University of Applied Sciences

Angewandte Computer-
und Biowissenschaften

HOCHSCHULE
MITTWEIDA
University of Applied Sciences

HOCHSCHULE
MITTWEIDA
University of Applied Sciences

HOCHSCHULE
MITTWEIDA
University of Applied Sciences

MASTERARBEIT

HOCHSCHULE
MITTWEIDA
University of Applied Sciences

Herr
Maximilian Jugl

Entwicklung eines Broker-Service für PPRL im Rahmen der verteilten Datenanalyse mit dem Personal Health Train

2022

University of Applied Sciences

Angewandte Computer-
und Biowissenschaften

HOCHSCHULE MITTWEIDA
University of Applied Sciences

HOCHSCHULE MITTWEIDA
University of Applied Sciences

HOCHSCHULE MITTWEIDA
University of Applied Sciences

MASTER THESIS

HOCHSCHULE MITTWEIDA
University of Applied Sciences

Herr
Maximilian Jugl

Development of a Broker Service for PPRL in the Context of Distributed Analytics with the Personal Health Train

2022

Fakultät **Angewandte Computer- und Biowissenschaften**

MASTERARBEIT

Entwicklung eines Broker-Service für PPRL im Rahmen der verteilten Datenanalyse mit dem Personal Health Train

Autor:
Maximilian Jugl

Studiengang:
Cybercrime/Cybersecurity

Seminargruppe:
CY19wC-M

Erstprüfer:
Prof. Dr.-Ing. Toralf Kirsten

Zweitprüfer:
M.Sc. Josefine Welk

Mittweida, April 2022

Bibliografische Angaben

Jugl, Maximilian: Entwicklung eines Broker-Service für PPRL im Rahmen der verteilten Datenanalyse mit dem Personal Health Train, 94 Seiten, 32 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften

Masterarbeit, 2022

Referat

In dieser Arbeit wird die Entwicklung einer Client-Server-Infrastruktur für die probabilistische Privacy Preserving Record Linkage (PPRL) vorgestellt. Ziel ist die Integration der entwickelten Dienste in eine Implementierung des Personal Health Train. Die Anwendbarkeit wird anhand von Fallbeispielen demonstriert und die Toleranz des PPRL-Ansatzes gegenüber kleinen Fehlern zwischen sonst übereinstimmenden Datensätzen hervorgehoben. Das Ergebnis ist eine robuste PPRL-Infrastruktur für den Einsatz in der verteilten Datenanalyse.

Abstract

This thesis presents the development of a client-server infrastructure for probabilistic privacy preserving record linkage (PPRL). The aim is to integrate the developed components into an implementation of the Personal Health Train. Applicability is demonstrated by means of selected case studies and error tolerance of the PPRL approach for data records with minor differences is highlighted. The result is a reliable PPRL infrastructure for use in distributed data analytics.

I. Inhaltsverzeichnis

Inhaltsverzeichnis I

Abbildungsverzeichnis II

Tabellenverzeichnis III

Listings IV

Vorwort V

1 Einleitung 1
- 1.1 Motivation . . . 1
- 1.2 Ähnliche Arbeiten . . . 2
- 1.3 Aufbau der Arbeit . . . 3

2 Record Linkage 4
- 2.1 Szenario und Lösungsansätze . . . 4
- 2.2 Deterministische Record Linkage . . . 5
- 2.3 Probabilistische Record Linkage . . . 7
- 2.4 Verwendung von Bloomfiltern für probabilistische Record Linkage . . . 9
- 2.5 Privacy Preserving Record Linkage . . . 12
  - 2.5.1 PPRL-Prozess . . . 12
  - 2.5.2 Two-party und Multi-party-Protokolle . . . 18
  - 2.5.3 Sicherheit in PPRL-Protokollen . . . 19

3 Personal Health Train 21
- 3.1 Konzeptioneller Überblick . . . 21
- 3.2 Anwendungen des PHT . . . 22
- 3.3 Funktionsweise von PHT-Implementierungen . . . 23

4 Konzeption 26

5 Encoder und Matcher 31
- 5.1 Encoder . . . 31
  - 5.1.1 Konzeptioneller Überblick . . . 31
  - 5.1.2 Datenmodell . . . 32
  - 5.1.3 Endpunkte . . . 34
  - 5.1.4 Datenvorverarbeitung . . . 35
  - 5.1.5 Maskierung . . . 37
- 5.2 Matcher . . . 39
  - 5.2.1 Konzeptioneller Überblick . . . 39
  - 5.2.2 Datenmodell . . . 40
  - 5.2.3 Endpunkte . . . 42
  - 5.2.4 Matching . . . 42

6 Broker **45**
6.1 Konzeptioneller Überblick . . . 45
6.2 Datenmodell . . . 47
6.3 Endpunkte . . . 49
6.4 Komponenten . . . 49
6.4.1 Zuordnung von Sessions . . . 49
6.4.2 Abbruchsmethoden . . . 51
6.4.3 Client für den Matcher . . . 52
6.4.4 Cache für Match-Ergebnisse . . . 53
6.4.5 Queue . . . 55
6.5 Arbeitszyklus . . . 57
6.5.1 Initialisierung . . . 58
6.5.2 Hintergrundprozess . . . 58
6.5.3 REST-Interface . . . 60

7 PHT-Integration **63**
7.1 PIX-System und Pseudonymisierung . . . 63
7.2 Resolver . . . 64
7.3 Deployment der Station-Services . . . 67
7.4 Zugklasse . . . 68

8 Ergebnisse **70**
8.1 Qualität des Matchings . . . 70
8.2 Testfall: 5. MII-Projektathon „Private Set Intersection“ . . . 73
8.3 Testfall: Synthetische Beispieldaten . . . 78
8.4 Performance-Analyse . . . 80
8.4.1 Bitweises Matching . . . 80
8.4.2 Laufzeit des Brokers . . . 82

9 Diskussion **84**
9.1 Qualitative Bewertung . . . 84
9.2 Bekannte Probleme . . . 85
9.3 Sicherheitsaspekte . . . 86
9.4 Ausblick . . . 87
9.5 Zusammenfassung . . . 89

A Anfrage und Antwort für den Encoder-Service **90**

B Anfrage und Antwort für den Match-Service **91**

C Ausgabe des Zuges für den MII-Projektathon Beispieldatensatz **92**

Literatur **93**

II. Abbildungsverzeichnis

2.1 Funktionsweise eines Bloomfilters 9
2.2 Darstellung eines möglichen PPRL-Prozesses 13
2.3 Maskierung von Daten mit Bloomfilter und q-Grammen 16
3.1 Schematischer Aufbau des PHT am Beispiel von drei Stationen 22
3.2 Kommunikation zwischen PHT-Stationen und zentraler Repository in PADME 24
4.1 Schematische Integration der PPRL-Infrastruktur in den PHT 26
4.2 Ausführungsphasen des PPRL-Zuges 29
5.1 Schematische Arbeitsweise des PPRL-Encoders 31
5.2 UML-Diagramm der Modellklassen in Verwendung des PPRL-Encoders 33
5.3 Prozess der Datenvorverarbeitung im PPRL-Encoder 35
5.4 Berechnung der Bloomfiltergröße in der Google Guava Implementierung 39
5.5 Schematische Arbeitsweise des PPRL-Matchers 40
5.6 UML-Diagramm der Modellklassen in Verwendung des PPRL-Matchers 41
5.7 Schnittmenge zweier Bitvektoren mit bitweiser Konjugation 43
5.8 Schnittmenge zweier Bitvektoren mit q-Grammen 43
5.9 Schnittmenge von zwei Bitvektoren unterschiedlicher Länge mit q-Grammen 44
6.1 Schematische Arbeitsweise des PPRL-Brokers 46
6.2 UML-Diagramm der Modellklassen in Verwendung des PPRL-Brokers 48
6.3 UML-Diagramm der Modellklassen für die Zuordnung von Sessions 50
6.4 UML-Diagramm der abstrakten Klasse für Session-Abbruchsmethoden 51
6.5 UML-Diagramm des Interfaces für den Matcher-Client 52
6.6 UML-Diagramm des Interfaces für den Bitvektor-Cache 53
6.7 UML-Diagramm des Interfaces für die Bitvektor-Queue 56
6.8 Queue-Layout in Redis für eine Beispiel-Session 57
6.9 Ablaufdiagramm des Workers im PPRL-Broker 59
6.10 Erzeugung der Queue aus neu hinzugefügten Bitvektoren 62
7.1 Kommunikationswege zwischen PPRL-Zug, Resolver und Broker 65
8.1 Genauigkeit, Sensitivität und F1-Score in Abhängigkeit von Schwellenwerten 72

8.2 Import der MII-Beispieldaten in E-PIX 74
8.3 Import der generierten MPIs in gPAS 74
8.4 Ergebnisse des PPRL-Durchlaufs auf Basis der MII-Beispieldaten 77
8.5 Ähnlichkeitsverteilung zwischen drei Clients mit je 100 Datensätzen 79

III. Tabellenverzeichnis

2.1 Beispieldatensätze für Record Linkage 4
2.2 Deterministische Record Linkage am Beispiel 6
2.3 Vergleich von Datensatzpaaren auf Ähnlichkeit 8
2.4 Maskierung von Daten mit Hashfunktionen 15
7.1 PIX-Datenschema und zugeordnete Pseudonyme an einem Beispiel 63
8.1 Vorgenommene Änderungen in synthetischen Testdaten 78
8.2 Anzahl der Bitvektor-Vergleichsoperationen pro Sekunde 81
8.3 Durchschnittliche Zeiten für Operationen des Brokers 83

IV. Listings

6.1 Cypher-Anfrage zum Erstellen von Bitvektor-Knoten 54
6.2 Cypher-Anfrage zum Einfügen von Matches zwischen Bitvektor-Knoten . . . 54
6.3 Cypher-Anfrage zum Abfragen von Matches für einen Client 55
6.4 Cypher-Anfragen zum Löschen von Knoten 55
7.1 Beispiel einer PPRL-Resolver Anfrage 66
7.2 Beispiel einer PPRL-Zug-Konfiguraiton 69
8.1 E-PIX Domänenkonfiguration für Private Set Intersection 75
8.2 Konfiguration des PPRL-Zuges für die MII-Beispieldaten 76
8.3 Timing-Code in der Redis-Queue-Implementierung 82
A.1 Beispiel einer PPRL-Encoder Anfrage 90
A.2 Beispiel einer PPRL-Encoder Antwort 90
B.1 Beispiel einer PPRL-Matcher Anfrage 91
B.2 Beispiel einer PPRL-Matcher Antwort 91
C.1 Ausgabe des PPRL-Zuges für die MII-Beispieldaten 92

V. Vorwort

Vor etwa anderthalb Jahren entschied ich mich, mein Leben komplett zu ändern. In den Monaten und Jahren zuvor bahnte sich eine sehr schwere depressive Phase an, welche durch die Folgen der COVID-19 Pandemie beschleunigt wurde. Rückblickend kann ich behaupten, dass ich die bislang schwerste Phase meines Erwachsenenlebens überstanden habe. Leider werden die psychischen Beschwerden Studierender nach wie vor nicht so ernst genommen, wie es sein sollte.

Die Deutsche Gesellschaft für Psychosomatische Medizin und Ärztliche Psychotherapie berichtete im Mai 2021 von einer Studie unter 30 000 Studierenden, welche die Selbsteinschätzung zu depressiven Symptomen im Vergleich zum Jahr 2019 untersuchte[1]. Das Ergebnis: ein drastischer Anstieg in Zukunftsängsten, Gefühlen von Vereinsamung, Verlust von gewohnten Strukturen, sozialen Kontakten und ein Anstieg im Substanzkonsum zur Selbstbewältigung. Der ZDF berichtete über eine starke Nachfrage an psychosozialen Beratungen bis zum Punkt der Überlastung. Studierende fühlen sich vernachlässigt und unbeachtet[2].

Obwohl das Thema psychischer Krankheiten in den letzten Jahrzehnten mehr in den Vordergrund gerückt ist, existieren leider immer noch eine Menge Vorurteile und Stigma. Offen über solche schweren Themen zu reden und sich als betroffene Person zu bekennen ist wichtig, um die Normalisierung des Themas psychischer Krankheiten im öffentlichen Diskurs voranzutreiben. Depressionen können jeden treffen, unabhängig von Lebensumstand, Herkunft oder Geschlecht. Jeder betroffenen Person muss zugehört und den Weg zur Selbsthilfe bereitet werden.

Ich danke an dieser Stelle meinen Eltern. Es lief nicht immer alles rund. Dennoch kann ich mir keine besseren Eltern wünschen. Ich danke Cassie, Belinda, Jakob und den ganzen Nerds. Ihr seid meine moralische Stütze in schweren Zeiten. Ich danke Maik. Du hast mir während meiner Bachelorarbeit durch eine schwere Phase geholfen, ohne es zu wissen. Ich danke meinen Therapeutinnen. Sie haben mit Ihrer Arbeit nicht nur mein, sondern auch sämtliche andere Leben zurück in die richtige Richtung gebogen. Ich danke Sascha, Lars und dem restlichen PHT-Team in Aachen. Es war eine großartige Zusammenarbeit mit euch, hoffentlich auch in der Zukunft. Zuletzt danke ich Toralf und Josi. Ihr habt mir unglaubliche Chancen eröffnet. Ich hoffe, dass unsere enge Zusammenarbeit weiterhin bestehen bleibt.

[1] Siehe https://www.dgpm.de/de/presse/presse-informationen/presse-information/auswirkungen-der-pandemie-studierende-leiden-stark-unter-einsamkeit-und-depression/

[2] Siehe https://www.zdf.de/nachrichten/panorama/corona-studierende-psyche-belastung-universitaet-100.html

1 Einleitung

Im Rahmen dieser Arbeit wird die Implementierung einer Client-Server-Infrastruktur für Privacy Preserving Record Linkage beschrieben. In diesem Kapitel wird die Notwendigkeit der Privacy Preserving Record Linkage erschlossen. Der bisherige Forschungsstand und bereits existierende Lösungen werden aufgegliedert. Anschließend wird der Aufbau der gesamten Arbeit vorgestellt.

1.1 Motivation

In der Medizininformatik hat die Arbeit an medizinischen Daten eine hohe Relevanz. Bekannte Anwendungen sind die Erstellung von Vorhersagemodellen für bestimmte Diagnosen, Entwicklung von neuen Behandlungsmethoden zur Unterstützung von medizinischen Personal und die Durchführung von Kohortenstudien [1, 2]. Hierfür müssen oftmals Datensätze aus verschiedenen Instituten miteinander verknüpft werden. Aber auch außerhalb der Medizininformatik finden sich eine Menge verschiedener Anwendungen für die Verknüpfung verteilter Daten. Zensusbüros verwenden Methoden der Datenverknüpfung, um Bevölkerungszahlen abzuschätzen [3]. Ein Großteil der Arbeit in Data Warehouses entfällt auf die Beseitigung von Redundanzen [4]. Und sogar in der Astronomie müssen Messwerte von unterschiedlichen Instrumenten verknüpft werden, um astronomische Objekte genauer zu beschreiben und zu identifizieren [5].

Die Daten an einer zentralen Stelle zum Zweck der Verknüpfung zu sammeln, wurde jedoch durch wachsende Datenschutzanforderungen innerhalb der letzten Jahrzehnte erheblich erschwert. Gesetzespakete wie die Datenschutzgrundverordnung in der Europäischen Union [6], der „Health Insurance Portability and Accountability Act“ in den Vereinigten Staaten von Amerika [7] und das „Data-matching Program“ in Australien [8] zielen auf den Schutz von personenbezogenen Daten ab. Die Weitergabe von sensitiven Daten über Institutionsgrenzen hinaus ist somit in den meisten Fällen nicht ohne intensive rechtliche Absicherungen möglich [9].

Die klassischen Methoden der Datenverknüpfung operieren meistens auf Klartext und bieten durch Pseudonymisierung abermals ein minimales Maß an Sicherheit. An dieser Stelle schaffen die Methoden der Privacy Preserving Record Linkage (PPRL) Abhilfe. Ihr Ziel besteht in der Zusammenführung von maskierten Daten ohne Rückschlüsse auf die ursprünglichen Daten zu ermöglichen. Ansätze aus der deterministischen und probabilistischen Record Linkage wurden im letzten Jahrzehnt um weitere Aspekte erweitert, um sensitive Daten zu verschleiern und zu schützen. An quelloffenen Implementierungen mangelt es jedoch zum Zeitpunkt der Arbeit.

In dieser Arbeit wird die Implementierung eines PPRL-Protokolls unter Verwendung von Bloomfiltern vorgestellt. Hierfür werden eine Reihe an Webdiensten konzipiert und umgesetzt, welche den Einsatz von PPRL mithilfe einfacher Schnittstellen und Funktionsaufrufen ermöglichen. Die Funktionsweise wird anhand einer Implementierung des Personal Health Train — einem Ansatz zur verteilten Datenanalyse — demonstriert und ausgewertet.

1.2 Ähnliche Arbeiten

Schnell et al. [10, 11] stellten als Erste ein PPRL-Protokoll unter Verwendung von Bloomfiltern vor. Zwei Teilnehmer, welche einen Abgleich ihrer Daten durchführen möchten, besitzen unterschiedliche Datenbasen mit einer Menge von Datensätzen. Sie führen eine Normalisierung der Datensätze aus und teilen diese in Textfragmente der Länge q auf. Diese werden q-Gramme genannt.

Für jeden Datensatz wird daraufhin ein Bloomfilter erzeugt, in den die q-Gramme eingefügt werden. Bloomfilter sind probabilistische Datenstrukturen, welche im Kontext der PPRL verwendet werden, um Daten zu verschleiern und dennoch ihre Vergleichbarkeit zu ähnlichen Daten zu erhalten. Anschließend senden die beiden Teilnehmer ihre Bloomfilter an eine zentrale Verknüpfungseinheit, welche ein Kreuzprodukt aus den zugesendeten Bloomfiltern bildet, um die Menge aller möglichen Bloomfilterpaare zu erhalten. Jedes Paar wird auf die Ähnlichkeit zueinander überprüft, indem die Anzahl der übereinstimmenden Bits berechnet wird. Mit dem Dice-Koeffizienten — einem Ähnlichkeitsmaß für Mengen im mathematischen Sinn — wird die Ähnlichkeit in einer Wahrscheinlichkeit der Übereinstimmung ausgedrückt. Liegt diese Wahrscheinlichkeit über einem vorher festgelegten Schwellenwert, so wird ein Paar als hinreichend ähnlich beziehungsweise identisch klassifiziert.

Die Notwendigkeit der Berechnung des Kreuzproduktes der Bitvektoren von mehreren Teilnehmern wirkt sich unmittelbar auf die Laufzeit des PPRL-Protokolls aus. Um die Rechenzeit für große Datenmengen in einen akzeptablen Zeitraum zu verschieben, setzten Karakasidis und Verykios [12] eine Blocking-Komponente ein, welche die Anzahl der Vergleiche drastisch reduziert. Beim Blocking werden ähnliche Datensätze an der Datenquelle noch vor der Maskierung miteinander gruppiert. Allerdings kommt dieser Ansatz zu Kosten der Genauigkeit des Matchings.

Die erste Anwendung von Bloomfiltern für PPRL im großen Maßstab fand durch Randall et al. [13] statt. An über 26 Millionen klinischen Datensätzen wurde die Anwendbarkeit dieses Ansatzes auf sehr große Datenbasen demonstriert und verifiziert. Ihr Ansatz entspricht dem von Schnell et al. mit einem zusätzlichen Blocking-Schritt vor dem Matching. Sie identifizieren das Blocking als einen wichtigen Schritt, um die Laufzeit des PPRL-Protokolls auf Basis von Bloomfiltern drastisch zu verkürzen.

Sehili et al. [14] verfolgten einen anderen Ansatz, um die Anzahl der Vergleichsoperationen zu minimieren. Sie bauen auf dem „PPJoin"-Algorithmus von Xiao et al. [15] auf und wenden diesen auf Bloomfilter an, um Paare, welche keine hinreichende Ähnlichkeit zueinander aufweisen, noch vor dem Matching zu eliminieren. Zudem führen sie eine Implementierung ihres adaptierten Algorithmus für die Berechnung auf GPUs durch und demonstrieren die dadurch möglichen Laufzeitgewinne. Zuvor stellten Bachteler et al. [16] bereits eine Methode des Filterns von Bloomfilterpaaren unter der Verwendung von Multibit-Trees vor. Diese Datenstruktur wurde ursprünglich für das schnelle Durchsuchen von molekularen Fingerabdrücken in Biodatenbanken konzipiert und wurde auf den PPRL-Anwendungsfall angepasst.

Neben Bloomfiltern wurde parallel an PPRL-Methoden auf Basis von Peer-to-peer-Kommunikation geforscht. Stammler et al. [17, 18] veröffentlichten „MainSEL", welche die Patientenverwaltungssoftware „Mainzelliste" um eine PPRL-Komponente basierend auf den Algorithmen der „EpiLink"-Software erweitert. Somit basiert die Sicherheit der Patientendaten nicht auf der Übertragung mittels Bloomfiltern, sondern auf der getunnelten Verbindung zwischen zwei Teilnehmern. Zum Zeitpunkt der Arbeit ist „MainSEL" die einzige quelloffene Implementierung eines funktionsfähigen PPRL-Protokolls.

1.3 Aufbau der Arbeit

In dieser Arbeit wird eine generische Client-Server-Infrastruktur für ein probabilistisches PPRL-Protokoll vorgestellt. Zuerst werden die Grundlagen im Gebiet der Datenverknüpfung hinführend zu den Methoden der PPRL erläutert. Zusätzlich werden Bloomfilter als unterstützende Datenstruktur und der Personal Health Train als Ansatz für die verteilte Datenanalyse präsentiert. Anschließend wird der Aufbau der Infrastruktur konzipiert.

Die Implementierung der einzelnen Softwarekomponenten wird sukzessiv beschrieben. Weiterhin wird die Umsetzung von zusätzlichen Diensten geschildert, welche die Integration des PPRL-Protokolls in das Konzept des PHT ermöglichen. Zuletzt wird eine quantitative und qualitative Analyse des Protokolls, hinsichtlich dessen Fähigkeit übereinstimmende Datensätze mit einer gewissen Fehlertoleranz zu identifizieren, durchgeführt. Bekannte Probleme in- und außerhalb des Rahmens der PPRL-Implementierung werden mit Lösungsvorschlägen für zukünftige Entwicklungen verdeutlicht.

2 Record Linkage

Record Linkage beschreibt das Verbinden von Informationen, welche sich auf die gleiche Entität beziehen. In Folgendem wird anhand eines motivierenden Beispiels in diesem Kapitel die Kernproblematik dargestellt, welche die Record Linkage zu lösen versucht. Es werden die deterministische und probabilistische Record Linkage mit ihren Vor- und Nachteilen vorgestellt und anschließend die Privacy Preserving Record Linkage als Rahmen um diese Methodiken präsentiert.

2.1 Szenario und Lösungsansätze

Eine Person wird mit Verdacht auf einen schweren Grippeverlauf in das *Heinrich-Blau-Krankenhaus* eingeliefert. Dort stellt sich heraus, dass sich das Krankheitsbild mit keinen der bekannten Influenzaviren deckt. An den Folgetagen treffen mehrere Meldungen von Patienten mit ähnlichen Symptomen und Krankheitsverläufen aus unterschiedlichen Kliniken ein. Von betroffenen Patienten entnommene Abstriche werden zur weiteren Untersuchung an die zuständigen Virologen in den nahegelegenen Laboren geschickt.

Wenige Tage später meldet das *Heinrich-Blau-Krankenhaus* eine neue bisher unbekannte Virenart. Um einer unkontrollierten Ausbreitung des Erregers zuvor zu kommen, wird um die Kooperation der Fluggesellschaften geworben. Sämtliche Passagierdaten der letzten zwei Wochen sollen auf die Identitäten der betroffenen Patienten überprüft werden. Passagiere, die mit einer der infizierten Personen in einem Flug saßen, sollen kontaktiert und gewarnt werden. *GeorgAir* ist eine dieser Fluggesellschaften. Exemplarisch sind in der Tabelle 2.1 Auszüge aus den Personendatenbeständen der beiden genannten Institutionen dargestellt.

Patienten-Nr.	**Vorname**	**Nachname**	**Geschlecht**	**Geburtsdatum**	**Krankenkasse**	...
131220933	Emma	Horn	Weiblich	24.10.1993	TK	...
131220934	Erica	Freund	Weiblich	19.12.1967	AOK	...
131220935	Yannik	Geiger	Männlich	04.05.1989	TK	...
131220936	Gunther	Auffahrt	Männlich	22.01.1944	IKK	...

(a) Patientendatenstamm des Heinrich-Blau-Krankenhauses

Kunden-Nr.	**Vorname**	**Nachname**	**Geschlecht**	**Geburtsdatum**	**Herkunftsland**	...
5378238	Rico	Nolte	Männlich	1992/12/24	Deutschland	...
5378239	Eva	Fiend	Weiblich	1977/10/09	Großbritannien	...
5378240	Peter	Bierwirth	Männlich	1971/02/11	Deutschland	...
5378241	Erika	Freund	Weiblich	1967/12/19	Deutschland	...

(b) Kundendatenstamm von GeorgAir

Tabelle 2.1: Beispieldatensätze für Record Linkage anhand zwei verschiedener Datenbasen. Zwei ähnliche Datensätze sind grau hinterlegt.

Es ist zu erkennen, dass Frau Freund in den Datensätzen beider Institutionen vertreten ist. Die beiden Datensätze zusammenzuführen, ist Ziel der Record Linkage. Jedoch sind bereits einige Probleme erkennbar, die den Prozess erschweren.

- Die Datenschemata zwischen den beiden Institutionen sind verschieden.
- Daten können unterschiedlich formatiert sein, wie zu sehen beim Geburtsdatum.
- In beiden Datenbasen sind nur eine begrenzte Anzahl an persönlichen Eigenschaften vorhanden.
- Durch technische oder menschliche Fehler können typographische Fehler zwischen sonst identischen Datensätzen auftreten.

Einer robusten Methode sollte es möglich sein, die Datensätze von Frau Freund zusammenzuführen trotz des Tippfehlers im Vornamen. Hierfür muss eine Definition für die Ähnlichkeit zweier Datensätze etabliert werden. Unter dem Begriff der Ähnlichkeit ist im Kontext der Datenverknüpfung die Gleichartigkeit zweier Entitäten zu verstehen [9].

In der deterministischen Record Linkage werden Datensätze anhand einer Reihe von Regeln auf ihre Ähnlichkeit zueinander überprüft. Solche Regeln können die exakte Übereinstimmung bestimmter identifizierender Attribute zwischen Datensätzen umfassen. Es können aber auch aus vorhandenen Attributen neue zusammengesetzte Schlüsselwerte abgeleitet werden, welche zwischen ähnlichen Datensätzen identisch sein sollen.

Bei der probabilistischen Record Linkage wird die Ähnlichkeit zwischen zwei Datensätzen als Wahrscheinlichkeit ausgedrückt. Diese Wahrscheinlichkeit sagt aus, wie wahrscheinlich es ist, dass sich zwei Datensätze auf die gleiche Entität beziehen. Hierfür werden diverse Ähnlichkeitsmaße auf Paare von Attributwerten angewendet und für einen Datensatz zusammengefasst. Üblicherweise werden zwei Datensätze als identisch angesehen, wenn die berechnete Wahrscheinlichkeit über einem gewissen Schwellenwert liegt. In den folgenden Abschnitten wird genauer auf die Methoden der deterministischen und probabilistischen Record Linkage anhand des genannten Beispiels eingegangen.

2.2 Deterministische Record Linkage

Bei der deterministischen Record Linkage werden als Kriterium für das Übereinstimmen von zwei Datensätzen eine oder mehrere Identifikatoren (kurz: IDs) verwendet, welche zwischen den Datensätzen identisch sein müssen [19]. Diese Eigenschaft wird Vergleichsschlüssel, im Englischen „match key“, genannt. Nur wenn der Vergleichsschlüssel zwischen zwei Datensätzen identisch ist, können sie zusammengeführt werden.

Sowohl das Krankenhaus als auch die Fluggesellschaft führen technische IDs in Tabelle 2.1. Meistens sind das fortlaufende Nummern oder Zeichenketten, welche innerhalb des Datenschemas garantiert einzigartig sind. Dazu zählen die Patientennummer im Krankenhausdatensatz und die Kundennummer im im Fluggesellschaftsdatensatz. Da diese jedoch nur innerhalb der jeweiligen Institution einzigartig sind, eignen sie sich nicht als Vergleichsschlüssel.

Aus der Datenmodellierung sind fachliche Schlüssel bekannt. Diese schließen Attribute ein, welche sich über einen langen Zeitraum kaum oder gar nicht ändern. Solche Schlüssel werden in der Datenverknüpfung auch „Quasi-Identifikatoren“ (kurz: QIDs) genannt. Werden mehrere QIDs miteinander verknüpft, so ist es möglich Personen mit hoher Genauigkeit zu identifizieren. [9]

Zu solchen Attributen zählen Vorname, Nachname, Geschlecht und Geburtstag, welche allesamt in beiden Schemata vertreten sind. Ein möglicher Vergleichsschlüssel könnte diese Eigenschaften auf Gleichheit überprüfen. Und obwohl mit Sicherheit in fehlerfreien Datensätzen alle relevanten Entitäten miteinander verknüpft werden könnten, ist dies im Fallbeispiel nicht gegeben. Frau Freund würde nicht erkannt werden aufgrund des Fehlers im Vornamen zwischen den beiden Datensätzen.

Eine Alternative bieten hier synthetische Schlüssel, welche beispielsweise aus einer Bildungsvorschrift erzeugt werden können. So können sich Krankenhaus und Fluggesellschaft darauf einigen, ihre Datenschemata um eine weitere Eigenschaft zu erweitern. Diese soll sich aus den ersten beiden Zeichen des Vornamens, den letzten beiden Zeichen des Nachnamens, des Geschlechts als einziges Zeichen und den letzten beiden Ziffern des Geburtsjahres zusammensetzen.

Patienten-Nr.	**Vorname**	**Nachname**	**Geschlecht**	**Geburtsdatum**	**Schlüssel**	...
131220933	Emma	Horn	Weiblich	24.10.1993	EMRNW93	...
131220934	Erica	Freund	Weiblich	19.12.1967	**ERNDW67**	...
131220935	Yannik	Geiger	Männlich	04.05.1989	YAERM89	...
131220936	Gunther	Auffahrt	Männlich	22.01.1944	GURTM44	...

(a) Patientendatenstamm des Heinrich-Blau-Krankenhauses

Kunden-Nr.	**Vorname**	**Nachname**	**Geschlecht**	**Geburtsdatum**	**Schlüssel**	...
5378238	Rico	Nolte	Männlich	1992/12/24	RITEM92	...
5378239	Eva	Fiend	Weiblich	1977/10/09	EVNDW77	...
5378240	Peter	Bierwirth	Männlich	1971/02/11	PETHM71	...
5378241	Erika	Freund	Weiblich	1967/12/19	**ERNDW67**	...

(b) Kundendatenstamm von GeorgAir

Tabelle 2.2: Deterministische Record Linkage am Beispiel mit synthetischen Schlüsseln. Übereinstimmende Schlüssel sind fett hervorgehoben.

Tabelle 2.2 stellt das Ergebnis dieses Prozesses dar. In diesem Fall würden die Datensätze von Frau Freund zusammengeführt werden. Diese Methode ist fehlertoleranter

als der exakte Vergleich von Vor-, Nachname, Geschlecht und Geburtsdatum, birgt allerdings das Risiko falsch positiver Übereinstimmungen. So würde beispielsweise der Datensatz von Frau Erltrud Grund, geboren am 04.05.1967, den gleichen Vergleichsschlüssel erzeugen, obwohl die beiden Personen offensichtlich verschieden voneinander sind. Solche Kollisionen können mit mehr persönlichen Eigenschaften und längeren Vergleichsschlüsseln vermieden werden, jedoch auf Kosten der Fehlertoleranz.

2.3 Probabilistische Record Linkage

Bei der probabilistischen Record Linkage wird die Ähnlichkeit zwischen zwei Datensätzen als Wahrscheinlichkeit betrachtet. Je höher diese Wahrscheinlichkeit ist, desto ähnlicher sind zwei Datensätze zueinander. Ein formales Modell hierfür lieferten Fellegi und Sunter [19, 20], welche zwei Schwellenwerte vorschlagen, um Datensätze paarweise in „Match“, „möglicher Match“ oder „kein Match“ zu unterteilen. Unter einem „Match“ sind Datensatzpaare klassifiziert, die garantiert auf die gleiche Entität verweisen. Ein „möglicher Match“ besteht zwischen Datensätzen, die eine gewisse Ähnlichkeit zueinander besitzen und einer manuellen Überprüfung unterzogen werden müssen. „Kein Match“ liegt in Datensatzpaaren vor, die keine besondere Ähnlichkeit zueinander aufweisen, also grundsätzlich verschieden voneinander sind.

Für die probabilistische Record Linkage muss zuerst das Kreuzprodukt der Datensätze aus den zu vergleichenden Datenbasen gebildet werden. Weiterhin muss eine Übereinstimmungsregel festgelegt werden. Diese besagt lediglich, welche Eigenschaften miteinander übereinstimmen müssen, damit ein Datensatzpaar als Match klassifiziert werden kann [21]. Im Gegensatz zur deterministischen Record Linkage ist es nunmehr möglich eine Fehlertoleranz zwischen zwei Datensätzen über Wahrscheinlichkeiten für einzelne Felder auszudrücken.

Attribut	**Wert a**	**Wert b**	$\mathrm{sim}(a,b)$
Vorname	Erica	Erika	0,8
Nachname	Freund	Freund	1,0
Geschlecht	W	W	1,0
Geburtstag	19.12.1967	19.12.1967	1,0

(a) Erica Freund und Erika Freund

Attribut	**Wert a**	**Wert b**	$\mathrm{sim}(a,b)$
Vorname	Erica	Eva	0,4
Nachname	Freund	Fiend	0,67
Geschlecht	W	W	1,0
Geburtstag	19.12.1967	09.10.1977	0,7

(b) Erica Freund und Eva Fiend

Tabelle 2.3: Vergleich von Datensatzpaaren auf Ähnlichkeit. Die Edit-Distanz wird als Ähnlichkeitsmaß paarweise auf Attributwerte angewendet.

Im Fallbeispiel wäre eine mögliche Übereinstimmungsregel „einig im Vorname, Nachname, Geschlecht und Geburtsdatum“. Das mathematische Modell von Fellegi und Sunter schreibt nicht vor, wie die Übereinstimmung zwischen zwei Eigenschaften zu bestimmen ist. Eine Möglichkeit bietet die Edit-Distanz, auch Levenshtein-Distanz genannt, welche die Anzahl der notwendigen Einfügungs-, Lösch- und Substitutionsoperationen zählt, die auf eine Zeichenkette angewendet werden müssen, um eine andere Zeichenkette zu erhalten. Tabelle 2.3 zeigt die berechneten Ähnlichkeitswerte unter der definierten Übereinstimmungsregel für ausgewählte Datensatzpaare. Die Ähnlichkeit ergibt sich aus der folgenden Formel, wobei $\mathrm{edit}(a,b)$ für die Edit-Distanz zwischen den Zeichenketten a und b steht.

$$\mathrm{sim}(a,b) = \frac{\mathrm{edit}(a,b)}{\max\left(|a|,|b|\right)}$$

Es ist zu erkennen, dass die Datensätze von Erica Freund aus dem Patientenstamm des Krankenhauses und Erika Freund aus der Kundenliste der Fluggesellschaft eine sehr hohe Ähnlichkeit zueinander aufweisen. Bevor eine Klassifizierung in „Match“, „möglicher Match“ oder „kein Match“ stattfinden kann, müssen die berechneten Ähnlichkeiten zusammengefasst und die genannten Schwellenwerte angewendet werden.

Die Wahl solcher Schwellenwerte ist jedoch nicht trivial. Nicht jede Eigenschaft ist gleichermaßen aussagekräftig für die Ähnlichkeit zweier Datensätze. Weiterhin ist die Wahl des Ähnlichkeitsmaßes von großer Bedeutung. Wird beispielsweise die Levenshtein-Distanz mit der obigen Formel verwendet, um die Ähnlichkeit zweier Zeichenketten zu berechnen, so werden bei einem typographischen Fehler kurze Zeichenketten stärker gewichtet als längere Zeichenketten.

Nicht zuletzt ist die statistische Unabhängigkeit verschiedener Eigenschaften nicht immer gegeben [19]. Wenn zwei Datensätze im Geburtsjahr übereinstimmen, so werden sie auch eher im Vornamen übereinstimmen. Das kann beispielsweise mit Babynamen zusammenhängen, die im jeweiligen Geburtsjahr populär waren. Trotz dessen ist es möglich, Übereinstimmungsregeln zu formulieren, welche eine hohe Genauigkeit aufweisen.

2.4 Verwendung von Bloomfiltern für probabilistische Record Linkage

Bloomfilter sind probabilistische Datenstrukturen, welche in der probabilistischen Record Linkage Verwendung finden um die Ähnlichkeit zweier Datensätze zueinander zu vergleichen. Sie haben sich für diesen Zweck als besonders schnell, ressourcenschonend, fehlertolerant und effizient erwiesen. In diesem Abschnitt werden Bloomfilter im

Allgemeinen beschrieben, bevor im Anschluss auf deren Verwendung in der Record Linkage eingegangen wird.

Bloomfilter wurden erstmals von Burton Howard Bloom [22] im Jahr 1970 beschrieben. Sie ermöglichen das schnelle Überprüfen, ob ein bestimmter Wert in einem Datenstrom bereits aufgetreten ist. Bloomfilter sind Bit-Arrays der Länge m, wobei alle Bits anfangs auf 0 gesetzt sind. Zusätzlich werden k unabhängige Hashfunktionen $h(x)$ benötigt. Eine Hashfunktion bildet beliebig große Daten auf einen festgelegten Wertebereich ab. Es wird davon ausgegangen, dass die Ausgabe jeder Hashfunktion über die Größe des Bloomfilters gleichverteilt ist. Das bedeutet, dass jedes Bit eine Wahrscheinlichkeit von annähernd $1/m$ besitzt, gesetzt zu werden.

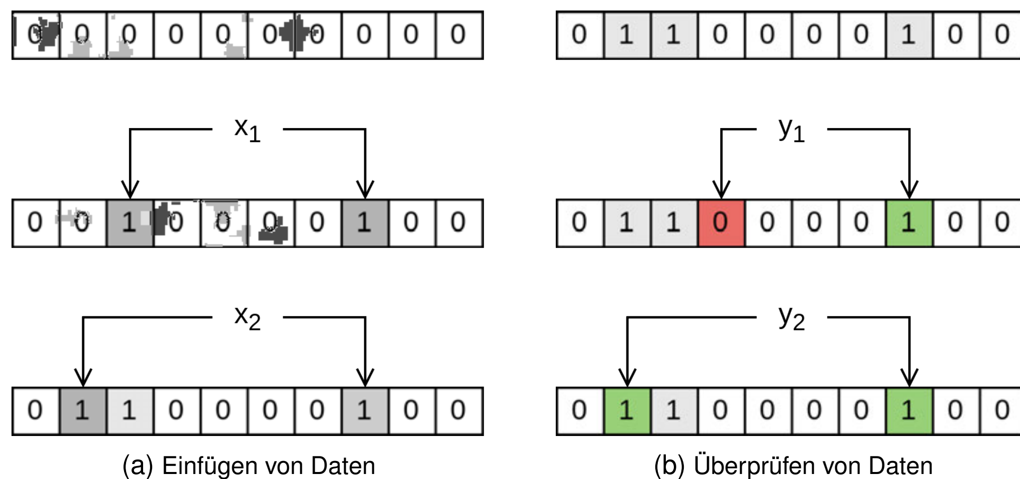


Abbildung 2.1: Funktionsweise eines Bloomfilters mit zwei Hashfunktionen und 10 Bits. Bits, welche beim Einfügen eines neuen Datums vorher noch nicht gesetzt waren, sind dunkelgrau hinterlegt. Übereinstimmende Bits beim Überprüfen der Existenz eines Datums im Bloomfilter sind grün, nicht übereinstimmende Bits sind rot gekennzeichnet. (adaptiert von [23])

Sei $X = \{x_0, x_1, \ldots, x_{a-1}\}$ eine Menge von Daten, die in einen Bloomfilter eingefügt werden soll. Für jedes Datum $x \in X$ werden zuerst alle Hashwerte $h_i(x)$ berechnet, wobei $0 \leq i < k$. Jeder Hashwert referenziert ein Bit im Bit-Array, welches folglich auf 1 gesetzt wird. Die Position im Bit-Array wird mit $h_i(x) \bmod m$ berechnet. Ist das Bit an der berechneten Position bereits gesetzt, so ändert sich nichts.

Sei $Y = \{y_0, y_1, \ldots, y_{b-1}\}$ eine Menge von Daten. Es soll überprüft werden, ob Daten aus Y in X vertreten sind. Wieder werden Positionen im Bit-Array mit den Hashwerten $h_i(y)$ berechnet. Ist für ein Datum y ein Bit an einer der berechneten Positionen nicht gesetzt, so wurde y mit Sicherheit noch nicht gesichtet. Sind allerdings alle Bits für ein Datum y gesetzt, so kann mit einer gewissen Wahrscheinlichkeit davon ausgegangen werden, dass y bereits gesichtet und in den Bloomfilter eingefügt wurde. Die beschriebenen Operationen — das Einfügen und das Überprüfen der Existenz von Daten in einem Bloomfilter — sind in Abbildung 2.1 dargestellt.

Sind alle Bits für ein Datum y gesetzt, dann kann dies auch durch Kollisionen mit den gesetzten Bits eines anderen Datums oder einer Kombination mehrerer Daten zustande kommen. Deswegen sind Bloomfilter mit einer Falsch-positiv-Rate p behaftet [23], welche von der Anzahl der Bits m, der Anzahl der Hashfunktionen k, der Anzahl der zu erwarteten Einfügen-Operationen n abhängig ist und wie folgt berechnet werden kann.

$$p = \left(1 - \mathrm{e}^{-\frac{kn}{m}}\right)^k$$

Daraus ergibt sich für die Parameter des Bloomfilters eine Proportionalität. Je mehr Hashfunktionen verwendet werden, je mehr Daten eingefügt werden, je weniger Bits vorhanden sind, desto höher ist die Wahrscheinlichkeit eines falsch positiven Ergebnisses bei der Überprüfung der Zugehörigkeit. Infolgedessen lässt sich für einen Bloomfilter eine optimale Anzahl an Hashfunktionen für eine Datenmenge mit einer vordefinierten Anzahl an Bits ermitteln, welche die Falsch-positiv-Rate minimiert.

$$k = \frac{m}{n} \ln 2$$

Hashfunktionen Für Bloomfilter können eine Vielzahl an Hashfunktionen verwendet werden. In der Bloomfilter-Implementierung in *Google Guava*[3] wird standardmäßig *MurmurHash3* verwendet. *MurmurHash* ist eine nicht-kryptografische Hashfunktion und zielt darauf ab sehr schnell und effizient Hashwerte berechnen zu können. [24]

Baqend's Bloomfilter[4] für Java unterstützt neben *MurmurHash3* weitere nicht-kryptografische Hashfunktionen als auch alle in Java verfügbaren kryptografischen Hashfunktionen, darunter *SHA-1* und *SHA-256*. Zudem ist die Implementierung eigener Hashverfahren zum Besetzen von Bloomfiltern in *Google Guava* als auch Baqend's Bloomfilter möglich.

Double Hashing Bei einer hohen Anzahl unterschiedlicher Hashfunktionen steigt der Rechenaufwand drastisch. Stattdessen ist es möglich, lediglich zwei Hashfunktionen zu kombinieren, um eine beliebige Menge an Hashwerten zu berechnen. Kirsch und Mitzenmacher [25] stellten diese Methode zuerst vor und stellten fest, dass sie vergleichbare Ergebnisse liefert ohne einer signifikanten Beeinflussung der Falsch-positiv-Rate. Im Kontext der PPRL wird dieser Ansatz als „Double Hashing" bezeichnet.

[3] Siehe `https://guava.dev/`

[4] Siehe `https://github.com/Baqend/Orestes-Bloomfilter`

Seien $h_1(x)$ und $h_2(x)$ zwei unabhängige Hashfunktionen, m die Größe des zu besetzenden Bloomfilters und k die gewünschte Anzahl an Hashwerten. Sei zudem $g_i(x)$ wie folgt definiert.

$$g_i(x) = h_1(x) + ih_2(x) \pmod m \qquad x,\ h(x),\ g(x) \in \mathbb{N};\ 0 \leq g(x) < m$$

Dann ist es möglich mit $g_i(x)$ und $1 \leq i \leq k$ die gewünschte Anzahl an Hashwerten zu berechnen. Statt der Ausgabe zwei verschiedener Hashfunktionen in der obigen Formel kann der berechnete Wert einer einzigen Hashfunktion auf zwei Hashwerte aufgeteilt werden. Am Beispiel von *SHA-256* kann die Ausgabe der Hashfunktion auf zwei 128-Bit Hashwerte zerlegt werden. Diese Optimierung ist sowohl in *Google Guava* als auch Baqend's Bloomfilter vertreten.

Enhanced Double Hashing Die Berechnung von Hashwerten mit der Formel im vorherigen Abschnitt birgt zwei signifikante Probleme. Ist das Ergebnis von $h_2(x)$ gleich Null, dann wird unabhängig von i stets derselbe Hashwert berechnet. Ist außerdem $h_2(x)$ ein Teiler von m, so können höchstens $\frac{m}{h_2(x)}$ verschiedene Hashwerte erzeugt werden. Dillinger et al. [26] schlagen mit der folgenden Formel eine Verbesserung für das Double Hashing vor, welche die beiden genannten Probleme umgeht.

$$g_i(x) = h_1(x) + ih_2(x) + \frac{i^3 - i}{6} \pmod m$$

Random Hashing Schnell et al. [27] empfehlen die Verwendung eines deterministischen Zufallszahlengenerators (im Englischen: pseudorandom number generator, kurz: PRNG), um die Indizes der zu setzenden Bits in einem Bloomfilter zu berechnen. Als Startwert des PRNGs wird ein Hashwert verwendet, welcher aus einem hashbasierten Nachrichtenauthentifizierungscode erzeugt wird. Somit wird sichergestellt, dass für dieselben Elemente stets dieselben Hashwerte erzeugt werden.

2.5 Privacy Preserving Record Linkage

Das gewählte Fallbeispiel für den institutionsübergreifenden Datenabgleich gestaltet sich mit den bisher beschriebenen Methoden der Record Linkage schwierig. Die Verwendung von personenbezogenen Daten über Institutionsgrenzen hinaus ist rechtlich abgesichert und restriktiv. In der Europäischen Union legt seit 2016 die Datenschutz-Grundverordnung den Grundstein für die Verwendung und den Austausch personenbezogener Daten in ihren Mitgliedsländern [6]. Für die Record Linkage bedeutet dies,

dass die klassischen Methoden an die stetig wachsenden Datenschutzanforderungen angepasst werden müssen.

Die Hauptherausforderung in der deterministischen Record Linkage liegt in der Notwendigkeit einer gemeinsamen ID zwischen mehreren Datenbasen. Universelle IDs wie die Personalausweisnummer kommen nicht in Frage, da bei weitem nicht jeder Dienstleister legitimiert ist diese zu erfassen. Und selbst bei der Verwendung zusammengesetzter Vergleichsschlüssel, wie in Tabelle 2.2 demonstriert, ist die Gefahr groß, dass bei einem Datenleck der zugewiesenen IDs die natürlichen Personen eindeutig identifiziert werden können. Solche IDs fungieren letzten Endes als Pseudonyme für die wahren Identitäten und bieten nur eine Scheinsicherheit.

Auch die probabilistische Record Linkage ist in ihren Grundzügen schwer mit aktuellen Datenschutzstandards vereinbar. Da ein Kreuzprodukt über mehrere Datenbasen erstellt werden muss, wird vorausgesetzt, dass die teilnehmenden Institutionen befähigt sind, personenbezogene Daten für diesen Zweck weiterzugeben. Die rechtlichen Hürden sind hierfür oftmals schwer überwindbar [28]. Das wirkt sich auch unmittelbar auf die Bestimmung eines angemessenen Schwellenwertes aus, da eine gewisse Grundkenntnis über die Ausprägungen der Daten an den einzelnen Institutionen noch vor der Durchführung der Record Linkage vorausgesetzt ist.

Über die vergangenen Jahrzehnte wurden neue Methoden für Record Linkage entwickelt, um den wachsenden Datenschutzbestimmungen gerecht zu werden. Diese Methoden werden eingeordnet in das Gebiet der Privacy Preserving Record Linkage, kurz PPRL.

2.5.1 PPRL-Prozess

Der PPRL-Prozess verläuft im Wesentlichen in zwei Phasen: die Verarbeitung der zu verknüpfenden Daten an den Datenquellen und die Zusammenführung von Daten [28]. Zuerst führen die Teilnehmer eine Maskierung der Datensätze aus ihren Datenbasen durch. Die Maskierung dient der Verschleierung der originalen, oftmals sensitiven, Daten. Maskierte Daten dürfen keine Rückschlüsse auf die Daten erlauben, aus denen sie generiert worden sind. In der Praxis muss also der Rückschluss auf die ursprünglichen Daten schwer oder unausführbar sein. Danach wird ausschließlich auf den maskierten Daten die Verknüpfung durchgeführt. Diese kann außerhalb der Institutionsgrenzen, oder auch direkt zwischen den einzelnen Teilnehmern, durchgeführt werden. In den folgenden Absätzen wird der PPRL-Prozess am Beispiel der Abbildung 2.2 in feinere Teilschritte untergliedert und beschrieben.

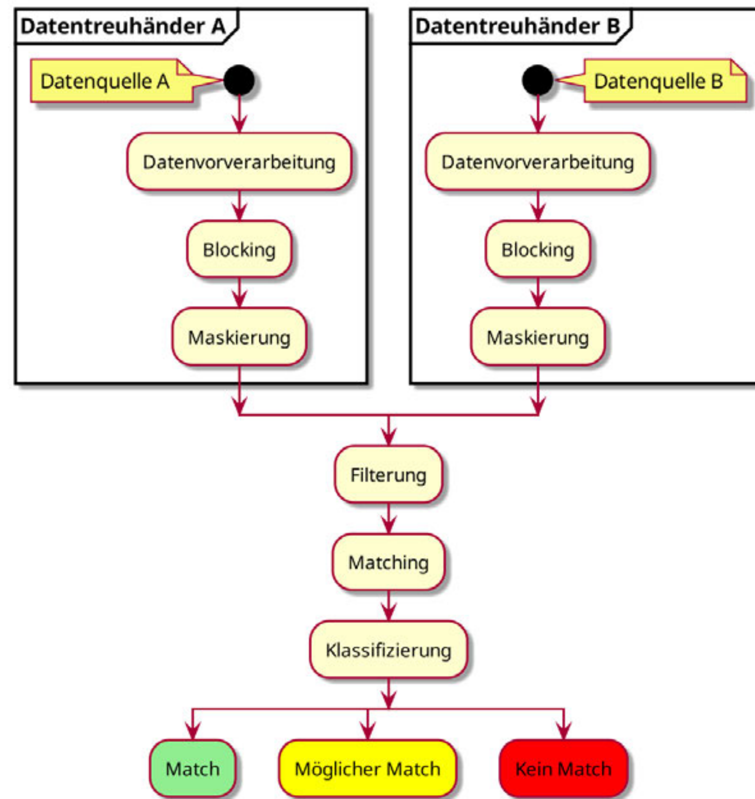


Abbildung 2.2: Darstellung eines möglichen PPRL-Prozesses. Zwei Teilnehmer führen eine Reihe an Vorverarbeitungs- und Maskierungsschritten an ihren eigenen Daten aus. Diese werden an einer zentralen Stelle gesammelt und verglichen. Je nach Grad der Ähnlichkeit erfolgt eine Klassifizierung in eine von drei Gruppen. (adaptiert von [28])

Vorverarbeitung Bei der Datenvorverarbeitung wird eine Standardisierung und Normalisierung der Daten durchgeführt [29]. Im Bereich der Standardisierung sind beispielsweise folgende Operationen möglich:

- Formatierung von Datumsangaben als „[Tag].[Monat].[Jahr]“ oder „DD.MM.YYYY“
- Geschlechtskodierung als „M“ für männlich, „W“ für weiblich und „D“ für divers
- Kodierung von Titeln wie „Doktor“ als „Dr.“, „Professor“ als „Prof.“
- Erweiterung von Kürzeln wie „Str.“ zu „Straße“

Bei der Normalisierung werden fehlende Werte behandelt und beispielsweise kategorische und numerische Werte in konsistente Darstellungen überführt [28]. Als Beispiel sei hier die Körpergröße genannt, welche in einer Datenbasis mit einer beliebigen Präzision angegeben werden kann. Hier macht es Sinn, die Anzahl der Nachkommastellen zu beschränken.

Blocking Das Beispiel in Abschnitt Szenario und Lösungsansätze zeigt einen Ausschnitt von zwei Datenbasen. Angenommen beide Institutionen führen Datensätze von je 10 000 Personen. Das hat zur Folge, dass insgesamt 100 Millionen Datensatzpaare verglichen werden müssen. Bei einer beliebigen Menge an Teilnehmern lässt sich dieser Verhalt formalisieren.

Sei $\Omega = \{X_1, X_2, \ldots, X_n\}$ die Menge aller Datenbasen, wobei X_i die Datenbasis der i-ten Institution darstellt. Dann lässt sich die Menge der durchzuführenden Vergleichsoperationen wie folgt berechnen.

$$\sum \left|X_i \times X_j\right| = \sum |X_i| \left|X_j\right| \quad \forall X_i, X_j \in \Omega; i \neq j; i < j$$

Die Anzahl der Vergleichsoperationen steigt rasant an, je mehr Datensätze und Datenbasen miteinander verknüpft werden müssen. Beim Blocking werden ähnliche Datensätze innerhalb einer Datenbasis noch vor der eigentlichen Verknüpfung miteinander gruppiert, um die Laufzeit des PPRL-Prozesses zu kürzen. [28]

Beim Blocking ist die Wahl der Blocking-Variablen entscheidend. Diese Variablen müssen in allen Datensätzen vertreten sein, ungefähr gleichverteilt sein und bestenfalls viele mögliche Ausprägungen und wenige Fehler enthalten [30]. Eine Möglichkeit des Blockings wurde bereits im Abschnitt Deterministische Record Linkage vorgestellt, jedoch mit einer anderen Anwendung. Die in diesem Fall gewählten Variablen sind der Vor- und Nachname, Geschlecht und Geburtsdatum. Mit einer Bildungsvorschrift wird für jeden Datensatz ein zusätzlicher Wert auf Grundlage der gewählten Variablen gebildet, welcher dann als Schlüssel für die Gruppierung verwendet wird. Das Geschlecht eignet sich jedoch nicht als Blocking-Variable, da es sehr wenige Ausprägungen besitzt.

Eine weitere Möglichkeit ist die Gruppierung von Entitäten mithilfe phonetischer Kodierungen. Dabei handelt es sich um Algorithmen, welche Zeichenketten aufgrund ihrer Klänge kodieren. Ein Beispiel im deutschsprachigen Raum ist die „Kölner Phonetik“ [31]. Diese weist jedem Zeichen in einer Zeichenkette einen Zahlencode zu, welcher kontextabhängig ist. So besitzen beispielsweise die Namen „Steffen“ und „Stephen“ den gleichen Code „8203306“. Im englischsprachigen Raum sind Soundex und Metaphone bekannte Vertreter phonetischer Kodierungen [28].

Es ist zudem möglich mehrere Blocking-Verfahren miteinander zu kombinieren. So wäre es möglich, die ersten drei Zeichen der Soundex-Kodierung des Vor- und Nachnamens zusammen mit dem Geburtsjahr als Schlüssel für das Blocking zu vereinen. Prinzipiell gibt es allerdings beim Blocking einen Kompromiss zu beachten. Je größer die Gruppen nach dem Blocking sind, desto weniger aussagekräftig sind die Ergebnisse nach dem Zusammenführen der gruppierten und maskierten Datensätze.

Maskierung Datensätze müssen vor der eigentlichen Verknüpfung maskiert werden. Dabei müssen Rückschlüsse von den maskierten zu den originalen Daten bestmöglich unterbunden werden. Es existiert eine Vielzahl an Methoden zum Maskieren von Daten, von denen eine Auswahl in dieser Arbeit vorgestellt wird. [28]

Eine Möglichkeit stellt die Verwendung von Hashfunktionen dar. Für jeden Datensatz werden die Werte der für die Verknüpfung relevanten Eigenschaften mit einer kryptografischen Hashfunktion verarbeitet. Der Nachteil dieses Ansatzes ist, dass nur exakte Übereinstimmungen gefunden werden können.

Vorname	Erica	Erika
Nachname	Freund	Freund
Geschlecht	W	W
Geburtsdatum	19.12.1967	19.12.1967
Hash	8c5599...	24a6b0...

(a) Originale Attributwerte

Vorname_Phon	07040	07040
Nachname_Phon	370062	370062
Geschlecht	W	W
Geburtsdatum	19.12.1967	19.12.1967
Hash	**b1a4cf...**	**b1a4cf...**

(b) Phonetisch kodierte Attributwerte

Tabelle 2.4: Maskierung von Daten mit *SHA-256*. Die Hashfunktion wird auf die Verkettung aller Attributwerte eines Datensatzes angewandt.

Abhilfe kann hier die Anwendung eines phonetischen Codes schaffen. Ein Beispiel mithilfe der „Kölner Phonetik“ ist in Tabelle 2.4 dargestellt. Für die Berechnung der Hashwerte werden Vor- und Nachname, Geschlecht und Geburtsdatum in genannter Reihenfolge in eine Zeichenkette kombiniert und mit der kryptografischen Hashfunktion *SHA-256* verarbeitet. Da „Erica“ und „Erika“ mithilfe des phonetischen Codes auf den gleichen Wert abgebildet werden, ist der resultierende Hashwert für die transformierten Datensätze gleich.

Eine weitere Möglichkeit der Maskierung wurde von Schnell et al. [11] vorgeschlagen unter der Verwendung von Bloomfiltern. Für einen Datensatz wird ein neuer Bloomfilter erstellt. Die Werte im Datensatz werden in Textfragmente der Länge q aufgeteilt, welche q-Gramme genannt werden[5]. Die q-Gramme werden daraufhin in den Bloomfilter eingefügt. Das zugrundeliegende Bit-Array ist nun repräsentativ für den Datensatz und kann für Verknüpfungsoperationen verwendet werden. Ein Beispiel anhand der Namen „Erika“ und „Erica“ ist in Abbildung 2.3 dargestellt. Anfangs- und Endbuchstaben werden mit Füllzeichen auf die Länge des q-Gramms gestreckt.

Bloomfilter in Verknüpfung mit q-Grammen sind praktisch, denn sie erlauben eine probabilistische Record Linkage auf Basis der besetzten Bits. Zwei Datensätze, die bis auf einige wenige typografische Fehler identisch sind, werden mit dieser Methoden eine ähnliche Besetzung der Bits in ihren Bloomfiltern aufweisen. Für den Bloomfilter sollten kryptografische Hashfunktionen verwendet werden, um den Rückschluss von gesetzten Bits auf die originalen Daten zu erschweren.

[5] In der Literatur wird öfter der Begriff n-Gramm statt q-Gramm verwendet. Da n in dieser Arbeit bereits bei Bloomfiltern für die Anzahl der zu erwartenden Elemente verwendet wird, soll q für die Länge von Textfragmenten stehen.

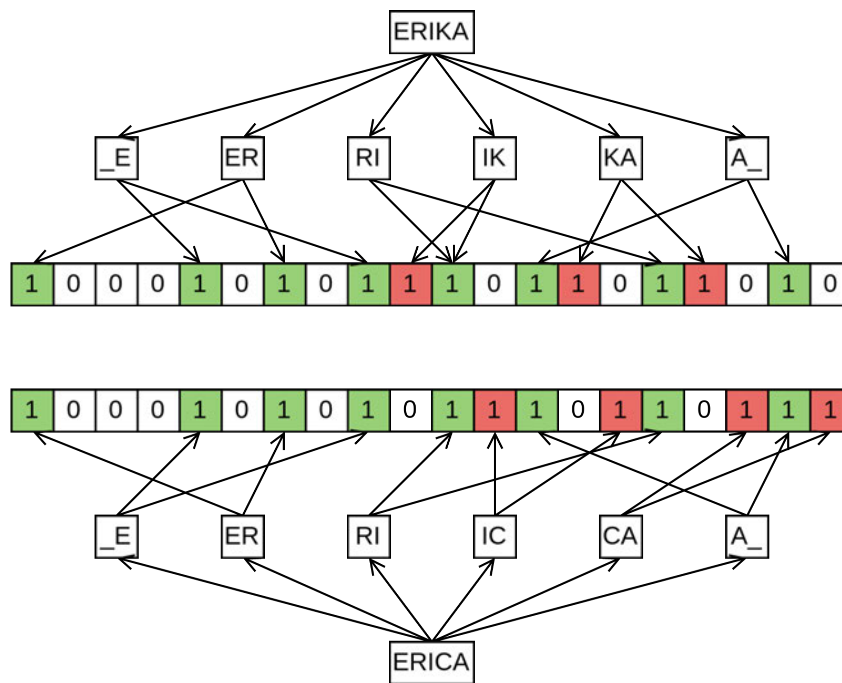


Abbildung 2.3: Maskierung von Daten mit Bloomfilter. Zeichenketten werden in q-Gramme aufgeteilt. Übereinstimmende Bits sind grün, nicht übereinstimmende Bits rot hervorgehoben. (k=2, m=20, q=2, adaptiert von [10])

Filterung Vor dem Abgleich der maskierten Daten kann zusätzlich ein Schritt zum Minimieren der durchzuführenden Vergleichsoperationen durchgeführt werden. Das Ziel des Filterns ist es, maskierte Daten vom Matching auszuschließen, die definitiv keine signifikanten Ähnlichkeiten zueinander aufweisen.

Bei der Verwendung von Bloomfiltern eignet sich beispielsweise ein einfacher Längenfilter, welcher überprüft, ob sich die zu vergleichenden Bloomfilter signifikant in ihrer Länge voneinander unterscheiden [28]. Wenn beispielsweise ein Schwellenwert von 95 % für das Matching angesetzt ist, dann können zwei Bloomfilter nicht dem gleichen Datensatz entstammen, wenn sich die Anzahl der gesetzten Bits um mindestens 5 % unterscheidet.

Weitere Ansätze erweitern den Längenfilter um mehrere Filter für eine weitaus effizientere Reduktion der Laufzeit des anschließenden Matchings [14]. Eine weitere Methode von Bachteler et al. [16] verwendet Multibit-Trees — eine baumförmige Datenstruktur, welche das schnelle Abfragen von Bitvektoren mit einer vordefinierten minimalen Ähnlichkeit zueinander ermöglicht.

Matching Das Matching bezeichnet das eigentliche Zusammenführen der maskierten Datensätze. Der Datenabgleich kann exakt oder schätzungsweise erfolgen, wobei in der Praxis genaue Übereinstimmungen aufgrund von typographischen Fehlern sehr selten sind. [32]

Für den Vergleich von Zeichenketten innerhalb von Datensätzen eignen sich Metriken wie die Edit-Distanz, welche im Absatz Maskierung bereits demonstriert wurde, oder aber auch das Jaro-Winkler-Maß. Letzteres rechnet zusätzlich ausgetauschte Zeichen in einer Zeichenkette ein und erlaubt deren Gewichtung. [33]

In Verbindung mit Bloomfiltern werden sowohl häufig die Overlap- und Dice-Koeffizienten als auch der Jaccard-Index verwendet [32]. Der Dice-Koeffizient wurde von Schnell et al. in ihrem PPRL-Ansatz vorgeschlagen, da dieser nicht anfällig gegenüber einer großen Anzahl an ungesetzten Bits ist [10].

Die Dice- $D(A,B)$ und Jaccard-Koeffizienten $J(A,B)$ werden wie folgt berechnet, wobei A und B für die Mengen der Indices der gesetzten Bits im jeweils ersten und zweiten Bloomfilter stehen.

$$
\begin{aligned}
D(A,B) &= \frac{2\,|A \cap B|}{|A| + |B|} \\
J(A,B) &= \frac{|A \cap B|}{|A| + |B| - |A \cap B|}
\end{aligned}
$$

Angewendet auf das Beispiel in Abbildung 2.3 ergibt sich für die Bloomfilter von „Erika“ und „Erica“ die folgende Schnittmenge.

$$
\begin{aligned}
A &= \{0,4,6,8,9,10,12,13,15,16,18\} \\
B &= \{0,4,6,8,10,11,12,14,15,17,18,19\} \\
A \cap B &= \{0,4,6,8,10,12,15,18\}
\end{aligned}
$$

Die Dice- und Jaccard-Koeffizienten berechnen sich wie folgt.

$$
\begin{aligned}
D(A,B) &= \frac{16}{11+12} = \frac{16}{23} = 0{,}69565\ldots \\
J(A,B) &= \frac{8}{11+12-8} = \frac{8}{15} = 0{,}53333\ldots
\end{aligned}
$$

Klassifizierung Nach dem Matching ist jedem maskierten Datensatzpaar eine Wahrscheinlichkeit der Übereinstimmung zugeordnet. Diese Wahrscheinlichkeit wird verwendet, um eine Klassifizierung in „Match“, „kein Match“ und gegebenenfalls „möglicher Match“ durchzuführen.

Eine einfache Methode ist die Verwendung eines konstanten Schwellenwertes t. Liegt die berechnete Wahrscheinlichkeit über t, so wird das dazugehörige Paar als „Match“ klassifiziert, andernfalls als „kein Match“. Durch das Hinzufügen eines weiteren Schwellenwertes unterhalb von t ist es möglich, eine weitere Unterteilung in „möglicher Match“ zu erzeugen. [32]

Der fixe Schwellenwert hat den Nachteil, dass die Güte der Klassifizierung stark zwischen Anwendungsfällen schwanken kann. Ein hoher Schwellenwert kann für große Datensätze mit vielen Ausprägungen sehr gute Ergebnisse liefern. Bestehen Datensätze allerdings aus wenigen Attributen, so kann bereits ein kleiner Fehler zwischen zwei sonst sehr ähnlichen Datensätzen die berechnete Ähnlichkeit schnell unter den Schwellenwert drücken.

Einen dynamischen Ansatz zur Schwellenwertbildung liefern Fellegi und Sunter, welcher a priori Schätzungen über Fehler zwischen zwei Datensätzen einfließen lässt [20]. Dieser Ansatz gestaltet sich im Kontext der PPRL allerdings um einiges schwieriger. Es ist aus Datenschutzbedenken schwer möglich, die Art und Häufigkeit der Fehler zwischen verschiedenen Datenbasen zu überprüfen, um eine Schätzung vor dem eigentlichen PPRL-Prozess durchzuführen.

2.5.2 Two-party und Multi-party-Protokolle

Ein weiterer entscheidender Aspekt für PPRL ist die Anzahl der Teilnehmer. Prinzipiell wird zwischen Protokollen mit zwei Teilnehmern und mehreren Teilnehmern unterschieden, auch „two party“ und „multi party“ genannt. In Two-party-Protokollen kommunizieren die beiden Teilnehmer direkt miteinander, also peer-to-peer. Bei Multi-party-Protokollen wird eine zentrale Verknüpfungseinheit benötigt, welche die maskierten Datensätze aller Teilnehmer annimmt, das Matching durchführt und die Ergebnisse allen Teilnehmern mitteilt. [32]

Die Notwendigkeit einer außenstehenden Partei, der alle Teilnehmer vertrauen müssen, macht das Konzept der Multi-party-Protokolle je nach Angriffsmuster anfälliger für Manipulation durch Dritte im Vergleich zu Two-party-Protokollen. Weiterhin können die Teilnehmer in Two-party-Protokollen das Matching und die anschließende Klassifikation lokal durchführen und müssen nicht auf die Ergebnisse einer externen Entität vertrauen. Die Anforderung an die kryptografisch sichere Kommunikation zwischen zwei Teilnehmern ist im Two-party-Ansatz allerdings um einiges höher. Zudem müssen erweiterte Sicherheitsvorkehrungen getroffen werden, um sicherzugehen, dass die Teilnehmer nichts über die Datensätze des jeweils anderen in Erfahrung bringen können. [32]

2.5.3 Sicherheit in PPRL-Protokollen

Die Art der Daten, mit der im Kontext von PPRL gearbeitet wird, fordert ein hohes Sicherheitsmaß an das Kommunikationsprotokoll zwischen allen Teilnehmern. Dafür wird eine Grunddefinition von Sicherheit in PPRL-Protokollen benötigt. Im besten Fall sind alle Teilnehmer integer und halten sich strikt an das Protokoll. Die folgenden Absätze bieten eine kurze Übersicht über die möglichen Angriffsformen auf PPRL-Protokolle und die Sicherheitsanforderungen, die sich daraus ableiten.

Angriffsmodelle Vatsalan et al. [28] beschreiben zwei grundlegende Klassen von Angriffsmodellen, die in Bezug auf PPRL möglich sind. Das „honest-but-curious"-Modell (HBC) geht von einem Angreifer aus, der sich zwar an das Protokoll hält, jedoch versucht, so viel wie möglich über die Daten der anderen Teilnehmer herauszufinden. Es ist wichtig zu bedenken, dass sich das HBC-Modell nicht nur auf einen Angreifer beschränkt. Absprachen zwischen mehreren Teilnehmern sind im HBC-Modell möglich und müssen in einem durchdachten PPRL-Protokoll unterbunden werden.

Das zweite Modell bezieht sich auf böswillige Angreifer. Ihre Kooperation am Protokoll ist zu keinem Zeitpunkt gegeben. Sie können andere Teilnehmer stören, Eingaben verfälschen, die Schritte im Protokoll in einer beliebigen Reihenfolge durchgehen und an verschiedenen Stellen aus- und wieder einsteigen.

Zum Zeitpunkt der Arbeit richten sich die meisten PPRL-Protokolle nach dem HBC-Modell, jedoch ohne die Möglichkeit der Absprache zwischen Teilnehmern [1, 32]. Das Modell, welches sich an böswilligen Teilnehmern ausrichtet, findet wenig Beachtung. Ein markanter Grund hierfür ist, dass der rechnerische Aufwand, um solche Angriffe zu vermeiden, schlichtweg zu hoch ist. Weitere Forschung in den Sicherheitsaspekten der PPRL wird benötigt, um aggressivere Angriffsszenarien zu behandeln.

Sicherheitsanforderungen Die folgenden Anforderungen basieren auf einer Zusammenfassung von Lindell et al. [1], welche eine Liste von Grundprinzipien definieren, um eine Sicherheitsdefinition im Kontext von PPRL einzugrenzen.

Jeder Teilnehmer darf nicht mehr Information in Erfahrung bringen, als ihm zusteht. Bezogen auf PPRL bedeutet dies konkret, dass ein Teilnehmer die Auskunft, ob für seine eigens eingebrachten Datensätze Übereinstimmungen bei anderen Teilnehmern gefunden wurden, erhalten darf. Es darf keine Information über die Eingaben der anderen Teilnehmer veröffentlicht oder auf diese Rückschlüsse gefasst werden.

Kein Teilnehmer darf die Eingaben der anderen Teilnehmer erfahren. Es darf einem Angreifer also nicht möglich sein, die eigenen Eingaben an die der anderen Teilnehmer

anzupassen. Ein Beispiel in PPRL ist die bewusste Herbeiführung von falsch positiven Übereinstimmungen, indem ein Angreifer die gleichen oder sehr ähnliche Eingaben wie andere Teilnehmer tätigt. Somit werden auch die Ergebnisse für ehrliche Teilnehmer verfälscht.

Zuletzt müssen jedem Teilnehmer, ob ehrlich oder nicht, Ergebnisse garantiert werden. Die Aussage dieser Anforderung ist zweierlei. Böswillige Angreifer dürfen ehrliche Teilnehmer nicht daran hindern, ihre Ergebnisse zu erhalten. Das entspricht dem IT-Grundschutzziel der Verfügbarkeit und kann als die Vermeidung eines „Denial of Service“-Angriffs gewertet werden. Weiterhin müssen die Ergebnisse für jeden Teilnehmer korrekt sein. Das bedeutet, dass auch ein böswilliger Angreifer, welcher beispielsweise eine Menge an zufälligen Bitvektoren generiert, auch Ergebnisse für diese erhält, wenn Übereinstimmungen identifiziert wurden.

3 Personal Health Train

Der Personal Health Train (PHT) ist ein Ansatz zur verteilten Datenanalyse. Statt Daten an einer zentralen Stelle für eine Datenanalyse zu sammeln, besteht das Ziel des PHTs in der Ausführung der Datenanalyse bei den Eigentümern der Daten. Der PHT wird im Rahmen dieser Arbeit verwendet, um Daten bei ihren Inhabern zu verarbeiten. In diesem Kapitel wird auf Herausforderungen eingegangen, welche der PHT mit seinem Ansatz versucht zu lösen. Weiterhin werden bereits durchgeführte Studien mithilfe des PHT vorgestellt und einige Implementierungen mit deren Funktionsweisen kurz erklärt.

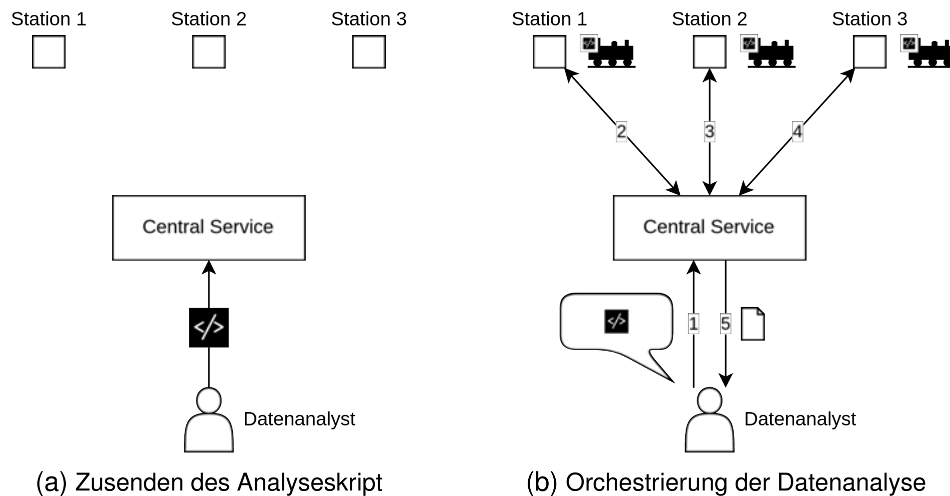
3.1 Konzeptioneller Überblick

Datenanalysen über mehrere Standorte hinweg durchzuführen, birgt eine Vielzahl von Problemen. Die eingesetzten Technologien zur Verwaltung von und zum Zugriff auf Daten sind sehr verschieden. Es ist unwahrscheinlich, dass sich zum Zweck der Datenanalyse viele Datentreuhänder auf eine universelle Lösung einigen. Und selbst wenn dem so wäre, dann würden sie für eine Datenanalyse die Kontrolle über den Zugriff auf ihre Daten behalten wollen [34]. Weiterhin zieht das Sammeln von studienrelevanten Daten an einem Ort einen immensen bürokratischen Aufwand mit sich und stellt hohe Anforderungen an die Sicherheit der Daten.

Diesen Problemen stellt sich der PHT. Der PHT ist ein Ansatz, welcher studienrelevante Daten bei ihren Eigentümern belässt, statt diese an einem zentralen Ort zum Zweck der Datenanalyse zu übertragen. Die Analysewerkzeuge werden also zu den Daten befördert und nicht umgekehrt. So löst der PHT die Problematik, dass sensitive Daten selten die Grenzen der Institution verlassen, die diese verwalten. Die Verwalter der Datenquellen müssen lediglich den Zugriff auf ihre Datenbestände gewährleisten und sind anderweitig an keine weiteren Softwarelösungen gebunden. [34]

Das Schema des PHT ist in Abbildung 3.1 dargestellt. Der PHT gleicht der Analogie eines Zuges, welcher von Station zu Station fährt. Dabei repräsentiert der Zug die Analyseaufgabe und die Stationen die Standorte, an denen sich die zu analysierenden Daten befinden.

Angenommen ein Wissenschaftler möchte eine Datenanalyse mit dem PHT an verschiedenen Standorten ausführen. Die Schnittstelle zwischen dem Wissenschaftler und dem PHT ist ein zentraler Dienst, der sogenannte „Central Service“ oder die „Central Station“, welcher die Ausführung von Analyseaufgaben orchestriert, die Kommunikation zwischen den Stationen ermöglicht und Ergebnisse sammelt.



(a) Zusenden des Analyseskript (b) Orchestrierung der Datenanalyse

Abbildung 3.1: Schematischer Aufbau des PHT am Beispiel von drei Stationen. Der Datenanalyst interagiert mit dem Central Service, um ein Analyseskript vorzubereiten und um dieses an die gewünschten Stationen zu senden.

Zuerst muss der Wissenschaftler definieren, welche Analyse ausgeführt werden soll. Dafür sendet er ein ausführbares Skript an den Central Service. Anschließend kann sich der Wissenschaftler auf dieses Skript beziehen und dem Central Service mitteilen, dass es an bestimmten Standorten ausgeführt werden soll. Dieser versendet das Skript in einem „Zug“ an die „Stationen“ an den angegebenen Standorten. Dies kann inkrementell oder parallel erfolgen. Der inkrementelle Fall ist in Abbildung 3.1b dargestellt, wo der Zug alle Stationen nacheinander besucht. An den Stationen wird das Skript ausgeführt und Artefakte, welche während der Ausführung entstehen, zurück an den Central Service geschickt. Dieser sammelt die Ergebnisse von allen Stationen und übermittelt diese nach der Beendigung aller Züge an den Wissenschaftler.

Der PHT ist nicht nur auf Datenanalysen beschränkt. In jedem Anwendungsfall, wo ein bestimmter Algorithmus oder eine Aufgabe an mehreren Standorten durchgeführt werden muss, kann der PHT eingesetzt werden. Vor allem im Bereich der Medizininformatik ist das Konzept des PHT lukrativ, da in den meisten Fällen auf sensitiven personenbezogenen Daten gearbeitet wird und diese selten ihre zugewiesenen Domänen verlassen dürfen. Auch maschinelles Lernen lässt sich auf den PHT-Ansatz anwenden, indem ein Modell an jeder Station an den eigenen Daten trainiert, validiert oder ausgeführt wird.

3.2 Anwendungen des PHT

Wie der Name suggeriert, ist der PHT-Ansatz vor allem für Datenanalysen in der medizinischen Domäne vorgesehen. Es wurden bereits eine Reihe von Studien mithilfe des PHT durchgeführt, welche die Notwendigkeit und Relevanz dieses Konzeptes unterstreichen. Einige dieser Studien werden in den folgenden Absätzen kurz vorgestellt.

Shi et al. [35] verwendeten den PHT-Ansatz, um ein Modell für das maschinelle Lernen auf einen bestimmten Biomarker aus Lungenkrebsbildern zu trainieren. Zwei öffentliche Datensätze wurden hierfür auf zwei Stationen aufgeteilt. An der ersten Station wurde das Modell trainiert, an der zweiten Station validiert. Die Ergebnisse zeigen keine signifikanten Abweichungen von einem Lernansatz, in dem alle Daten vorher an einer zentralen Stelle aggregiert werden.

In einem wesentlich größeren Maßstab zeigten Deist et al. [36] die Funktionsweise des PHT an über 20 000 Lungenkrebs-Datensätzen, welche über acht verschiedene Institute ausgewertet wurden. Auf Grundlage der Daten an jeder Station wurden statistische Werte berechnet, die anschließend für das Training einer logistischen Regression verwendet wurden.

Mou et al. [37] verwendeten Bilder aus den Hautläsion-Datensätzen der ISIC 2019 Challenge, um diese einer Melanom-Klassifikation zu unterziehen. Hierfür wurde ein Modell zum maschinellen Lernen inkrementell an drei verschiedenen Stationen trainiert. Auch hier konnten vergleichbare Ergebnisse zu einem zentralisierten Ansatz gemessen werden. Alle genannten Studien demonstrieren, dass der PHT einen äußerst nützlichen Ansatz für die verteilte Datenanalyse bietet, welcher den Schutz der Patientendaten an den einzelnen Stationen effektiv wahrt.

3.3 Funktionsweise von PHT-Implementierungen

Beim PHT handelt es sich lediglich um einen Ansatz. Die Implementierung obliegt Entwicklerteams. Da der PHT ein relativ neues Konzept ist, existieren zum Zeitpunkt der Arbeit noch keine Lösungen, welche den PHT-Ansatz in breiter Masse anwendbar realisieren. Es sind allerdings schon einige Pilotprojekte in der Entwicklung, welche sich langsam etablieren und in den folgenden Abschnitten erwähnt werden.

Eine Implementierung für die verteilte Datenanalyse, welche noch vor der Konzeption des PHT geschaffen wurde, ist *DataSHIELD*[6] [38]. Wie der PHT beabsichtigt *DataSHIELD* die Analysewerkzeuge zu den Daten zu bringen. Im Gegensatz zum PHT sind Teilnehmer in *DataSHIELD* an eine *Opal*[7]-Datenbank gebunden. Analyseskripte müssen in der statistischen Programmiersprache *R* formuliert und von einem *R*-Server ausgeführt werden. Weiterhin müssen die Daten an jeder Station zuerst in das *Opal*-System eingepflegt werden, was einen zusätzlichen Administrationsaufwand darstellt.

Zwei Implementierungen, welche sich direkt an dem Konzept des PHTs orientieren, sind *PADME*[8] der RWTH Aachen [39] und *VANTAGE6*[9] des Integraal Kankercentrum

[6] Siehe `https://datashield.org/`
[7] Siehe `https://www.obiba.org/pages/products/opal/`
[8] Siehe `https://padme-analytics.de/`
[9] Siehe `https://vantage6.ai/`

Nederland [40]. Beide stützen ihre Infrastruktur auf die Containervirtualisierungslösung Docker[10].

In der Docker-Terminologie wird ein ausführbares Skript oder Programm als Container bezeichnet. Ein Container enthält alle notwendigen Dateien, Abhängigkeiten und Komponenten, die für die Ausführung eines Programms notwendig sind. Container repräsentieren isolierte Ausführungsumgebungen, welche getrennt vom System arbeiten, auf dem Docker installiert ist.

Container werden aus Docker Images erzeugt. Ein Docker Image enthält die Vorlage für die Erstellung eines neuen Containers. Um ein solches Docker Image zu erstellen, wird ein *Dockerfile* benötigt. Dieses wird vom Wissenschaftler entworfen und enthält Instruktionen zur Installation notwendiger Softwarekomponenten, um das eigene Analyseskript oder -programm auszuführen. Insofern steht dem Wissenschaftler die Wahl der Werkzeuge für die Datenanalyse komplett frei.

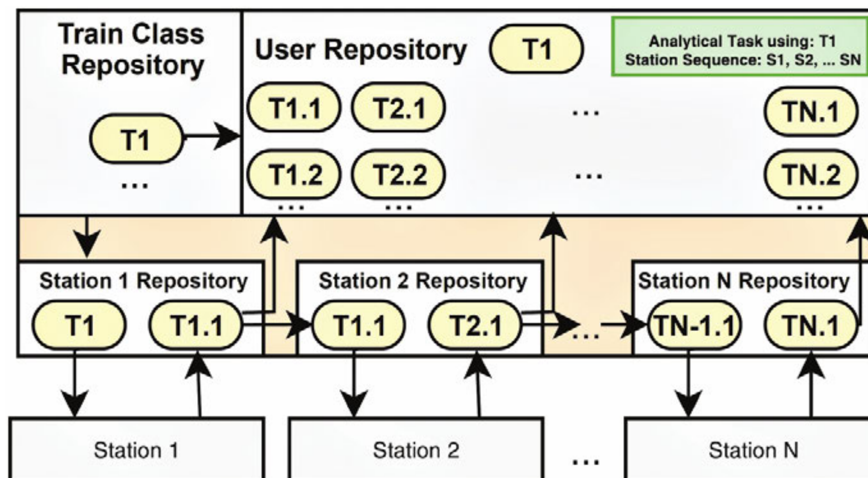


Abbildung 3.2: Kommunikation zwischen PHT-Stationen und zentraler Zugklassen-Repository in *PADME*. Ein Zugimage wird von Station zu Station gesendet und die Ergebnisse der Ausführung gespeichert. (übernommen von [39])

Der Central Service verwaltet Docker Images. Nach der Definition einer Analyseaufgabe als *Dockerfile* und der Übermittlung an den Central Service kann der Wissenschaftler das erzeugte Zugimage und eine Reihe von Stationen auswählen. Dies ist in Abbildung 3.2 am Beispiel der Implementierung von *PADME* dargestellt. Die erste Station wird über ein neues Zugimage informiert und lädt dieses herunter. Aus dem Zugimage wird ein neuer Container erstellt, ausgeführt, und das Ergebnis als neues Docker Image an den Central Service zurückgesendet. Die Ergebnisse werden vom Central Service extrahiert und das aktualisierte Zugimage wird an die nächste Station gesendet. Dieser Prozess wird für alle Stationen durchgeführt.

[10] Siehe `https://www.docker.com/`

Im Gegensatz zu *PADME* ist *VANTAGE6* speziell auf „federated learning“, also dem parallelen Ausführen von Datenanalysen, ausgelegt. Nur die Ergebnisse einer Ausführung an einer Station werden zurück an den Central Service gesendet. Die Ergebnisse können jedoch, auf Anforderung des Wissenschaftlers, erneut aggregiert und zurück an die Stationen geschickt werden, beispielsweise für eine weitere Lernphase im Kontext des maschinellen Lernens. *PADME* erlaubt zum Zeitpunkt der Arbeit das sequentielle Ausführen von Analyseaufgaben, jedoch wird aktiv an einer Implementierung des „federated learning“ gearbeitet.

4 Konzeption

Im Rahmen dieser Arbeit wird ein probabilistisches PPRL-Protokoll auf Basis von Bloomfiltern beschrieben und die notwendigen Softwarekomponenten implementiert. Die PHT-Implementierung *PADME* wird als unterstützende Ausführungsumgebung verwendet. Es werden einige Komponenten beschrieben, welche für den speziellen Einsatz in der PHT-Infrastruktur vorgesehen sind. Dennoch sind die PPRL-Kernkomponenten so konzipiert, sodass sie auch außerhalb des PHT in jeder generischen Umgebung mit Internetzugang verwendet werden können.

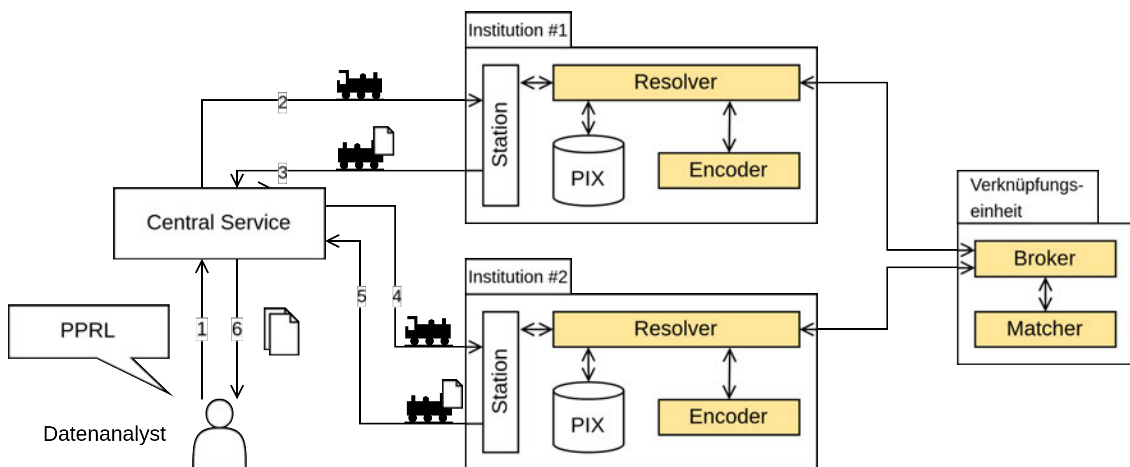


Abbildung 4.1: Schematische Integration der PPRL-Infrastruktur in den PHT. Implementierte Komponenten sind farbig hinterlegt.

Der schematische Aufbau der PPRL-Architektur im PHT ist in Abbildung 4.1 dargestellt. An jeder Station befindet sich ein PIX-System, welches für die Referenzierung von Patienteninformationen verwendet wird. In der geplanten Architektur dient das PIX-System dem Identitäts- und Pseudonymmanagement. Weiterhin befindet sich an jeder Station ein Encoder-Service, welcher Datensätze entgegennimmt und maskiert. Datensätze werden über Pseudonyme adressiert, welche auf der Seite der Station von einem Resolver-Service entgegengenommen, aufgelöst, maskiert und abgeschickt werden. Den Vergleich von maskierten Datensätzen über Institutionsgrenzen hinweg ermöglicht der Broker-Service in Kombination mit dem Matcher-Service. Letzterer ist für die Berechnung der Ähnlichkeit von Datensätzen zueinander zuständig. Zuletzt dient eine vordefinierte Zugklasse dem einfachen Ausführen von PPRL in der PHT-Infrastruktur. In den folgenden Absätzen wird der erwartete Funktionsumfang der einzelnen Komponenten genauer beschrieben.

Kodierung und Maskierung Für die Schritte der Kodierung und Maskierung im PPRL-Prozess ist der Encoder zuständig. Der Encoder ist, wie alle weiteren Komponenten,

welche in diesem Kapitel beschrieben werden, ein einfacher Webservice. Jedem generischen HTTP-Client soll es möglich sein, mit dem Encoder und allen weiteren Webservices zu kommunizieren. Deswegen bieten sie ihre Funktionen als REST-Schnittstellen an.

Die Ausprägungen der zu verknüpfenden Daten zwischen Stationen können sehr verschieden ausfallen. Beispiele sind unterschiedliche Konventionen bei der Groß- und Kleinschreibung oder bei der Präzision von Fließkommazahlen. Hierfür sollten Schemata für ausgewählte Attribute definierbar sein, welche aussagen, wie ein Attribut zu interpretieren ist. Dem Encoder muss es möglich sein, aus einem solchen Schema eine Reihe von Vorverarbeitungsschritten abzuleiten und anzuwenden.

Anschließend wird für jeden Datensatz ein Bloomfilter angelegt. Damit Bloomfilter über jede Station hinweg auf die gleiche Art und Weise erzeugt und besetzt werden, muss zusätzlich eine Konfiguration für den Encoder angegeben werden. Auch diese Konfiguration muss über alle Stationen, die ihre Daten miteinander abgleichen wollen, konsistent sein. Die Bloomfilter werden daraufhin in eine textuelle Repräsentation umgewandelt. Diese Darstellung wird fortan als Bitvektor bezeichnet.

Matching Das Matching von Bitvektorpaaren untereinander wird vom einem weiteren Webservice übernommen: dem Matcher. Ihre Ähnlichkeit zueinander wird berechnet und resultiert in einer Wahrscheinlichkeit. Diese sagt aus, wie wahrscheinlich es ist, dass die Datensätze, aus denen die beiden Bitvektoren erzeugt wurden, auf die gleiche Person in der echten Welt verweisen.

Für die Berechnung der Ähnlichkeit können verschiedene Metriken verwendet werden. Zudem soll es möglich sein, alle Bitvektorpaare mit ihren berechneten Ähnlichkeiten ausgeben zu lassen oder nur diejenigen, die über einem festen Schwellenwert liegen. Auch für den Matcher wird somit neben den zu vergleichenden Bitvektoren eine Match-Konfiguration benötigt, welche die Parameter für das Matching festlegt.

Verteilte Datenverknüpfung Jede Station verfügt über einen Encoder-Service. Um jedoch das Matching von erzeugten Bitvektoren über eine beliebige Anzahl von Standorten hinweg zu ermöglichen, müssen diese zuerst an einem zentralen Ort gesammelt werden. Die Verwaltung von Bitvektoren und das Ausführen des Matchings mithilfe des Matchers übernimmt der Broker-Service. Er ist die zentrale Ansprechstelle für Stationen, die ihre Bitvektoren mit denen anderer Teilnehmer abgleichen wollen.

Gruppen von Teilnehmern werden in Match Sessions organisiert. Auf einer abstrakten Ebene sind solche Sessions lediglich Sammlungen von Bitvektoren, die miteinander verglichen werden müssen. Es soll jedem möglich sein, eine neue Match Session zu

starten. Teilnehmer einer Session können ihre generierten Bitvektoren an den Broker unter der Kennung der Session zusenden. Erhält der Broker von mindestens zwei Teilnehmern Bitvektoren, so beginnt das Matching.

Schritt für Schritt werden die Bitvektorpaare an den Matcher gesendet und die berechneten Ähnlichkeiten für gefundene Matches zwischengespeichert. Aufgrund der Matching-Laufzeit für eine große Menge an Bitvektoren soll es jedem Teilnehmer einer Session möglich sein, den aktuellen Fortschritt des Matchings abzufragen. Ist das Matching abgeschlossen, so sollen die Teilnehmer die Ergebnisse des Matchings für die von ihnen zugesendeten Bitvektoren abrufen können. Nach dem Ende der Session sollen alle Zwischenergebnisse für die Session aus dem Zwischenspeicher entfernt werden.

Der Lebenszyklus einer Match Session teilt sich somit in folgende Schritte auf:

1. Erstellen der Session
2. Empfangen von Bitvektoren
3. Erstellen von Bitvektorpaaren und Senden an den Matcher
4. Zwischenspeichern der Ergebnisse des Matchers
5. Match-Ergebnisse auf Anfrage an Teilnehmer senden
6. Schließen der Session und Löschen des Zwischenspeichers

Integration in den PHT Voraussetzung für eine erfolgreiche Integration der PPRL-Infrastruktur in den PHT ist eine PPRL-Zugklasse. Diese muss vom Wissenschaftler, welcher eine Record Linkage durchführen möchte, konfigurierbar sein. Hierzu zählt in erster Linie die Kennung der Session, an welche die generierten Bitvektoren an den einzelnen Stationen zugesendet werden sollen. Weiterhin müssen eine Encoder-Konfiguration und Attributschemata Teil der Zugklasse sein, sodass die Daten an jeder Station auf die gleiche Art und Weise verarbeitet und kodiert werden.

Ein PPRL-Durchlauf erfolgt, wie in Abbildung 4.2 zu sehen ist, in zwei Phasen. In der ersten Phase werden Bitvektoren an jeder Station generiert und an den Broker gesendet. Nach dem Abschluss des Matchings werden in der zweiten Phase an jeder Station die verfügbaren Ergebnisse vom Broker abgerufen. Auf die einzelnen Phasen werden in den folgenden Absätzen genauer eingegangen.

Vor dem Start der ersten Phase muss eine neue Session am Broker registriert werden. Das geschieht entweder manuell oder automatisiert während der Erstellung des Zuges. Mit der erhaltenen Session-Kennung wird der PPRL-Zug für die erste Phase ausgestattet. Der Wissenschaftler fügt dem Zug die Konfiguration für den Encoder-Service

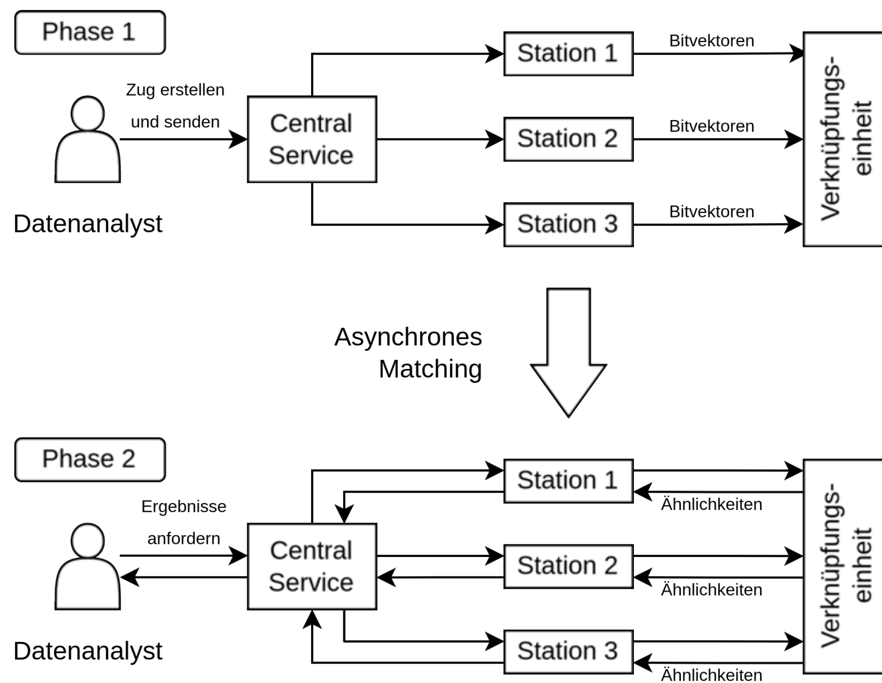


Abbildung 4.2: Ausführungsphasen des PPRL-Zuges. In der ersten Phase werden generierte Bitvektoren von den Stationen an die Verknüpfungseinheit, bestehend aus Broker- und Match-Service, gesendet. In der zweiten Phase werden die Ergebnisse an den einzelnen Stationen von der Verknüpfungseinheit abgerufen.

an den einzelnen Stationen bei. Erreicht der Zug eine Station, so müssen zuerst die dort verfügbaren Pseudonyme abgefragt werden. Diese müssen aus einem an der Station bereitgestellten Datenbestand extrahiert werden. Die Pseudonyme werden dem Resolver-Service an der Station übergeben. Der Resolver löst die Pseudonyme mithilfe des PIX-Systems vor Ort auf, sendet die ermittelten Datensätze an den Encoder und schickt die generierten Bitvektoren zusammen mit der Session-Kennung an den Broker. Gleichzeitig konstruiert der Resolver eine Zuordnung von Bitvektoren auf Pseudonyme, welche für die zweite Phase notwendig ist und gespeichert wird. Für jede teilnehmende Station wird dieser Prozess wiederholt.

An den Stationen kann der Matching-Fortschritt sukzessiv verfolgt werden. Ist die erste Phase des Zuges beendet und ist das Matching abgeschlossen, so kann die zweite Ausführungsphase beginnen. Erreicht nun der Zug eine Station, so wird erneut der Resolver-Service kontaktiert. In der zweiten Phase werden lediglich Ergebnisse für die Station vom Resolver abgefragt. Dieser stellt wiederum eine Anfrage an den Broker, welcher die gematchten Bitvektoren für die Station bereitstellt. Die Bitvektoren werden mithilfe der in der vorherigen Phase erstellten Zuordnung in Pseudonyme umgewandelt, welche dann als Antwort an den Zug übergeben werden. Vor Ort können anschließend die Ergebnisse ausgewertet werden.

Zusammenfassung Für die grundlegende PPRL-Infrastruktur müssen folgende Komponenten implementiert werden:

- Encoder: Konvertierung von Datensätzen in Bitvektoren
- Matcher: Berechnung der Ähnlichkeit von Bitvektorpaaren zueinander
- Broker: Verwalten von Match Sessions, Aggregation und Sammeln von Match-Ergebnissen
- Resolver: Auflösen von studienspezifischen Pseudonymen und Interaktion mit dem PIX-System und Encoder vor Ort, Kommunikation mit dem Broker
- PPRL-Zug Skript: Senden von Pseudonymen an den Resolver und Abfragen von Match-Ergebnissen an jeder Station

5 Encoder und Matcher

In diesem Kapitel wird die Implementierung von zwei Webservices vorgestellt. Zuerst wird der Encoder für die Maskierung von Datensätzen präsentiert. Anschließend wird der Matcher beschrieben, welcher den Vergleich von maskierten Datensätzen zueinander vornimmt.

5.1 Encoder

Der Encoder vereint die Schritte der Vorverarbeitung und der Maskierung im PPRL-Prozess in einem Service. Wie der Matcher und der Broker ist der Encoder als JAX-RS-Anwendung implementiert. In den folgenden Abschnitten wird der erwartete Funktionsumfang des Encoders vorgestellt, sein Datenmodell und die wichtigsten Aspekte dessen Implementierung beschrieben.

5.1.1 Konzeptioneller Überblick

Die Aufgabe des Encoders besteht in der Verarbeitung und Maskierung eingehender Datensätze. Für jedes Attribut eines Datensatzes muss eine Reihe von Validierungs- und Normalisierungsoperationen durchgeführt werden. Jedes Attribut wird deswegen mit einem Schema versehen, welches den Datentyp und weitere Optionen für die Verarbeitung des Attributs festlegt.

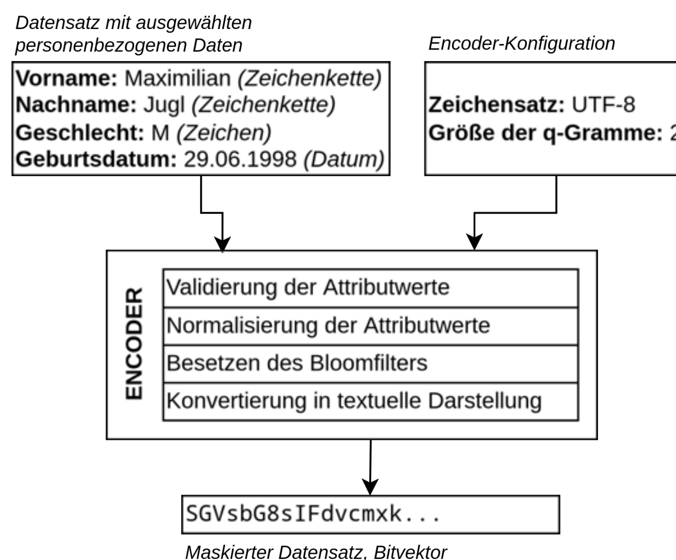


Abbildung 5.1: Schematische Arbeitsweise des PPRL-Encoders. Ein Datensatz mit einem Attributschema wird vom Encoder in einer Reihe von Vorverarbeitungs- und Maskierungsschritten in einen Bitvektor umgewandelt.

Die Arbeitsweise des Encoders ist in Abbildung 5.1 mit einem Beispieldatensatz dargestellt. Das Attribut „Geschlecht“ soll als einfaches Beispiel für die Vorverarbeitungsschritte des Encoders dienen. Bei der Validierung kann das Geschlecht auf eine Reihe von zulässigen Ausprägungen überprüft werden. Als Option könnte eine Liste von gültigen Zeichen, wobei jedes Zeichen eine Geschlechteridentität, an den Encoder übergeben werden. Entspricht das angegebene Geschlecht in einem Datensatz nicht dieser Anforderung, so wird die Vorverarbeitung abgebrochen. Bei der anschließenden Normalisierung und Standardisierung kann beispielsweise das Geschlecht in dessen Kleinschreibung umgewandelt werden, um eine konsistente Darstellung des Geschlechts über alle Datensätze hinweg zu erzeugen.

Nach der Datenvorverarbeitung wird für jeden Datensatz ein Bloomfilter angelegt. Um die Bloomfilter zu parametrisieren, muss zusätzlich eine Konfiguration an den Encoder übergeben werden. Alle Attribute des Datensatzes werden in q-Gramme aufgeteilt und in den Bloomfilter eingefügt. Das zugrundeliegende Bit-Array des Bloomfilters wird in eine kompakte und leicht übertragbare textuelle Bitvektor-Repräsentation konvertiert. Die Kombination aus Attributschemata und Encoder-Konfiguration ist Voraussetzung dafür, dass jeder Datensatz an jeder Station in einem PPRL-Durchlauf mit den gleichen Einstellungen erzeugt wird.

5.1.2 Datenmodell

Das Klassendiagramm für das Modell des PPRL-Encoders ist in Abbildung 5.2 dargestellt. `DataEntity` repräsentiert einen Datensatz aus einer Datenquelle, welcher sich auf eine Entität bezieht. Eine Entität besitzt einen Identifikator und eine Liste von Attributnamen und -werten. Letztere werden als Instanzen von `DataEntityElement` dargestellt. Diese besitzen genau einen Namen und einen Wert. Bei beiden Feldern handelt es sich um Zeichenketten.

Ein `EncoderRequest` setzt sich aus einer Menge von Entitäten, einer Konfiguration für die Generierung von Bitvektoren und einer Liste von Attributschemata zusammen. Die Konfiguration umfasst eine Reihe von Parametern für die Erstellung von Bloomfiltern. Dazu zählt der Zeichensatz, in dem die Attributwerte einer Entität kodiert sind, und ein Seed-Wert, der in jeden Bloomfilter eingefügt werden soll. Weiterhin ist die Generierungsfunktion angegeben, welche die gewünschte Erzeugung der q-Gramme beschreibt.

Zudem besitzt ein `EncoderRequest` Schemata für Attribute. `AttributeSchema` repräsentiert ein solches Schema. Ein Schema kann stets nur auf ein Attribut, identifiziert durch einen Attributnamen, angewendet werden. Zusätzlich zum Attributnamen führt die Klasse den Datentyp und eine Zuordnung von weiteren Optionen, welche zu berücksichtigen sind. `DataType` ist eine Auflistung der akzeptierten Datentypen.

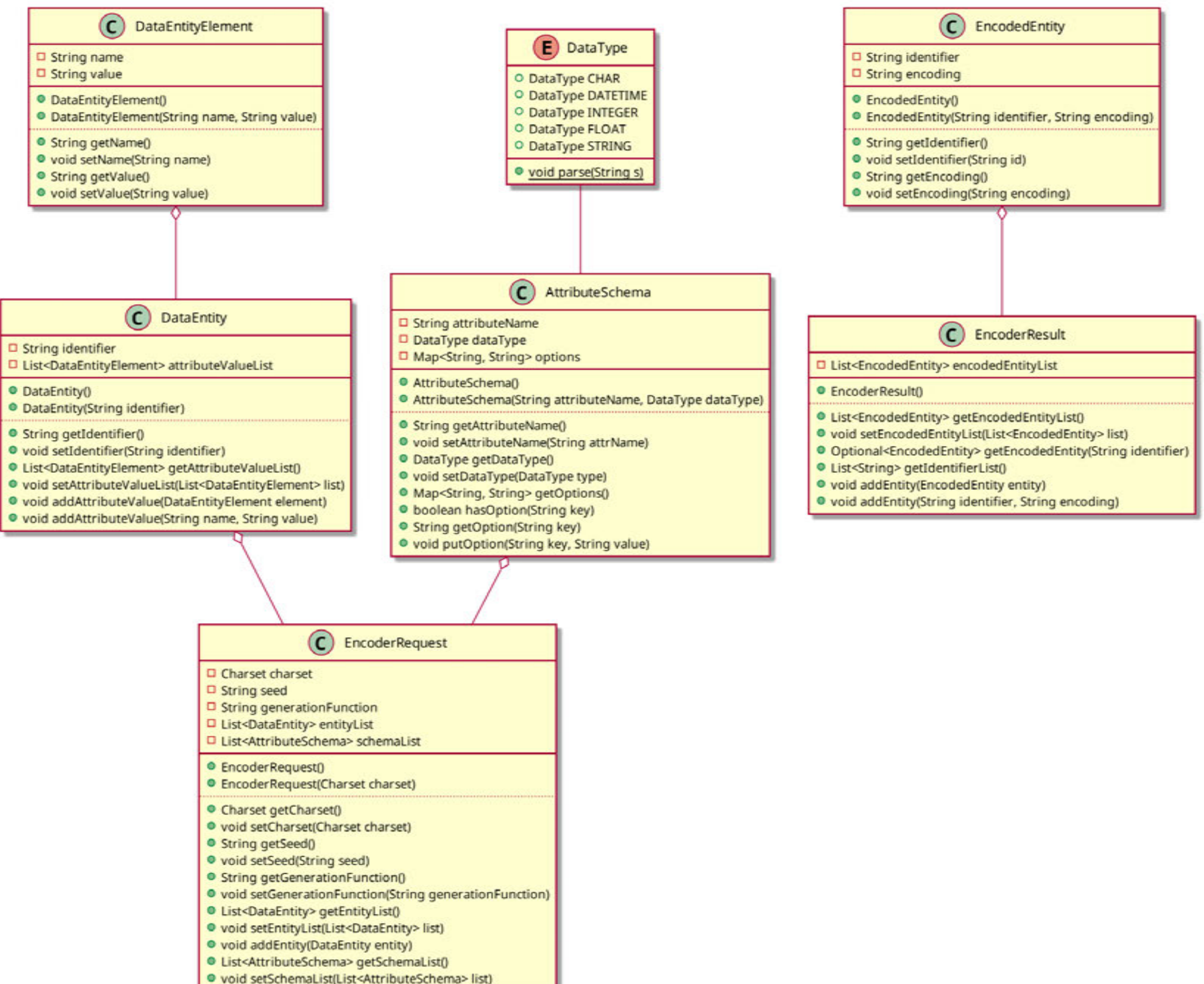


Abbildung 5.2: UML-Diagramm der Modellklassen in Verwendung des PPRL-Encoders

Die Antwort des Encoders — `EncoderResult` — ist eine Liste von maskierten Entitäten, repräsentiert durch `EncodedEntity`. Diese bestehen aus zwei Feldern: einem Identifikator und einer textuellen Repräsentation eines erzeugten Bloomfilters. Die Identifikatoren aus der Antwort des PPRL-Encoders korrespondieren mit den Identifikatoren, die in der Anfrage an den Encoder übergeben worden. Wenn also eine Anfrage eine `DataEntity` mit dem Identifikator „0001“ besitzt, so enthält die zugehörige `EncodedEntity` mit dem Identifikator „0001“ in der Antwort des Encoders die Bitvektorrepräsentation dieser Entität. Somit kann jede maskierte Entität in der Antwort des Encoders mit den Entitäten aus der eingehenden Anfrage verknüpft werden.

Die Listings in Anhang A zeigen eine Beispielanfrage und -antwort, welche den erwarteten Arbeitsablauf des PPRL-Encoders verdeutlichen. In der Anfrage ist eine Entität mit dem Identifikator „0001“ enthalten. Diese Entität besitzt zwei Attribute: Vorname und Körpergröße. Die Werte für die beiden Attribute werden als Strings übergeben. Zusätzlich sind zwei Attributschemata hinterlegt. Diese sagen aus, dass der Vorname als String und die Körpergröße als Gleitkommazahl behandelt werden sollen. Weiterhin soll die Körpergröße auf eine Nachkommastelle gerundet werden.

Die Antwort auf diese Anfrage besitzt eine maskierte Entität, dessen Identifikator mit dem aus der Anfrage korrespondiert. Das `encoding`-Feld enthält die textuelle Repräsentation des Bloomfilters, welcher entsprechend der Konfiguration in der Anfrage aus den Attributen der dazugehörigen Entität erzeugt wurde.

5.1.3 Endpunkte

Der Encoder beschriebt zwei Endpunkte, welche auch vom Matcher und vom Broker übernommen werden und an dieser Stelle nur einmal erwähnt werden. Der Wurzelpfad / liefert eine Liste aller vom Webservice zur Verfügung gestellten Endpunkte. Zusätzlich existiert der Endpunkt `/health`, welcher lediglich den Statuscode *200 OK* sendet. Dieser dient dem Monitoring des Services.

Speziell für den Encoder existiert der Endpunkt `/generator-functions`. Dieser sendet eine Liste von allen möglichen Werten zurück, die in das `generationFunction`-Feld der Encoder-Anfrage eingetragen werden können. Zum Zeitpunkt dieser Arbeit unterstützt der Encoder die Generierung von Bloomfiltern mit Bi- und Trigrammen, also q-Grammen der Länge zwei und drei.

Zuletzt stellt der Encoder den Endpunkt `/encode` bereit. Mit einer POST-Anfrage kann die Maskierung einer Liste von Entitäten angefordert werden. Das Format der Anfrage ergibt sich aus dem Modell, welches im vorhergehenden Abschnitt beschrieben ist.

5.1.4 Datenvorverarbeitung

Erhält der Encoder eine Anfrage, so beginnt er die Liste der maskierten Entitäten aufzubauen. Dafür werden alle Entitäten nacheinander zuerst vorverarbeitet und dann in jeweils einen Bloomfilter anhand der Konfiguration in der Anfrage eingefügt. Die Art und Weise der Vorverarbeitung ergibt sich aus der Liste der Attributschemata in der Anfrage, sofern vorhanden. Alle Schemata sind hierbei auf alle angegebenen Entitäten anzuwenden. Der allgemeine Ablauf ist in Abbildung 5.3 dargestellt und wird in den nächsten Absätzen näher beschrieben.

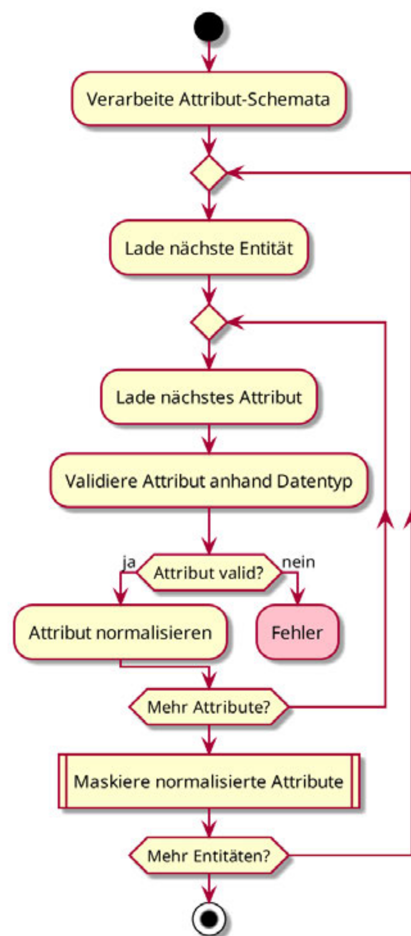


Abbildung 5.3: Prozess der Datenvorverarbeitung im PPRL-Encoder. Attributwerte einer Entität werden entsprechend der vorher geladenen Attributschemata validiert, normalisiert und anschließend maskiert.

Zuerst werden alle Attributschemata ausgewertet. Für jeden Datentyp existiert eine Klasse, welche die Klasse `AttributeProcessor` erbt und die Verarbeitungsschritte zur Validierung und Normalisierung eines Attributwerts durchführt. Der Matcher führt eine Zuordnung von Attributnamen auf Instanzen solcher Klassen, sodass für jedes Attribut einer Entität die geforderte Normalisierung durchgeführt wird. Die Datentypen und ihre Vorverarbeitungsschritte sind auf den nächsten Seiten aufgeführt.

Strings Für Strings wird der `StringAttributeProcessor` verwendet. Wenn für ein Attribut kein Schema hinterlegt ist, dann wird dieser Prozessor standardmäßig verwendet. Eine Validierung findet nicht statt, da alle Attributwerte sowieso als Strings gesendet werden. Die Klasse führt eine Reihe an gewöhnlichen String-Transformationen durch. Zuerst wird der String in dessen Kleinschreibung umgewandelt. Es werden Tabulatoren, Zeilenumbrüche und Punkte durch Leerzeichen ersetzt. Deutsche Umlaute werden ersetzt, beispielsweise „ä“ durch „ae“, „ö“ durch „oe“ und so weiter. Mehrere Leerzeichen hintereinander werden in ein einziges Leerzeichen zusammengefasst und Leerräume am Anfang und Ende des Strings werden entfernt.

Einzelne Zeichen Für einzelne Zeichen wird der `CharAttributeProcessor` verwendet. Dieser führt lediglich eine Validierung durch und gibt einen Fehler zurück, wenn der zu verarbeitende Attributwert nicht exakt ein Zeichen lang ist.

Ganzzahlen Der `IntegerAttributeProcessor` ist für die Verarbeitung von Ganzzahl-Attributen zuständig. Die Basis der übergebenen Zahl kann als Option angegeben werden. Ist diese nicht vorhanden, so wird der Attributwert als eine Zahl mit der Basis 10 angenommen. Bei der Validierung wird der String mit der ermittelten Basis in eine Ganzzahl umgewandelt. Gelingt dies nicht, so gibt der Prozessor einen Fehler zurück. Die konvertierte Ganzzahl wird in der Basis 10 anschließend wieder in einen String konvertiert. Das hat zur Folge, dass führende Nullen im Attributwert eliminiert werden.

Fließkommazahlen Die Verarbeitung von Fließkommazahl-Attributen übernimmt der `FloatAttributeProcessor`. Für Fließkommazahlen kann die Anzahl der Nachkommastellen über eine Option festgelegt werden. Ist diese nicht vorhanden, so wird diese Anzahl auf sechs beschränkt. Der Attributwert wird validiert, indem der Attributwert in eine Gleitkommazahl konvertiert wird. Wie bei der Validierung der Ganzzahlen wird bei Misslingen der Konvertierung ein Fehler zurückgegeben. Eine weitere Fehlerbedingung tritt ein, wenn die konvertierte Fließkommazahl, nach dem IEEE 754 Standard[11], unendlich ist.

Anschließend wird die Zahl auf die definierte Anzahl der Nachkommastellen gerundet. Die Rundungsregel entspricht dem kaufmännischen Runden nach der DIN-Norm DIN 1333[12]. Zuletzt wird die gerundete Zahl in einen String konvertiert. Wie bei der Verarbeitung von Ganzzahlen hat dies den Effekt, dass führende Nullen vom Attributwert entfernt werden.

[11] Siehe https://standards.ieee.org/ieee/754/6210/

[12] Siehe https://standards.globalspec.com/std/398343/DIN%201333

Datums- und Zeitangaben Der `DateTimeAttributeProcessor` nimmt die Verarbeitung von Datums- und Zeitangaben vor. Bei der Validierung wird der Attributwert auf das Format `dddd.MM.yyyy[ HH:mm[:ss]]` überprüft. Es müssen also folgende Datumsfelder in der gegebenen Reihenfolge vorhanden sein:

1. Tag (ein- bis zweistellig), Monat (ein- bis zweistellig) und Jahr (vierstellig), getrennt mit Punkten
2. Optional: Leerzeichen, gefolgt von Stunde und Minute (je ein- bis zweistellig), getrennt mit Doppelpunkt
3. Optional: Doppelpunkt, gefolgt von Sekunde (ein- bis zweistellig)

Fehlende Zeitangaben werden durch Nullen ersetzt. Gültige Werte sind beispielsweise „10.01.2022“, „10.01.2022 08:48“ und „10.01.2022 08:48:22“. Im Gegensatz stellen „10/01 08:48“, „08:48:22“ und „2022“ fehlerhafte Datumsangaben dar. Entspricht der Attributwert nicht dem beschrieben Format, so wird mit einem Fehler abgebrochen.

Das Format, in welches die Datums- und Zeitangabe umgewandelt werden soll, kann als Option festgelegt werden. Ist dieses Format nicht vorhanden, so werden standardmäßig Tag, Monat, Jahr, Stunde, Minute und Sekunde der Reihe nach, ohne Trennzeichen und Leerräume, zusammengeführt. Das entsprechende Format hierfür ist `ddMMyyyyHHmmss`.

Ein benutzerdefiniertes Datumsformat kann beispielsweise nützlich sein, um Zeitangaben bei der Generierung des Bitvektors zu ignorieren. Ist zum Beispiel in einem Datenschema das Geburtsdatum einer Person die Uhrzeit mitgeführt, dann kann dieses durch die Angabe eines Formats ohne Zeitangabe, wie `ddMMyyyy`, vernachlässigt werden. Die Java-Dokumentation der Klasse `DateTimeFormatter` führt hierfür erlaubte Formatangaben[13].

5.1.5 Maskierung

Nachdem alle Attribute einer Entität validiert und normalisiert wurden, werden diese maskiert. Für jede Entität wird ein neuer Bloomfilter erzeugt unter der Verwendung der Implementierung von *Google Guava*. Die Gründe hierfür liegen in der Robustheit der *Guava*-Bibliothek als Ganzes. Weiterhin legen die Entwickler viel Wert auf vollständige Tests und Performance ihres Codes.

Ein Bloomfilter benötigt zwei Parameter: die Anzahl der Hashfunktionen und die Anzahl der Bits. *Google Guava* lässt das manuelle Setzen dieser Parameter nicht zu. Stattdessen werden sie in der *Google Guava* Implementierung aus zwei anderen Eigenschaften

[13] Siehe https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html

eines Bloomfilters berechnet: der Anzahl der erwarteten Elemente und der gewünschten Falsch-positiv-Rate. Letztere sagt aus, wie hoch die Wahrscheinlichkeit eines falsch positiven Ergebnisses sein soll, wenn die Existenz eines Elementes im Bloomfilter überprüft werden soll.

Die q-Gramme der normalisierten Attributwerte entsprechen den einzufügenden Elementen. Standardmäßig ist die Falsch-positiv-Rate in der *Google Guava* Implementierung auf 3 % festgelegt. Weiterhin wird *MurmurHash3* als Hashfunktion in Verbindung mit der Double-Hashing-Verfahren von Kirsch und Mitzenmacher [25] verwendet, um eine beliebige Anzahl an Hashwerten zu berechnen.

Folglich muss vor der Erstellung des Bloomfilters die Anzahl der Elemente abgeschätzt werden. Sei l die Zeichenlänge eines Attributwertes und q die Größe der q-Gramme, in die der Wert zerlegt werden soll. Dann entspricht $l+q-1$ der Anzahl der q-Gramme für diese Zeichenkette. Diese Berechnung wird für alle normalisierten Attributwerte durchgeführt und die Ergebnisse aufsummiert. Das Endergebnis wird verwendet, um einen neuen Bloomfilter zu instanziieren.

Die Größe des Bloomfilters m in der *Google Guava* Implementierung berechnet sich in Abhängigkeit der einzufügenden Elemente n und der Falsch-positiv-Rate p wie folgt[14]:

$$m = \left\lfloor \frac{-n \ln p}{(\ln 2)^2} \right\rfloor$$

Das hat zur Folge, dass mit jedem neu eingefügten Element die Größe des Bloomfilters linear ansteigt. Angenommen es existieren zwei Datensätze, welche identisch zueinander sind bis auf den Vornamen, beispielsweise „Lea“ und „Lena“. Da der Datensatz von „Lena“ ein Zeichen mehr besitzt als der von „Lea“, und dementsprechend ein weiteres q-Gramm erzeugt wird, wird automatisch der Bloomfilter von „Lena“ vergrößert. Das kann bewirken, dass sich die Besetzung der Bits zwischen den generierten Bloomfiltern für die beiden Datensätze stark voneinander unterscheidet.

Um dem entgegenzuwirken, wird die berechnete Anzahl der q-Gramme auf das nächste Vielfache von 32 angehoben. Dadurch ergibt sich ein treppenförmiger Anstieg der Bloomfiltergröße, wie in Abbildung 5.4 zu sehen ist. Die Zahl ist so gewählt, dass mit jedem fehlenden oder überflüssigen Zeichen eine $\frac{1}{32} \approx 3{,}13\%$ Chance besteht, dass dieses die Größe des Bloomfilters drastisch ändert. Somit ergibt sich ein annehmbarer Kompromiss zwischen Fehleranfälligkeit und drastischen Größenunterschieden zwischen Bloomfiltern. Es schließt nicht vollständig aus, dass Fehler wie am Beispiel von

[14] Siehe hierfür die Funktion `optimalNumOfBits`: `https://github.com/google/guava/blob/6ad72dd8e97688745267edfa7d0da7a432f57170/guava/src/com/google/common/hash/BloomFilter.java#L529`

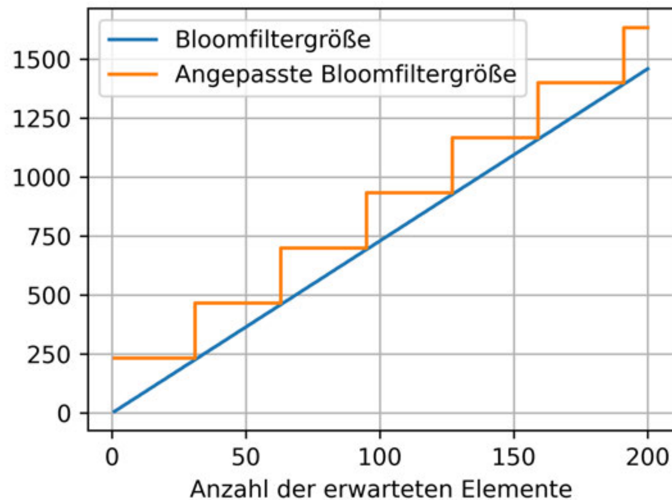


Abbildung 5.4: Berechnung der Bloomfiltergröße in der *Google Guava* Implementierung in Abhängigkeit der zu erwartenden Elemente mit der standardmäßigen Falsch-positiv-Rate von 3 %

„Lea" und „Lena" weiterhin auftreten. Jedoch ist somit dieser Fehler kontrollierbar und kann auf ein bestimmtes Maß vom Entwickler minimiert werden.

Anschließend werden die normalisierten Attributwerte einer Entität in q-Gramme aufgeteilt und in den erstellten Bloomfilter eingefügt. *Google Guava* speichert die gesetzten Bits in einem `long`-Array. Dessen Inhalt kann in jede Datenstruktur geschrieben werden, welche die Java-Klasse `OutputStream` erbt. Der Inhalt des Bloomfilters wird mit einem `ByteArrayOutputStream` in ein `byte`-Array geschrieben, welches mit *Base64* kodiert wird. *Base64* erlaubt eine kompakte Übertragung von binären Datenstrukturen, da es 24 Bits mit 4 *ASCII*-Zeichen kodiert.

Der gesamte Prozess auf Datenvorverarbeitung und Maskierung wird für jede gesendete Entität durchgeführt. Das Ergebnis ist die Liste aller kodierten Entitäten mit ihren eingangs beschriebenen Identifikatoren.

5.2 Matcher

Die Aufgabe des Matchers besteht in der Berechnung der Ähnlichkeit von Bitvektorpaaren. In diesem Abschnitt wird das Konzept des Matchers vorgestellt. Anschließend wird sein Datenmodell und seine Funktionsweise beschrieben.

5.2.1 Konzeptioneller Überblick

Der Matcher ist für den Vergleich von Bitvektorpaaren vorgesehen. Hierfür erhält der Matcher zwei Listen von Bitvektoren, welche auf ihre Ähnlichkeit zueinander überprüft

werden sollen. In Abbildung 5.5, und auf den folgenden Seiten, werden diese Listen „Domain“ und „Range“ genannt – der englischen Übersetzung von „Definitionsbereich“ und „Wertebereich“ entstammend. Der Matcher erhält zusätzlich eine Konfiguration, welche das anzuwendende Ähnlichkeitsmaß und die Parameter für die Ermittlung der Ergebnismenge enthält.

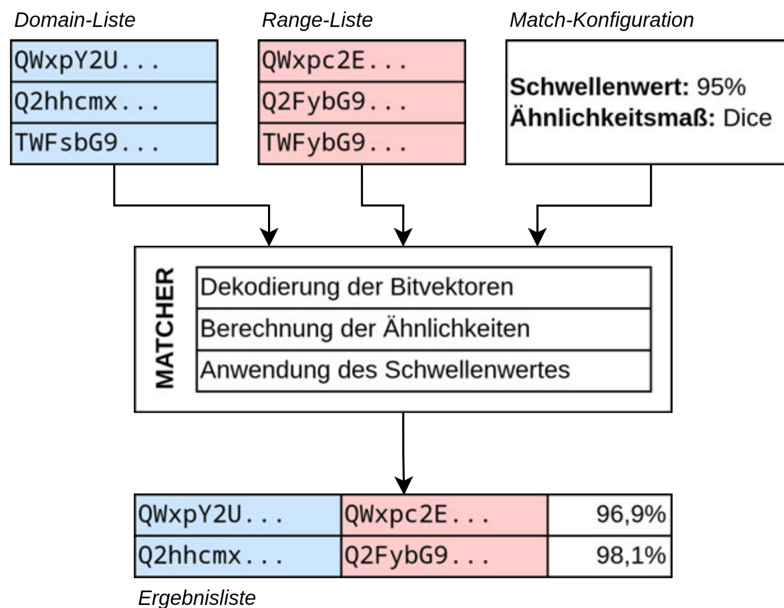


Abbildung 5.5: Schematische Arbeitsweise des PPRL-Matchers. Zwei Bitvektor-Listen werden zusammen mit einer Match-Konfiguration an den Matcher übergeben. Dieser berechnet die Ähnlichkeit zwischen Bitvektorpaaren und gibt Ergebnisse entsprechend der Konfiguration aus.

Die Abbildung zeigt ein Beispiel anhand von zwei Listen mit je drei Bitvektoren. Durch die Match-Konfiguration wird die Ähnlichkeit zwischen Bitvektoren mithilfe des Dice-Koeffizienten berechnet. Die Ergebnismenge enthält nur Bitvektorpaare mit einer Ähnlichkeit von mindestens 95 %. Ähnlichkeitsmaß und Schwellenwert sollen frei konfigurierbar sein.

5.2.2 Datenmodell

Die Modellklassen für den Matcher sind in Abbildung 5.6 zu sehen. `MatchEntity` beschreibt eine maskierte Entität. Ihr einziges Feld ist der Bitvektor, der vom PPRL-Encoder für die Attribute der Entität erzeugt wurde. Eine `MatchRequest` enthält die Domain- und Range-Liste, welche die zu vergleichenden Bitvektoren enthalten.

Zusätzlich ist mit jeder Anfrage an den Encoder eine `MatchConfiguration` enthalten. Diese beschreibt, wie das Matching durchgeführt werden soll. Insgesamt besteht `MatchConfiguration` aus vier Parametern. Der Wert von `matchFunction` gibt an, welche Ähnlichkeitsmetrik verwendet werden soll und ob die übermittelten Bloomfilter noch-

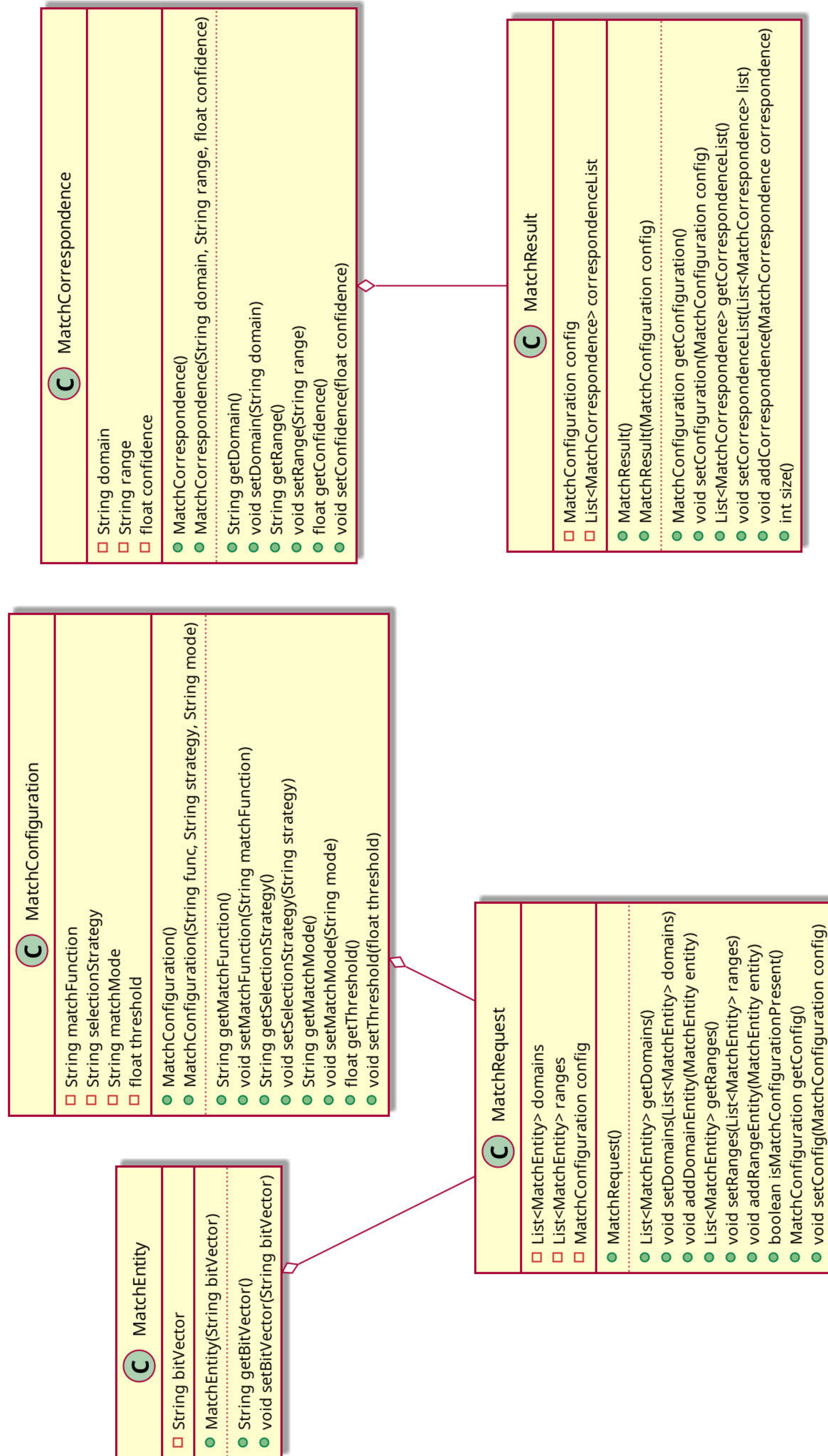


Abbildung 5.6: UML-Diagramm der Modellklassen in Verwendung des PPRL-Matchers

mals in q-Gramme aufgeteilt werden sollen. Das Feld `selectionStrategy` sagt aus, ob und wie Ergebnisse gefiltert werden sollen. Die Vorgehensweise, wie die übergebenen Listen von Bitvektoren miteinander verglichen werden sollen, kann mit `matchMode` festgelegt werden. Zuletzt kann ein Schwellenwert mit `threshold` festgelegt werden, welcher angewendet wird, sofern die ausgewählte `selectionStrategy` dies zulässt.

Der Matcher antwortet auf eine Match-Anfrage mit einer Liste von berechneten Ähnlichkeiten für übermittelte Bitvektorpaare, welche mit `MatchCorrespondence` repräsentiert werden. Zudem enthält jedes `MatchResult` eine Kopie der Match-Konfiguration, die in der Anfrage angegeben wurde.

Die Listings in Anhang B zeigen eine mögliche Anfrage an den Matcher und die darauf entstehende Antwort. In der Anfrage sind je ein Bitvektor in der Domain- und in der Range-Liste enthalten, deren Ähnlichkeit zueinander berechnet und der Antwort beigefügt ist. Die Konfiguration sagt aus, dass der Dice-Koeffizient verwendet und alle Ergebnisse zurückgegeben werden sollen.

5.2.3 Endpunkte

Zusätzlich zum Wurzel- und zum Monitoring-Pfad definiert der Match-Service eine Reihe an Endpunkten, mit denen die möglichen Parameter für das Matching abgefragt werden können. `/match-methods` listet alle Werte, die an der Stelle von `matchFunction` in `MatchConfiguration` eingesetzt werden können. Analog bietet dies `/match-modes` für das `matchMode`-Feld und `/selection-strategies` für `selectionStrategy`. Der `/match`-Endpunkt führt das eigentliche Matching auf Grundlage von vorhin beschriebenen Match-Anfragen durch.

5.2.4 Matching

Die Match-Konfiguration beeinflusst die Art und Weise des Abgleichs von Bitvektorpaaren untereinander. Der Einfluss der darin enthaltenen Parameter wird in den folgenden Absätzen beschrieben.

Match-Modi Zum Zeitpunkt der Arbeit sind zwei Modi implementiert: paar- und kreuzweises Matching. Angenommen $D = \{d_1, d_2, \ldots, d_m\}$ und $R = \{r_1, r_2, \ldots, r_n\}$ seien jeweils die Mengen aller Bitvektoren in der Domain- und Range-Liste. Beim paarweisen Matching werden die Elemente beider Listen — wie der Name bereits suggeriert — paarweise miteinander abgeglichen. Das bedeutet, dass zuerst d_1 mit r_1, d_2 mit r_2 und so weiter verglichen werden. Als Grundvoraussetzung gilt allerdings, dass die beiden Listen gleich lang sind, also $n = m$ ist. Ist dem nicht so, so wird die Anfrage vom Matcher abgewiesen. Beim kreuzweisen Matching wird zuerst das Kreuzprodukt über beide

Listen gebildet. Es wird also d_1 mit r_1 bis r_n abgeglichen, anschließend d_2 mit r_1 bis r_n und so weiter. Das paar- und kreuzweise Matching hat somit eine Zeitkomplexität von jeweils $\mathrm{O}(n)$ und $\mathrm{O}(n^2)$.

Das kreuzweise Matching hat den Vorteil, dass mit kürzeren Domain- und Range-Listen weitaus mehr Vergleichsoperationen durchgeführt werden können als beim paarweisen Matching. Jedoch ist nicht nur die Zeitkomplexität beim kreuzweisen Matching quadratisch, sondern auch der Anspruch auf den Hauptspeicher. Beim paarweisen Matching sind die Laufzeit- und Speicheranforderungen besser vorhersehbar.

Ähnlichkeitsmetriken Um die Ähnlichkeit zweier Bitvektoren miteinander zu vergleichen, werden die übereinstimmenden Positionen der gesetzten Bits gezählt. Der Match-Service implementiert den Dice-Koeffizient, den Jaccard-Index und die Kosinus-Ähnlichkeit. Alle benötigen drei Werte: die Anzahl der gesetzten Bits in den beiden Bitvektoren und deren Schnittmenge.

```
    A = 01001000 01101001
    B = 01101110 01100001
---------------------------
A & B = 01001000 01100001
```

Abbildung 5.7: Schnittmenge zweier Bitvektoren mit bitweiser Konjugation

Es gibt zwei Arten, auf die Bitvektoren hinsichtlich ihrer Ähnlichkeit überprüft werden können: bitweiser und direkter Vergleich mit q-Grammen. Der bitweise Vergleich ist sehr effizient, da bereits auf binären Daten gearbeitet wird. Um die Schnittmenge der Bitvektoren zu bestimmen, müssen sie lediglich mit einer bitweisen Konjugation miteinander verknüpft werden, wie in Abbildung 5.7 zu sehen ist.

```
          A = 01001000 01101001
          B = 01101110 01100001
----------------------------------------------------------
        A_2 = 01 10 00 01 10 00 00 00 01 11 10 01 10 00 01
        B_2 = 01 11 10 01 11 11 10 00 01 11 10 00 00 00 01
----------------------------------------------------------
  A_2 & B_2 =  1  0  0  1  0  0  0  1  1  1  1  0  0  1  1
```

Abbildung 5.8: Schnittmenge von zwei Bitvektoren mit q-Grammen

Bei der zweiten Variante werden ähnlich wie bei der Maskierung von Attributwerten die Bitvektoren in q-Gramme aufgeteilt. Es werden jedoch nicht die Bits am Rande mit Leerzeichen aufgefüllt. Die q-Gramme werden der Reihe nach auf Gleichheit untersucht. Ein Beispiel mit Bigrammen ist in Abbildung 5.8 dargestellt. Die Schnittmenge wird bestimmt, indem nacheinander die q-Gramme auf Gleichheit überprüft werden.

```
          A = 01001000
          B = 01101110 01100001
-----------------------------------------------------------
        A_2 = 01 10 00 01 10 00 00 XX XX XX XX XX XX XX XX
        B_2 = 01 11 10 01 11 11 10 00 01 11 10 00 00 00 01
-----------------------------------------------------------
  A_2 & B_2 =  1  0  0  1  0  0  0  0  0  0  0  0  0  0  0
```

Abbildung 5.9: Schnittmenge von zwei Bitvektoren unterschiedlicher Länge mit q-Grammen

Ein Sonderfall tritt ein, wenn die Bitvektoren wie in Abbildung 5.9 verschiedene Längen aufweisen. Dieser Fall muss beim bitweisen Vergleich nicht berücksichtigt werden, da sowieso nur die gesetzten Bits auf Gleichheit überprüft werden. Bei der Variante mit q-Grammen hingegen muss der kürzere Bitvektor nach der Segmentierung auf die gleiche Länge wie die des längeren Bitvektors erweitert werden. Dabei sind alle erweiterten q-Gramme des kürzeren Bitvektors so zu behandeln, als würden sie nicht mit denen des längeren Bitvektors übereinstimmen.

Auswahl der Ergebnisse Die Liste der berechneten Ähnlichkeiten für jedes Bitvektorpaar kann gefiltert werden. Der Matcher bietet hierfür vier Möglichkeiten. Die einfachsten Varianten sind das Zurückgeben der gesamten Ergebnisliste und das Anwenden eines Schwellenwertes, welcher mit der Match-Konfiguration mitgeliefert wird.

Die anderen beiden Varianten sind nur anwendbar auf den kreuzweisen Vergleich von Bitvektoren und findet im weiteren Verlauf bei der Beschreibung des Brokers keine Anwendung. Dennoch werden sie der Vollständigkeit halber hier erwähnt. Eine weitere Methode ist die Anwendung des „Stable Marriage“-Problems auf alle Bitvektorpaare, um einen optimalen Match für jedes Element aus der Domain- und der Range-Liste zu erhalten. Zusätzlich existiert die Möglichkeit der Sortierung aller berechneten Ähnlichkeiten in absteigender Reihenfolge. Die resultierende Liste wird verwendet, um jedem Bitvektor aus der Domain- und Range-Liste jeweils denjenigen Bitvektor aus der anderen Liste zuzuordnen, der noch nicht mit einem anderen Bitvektor verknüpft ist und zu ihm die höchste Ähnlichkeit aufweist.

6 Broker

Der Broker ist die umfangreichste Komponente im Rahmen dieser Arbeit. Seine Aufgabe besteht in der Durchführung von PPRL für eine beliebige Anzahl an Teilnehmern. Hierfür muss die Kommunikation zwischen den Stationen und dem Matcher orchestriert und Zwischenergebnisse während des Matchings gespeichert werden. Weiterhin muss der Fortschritt des Matchings, aufgrund der langen Laufzeit für große Mengen an Bitvektoren, für jeden Teilnehmer einsehbar sein. In diesem Kapitel wird das Konzept, der Aufbau und die Funktionsweise des Brokers umfassend beschrieben.

6.1 Konzeptioneller Überblick

Der Broker steht als Schnittstelle zwischen den Stationen, die ihre Daten mit PPRL verknüpfen wollen, und drei Diensten: einem Cache, einer Queue und einem Match-Service. Diese Dienste werden benötigt, um das asynchrone Matching von Bitvektoren durchführen zu können. Die Kommunikation zwischen dem Broker und den genannten Diensten wird in den folgenden Absätzen beschrieben. Somit wird deren Notwendigkeit unterstrichen und die Abläufe illustriert, die durch eingehende Anfragen ausgelöst werden.

Die Verwendung des Brokers umfasst sechs Phasen, welche in Abbildung 6.1 dargestellt sind. Zuerst muss eine Session am Broker registriert werden. Das geschieht im Prozess der Zugerstellung. In einer Session werden Bitvektoren von allen teilnehmenden Stationen gesammelt. Die Bitvektoren werden vom Resolver, welcher sich an den jeweiligen Stationen befindet, an den Broker gesendet. Der Broker nimmt die Bitvektoren entgegen und speichert diese in einem Cache. Die Aufgabe des Caches besteht in der Verwaltung von Bitvektoren und dem Speichern von Übereinstimmungen, welche später vom Match-Service gemeldet werden.

Wenn Bitvektoren von mehr als zwei Stationen zugesendet wurden, so lädt der Broker Bitvektoren in der jeweiligen Session aus dem Cache, um Bitvektorpaare zu erzeugen. Diese werden anschließend an eine für die Session dedizierte Queue übergeben. Im Hintergrund führt der Broker das Matching durch. Da der Match-Service eine Konfiguration benötigt, muss diese bei der Registrierung der Session angegeben werden. Wenn der Match-Service Ergebnisse vermeldet, dann werden diese im Cache gespeichert. Während des Matchings soll es jeder Station möglich sein, den Fortschritt des Matchings abzufragen. Hierfür bezieht der Broker den Status der Queue ein.

Jede Station kann anschließend die Ergebnisse für die eigens beigesteuerten Bitvektoren vom Broker abfragen. Der Broker stellt eine entsprechende Anfrage an den Cache

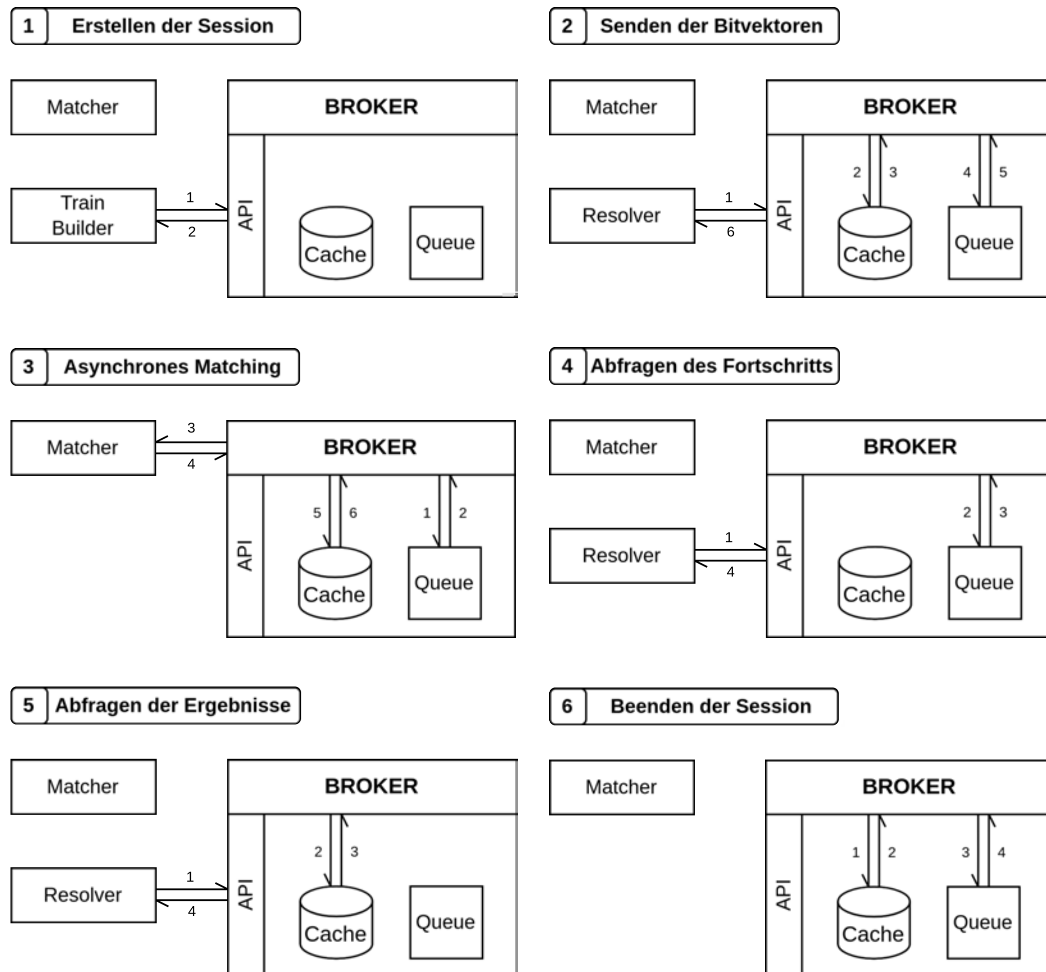


Abbildung 6.1: Schematische Arbeitsweise des PPRL-Brokers. Zuerst wird eine Session am Broker erstellt. Für diese Session können Bitvektoren an den Broker gesendet werden, welche von ihm im Cache gespeichert werden. Sind Bitvektoren von mindestens zwei verschiedenen Clients vorhanden, werden diese der Queue hinzugefügt. Im Hintergrund führt der Broker das Matching aus. Bitvektorpaare werden aus der Queue gelesen, dem Matcher übergeben und Ergebnisse im Cache gespeichert. Währenddessen kann der Fortschritt der Session abgefragt werden. Hierfür wird die Länge der Queue eingerechnet. Ergebnisse werden vom Broker über den Cache ausgelesen. Wird die Session beendet, so werden alle Session-Daten von Cache und Queue gelöscht.

und gibt diese Ergebnisse an den Resolver der jeweiligen Station weiter. Anschließend wird die Session beendet. Cache und Queue werden von Daten, die zur Session gehören, bereinigt.

Eine Session sollte automatisch als auch manuell beendet werden können. Das bedeutet, dass das Löschen von Daten in einer Session durch eine direkte Anfrage an den Broker ausgelöst werden soll. Aber auch das Festsetzen einer Laufzeit für eine Session soll ermöglicht werden. Somit kann sichergestellt werden, dass alle Daten einer Session ab einem bestimmten Zeitpunkt mit Sicherheit gelöscht sind. Das bedeutet, dass zusätzlich zur Match-Konfiguration eine Abbruchsmethode bei der Registrierung einer Session am Broker angegeben werden sollte.

6.2 Datenmodell

Die Übersicht aller Modellklassen für den Broker ist in Abbildung 6.2 dargestellt. Eine `BitVectorEntity` besitzt, analog der `MatchEntity` für den Matcher, ein Feld: der Bitvektor einer maskierten Entität. Jede `SubmitRequest` enthält eine `BitVectorEntity`-Liste. `MatchedBitVectorEntity` erbt alle Eigenschaften von `BitVectorEntity` und erweitert diese um das Feld `confidence`. Dieses Feld beschriebt, wie hoch die Ähnlichkeit zu einem anderen Bitvektor eines fremden Clients in der gleichen Match Session ist. Im Gegensatz zur Matcher-Klasse `MatchConfidence` wird der übereinstimmende Bitvektor nicht mitgeführt. Eine Liste von `MatchedBitVectorEntity` ist in jeder `ResultResponse` enthalten.

Die Informationen zur Beendigung einer Session sind in `SessionCancellation` enthalten. Sie enthält den Namen der Methode und eine Liste von Optionen, die im Zusammenhang mit der gewählten Methode einbezogen werden sollen. Die Interpretation der angegebenen Optionen sind abhängig von der jeweiligen Implementierung der Abbruchsmethode und wird im Laufe dieses Kapitels beschrieben. Jede `SessionRequest` enthält eine `SessionCancellation<`. Zusätzlich muss eine Match-Konfiguration übergeben werden, damit der Broker das Matching eigenständig verwalten kann.

Weiterhin existieren zwei Klassen, die lediglich als Halter für einzelne Antwortwerte dienen. `ProgressResponse` führt das Feld `progress`, welches über den Fortschritt der laufenden Match-Session informiert. Zuletzt wird eine generische Antwort mit einem geheimen Wert, welcher im Feld `secret` gespeichert ist, durch `SecretResponse` bereitgestellt.

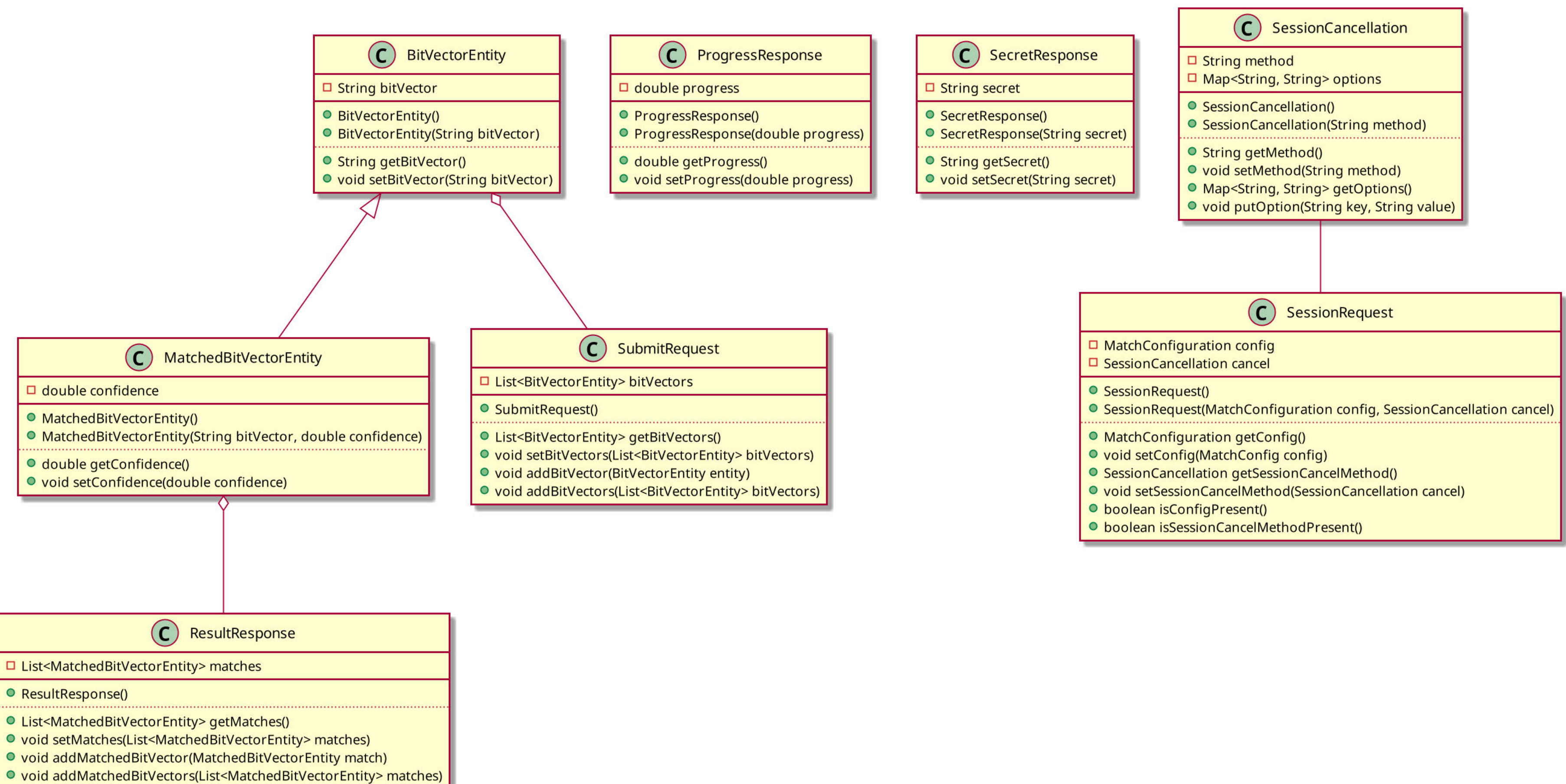


Abbildung 6.2: UML-Diagramm der Modellklassen in Verwendung des PPRL-Brokers

6.3 Endpunkte

Die Funktionen des `/session`-Endpunktes umfassen das Erstellen und das Beenden von Match Sessions. Mit einer entsprechenden `POST`-Anfrage kann jeder Client eine neue Session starten und im Gegenzug ein Secret erhalten, welches die Session eindeutig identifiziert. Das erhaltene Secret dient gleichzeitig als authentifizierendes Merkmal für viele weitere Endpunkte. Um beispielsweise eine Session mit einer `DELETE`-Anfrage an den gleichen Endpunkt zu beenden, wird stets das Session Secret gefordert. Je nach gewählter Methode zur Beendigung der Session können allerdings auch weitere Parameter notwendig sein. Die Liste aller gültigen Methoden kann unter `/session-cancellation-methods` abgefragt werden.

Der Funktionsumfang des `/submit`-Pfades ist ebenso zweigeteilt, jedoch richtet sich dieser explizit an Teilnehmer einer Match Session. Mit einer `GET`-Anfrage und einem gültigen Session Secret wird ein Client Secret erzeugt. Dieses Client Secret muss in Verbindung mit einer anschließenden `POST`-Anfrage gesendet werden, um eine Liste von Bitvektoren an den Matcher zu übermitteln.

Um den Fortschritt einer Match Session zu beurteilen, wird der `/progress`-Endpunkt zur Verfügung gestellt. Dieser meldet, unter Angabe eines gültigen Session Secrets, wie weit das Matching bereits fortgeschritten ist. Der Fortschritt wird in Prozent angegeben. Zuletzt kann jeder Client jederzeit mit dem eigenen Client Secret den `/result`-Endpoint verwenden, um eine Liste von Matches für die eigenen übermittelten Bitvektoren zu erhalten.

6.4 Komponenten

Bevor auf den Arbeitszyklus des Brokers eingegangen wird, ist eine Beschreibung der Komponenten notwendig, von denen der Broker abhängig ist. In den folgenden Abschnitten wird erläutert, wie Sessions im Broker repräsentiert und verwaltet werden. Weiterhin werden die Implementierungsdetails der Anbindung externer Dienste beschrieben. Zu diesen Diensten zählen der Match-Service, der Cache zum Speichern von Match-Ergebnissen und die Queue zum Auflisten von Bitvektorpaaren, die noch abgeglichen werden müssen.

6.4.1 Zuordnung von Sessions

Der Broker führt eine Liste aller laufenden Match Sessions. Diese Liste muss von vielen Teilen der Anwendung erreichbar, zentral und konsistent sein. Zu diesem Zweck existiert der `MatchSessionMapper`, welcher Referenzen zu `MatchSession`-Instanzen verwaltet.

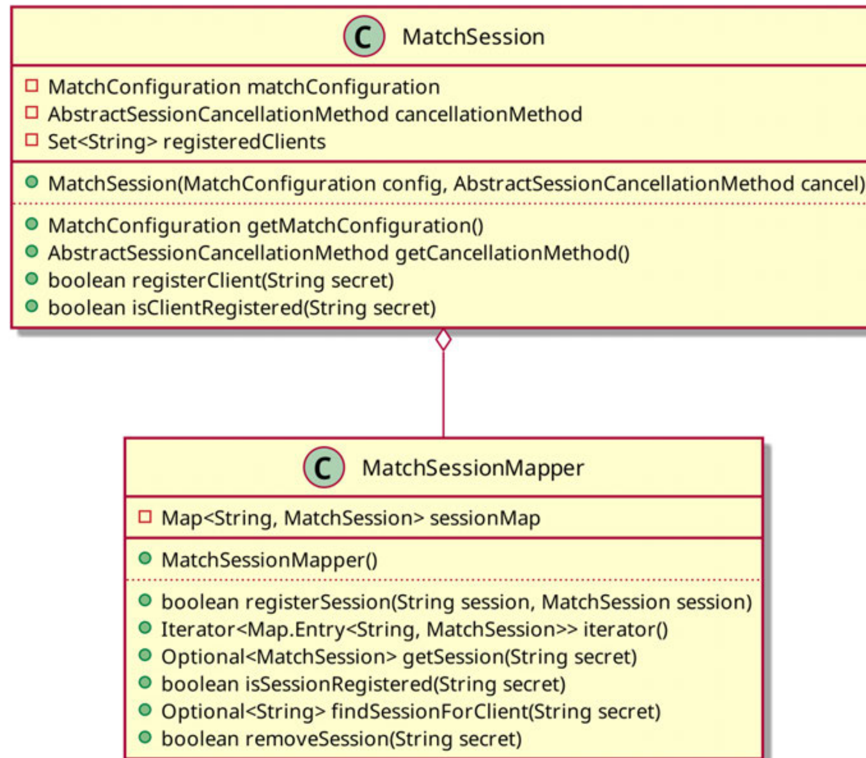


Abbildung 6.3: UML-Diagramm der Modellklassen für die globale Zuordnung von Sessions zu ihren Session Secrets

Der Aufbau dieser beiden Klassen ist in Abbildung 6.3 aufgeführt. `MatchSession` ist eine Zusammenfassung aller Felder, die in einer Session-Anfrage an den Broker gesendet werden. Sie enthält sowohl die Konfiguration für den Matcher als auch eine Abbruchsmethode, welche Auskunft erteilt, wann die Session-Instanz beendet werden kann. Weiterhin enthält eine `MatchSession`-Instanz eine Menge eindeutiger Secrets, die dieser Session angehören und an teilnehmende Clients verteilt wurden.

`MatchSession` verfügt über die Funktion `registerClient`. Sie fügt Client Secrets der Session hinzu und gibt nur dann `true` zurück, wenn das übergebene Secret noch nicht in der Menge der bereits registrierten Secrets enthalten ist. `isClientRegistered` hingegen überprüft, ob ein Client Secret in einer Session vorhanden ist und gibt im Erfolgsfall `true`, andernfalls `false`, zurück.

`MatchSessionMapper` führt eine Zuordnung von Session Secrets zu `MatchSession`-Instanzen. Darüber hinaus besitzt er die Funktion `registerSession`, welche nur einen Erfolg vermeldet, wenn das übergebene Secret noch nicht einer anderen Session zugewiesen wurde. In diesem Fall gibt die Funktion `true`, andernfalls `false`, zurück. Die Funktion `findSessionForClient` iteriert über alle aktiven Sessions und überprüft, ob eine Session über das gegebene Client Secret verfügt. Ist das der Fall, so wird das Secret dieser Session zurückgegeben.

6.4.2 Abbruchsmethoden

Die Klasse `AbstractSessionCancellationMethod` gilt als Vorlage für alle Methoden, mit denen eine Session abgebrochen oder beendet werden kann. Ihre Struktur ist in Abbildung 6.4 zu sehen. Eine neue Instanz einer solchen Implementierung erhält eine Liste von Optionen. Wie diese zu interpretieren sind, hängt von der jeweiligen Methode ab.

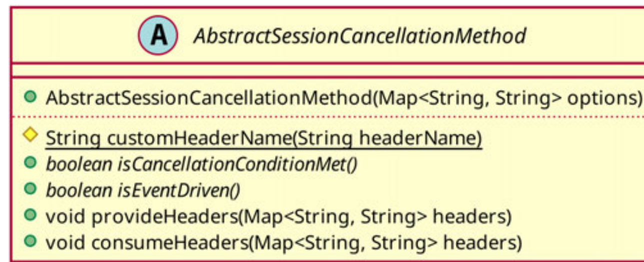


Abbildung 6.4: UML-Diagramm der abstrakten Klasse für Session-Abbruchsmethoden

Jede Implementierung muss die beiden Funktionen `isCancellationConditionMet` und `isEventDriven` überschrieben. Erstere gibt zurück, ob die Session, die mit dieser Methode assoziiert ist, beendet werden kann. Dies kann auf zwei Weisen eintreten: ereignisgesteuert oder zeitgesteuert. Dafür ist der Rückgabewert von `isEventDriven` ausschlaggebend.

Gibt `isEventDriven` den Wert `true` zurück, so ist die Methode von Eingaben von außerhalb abhängig, um eine Session zu beenden. Das geschieht über Anfragen an den `/session`-Endpunkt. Wird eine neue Session erstellt mit einer ereignisgesteuerten Methode, so wird die Funktion `provideHeaders` aufgerufen. Der Methode ist es möglich, spezielle HTTP-Header festzulegen, die zur Antwort auf die Session-Anfrage angehängt werden sollen. Diese Header können an spezielle Bedingungen gebunden sein, unter denen die Session beendet werden kann. Wichtig ist, dass die Namen dieser Header mit `X-` anfangen müssen, um zu signalisieren, dass es sich um applikationsspezifische Header handelt.

Wird die Beendigung einer Session über den `/session`-Endpoint angefordert, so werden alle Header aus der eingehenden Anfrage, die mit `X-` beginnen, an die Funktion `consumeHeaders` der an die Session gekoppelte Abbruchsmethode übergeben. Die Methode kann daraufhin die Header verarbeiten und den eigenen Status aktualisieren.

Gibt hingegen `isEventDriven` den Wert `false` zurück, so muss die Methode eigenständig entscheiden, wann die verknüpfte Session zu beenden ist. Hierfür wird die Methode `isCancellationConditionMet` periodisch abgerufen. `provideHeaders` kann verwendet werden, um den anfragenden Client über die Details der Abbruchbedingung zu informieren.

Zum Zeitpunkt der Arbeit existieren drei Methoden zum Beenden einer Session. Die „Simple“-Methode ist ereignisgesteuert und erlaubt die Beendigung einer Session zu jedem Zeitpunkt. Das bedeutet, dass jedermann im Besitz des Session Secrets die Session jederzeit über den `/session`-Endpunkt beenden kann. Diese Methode ist für Test- und Evaluationszwecke geeignet, jedoch in der Praxis nicht sehr praktikabel. Es ist davon auszugehen, dass alle teilnehmenden Clients ebenfalls im Besitz des Session Secrets sind und somit die Befugnis haben, die Session vorzeitig zu beenden.

Die „Token“-Methode ist ebenfalls ereignisgesteuert, erzeugt jedoch bei der Initialisierung ein zusätzliches Secret. Dieses Secret wird beim Aufruf von `provideHeaders` als Header angehängt. Nur in der Kombination vom Session Secret und dem von der Token-Methode erzeugten Secret ist es möglich, eine Session mit dieser Methode zu beenden. Somit ist ausschließlich der Verwalter der Session im Besitz der notwendigen Legitimation, um eine Session zu beenden.

Die dritte Methode ist die „Timeout“-Methode. Diese ist zeitgesteuert und wartet auf einen Zeitpunkt in der Zukunft, ab dem die Session zu beenden ist. Dieses Intervall kann über eine Timeout-Option gesteuert werden. Ist diese Option nicht gesetzt, so beträgt das Standard-Intervall einen Tag. Diese Intervall ist lediglich eine Schätzung, wie lange eine Session bei mehreren Teilnehmern dauern könnte, bis jeder seine Bitvektoren zugesendet und die Ergebnisse abgerufen hat.

6.4.3 Client für den Matcher

Der Broker agiert als Schnittstelle zwischen Clients und dem Matcher. Das bedeutet, dass er in der Lage sein muss, Anfragen an den Matcher zu stellen. Die wichtigsten Funktionen zur Kommunikation mit dem Matcher werden durch das Interface `MatchRequestSender` beschrieben. Das Interface ist in Abbildung 6.5 zu sehen.

Abbildung 6.5: UML-Diagramm des Interfaces für den Matcher-Client

Der Funktionsumfang ist sehr kompakt und beschreibt nur die Aufrufe, die vom Broker benötigt werden. Die Funktionen `getMatchMethods` und `getSelectionStrategies` geben jeweils die unterstützten Ähnlichkeitsmetriken und Filterstrategien eines Matchers zurück. `matchEntities` besitzt zwei Signaturen, die sich von ihrer Funktionsweise allerdings nicht unterscheiden. Beide stellen aufgrund der übergebenen Parameter eine Match-Anfrage an den Matcher. Im Broker befindet sich eine einfache Implementierung auf Basis des Glassfish Jersey HTTP Clients.

6.4.4 Cache für Match-Ergebnisse

Für das Zwischenspeichern von Match-Ergebnissen wird die Graphdatenbank *Neo4j*[15] verwendet. Bitvektoren werden als Knoten modelliert und die Kanten repräsentieren die Ähnlichkeiten von Bitvektoren untereinander. Diese Art der Modellierung wurde gewählt, weil somit vor allem Bitvektoren mit mehreren Übereinstimmungen einfach eingepflegt und abgefragt werden können.

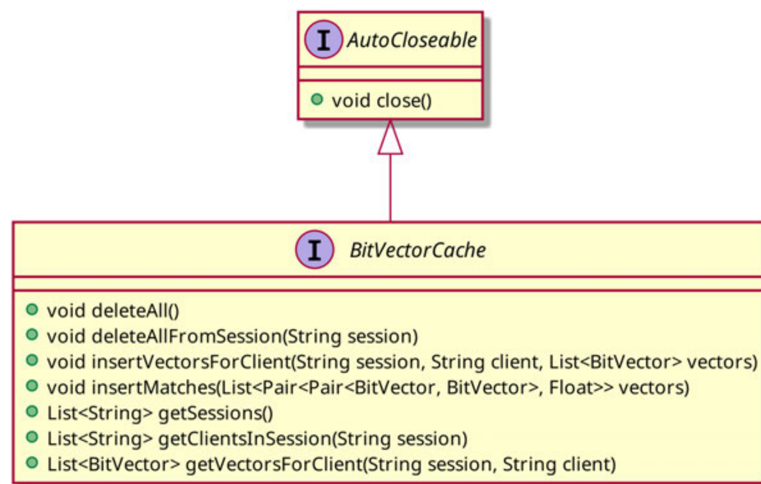


Abbildung 6.6: UML-Diagramm des Interfaces für den Bitvektor-Cache

Für die Cache-Implementierung mit *Neo4j* und alternative Implementierungen in der Zukunft beschreibt das Interface `BitVectorCache` den erwarteten Funktionsumfang. Das Interface ist in Abbildung 6.6 dargestellt. `BitVectorCache` erbt die Funktion `close` der Java-Standard-Interfaces `AutoCloseable`, um mögliche offene Verbindungen bei Beendigung des Webservices zu schließen. Das Löschen von Einträgen aus dem Cache wird mit den Funktionen `deleteAll` und `deleteAllFromSession` erzielt. Erstere löscht alle Einträge, letztere entfernt nur Einträge, die mit einer bestimmten Session verknüpft sind.

Die Funktion `insertVectorsForClient` erzeugt neue Bitvektor-Einträge im Cache. Dafür erhält die Funktion die Kennzeichnung der Session, die Client-Kennung und die Liste der vom Client übermittelten Bitvektoren. Das Einfügen der eigentlichen Relationen der Bitvektoren untereinander wird durch die Funktion `insertMatches` gehandhabt. Sie erhält eine Liste von Bitvektorpaaren und deren berechnete Ähnlichkeiten zueinander. Es wird vorausgesetzt, dass die Bitvektoren vor dem Einfügen der Matches bereits im Cache vorhanden sind.

Zuletzt sind einige Funktionen definiert, um den Datenbestand des Caches abzufragen. `getSessions` gibt die Liste aller vorhandenen Session-Kennungen zurück. Die Funktion `getClientsInSession` gibt unter Angabe einer Session-Kennung die Liste aller

[15] Siehe `https://neo4j.com/`

Client-Kennungen zurück, die an dieser Session teilnehmen. Weiterhin ist die Funktion `getVectorsForClient` beschrieben, welche für einen Client in einer Session die Liste aller übermittelten Bitvektoren zurückgibt.

Für die Anbindung von *Neo4j* wird der offizielle *Neo4j*-Java-Graphdatenbanktreiber[16] verwendet. Alle Anfragen an die Datenbank sind in der Anfragesprache *Cypher*[17] formuliert. Der Datenbanktreiber besitzt die Möglichkeit Knoten- und Kantenobjekte aus einer Antwort auf eine Anfrage direkt in native Datentypen für Java umzuwandeln und umgekehrt.

```
FOREACH (vector in $vectors |
    CREATE (:BitVector {
        session: $session,
        client: $client,
        value: vector
    })
)
```

Listing 6.1: Cypher-Anfrage zum Erstellen von Bitvektor-Knoten

Die Anfrage zum Erstellen von Bitvektorknoten in Listing 6.1 zeigt zugleich das Datenschema eines solchen Knotens auf. Jeder Bitvektorknoten besitzt drei Eigenschaften: die zugehörige Session, die Client-Kennung und den eigentlichen Wert des Bitvektors. Die Menge aller einzufügenden Bitvektoren wird als Liste von Strings gesendet. Über die Liste wird dann mit `FOREACH` iteriert. Das `CREATE` in Verbindung mit `FOREACH` ist wesentlich schneller als für jeden einzufügenden Bitvektor ein einzelnes `CREATE`-Statement abzusetzen.

```
MATCH (a:BitVector { value: $domain }) WITH a
MATCH (b:BitVector { value: $range })
WHERE a.session = b.session AND a.client <> b.client
MERGE (a)-[:IS_SIMILAR_TO { confidence: $conf }]->(b)
```

Listing 6.2: Cypher-Anfrage zum Einfügen von Matches zwischen Bitvektor-Knoten

Für das Einfügen von Kanten zwischen den Knoten müssen zuerst die Knoten, welche miteinander verbunden werden sollen, ermittelt werden. `domain` und `range` in Listing 6.2 stehen für die Werte des Bitvektorpaars, für das eine Ähnlichkeit berechnet wurde. Für die Bitvektorknoten werden zwei getrennte Retrieval-Statements durchgeführt, welche dann mit `WITH` verknüpft werden.

Nachdem die Knoten für das übermittelte Bitvektorpaar identifiziert wurden, werden diese nach zwei Anforderungen gefiltert. Die Bitvektoren müssen in der selben Session

[16] Siehe https://neo4j.com/developer/java/
[17] Siehe https://neo4j.com/developer/cypher/

auftreten und sie dürfen nicht vom gleichen Client stammen. Somit wird sichergestellt, dass nur Bitvektoren von unterschiedlichen Clients miteinander verknüpft werden. Zuletzt werden mit einem `MERGE` die Kanten zwischen den ermittelten Knoten erstellt.

```
MATCH (a:BitVector {
    session: $session,
    client: $client
})-[:IS_SIMILAR_TO]-(b:BitVector {
    session: $session
})
WHERE a.client <> b.client
RETURN DISTINCT a.value, s.confidence
```

Listing 6.3: Cypher-Anfrage zum Abfragen von Matches für einen Client

Die Abfrage für gefundene Matches für einen Client ist in Listing 6.3 aufgezeigt. Es wird nach Knoten gesucht, die durch eine Kante miteinander verbunden sind. Auch hier sind wieder die Grundbedingungen vertreten, dass diese Knoten der gleichen Session angehören und die assoziierten Bitvektorwerte von unterschiedlichen Clients stammen müssen. Das Ergebnis der Anfrage erlaubt dem abfragenden Client keinen Aufschluss über fremde Bitvektoren, die mit den eigenen gematcht wurden.

```
MATCH (n) DETACH DELETE n
MATCH (n:BitVector { session: $session }) DETACH DELETE n
```

Listing 6.4: Cypher-Anfragen zum Löschen von Knoten

Das Listing 6.4 fasst die beiden Varianten zusammen, mit denen Bitvektoren aus dem Cache entfernt werden können. Die Anweisung `DETACH DELETE` löscht alle Knoten und dazugehörige Kanten, die mit der Anfrage übereinstimmen. Das Löschen des gesamten Datenbestandes im Cache wird mit der ersten Anfrage im Listing realisiert. Die zweite Anfrage löscht alle Knoten und Kanten, die einer bestimmten Session angehören.

6.4.5 Queue

Für die Queue-Implementierung wird *Redis*[18] verwendet. *Redis* ist ein sehr schneller und effizienter Schlüssel-Werte-Speicher. Die Vielzahl an verfügbaren Kommandos ermöglicht auch das Erstellen und Verwalten von Listen. Für die Queue existiert, wie für die Cache-Implementierung, ein Interface `BitVectorMatchQueue` mit einer Menge von Funktionen, die jede Queue-Implementierung unterstützen sollte. Der Funktionsumfang dieses Interfaces ist in Abbildung 6.7 dargestellt.

[18] Siehe https://redis.io/

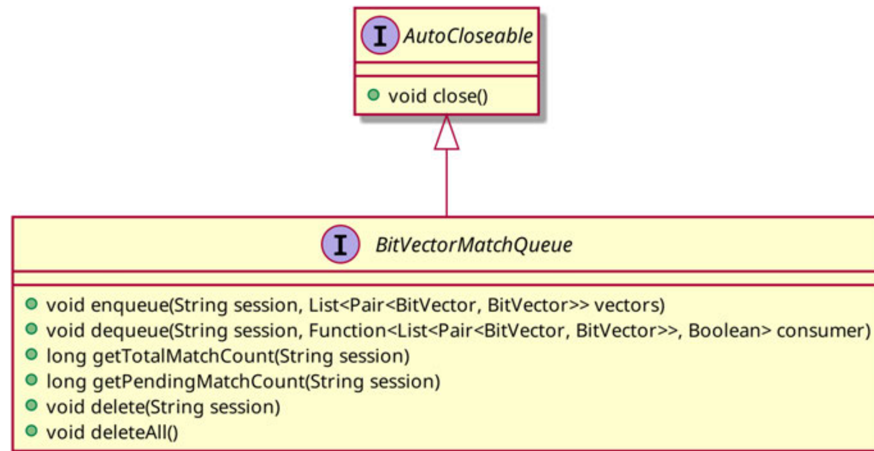


Abbildung 6.7: UML-Diagramm des Interfaces für die Bitvektor-Queue

Wie das Cache-Interface erbt `BitVectorMatchQueue` die Funktion des `AutoCloseable`-Interfaces. Die Funktion `enqueue` reiht eine Liste von Bitvektorpaaren zum Matching für eine bestimmte Session ein. Das Gegenstück ist die Funktion `dequeue`, welche eine bestimmte Menge an Bitvektorpaaren aus der Liste entfernt. Der Funktionsaufruf benötigt wieder die Kennung der Session und eine Callback-Funktion. Eine gültige Callback-Methode muss eine Liste von Bitvektorpaaren als Argument annehmen und einen Boolean zurückgeben.

Der Broker fordert eine Liste von Bitvektorpaaren von der Queue an. Bevor diese Paare jedoch als abgearbeitet zu betrachten sind, müssen diese erst an den Matcher gesendet und in den Cache eingefügt werden. Schlägt einer dieser Schritte fehl, beispielsweise durch unerreichbare Dienste, dann muss die Queue in der Lage sein, die Liste der Bitvektorpaare wiederherzustellen. Deswegen erhält `dequeue` eine Callback-Funktion als Parameter. Die Funktion gibt `true` zurück, um der Queue zu signalisieren, dass die übergebenen Bitvektorpaare korrekt verarbeitet wurden. In diesem Fall kann die Queue entweder das Entfernen der ermittelten Paare finalisieren oder bei einem Rückgabewert von `false` einen Rollback durchführen.

Die Funktion `getTotalMatchCount` gibt die Anzahl aller Bitvektorpaare zurück, die in der gesamten Laufzeit einer Session in die Queue eingefügt wurden. Im Zusammenspiel mit der Funktion `getPendingMatchCount`, welche die gegenwärtige Länge der Queue zurückgibt, lässt sich der Fortschritt des Matchings für eine Session ermitteln. Wie im Cache-Interface muss jede Queue-Implementierung die Funktionen `delete` und `deleteAll` bereitstellen. Diese löschen jeweils den gesamten Datenbestand oder die Queue für eine bestimmte Session.

Für die Queue-Implementierung auf Basis von *Redis* wird die Java-Bibliothek *Jedis* verwendet[19]. Da *Redis* ein reiner Schlüssel-Werte-Speicher ist, müssen die Schlüssel für

[19] Siehe `https://github.com/redis/jedis/`

unterschiedliche Sessions so benannt werden, sodass diese nicht miteinander kollidieren. Aus diesem Grund enthalten alle Schlüssel für eine Session den Session-Identifier als Präfix, getrennt mit einem Doppelpunkt.

```
example:count    3
example:queue    [ SGFiZW4gU2llIHNjaG9u:ZWlubWFsIEZRTGl0ZQ==
    YXVmIEdpdEh1YiB2b24=:VXNlciAicGF3bGFzemN6eWsi
    Z2VzZWhlbj8gU2Vocg==:Z3V0ZXMgUHJvamVrdCE= ]
```

Abbildung 6.8: Queue-Layout in *Redis* für eine Beispiel-Session. Bitvektorpaare sind durch Doppelpunkte getrennt und in einer Liste gespeichert. Die Anzahl aller jemals eingereihten Bitvektorpaare wird in einem zusätzlichen Schlüssel erfasst.

Angenommen eine Session mit dem Secret „example“ wird erstellt. Haben mindestens zwei Clients ihre Bitvektoren an den Broker für diese Session übermittelt, so wird eine Queue angelegt. In *Redis* wird mithilfe des Befehls `RPUSH` eine neue Liste unter dem Schlüssel `example:queue` angelegt. Die Bitvektoren werden in ihrer *Base64*-Repräsentation mit Doppelpunkten verknüpft, sodass ein Element der Liste ein Bitvektorpaar darstellt. Gleichzeitig wird ein neuer Zähler unter dem Schlüssel `example:count` erstellt, welcher sofort um die Anzahl der eingefügten Listenelemente erhöht wird. Ein Beispiel mit drei Bitvektorpaaren ist in Abbildung 6.8 dargestellt.

Beim Aufruf von `dequeue` wird das `LRANGE`-Kommando verwendet, um eine bestimmte Anzahl an Bitvektorpaaren aus der Liste zu lesen. Dabei werden sie noch nicht aus der Liste entfernt. Die Anzahl der Bitvektorpaare, die mit einem Mal geladen werden sollen, kann über eine Umgebungsvariable definiert werden. Standardmäßig ist dieser Wert auf 5000 festgelegt. Die geladenen Bitvektorpaare werden an die Callback-Funktion übergeben. Gibt diese `true` zurück, so wird `LTRIM` verwendet, um die übergebenen Bitvektorpaare zu entfernen.

Der Wert des Schlüssels `example:count` wird verwendet, um den Rückgabewert der Funktion `getTotalMatchCount` zu bestimmen. Die Anzahl der gegenwärtigen Bitvektorpaare in der Queue wird mit dem `LLEN`-Kommando im Bezug auf die Liste in *Redis* ermittelt. Für das Löschen von Queues, beziehungsweise der gesamten Datenbasis, wird `DEL` verwendet. Einzelne Sessions können einfach entfernt werden, indem die erstellten Schlüssel für die Session explizit an den Befehl übergeben werden. Um alle Daten aus *Redis* zu löschen, müssen zuerst alle noch vorhandenen Schlüssel mit dem `KEYS`-Befehl ermittelt werden. Das Ergebnis dieses Befehls kann anschließend an `DEL` weitergegeben werden.

6.5 Arbeitszyklus

Schnittstelle zwischen dem REST-Interface des Brokers und dessen internen Komponenten ist die Klasse `MatchSessionBackgroundWorker`. Die Funktionen der Klasse

sind so konzipiert, dass die Klassen für die Endpunkte des Brokers meistens nur einen einzigen Funktionsaufruf tätigen müssen, um eine Anfrage zu bearbeiten. Weiterhin ist es die Aufgabe des Workers, das Matching im Hintergrund auszuführen und sicherzustellen, dass die Ergebnisse persistiert werden. In diesem Abschnitt wird die Arbeitsweise des Workers beschrieben. Somit werden die bisher beschriebenen Teilkomponenten des Brokers zusammengeführt.

6.5.1 Initialisierung

Bei der Initialisierung des Workers wird ein `Context`-Objekt erstellt. Dabei handelt es sich um eine interne Klasse, welche Referenzen zu einer Queue-, Cache-, Mapper- und Match-Client-Implementierung enthält. Dieses `Context`-Objekt wird innerhalb des Workers verwendet und an andere Klassen weitergegeben für einen konsistenten Zugriff auf globale Objekte.

Weiterhin werden zwei Threads im Hintergrund gestartet. Ein Thread ist für den Worker selbst ausgelegt. Da dieser das `Runnable`-Interface implementiert, ist es für ihn möglich einen Codeteil asynchron auszuführen. Der zweite Thread ist für die Ausführung des Matchings und der Persistierung von Matchergebnissen vorgesehen. Mit der Initialisierung des Workers wird der asynchrone Codeteil des Workers automatisch in den ersten Thread verlagert und sofort ausgeführt.

Um den synchronen Zugriff auf die internen Variablen des Workers zu gewährleisten, wird zudem ein Lock-Objekt erstellt. Jede Funktion, die etwas am Zustand des Workers ändern möchte, muss zuerst den Zugriff auf das Lock-Objekt in einem Synchronisationsblock erlangen.

6.5.2 Hintergrundprozess

Der asynchrone Teil des Workers ist vereinfacht in Abbildung 6.9 dargestellt. Unmittelbar nach dem Start des Hintergrundprozesses werden Cache und Queue vollständig gelöscht. Daraufhin tritt eine Schleife ein, in der eine Reihe von Aktionen regelmäßig durchgeführt wird. Diese Schleife bleibt so lange aktiv bis entweder der Webservice heruntergefahren wird oder die Terminierung des Workers explizit angefordert wird.

Als erstes werden die Abbruchbedingungen aller Sessions überprüft, die auf einer nicht-ereignisbasierten Methode basieren. Wird von einer solchen Session signalisiert, dass sie abgebrochen werden kann, so wird das zugeordnete Secret in einer internen Liste gespeichert. Für jede Session in dieser Liste werden die korrespondierenden Einträge im Cache und in der Queue sofort gelöscht. Zudem wird die Session aus dem internen Mapping entfernt.

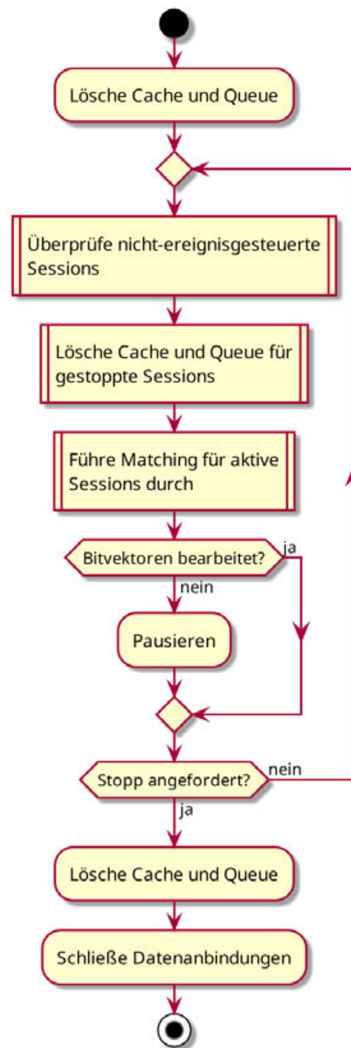


Abbildung 6.9: Ablaufdiagramm des Workers im PPRL-Broker. Cache und Queue werden zu Beginn und Schluss gelöscht. Währenddessen werden abgebrochene Sessions gelöscht und das Matching von Bitvektorpaaren in einer Dauerschleife durchgeführt.

Zuletzt wird über alle noch aktiven Sessions iteriert. Für diese Sessions wird jeweils ein neuer separater Thread gestartet, welcher für die Durchführung des Matchings verantwortlich ist. Dafür werden zunächst die nächsten Bitvektorpaare für eine Session aus der Queue entnommen.

Die Match-Konfiguration für die jeweilige Session wird ermittelt. Alle Parameter bleiben bestehen bis auf den Match-Modus, welcher zu paarweisem Matching überschrieben wird. Daraufhin werden die Bitvektorpaare zusammen mit der angepassten Match-Konfiguration an den Matcher gesendet. Meldet der Matcher Ergebnisse zurück, so werden diese im Cache gespeichert. Der Job gibt als Ergebnis die Anzahl der Bitvektorpaare aus der Queue zurück, die erfolgreich abgearbeitet wurden.

Die Rückgabewerte werden vom Worker verarbeitet. Meldet keiner der Jobs, dass die jeweilige Queue verkürzt wurde, so geht der Worker in einen kurzen Leerlauf über. Diese kann über eine Umgebungsvariable definiert werden und ist standardmäßig auf fünf Sekunden festgelegt. Wurde allerdings mindestens eine Queue bearbeitet, so wird die Pause übersprungen und die Schleife beginnt von vorn.

Der Grund für dieses Verhalten liegt in der Tatsache, dass der Broker sich in den meisten Fällen für sehr lange Zeiten im Leerlauf befindet. In dieser Zeit sollte der Broker so wenig wie möglich Ressourcen beanspruchen. Sind jedoch Bitvektoren miteinander zu vergleichen, so sollte der Broker diese so schnell wie möglich abarbeiten um alsbald wieder in den Leerlauf zu kommen.

Wird die Terminierung des Workers angefordert, so bricht dieser aus der Schleife bei der nächsten Gelegenheit aus. Das bedeutet, dass der Worker die restlichen Schritte noch einmal abarbeitet wenn er sich im Moment am Beginn der Schleife befindet. Wurde die Schleife verlassen, so werden erneut Cache und Queue vom gesamten Datenbestand bereinigt und die Verbindungen zu diesen Services geschlossen.

6.5.3 REST-Interface

Wie eingangs erwähnt, ist der Worker die Schnittstelle für alle REST-Endpunkte. Wird eine neue Session über den `/session`-Endpoint erstellt, so wird zuerst die übergebene Match-Konfiguration auf ihre Gültigkeit überprüft. Das wird realisiert, indem die übertragene Ähnlichkeitsmetrik und Filterstrategie entnommen und zur Verifizierung gegen die unterstützten Methoden des Matchers geprüft werden. Weiterhin wird versucht, die angegebene Abbruchsmethode aufzulösen.

Mit der Match-Konfiguration und Abbruchsmethode wird ein neues Session-Objekt erstellt. Dieses wird an den Worker gegeben um ein neues Secret zu erstellen und um die Session im globalen Kontext zu registrieren. Jedes vom Broker erstellte Secret wird über Java’s `SecureRandom`-Klasse[20] erzeugt. Hierfür wird ein Byte-Array der Länge 32 erstellt und mit zufälligen, kryptografisch sicher erzeugten Bytes befüllt. Der Inhalt des daraus resultierenden Byte-Arrays wird hexadezimal kodiert. Bevor das Secret zurück an den anfragenden Client geschickt wird, werden die von der Abbruchsmethode erzeugten Header an die HTTP-Antwort angehängt.

In allen folgenden Anfragen muss das Session Secret als Legitimation hinterlegt werden. Hierfür muss der `Authorization`-Header mit jeder Anfrage gesetzt und gültig sein. Dafür wird das „Bearer“-Authentifizierungsschema verwendet. Angenommen der Broker generiert ein Session Secret `secretfoobar`, dann muss der Authentifizierungsheader wie folgt formatiert sein: `Authorization: Bearer secretfoobar`. Stellt der

[20] Siehe https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/security/SecureRandom.html

Broker während einer Anfrage fest, dass das angegebene Secret ungültig ist, so wird sie mit einem „Unauthorized"-Status zurückgewiesen.

Soll eine Session über den gleichen `/session`-Endpunkt beendet werden, so werden alle benutzerdefinierten Header, die mit `X-` beginnen, aus der Anfrage gelesen. Das Session Secret und die ermittelten Header werden an den Worker weitergegeben, welcher zuerst die Abbruchmethode der Session ermittelt. Ist diese nicht ereignisgesteuert, so wird die Anfrage sofort zurückgewiesen. Andernfalls werden die Header an die Abbruchsmethode übergeben, damit sie ihren internen Zustand aktualisieren kann. Ist nach diesem Schritt die Abbruchsbedingung eingetreten, so wird das Session Secret zur internen Liste hinzugefügt, welche vom Hintergrundprozess des Workers abgearbeitet wird um Sessions zu beenden.

Möchte ein Client an einer Session teilhaben, so muss der `/submit`-Endpunkt verwendet werden. Das in der Anfrage enthaltene Session Secret wird an den Worker weitergegeben. Der Worker ermittelt das Session-Objekt, erstellt ein neues Client Secret und gibt es an die Session weiter. Anschließend gibt der Worker das generierte Client Secret zurück.

Um Bitvektoren zu senden, verwendet der Client nun nicht mehr das Session Secret als Legitimation, sondern das erstellte Client Secret für den gleichen Endpunkt. Die Liste der übermittelten Bitvektoren, zusammen mit dem Client Secret, werden an den Worker übergeben. Der Worker versucht nun für das gegebene Client Secret die zugehörige Session zu ermitteln. Dafür wird über alle Session-Objekte iteriert und überprüft, ob eine Session existiert, die das Client Secret kennt. Existiert keine solche Session, so wird die Anfrage zurückgewiesen. Ansonsten werden die Bitvektoren in einem separaten Thread in den Cache eingefügt.

Hat mindestens ein weiterer Client in der gleiche Session bereits Bitvektoren gesendet, so wird zudem die Queue befüllt. Hierfür werden sukzessiv alle anderen Clients ermittelt, die ebenfalls an der Session teilnehmen. Aus den übermittelten Bitvektoren und den Bitvektoren der anderen Clients werden Paare gebildet. Diese werden daraufhin an die Queue übermittelt. Dieser Prozess ist in Abbildung 6.10 veranschaulicht.

Jeder im Besitz des Session Secrets kann den Fortschritt des Matchings über den `/progress`-Endpunkt abfragen. Das Secret wird an den Worker übergeben, welcher mit der Queue kommuniziert um deren aktuelle Länge l und die Anzahl aller jemals eingefügten Bitvektorpaare für die zugehörige Session l_0 zu ermitteln. Der Fortschritt p für das Matching lässt sich daraufhin folgendermaßen berechnen:

$$p(l) = 1 - \frac{l}{l_0}$$

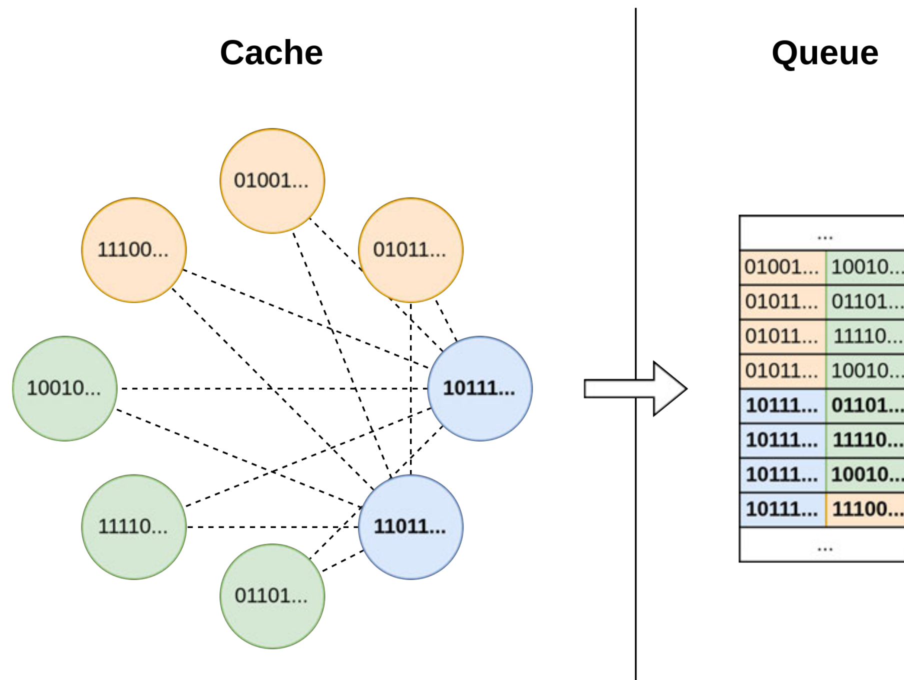


Abbildung 6.10: Erzeugung der Queue aus neu hinzugefügten (fett markierten) Bitvektoren. Die Links zwischen den Knoten im Cache stellen die paarweisen Vergleiche dar, die noch durchgeführt werden müssen. Unterschiedliche Farben zwischen den Knoten stellen die Clients dar, von denen die Bitvektoren zugesendet wurden.

Möchte ein Client die Ergebnisse des Matchings anfordern, so identifiziert er sich gegenüber dem Broker erneut mit dem entsprechenden Client Secret. Das Secret erhält der Worker, welches er in die zugehörige Match Session auflöst. Die Kombination aus Session und Client Secret wird verwendet um die eingetragenen Matches vom Cache anzufordern und an den Client zu senden.

7 PHT-Integration

Für die erfolgreiche Integration in die PHT-Infrastruktur werden noch einige Komponenten an den jeweiligen Stationen benötigt. Zudem muss noch eine Zugklasse entworfen werden, welche PPRL-Durchläufe in der PHT-Infrastruktur ermöglicht. In diesem Kapitel wird zuerst auf PIX-Systeme als Speicher für Patientendaten eingegangen und die daraus entstehende Notwendigkeit der Pseudonymisierung für Studienzwecke. Anschließend wird die Implementierung des Resolvers als weiterer Webservice und die PPRL-Zugklasse vorgestellt.

7.1 PIX-System und Pseudonymisierung

Für medizinische Studien müssen abrufbare Personendaten in einem PIX-System an der Station vorliegen. Jede Identität muss mit einem Master Patient Index (MPI) versehen sein, welche diese eindeutig identifiziert. MPIs sind langlebige Identifikatoren. In einem Patientendatenstamm, welcher frei von jeglichen Redundanzen ist, besitzt jeder Patient einen eindeutigen MPI, welcher sich nicht ändert. Die Herausgabe von MPIs für Studienzwecke ist somit aus der Datenschutzperspektive gravierend und nicht tragbar.

Pseudonym	MPI
study_483918485	11000000726
study_910023182	11000000727
study_339025714	11000000728

MPI	Vorname	Nachname	...
11000000726	Erica	Freund	...
11000000727	Christin	Kessler	...
11000000728	Enno	Steffens	...

Tabelle 7.1: PIX-Datenschema und zugeordnete Pseudonyme an einem Beispiel

Stattdessen muss für jede Person in einer Studie ein Pseudonym erzeugt werden. Pseudonyme sind kurzlebige Identifikatoren und können in beliebiger Menge erstellt und wieder gelöscht werden. Zwar fungieren sie nur stellvertretend für die eigentliche Identität und deren MPI, allerdings sind sie aufgrund ihrer Kurzlebigkeit keine dauerhaften Identifikatoren und können jederzeit ihre Gültigkeit verlieren. Werden generierte Pseudonyme für eine Datenanalyse, wie bei der PPRL, nach der Verwendung sofort gelöscht und nicht wiederverwendet, so bieten sie eine adäquaten Schutz vor der Wiedererkennung bestimmter Identitäten. Ein Beispiel für eine Zuordnung zwischen Identitäten und Pseudonymen ist in Tabelle 7.1 zu sehen.

Die eingesetzten PIX-Systeme in der echten Welt sind vielfältig. Eine umfassende Lösung, welche sich an alle möglichen PIX-Systeme anbinden und Patientendaten extrahieren kann, wäre wünschenswert, würde aber den Rahmen dieser Arbeit sprengen. Stattdessen werden im Rahmen dieser Arbeit die Produkte der MOSAIC-Suite der THS

Greifswald verwendet[21]. Genauer werden E-PIX als PIX-System und gPAS als Pseudonymisierungsdienst verwendet.

Beide Dienste sind Webanwendungen, die auf einem Server installiert werden können. In der Architektur des PHT werden E-PIX und gPAS als separate Dienste neben der Software an der Station bereitgestellt. Sie erfüllen mehrere Anforderungen, welche die Integration in den PHT und die Verwendung mit den bereits beschriebenen Diensten in der PPRL-Architektur vereinfachen. Beide Dienste sind quelloffen, verfügen über ein nutzerfreundliches Web-Interface, besitzen vordefinierte und klar dokumentierte Datenschemata und sind über SOAP-Schnittstellen abrufbar.

Datenbestände in E-PIX und gPAS werden in Domänen organisiert. Eine Domäne besitzt stets einen Namen und ist auf einer abstrakten Ebene als isolierte Studie oder Institution anzusehen. Allgemein gilt, dass eine Domäne redundanzfrei sein muss. Deswegen implementiert E-PIX eine lokale Record Linkage, um Dubletten in den Datensätzen einer Domäne zu identifizieren und gegebenenfalls zusammenzuführen.

7.2 Resolver

Um Pseudonyme aufzulösen und um aus den daraus abgeleiteten personenbezogenen Daten Bitvektoren zu erzeugen, wird ein weiterer Dienst benötigt: der Resolver. Der Resolver ist die einzige Komponente, mit welcher der PPRL-Zug kommuniziert. Deswegen muss es dem Resolver neben dem Auflösen von Pseudonymen möglich sein, die Ergebnisse für eine Station abzufragen und diese pseudonymisiert dem Zug bereitzustellen.

Der Resolver ist, entgegen den bisher beschriebenen Diensten, in Python geschrieben. Die Hauptgründe hierfür sind, dass der Resolver ein leichtgewichtiger Dienst ist und die Python-Bibliothek *Zeep*[22] eine einfache Verwendung von SOAP-APIs ermöglicht.

Der Resolver benötigt zum Start URLs zu den Diensten, von denen er abhängig ist. Dazu zählen E-PIX, gPAS und der Encoder, welche alle neben dem Resolver an der Station gehostet werden. Weiterhin muss eine URL zu einem Broker hinterlegt werden. Die Konfiguration dieser URLs geschieht über Umgebungsvariablen.

Die beiden Hauptaufgaben des Resolvers, also das Auflösen von Pseudonymen und Bereitstellen von Ergebnissen, sind in Abbildung 7.1 dargestellt. Das Diagramm enthält alle Kommunikationswege zwischen Resolver, Broker, PPRL-Zug und den Diensten, die an der Station vorhanden sind. Die einzelnen Schritte werden auf den folgenden Seiten beschrieben.

[21] Siehe `https://www.ths-greifswald.de/projekte/mosaic-projekt/`
[22] Siehe `https://docs.python-zeep.org/en/master/`

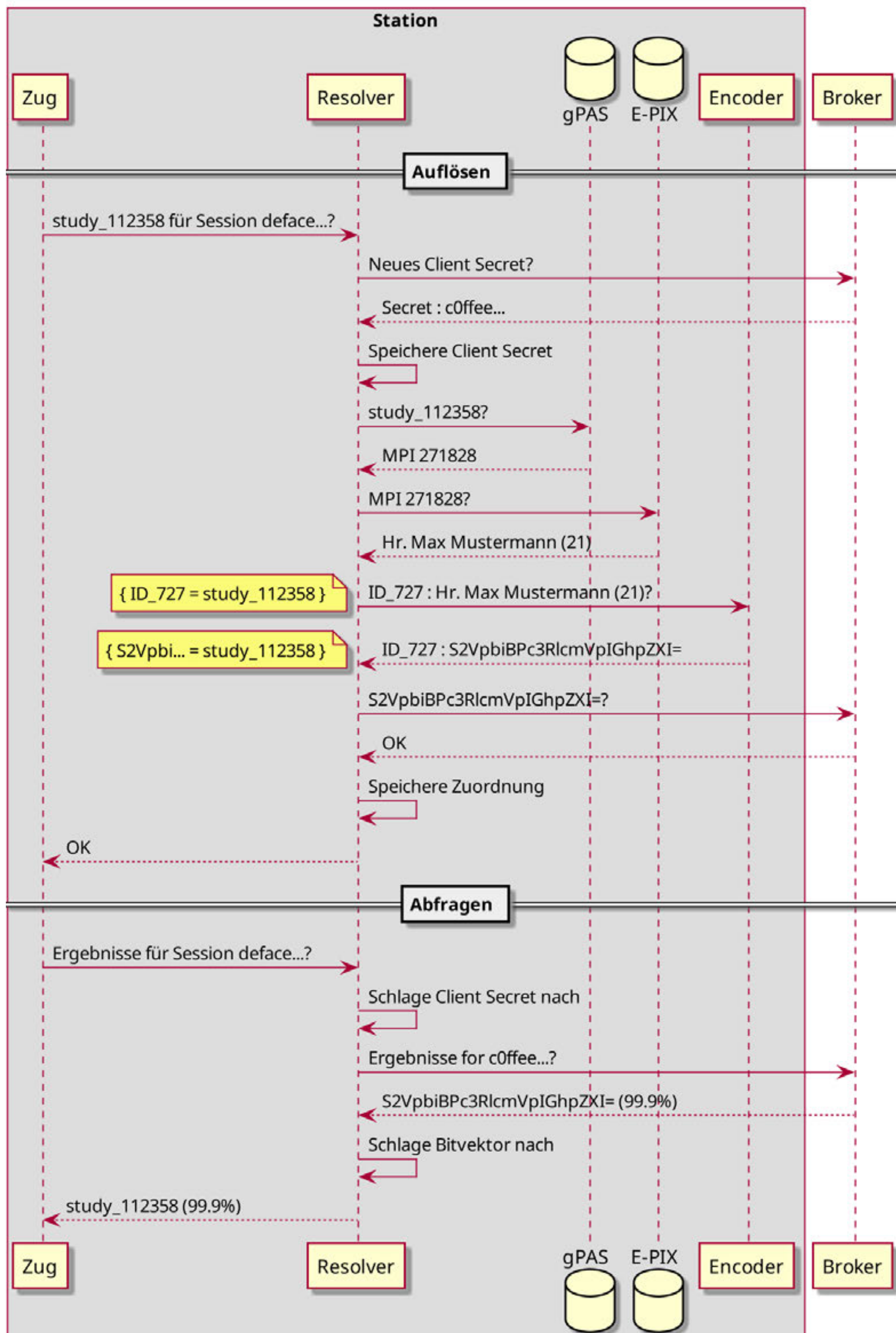


Abbildung 7.1: Kommunikationswege zwischen PPRL-Zug, Resolver und Broker anhand eines Beispiels. In der ersten Phase erhält der Resolver Pseudonyme, welche er auflöst, mithilfe des Encoders maskiert und an den Broker sendet. In der zweiten Phase fragt der Resolver Ergebnisse vom Broker ab. Währenddessen ist der Resolver vollständig für die Verwaltung des Client Secrets für die Session verantwortlich.

Auflösen von Pseudonymen Der Resolver nimmt am Wurzelpfad Anfragen an. Um Pseudonyme an den Resolver zu übermitteln, muss eine POST-Anfrage mit dem Aufbau in Listing 7.1 gesendet werden. Die Anfrage enthält eine Konfiguration für den lokalen Encoder-Service und eine Liste von aufzulösenden Pseudonymen. Weiterhin sind domänenspezifische Parameter enthalten, darunter der Name der Domäne, wie sie in E-PIX und gPAS vorhanden ist, und die Liste der Attribute, die für jeden aufgelösten Datensatz für das Matching einbezogen werden sollen. Eine Liste von gültigen Attributnamen kann über den `/attributes`-Endpunkt des Resolvers mit einer GET-Anfrage ermittelt werden.

```
{
  "client-config": {
    "domain": "study",
    "attributes": ["first_name", "last_name", "gender",
      "birth_date"]
  },
  "encoder-config": {
    "charset": "UTF-8",
    "seed": "s3crets33d",
    "generation-function":
       "BIGRAM_ATTRIBUTE_BLOOM_FILTER"
  },
  "pseudonyms": ["study_483918485", "study_910023182",
    "study_339025714"]
}
```

Listing 7.1: Beispiel einer PPRL-Resolver Anfrage

Jede Anfrage muss authentifiziert sein. Wie beim Broker muss ein `Authorization`-Header in der Anfrage vorhanden sein, welcher das Secret der am Broker registrierten Session enthält. Zuerst wird mit dem erhaltenen Secret ein neues Client Secret am Broker generiert. Das erhaltene Client Secret wird lokal in einem temporären Verzeichnis gespeichert.

Jedes Pseudonym wird nun sukzessiv mit gPAS aufgelöst und der erhaltene MPI in E-PIX nachgeschlagen, um die personenbezogenen Daten zu erhalten. Dafür werden die SOAP-Schnittstellen der beiden Dienste verwendet. gPAS stellt hierfür die Funktion `getValueFor` bereit, welche den originalen Wert für ein Pseudonym zurückgibt. Mit einem MPI wird in E-PIX die Funktion `getPersonsByMPI` verwendet, um den Datensatz, welcher mit dem gegebenen MPI verknüpft ist, zu ermitteln. Schlägt eine der beiden Funktionen fehl, also kann entweder das Pseudonym nicht aufgelöst oder der MPI nicht nachgeschlagen werden, so wird das Pseudonym übersprungen.

Währenddessen wird eine Anfrage an den Encoder vorbereitet. Der Encoder benötigt eine Liste von Entitäten, welche maskiert werden sollen. Die Entitäten setzen sich aus

den aufgelösten Datensätzen mit der Auswahl der in der Anfrage enthaltenen Attribute zusammen. Zudem erhält jede Entität einen zufällig generierten Identifikator. Auf dieser Basis erstellt der Resolver eine Zuordnung von Pseudonymen auf Identifikatoren für den Encoder.

Diese Zuordnung wird benötigt, sobald der Encoder eine Antwort zurückgibt. Jeder generierte Bitvektor ist mit dem Identifikator aus der ursprünglichen Anfrage versehen. Somit kann der Resolver eine weitere Zuordnung von Bitvektoren zu Pseudonymen erstellen. Bevor diese Zuordnung lokal gespeichert werden kann, werden die generierten Bitvektoren unter Verwendung des anfangs generierten Client Secrets an den Broker gesendet. Erst wenn der Broker den Empfang der Bitvektoren bestätigt, wird die Zuordnung zusammen mit dem Client Secret gespeichert. Damit ist die „Auflösungsphase“ des Resolvers abgeschlossen und die Anfrage abgearbeitet.

Abrufen von Ergebnissen Bei einer authentifizierten GET-Anfrage am Wurzelpfad lädt der Resolver die Match-Ergebnisse. Zuerst wird überprüft, ob für das übergebene Session Secret bereits lokal ein Client Secret und dementsprechend eine Bitvektor-Zuordnung hinterlegt ist. Ist das der Fall, so werden die Ergebnisse vom Broker unter Verwendung des lokal gespeicherten Client Secrets abgerufen.

Der Broker gibt jedoch nur eine Liste von Bitvektoren zurück, die vom Resolver zugesendet wurden. Diese Bitvektoren werden mithilfe der lokal gespeicherten Zuordnung von Bitvektoren zu Pseudonymen aufgelöst. Der Resolver sendet anschließend eine Antwort mit einem identischen Aufbau wie die Ergebnis-Antwort des Brokers. Statt der Bitvektoren werden jedoch die Pseudonyme eingesetzt, die in der ursprünglichen Anfrage an den Resolver für die gegebene Session übermittelt wurden.

7.3 Deployment der Station-Services

Für eine einfache Einrichtung der notwendigen Dienste, die an einer Station verfügbar sein müssen, wird eine Docker Compose Datei zur Verfügung gestellt. Docker Compose[23] ermöglicht die Definition mehrerer Dienste in einer Datei und deren Ausführung mit einem Kommando. So wird die Konfiguration komplexer Dienste und deren Abhängigkeiten stark vereinfacht. Die Datei umfasst Service-Definitionen für E-PIX, gPAS, MySQL — welches von E-PIX und gPAS benötigt wird —, den Encoder- und den Resolver-Service. Alle Services werden auf dem Port 8080 gehostet und eingehende Anfragen werden von einem Reverse Proxy an die zuständigen Dienste weitergeleitet.

Anfragen auf dem `/epix`-Pfad werden an E-PIX weitergeleitet. Analog werden Anfragen auf dem `/gpas`-Pfad an gPAS übergeben. Eine Besonderheit ist der `/static`-Pfad. Ne-

[23] Siehe `https://docs.docker.com/compose/`

ben der Docker Compose Datei existiert ein leeres Verzeichnis mit dem Namen „static“, in dem Dateien abgelegt werden können. Dieses Verzeichnis wird in den Container des Reverse Proxys eingebunden, sodass diese Dateien über den `/static`-Pfad erreichbar sind. Das Verzeichnis dient dem Ablegen einer Datei, in der eine Liste aller an der Station verfügbaren Pseudonyme enthalten ist. Somit wird der Zugriff auf die Pseudonyme für den PPRL-Zug garantiert. Alle anderen Anfragen werden an den Resolver weitergeleitet und gegebenenfalls abgewiesen, wenn der Resolver sie nicht bearbeiten kann.

7.4 Zugklasse

Das Skript für den PPRL-Zug basiert auf einer Python-Bibliothek, welche speziell für die Interaktion mit den in dieser Arbeit beschriebenen Diensten entwickelt wurde. Die Endpunkte jedes Dienstes werden in der Bibliothek *pprl*[24] als einfache in Python integrierbare Funktionen zur Verfügung gestellt. Für jeden Dienst existiert eine Klasse, welche als einfacher HTTP-Client auf Basis der Bibliothek *requests*[25] fungiert. Das PPRL-Skript, welches im Zug ausgeführt wird, verwendet lediglich die Client-Klasse für den Resolver.

Das *Dockerfile* für das Zugimage spezifiziert folgende Umgebungsvariablen, die vom Stationsbetreiber vor der Ausführung des Zuges festgelegt werden müssen:

- Session Secret: Secret für die Match Session
- Resolver URL: URL, unter welcher der Reverse Proxy an der Station auf Anfragen wartet
- Domäne: Datendomäne, unter welcher in E-PIX und gPAS jeweils die personenbezogenen Daten und Pseudonyme gespeichert sind
- Ausführungsphase: entweder erste oder zweite Phase
- Pseudonym-URL: URL, unter welcher die Liste von Pseudonymen an der Station erreichbar ist

Der Zug enthält eine Konfigurationsdatei, welche für jeden PPRL-Durchlauf angepasst werden muss. Ein Beispiel für eine solche Konfiguration ist in Listing 7.2 zu sehen. Die Konfiguration umfasst die Einstellungen für den Encoder und die Liste von Attributen, die an jeder Station an den Resolver weitergegeben wird.

Das Skript, welches im Zug ausgeführt wird, überprüft zuerst, ob alle geforderten Umgebungsvariablen gesetzt und gültig sind. Die Variable für die Phase des Zuges kann zwei Ausprägungen annehmen: „submit“ und „result“.

[24] Siehe https://pypi.org/project/pprl/
[25] Siehe https://pypi.org/project/requests/

```
{
  "encoderConfig": {
    "charset": "UTF-8",
    "seed": "cc91b3439e04f4ce545c78d7dc6f940e",
    "generationFunction":
       "TRIGRAM_ATTRIBUTE_BLOOM_FILTER"
  },
  "attributes": ["first_name", "last_name", "gender",
     "birth_date"]
}
```

Listing 7.2: Beispiel einer PPRL-Zug-Konfiguraiton

Ist die Phase auf „submit“ gesetzt, so wird zuerst die Datei mit der Liste von Pseudonymen von der angegebenen URL geladen. Anschließend wird eine neue POST-Anfrage an den angegebenen Resolver gestellt. Die Anfrage setzt sich aus den Pseudonymen, der Zugkonfiguration und den Umgebungsvariablen zusammen. Ist die Phase hingegen auf „result“ gesetzt, so wird eine GET-Anfrage an den Resolver geschickt. Die Liste aller gefundenen Matches an der Station werden einer CSV-Datei mit dem Namen „results.csv“ angehängt.

Der Zug ist so ausgelegt, dass alle Variablen für einen Durchlauf identisch bleiben bis auf die Phase, welche zwischen der ersten und zweiten Ausführung angepasst werden muss. Das Endergebnis nach dem zweiten Zugdurchlauf ist eine Liste von Pseudonymen zusammen mit den zugeordneten Wahrscheinlichkeiten eines Matches.

8 Ergebnisse

In diesem Kapitel wird zuerst ein Blick auf die Qualität des Matchings anhand des Encoders und des Matchings geworfen. Daraufhin wird die Funktionsweise des Brokers an zwei Fallbeispielen demonstriert, eines davon angewendet auf den PHT. Abschließend wird die Performance des Brokers und der zentralen Match-Funktionen beleuchtet, um zu ermitteln, ob der Broker großen Datenmengen standhält.

8.1 Qualität des Matchings

Der Broker verarbeitet Ergebnisse vom Matcher und Encoder. Die Qualität des Matchings hängt dementsprechend von diesen beiden Diensten ab. In diesem Abschnitt werden die berechneten Ähnlichkeiten des Matchers näher untersucht, wenn Datensätze bestimmte Fehler zueinander aufweisen.

Für die Durchführung wird eine CSV-Datei mit 10 000 Datensätzen erstellt. Jeder Datensatz umfasst einen zufällig generierten Vor- und Nachnamen, Geschlecht, Geburtstag, Straße, Stadt und Postleitzahl. Die Daten werden mithilfe der Python-Bibliothek *Faker*[26] generiert. Auf Basis dieser CSV-Datei wird eine zweite Datei erzeugt. Für jeden Datensatz tritt durch Zufall eines der folgenden Szenarien ein:

- Der Datensatz wird unverändert kopiert.
- Ein Attribut des Datensatzes wird minimal geändert, das heißt es wird entweder ein Zeichen hinzugefügt, gelöscht oder ausgetauscht.
- Drei Attribute des Datensatzes werden durch neue Werte ersetzt.
- Der Datensatz wird verworfen und ein neuer zufälliger Datensatz wird generiert.

Die Datensätze werden mithilfe des Encoders maskiert. Alle Attribute werden als Strings interpretiert bis auf das Geburtsdatum und das Geschlecht, welche jeweils im Schema-Feld der Encoder-Anfrage als Datumsangabe und als einzelnes Zeichen deklariert werden. In zwei Durchläufen werden die Bloomfilter einmal mit Bigrammen, einmal mit Trigrammen besetzt — also q-Grammen der Länge zwei und drei. Anschließend werden die Bitvektoren paarweise an den Matcher gesendet ohne Begrenzung durch einen Schwellenwert.

Die Ergebnisse des Matchers werden anschließend mit der Art der Fehler zwischen den Datensätzen, welche die jeweiligen Bitvektorpaare repräsentieren, zusammengeführt. Als Match werden alle Paare eingestuft, deren Datensätze zueinander entweder

[26] Siehe `https://faker.readthedocs.io/en/master/`

identisch sind oder einen einzigen Fehler aufweisen. Alle restlichen Paare werden nicht als Matches kategorisiert.

Abbildung 8.1 zeigt eine Reihe statistischer Maße für das Ergebnis des Matchings bei unterschiedlichen Schwellenwerten. Abgebildet sind die Genauigkeit, die Sensitivität und der F1-Score. Bei einem Schwellenwert von 100 % beträgt die Sensitivität etwa 50 %, da alle Datensätze, in denen mehrere oder alle Attribute ausgetauscht wurden, korrekterweise als nicht übereinstimmend klassifiziert werden. Andererseits beträgt die Genauigkeit bei einem Schwellenwert von 0 % zirka 50 %, da ebendiese Datensätze nun fälschlicherweise als Matches klassifiziert werden.

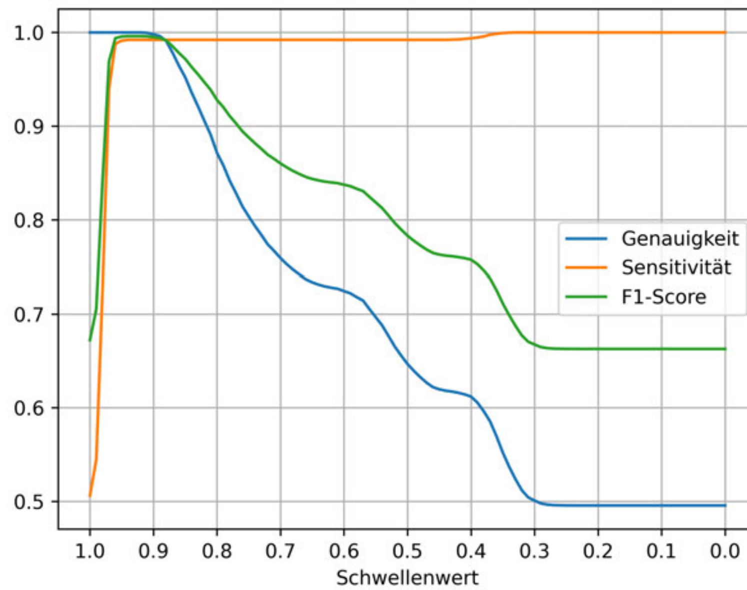
Es ist zu erkennen, dass sowohl für Bi- als auch Trigramme der F1-Score für Schwellenwerte im Bereich von 90 % bis zirka 95 % sehr stabil bleibt und nahezu 100 % beträgt. Bei Schwellenwerten höher als 95 % nimmt der F1-Score stark ab. Begründet ist das in der Tatsache, dass Datensätze, welche kleine Fehler zwischeneinander aufweisen, bei höheren Schwellenwerten nicht mehr als Matches klassifiziert werden.

Ab einem Schwellenwert von 90 % und abwärts fällt der F1-Score erneut. Nun werden Datensätze, welche keine signifikanten Ähnlichkeiten zueinander aufweisen, als Matches klassifiziert. Das zeigt sich in der stark fallenden Genauigkeitskurve, welche wie die Kurve für den F1-Score bei der Verwendung von Trigrammen in der Maskierung weniger Hügel aufwirft als bei Bigrammen.

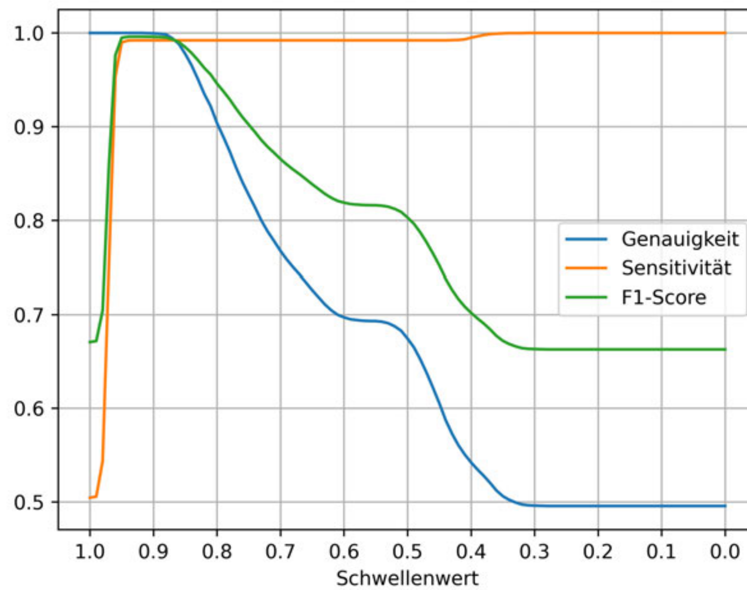
Eine mögliche Erklärung für diese Beobachtung liegt in der statistischen Verteilung von q-Grammen. Der Zeichenvorrat des Encoders, aus dem q-Gramme bestehen, enthält 37 einzigartige Zeichen: 26 Buchstaben, 10 Ziffern und ein Leerzeichen. Es existieren demzufolge 1369 einzigartige Bigramme und 50 653 Trigramme. Diese sind jedoch nicht gleichverteilt und sind von der Sprache, auf die sie angewendet werden, abhängig. Bei einer größeren Menge an q-Grammen ist zu erwarten, dass sich die Häufigkeitsverteilung stärker ausbreitet.

Es ist möglich, dass in Abbildung 8.1a einige wenige Bigramme die Mehrzahl aller auftretenden Textfragmente repräsentieren. Das würde die starken Abfälle im Bereich von 60 % und 40 % erklären, wenn die Bitmuster für diese q-Gramme nicht mehr miteinander gematcht werden. In Abbildung 8.1b ist davon auszugehen, dass weniger Trigramme existieren, welche sehr häufig auftauchen. Dementsprechend verlaufen die Kurven für die Genauigkeit und den F1-Score glatter bevor sie ab 50 % einbrechen, wenn verschiedene Datensätze als Matches klassifiziert werden.

Zusammenfassend ist für die generierten Daten ein Schwellenwert im Bereich von etwa 90 % bis 95 % angemessen. Dieser Schwellenwert kann bei größeren q-Grammen abgesenkt werden. Jedoch sinkt die Genauigkeit des Matchers bei tieferen Schwellenwerten stark ab.



(a) Aufteilung in Bigramme (q=2)



(b) Aufteilung in Trigramme (q=3)

Abbildung 8.1: Genauigkeit, Sensitivität und F1-Score in Abhängigkeit von Schwellenwerten. Es wurden 10 000 Datensätze mit variierenden unterschiedlichen Merkmalen abgeglichen. Wahre Übereinstimmungen durften nur minimale Änderungen zueinander aufweisen.

8.2 Testfall: 5. MII-Projektathon „Private Set Intersection“

Im Rahmen des 5. MII-Projektathons wurde ein Beispielszenario für den Anwendungsfall der „Private Set Intersection“, also der „privaten Schnittmenge“, vorgestellt. Die Testdaten bestehen aus drei CSV-Dateien, welche jeweils einem Teilnehmer zugewiesen werden. Jede Datei enthält 100 synthetisch generierte Patientendatensätze. Unter den drei Dateien existieren Dubletten, welche es zu ermitteln gilt. In diesem Abschnitt sollen die Beispieldaten mit der in dieser Arbeit entwickelten PPRL-Infrastruktur mithilfe des PHT auf Dubletten untersucht werden.

Analyse des Datenschemas Die Beispieldaten besitzen eine Vielzahl an Attributen für jeden Datensatz. Von denen lassen sich allerdings nur eine Auswahl direkt in das Datenschema des E-PIX übertragen. Darunter fallen der Vorname, Nachname, Geburtsname, Geschlecht, Geburtsdatum, Wohnort und Postleitzahl. Eine gesonderte Konfiguration muss vorgenommen werden für die restlichen Attribute. Diese umfassen einen Wahrheitswert, ob ein Patient gestorben ist, den Todeszeitpunkt, den letzten Lebenszeitpunkt, der Name der Krankenversicherung und die Versichertennummer.

Um diese Attribute im E-PIX abzubilden, muss die Konfiguration der Studiendomäne angepasst werden. Die geänderten Felder der Konfiguration sind in Listing 8.1 zu sehen. Die Liste der notwendigen Felder wird um alle Felder erweitert, die laut der E-PIX-Dokumentation als erforderlich gekennzeichnet werden dürfen. Weiterhin führt E-PIX in seinem Datenschema 10 Attribute, welche frei belegt werden können. In der Konfiguration wird ein Teil dieser Felder verwendet, um die Attribute entsprechend des Datenschemas aus den Beispieldaten anzupassen.

Vorbereitung des Durchlaufes An dem Testlauf nehmen drei Stationsbetreiber teil. Neben ihren Stationen setzen sie den Resolver, den Encoder, E-PIX und gPAS mithilfe der mitgelieferten Docker Compose Datei an einem unbesetzten Port auf.

Jeder Stationsbetreiber erhält eine der drei CSV-Dateien. Zuerst erstellt jeder Teilnehmer im E-PIX eine neue Domäne namens „smpc_psi“ mit der angepassten Konfiguration. Über den Menüpunkt „Import“ wird dann die zugewiesene CSV-Datei hochgeladen. Vor dem eigentlichen Import müssen die Spalten der CSV-Datei dem Datenschema der Domäne zugewiesen werden. Diese Zuordnung ist in Abbildung 8.2 zu sehen. Die Felder „PatientKontext“, „PatientIdentifikator“ und „Informationsquelle“ werden nicht importiert, da diese sich auf die fiktive Datenquelle beziehen, aus der die Beispieldaten stammen. Nach dem erfolgreichen Import kann über den Menüpunkt „Export“ die Liste der generierten MPIs für alle importierten Patienten heruntergeladen werden.

Abbildung 8.2: Import der MII-Beispieldaten in E-PIX. Die Spalten aus der CSV-Datei werden den korrespondierenden Spalten im E-PIX-Datenschema zugeordnet.

Abbildung 8.3: Import der generierten MPIs in gPAS. Für eine Liste von MPIs wird mit diesen Einstellungen eine Liste neuer Pseudonyme erstellt.

```xml
<ma:MatchingConfiguration>
  <!--
    Nicht gelistete Felder entstammen der mitgelieferten
       Beispielkonfiguration des E-PIX.
  -->
  <required-fields>
    <name>firstName</name>
    <name>lastName</name>
    <name>mothersMaidenName</name>
    <name>gender</name>
    <name>birthDate</name>
    <name>city</name>
    <name>zipCode</name>
  </required-fields>
  <value-fields-mapping>
    <value1>Patient verstorben</value1>
    <value2>KV-Name</value2>
    <value3>KV-Nummer</value3>
    <value4>Todeszeitpunkt</value4>
    <value5>Letzter lebend Zeitpunkt</value5>
    <value8>Bloomfilter</value8>
  </value-fields-mapping>
</ma:MatchingConfiguration>
```

Listing 8.1: E-PIX Domänenkonfiguration für Private Set Intersection

Anschließend wird zu gPAS gewechselt. Wieder wird eine neue Domäne mit dem gleichen Namen „smpc_psi“ erstellt. Jeder Teilnehmer entscheidet sich für ein Präfix, welches die eigene Station repräsentiert. Somit werden Kollisionen in den generierten Pseudonymen zwischen den einzelnen Stationen vermieden. Über den Menüpunkt „Verarbeiten“ kann die von E-PIX generierte CSV-Datei hochgeladen werden. Nach der Auswahl der MPI-Spalte muss spezifiziert werden, wie die Pseudonyme generiert werden sollen. Die Importoptionen sind in Abbildung 8.3 zu sehen. Die Liste der generierten Pseudonyme wird dann über den Menüpunkt „Export“ heruntergeladen. Anschließend entfernen die Betreiber der Stationen die MPI-Spalte aus dem gPAS-Export und speichern die Liste der Pseudonyme im Verzeichnis für statische Dateien des an der Station laufenden Reverse Proxys.

Ausführung des Durchlaufes Haben alle Stationsbetreiber ihre Vorbereitungen beendet, so kann der Datenanalyst den PPRL-Zug vorbereiten. Die vollständige Konfigurationsdatei, welche dem PPRL-Zug beigefügt wird, ist in Listing 8.2 aufgeführt. Der Seed-Wert ist frei wählbar und nicht für den Erfolg des Beispielszenarios ausschlaggebend. Die Liste der Attributnamen entspricht den Spaltennamen im E-PIX-Datenschema, welche in der Domänenkonfiguration spezifiziert wurden.

```
{
  "encoderConfig": {
    "charset": "UTF-8",
    "seed": "1d3d290ff93dde4fc03072e9c891ab30",
    "generationFunction":
      "TRIGRAM_ATTRIBUTE_BLOOM_FILTER"
  },
  "attributes": ["first_name", "last_name", "gender",
    "birth_date", "mothers_maiden_name",
    "contact.city", "contact.zip_code", "value1",
    "value2", "value3", "value4", "value5"]
}
```

Listing 8.2: Konfiguration des PPRL-Zuges für die MII-Beispieldaten

Als nächstes erstellt der Datenanalyst eine neue Session am Broker. Das Session Secret wird über einen sicheren Kommunikationskanal an die Betreiber der teilnehmenden Stationen zugesendet. Anschließend generiert der Datenanalyst einen neuen Zug auf Basis der angepassten Zugklasse und wählt die teilnehmenden Stationen als Ziele aus. Die Stationsbetreiber füllen im Durchlauf die Variablen für jeden Zug aus, abhängig vom Port, an dem die zusätzliche Software gehostet wird. Wichtig ist, dass die Variable für die Zug-Phase im ersten Durchlauf auf „submit“, im zweiten Durchlauf auf „result“ gesetzt ist. An jeder Station gibt der PPRL-Zug eine Liste von gematchten Pseudonymen zusammen mit den berechneten Konfidenzen aus.

Ergebnisse Der beschriebene Ablauf wurde mit Stationen an der Hochschule Mittweida, dem Universitätsklinikum Leipzig und der RWTH Aachen durchgeführt. Der Broker wurde an der Hochschule Mittweida gehostet. Für die Session wurde ein Schwellenwert von 90 % festgelegt. Die reine Ausgabe des Zuges nach dem zweiten Durchlauf ist im Anhang C beigefügt.

Da die Pseudonyme an sich keine direkte Aussage über den Erfolg des Durchlaufs ermöglichen, wurde eine weitere Zugklasse erstellt. Diesem Zug wird die Liste aller Pseudonyme übergeben, für die Übereinstimmungen gefunden wurden. Die Aufgabe des Zuges an jeder Station ist das Auflösen der mitgelieferten Pseudonyme in Patientendaten. Diese Daten werden anschließend in einer Datei als Artefakt im Zug gespeichert. Offensichtlich wird ein solcher Zug nie in einem echten Szenario zur Anwendung kommen, da das Speichern von personenbezogenen Daten für den Analysten nicht datenschutzkonform ist. Deswegen dient dieser Zug ausschließlich dem Zweck der Auswertung von Beispieldaten. Die aufbereiteten Ergebnisse dieses finalen Zugdurchlaufs sind in Abbildung 8.4 zu sehen. Übereinstimmende Datensätze sind farblich kodiert. Falsch positive Übereinstimmungen sind kursiv gekennzeichnet.

set	match	pseudonym	first_name	last_name	gender	birth_date	mothers_ma…	city	zip_code	dead?	insur_name	insur_num	death_time	alive_time	conf
B	C	hsmw_85498055…	Adele	Roberto	F	1984-12-01	Roberto	Naranjo	85161	nein	BKK_800	8002001650		26.09.20	1,00
B	C	hsmw_60445899…	Reserves	Delucia	M	2019-04-01	Delucia	Tomado	13864	nein	TKK_900	9002015332		08.01.20	1,00
B	C	hsmw_30094915…	Berenice	Roberto	F	1949-04-16	Roberto	Tomado	13860	nein	TKK_900	9002005289		14.09.20	1,00
B	C	hsmw_98458444…	Oerjan	Bramhall	F	1985-07-01	Bramhall	Lamakera Dua	40666	nein	AOK_110	1102005343		06.11.20	1,00
B	CC	hsmw_04554136…	Bjoern	Twinborough	F	1979-01-16	Huburn	Kaustinen	23765	nein	AOK_100	1002018109		02.10.20	0,91
B	AC	hsmw_28626216…	Mailys	Sivier	M	1992-01-16	Blasli	Kaustinen	23768	nein	AOK_100	1001012397		30.07.20	1,00
B	A	hsmw_89384439…	Cloe	Silverman	F	1985-07-01	Silverman	Ya bad	43766	nein	BAR_400	4001016480		24.12.20	1,00
B	A	hsmw_36774099…	Gerald	Pinock	F	1940-04-16	Anthiftle	Tomado	13865	nein	TKK_900	9001003186		31.01.20	1,00
B	A	hsmw_65567166…	Leane	Strickett	F	1963-10-01	Strickett	Dukuhsia	63766	nein	AOK_600	6001001771		19.09.20	1,00
A	BC	rwth_825863076	Mailys	Sivier	M	1992-01-16	Blasli	Kaustinen	23768	nein	AOK_100	1001012397		30.07.20	1,00
A	B	rwth_084669279	Cloe	Silverman	F	1985-07-01	Silverman	Ya bad	43766	nein	BAR_400	4001016480		24.12.20	1,00
A	B	rwth_703478825	Gerald	Pinock	F	1940-04-16	Anthiftle	Tomado	13865	nein	TKK_900	9001003186		31.01.20	1,00
A	B	rwth_836575770	Leane	Strickett	F	1963-10-01	Strickett	Dukuhsia	63766	nein	AOK_600	6001001771		19.09.20	1,00
A	C	rwth_151573877	Maiwenn	Duinkerk	M	2002-04-16	Twinborough	Hougong	13762	nein	TKK_900	9001001625		10.07.20	1,00
A	C	rwth_960890472	Wa	Kamall	M	1961-04-01	Kamall	Hougong	13761	nein	TKK_900	9001003133		17.03.20	1,00
A	C	rwth_332094188	Maerta	Duinkerk	F	1963-01-16	Twinborough	Kaustinen	23762	nein	AOK_100	1001013797		05.11.20	1,00
A	C	rwth_499344120	Anae	Saltsberger	M	1952-01-16	Holtum	Kaustinen	23768	nein	AOK_100	1001008185		22.04.20	1,00
C	B	UKL_029173425	Adele	Roberto	F	1984-12-01	Roberto	Naranjo	85161	nein	BKK_800	8002001650		26.09.20	1,00
C	B	UKL_802864209	Reserves	Delucia	M	2019-04-01	Delucia	Tomado	13864	nein	TKK_900	9002015332		08.01.20	1,00
C	B	UKL_688867753	Berenice	Roberto	F	1949-04-16	Roberto	Tomado	13860	nein	TKK_900	9002005289		14.09.20	1,00
C	B	UKL_759210331	Oerjan	Bramhall	F	1985-07-01	Bramhall	Lamakera Dua	40666	nein	AOK_110	1102005343		06.11.20	1,00
C	B	UKL_680188676	Bjoern	Twinborough	F	1979-01-16	Huburn	Kaustinen	23765	nein	AOK_100	1002018109		02.10.20	1,00
C	AB	UKL_814426361	Mailys	Sivier	M	1992-01-16	Blasli	Kaustinen	23768	nein	AOK_100	1001012397		30.07.20	1,00
C	*A*	*UKL_018574329*	*Lie*	*Twinborough*	*F*	*1971-01-16*	*Twinborough*	*Kaustinen*	*23764*	*nein*	*AOK_100*	*1003000356*		*10.09.20*	*0,91*
C	C	UKL_329331423	Maiwenn	Duinkerk	M	2002-04-16	Twinborough	Hougong	13762	nein	TKK_900	9001001625		10.07.20	1,00
C	C	UKL_788144199	Wa	Kamall	M	1961-04-01	Kamall	Hougong	13761	nein	TKK_900	9001003133		17.03.20	1,00
C	C	UKL_882465929	Maerta	Duinkerk	F	1963-01-16	Twinborough	Kaustinen	23762	nein	AOK_100	1001013797		05.11.20	1,00
C	C	UKL_746146882	Anae	Saltsberger	M	1952-01-16	Holtum	Kaustinen	23768	nein	AOK_100	1001008185		22.04.20	1,00

Abbildung 8.4: Ergebnisse des PPRL-Durchlaufs auf Basis der MII-Beispieldaten. Erkannte Übereinstimmungen sind farblich hinterlegt. Falsch positive Ergebnisse sind kursiv.

Bis auf ein falsch positives Ergebnis konnten alle erwarteten Übereinstimmungen identifiziert werden. Dieses Beispiel demonstriert somit klar die erfolgreiche Integration der PPRL-Infrastruktur in die Aachener PHT-Architektur.

8.3 Testfall: Synthetische Beispieldaten

Ein weiteres Beispiel soll die korrekte Funktionsweise der Dienste anhand synthetischer Daten mit manuell eingefügten typographischen Fehlern demonstrieren. Encoder, Matcher und Broker werden auf dem gleichen Computer gehostet. An dem Beispiel beteiligen sich drei Clients, welche alle über ein Python-Skript erstellt und gesteuert werden.

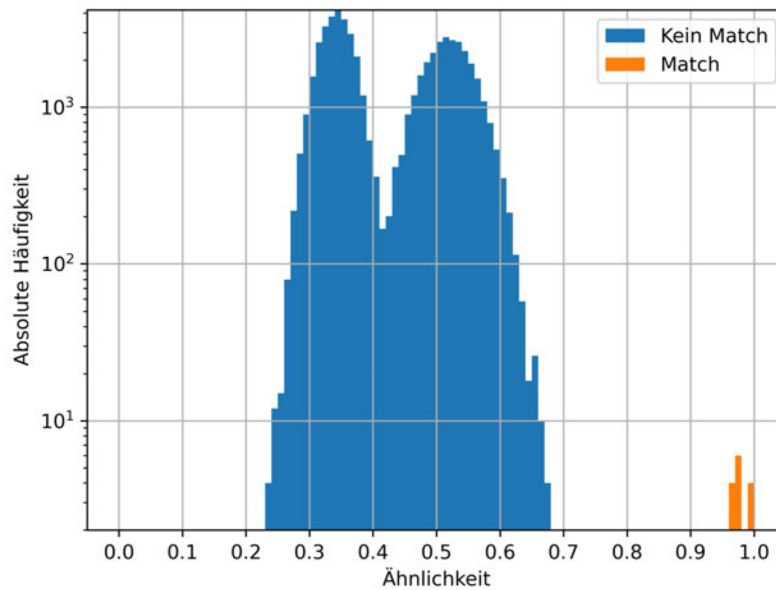
Generierung der Daten Für die Generierung von Beispieldaten wird erneut die Python-Bibliothek *Faker* verwendet. Jeder Datensatz umfasst die Eigenschaften Vorname, Nachname, Geschlecht, Geburtsdatum, Straße, Stadt und Postleitzahl. Zu Beginn werden drei Datensätze zufällig generiert. Für jeden Client werden daraufhin zusätzliche Datensätze generiert, sodass die Datenbasis jedes Clients insgesamt 1000 Datensätze umfasst. Das bedeutet, dass nach diesem Schritt alle drei Clients jeweils drei Datensätze besitzen, die alle zueinander identisch sind.

Anschließend werden manuelle Änderungen an den drei identischen Datensätzen in den Datenbasen der drei Clients vorgenommen. Die Art der Fehler umfasst das Löschen, Einfügen und Austauschen einzelner Zeichen. Diese Fehler sind jedoch nicht zufällig, sondern lassen den geänderten Attributwert ohne weiteres Vorwissen noch plausibel erscheinen. Die eingetragenen Fehler am durchgeführten Beispiel sind in Tabelle 8.1 zusammengefasst.

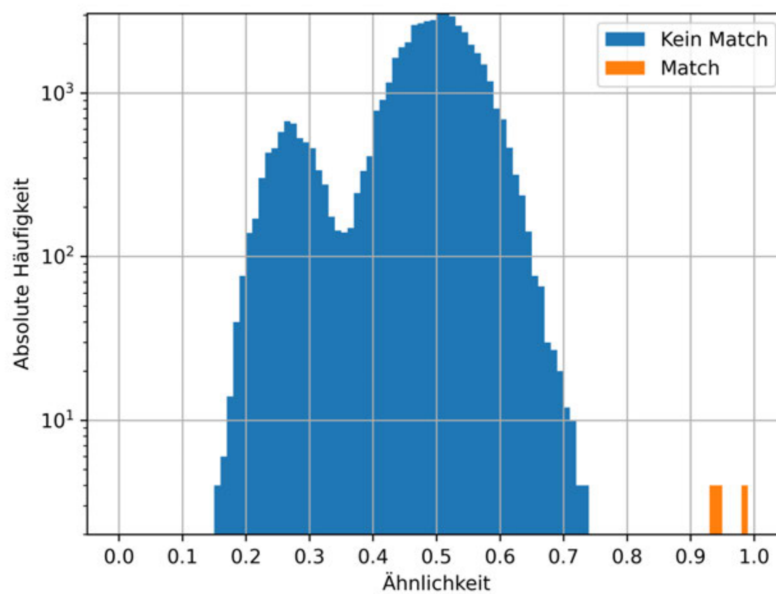
Datensatz	Attribut	Originalwert	Änderung	Betroffene Clients
1	Vorname	Michaela	Erstes „a“ entfernt	2
	Geburtstag	21.10.1943	„21“ zu „20“ geändert	3
	Straße	Ladeckallee 7	„7“ zu „17“ geändert	3
2	Vorname	Elli	„s“ angehängt	2+3
	Straße	Mallersdorf	Erstes „l“ entfernt	3
3	Nachname	Preiß	„ß“ zu „s“ geändert	2
	Vorname	Norma	„n“ angehängt	3
	Straße	Irmgard-Küchnert-Allee	„t“ entfernt	3

Tabelle 8.1: Vorgenommene Änderungen in synthetischen Testdaten

Ausführung des Testfalls Das Python-Skript benötigt für jeden zu erstellenden Client eine Liste von Datensätzen. Zusätzlich wird eine Konfigurationsdatei übergeben, welche die Schemata für einzelne Attribute festlegt. Zuerst wird durch das Skript eine neue Session am Broker erstellt. Danach werden alle Clients nach den gleichen Schritten abgearbeitet. Alle Datensätze für diesen Client werden mithilfe des Encoders



(a) Datensätze mit sieben Attributen



(b) Datensätze mit vier Attributen

Abbildung 8.5: Ähnlichkeitsverteilung zwischen drei Clients mit je 100 Datensätzen. Drei Datensätze sind bis auf kleine typographische Fehler zwischen den Clients identisch. Die restlichen Datensätze sind zufällig generiert und weisen keine Ähnlichkeit zueinander auf.

in Bitvektoren umgewandelt. Das Skript führt eine Zuordnung von Bitvektoren zu den originalen Datensätzen. Anschließend wird am Broker ein neues Client Secret erzeugt, welches sofort verwendet wird, um die generierten Bitvektoren zu senden. Nachdem alle Clients ihre Bitvektoren gesendet haben, wartet das Skript auf den Abschluss des Matchings. Zuletzt werden für jeden Client die Ergebnisse vom Broker abgerufen und mithilfe der angelegten Zuordnung zurück in die originalen Datensätze übersetzt.

Ergebnisse Der Broker konnte alle drei Datensätze zwischen den einzelnen Clients identifizieren und zusammenführen. Um die Wahl des Schwellenwertes zu bewerten, wurde der beschriebene Ablauf mit 100 statt 1000 Datensätzen pro Client wiederholt und der Schwellenwert für das Matching auf 0 % gesetzt. Damit erhalten alle Clients eine Liste über alle berechneten Ähnlichkeiten zu Datensätzen von allen anderen Clients.

Die Verteilung der berechneten Schwellenwerte für alle Bitvektorpaare ist in Abbildung 8.5a dargestellt. Sie zeigt, dass 95 % für die gegebene Datenbasis die obere Grenze für den Schwellenwert repräsentiert, ohne falsch negative Ergebnisse zu erhalten. Andersrum befindet sich die untere Grenze für den Schwellenwert bei ungefähr 70 %, bevor sich die ersten falsch positiven Ergebnisse manifestieren würden.

Werden die Attribute reduziert auf Vorname, Nachname, Geburtsdatum und Geschlecht, so wird die Kurve der Non-Matches wesentlich breiter, wie in Abbildung 8.5b zu sehen ist. Der obere Schwellenwert, um alle erwarteten Matches ohne falsch negative Ergebnisse zu erhalten, ist zudem reduziert auf 93 %.

8.4 Performance-Analyse

Der Broker ist nur in einer Produktionsumgebung anwendbar, wenn dieser großen Datenmengen mithalten kann. In diesem Abschnitt soll abschließend auf die Performance des Matchings und des Brokers eingegangen werden. Alle Performance-Tests wurden auf einem Lenovo ThinkPad L480 ausgeführt. Dieser ist ausgestattet mit einer Intel i5-8250U CPU und 8 GB DDR4-2400 RAM. Der Laptop läuft auf Arch Linux mit dem Linux LTS Kernel 5.15.15-1. Weitere relevante Spezifikationen werden in den folgenden Absätzen erwähnt.

8.4.1 Bitweises Matching

In einer Session ist das Matching die rechenintensivste Phase. Demzufolge muss das Matching sehr performant sein, um auch bei längeren Bitvektoren und mehreren Millionen zu vergleichenden Bitvektorpaaren Ergebnisse in einer sensiblen Zeit berechnen können. Der bitweise Vergleich von Bitvektoren unter der Verwendung des Dice-

Koeffizienten wurde hierfür einem Benchmark unterzogen. Ziel ist die Ermittlung der möglichen Ausführungen pro Sekunde.

Für die Durchführung der Benchmarks wurde die *Java Microbenchmark Harness* (JMH)[27] in der Version 1.34 verwendet. Unterschiedliche Bitvektorlängen von 16 bis 1024 Bytes mit zufälligen Werten wurden für die Benchmarks verwendet. Die Benchmarks wurden in insgesamt drei verschiedenen Prozessen ausgeführt mit jeweils jeweils 10 Messungen. Die ersten fünf Messungen werden verworfen. Somit werden Ausreißer durch Kaltstarts vermieden. Aus den letzten fünf Messungen wird der Mittelwert gebildet. Werte für die Bitvektoren wurden zwischen den Durchläufen neu generiert. Ausgeführt wurden die Tests in der OpenJDK JVM in der Version 11.0.13.

Bytes	**Vergleichsoperationen** $\left[10^6\,\mathrm{s}^{-1}\right]$
16	41,111 ± 0,397
32	35,198 ± 0,236
64	27,707 ± 0,522
128	21,687 ± 0,244
256	14,860 ± 0,091
512	8,914 ± 0,029
1024	4,819 ± 0,038

Tabelle 8.2: Anzahl der Bitvektor-Vergleichsoperationen pro Sekunde

Die Ergebnisse sind in Tabelle 8.2 aufgeführt. Wie zu erwarten nimmt die Anzahl der Vergleichsoperationen bei größeren Bitvektoren ab. Das ist damit begründet, dass Bitvektoren im Hauptspeicher als Arrays von 64-Bit Ganzzahlen gespeichert werden. Beim Vergleich werden zwei Arrays miteinander mit einer bitweisen Konjugation verknüpft, was bedeutet, dass diese Operation auf jedes Element aus den beiden Arrays paarweise angewendet werden muss. Längere Arrays bedeuten mehr Konjugationen. Dennoch ist eine solche bitweise Operation sehr effizient. Der Großteil der Rechenzeit beläuft sich auf das Iterieren und die Zugriffe auf das Array.

Dieser Benchmark deckt Bloomfilter mit 128 bis 8192 Bits ab. Da sich die Größe der Bloomfilter in der Implementierung von *Google Guava* nach der Anzahl der zu erwartenden Elemente — also der Anzahl der q-Gramme — richtet, bietet dieser Benchmark eine realistische Einschätzung über die reine Performance des Matchings. Die gleiche Formel, die für die Berechnung der Größe des *Google Guava* Bloomfilters verwendet wird, kann umgestellt werden um die Anzahl der zu erwartenden Elemente abzuleiten. Für eine Bloomfiltergröße von 8192 Bits ergeben sich 1122 Elemente. Bei einem Datensatz, welcher Vor- und Nachname, Geburtsdatum und Geschlecht umfasst, würde die Anzahl der q-Gramme meistens im zweistelligen Bereich liegen. Werden weitere Attribute eingerechnet, dann kann diese Zahl im dreistelligen Bereich liegen. Selbst dann ist es jedoch sehr unwahrscheinlich, dass ein Datensatz sich in mehr als 1000 einzigartige

[27] Siehe `https://github.com/openjdk/jmh`

q-Gramme zerlegen lässt und somit die Performance des Matchings stark beeinflusst. Auch bei längeren Bitvektoren ist das Matching performant genug, um keinen Engpass im PPRL-Prozess darzustellen.

8.4.2 Laufzeit des Brokers

Der Broker wurde für den Benchmark um eine Timer-Klasse erweitert, welche Codeteile im Nanosekundenbereich messen kann. Es werden nur Codeteile im Broker gemessen, in denen auf Antworten von externen Diensten gewartet wird. Ein Beispiel ist in Listing 8.3 zu sehen. Hier wird allein die Dauer zur Verarbeitung des `LLEN`-Kommandos in *Redis* inklusive eventueller Netzwerklatenzen gemessen.

```
public long getPendingMatchCount(String session) {
    requireNonNull(session);

    try (Jedis j = pool.getResource()) {
        // llen = length of list
        Timer.start("queue:getPendingMatchCount");
        long l = j.llen(key(session, KEY_QUEUE));
        Timer.stop("queue:getPendingMatchCount");
        return l;
    }
}
```

Listing 8.3: Timing-Code in der Redis-Queue-Implementierung

Ein Python-Skript erstellt für eine Session zwei Clients. Für beide Clients werden 1000 Bitvektoren mit 64 Bytes zufällig erstellt, wobei 10 Bitvektoren zueinander identisch sind. Das Matching wird mit einem Schwellenwert von 99 % ausgeführt. Die Bitvektoren der Clients werden an den Broker geschickt, das Matching abgewartet und die Ergebnisse abgerufen. Der ganze Ablauf wird insgesamt sechs Mal durchgeführt, einmal zur Vermeidung von Kaltstarts und fünfmal zum Messen. Anschließend werden die gemessenen Codeteile des Brokers ausgewertet. Für jeden gemessenen Codeteil wird der Mittelwert berechnet. Der Broker läuft mit den Standardeinstellungen.

Die Ergebnisse dieses Benchmarks sind in Tabelle 8.3 dargestellt. Beim Senden der Bitvektoren werden für jeden Client sofort 1000 Bitvektoren in den Cache eingefügt. Das Einfügen von Bitvektorpaaren in die Queue erfolgt schrittweise in Schritten von je 1000 Paaren. Auch wenn der Cache scheinbar viel mehr Zeit braucht um die Bitvektoren zu speichern, wird die eine Aktion der Queue mehrere Male ausgeführt. Demzufolge beläuft sich die tatsächliche Zeit dieser Phase auf etwa zwei bis drei Sekunden bei der gegebenen Datenmenge. Da das Senden der Bitvektoren keine zeitkritische Phase ist, sind die gemessenen Zeiten im akzeptablen Rahmen.

Phase	Komponente	Aktion	Zeit $[\mathrm{ms}]$
Senden der Bitvektoren	Queue	Einreihen von Bitvektorpaaren	2,763
	Cache	Clients abfragen	7,005
		Bitvektoren für Client abfragen	11,205
		Bitvektoren einfügen	46,294
Matching	Queue	Bitvektorpaare abfragen	2,508
		Bitvektorpaare entfernen	0,217
	Cache	Sessions abfragen	5,640
		Matches einfügen	40,493
	Matcher	Matching von Bitvektorpaaren	13,999
Löschen der Session	Cache	Session löschen	72,983
	Queue	Session löschen	0,778
Session-Fortschritt	Queue	Länge der Queue abfragen	0,135
		Anzahl aller Bitvektorpaare abfragen	0,204
Match-Ergebnisse	Cache	Matches abfragen	4,274
Sonstiges	Matcher	Ähnlichkeitsfunktionen abfragen	2,322
		Filterstrategien abfragen	1,578

Tabelle 8.3: Durchschnittliche Zeiten für Operationen des Brokers

Beim Matching wird die Queue der Session in Schritten von 5000 Bitvektorpaaren abgearbeitet. Die Queue antwortet dementsprechend schnell. Interessant sind die Antwortzeiten des Matchers, welche Aufschluss über die tatsächliche Anzahl der Vergleichsoperationen pro Sekunde gibt im Vergleich zu der praktisch möglichen Anzahl aus dem vorhergehenden Abschnitt. Bei 5000 Bitvektorpaaren in 13,999 ms ergibt sich ein Wert von zirka 357 000 Vergleichsoperationen pro Sekunde. In diesem Wert sind die Netzwerklatenzen und die Zeit, die benötigt wird um die Anfrage zu bearbeiten, inbegriffen. Es ist also davon auszugehen, dass der Broker in einer akzeptablen Zeit mehrere Millionen Bitvektorpaare verwalten kann. Das Matching von zwei Datenbasen mit je 10 000 Datensätzen könnte in unter fünf Minuten, bei je 100 000 Datensätzen in knapp unter acht Stunden, durchgeführt werden. Auch das Einfügen von Matches in den Cache ist sehr gut optimiert, ohne die Laufzeit des Matchings stark zu beeinflussen.

Das Löschen von Sessions ist ebenfalls nicht zeitkritisch. Dennoch zeichnet sich hier ab, dass *Neo4j* als Cache wesentlich länger braucht um Daten zu löschen als die Queue basierend auf *Redis*. Da diese Aktionen in einem separaten Thread ausgeführt werden, wird der Verwalter der Session nicht auf diese Latenzen aufmerksam werden. Dennoch ist dieser Unterschied bemerkenswert.

Die Ausführungszeiten der restlichen Aktionen sind angemessen. Das Ermitteln des Fortschritts einer Session basiert auf zwei einfachen *Redis*-Kommandos. Auch das Abfragen der Ergebnisse aus dem Cache ist sehr gut optimiert, wobei sich die Laufzeit dieser Anfrage wahrscheinlich vergrößern wird je mehr Matches gefunden und somit in den Cache eingetragen werden. Die Anfragen an den Matcher sind ebenfalls sehr schnell, da im Endeffekt nur konstante Werte abfragt werden und die Anfrage und Antwort dementsprechend schnell verarbeitet werden können. Insgesamt sind keine größeren Engpässe in der Funktionsweise des Brokers erkennbar.

9 Diskussion

Abschließend werden in diesem Kapitel einige Beobachtungen in den Ergebnissen aus dem vorherigen Kapitel geschildert und erläutert. Es werden einige Probleme der aktuellen Implementierung mit möglichen Lösungsvorschlägen aufgezeigt. Zudem wird eine Analyse der Sicherheit anhand aktueller Angriffe auf Bloomfilter-gestützte PPRL-Protokolle durchgeführt und eine Liste von zusätzlichen Features vorgestellt, welche die Richtung für die zukünftige Entwicklung der Webservices vorgeben.

9.1 Qualitative Bewertung

Die Ergebnisse aus dem vorherigen Kapitel zeigen eindeutig, dass es möglich ist mit den entwickelten Komponenten eine fehlertolerante Verknüpfung von ähnlichen Datensätzen über Institutionsgrenzen hinweg durchzuführen. Somit können Entitäten mit ähnlichen Datensätzen an verschiedenen Standorten identifiziert und zusammengeführt werden. Insgesamt versteht sich die entwickelte PPRL-Architektur als ein Art Vorverarbeitungsschritt in einer groß angelegten Datenanalyse. Die Ergebnisse eines PPRL-Durchlaufs können verwendet werden, um weitere Entscheidungen an einer Station zu treffen. Als Beispiel sei der gezielte Ausschluss doppelter Datensätze beim Training eines Modells für das maschinelle Lernen genannt.

Dennoch zeigen die Ergebnisse, dass für die erfolgreiche Datenverknüpfung einige Aspekte zu beachten sind. Die Qualität der Ergebnisse hängt hauptsächlich von Encoder und Matcher ab. An dem Beispiel aus dem 5. MII-Projektathon, welches in Abschnitt 8.2 demonstriert wurde, können einige Ursachen für falsch positive Ergebnisse abgeleitet werden.

Die Angabe von geeigneten Attributschemata hat eine hohe Relevanz. Im genannten Beispiel wurden die Datentypen aus dem Datenschema des E-PIX automatisch abgeleitet. Das hatte zur Folge, dass der Lebenszeitpunkt als Zeichenkette und nicht als Datumsangabe interpretiert wurde. Prinzipiell sind Attributschemata so zu definieren, dass zufällige Übereinstimmungen zwischen verschiedenen Datensätzen so gut wie möglich ausgeschlossen sind. Das bedeutet, dass Attribute mit ihren korrekten Datentypen annotiert und gegebenenfalls um weitere Optionen für den Encoder erweitert werden müssen. Der Datenanalyst muss also eng mit der Art und den Ausprägungen der zu verknüpfenden Daten vertraut sein.

Ein weiterer wichtiger Faktor ist die Wahl des Schwellenwertes. Für die Projektathon-Beispieldaten wurde der Schwellenwert zu niedrig angesetzt. Mit der Kenntnis, dass nur exakte Übereinstimmungen gesucht werden, hätte der Schwellenwert auf 100 % gesetzt

werden können. Allerdings zeigt das Beispiel mit synthetischen Daten aus Abschnitt 8.3, dass die Wahl eines geeigneten Schwellenwertes bei Daten mit typographischen Fehlern keineswegs trivial ist. Die Tatsache, dass die echten Daten, welche verknüpft werden sollen, für den Datenanalysten nicht einsehbar sind, erschwert diese Entscheidung. Auch hier ist wieder die Expertise des Datenanalysten gefragt. Als Entscheidungshilfe können Beispieldaten verwendet werden, welche den realen Daten sehr ähnlich sind, um einen geeigneten Schwellenwert zu ermitteln. Jedoch zeigen die Ergebnisse der synthetischen Daten, dass zwischen ähnlichen und grundsätzlich verschiedenen Datensätzen eine große Differenz in den berechneten Ähnlichkeiten existiert. Somit existiert in der derzeitigen PPRL-Implementierung mit den vorgeführten Daten ein angenehmer Spielraum für den Schwellenwert, in dem falsch positive als auch falsch negative Ergebnisse bestmöglich vermieden werden können.

9.2 Bekannte Probleme

Denial of Service Obwohl noch nicht getestet, sind verschiedene klassische Netzwerkangriffe auf den Broker als zentrale Komponente in der PPRL-Infrastruktur möglich. Da es keine Grenze für die Größe von Anfragen gibt, ist es möglich beliebig viele Bitvektoren an den Broker zu senden. Das kann unter Umständen zu einer Ausreizung des Hauptspeichers führen, welcher von der JVM zur Verfügung gestellt wird. Eine weitere Möglichkeit ist das Senden von vielen Anfragen hintereinander, was beispielsweise durch Verlangsamung weiterer einkommender Anfragen zu einem „Denial of Service“ führen kann. Für eine Anwendung des Brokers in einer Produktionsumgebung ist die Verwendung eines Proxy somit unerlässlich. Dem Proxy muss es möglich sein, Rate Limiting durchzuführen und Anfragen abzuweisen, welche eine bestimmte Größe überschreiten.

Bloomfilter-Implementierung Im Laufe der Arbeit stellte sich heraus, dass *Google Guava*’s Bloomfilter-Implementierung für den Einsatz in PPRL-Protokollen nur bedingt geeignet ist. Wie schon diskutiert hängt die Größe der Bloomfilter von der Anzahl der erwarteten Elemente ab. Die Größe des Bloomfilters und die Anzahl der anzuwendenden Hashfunktionen lassen sich nicht konfigurieren. Eine Möglichkeit ist es, die Funktionen, die in der Implementierung zur Berechnung der genannten Parameter verwendet werden, auf die gewünschten Parameter umzustellen. Einfacher ist es jedoch, eine andere Bloomfilter-Implementierung zu adaptieren. Genannt sei an dieser Stelle zum Beispiel Baqend’s Bloomfilter.

PHT-Integration Die Übertragung von Session Secrets hängt vom gewählten Kommunikationskanal zwischen dem Datenanalysten und den Stationsbetreibern ab. Ist dieser

Kanal nicht abgesichert, so besteht das Risiko einer unerwarteten Freigabe des Secrets. Eine Lösung hierfür wäre die Implementierung von Klassenvariablen in die PHT-Infrastruktur. Das bedeutet, dass es dem Datenanalysten vor dem Senden eines Zuges möglich sein soll, bestimmte Variablen schon im Voraus zu setzen, ohne dass die Stationsbetreiber an ihnen etwas ändern können. Damit würde sich beispielsweise auch die Eingabe der Datendomäne an jeder Station erübrigen.

Vertrauen in den Broker Dem Betreiber des Brokers muss von jeder Station vertraut werden. Auch wenn der Betreiber des Broker nur Bitvektoren erhält, existieren einige Angriffe, welche den Rückschluss auf originale Daten, die zur Besetzung des Bloomfilters verwendet wurden, ermöglichen. Solche Angriffe werden unter anderem im nächsten Abschnitt beschrieben. Prinzipiell ist es dennoch wünschenswert, dass der Broker als eine austauschbare Komponente betrachtet werden kann. Ist beispielsweise eine Broker-Instanz an einer sonst vertrauenswürdigen Institution nicht verfügbar, so soll es kein Problem sein eine eigene, öffentliche Broker-Instanz bereitzustellen, ohne erst um das Vertrauen der Teilnehmer einer Session werben zu müssen.

9.3 Sicherheitsaspekte

Es existieren eine Reihe von Angriffen auf Bloomfilter. Ziel solcher Angriffe ist es, die Daten, die in einen Bloomfilter eingefügt wurden, zu rekonstruieren. Bei Wörterbuchangriffen wird ein Korpus von bekannten Daten in einen Bloomfilter eingefügt, um zu überprüfen welche Bits für welches Datum gesetzt werden [32]. Dafür muss zum einen der Bloomfilter die gleichen Parameter wie der anzugreifende Filter besitzen. Zum anderen muss der gleiche Maskierungsprozess auf den Korpus angewendet werden. Wörterbuchangriffe können vermieden werden, indem für das Hashing ein kryptografischer Schlüssel verwendet wird, entweder in der Form eines Seed-Wertes oder in Verbindung mit einem hash-basierten Nachrichtenauthentifizierungscode (HMAC).

Häufiger sind Frequenzangriffe, welche die gesetzten Bits zwischen Bloomfiltern miteinander vergleichen, um auf Daten zurückzuschließen, die in einer Mehrheit von Bloomfiltern gesetzt sind [32]. Wenn beispielsweise zwei Bloomfilter konstruiert werden für die Namen „Alex“ und „Alexa“, wobei die Namen zuvor in Bigramme aufgeteilt werden, so wird ein Großteil der gesetzten Bits zwischen ihnen identisch sein. Nimmt man zusätzlich die Namen „Albert“, „Alice“ und „Alfonso“ her, dann ist es möglich, die Bitposition zu bestimmen, welche mit dem Bigramm „al“ korrespondiert. Dieser Angriff funktioniert trotz Seeding und HMAC, auch wenn der Schlüssel dem Angreifer nicht bekannt ist.

Ein „constraint satisfaction“ Angriff von Kuzu et al. [41] auf Bloomfilter mit Parametern, welche in der Literatur häufig vorgeschlagen werden, stellte sich als sehr wirksam heraus. Die Autoren empfehlen größere q-Gramme und ein kleineres Verhältnis von einge-

setzten Hashfunktionen zu Bits im Bloomfilter. Eine weitere, sehr effiziente Kryptoanalyse von Christen et al. [42] basiert auf einem Frequenzangriff und ist unabhängig von den Bloomfilter-Parametern. Die Rekonstruktion von sensitiven Daten aus einer großen Menge von Bloomfiltern ist einfach durchzuführen, sofern ein Korpus von bekannten Daten existiert. Die Maskierung von sensitiven Daten mit Bloomfiltern ohne weitere Sicherheitsvorkehrungen ist unsicher. Somit bietet das in dieser Arbeit vorgestellte PPRL-Protokoll keine Sicherheit für echte Personendaten.

Christen et al. [42] stellen einige Methoden zum Härten von Bloomfiltern zum Einsatz in PPRL-Protokollen vor. Einfache Möglichkeiten stellen das Seeding von q-Grammen und, bekräftigt durch Schnell et al. [27], das Randomisieren und Balancieren von Bloomfiltern, sodass 50 % der Bits gesetzt sind.

Eine weitere, robuste Möglichkeit, vorgeschlagen von Durham et al. [43], ist der Einsatz von „record-level" Bloomfilter (RBF). „Field-level" Bloomfilter (FBF) verfolgen den bereits bekannten Ansatz. Einzufügende Daten werden in q-Gramme aufgeteilt und die Textfragmente werden in den Bloomfilter eingefügt. RBFs hingegen setzen sich aus den FBFs der einzelnen Eigenschaften eines Datensatzes zusammen. Die Bitpositionen im RBF werden zufällig aus den FBFs gezogen, sodass eine gleichmäßige Verteilung von Bits im resultierenden Bloomfilter erzielt wird. Diese Methode ist resistent gegen die Angriffsmethode von Kuzu et al. und ermöglicht auch die Gewichtung von Feldern in einem Datensatz.

9.4 Ausblick

Metadaten Die Aussagekraft der Ergebnisse eines PPRL-Durchlaufs kann noch verbessert werden. Stationsbetreiber erhalten lediglich eine Liste von Pseudonymen mit zugeordneten Wahrscheinlichkeiten. Um informierte Entscheidungen über die Ergebnisse eines PPRL-Durchlaufs zu treffen, ist beispielsweise der Einsatz von Metadaten für jeden Bitvektor möglich. Jeder Bitvektor könnte zusätzliche Daten erhalten, zum Beispiel das Änderungsdatum des zugrundeliegenden Datensatzes. Diese Metadaten würden dann ebenfalls in den Ergebnissen präsentiert werden. So wäre es dann möglich, bei einer Dublette nur den aktuellsten Datensatz anzufordern. Es ist jedoch wichtig, dass die Metadaten keine Rückschlüsse auf die ursprünglichen Datensätze erlauben.

Parallelisierung der Queue Ursprünglich wurde *Neo4j* eingesetzt, um sowohl Ergebnisse zu speichern als auch unverknüpfte Bitvektorpaare abzufragen. Letzteres hat sich als sehr langsam herausgestellt, weswegen *Redis* als Queue verwendet wird. Zwar ist *Redis* schnell und funktioniert problemlos, dennoch ist es in erster Linie ein Schlüssel-Werte-Speicher. Die Rollback-Funktionen mussten von Hand in Java implementiert werden. Weiterhin kann das Abarbeiten der Queue in diesem Zustand nicht parallelisiert

werden. Genauso ist das Warten auf eine Antwort vom Matcher und dem darauffolgenden Einfügen von Ergebnissen in den Cache eine umständliche Lösung für ein Problem, welches sich anderweitig lösen lässt. Viel effizienter wäre die Verwendung eines Message Brokers wie *RabbitMQ*[28]. Mit einer designierten Message Queue kann der Broker „Arbeitspakete“ zur asynchronen Verarbeitung vorbereiten, welche dann von parallel laufenden Prozessen aufgegriffen und bearbeitet werden. Die Anzahl der Prozesse kann je nach Rechenkapazität beliebig angepasst werden.

Bloomfilter-Parametrisierung Wenn die Bloomfilter-Implementierung in der jetzigen PPRL-Infrastruktur ersetzt wird, dann müssen die Größe des Filters und die Anzahl der Hashfunktionen als Parameter im Kodierungsschritt übergeben werden. Diese sind für Laien nicht sehr intuitiv zu verstehen. Die Wahl der Parameter allein um ein qualitativ hochwertiges Matching zu erzeugen, gestaltet sich schwierig. Darüber hinaus existiert zu dem Zeitpunkt der Arbeit kein Konsens, wie diese Parameter zu besetzen sind. Bislang hat jedes vorgeschlagene PPRL-Protokoll, inklusive das in dieser Arbeit vorgestellte, die Parameter an Testdaten angepasst, ohne jedoch einen generischen Algorithmus zur Wahl der Parameter für beliebige Daten vorzuschlagen. Zukünftige Forschung sollte sich dem Ermitteln eines Maßstabes für die Besetzung von Bloomfilter-Parametern anhand der Art der Originaldaten widmen. Weiterhin sollte die Möglichkeit erkundet werden, ob andere, leicht verständliche Eigenschaften eines Bloomfilters als Maße für das Matching verwendet werden können, um daraus Bloomfiltergröße und Anzahl der Hashfunktionen abzuleiten.

Laufzeitanalyse Ein Schritt eines generischen PPRL-Protokolls, welcher in dieser Arbeit nur kurz in Betracht gezogen wurde, ist das Filtern als Bereinigungsschritt vor dem Matching. Eingangs wurden Methoden wie P4Join und Multibit-Trees vorgestellt, um die Anzahl der Vergleichsoperationen zu minimieren. Da jedoch das bitweise Matching an sich schon sehr performant ist, stellt sich die Frage ob durch einen zusätzlichen Filter-Schritt die Laufzeit des in dieser Arbeit vorgestellten PPRL-Protokolls signifikant reduziert werden kann. Ebenso interessant ist der Einsatz von GPUs, um das Matching zu beschleunigen. Matching-Routinen können in GPU-Kernel ausgelagert werden, welche anschließend massiv parallel ausgeführt werden können.

9.5 Zusammenfassung

Durch die Wahlfreiheit eines jeden Patienten, entstehen an mehreren medizinischen Instituten Daten, welche sich auf die gleiche Person beziehen. Diese Daten zu erkennen und zu verknüpfen ist ein wichtiger Schritt, um die Qualität der Ergebnisse in der

[28] Siehe `https://www.rabbitmq.com/`

medizinischen Datenanalyse zu erhöhen und die Laufzeit durch Vermeidung unnötiger Rechenoperationen zu reduzieren. Für diese Zwecke wurde in dieser Arbeit eine PPRL-Infrastruktur zum Einsatz in Systemen zur verteilten Datenanalyse konzipiert, implementiert und ausgewertet. Der PHT-Ansatz wurde als unterstützende Ausführungsplattform gewählt, um auf Daten direkt an den Stationen zugreifen zu können, statt diese an einer zentralen Stelle zu sammeln. Somit werden personenbezogene und medizinische Daten effektiv geschützt.

Die Maskierung im PPRL-Protokoll wurde mit Bloomfiltern realisiert, um dem Rückschluss auf die originalen Daten erheblich zu erschweren und um die Vergleichbarkeit zwischen ähnlichen Datensätzen zu garantieren. Anhand zwei gewählter Fallbeispiele wurde die Funktionsweise der PPRL-Infrastruktur im PHT und die Toleranz gegenüber typographischen Fehlern demonstriert und bewiesen. Performance-Analysen belegen, dass die entwickelten Komponenten auch sehr großen Datenmengen standhalten und Ergebnisse in einem akzeptablen Zeitrahmen vorweisen können. Zukünftige Forschung sollte sich vor allem der kryptografischen Härtung des PPRL-Protokolls widmen. Weitere Verbesserungen, um die Verwendbarkeit der entwickelten Komponenten zu erhöhen, sind ebenfalls ratsam.

Anhang A: Anfrage und Antwort für den Encoder-Service


```
{
  "charset": "UTF-8",
  "seed": "foobar",
  "generation-function": "TRIGRAM_ATTRIBUTE_BLOOM_FILTER",
  "entity-list": [
    {
      "identifier": "0001",
      "attribute-value-list": [
        {
          "name": "first-name",
          "value": "Maximilian"
        },
        {
          "name": "height",
          "value": "1.85"
        }
      ]
    }
  ],
  "schema-list": [
    {
      "attribute-name": "first-name",
      "data-type": "string"
    },
    {
      "attribute-name": "height",
      "data-type": "float",
      "sig-figs": "1"
    }
  ]
}
```


Listing A.1: Beispiel einer PPRL-Encoder Anfrage


```
{
  "entity-list": [
    {
      "identifier": "0001",
      "encoding": "TmEsIG5ldWdpZXJpZz8="
    }
  ]
}
```


Listing A.2: Beispiel einer PPRL-Encoder Antwort

Anhang B: Anfrage und Antwort für den Match-Service


```
{
  "match-configuration": {
    "match-function": "MONOGRAM_DICE",
    "selection-strategy": "FULL_RESULT",
    "match-mode": "PAIRWISE",
    "threshold": 0
  },
  "domain-entity-list": [
    {
      "bit-vector": "NDE4IEknbSBhIHRlYXBvdA=="
    }
  ],
  "range-entity-list": [
    {
      "bit-vector": "NDA0IE5vdCBGb3VuZA=="
    }
  ]
}
```


Listing B.1: Beispiel einer PPRL-Matcher Anfrage


```
{
  "match-configuration": {
    "match-function": "MONOGRAM_DICE",
    "selection-strategy": "FULL_RESULT",
    "match-mode": "PAIRWISE",
    "threshold": 0
  },
  "correspondence-list": [
    {
      "domain-bit-vector": "NDE4IEknbSBhIHRlYXBvdA==",
      "range-bit-vector": "NDA0IE5vdCBGb3VuZA==",
      "confidence": 0.42
    }
  ]
}
```


Listing B.2: Beispiel einer PPRL-Matcher Antwort

Anhang C: Ausgabe des Zuges für den MII-Projektathon Beispieldatensatz

```
hsmw_854980555 ,1.0
hsmw_604458991 ,1.0
hsmw_300949159 ,1.0
hsmw_984584448 ,1.0
hsmw_045541360 ,1.0
hsmw_286262166 ,1.0
hsmw_893844399 ,1.0
hsmw_367740994 ,1.0
hsmw_655671668 ,1.0
hsmw_045541360 ,0.9077380895614624
rwth_825863076 ,1.0
rwth_084669279 ,1.0
rwth_703478825 ,1.0
rwth_836575770 ,1.0
rwth_151573877 ,1.0
rwth_960890472 ,1.0
rwth_332094188 ,1.0
rwth_499344120 ,1.0
UKL_029173425 ,1.0
UKL_802864209 ,1.0
UKL_688867753 ,1.0
UKL_759210331 ,1.0
UKL_680188676 ,1.0
UKL_814426361 ,1.0
UKL_018574329 ,0.9077380895614624
UKL_329331423 ,1.0
UKL_788144199 ,1.0
UKL_882465929 ,1.0
UKL_746146882 ,1.0
```

Listing C.1: Ausgabe des PPRL-Zuges für die MII-Beispieldaten

Literatur

[1] Yehida Lindell. *Secure Multiparty Computation for Privacy Preserving Data Mining*. Encyclopedia of Data Warehousing and Mining. ISBN: 9781591405573 Pages: 1005-1009 Publisher: IGI Global. 2005. DOI: 10.4018/978-1-59140-557-3.ch189. URL: https://www.igi-global.com/chapter/encyclopedia-data-warehousing-mining/www.igi-global.com/chapter/encyclopedia-data-warehousing-mining/10743 (besucht am 10. 12. 2021).

[2] G. R. Howe. „Use of computerized record linkage in cohort studies". In: *Epidemiologic Reviews* 20.1 (1998), S. 112–121. ISSN: 0193-936X. DOI: 10.1093/oxfordjournals.epirev.a017966.

[3] William E. Winkler. „Matching and Record Linkage". In: *Business Survey Methods*. Section: 20 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118150504.ch20. John Wiley & Sons, Ltd, 1995, S. 353–384. ISBN: 978-1-118-15050-4. DOI: 10.1002/9781118150504.ch20. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118150504.ch20 (besucht am 23. 03. 2022).

[4] William E. Winkler. *Overview of record linkage and current research directions*. BUREAU OF THE CENSUS, 2006.

[5] Tamás Budavári und Thomas J. Loredo. „Probabilistic Record Linkage in Astronomy: Directional Cross-Identification and Beyond". In: *Annual Review of Statistics and Its Application* 2.1 (2015). _eprint: https://doi.org/10.1146/annurev-statistics-010814-020231, S. 113–139. DOI: 10.1146/annurev-statistics-010814-020231. URL: https://doi.org/10.1146/annurev-statistics-010814-020231 (besucht am 22. 03. 2022).

[6] *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance)*. Legislative Body: EP, CONSIL. 27. April 2016. URL: http://data.europa.eu/eli/reg/2016/679/oj/eng (besucht am 04. 01. 2022).

[7] *Health Insurance Portability and Accountability Act of 1996 (HIPAA) | CDC*. 21. Februar 2019. URL: https://www.cdc.gov/phlp/publications/topic/hipaa.html (besucht am 01. 02. 2022).

[8] DSS. *Data-matching Program (Assistance and Tax) Act 1990*. Archive Location: au Publisher: Attorney-General's Department. URL: https://www.legislation.gov.au/Details/C2020C00371/Html/Text,%20http://www.legislation.gov.au/Details/C2020C00371 (besucht am 01. 02. 2022).

[9] Peter Christen, Thilina Ranbaduge und Rainer Schnell. *Linking sensitive data: methods and techniques for practical privacy-preserving information sharing*. OCLC: 1204136118. 2020. ISBN: 978-3-030-59706-1. URL: http://public.eblib.com/choice/PublicFullRecord.aspx?p=6381213 (besucht am 14. 03. 2022).

[10] Rainer Schnell, Tobias Bachteler und Jörg Reiher. „Privacy-preserving record linkage using Bloom filters". In: *BMC Medical Informatics and Decision Making* 9.1 (25. August 2009), S. 41. ISSN: 1472-6947. DOI: 10.1186/1472-6947-9-41. URL: https://doi.org/10.1186/1472-6947-9-41 (besucht am 13. 12. 2021).

[11] Rainer Schnell, Tobias Bachteler und Jörg Reiher. *A Novel Error-Tolerant Anonymous Linking Code*. SSRN Scholarly Paper ID 3549247. Rochester, NY: Social Science Research Network, 16. November 2011. DOI: 10.2139/ssrn.3549247. URL: https://papers.ssrn.com/abstract=3549247 (besucht am 14. 03. 2022).

[12] Alexandros Karakasidis und Vassilios S. Verykios. „Secure Blocking + Secure Matching = Secure Record Linkage". In: *Journal of Computing Science and Engineering* 5.3 (2011). Publisher: Korean Institute of Information Scientists and Engineers, S. 223–235. ISSN: 1976-4677. DOI: 10.5626/JCSE.2011.5.3.223. URL: https://www.koreascience.or.kr/article/JAKO201128762648380.page (besucht am 15. 12. 2021).

[13] Sean M. Randall u. a. „Privacy-preserving record linkage on large real world datasets". In: *Journal of Biomedical Informatics*. Special Issue on Informatics Methods in Medical Privacy 50 (1. August 2014), S. 205–212. ISSN: 1532-0464. DOI: 10.1016/j.jbi.2013.12.003. URL: https://www.sciencedirect.com/science/article/pii/S1532046413001949 (besucht am 14. 12. 2021).

[14] Ziad Sehili u. a. *Privacy preserving record linkage with ppjoin*. Accepted: 2017-06-30T11:40:54Z ISSN: 1617-5468. Gesellschaft für Informatik e.V., 2015. ISBN: 978-3-88579-635-0. URL: http://dl.gi.de/handle/20.500.12116/2453 (besucht am 15. 12. 2021).

[15] Chuan Xiao u. a. „Efficient similarity joins for near-duplicate detection". In: *ACM Transactions on Database Systems* 36.3 (26. August 2011), 15:1–15:41. ISSN: 0362-5915. DOI: 10.1145/2000824.2000825. URL: https://doi.org/10.1145/2000824.2000825 (besucht am 15. 12. 2021).

[16] Tobias Bachteler, Jörg Reiher und Rainer Schnell. *Similarity Filtering with Multibit Trees for Record Linkage*. SSRN Scholarly Paper ID 3530899. Rochester, NY: Social Science Research Network, 15. März 2013. DOI: 10.2139/ssrn.3530899. URL: https://papers.ssrn.com/abstract=3530899 (besucht am 16. 12. 2021).

[17] Sebastian Stammler u. a. „Mainzelliste SecureEpiLinker (MainSEL): Privacy-Preserving Record Linkage using Secure Multi-Party Computation“. In: *Bioinformatics* (1. September 2020), btaa764. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btaa764. URL: https://doi.org/10.1093/bioinformatics/btaa764 (besucht am 10. 12. 2021).

[18] P. Contiero u. a. „The EpiLink Record Linkage Software“. In: *Methods of Information in Medicine* 44.1 (2005). Publisher: Schattauer GmbH, S. 66–71. ISSN: 0026-1270, 2511-705X. DOI: 10.1055/s-0038-1633924. URL: http://www.thieme-connect.de/DOI/DOI?10.1055/s-0038-1633924 (besucht am 16. 12. 2021).

[19] Thomas N. Herzog, Fritz J. Scheuren und William E. Winkler. „Record Linkage – Methodology“. In: *Data Quality and Record Linkage Techniques*. Hrsg. von Thomas N. Herzog, Fritz J. Scheuren und William E. Winkler. New York, NY: Springer, 2007, S. 81–92. ISBN: 978-0-387-69505-1. DOI: 10.1007/0-387-69505-2_8. URL: https://doi.org/10.1007/0-387-69505-2_8 (besucht am 03. 01. 2022).

[20] Ivan P. Fellegi und Alan B. Sunter. „A Theory for Record Linkage“. In: *Journal of the American Statistical Association* 64.328 (1. Dezember 1969). Publisher: Taylor & Francis _eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1969.10501049, S. 1183–1210. ISSN: 0162-1459. DOI: 10.1080/01621459.1969.10501049. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1969.10501049 (besucht am 15. 12. 2021).

[21] Adrian Sayers u. a. „Probabilistic record linkage“. In: *International Journal of Epidemiology* 45.3 (Juni 2016), S. 954–964. ISSN: 0300-5771. DOI: 10.1093/ije/dyv322. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5005943/ (besucht am 04. 01. 2022).

[22] Burton H. Bloom. „Space/time trade-offs in hash coding with allowable errors“. In: *Communications of the ACM* 13.7 (1. Juli 1970), S. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: https://doi.org/10.1145/362686.362692 (besucht am 13. 12. 2021).

[23] Andrei Broder und Michael Mitzenmacher. „Network Applications of Bloom Filters: A Survey“. In: *Internet Mathematics* 1.4 (1. Januar 2004). Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/15427951.2004.10129096, S. 485–509. ISSN: 1542-7951. DOI: 10.1080/15427951.2004.10129096. URL: https://doi.org/10.1080/15427951.2004.10129096 (besucht am 03. 01. 2022).

[24] Adam Horvath. *MurMurHash3, an ultra fast hash algorithm for C# / .NET*. 10. August 2012. URL: http://blog.teamleadnet.com/2012/08/murmurhash3-ultra-fast-hash-algorithm.html (besucht am 23. 03. 2022).

[25] Adam Kirsch und Michael Mitzenmacher. „Less Hashing, Same Performance: Building a Better Bloom Filter“. In: *Algorithms – ESA 2006*. Hrsg. von Yossi Azar und Thomas Erlebach. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, S. 456–467. ISBN: 978-3-540-38876-0. DOI: 10.1007/11841036_42.

[26] Peter C. Dillinger und Panagiotis Manolios. „Bloom Filters in Probabilistic Verification“. In: *Formal Methods in Computer-Aided Design*. Hrsg. von Alan J. Hu und Andrew K. Martin. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, S. 367–381. ISBN: 978-3-540-30494-4. DOI: 10.1007/978-3-540-30494-4_26.

[27] Rainer Schnell und Christian Borgs. „Randomized Response and Balanced Bloom Filters for Privacy Preserving Record Linkage“. In: *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. 2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW). ISSN: 2375-9259. Dezember 2016, S. 218–224. DOI: 10.1109/ICDMW.2016.0038.

[28] Dinusha Vatsalan u. a. „Privacy-Preserving Record Linkage for Big Data: Current Approaches and Research Challenges“. In: *Handbook of Big Data Technologies*. Hrsg. von Albert Y. Zomaya und Sherif Sakr. Cham: Springer International Publishing, 2017, S. 851–895. ISBN: 978-3-319-49340-4. DOI: 10.1007/978-3-319-49340-4_25. URL: https://doi.org/10.1007/978-3-319-49340-4_25 (besucht am 10. 12. 2021).

[29] Thomas N. Herzog, Fritz J. Scheuren und William E. Winkler. „Standardization and Parsing“. In: *Data Quality and Record Linkage Techniques*. Hrsg. von Thomas N. Herzog, Fritz J. Scheuren und William E. Winkler. New York, NY: Springer, 2007, S. 107–114. ISBN: 978-0-387-69505-1. DOI: 10.1007/0-387-69505-2_10. URL: https://doi.org/10.1007/0-387-69505-2_10 (besucht am 04. 01. 2022).

[30] Thomas N. Herzog, Fritz J. Scheuren und William E. Winkler. „Blocking“. In: *Data Quality and Record Linkage Techniques*. Hrsg. von Thomas N. Herzog, Fritz J. Scheuren und William E. Winkler. New York, NY: Springer, 2007, S. 123–130. ISBN: 978-0-387-69505-1. DOI: 10.1007/0-387-69505-2_12. URL: https://doi.org/10.1007/0-387-69505-2_12 (besucht am 04. 01. 2022).

[31] Hans Joachim Postel. „Die Kölner Phonetik. Ein Verfahren zur Identifizierung von Personennamen auf der Grundlage der Gestaltanalyse“. In: *IBM-Nachrichten* 19 (1969), S. 925–931.

[32] Dinusha Vatsalan, Peter Christen und Vassilios S. Verykios. „A taxonomy of privacy-preserving record linkage techniques“. In: *Information Systems* 38.6 (1. September 2013), S. 946–969. ISSN: 0306-4379. DOI: 10.1016/j.is.2012.11.005. URL: https://www.sciencedirect.com/science/article/pii/S0306437912001470 (besucht am 05. 01. 2022).

[33] Thomas N. Herzog, Fritz J. Scheuren und William E. Winkler. „String Comparator Metrics for Typographical Error“. In: *Data Quality and Record Linkage Techniques*. Hrsg. von Thomas N. Herzog, Fritz J. Scheuren und William E. Winkler. New York, NY: Springer, 2007, S. 131–135. ISBN: 978-0-387-69505-1. DOI: 10.1007/0-387-69505-2_13. URL: https://doi.org/10.1007/0-387-69505-2_13 (besucht am 04. 01. 2022).

[34] Oya Beyan u. a. „Distributed Analytics on Sensitive Medical Data: The Personal Health Train“. In: *Data Intelligence* 2.1 (Januar 2020), S. 96–107. ISSN: 2641-435X. DOI: 10.1162/dint_a_00032. URL: https://direct.mit.edu/dint/article/2/1-2/96-107/9997 (besucht am 27. 10. 2021).

[35] Zhenwei Shi u. a. „Distributed radiomics as a signature validation study using the Personal Health Train infrastructure“. In: *Scientific Data* 6.1 (22. Oktober 2019). Number: 1 Publisher: Nature Publishing Group, S. 218. ISSN: 2052-4463. DOI: 10.1038/s41597-019-0241-0. URL: https://www.nature.com/articles/s41597-019-0241-0 (besucht am 23. 03. 2022).

[36] Timo M. Deist u. a. „Distributed learning on 20 000+ lung cancer patients – The Personal Health Train“. In: *Radiotherapy and Oncology* 144 (1. März 2020), S. 189–200. ISSN: 0167-8140. DOI: 10.1016/j.radonc.2019.11.019. URL: https://www.sciencedirect.com/science/article/pii/S0167814019334899 (besucht am 23. 03. 2022).

[37] Yongli Mou u. a. „Distributed Learning for Melanoma Classification using Personal Health Train“. In: *arXiv:2103.13226 [cs]* (24. März 2021). arXiv: 2103.13226. URL: http://arxiv.org/abs/2103.13226 (besucht am 23. 03. 2022).

[38] Amadou Gaye u. a. „DataSHIELD: taking the analysis to the data, not the data to the analysis“. In: *International Journal of Epidemiology* 43.6 (1. Dezember 2014), S. 1929–1944. ISSN: 0300-5771. DOI: 10.1093/ije/dyu188. URL: https://doi.org/10.1093/ije/dyu188 (besucht am 08. 03. 2022).

[39] Sascha Welten u. a. „A Privacy-Preserving Distributed Analytics Platform for Health Care Data“. In: *Methods of Information in Medicine* (17. Januar 2022). Publisher: Georg Thieme Verlag KG. ISSN: 0026-1270, 2511-705X. DOI: 10.1055/s-0041-1740564. URL: http://www.thieme-connect.de/DOI/DOI?10.1055/s-0041-1740564 (besucht am 17. 01. 2022).

[40] Arturo Moncada-Torres u. a. „VANTAGE6: an open source priVAcy preserviNg federaTed leArninG infrastructurE for Secure Insight eXchange“. In: *AMIA Annual Symposium Proceedings* 2020 (25. Januar 2021), S. 870–877. ISSN: 1942-597X. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8075508/ (besucht am 08. 03. 2022).

[41] Mehmet Kuzu u. a. „A Constraint Satisfaction Cryptanalysis of Bloom Filters in Private Record Linkage“. In: *Privacy Enhancing Technologies*. Hrsg. von Simone Fischer-Hübner und Nicholas Hopper. Lecture Notes in Computer Science.

Berlin, Heidelberg: Springer, 2011, S. 226–245. ISBN: 978-3-642-22263-4. DOI: 10.1007/978-3-642-22263-4_13.

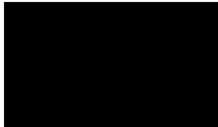
[42] Peter Christen u. a. „Efficient Cryptanalysis of Bloom Filters for Privacy-Preserving Record Linkage“. In: *Advances in Knowledge Discovery and Data Mining*. Hrsg. von Jinho Kim u. a. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, S. 628–640. ISBN: 978-3-319-57454-7. DOI: 10.1007/978-3-319-57454-7_49.

[43] Elizabeth A. Durham u. a. „Composite Bloom Filters for Secure Record Linkage“. In: *IEEE Transactions on Knowledge and Data Engineering* 26.12 (Dezember 2014). Conference Name: IEEE Transactions on Knowledge and Data Engineering, S. 2956–2968. ISSN: 1558-2191. DOI: 10.1109/TKDE.2013.91.

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.



Mittweida, 4. April 2022