Auslaufen eines alten Tupels weitergeschoben, während es bei der Stapelverarbeitung (verzögerte Aktualisierung) wartet, bis sich mehrere Änderungen ansammeln konnten. Beim *Sliding-Fenster* wird mit jedem ein- oder wieder austretenden Tupel eine Änderung veranlasst. Dies ist daher die schnellstmögliche Form. Beim *Jumping-Fenster* erfolgt die Aktualisierung stets zu einem ausgewählten Zeitpunkt, wobei dieser kleiner oder gleich der Fenstergröße bleiben soll. Bis auf den äußersten Fall ist daher potenziell mit Überlappung zu rechnen. Beim *Tumbling-Fenster* entspricht das Aktualisierungsintervall genau der Fenstergröße, sodass hier auf direkt angrenzenden Partitionen gearbeitet wird. Beim *Session-Fenster* übersteigt die Aktualisierungsintervall die Fenstergröße, sodass inaktive Phasen entstehen, in denen Tupel übersprungen beziehungsweise nie von Interesse sein werden.

3.6 Continuous Query Language

Die Continuous Query Language (CQL) ist eine deklarative Anfragesprache für Datenströme in DSMS. Es handelt sich um eine Anfang der 2000er entwickelte SQL-Erweiterung von [ABW03] der Stanford University, die exemplarisch die Konvertierbarkeit von Relationen und Datenströmen beschreibt.



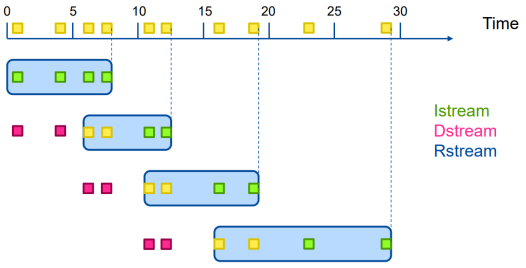
Das Arbeiten mit Streams verläuft dabei nach den folgenden drei Schritte ab:

1. Der Fenster-Operator selektiert aus einem unendlichen Stream ein spezifisches endliches Teilstück. Diese zeitdynamische Relation wird schlichtweg Fenster genannt.
2. Das aktuelle Fenster dient dann als Basisrelation für SQL-Anfragen. Es kann mit historischen Sichten sowie persistenten Daten kombiniert werden und einen weitreichenden Umfang relationaler Operatoren nutzen, solange es kein Aufwandsrisiko darstellt.
3. Auf Grundlage der aktuellen Ergebnismenge werden schließlich die neusten Stream-Elemente erzeugt. Da neben dem reinen aktuellen Fenster auch ein Differenzmengenvergleich mit dem zeitnächsten Nachbar sinnhaft sein könnte, stehen insgesamt drei stream-erzeugende Operatoren zur Verfügung.

Das Fehlen eines direkten Strom-zu-Strom-Operators rührt daher, dass er bewusst auf indirektem Wege über die anderen drei Opertionsschritte gebildet wird. Und auch wenn die Pipeline zentriert darauf ausgelegt ist, könnte ebenso mit anderer Zielsetzung konvertiert werden.

3.7 Relation zu Stream

Die Konvertierung von Relationen zu Datenströmen geschieht durch einen aktualitätsgetriebenen, fortlaufenden Append-Prozess, der auf Grundlage der aktuellsten Ergebnismengen stattfindet. Allgemein stehen dafür drei Konvertierungsoperatoren zur Auswahl, die verschiedene Zeitpunkte und Abbildungshäufigkeiten ermöglichen.



Istream(R): Insert-Stream
Der Einfüge-Strom gibt all jene Ergebnisse der aktuellsten Fensterinstanz aus, die gegenüber der vorherigen neu hinzugekommen sind. Jeder Eintrag wird einmalig und schnellstmöglich in den Strom geleitet.

Dstream(R): Delete-Stream
Der Lösch-Strom ist eine verzögerte Ausgabe, der neue Ergebnisse erst ausgibt, sobald diese aus der aktuellsten Ergebnismenge verdrängt, also sozusagen entfernt sind. Jeder Eintrag wird einmalig und mit ereignisspezifischer Verzögerung in den Strom geleitet.

Rstream(R): Relation-Stream
Der Relationen-Strom gibt auf direktem Wege alle Ergebnisse der aktuellsten Fensterinstanz aus. Bei Fensterüberlappung bis hin zur Monotonie kann diese Abbildungsform ein hohes Maß an Redundanz erzeugen.

3.8 Eigene Begrifflichkeiten zur vertiefenden Beschreibung

3.8.1 Lebensdauer-Abstufungen von Fenstern

Im theoretischen Sinne ist ein Datenstrom für Fensterabfragen als Gerade denkbar, die sich in beiden Richtungen ablaufen ließe. In der Realität muss es jedoch immer einen zeitlichen Ursprung geben, sodass man sich nie endlos in die Vergangenheit zurückbewegen kann und ein Datenstrom ideell gesehen einem Strahl entsprechen würde. In der Praxis wird es obendrein auch immer einen vorzeitigen Abbruch geben, da kein Programm ewig ausgeführt werden kann, sodass rein faktisch immer auf einer Strecke bzw. einem endlichen Bereich gearbeitet wird.

Unter ideellen Realitätsbedingungen wären unendlich viele Fenster möglich, falls der Datenstrom fortlaufend anwachsen und der Fensterbereich niemals permanent leer werden kann. Rein praktisch wird sich aber immer irgendein Abbruchgrund ergeben, der die endlose Weiterführung zunichte macht. Die hauptsächlichen Auslöser sind: schwindender Nutzen der gestellten Anfrage, Informationsausfälle aufgrund von Lieferabhängigkeiten bei Streamketten, Ressourcenlimitierung, Obsoleszenz der Hardware; im äußersten Grenzfall die begrenzte Existenz der Erde oder des Universums, wenn man so möchte. Von der reinen Möglichkeit her sei eine vorzeitige Terminierung regulär ohne weitere Erwähnung mit eingepreist, in den nachfolgenden Betrachtungen jedoch ausgeschlossen. Sie ließe sich stets herbeiführen, ist aber nie das primäre Ziel, sodass dieser triviale Aspekt nahezu nichts Inhaltliches beitragen kann.

unendlich: Die Fensteranzahl ist potenziell unendlich, falls ideell ein ständiger Streamzuwachs bzgl. der relevanten Laufrichtungen und ein endloses Bewegungsverhalten, das sich nicht von selbst erschöpft, gegeben sind. Im Übrigen ist der Unendlichkeitsbegriff hier streng im explorativen Sinne zu verstehen: Es werden nämlich tatsächlich neue Einträge und damit sich stets unterscheidende Fenster angestrebt.

endlich: Die Fensteranzahl ist endlich, falls ein endlicher Stream oder ein von selbst endendes Bewegungsverhalten des Fensters gegeben ist. Zum einen ist dies der Fall, falls beide Enden eines Fensters an irgendein Streamende stoßen sollten, wodurch es fortan stehend oder

leer wäre. Streng genommen trifft dies also schon auf jede Aufzeichnung zu. Zum anderen gilt dies, falls das Bewegungsverhalten schrumpfend oder stehend angelegt ist und anschließend terminiert. Jenseits der vereinfachten bzw. stringenten Bewegungsmuster kommt es natürlich nur auf die absehbare Selbstterminierung an – ein schrumpfendes Echtzeitfenster, das seine Größe immerzu wiederauflädt, kann sich natürlich ebenso gut unendlich entwickeln.

quasi-unendlich: Die Fensteranzahl ist quasi-unendlich, falls ein endlicher Stream und potenziell unendliches Bewegungsverhalten gegeben ist, wobei während der Ausführungszeit kein Stream- bzw. Aufzeichnungsende erwartungsgemäß erreicht wird. Unter diesen Umständen sei quasi dasselbe Verhalten wie im echten unendlichen Fall gewährleistet.

schein-unendlich: Die Fensteranzahl ist schein-unendlich, falls periodische Bewegungsmuster oder randomisierte Bewegungsverhalten auf einem endlichen Streamausschnitt angewandt werden, sodass schlussendlich nur mit erwartbarer Wiederkehr dieselben Fenster auftreten. Innerhalb dieser Arbeit ist der Begriff aufgrund der echten (explorativen) Fortbewegung nicht weiter von Gebrauch, da bei Unendlichkeit sowie Quasi-Unendlichkeit eine Scheinunendlichkeit im Folgenden ausgeschlossen ist. Allerdings zeigt es, dass der Begriff prinzipiell stets nützlich ist, um die Begriffe unendlich und quasi-unendlich weiter zu verschärfen.

eins: Die Fensteranzahl ist eins, falls nur ein einziges Fenster zu erwarten ist. Dieser Sonderfall tritt einzig ein, falls die Anfrage von vornherein als fixed window formuliert wurde. Im Grunde gleicht das Ganze dann einer Bereichsanfrage.

null: Eine Fensteranzahl von null kann multiple Gründen haben. In der theoretischen Betrachtung ist dies bereits absehbar, falls irreguläre, widersprüchliche oder nicht realisierbare Dinge definiert werden. Dazu zählen etwa unzulässige Fensterdefinitionen, wo das Ende zeitlich vor dem Anfang liegen soll. Im Besonderen gilt das auch für eine Fortbewegung jenseits des Ursprungs bzw. unzulässige Bewegungsverhalten: Über reine Echtzeitdaten, die stets zukunftsgerichtet sind, kann niemals ein vergangenheitsgerichtetes Fenster laufen. (In der Praxis kommen dann unvorhersehbare Umstände wie das völlige Ausbleiben von Einträgen oder das Unterschreitung der Mindestkapazität hinzu, die theoretisch betrachtet aber noch nicht von Belang sind.)

3.8.2 Größen zur Fensterbeschreibung

Fenstergröße

Die Fenstergröße bemisst den Intervallraum entlang des Streams, in dem sich die gegenwärtigen Einträge befinden. Die allgemeinste Variablenbezeichnung der Fenstergröße lautet im Folgenden $windowSize \in \mathbb{R}^+$. Die Einheit ist hier noch nicht näher spezifiziert, sodass sie vom jeweiligen Kontext definiert wird.

- Im Falle eines Tupelfensters wird die Fenstergröße nachfolgend Kapazität (*capacity*) genannt und in der Anzahl von Einträgen bemessen. Für die Kapazität gilt $c \in \mathbb{N}^+$.
- Im Falle eines Zeitfensters wird die Fenstergröße nachfolgend Zeitspanne oder Dauer (*duration*) genannt und in der Einheit Zeit bemessen. Für die Dauer gilt $d \in \mathbb{R}^+$.

Sprungweite / Hop

Die Sprungweite beziehungsweise der Hop bemisst die Verschiebung eines Fensterendes, durch welche neue Einträge hinzukommen oder bestehende Einträge vergessen werden können. Die allgemeinste Bezeichnung lautet im Folgenden $hop \in \mathbb{R}$. Auch hier ist die Einheit nicht näher spezifiziert und erschließt sich dadurch aus dem Kontext. Innerhalb dieser Arbeit und in der Praxis sind die Sprungwerte im Regelfall konstant. Nichtsdestotrotz könnten sie aber auch variabel angelegt werden.

- Im Falle eines Tupelfensters wird die Sprungweite nachfolgend $hopTuple \in \mathbb{Z}$ genannt und in der Anzahl von Einträgen bemessen.
- Im Falle eines Zeitfensters wird die Sprungweite nachfolgend $hopTime \in \mathbb{R}$ genannt und in der Einheit Zeit bemessen.

Sei hop_A repräsentativ die gegenwärtige Sprungweite des Fensteranfangs und hop_E die Sprungweite des Fensterendes. Im Falle von Tupelfenstern gelte allgemein hop_A, $\text{hop}_E \in \mathbb{Z}$, im Falle von Zeitfenstern hop_A, $\text{hop}_E \in \mathbb{R}$.

Bewegungsverhalten / Richtung

Das Bewegungsverhalten der Fenstergrenzen wird im Folgenden dreiwertig repräsentiert durch:

- ◄ als Sprung zu älteren Einträgen (vergangenheitsgerichtet)
- ▽ als Halt am gegenwärtigen Eintrag (stehend)
- ► als Sprung zu neueren Einträgen (zukunftsgerichtet)

Das jeweilige Richtung ermittelt sich dann über:

$$\text{Sprungweitenrichtung } dir_X := \text{sgn}(hop_X) \in \{-1, 0, +1\} = \{\blacktriangleleft, \triangledown, \blacktriangleright\}$$

Zuwachsverhalten

Reduziert auf das reine Vorhandensein im boolschen Sinne ließe sich das Zuwachsverhalten bereits mit dem Zeichenvorrat des Bewegungsverhalten ausdrücken:

$$\text{Zuwachsrichtungen } growth \subseteq \{-1, +1\} = \{\blacktriangleleft, \blacktriangleright\}$$

Da eine makroskopische Mitbetrachtung der Grenzen jedoch zusätzliche Konkretisierungen ermöglicht, seien folgende drei Grenzzuwachsfälle eingeführt:

- $\rightarrow$ als potentiell unendliches Wachstum
- $\dashv$ als endliches Wachstum, dessen Ende erwartungsgemäß erreicht wird
- $\rightarrow\!\!\!|$ als endliches Wachstum, dessen Ende erwartungsgemäß nie erreicht wird

Das deutlich detailliertere Zuwachsverhalten bildet sich dann aus:

$$\text{Zuwachsrichtungen } growth \in \{\leftarrow, |\!\leftarrow, \vdash\} \times \{\rightarrow, \rightarrow\!\!|, \dashv\}$$

Wachstumsumgebungen

Nicht alle Fenstertypen können beliebig eingesetzt werden, so wie es ideell möglich sein könnte. Die praktischen Gegebenheiten sorgen für eine maßgebliche Beschränkung des Anwendungsraumes. Zwecks begrifflicher Abgrenzung und weiterem Verständnis werden namentlich folgende Wachstumsumgebungen eingeführt:

$\longleftrightarrow$ als theoretisch unendliche Bidirektionalität

$\longmapsto$ als Echtzeit-Entwicklung

$\longmapsto\!\!\shortmid$ als (endliche) Aufzeichnung

$\shortmid\!\!\longmapsto\!\!\shortmid$ als quasi-unendliche Aufzeichnung

$\shortmid\!\!\longleftarrow$ als umgekehrte Echtzeit

Die unendliche Bidirektionalität $\longleftrightarrow$ entspricht der theoretischen Idealumgebung, in welcher der Fensterentwicklung keinerlei Grenzen gesetzt ist. Dieser Fall ist aber nahezu theoretisch, da die Bindung an einen Informationsursprung tatsächliche Bidirektionalität unmöglich macht. Eine seltene Ausnahme davon wäre bei Einträgen gegeben, die sich nach Bedarf komplett funktional berechnen lassen.

Dementsprechend ist im Besonderen die umgekehrte Echtzeitentwicklung $\shortmid\!\!\longleftarrow$ ein Theoriekonstrukt, das nicht in der Praxis Anwendung finden kann. Sie entspricht dem invertierten Zeitverlauf, der sich der menschlichen Wahrnehmung entzieht – ganz wie in Christopher Nolans Film „Tenet“, wenn man so möchte. Ein Bewegungsverhalten, das auf Dauer dem Informationszuwachs entgegenläuft, kann nur rückwirkend und in endlichem Maße über Aufzeichnungen aufrechterhalten werden.

Die Echtzeitendwicklung $\longmapsto$ entspricht der zukunftsgerichteten Entwicklung. Sie entspricht dem allgemeinen Zeitverständnis, entfernt sich fortlaufend vom absoluten Streambeginn und ist die einzige Umgebung, die sich im praktischen Sinne potentiell unendlich entwickeln kann.

Sei das Medium nun wiederum eine Streamaufzeichnung. Per Suchstruktur lässt sich ohne nennenswerte Verzögerung an einen beliebigen Aufzeichnungspunkt springen. Außerdem ist ganz nach Bedarf ein Wertezuwachs in beide Richtungen möglich: vergangenheitsgerichtet, zukunftsgerichtet oder beidseitig gerichtet.

Der Begriff endliche Aufzeichnung $\longmapsto\!\!\shortmid$ soll im Folgenden betonen, dass sich die Streamgrenzen zur Laufzeit einer Anfrage explizit erreichen lassen, was die Bildung des letztmöglichen Fensters und somit eine automatische Terminierung bedeutet.

Der Gegenbegriff quasi-unendliche Aufzeichnung $\shortmid\!\!\longmapsto\!\!\shortmid$ meint im Folgenden, dass das Erreichen von Streamgrenzen zur Laufzeit ausgeschlossen ist; hier herrschen also quasi-unendliche Idealbedingungen vor, die erwartungsgemäß bei immensen Datenmengen bzw. einem frühzeitigen Abbruch durch den Nutzer angenommen werden können.

Verarbeitung

Das Zuwachsverhalten der Daten und das Bewegungsverhalten eines Fensters kann im Zusammenspiel harmonisieren oder konfligieren. Im Idealfall ist das Bewegungsverhalten des Fensters durch die Zuwachsrichtung(en) abgedeckt, was eine direkte Verarbeitung in Echtzeit ermöglicht, die potenziell unendlich fortgeführt werden könnte. Andernfalls müsste zwangsläufig auf Aufzeichnungen zurückgegriffen werden.

echtzeit-unendlich: Bei echtzeit-unendlicher Verarbeitung handelt es sich um Echtzeitdaten, die direkt verarbeitet werden und aus denen (durch erwartbares Nichterreichen des Datenstromendes) potenziell unendlich viele Fenster hervorgehen.

echtzeit-endlich: Bei echtzeit-endlicher Verarbeitung handelt es sich um Echtzeitdaten, die direkt verarbeitet werden können und aus denen (durch erwartbares Erreichen des Datenstromendes) endlich viele Fenster hervorgehen.

Quasi-Aufzeichnung: Eine Quasi-Aufzeichnung ist eine zweiteilige Technik, die notwendig wird, falls der Eintragszuwachs und das Bewegungsverhalten in gegenteilige Richtungen verläuft und die Datenerfassung aufgrund Echtzeitcharakter noch aussteht. Konträre Richtungen sind besonders kontraproduktiv, da eine potenziell unendliche Fortführung von vornherein unmöglich ist; erschwerend kommt hinzu, dass dies streng genommen nur retrospektiv mit Aufzeichnungen umsetzbar ist. Falls die Anfrage definiert ist, die Daten aber noch in der Zukunft liegen, muss also zunächst auf deren vollständige Erfassung gewartet werden; erst im Anschluss kann die Fensterbildung beginnen.

Momentaufnahme: Eine Momentaufnahme ist eine Anfrage auf einen Datenstrom, die nur ein einziges Fenster bilden möchten. In diesem Sinne ist es also ein stehendes Fenster – und so gesehen das Warten auf eine in Echtzeit generierte Bereichsanfrage.

Praxis

Die folgenden Tabellen schlüsseln unter Gebrauch der soeben eigens eingeführten Begriffe das Verhalten der wesentlichen Wachstumsumgebungen je Fenstertyp auf. Zur Erleichterung der gedanklichen Vorstellbarkeit der erlaubten bzw. konfligierenden Bewegungsrichtungen wurde statt der üblichen Wort- oder Zahlennotation im Folgenden die richtungsangebende Dreiecksnotation benutzt. Eine begriffliche Komplettübersicht inklusive allen vier synonymen Hop-Entsprechungen ist bereits in Abschnitt 3.5.2 zu finden.

Im Falle der Echtzeitentwicklung $\longmapsto$ gilt:

hop_A	hop_E	Unterteilung	Fensteranzahl	Terminierung	Verarbeitung	Fenstertypänderung
◀	◀	$hop_A<hop_E$	endlich	Fenster oder Stream	verzögert, Quasi-Aufz.	ggf. einseitig schrumpfend
		$hop_A=hop_E$	endlich	Stream	verzögert, Quasi-Aufz.	irgendwann einseitig schrumpfend
		$hop_A>hop_E$	endlich	Stream	verzögert, Quasi-Aufz.	irgendwann einseitig schrumpfend
◀	▽		endlich	Stream	verzögert, Quasi-Aufz.	irgendwann stehend
◀	▶		unendlich	nein	direkt, Echtzeit	irgendwann einseitig
▽	▽		einmalig	Fenster	verzögert, Quasi-Aufz.	keine
▽	◀		endlich	Fenster	verzögert, Quasi-Aufz.	keine
▽	▶		unendlich	nein	direkt, Echtzeit	keine
▶	◀		endlich	Fenster	verzögert, Quasi-Aufz.	keine
▶	▽		endlich	Fenster	verzögert, Quasi-Aufz.	keine
▶	▶	$hop_A<hop_E$	unendlich	nein	direkt, Echtzeit	keine
		$hop_A=hop_E$	unendlich	nein	direkt, Echtzeit	keine
		$hop_A>hop_E$	endlich	Fenster	direkt, Echtzeit	keine

Im Falle von einer (endlichen) Aufzeichnung $\longmapsto\!\!\!\!\mid$ gilt:

hop_A	hop_E	Unterteilung	Fensteranzahl	Terminierung	Verarbeitung	Fenstertypänderung
◀	◀	$hop_A<hop_E$	endlich	Fenster oder Stream	direkt, Echtzeit	ggf. einseitig schrumpfend
		$hop_A=hop_E$	endlich	Stream	direkt, Echtzeit	irgendwann einseitig schrumpfend
		$hop_A>hop_E$	endlich	Stream	direkt, Echtzeit	irgendwann einseitig schrumpfend
◀	▽		endlich	Stream	direkt, Echtzeit	irgendwann stehend
◀	▶		endlich	Stream	direkt, Echtzeit	irgendwann stehend
▽	▽		einmalig	Fenster	direkt, Echtzeit	keine
▽	◀		endlich	Fenster	direkt, Echtzeit	keine
▽	▶		endlich	Stream	direkt, Echtzeit	irgendwann stehend
▶	◀		endlich	Fenster	direkt, Echtzeit	keine
▶	▽		endlich	Fenster	direkt, Echtzeit	keine
▶	▶	$hop_A<hop_E$	endlich	Stream	direkt, Echtzeit	irgendwann einseitig schrumpfend
		$hop_A=hop_E$	endlich	Stream	direkt, Echtzeit	irgendwann einseitig schrumpfend
		$hop_A>hop_E$	endlich	Fenster oder Stream	direkt, Echtzeit	ggf. einseitig schrumpfend

Die Echtzeitumgebung und die Aufzeichnungsumgebung stellen im allgemeinen Sprechen die beiden Praxisumgebungen dar. Beide Tabellen sollen zeigen, wie sehr sie jeweils durch die Endlichkeit der Streamenden limitiert werden. Außerdem soll die Echzeit-Tabelle verdeutlichen, dass die entsprechende Umgebung hinsichtlich Vergangenheitsentwicklungen nahezu komplett beschränkt ist. Im eigentlichen Sinne ist wie gesagt gar keine direkte Vergangenheitsbewegung möglich; es kann höchstens mit entsprechender Verzögerung auf zukünftige Daten antizipierend gewartet werden, um dann schnellstmögliche (und dabei stets endliche) Erkenntnisse liefern zu können.

Im Falle einer quasi-unendlichen Aufzeichnung $\nrightarrow$ gilt:

hop_A	hop_E	Unterteilung	Fensteranzahl	Terminierung	Verarbeitung	Fenstertypänderung
◄	◄	$hop_A<hop_E$	endlich	Fenster	direkt, Aufzeichnung	keine
		$hop_A=hop_E$	quasi-unendlich	nein	direkt, Aufzeichnung	keine
		$hop_A>hop_E$	quasi-unendlich	nein	direkt, Aufzeichnung	keine
◄	▽		quasi-unendlich	nein	direkt, Aufzeichnung	keine
◄	►		quasi-unendlich	nein	direkt, Aufzeichnung	keine
▽	▽		einmalig	Fenster	direkt, Aufzeichnung	keine
▽	◄		endlich	Fenster	direkt, Aufzeichnung	keine
▽	►		quasi-unendlich	nein	direkt, Aufzeichnung	keine
►	◄		endlich	Fenster	direkt, Aufzeichnung	keine
►	▽		endlich	Fenster	direkt, Aufzeichnung	keine
►	►	$hop_A<hop_E$	quasi-unendlich	nein	direkt, Aufzeichnung	keine
		$hop_A=hop_E$	quasi-unendlich	nein	direkt, Aufzeichnung	keine
		$hop_A>hop_E$	endlich	Fenster	direkt, Aufzeichnung	keine

Im Falle der theoretischen Bidirektionalität $\longleftrightarrow$ gilt:

hop_A	hop_E	Unterteilung	Fensteranzahl	Terminierung	Verarbeitung	Fenstertypänderung
◄	◄	$hop_A<hop_E$	endlich	Fenster	direkt, Aufzeichnung	keine
		$hop_A=hop_E$	unendlich	nein	direkt, Aufzeichnung	keine
		$hop_A>hop_E$	unendlich	nein	direkt, Aufzeichnung	keine
◄	▽		unendlich	nein	direkt, Aufzeichnung	keine
◄	►		unendlich	nein	direkt, Aufzeichnung	keine
▽	▽		einmalig	Fenster	direkt, Aufzeichnung	keine
▽	◄		endlich	Fenster	direkt, Aufzeichnung	keine
▽	►		unendlich	nein	direkt, Aufzeichnung	keine
►	◄		endlich	Fenster	direkt, Aufzeichnung	keine
►	▽		endlich	Fenster	direkt, Aufzeichnung	keine
►	►	$hop_A<hop_E$	unendlich	nein	direkt, Aufzeichnung	keine
		$hop_A=hop_E$	unendlich	nein	direkt, Aufzeichnung	keine
		$hop_A>hop_E$	endlich	Fenster	direkt, Aufzeichnung	keine

Die quasi-unendliche Umgebung und die theoretische bidirektionale Umgebung stellen die wünschenswerte Idealumgebungen dar. Der Unterschied besteht lediglich in der begrifflichen Ersetzungen zwischen quasi-unendlich und unendlich, ohne dass sich aus Anwendungssicht etwas ändert. Ersteres ist idealisierte Praxis, Letzteres das rein theoretische Äquivalent. Außerdem zu erkennen: Falls die Fensteranzahl endlich ist bzw. das Bewegungsverhalten terminiert, so liegt dies einzig am bewusst vom Nutzer vorgegebenen Fenstertyp.

4 Übersicht zu Fenstertypen

Dieses Kapitel widmet sich der Beschreibung und Veranschaulichung einer Vielzahl von Fenstertypen. Das Bewegungsverhalten und weitere Kategorien aus Abschnitt 3.5.2 werden dezidiert kombiniert, um einen möglichst weitreichenden Überblick zu schaffen und die selbstgewählten Begrifflichkeiten zu festigen. Natürlich ist es schon aufgrund von Freiheiten bei der Definitionsausrichtung unmöglich, hier Vollständigkeit herzustellen, zumal mehrere Einschränkungen aufgestellt wurden. Dies wäre unter anderem, dass der Fokus auf konstanten Wert liegen soll. Außerdem wird vorausgesetzt, dass die eingeschlagene Bewegungsrichtung konsequent beibehalten wird. Nichtsdestotrotz wird aber ebenso wiederholt die Gelegenheit genutzt, um weitere Ansätze anzudeuten.

Die Beispielbilder dieses Abschnitts wurden übrigens nicht von Grund auf neu erstellt, sondern unter Zuhilfenahme einer Vorlage geschaffen. Die ersten Grafiken dieses Abschnitts basieren nämlich auf Vorlesungsfolien von [Müh22]. Mittels Bildeditierung wurden dann etliche weitere Beispiele erzeugt, die sich nahtlos in jenes Template einfügen. Zudem beschränken sich die Bilder auf eine zukunftsgerichtete Darstellung. Auf eine Unterscheidung zwischen zukunfts- und vergangenheitsgerichtetem Bewegungsverhalten wurde aufgrund von Redundanz und zu geringem Mehrwert verzichtet. Die eingeführten formalen Bedingungen lassen dennoch eine Beschreitung beider Richtungen zu, was zumeist an den Betragsfunktionsstrichen zu erkennen ist.

Konventionen zur Erleichterung

Einige Fenstertypen gestatten sinnvolle Namensvereinfachungen, durch welche eine kontextuelle Loslösung von den beiden Rollen hop_A und hop_E vollzogen werden kann. In solchen Fällen ist es dann unerheblich, ob es sich um die Sprungweite des Fensteranfangs oder der Fensterendes handelt. Viele Namen sind in diesem Abschnitt und der Arbeit selbst kaum von Relevanz. Es geht vielmehr um den pragmatischen Nutzen an sich, sofern andere wissenschaftliche Arbeiten darauf aufzubauen gedenken. Die meisten Vereinfachungen werden in ihrem jeweiligen Unterabschnitt verkündet. Die allerwichtigsten und abschnittsübergreifenden Vereinfachung seien jedoch schon an dieser Stelle eingeführt:

Sofern wertgleiche Sprungweiten vorliegen, also falls $hop_A{=}hop_E$ gilt, ist eine grundsätzliche Verkürzung auf hop möglich.

Der Übersichtlichkeit und Länge wegen wurde aufgrund des vielfachen Gebrauchs darauf verzichtet, die spezifischeren Ausdrücke $hopTuple$ und $hopTime$ zu benutzen. Außerdem ist dadurch die Regelähnlichkeit zwischen Zeit- und Tupelebene besser erkennbar.

Wenn das Umfeld als Tupelfenster benannt ist, sollte daher immer die gedankliche Ersetzung zum noch expliziteren $hop := hopTuple \in \mathbb{Z}$ mitbedacht werden. Zudem gilt wie bereits eingeführt, dass die Fensterkapazität $c \in \mathbb{N}^+$ beträgt. Die Größen hop und c sind hier nutzerseitig vordefinierte Konstanten. Die Zeitspanne d wird zu einer nebensächlichen Größe, die durch jene beeinflusst wird.

Wenn das Umfeld als Zeitfenster benannt ist, sollte daher immer die gedankliche Ersetzung zum noch expliziteren $hop := hopTime \in \mathbb{R}$ mitbedacht werden. Zudem gilt wie bereits

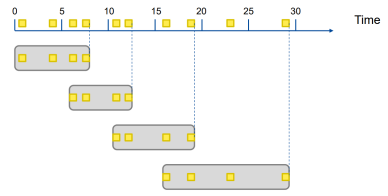
eingeführt, dass die Zeitspanne $d \in \mathbb{R}^+$ beträgt. Die Größen *hop* und d sind hier nutzerseitig vordefinierte Konstanten. Die Fensterkapazität c wird zu einer nebensächlichen Größe, die durch jene beeinflusst wird.

Abkürzung – **i.d.A.:** innerhalb dieser Arbeit. Dies soll kenntlich machen, dass es sich um eine enger gefasste Festlegung handelt, wahlweise aus Pragmatik oder weil es dem häufig gewählten Regelfall entspricht. Fernab dieser Arbeit könnten jedoch auch freiere Definitionen möglich sein.

4.1 Jumping-Fenster

Jumping-Fenster sind Fenster fester Größe, die sich (gleichbleibend) in eine bestimmte Richtung fortbewegen. Innerhalb dieser Arbeit gilt verschärfend, dass benachbarte Folgefenster überlappen oder partitionierend an einander angrenzen.

Jumping-Tupelfenster

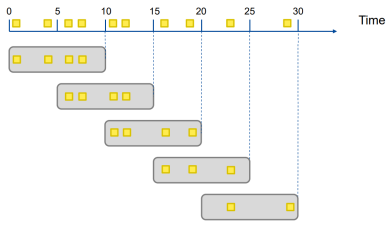


c=4 Einträge, hop=2 Einträge

Fensterkapazität	konstant
Zeitspanne	variabel
Aktualisierungsverhalten	eintragszyklisch
Überschneidung	ungewiss, tendenziell überlappend
Wachstum	nichtmonoton
Eintragsberücksichtigung	[1, n]
Vorbedingungen	$hop_A{=}hop_E$ $0 < \lvert hop \rvert \leq c$

Spezialfälle: $tumblingTuple$, $slidingTuple \subset jumpingTuple$

Jumping-Zeitfenster



d=10 s, hop=5 s

Fensterkapazität	variabel
Zeitspanne	konstant
Aktualisierungsverhalten	zeitzyklisch
Überschneidung	ungewiss, tendenziell überlappend
Wachstum	nichtmonoton
Eintragsberücksichtigung	[1, n]
Vorbedingungen	$hop_A{=}hop_E$ $0 < \lvert hop \rvert \leq d$

Spezialfälle: $tumblingTime \subset jumpingTime$ $(slidingTime \not\subset jumpingTime)$

Sliding-Zeitfenster stellen im Gegensatz zu Sliding-Tupelfenstern keine Spezialform eines Jumpings dar. Eine Erklärung folgt im spezifischen Abschnitt 4.3.

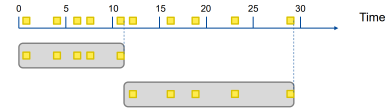
Anmerkung:

Eine tolerantere Jumping-Fenster-Definition wäre auch über die Formel $hop_A{=}hop_E \wedge 0{<}|hop|$ beschreibbar. Im Rahmen dieser Arbeit wurde jedoch noch zusätzlich zwischen $|hop| \leq windowSize$ und $|hop| > windowSize$ unterschieden. Letztere Unterform findet sich in Abschnitt 4.4 und wird im Folgenden Session-Jumping-Fenster genannt.

4.2 Tumbling-Fenster

Tumbling-Fenster sind Fenster fester Größe, die sich nachbarschaftlich disjunkt in eine bestimmte Richtung fortbewegen.

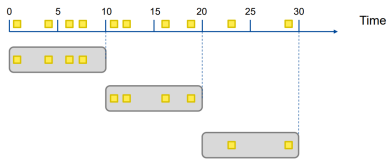
Tumbling-Tupelfenster



hop=5 Einträge

Fensterkapazität	konstant
Zeitspanne	variabel
Aktualisierungsverhalten	eintragszyklisch
Überschneidung	disjunkt
Wachstum	nichtmonoton
Eintragsberücksichtigung	[1, 1]
Vorbedingungen	$hop_A{=}hop_E$ $0 < \|hop\| = c$

Tumbling-Zeitfenster



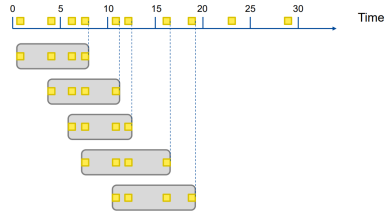
hop=10 s

Fensterkapazität	variabel
Zeitspanne	konstant
Aktualisierungsverhalten	zeitzyklisch
Überschneidung	disjunkt
Wachstum	nichtmonoton
Eintragsberücksichtigung	[1, 1]
Vorbedingungen	$hop_A{=}hop_E$ $0 < \|hop\| = d$

4.3 Sliding-Fenster

Sliding-Fenster sind Fenster fester Größe, die sich eintragsweise in eine bestimmte Richtung fortbewegen.

Sliding-Tupelfenster

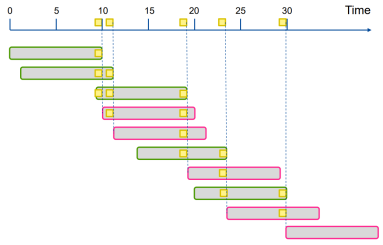


c=4 Einträge

Fensterkapazität	konstant
Zeitspanne	variabel
Aktualisierungsverhalten	eintragszyklisch
Überschneidung	überlappend
Wachstum	nichtmonoton
Eintragsberücksichtigung	$[1, n] \rightarrow c$
Vorbedingungen	$hop_A{=}hop_E$ $\lvert hop\rvert = 1$

Sliding-Zeitfenster

Sliding-Zeitfenster heben sich von allen anderen Zeitfenstern im Besonderen durch ihre kontraintuitive Funktionsweise ab. Während sich einzelne Eintragsveränderungen bei Tupelfenstern gezielt erfassen lassen, widerstrebt es intuitiv einem zeitorientieren Ansatz, da es sich um unbestimmte Ereignisse handelt. Die Fensterentwicklung wandelt sich in diesem Falle daher vielmehr zu einer ereignisgetriebenen. Jeder neue Eintrag löst während seiner Berücksichtigung zwei Fenster aus: Eines bei dessen Eintritt (grün), wo er bei der Auswertung berücksichtigt wird, und eines bei dessen Austritt (pink), wo er nicht mehr berücksichtigt wird. Dies hat unter anderem zur Folge, dass selbst ausgedehnte eintragslose Phasen zu einzelnen leeren Fenstern komprimiert werden.



d=10 s

Fensterkapazität	variabel
Zeitspanne	konstant
Aktualisierungsverhalten	zeitvariabel (ereignisgetrieben)
Überschneidung	überlappend
Wachstum	nichtmonoton
Eintragsberücksichtigung	[1, n]
Vorbedingungen	$hop_A{=}hop_E$ $\lvert hop\rvert \in \mathbb{R}^+$

Nicht-Spezialfall: $slidingTime \not\subset jumpingTime$ $(slidingTuple \subset jumpingTuple)$

Aus dem ereignisgetriebenen Fensterverhalten folgt, dass die Hops eine nicht vorhersagbare dynamische Größe besitzen. Ein solches Verhalten lässt sich nutzerseitig nahezu unmöglich über einen konstanten, geschweige denn funktionalen Hop, beschreiben. Dementsprechend ist es ausgeschlossen, dass es sich um den Spezialfall eines Jumping-Zeitfenster handelt.

4.4 Session-Fenster

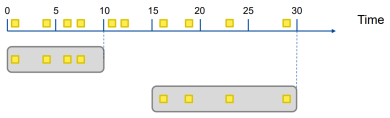
Session-Fenster sind gleichgerichtete disjunkte Fenster, deren (hinterer) Sprung stets größer als die (gegenwärtige) Fenstergröße ist. Dadurch gibt es potenziell Einträge, die in keinem einzigen Fenster Berücksichtigung finden. Dies ist natürlich eine recht weit gefasste Definition, die auch sich dynamisch ändernde Hops und Fenstergrößen gestattet, solange stets

der nötige Mindestabstand zueinander durch ständige Neuanpassungen gewahrt wird. Innerhalb dieser Arbeit wird dagegen die enger gefasste Definition genutzt. Im Folgenden gelten die verschärfenden Zusatzbedingungen, dass die Fenstergröße konstant festgelegt ist und Wertgleichheit zwischen den beiden Hop-Enden herrscht. Auf diese Weise soll eine schärfere Unterscheidung von Jumping-Fenstern erzielt werden.

Das Session-Konzept lässt sich beispielsweise sehr gut auf Mikroblogging- oder Feedbackdiensten übertragen. Ein Musterbeispiel wäre das Analysieren von Diskussionsfäden und Gesprächstrends. Über eine ignorierende Mindestwartezeit könnten impulsive und automatisierte Erstantworten von vornherein ausgesiebt werden. Außerdem könnte man mittels dieser Technik die Analyselast reduzieren. Anstatt das gesamte Nachrichtenaufkommen permanent zu untersuchen, könnte es bereits völlig ausreichen, wenn man alle zehn Minuten die letzten zwei analysiert. Dies wäre im Besonderen bei zähen bzw. nicht allzu sprunghaften Entwicklungen effektiv, wo eine Stichprobe schon nahezu die gleiche Aussagekraft wie eine Daueranalyse liefern könnte.

vereinfachende Konvention: Da stets eine Mindestsprungweite $|hop| > windowSize$ vorliegen muss, könnte der Hop auch äquivalent als $timeout$-Hop ausgedrückt werden, was der Form $timeout := hop - sgn(hop) \cdot windowSize$ entsprechen würde. Timeout ist hier im generellen Sinne als Pause, Unterbrechung bzw. inaktive Phase ohne zwingenden Bezug zur physikalischen Größe Zeit zu verstehen.

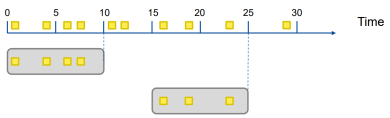
Session-Tupelfenster (streng genommen: Session-Jumping-Tupelfenster)



c=4 Einträge,
hop=6 → timeout=2 Einträge

Fensterkapazität	konstant
Zeitspanne	variabel
Aktualisierungsverhalten	i.d.A. eintragszyklisch
Überschneidung	disjunkt
Wachstum	nichtmonoton
Eintragsberücksichtigung	[0, 1]
Vorbedingungen	$hop_A{=}hop_E$ $\|hop\| > c > 0$ $timeout := hop - \text{sgn}(hop) \cdot c$

Session-Zeitfenster (streng genommen: Session-Jumping-Zeitfenster)



d=10 s,
hop=15 s → timeout=5 s

Fensterkapazität	variabel
Zeitspanne	konstant
Aktualisierungsverhalten	i.d.A. zeitzyklisch
Überschneidung	disjunkt
Wachstum	nichtmonoton
Eintragsberücksichtigung	[0, 1]
Vorbedingungen	$hop_A{=}hop_E$ $\|hop\| > d > 0$ $timeout := hop - \text{sgn}(hop) \cdot d$

Sonderformen

Durch das ständige Abwechseln berücksichtigter und ignorierter Intervalle eignen sich Session-Fenster auch zur Spezifikation weiterer Sonderformen. Als Erweiterung könnte ein Wechsel zwischen Zeit- und Tupelverhalten denkbar sein. Zum einen ließen sich Tupelfenster mit zeitlichem Timeout definieren, zum anderen Zeitfenster mit Tupel-Timeout.

Darüber hinaus ließen sich auch Session-Zeitfenster mit variabler Länge definieren. Sie würden entlang eines zeitlichen Schwellenwerts *threshold* gebildet werden. Ein unfertiges neues Fenster würde stets weitere Einträge aufnehmen, solange die aufeinanderfolgenden Zeitabstände innerhalb des Schwellenwertes liegen. Erst wenn eine Überschreitung erfolgt, wäre die Fensterbildung abgeschlossen und es würde mit dem nächsten Fenster begonnen werden würden. Ein solcher Typ setzt eine grobe Kenntnis des Datenstroms voraus und eignet sich nicht jede alle Abfragen, da lange bis ewige Sammelphasen auf ausreichend dichten Streams möglicherweise nicht gewollt sind.

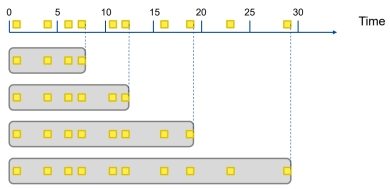
4.5 Einseitige Landmark-Fenster

Einseitige Landmark-Fenster sind monotone Fenster mit einem stehenden und einem beweglichen Hop, die sich einseitig in eine bestimmte Richtung fortentwickeln. Dies kann wahlweise in Form von Wachstum und Schrumpfung geschehen.

vereinfachende Konvention: Da einer der beiden Hops den stehenden Wert 0 annimmt, kann der andere verkürzend *hop* genannt werden, falls bloß der Sprungwert an sich referiert wird.

$$hop = \begin{cases} hop_A & falls \quad hop_E{=}0 \\ hop_E & falls \quad hop_A{=}0 \end{cases}$$

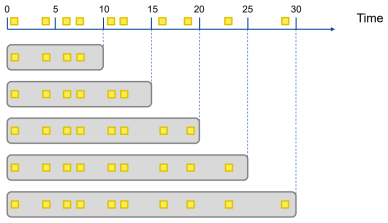
Einseitig wachsendes Landmark-Tupelfenster



hop=2 Einträge, c-start=4 Einträge

Fensterkapazität	variabel, i.d.A. konstant zunehmend
Zeitspanne	variabel, zunehmend
Aktualisierungsverhalten	i.d.A. eintragszyklisch
Überschneidung	überlappend
Wachstum	monoton wachsend
Eintragsberücksichtigung	$\rightarrow \infty$
Vorbedingungen	$hop_A{=}0, hop_E{>}0 \vee hop_A{<}0, hop_E{=}0$
häufiger Defaultwert	$c_{start}{=}0$

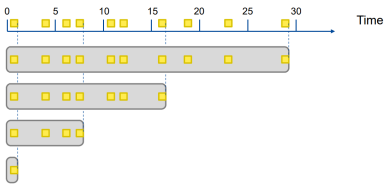
Einseitig wachsendes Landmark-Zeitfenster



hop=5 s, d-start=10 s

Fensterkapazität	variabel, zunehmend
Zeitspanne	variabel, i.d.A. konstant zunehmend
Aktualisierungsverhalten	i.d.A. zeitzyklisch
Überschneidung	überlappend
Wachstum	monoton wachsend
Eintragsberücksichtigung	$\rightarrow \infty$
Vorbedingungen	hop_A=0, hop_E>0 $\vee$ hop_A<0, hop_E=0
häufiger Defaultwert	d_{start}=0

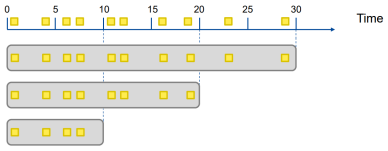
Einseitig schrumpfendes Landmark-Tupelfenster



hopA=0, hopE=−3, c-start=10 Einträge

Fensterkapazität	variabel, i.d.A. konstant abnehmend
Zeitspanne	variabel, abnehmend
Aktualisierungsverhalten	i.d.A. eintragszyklisch
Überschneidung	überlappend
Wachstum	monoton schrumpfend
Eintragsberücksichtigung	[1, n]
Vorbedingungen	hop_A=0, hop_E<0 $\vee$ hop_A>0, hop_E=0 c_{start}>0
angestrebte Verhältnisse	$c_{start} \gg \lvert hop \rvert$

Einseitig schrumpfendes Landmark-Zeitfenster



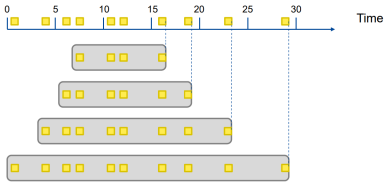
hopA=0 s, hopE=−10 s, d-start=30 s

Fensterkapazität	variabel, abnehmend
Zeitspanne	variabel, i.d.A. konstant abnehmend
Aktualisierungsverhalten	i.d.A. zeitzyklisch
Überschneidung	überlappend
Wachstum	monoton schrumpfend
Eintragsberücksichtigung	[1, n]
Vorbedingungen	hop_A=0, hop_E<0 $\vee$ hop_A>0, hop_E=0 d_{start}>0
angestrebte Verhältnisse	$d_{start} \gg \lvert hop \rvert$

4.6 Beidseitige Landmark-Fenster

Beidseitige Landmark-Fenster sind monotone Fenster mit beweglichen Hops, die sich beidseitig in gegensätzliche Richtungen fortentwickeln. Dies kann wahlweise in Form von Wachstum und Schrumpfung geschehen.

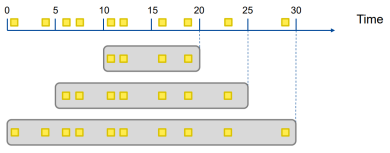
Beidseitig wachsendes Landmark-Tupelfenster



hopA=−1, hopE=1, c-start=4 Einträge

Fensterkapazität	variabel, i.d.A. konstant zunehmend
Zeitspanne	variabel, zunehmend
Aktualisierungsverhalten	i.d.A. eintragszyklisch
Überschneidung	überlappend
Wachstum	monoton wachsend
Eintragsberücksichtigung	$\rightarrow \infty$
Vorbedingungen	hop_A<0, hop_E>0
häufiger Defaultwert	c_{start}=0

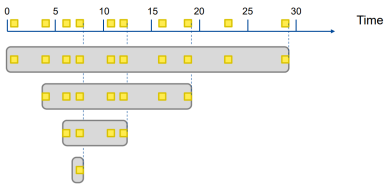
Beidseitig wachsendes Landmark-Zeitfenster



hopA=−5 s, hopE=5 s, d-start=10 s

Fensterkapazität	variabel, zunehmend
Zeitspanne	variabel, i.d.A. konstant zunehmend
Aktualisierungsverhalten	i.d.A. eintragszyklisch
Überschneidung	überlappend
Wachstum	monoton wachsend
Eintragsberücksichtigung	$\rightarrow \infty$
Vorbedingungen	hop_A<0, hop_E>0
häufiger Defaultwert	d_{start}=0

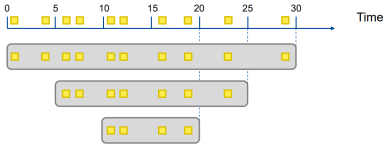
Beidseitig schrumpfendes Landmark-Tupelfenster



hopA=1, hopE=−2, c-start=10 Einträge

Fensterkapazität	variabel, i.d.A. konstant abnehmend
Zeitspanne	variabel, abnehmend
Aktualisierungsverhalten	i.d.A. eintragszyklisch
Überschneidung	überlappend
Wachstum	monoton schrumpfend
Eintragsberücksichtigung	[1, n]
Vorbedingungen	hop_A>0, hop_E<0
	c_{start}>0
angestrebte Verhältnisse	$c_{start} \gg \lvert hop \rvert$

Beidseitig schrumpfendes Landmark-Zeitfenster



hopA=5 s, hopE=−5 s, d-start=30 s

Fensterkapazität	variabel, abnehmend
Zeitspanne	variabel, i.d.A. konstant abnehmend
Aktualisierungsverhalten	i.d.A. zeitzyklisch
Überschneidung	überlappend
Wachstum	monoton schrumpfend
Eintragsberücksichtigung	[1, n]
Vorbedingungen	hop_A>0, hop_E<0
	d_{start}>0
angestrebte Verhältnisse	$d_{start} \gg \lvert hop \rvert$

Vereinfachende Konventionen

beidseitig wachsend: Für beidseitig wachsende Fenster wäre aus pragmatischer Sicht eine Umbenennung auf $hop_{past} := hop_A$, $hop_{future} := hop_E$ möglich.

beidseitig schrumpfend: Aufgrund des Schrumpfens kann die Verwendung derselben Notation hier mitunter kontraproduktiv sein bzw. missverständlich wirken. Im Sinne des Hops müsste es die Bewegungsrichtung repräsentieren, was dann $hop_{future} := hop_A$, $hop_{past} := hop_E$ entsprechen würde. Andererseits könnte die jeweilige Rolle auch positionsabhängig gedeutet werden. Das Vergangenheitsende betreibt nämlich Fensterabbau in die Zukunft und das Zukunftsende baut wiederum vergangenheitsgerichtet ab. Ohne eine explizite Erklärung wäre also davon abzuraten.

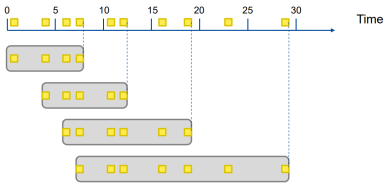
4.7 Fenster mit relativer Größenentwicklung

Fenster mit relativer Größenentwicklung (kurz: relative Fenster) sind Dies kann wahlweise in Form von Wachstum und Schrumpfung geschehen.

Vereinfachende Konvention

$$hop_{slow} = \begin{cases} hop_A & falls \quad |hop_A| < |hop_E| \\ hop_E & falls \quad |hop_E| < |hop_A| \end{cases} \qquad hop_{fast} = \begin{cases} hop_A & falls \quad |hop_A| > |hop_E| \\ hop_E & falls \quad |hop_E| > |hop_A| \end{cases}$$

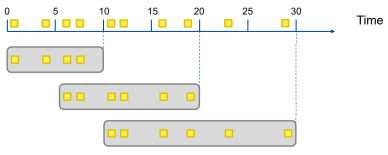
Relativ wachsendes Tupelfenster



hopA=1, hopE=2, c-start=4 Einträge

Fensterkapazität	variabel, i.d.A. konstant zunehmend
Zeitspanne	variabel, tendenziell zunehmend
Aktualisierungsverhalten	i.d.A. eintragszyklisch
Überschneidung	ungewiss, tendenziell überlappend
Wachstum	nichtmonoton
Eintragsberücksichtigung	[0, n]
Vorbedingungen	$0<hop_A<hop_E \vee hop_A<hop_E<0$
häufiger Defaultwert	$c_{start}=0$

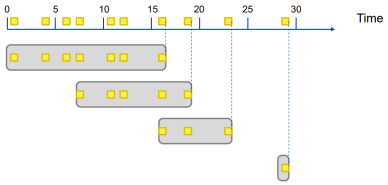
Relativ wachsendes Zeitfenster



hopA=5 s, hopE=10 s, d-start=10 s

Fensterkapazität	variabel, tendenziell zunehmend
Zeitspanne	variabel, i.d.A. konstant zunehmend
Aktualisierungsverhalten	i.d.A. zeitzyklisch
Überschneidung	ungewiss, tendenziell überlappend
Wachstum	nichtmonoton
Eintragsberücksichtigung	[0, n]
Vorbedingungen	$0<hop_A<hop_E \vee hop_A<hop_E<0$
häufiger Defaultwert	$d_{start}=0$

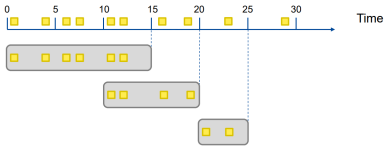
Relativ schrumpfendes Tupelfenster



hopA=3, hopE=1, c-start=7 Einträge

Fensterkapazität	variabel, i.d.A. konstant abnehmend
Zeitspanne	variabel, tendenziell abnehmend
Aktualisierungsverhalten	i.d.A. eintragszyklisch
Überschneidung	ungewiss, tendenziell überlappend
Wachstum	nichtmonoton
Eintragsberücksichtigung	[0, n]
Vorbedingungen	$0<hop_E<hop_A \vee hop_E<hop_A<0$ $c_{start}>0$
angestrebte Verhältnisse	$c_{start} \gg \lvert hop \rvert$

Relativ schrumpfendes Zeitfenster



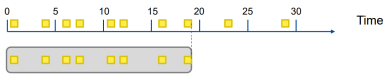
hopA=10 s, hopE=5 s, d-start=15 s

Fensterkapazität	variabel, tendenziell abnehmend
Zeitspanne	variabel, i.d.A. konstant abnehmend
Aktualisierungsverhalten	i.d.A. zeitzyklisch
Überschneidung	ungewiss, tendenziell überlappend
Wachstum	nichtmonoton
Eintragsberücksichtigung	[0, n]
Vorbedingungen	$0<hop_E<hop_A \vee hop_E<hop_A<0$ $d_{start}>0$
angestrebte Verhältnisse	$d_{start} \gg \lvert hop \rvert$

4.8 Stehende Fenster

Stehende Fenster sind im Folgenden Einzelfenster mit stehenden Hops, welche keine weitere Entwicklung vollziehen. Sie gleichen so gesehen normalen Bereichsanfragen. Hiervon nochmals deutlich abzugrenzen sind die stehenden Prädikatfenster.

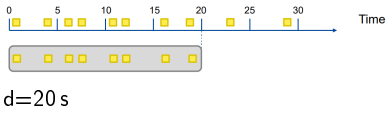
Stehendes Tupelfenster



c=8 Einträge

Fensterkapazität	konstant
Zeitspanne	konstant (aufgrund Einzelfenster)
Aktualisierungsverhalten	keines
Überschneidung	keine
Wachstum	keines
Eintragsberücksichtigung	[1, 1]
Vorbedingungen	hop_A=0, hop_E=0

Stehendes Zeitfenster



Fensterkapazität	konstant (aufgrund Einzelfenster)
Zeitspanne	konstant
Aktualisierungsverhalten	keines
Fensterentwicklung	keine
Eintragsberücksichtigung	[1, 1]
Vorbedingungen	hop_A=0, hop_E=0

4.9 Dynamische Sonderformen

Innerhalb dieser Arbeit bewegen sich Fenstervariablen bekanntlich im konstanten Spektrum. Gleichwohl könnten Größen wie die Sprungweite oder die Fenstergröße auch deutlich dynamischer angelegt werden. Denkbar wären beispielsweise: (1) periodische Wertmuster; (2) funktionale Wertentwicklungen; (3) randomisierte Wertentwicklungen mit ein- oder beidseitig beschränkten Intervallgrenzen; (4) randomisierte Wertentwicklungen mit Erwartungswert und Varianz (5) im Extremfall sogar vollkommene Randomisierung.

Mittels solcher Techniken ließe sich beispielsweise das Bewegungsverhalten von Landmarks dynamisieren. Darüber hinaus könnte man damit die Hops von sich relativ entwickelnden Fenstern so abstimmen, dass sie tumbling-ähnlich partitionieren. Fernab gäbe es außerdem noch die Möglichkeit, dass man für die Sprünge auf verschiedenen Wertebereiche zugleich fortbewegt. Einer der beiden Hops könnte tupelorientiert springen, der andere dagegen zeitorientiert. Und selbst dieses Verhalten muss nicht dauerhaft festgeschrieben sein.

5 Problemklassen

Die Kombination von Anonymisierungsoperatoren mit Fensterverarbeitung kann aus Privacy-Sicht mehrere neue Risiken hervorrufen, die es nach Möglichkeit zu vermeiden gilt. Die neuen Problemarten, die sich daraus ergeben könnten, werden im Folgenden antizipierend ergründet, wobei von den (im Anschluss) vorgestellten minimalistischen ‚naiven' Operatoransätzen ausgegangen wird. Darauffolgend wird untersucht, mit welcher Anfälligkeit bestimmte Fenstertypen davon betroffen wären, ob sich besondere Eigenschaften abzeichnen und wie eine Problemlösung aussehen könnte. Danach werden diese Erkenntnisse genutzt, um erweiterte Operatoren abzuleiten, die die theoretischen Minimaloperatoren nicht nur allgemein verfeinern, sondern auch über elementare Zusatzfunktionen zur Stromdatentauglichkeit verfügen.

Anonymisierungsoperator	P1	P2	P3
Differential Privacy: α^D	ja	nein	nein
Generalisierung: α^G	ja	ja	ja
Slicing: α^S	ja	ja	ja

Abbildung 10: vorliegende Probleme je operationalisierter Anonymisierungstechnik

Problemfall P1: gedächtnisloses Anonymisieren desselben Eintrags

Fensterüberlappung eröffnet potenziellen Angreifern die Möglichkeit, eine Vielzahl von Informationen abzuleiten, sobald ein Eintrag in mehreren Fenstern vorkommt.

Problemfall P1-D: gedächtnisloses Mehrfachverrauschung desselben Eintrags

Falls derselbe Eintrag aus einem Strom aufgrund von Fensterüberlappung in einer Vielzahl von Fenstern vorkommt, ergibt sich daraus ein zusätzliches Einfallstor für potenzielle Angreifer.

Man stelle sich nun ein bescheiden großes Tupelfenster mit Kapazität $k = 10$ und einer Verrauschungswahrscheinlichkeit von $p = 0.3$ für irgendein schützenswertes Attribut vor. Ein bestimmter Eintrag würde dann unterschiedlich positioniert in zehn aufeinanderfolgenden Sliding-Fenstern auftauchen, bis er nicht mehr relevant ist. Falls alle Fenster naiv verraucht werden würden, würde zehnmal per Münzwurf entschieden, ob der Ursprungswert mit 70-prozentiger Wahrscheinlichkeit beibehalten wird – und andernfalls zufällig verrauscht. In einem erwartungswertgetreuen Idealfall käme nun siebenmal der beibehaltene Originalwert A vor und je einmal die Verrauschungswerte B, C und D.

Das Problematische daran sind nun zwei Dinge: Zum einen ergibt sich ein statistisches Ungleichgewicht – ein Angreifer wird bei einer fensterübergreifenden Häufigkeitszählung am häufigsten das Wiederauftreten unveränderter Werte beobachten können. Gelegentlich abgeänderte Werte stechen dann wegen ihrer Seltenheit sofort heraus und können leicht aussortiert werden. Zum anderen wird dies obendrein dadurch erleichtert, dass sich der Alternativwert im Falle einer tatsächlichen Abänderung zumeist nicht funktionell bestimmt,

sondern obendrein dem Zufall obliegt – bei ausreichender Wertebereichvarianz tauchen verrauschte Einträge dann sogar nahezu singulär auf.

Die Schlussfolgerung daraus lautet, dass es bereits ab geringer Kapazität nahezu zwingend ist, nicht gedächtnislos zu verrauschen. Wenn ein Eintrag in mehreren Fenstern auftritt, sollte die schützende DP-Manipulation nach erster Ausführung konsequent erinnert bzw. beibehalten werden, falls diese weitergereicht bzw. öffentlich gemacht wird.

Problemfall P1-S: gedächtnisloses Mehrfach-Slicen desselben Eintrags

Beim Slicing ließen sich aufrechterhaltene Attributbeziehungen (Korrelationsmengen) bei seltenen bis einzigartigen Werten bereits über wenige Fenstervergleiche erkennen, da die Chance auf wiederholtes Zusammentreffen nach der Permutation immer unwahrscheinlicher wird. Ein einziger signifikanter Eintrag könnte bereits genügen, um dies für einige Attribute nach nur einem Fenstervergleich, also zweifachen Eintragsvorkommen, mit Sicherheit sagen zu können. Der Ausnahmefall wäre der zufällige Erhalt aller Einträge, was statistisch jedoch gegen Null geht und den Anonymisierungsbestrebungen ohnehin zuwiderläuft. In manchen Fällen können die erhaltenen Attributbeziehungen öffentlich bekannt sein, vorzugsweise aber ebenso nichtöffentliches Wissen darstellen, das für Dritte unzugänglich bleiben sollte. Das rückwirkende Ermitteln der Korrelationsmengen über Fenstervergleiche ist im Übrigen nochmals scharf von der rückwirkenden Ermittelung der ursprünglichen Attributbeziehungen zu unterscheiden. Da der zweite Aspekt weitaus komplexer ist, sei an dieser Stelle schlicht auf den später folgenden Abschnitt 7.2 mit Hauptaugenmerk auf die Stufen 2 bis 4 verwiesen, wo jener Sachverhalt ausführlich untersucht wurde.

Weiterhin lässt sich über seltene bis einzigartige Werte vermuten, inwieweit geslicte Attributwerte, die vom selben Eintrag stammen könnten, maximal auseinanderliegen. Durch Beobachtung aufeinanderfolgender Fenster ließe sich die Größe der Slicing-Äquivalenzklasse sowie die konkreten Splitstellen ermitteln. Dies böte dann ein Einfallstor für all jene entanonymisierenden Angriffe, die typischerweise bei k-Anonymität möglich sind – also die Eingrenzung auf potenzielle Werteverteilungen, dadurch Homogenitätsattacken, et cetera.

Problemfall P1-G: gedächtnisloses Mehrfach-Generalisieren desselben Eintrags

Beim Generalisieren eines Eintrags stünden fensterbedingt unterschiedliche Wertepools zur Verfügung. Daher wäre es möglich, dass die zu generalisierenden Werten und deren Informationsgehalt bzw. Granularität gleich bleiben, sich verbessern oder verschlechtern könnten. Für sich betrachtet ist dies zunächst nichts Schlechtes – eine Verbesserung wäre ja sogar begrüßenswert. Aufgrund der Wandelbarkeit der Generalisierungswerte könnte dies jedoch schwankende bis inkonsequente Ergebnismengen nach sich ziehen, falls noch höherliegende Operatoren wie Joins, Selektionen, Aggregate etc. im Anschluss folgen sollten.

Ein schwerwiegendes Bedrohungspotential ergibt sich, falls die Generalisierung dynamisch verfährt bzw. mehrere Hierarchien adaptiv zur Auswahl stünden. Man stelle sich eine Zusammenfassung zu Bereichsintervallen vor: einmal zu $[0-45]$, einmal zu $[42-100]$; hier ließe sich der Individualwert in erheblichem Maße auf $[42-45]$ eingrenzen. Oder man stelle sich unterdrückte Zeichenketten vor: einmal verkürzt zu „Rob. O.“, einmal zu „Robert ****“ und

einmal zu „**** Oppenheimer“; hieraus ließe sich „Robert Oppenheimer“ rekonstruieren. Und ebenso ginge es auf einer noch abstrakteren Inhaltsebene bei Taxonomien: bei „Irland (als Insel)“ und „United Kingdom“ könnte man eingrenzend auf „Nordirland“ schließen. Es würde jedenfalls auf lange Sicht zusätzliches Wissen offenbart werden, wodurch man intersektionell gefolgert immer näher an die Ursprungswerte herankommen könnte. Dabei sind es dann sensitive und sonstige Attribute, die spezifische Wiederherstellungen rückwirkend ermöglichen würden. Besonders akut wäre die Gefahr bei einer langen Verweildauer der Ausgangsdaten, also großen Slidings oder gar Landmarks.

Die Schlussfolgerungen daraus: Es ist definitiv zwingend, während der Fensterlebensdauer eines Ausgangswertes an einer einzigen Generalisierungshierachrie festzuhalten. Eine vielfältige Auswahl an Hierarchien ist zur Erzielung der bestmöglichen Granularität gestattet, ein Wechsel nach erstmaliger Gruppenbildung dann allerdings untersagt. Bei Nichtmonotonie darf vom erstmaligen Generalisierungswert über die gesamte Lebensdauer des Ursprungseintrags nicht mehr abgewichen werden. Bei Monotonie ist es durchaus zulässig, den Informationsgehalt entlang der Starthierarchie in Zukunft zu verfeinern.

Lösung von P1: *skip*-Parameter

Eine weitreichende, einfache Lösung für die meisten Fenstertypen wäre die Einführung eines zusätzlichen *skip*-Attributes, mit dem sich eine ausgewählte Anzahl an Einträgen unverändert übernehmen ließe. Unter der Annahme, dass ein bestimmter bereits in einem früheren Fenster berücksichtigt wurde, könnte so das erstmalige Anonymisierungsergebnis konsequent beibehalten werden. Ein Anonymisierungsoperator würde diesen abgezählten Bereich dann überspringen und erst danach ansetzen. Auf direktem Wege ließe sich der Bereich über den Zeitstempel abstecken und auf indirektem Wege auch im Falle einer aufeinanderfolgenden abgezählten Anzahl von Tupeln. Denkbare Konventionen wären u. a.:

(1) eine Zeitstempel-Bereichsnotation $skip = [t_1, t_2]$ bzw. als Positionsindex $skip = [n, m]$

(2) bei vordefiniertem Start t_1 auch in verkürzter Form als $skip = t_2$ (oder umgekehrt bei vordefiniertem Endwert)

(3) neben einer positiven Abzähl-Notation $skip = n$ (beispielsweise zum Auswählen der ersten n Einträge) ist auch eine negative Abzähl-Notation $skip = -n$ (etwa zum Ausblenden der letzten n Einträge) möglich; ob jeweils am Anfang oder Ende begonnen wird, ist anwendungsabhängig zusätzlich festzulegen

Ganz nebenbei sind Notationen wie (2) besonders praktisch, weil sich so kollektiv und auf unkomplizierte Weise die Splitstellen mitverschieben lassen.

Problemfall P2: unzureichende Fenstergröße

Ein zweites grundlegendes Problem ergibt sich beim naiven Slicing- und Generalisierungsoperator parameterbedingt, wenn die Fensterkapazität c die schutzbietende Mindestgröße a unterschreiten sollte. Der einfache Verrauschungsoperator ist hiervon nicht betroffen, da er aufgrund seines zeilenweisen Arbeitens horizontal komplett unabhängig ist. Maßgeblichen

Einfluss nimmt der Wertebereich des Fensters. Bei Tupelfenstern sind die Größen von vornherein bekannt und eventuelle Änderungen absehbar, bei Zeitfenstern dagegen nicht, sodass man zwischen drei Fällen unterscheiden kann:

(A) Wenn bei Tupelfenstern die Fensterkapazität der schützenden Mindestgröße entspricht oder größer als Letztere ist, also $c \geq a$ gilt, besteht das Problem nicht.

(B) Wenn bei Tupelfenstern die Fensterkapazitätt die schützenden Mindestgröße unterschreitet, also $c < a$ gilt, kann nicht die geforderte Privacy gewährleistet werden. Dies würde die leere Ergebnismenge bzw. Ergebnisunterdrückung erzwingen. Bei gleichbleibender Fenstergröße hieße das sogar, dass Ergebnisse grundsätzlich ausbleiben müssten.

(C) Das Problem von Zeitfenstern besteht in ihrer variablen, regulär nicht abschätzbaren Tupelanzahl. Dies hat zum einen ein kaum bis gar nicht abschätzbares Entwicklungsverhalten zur Folge, dass immerzu zwischen (A) und (B) wechseln könnte, und zum anderen, dass die aktuelle Fensterkapazität erst zur Laufzeit bekannt ist. In Einzelfällen ließe sich über eine erwartbare Mindest- und Höchstanzahl zumindest Klarheit schaffen, aber auch das würde keine Lösung des eigentlichen Problems darstellen.

Lösung von P2.B

Die Lösung von (B) besteht im Ansparen einer zusätzlichen Anzahl an Einträgen, um die gewünschte schützende Mindestgruppengröße gewährleisten zu können. Um wiederum dann der gewünschten Fensterkapazität zu entsprechen, werden die internen zusätzlichen Einträge wieder ausgeblendet. Praktisch hieße das für ein reines Einzelfenster (ohne Mitbetrachtung anderer Probleme), dass bei jedem unterschreitenden c die faktische Arbeitskapazität intern auf a ausgeweitet und der $a-c$ große Zusatz abschließend wieder wegselektiert werden muss.

Dieser Ansatz setzt dabei zwei Kompromisse voraus: Zum einen wird es als vertretbar angesehen, sich mit einem größeren Oberfenster zu behelfen, was für den Nutzer sowohl transparent als auch intransparent ablaufen kann. Eine zeitliche Verzögerung und höherer Speicherplatz wird aufgrund der nötigen Sammlung zusätzlicher Einträge bereitwillig in Kauf genommen. Zum anderen billigt man eine gewisse zeitliche Pervertierung, die durch vorweggenommes Wissen ergebnisbezogen auftreten könnte.

Im Besonderen könnte sich der Mechanismus negativ auf das Slicing auswirken. Es werden dort nämlich behelfsmäßige Zusatzwerte ins Fenster hineingeslict, während zugleich im Fenster befindliche dadurch hinausgedrängt werden. Einerseits werden also zeitnahe Zukunftswerte eingeflochten, die streng genommen noch gar nicht vorgekommen wären, und relevante Werte wiederum verdrängt, was Anfragen mit strengem Zeitbezug sowie maßgeblicher Fensterkapazität stark verfälschen könnte. In vielen Fällen erfährt dieser Umstand allerdings eine erhebliche Abmilderung. Das geschieht einerseits durch $c \gg a - c$, falls der verfälschende Anteil relativ gering ausfallen sollte; außerdem verhält es sich bei Fenstertypen wie Tumbling oder Landmark dann bloß wie ein vorgezogener bzw. verzögerter Transfer, da (unter Einbezug der P1-Lösung) ohnehin ein zwangsläufiges Auftauchen im Nachbarfenster garantiert ist; und bei wachsenden Fenstern ist dieses Problem sowieso nur ein zeitweiliges. Gleichwohl ist das Problem und dessen Lösung äußerst wichtig, besonders da es als Grundlage für P3 anzusehen ist.

Anders ist es dagegen bei der Generalisierung, wo sich der Mechanismus unproblematisch und jenseits des Größenproblems weitergedacht sogar zum Positiven auswirkt. Davon ausgehend, dass die Reihenfolge nicht abschließend manipuliert wird, um die gebildeten Gruppen hervorzuheben, bleibt die Eintragsabfolge unangetastet. Das Extrawissen ermöglicht dann einfach die gewünschte k-Anonymität, selbst wenn die Gruppe zunächst unvollständig erscheint (erst in späteren Ergebnismengen in den sichtbaren Bereich rücken) oder einige Einträge sogar nur im Hintergrund bleiben, falls sie aufgrund des Bewegungsverhaltens nie ins sichtbare Fenster geraten würden. Unter Beherzigung aller Schlussfolgerungen aus P1-G könnte der unsichtbare Abschnitt sogar noch größer als nötig gewählt werden. Der zusätzliche Wertepool könnte dann nochmals deutlich feinere Werte begünstigen. Es sei jedoch noch einmal betont, dass auch reine Hintergrundwerte nicht mehrere Gruppen zugleich bzw. zeitlich versetzt aufwerten dürfen, bloß weil sie gerade als die optimale Wahl erscheinen. Auch sie stellen allokierbare Ressourcen dar, die sich dann eventuell einer bestimmten Gruppe bzw. Wertehierarchie verschreiben. Nicht zuletzt funktioniert der Gedanke nur, wenn manche Einträge zeitlebens nie sichtbar ins Fenster gerieten – gerade deshalb ist die pragmatische aufeinanderfolgende Gruppierung weitaus universeller, wenngleich suboptimal. Gleichwohl kann der Mechanismus noch neues Potential eröffnen: Bei Session-Fenstern etwa gibt es einen grundsätzlichen Überschuss, der stets im Verborgenen bliebe. Ebendiesen Wertepool könnte man, sofern gestattet, dann nutzbringend mit einsetzen, um deutlich feinere Gruppen zu bilden. Auch diesem müsste man bei ständiger Zunahme allerdings irgendwann Grenzen setzen.

Lösung von P2.C

Die Lösung von (C) baut grundlegend auf der jener von (B) auf. Die einzige Erschwerung besteht in der Kapazitätsungewissheit, die obendrein ein dynamisches Wechselverhalten zwischen (A) und (B) zur Laufzeit erfordert. Denn während die Kapazitätserweiterung bei Tupelfenstern bereits zur Erstell- bzw. Kompilierzeit vorab problemlos einplanbar ist, müsste sie bei Zeitfenstern dynamisch zur Laufzeit erfolgen. In diesem Falle müsste also über eine Mischform aus Zeit- und Tupelfenstern nachgedacht werden. Das Ganze verliefe dann nach dem Schema: sammle zunächst die Tupel über den zu beobachtenden Zeitraum; sammle anschließend noch die n veranschlagten Tupel für die nötige Zusatzkapazität; führe dann auf Grundlage dessen die Berechnung (Anonymisierung) durch; beschränke die sichtbare Ergebnismenge bzw. Grundlage für höher liegende Auswertungen jedoch rein auf den sichtbaren Zeitfensterabschnitt.

Problemfall P3: unzureichender Hop-Zuwachs

Ein drittes Problem ergibt sich beim naiven Generalisierungs- und Slicing-Operator, wenn die Anzahl neuer Einträge (pro Hop) die schützende Mindestgröße a unterschreiten sollte. Der einfache Verrauschungsoperator ist davon unbetroffen, weil er aufgrund seines zeilenweisen Verrauschens horizontal unabhängig arbeitet. Rein für sich betrachtet wirkt es zunächst unproblematisch, da vielmehr ausschlaggebend scheint, dass gemäß P2 eine ausreichende Fenstergröße gegeben ist. Allerdings resultiert das Problem intersektionell aus dem Werteerhalt-Lösungsansatz von P1, sobald man diesen mit umsetzen möchte, und zu wenigen neuen Einträgen pro Hop, die für sich genommen nicht zur Eingliederung ausreichen.

Im Kern spiegelt es daher genau das P2.B-Problem wider, jedoch in generalisierter Abwandlung, bei der sich die ausschlaggebende Kenngröße von der Kapazität zum Eintragszuwachs durch den Hop gewandelt hat. Anstelle einer vollständigen Anonymisierung des gewünschten Abschnitts geht dem Ganzen ein bereits fertig anonymisierte Abschnitt voraus, der überlappungsbedingt aus vorherigen Fenstern übrig blieb und unverändert übernommen werden soll, was den zu leistenden Anonymisierungsaufwand nützlicherweise minimiert. Im Jumping-Fall bedeutet dies den zeitweiligen Erhalt bis zur Verdrängung, im Landmark-Fall den fortwährenden Erhalt. Falls $hop < a$ gilt, kann es sogar einzelne bis mehrere Fenster geben, die aufgrund des versteckten Überschusses faktisch gar nichts Neues anonymisieren müssen, bis dieser aufgebraucht ist. Das P2.B-Problem hat quasi Startfenster bzw. stets disjunkte Fenster mit einem *skip*-Wert von 0 abgedeckt. P3 muss wiederum auch mit nachfolgenden Fenstern umgehen können, die auf früheren aufbauen.

Der schon erwähnte *skip*-Bereich gibt an, inwieweit die Fensterkapazität durch bereits bestehende Einträge beansprucht wird. Der eigentliche Arbeitsraum zur Anonymisierung neuer Einträge nimmt daher eine Größe von $|\,r\,| - skip$ an. Im äußersten Falle könnte sich der Zuwachs auf einen einzigen Eintrag belaufen, wie es etwa bei Sliding-Fenstern der Fall ist. Die allgemeine Lösung besteht nun wie bei P2 in einer internen Erweiterung. Um stets eine konsistente neue Slicing- oder Generalisierungsgruppe bilden zu können, wäre dementsprechend eine versteckte Kapazitätserweiterung um $a - 1$ Einträge mit einzuplanen.

In Kombination resultiert aus der nötigen Mindestgröße, dem Werteerhalt mehrfach berücksichtigter Einträge und dem erweiterten ausgeblendeten Zuwachs zum konsistenten Anonymisierungsbetrieb bei Stromdaten eine interne erweiterte Fensterkapazität c_{extend} und ein abzählender Zahlenparameter *cut*, der das ausgegebene Ergebnis wieder auf die Originalkapazität herunterkürzt:

$$c_{extend} := c + (a - 1)$$
$$cut := c \quad \text{bzw. in alternativer Notation:} \quad cut := -(a - 1)$$

Ein hoher Eintragszuwachs kann insbesondere bei der Generalisierung von Vorteil sein. Damit sie fortlaufend funktioniert, bedarf es zur Eingliederung neuer Einträge zunächst das gesicherte Minimum der Mindestgröße a bzw. k, um eine weitere Gruppe bilden zu können. Falls sich bei größerer Sprungweite oder ausreichend versteckter Zusatzkapazität zudem mehrere Gruppen zugleich bilden ließen, würden sich noch weitere Einflussräume eröffnen. Ab einer Anzahl von $2a$ Einträgen bzw. zwei Gruppen könnte der Nutzer nämlich auch über den Aufwand und die Granularität verfügen.

Zuwachs	Gruppierungsoptionen	Bemerkung
0 bis a−1	–	unzureichende Anzahl
a bis 2a−1	keine	voraussichtlich suboptimale Granularität, effizient, stets eine Gruppe, forciert ein Aufeinanderfolgen
ab 2a	aufeinanderfolgend	voraussichtlich suboptimale Granularität, effizient, mehrere Gruppen, unveränderte Reihenfolge
	zerstreut	feingranular, aufwendiger, zerstreute Gruppen, unveränderte Reihenfolge
	umstrukturiert	feingranular, aufwendiger, Reihenfolgenänderung zum Sichtbarmachen der Gruppen
	dynamisch	zerstreut mit einer versteckten optionalen Zusatzkapazität

Abbildung 11: auswählbare Gruppierungsoptionen für den Nutzer

Die direkt aufeinanderfolgende Gruppierung ist dabei tendenziell suboptimal, wenn willkürliche bzw. weit auseinanderliegende Werten (hinsichtlich der Generalisierungshierarchie) zu erwarten sind. Wenn dagegen eine natürliche Nähe (etwa bei zeitnahen Messwerten) besteht oder der Stream eine gewisse Vorgruppierung oder gar Vorsortierung mitliefert, dürfte der Informationsgehalt deutlich besser ausfallen.

6 Differential-Privacy-Operator

Dieses Kapitel widmet sich der Operationalisierung von einfacher Verrauschung. Zuerst wird in Abschnitt 6.1 der minimalistisch gehaltene Differential-Privacy-Operator α^D vorgestellt, der aus Kapitel 6 der damals selbstverfassten Bachelorarbeit [Kle20] stammt. Als Nächstes werden in Abschnitt 6.2 die Schwächen dieser minimalistischen Variante hinsichtlich Fensterverarbeitung ergründet. Abschließend wird in Abschnitt 6.3 der erweiterte Differential-Privacy-Operator α^{D+} eingeführt.

6.1 Minimalistischer DP-Operator α^D

6.1.1 Aufbau

Der minimalistische Differential-Privacy-Operator setzt sich folgendermaßen zusammen:

$$\alpha^D{}_{P=\{(A_1,p_1),\ldots,(A_n,p_n)\}}(r)$$

α^D	Bezeichner des Operators α symbolisiert Anonymisierung D bezeichnet das genaue Verfahren (Differential Privacy)
P	die Menge der zu verrauschenden Attribute
A_i	ein Attribut, $A_i \in R$
p_i	die Wahrscheinlichkeit, dass A_i verrauscht wird; $0 \leq p_i \leq 1$
r	die Basisrelation
	weiterhin vorausgesetzt als global Parameter im Hintergrund:
F_i	Wahrscheinlichkeitsverteilung(en) für A_i

Weitere formelle Beschreibungen folgen im anschließenden Abschnitt 6.1.3, da zunächst noch alle getroffenen Annahmen und Einschränkungen erklärt werden sollten.

6.1.2 Annahmen und Einschränkungen

Ein Verrauschungsoperator kann auf verschiedene Ansätze und Techniken ausgerichtet sein, sodass ein jeder über ziemlich spezielle Regeln verfügt. Da eine derart ausführliche Untersuchung aber nicht möglich ist, soll sie sich auf einen relativ allgemeinen Fall beschränken.

Grundsätzlich sind bei Verrauschung zweierlei Ansätze denkbar: einerseits das einfache Randomized-Response-Konzept, andererseits das ε-DP-Prinzip. Der erste Ansatz erfordert die Einführung einer allgemeinen Verrauschungswahrscheinlichkeit, mit der die Frage nach der Beeinflussung einer Zeile stets und unabhängig aufs Neue gestellt wird. Der zweite Ansatz gibt dagegen vor, wie sehr die veränderte Häufigkeit von Werten aus relativer Sicht maximal abweichen darf. Diese Methode hängt von der Gesamtheit der Einträge ab, ist also nicht zeilenunabhängig. Die Minimalform dieses Ansatzes würde sich auf ε beschränken, in der erweiterten Form wäre zusätzlich δ mit bedacht. Aus formaler Sicht ergeben sich drei verschiedene Ausführungen:

$$\text{Münzwurf-Variante:} \quad {\alpha^D}_{P=\{(A_1,p_1),\ldots,(A_n,p_n)\}}(r)$$

$$(\varepsilon,\delta)\text{-Variante:} \quad {\alpha^D}_{E=\{(A_1,\varepsilon_1,\delta_1),\ldots,(A_n,\varepsilon_n,\delta_n)\}}(r)$$

$$\varepsilon\text{-Variante:} \quad {\alpha^D}_{E'=\{(A_1,\varepsilon_1),\ldots,(A_n,\varepsilon_n)\}}(r) \;=\; {\alpha^D}_{E=\{(A_1,\varepsilon_1,\delta_1=0),\ldots,(A_n,\varepsilon_n,\delta_n=0)\}}(r)$$

Da der zeilenunabhängige Ansatz am ehesten kontrollierbar ist bzw. die Auswirkungen individuell sind, wird sich alle weitere Untersuchung auf ${\alpha^D}_P$ beschränken.

Während zeilenbezogen die allgemeine Frage ist, ob verrauscht wird oder nicht, geht es spaltenbezogen vorrangig darum, inwieweit sich wertespezifische Veränderungen auswirken. Verteilungen müssen etwa nicht zwangsweise auf ein einziges Attribut beschränkt sein, sondern können sich ebenso gut auf mehrere Attribute beziehen. Es wäre auch möglich, dass die Verrauschung wertabhängig ist – zum einen durch die echten bzw. Vorgängerwerte, zum anderen durch Werte von unbetroffenen Attributen. Nicht zuletzt würde dies noch weitere Fragen eröffnen: Wie sähe es dann beispielsweise mit der zeitlichen Beeinflussung aus? Würde nur auf Basis des Ausgangszustandes verrauscht werden oder dynamisch? Da eine gesonderte Untersuchung viel zu sehr in die Tiefe ginge, wird auch hier nur der allgemeinste Ansatz weiterverfolgt. Es sei daher festgelegt, dass Attribute ausschließlich einzeln und unabhängig voneinander verrauscht werden.

Des Weiteren gäbe es noch einige triviale und weniger bedeutsame Dinge, die der Klarheit und Formalisierung wegen angesprochen werden sollten. Es sei sichergestellt, dass jedes Attribut, das man verrauschen möchte, sowohl vorkommt als auch eine entsprechende Verrauschungsverteilung besitzt. Alle Attribute tauchen in der Verrauschungsmenge nur einmalig auf, da sonst unklar wäre, auf Grundlage welches Wahrscheinlichkeitswertes man überhaupt verrauschen sollte. Attribute, die nicht in der Verrauschungsmenge vorkommen, werden wahrheitsgemäß belassen bzw. so behandelt, als sei $p_i = 0$. Zur weiteren Vereinfachung sei außerdem sichergestellt, dass $p_i > 0$ gilt, also bloß Attribute mit echter Verrauschungsabsicht aufgeführt werden.

6.1.3 Ausführliche Formalisierung

Sei $p_i \in [0,1]$ die Wahrscheinlichkeit, mit der ein Wert a_{ij} zufällig verrauscht werden soll. Im Umkehrschluss meint die Gegenwahrscheinlichkeit $1 - p_i$ Wertetreuheit, also keine Veränderung. Eine bestimmte Verrauschungswahrscheinlichkeit p_i gilt im Übrigen für alle Werte eines Attributes A_i. Im Falle einer Verrauschung existiert immer eine eindeutige Verteilung. Sie hängt auf jeden Fall vom Attribut am, unter Umständen auch noch vom Ausgangswert selbst. Eine statische Verteilung sei eine Verteilung, die für alle a_{ij}-Werte eines Attributes A_i gleich aussieht. Eine dynamische Verteilung sei dagegen zusätzlich vom jeweiligen Ausgangswert a_{ij} abhängig. b_i ist im Folgenden ein boolescher Wahrheitswert, bei welchem dynamisch *wahr* und statisch *falsch* symbolisiert. Abbildung 12 verbildlicht noch einmal die namentliche Attributeinteilung der zu verzerrenden Ausgangswerte.

A_1	...	A_i	...	A_n
a_{11}	...	a_{i1}	...	a_{n1}
...	...	...	...	...
a_{1j}	...	a_{ij}	...	a_{nj}
...	...	...	...	...
a_{1m}	...	a_{im}	...	a_{nm}

Abbildung 12: Unverrauschte Ausgangswerte und deren Attributzuordnung

Der Algorithmus 1, wenngleich nur Pseudocode, verdeutlicht, wie alle vorgestellten Faktoren im zufallsgesteuerten Entscheidungsprozess agieren.

Algorithmus 1: Randomisierter Mechanismus

Data: Verteilungsname A_i, Ausgangswert a_{ij}, Verrauschungswahrscheinlichkeit p_i, dynamische Verzerrung b_i (Wahrheitswert)
Result: Wert, der vielleicht verrauscht wurde, vielleicht aber auch unverändert ist
randomized_Response(A_i,a_{ij},p_i,b_i);
if *random()* $\leq 1 - p_i$ **then**
  return a_{ij};
else
  if $b_i = false$ **then**
    return statische_Verteilung(A_i, random());
  else
    return dynamische_Verteilung(A_i, a_{ij}, random());
  end
end

Zu den Verteilungen selbst lässt sich nichts weiter konkretisieren. Den Operator interessiert höchstens, dass sie vorhanden beziehungsweise zugewiesen sind. Nichtsdestotrotz sei an dieser Stelle zwischen typischen Formen unterschieden. Trotz der Beschränkung, dass nur klar von einander abgegrenzte Einzelattribute erlaubt sind, können die Verteilungen hochvariabel sein: sie könnten diskret oder stetig sein – endlich, abzählbar unendlich oder überabzählbar – durch einen Algorithmus funktional beschreibbar oder auch nicht. Daher ist nicht viel mehr möglich, als zu beiden Formen ein exemplarisches Beispiel anzuführen.

Statische Verteilung

Eine statische Verteilung gilt universell für alle konkreten Werte eines Attributes. Ein einfaches Beispiel wäre eine endliche Verteilung, die alle Tage eines Jahres unabhängig vom Jahr selbst abbildet. Wäre nicht der Schaltjahr-Tag zu berücksichtigen, wäre es sogar eine Gleichverteilung. Abbildung 13 verbildlicht das Ganze noch einmal. Natürlich haben statische Verteilungen klare Grenzen in dem, was sie beschreiben können. Im Grunde sind sie nur ein Spezialfall. Würde man zusätzlich das Jahr mit einfließen lassen und einen konkreten Tag auf dessen jeweiliges Jahr verteilen wollen, müsste man nämlich zu einer dynamischen Verteilung greifen.

Wahr-scheinlichkeit	Wert
4/1461	01.01.
4/1461	02.01.
4/1461	03.01.
...	...
4/1461	28.02.
1/1461	29.02.
4/1461	01.03.
...	...
4/1461	31.12.

Abbildung 13: Statische Verteilung

Dynamische Verteilung

Ein exemplarisches Beispiel für eine dynamische Verteilung sei eine Verteilung, die einen beliebigen Tag mit hoher Wahrscheinlichkeit zu einem seiner umliegenden Nachbartage verrauscht. Der ursprüngliche Wert x würde in diesem Falle sogar bewusst vermieden werden. Für Abbildung 14 wurde im Konkreten der 08.03. ausgewählt. Während eine statische Wissensbasis wie in Abb. 13 unmittelbar vorliegen würde, könnte eine dynamische wiederum etwa durch eine diskretisierende Funktion generiert werden, was im gewählten Beispiel einer 2-welligen Funktion gleichkommen würde. Es sei noch einmal betont, dass dieses Beispiel nur einen Bruchteil der denkbaren Möglichkeiten zeigt, denn viele dynamische Verhalten lassen sich nicht algorithmisch, sondern nur händisch abbilden.

Wahr-scheinlichkeit	Wert
...	...
0.05	$x-3$
0.15	$x-2$
0.20	$x-1$
0.05	x
0.20	$x+1$
0.15	$x+2$
0.05	$x+3$
...	...

$\Longrightarrow$

Wahr-scheinlichkeit	Wert
...	...
0.05	05.03.
0.15	06.03.
0.20	07.03.
0.05	08.03.
0.20	09.03.
0.15	10.03.
0.05	11.03.
...	...

Abbildung 14: Dynamische Verteilung

6.1.4 Beispiel

Zum genauen Verständnis des α^D-Operators sei dessen Wirkungsweise an einem Beispiel verdeutlicht. Eine kurze inhaltliche Kontextualisierung ist in Abschnitt 2.7 zu finden.

Die Ausgangsrelation r setzt sich aus dem Schema $R = \{A, B, C, D\}$ zusammen. Ebendiese Relation gilt es mittels einfacher Verrauschung nun zu anonymisieren. Attribut A soll mit einer Wahrscheinlichkeit von 0.16 verrauscht werden, Attribut B mit einer Wahrscheinlichkeit von 0.33, Attribut C mit einer Wahrscheinlichkeit von 0.5 und Attribut D soll unverrauscht bleiben. Aus all diesen Forderungen ergibt sich folgender Ausdruck:

$$r_a := \alpha^D{}_{P=\{(A,0.16),(B,0.33),(C,0.5)\}}(r)$$

Die Verrauschungsverteilungen von B und C sind in Abbildung 15 zu finden und beide statisch. Die Verrauschungsverteilungen von A sei dynamisch und verhält sich wie jene, die in Abbildung 14 ersichtlich ist. Sollte ein Wert aus A verrauscht werden, erfolgt also höchstwahrscheinlich eine Verzerrung zu einem Tag, der nicht weit vom ursprünglichen entfernt ist.

B	p(b)
00	0.410
A0	0.358
AA	0.078
B0	0.102
AB	0.045
BB	0.007

C	p(c)
+	0.85
−	0.15

Abbildung 15: statische Verteilungen für die Verrauschung von B und C

Abbildung 16 zeigt exemplarisch, wie die Anwendung des Operators aussehen könnte. Alle manipulierten Werte sind im Folgenden rot hervorgehoben. Die Ungewissheit des Zufalls erlaubt es im Extremen, dass einerseits alles, andererseits auch nichts von A, B und C verändert werden könnte. Ob tendenziell viele oder wenige Werte verzerrt werden, ist abhängig von der jeweiligen Verrauschungswahrscheinlichkeit. Damit das aktuelle Beispiel und die noch folgenden auch wirklich ein besseres Verständnis begünstigen, spiegelt sich die Verrauschungswahrscheinlichkeit im Allgemeinen in der Anzahl der verrauschten Werte wider, was erwartungsgemäß rund $|r| \cdot p_i$ verrauschte Einträge bedeutet. Auf unwahrscheinliche Fälle wird also verzichtet.

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	−	32
20.03.	A0	+	36
25.03.	A0	+	42
31.03.	B0	−	56

$\Longrightarrow$

A	B	C	D
28.12.	00	+	18
05.01.	A0	+	24
31.01.	B0	+	32
20.03.	00	+	36
25.03.	A0	+	42
31.03.	B0	−	56

Abbildung 16: links r, rechts r_a

6.2 Schwächen von α^D-Operator je Fenstertyp

In diesem Abschnitt werden fallspezifisch die Schwächen des minimalistischen α^D-Operators hinsichtlich aller wesentlichen Fenstertypen beleuchtet.

6.2.1 Tupelfenster

Bei Gebrauch des minimalistischen Operators lassen sich bei Tupelfenstern abhängig vom Fenstertyp folgende Stromdaten-Probleme konstatieren:

Tupelfenster-Typ	P1	P2	P3
stehend	nein	nein	nein
Tumbling	nein	nein	nein
Sliding	ja	nein	nein
Jumping	ggf.	nein	nein
Session	nein	nein	nein
einseitig wachsendes LM	ja	nein	nein
relativ wachsend	ggf.	nein	nein
einseitig schrumpfendes LM	ja	nein	nein
relativ schrumpfend	ggf.	nein	nein
beidseitig wachsendes LM	ja	nein	nein
beidseitig schrumpfendes LM	ja	nein	nein

Abbildung 17: Tupelfenster-Schwächen von α^D

Die einfache Verrauschung ist nur für Mehrfachverrauschung (P1) anfällig. Aufgrund des zeilenweisen unabhängigen Vorgehens gibt es keine kritischen Mindestgrößen, die sich unterschreiten lassen, sodass sie von der unzureichenden Fenstergröße (P2) und dem unzureichenden Zuwachs (P3) gänzlich unbetroffen ist.

P1 - Mehrfachverrauschung

Auf Tumbling- und Session-Fenster ist der damalige Operator bedenkenlos anwendbar, da eine grundsätzliche Schnittmengenfremdheit gegeben ist. Ein stehendes Fenster entspricht mangels Folgefenster sogar dem typischen Anwendungsfall, für den der damalige Operator konzipiert wurde.

Jumping-Fenster sind für naive Mehrfachverrauschung verschieden stark empfänglich. Abseits des Tumbling-Sonderfalls ist stets eine Gefahr gegeben. Bei einer gewählten Hopgröße hop und Tupelkapazität c überschneiden sich stets $c-|hop|$ Einträge mit dem Nachbarfenster. Im laufenden Betrieb könnte derselbe Eintrag $\lfloor \frac{c}{|hop|} \rfloor$ oder $\lceil \frac{c}{|hop|} \rceil$ Mal naiv verrauscht vorkommen – einzig bei der kapazitätssammelnden Startphase wäre es weniger, sofern sie nicht ausgeblendet wird. Bei Fenstern mit winziger Schnittmenge (sehr nahe am Tumbling) sind daher nur vereinzelte Einträge gefährdet. Ein Sliding birgt im Gegensatz dazu die stärkste nichtmonotone Gefahr, da derselbe Eintrag im laufenden Betrieb ganze c-mal vorkommt.

Die allergrößte Gefahr geht allerdings von monotonen Landmarks aus. Bei einseitig und beidseitig wachsenden Landmarks ist die Gefahr sogar permanent, da Einträge niemals verschwinden. Einseitig und beidseitig schrumpfende Landmarks sind fast gleichermaßen ge-

fährlich, was einzig dadurch abgeschwächt wird, dass diese aufgrund des Schrumpfens nicht ewig bestehen werden.

Auch Fenster mit relativer Größenentwicklung bergen eine klare Gefahr. Das Problem P1 besteht nicht, solange sich die Fenster wie Tumblings oder Sessions mit veränderlicher Größe verhalten; im zunehmenden Fall müssten die Hops jedoch dynamisch mitwachsen können. Im Falle einer konstanten Hop-Größe ist eine Überlappungsgefahr daher kaum vermeidbar. Bei relativem Wachstum geschieht eine Überlappung sogar irgendwann zwangsläufig und wächst mit.

Die folgende Übersicht schlüsselt abschließend, soweit es sich näherungsweise formalisieren ließe, die zu erwartende Berücksichtigungshäufigkeit in konkreten Zahlen auf. Wie bereits in Kapitel 4 steht hop hier kontextuell verkürzt für *hopTuple*.

Tupelfenster-Typ	Berücksichtigungsanzahl (= Verrauschungsanzahl)	P1
stehend	einmalig	nein
Tumbling	einmalig	nein
Sliding	c-fach	ja
Jumping	bei $\lvert hop\rvert < c$: $\lfloor \frac{c}{\lvert hop\rvert} \rfloor$ oder $\lceil \frac{c}{\lvert hop\rvert} \rceil$	ja
	bei $\lvert hop\rvert = c$: einmalig	nein
Session	einmalig oder null	nein
einseitig wachsendes LM	potenziell unendlich oft	ja
relativ wachsend	bei dynamischer Disjunktheit: einmalig	nein
	ansonsten: bei Einträgen des n-ten Fensters von $\lceil \frac{c_{start} + (n-1)\cdot(\lvert hop_{fast}\rvert - \lvert hop_{slow}\rvert)}{\lvert hop_{fast}\rvert} \rceil$ (älteste Fenstereinträge) bis $\lceil \frac{c_{start} + (n-1)\cdot(\lvert hop_{fast}\rvert - \lvert hop_{slow}\rvert)}{\lvert hop_{slow}\rvert} \rceil$ (neuste Fenstereinträge) reichend	ja
einseitig schrumpfendes LM	je nach Position von einmalig bis $\lceil \frac{c_{start}}{hop} \rceil$ reichend	ja
relativ schrumpfend	bei Disjunktheit: einmalig	nein
	ansonsten: von maximal $\lceil \frac{c_{start} + n\cdot(\lvert hop_{slow}\rvert - \lvert hop_{fast}\rvert)}{\lvert hop_{fast}\rvert} \rceil - initDiscount$ beim $n = \lceil \frac{c_{start}}{\lvert hop_{fast}\rvert} \rceil$-ten Fenster mit $initDiscount \approx$ ReLU $\lfloor \frac{c_{start} - 1 - n\cdot\lvert hop_{fast}\rvert}{\lvert hop_{slow}\rvert} \rfloor$ (Startphasenabzug) bis einmalig reichend	ja
beidseitig wachsendes LM	potenziell unendlich oft	ja
beidseitig schrumpfendes LM	je nach Position von einmalig bis $\lceil \frac{c_{start}}{\lvert hop_{start}\rvert + \lvert hop_{end}\rvert} \rceil$ reichend	ja

Abbildung 18: Berücksichtigungsanzahl bei Tupelfenstern

6.2.2 Zeitfenster

Die Stromdaten-Probleme von Zeitfenstern sind deckungsgleich mit den bereits bekannten von Tupelfestern. Im Allgemeinen kann daher auch für Zeitfenster einfach auf Abbildung 17 verwiesen werden. Da die Fensterentwicklung nun zeitlich quantifiziert wird, ist die Tupelanzahl allerdings erst zur Laufzeit ermittelbar. Die Berücksichtigungsanzahl ließe sich dennoch zumeist wie in Abbildung 18 ermitteln, nur gegebenenfalls mit den Ersetzungen $c \to d$ und $hopTuple \to hopTime$. Der Hauptunterschied bestünde beim Sliding, dessen

Eintragsberücksichtigung nun individuell von einmalig bis beliebig endlich oft reichen kann. An der Berechnungsgrundlage selbst ändert sich schematisch kaum etwas; vielmehr besteht die eigentliche Änderung in der schwankenden Konzentration neu hinzukommender Einträge.

6.3 Erweiterter DP-Operator α^{D+}

6.3.1 Aufbau

Der verfeinerte Differential-Privacy-Operator setzt sich folgendermaßen zusammen:

$$\alpha^{D+}{}_{P=\{(A_1,p_1),\ldots,(A_n,p_n)\}\,[;skip;reverse]}(r)$$

α^{D+}	Bezeichner des Operators α symbolisiert Anonymisierung D bezeichnet das genaue Verfahren (Differential Privacy) $+$ als abgrenzender Indikator für den erweiterten Operator
P	die Menge der zu verrauschenden Attribute
A_i	ein Attribut, $A_i \in R$
p_i	die Wahrscheinlichkeit, dass A_i verrauscht wird; $0 \leq p_i \leq 1$
r	die Basisrelation
	weiterhin vorausgesetzt als globale Parameter im Hintergrund:
F_i	Wahrscheinlichkeitsverteilung(en) für A_i
	neue optionale Parameter:
$(skip)$	Anzahl unveränderter Einträge zu Beginn, deren Verrauschung übersprungen wird ○ Default: $skip = 0$
$(reverse)$	optionale Richtungsumkehr der Arbeitsweise ○ Default: $reverse := false$ Anmerkung: Die Standard-Arbeitsweise des Operators ist von oben nach unten ausgelegt. In Bezug auf Stromdaten wird im Folgenden nämlich angenommen, dass sich die ältesten Einträge eines Fensters am Anfang beziehungsweise die neusten Einträge am unteren Ende befinden. Sollte eine Arbeitsweise in gegenteiliger Richtung nötig sein, würde die komplette Funktionsweise quasi gespiegelt bzw. auf den Kopf gestellt werden.

6.3.2 Annahmen und Einschränkungen

Die Annahmen und Einschränkungen haben sich gegenüber 6.1.2 im Kern nicht verändert. Sie werden höchstens an einigen Stellen explizit überschrieben, wenn der Operator um Alternativen bereichert wurde.

6.3.3 Beispiele

Fallbeispiel 1: Default

Das Defaultverhalten des α^{D+}-Operators mit *skip*-Wert 0 ist deckungsgleich mit dessen Vorläufer α^{D}, was nochmals kurz durch Abb. 19 veranschaulicht werden soll.

$$\alpha^{D+}{}_{P}(r) \equiv \alpha^{D+}{}_{P;skip=0;reverse=false}(r) \iff \alpha^{D}{}_{P}(r)$$

$$r_2 := \alpha^{D+}{}_{P=\{(A,0.16),(B,0.33),(C,0.5)\}}(r_1)$$

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	−	32
20.03.	A0	+	36
25.03.	A0	+	42
31.03.	B0	−	56

$\Longrightarrow$

A	B	C	D
28.12.	00	+	18
05.01.	A0	+	24
31.01.	B0	+	32
20.03.	00	+	36
25.03.	A0	+	42
31.03.	B0	−	56

Abbildung 19: links r_1, rechts r_2

Fallbeispiel 2: *skip*-Parameter

Die Abbildung 20 veranschaulicht exemplarisch den Gebrauch des neuen *skip*-Parameters. Bei $skip = 4$ werden die ersten vier Einträge, die im entsprechenden Schritt eingegraut hervorgehoben wurden, unverändert übernommen. Hierbei sei deutlich darauf hingewiesen, dass der privacy-konforme Gebrauch dann vollständig dem Nutzer obliegt. Falls die vier obersten Zeilen von r_1 nicht bereits anonymisiert wurden und Relation r_2 dann veröffentlicht wird, hätte man diese vier, auch wenn es für außenstehende Betrachter natürlich nicht erkennbar ist, beim Schutzvorgang übergangen.

$$r_2 := \alpha^{D+}{}_{P=\{(A,0.25),(B,0.5),(C,1.0)\};skip=4}(r_1)$$

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	−	32
20.03.	A0	+	36
25.03.	A0	+	42
31.03.	B0	−	56
02.05.	A0	+	49
08.05.	A0	+	58

$\Longrightarrow$

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	−	32
20.03.	A0	+	36
25.03.	AA	+	42
04.04.	B0	+	56
02.05.	AB	−	49
08.05.	A0	+	58

Abbildung 20: links r_1, rechts r_2

Korrekterweise würde der *skip*-Parameter eingesetzt werden, um an bereits verrauschte Einträge anzuschließen, was Abb. 21 verdeutlichen soll.

$$r_2 := \alpha^{D+}{}_{P=\{(A,0.25),(B,0.5),(C,1.0)\}}(r_1)$$
$$r_4 := r_2 \cup r_3$$
$$r_5 := \alpha^{D+}{}_{P=\{(A,0.25),(B,0.5),(C,1.0)\};skip=4}(r_4)$$

A	B	C	D
01.01.	A0	+	18
24.12.	00	−	24
31.01.	B0	+	32
20.03.	AB	+	36

$\Longrightarrow$

A	B	C	D
01.01.	A0	+	18
24.12.	00	−	24
31.01.	B0	+	32
20.03.	AB	+	36
25.03.	A0	+	42
31.03.	B0	−	56
02.05.	A0	+	49
08.05.	A0	+	58

$\Longrightarrow$

A	B	C	D
01.01.	A0	+	18
24.12.	00	−	24
31.01.	B0	+	32
20.03.	AB	+	36
25.03.	AA	+	42
04.04.	B0	+	56
02.05.	AB	−	49
08.05.	A0	+	58

Abbildung 21: von links nach rechts: r_2, r_4 und r_5

Fallbeispiel 3: Stromdaten am Beispiel eines Jumping-Tupelfensters

Die Abbildungen 22 und 23 veranschaulichen in acht Schritten die exemplarische Nutzung des α^{D+}-Operators zur Verrauschung eines Jumping-Tupelfensters mit einer Kapazität von acht Einträgen und einer zukunftsgerichteten Sprungweite von drei Einträgen. Die Relation r_1 repräsentiert das unanonymisierte Startfenster, das es zunächst vollständig zu anonymisieren gilt. In Folgeschritten sind nur noch neu hinzugekommene Einträge zu verrauschen. Die ungeraden Relationen zeigen im Resultat jeweils Fensterverschiebungen, die geraden zeigen wiederum Verrauschungen.

$$r_2 := \alpha^{D+}{}_{P=\{(A,0.2),(B,0.33)\}}(r_1)$$
$$r_3 := JUMPING_{type=tuple,hop=3}(r_2)$$
$$r_4 := \alpha^{D+}{}_{P=\{(A,0.2),(B,0.33)\};skip=5}(r_3)$$
$$r_5 := JUMPING_{type=tuple,hop=3}(r_4)$$
$$r_6 := \alpha^{D+}{}_{P=\{(A,0.2),(B,0.33)\};skip=5}(r_5)$$
$$r_7 := JUMPING_{type=tuple,hop=3}(r_6)$$
$$r_8 := \alpha^{D+}{}_{P=\{(A,0.2),(B,0.33)\};skip=5}(r_7)$$

ID	A	B
1	09.09.	AB
2	10.09.	00
3	12.03.	B0
4	14.05.	AA
5	27.11.	A0
6	16.09.	BB
7	21.10.	00
8	26.10.	00

Relation 1

$\Longrightarrow$

ID	A	B
1	09.09.	BB
2	10.09.	00
3	06.06.	B0
4	14.05.	AA
5	27.11.	B0
6	16.09.	BB
7	02.03	00
8	26.10.	A0

Relation 2

$\Longrightarrow$

ID	A	B
4	14.05.	AA
5	27.11.	B0
6	16.09.	BB
7	02.03	00
8	26.10.	A0
9	28.10.	B0
10	08.06.	A0
11	04.02.	A0

Relation 3

$\Longrightarrow$

ID	A	B
4	14.05.	AA
5	27.11.	B0
6	16.09.	BB
7	02.03	00
8	26.10.	A0
9	28.10.	AB
10	04.06	A0
11	04.02.	A0

Relation 4

$\Longrightarrow$ ...

Abbildung 22: von links nach rechts: r_1 bis r_4 (Teil 1 von 2)

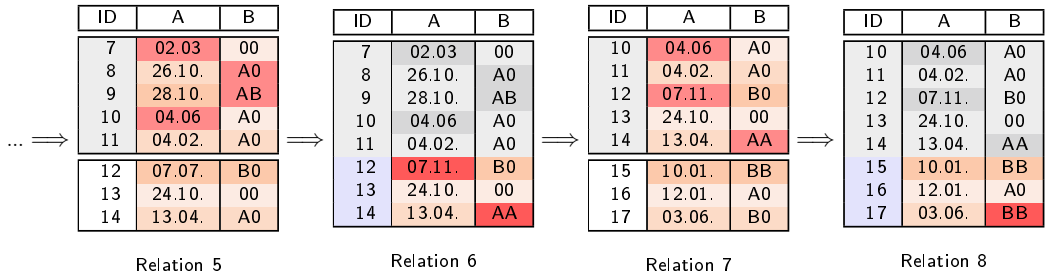


Abbildung 23: von links nach rechts: r_5 bis r_8 (Teil 2 von 2)

Fallbeispiel 4: Umkehr der Arbeitsweise

Falls das Anwendungsumfeld erfordert, dass sich die zu überspringenden Einträge am unteren Relationenende befinden, seien abschließend auch noch drei anregende Notationen zur Umkehr der Arbeitsweise vorgeschlagen. Da der DP-Operator sehr reduziert und klar definiert ist, ist sogar eine problemlose Gleichsetzung der drei Notationen möglich; bei den anderen beiden Operatoren, deren Einführung und Erweiterung noch aussteht, sind die Notationsmöglichkeiten aufgrund der steigenden Parameteranzahl dagegen deutlich eingeschränkter. Zwecks operatorübergreifender Einheitlichkeit wird daher ganz klar die erste Notation präferiert.

Varianten: $\alpha^{D+}{}_{P;skip=n;reverse=true}(r) \equiv \alpha^{D+}{}_{P;skip=-n}(r) \equiv \alpha^{D+}{}_{P;skip_A=0,skip_E=n}(r)$

Präferenz und Default: $\alpha^{D+}{}_{P}(r) \equiv \alpha^{D+}{}_{P;[...];reverse=false}(r)$

$r_2 := \alpha^{D+}{}_{P=\{(A,0.25),(B,0.5),(C,1.0)\};skip=4,reverse=true}(r_1)$

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	−	32
20.03.	A0	+	36
25.03.	A0	+	42
31.03.	B0	−	56
02.05.	A0	+	49
08.05.	A0	+	58

$\Longrightarrow$

A	B	C	D
01.01.	A0	+	18
24.12.	00	−	24
31.01.	B0	+	32
20.03.	AB	+	36
25.03.	A0	+	42
31.03.	B0	−	56
02.05.	A0	+	49
08.05.	A0	+	58

Abbildung 24: links r_1, rechts r_2

7 Slicing-Operator

Dieses Kapitel widmet sich der Operationalisierung des Slicings. Zuerst wird in Abschnitt 7.1 der minimalistisch gehaltene Slicing-Operator α^S vorgestellt, der aus Kapitel 5 der damals selbstverfassten Bachelorarbeit [Kle20] stammt. Als Nächstes werden in Abschnitt 7.2 die Schwächen dieser minimalistischen Variante hinsichtlich Fensterverarbeitung ergründet. Abschließend wird in Abschnitt 7.3 der erweiterte Slicing-Operator α^{S+} eingeführt.

7.1 Minimalistischer Slicing-Operator α^S

7.1.1 Aufbau

Der minimalistische Slicing-Operator setzt sich folgendermaßen zusammen:

$$\alpha^S{}_{V=\{V_1,\ldots,V_n\},sliceSize}(r)$$

α^S	Bezeichner des Operators
	α symbolisiert Anonymisierung
	S bezeichnet das genaue Verfahren (Slicing)
V	Menge der Korrelationsmengen
V_i	Korrelationsmenge, $V_i \subseteq R$
$sliceSize$	Slicing-Größe
r	Basisrelation

Eine formale Beschreibung im Detail folgt im anschließenden Abschnitt 7.1.3, da zunächst noch die getroffene Annahmen erklärt werden sollten.

7.1.2 Annahmen und Einschränkungen

Um den beschriebenen Operator so schlicht wie möglich zu halten, sind mehrere Annahmen und Einschränkungen getroffen worden. Den internen Trennungs- und anschließenden Zusammenführungsvorgang wird man jenseits der Parameter V und $sliceSize$ nicht beeinflussen können. Der horizontale Vorgang wird immer mittels einer zusätzlichen temporären Hilfs-ID vollzogen. Man wird keinen Einfluss auf sie nehmen können, sodass es rein äußerlich so erscheint, als hätte sie am Ende gar nicht existiert.

Das konkrete horizontale Aufteilungsschema sowie die Slicing-Größe sollten nach Möglichkeit nicht öffentlich bekannt sein. Ansonsten könnten Angreifer durch deren Kenntnis die gleichen Angriffe wie bei der k-Anonymität ausführen, was etwa Homogenitätsattacken und Angriffe mit Hintergrundwissen zulassen würde. Die horizontale Aufteilung kann nach verschiedenen größenabhängigen Mustern erfolgen. Idealerweise würde die Eintragsanzahl $|r|$ einem Vielfachen der Slicing-Größe entsprechen, sodass einfach n Slice-Gruppen der Größe $sliceSize$ gebildet werden müssten. Da dies aber nicht immer der Fall ist, müssen einige Gruppen bzw. Splitstellen größer angesetzt werden, um die unzureichende Restgruppe zu tilgen.

Um den Operator so minimalistisch wie möglich zu halten, wird im Folgenden ein triviales horizontales Auswahlverfahren als Standard festgelegt. Die Abbildungen 25–27 veranschau-

lichen drei naheliegende Verfahren zur Splitstellensetzung bei einer Slicing-Größe von 3 für die Relationengrößen 3 bis 12. Der minimalistische Slice-Operator wird als Standard das Schema Top-down benutzen. Solange es möglich ist, werden von oben beginnend Splitstellen der Größe *spliceSize* gesetzt. Die abschließende unterste Gruppe verfügt über eine variable Übergröße, die von *sliceSize* bis $2 \cdot sliceSize - 1$ reicht, um alle überschüssigen Einträge aufnehmen zu können.

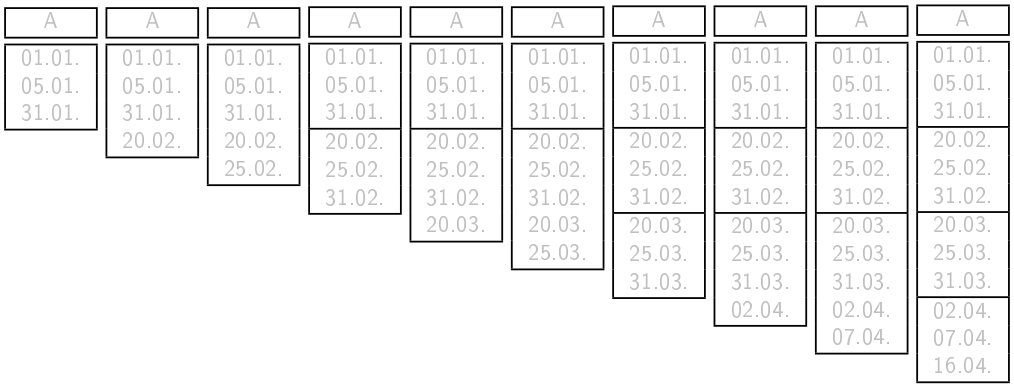


Abbildung 25: Splitstellensetzung nach dem Schema Top-down mit einer potenziell übergroßen, auffangenden Gruppe am Ende

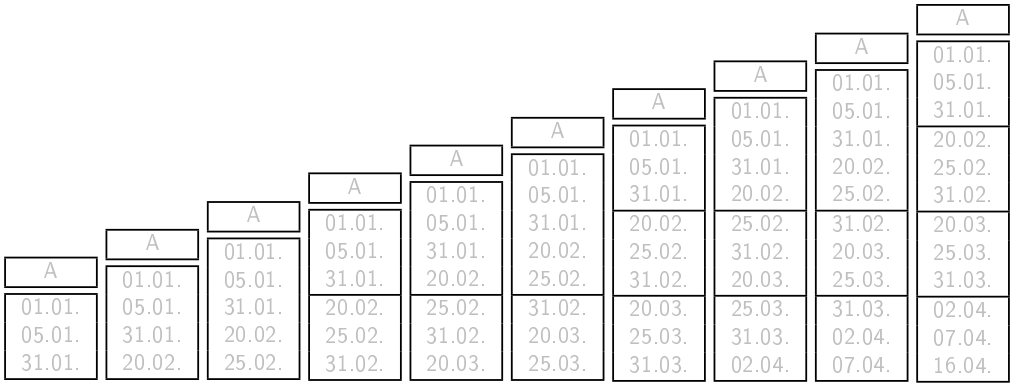


Abbildung 26: Splitstellensetzung nach dem Schema Bottom-up mit einer potenziell übergroßen, auffangenden Gruppe am Anfang

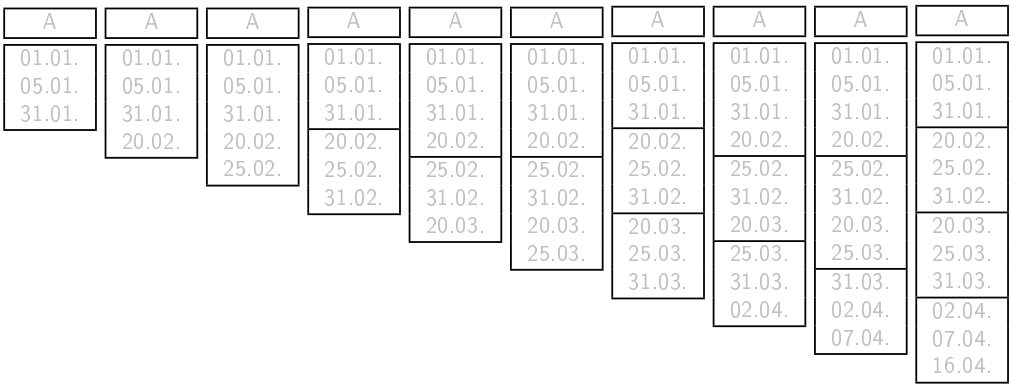


Abbildung 27: Splitstellensetzung, wo die Übergröße per Round Robin (von oben beginnend) auf die ersten $spliceSize - 1$ Gruppen sanft verteilt wird

Beim Neuanordnen der Einträge wird nachfolgend eine randomisierte Permutation vorausgesetzt. Bei einer der Korrelationsgruppen könnte man theoretisch auch auf eine Sortierung zurückgreifen, allerdings nicht bei mehreren, da Sortiermuster in Kombination weitere Schwachstellen eröffnen könnten, sofern der ursprünglichen Relation eine korrelationsgruppenübergreifende Vorsortierung zugrunde liegt. Dieselbe Sortierrichtung würde nämlich die Originalbeziehungen zweier Gruppen bewahren, was die Anonymisierungsabsicht faktisch untergraben würden; für Außenstehende ist dieser fahrlässige Einsatz zumindest nicht eindeutig erkennbar, da sich die Sortierung auch als nachträgliches Mittel zwecks Übersichtlichkeit auslegen ließe. Entgegengesetzte Sortierrichtungen könnte man dagegen sogar offensichtlich erkennen und würden dann nicht nur die Gruppen offenbaren, sondern obendrein auch die Originalbeziehungen, da eine der Sortierungen bloß zurückgespiegelt werden müsste, was diese Kombination noch gefährlicher macht. Ergo sollte man grundsätzlich randomisierte Permutationen nutzen.

Die vertikale Aufteilung wird im Folgenden als Partitionierung vorausgesetzt. Der korrekte Operatorgebrauch verlangt also ab, dass jedes Attribut auf genau eine Korrelationsgruppe aufgeteilt sein muss. Während bei der horizontalen Aufteilung eine Partitionierung nämlich zwingend ist, könnte vertikal aber auch eine freiere Aufteilung angestrebt werden. Man könnte natürlich erlauben, dass ein Attribut in einer, keiner oder mehreren Korrelationsgruppen auftauchen dürfte, was eine Ausblendung oder Mehrfachverwendung gestatten würde. Das Ausblenden unbrauchbarer Attribute wäre allerdings nichts weiter als eine dazwischengeschaltete Projektion, wofür bereits der gesonderte Operator existiert, sodass dieses Feature keine nützliche Neuerung einführen würde. Die Wiederverwendung in mehreren Korrelationsmengen kann wiederum dazu führen, dass aufgrund der Überlappung relativ wahrscheinliche bis definitive Rückschlüsse möglich sind, was in den Abbildungen 28 und 29 zu sehen ist.

Die Tücken der Mehrfachverwendung seien also noch etwas genauer anhand von drei Beispielen veranschaulicht. Abbildung 28 zeigt die offensichtliche Gefahr, sobald ein Attribut wie A nahezu bis absolut quasi-identifizierend sein sollte. Durch QID-Selbstverbund ließen sich zwei Korrelationsmengen ohne Weiteres wieder verbinden. Falls das Attribut für beide Korrelationsmengen unabdingbar ist, kann man sich deren Trennung im Grunde ersparen

und sie zu einer zusammenfassen. Ist eine Attributtrennung wiederum zwingend, muss sich das QID-Attribut für eine Seite entscheiden und darf auch insgesamt nur einmalig genutzt werden.

B	C	A.K1	A.K2	D
B0	–	31.01.	01.01.	18
AB	–	14.01.	05.01.	24
00	+	05.01.	31.01.	62
00	+	01.01.	14.01.	32
A0	+	07.02.	31.03.	56
A0	+	20.03.	25.03.	42
B0	–	25.03.	20.03.	36
BB	–	31.03.	07.02.	28

Abbildung 28: geslicte Relation mit quasi-identifizierendem Mehrfachattribut A

Abbildung 29 zeigt das weniger offensichtliche Problem, sobald ein Attribut wie B lokal quasi-identifizierend sein sollte. Attributwerte könnten sich zwar wiederkehrend wiederholen, aber möglicherweise nicht naheliegend genug auf lokaler Ebene. Während ein Selbstverbund auf Basis der gesamten Relation noch schützend erscheint, da es die Eintragsanzahl vervielfacht, könnten lokale Selbstverbünde relativ eindeutige Informationen preisgeben, sofern man in etwa die Gruppengröße bzw. die Splitstellen kennt. Auch hier müsste man sich daher entscheiden, ob man das fragliche Attribut doch partitioniert oder die Korrelationsmengen besser gleich zusammenführt.

A	C	B.K1	B.K2	D
31.01.	–	B0	00	18
14.01.	–	AB	AA	24
05.01.	+	AA	B0	62
01.01.	+	00	AB	32
07.02.	+	00	AB	56
20.03.	+	00	AA	42
25.03.	–	AA	00	36
31.03.	–	AB	00	28

Abbildung 29: gelicte Relation mit lokal quasi-identifizierendem Mehrfachattribut B

Der einzig vertretbare Fall ist in Abbildung 30 zu sehen. Erst wenn sich Attributwerte auch gruppenintern häufen, würde die Mehrfachverwendung von Attributen akzeptabel sein. Einen Schutz für vereinzelte lokale Ausreißer gibt es jedoch nicht, sodass man dennoch ein paar singuläre korrelationsmengenübergreifende Infos ableiten könnte. Um den Ausreißereffekt vorbeugend kleinzuhalten, wäre es daher äußerst vorteilhaft, die Mehrfachattribute noch vor dem Slicen zu clustern bzw. nach ebendiesen vorzusortieren.

A	B	C.K1	C.K2	D
31.01.	B0	−	+	18
14.01.	AB	−	+	24
05.01.	00	+	−	62
01.01.	00	+	−	32
07.02.	A0	+	+	56
20.03.	A0	+	−	42
25.03.	A0	−	+	36
31.03.	B0	+	+	28

Abbildung 30: gelicte Relation mit einem Mehrfachattribut C, dessen Werte sich bereits zumeist lokal uneindeutig häufen

Zudem müsste noch zwangsläufig der Umgang mit der Namenseindeutigkeit geklärt werden, da derselbe Attributname nicht mehrfach vorkommen darf. Im Falle einer vertikalen Partitionierung stellt sich dieses Problem logischerweise nicht, aber es sei dennoch kurz angeschnitten. Sei die gewünschte Slice-Auswahl beispielsweise $V = \{\{A, B, C, D\}, \{C, D, E\}\}$. Die einfachste Lösung bestünde in einer Korrelationsgruppen-Indexzahl, was den Attributnamensraum exemplarisch zu $R = \{A, B, C1, D1, C2, D2, E\}$ umwandeln würde. Der Vorteil dieser Methode ist der Erhalt der Übersichtlichkeit, der Nachteil die sich kaum von selbst erschließende Zuordnung. Damit kommt sie zwar dem allgemeinen Geheimhaltungsaspekt zugute, eignet sich aber nur bedingt als theoretische Notation und könnte in der Praxis rasch zu Fehlablesungen verleiten. Eine alternative Lösung bestünde in der angehängten Namensverkettung von Korrelationsattributen – wahlweise alle oder reduziert auf die einmaligen Korrelationsattribute. In der Minimalschreibweise würde der Namensraum dann $R = \{A, B, C.AB, D.AB, C.E, D.E, E\}$ lauten. Jenseits kurzer Namen bzw. theoretischer Betrachtungen wird dies natürlich rasch unübersichtlich, sodass man auf Abkürzungen oder minimal eindeutige Präfixe ausweichen müsste.

Zuletzt sei noch angemerkt, dass davon ausgegangen wird, dass die Attributauswahl nach bestem Wissen getroffen wurde. Die zusammengestellte Auswahl wird somit nicht in Frage gestellt.

7.1.3 Ausführliche Formalisierung

Im Folgenden wird veranschaulicht, wie die genaue Zerteilung, Neuordnung und Zusammenführung der Basisrelation aussieht. Die horizontale Aufteilung muss per Definition vollständig disjunkt verlaufen. Formal heißt das:

$$r := \sigma_{H_1}(r) \cup ... \cup \sigma_{H_n}(r)$$

$$\sigma_{H_i}(r) \cap \sigma_{H_j}(r) = \{\} \quad \text{für} \quad \forall i, \forall j : \; i, j \in \{1, ..., n\} \quad \text{und} \quad i \neq j$$

Weiterhin gilt bei Splitstellensetzung nach dem Schema Top-down:

$$n := \left\lfloor \frac{|\,r\,|}{sliceSize} \right\rfloor$$

$$\begin{aligned} |\sigma_{H_1}(r)| &= sliceSize \\ ... &= sliceSize \\ |\sigma_{H_{n-1}}(r)| &= sliceSize \\ |\sigma_{H_n}(r)| &= sliceSize + (\,|\,r\,| \bmod sliceSize\,) \end{aligned}$$

Bei den H-Ausdrücken handelt es sich um Selektionsprädikate, durch die die disjunkte horizontale Teilung vorgenommen wird. Die vertikale Aufteilung muss wie schon erwähnt nicht zwingend vollständig disjunkt sein, für unseren Operator aber schon. Sie wird zwar mit dem Hintergrund gewählt, was miteinander korreliert und was nicht, lässt sich im Prinzip aber frei bestimmen. Die genaue Aufteilung wird durch die Parameter des Operators vorgegeben. Auf formaler Ebene bedeutet das:

$$R = V_1 \cup ... \cup V_m$$

$$V_i \cap V_j = \{\} \quad \text{für} \quad \forall i, \forall j : \; i, j \in \{1, ..., m\} \quad \text{und} \quad i \neq j$$

$$V = \{V_1, ..., V_m\}$$

Der spezifische Slicing-Prozess ist eine vielschrittige Anwendung einzelner Operatoren auf diverse gebildete Teilrelationen. Der genaue Prozess ist sowohl schrittweise formalisiert als auch visuell gebündelt in Form von Abbildung 31 zu finden. Da wir uns innerhalb der gängigen relationalen Algebra bewegen und etwas erzielen möchten, was nicht regulär dazugehört, ist die formale Einführung eines zusätzlichen Operators nötig.

ω-Operator: $\omega_L(r)$

Der Operator ω aus [HSS19] dient dazu, die Algebra um das Konzept der Sortierung zu erweitern. Sei r im Folgenden eine Relation und L eine Liste von Attributen aus r. Durch Anwendung von $\omega_L(r)$ wird r lexikographisch nach L sortiert. Da eine Relation hierbei strenggenommen in eine Liste überführt werden muss, ist es in der erweiterten relationalen Algebra zumeist die letzte Operation, die man ausführt. In diesem Falle aber nicht, da es einzig bei der Neuordnung von Einträgen helfen soll. Nach anschließender Zusammenführung aller Teilrelationen ist die Reihenfolge wieder belanglos. Eine Sortierung kann wahlweise aufsteigend, absteigend oder randomisiert ausgerichtet sein. Zum Bau des Slicing-Operators bedarf es im Speziellen der randomisierten Abfolge, was gleichbedeutend mit einer zufälligen Permutation ist. Es sei noch angemerkt, dass der Begriff Sortierung im Falle einer Randomisierung vorzugsweise zu vermeiden ist, da es einen inhärenten Widerspruch birgt.

Schritte

Ein Slicing-Vorgang setzt sich aus insgesamt fünf Einzelschritten zusammen. Der erste Schritt ist die horizontale Aufteilung der Basisrelation. Im anschließenden Schritt wird die vertikale Aufteilung vorgenommen. Im dritten Schritt erfolgt dann die Permutation. Im vierten und fünften Schritt geschieht die erneute Zusammenführung, zuerst vertikal, dann horizontal.

Schritt 1 :	$\forall i \in \{1,...,n\}$	$r_{1\langle i\rangle}$	:=	$\sigma_{H_i}(r)$
Schritt 2 :	$\forall i \in \{1,...,n\}, \forall j \in \{1,...,m\}$	$r_{2\langle i,j\rangle}$	:=	$\pi_{V_j}(r_{1\langle i\rangle})$
Schritt 3 :	$\forall i \in \{1,...,n\}, \forall j \in \{1,...,m\}$	$r_{3\langle i,j\rangle}$	:=	$\omega_{P_j}(r_{2\langle i,j\rangle})$
Schritt 4 :	$\forall i \in \{1,...,n\}$	$r_{4\langle i\rangle}$	:=	$r_{3\langle i,1\rangle} \bowtie_{ID} ... \bowtie_{ID} r_{3\langle i,m\rangle}$
Schritt 5 :		r_5	:=	$\bigcup_{i=1}^{n}(r_{4\langle i\rangle})$

Abbildung 31, von unten nach oben zu lesen, veranschaulicht den exemplarischen Prozess noch etwas kompakter. Da sich die ID-Vergabe schlecht einbinden ließ, wurde dessen Vergabe und Ausblendung weggelassen.

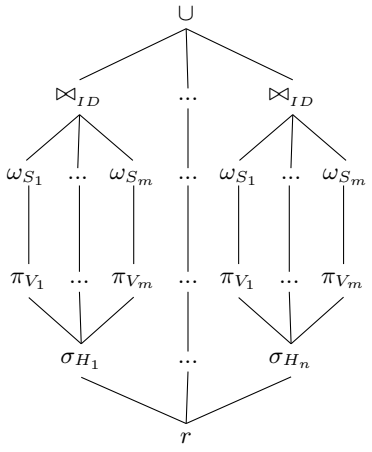


Abbildung 31: α^S-Operator

Darüber hinaus sei noch angemerkt, dass die Anordnung der Einzeloperatoren nicht zwingend ist. Sowohl beim Teilungs- als auch Zusammenführungsprozess ließe sich der horizontale mit dem vertikalen Schritt problemlos vertauschen. Folglich gäbe es gleich vier äquivalente Beschreibungsmöglichkeiten, was Abbildung 32 nochmals unterstreichen soll. Zur Veranschaulichung von α^S wurde also rein beispielhaft Möglichkeit A genutzt.

	Möglichkeit A	Möglichkeit B	Möglichkeit C	Möglichkeit D
Schritt 1	σ_H	σ_H	π_V	π_V
Schritt 2	π_V	π_V	σ_H	σ_H
Schritt 3	ω_S	ω_S	ω_S	ω_S
Schritt 4	$\bowtie_{ID}$	$\bigcup$	$\bowtie_{ID}$	$\bigcup$
Schritt 5	$\bigcup$	$\bowtie_{ID}$	$\bigcup$	$\bowtie_{ID}$

Abbildung 32: potenzielle Beschreibungsformen

7.1.4 Beispiel

Zum genauen Verständnis des α^S-Operators sei dessen Wirkungsweise nun an einem Beispiel verdeutlicht. Eine kurze inhaltliche Kontextualisierung ist in Abschnitt 2.7 zu finden.

Die Ausgangsrelation r setzt sich aus dem Schema $R = \{A, B, C, D\}$ zusammen. Ebendiese Relation gilt es mittels Slicing nun zu anonymisieren. Die vorliegenden Attribute werden in die drei Korrelationsmengen $V_1 = \{A\}$, $V_2 = \{B, C\}$ und $V_3 = \{D\}$ aufgeteilt. Die horizontale Splitgröße soll im Folgenden 3 betragen. Aus all diesen Forderungen ergibt sich dann folgender Ausdruck:

$$r_a := \alpha^S{}_{V=\{\{A\},\{B,C\},\{D\}\},sliceSize=3}(r)$$

Abbildung 33 zeigt an einem Datenbestand, wie ein exemplarisches Resultat aussehen könnte, da bekanntlich eine Vielzahl von Permutationen möglich ist.

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	−	32
20.03.	A0	+	36
25.03.	A0	+	42
31.03.	B0	−	56

$\Longrightarrow$

A	B	C	D
31.01.	00	+	18
05.01.	00	+	32
01.01.	B0	−	24
20.03.	A0	+	36
25.03.	B0	−	42
31.03.	A0	+	56

Abbildung 33: links r, rechts r_a

7.2 Schwächen von α^S-Operator je Fenstertyp

In diesem Abschnitt werden fallspezifisch die Schwächen des minimalistischen α^S-Operators hinsichtlich aller wesentlichen Fenstertypen beleuchtet.

7.2.1 Tupelfenster

Bei Gebrauch des minimalistischen Operators lassen sich bei Tupelfenstern abhängig vom Fenstertyp folgende Stromdaten-Probleme konstatieren:

Tupelfenster-Typ	P1	P2 (ggf. bei $c < a$)	P3 (ggf.)
stehend	nein	einmalig (existenziell)	gar keine Folgefenster
Tumbling	nein	durchgehend	bei $c < a$ durchgehend
Sliding	ja	durchgehend	grundsätzlich, da $\|hop\| := 1 < (a \geq 2))$
Jumping	ggf.	durchgehend	bei $\|hop\| < a$ durchgehend
Session	nein	durchgehend	bei $c < a$ durchgehend
einseitig wachsendes LM	ja	zu Beginn	bei $\|hop\| < a$
relativ wachsend	ggf.	zu Beginn	bei $\|hop_{fast}\| < a$
einseitig schrumpfendes LM	ja	irgendwann immer	gar kein Zuwachs
relativ schrumpfend	ggf.	irgendwann immer	bei $\|hop_{slow}\| < a$
beidseitig wachsendes LM	ja	zu Beginn	bei $\|hop_{past}\| + \|hop_{future}\| < a$
beidseitig schrumpfendes LM	ja	irgendwann immer	gar kein Zuwachs

Abbildung 34: Tupelfenster-Schwächen von α^S

Slicing ist für alle drei Problemarten anfällig. P1 ist verfahrenunabhängig durch die Überlappung bedingt, weshalb es sich genau wie beim bereits untersuchten Verrauschen verhält. Neu hinzugekommen sind das Mindestgrößenproblem P2 und Zuwachsproblem P3. Sie werden bei konfligierenden Minimalanforderungen verursacht und sind bei Tupelfenstern bereits zur Kompilierzeit bzw. mit der Stellung der Abfrage absehbar.

P1 - Mehrfachslicing

Das grundlegende Problemmuster von P1 ist dasselbe wie beim einfachen Verrauschen, stets bedingt durch die Schnittmengen. Die Auswirkungen sind allerdings andere, da es sich nunmehr um ein zeilenübergreifendes Verfahren handelt. Je nach Fenstertyp kann Mehrfachslicing dabei verschiedene Wahrscheinlichkeitsverteilungen hinsichtlich des ursprünglichen Eintrags erzeugen. Das folgende selbst eingeführte Stufenmodell versucht, diese Verteilungen näherungsweise zu klassifizieren. Hierbei wird von randomisiertem Slicing ausgegangen.

Stufen der Slicing-Sicherheit

Stufe 0 Diese Stufe entspricht einmaligem und gesamtheitlichem Slicing. Gesamtheitlich meint hier randomisiert als eine einzige riesige Gruppe, die auf Splitstellen verzichtet, und einmalig soll bedeuten, dass der Ursprungseintrag niemals erneut geslict wird. Mit zunehmender Größe wird dies natürlich impraktikabel und ist dementsprechend niemals streamingtauglich. Dennoch sei diese Stufe eingeführt, da sie so ziemlich den größtmöglichen Schutz offerieren kann.

Stufe 1 Diese Stufe entspricht einmaligem Slicing mit nicht öffentlich bekannten Splitstellen. Ein geslicter Eintragsteil ist potenziell auf a benachbarte Einträge zurückführbar, was eine 1/a-Gleichverteilung impliziert. Da die Splitstellen jedoch aus öffentlicher Sicht geheim sind und rein anhand der Daten nicht erkennbar sein sollten, sofern Zusatzwissen ausgeschlossen ist, können Angreifer diesbezüglich nur Mutmaßungen anstellen. Auch wenn die naheliegendsten Einträge allgemein wahrscheinlich sind, lässt sich probabilistisch keine genau Grenze ziehen. Diese Stufe verkörpert den typischen Anwendungsfall ohne Streaming-Kontext und stellt realistisch betrachtet den anzustrebenden Schutz dar.

Stufe 2 Diese Stufe entspricht Mehrfachslicing mit unveränderten bzw. synchron mitverschobenen Splitstellen, was bei $hop \bmod a = 0$ der Fall ist. Ein geslicter Eintragsteil ist wie bei Stufe 1 auf potenziell a benachbarte Einträge zurückführbar, was eine 1/a-Gleichverteilung impliziert. Der Unterschied ist hier, dass die Splitstellen zwar ebenfalls nicht öffentlich bekannt sind, jedoch sich selbst und damit auch die Slice-Gruppen durch das wiederholte Slicing indirekt offenbaren. An der zugrunde liegenden Verteilung selbst hat sich nichts geändert, allerdings findet nun eine gleichsetzende Eingrenzung der öffentlich vermuteten Verteilung statt. Äquivalent gesprochen ist diese Stufe gleichbedeutend mit einmaligem Slicing, bei dem die Splitstellen bewusst mitveröffentlicht wurden. Infolgedessen erleichtert es Angreifern zwar die Ausführung generalisierungstypischer Angriffe, der Minimalschutz ist jedoch weiterhin gegeben, sodass auch diese Stufe vertretbar sein kann. Sobald die Splitstellen bzw. Gruppen nach ausreichend Output-Beobachtung hinreichend erkennbar sind, gibt es für Angreifer keinen weiteren direkten Mehrwert, da die beobachtbare Verteilung gen unendlich nur noch zementiert wird. Hieran zeigt sich auf absurde Weise, dass blindes Mehrfachslicing (besonders hinsichtlich Landmarks) sogar aus Angreifersicht recht schnell zu einer reinen Ressourcenverschwendung verkommt.

Stufe 3 Diese Stufe entspricht Mehrfachslicing mit wandernden Splitstellen, welche auf Dauer eine $2a - 1$ große, distanzabhängig diskrete, pyramidenähnliche Verteilung mit gröberer Auflösung induzieren – mit der zufälligen Rekombination der Ausgangswerte als einer von mehreren gleich wahrscheinlichen Kandidaten im Zentrum. Dieser Effekt ist zu beobachten, falls $hop \bmod a \neq 0$ und $ggT(a, |hop|) \geq 2$ gilt. Der stufenförmige Auflösungsgrad lässt sich anhand von $ggT(a, |hop|)$ berechnen. Bei einer Slicing-Mindestgröße von $a = 12$ erzeugt ein konstanter Hop von 3, 9 oder 15 beispielsweise einen Auflösungsgrad von 3; ein Hop von 4, 8 oder 16 würde wiederum einen Auflösungsgrad von 4 zur Folge haben; ein Hop von 6, 18 oder -6 würde einen Auflösungsgrad von 6 erzeugen. Diese Stufe ist als Bindeglied zwischen Stufe 2 und 4 anzusehen, welche so gesehen nur die beiden exkludierten äußeren Extremfälle sind: Bei Stufe 4 betrüge der Auflösungsgrad nämlich 1, wodurch sich ein klarer Topkandidat herausbildet, der sich am häufigsten beobachten ließe; bei Stufe 2 würde die gröbste Auflösung wiederum dafür sorgen, dass die pyramidenähnliche Stufenverteilung kollabiert bzw. in eine Gleichverteilung mündet. Diese Stufe ist nicht mehr tolerierbar, da die angestrebte Slicinggröße quasi um ein Vielfaches geteilt wird. Nichtsdestotrotz lässt sich positiv attestieren, dass der Ursprungseintrag aufgrund mehrerer potenzieller Spitzenkandidanten weiterhin uneindeutig bleibt, was einem Mindestschutz der Slice-Größe 2 gleichkommt.

Stufe 4 Diese Stufe entspricht Mehrfachslicing mit wandernden Splitstellen, welche auf Dauer eine $2a-1$ große, distanzabhängig diskrete Pyramidenverteilung induzieren – mit der

zufälligen Rekombination der Ausgangswerte als wahrscheinlichstes Zentrum. Dieser Effekt ist zu beobachten, falls hop mod $a \neq 0$ und $ggT(a, |hop|) = 1$ gilt. Diese Stufe ist keinesfalls mehr tolerierbar, da sich Ursprungseinträge mit genügend Aufwand faktisch rekonstruieren ließen. Das universelle Musterbeispiel dafür ist das Sliding-Fenster, wo die Splitstellen mit jedem Folgefenster kollektiv und in dieselbe Richtung um einen Eintrag weiterrücken. Auch in den anderen Fällen wird dieselbe Verteilung wie bei $|hop| = 1$ induziert, nur ist das zyklische Überlappungsmuster weniger offensichtlich. Abschließend sei noch angemerkt, dass es natürlich ein Mindestmaß an Beobachtbarkeit bzw. Verweildauer geslicter Einträge bedarf, damit Angreifer überhaupt aussagekräftige Verteilungen auf Basis der randomisieren Samples konstruieren können.

Stufe 5 In dieser Stufe sind Originaleinträge schutzlos offengelegt, was gleichbedeutend mit einer obsoleten Slice-Größe von $a = 1$ ist. Dieses Extrem lässt sich nur über die Falsch- oder Nichtverwendung des Operators erreichen.

Sicherheit je Tupelfenster-Fenstertyp

Es sei angenommen, dass die Splitstellen bzw. die Slice-Mindestgröße von offizieller Seite geheimgehalten werden. Außerdem wird der Einfluss der übergroßen (letzten) Gruppe, welche überschüssige Einträge aufnimmt, auf die Verteilungen vernachlässigt.

stehend	Stufe 0 bis 1
Tumbling	Stufe 1
Sliding	Stufe 4
Jumping	Stufe 1 bis 4 Stufe 1 bei $\|hop\| = c$ (entspricht Tumbling) Stufe 2 bei hop mod $a = 0$ Stufe 3 bei hop mod $a \neq 0$ und $ggT(a, \|hop\|) \geq 2$ Stufe 4 bei hop mod $a \neq 0$ und $ggT(a, \|hop\|) = 1$ (enthält Sliding)
Session	Stufe 1
einseitig wachsendes LM	Stufe 2 bis 4 bei $hop > 0$ (zukunftsgerichteter Aufbau): Stufe 2 bei $hop < 0$ (vergangenheitsgerichteter Aufbau): Stufe 2 bei hop mod $a = 0$ Stufe 3 bei hop mod $a \neq 0$ und $ggT(a, \|hop\|) \geq 2$ Stufe 4 bei hop mod $a \neq 0$ und $ggT(a, \|hop\|) = 1$ einfacher Verbesserungsvorschlag: Beim vergangenheitsgerichteten Aufbau könnte eine umgekehrte Splitstellensetzung ratsam sein, die am Relationenende ansetzt und dann nach dem Schema Bottom-up verfährt. Der eventuelle Überschuss käme dann am Relationenanfang zustande. Auf diese Weise würde ebenfalls Stufe 2 gewährleistet sein.
relativ wachsend	Stufe 1 bis 4 Stufe 1 bei $\|hop_A\| \geq c_{latest}$ und $hop > 0$ (erfordert mitwachsende Hops) oder $\|hop_E\| \geq c_{latest}$ und $hop < 0$ (erfordert mitwachsende Hops) Stufe 2 bei hop_A mod $a = 0$ Stufe 3 bei hop_A mod $a \neq 0$ und $ggT(a, \|hop_A\|) \geq 2$ Stufe 4 bei hop_A mod $a \neq 0$ und $ggT(a, \|hop_A\|) = 1$ Anmerkung: Bei $hop > 0$ kommt hop_A die Rolle des langsameren Hops zu, bei $hop < 0$ übernimmt hop_E die des langsameren Hops. c_{latest} gibt die jeweils aktuelle Fensterkapazität an. Stufe 1 kann auf Dauer nur durch eine dynamische wachsende Mitanpassung der Hops gewährleistet werden.

einseitig schrumpfendes LM	Stufe 2 bis 4 bei $hop < 0$ (vergangenheitsgerichteter Abbau): Stufe 2 bei $hop > 0$ (zukunftsgerichteter Abbau): Stufe 2 bei hop mod $a = 0$ Stufe 3 bei hop mod $a \neq 0$ und $ggT(a, \lvert hop \rvert) \geq 2$ Stufe 4 bei hop mod $a \neq 0$ und $ggT(a, \lvert hop \rvert) = 1$ einfacher Verbesserungsvorschlag: Bottom-up Splitstellensetzung beim zukunftsgerichteten Abbau, sodass ebenfalls Stufe 2 gewährleistet sein würde.
relativ schrumpfend	Stufe 1 bis 4 Stufe 1 bei $\lvert hop_A \rvert \geq c_{init}$ und $hop > 0$ oder $\lvert hop_E \rvert \geq c_{init}$ und $hop < 0$ Stufe 2 bei hop_A mod $a = 0$ Stufe 3 bei hop_A mod $a \neq 0$ und $ggT(a, \lvert hop_A \rvert) \geq 2$ Stufe 4 bei hop_A mod $a \neq 0$ und $ggT(a, \lvert hop_A \rvert) = 1$ Anmerkung: Bei konstanten Hops ist es nicht möglich, dass der schnellere Hop einen Betrag von 1 hat. Der langsamere Hop kann die Sprungweite von nur einem Tupel nicht unterbieten, da es sich sonst nicht mehr um relatives Schrumpfen handeln würde. c_{init} gibt die initiale Fensterkapazität an, die nötig wäre, damit session-ähnliches Schrumpfen auf Stufe 1 passieren würde. Im Gegensatz zum relativ schrumpfenden LM kann Stufe 1 hier bereits mit konstanten Hops gewährleistet werden. Eine dynamische Mitverkleinerung der Hops ist bloß nötig, um den inaktiven Raum noch kleiner zu halten oder um tumbling-ähnliches Schrumpfen zu erzielen.
beidseitig wachsendes LM	Stufe 2 bis 4 Stufe 2 bei hop_A mod $a = 0$ Stufe 3 bei hop_A mod $a \neq 0$ und $ggT(a, \lvert hop_A \rvert) \geq 2$ Stufe 4 bei hop_A mod $a \neq 0$ und $ggT(a, \lvert hop_A \rvert) = 1$ mit $hop_A \leq -1$ als ausbauender Hop des Fensteranfangs einfacher Verbesserungsvorschlag: Man könnte zwei horizontale Partitionen einführen: eine Partition für vergangenheitsgerichtete und eine für zukunftsgerichtete Einträge. Die zukunkftsgerichtete Partition würde die regulären Top-down-Splitstellen kriegen, die vergangenheitsgerichtete Partition die bottom-up-Splitstellen. Auf diese Weise könnte man Stufe 2 gewährleisten.
beidseitig schrumpfendes LM	Stufe 2 bis 4 Stufe 2 bei hop_A mod $a = 0$ Stufe 3 bei hop_A mod $a \neq 0$ und $ggT(a, \lvert hop_A \rvert) \geq 2$ Stufe 4 bei hop_A mod $a \neq 0$ und $ggT(a, \lvert hop_A \rvert) = 1$ mit $hop_A \geq 1$ als abbauender Hop des Fensteranfangs einfacher Verbesserungsvorschlag: Man könnte von vornherein das Zentrum bestimmen, wo der beidseitige Abbau enden würde. An diesem Punkt könnte man die Relation dann partitionieren und wie beim Verbesserungsvorschlag des beidseitig wachsenden Landmarks verfahren. Auf diese Weise ließe sich allgemein Stufe 2 gewährleisten.

Der maßgeblich entscheidende Hop ist hier immer die Fensteranfangs-Sprungweite hop_A, weil die Splitstellen nach dem Schema Top-down gesetzt werden. Würde Splitstellen nach dem Schema Bottum-up gesetzt werden, wäre stattdessen die Fensterende-Sprungweite hop_E entscheidend.

P2 - Unterschreitung der Slice-Mindestgröße

Bei konstanter Fensterkapazität ist das Problem existenzbedrohend, da eine permanente Unterschreitung eine ständige vorsorgliche Ergebnisunterdrückung bedeuten würde, was quasi der leeren Menge gleichkommt. Dies würde im Wesentlichen zwar nur sehr kleine Fenster betreffen, aber dennoch einen herben Verlust bedeuten, da es Kleinstanfragen grundsätzlich verunmöglichen würde.

Bei wachsender Fensterkapazität wird die Mindestanforderung nur bei einigen Fenstern zu Beginn verletzt, die man durch vorsorgliche Mechanismen unterdrücken müsste. Auf lange

Sicht erscheint der anfängliche Verlust im Verhältnis nichtig und könnte daher vernachlässigt werden.

Bei schrumpfender Fensterkapazität werden ggf. nur einige Fenster am Ende verletzt, die man vorsorglich unterdrücken müsste, was einem etwas früheren Abbruch gleichkommt. Der Verlust steht dabei im Verhältnis zum Startfenster und wäre bei einem kleinen Startfenster dementsprechend groß. Aus Anwendungssicht ist dies jedoch relativ unwahrscheinlich, da möglichst langlebige Analysen anvisiert werden, was im schrumpfenden Fall immer große Startfenster erfordert. Bei einem einmaligen langlebigen Abbau wäre der unterdrückte Verlust daher so gut wie vernachlässigbar. Dennoch ist dies nicht in jedem Anwendungsfall ratsam. Beim periodischen Abbau besonders kleiner Fenster skaliert der Verlust nämlich mit, was eine ständige, nicht unbedenkliche Verfälschung bedeutet.

P3 - Unterschreitung des Mindestzuwachses

Bei naivem Gebrauch ist Problem P3 nicht per se zu beobachten. Es ergibt sich vielmehr erst als weiteres Problem, sobald man P1 und P2 zu lösen versucht. Das Problem verhält sich dann analog zu P2: Bei konstantem Tupelzuwachs besteht das Problem ggf. durchgehend, bei steigendem Tupelzuwachs nur anfänglich, bei schrumpfendem Tupelzuwachs am Ende immer. Da der Zuwachs bei Tupelfenstern jedoch allgemein konstant gehalten wird, ist P3 dadurch ein existenzielles Problem für all jene Fensteranfragen, die die allerfeinsten Entwicklungen analysieren wollen.

7.2.2 Zeitfenster

Bei Zeitfenstern kommt erschwerend hinzu, dass die Tupelentwicklung ungewiss ist bzw. erst zur Laufzeit bekannt wird. Dementsprechend stellt Abb. 35 eine zusätzliche Verschärfung von Abb. 34 dar. P1 ist verfahrenunabhängig durch die Überlappung bedingt, weshalb es diesbezüglich keine Veränderung gibt. Die Verschärfungen betreffen das Mindestkapazitätsproblem P2 und das Mindestzuwachsproblem P3.

Zeitfenster-Typ	P1	P2 (ggf. bei $c < a$)	P3 (ggf.)
stehend	nein	einmalig (existenziell)	gar keine Folgefenster
Tumbling	nein	stets ungewiss	stets ungewiss
Sliding	ja	stets ungewiss	grundsätzlich, da $\|hop\| := 1 < (a \geq 2))$
Jumping	ggf.	stets ungewiss	stets ungewiss
Session	nein	stets ungewiss	stets ungewiss
einseitig wachsendes LM	ja	zu Beginn	stets ungewiss
relativ wachsend	ggf.	immer unwahrscheinlicher	stets ungewiss
einseitig schrumpfendes LM	ja	irgendwann immer	gar kein Zuwachs
relativ schrumpfend	ggf.	immer wahrscheinlicher	stets ungewiss
beidseitig wachsendes LM	ja	zu Beginn	stets ungewiss
beidseitig schrumpfendes LM	ja	irgendwann immer	gar kein Zuwachs

Abbildung 35: Zeitfenster-Schwächen von α^S

7.3 Erweiterter Slicing-Operator α^{S+}

7.3.1 Aufbau

Der verfeinerte Slicing-Operator setzt sich folgendermaßen zusammen:

$$\alpha^{S+}{}_{V=\{V_1,\ldots,V_n\},sliceSize,rest\,[;pattern,skip,cut;reverse]}(r)$$

α^{S+}	Bezeichner des Operators α symbolisiert Anonymisierung S bezeichnet das genaue Verfahren (Slicing) $+$ als abgrenzender Indikator für den erweiterten Operator
V	Menge der Korrelationsmengen
V_i	Korrelationsmenge, $V_i \subseteq R$
$sliceSize$	Slicing-Größe
r	Basisrelation
	neue Pflichtparameter:
$rest$	Umgang mit überschüssigen Einträgen bei einer unvollständigen Gruppe • include: bestehende Gruppen vergrößern • suppress: unvollständige Gruppe weglassen • unchanged: unverändert mit ausgeben (zur internen Speicherung bzw. Verarbeitung in späteren Iterationsschritten; NICHT für Direktausgaben vorgesehen) ○ potenzielle Default-Präferenz: $rest :=$ include
	neue optionale Parameter:
$(skip)$	Anzahl unveränderter Einträge zu Beginn, deren Slicing übersprungen wird ○ Default: $skip := 0$
$(pattern)$	Muster zur Spiltstellensetzung • Top-down: oben beginnend, potenzieller Rest am Ende • Bottom-up: unten beginnend, potenzieller Rest am Anfang • ... (weitere Alternativen denkbar) ○ Default: $pattern :=$ Top-down Anmerkung: Sondergröße bei $rest :=$ include reicht von $sliceSize$ bis $2 \cdot sliceSize - 1$
(cut)	manuell abgezählte Selektion, um das untere Eintragsende abschließend eigenständig beschneiden zu können, was das Trimmen einer geslicten Gruppe ermöglichen soll • positive Zahl: Auswahl der ersten n Einträge • negative Zahl: Ausblenden der letzten n Einträge ○ Default: Auswahl aller Einträge, quasi $cut := \lvert r \rvert \equiv -0$ (Achtung, hier gilt: $0 \not\equiv -0$) Anmerkung: bei Anwendung auf Stromdaten würde man die Fenstergröße a priori etwas größer ansetzen und diesen Überschuss dann durch das Abschneiden wieder verbergen; hierzu sei auf die Formel unter dem P3-Abschnitt von Kapitel 5 verwiesen.
$(reverse)$	optionale Richtungsumkehr der Arbeitsweise ○ Default: $reverse := false$ Anmerkung: Die Standard-Arbeitsweise des Operators ist von oben nach unten ausgelegt. In Bezug auf Stromdaten wird im Folgenden nämlich angenommen, dass sich die ältesten Einträge eines Fensters am Anfang beziehungsweise die neusten Einträge am unteren Ende befinden. Sollte eine Arbeitsweise in gegenteiliger Richtung nötig sein, würde die komplette Funktionsweise quasi gespiegelt bzw. auf den Kopf gestellt werden.

7.3.2 Annahmen und Einschränkungen

Die Annahmen und Einschränkungen haben sich gegenüber 7.1.2 im Kern nicht verändert. Sie werden höchstens an einigen Stellen explizit überschrieben, wenn der Operator um Alternativen bereichert wurde.

7.3.3 Beispiele

Fallbeispiel 1: Default

Das unbestimmte bzw. noch nicht vollständig konkretisierte Defaultverhalten des Operators entspricht in verkürzter Form:

$$\alpha^{S+}{}_{V,sliceSize,rest}(r) \equiv \alpha^{S+}{}_{V,sliceSize,rest;skip=0,pattern=TopDown,cut=-0;reverse=false}(r)$$

Im Falle von $rest := include$ ist das Verhalten des α^{S+}-Operators sogar deckungsgleich mit dessen Vorläufer α^{S}, was nochmals durch Abb. 36 veranschaulicht werden soll.

$$\alpha^{S+}{}_{V,sliceSize,rest=include}(r) \Longleftrightarrow \alpha^{S}{}_{V,sliceSize}(r)$$

$$r_2 := \alpha^{S+}{}_{V=\{\{A\},\{B,C\},\{D\}\},sliceSize=3,rest=include}(r_1)$$

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	−	32
20.03.	A0	+	36
25.03.	A0	+	42
31.03.	B0	−	56
02.05.	A0	+	49

$\Longrightarrow$

A	B	C	D
31.01.	00	+	18
05.01.	00	+	32
01.01.	B0	−	24
20.03.	A0	+	36
25.03.	A0	+	42
31.03.	B0	−	49
02.05.	A0	+	56

Abbildung 36: links r_1, rechts r_2

Fallbeispiel 2: *rest*-Parameter

In Abbildung 37 werden die drei verschiedenen Modi des *rest*-Parameters veranschaulicht. Aufgrund einer Relationengröße von $|r| = 8$ und einer Slice-Größe von 3 ergibt sich aufgrund von $8 \ mod \ 3 = 2$ ein Rest von zwei Einträgen. Zunächst sei nochmals angemerkt, dass als Default-Muster zudem $pattern = TopDown$ festgelegt worden ist, sodass in allen drei Fällen die untersten beiden Einträge den unvollständigen Rest bilden. Relation r_2 zeigt die Eingliederung des Restes in die letzte vollständige Slice-Gruppe, die wie in diesem Fall auf eine Übergröße von bis zu fünf Einträgen anwachsen kann; dieses Vorgehen wurde auch schon beim damaligen α^{S}-Operator präferiert. Relation r_3 zeigt die Unterdrückung der untersten beiden Einträge. Das Unterdrücken findet noch vor dem Anonymisieren statt, sodass ein Hineinslicen ausgeschlossen ist; auf diese Weise grenzt sich der Modus auch klar von einem nachträglich erfolgenden Abschneiden durch den *cut*-Parameter ab. Relation r_4 zeigt, wie die untersten beiden Einträge unverändert übernommen wurden.

$$r_2 := \alpha^{S+}{}_{V=\{\{A\},\{B,C\},\{D\}\},sliceSize=3,rest=include}(r_1)$$
$$r_3 := \alpha^{S+}{}_{V=\{\{A\},\{B,C\},\{D\}\},sliceSize=3,rest=suppress}(r_1)$$
$$r_4 := \alpha^{S+}{}_{V=\{\{A\},\{B,C\},\{D\}\},sliceSize=3,rest=unchanged}(r_1)$$

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	−	32
20.03.	A0	+	36
25.03.	A0	+	42
31.03.	B0	−	56
02.05.	A0	+	49
08.05.	A0	+	58

$\Longrightarrow$

A	B	C	D
31.01.	00	+	18
05.01.	00	+	32
01.01.	B0	−	24
20.03.	A0	+	36
25.03.	A0	+	42
31.03.	A0	+	56
02.05.	B0	−	58
08.05.	A0	+	49

A	B	C	D
31.01.	00	+	18
05.01.	00	+	32
01.01.	B0	−	24
20.03.	A0	+	36
25.03.	B0	−	42
31.03.	A0	+	56

A	B	C	D
31.01.	00	+	18
05.01.	00	+	32
01.01.	B0	−	24
20.03.	A0	+	36
25.03.	B0	−	42
31.03.	A0	+	56
02.05.	A0	+	49
08.05.	A0	+	58

Abbildung 37: von links nach rechts: r_1 bis r_4

Fallbeispiel 3: *skip*-Parameter

Die Abbildung 38 veranschaulicht den exemplarischen Gebrauch des *skip*-Parameters. Das Überspringen an sich verläuft wie beim *skip*-Parameter des α^{D+}-Operators, nur dass das daran anschließende Anonymisierungsverfahren dann ein anderes ist. Ein nützlicher Nebeneffekt des Überspringens besteht darin, dass es eine Verschiebung der Splitstellen bewirkt, sobald $skip \ mod \ sliceSize \neq 0$ gilt. Bei Relation r_2 mit $skip = 4$ werden die ersten vier Einträge, die in durchgehendem Hellgrau hervorgehoben wurden, unverändert übernommen. Die Splitstellen der geslicten Gruppen sind dabei deckungsgleich mit dem Standardfall ohne *skip*, sprich $skip = 0$. Bei Relation r_3 mit $skip = 3$ werden die ersten drei Einträge unverändert übernommen. In diesem Fall findet eine Verschiebung der Splitstellen statt. Erneut sei deutlich darauf hingewiesen, dass der privacy-konforme Gebrauch des Parameters hier vollständig dem Nutzer obliegt. Falls die hier übersprungenen Einträge nicht a priori schon einmal anonymisiert wurden und die Relationen r_2 und r_3 veröffentlicht werden sollten, hätte man diese Einträge, auch wenn es für außenstehende Betrachter natürlich nicht erkennbar ist, beim Schutzvorgang übergangen.

$$r_2 := \alpha^{S+}{}_{V=\{\{A\},\{B,C\},\{D\}\},sliceSize=2,rest=include;skip=4}(r_1)$$
$$r_3 := \alpha^{S+}{}_{V=\{\{A\},\{B,C\},\{D\}\},sliceSize=2,rest=include;skip=3}(r_1)$$

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	−	32
20.03.	A0	+	36
25.03.	A0	+	42
31.03.	B0	−	56
02.05.	A0	+	49
08.05.	A0	+	58
09.05.	B0	−	28
24.06.	AA	−	19

$\Longrightarrow$

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	−	32
20.03.	A0	+	36
25.03.	B0	−	56
31.03.	A0	+	42
08.05.	A0	+	58
02.05.	A0	+	49
09.05.	AA	−	28
24.06.	B0	−	19

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	−	32
20.03.	A0	+	42
25.03.	A0	+	36
02.05.	B0	−	49
31.03.	A0	+	56
24.06.	A0	+	28
09.05.	B0	−	19
08.05.	AA	−	58

Abbildung 38: von links nach rechts: r_1 bis r_3

Korrekterweise würde der *skip*-Parameter eingesetzt werden, um an bereits geslicte Einträge anzuschließen, was Abb. 39 verdeutlichen soll.

$$r_2 := \alpha^{S+}{}_{V=\{\{A\},\{B,C\},\{D\}\},sliceSize=4,rest=include}(r_1)$$
$$r_4 := r_2 \cup r_3$$
$$r_5 := \alpha^{S+}{}_{V=\{\{A\},\{B,C\},\{D\}\},sliceSize=4,rest=include;skip=4}(r_4)$$

A	B	C	D
20.03.	00	+	18
31.01.	00	+	36
05.01.	B0	−	32
01.01.	A0	+	24

$\Longrightarrow$

A	B	C	D
20.03.	00	+	18
31.01.	00	+	36
05.01.	B0	−	32
01.01.	A0	+	24
25.03.	A0	+	42
31.03.	B0	−	56
02.05.	A0	+	49
08.05.	A0	+	58

$\Longrightarrow$

A	B	C	D
20.03.	00	+	18
31.01.	00	+	36
05.01.	B0	−	32
01.01.	A0	+	24
25.03.	A0	+	49
31.03.	A0	+	56
02.05.	A0	+	42
08.05.	A0	−	58

Abbildung 39: von links nach rechts: r_2, r_4 und r_5

Fallbeispiel 4: *cut*-Parameter

Die Abbildung 40 veranschaulicht den exemplarischen Gebrauch des *cut*-Parameters. Das Abschneiden erlaubt das abgezählte Ausblenden der letzten Einträge und ist üblicherweise zum Verkürzen der letzten Gruppe gedacht. In gewisser Weise kann der *cut*-Parameter als entkoppelter vierter *rest*-Modus begriffen und genutzt werden, der dem überschüssigen Rest ein potenzielles Hineinslicen gestattet und erst danach den ungewollten Überschuss unterdrückt. Relation r_2 repräsentiert hierbei den internen Zwischenschritt, bevor dann wie in r_3 das abschließende Abschneiden erfolgt.

$$r_2 := \alpha^{S+}{}_{V=\{\{A\},\{B,C\},\{D\}\},sliceSize=3,rest=include,cut=-2}(r_1)$$
$$r_3 := \alpha^{S+}{}_{V=\{\{A\},\{B,C\},\{D\}\},sliceSize=3,rest=include}(r_1)$$

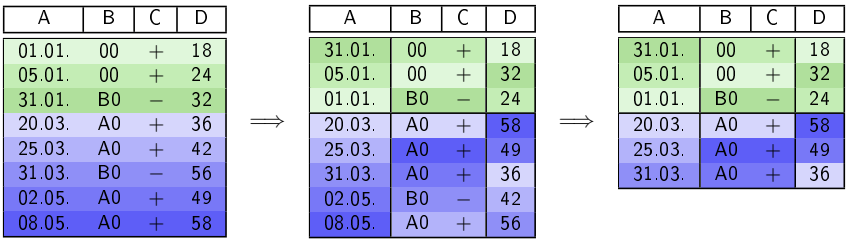


Abbildung 40: von links nach rechts: r_1 bis r_3

Besonders nützlich kann sich dieser Parameter bei Datenströmen erweisen, was das hieran anschließende Fallbeispiel veranschaulichen soll. Er erlaubt nämlich das Im-Hintergrund-Halten von zusätzlichen Werten, die noch nicht ergebnis-, aber schon berechnungsrelevant sind.

Fallbeispiel 5: Stromdaten am Beispiel eines Jumping-Tupelfensters

Die Abbildungen 41 und 42 veranschaulichen in zehn Schritten die exemplarische Nutzung des α^{S+}-Operators zum Slicen eines Jumping-Tupelfensters mit einer Ausgabekapazität von sechs Einträgen, einer Slicing-Größe von vier Einträgen und einer zukunftsgerichteten Sprungweite von drei Einträgen. Um eine konsistent geslicte Ausgabekapazität von sechs Einträgen zu erzielen, ist eine interne Erweiterung auf neun Einträge nötig, wobei die letzten drei nie weiter- bzw. ausgegeben werden. Die Relation r_1 repräsentiert das unanonymisierte Startfenster, das es zunächst (bis auf den Rest) vollständig zu anonymisieren gilt. In Folgeschritten sind nur noch alle neu hinzugekommene Einträge, die eine vollständige Gruppe ausbilden, zu slicen. Die ungeraden Relationen zeigen im Resultat jeweils die Fensterverschiebungen, die geraden wiederum gezieltes Permutieren. Generell ist zwischen der intern gespeicherten Relation und dem weiter- bzw. ausgegebenen Ergebnis zu unterscheiden. Schritte, die eine Ergebnisausgabe repräsentieren, sind am Hochkomma gemäß der Notation $r_n{}'$ zu erkennen. Intern gespeicherte Relationen r_n kommen dementsprechend ohne einen Oberstrich aus. Aus Pragmatik und zum Verständnis wurde darauf verzichtet, zusätzlich noch einen separaten Cut-Operator einzuführen. Es sei deshalb im Folgenden garantiert, dass das randomisierte Permutieren bei r_n und $r_n{}'$ den gleichen Ausgang nimmt.

$$r_2 := \alpha^{S+}{}_{V=\{\{A\},\{B\}\},sliceSize=4,rest=unchanged}(r_1)$$
$$r_2{}' := \alpha^{S+}{}_{V=\{\{A\},\{B\}\},sliceSize=4,rest=unchanged;cut=-3\equiv 6}(r_1)$$

$$r_3 := JUMPING_{type=tuple,hop=3}(r_2)$$
$$r_4 := \alpha^{S+}{}_{V=\{\{A\},\{B\}\},sliceSize=4,rest=unchanged;skip=5}(r_3)$$
$$r_4{}' := \alpha^{S+}{}_{V=\{\{A\},\{B\}\},sliceSize=4,rest=unchanged;skip=5,cut=-3}(r_3)$$

$$r_5 := JUMPING_{type=tuple,hop=3}(r_4)$$
$$r_6 := \alpha^{S+}{}_{V=\{\{A\},\{B\}\},sliceSize=4,rest=unchanged;skip=6\equiv 9}(r_5) \equiv r_5$$
$$r_6{}' := \alpha^{S+}{}_{V=\{\{A\},\{B\}\},sliceSize=4,rest=unchanged;skip=6,cut=-3}(r_5)$$

$$r_7 := JUMPING_{type=tuple,hop=3}(r_6)$$
$$r_8 := \alpha^{S+}{}_{V=\{\{A\},\{B\}\},sliceSize=4,rest=unchanged;skip=3}(r_7)$$
$$r_8{}' := \alpha^{S+}{}_{V=\{\{A\},\{B\}\},sliceSize=4,rest=unchanged;skip=3,cut=-3}(r_7)$$

$$r_9 := JUMPING_{type=tuple,hop=3}(r_8)$$
$$r_{10} := \alpha^{S+}{}_{V=\{\{A\},\{B\}\},sliceSize=4,rest=unchanged;skip=4}(r_9)$$
$$r_{10}{}' := \alpha^{S+}{}_{V=\{\{A\},\{B\}\},sliceSize=4,rest=unchanged;skip=4,cut=-3}(r_9)$$

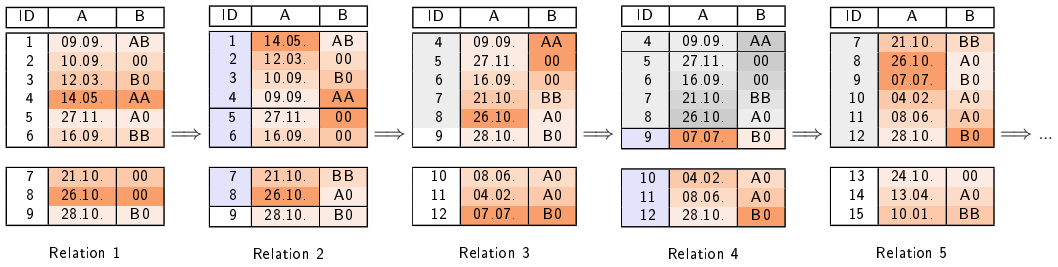


Abbildung 41: von links nach rechts: r_1 bis r_5 (Teil 1 von 2)

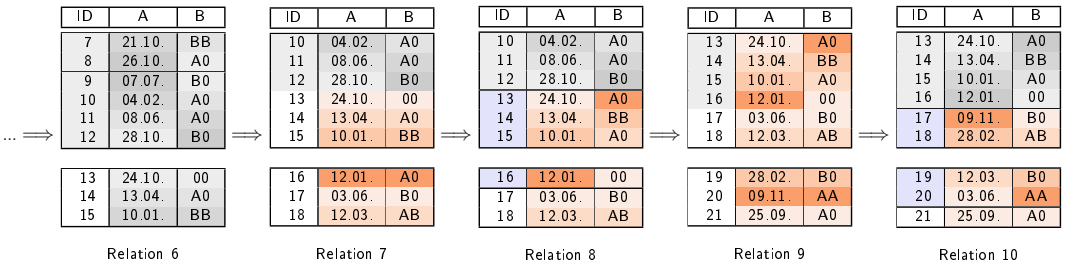


Abbildung 42: von links nach rechts: r_6 bis r_{10} (Teil 2 von 2)

Fallbeispiel 6: Umkehr der Arbeitsweise

Falls es das Anwendungsumfeld erfordert, dass sich die zu überspringenden Einträge am unteren Relationenende befinden, sei abschließend auch noch eine anregende Notationen zur Umkehr der Arbeitsweise vorgeschlagen. Aufgrund der Spiegelung würde der *skip*-Parameter am unteren Ende ansetzen und der *cut*-Parameter die Relation nun am oberen Ende beschneiden. Der α^{D+} in Abschnitt 6.3.3 ließ diesbezüglich noch relativ freiheitlich drei unterschiedliche Ansätze zu. Beim erweiterten Slicing ist der *skip*-Parameter jedoch mit dem am anderen Relationenende ansetzenden *cut*-Parameter gekoppelt, was einen zweiten oder richtungsumkehrbaren *skip*-Paramter verhindert bzw. schwer erklär- und einbindbar macht. Aus diesem Grunde sei als intuitive Notation auch hier einfach ein zusätzlicher optionaler *reverse*-Parameter bevorzugt. Abschließend sei noch angemerkt, dass die Splitstellenmuster

Top-down und Bottom-up aufgrund der Umkehr nun missverständlich aufgefasst werden könnten. Es wäre daher notwendigerweise noch zu explizieren, ob *top* und *bottom* positionell das konkrete Relationenende meinen oder sie sich davon losgelöst auf den bearbeitungsbezogen, umkehrbaren Ursprung beziehen. Im Grunde legt es sogar den Schluss nahe, dass das einseitig gerichtete Definieren von Mustern völlig ausreicht, sobald man das umgekehrte Muster auch indirekt über *reverse* erzielen kann. In diesem Falle wäre es natürlich Top-down, da das Beschneiden der untersten Gruppe funktionell mit dem *cut*-Parameter abgestimmt ist. Die Abbildung 43 veranschaulicht nochmals exemplarisch, wie eine umgekehrte Arbeitsweise über $reverse := true$ aussehen könnte.

$$\text{Default: } \alpha^{S+}{}_{V,sliceSize,rest}(r) \equiv \alpha^{S+}{}_{V,sliceSize,rest;[...];reverse=false}(r)$$

$$r_2 := \alpha^{S+}{}_{V=\{\{A\},\{B,C\},\{D\}\},sliceSize=2,rest=include;skip=3;reverse=true}(r_1)$$
$$r_3 := \alpha^{S+}{}_{V=\{\{A\},\{B,C\},\{D\}\},sliceSize=2,rest=include;skip=3,cut=-2;reverse=true}(r_1)$$

A	B	C	D
24.06.	AA	−	19
09.05.	B0	−	28
08.05.	A0	+	58
02.05.	A0	+	49
31.03.	B0	−	56
25.03.	A0	+	42
20.03.	A0	+	36
31.01.	B0	−	32
05.01.	00	+	24
01.01.	00	+	18

$\Longrightarrow$

A	B	C	D
08.05.	AA	−	58
09.05.	B0	−	19
24.06.	A0	+	28
31.03.	A0	+	56
02.05.	B0	−	49
25.03.	A0	+	36
20.03.	A0	+	42
31.01.	B0	−	32
05.01.	00	+	24
01.01.	00	+	18

$\Longrightarrow$

A	B	C	D
24.06.	A0	+	28
31.03.	A0	+	56
02.05.	B0	−	49
25.03.	A0	+	36
20.03.	A0	+	42
31.01.	B0	−	32
05.01.	00	+	24
01.01.	00	+	18

Abbildung 43: von links nach rechts: r_1 bis r_3

8 Generalisierungs-Operator

Dieses Kapitel widmet sich der operationalen Verwendung von Generalisierung hinsichtlich k-Anonymität. Zuerst wird in Abschnitt 8.1 der minimalistisch gehaltene Generalisierungs-Operator α^G vorgestellt, der aus Kapitel 4 der damals selbstverfassten Bachelorarbeit [Kle20] stammt. Als Nächstes werden in Abschnitt 8.2 die Schwächen dieser minimalistischen Variante hinsichtlich Fensterverarbeitung ergründet. Abschließend wird in Abschnitt 8.4 der erweiterte Generalisierungs-Operator α^{G+} eingeführt.

8.1 Minimalistischer Generalisierungs-Operator α^G

8.1.1 Aufbau

Der minimalistische Generalisierungs-Operator setzt sich folgendermaßen zusammen:

$$\alpha^G{}_{X,\,k}(r)$$

α^G	Bezeichner des Operators
	α symbolisiert Anonymisierung
	G bezeichnet das genaue Verfahren (Generalisierung)
X	Menge der Generalisierungsattribute, $X \subseteq R$
k	Grad der k-Anonymität, $k \in \mathbb{N}$
r	Basisrelation
	weiterhin vorausgesetzt als globale Parameter im Hintergrund:
DGH_i	Generalisierungshierarchie(n) für Attribut $A_i \in X$

Im Rahmen dieser Arbeit werden sich alle Untersuchungen bezüglich der Generalisierung auf k-Anonymität beschränken. Obendrein auf l-Diversität und t-closeness zu achten, würde zu weit gehen und wohl zu restriktiv für Datenströme sein. Im Sinne hierauf aufbauender Forschung ließe sich der Operator aber problemlos um l-Diversität und t-closeness erweitern. Ein wie hier rein auf k-Anonymität bedachter Operator wäre demnach ein Spezialfall. Für die anderen beiden Metriken wird angenommen, dass diese uneingeschränkt gelten können. Im erweiterten Sinne gilt daher überall 1-Diversität und 1-closeness. Erstere gestattet bereits uniforme Äquivalenzklassen und letztere erlaubt zudem, dass keinerlei Mindestähnlichkeit zur Gesamtverteilung bestehen muss.

$$\alpha^G{}_{X,k}(r) = \alpha^G{}_{X,k,l=1,t=1}(r)$$

8.1.2 Annahmen und Einschränkungen

Damit der vorgestellte Operator im Sinne der damit aufgestellten Regeln sowohl minimalistisch als auch vielseitig ist, sei der Klarheit wegen explizit genannt, wozu er imstande ist und wozu nicht. Es sollte relativ klar sein, dass eine minimale Anonymisierung angestrebt wird. Die Verallgemeinerung von Werten erfolgt daher nicht stärker als nötig.

Bei mehreren Generalisierungsattributen könnten sich aufgrund vielfältiger Generalisierungsebenen mehrere Möglichkeiten ergeben, wie sich eine Generalisierung erreichen ließe –

zugunsten des einen oder aber des anderen Attributes. Welche der Konstellationen schlussendlich bevorzugt wird, ist im Folgenden gleichgültig, sodass man den zugrundeliegenden Generalisierungsalgorithmus nicht in seiner internen Funktionsweise verstehen muss.

Des Weiteren werden alle Werte innerhalb eines Generalisierungsattributes zu Werten gleicher Ebene zusammengefasst. Auf diese Weise wird untereinander eine klare, werteseitige Abgrenzung sichergestellt.

Außerdem wird angenommen, dass die Generalisierung nach eigenem Ermessen erfolgt. Die genaue Wahl der Quasi-Identifikatoren muss somit selbstständig geschehen, nach bestem Wissen und wird nicht weiter in Frage gestellt.

Zudem sind die angegebenen Generalisierungsattribute auch wirklich Teil der Ausgangsrelation. Darüber hinaus existiert zu jedem dieser Attribute genau eine zielführende Hierarchie. Es gäbe nämlich auch die Möglichkeit, dass mehrere vorliegen, unter denen man dann entweder auswählt oder aber automatisch die bessere genommen werden würde.

Weiterhin sei klargestellt, dass sich die Generalisierungshierarchien immer nur an Einzelattribute richten. Es gibt nämlich auch Techniken, bei denen sie im Sinne der Generalisierung auf mehrere ausgelegt sein können.

Darüber hinaus wird prinzipiell vorausgesetzt, dass für jeden zu anonymisierenden Datenbestand mindestens so viele Einträge vorliegen, wie es der Anonymitätsgrad verlangt. Wenn die Einträge nicht einmal für eine einzige k-große Gruppe reichen, ist es unmöglich, die geforderte Anonymität in irgendeiner Form zu erfüllen. Sollte es dennoch dazu kommen, würde alle Einträge unterdrückt werden, was der leeren Ergebnismenge gleichkommt.

8.1.3 Beispiel

Zum genauen Verständnis des α^G-Operators sei dessen Wirkungsweise am besten an einem Beispiel erklärt. Eine kurze inhaltliche Kontextualisierung ist in Abschnitt 2.7 zu finden.

Die Ausgangsrelation r setzt sich aus dem Schema $R = \{A, B, C, D\}$ zusammen. Ebendiese Relation soll nun mittels Generalisierung 3-anonym gemacht werden. Die zu generalisierenden Attribute sind hierbei A und C. Das C steht für den Rhesusfaktor, das A für den Entnahmetag innerhalb eines nicht näher spezifizierten Jahres. Komprimiert dargestellt sind folgende Generalisierungen möglich:

$$A\text{: Tag} \longrightarrow \text{Monat} \longrightarrow \text{Quartal} \longrightarrow \text{Semester} \longrightarrow \text{irgendwann}$$

$$C\text{: } + \longrightarrow \pm,\ - \longrightarrow \pm$$

Eine ausführlichere Baumdarstellung ist weiter unten in Abbildung 45 zu finden. Aus all diesen Forderungen ergibt sich dann folgender Ausdruck:

$$r_a := \alpha^G{}_{X=\{A,C\},k=3}(r)$$

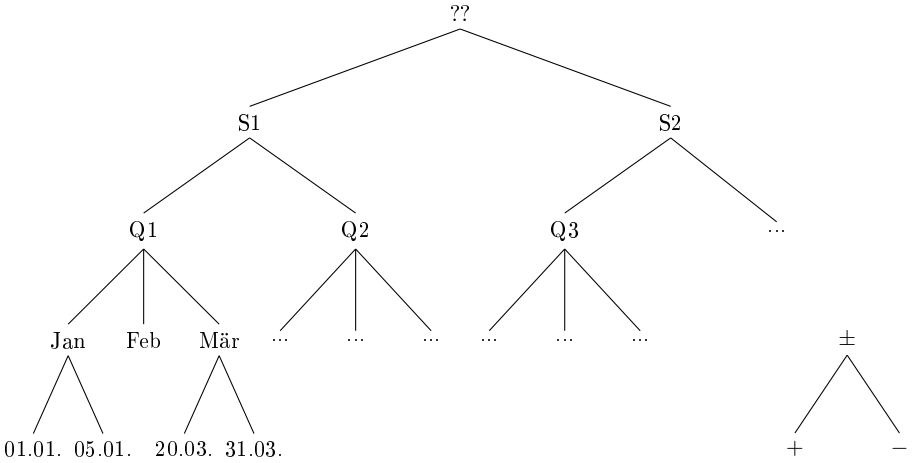
Abbildung 44 zeigt an einem exemplarischen Datenbestand, wie das Resultat aussehen könnte. Um hinsichtlich der Tage 3-anonyme Gruppen zu schaffen, bedarf es mindestens einer Generalisierung zu Monaten, und hinsichtlich des Rhesusfaktors bleibt keine andere Wahl als stärkstmöglich zu generalisieren. Dies wäre zunächst natürlich nur eine Einzelbetrachtung und darauf aufbauend müsste dann geschaut werden, zu wessen Gunsten man die Forderung im Zusammenspiel erfüllten könnte. In diesem Beispiel erbringen A und C die geforderte Anonymität aber ebenso in Kombination.

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
05.01.	B0	−	32
20.03.	A0	+	36
20.03.	A0	−	42
31.03.	B0	+	56

$\Longrightarrow$

A	B	C	D
Jan	00	±	18
Jan	00	±	24
Jan	B0	±	32
Mär	A0	±	36
Mär	A0	±	42
Mär	B0	±	56

Abbildung 44: links r, rechts r_a



(a) DGH_A (b) DGH_C

Abbildung 45: Generalisierungshierarchien

8.2 Schwächen von α^G-Operator je Fenstertyp

In diesem Abschnitt werden fallspezifisch die Schwächen des minimalistischen α^G-Operators hinsichtlich aller wesentlichen Fenstertypen beleuchtet.

8.2.1 Tupelfenster

Bei Gebrauch des minimalistischen Operators lassen sich bei Tupelfenstern abhängig vom Fenstertyp folgende Stromdaten-Probleme konstatieren:

Tupelfenster-Typ	P1	P2 (ggf. bei $c < k$)	P3 (ggf.)
stehend	nein	einmalig (existenziell)	gar keine Folgefenster
Tumbling	nein	durchgehend	bei $c < k$ durchgehend
Sliding	ja	durchgehend	grundsätzlich, da $\|hop\| := 1 < (k \geq 2)$)
Jumping	ggf.	durchgehend	bei $\|hop\| < k$ durchgehend
Session	nein	durchgehend	bei $c < k$ durchgehend
einseitig wachsendes LM	ja	zu Beginn	bei $\|hop\| < k$
relativ wachsend	ggf.	zu Beginn	bei $\|hop_{fast}\| < k$
einseitig schrumpfendes LM	ja	irgendwann immer	gar kein Zuwachs
relativ schrumpfend	ggf.	irgendwann immer	bei $\|hop_{slow}\| < k$
beidseitig wachsendes LM	ja	zu Beginn	bei $\|hop_{past}\| + \|hop_{future}\| < k$
beidseitig schrumpfendes LM	ja	irgendwann immer	gar kein Zuwachs

Abbildung 46: Tupelfenster-Schwächen von α^G

Generalisierung ist für alle drei Problemarten anfällig. P1 ist verfahrenunabhängig durch die Überlappung bedingt, weshalb es sich genau wie beim Verrauschen und Slicen verhält. Wie bereits beim Slicing ist auch das Mindestgrößenproblem P2 und Zuwachsproblem P3 existent. Sie werden bei konfligierenden Minimalanforderungen verursacht und sind bei Tupelfenstern bereits zur Kompilierzeit bzw. mit der Stellung der Abfrage absehbar. Außerdem lässt sich konstatieren, dass Abb. 46 deckungsgleich mit Abb. 34 ist. Die äußerliche Problemanfälligkeit ist bei Generalisierung und Slicing also gleich, was eine gewisse grundlegende Ähnlichkeit der beiden Verfahren zum Ausdruck bringt.

P1 - Mehrfachgeneralisierung

Das grundlegende Problemmuster von P1 ist dasselbe wie beim einfachen Verrauschen und Slicing, stets bedingt durch die Schnittmengen. Die Auswirkungen sind allerdings auch hier verfahrensspezifisch andere, da es sich um ein zeilenübergreifendes Verfahren handelt, da noch freier als das lokal überschaubare Slicing arbeiten kann.

Die Gefahr, die vom Mehrfachgeneralisieren ausgeht, hängt vom Freiheitsgrad bei der Wahl nach Generalisierungshierarchien ab. Eine Generalisierung kann bekanntlich sowohl ein Attribut als mehrere Attribute betreffen, was jeweils eigenständige Hierarchien erfordert; in der Kombination sind sie dann als Einheit zu verstehen. Im Folgenden wird zwischen drei Arten unterschieden. Bei einer statischen Generalisierung sind die Generalisierungshierarchien mit dem Relationenschema (bzw. Stream) fest verbunden. Anfänglich darf selbstverständlich eine freie Auswahl anhand aller Optionen getroffen werden, aber mit der ersten Ergebnisveröffentlichung müssen die ausgewählten Hierarchien künftig konsequent beibehalten werden.

Bei einer dynamischen Generalisierung dürfen die Generalisierungshierarchien jederzeit gewechselt werden. Eine dynamische Generalisierung ist in jedem Fall freier als eine statische; falls selbst eine statische Generalisierung Gefahren birgt, müsste das Anonymisieren bzw. Veröffentlichen aus Sicherheitsgründen untersagt sein.

Tupelfenster-Typ	erlaubte Generalisierung	Gefahrenpotential
stehend	dynamisch	keins
Tumbling	dynamisch	keins
Sliding	keine	eindeutig vorhanden
Jumping	keine	eindeutig vorhanden
Session	dynamisch	keins
einseitig wachsendes LM	statisch (Upgrade)	ggf.
relativ wachsend	keine	eindeutig vorhanden
einseitig schrumpfendes LM	statisch (Downgrade)	ggf.
relativ schrumpfend	keine	eindeutig vorhanden
beidseitig wachsendes LM	statisch (Upgrade)	ggf.
beidseitig schrumpfendes LM	statisch (Downgrade)	ggf.

Die disjunkten Fenstertypen Tumbling und Session lassen sich bedenkenlos dynamisch anonymisieren, da es nie zu einer Mehrfachgeneralisierung kommen kann. Auch ein stehendes Fenster funktioniert ohne Weiteres, da nur ein einziges Fenster gebildet wird; hierbei sei natürlich vorausgesetzt, dass es sich wirklich um eine einmalige Abfrage handelt, die nicht mehrfach öffentlich gemacht wird.

Monotones Fensterwachstum ist zulässig, falls sichergestellt ist, dass es sich um eine statische Generalisierung handelt. Bei ein- und beidseitig wachsenden Landmarks kommt es mit fortschreitender Zeit zur Verfeinerung von Werten, bei ein- und beidseitig schrumpfenden Landmarks zur Verschlechterung. Eine dynamische Generalisierung birgt die Gefahr, dass ein Eintrag bei anderen Kandidaten bzw. Peers unterkommt, was den Gruppenschutz zerstört und intersektionelles Wissen preisgibt.

Die restlichen Fenstertypen Sliding, Jumping und die sich relativ entwickelnden Fenster sind beim Gebrauch des naiven Operators grundsätzlich unsicher. Selbst eine statische Generalisierung ist hier gefährlich, da die Nichtmonotonie gebildete Gruppen irgendwann auflöst. Die verbleibenden Kandidaten müssten sich dann mit neueren zusammentun, was möglicherweise sogar zu feineren Werte als zuvor resultieren könnte. Dieses gedächtnislose Agieren würde den Schutz der früheren Gruppe aber rückwirkend zerstören.

P2 - Unterschreitung der Generalisierungs-Mindestgröße

Beim Generalisieren lässt sich bezüglich der Mindestgröße dasselbe wie beim Slicing konstatieren, weshalb dieser Abschnitt deckungsgleich mit Abschnitt 7.2.1 ist.

Bei konstanter Fensterkapazität ist das Problem existenzbedrohend, da eine permanente Unterschreitung eine ständige vorsorgliche Ergebnisunterdrückung bedeuten würde, was quasi der leeren Menge gleichkommt. Dies würde im Wesentlichen zwar nur sehr kleine Fenster betreffen, aber dennoch einen herben Verlust bedeuten, da es Kleinstanfragen grundsätzlich verunmöglichen würde.

Bei wachsender Fensterkapazität wird die Mindestanforderung nur bei einigen Fenstern zu Beginn verletzt, die man durch vorsorgliche Mechanismen unterdrücken müsste. Auf lange Sicht erscheint der anfängliche Verlust im Verhältnis nichtig und könnte daher vernachlässigt werden.

Bei schrumpfender Fensterkapazität werden ggf. nur einige Fenster am Ende verletzt, die man vorsorglich unterdrücken müsste, was einem etwas früheren Abbruch gleichkommt. Der Verlust steht dabei im Verhältnis zum Startfenster und wäre bei einem kleinen Startfenster dementsprechend groß. Aus Anwendungssicht ist dies jedoch relativ unwahrscheinlich, da möglichst langlebige Analysen anvisiert werden, was im schrumpfenden Fall immer große Startfenster erfordert. Bei einem einmaligen langlebigen Abbau wäre der unterdrückte Verlust daher so gut wie vernachlässigbar. Dennoch ist dies nicht in jedem Anwendungsfall ratsam. Beim periodischen Abbau besonders kleiner Fenster skaliert der Verlust nämlich mit, was eine ständige, nicht unbedenkliche Verfälschung bedeutet.

P3 - Unterschreitung des Mindestzuwachses

Beim Generalisieren lässt sich auch bezüglich des Mindestzuwachses dasselbe wie beim Slicing konstatieren, weshalb dieser Abschnitt deckungsgleich mit Abschnitt 7.2.1 ist.

Bei naivem Gebrauch ist Problem P3 nicht per se zu beobachten. Es ergibt sich vielmehr erst als weiteres Problem, sobald man P1 und P2 zu lösen versucht. Das Problem verhält sich dann analog zu P2: Bei konstantem Tupelzuwachs besteht das Problem ggf. durchgehend, bei steigendem Tupelzuwachs nur anfänglich, bei schrumpfendem Tupelzuwachs am Ende immer. Da der Zuwachs bei Tupelfenstern jedoch allgemein konstant gehalten wird, ist P3 dadurch ein existenzielles Problem für all jene Fensteranfragen, die die allerfeinsten Entwicklungen analysieren wollen.

8.2.2 Zeitfenster

Bei Zeitfenstern kommt erschwerend hinzu, dass die Tupelentwicklung ungewiss ist bzw. erst zur Laufzeit bekannt wird. Dementsprechend stellt Abb. 47 eine zusätzliche Verschärfung von Abb. 46 dar. P1 ist verfahrenunabhängig durch die Überlappung bedingt, weshalb es diesbezüglich keine Veränderung gibt. Die Verschärfungen betreffen das Mindestkapazitätsproblem P2 und das Mindestzuwachsproblem P3. Auch bei der Zeitfenster-Verschärfung lässt sich konstatieren, dass Abb. 47 deckungsgleich mit Abb. 35 ist. Die äußerlichen Problemanfälligkeiten von Generalisierung und Slicing gleichen sich analog also auch bei Zeitfenstern.

Zeitfenster-Typ	P1	P2 (ggf. bei $c < a$)	P3 (ggf.)
stehend	nein	einmalig (existenziell)	gar keine Folgefenster
Tumbling	nein	stets ungewiss	stets ungewiss
Sliding	ja	stets ungewiss	grundsätzlich, da $\|hop\| := 1 < (k \geq 2)$)
Jumping	ggf.	stets ungewiss	stets ungewiss
Session	nein	stets ungewiss	stets ungewiss
einseitig wachsendes LM	ja	zu Beginn	stets ungewiss
relativ wachsend	ggf.	immer unwahrscheinlicher	stets ungewiss
einseitig schrumpfendes LM	ja	irgendwann immer	gar kein Zuwachs
relativ schrumpfend	ggf.	immer wahrscheinlicher	stets ungewiss
beidseitig wachsendes LM	ja	zu Beginn	stets ungewiss
beidseitig schrumpfendes LM	ja	irgendwann immer	gar kein Zuwachs

Abbildung 47: Zeitfenster-Schwächen von α^G

8.3 Besondere funktionale Modi

Von allen operationalisierten Verfahren kann im Besonderen die Generalisierung extrem hohe Freiheitsgrade bieten. Die Schattenseite dieser Vielfalt an Möglichkeiten ist natürlich, dass sich dies nur schwer beschreiben bzw. kompakt vereinheitlichen lässt. Da es den Rahmen sprengen würde, wird sich diese Arbeit auf die Umreißung der grundlegenden Optionen beschränken, um den potenziellen Möglichkeitsraum zumindest anzudeuten.

Unmittelbare Nachbarschaftsgruppe

Der Pool an Ausgangsdaten ist bekanntlich maßgeblich für die Informationsgüte beim Generalisierungsprozess. Ein hoher Aufwand zwecks Feingranularität ist jedoch zumeist nicht mit der Zeit- und Speicherbeschränkung bei Stromdaten vereinbar. Als effizientester Defaultmodus sei für Stromdaten daher die unmittelbare Nachbarschaft empfohlen, da dieser Modus universell auf alle Fenstertypen anwendbar ist und eine klar erwartbare Wirkung erzielt. Das unmittelbare lokale Abstecken der Gruppen a priori sorgt zwar nur bedingt für die feinsten Ergebnisse bzw. macht sie ungewiss, es verläuft aber schnell und eindeutig. Zudem lässt sich das Ganze bestens parallelisieren und der Gruppenschutz ist auf den ersten Blick erkennbar. Und wie bereits zuvor erwähnt sind relativ feingranulare Werte zumindest dann gewiss, wenn die Streamabfolge eine intrinsische Wertenähe in Form von Sortierung oder Clustering innehaben sollte.

Freie Gruppenbildung

Die freie Gruppenbildung strebt feingranulare Werte mit dem größten Informationsgehalt an. Sie ist aufwendiger und stößt insbesondere in einem Streamingumfeld zwangsläufig an gewisse Grenzen. Unterhalb der 2-Gruppen-Grenzen, also bis zu einem geringen Zuwachs von $2 \cdot k - 1$ Einträgen, ist der Modus von der unmittelbaren Nachbarschaftsgruppierung im Ergebnis nicht zu unterscheiden. Nach erfolgter Generalisierung lässt sich noch differenzieren, ob generalisierte Einträge an ihrer ursprünglichen Position bleiben oder zum Sichtbarmachen der Gruppen umpositioniert werden sollten. Zerstreute Werte sind zwar weniger nutzerfreundlich, erschweren aber unmittelbare Attacken und sind zudem sinnvoll, falls die initiale Reihenfolge auch in höheren Verarbeitungsschritten noch eine Rolle spielen sollte.

Im Session-Fall ist vorteilhafterweise gewährleistet, dass der Zusatzwerte-Raum fortwährend verborgen bleibt und anwächst. Theoretisch hieße das, dass mit fortschreitender Zeit eine immer größer werdende Wertebasis zu Verfügung stünde. Aus praktischer Sicht ist allerdings irgendwann eine Limitierung nötig, falls man zeitnahe Ergebnisse liefern möchte. Weiterhin sei noch angemerkt, dass dieser Gedanke die generalisierende Anonymisierung quasi auf die Spitze treibt: Auf lange Sicht würden Generalisierungsgruppen zu immer feineren Werten konvergieren, also weitestmöglich den Originalwerten annähern, da sich mit anwachsendem Überschuss auch immer ähnlichere Werte häufen können. Solange die Quasi-Identifikatoren ihrer Schlüsselfunktion gerecht bleiben, sind willkürliche Wiederholungen natürlich ausgeschlossen, da sonst per Definition keine Eindeutigkeit mehr gewährleistet sein würde, sodass immer irgendeine Form von Generalisierung unternommen werden muss, weil keine Gruppe rein aus wertgleichen Dopplungen bestehen kann. Sollten die QIDs dagegen nur auf einer Schätzung bzw. unbedachten Nutzerauswahl beruhen, könnten sich genug Wiederholungen ereignen, sodass sogar Originalswerte offenbart werden würden. Bei stetigen Werten und immenser Vielfalt aufgrund von Attributpaarungen ist dies jedoch sehr unwahrscheinlich und selbst für moderate Wertebereiche, wo die Möglichkeit bestünde, würde dies durch die Speicherbegrenzung weitgehend unterbunden werden. Dennoch sollte man fortlaufend anwachsende Datensätze wie beim Landmark und ganz besonders versteckt unterstützende Datensätze grundsätzlich kritisch betrachten, da man die besten privacy-konformen Werte anstrebt und Nutzerfehler hier leicht weitreichende Folgen haben können. Will man es Angreifern nicht allzu leicht machen, hat daher auch eine sehr begrenzte Wissensbasis ihre Vorzüge.

Beim Jumping und dessen Sonderformen ist es dagegen nur eine Frage der Zeit, bis jeder Wert in den sichtbaren Fensterausschnitt rückt. Hier wäre also kluges Management von Nöten, damit nie zu gierig vorgegangen wird.

Bei monoton wachsenden Fenstern, also einseitig oder beidseitig wachsenden Landmarks, erscheinen zwei Strategien realistisch denkbar: Die erste Strategie: Die Zusatzkapazität wird zur Erzielung noch feinerer dazukommender Werte genutzt. Mit der Sichtbarwerdung ist ihr Wert dann allerdings zementiert, sodass ältere Werte keinerlei Verbesserung erhalten. Die zweite Strategie: Auf die Zusatzkapazität wird verzichtet. Stattdessen wird gestattet, dass sich Generalisierungswerte unter Beibehaltung der zu Beginn ausgewählten Hierarchie im weiteren Verlauf jederzeit verfeinern lassen. Man könnte natürlich beides zugleich zulassen, aber die ständige Verfeinerung bietet schon mehr als genug Potential, sodass es einer Ressourcenverschwendung gleichkäme.

Bei monoton schrumpfenden Fenstern und stehenden Fenstern hat das Konzept besonders wenig Sinn, da bei der Fensterentwicklung niemals irgendein Zuwachs zu erwarten ist. Sichtbare und versteckte Werte wären hier permanent von einander getrennt. Eine Verfeinerung des Startfensters wäre natürlich möglich, müsste für Außenstehende aber auch als solche gekennzeichnet werden, da es sonst selbst mit fortschreitender Zeit wie eine unzureichende Anonymisierung erscheinen würde.

Fenster mit relativer Größenentwicklung lassen sich besonders schwer pauschalisieren, da es weiterer Unterscheidung bedarf. Je nach Gegebenheit werden diverse schon bekannte Strategien wieder aufgegriffen. Relativ wachsende disjunkte Fenster, also mit tumbling- und

session-ähnlichem Charakter, lassen sich zwar nicht aufbessern, profitieren bei freier Gruppenbildung aber bereits von ihrer zunehmenden Größe, was die aufbessernde Zusatzkapazität obsolet macht. Bei relativ wachsenden Fenstern mit Schnittmenge könnten dieselben Strategien wie beim monoton wachsenden angewandt werden.

Relativ schrumpfende Fenster, die tumbling-typisch angrenzen, könnten die schwindende Wertebasis durch Zusatzwerte aus naher Zukunft kompensieren. Relativ schrumpfende mit session-typischen Zwischenräumen könnten ihre schwindende Wertebasis noch effektiver durch die ausgeblendeten Zwischenräume ausgleichen. Bei relativ schrumpfenden mit Schnittmenge könnten dieselben zwei Strategien wie beim monoton wachsenden angewandt werden – möglicherweise sogar in Kombination, da feingranulare Wert hier in besonderem Maße erschwert werden.

Versteckte optionale Zusatzkapazität

Eine versteckte Zusatzkapazität kann bereits vorteilhaft sein, um bei zeilenübergreifenden Gruppenverfahren genug verborgenes Fassungsvermögen zu besitzen, falls der sichtbare Zuwachs zu gering sein sollte. Da die Gruppierung beim Slicing räumlich klar definiert ist, kann man zwecks parallelisierbarer Stapelverarbeitung mehr als eine zusätzliche Gruppen auf einmal abarbeiten, um bei winzigen Sprüngen größere Ruhephasen zu ermöglichen, in denen de facto nichts anonymisiert werden muss, weil genug vorgeslicte versteckte Werte bereitstehen. Auf informativer Ebene lässt sich darüber allerdings nichts dazugewinnen.

Ganz anders sieht es wiederum bei der Generalisierung aus, wo sich der granulare Informationsgehalt theoretisch mit jedem weiteren Eintrag erheblich aufbessern ließe. Rein praktisch wird diese Möglichkeit natürlich immer an irgendeine Grenze stoßen: (1) Speicherlimitierung; (2) der Detailgrad einer nicht dynamischen Hierarchie ist irgendwann ausgereizt; (3) der informative Zugewinn wird zunehmend geringer bzw. ist den zusätzlichen Aufwand nicht wert; (4) der Zeitaufwand übersteigt die tolerierte Ergebnisverzögerung.

Weiterhin lässt sich noch differenzieren, in welchem Umfang die versteckte Zusatzkapazität mitgeneralisiert werden muss. Während sichtbare Einträge immer anonymisiert sein müssen, ist es bei nicht sichtbaren eine optionale Angelegenheit. Der einfachste Fall wäre die vollständige Anonymisierung, also mitsamt aller versteckten Einträgen. Da dieser Modus keinerlei Unterscheidung verlangt, eignet er sich am ehesten als Standardfall.

Eine dynamischere Variante bestünde darin, nur die nötigsten Einträge zu generalisieren. Bloß Einträge, die in den sichtbaren Bereich aufrücken, müssten anonymisiert werden, was vereinzelt ohnehin schon geschehen ist. Im Grunde sind Einträge dann nur nach ganz lose an den sichtbaren Bereich gebunden. Statt Gruppen direkt innerhalb des sichtbaren In-Group-Bereichs aufzubauen, bestünde aufgrund der Optimierungssuche nach Gleichgesinnten die Tendenz, ganz entfernt noch ähnlichere Einträge zu finden – ein optimierungswilliges Streben, das durchaus mit dem Aufkommen von Internetforen oder Datingplattformen vergleichbar ist. Im äußersten Fall wäre es sogar so, dass jeder sichtbar gewordene Eintrag eine neue Gruppe mit je $k-1$ Einträgen der versteckten Out-Group ausformen könnte, sofern die Zusatzkapazität nicht kleiner angelegt ist. Dementsprechend müsste man generell aufpassen, dass niemals zu gierig vorgegangen wird bzw. die Verhältnisse stimmen, da es ansonsten nicht viel mehr als ein qualitativer Startbonus wäre, der irgendwann erschöpft ist. Im Vergleich

tun sich dann folgende zwei Extreme auf: Falls man geschlossen bzw. ohne Zusatzkapazität anonymisieren könnte, was z.B. auf ausreichend große Tumbling-Fenster zutrifft, hat man im Minimalfall nur die In-Group selbst. Generalisiert man dagegen frei und mit Zusatzkapazität, müsste man im Ernstfall insgesamt das k-Fache der sichtbar werdenden In-Group aufbringen – sofern man die Anzahl an Out-Group-Kandidaten nicht bewusst beschränkt, um die Gruppenbildung innerhalb der In-Group zu bestärken – und über die Hops dann im Schnitt ebenso viel wieder dazuwachsen lassen, damit sich der Vorrat auf Dauer niemals erschöpfen würde.

8.4 Erweiterter Generalisierungs-Operator α^{G+}

8.4.1 Aufbau

Der verfeinerte Generalisierungs-Operator setzt sich folgendermaßen zusammen:

$$\alpha^{G+}{}_{X,k,grouping[;rest,skip,optional,cut;reverse]}(r)$$

α^{G+}	Bezeichner des Operators α symbolisiert Anonymisierung G bezeichnet das genaue Verfahren (Generalisierung) $+$ als abgrenzender Indikator für den erweiterten Operator
X	Menge der Generalisierungsattribute, $X \subseteq R$
k	Grad der k-Anonymität, $k \in \mathbb{N}$
r	Basisrelation
	weiterhin vorausgesetzt als globale Parameter im Hintergrund:
DGH_i	Generalisierungshierarchie(n) für Attribut $A_i \in X$
	neue Plichtparameter:
$grouping$	Gruppenbildung • *neighbouring*: unmittelbare Nachbarschaftsabfolge (schnell, Streaming-Default) • *free-fixed*: freie Gruppenbildung mit zerstreuten Gruppen, wo die generalisierten Werte an ihrer Ausgangsposition bleiben • (*free-rearranged*: freie Gruppenbildung mit Neupositionierung zum Sichtbarmachen der Gruppen → indirekt über *free-fixed* + Clustering oder Sortierung) • *free-dynamic*: freie Gruppenbildung, erweitert um einen optionalen Generalisierungspool am untersten Ende; notwendigerweise zerstreut
	neue optionale Parameter:
($rest$)	Umgang mit überschüssigen Einträgen bei einer unvollständigen letzten Gruppe Vorbedingung: $grouping :=$ *neighbouring* • *include*: zur letzten vollen Gruppe hinzufügen (Sondergröße: k bis $2 \cdot k - 1$) • *suppress*: unvollständige Gruppe weglassen • *unchanged*: unverändert mit ausgeben (zur internen Speicherung bzw. Verarbeitung in späteren Iterationsschritten; NICHT für Direktausgaben vorgesehen) ◦ Default: $rest :=$ *include*
($skip$)	Anzahl unveränderter Einträge zu Beginn, deren Generalisierung übersprungen wird ◦ Default: $skip := 0$
($optional$)	Anzahl von Einträgen am unteren Eintragsende, deren Generalisierung optional ist, was einen potenziellen Zusatzpool für noch bessere Kandidaten eröffnen soll Vorbedingung: $grouping :=$ *free-dynamic* ◦ Default: $optional := 0$
(cut)	manuell abgezählte Selektion, um das untere Eintragsende abschließend eigenständig beschneiden zu können, was zum Trimmen einer Gruppe und zum Ausblenden unveränderter sowie optionaler Einträge nützlich ist • positive Zahl: Auswahl der ersten n Einträge • negative Zahl: Ausblenden der letzten n Einträge
($reverse$)	optionale Richtungsumkehr der Arbeitsweise ◦ Default: $reverse := false$ Anmerkung: Die Standard-Arbeitsweise des Operators ist von oben nach unten ausgelegt. In Bezug auf Stromdaten wird im Folgenden nämlich angenommen, dass sich die ältesten Einträge eines Fensters am Anfang beziehungsweise die neusten Einträge am unteren Ende befinden. Sollte eine Arbeitsweise in gegenteiliger Richtung nötig sein, würde die komplette Funktionsweise quasi gespiegelt bzw. auf den Kopf gestellt werden.

8.4.2 Annahmen und Einschränkungen

Die Annahmen und Einschränkungen haben sich gegenüber 8.1.2 im Kern nicht verändert. Sie werden höchstens an einigen Stellen explizit überschrieben, wenn der Operator um Alternativen bereichert wurde.

Weitere integrierbare Steuerungsparameter

Neben den eingeführten Parametern wären noch zwei weitere denkbar, die zwecks pragmatischer Reduktion ausgelassen wurden. Mit einem Argument könnte man gruppenübergreifend über ein einheitliche Generalisierungsebene entscheiden. Da das Hauptziel üblicherweise feingranulare Werte sind, sind im Regelfall natürlich unabhängige Ebenen je Gruppe bevorzugt. Einheitliche Ebenen wären dementsprechend eine sehr spezielle Nutzerpräferenz, wo alle feineren Ebenen auf das Niveau gröberer Ausreißer abgeschwächt werden würden. Ein Vorteil dieser härteren bzw. übervorsichtigen Maßnahme wäre zumindest, dass es intersektionelle Angriffe präventiv abschwächt.

Weiterhin könnte man noch ein Argument einführen, mit dem sich ein Hierarchiewechsel erlauben ließe, falls mehrere Hierarchien zur Verfügung stünden. Wie allerdings schon mehrfach darauf hingewiesen wurde, ist hier aus Datenschutzsicht klare Vorsicht geboten. Es sollte vorzugsweise sichergestellt werden, dass Einträge nur einmalig berücksichtigt werden oder bei Wiederholung ein entsprechender Werteerhalt sichergestellt ist.

8.4.3 Beispiele

Fallbeispiel 1: Default

Das unbestimmte bzw. noch nicht vollständig konkretisierte Defaultverhalten des Operators entspricht in verkürzter Form:

$$\alpha^{G+}{}_{X,k,grouping}(r) \equiv \alpha^{G+}{}_{X,k,grouping;rest=include,skip=0,optional=0,cut=-0;reverse=false}(r)$$

Innerhalb der eigenen Bachelorarbeit wurde die Generalisierung anhand von Fallbeispielen sehr pragmatisch und reduziert gehalten. Aufgrund der häufigen Vorsortierung nach den Generalisierungsattributen, der Schlichtheit der Hierarchien und der minimalen aufeinanderfolgenden Gruppengrößen mag es auf den ersten Blick so wirken bzw. zumeist so sein, als wäre $include := neighbouring$ genutzt worden. Genau genommen wurde jedoch vielmehr eine weitestgehend optimale Generalisierung über $include := freeRearranged$ mit Clustering (also Sichtbarmachung der Gruppen ohne strenge Sortierung) genutzt, was im Besonderen durch die zusätzliche Untersuchung im Anhang deutlich wird. Somit wäre der α^{G+}-Operators im Falle von $grouping := freeRearrangedClustering$ allgemein deckungsgleich mit dessen Vorläufer α^{G}, was nochmals durch Abb. 48 veranschaulicht werden soll.

$$\alpha^{G+}{}_{X,k,grouping=freeRearranged;}(r) \Longleftrightarrow \alpha^{G}{}_{X,k}(r)$$

$$r_2 := \alpha^{G+}{}_{X=\{A,C\},k=3,grouping=freeRearranged}(r_1)$$

A	B	C	D
20.03.	A0	+	36
20.03.	A0	−	42
31.03.	B0	+	56
01.01.	00	+	18
01.01.	00	+	24
14.02.	AA	+	21
11.02.	BB	+	28
05.01.	B0	−	32
20.02.	B0	+	34

$\Longrightarrow$

A	B	C	D
Mär	A0	±	36
Mär	A0	±	42
Mär	B0	±	56
Jan	00	±	18
Jan	00	±	24
Jan	B0	±	32
Feb	AA	+	21
Feb	BB	+	28
Feb	B0	+	34

Abbildung 48: links r_1, rechts r_2

Fallbeispiel 2: *grouping*-**Parameter**

Die Abbildungen 49 bis 52 veranschaulichen die verschiedenen potenziellen Gruppierungsmodi des *grouping*-Parameters. Die Relation r_2 in Abb. 49 zeigt die nachbarschaftliche Gruppenbildung. Bei dieser werden lokal aufeinanderfolgende Gruppen mit Minimalgröße k nach dem Muster Top-down gebildet. Über den *rest*-Parameter (siehe Fallbeispiel 3) lässt sich noch spezifizieren, wie mit der potenziell unvollständigen letzten Gruppe am unteren Ende umgegangen wird. Kurzum lässt sich sagen, dass der Nachbarschaftsmodus quasi wie die Splitstellensetzung des Slicings funktioniert. Der Vorteil dieses Ansatzes besteht in der äußerst effizienten lokalen Berechnung, der Nachteil in den potenziell Kauf genommenen Abstrichen bezüglich der Granularität. Dieser Kompromiss kann im Besonderen bei Stromdaten nützlich sein. Nebenbei sei nochmals angemerkt, dass der Granularitätsnachteil bereits bei einer natürlichen lokalen Wertenähe entlang des Stroms oder durch Vorclusterung (bis hin zur Vorsortierung) weitgehend minimiert werden kann.

$$r_2 := \alpha^{G+}{}_{X=\{A\},k=3,grouping=neighbouring}(r_1)$$

A	B	C	D
20.03.	A0	+	36
20.03.	A0	+	42
31.03.	B0	−	56
01.01.	00	+	18
01.01.	00	+	24
14.02.	AA	+	21
11.02.	BB	+	28
05.01.	B0	−	32
20.02.	B0	+	34
09.03.	B0	+	19

$\Longrightarrow$

A	B	C	D
Mär	A0	+	36
Mär	A0	+	42
Mär	B0	−	56
Jan-Feb	00	+	18
Jan-Feb	00	+	24
Jan-Feb	AA	+	21
Jan-Mär	BB	+	28
Jan-Mär	B0	−	32
Jan-Mär	B0	+	34
Jan-Mär	B0	+	19

Abbildung 49: links r_1, rechts r_2

Relation r_3 in Abb. 50 zeigt eine freie Gruppenbildung mit zerstreuten Gruppenmitgliedern, wo alle Einträge fixiert bzw. positionell unverändert bleiben.

$$r_3 := \alpha^{G+}{}_{X=\{A\},k=3,grouping=freeFixed}(r_1)$$

A	B	C	D
20.03.	A0	+	36
20.03.	A0	+	42
31.03.	B0	−	56
01.01.	00	+	18
01.01.	00	+	24
14.02.	AA	+	21
11.02.	BB	+	28
05.01.	B0	−	32
20.02.	B0	+	34
09.03.	B0	+	19

$\Longrightarrow$

A	B	C	D
Mär	A0	+	36
Mär	A0	+	42
Mär	B0	−	56
Jan	00	+	18
Jan	00	+	24
Feb	AA	+	21
Feb	BB	+	28
Jan	B0	−	32
Feb	B0	+	34
Mär	B0	+	19

Abbildung 50: links r_1, rechts r_3

Die Relationen r_4 und r_5 in Abb. 51 zeigen eine freie Gruppenbildung mit abschließender Neupositionierung zum Sichtbarmachen der entstandenen Gruppen. Relation r_4 nutzt eine aufsteigende lexikalische Sortierung nach dem Generalisierungsattribut A. Beim Clustering in Relation r_5 ist sortierungsunabhängig das initiale Auftreten ausschlaggebend, sodass alle späteren bzw. tiefer liegenden Mitglieder einer Gruppierung direkt unter das zuvorderst auftretende Gruppenmitglied geschoben werden. Bei diesem Clusteransatz wird also in Teilen versucht, eine gewisse zeitkausale Abfolge zu bewahren; alternativ könnte die Gruppierung aber z. B. ebenso gut eine minimalistische Veränderung nach dem Vorbild der Levenshtein-Distanz anstreben.

$$r_4 := \alpha^{G+}{}_{X=\{A\},k=3,grouping=freeRearrangedSort}(r_1)$$
$$r_5 := \alpha^{G+}{}_{X=\{A\},k=3,grouping=freeRearrangedClustering}(r_1)$$

A	B	C	D
20.03.	A0	+	36
20.03.	A0	+	42
31.03.	B0	−	56
01.01.	00	+	18
01.01.	00	+	24
14.02.	AA	+	21
11.02.	BB	+	28
05.01.	B0	−	32
20.02.	B0	+	34
09.03.	B0	+	19

$\Longrightarrow$

A	B	C	D
Feb	AA	+	21
Feb	BB	+	28
Feb	B0	+	34
Jan	00	+	18
Jan	00	+	24
Jan	B0	−	32
Mär	A0	+	36
Mär	A0	+	42
Mär	B0	−	56
Mär	B0	+	19

A	B	C	D
Mär	A0	+	36
Mär	A0	+	42
Mär	B0	−	56
Mär	B0	+	19
Jan	00	+	18
Jan	00	+	24
Jan	B0	−	32
Feb	AA	+	21
Feb	BB	+	28
Feb	B0	+	34

Abbildung 51: von links nach rechts: r_1, r_4 und r_5

Abb. 52 zeigt eine freie Gruppenbildung mit dynamischer Aufbesserung durch Verwendung bzw. Vorwegnahme einer optionalen Zusatzkapazität. Der Nutzung der Zusatzkapazität kann dabei verschiedene Direktiven verfolgen: (1) etwa die Priorisierung ausgewählter Wertebereiche; (2) die generelle Verfeinerung besonders feingranularer Gruppen, denen es noch an einzelnen Werten zum Erreichen der nächsthöheren Generalisierungsebene mangelt; (3) die vorteilhafteste Einbindung bzw. Aufwertung von Ausreißern. Relation r_6 repräsentiert eine zweiteilige Startrelation, die es zu anonymisieren gilt. Die oberen neun Einträge sind dabei

verpflichtend zu generalisieren, die unteren acht sind optional. Relation r_8 zeigt, wie das Ergebnis einer freien fixierten Gruppenbildung ohne Zusatzkapazität ausfallen würde. Die in der Mitte befindliche Relation r_7 zeigt eine von vielen möglichen Aufwertungen, die sich je nach Zielsetzung und Reduziertheit erzielen ließen.

$$\alpha^{G+}{}_{X,k,freeDynamic,optional=0}(r) \equiv \alpha^{G+}{}_{X,k,grouping=freeFixed}(r)$$

$$\begin{aligned} r_7 &:= \alpha^{G+}{}_{X=\{A\},k=3,grouping=freeDynamic,optional=8}(r_6) \\ r_7{}' &:= \alpha^{G+}{}_{X=\{A\},k=3,grouping=freeDynamic,optional=8,cut=-8}(r_6) \\ r_8 = r_8{}' &:= \alpha^{G+}{}_{X=\{A\},k=3,grouping=freeFixed}(r_6) \end{aligned}$$

A	B	C	D
20.03.	A0	+	36
20.03.	A0	+	42
31.03.	B0	−	56
06.12.	BB	−	28
01.01.	00	+	18
01.01.	00	+	24
05.01.	B0	−	32
14.02.	AA	+	21
11.02.	BB	+	28

$\Longrightarrow$

A	B	C	D
Mär	A0	+	36
Mär	A0	+	42
Mär	B0	−	56
Jan-Dez	BB	−	28
01.01.	00	+	18
01.01.	00	+	24
Jan-Dez	B0	−	32
Feb	AA	+	21
Feb	BB	+	28

A	B	C	D
Mär	A0	+	36
Mär	A0	+	42
Mär	B0	−	56
Feb-Dez	BB	−	28
Jan	00	+	18
Jan	00	+	24
Jan	B0	−	32
Feb-Dez	AA	+	21
Feb-Dez	BB	+	28

20.02.	B0	+	34
02.04.	A0	−	56
27.04.	BB	+	21
15.04.	B0	+	28
01.01.	B0	+	19
02.06.	00	−	37
01.09.	00	−	48
03.06.	AA	+	33

Feb	B0	+	34
02.04.	A0	−	56
27.04.	BB	+	21
15.04.	B0	+	28
01.01.	B0	+	19
02.06.	00	−	37
Jan-Dez	00	−	48
03.06.	AA	+	33

Abbildung 52: von links nach rechts: r_6 bis r_8

Fallbeispiel 3: *rest*-Parameter

In Abbildung 53 werden die drei verschiedenen Modi des *rest*-Parameters veranschaulicht, sobald als Vorbedingung der Nachtbarschaftsmodus *grouping* := *neighbouring* ausgewählt wurde. Aufgrund einer Relationengröße von $|r| = 11$ und einer Generalisierungsgröße von $k = 3$ ergibt sich aufgrund von $11\ mod\ 3 = 2$ ein Rest von zwei Einträgen. Zunächst sei nochmals angemerkt, dass der potenzielle Rest immer am unteren Ende, also stets nach dem Muster Top-down entsteht; zum einen ist nämlich der *cut*-Parameter darauf abgestimmt, zum anderen könnte man das Muster Bottom-up auch einfach über einen zusätzlichen *reverse*-Parameter erzielen, was bereits im Fallbeispiel 6 von Abschnitt 7.3.3 angesprochen wurde. Relation r_2 zeigt die Eingliederung des Restes in die letzte vollständige Generalisierungsgruppe, die wie in diesem Fall auf eine Übergröße von bis zu fünf Einträgen anwachsen kann; dieses inkludierende Vorgehen wurde auch schon beim damaligen α^G-Operator präferiert. Relation r_3 zeigt die Unterdrückung der untersten beiden Einträge. Das Unterdrücken findet

noch vor dem Anonymisieren statt, sodass keine unnötige Verschlechterung bewirkt wird; auf diese Weise grenzt sich der Modus auch klar von einem nachträglich erfolgenden Abschneiden durch den *cut*-Parameter ab. Relation r_4 zeigt zuletzt, wie die untersten beiden Einträge unverändert übernommen wurden.

$$r_2 := \alpha^{G+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=include}(r_1)$$
$$r_3 := \alpha^{G+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=suppress}(r_1)$$
$$r_4 := \alpha^{G+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=unchanged}(r_1)$$

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	–	32
20.03.	A0	+	36
13.01.	A0	+	42
31.03.	B0	+	56
02.05.	A0	–	49
08.05.	A0	–	58
09.05.	B0	–	28
24.06.	AA	+	19
24.09.	00	–	73

$\Longrightarrow$

A	B	C	D
Jan	00	+	18
Jan	00	+	24
Jan	B0	–	32
Q1	A0	+	36
Q1	A0	+	42
Q1	B0	+	56
Mai-Sep	A0	–	49
Mai-Sep	A0	–	58
Mai-Sep	B0	–	28
Mai-Sep	AA	+	19
Mai-Sep	00	–	73

A	B	C	D
Jan	00	+	18
Jan	00	+	24
Jan	B0	–	32
Q1	A0	+	36
Q1	A0	+	42
Q1	B0	+	56
Mai	A0	–	49
Mai	A0	–	58
Mai	B0	–	28

A	B	C	D
Jan	00	+	18
Jan	00	+	24
Jan	B0	–	32
Q1	A0	+	36
Q1	A0	+	42
Q1	B0	+	56
Mai	A0	–	49
Mai	A0	–	58
Mai	B0	–	28
24.06.	AA	+	19
24.09.	00	–	73

Abbildung 53: von links nach rechts: r_1 bis r_4

Fallbeispiel 4: *skip*-Parameter

Die Abbildung 54 veranschaulicht den exemplarischen Gebrauch des *skip*-Parameters. Das Überspringen an sich verläuft wie beim *skip*-Parameter der Operatoren α^{D+} und α^{S+}, nur dass das daran anschließende Anonymisierungsverfahren dann ein anderes ist. Beim Nachbarschaftsmodus $grouping := neighbouring$ hat das Überspringen den Nebeneffekt, dass quasi wie beim Slicing Splitstellen entstehen, die sich abhängig vom *skip*-Parameter konstant mitverschieben würden, was bei unbedachter Mehrfachgeneralisierung intersektionelles Wissen preisgeben könnte. Der siebte Eintrag in der Abbildung ließe sich beispielsweise singulär auf den Monat Juni eingrenzen, da sich all dessen Gruppenmitglieder im relationenübergreifenden Vergleich auf März oder Mai einschränken lassen. Bei den freien Gruppenbildungsverfahren liegt ein solcher Nebeneffekte nicht vor. Erneut sei deutlich darauf hingewiesen, dass der privacy-konforme Gebrauch des Parameters hier vollständig dem Nutzer obliegt. Falls die hier übersprungenen Einträge nicht a priori schon einmal anonymisiert wurden und die Relationen r_2 und r_3 veröffentlicht werden sollten, hätte man diese Einträge beim Schutzvorgang übergangen, was im Falle von Generalisierung sogar explizit erkennbar sein würde.

$$r_2 := \alpha^{G+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=include,skip=4}(r_1)$$
$$r_3 := \alpha^{G+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=include,skip=3}(r_1)$$

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	−	32
20.03.	A0	+	36
25.03.	A0	+	42
31.03.	B0	−	56
24.06.	A0	+	49
08.05.	A0	+	58
09.05.	B0	−	28
02.05.	AA	−	19

$\Longrightarrow$

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	−	32
20.03.	A0	+	36
Mär-Jun	A0	+	42
Mär-Jun	B0	−	56
Mär-Jun	A0	+	49
Mai	A0	+	58
Mai	B0	−	28
Mai	AA	−	19

A	B	C	D
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	−	32
Mär	A0	+	36
Mär	A0	+	42
Mär	B0	−	56
Mai-Jun	A0	+	49
Mai-Jun	A0	+	58
Mai-Jun	B0	−	28
Mai-Jun	AA	−	19

Abbildung 54: von links nach rechts: r_1 bis r_3

Korrekterweise würde der *skip*-Parameter eingesetzt werden, um an bereits generalisierte Einträge anzuschließen, was Abb. 55 verdeutlichen soll.

$$r_2 := \alpha^{G+}_{X=\{A\},k=2,grouping=neighbouring;rest=include}(r_1)$$
$$r_4 := r_2 \cup r_3$$
$$r_5 := \alpha^{G+}_{X=\{A\},,k=2,grouping=neighbouring;rest=include,skip=4}(r_4)$$

A	B	C	D
Jan	00	+	18
Jan	00	+	24
Mär	B0	−	32
Mär	A0	+	36

$\Longrightarrow$

A	B	C	D
Jan	00	+	18
Jan	00	+	24
Mär	B0	−	32
Mär	A0	+	36
25.03.	A0	+	42
31.01.	B0	−	56
02.05.	A0	+	49
08.05.	A0	+	58

$\Longrightarrow$

A	B	C	D
Jan	00	+	18
Jan	00	+	24
Mär	B0	−	32
Mär	A0	+	36
Q1	A0	+	42
Q1	B0	−	56
Mai	A0	+	49
Mai	A0	+	58

Abbildung 55: von links nach rechts: r_2, r_4 und r_5

Fallbeispiel 5: *cut*-Parameter

Die Abbildung 56 veranschaulicht den exemplarischen Gebrauch des *cut*-Parameters. Das Abschneiden erlaubt das abgezählte Ausblenden der letzten Einträge und ist üblicherweise zum Verkürzen der letzten Gruppe gedacht. In gewisser Weise kann der *cut*-Parameter als entkoppelter vierter *rest*-Modus begriffen und genutzt werden, der den überschüssigen Rest mitgeneralisiert und erst danach den ungewollten Überschuss unterdrückt. Relation r_2 repräsentiert hierbei den internen Zwischenschritt, bevor dann wie in r_3 das abschließende Abschneiden erfolgt.

$$r_2 := \alpha^{G+}_{X=\{A\},k=3,grouping=neighbouring;rest=include,cut=-2}(r_1)$$
$$r_3 := \alpha^{G+}_{X=\{A\},k=3,grouping=neighbouring;rest=include}(r_1)$$

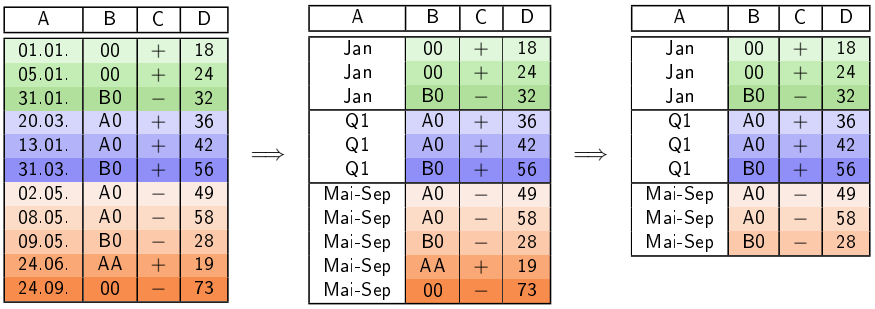


Abbildung 56: von links nach rechts: r_1 bis r_3

Besonders nützlich kann sich dieser Parameter bei Datenströmen erweisen, was das hieran anschließende Fallbeispiel veranschaulichen soll. Er erlaubt nämlich das Im-Hintergrund-Halten von zusätzlichen Werten, die noch nicht ergebnis-, aber schon berechnungsrelevant sind.

Fallbeispiel 6: Stromdaten am Beispiel eines Jumping-Tupelfensters

Die Abbildungen 57 und 58 veranschaulichen in zehn Schritten die exemplarische Nutzung des α^{G+}-Operators zum Generalisieren eines Jumping-Tupelfensters mit einer Ausgabekapazität von neun Einträgen, einer k-Anonymität von drei Einträgen und einer zukunftsgerichteten Sprungweite von vier Einträgen unter dem. Um eine konsistent generalisierte Ausgabekapazität von neun Einträgen zu erzielen, ist eine interne Erweiterung auf elf Einträge nötig, wobei die letzten beiden Einträge nie weiter- bzw. ausgegeben werden. Die Relation r_1 repräsentiert das unanonymisierte Startfenster, das es zunächst (bis auf den Rest) vollständig zu anonymisieren gilt. In Folgeschritten sind nur noch alle neu hinzugekommene Einträge, die eine vollständige Gruppe ausbilden, zu generalisieren. Die ungeraden Relationen zeigen im Resultat jeweils die Fensterverschiebungen, die geraden wiederum gezieltes Generalisieren. Generell ist zwischen der intern gespeicherten Relation und dem weiter- bzw. ausgegebenen Ergebnis zu unterscheiden. Schritte, die eine Ergebnisausgabe repräsentieren, sind am Hochkomma gemäß der Notation $r_n{}'$ zu erkennen. Intern gespeicherte Relationen r_n kommen dementsprechend ohne einen Oberstrich aus. Aus Pragmatik und zum Verständnis wurde darauf verzichtet, zusätzlich noch einen separaten Cut-Operator einzuführen.

$$r_2 := \alpha^{G+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=unchanged}(r_1)$$
$$r_2{}' := \alpha^{G+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=unchanged,cut=-2\equiv 9}(r_1)$$

$$r_3 := JUMPING_{type=tuple,hop=4}(r_2)$$
$$r_4 := \alpha^{G+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=unchanged,skip=5}(r_3)$$
$$r_4{}' := \alpha^{G+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=unchanged,skip=5,cut=-2}(r_3)$$

$$r_5 := JUMPING_{type=tuple,hop=4}(r_4)$$
$$r_6 := \alpha^{S+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=unchanged,skip=7}(r_5)$$

$$r_6{}' := \alpha^{S+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=unchanged,skip=7,cut=-2}(r_5)$$

$$r_7 := JUMPING_{type=tuple,hop=4}(r_6)$$
$$r_8 := \alpha^{S+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=unchanged,skip=6}(r_7)$$
$$r_8{}' := \alpha^{S+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=unchanged,skip=6,cut=-2}(r_7)$$

$$r_9 := JUMPING_{type=tuple,hop=4}(r_8)$$
$$r_{10} := \alpha^{S+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=unchanged,skip=5}(r_9)$$
$$r_{10}{}' := \alpha^{S+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=unchanged,skip=5,cut=-2}(r_9)$$

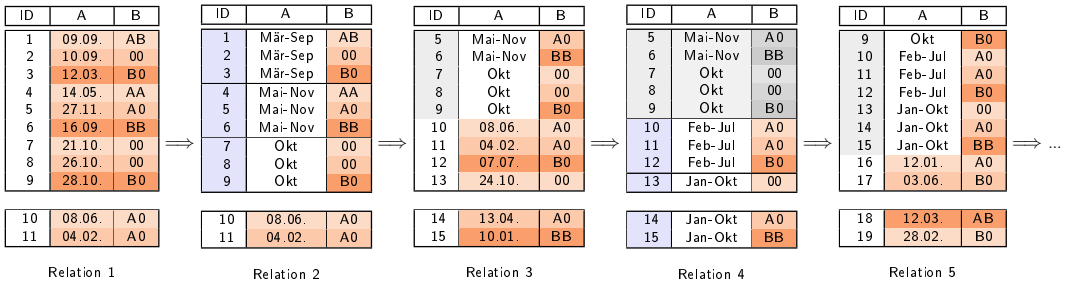


Abbildung 57: von links nach rechts: r_1 bis r_5 (Teil 1 von 2)

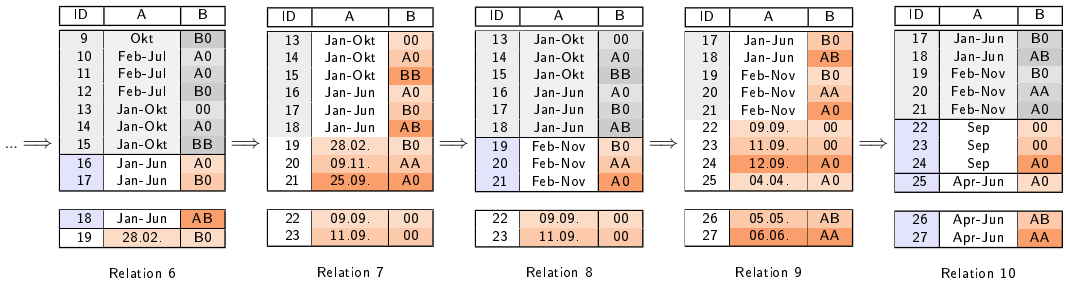


Abbildung 58: von links nach rechts: r_6 bis r_{10} (Teil 2 von 2)

Fallbeispiel 7: Umkehr der Arbeitsweise

Falls es das Anwendungsumfeld erfordert, dass sich die zu überspringenden Einträge am unteren Relationenende befinden, sei abschließend auch noch eine anregende Notationen zur Umkehr der Arbeitsweise vorgeschlagen. Aufgrund der Spiegelung würde der *skip*-Parameter am unteren Ende ansetzen und die optionale Zusatzkapazität sowie der beschneidende *cut*-Parameter würden wiederum am oberen Ende ansetzen. Wie bereits bei α^{S+} liegt nun eine enge Kopplung von Parameter vor, die eine Erweiterungen des *skip*-Parameters zu kompliziert macht. Aus diesem Grunde sei als intuitive Notation auch hier einfach ein zusätzlicher

optionaler *reverse*-Parameter bevorzugt. Ein exemplarisches Beispiel zur Veranschaulichung ist abschließend in Abb. 59 zu finden.

$$\text{Default: } \alpha^{G+}{}_{X,k,grouping}(r) \equiv \alpha^{G+}{}_{X,k,grouping;[...];reverse=false}(r)$$

$$r_2 := \alpha^{G+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=include,skip=3,reverse=true}(r_1)$$
$$r_3 := \alpha^{G+}{}_{X=\{A\},k=3,grouping=neighbouring;rest=include,skip=3,cut=-2,reverse=true}(r_1)$$

A	B	C	D
02.05.	A0	–	49
08.05.	A0	–	58
09.05.	B0	–	28
24.06.	AA	+	19
24.09.	00	–	73
20.03.	A0	+	36
13.01.	A0	+	42
31.03.	B0	+	56
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	–	32

$\Longrightarrow$

A	B	C	D
Mai-Sep	A0	–	49
Mai-Sep	A0	–	58
Mai-Sep	B0	–	28
Mai-Sep	AA	+	19
Mai-Sep	00	–	73
Q1	A0	+	36
Q1	A0	+	42
Q1	B0	+	56
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	–	32

$\Longrightarrow$

A	B	C	D
Mai-Sep	B0	–	28
Mai-Sep	AA	+	19
Mai-Sep	00	–	73
Q1	A0	+	36
Q1	A0	+	42
Q1	B0	+	56
01.01.	00	+	18
05.01.	00	+	24
31.01.	B0	–	32

Abbildung 59: von links nach rechts: r_1 bis r_3

9 Praxisteil

Dieses Kapitel widmet sich der Realisierung einer konkreten Anonymisierungstechnik, die eine datenschutzkonforme Arbeitsweise in einem Stromdatenumfeld erzielen soll. Zunächst werden kurz die naheliegenden Möglichkeiten anhand der bereits eingeführten Fallbeispiele abgewogen. Danach werden alle Parameter herausgestellt, die sich potenziell verändern lassen. Im Anschluss wird kurz ein Stromdatenerzeuger erklärt, das aus einer Lehrveranstaltung stammt und als Grundlage für die eigenen Zwecke modifiziert wurde. Danach folgen wiederum mehrere Codefragmente an, das konkrete Verhalten in Python- oder Pseudocode beschreiben. Anschließend werden kurz die erforderlichen Schritte zur Programmausführung genannt. Zuletzt wurde noch eine exemplarische Kommandozeilenausgabe mit angefügt.

9.1 Realisierungsoptionen

Aufgrund des theoretischen Schwerpunkts und der Vielzahl an konzeptionellen Ideen ist es zuallererst wohl relativ verständlich, dass nur ein Bruchteil im Zuge dieser Arbeit überhaupt umgesetzt werden könnte. Da als Ausgangsbeispiel die exemplarische Vereinbarkeit beider Techniken im Vordergrund steht, wurde als Fenstertyp im Folgenden ein Jumping-Tupelfenster gewählt. In jedem Operator-Kapitel wurde ein solches sogar praktischerweise als Fallbeispiel veranschaulicht, das zeigt, wie dessen datenschutzkonforme Anonymisierung in den ersten vier bis fünf Iterationsschritten aussehen könnte. Die Abbildung 60 verweist nochmals direkt zum jeweiligen Fallbeispiel.

Operator	Anonymisierungstechnik	Verweis	Umsetzung
α^{D+}	einfache Verrauschung	Abschnitt 6.3.3, Fallbeispiel 3	nein
α^{S+}	Slicing	Abschnitt 7.3.3, Fallbeispiel 5	ja
α^{G+}	k-anonyme Generalisierung	Abschnitt 8.4.3, Fallbeispiel 6	nein

Abbildung 60: Umsetzungsoptionen hinsichtlich der verwendeten Fallbeispiele

Schlussendlich wurde sich für eine Umsetzung der Slicing-Technik, also Fallbeispiel 5 aus Abschnitt 7.3.3 entschieden. Die einfache Verrauschung wäre aufgrund der zeilenunhängigen Arbeitsweise selbst beim erweiterten Operator doch etwas zu schlicht gewesen. Bei der Generalisierung hätte wiederum bereits das reine Verfahren einen Großteil der Entwicklung in Anspruch genommen. Der Slicing-Operator kann fast genauso funktionsreich wie der Generalisierungsoperator sein, lässt sich aber schon durch ein simples Permutieren benachbarter Felder verwirklichen. Nicht zuletzt teilen sich beide erweiterten Operatoren dieselben Parameter, sodass vieles als Blaupause verwendet werden könnte. Die eigentliche Technik geschieht lokal und ist am einfachsten implementiert, da sie keinerlei Wissenbasis wie DGHs oder Verrauschungstabellen erfordert. Das Slicing-Verfahren stellt deshalb den bestmöglichen Mittelweg dar.

9.2 Einstellbare Parameter und Konstanten

Aufgrund der Vielzahl an funktionellen Parameter ist im Rahmen dieser Arbeit nur eine reduzierte Implementierung des α^{S+}-Operators möglich gewesen. Die manipulierbaren Parameter beschränken sich auf die allerwichtigsten: die Slicing-Größe *sliceSize*, die Sprungweite

hop sowie die Fensterkapazität c. Die erweiterte Fensterkapazität c_{extend} nimmt dementsprechend funktional abgeleitet die Größe $c + sliceSize - 1$ an. Auch iterative Parameter wie die Anzahl der übersprungenen Einträge *skip* oder der Resteinträge *remainder* werden automatisch für jeden Schleifendurchlauf bestimmt. Die restlichen Verhaltensoptionen sind über den Quelltext vorgeschrieben und würden härtere Eingriffe bzw. zusätzliche Entwicklungszeit erfordern. Eine noch detaillierte Aufschlüsselung diesbezüglich ist Abb. 61 zu entnehmen.

Bezeichner	Bedeutung	Veränderbarkeit	Anmerkung
sliceSize	Slicing-Größe	Parameter, im Quelltext veränderbar	zur Laufzeit konstant
hop	Sprungweite	Parameter, im Quelltext veränderbar	zur Laufzeit konstant
c	sichtbare Fensterkapazität	Parameter, im Quelltext veränderbar	zur Laufzeit konstant
c_extend	erweiterte Fensterkapazität	abgeleitete Konstante, automatische Berechnung	zur Laufzeit konstant
skip	Anzahl übersprungener Einträge	Iterationsvariable, automatische Berechnung	–
remainder	Anzahl von Resteinträgen ohne vollständige Gruppe	Iterationsvariable, automatische Berechnung	–
rest	Modus zum Umgang mit Einträgen einer unvollständigen Gruppe	hart codiert, im Quelltext veränderbar	$rest := unchanged$
V	Korrelationsmenge	hart codiert, im Quelltext veränderbar	$V := \{\{A\}, \{B\}\}$
pattern	Muster der Splitstellensetzung	hart codiert, gegenwärtig schwerlich veränderbar	$pattern := topDown$
reverse	Richtungsumkehr der Arbeitsweise	hart codiert, gegenwärtig schwerlich veränderbar	$reverse := false$
–	Iterationspausen	über Zusatzparameter in der Kommandozeile	siehe Abb. 62
–	Zufallszahlen-Seed für reproduzierbare Permutationen	hart codiert, im Quelltext veränderbar	standardmäßig auskommentiert

Abbildung 61: Veränderlichkeit von Parametern

9.3 Python-Vorlage

Das Stromdatenkonzept wurde freundlicherweise über einen Implementierungsansatz aus einem Data-Streaming-Projekt der „Integrierten Lehrveranstaltung: Big Data Processing (ehemals DB III)“ bereitgestellt, der nachfolgend als Grundlage genutzt wurde. Es handelt sich hierbei um einen in Python geschriebenen Minimalansatz zur Realisierung von Datenströmen. Ein Datenstrom bzw. ein Fenster wird minimalistisch als Tupelfenster fester Größenbegrenzung in Form eines Ringspeichers mit sich schrittweise verschiebenden Lese- und Schreibpositionen definiert, auf welchem sich dann diverse Methoden ausführen lassen. Über die Methoden `get()` und `put()` können Tupel aufgenommen und wieder abgestoßen werden. Nennenswerte Besonderheiten bestehen darin, dass hierbei ein Multithreading-Ansatz verfolgt wird und leere sowie volle Fenster deshalb eine blockierende Wirkung erzielen. Die genaue Funktionsweise und Implementierung ist im Kern allerdings gar nicht wirklich von Belang, da es ja vielmehr auf die generelle Vereinbarkeit von Slicing mit Stromdaten ankommt. Aspekte wie der Ringspeicher und nutzerfreundliche print-Ausgaben wurden daher grundsätzlich entfernt und Aspekte wie das Multithreading nur an elementaren Stellen bewahrt. Die folgende Auflistung hebt die wesentlichen Modifikationen gegenüber der ursprünglichen Python-Vorlage hervor:

```
strthread.py ⇒ stream-slicer.py
  class stream
    def __init__(self, name, winsize)
    def __len__(self)
    def __str__(self)
    def _isfull(self)
```

```
  def _isempty(self)
  def _enqueue(self, t)
  def _dequeue(self)
  def put(self, t)
  def get(self)
  def inspect(self)
  def inspectAt(self, index) [NEU]
  def flush(self) [NEU]
  def slice(self) [NEU]

### ab hier gelistete Funktionen existieren außerhalb der obigen Klasse
def pausingOption() [NEU]
def fountain(oStream1, oStream2) [MODIFIZIERT]
def sink(iStream1, iStream2, iStream3) [MODIFIZIERT]
def filter(iStream, oStream) [GELÖSCHT]
def filterMinMax(iStream, oStream) [GELÖSCHT]
def internal(iStream1, iStream2, oStream) [NEU]
```

9.4 Bibliotheken

Zur Inbetriebnahme des Programms ist dabei das Vorhandensein bzw. Laden folgender Python-Bibliotheken nötig:

```
1 import threading
2 import time
3 import random
4 import logging
5 import numpy as np
6 import sys
```

Die Bibliotheken `numpy` und `sys` sind aufgrund der eigenen Erweiterungen nötig geworden. Die Vorlage nutze noch `datetime`, was in der eigenen Fassung aber obsolet wurde, da die generierten Zeitstempel nun diskret gezählt werden.

9.5 Implementierung

Das erste Codefragment verdeutlicht den Initialisierungsprozess und im Besonderen die drei veränderlichen Parameter hervor. Es dürfte wohl relativ klar sein, dass es sich bei allen um positive natürliche Zahlen handeln muss.

```
1  ## changeable parameters
2  c = 6  ## visible capacity
3  sliceSize = 4  ## group size for slicing
4  HOP = 3  ## step size of the jumping tuple window
5
6  ## functionally determined parameter
7  c_extend = c + sliceSize - 1
8
9  ## random seed to reproduce the permutation of a certain test case
10 #np.random.seed(123456)
11
12 st1 = stream("Stream 1", c_extend)  ## original unanonymized data
13 st2 = stream("Stream 2", c_extend)  ## internally saved slicing data window
14 st3 = stream("Stream 3", c)  ## anonymized resultsets
15
16 ## create threads for three operators
```

```
## one source (fountain), one filter (internal), and one destination (sink)
src = threading.Thread(name='fountain', target=fountain, args=(st1,st2))
dst = threading.Thread(name='sink', target=sink, args=(st3,))
flt = threading.Thread(name='internal', target=internal, args=(st1,st2,st3))

dst.start()
src.start()
flt.start()
```

Das zweite Codefragment beschreibt den Tupelerzeuger, welcher die generative Quelle des Datenstroms darstellt. Da es im Konkreten um einen reinen Testfall geht, mit dem das korrekte Permutieren schnell und einfach überprüfbar sein soll, werden eindeutig nachvollziehbare Tripel nach dem Schema `(1, A1, B1), (2, A2, B2), (3, A3, B3)` ... erzeugt.

```
T_COUNT = 0  ## counts the number of produced tuples
...
## oStream1 = ORIGINAL, oStream2 = INTERNAL
def fountain(oStream1, oStream2):
  while True:
    while (not oStream1._isfull()) and (not oStream2._isfull()):
      global T_COUNT
      T_COUNT = T_COUNT + 1
      timestamp = T_COUNT

      attrA = "A" + str(timestamp)
      attrB = "B" + str(timestamp)
      t = (timestamp, attrA, attrB)

      oStream1.put(t)  ## just to have the unanonymized original as a reference
      oStream2.put(t)  ## internal storage
  time.sleep(random.random())  ## mimic heavy duty
```

Für die elementarsten beiden Algorithmen, die im besonderen Maße verdichtet werden mussten, wird im Folgenden zu Pseudocode gewechselt. Algorithmus 2 beschreibt das grundlegende Slicing-Prinzip, dass auf jedes neue Fenster angewandt wird.

Algorithmus 2: Stromdaten-Slicing eines Jumping-Tupelfenster

Data: Datenstrom `stream`, Tupelkapazität c, Sprungweite `HOP`, Slicing-Größe `SLICESIZE`
Result: geslicter Tupelspeicher `storage` und dementsprechend beschnittene Ergebnismenge `resultSet` für jedes neue Fenster

Tupelanzahl `SKIP` := 0;
Tupelanzahl `REMAINDER` := 0;

Tupelkapazität $c_{extend} := c +$ `SLICESIZE` $- 1$;

Tupelspeicher `storage`$[c_{extend}]$;
Ergebnismenge `resultSet`$[c]$;

`storage` := $JUMPING_{type=tuple,hop=c_{extend}}$(`storage`) *from* `stream`;

while *true* **do**
  `storage` := $JUMPING_{type=tuple,hop=HOP}(storage)$ *from* `stream`;
  `storage` := $\alpha^{S+}{}_{V,sliceSize=SLICESIZE,rest=unchanged,skip=SKIP}(storage)$;
  `resultSet` := `storage.getRange(0, c - 1)`;
  output `resultSet`;
end

Algorithmus 3 beschreibt ergänzend den iterativen Ablauf des Slicing-Operators. Das Permutieren setzt in dieser Implementierung erst mit dem iterativen Erreichen des letzten Mitglieds einer Gruppe ein.

Algorithmus 3: $\alpha^{S+}{}_{V,sliceSize=SLICESIZE,rest=unchanged,skip=SKIP}(storage)$

```
Data: Tupelspeicher storage, Tupelkapazität c_extend, Slicing-Größe SLICESIZE,
      Tupelanzahl SKIP, Tupelanzahl REMAINDER
Result: geslicter Tupelspeicher storage
/* Anmerkung: c_extend = storage.size                                   */
sliceCount := 1;
REMAINDER := (c_extend − SKIP) mod SLICESIZE;
Tupelgruppe group[SLICESIZE];
for i in range(c_extend) do
    if i < SKIP then
        /* grundsätzlich keine Eintragsveränderung                      */
    end
    if SKIP ≤ i < c_extend − REMAINDER then
        if sliceCounter mod SLICESIZE = 0 then
            rand1 = np.random.permutation(SLICESIZE);
            rand2 = np.random.permutation(SLICESIZE);
            for j in range(SLICESIZE) do
                pID := i − SLICESIZE + 1 + j;
                pr1 := i − SLICESIZE + 1 + rand1[j];
                pr2 := i − SLICESIZE + 1 + rand2[j];
                argID := storage[pID][0];
                arg1 := storage[pr1][1];
                arg2 := storage[pr2][2];
                group[j] := (argID, arg1, arg2);
            end
            for j in range(SLICESIZE) do
                pID := i − SLICESIZE + 1 + j;
                storage[pID] := group[j]
            end
        end
        sliceCount := sliceCount + 1;
    end
    if c_extend − REMAINDER ≤ i then
        /* keine Einträgsveränderung im Modus rest := unchanged         */
    end
end
SKIP = c_extend − REMAINDER − HOP;
return storage;
```

9.6 Programmausführung

Da es sich beim abgegebenen Programm um eine einzelne Datei handelt, ist dessen Ausführung relativ einfach und dessen Nutzung überdies äußerst geradlinig gehalten. Insgesamt sind die folgenden vier Schritte erforderlich:

1. Python installieren
2. zusätzlich erforderliche Bibliotheken nachladen
3. per Kommandozeile über `cd` ... den Dateipfad ändern
4. Start über den Aufruf `python stream-slicer.py`

Da ein Strom potenziell unendlich laufen kann, sind zur Nachvollziehbarkeit der Ausgaben und zur verbesserten Vorführung sogar vier Aufrufoptionen möglich. Das Programm kann dadurch wahlweise mit oder ohne automatischen Pausen ausgeführt werden. Zur Auswahl stünden:

Aufrufoption	iterative Unterbrechungsform
`python stream-slicer.py`	keine
`python stream-slicer.py pause`	Fortsetzung über Drücken der Eingabetaste
`python stream-slicer.py sleep`	pausiert standardmäßig für 2 Sekunden
`python stream-slicer.py sleep [sec]`	pausiert für eine ausgewählte Anzahl von Sekunden

Abbildung 62: Aufrufoptionen

Da der Strom potenziell unendlich laufen kann, wird das Programm ohne Eingreifen des Nutzers kein Ende finden. Der Nutzer muss also wahlweise das Programm oder die Kommandozeilenumgebung terminieren. Für unerfahrene Nutzer sei noch erwähnt, dass die Tastenkombination zum Herbeiführen eines Abbruchs kann je nach Python-Version und Betriebssystem variieren kann und sich daher nicht pauschal sagen lässt. Im Falle von aktuellen Windows-Versionen und Python 3 wäre erwartungsgemäß u. a. das Drücken von `Strg` + `Pause` / `Untbr` möglich.

9.7 Exemplarische Kommandozeilenausgabe

Abschließend sei noch eine exemplarische Kommandozeilenausgabe der ersten beiden Resultsets aufgelistet. Die abgebildete Ausgabe ließe sich mittels Randomisierungsseed `np.random.seed(123456)` jederzeit reproduzieren. Die entsprechende Seed-Zeile befindet sich bereits ganz weit unten im Quelltextende der abgegebenen Fassung, wurde standardmäßig allerdings auskommentiert.

```
SLICING of a JUMPING TUPLE WINDOW

SOURCE CODE VARIABLES:
slicing size: sliceSize=4
visible capacity: c=6
step size: hop=3

FUNCTIONALLY DETERMINED PARAMETERS:
internal capacity: c_extend=9 (the last 3 entries will be cut off before publishing)
skip and remainder are automatically calculated for every new iteration

HARD CODED CONSTANTS:
pattern: Top-down
rest: unchanged values
V = {{A},{B}} (implemented example is currently only capable of strict single attribute
partitioning)


[DEBUG] (fountain  ) produced 1 @ A1 @ B1
[DEBUG] (fountain  ) produced 2 @ A2 @ B2
[DEBUG] (fountain  ) produced 3 @ A3 @ B3
[DEBUG] (fountain  ) produced 4 @ A4 @ B4
[DEBUG] (fountain  ) produced 5 @ A5 @ B5
[DEBUG] (fountain  ) produced 6 @ A6 @ B6
[DEBUG] (fountain  ) produced 7 @ A7 @ B7
[DEBUG] (fountain  ) produced 8 @ A8 @ B8
[DEBUG] (fountain  ) produced 9 @ A9 @ B9
INTERNAL WINDOW (before slicing) = stream(name=Stream 2, q=[(1, 'A1', 'B1'), (2, 'A2', 'B2')
, (3, 'A3', 'B3'), (4, 'A4', 'B4'), (5, 'A5', 'B5'), (6, 'A6', 'B6'), (7, 'A7', 'B7'), (8, '
A8', 'B8'), (9, 'A9', 'B9')], cnt=9, rpos=0, wpos=0)


Slicing is STARTING ...
skip:0, rest:1
pos:0, i:0, (1, 'A1', 'B1') -> (1, 'A4', 'B3') - sliced
pos:1, i:1, (2, 'A2', 'B2') -> (2, 'A1', 'B4') - sliced
pos:2, i:2, (3, 'A3', 'B3') -> (3, 'A3', 'B1') - sliced
pos:3, i:3, (4, 'A4', 'B4') -> (4, 'A2', 'B2') - sliced (end of group)
pos:4, i:4, (5, 'A5', 'B5') -> (5, 'A7', 'B6') - sliced
pos:5, i:5, (6, 'A6', 'B6') -> (6, 'A6', 'B7') - sliced
pos:6, i:6, (7, 'A7', 'B7') -> (7, 'A5', 'B8') - sliced
pos:7, i:7, (8, 'A8', 'B8') -> (8, 'A8', 'B5') - sliced (end of group)
pos:8, i:8, (9, 'A9', 'B9') - rest=unchanged
Slicing is DONE.


CUT 9 down to 6, leave out the timestamp (when publishing it)
(1, 'A4', 'B3')
(2, 'A1', 'B4')
(3, 'A3', 'B1')
(4, 'A2', 'B2')
(5, 'A7', 'B6')
(6, 'A6', 'B7')
RESULTSET 1 = stream(name=Stream 3, q=[('A4', 'B3'), ('A1', 'B4'), ('A3', 'B1'), ('A2', 'B2
'), ('A7', 'B6'), ('A6', 'B7')], cnt=6, rpos=0, wpos=0)


Press enter to continue
[DEBUG] (fountain  ) produced 10 @ A10 @ B10
```

```
[DEBUG] (fountain  ) produced 11 @ A11 @ B11
[DEBUG] (fountain  ) produced 12 @ A12 @ B12
INTERNAL WINDOW (before slicing) = stream(name=Stream 2, q=[(10, 'A10', 'B10'), (11, 'A11',
'B11'), (12, 'A12', 'B12'), (4, 'A2', 'B2'), (5, 'A7', 'B6'), (6, 'A6', 'B7'), (7, 'A5', 'B8
'), (8, 'A8', 'B5'), (9, 'A9', 'B9')], cnt=9, rpos=3, wpos=3)


Slicing is STARTING ...
skip:5, rest:0
pos:3, i:0, (4, 'A2', 'B2') - skipped
pos:4, i:1, (5, 'A7', 'B6') - skipped
pos:5, i:2, (6, 'A6', 'B7') - skipped
pos:6, i:3, (7, 'A5', 'B8') - skipped
pos:7, i:4, (8, 'A8', 'B5') - skipped
pos:8, i:5, (9, 'A9', 'B9') -> (9, 'A12', 'B10') - sliced
pos:0, i:6, (10, 'A10', 'B10') -> (10, 'A10', 'B12') - sliced
pos:1, i:7, (11, 'A11', 'B11') -> (11, 'A9', 'B11') - sliced
pos:2, i:8, (12, 'A12', 'B12') -> (12, 'A11', 'B9') - sliced (end of group)
Slicing is DONE.


CUT 9 down to 6, leave out the timestamp (when publishing it)
(4, 'A2', 'B2')
(5, 'A7', 'B6')
(6, 'A6', 'B7')
(7, 'A5', 'B8')
(8, 'A8', 'B5')
(9, 'A12', 'B10')
RESULTSET 2 = stream(name=Stream 3, q=[('A2', 'B2'), ('A7', 'B6'), ('A6', 'B7'), ('A5', 'B8
'), ('A8', 'B5'), ('A12', 'B10')], cnt=6, rpos=0, wpos=0)


Press enter to continue
```

10 Schlusswort

Im Rahmen dieser Masterarbeit wurde sich tiefgreifend mit der Operationalisierbarkeit von k-Anonymität, Slicing und einfacher Verrauschung auseinandergesetzt und darüber hinaus ein reichhaltiges Konzept zur datenschutzkonformen Vereinbarkeit mit Stromdaten erarbeitet. Es hat sich gezeigt, dass alle drei Techniken über gewisse Zusatzmaßnahmen ebenso in einem Stromdatenumfeld betrieben werden können. Hierbei wurden aus Privacy-Sicht drei neue Problemklassen ermittelt, die durch Kombination der Anwendungsgebiete resultieren, und durch selbst entwickelte Lösungsansätze wiederum überwunden werden konnten. Bezüglich selbst entstandener Definitionen seien nochmals einige selbst erarbeiteten Namen, Klassifizierungen und Gedanken hervorgehoben, die durchaus aufgreifenswert sein könnten. Dies wären vornehmlich: ein eigenes Klassifizierungsschema von Fenstern hinsichtlich der Sprungweiten, wo auch etliche randständige Fenstertypen mit eigenen Namen bedacht wurden; eigene Konventionen bezüglich der Fensterlebensdauer, dem Zuwachs- und Bewegungsverhalten, der Wachstumsumgebung sowie der Verarbeitungsform; eine sehr umfangreiche Übersicht von Fenstertypen.

Die maßgebliche Größe für die Kombinierbarkeit von Stromdaten und Anonymisierungstechniken ist dabei die horizontale Lokalität der jeweiligen Verfahren. Die einfache Verrauschung erforderte die wenigsten Modifikationen. Dahinter reiht sich das Slicing ein, das sich trotz mehrerer Vorschriften ebenfalls ohne sonderliche Einbußen ausführen ließe. Das Schlusslicht bildet die Generalisierung, die im Kern die gleichen Lösungsansätze wie beim Slicing aufgreift, nun allerdings zunehmend lokaler agieren muss. Es ist wohl nicht unwahrscheinlich, dass die erdachten Lösungsansätze auch für andere zeilenübergreifende Anonymisierungsverfahren hilfreich sein könnten. Man kann wohl sagen, dass die Vereinbarkeit bzw. privacykonforme Beherrschbarkeit generell schwindet, sobald optimale bzw. großräumige Techniken angestrebt werden. Für jene dürften härtere Einschränkungen bzw. qualitative Einbußen also kaum vermeidbar sein.

Zuletzt seien noch einige Aspekte hervorgehoben, welche diese Arbeit limitierungsbedingt nur anreißen konnte. Generell beschränkte sich die praktische Umsetzung konzeptuell auf die elementaren *skip*- und *cut*-Parameter sowie die Slicing-Technik. Hätte es die Zeit erlaubt, hätte auch definitiv Interesse an der Vollimplementierung des erweiterten Slicing-Operators bestanden. Zu ergänzen wären vorrangig die *include*- und *reverse*-Parameter und die direkte Auswählbarkeit der Korrelationsmengen. Außerdem wären noch durchaus eine Kommandozeileneinbettung und alternative Wertgeneratoren überlegenswert gewesen. Das Gleiche lässt sich sicherlich auch für die Generalisierung sagen, die darüber hinaus noch diverse weitere Möglichkeiten und Herausforderungen geboten hätte, beispielsweise die Gruppierungsoptionen und im Besonderen das dynamische Gruppieren mit optionaler Zusatzkapazität und verschiedenen Zielsetzungen. Außerdem wäre gewiss der Umgang mit dynamischen Fenstergrößen und Zeitfenstern interessant gewesen. Nicht zuletzt gäbe es noch Spezialtechniken und andere Metriken, die gar nicht untersucht wurden. Darüber hinaus sollte ebenso noch einmal festgehalten werden, dass sich diese Arbeit ganz klar auf konstante Schrittweiten beschränkte, von einer Nichtwiederkehr vergessener Einträge ausging und sich auf eine direkt am Datenstrom ansetzende Anonymisierung fokussiert hat. Außerdem mussten Aspekte wie eine Evaluation der Performance oder die Parallelisierbarkeit außer Acht gelassen werden, da die umfangreiche Konzeptionierung eindeutig im Vordergrund stand. Abschließend lässt

sich sicherlich sagen, dass die Kombination von Anonymisierungstechniken mit Stromdaten definitiv noch einiges an Potenzial für die Zukunft bietet.

Literatur

[ABW03] Arasu, Arvind, Shivnath Babu und Jennifer Widom: *The CQL Continuous Query Language: Semantic Foundations and Query Execution.* Technischer Bericht 2003-67, Stanford InfoLab, 2003. `http://ilpubs.stanford.edu:8090/758/`, zuletzt aufgerufen am 17.11.2024.

[ABW06] Arasu, Arvind, Shivnath Babu und Jennifer Widom: *The CQL continuous query language: semantic foundations and query execution.* The VLDB Journal, 15:121–142, 2006.

[BFO04] Bry, François, Tim Furche und Dan Olteanu: *Datenströme.* Informatik Spektrum, 27(2):168–171, 2004.

[DKM+06] Dwork, Cynthia, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov und Moni Naor: *Our Data, Ourselves: Privacy via Distributed Noise Generation.* In: *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Band 4004 der Reihe *Lecture Notes in Computer Science*, Seiten 486–503. Springer, 2006.

[Dwo06] Dwork, Cynthia: *Differential Privacy.* In: *33rd International Colloquium on Automata, Languages and Programming, part II (ICALP 2006)*, Band 4052 der Reihe *Lecture Notes in Computer Science*, Seiten 1–12. Springer, 2006.

[Dwo08] Dwork, Cynthia: *Differential Privacy: A Survey of Results.* In: *TAMC*, Band 4978 der Reihe *Lecture Notes in Computer Science*, Seiten 1–19. Springer, 2008.

[FS10] Friedman, Arik und Assaf Schuster: *Data Mining with Differential Privacy.* In: *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Seiten 493–502. Association for Computing Machinery, 2010.

[GAE06] Ghanem, Thanaa, Walid G. Aref und Ahmed Elmagarmid: *Exploiting predicate-window semantics over data streams.* ACM SIGMOD Record, 35:3–8, März 2006.

[GH15] Grunert, Hannes und Andreas Heuer: *Slicing in Assistenzsystemen - Wie trotz Anonymisierung von Daten wertvolle Analyseergebnisse gewonnen werden können.* In: *Proceedings of the 27th GI-Workshop Grundlagen von Datenbanken, Gommern, Germany, May 26-29, 2015*, Seiten 24–29. CEUR-WS.org, 2015.

[GMM+03] Guha, Sudipto, Adam Meyerson, Nina Mishra, Rajeev Motwani und Liadan O'Callaghan: *Clustering Data Streams: Theory and Practice.* IEEE Transactions on Knowledge and Data Engineering, 15(3):515–528, 2003.

[GÖ03] Golab, Lukasz und M. Tamer Özsu: *Issues in Data Stream Management.* ACM SIGMOD Record, 32(2):5–14, 2003.

[Hau07] Hauf, Dietmar: *Allgemeine Konzepte K-Anonymity, l-Diversity and T-Closeness*, 2007. `https://dbis.ipd.kit.edu/img/content/SS07Hauf_kAnonym.pdf`, zuletzt aufgerufen am 17.11.2019.

[HSS19] Heuer, Andreas, Gunter Saake und Kai-Uwe Sattler: *Datenbanken - Implementierungstechniken*, Kapitel 10, Seite 403. MITP, 4. Auflage, 2019.

[Kle20] Klein, Eric: *Parallele Anonymisierung von großen Datenbeständen.* Bachelorarbeit, Universität Rostock, 2020. `https://eprints.dbis.informatik.uni-rostock.de/1012/`, zuletzt aufgerufen am 28.10.2024.

[LLV07] Li, Ninghui, Tiancheng Li und Suresh Venkatasubramanian: *t-Closeness: Privacy Beyond k-Anonymity and l-Diversity.* In: *23rd International Conference on Data Engineering (ICDE 2007)*, Seiten 106–115. IEEE Computer Society, 2007.

[LLXY04] Lin, Xuemin, Hongjun Lu, Jian Xu und Jeffrey Xu Yu: *Continuously maintaining quantile summaries of the most recent N elements over a data stream.* In: *Proceedings. 20th International Conference on Data Engineering*, Seiten 362–373, 2004.

[LLZM12] Li, Tiancheng, Ninghui Li, Jian Zhang und Ian Molloy: *Slicing: A New Approach for Privacy Preserving Data Publishing.* IEEE Transactions on Knowledge and Data Engineering, 24(3):561–574, 2012.

[LWZ04] Law, Yan-Nei, Haixun Wang und Carlo Zaniolo: *Query Languages and Data Models for Database Sequences and Data Streams.* In: *Proceedings of the 30th International Conference on Very large Data Bases*, Seiten 492–503, 2004.

[MKGV07] Machanavajjhala, Ashwin, Daniel Kifer, Johannes Gehrke und Muthuramakrishnan Venkitasubramaniam: *L-diversity: Privacy Beyond K-anonymity.* ACM Transactions on Knowledge Discovery from Data, 1(1), 2007.

[Müh22] Mühl, Gero: *Event-Driven Architectures.* Vorlesungsfoliensatz 10: Stream Processing, Sommersemester, 2022.

[NJTF15] Nielsen, Jan Hendrik, Daniel Janusz, Jochen Taeschner und Johann-Christoph Freytag: *D2Pt: Privacy-Aware Multiparty Data Publication.* In: *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, Seiten 105–124. Gesellschaft für Informatik e.V., 2015.

[RMCZ06] Ryvkina, Esther, Anurag S. Maskey, Mitch Cherniack und Stan Zdonik: *Revision Processing in a Stream Processing Engine: A High-Level Design.* In: *22nd International Conference on Data Engineering (ICDE)*, Seite 141, 2006.

[Sam01] Samarati, Pierangela: *Protecting Respondents' Identities in Microdata Release.* IEEE Transactions on Knowledge and Data Engineering, 13(6):1010–1027, 2001.

[SS98] Samarati, Pierangela und Latanya Sweeney: *Protecting Privacy when Disclosing Information: k-Anonymity and Its Enforcement through Generalization and Suppression.* Technischer Bericht, 1998.

[Swe02] Sweeney, Latanya: *Achieving K-anonymity Privacy Protection Using Generalization and Suppression.* International Journal on Uncertainty, Fuzziness and Knowledge-based Systems, 10(5):571–588, 2002.

[TMSF03] Tucker, Peter A., David Maier, Tim Sheard und Leonidas Fegaras: *Exploiting Punctuation Semantics in Continuous Data Streams.* IEEE Transactions on Knowledge and Data Engineering, 15(3):555–568, 2003.

[War65] Warner, Stanley L.: *Randomized Response: A Survey Technique for Eliminating Evasive Answer Bias.* Journal of the American Statistical Association, 60(309):63–69, 1965.

[WDR06] Wu, Eugene, Yanlei Diao und Shariq Rizvi: *High-performance complex event processing over streams.* In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, Seiten 407–418, 2006.

[Wik13] Wikipedia: *Nichts-zu-verbergen-Argument*, 2013. `https://de.wikipedia.org/w/index.php?title=Nichts-zu-verbergen-Argument&oldid=248573491`, zuletzt aufgerufen am 07.12.2024.

[ZS03] Zhu, Yunyue und Dennis Shasha: *Efficient elastic burst detection in data streams.* In: *Proceedings. 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Seiten 336–345, 2003.

[ÖV11] Özsu, M. Tamer und Patrick Valduriez: *Principles of Distributed Database Systems*, Kapitel 18, Seiten 723–744. Springer, 3. Auflage, 2011.

Selbständigkeitserklärung

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

Rostock, den 10.12.2024