Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik
Abteilung Datenbanken
&
mgm technology partners GmbH

Implementierung und Evaluation von parallelen Verfahren mit Flink zum Schutz personenbezogener Daten beim Record-Linkage

Masterarbeit

vorgelegt von

Martin Franke

Betreuer: **Ziad Sehili**
Fakultät für Mathematik und Informatik
Abteilung Datenbanken
Prof. Dr. Erhard Rahm

Michael Schmeißer
mgm technology partners GmbH

April 2017

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1
1.2 Ziele der Arbeit 5
1.3 Aufbau der Arbeit 6

2 Grundlagen und verwandte Arbeiten **7**

2.1 Privacy-preserving-Record-Linkage 7
2.1.1 Definition und Komplexität 7
2.1.2 PPRL-Prozess 11
2.1.3 Evaluierung 17
2.2 Bloom-Filter 24
2.2.1 Definition 24
2.2.2 Wahl der Parameter und Falsch-positiv-Rate 27
2.2.3 Bloom-Filter im PPRL 29
2.3 Blocking-Verfahren 32
2.3.1 Locality-sensitive-Hashing (LSH) 33
2.3.2 Phonetisches Blocking 37
2.4 Flink 39
2.4.1 Einführung 39
2.4.2 Flink-Programme 40
2.4.3 Verteilte Laufzeitumgebung 43
2.4.4 Flink-APIs 45
2.4.5 Monitoring mit Flink 47
2.5 Verwandte Arbeiten 49

3 Parallel-Privacy-preserving-Record-Linkage mit Flink **52**

3.1 Grundkonzept 52
3.2 Umzusetzende Blocking-Verfahren 53
3.3 Architektur 57
3.4 Vorverarbeitung 57

3.5 Linkage 59
3.5.1 Nested-Loop 60
3.5.2 LSH 61
3.5.3 Phonetisches Blocking 65
3.6 Evaluierung 65

4 Evaluierung **69**
4.1 Datengrundlage und verwendete Parameter 69
4.2 Systemkonfiguration 76
4.3 Methodik 76
4.4 Ergebnisse 77
4.4.1 Qualität 78
4.4.2 Skalierbarkeit und Speedup 83
4.5 Fazit 91

5 Zusammenfassung und Ausblick **93**

Abkürzungsverzeichnis **96**

Symbolverzeichnis **98**

Abbildungsverzeichnis **101**

Tabellenverzeichnis **103**

Literaturverzeichnis **104**

Selbständigkeitserklärung **116**

Kapitel 1

Einleitung

1.1 Motivation

Jeder Mensch erzeugt Daten. Diese Daten können dabei sehr bewusst und aktiv erzeugt werden, so beispielsweise durch das Verfassen von E-Mails und Tweets oder durch die Nutzung von sozialen Netzwerken und den damit verbundenen Austausch von Inhalten jeglicher Art. Aber diese Daten können auch eher unbewusst und passiv erzeugt werden: Zum Beispiel durch die bloße Geburt und damit Registrierung im Einwohnermeldeamt, aber auch durch Umzüge, einen Krankenhausaufenthalt oder durch die Aufnahme einer Beschäftigung. Alle diese Aktivitäten und Ereignisse erzeugen Daten, welche mit den Personen, die sie ausgelöst haben, assoziiert sind. Auch wenn nicht alle diese Daten auf den ersten Blick von Interesse erscheinen, so können dennoch verschiedenste Erkenntnisse aus Analyse dieser Daten gewonnen werden. Aus den vorherigen Beispielen lassen sich zum Beispiel wichtige Erkenntnisse über die Demographie eines Landes oder über aktuelle epidemiologische Entwicklungen ableiten. Bei der Analyse solcher Daten ist es jedoch oft notwendig, dass Datensätze und Informationen von verschiedenen Datenquellen berücksichtigt und miteinander verbunden werden. Die Gründe hierfür sind, dass durch eine Verbindung mehrerer Datenquellen zusätzliche Informationen zur Verfügung stehen und die Datenqualität verbessert werden kann [Chr12a, EIV07].

Angenommen es soll eine Studie zur Verkehrssicherheit mit der Forschungsfrage *"Verursachen Personen mit einer Mobilfunk-Flatrate mehr Verkehrsunfälle?"* durchgeführt werden. Unterschiedliche Organisationen und Einrichtungen (Parteien) speichern hierbei verschiedene Daten, die im Zusammenhang mit einem Verkehrsunfall stehen. Hiervon betroffen sind beispielsweise die Polizei sowie eventuell die Feuerwehr und der Rettungsdienst, außerdem die Kfz-Versicherungen der Unfallbeteiligten und möglicherweise auch die Krankenkassen, Krankenhäuser und Ärzte, die die Behandlung von Verletzun-

gen oder Folgeschäden verfolgen bzw. durchführen. Andererseits speichert der jeweilige Mobilfunkanbieter, welcher Vertragstyp mit der entsprechenden Person vereinbart wurde. Problematisch hierbei ist, die Datensätze aus den verschiedenen Datenquellen zu identifizieren, welche dieselbe Person referenzieren. Zwar ist jeder Datensatz in der Datenbank der jeweiligen Partei eindeutig über einen einzigartigen Identifikator referenzierbar, jedoch sind diese Identifikatoren in der Regel nur lokal gültig [VCV13]. Weiterhin handelt es sich bei den Daten der einzelnen Parteien um teilweise sehr sensible und persönliche Informationen, welche strikten Datenschutzvorschriften und Gesetzen unterliegen.[1] Außerdem ist der Schutz solcher personenbezogenen Daten notwendig, da durch Enthüllung dieser Daten Nachteile für die betroffene Person entstehen können: Zum einen stellt die Enthüllung der Daten einen Einschnitt in die Privatsphäre der Person dar. Zum anderen können die gewonnenen Informationen zur Benachteiligung der Person führen. Beispielsweise kann eine Person aufgrund der Veröffentlichung aller Krankheiten sozial oder beruflich ausgegrenzt werden oder eine enthüllte (E-Mail-)Adresse kann zum Erhalt unerwünschter Werbungs- oder Spam-Nachrichten führen.

Das oben beschriebene Szenario mit den daraus folgenden Problemen wird vom *Privacy-preserving-Record-Linkage (PPRL)* adressiert. PPRL bezeichnet den Prozess der Verbindung (Linkage) von Datensätzen (Records) aus verschiedenen Datenquellen. Das Ziel des PPRL ist es, die Datensätze bzw. Objekte zu identifizieren, welche demselben Realwelt-Objekt entsprechen. Zusätzlich soll bei diesem Prozess die Privatsphäre gesichert werden, weshalb beim Linkage-Prozess keine Erkennungsmerkmale oder sonstige Informationen enthüllt werden dürfen, mit deren Hilfe es möglich wäre, Objekte einer Datenquelle zu identifizieren oder Rückschlüsse auf sensible Daten zu ziehen. Als Ergebnis erzeugt das PPRL daher nur die übereinstimmenden Datensätze bzw. nur ausgewählte Attribute oder Eigenschaften dieser Datensätze.

Neben dem vorgestellten Szenario gibt es für das PPRL noch zahlreiche weitere Anwendungsgebiete und Einsatzmöglichkeiten: Grundsätzlich kann es überall dort zum Einsatz kommen, wo verschiedene autonome Parteien ihre Daten miteinander abgleichen wollen, vor allem um sie gegenseitig zu integrieren, die Datenqualität zu verbessern oder erweiterte Analysemöglichkeiten zu schaffen. Ein Anwendungsgebiet liegt im Bereich der Medizin und Gesundheitsfürsorge [Dur12, VCV13]. Hier sind verschiedene medizinische Versorgungseinrichtungen, wie Krankenhäuser und Arztpraxen, im Besitz von Patientendaten. Ein Zusammenführen der Daten aus diesen verschiedenen Quellen kann die Qualität der Patientenbehandlung verbessern, da dadurch alle Patientendaten verfügbar gemacht wer-

[1] Für Deutschland sind dies unter anderem das Allgemeine Persönlichkeitsrecht, das Bundesdatenschutzgesetz (BDSG), das Telemediengesetz (TMG), die Charta der Grundrechte der Europäischen Union (EGRC), die Datenschutz Grundverordnung (EU-DSGVO) sowie die europäische Menschenrechtskonvention (EMRK) [Buc12].

den. Wenn alle Patientendaten vorliegen, kann dies sogar zur Entdeckung von bisher unbekannten Korrelationen führen, beispielsweise indem eine Wechselwirkung von verschiedenen Medikamenten, welche vom Patienten eingenommen werden, aufgedeckt wird. Darüber hinaus sind Kosteneinsparungen möglich, da keine Untersuchung mehrfach veranlasst werden muss, um das Fehlen eines Befundes zu kompensieren.

Die Durchführung des PPRL wird allerdings durch verschiedene Probleme und Herausforderungen erschwert. Grundsätzlich lassen sich dabei drei wesentliche Herausforderungen unterschieden. Die Herausforderungen sind in Abbildung 1.1 dargestellt und betreffen die Skalierbarkeit, die Qualität und die Sicherheit (Privacy) [VCV13, VSCR17]. Alle diese Herausforderungen beeinflussen sich dabei gegenseitig, wodurch weitere Probleme entstehen. Zusätzlich stellen die Eigenschaften von *Big Data* eine große Herausforderung für das PPRL dar. Der Begriff Big Data lässt sich dabei durch fünf V's charakterisieren, welche *Volume*, *Velocity*, *Variety*, *Veracity* und *Value* sind [DGDLM13]. Zusammen stehen diese Eigenschaften einerseits für enorme Mengen an Daten (Volume), die als gewaltige Datenströme in sehr kurzer Zeit generiert und ausgewertet werden müssen (Velocity). Andererseits stammen die Daten entweder aus vielen unterschiedlichen Datenquellen und liegen dabei in verschiedenen Formaten vor oder sie weisen vielfältige Datentypen auf (Variety). Zudem sind die Daten von unterschiedlicher Qualität, das heißt, sie können Fehler, Abweichungen oder Inkonsistenzen aufweisen (Veracity). Schließlich soll aus der Analyse solcher Daten möglichst großer Mehrwert erzielt werden (Value) [DGDLM13]. Die konkreten Herausforderungen, die sich dadurch insgesamt für das PPRL ergeben, sind wie folgt:

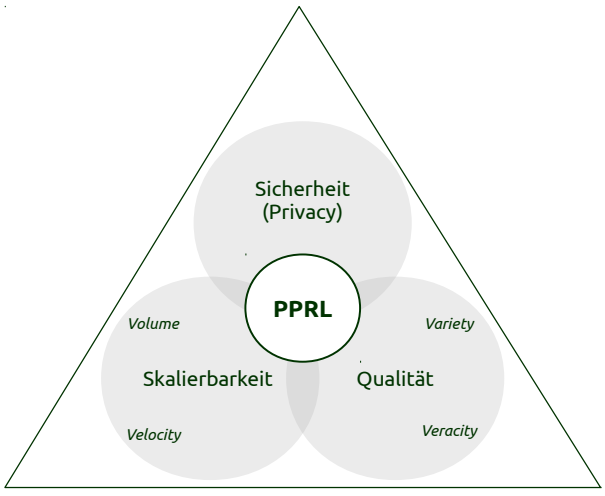


Abbildung 1.1: Herausforderungen beim PPRL.

Skalierbarkeit Wie bereits aus dem einleitenden Beispiel erkennbar ist, können potentiell sehr viele verschiedene Parteien am PPRL beteiligt sein. Hinzu kommt, dass da-

bei jede Partei sehr große Datenmengen mit mehreren Millionen Datensätzen verwalten kann. Derart große Datenmengen sind dabei keineswegs untypisch. Reale Beispiele hierfür sind unter anderem die Deutsche Telekom AG mit ca. 41,8 Millionen Mobilfunk-Kunden deutschlandweit [Deu], die Allianz Versicherung mit ca. 20 Millionen Kunden [All] oder die AOK Krankenkasse mit ca. 25,5 Millionen Versicherten [Bun]. Der naive Ansatz zur Durchführung des PPRL besteht darin, dass alle möglichen Datensatzpaare miteinander verglichen werden [VCV13]. Bereits für zwei Parteien würde dieses Vorgehen dazu führen, dass die Anzahl der notwendigen Vergleiche gleich dem Produkt der beiden Datenbankgrößen wäre. Enthalten beispielsweise beide Datenbanken 100.000 Datensätze, so würde dies bereits zu $100.000 \cdot 100.000 = 10^{10}$ Vergleichen führen. Der naive Vergleich aller möglichen Datensatzpaare stellt damit den Flaschenhals im PPRL-Prozess dar und ist in realen Anwendungen meist nicht durchführbar. Aus diesem Grund kommen Blocking-Verfahren zum Einsatz, deren Ziel es ist, den Suchraum möglichst weit einzuschränken, indem unähnliche Datensätze frühzeitig von weiteren Vergleichen ausgeschlossen werden [BCC+03].

Qualität Zur Identifizierung gleichartiger Objekte werden beim PPRL die codierten Datensätze aus den verschiedenen Datenquellen mit Hilfe von Ähnlichkeitsfunktionen miteinander verglichen [VCV13]. Die Berechnung der Ähnlichkeit basiert dabei auf dem Vergleich mehrerer Attribute der Datensätze. Solche Attribute sind zum Beispiel der Name, die Adresse oder das Alter einer Person [Dur12]. Dieser Vergleich wird jedoch dadurch erschwert, dass reale Daten oft Fehler und Variationen aufweisen (*Dirty Data*) [HS98] oder Attributwerte schlimmstenfalls gänzlich fehlen (vgl. Veracity). PPRL-Verfahren müssen aus diesem Grund eine gewisse Toleranz gegenüber solchen fehlerhaften Daten gewährleisten, sodass die Qualität der Ergebnisse gesichert wird.

Sicherheit (Privacy) Die Einhaltung des Datenschutzes sowie die Sicherung der Privatsphäre stellt die dritte wesentliche Herausforderung beim PPRL dar. Um diese Forderungen zu gewährleisten, müssen die Datensätze codiert (maskiert) werden. Hierbei muss jedoch sichergestellt werden, dass durch die Maskierung keine wesentlichen Eigenschaften der Daten, welche für das Linkage nötig sind, zerstört werden. Anderenfalls würde dies zu einer Minderung der Ergebnisqualität führen. Außerdem muss die Ähnlichkeitsberechnung auf den codierten Daten möglichst effizient sein, da sonst die Leistungsfähigkeit und die Skalierbarkeit des PPRL reduziert werden.

1.2 Ziele der Arbeit

Die grundlegende Herausforderung beim PPRL ist die Erreichung eines performanten, skalierbaren Linkage-Prozesses unter Erzielung einer hohen Genauigkeit und Qualität sowie gleichzeitiger Sicherung der Privatsphäre und des Datenschutzes. Trotz der Verwendung von Blocking-Verfahren zur Reduzierung der notwendigen Vergleiche bleibt der Berechnungsaufwand, vor allem für viele Parteien und mehrere Millionen Datensätze, sehr hoch. Das Ziel dieser Arbeit ist daher die Parallelisierung des PPRL durch die Untersuchung, Implementierung und Evaluation von parallelen PPRL-Verfahren.

Das *Parallel-Privacy-preserving-Record-Linkage (P3RL)* verspricht eine Verbesserung der Ausführungszeit proportional zur Anzahl der eingesetzten Rechner bzw. Prozessoren im Computer-Cluster [DBGH11, KL07, KKH+10]. Blocking-Verfahren können beim P3RL genutzt werden, um die zu vergleichenden Datensätze auf die Rechner im Cluster zu partitionieren und danach die Ähnlichkeitsberechnung für die einzelnen Blöcke parallel durchzuführen. Damit hat das Blocking weiterhin großen Einfluss, wodurch die Leistungsfähigkeit des P3RL ebenfalls vom Blocking abhängt. Aus diesem Grund liegt das Hauptaugenmerk dieser Arbeit auf der Anpassung und Umsetzung von Blocking-Verfahren für das P3RL.

In diesem Zusammenhang sollen als Blocking-Verfahren das *Locality-sensitive-Hashing (LSH)* [IM98, KL10, Dur12] und phonetisches Blocking [KV09, Chr12b] angepasst und umgesetzt werden. Außerdem sollen beide Techniken hinsichtlich ihrer Performanz und Qualität miteinander verglichen werden. Bezüglich der Privacy-Technik zur Maskierung der Daten sollen ausschließlich *Bloom-Filter* [Blo70, SBR09, SBR11] zur Anwendung kommen. Weiterhin wird das Linkage zunächst auf zwei Parteien beschränkt. Dabei soll das Linkage von einer vertrauenswürdigen dritten Partei, der sogenannten Linkage-Unit [VCV13] durchgeführt werden. Die Nutzung einer solchen Linkage-Unit ist bestens für das P3RL geeignet, da diese ein leistungsfähiges Computer-Cluster betreiben kann, auf welchem die parallelen Verfahren durchgeführt werden können. Schließlich soll die Umsetzung des P3RL mit Hilfe von *Apache Flink*[2] erfolgen. Als Cluster-Computing-Framework erleichtert Flink die Entwicklung von parallelen Programmen zur Ausführung auf Cluster-Umgebungen [TRH+15].

Zusammenfassend werden im Rahmen dieser Arbeit folgende Ziele verfolgt:

- Realisierung des P3RL unter Verwendung von Flink und unter Abdeckung des gesamten PPRL-Prozesses.
- Anpassung und Umsetzung von verschiedenen Blocking-Verfahren (LSH, phoneti-

[2] `http://flink.apache.org/`, Zugriff: 06.04.2017

sches Blocking) mit Berücksichtigung verschiedener Parameter bzw. Vorgehensweisen.

- Evaluierung der umgesetzten Verfahren unter Analyse verschiedener Metriken zur Beurteilung der Skalierbarkeit und der Qualität des Linkage-Prozesses.
- Entwicklung eines Frameworks mit einer flexiblen und erweiterbaren Architektur, die möglichst einfache und automatisierte Vergleichsmöglichkeiten von umgesetzten Verfahren für das P3RL ermöglicht.

1.3 Aufbau der Arbeit

Die nachfolgenden Kapitel der Arbeit gliedern sich wie folgt:

Kapitel 2 behandelt die theoretischen Grundlagen der Arbeit. Neben einer genauen Betrachtung des PPRL und der Grundlagen bezüglich der verwendeten Blocking-Verfahren beinhaltet dies die Beschreibung der Konzepte und der Funktionsweise von Flink. Des Weiteren werden verwandte Arbeiten vorgestellt und Unterschiede zu dieser Arbeit hervorgehoben.

Kapitel 3 stellt die Konzepte für die Umsetzung des P3RL unter Verwendung von Flink vor. Hierbei werden die realisierten Verfahren näher erklärt und veranschaulicht. Außerdem wird auf verschiedene Details bezüglich der Implementierung eingegangen.

Kapitel 4 beschäftigt sich mit der Evaluierung der entwickelten Verfahren. Hierbei werden die durchgeführten Testläufe beschrieben und es erfolgt die Auswertung und Analyse der Ergebnisse.

Kapitel 5 gibt eine Zusammenfassung der Arbeit und bietet Empfehlungen für weiterführende Schritte.

Kapitel 2

Grundlagen und verwandte Arbeiten

Das folgende Kapitel behandelt die theoretischen Grundlagen der Arbeit. Hierbei werden zunächst die Grundlagen zum Privacy-preserving-Record-Linkage näher erläutert. Außerdem werden Bloom-Filter vorgestellt und es wird gezeigt, wie diese im Rahmen des PPRL Anwendung finden. In Abschnitt 2.3 werden die Blocking-Verfahren eingeführt, welche für das P3RL umgesetzt werden sollen. Danach folgen die wesentlichen Konzepte, die für den Umgang mit Flink relevant sind. In Abschnitt 2.5 werden schließlich verwandte Arbeiten vorgestellt und gegen diese Arbeit abgegrenzt.

2.1 Privacy-preserving-Record-Linkage

2.1.1 Definition und Komplexität

Im Folgenden werden die Begriffe Record-Linkage (RL) und Privacy-preserving-Record-Linkage (PPRL) definiert und erläutert. Darüber hinaus wird die Komplexität des Problems untersucht. Abgesehen von den gekennzeichneten Ausnahmen beruhen die Definitionen und die zugehörigen Ausführungen auf [VCV13] und [FS69].

Record-Linkage Es seien $P_1, \ldots, P_p$ die p Eigentümer ihrer entsprechenden Datenbank $D_1, \ldots, D_p$. Die Eigentümer (Parteien) möchten ermitteln, welche ihrer Datensätze (Records) $r_a^1 \in D_1, \ldots, r_c^p \in D_p$ übereinstimmen, in dem Sinn, dass sie das gleiche Realwelt-Objekt referenzieren. Die Entscheidung, ob zwei oder mehrere Datensätze übereinstimmen, erfolgt durch Anwendung eines Entscheidungsmodells ε. Dieses ordnet Paare bzw. Mengen von Datensätzen den Klassen M, für übereinstimmende Datensätze (*Matches*) und U, für nicht übereinstimmende Datensätze (*Non-Matches*), zu.

Für $p = 2$ mit den Parteien P_1 und P_2 ergibt sich die Menge aller geordneten Paare von Datensätzen C als

$$C = D_1 \times D_2 = \{(x, y) : x \in D_1, y \in D_2\}. \tag{2.1}$$

Die Menge aller Matches M und die Menge aller Non-Matches U berechnet sich dann wie folgt:

$$M = \{(x, y) : x \overset{\varepsilon}{=} y, x \in D_1, y \in D_2\} \tag{2.2}$$

$$U = \{(x, y) : x \overset{\varepsilon}{\neq} y, x \in D_1, y \in D_2\}. \tag{2.3}$$

Hierbei wird durch $x \overset{\varepsilon}{=} y$ bzw. $x \overset{\varepsilon}{\neq} y$ gekennzeichnet, dass die Datensätze x und y unter dem Entscheidungsmodell als gleich (übereinstimmend) bzw. ungleich (nicht übereinstimmend) angesehen werden. Für $p > 2$ lassen sich diese Berechnungen erweitern, indem statt der Record-Paare (x, y) die Record-Mengen $(x, y, \ldots, z)$ mit $x \in D_1, y \in D_2, \ldots, z \in D_p$ betrachtet werden.

Aufgrund des Fehlens von eindeutigen Identifikatoren, welche über verschiedene Datenbanken hinweg gültig sind, basiert diese Klassifikation der Record-Paare bzw. Record-Mengen auf dem Vergleich von Attributwerten. Die hierfür genutzten Attribute müssen so gewählt werden, dass sie gemeinsam die eindeutige Identifikation eines Datensatzes ermöglichen. Wird diese Eigenschaft gewährleistet, so werden die Attribute als Quasi-Identifikatoren (QIDs) [MX07] bezeichnet.

Andere Bezeichnungen für das Record-Linkage sind unter anderem Data-Matching, Entity-Resolution oder Entity-Matching [KR10, Chr12a].

Privacy-preserving-Record-Linkage Das PPRL erweitert das RL durch die Anforderung, die Privatsphäre der von den Datensätzen referenzierten Realwelt-Objekte (Entitäten) zu schützen und zu erhalten. Dazu ist sicherzustellen, dass beim Linkage-Prozess nicht die Werte der einzelnen Datensätze offengelegt werden. Das bedeutet, dass keine Partei P_i die Werte eines Datensatzes $r^j \in D_j$ einer anderen Partei P_j mit $i \neq j$ erfahren darf. Dies schließt auch mit ein, dass keine Erkennungsmerkmale oder sonstige Informationen enthüllt werden dürfen, mit deren Hilfe es möglich wäre, Realwelt-Objekte zu identifizieren oder Rückschlüsse auf sensitive Informationen dieser zu ermöglichen. Alleinig für die übereinstimmenden Paare bzw. Mengen von Datensätzen der Klasse M sind die Parteien bereit, ausgewählte Attributwerte für die anderen Parteien oder eine externe

dritte Partei zu Analyse- oder Forschungszwecken offenzulegen.

Komplexität PPRL Die Komplexität des Record-Linkages bzw. Privacy-preserving-Record-Linkages ergibt sich aus der Anzahl der zu vergleichenden Record-Paare bzw. Record-Mengen. Jede Datenbank D_i der Partei P_i enthalte $n_i = |D_i|$ Datensätze. Der naive Ansatz, um alle übereinstimmenden Paare bzw. Mengen von Datensätzen zu finden, besteht darin, alle möglichen Paare bzw. Mengen von Datensätzen miteinander zu vergleichen. Sei zunächst $p = 2$. Dies entspricht dem Linkage mit den zwei Parteien P_1 und P_2. Wie aus Gleichung 2.1 ersichtlich wird, ergibt sich die Anzahl aller möglichen Paare von Records als kartesisches Produkt der Records der beiden zugehörigen Datenbanken D_1 und D_2. Demzufolge müssen hierbei $n_1 \cdot n_2$ Paare miteinander verglichen werden. Dies führt zu quadratischer Komplexität $O(\mathring{n}^2)$ in Abhängigkeit zu der durchschnittlichen Anzahl der Records $\mathring{n}$ in den beiden Datenbanken.

Für $p > 2$ müssen beim naiven Ansatz anstatt aller Record-Paare alle möglichen Record-Mengen miteinander verglichen werden. Alle möglichen Record-Mengen ergeben sich ebenso wie für $p = 2$ aus dem kartesischen Produkt der Records der Datenbanken aller am PPRL teilnehmenden Parteien und damit als $D_1 \times \ldots \times D_p$. Somit sind insgesamt $\prod_{i=1}^{p}(n_i) = (n_1 \cdot n_2 \cdot \ldots \cdot n_p)$ Vergleiche erforderlich. Dies führt zu einer exponentiellen Komplexität $O(\mathring{n}^p)$ in Abhängigkeit zu der durchschnittlichen Anzahl der Records $\mathring{n}$ in allen Datenbanken sowie zu der Anzahl der Parteien p. Hierbei ist jedoch zu beachten, dass der naive Ansatz zu vielen unnötigen Vergleichen führt. Ist bekannt, dass $x \neq y$ für das Record-Paar (x, y) mit $x \in D_i$, $y \in D_j$ und $i < j$ gilt, dann müssen alle Record-Mengen die dieses Paar enthalten, nicht mehr berücksichtigt werden, da sie zu keinem Match mehr führen können. Damit genügt es, alle Record-Paare (x, y) $\forall i, j : i \neq j$ mit $x \in D_i$, $y \in D_j$ zu vergleichen und daraus die übereinstimmenden Record-Mengen zu berechnen. Dieser paarweise Vergleich erfordert insgesamt $\binom{p}{2} \cdot \mathring{n}^2$ Vergleiche, was zu einer Komplexität von $O(p^2 \cdot \mathring{n}^2)$ führt.

Beispiel Im einleitenden Beispiel-Szenario aus Abschnitt 1.1 soll die Forschungsfrage *"Verursachen Personen mit einer Mobilfunk-Flatrate mehr Verkehrsunfälle?"* beantwortet werden. Um diese Frage zu beantworten, sollen im Folgenden zwei Parteien berücksichtigt werden: Partei A und Partei B. Partei A ist ein Mobilfunkanbieter, der verschiedene Informationen über seine Kunden in einer Datenbank D_A speichert. Tabelle 2.1 zeigt ein Beispiel für eine solche Datenbank. Gleiches gilt für die Partei B, welche eine Kfz-Versicherung darstellt und die Daten von Versicherten in der Datenbank D_B speichert (siehe Tabelle 2.2). Um die Forschungsfrage zu beantworten, muss bestimmt werden, welche Personen aus D_B mit der Eigenschaft 'Versicherungsfall = Verkehrsunfalls' gleichzeitig in der Datenbank D_A vorkommen und dabei 'Vertragstyp = Flatrate' gilt. Dies zu bestimmen ist jedoch nicht direkt möglich: Einerseits sind die beiden Primärschlüssel 'KNR'

und 'VID' nur lokal gültig, weshalb in beiden Datenbanken vorhandene QIDs, wie Name und Adresse, genutzt werden müssen. Andererseits darf keine der beiden Parteien aufgrund des Datenschutzes und der Einhaltung der Privatsphäre die Daten seiner Kunden bzw. Versicherten weitergeben oder offenlegen. Um dieses Problem zu lösen, können demzufolge PPRL-Verfahren eingesetzt werden. Durch das PPRL soll dabei nur die Anzahl von Unfallverursachern ermittelt werden, die eine Mobilfunk-Flatrate abgeschlossen haben. Die anderen Attribute werden nur für das Linkage benötigt und müssen nicht veröffentlicht werden. Die beschriebene Fragestellung lässt sich noch erweitern, beispielsweise indem man sich auf Unfälle mit bestimmten Folgen (zum Beispiel Schädel-Hirn-Trauma) beschränkt. Hierzu müsste eine weitere Partei C berücksichtigt werden, welche eine Gesundheitsdatenbank verwaltet, wie beispielsweise eine Krankenversicherung oder ein Krankenhausverbund.

KNR	**Vorname**	**Nachname**	**Alter**	**Adresse**	**Vertragstyp**
A1	Anna	Schmitt	18	Bahnhofstraße 7, 43110	Flatrate
A2	Bernd	Meier	66	Hauptstraße 24, 05752	Flatrate
A3	Christian	Koch	39	Mühlstraße 6, 9099	Zeittarif
A4	Klaus	Becker	42	Poststraße 14, 05099	Zeittarif

Tabelle 2.1: Beispiel: Datenbank D_A der Partei A (Mobilfunkanbieter).

VID	**Name**	**Geb.Dat.**	**Sex**	**Adresse**	**Versicherungsfall**
B1	Meier, Bernt	20.05.51	m	Hauptstr. 24a, 05752	Verkehrsunfall
B2	Schulz, Elke	30.02.80	w	Marktstr. 40, 62472	Diebstahl
B3	Jahn, Otto	07.10.69	-	Am Anger 32, 62877	Vandalismus
B4	Schmidt, Anna	19.03.99	w	Bahnhofstr. 7, 43110	Verkehrsunfall

Tabelle 2.2: Beispiel: Datenbank D_B der Partei B (Kfz-Versicherung).

Zur Bestimmung der Komplexität des PPRL für das Beispiel können erneut die zwei Datenbanken für die Parteien A und B entsprechend der Tabellen 2.1 und 2.2 betrachtet werden. Hierbei gilt $p = 2$ mit $n_A = n_B = 4$. Um alle Record-Paare miteinander zu vergleichen, sind insgesamt $n_A \cdot n_B = 4 \cdot 4 = 16$ Vergleiche notwendig. Für die Menge aller Record-Paare C gilt $C = \{(A1, B1), (A1, B2), (A1, B3), (A1, B4), (A2, B1), (A2, B2), \ldots, (A4, B3), (A4, B4)\}$. Nach der Klassifikation der Record-Paare könnte sich für die Mengen M und U folgende Zuordnungen ergeben: $M = \{(A1, B4), (A2, B2)\}$ und $U = C \backslash M$.

2.1.2 PPRL-Prozess

Der folgende Unterabschnitt erläutert den komplexen Prozess des PPRL. Die Ausführungen gründen sich dabei auf [VCV13] und [Chr12a].

In Abbildung 2.1 ist in Anlehnung an [VCV13] der grundlegende PPRL-Prozess für zwei Parteien dargestellt. Die einzelnen Schritte werden in den nachfolgenden Unterabschnitten näher betrachtet. Wichtig ist, dass sich die am PPRL teilnehmenden Parteien im Vorfeld auf diverse Parameter verständigen müssen. Jeder Schritt des PPRL erfordert bestimmte Parameter, die großen Einfluss auf die Qualität der Ergebnisse und die Leistungsfähigkeit des PPRL-Prozesses haben. Bei allen PPRL-Verfahren ist es notwendig, dass die Parteien gemeinsam festlegen, welche Attribute der Datensätze für das Linkage benutzt werden. Hierfür ist es zunächst erforderlich, die gemeinsamen Schema-Elemente innerhalb der Datenquellen der Parteien zu identifizieren (*Schema Matching* [BBR11]). Außerdem muss festgelegt werden, welche Verfahren bei der Durchführung des PPRL zum Einsatz kommen sollen. Weitere notwendige Parameter werden an entsprechender Stelle bzw. im entsprechenden Schritt eingeführt und erläutert. Der Unterschied zwischen dem PPRL im Gegensatz zum RL-Prozess besteht darin, dass in jedem Schritt die Sicherung der Privatsphäre berücksichtigt werden muss. Während die Vorverarbeitung noch lokal auf uncodierten Daten durchgeführt wird, wird in allen nachfolgenden Schritten auf codierten Daten gearbeitet.

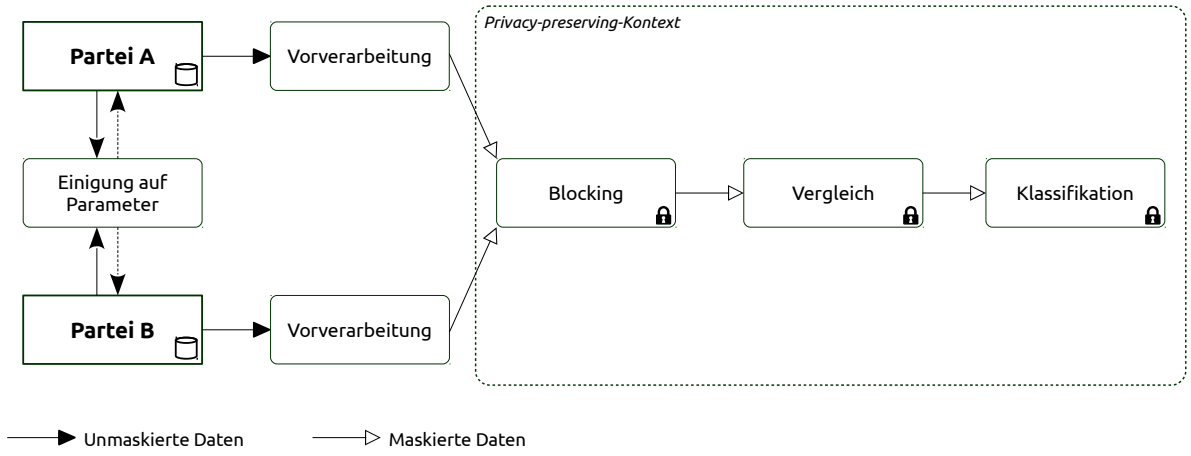


Abbildung 2.1: Grundlegender PPRL-Prozess für zwei Parteien.

2.1.2.1 Vorverarbeitung

Der erste Schritt des PPRL-Prozesses ist die Vorverarbeitung der Daten. Die Vorverarbeitung wird dabei unabhängig (lokal) von jeder Partei i durchgeführt ($\forall i \in \{1, \ldots, p\}$).

Dieser Schritt besteht dabei wiederum aus zwei Teilschritten, welche nachfolgend beschrieben werden.

Der ersten Schritt umfasst das *Data-Cleaning* sowie die *Standardisierung* der Daten. Dieser Schritt ist zunächst unabhängig von der gewählten Technik zur Erhaltung der Privatsphäre und wird ebenso beim klassischen Record-Linkage durchgeführt. Das grundlegende Ziel ist es, die Datenqualität und damit die Qualität des PPRL zu erhöhen [RFBS13]. Als Data-Cleaning wird die *Säuberung* der Daten bezeichnet. Diese Bezeichnung ist darauf zurückzuführen, dass reale Daten oft Fehler bzw. Variationen aufweisen oder unvollständig sind (z. B. fehlende Attributwerte aufweisen) [RD00], weshalb man von *Dirty Data* [HS98] spricht. Diese Datenqualitätsprobleme sollen mit dem Data-Cleaning behoben oder zumindest abgemildert werden. Das wird erreicht, indem beispielsweise fehlende Werte ausgefüllt, ungewollte Werte entfernt oder Inkonsistenzen in der Datenrepräsentation und Codierung aufgelöst werden. Neben dem Data-Cleaning werden die Daten außerdem standardisiert. Das bedeutet, dass sich die Parteien auf gemeinsame Standards bezüglich der Repräsentation und Codierung der Daten einigen, ihre Daten entsprechend anpassen und damit in eine wohldefinierte und konsistente Form bringen. Auch wenn die Trennung dieser beiden Schritte in der Praxis nicht so strikt erfolgen muss, ist es so, dass sich das Data-Cleaning und die Standardisierung verschiedenen Datenqualitätsproblemen widmen. So ist das Ziel des Data-Cleanings, Single-Source-Datenqualitätsprobleme [RD00] zu lösen. Gemeint sind hiermit Probleme, die bereits in einer Datenquelle vorhanden sind, wie beispielsweise fehlerhafte Datenwerte. Auf der anderen Seite sollen durch die Standardisierung Multi-Source-Datenqualitätsprobleme [RD00] gelöst werden. Diese Probleme entstehen erst durch Betrachtung mehrerer Datenquellen und liegen meist heterogenen Datenmodellen und verschiedenen Repräsentationsmöglichkeiten zugrunde. Für weitere Aspekte der Datenqualität und damit einhergehende Probleme sei auf [BS06] sowie [RD00] und [Chr12a] verwiesen.

Zur Verdeutlichung der Notwendigkeit dieser beiden Schritte können erneut die Beispiel-Datenbanken aus Tabelle 2.1 und 2.2 betrachtet werden. Hierbei fallen unter anderem folgende Probleme auf:

- Fehlende Werte:
  - Fehlendes Geschlecht bei Datensatz B3
- Unzulässige bzw. fehlerhafte Werte:
  - Fehlende Zahl bei der Postleitzahl von Datensatz A3
  - Ungültiger Datumswert beim Geburtsdatum von Datensatz B2
- Unterschiede in der Repräsentation der Daten:

  - Abkürzung von Straßennamen in D_B
  - Ein Attribut Name in D_B im Gegensatz zur Trennung zwischen Vor- und Nachname in D_A
  - Alter vs. Geburtsdatum
- Phonetische Variationen von Namen:
  - Schmitt vs. Schmidt
  - Bernd vs. Bernt.

Der zweite Schritt der Vorverarbeitung umfasst die *Maskierung* bzw. *Codierung* der Daten. Dieser Schritt ist wesentlich für das PPRL, da es die Wahrung der Privatsphäre sicherstellt. Ziel dieses Schrittes ist es, die Datensätze so zu codieren bzw. pseudonymisieren, dass einerseits die anderen Parteien oder ein Angreifer keine sensitiven Informationen aus den codierten Daten erhalten können und andererseits, dass bestimmte funktionale Beziehungen und Eigenschaften zwischen den codierten Daten und den Ursprungsdaten erhalten bleiben [Fie05]. Die zweite Forderung ist dabei von großer Bedeutung für die Durchführbarkeit des Linkages und zudem essentiell für die Qualität der Ergebnisse des PPRL. Auch bei diesem Schritt muss das Problem der Dirty Data berücksichtigt werden. Aus diesem Grund eignen sich kryptologische Hashfunktionen [KCB97, Bru96] nicht für das PPRL, da im Allgemeinen schon die Änderung eines Zeichens zu einem anderen Hashwert führt. Hiermit lassen sich demnach nur die Korrespondenzen finden, welche in jedem berücksichtigten Attributwert exakt übereinstimmen [DQB95]. Für das PPRL wurden daher verschiedene Techniken zur Codierung der Daten und damit zur Sicherstellung der Privatsphäre untersucht und entwickelt (siehe [VCV13]). Im Rahmen dieser Arbeit sollen *Bloom-Filter* [Blo70] als Privacy-Technik und damit zur Maskierung eingesetzt werden. Die Behandlung von Bloom-Filtern erfolgt in Abschnitt 2.2.

Insgesamt bewirkt die Vorverarbeitung der Daten die Überführung aller Datensätze $r \in D_i$ in ihre entsprechende maskierte bzw. codierte Repräsentation $\tilde{r}$. Damit wird gleichzeitig jede Datenquelle D_i in $\tilde{D}_i$ überführt, wobei $\tilde{D}_i$ alle maskierten Datensätze der Partei i enthält.

Zur Durchführung der Vorverarbeitung ist es notwendig, dass sich die Parteien vorab auf ein einheitliches Vorgehen bezüglich der Schritte der Vorverarbeitung einigen. Außerdem muss festgelegt werden, welche Attribute für das Linkage benutzt werden sollen sowie welche Maskierungstechnik zum Einsatz kommen soll. Weitere Parameter ergeben sich in Abhängigkeit von der gewählten Maskierungstechnik.

2.1.2.2 Blocking

Der zweite Schritt im PPRL-Prozess ist das *Blocking*, der teilweise auch als *Indexing* oder *Filterung* bezeichnet wird [BCC+03, Chr12b, VSCR17]. Ziel des Blockings ist es, die Anzahl der notwendigen Record-Vergleiche zu reduzieren. Wie in Unterabschnitt 2.1.1 gezeigt, ergibt sich im Fall von $p = 2$ die Anzahl aller möglichen Record-Paare aus dem kartesischen Produkt der Datensätze aus den zwei Datenbanken (Gleichung 2.1). Das heißt, dass jeder Datensatz aus der Datenbank der einen Partei mit jedem Datensatz aus der Datenbank der anderen Partei verglichen werden muss ($O(\mathring{n}^2)$). Dieser Schritt stellt damit den Flaschenhals im PPRL dar [CG07], auch weil die Vergleiche in Abhängigkeit der Privacy-Technik aufwändig sein können. In Abschnitt 1.1 wurde bereits berechnet, dass sich für zwei Datenbanken mit jeweils 100.000 Records 10^{10} notwendige Vergleiche ergeben. Selbst wenn ein Rechner 100.000 Vergleiche pro Sekunde durchführen kann, so wären $10^5 s \approx 1667 min \approx 28 h$ notwendig, um alle Vergleiche durchzuführen. Klar ist, dass sehr viele dieser Vergleiche nicht zu Matches führen können, da unter der Bedingung, dass die Datenbanken duplikatfrei sind und jeder Record genau eine Korrespondenz in der anderen Datenbank hat, insgesamt maximal 100.000 Matches gefunden werden können. Das Blocking dient daher dazu, möglichst viele Record-Paare, welche mit hoher Wahrscheinlichkeit nicht übereinstimmen (bzw. unähnlich sind) von einem genauen Vergleich auszuschließen [BCC+03]. Die verbleibenden Paare von Datensätzen werden als *Kandidaten-Record-Paare* bezeichnet. Diese werden im Detail miteinander verglichen, da die Wahrscheinlichkeit, dass es sich bei diesen um einen Match handelt, höher ist. Für die Menge der Kandidaten-Record-Paare C_{Kand} sollte $C_{Kand} \subsetneq C$ sowie $|C_{Kand}| \ll |C|$ gelten. Das Blocking sollte demnach den Suchraum C aller möglichen Record-Paare deutlich reduzieren. Außerdem muss gelten, dass die Zeit oder der Aufwand, der für das Blocking benötigt wird, wesentlich kleiner ist, als die Zeit oder der Aufwand der notwendig wäre, um alle möglichen Record-Paare C zu vergleichen [Dur12]. Diese Forderung stellt sicher, dass die Gewinne, die sich durch eingesparte Vergleiche ergeben, nicht von zu aufwändigen Blocking-Verfahren neutralisiert werden. Im Gegensatz zum RL, wo das Blocking ebenfalls eingesetzt wird, ist das Blocking beim PPRL abhängig von der Codierung der Daten, also der Privacy-Technik. Zudem muss natürlich das Blocking ebenfalls die Privatsphäre schützen und darf keine sensitiven Informationen verarbeiten.

Insgesamt hat das Blocking damit großen Einfluss auf die Leistungsfähigkeit und Skalierbarkeit des gesamten PPRL. Da der Schwerpunkt dieser Arbeit auf der Verbesserung der Skalierbarkeit des PPRL durch Erweiterung auf das P3RL liegt, wird daher auch dem Blocking eine besondere Bedeutung beigemessen.

Wie eingangs erwähnt, existieren für das Blocking verschiedene Bezeichnungen. Diese

resultieren daraus, dass eine Vielzahl an Blocking-Methoden für das RL und PPRL entwickelt wurden und diese teilweise verschiedenen Grundideen folgen. Eine Übersicht über die verschieden Verfahren bieten unter anderem [VCV13], [VSCR17] und [Chr12a].

Im Rahmen dieser Arbeit ist vor allem das traditionelle Standard-Blocking [FS69] von Bedeutung. Hierbei wird ein sogenannter *Blocking-Key* definiert. Der Blocking-Key BK wird aus einem oder mehreren Attributen oder aus Segmenten von Attributen konstruiert. Damit stellt der Blocking-Key ein Ähnlichkeitskriterium dar, anhand dessen die Menge aller Records $R = \bigcup_{i=1}^{p}(\tilde{D}_i)$ in b disjunkte Blöcke $B_1, \ldots, B_b$ mit $B_i \subseteq R$ und $B_i \cap B_j = \emptyset \quad \forall i, j : i \neq j \wedge i, j \in \{1, \ldots, b\}$ partitioniert wird, sodass gilt $\forall x, y \in B_i : BK(x) = BK(y)$. Für die Anzahl der resultierenden Blöcke b gilt, dass maximal so viele Blöcke erzeugt werden können, wie Wertkombinationen für den Blocking-Key existieren. Die einzelnen Blöcke B sollen ähnliche Datensätze, also solche, die in ihrem Blocking-Key übereinstimmen, zusammenbringen. Die Kandidaten-Record-Paare werden dann nur innerhalb der einzelnen Blöcke erzeugt, sodass

$$C_{Kand} = \bigcup_{i=1}^{b} \{(x, y) : x, y \in B_i \wedge x \neq y \wedge (x \in \tilde{D}_j \Rightarrow y \notin \tilde{D}_j)\}. \tag{2.4}$$

Im Idealfall enthält jeder der b Blöcke $\frac{\bar{n}}{b}$ Datensätze, wobei $\bar{n}$ die Summe aller Datensätze der Parteien ist. Pro Block ergeben sich dann $O\left((\bar{n}/b)^2\right)$ Vergleiche. Dies führt zu insgesamt $O\left(\bar{n}^2/b\right)$ Vergleichen.

Für das laufende Beispiel mit den beiden Beispiel-Datenbanken aus Tabelle 2.1 und 2.2 könnte man das Blocking vornehmen, indem der erste Buchstabe des Nachnamens als Blocking-Key gewählt wird. Dies ergibt die Werte 'S', 'M', 'K', 'B' und 'J' als Blocking-Key für die angegebenen Datensätze. Damit ist $b_{\max} = 26$ (Buchstaben A – Z), aber tatsächlich gilt $b = 5$. Statt $|C| = 16$ notwendigen Vergleichen ohne Blocking ergeben sich damit nur noch $|C_{Kand}| = |\{(A1, B2), (A1, B4), (A2, B1)\}| = 3$ Kandidaten-Record-Paare und damit Vergleiche unter Verwendung des Blockings.

Auch beim Blocking wirkt sich das Problem der Dirty Data negativ aus. Konnten Fehler während der Vorverarbeitung der Daten nicht gefunden bzw. korrigiert werden, kann sich dies auf den Blocking-Key auswirken. Angenommen der erste Buchstabe des Nachnamens ist in einem Datensatz verändert (z. B. könnte durch einen Eingabefehler der Nachname im Beispiel-Datensatz A1 anstelle von '**S**chmitt' zu '**D**chmitt' verfälscht sein). Das führt dazu, dass sich der Blocking-Key von Datensatz A1 entsprechend zu 'D' ändert. Dies hat dann zur Folge, dass der Datensatz A1 einem anderen Block zugeordnet wird und dadurch die Übereinstimmung mit dem Datensatz B4 nicht aufgedeckt werden kann. In diesem Fall wird das Resultat als *false-negative* bezeichnet, da A1 und B4 fälschlicherweise als Non-

Match klassifiziert werden. Aus diesem Grund ist es von Vorteil, wenn der Blocking-Key bestimmte Fehler toleriert. In diesem Zusammenhang lässt sich ein Trade-off zwischen der Komplexität, also dem Grad der Reduktion des Suchraums durch das Blocking, und der Qualität der Linkage-Ergebnisse erkennen [BCC+03]. Wählt man den Blocking-Key zu unspezifisch, so ergibt sich hierdurch eine geringe Anzahl an Blöcken b. Das hat zur Folge, dass die einzelnen Blöcke sehr groß werden, das heißt viele Datensätze enthalten. Dieser Effekt wird mit zunehmender Datengröße immer weiter verstärkt. Diese wenigen, großen Blöcke führen letztlich zu mehr Kandidaten-Record-Paaren und damit zu mehr Vergleichen. Die Wahrscheinlichkeit, Record-Paare aufgrund von Datenfehlern nicht als Match zu klassifizieren, ist dadurch allerdings geringer [Chr12b]. Wählt man den Blocking-Key andererseits zu spezifisch, dann steigt die Anzahl der Blöcke. Das wiederum führt dazu, dass weniger Datensätze den einzelnen Blöcken zugeordnet werden, wodurch entsprechend weniger Kandidaten-Record-Paare erzeugt werden. Hierbei ist aber die Wahrscheinlichkeit höher, übereinstimmende Datensätze nicht als Match zu klassifizieren, was die Qualität des PPRL negativ beeinflusst.

Im Rahmen dieser Arbeit soll als Blocking-Verfahren zum einen das phonetische Blocking und zum anderen das Locality-sensitive-Hashing (LSH) umgesetzt und angewandt werden. Die Beschreibung der Verfahren wird in Unterabschnitt 2.3.2 bzw. in Unterabschnitt 2.3.1 vorgenommen.

2.1.2.3 Vergleich

Der nächste Schritt im PPRL-Prozess ist der *Vergleich* der Kandidaten-Record-Paare. Hierbei werden die zuvor bestimmten Kandidaten-Record-Paare unter Anwendung von Vergleichs- bzw. Ähnlichkeitsfunktionen detailliert miteinander verglichen. Eine solche Funktion erhält als Eingabe ein Kandidaten-Record-Paar und liefert meist einen numerischen Wert im Bereich $[0, 1]$. Der Wert 0 bedeutet dabei, dass sich die beiden Datensätze (bzw. ihre entsprechende maskierte Repräsentation) vollständig unähnlich sind. Der Wert 1 bedeutet hingegen, dass sich die beiden Datensätze vollständig ähnlich sind. Die verwendete Vergleichsfunktion ist abhängig von der gewählten Privacy-Technik zur Maskierung der Daten. Insgesamt existiert eine Vielzahl solcher Vergleichsfunktionen (siehe [Chr12a] oder [VCV13]). Im Rahmen dieser Arbeit kommen Bloom-Filter als Privacy-Technik zum Einsatz. Wie diese miteinander verglichen werden können und welche Ähnlichkeits- bzw. Distanzmaße für Bloom-Filter existieren, wird in Abschnitt 2.2 behandelt.

2.1.2.4 Klassifikation

Der letzte Schritt im PPRL-Prozess ist die *Klassifikation*. In diesem Schritt findet die Zuordnung der Kandidaten-Record-Paare zu den beiden Mengen bzw. Klassen von übereinstimmenden Datensätzen (*Matches*) M und nicht übereinstimmenden Datensätzen (*Non-Matches*) U statt (siehe Unterabschnitt 2.1.1). Auch hier gibt es mehrere Möglichkeiten, wie diese Klassifizierung vorgenommen wird. Im Rahmen dieser Arbeit erfolgt die Klassifizierung durch Nutzung eines Schwellwerts t. Diese Art der Klassifizierung wird sehr häufig eingesetzt [VCV13] und erfolgt für ein Kandidaten-Record-Paar $(x, y) \in C_{Kand}$ mit ihrer entsprechenden Ähnlichkeit, gekennzeichnet durch $Sim(\cdot, \cdot)$, wie folgt:

$$Sim(x, y) < t \quad \Rightarrow \quad (x, y) \in U \tag{2.5}$$

$$Sim(x, y) \geq t \quad \Rightarrow \quad (x, y) \in M \tag{2.6}$$

Für andere Verfahren zur Klassifizierung der Kandidaten-Record-Paare sei auf [VCV13] verwiesen.

2.1.3 Evaluierung

Die Evaluierung des PPRL kann als zusätzlicher Schritt im PPRL-Prozess angesehen werden. Allerdings ist in realen Anwendungen die Bewertung der Qualität der Ergebnisse durch die Sicherung der Privatsphäre sehr schwierig [BMSB09, CG07]. Dies liegt darin begründet, dass statt der Ausgangs-Datensätze nur die entsprechend codierten Datensätze eingesehen werden können. Außerdem erfolgt die Evaluierung vor dem operationalen Betrieb und wird aus diesen Gründen hier eigenständig betrachtet. Das PPRL kann anhand von drei wesentlichen Kriterien evaluiert werden, diese sind nach [Vat14, VCOV14, VCV13]:

1. Skalierbarkeit
2. Qualität
3. Privatsphäre (Privacy).

Wie diese drei Kriterien evaluiert werden können, wird in den folgenden Unterabschnitten erläutert. Die Ausführungen beruhen dabei auf [CG07], [VCV13], [VCOV14], [VSCR17] und [EVE02].

2.1.3.1 Skalierbarkeit

Der erste Aspekt, welcher bei der Evaluierung des PPRL berücksichtigt wird, ist die Skalierbarkeit. Hierbei ist vor allem die Berechnungskomplexität der eingesetzten Techniken bzw. Verfahren von Bedeutung. Wie in Unterabschnitt 2.1.1 gezeigt und in Unterunterabschnitt 2.1.2.2 weiter erläutert, bestimmt im Wesentlichen die Anzahl der Kandidaten-Record-Paare $|C_{Kand}|$ die Komplexität des PPRL. Dabei werden Blocking-Verfahren eingesetzt, um den Suchraum, bestehend aus allen möglichen Record-Paaren C, möglichst umfassend einzuschränken. Das eingesetzte Blocking-Verfahren hat damit großen Einfluss auf die Leistungsfähigkeit und Skalierbarkeit eines PPRL-Verfahrens. Die Effizienz des Blocking-Verfahrens kann mit Hilfe der *Reduction-Rate* (RR) [EVE02] gemessen werden. Die Berechnung der Reduction-Rate erfolgt nach Gleichung 2.7. Die Reduction-Rate hat den Vorteil, dass kein Wissen über die einzelnen Datensätze benötigt wird. Stattdessen ist es nur notwendig zu bestimmen, wie viele Datensätze jede Partei als Eingabe übermittelt hat und wie hoch die Anzahl der Kandidaten-Record-Paare ist, die das PPRL-Verfahren erzeugt. Damit kann diese Metrik auch unter Einhaltung der Privatsphäre evaluiert werden.

$$RR = 1 - \frac{|C_{Kand}|}{|C|} \tag{2.7}$$

Die Reduction-Rate bestimmt demnach, inwieweit das Blocking-Verfahren den Suchraum einschränken konnte, das heißt, wie viele Kandidaten-Record-Paare im Verhältnis zu der Anzahl aller möglichen Record-Paare generiert wurden. Eine niedrige Reduction-Rate bedeutet, dass das eingesetzte Blocking-Verfahren ineffizient ist und keine ($RR = 0$) oder nur verhältnismäßig wenig Datensatz-Paare vom Vergleich ausgeschlossen wurden. Ist die Reduction-Rate hingegen sehr hoch ($RR \approx 1$), so war das Blocking sehr effizient und es konnten nahezu alle unnötigen Vergleiche von Record-Paaren eingespart werden.

Für das mitlaufende Beispiel und dem Blocking-Ansatz aus Unterunterabschnitt 2.1.2.2 ergibt sich mit den Werten $|C| = 16$ und $|C_{Kand}| = 3$ eine Reduction-Rate von $RR = 1 - \frac{3}{16} = 1 - 0.1875 = 0.8125$.

Neben der Reduction-Rate können noch weitere Metriken bezüglich der Komplexität des PPRL berücksichtigt werden. Diese sind jedoch unter anderem abhängig von der genauen Spezifikation der eingesetzten Rechner, der Computerplattform sowie der vorhandenen Netzwerkinfrastruktur [VCV13]. Weitere Metriken nach [VCV13] sind beispielsweise

- Laufzeit
- Speicherverbrauch

- Kommunikationskosten.

2.1.3.2 Qualität

Die Bestimmung der Qualität eines PPRL-Verfahrens ist ebenfalls von großer Bedeutung für dessen praktische Einsatzfähigkeit. Wie bereits erwähnt, wird das PPRL durch das Problem der Dirty Data erschwert. Daher ist eine Anforderung an das PPRL eine gewisse Toleranz gegenüber solchen unsauberen Daten zu erreichen. Dies bedeutet, dass das PPRL in der Lage sein soll, in Wirklichkeit übereinstimmende Datensätze als solche zu erkennen, auch wenn diese zu einem gewissen Grad verschmutzt (unsauber) sind, also beispielsweise Fehler oder Inkonsistenzen aufweisen. Des Weiteren sollen wesentliche Eigenschaften und Beziehungen zwischen den Datensätzen auch nach der Maskierung der Datensätze erhalten bleiben, sodass die Durchführbarkeit eines präzisen Linkages gewährleistet bleibt. Damit einher geht, dass sich übereinstimmende Datensätze deutlich von nicht übereinstimmenden Datensätzen abheben. Schließlich soll das eingesetzte Blocking-Verfahren zwar möglichst viele unnötige Vergleiche verhindern, aber dennoch sollten keine oder nur so wenig wie möglich übereinstimmende Record-Paare von einem genauen Vergleich ausgeschlossen werden.

Um die Anforderungen an die Qualität des PPRL zu präzisieren, ist eine genauere Betrachtung der Klassifizierung von Record-Paaren notwendig. Wie in Unterabschnitt 2.1.1 erläutert, klassifiziert das PPRL-Verfahren jedes Record-Paar in genau eine der beiden Mengen U oder M. Die Menge U enthält dabei die vom PPRL-Verfahren als Non-Match (nicht übereinstimmend) klassifizierten Record-Paare. Hingegen enthält die Menge M die vom PPRL-Verfahren als Match (übereinstimmend) klassifizierten Record-Paare.

Sei nun $\hat{M}$ die Menge der in Wirklichkeit übereinstimmenden Record-Paare (bzw. Record-Mengen). Weiter sei $\hat{U}$ die Menge der in Wirklichkeit *nicht* übereinstimmenden Record-Paare (bzw. -Mengen). Dann kann die Klassifikation der Record-Paare in M und U nach [Faw04] in vier Gruppen unterteilt werden. Diese vier Gruppen sind in Tabelle 2.3 dargestellt.

Klassifikation / Wirklichkeit	Match (M)	Non-Match (U)
Match ($\hat{M}$)	True-Match *true-positive* (TP)	False-non-Match *false-negative* (FN)
Non-Match ($\hat{U}$)	False-Match *false-positive* (FP)	True-non-Match *true-negative* (TN)

Tabelle 2.3: Konfusionsmatrix bezüglich der Klassifikation von Record-Paaren.

Wie daraus ersichtlich wird, wird ein korrekt als Match klassifiziertes Record-Paar als True-Match oder *true-positive* bezeichnet. Ein Record-Paar, welches korrekt als Non-Match klassifiziert wird, erhält die Bezeichnung True-non-Match bzw. *true-negative*. Wird ein in Wirklichkeit übereinstimmendes Record-Paar (aus $\hat{M}$) fälschlicherweise als Non-Match (in U) klassifiziert, so wird dieses als False-non-Match oder *false-negative* bezeichnet. Schließlich wird ein in Wirklichkeit nicht übereinstimmendes Record-Paar (aus $\hat{U}$), welches fälschlicherweise als Match (in M) klassifiziert wurde, als False-Match bzw. *false-positive* bezeichnet. Diese vier Gruppen erzeugen dabei jeweils die in Klammern angegeben Mengen. Dementsprechend ist TP die Menge aller True-Matches, FN die Menge aller False-non-Matches, FP die Menge aller False-Matches und TN die Menge aller True-non-Matches. Damit ergeben sich folgende Zusammensetzungen für die Mengen M und U bzw. $\hat{M}$ und $\hat{U}$:

$$M = TP \cup FP \tag{2.8}$$

$$U = FN \cup TN \tag{2.9}$$

$$\hat{M} = TP \cup FN \tag{2.10}$$

$$\hat{U} = FP \cup TN \tag{2.11}$$

Um eine hohe Qualität zu erreichen, sollte das PPRL-Verfahren dementsprechend idealerweise möglichst kleine Werte für $|FP|$ und $|FN|$ erzeugen.

Zur Evaluierung der Qualität wurden zahlreiche Metriken auf Basis obiger Überlegungen entwickelt. Im Rahmen dieser Arbeit sind vor allem die *Pairs-Completeness* [EVE02], die *Pairs-Quality* [VCV13] und *F-Measure* [BYRN99] von Bedeutung. Die Berechnung dieser Qualitätsmetriken erfolgt entsprechend den Gleichungen 2.12, 2.13 sowie 2.14. Die Pairs-Completeness entspricht dabei dem Anteil der Record-Paare, welche korrekt als Match klassifiziert wurden, im Verhältnis zu der Anzahl aller in Wirklichkeit übereinstimmenden Record-Paare. Die Pairs-Completeness gibt demzufolge an, wie viele von den in Wirklichkeit existenten Matches gefunden wurden. Sie ist gleich dem Recall (Sensitivität) [BYRN99]. Die Pairs-Quality entspricht dem Anteil der Record-Paare, welche korrekt als Match klassifiziert wurden, im Verhältnis zu der Anzahl aller als Match klassifizierten Record-Paare. Damit gibt die Pairs-Quality an, wie viele der gefundenen Matches, auch in Wirklichkeit übereinstimmen. Die Pairs-Quality ist gleich der Precision (Genau-

igkeit) [BYRN99]. F-Measure ist das harmonische Mittel zwischen der Pairs-Quality und der Pairs-Completeness. F-Measure produziert genau dann hohe Werte, wenn sowohl die Pairs-Quality, als auch die Pairs-Completeness hohe Werte liefern. Für weitere Metriken, die zur Evaluierung der Qualität des PPRL zum Einsatz kommen, sei auf [CG07] verwiesen.

Das Problem bei der Evaluierung der Qualität des PPRL ist, dass in realen Anwendungen die Mengen $\hat{M}$ und $\hat{U}$ unbekannt sind. Weiterhin können auch die Klassen TP, TN, FP und FN nicht bestimmt werden, da dazu Einsicht in die uncodierten Records bzw. deren Attributwerte notwendig wäre. Dies ist aufgrund der Einhaltung des Datenschutzes sowie der Privatsphäre nicht möglich. Darüber hinaus wäre die manuelle Untersuchung der Datensätze bei großem Datenvolumen sehr aufwändig. Aus diesem Grund können die Qualitätsmetriken in realen Anwendungen nicht ausgewertet werden. Jedoch können sie im Zusammenhang mit Trainings- bzw. Testdaten genutzt werden, wodurch sie beim Vergleich verschiedener Ansätze für das PPRL helfen.

$$PC = \frac{|TP|}{|\hat{M}|} = \frac{|TP|}{|TP| + |FN|} \tag{2.12}$$

$$PQ = \frac{|TP|}{|M|} = \frac{|TP|}{|TP| + |FP|} \tag{2.13}$$

$$FM = \frac{2 \cdot PC \cdot PQ}{PC + PQ} \tag{2.14}$$

2.1.3.3 Privacy

Der dritte Aspekt bei der Evaluierung eines PPRL-Verfahrens betrifft die Analyse, inwieweit die Privatsphäre (Privacy) und der Datenschutz eingehalten werden. Dieser Aspekt ist damit entscheidend für die Praktikabilität des PPRL, denn ohne die Einhaltung und den Schutz der Privatsphäre würde das PPRL seinen Zweck bzw. seine Funktion nicht erfüllen. Bei der Evaluierung der Privacy sind nach [VSCR17] im Wesentlichen vier Dimensionen zu berücksichtigen:

(1) Anzahl der Parteien mit deren Rolle,

(2) Angriffsmodelle (Adversary-Models),

(3) Privacy-Technik zur Maskierung der Daten und

(4) Angriffsarten (Attacks).

Darüber hinaus wurden verschiedene Privacy-Metriken entwickelt, um die Vergleichbarkeit von verschiedenen PPRL-Verfahren zu verbessern. Diese Privacy-Metriken messen dabei, inwieweit die Privatsphäre von einem PPRL-Verfahren geschützt bzw. gesichert wird. Da der Fokus dieser Arbeit auf der Verbesserung der Skalierbarkeit des PPRL liegt, erfolgt im Rahmen dieser Arbeit keine vollständige oder formale Analyse und Evaluierung der Privacy der eingesetzten Verfahren. Eine solche Analyse ist außerdem, wie nachfolgend noch weiter ausgeführt wird, sehr komplex und wird durch verschiedene Probleme erschwert. Aus diesem Grund wird auf die Einführung von konkreten Privacy-Metriken verzichtet und stattdessen auf [VCOV14] und [VCV13] verwiesen. Im Folgenden werden die Dimensionen, welche bei der Evaluierung der Privacy berücksichtigt werden müssen, erläutert.

Anzahl der Parteien PPRL-Verfahren werden danach unterteilt, für wie viele Parteien p sie die Durchführung des Linkages ermöglichen. Generell wird hierbei zwischen dem Linkage zwischen zwei Parteien ($p = 2$) und mehr als zwei Parteien ($p > 2$) differenziert. Zusätzlich wird berücksichtigt, wie das Linkage zwischen den Parteien durchgeführt wird: Entweder das Linkage wird von den Parteien selbst durchgeführt oder aber eine unabhängige und vertrauenswürdige dritte Partei, die sogenannte *Linkage-Unit* [VCV13], führt das Linkage durch. Anhand dieser Unterscheidungsmerkmale werden die PPRL-Verfahren entsprechend der Tabelle 2.4 in verschiedene Protokolltypen kategorisiert.

Linkage-Unit / Anzahl Parteien	Ja	Nein
$p = 2$	Two-Party-Protokoll	Three-Party-Protokoll
$p > 2$	Multi-Party-Protokoll	

Tabelle 2.4: Protokolltypen von PPRL-Verfahren.

Angriffsmodelle Die Sicherheit von PPRL-Verfahren wird unter der Annahme von verschiedenen Angriffsmodellen evaluiert. Ein Angriffsmodell, welches häufig angenommen wird, ist das *Honest-But-Curious*-Modell. Hierbei folgen die Parteien dem Protokoll (honest), jedoch versuchen sie möglichst viele Rückschlüsse und Informationen über die Original-Datensätze der anderen Parteien zu erhalten [HF10, LP09]. Dies schließt mit ein, dass sich zwei oder mehr Parteien zusammenschließen könnten, um sensitive Informationen über die Original-Datensätze der anderen Parteien zu erhalten. Ein solcher Zusammenschluss wird als Kollusion bezeichnet und kann auch mit der Linkage-Unit eingegangen werden.

Ein weiteres Angriffsmodell unterstellt den am PPRL teilnehmenden Parteien, dass sie sich böswillig (malicious) verhalten. Bei diesem Modell wird angenommen, dass sich die Parteien willkürlich verhalten [LP07]. Damit ist gemeint, dass Parteien möglicherweise nicht dem Protokoll folgen, das Protokoll zu einem beliebigen Zeitpunkt abbrechen oder arbiträre Eingabedaten senden. Dieses Modell wurde bisher nur selten betrachtet, da eine Analyse durch das unvorhersehbare und verschiedenartige Verhalten der Parteien schwierig ist [VCV13, VSCR17].

Angriffsarten Neben den Angriffsmodellen können verschiedene Angriffsarten für das PPRL untersucht werden. Hierbei ist es allerdings von der gewählten Privacy-Technik abhängig, inwieweit ein Angriff anwendbar bzw. Erfolg versprechend ist. Nach [VCOV14] und [VSCR17] müssen beim PPRL unter anderem folgende Angriffsarten berücksichtigt werden:

1. **Wörterbuchangriff:** Bei dem Wörterbuchangriff (Dictionary-Attack) wird angenommen, dass ein Angreifer das eingesetzte Maskierungsverfahren sowie die entsprechenden Parameter kennt, sodass er eine globale Datenmenge mit häufig vorkommenden Werten entsprechend codieren kann. Nun wird versucht, übereinstimmende Werte oder Datensätze mit Hilfe der globalen Datenmenge zu finden. Ist dies erfolgreich, so ermöglicht das den Rückschluss auf die uncodierten Werte. Zur Verhinderung dieser Angriffe können Keyed-Hash-Message-Authentication-Codes (HMACs) [KCB97] bei der Maskierung eingesetzt werden.

2. **Häufigkeitsanalyse:** Bei der Häufigkeitsanalyse (Frequency-Attack) wird die Verteilung von maskierten Werten untersucht. Unter Nutzung der Verteilung von bekannten unmaskierten Werten (z. B. Namenshäufigkeiten) kann so versucht werden, Übereinstimmungen in den Werten zu finden. Die Verwendung von HMACs ändert nicht die Häufigkeitsverteilung, weshalb diese hier keinen Schutz bieten. Um Häufigkeitsanalysen zu erschweren, kann stattdessen versucht werden, die Verteilung von Werten künstlich zu beeinflussen. Dies kann erfolgen, indem zusätzliche Werte oder Datensätze in die Datenbanken eingefügt werden. Diese können als Art Attrappe gesehen werden und sind so real nicht in den Datensätzen bzw. Datenbanken enthalten. Das Problem bei diesem Ansatz ist jedoch, dass sich die zusätzlichen Werte bzw. Datensätze sowohl auf die Linkage-Qualität als auch auf die Skalierbarkeit negativ auswirken, da einerseits die Wahrscheinlichkeit für falsch klassifizierte Records zunimmt und andererseits generell mehr Records berücksichtigt werden müssen [KVC12].

3. **Kompositionsangriff:** Dieser Angriff nutzt zusätzliches Wissen über die Datensätze bzw. die Datenbanken der einzelnen Parteien. Dieses Wissen über die Zusammen-

setzung einzelner Datenbestände kann kombiniert werden, um so sensitive Informationen über einzelne Datensätze abzuleiten [GKS08].

4. **Kollusion:** Kollusion bezeichnet den Zusammenschluss von zwei oder mehr Parteien, die gemeinsam versuchen, sensitive Informationen von anderen Parteien zu erhalten.

2.2 Bloom-Filter

Im folgenden Abschnitt werden Bloom-Filter vorgestellt und es wird beschrieben, wie diese für das PPRL als Privacy-Technik genutzt werden können. Bloom-Filter wurden erstmals in [Blo70] unabhängig vom PPRL vorgestellt. Ursprünglich wurden sie entwickelt, um beim Test auf Mengenzugehörigkeit zum Einsatz zu kommen [Blo70]. Vor allem wegen ihrer Speichereffizienz finden sie jedoch auch bei Netzwerkprotokollen und im Datenbankbereich Anwendung [BM04]. Erst später wurden sie in [SBR09] für das PPRL vorgeschlagen. Seitdem existieren mehrere Varianten für den Einsatz von Bloom-Filtern für das PPRL [SBR11, DKX+14, Dur12, VCOV14].

Im Rahmen dieser Arbeit sollen ausschließlich Bloom-Filter zur Maskierung der Datensätze zur Anwendung kommen. Dies liegt darin begründet, dass Bloom-Filter sehr häufig für das PPRL genutzt werden [VCV13] und außerdem bei korrekter Nutzung als sicher [SB16, NSKS14] und effizient [Vat14, VC16a, VC14, LYC+06] gelten. Außerdem kommen Bloom-Filter bereits in verschiedenen realen Anwendungen zum Einsatz [SRB14, RFB+14, KRM+12, Roc13].

Die nachfolgenden Ausführungen zu den Grundlagen von Bloom-Filtern beruhen auf [Blo70], [MU05] sowie [BM04]. Ferner beziehen sich die Erklärungen zum Einsatz von Bloom-Filtern im PPRL im Wesentlichen auf [SBR09], [SBR11], [NSKS14] und [DKX+14].

2.2.1 Definition

Ein Bloom-Filter (Bf) ist ein Bit-Array der Länge m. Die einzelnen Bits werden über ihre Position von $0, \ldots, m-1$ adressiert. Initial werden alle Bits auf 0 gesetzt. Sei $E = \{e_1, \ldots, e_\omega\}$ eine Menge beliebiger aber bestimmter Elemente, die in den Bloom-Filter aufgenommen (bzw. repräsentiert) werden sollen. Es werden k unabhängige Hashfunktionen $H_1, \ldots, H_k$ gewählt, welche als Eingabe Elemente aus der Menge E erlauben und als Ausgabe einen Wert innerhalb des Wertebereichs von $[0, m-1]$ erzeugen. Dieser Ausgabewert kann damit als Position innerhalb des Bloom-Filters interpretiert werden. Für die Anzahl an Hashfunktionen k muss generell $k \cdot \omega < m$ gelten.

Um ein Element $e \in E$ in den Bloom-Filter aufzunehmen (zu repräsentieren), wird dazu jede der k Hashfunktionen für das Element e berechnet. Jede Hashfunktion H_i mit $1 \leq i \leq k$ liefert dann eine Position j_i mit $H_i(e) = j_i$ innerhalb des Bloom-Filters. Demnach werden für ein Element e die Positionen $j_1, \ldots, j_k$ erzeugt. Die Bits an diesen insgesamt k Positionen, also Ausgabewerten der Hashfunktionen, werden im Anschluss auf 1 gesetzt. Damit gilt $\forall j \in \{j_1, \ldots, j_k\} : Bf(j) = 1$, wobei $Bf(j)$ den Wert des Bits an Position j liefert. Hierbei ist es möglich, dass von zwei verschiedenen Hashfunktionen H_x und H_y der gleiche Ausgabewert erzeugt wird, also $H_x(e) = H_y(e) = j_x = j_y$ gilt. Bei einer solchen *Kollision* bleibt das Bit an der entsprechenden Position auf 1 gesetzt. Um alle Elemente der Menge E in den Bloom-Filter aufzunehmen, wird dieser Vorgang für jedes Element durchgeführt. Auch hierbei können Kollisionen auftreten, indem von einer oder mehreren Hashfunktionen dieselbe Position für zwei unterschiedliche Elemente e_1 und e_2 mit $e_1 \neq e_2$ erzeugt wird: Zum einen können zwei verschiedene Hashfunktionen H_x und H_y mit $H_x(e_1) = H_y(e_2)$ dieselbe Position für die beiden Elemente erzeugen. Zum anderen ist es möglich, dass eine Hashfunktion H_i mit $H_i(e_1) = H_i(e_2)$ dieselbe Position für die Elemente e_1 und e_2 berechnet. Die Ursache hierfür ist, dass Hashfunktionen im Allgemeinen nicht injektiv sind. Um die vorhandenen Bits im Bit-Array vollständig auszunutzen, sollten die gewählten Hashfunktionen jedoch surjektiv sein.

Zum Test, ob ein Element in den Bloom-Filter aufgenommen wurde, werden ebenfalls die Hashfunktionen $H_1, \ldots, H_k$ für das Element berechnet und damit die Positionen $j_1, \ldots, j_k$ bestimmt. Falls ein Bit an bereits einer einzigen Position auf 0 gesetzt ist, also gilt $\exists j \in \{j_1, \ldots, j_k\} : Bf(j) \neq 1$, so ist das Element definitiv nicht im Bloom-Filter enthalten. Sind hingegen die Bits an jeder der Positionen auf 1 gesetzt, also gilt $\forall j \in \{j_1, \ldots, j_k\} : Bf(j) = 1$, so ist das Element *wahrscheinlich* im Bloom-Filter enthalten. Eine genaue Bestimmung, ob $e \in Bf$ gilt, ist aufgrund von möglichen Kollisionen nicht möglich. So könnte die notwendige Menge an gesetzten Positionen auch durch (mehrere) andere Elemente (gemeinschaftlich) gesetzt worden sein. Aus diesem Grund können Bloom-Filter falsch-positive Ergebnisse liefern, indem der beschriebene Test die Aussage $e \in Bf$ liefert, obwohl in Wirklichkeit $e \notin Bf$ gilt.

Weiterhin sind AND- ($\circ$), OR- ($\diamond$) und XOR-Operationen ($\oplus$) auf Bloom-Filtern definiert. Voraussetzung hierfür ist, dass die Operanden Bloom-Filter die gleiche Länge m besitzen und die gleichen Hashfunktionen $H_1, \ldots, H_k$ nutzen. Seien nun Bf_1 und Bf_2 zwei Bloom-Filter, welche diese Vorraussetzungen erfüllen und die Elementmengen E_1 bzw. E_2 repräsentieren. Weiter sei Bf_{res} der Bloom-Filter, welcher das Ergebnis der jeweiligen Operation darstellt. Dann gilt, dass

$$Bf_1 \circ Bf_2 = Bf_{res}, \text{ sodass} \\ [\forall i \in \{0, \ldots, m-1\} : Bf_{res}(i) = (Bf_1(i) \wedge Bf_2(i))] \tag{2.15}$$

$$Bf_1 \diamond Bf_2 = Bf_{res}, \text{ sodass} \\ [\forall i \in \{0, \ldots, m-1\} : Bf_{res}(i) = (Bf_1(i) \vee Bf_2(i))] \tag{2.16}$$

$$Bf_1 \oplus Bf_2 = Bf_{res}, \text{ sodass} \\ [\forall i \in \{0, \ldots, m-1\} : Bf_{res}(i) = ((Bf_1(i) \vee Bf_2(i)) \\ \wedge \neg(Bf_1(i) \wedge Bf_2(i)))]. \tag{2.17}$$

Die Ausführung einer solchen Operation führt dazu, dass der resultierende Bloom-Filter Bf_{res} nun ebenfalls bestimmte Teile, der von den Bloom-Filtern Bf_1 und Bf_2 repräsentierten Mengen E_1 bzw. E_2, repräsentiert. Es gilt:

$$Bf_1 \circ Bf_2 \approx E_1 \cap E_2 \tag{2.18}$$

$$Bf_1 \diamond Bf_2 = E_1 \cup E_2 \tag{2.19}$$

$$Bf_1 \oplus Bf_2 \approx (E_1 \cup E_2) \backslash (E_1 \cap E_2). \tag{2.20}$$

Dabei wird durch $\approx$ gekennzeichnet, dass die Operation auf den beiden Bloom-Filtern die nachfolgende Mengenoperation abschätzt. So kann der Schnitt der beiden Ursprungsmengen E_1 und E_2 nur abgeschätzt werden. Der Grund hierfür sind mögliche Kollisionen. Wenn ein Bit an einer Stelle i im resultierenden Bloom-Filter auf 1 gesetzt ist, dann gilt $Bf_{res}(i) = 1$. Prinzipiell gilt $Bf_{res}(i) = 1 \Leftrightarrow (Bf_1(i) = 1 \wedge Bf_2(i) = 1)$. Jedoch müssen mehrere Bits an den Stellen $i_1, \cdots, i_\iota$ in den Bloom-Filtern Bf_1 und Bf_2 nicht durch dasselbe Element (aus der Schnittmenge) gesetzt worden sein. Vielmehr können die entsprechenden Bits durch verschiedene Elemente, welche Teil der Schnittmenge oder kein Teil der Schnittmenge sind, gesetzt werden.

Schließlich ist es möglich, die Ähnlichkeit von Bloom-Filtern zu bestimmen. Die Grundlage hierfür bildet die Bestimmung der Ähnlichkeit zweier beliebiger Mengen, wozu verschiedene Ähnlichkeitsmaße zur Anwendung kommen. Die zwei wichtigsten Ähnlichkeitsmaße im Rahmen dieser Arbeit sind der Dice- [Dic45] und der Jaccard-Koeffizient [Jac12]. Die Ähnlichkeit zweier beliebiger Mengen X und Y kann mit diesen wie folgt berechnet werden:

$$Sim_{Dice}(X,Y) = \frac{2 \cdot |X \cap Y|}{|X| + |Y|} \tag{2.21}$$

$$Sim_{Jaccard}(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} = \frac{|X \cap Y|}{|X| + |Y| - |X \cap Y|}. \tag{2.22}$$

Wie oben beschrieben, kann ein Bloom-Filter eine Menge E repräsentieren, indem alle Elemente $e \in E$ in den Bloom-Filter aufgenommen werden. Aus diesem Grund lässt sich die Ähnlichkeitsbestimmung zwischen Mengen auf Bloom-Filter übertragen. Die Voraussetzung hierfür ist ebenfalls, dass die Bloom-Filter die gleiche Länge m besitzen und die gleichen Hashfunktionen $H_1, \ldots, H_k$ nutzen. Für die zwei Bloom-Filter Bf_1 und Bf_2, welche die Elementmengen E_1 bzw. E_2 repräsentieren, gilt:

$$Sim_{Dice}(Bf_1, Bf_2) = \frac{2 \cdot \|Bf_1 \circ Bf_2\|_1}{\|Bf_1\|_1 + \|Bf_2\|_1} \tag{2.23}$$

$$Sim_{Jaccard}(Bf_1, Bf_2) = \frac{\|Bf_1 \circ Bf_2\|_1}{\|Bf_1 \diamond Bf_2\|_1} = \frac{\|Bf_1 \circ Bf_2\|_1}{\|Bf_1\|_1 + \|Bf_2\|_1 - \|Bf_1 \circ Bf_2\|_1}. \tag{2.24}$$

Hierbei entspricht $\|\cdot\|_1$ bzw. $\|\cdot\|_0$ der Anzahl der 1- bzw. 0-Bits im Bit-Array des entsprechenden Bloom-Filters. Alternativ kann die Dice-Similarity auch mit Hilfe der XOR-Operation ($\oplus$) durch $Sim_{Dice}(Bf_1, Bf_2) = \frac{\|Bf_1 \oplus Bf_2\|_0}{m}$ berechnet werden.

2.2.2 Wahl der Parameter und Falsch-positiv-Rate

Insgesamt müssen bei Bloom-Filtern vier Parameter berücksichtigt werden. Diese sind

- m: Länge des Bit-Arrays
- k: Anzahl der Hashfunktionen
- ω: Anzahl der aufzunehmenden Elemente.

Jeder dieser Parameter hat Einfluss auf die Wahrscheinlichkeit von falsch-positiven Ergebnissen. Diese Wahrscheinlichkeit des Auftretens von falsch-positiven Ergebnissen kann mit der Falsch-positiv-Rate (fpr) nach [MU05] wie folgt abgeschätzt werden:

$$fpr = (1 - \mu)^k. \tag{2.25}$$

Hierbei entspricht μ dem Anteil der 0-Bits im Bloom-Filter, nachdem alle Elemente aus E in den Bloom-Filter aufgenommen wurden. Hierbei gilt [MU05]:

$$\mu = \left(1 - \frac{1}{m}\right)^{k \cdot \omega}. \tag{2.26}$$

In [BM04, MU05] wurde gezeigt, dass die Falsch-positiv-Rate *fpr* minimiert wird, wenn $\mu = \frac{1}{2}$ gilt. Damit sollte möglichst genau die Hälfte aller Bits im Bloom-Filter auf 0 bzw. 1 gesetzt sein. Die Falsch-positiv-Rate berechnet sich dann wie folgt:

$$fpr = \left(1 - \frac{1}{2}\right)^{k} = \left(\frac{1}{2}\right)^{k} = 2^{-k}. \tag{2.27}$$

Hieraus kann abgeleitet werden, wie die Anzahl der Hashfunktionen k optimalerweise bestimmt werden kann. Es gilt:

$$k_{opt} = \ln(2) \cdot \frac{m}{\omega} \tag{2.28}$$

bzw.

$$k_{opt} = \frac{\ln\left(\frac{1}{fpr}\right)}{\ln(2)}. \tag{2.29}$$

Aus Gleichung 2.28 kann außerdem die benötigte Länge des Bloom-Filters abgeleitet werden, folglich gilt:

$$m_{opt} = \frac{k \cdot \omega}{\ln(2)}. \tag{2.30}$$

Zusammenfassend bewirkt eine höhere Anzahl von Hashfunktionen k, dass die Falsch-positiv-Rate reduziert wird. Der Grund hierfür ist, dass mehr Hashfunktionen die Wahrscheinlichkeit erhöhen, ein 0-Bit für ein Element e zu finden, für das gilt $e \notin Bf$ bzw. $e \notin E$. Dies gilt jedoch nur dann, wenn die Länge des Bloom-Filters entsprechend an die Anzahl der Hashfunktionen angepasst wird. Eine hohe Anzahl an Hashfunktionen würde ansonsten (wenn die Länge des Bloom-Filters zu klein gewählt wurde) zu mehr 1-Bits führen. Demzufolge würde der Anteil der 0-Bits im Bloom-Filter reduziert, wodurch wiederum die Wahrscheinlichkeit erhöht wird, für ein Element, welches nicht in der Menge E enthalten ist, ein 1-Bit zu finden. Andererseits verursachen eine größere Anzahl Hashfunktionen und ein längeres Bit-Array auch einen erhöhten Aufwand für die Bloom-Filter-Operationen, wodurch die Effizienz vermindert werden kann. Im Allgemeinen ist die Wahl der Parameter stark abhängig von der Art der Anwendung und dem geplanten Einsatzzweck der Bloom-Filter.

Das Vorgehen zur Wahl der Bloom-Filter-Parameter ist im Rahmen dieser Arbeit wie folgt: Zunächst wird die Anzahl der Hashfunktionen k mit Hilfe von Gleichung 2.29 bestimmt. Hierzu wird ein akzeptierbarer Wert für die *fpr* gewählt. Nachfolgend wird die Anzahl der durchschnittlich aufzunehmenden Elemente ω abgeschätzt (zur genauen Vorgehensweise siehe Abschnitt 4.1). Mit Hilfe dieser Werte kann dann die erforderliche Länge m des Bloom-Filters durch Gleichung 2.30 bestimmt werden.

2.2.3 Bloom-Filter im PPRL

Bloom-Filter wurden erstmals in [SBR09] für das PPRL vorgeschlagen. Die grundlegende Idee besteht darin, für jeden Datensatz (Record) einen Bloom-Filter zu erzeugen. Dabei wird die Menge der Attributwerte der QIDs, also der Attribute, welche für das PPRL benutzt werden sollen, als die vom Bloom-Filter zu repräsentierende Menge E gewählt. Für einen Record $r_i^j \in D_j$ ergibt sich somit die Menge $E_i^j = \{e_1^{i,j}, \ldots, e_\eta^{i,j}\}$ der Attributwerte der QIDs, wobei η gleich der Anzahl der QID-Attribute entspricht.

Um das Problem der Dirty Data zu berücksichtigen, werden die Elemente der Menge E_i^j jedoch nicht direkt in den Bloom-Filter aufgenommen. Stattdessen wird die Menge aller *Q-Gramme* [CC04] $\bar{E}_i^j$ aus der Menge E_i^j erzeugt und alle Elemente der Menge $\bar{E}_i^j$, d. h. alle Q-Gramme, werden in den Bloom-Filter aufgenommen. Dazu wird für jeden Attributwert $e_1^{i,j}, \ldots, e_\eta^{i,j} \in E_i^j$ die Menge seiner Q-Gramme $QG(e_1^{i,j}), \ldots, QG(e_\eta^{i,j})$ erzeugt. Die Menge aller Q-Gramme ergibt sich dann als Vereinigung der einzelnen Q-Gramm-Mengen, demzufolge als $\bar{E}_i^j = QG(e_1^{i,j}) \cup \ldots \cup QG(e_\eta^{i,j})$.

Hierbei ist ein Q-Gram ein Teilstring der Länge Q einer beliebigen Zeichenkette [CC04]. Beispielsweise ist die Menge der 3-Gramme (Trigramme), die aus dem Namen 'Bernd' gebildet werden können, {'Ber', 'ern', 'rnd'}. Q-Gramme werden häufig im Record-Linkage eingesetzt [BCC+03, Chr12b], da mit ihrer Hilfe die Ähnlichkeit von zwei Zeichenketten bestimmt werden kann [CC04, Kuk92, vBDS88]. Soll zum Beispiel die Ähnlichkeit der beiden Zeichenketten 'Bern**d**' und 'Bern**t**' bestimmt werden, so kann man für $Q = 3$ die Mengen der Trigramme, also {'Ber', 'ern', 'rnd'} und {'Ber', 'ern', 'rnt'}, betrachten. Vergleicht man die beiden Mengen (mit Hilfe von Gleichung 2.21 bzw. Gleichung 2.22), so wird deutlich, dass von den insgesamt sechs Trigrammen genau zwei übereinstimmen. Dies führt zu einer Dice-Similarity von $\frac{2 \cdot 2}{3+3} = \frac{4}{6} = \frac{2}{3}$ und einer Jaccard-Similarity von $\frac{2}{4} = \frac{1}{2}$.

Nachdem dann die Menge $\bar{E}_i^j$ aus E_i^j erzeugt wurde, können die einzelnen Elemente aus $\bar{E}_i^j$ in einen Bloom-Filter Bf_i^j aufgenommen werden. Dabei wird für jedes Element entsprechend den Ausführungen in Unterabschnitt 2.2.1 vorgegangen. Der resultierende Bloom-Filter Bf_i^j repräsentiert dann den Record $r_i^j \in D_j$. Dieses Vorgehen wird von den einzelnen

Parteien für alle Records ihrer Datenbank wiederholt.

Die Erzeugung der Menge aller Q-Gramme führt dazu, dass die Länge Q der Q-Gramme als zusätzlicher Parameter für die Bildung von Bloom-Filtern berücksichtigt werden muss. Die am PPRL teilnehmenden Parteien müssen sich natürlich im Vorfeld auf alle genannten Parameter, welche zur Erzeugung von Bloom-Filtern notwendig sind, verständigen. Wichtig für das PPRL ist, dass kryptologische Einweghashfunktionen [KCB97, Bru96] genutzt werden, sodass die aufgenommenen Elemente, also die Q-Gramme, nicht rekonstruiert werden können [SBR09].

In Abbildung 2.2 ist das Vorgehen zur Erstellung eines Bloom-Filters zusammenfassend dargestellt. Hierbei wurde vereinfacht angenommen, dass nur ein Attribut (Vorname) für das Linkage berücksichtigt werden soll. Als Datensatz wurde der Beispiel-Datensatz A2 aus dem laufenden Beispiel von Unterabschnitt 2.1.1 gewählt. Die Parameter des Bloom-Filters wurden beispielhaft mit $m = 16$, $k = 3$ und $Q = 3$ gewählt. Initial wird zunächst der leere Bloom-Filter erzeugt, indem alle Bits auf 0 gesetzt werden. Im Anschluss wird aus der Menge der gewählten Attributwerte die Menge der Q-Gramme erzeugt. In diesem Fall betrifft das nur den Attributwert 'Bernd', welcher den Vornamen der Person aus Datensatz A2 darstellt. Wie bereits gezeigt, ist die Menge aller Q-Gramme hierfür {'Ber', 'ern', 'rnd'}. Nachfolgend werden für jedes resultierende Q-Gramm die Hashfunktionen $H_1, \ldots, H_3$ berechnet und die Bits an den ausgegebenen Positionen auf 1 gesetzt. Wie zu erkennen ist, tritt hierbei eine Kollision zwischen $H_3(\text{Ber})$ und $H_2(\text{rnd})$ auf, da gilt $H_3(\text{Ber}) = H_2(\text{rnd}) = 9$.

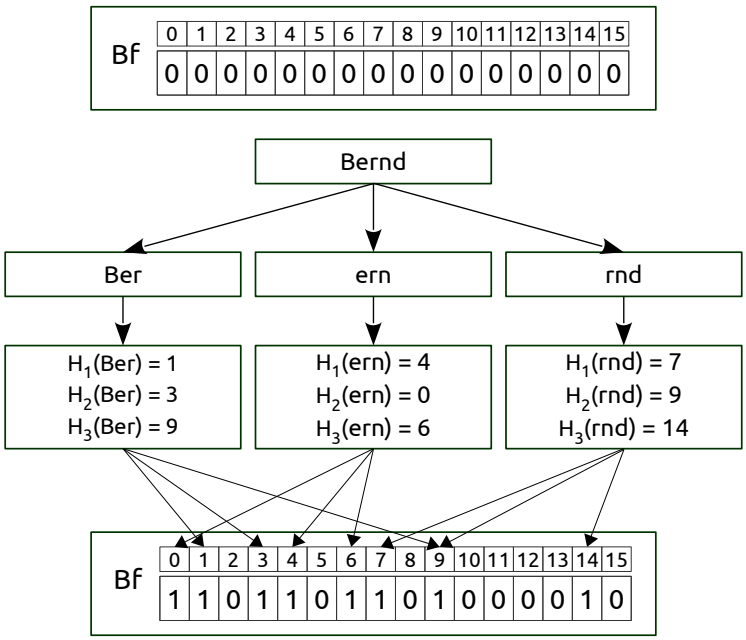


Abbildung 2.2: Beispiel: Erstellung eines Bloom-Filters beim PPRL.

Im Rahmen des PPRL wird, wie in Unterunterabschnitt 2.1.2.1 beschrieben, eine Vorverarbeitung der Daten durchgeführt. Im Zusammenhang mit Q-Grammen ist es möglich, die Attributwerte in zusätzliche Füllzeichen (z. B. '#') einzubetten [vBDS88], beispielsweise indem 'Bernd' zu '##Bernd##' transformiert wird. Im Fall von Trigrammen führt dies zur Menge {'##B', '#Be', 'Ber', 'ern', 'rnd', 'nd#', 'd##'}. Das hat zur Folge, dass der Anfang und das Ende einer Zeichenkette stärker gewichtet werden.

Wie in Unterabschnitt 2.2.1 beschrieben, ist es möglich, die Ähnlichkeit zweier Bloom-Filter zu bestimmen. Da ein Bloom-Filter einen Record $r_i^j \in D_j$ repräsentiert, ermöglicht der Vergleich der Bloom-Filter einen indirekten Vergleich der Records, welche die Bloom-Filter repräsentieren. Im obigen Beispiel wurde für den Datensatz A2 der Bloom-Filter $Bf_{A2}^A = [1101101101000010]$ bestimmt. Das gleiche Vorgehen könnte zu einem Bloom-Filter $Bf_{B1}^B = [1101101001011101]$ für den Datensatz B1 mit dem zu repräsentierenden Attributwert 'Bernt' führen. Auch hier werden die Trigramme 'Ber' und 'ern' in den Bloom-Filter aufgenommen, weshalb für diese die gleichen Positionen gesetzt werden wie für den Bloom-Filter Bf_{A2}^A. Das Trigramm 'rn**t**' hingegen erzeugt andere Positionen als 'rn**d**', was zu vier veränderten Bits im Vergleich zum Bloom-Filter Bf_{A2}^A führt. Die Ähnlichkeit der beiden Bloom-Filter Bf_{A2}^A und Bf_{B1}^B berechnet sich nach Gleichung 2.24 für die Jaccard-Similarity wie folgt:

$$Sim_{Jaccard}(Bf_{A2}^A, Bf_{B1}^B) = \frac{\|Bf_{A2}^A \circ Bf_{B1}^B\|_1}{\|Bf_{A2}^A \diamond Bf_{B1}^B\|_1} = \frac{\|1101101001000000\|_1}{\|1101101101011111\|_1} = \frac{6}{12} = \frac{1}{2}.$$

Die resultierende Ähnlichkeit stimmt dabei mit der oben berechneten Ähnlichkeit der Trigramm-Mengen der beiden Zeichenketten 'Bernd' und 'Bernt' überein. Im Allgemeinen lassen sich mit Bloom-Filtern ähnlich gute Ergebnisse erzielen, wie beim Vergleich der Originaldaten möglich sind [SBR09].

Das hier vorgestellte Vorgehen beschränkt sich auf die Aufnahme von Q-Grammen in die Bloom-Filter. Q-Gramme können dabei zwar für beliebige Zeichenketten genutzt werden, es ist jedoch sinnvoll für numerische Werte, zum Beispiel bei Altersangaben oder Geburtsdaten, andere Techniken einzusetzen. Für weitere Details hierzu sei auf [VC16b] verwiesen.

2.2.3.1 Arten von Bloom-Filtern

Beim zuvor beschriebenen Vorgehen zur Bildung eines Bloom-Filters werden die Attributwerte aller QIDs eines Records in einen einzigen Bloom-Filter aufgenommen. Der dabei entstehende Bloom-Filter wird auch als *Cryptographic-Longterm-Key (CLK)* bezeichnet. Diese Art der Konstruktion wurde erstmals in [SBR11] eingeführt.

Eine andere Möglichkeit zur Konstruktion von Bloom-Filtern wurde in [Dur12] vorgestellt. Hierbei werden die Q-Gramm-Mengen für die einzelnen Attributwerte zunächst in separate Bloom-Filter aufgenommen. Die verschiedenen entstehenden Field-Level-Bloom-Filter (FBF) werden dann zu einem *Record-Level-Bloom-Filter (RBF)* zusammengesetzt. Dies geschieht, indem mehrere Bits aus den FBFs ausgewählt werden, wobei die einzelnen Attribute unterschiedlich gewichtet werden können, sodass mehr Bits aus einem FBF ausgewählt werden.

Schließlich wurde in [VCOV14] eine Variante vorgestellt, welche die beiden obigen Ansätze kombiniert. Hierbei wird, wie beim CLK, nur ein Bloom-Filter genutzt und erzeugt. Jedoch werden unterschiedlich viele Hashfunktionen für die verschiedenen Attribute gewählt. Dies ermöglicht ebenfalls eine Gewichtung der einzelnen Attribute. Diese Variante wird als *CLKRBF* bezeichnet [VCOV14].

2.2.3.2 Privacy

Es ist offensichtlich, dass Bloom-Filter die repräsentierten Attributwerte der QIDs maskieren: Ohne Kenntnis der genutzten Hashfunktionen ist es schwierig zu bestimmen, welche Q-Gramme in dem Bloom-Filter enthalten sind. Selbst das Wissen, dass bestimmte Q-Gramme im Bloom-Filter enthalten sind, liefert noch nicht die repräsentierten Attributwerte. Außerdem können die Q-Gramme von verschiedenen Attributen stammen, was eine Zuordnung zusätzlich erschwert. Allerdings sind Bloom-Filter anfällig für Häufigkeitsanalysen [SBR09, SBR11], da sich die Häufigkeit von Q-Grammen auch in den resultierenden Bloom-Filtern durchsetzt. Das macht sich deutlich, indem Bloom-Filter häufig gesetzte Bitpositionen aufweisen, die in vielen oder eventuell sogar allen Bloom-Filtern vorhanden sind. Angriffe auf Bloom-Filter werden beispielsweise in [KKDM11] und [KKD+13] beschrieben. Generell gibt es einen Trade-off zwischen der Sicherheit (Privacy) und der Qualität von Bloom-Filtern [SBR11]. Dies betrifft das Verhältnis zwischen der Anzahl der Hashfunktionen k und der Länge der Bloom-Filter m. Ist $\frac{k}{m}$ hoch, so sind Bloom-Filter sicherer, da mehr Kollisionen entstehen, was die Zuordnung von einzelnen Bits zu Q-Grammen erschwert. Ist hingegen $\frac{k}{m}$ niedrig, so sind Kollisionen unwahrscheinlicher, was die Qualität der Bloom-Filter erhöht. Für weitere Details zur Sicherheit von Bloom-Filtern sei auf [NSKS14], [KS14] und [SB16] verwiesen.

2.3 Blocking-Verfahren

Das Ziel dieser Arbeit ist die Verbesserung der Skalierbarkeit des PPRL durch Erweiterung auf das P3RL, also durch die Parallelisierung von PPRL-Verfahren. Wie in Un-

terunterabschnitt 2.1.2.2 beschrieben, hat das Blocking sehr großen Einfluss auf die Leistungsfähigkeit und Skalierbarkeit des PPRL. Aus diesem Grund ist die Wahl der Blocking-Verfahren von wesentlicher Bedeutung. Zudem kann das Blocking genutzt werden, um die Datensätze auf die verfügbaren Rechner im Cluster zu partitionieren, sodass der Vergleich und die Ähnlichkeitsbestimmung von Datensätzen innerhalb der einzelnen Blöcke parallel erfolgen kann.

Der Fokus der Arbeit liegt auf der Anpassung und Umsetzung des *Locality-sensitive-Hashings (LSH)* [GIM+99]. LSH bietet den Vorteil, dass es direkt auf Bloom-Filtern bzw. Bit-Arrays angewandt werden kann. Außerdem erzielte es gute Ergebnisse hinsichtlich Performanz und Linkage-Qualität [Dur12, KV13, KV14, KV15] und benötigt kein globales Wissen, womit es sich gut zur Parallelisierung eignet.

Zum Vergleich soll außerdem *phonetisches Blocking* umgesetzt werden. Dieses wird sehr häufig, vor allem beim RL, aber auch beim PPRL, eingesetzt [Chr12a, VCV13, KV09, KVC12]. Dadurch sind bereits mehrere etablierte und oft genutzte phonetische Codierungen verfügbar [Chr06, Chr12b, VCV13].

Die nachfolgenden Ausführungen bezüglich LSH basieren auf [IM98], [GIM+99], [Ind04], [AI08], [HPIM12], [LRU14] sowie auf [KL10], [Dur12], [KV13] und [KV14]. Die Erklärungen bezüglich des phonetischen Blockings gründen auf [Chr06], [Chr12a], [Chr12b] sowie [VCV13].

2.3.1 Locality-sensitive-Hashing (LSH)

LSH wurde in [IM98] für das Nearest-Neighbor-Search-Problem in hochdimensionalen Räumen vorgestellt. In [Dur12] wurde es dann erstmals für das Blocking unter Verwendung von Bloom-Filtern im Rahmen des PPRL eingesetzt.

2.3.1.1 Definition

Sei $\mathcal{F}$ eine Menge bzw. Familie von Funktionen. Weiter sei $d(\cdot,\cdot)$ ein Distanzmaß [LRU14] und Z eine Menge mit Elementen. Für zwei Abstände d_1, d_2 mit $d_1 < d_2$ und zwei Wahrscheinlichkeiten pr_1, pr_2 mit $pr_1 > pr_2$ heißt $\mathcal{F}$ (d_1, d_2, pr_1, pr_2)-*sensitiv*, wenn für alle Funktionen $f \in \mathcal{F}$ und für alle Elemente $x, y \in Z$ gilt:

- $d(x,y) \leq d_1 \quad \Rightarrow \quad \mathbb{P}(f(x) = f(y)) \geq pr_1$
- $d(x,y) \geq d_2 \quad \Rightarrow \quad \mathbb{P}(f(x) = f(y)) \leq pr_2$.

Durch die Eigenschaft der (d_1, d_2, pr_1, pr_2)-Sensitivität wird gewährleistet, dass die Wahrscheinlichkeit, dass eine Funktion $f \in \mathcal{F}$ für zwei Elemente, deren Distanz kleiner oder gleich d_1 ist, den gleichen Wert liefert, mindestens pr_1 ist. Andererseits wird sichergestellt, dass, wenn die Distanz zweier Elemente größer oder gleich d_2 ist, die Wahrscheinlichkeit, dass die Funktion f für die beiden Elemente den gleichen Wert liefert, nicht größer als pr_2 ist.

LSH ist nun eine probabilistische Methode, welche eine Menge von Hashfunktionen $\mathcal{F}$ benutzt, für die genau diese Eigenschaft der (d_1, d_2, pr_1, pr_2)-Sensitivität erfüllt ist. Die Familie der Hashfunktionen $\mathcal{F}$ wird dabei so gewählt, dass sie sensitiv zu einem bestimmten Distanzmaß d ist. Im Zusammenhang mit dem PPRL bzw. Bloom-Filtern sind dabei vor allem die Jaccard- und die Hamming-Distanz von Bedeutung. Für zwei beliebige Mengen X und Y sind diese wie folgt definiert:

$$d_{Jaccard}(X, Y) = 1 - Sim_{Jaccard}(X, Y) \tag{2.31}$$

$$d_{Hamming}(X, Y) = (X \cup Y) \backslash (X \cap Y) = X \oplus Y. \tag{2.32}$$

Hierbei entspricht $Sim_{Jaccard}(\cdot, \cdot)$ der Jaccard-Similarity aus Gleichung 2.24 und $\oplus$ kennzeichnet die XOR-Verknüpfung zweier Mengen. Das LSH, bei dem die Familie von Hashfunktionen $\mathcal{F}$ sensitiv zu der Jaccard-Distanz gewählt wird, wird auch als *Jaccard-LSH (JLSH)* bezeichnet. Ist $\mathcal{F}$ hingegen sensitiv zu der Hamming-Distanz, so wird es als *Hamming-LSH (HLSH)* bezeichnet.

Schließlich müssen noch die genauen Funktionen $f \in \mathcal{F}$ bestimmt werden, die sensitiv zum entsprechenden Distanzmaß (Jaccard- bzw. Hamming-Distanz) sind bzw. dieses approximieren. Hierzu sei im Folgenden Z eine Menge von Bloom-Filtern bzw. Bit-Arrays der Länge m. Zur Annäherung der Jaccard-Distanz wird die Klasse der *MinHash*-Funktionen genutzt. Diese ergibt sich als

$$\mathcal{F}_{Jaccard} = \{f_i : f_i(x) = \min_{\text{1-Bit}}\{\pi_i(x)\},\ x \in Z\}. \tag{2.33}$$

Eine MinHash-Funktion f_i erhält als Eingabe einen Bloom-Filter bzw. ein Bit-Array und permutiert zunächst die Bits durch eine Permutation π_i. Als Ausgabe liefert die Funktion f_i dann die Position des ersten gesetzten Bits (1-Bit) entsprechend des permutierten Bloom-Filters. Zur Illustration sei $Bf_1 = [11000011]$ ein einfacher Bloom-Filter, $\pi_1 = (2, 3, 0, 1, 6, 7, 4, 5)$ eine Permutation und $f_1(\cdot) = \min_{\text{1-Bit}}\{\pi_1(\cdot)\}$ die entsprechende MinHash-Funktion bezüglich der Permutation π_1. Für den Bloom-Filter Bf_1 ergibt sich

damit $f_1(Bf_1) = \min_{\text{1-Bit}}\{\pi_1(Bf_1)\} = \min_{\text{1-Bit}}\{\pi_1(11000011)\} = \min_{\text{1-Bit}}\{(00111100)\} = 2$ als Hashwert. Für den Beweis, dass die MinHash-Funktionen zur Approximation der Jaccard-Distanz genutzt werden können, sei auf [Bro97] bzw. [LRU14] verwiesen.

Die Menge der Funktionen $\mathcal{F}$, die zur Annäherung an die Hamming-Distanz eingesetzt werden, ist wie folgt definiert:

$$\mathcal{F}_{Hamming} = \{f_i : f_i(x) = x(i), x \in Z, \forall i \in \{0, \ldots, m-1\}\}. \tag{2.34}$$

Eine solche Funktion $f_i \in \mathcal{F}_{Hamming}$ erhält wieder als Eingabe einen Bloom-Filter bzw. ein Bit-Array. Als Ausgabe liefert die Funktion f_i hierbei den Wert des Bits an der Position i des entsprechenden Bloom-Filters. Derartige Funktionen lassen sich einfach dadurch erzeugen, indem der Wert des Bits an einer zufälligen Position des Bloom-Filters zurückgegeben wird. Erneut sei zur Illustration $Bf_1 = [11000011]$ ein Bloom-Filter. Die Funktion $f_2(\cdot) \in \mathcal{F}_{Hamming}$ liefert angewandt auf Bf_1 den Wert 0, da das Bit an Position 2 des Bloom-Filters Bf_1 nicht gesetzt ist. Der Beweis, dass derartige Funktionen die Hamming-Distanz approximieren, wird in [IM98] geführt.

2.3.1.2 Nutzung für das Blocking

LSH lässt sich für das Blocking im Rahmen des PPRL einsetzen, indem eine Familie von Hashfunktionen $\mathcal{F}$ ($\mathcal{F}_{Jaccard}$ bzw. $\mathcal{F}_{Hamming}$) genutzt wird, um einen Blocking-Key zur Partitionierung der Records zu erzeugen. Durch LSH wird damit gewährleistet, dass ähnliche Records mit hoher Wahrscheinlichkeit demselben Block zugeordnet werden, wohingegen dies für unähnliche Records nur mit sehr niedriger Wahrscheinlichkeit zutrifft. Die Erzeugung eines Blocking-Keys BK erfolgt dadurch, dass die Ergebnisse von Ψ Hashfunktionen $f_{\lambda_1}, \ldots, f_{\lambda_\Psi} \in \mathcal{F}$ zu einem Key zusammengefasst werden, sodass für einen Bloom-Filter Bf_i^j gilt $BK(Bf_i^j) = f_{\lambda_1}(Bf_i^j) \odot \ldots \odot f_{\lambda_\Psi}(Bf_i^j)$. Hierbei bezeichne $\odot$ die Konkatenation der entsprechenden Funktionswerte und für $\mathcal{F}_{Hamming}$ gilt, dass $\lambda_1, \ldots, \lambda_\Psi \in \{0, \ldots, m-1\}$. Da der entstehende Blocking-Key BK aus genau Ψ Funktionswerten gebildet wird, bezeichnet man mit Ψ auch die Länge des Blocking-Keys.

Bei LSH besteht grundsätzlich die Möglichkeit, dass zwei Elemente x und y, trotz hinreichender Ähnlichkeit ($d(x, y) \leq d_1$), nicht die gleichen Funktionswerte für eine Hashfunktion aus der Funktionsfamilie $\mathcal{F}$ erzeugen. Aus diesem Grund wird von der ursprünglichen Idee des Standard-Blockings abgewichen, wo nur ein Blocking-Key erzeugt wird, der jeweils disjunkte Blöcke liefert (vgl. Unterunterabschnitt 2.1.2.2). Stattdessen können bei LSH mehrere Blocking-Keys $BK_1, \ldots, BK_\Lambda$ erzeugt werden, insgesamt also Λ Blocking-Keys. Jeder dieser Λ Blocking-Keys wird dabei nach dem oben beschriebenen Vorgehen

erzeugt. Durch die mehrfache Erzeugung von Blocking-Keys wird mit Λ auch die Anzahl der Iterationen bezeichnet. Die Verwendung von mehr als einem Blocking-Key führt weiterhin dazu, dass die entstehenden Blöcke nicht mehr disjunkt sind. Außerdem ist es möglich, dass sich Datensätze in mehr als einem Block treffen und dadurch doppelte Vergleiche ausgeführt werden. Grundsätzlich haben die beiden Parameter Ψ und Λ wesentlichen Einfluss auf die Leistungsfähigkeit von LSH. Wird Ψ größer gewählt, so ist die Wahrscheinlichkeit höher, dass sich nur sehr ähnliche Datensätze in den einzelnen Blöcken befinden. Dadurch sind die Blöcke kleiner, weshalb weniger Vergleiche durchgeführt werden, womit die Performanz des PPRL erhöht wird. Durch kleinere Blöcke steigt jedoch auch die Wahrscheinlichkeit, eigentlich übereinstimmende Datensätze aufgrund von fehlerhaften Daten zu verpassen. Wird Λ größer gewählt, so wird die Wahrscheinlichkeit erhöht, dass zwei Datensätze demselben Block zugeordnet werden. Wie bereits erwähnt, können jedoch Kandidaten-Paare mehrfach erzeugt werden. Außerdem steigt auch die Wahrscheinlichkeit, dass zwei nicht übereinstimmende Datensätze aufgrund von partiellen Ähnlichkeiten demselben Block zugeordnet werden. Insgesamt verschlechtert sich durch eine größere Anzahl von Blocking-Keys Λ die Skalierbarkeit.

2.3.1.3 Wahl der Parameter

Wie zuvor beschrieben haben die beiden Parameter Ψ und Λ großen Einfluss auf die Leistungsfähigkeit von LSH. Um diese Parameter bestmöglich auszuwählen, muss dazu zunächst betrachtet werden, mit welcher Wahrscheinlichkeit zwei Elemente bzw. Bloom-Filter x und y mindestens einmal demselben Block zugeordnet werden. Es gilt:

$$\mathbb{P}(\exists i \in \{1, \ldots, \Lambda\} : BK_i(x) = BK_i(y)) \geq \left(1 - (1 - (Sim(x, y)^{\Psi})^{\Lambda}\right). \tag{2.35}$$

Hierbei ist $Sim(x, y)^{\Psi}$ die Wahrscheinlichkeit, dass die beiden Elemente x und y in einem Blocking-Key übereinstimmen. Die Wahrscheinlichkeit, dass eine Hashfunktion $f \in \mathcal{F}$, die zur Bildung des Blocking-Keys genutzt wird, für x und y denselben Funktionswert erzeugt, beträgt dabei genau $Sim(x, y)$ [LRU14]. Die Wahrscheinlichkeit $Sim(x, y)^{\Psi}$ hängt damit grundlegend davon ab, wie ähnlich sich die beiden Elemente sind. Hohe Werte für $Sim(x, y)$ erhöhen dabei diese Wahrscheinlichkeit, da dann sehr viele der Bits übereinstimmen. Andererseits reduzieren größere Werte für Ψ diese Wahrscheinlichkeit, denn dadurch werden mehr Funktionen berechnet und so die Wahrscheinlichkeit erhöht, dass ein unterschiedliches Bit zwischen x und y zu einem anderen Hashwert führt. Weiter ist $(1 - (Sim(x, y)^{\Psi})^{\Lambda}$ die Wahrscheinlichkeit dafür, dass die Elemente x und y in keinem der Λ Blocking-Keys übereinstimmen.

Angenommen für die Ähnlichkeit zwischen x und y gilt $Sim(x, y) = 0{,}8$. Bereits ein Blocking-Key der Länge $\Psi = 5$ führt dazu, dass die Wahrscheinlichkeit, dass sich die zwei Elemente in einem Blocking-Key treffen, nur $(0{,}8)^5 \approx 0{,}33 = 33\,\%$ beträgt. Durch Erhöhung der Anzahl der Blocking-Keys auf beispielsweise $\Lambda = 8$, kann die Wahrscheinlichkeit, dass sich die beiden Elemente in wenigstens einem der acht Blocking-Keys treffen, auf $1 - (1 - (0{,}8)^5)^8 \approx 0{,}96 = 96\,\%$ erhöht werden.

Die optimale Anzahl an Iterationen bzw. Blocking-Keys lässt sich nach [AI08] in Abhängigkeit von Ψ wie folgt bestimmen:

$$\Lambda_{opt} = \left\lceil \frac{\ln(\delta)}{\ln(1 - \tau^{\Psi})} \right\rceil . \tag{2.36}$$

Hierbei kennzeichnet τ einen Schwellwert, welcher die Ähnlichkeit von zwei Elementen bzw. Bloom-Filtern angibt, die mit einer Wahrscheinlichkeit von wenigstens $1 - \delta$ vom LSH-Verfahren als Kandidaten-Record-Paar gebildet werden sollen . Der Parameter δ gibt dabei die tolerierbare Fehlerwahrscheinlichkeit für False-non-Matches an. Wird beispielsweise eine Fehlerwahrscheinlichkeit von $\delta = 0{,}01$ akzeptiert, dann ergibt sich für $\tau = 0{,}8$ und $\Psi = 5$ die optimale Anzahl an Iterationen als $\Lambda_{opt} = \lceil \ln(0{,}01) / \ln(1 - (0{,}8)^5) \rceil = 12$.

2.3.2 Phonetisches Blocking

Beim phonetischen Blocking wird eine phonetische Codierung erzeugt, sodass Datensätze mit ähnlich klingenden Attributwerten, also ähnlicher Aussprache, dem gleichen Block zugeordnet werden. Als QIDs werden beim PPRL häufig Vor- und Nachnamen gewählt. Auch bei diesen Attributen treten Datenqualitätsprobleme (Dirty Data) auf. Beispielsweise existieren für eine Vielzahl an Namen verschiedene Schreibweisen. Bei der Aufnahme der Daten kann es dann passieren, dass unterschiedliche Schreibweisen benutzt werden. Solche Fehler bzw. Variationen entstehen vor allem, wenn Personeninformationen über das Telefon oder im Gespräch aufgenommen werden. Im laufenden Beispiel von Unterabschnitt 2.1.1 sieht man zum Beispiel die unterschiedlichen Schreibweisen von 'Bern**d**' und 'Bern**t**' sowie von 'Schmi**t**t' und 'Schmi**d**t'. Um solche Variationen zu kompensieren eignen sich phonetische Codierungen, da sie für ähnlich klingende Namen dieselbe Codierungen erzeugen, also ähnlich klingende Namen in einem Code zusammenfassen. Für das PPRL bedeutend ist, dass phonetische Codierungen durch das Zusammenfassen mehrerer Werte zu einem phonetischen Code automatisch einen gewissen Grad an Privacy bieten [KVC12]. Problematisch ist jedoch, dass phonetische Codierungen anfällig für Häufigkeitsanalysen sind [VCOV14]. Außerdem sind sie sprachabhängig [QBD96, SBB04] und decken nur phonetische Variationen ab, weswegen Eingabefehler (Vertauschung, Einfügen oder Auslassen

von Buchstaben) meist nicht kompensiert werden können [Chr06].

Eine der bekanntesten und gleichzeitig eine der ältesten phonetischen Codierungen ist *Soundex* [OR18]. Diese phonetische Codierung wird zum phonetischen Blocking im Rahmen dieser Arbeit eingesetzt. Soundex wird sehr häufig benutzt und zeichnet sich durch seine Einfachheit und eine effiziente Berechnung aus [Chr12a]. Jeder Soundex-Code besteht aus einem Zeichen (Buchstaben) gefolgt von drei Zahlen. Das Vorgehen zur Erstellung des Soundex-Codes eines Namens bzw. einer Zeichenkette ist nach [OR18] wie folgt: Zunächst wird der erste Buchstabe aus dem Namen extrahiert. Dieser Buchstabe entspricht dem ersten Zeichen im Soundex-Code. Die restlichen Zeichen werden entsprechend der Tabelle 2.5 [OR18] codiert. Hierbei entspricht '–', dass die Buchstaben ignoriert werden, demnach also keinen Code erzeugen. Als letztes werden dann gleiche aufeinander folgende Codes zusammengefasst. Außerdem wird der bisherige Soundex-Code auf drei Ziffern gekürzt bzw. erweitert, indem überzählige Ziffern entfernt werden oder der Code mit zusätzlichen Nullen ('0') aufgefüllt wird. Beispielweise ergibt sich für den Namen 'Bern**d**' der Soundex-Code 'B653'. Ebenso erzeugt der Name 'Bern**t**' den Soundex-Code 'B653'. Wie aus diesem Vorgehen jedoch ersichtlich wird, ist ein Nachteil von Soundex, dass ein Unterschied im ersten Zeichen des Namens zu einem anderen Soundex-Code führt. Außerdem bewirkt das Kürzen des Soundex-Codes auf nur drei Ziffern, dass bei längeren Namen das Ende nicht berücksichtigt wird [Chr12a].

Code	**Buchstabe**
–	A, E, I, O, U, H, W, Y
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M, N
6	R

Tabelle 2.5: Soundex Regeltabelle.

Neben Soundex existieren noch zahlreiche weitere phonetische Codierungen. Häufig genutzt werden unter anderem NYIIS [BS92], Double-Metaphone [Phi00] oder die für deutsche Namen relevante Kölner Phonetik [Pos69].

2.4 Flink

Die Umsetzung des P3RL soll mit Hilfe von Apache Flink erfolgen. Aus diesem Grund werden im folgenden Abschnitt die grundlegenden Konzepte und die Funktionsweise von Flink erläutert. Der Fokus des Abschnitts liegt auf der Beschreibung des Aufbaus von Flink-Programmen. Dabei wird betrachtet, wie diese auf einem Computer-Cluster ausgeführt werden. Außerdem wird darauf eingegangen, wie die Ausführung von Flink-Programmen überwacht, analysiert und damit bewertet werden kann. Wenn nicht anders gekennzeichnet, beziehen sich alle Ausführungen des Abschnitts auf die Dokumentation von Flink in [Thea]. Im Einzelnen beziehen sich die Ausführungen von Unterabschnitt 2.4.1 auf [Then, Them, Thed, Thei, Thec], von Unterabschnitt 2.4.2 und Unterabschnitt 2.4.3 auf [Thee, Thef, Thej], von Unterabschnitt 2.4.4 auf [Theb, Theg, Theh, Thec] und von Unterabschnitt 2.4.5 auf [Thel, Theb, Thek].

2.4.1 Einführung

Apache Flink ist ein quelloffenes Framework zur verteilten Verarbeitung von Datenströmen [Theo]. Es erleichtert die Entwicklung von parallelen Programmen zur Ausführung auf Cluster-Umgebungen. Flink kann hierbei eigenständig auf einem statischen und unter Umständen heterogenen Cluster, aber auch cloudbasiert beispielsweise in Kombination mit einem Google-compute-Engine-Cluster (GCE-Cluster)[3] ausgeführt werden. Außerdem ist die lokale Ausführung auf einem einzelnen Rechner möglich.

Mit Flink lassen sich zwei Arten von Datensätzen verarbeiten: Einerseits unendliche (*unbounded*) Datensätze, welche einem kontinuierlichen Datenstrom entsprechen, und andererseits endliche, unveränderbare (*bounded*) Datensätze. Unendliche Datensätze werden beispielsweise durch physikalische Sensoren erzeugt, welche ununterbrochen, meist innerhalb bestimmter Zeitintervalle, aktuelle Messwerte liefern.

Daneben unterstützt Flink zwei Ausführungsmodelle, welche bestimmen, wie die Verarbeitung der Daten erfolgt: Streaming und Batch. Beim Streaming erfolgt die Verarbeitung der Daten stetig, solange neue Daten erzeugt werden. Im Gegensatz dazu bezeichnet die Batch-Verarbeitung die einmalige und vollständige Verarbeitung der Daten. Hierbei läuft die Verarbeitung der Daten in endlicher Zeit ab und die genutzten Ressourcen werden freigegeben, sobald das Programm terminiert.

Eine Besonderheit von Flink ist, dass endliche Datensätze als Spezialfall von unendlichen Datensätzen angesehen werden. Das bedeutet, dass Flink einen endlichen Datensatz

[3] `https://cloud.google.com/compute/`, Zugriff: 10.03.2017

intern als endlichen Datenstrom behandelt. Hierdurch werden endliche und unendliche Datensätze in Flink grundsätzlich gleich behandelt, was ermöglicht, sie auf derselben verteilten Datenfluss-Engine von Flink auszuführen. Aus diesem Grund bildet die Stream-Verarbeitung den Kern der Flink-Architektur. Die Batch-Verarbeitung hingegen wird als Spezialfall des Streamings behandelt.

Flink kann in einer Vielzahl von Anwendungsfällen eingesetzt werden. Es ist besonders für Large-scale-Anwendungen geeignet, die auf großen Computer-Clustern mit bis zu tausenden Knoten ausgeführt werden sollen. Demzufolge kann Flink eingesetzt werden, um eine schnelle Verarbeitung auch von sehr großen Datenmengen zu ermöglichen. Außerdem kann Flink präzise Ergebnisse liefern, da es auch verzögerte (*late-arriving*) Daten oder Daten, welche in der falschen Reihenfolge eintreffen (*out-of-order*), berücksichtigt. Ein weiterer Vorteil von Flink ist, dass es eine Vielzahl an bekannten Frameworks unterstützt. Neben dem Hadoop-distributed-File-System (HDFS)[4] sind dies unter anderem Apache Cassandra[5] oder Apache Kafka[6]. Schließlich bietet Flink mehrere Bibliotheken und APIs zur Erstellung von Programmen an. Neben der DataStream-API zur Stream-Verarbeitung und der DataSet-API zur Batch-Verarbeitung (siehe Unterabschnitt 2.4.4) bietet Flink beispielsweise FlinkML für Maschinelles Lernen, Gelly für die Graphverarbeitung und eine Table-API für die Verarbeitung von relationalen Daten. Die Programmierung kann mit Scala oder Java erfolgen, für die entsprechende Versionen der APIs zur Verfügung stehen. In Rahmen dieser Arbeit erfolgt die Programmierung mit Java, weshalb die Scala-Aspekte nicht näher betrachtet werden.

2.4.2 Flink-Programme

Ein Flink-Programm ist ein Programm, welches Transformationen auf Datenströmen (Streams) durchführt. Transformationen sind Operationen, die einen oder mehrere Streams als Eingabe erhalten und als Ausgabe einen oder mehrere Streams erzeugen. Mehrere Transformationen können miteinander kombiniert werden, um so komplexere Operationen zu realisieren. Streams können potentiell unendlich sein und bestehen aus einzelnen Datensätzen (Records).

Jedes Flink-Programm erhält als Eingabe eine oder mehrere Datenquellen (*Source*), welche die Eingabe-Streams erzeugen. Eingabe-Streams können beispielsweise durch Einlesen von (verteilten) Dateien oder durch Abfrage von Datenbanken erzeugt werden. Im Flink-Programm werden dann Transformationen spezifiziert, welche benutzerdefinierte Funk-

[4] `http://hadoop.apache.org/`, Zufriff: 10.03.2017
[5] `http://cassandra.apache.org/`, Zugriff: 10.03.2017
[6] `https://kafka.apache.org/`, Zugriff: 10.03.2017

tionen realisieren und die Eingabe-Daten auf gewünschte Art und Weise verarbeiten bzw. verändern. Beispiele für Transformationen sind Mappings, Filterungen und Aggregationen. Das Flink-Programm endet in einer oder mehreren Datensenken (*Sink*). Diese Datensenken konsumieren die erzeugten Ausgabe-Streams und geben die Ergebnisse zurück. Die Ausgabe der Ergebnisse kann beispielsweise durch Schreiben der Daten in (verteilte) Dateien erfolgen.

Flink-Programme werden zur Ausführung in *Streaming-Dataflows* übersetzt. Ein solcher kann als gerichteter azyklischer Graph (DAG) dargestellt werden. Er enthält die Datenquellen und Datensenken bzw. den jeweiligen Operator als Start- bzw. Endknoten. Die im Programm spezifizierten Transformationen werden auf Transformationsoperatoren abgebildet, wobei eine Transformation einen oder mehrere Transformationsoperatoren erzeugt. In Anlehnung an [Thee] ist in Abbildung 2.3 ein Beispiel für den Streaming-Dataflow eines einfachen Flink-Programmes dargestellt. Das dargestellte Flink-Programm enthält zwei Transformationen, die durch 'map' und 'groupBy/reduce' gekennzeichnet sind. Dabei entspricht die erste Transformation einem Mapping und die zweite einer Gruppierung mit anschließender Aggregierung der Daten. Für genauere Erklärungen zu den Transformationen in Flink siehe Unterabschnitt 2.4.4.

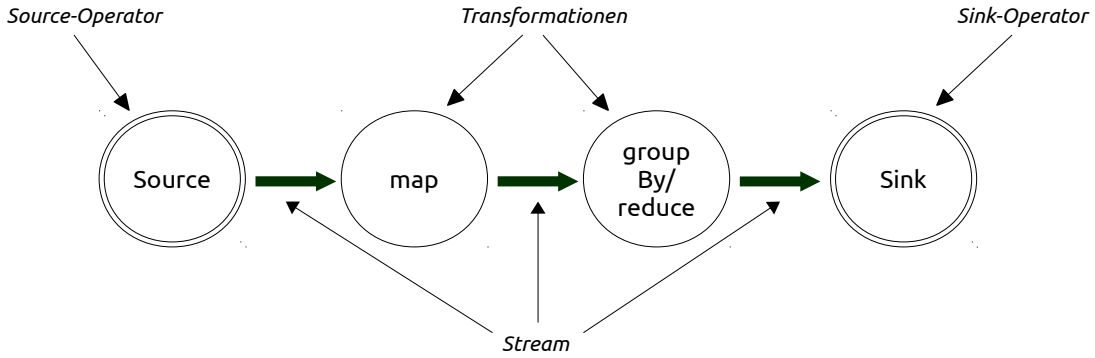


Abbildung 2.3: Beispiel: Streaming-Dataflow-Graph eines einfachen Flink-Programms.

Durch den oben beschriebenen Aufbau von Flink-Programmen sind sie bereits von sich aus zur verteilten und parallelen Ausführung geeignet. Die Streams können in *Stream-Partitions* aufgespalten werden, wodurch Operatoren gleichzeitig auf den einzelnen Partitionen des Streams ausgeführt werden können. Ein Operator wird dazu in *Operator-Subtasks* aufgeteilt, welche unabhängig voneinander in verschiedenen Threads und auf verschiedenen Rechnern ausgeführt werden können. Die Anzahl der Operator-Subtasks wird durch den Parallelitätsgrad des jeweiligen Operators bestimmt. In Flink kann jedem Operator ein eigener Parallelitätsgrad zugewiesen werden. Darüber hinaus ist es auch möglich, einem gesamten Programm und damit allen Operatoren des Programms einen Defaultwert für die Parallelität zuzuweisen. Dies entspricht dann der Flink-Job-

Parallelität, welche im Folgenden mit Υ bezeichnet wird. Wichtig ist, dass der maximal erlaubte Parallelitätsgrad von Flink bzw. des Flink-Clusters nicht überschritten wird. Dieser wird durch die Anzahl der verfügbaren *Task-Slots* (siehe Unterabschnitt 2.4.3) im Flink-Cluster bestimmt.

Zwischen den Operatoren werden die Daten des Streams unterschiedlich weiter transportiert. Dies ist abhängig von der Art zweier aufeinander folgender Operatoren. Manche Operatoren erfordern eine Umverteilung der Daten und damit eine Änderung der Partitionierung des Streams. Hierbei spricht man von *Redistributing-Streams*, also umzuverteilenden Streams. Bei diesen sendet jeder Operator-Subtask die Daten zu verschiedenen Zieltasks. Eine Ordnung innerhalb der Daten wird nur je Paar von Sende- und Empfangstask beibehalten. Diese Art von Umverteilung tritt beispielsweise vor Gruppierungen mit anschließenden Aggregationen auf. Erfordern Operatoren hingegen keine Umverteilung der Daten, so bleibt die Partitionierung der Streams sowie die Ordnung der Elemente innerhalb des Streams erhalten. Hierbei werden die Daten zwischen zwei Operatoren einfach weitergeleitet, weswegen diese als *Forwarding-Streams* bezeichnet werden. In Anlehnung an [Thee] ist in Abbildung 2.4 die parallele Sicht des Streaming-Dataflow-Graphen aus Abbildung 2.3 abgebildet. Die Parallelität der Operatoren in diesem Beispiel beträgt 2 außer für die Datensenke, welche nur mit Parallelität 1 ausgeführt wird. Das bedeutet, dass die Daten zunächst parallel von zwei Source-Operatoren gelesen werden. Diese erzeugen jeweils eine Stream-Partition, welche zusammen den gesamten Eingabe-Stream bilden. Die Daten zwischen dem Subtask [1] (bzw. [2]) des Source-Operators werden nachfolgend direkt zum Subtask [1] (bzw. [2]) des Map-Operators weitergeleitet. Hierbei handelt es sich also um Forwarding-Streams. Im Gegensatz dazu werden die Daten zwischen den Subtasks des Map-Operators und denen des GroupBy/reduce-Operators umverteilt, da dies die Gruppierung erforderlich macht. Schließlich erfolgt die letzte Umverteilung der Daten zwischen den Subtasks des GroupBy/reduce-Operators und dem Sink-Operator. Dies ist notwendig, da der Sink-Operator mit Parallelität 1 ausgeführt wird, weswegen die Daten hierfür zusammengeführt werden müssen.

Aufeinander folgende Operator-Subtasks können außerdem, falls sie keine Umverteilung der Daten erfordern, miteinander verkettet werden. Diese Verkettung führt zur Bildung von Tasks, wobei ein Task aus einem oder mehreren Subtasks besteht. Jeder Task wird dann innerhalb eines Threads ausgeführt. Diese Verkettung von Operatoren wird von Flink eingesetzt, da es den Overhead beim Wechsel von Threads sowie die Latenz verringert und den Durchsatz erhöht. Für das Beispiel aus Abbildung 2.3 bzw. Abbildung 2.4 bedeutet dies, dass der Source- und Map-Operator miteinander verkettet werden. Damit ergeben sich insgesamt drei Tasks. Da die ersten beiden Tasks mit Parallelität 2 ausgeführt werden, ergeben sich insgesamt fünf Subtasks und damit Threads. Es ergeben sich jeweils zwei

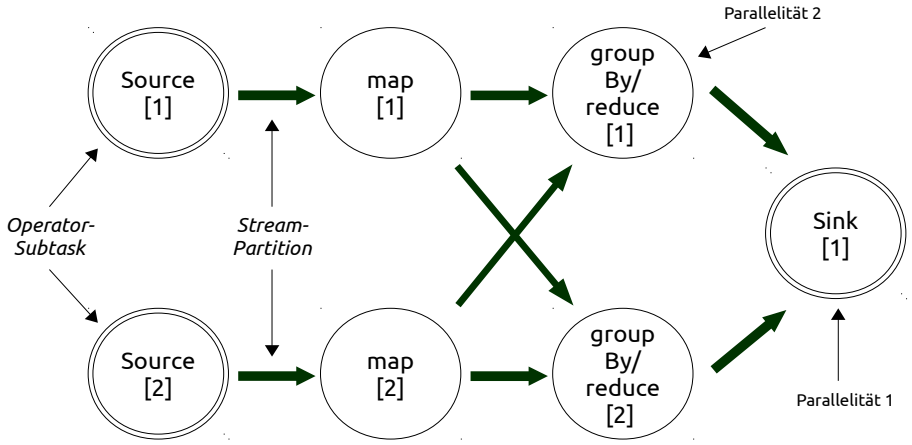


Abbildung 2.4: Beispiel: Paralleler Streaming-Dataflow-Graph eines einfachen Flink-Programms.

Subtasks für den verketteten Source-map-Operator sowie den GroupBy/reduce-Operator und ein Subtask für den Sink-Operator aufgrund der Parallelität von 1.

2.4.3 Verteilte Laufzeitumgebung

Flink startet während der Laufzeit zwei Arten von Prozessen, die als Job-Manager und Task-Manager bezeichnet werden:

Job-Manager Ein Job-Manager (JM) entspricht einem Koordinator- bzw. Master-Knoten. Dieser koordiniert die verteilte Ausführung und legt dazu unter anderem den Ablaufplan für Tasks fest und reagiert auf beendete Tasks sowie auf Fehler während der Ausführung. In einem Flink-Cluster existiert mindestens ein Job-Manager. Falls mehrere Job-Manager in einem Cluster vorhanden sind, so übernimmt einer von ihnen die Führungsrolle und wird als Leader bezeichnet, während die anderen sich im Standby-Modus befinden.

Task-Manager Ein Task-Manager (TM) entspricht einem Worker-Knoten. Dieser führt Teile des parallelen Programms, die Tasks bzw. Subtasks eines Dataflows aus. Dazu ist jeder Task-Manager mit einem Job-Manager verbunden. In einem Flink-Cluster gibt es mindestens einen Task-Manager. Zur Festlegung wie viele Tasks ein TM akzeptiert, wird die Anzahl seiner *Task-Slots (TS)* definiert. Ein oder mehrere Subtasks des Jobs werden den Task-Slots zugewiesen und dort in separaten Threads ausgeführt. Jeder Task-Manager ist ein JVM-Prozess, der diese Threads abarbeitet. Des Weiteren entspricht jeder Task-Slot einem festen Anteil der dem Task-Manager zugewiesenen Ressourcen. Werden einem Task-Manager zum Beispiel vier Task-Slots zugewiesen, so erhält jeder $\frac{1}{4}$ des vom TM verwalteten Speichers. Diese Aufteilung der Ressourcen wird als *Resource-Slotting* bezeichnet

und hat den Vorteil, dass das konkurrieren um den verwalteten Speicher zwischen Tasks von verschiedenen Jobs vermieden wird. Allerdings betrifft das Resource-Slotting nur die Speicheraufteilung und führt zu keiner Isolation der CPU. Außerdem ist es möglich, dass sich mehrere Subtasks des gleichen Jobs einen Task-Slot teilen (*Slot-Sharing*). Dadurch kann ein Task-Slot die gesamte Pipeline eines Jobs beinhalten. Genauer kann in einem Task-Slot eine Pipeline von sukzessiven Tasks parallel abgearbeitet werden. Weiterhin benötigt ein Flink-Cluster mindestens so viele Task-Slots, wie der maximale Parallelitätsgrad mit dem die Operatoren ausgeführt werden sollen. Wie bereits erwähnt, wird die Parallelität mit der ein Flink-Programm maximal ausgeführt werden kann durch die Anzahl der vorhandenen Task-Slots bestimmt. Zu beachten ist, dass die Flink-Job-Parallelität im Allgemeinen nicht gleich der Anzahl der eingesetzten Worker-Knoten ist. Stattdessen gibt die Flink-Job-Parallelität an, wie viele Operator-Subtasks jeder Operator des Flink-Programms erzeugt.

Dem Beispiel aus Unterabschnitt 2.4.2 folgend, ist in Abbildung 2.5 die Zuordnung der Subtasks zu den Task-Slots für zwei Task-Manager mit jeweils einem Task-Slot dargestellt. Hierdurch ergeben sich insgesamt zwei verfügbare Task-Slots im Flink-Cluster, was die minimal erforderliche Anzahl an Task-Slots für dieses Beispiel darstellt. Weiterhin ist erkennbar, dass der Task-Slot des Task-Managers 1 die komplette Pipeline des Jobs mit der Sequenz 'Source-map-groupBy/reduce-Sink' umfasst.

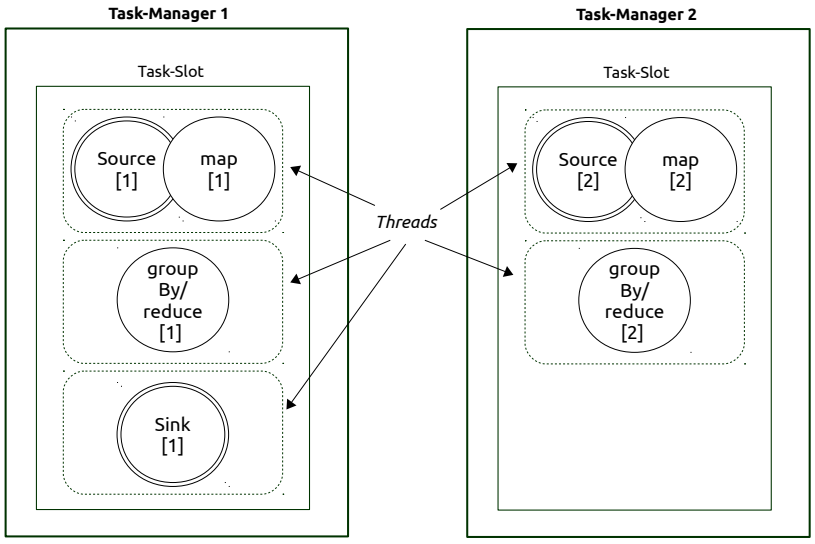


Abbildung 2.5: Beispiel: Zuordnung von Subtasks zu Task-Slots.

Neben diesen beiden Prozessen gibt es noch einen *Client*, welcher einen Dataflow vorbereitet und zu dem Job-Manager sendet. Der Client ist kein Teil der Programmausführung von Flink, weswegen er, nachdem er den Start des Programms ausgelöst hat, nicht mehr aktiv sein muss. Das Auslösen der Programmausführung geschieht entweder als Teil des Java-

bzw. Scala-Programms oder in Form eines Kommandozeilenprozesses. In Abbildung 2.6 ist in Anlehnung an [Thef] der grundlegende Aufbau der Flink-Laufzeitumgebung dargestellt. Der Client erzeugt aus dem Programmcode einen Dataflow- bzw. Job-Graphen. Dieser ist eine Repräsentation des Dataflows mit den Operatoren und ihren Eigenschaften (z. B. Parallelitätsgrad) sowie den jeweiligen Zwischenergebnissen. Dieser Dataflow-Graph wird zum Starten eines Jobs an den Job-Manager gesendet. Der Job-Manager überführt den Graphen dann in eine parallele Repräsentation. Der Job-Manager legt den Ablaufplan der Tasks fest und weist sie den Task-Managern bzw. ihren Task-Slots zu. Zwischen den Task-Managern können Zwischenergebnisse in Form von Data-Streams ausgetauscht werden. Die Task-Manager senden regelmäßig Heartbeat-Nachrichten an den Job-Manager. Außerdem leiten sie den Status der Tasks sowie diverse Statistiken zum Job-Manager weiter. Der Job-Manager überwacht dabei den gesamten Job-Status und damit die Ausführung des Programms. Statusmeldungen sowie Ergebnisse können dann zurück an den Client gesendet werden.

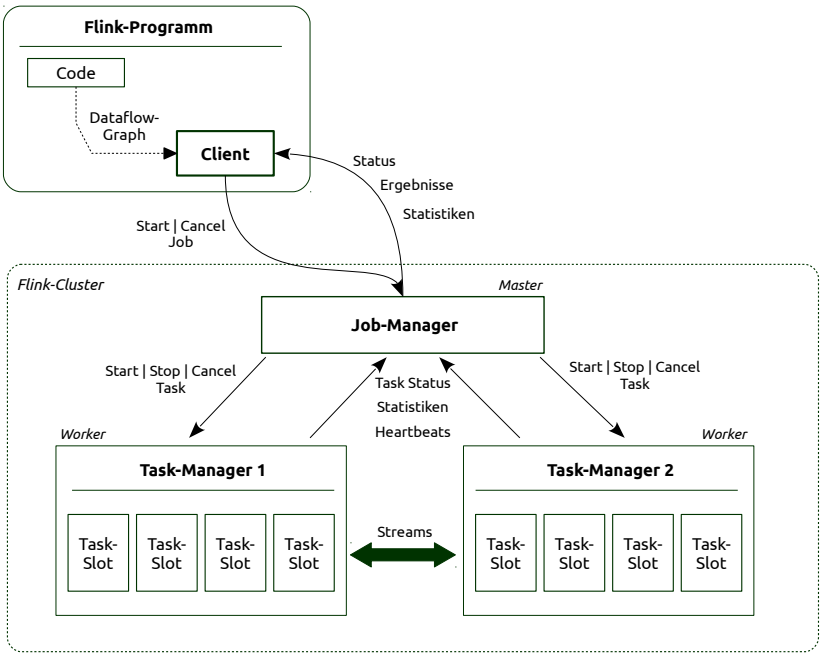


Abbildung 2.6: Architektur der Flink-Laufzeitumgebung.

2.4.4 Flink-APIs

Flink-Programme zeichnen sich dadurch aus, dass sie durch Transformationen spezifiziert sind. Zur Erstellung von Programmen werden von Flink zwei APIs angeboten. Zur Batch-Verarbeitung steht die DataSet-API und für die Stream-Verarbeitung die DataStream-API zur Verfügung.

Durch verschiedene Compilierungsprozesse kann aus dem Programm, welches mit Hilfe der APIs spezifiziert wird, ein Job-Graph erstellt werden. Dieser dient als Eingabe für den Job-Manager und kann als paralleler Datenfluss mit verschiedenen Tasks, die Streams konsumieren und erzeugen, angesehen werden.

Äquivalent zu der DataSet- und der DataStream-API werden von Flink die zwei Klassen `DataSet` und `DataStream` bereitgestellt. Beide Klassen repräsentieren die Daten in einem Flink-Programm und können als unveränderliche Sammlung von Daten angesehen werden. Nach der Erstellung können keine Daten mehr hinzugefügt oder entfernt werden. Stattdessen wird die Datensammlung durch eine Datenquelle (Source) erzeugt und durch Anwendung von Transformationen verändert. Die Transformationen sind durch die Methoden der APIs realisiert und können so zur Erstellung des Flink-Programms genutzt werden.

Für die Datentypen der Elemente in einem DataSet bzw. DataStream gibt es einige Einschränkungen, da Flink diese analysiert, um effiziente Ausführungsstrategien zu bestimmen. Unter anderem werden folgende Datentypen unterstützt:

- Tupel-Typen
- Plain-old-Java-Objects (POJOs)
- Primitive Typen (z. B. `Integer`, `String`, `Double`)
- Reguläre Klassentypen.

Tupel-Typen sind zusammengesetzte Datentypen, welche aus einer festen Anzahl von Feldern verschiedenen Typs bestehen. Hierbei ist es auch möglich, mehrere Tupel-Typen ineinander zu verschachteln. POJOs sind Java-Klassen, die bestimmte Eigenschaften erfüllen. Zunächst muss die Klasse als `public` gekennzeichnet sein. Weiterhin muss die Klasse einen öffentlichen Default-Konstruktor ohne Argumente haben und alle Felder müssen ebenfalls als `public` gekennzeichnet oder durch Getter- bzw. Setter-Funktionen zugreifbar sein und von Flink unterstützt werden. Schließlich sind reguläre Klassentypen Klassen, welche keine POJOs sind. Diese müssen jedoch serialisierbare Felder besitzen. Der Vorteil von POJOs ist, dass Flink diese effizienter verarbeiten kann als reguläre Klassentypen.

Einige Transformationen erfordern eine Gruppierung der Elemente und damit die Spezifizierung von *Keys*. Keys werden in Flink virtuell über die Daten definiert. Beispielsweise können als Key für Tupel-Typen ein oder mehrere Felder dienen. Gleiches gilt auch für die Felder eines POJO. Allerdings gibt es auch für Keys einige Einschränkungen bezüglich des Datentyps, beispielsweise müssen generische Typen das `Comparable`-Interface implementieren.

Während die bisherigen Ausführungen sowohl für die DataSet- als auch die DataStream-API zutreffen, unterscheiden sich die angebotenen Transformationen und deren genaue Funktionsweise je nach gewählter API. Da die DataStream-API nicht relevant für diese Arbeit ist, wird sie im Folgenden nicht näher betrachtet. Die DataSet-API bietet zahlreiche Transformationen auf DataSets an, durch deren Kombination komplexe Flink-Programme erstellt werden können. Eine Auswahl von typischen und für diese Arbeit relevanten Transformationen ist in Tabelle 2.6 dargestellt.

Transformation	**Beschreibung**
Map	Anwendung einer benutzerdefinierten Funktion auf jedes Element, wobei ein Eingabe-Element genau ein Ausgabe-Element erzeugt.
FlatMap	Anwendung einer benutzerdefinierten Funktion auf jedes Element, wobei ein Eingabe-Element beliebig viele Ausgabe-Elemente erzeugt.
Filter	Anwendung einer benutzerdefinierten Funktion auf jedes Element, welche als Ausgabe einen Wahrheitswert (true oder false) liefert. Nur Elemente, für welche der Wert true zurückgeliefert wird, bleiben erhalten. Die anderen Elemente werden entfernt. Die Funktion darf keine Modifikation an den Elementen vornehmen.
Reduce	Anwendung einer benutzerdefinierten Funktion auf jedes Element, wobei sukzessive zwei Elemente in ein Element kombiniert werden, sodass am Ende ein Ausgabe-Element erzeugt wird. Die Reduce-Transformation kann auf die gesamte Datenmenge oder auf eine gruppierte Datenmenge angewandt werden. Letzteres führt zu einem Ausgabe-Element pro Gruppe.
GroupReduce	Anwendung einer benutzerdefinierten Funktion auf jede Gruppe einer gruppierten Datenmenge. Hierbei dient die gesamte Gruppe als Eingabe und als Ausgabe können beliebig viele Elemente erzeugt werden.
Cross	Die Cross-Transformation erzeugt das kartesische Produkt zweier DataSets. Dazu erzeugt es alle Paare von Elementen aus den beiden Eingabe-Datensätzen.
Union	Die Union-Transformation vereinigt zwei Eingabe-DataSets zu einem Ausgabe-DataSet.

Tabelle 2.6: Auswahl einiger Transformationen der DataSet-API.

2.4.5 Monitoring mit Flink

Um den Status sowie Statistiken von laufenden und abgeschlossenen Jobs zu erhalten, bietet Flink eine REST-API an, welche in Form eines Webservers Teil des Job-Managers

ist. Diese Monitoring-API liefert Daten im JSON-Format und erlaubt eine Vielzahl von HTTP-GET-Anfragen mit denen man beispielsweise folgende Informationen erhält:

- Zusammenfassung des Status des Flink-Clusters
- Liste aller Jobs (gruppiert nach ihrem Status)
- Detaillierte Informationen über einen Job, z. B.
  - ID
  - Name
  - Start- und Endzeit
  - Ausführungszeit (Dauer)
  - Benutzerdefinierte Ausführungskonfiguration
  - Informationen über einzelne Tasks bzw. Operationen
  - Dataflow-Plan
  - Metriken und Akkumulatoren.

Daneben ist es auch möglich, Jobs mit POST-Anfragen zu starten bzw. mit DELETE-Anfragen abzubrechen. Wie in obiger Auflistung ersichtlich, bietet Flink die Möglichkeit Metriken und Akkumulatoren zu sammeln und über die REST-API zugänglich zu machen. Dies ermöglicht es, verschiedene Informationen während der Programmausführung zu erfassen und später auszuwerten. Mit Hilfe eines Akkumulators können beliebige akkumulierte Werte erfasst werden. Dazu hat jeder Akkumulator eine Funktion zum Hinzufügen von Werten (add) und eine Funktion, welche aus mehreren Teil-Akkumulatoren den akkumulierten Wert berechnet (merge). Durch die Möglichkeit der Zusammenführung von mehreren Teil-Akkumulatoren eignen sich diese zur globalen Erfassung von Werten, da sie die Werte von verschiedenen Ausführungseinheiten miteinander kombinieren können. Ein einfaches Beispiel für einen Akkumulator, welcher von Flink unterstützt wird, ist ein Zähler. Hierbei kann ein Zähler erhöht werden (add) und mehrere Zähler können durch Summenbildung zusammengefasst werden (merge). Außer den Zählern werden auch Histogramme als Umsetzung für einen Akkumulator von Flink angeboten. Neben Akkumulatoren werden außerdem verschiedene Metriken von Flink unterstützt. Diese sind in Tabelle 2.7 aufgeführt. Im Gegensatz zu Akkumulatoren erfassen Metriken nur lokale Werte, das heißt, sie werden nicht auf Ebene des Task-Managers oder Job-Managers zusammengefasst.

Zusätzlich zu den Informationen bezüglich laufender und abgeschlossener Flink-Jobs erfasst Flink diverse Systemmetriken. Diese können über konfigurierte Reporter, die un-

abhängig von der REST-API sind, veröffentlicht werden. Einige Beispiele für solche Systemmetriken sind:

- CPU-Ausnutzung der JVM
- Genutzter Heap-Speicher
- Anzahl aktiver Threads
- Anzahl gelesener und übermittelter Bytes
- Anzahl gelesener und übermittelter Records
- Häufigkeit und Dauer der Garbage-Collection.

Typ	**Beschreibung**
Zähler	Dienen zum Zählen von Ereignissen und können dazu erhöht oder verringert werden.
Gauge (Messwert)	Ein Gauge wird genutzt, um einen Messwert beliebigen Typs zu veröffentlichen.
Histogramm	Histogramme messen die Verteilung von Werten des Typs `long`.
Meter	Ein Meter misst den Durchsatz, indem es das Eintreten von Ereignissen registriert.

Tabelle 2.7: Flink-Metriken.

Insgesamt bietet Flink damit umfangreiche Möglichkeiten, die Ausführung von Programmen zu überwachen und Metriken bzw. Statistiken zu sammeln. Dadurch wird die Evaluierung von Flink-Programmen wesentlich vereinfacht.

2.5 Verwandte Arbeiten

Das Record-Linkage sowie das Privacy-preserving-Record-Linkage werden zusammen seit mehr als 70 Jahren erforscht. Insgesamt lässt sich daher in diesem Bereich ein Literaturangebot in erheblichem Umfang finden. Aus diesem Grund sollen in diesem Abschnitt kurz die wichtigsten Arbeiten mit Relevanz für diese Arbeit vorgestellt werden. Grundlegend wird hierbei zwischen dem herkömmlichen Record-Linkage und dem Privacy-preserving-Record-Linkage unterschieden. Zusätzlich wird betrachtet, welche Anstrengungen zur Parallelisierung des RL bzw. des PPRL unternommen wurden.

Record-Linkage Die ersten wissenschaftlichen Betrachtungen bezüglich des Record-Linkages lassen sich ungefähr seit Ende der 40er Jahre des letzten Jahrhunderts finden [Dun46]. Die Anwendungsgebiete lagen damals vor allem im Bereich der Volkszählung

(Census-Daten) und der Demographie, beispielsweise zum Abgleich und zur Auswertung von registrierten Lebensereignissen (Geburt, Heirat, Tod) [NKAJ59, NK62, New67]. Aber auch medizinische Studien, wie die Untersuchung von genetischen Defekten bzw. erblichen Krankheiten, wurden bereits berücksichtigt [NKAJ59]. Die mathematischen Grundlagen für das RL folgten in [FS69]. Über die Zeit wurden verschiedene Untersuchungen durchgeführt und es entstanden zahlreiche Verfahren zur Durchführung des RL. Einen Überblick über die Resultate und Methoden sind unter anderem in [HSW07], [EIV07], [KR10] und [Chr12a] zu finden.

Privacy-preserving-Record-Linkage Die zusätzliche Anforderung, das Record-Linkage unter Einhaltung der Privatsphäre und des Datenschutzes durchzuführen, führte zur Untersuchung neuer Aspekte und damit zur Entwicklung neuer Verfahren. Ein umfangreicher Überblick ist vor allem in [VCV13] zu finden. Phonetisches Blocking bzw. phonetische Codierungen im Allgemeinen werden dabei zum Beispiel in [KV09] und [KVC12] betrachtet. Verfahren unter Verwendung von LSH wurden ebenfalls bereits mehrfach untersucht und entwickelt, so vor allem in [KL10], [Dur12], [KV15] und [KV16].

Parallel-Record-Linkage Zur Verbesserung der Skalierbarkeit wurden für das RL verschiedene Ansätze entwickelt. Zum einen werden Graphikprozessoren (GPUs) genutzt, um das RL zu parallelisieren. Beispiele hierfür sind [VCL10], [FPS+13] und [NKH+13]. Eine andere Möglichkeit stellt die Parallelisierung mit Hilfe von Hadoop MapReduce[7] dar und ist unter anderem in [KL07], [WWL+10], [DBGH11] und [KTR12a] vorzufinden. Vergleichbar zu Flink lassen sich mit dem MapReduce-Framework von Hadoop einfach parallel ausführbare Programme erstellen. Das generelle Vorgehen der Ansätze ist, dass die Daten in der Map-Phase gelesen und in Blöcke partitioniert werden, sodass ähnliche Datensätze demselben Block zugeordnet werden. In der Reduce-Phase erhalten die Rechner alle Datensätze eines Blocks und führen dann den Vergleich bzw. die Ähnlichkeitsberechnung für die Record-Paare durch. Damit bilden derartige Verfahren eine gewisse Grundlage für das im Rahmen dieser Arbeit entwickelte P3RL mit Flink.

Parallel-Privacy-preserving-Record-Linkage Auch für das P3RL existieren bereits einige Verfahren. Ebenfalls kommen hier GPUs und Verfahren basierend auf MapReduce zum Einsatz. So wird in [SKB+15] ein Verfahren vorgestellt, welches Filter-Techniken zur Reduzierung des Suchraums einsetzt. Diese Filter-Techniken werden auf Bloom-Filter angewandt und können parallel auf GPUs ausgeführt werden. In [KV13, KV14] wird erneut MapReduce genutzt, um das PPRL zu parallelisieren. Außerdem kommen Bloom-Filter in Verbindung mit LSH als Blocking-Verfahren zur Anwendung. Damit können diese beiden Arbeiten als wesentliche Grundlage dieser Arbeit angesehen werden. Wesentlicher Unterschied ist zunächst, dass im Rahmen dieser Arbeit Flink statt MapReduce

[7] `http://hadoop.apache.org/`, Zugriff: 10.03.2017

zur Anwendung kommt. Wie in Abschnitt 2.4 beschrieben, unterstützt Flink zahlreiche Funktionen und erweitert damit die Möglichkeiten von MapReduce. Ferner kommt in [KV13] bzw. [KV14] MinHashing (JLSH) zur Anwendung, wohingegen im Rahmen dieser Arbeit das HLSH eingesetzt wird (siehe hierzu Unterabschnitt 2.3.1). Der generelle Ansatz der beiden Arbeiten ist wie folgt: Zu Beginn werden die Datensätze im HDFS gespeichert. In der Map-Phase werden die LSH-Keys berechnet. Anhand dieser erfolgt die Aufteilung (Gruppierung) der Records zu den Reduce-Tasks. In der Reduce-Phase wird schließlich die Ähnlichkeit zwischen allen Bloom-Filtern mit übereinstimmenden LSH-Keys bestimmt. Um das Problem von doppelten Kandidaten-Record-Paaren, die im Zusammenhang mit LSH entstehen können, zu vermeiden, wurde eine Strategie entwickelt, bei der zwei MapReduce-Jobs verkettet ausgeführt werden. Der erste MapReduce-Job berechnet dabei nur die Kandidaten-Record-Paare und gibt deren Identifikatoren (IDs) aus. Der zweite Job leitet die Ergebnisse des ersten Jobs an die Reduce-Tasks weiter, sodass eine Gruppierung von identischen Kandidaten-Record-Paaren erfolgt, wodurch mehrfache Vergleiche derselben Kandidaten-Paare vermieden werden. Problematisch bei den beiden Arbeiten ist, dass die Evaluation der umgesetzten Verfahren nur für bis zu vier Knoten und nur für maximal 300.000 Datensätze durchgeführt wurde. In der Realität hingegen sind Datenmengen mit mehreren Millionen Datensätzen üblich. Aus der durchgeführten Evaluation kann deshalb die praktische Einsatzfähigkeit der Verfahren beim Umgang mit großen Datenmengen nicht vollständig beurteilt werden.

Kapitel 3

Parallel-Privacy-preserving-Record-Linkage mit Flink

In diesem Kapitel wird zunächst das Grundkonzept zur Durchführung des P3RL unter Verwendung von Flink näher erläutert. Weiterhin wird das im Rahmen dieser Arbeit entwickelte Java-Framework[8] zur Durchführung des P3RL beschrieben. Dabei wird auf die umgesetzten Verfahren näher eingegangen und es wird dargelegt, wie die Realisierung der Verfahren mit Flink erfolgt.

3.1 Grundkonzept

Ein wesentliches Ziel dieser Arbeit ist es, den gesamten PPRL-Prozess einschließlich der Evaluation von Verfahren effizienter zu gestalten. Aus diesem Grund soll der komplette PPRL-Prozess mit Flink umgesetzt werden. Eingeschlossen hiervon ist insbesondere der Schritt der Vorverarbeitung, welcher lokal von den einzelnen Parteien durchgeführt wird. Wie in Unterunterabschnitt 2.1.2.1 beschrieben, werden in diesem Schritt das Data-Cleaning und die Maskierung der einzelnen Datensätze durchgeführt. Da die Parteien mitunter mehrere Millionen Datensätze verwalten, ist dieser Schritt ebenfalls rechenintensiv und kann zu langen Ausführungszeiten führen. Bei der Verwendung von Bloom-Filtern müssen beispielsweise für jedes entstehende Q-Gramm k Hashfunktionen berechnet werden. Eine einzelne solche Berechnung ist zwar günstig, in der Summe können diese jedoch signifikant Zeit in Anspruch nehmen. Daher ist auch für diesen Schritt eine Parallelisierung mit Flink wünschenswert. Jede Partei könnte dazu ein (eher kleines) Computer-Cluster betreiben, auf dem dann die Vorverarbeitung der Daten durchgeführt wird. Vorteilhaft

[8] `https://github.com/gen-too/master_thesis`

bei diesem Vorgehen ist, dass so die Maskierung der Daten schneller erfolgen kann. Vor allem bei sehr großen Unternehmen oder bei sozialen Netzwerken verändern sich die zur Zeit registrierten Kunden bzw. Nutzer stetig. Einerseits können sich Personen stets abmelden oder ihren Vertrag kündigen. Genauso können auch jederzeit neue Personen in die Datenbank aufgenommen werden. Andererseits ändern sich die gespeicherten Daten, beispielsweise durch einen Umzug einer Person. Dies führt dazu, dass die alte Codierung entfernt und eine neue Codierung berechnet werden muss. Gerade bei Anwendungen, welche ein kurzfristiges Linkage in Echtzeit verlangen, ist die Parallelisierung der Vorverarbeitung daher sinnvoll.

Die Grundidee für die Durchführung des parallelen Linkages entspricht im Wesentlichen dem bekannten Vorgehen beim PPRL. Die am PPRL teilnehmenden Parteien senden zunächst ihre codierten Daten an die Linkage-Unit. Diese speichert die Daten anschließend im HDFS. Durch die Speicherung im HDFS werden die Daten in kleine Blöcke partitioniert, auf die einzelnen Knoten verteilt und nach Bedarf repliziert. Zu Beginn des eigentlichen Linkages werden die so verteilten Daten parallel von den Rechnern im Cluster gelesen. Im Anschluss daran wird das Blocking nach einem zuvor bestimmten Verfahren durchgeführt. Unabhängig von den genauen Verfahren werden die Datensätze anhand ihres Blocking-Keys im Cluster umverteilt, sodass ein Rechner alle Datensätze eines Blocks erhält. Die einzelnen Rechner erzeugen dann parallel die Kandidaten-Record-Paare innerhalb der Blöcke, bestimmen die Ähnlichkeit zwischen den Datensätzen und führen letztlich die Klassifikation durch. Alle Paare mit einem hinreichenden Ähnlichkeitswert werden als Match betrachtet und von den Rechnern als Ergebnis in das HDFS geschrieben. Hierbei können die einzelnen Rechner ihre Ergebnisse parallel in eine oder mehrere Dateien schreiben.

Zur Evaluierung kann die REST-Schnittstelle des Flink-Job-Managers in Verbindung mit den Metriken und Akkumulatoren, welche während der Ausführung gesammelt bzw. berechnet werden, genutzt werden. Metriken und Akkumulatoren können in Flink genutzt werden, um wichtige Kennzahlen, wie beispielsweise die Anzahl der generierten Kandidaten-Record-Paare, zugreifbar zu machen. Auf dieser Basis lassen sich dann Erkenntnisse über das genutzte Verfahren gewinnen.

3.2 Umzusetzende Blocking-Verfahren

Wie in Unterunterabschnitt 2.1.2.2 beschrieben, ist das Blocking unerlässlich für ein performantes und skalierbares PPRL. Auch für die Parallelisierung ist entscheidend, welches Blocking-Verfahren eingesetzt wird. Bestenfalls sollte das Blocking die Datensätze

gleichmäßig über die Rechner im Cluster verteilen, sodass maximaler Gewinn aus der Parallelisierung des PPRL erzielt werden kann. In diesem Zusammenhang ist es von Interesse, wie gut sich die verschiedenen, bereits entwickelten Blocking-Verfahren für die Parallelisierung eignen und inwieweit durch die Parallelisierung neue Probleme und Herausforderungen entstehen. Der Fokus der Arbeit liegt daher auf der Umsetzung und Analyse mehrerer Blocking-Verfahren für das P3RL. Insgesamt werden im Rahmen dieser Arbeit die folgenden drei Möglichkeiten zur Durchführung des Blockings betrachtet:

Nested-Loop Beim Nested-Loop werden alle möglichen Record-Paare betrachtet, womit dieser Ansatz vollständig auf die Durchführung des Blockings verzichtet. Die Bezeichnung des Verfahrens ist darauf zurückzuführen, dass die Menge aller Record-Paare durch zwei ineinander geschachtelte Schleifen über die zwei Eingabe-Datenmengen berechnet werden kann. Der Grund für die Umsetzung dieses Verfahren ist, dass dieser Ansatz sich als Basis für den Vergleich verschiedener Verfahren eignet. Da alle möglichen Record-Paare miteinander verglichen werden, erzeugt dieses Verfahren die maximal mögliche Pairs-Completeness und einen sehr guten Richtwert für die Pairs-Quality (vgl. Unterabschnitt 2.1.3).[9] Außerdem sollte Nested-Loop hinsichtlich der Ausführungszeit den schlechtesten Wert liefern. Wird durch ein Blocking-Verfahren eine längere Ausführungszeit als für Nested-Loop erreicht, so werden entweder sehr viele Vergleiche doppelt ausgeführt oder das Blocking-Verfahren ist so aufwändig, dass kein Performanz-Vorteil erreicht werden kann.

Locality-sensitive-Hashing (LSH) LSH wurde detailliert in Unterabschnitt 2.3.1 erläutert. Im Rahmen dieser Arbeit kommt jedoch ausschließlich der HLSH-Ansatz zur Anwendung. Im Zusammenhang mit dem P3RL ist von besonderem Interesse, wie sich die verschiedenen Parameter auf die Parallelisierung auswirken. Beispielsweise führt eine höhere Anzahl von Keys bzw. Iterationen dazu, dass mehr Datensätze im Cluster umverteilt werden müssen.

Phonetisches Blocking Die Grundlagen vom phonetischen Blocking wurden bereits in Unterabschnitt 2.3.2 dargestellt. Im Rahmen dieser Arbeit wird jedoch ein eigener, etwas modifizierter Ansatz verfolgt. Die Grundidee ist wie folgt: Bei der Vorverarbeitung der Daten wird für jeden Datensatz $r_i^j \in D_j$ eine phonetische Codierung ξ_i^j für einen ausgewählten Attributwert erzeugt. Dieser Attributwert entspricht dem Wert des Attributs, auf welchem das Blocking durchgeführt werden soll. Der resultierende phonetische Code ξ_i^j wird dann in einen Bloom-Filter $Bf_{\xi_i^j}$ aufgenommen. Der Bloom-Filter $Bf_{\xi_i^j}$ wird dann gemeinsam mit dem Bloom-Filter Bf_i^j, welcher den gesamten Datensatz r_i^j repräsentiert,

[9] Durch Anwendung eines Blocking-Verfahrens ist es möglich, bessere Werte für die Pairs-Quality zu erreichen. Die Wahrscheinlichkeit hierfür ist allerdings gering, da dazu Record-Paare vom genauen Vergleich ausgeschlossen werden müssten, die eigentlich hinreichend ähnlich sind, um als Match klassifiziert zu werden, aber in der Realität ein Non-Match sind.

an die Linkage-Unit geschickt. Die Linkage-Unit benutzt schließlich den Bloom-Filter $Bf_{\xi_i^j}$ als Blocking-Key, sodass alle Datensätze, bei denen dieser Bloom-Filter exakt die gleichen gesetzten Bitpositionen aufweist, demselben Block zugeordnet werden.

Grundsätzlich entspricht dieses Vorgehen dem herkömmlichen Standard-Blocking, allerdings erfolgt zusätzlich eine Maskierung des phonetischen Codes durch Aufnahme in einen Bloom-Filter. Wie bereits in Unterabschnitt 2.3.2 beschrieben, bieten phonetische Codierungen bereits ein gewisses Maß an Privacy, jedoch ist durch die Aufnahme des Codes in einen Bloom-Filter der genaue Wert des Codes nicht sofort erkennbar. Zwar setzen sich häufige phonetische Codes auch in den Bloom-Filtern durch, aber durch geeignete Wahl der Parameter des Bloom-Filters, können die Häufigkeiten durch Kollisionen verschoben werden. Andererseits kann dies dazu führen, dass mehrere phonetische Codes den gleichen Bloom-Filter erzeugen, was der Skalierbarkeit abträglich ist, da dann mehr unähnliche Datensätze demselben Block zugeordnet werden.

Zur Verbesserung der Skalierbarkeit und der Privacy wird außerdem noch eine weitere Variante dieses Verfahrens untersucht. Hierbei wird ein zweites Attribut gewählt, welches für das Blocking benutzt werden kann. Für dieses Attribut wird dann ebenfalls eine entsprechende phonetische Codierung ζ_i^j für den Datensatz r_i^j erzeugt. Wie beim obigen Vorgehen wird der berechnete phonetische Code ξ_i^j in einen Bloom-Filter $Bf_{\xi_i^j}$ aufgenommen. Hierbei erhält jedoch jede Hashfunktion, welche zur Erzeugung des Bloom-Filters $Bf_{\xi_i^j}$ genutzt wird, zusätzlich den Code ζ_i^j als Eingabe. Jede Hashfunktion erhält dadurch den phonetischen Code ξ_i^j konkateniert mit dem phonetischen Code ζ_i^j als Eingabe. Damit dient der phonetische Code ζ_i^j als *Salt* für die einzelnen Hashfunktionen. Die Idee, einen Salt für die Hashfunktionen des Bloom-Filters zu verwenden, stammt ursprünglich aus [NSKS14] und wird dort zur Härtung von Bloom-Filtern als erweiterter Schutz gegenüber möglichen Angriffen vorgeschlagen. Die Nutzung von ζ_i^j führt dazu, dass die einzelnen Hashfunktionen nur dann dieselben Werte liefern, wenn beide Codes übereinstimmen. Dieses Vorgehen hat zum Vorteil, dass nun auch die Häufigkeitsverteilung der gesetzten Bits von beiden anstatt von einem Attribut abhängig ist. Prinzipiell wird dadurch erschwert, dass aus der Häufigkeitsverteilung der Bits Rückschlüsse auf die tatsächlichen Attributwerte möglich sind. Weiterhin werden Datensätze nur dann demselben Block zugeordnet, wenn wiederum beide Codes übereinstimmen. Dies führt zu kleineren Blöcken mit sehr ähnlichen Datensätzen. Der Nachteil dieses Ansatzes ist, dass die Wahrscheinlichkeit von False-non-Matches erhöht wird. Enthalten die für das Blocking gewählten Attribute Fehler, die nicht von den phonetischen Codierungen kompensiert werden können, so werden zwei eigentlich übereinstimmende Datensätze nicht mehr demselben Block zugeordnet und damit vom Vergleich ausgeschlossen. Dieser Umstand trifft zwar auch für den obigen Ansatz zu, jedoch ist dort der Blocking-Key nur von einem Attribut abhängig. Generell sollten

bei beiden Ansätzen möglichst fehlerunanfällige und unveränderliche Attribute gewählt werden.

Im weiteren Verlauf der Arbeit werden die beiden beschriebenen Verfahren als *Phonetic-Blocking (PB)* und *Salted-phonetic-Blocking (SPB)* bezeichnet. Zur Illustration der beiden Ansätze ist das Vorgehen zur Bildung des Blocking-Keys entsprechend des jeweiligen Ansatzes in Abbildung 3.1 dargestellt. Hierbei wurde erneut auf den Datensatz A2 aus dem laufenden Beispiel von Unterabschnitt 2.1.1 zurückgegriffen. Zur Erzeugung der phonetischen Codierung kommt außerdem Soundex zum Einsatz. Beim PB wird der Bloom-Filter, welcher als Blocking-Key genutzt werden soll, aus dem phonetisch codierten Nachnamen ('Meier') erzeugt. Dies entspricht einem Blocking auf dem Attribut Nachname. Jede Hashfunktion des Bloom-Filters erhält dabei jedoch den codierten Wert ('M460') als Eingabe. Beim SPB wird ebenfalls die phonetische Codierung des Nachnamens erzeugt. Zusätzlich wird hierbei jedoch außerdem der Vorname ('Bernd') phonetisch codiert. Die beiden phonetischen Codes 'M460' und 'B653' werden konkateniert und dienen zusammen als Eingabe für die Hashfunktionen des Blocking-Bloom-Filters. Dies entspricht dabei einem gemeinsamen Blocking auf den Nachnamen und den Vornamen.

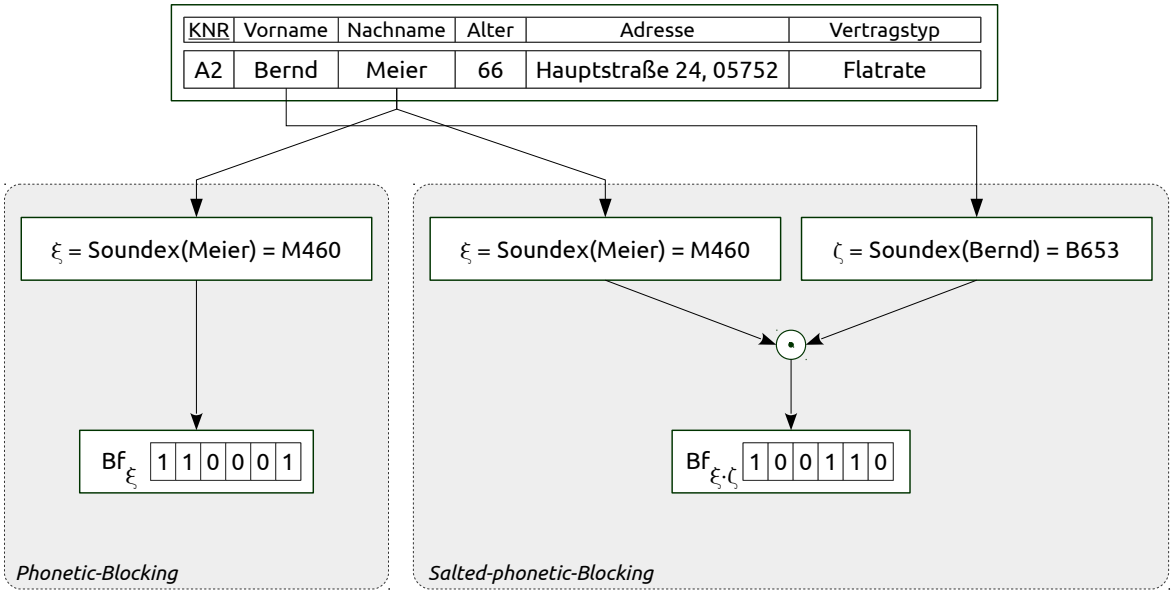


Abbildung 3.1: Beispiel: Bildung des Blocking-Keys beim Phonetic-Blocking bzw. Salted-phonetic-Blocking.

3.3 Architektur

Aus dem im Abschnitt 3.1 beschriebenen Grundkonzept wurde eine Architektur für das im Rahmen dieser Arbeit entwickelte Framework zur Durchführung des P3RL mit Flink entwickelt. Die entsprechende Architektur ist in Abbildung 3.2 dargestellt. Die einzelnen Komponenten decken sich dabei grundsätzlich mit den zugehörigen Schritten des PPRL-Prozesses. Die nachfolgenden Abschnitte beschreiben die einzelnen Komponenten im Detail und erläutern zudem, wie die Verfahren und Techniken konkret umgesetzt wurden und wie deren Ausführung in Flink erfolgt.

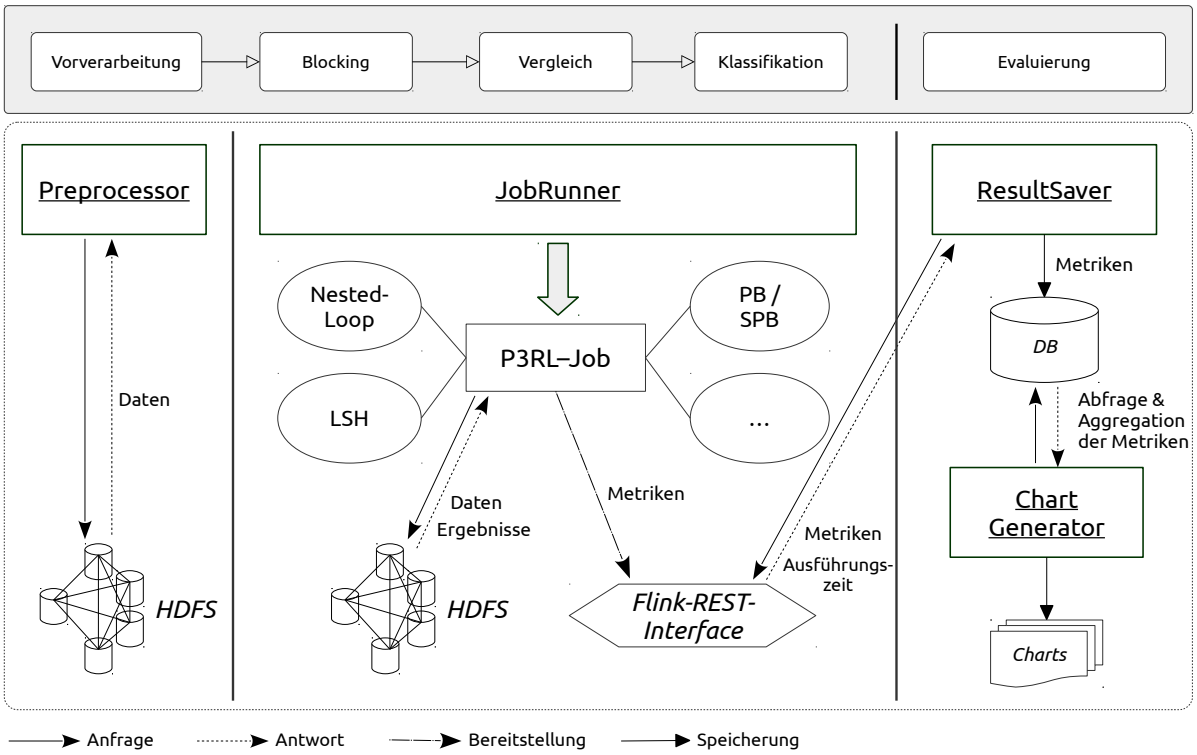


Abbildung 3.2: Architektur des entwickelten Frameworks für das P3RL mit Flink.

3.4 Vorverarbeitung

Der Schritt der Vorverarbeitung wird von dem `Preprocessor` ausgeführt und beginnt damit, dass die Daten eingelesen werden. Öffentlich verfügbare Testdaten liegen oftmals im CSV-Dateiformat vor (siehe Abschnitt 4.1). Hierbei liegen die Daten in einfachen Textdateien vor und sind nur durch bestimmte Trennzeichen voneinander abgegrenzt. Die Trennzeichen dienen dabei zur Unterscheidung zwischen den einzelnen Datensätzen und den Datenfeldern (Attributwerten). Flink bietet eine Klasse zum Lesen solcher Daten

an, die auch bereits verteilt im HDFS gespeichert sein können. Diese wird genutzt, um die Daten aus der CSV-Datei in ein DataSet zu laden (vgl. Unterabschnitt 2.4.4). Ein DataSet besteht hierbei aus Objekten der Klasse `Person`. Die Klasse Person erfüllt die Eigenschaften eines POJOs und deckt typische Attribute von Personen, wie zum Beispiel Vorname, Nachname, Alter, Geschlecht und Adresse, ab. Die Nutzung einer solchen Klasse zur Repräsentation von Personen hat den Vorteil, dass nur bestimmte Attribute aus der CSV-Datei gelesen werden können, welche dann den entsprechenden Feldern der Klasse zugeordnet werden. Dies erhöht die Flexibilität, welche notwendig ist, da beim PPRL je nach Einsatzgebiet und teilnehmenden Parteien unterschiedliche Attribute für das Linkage relevant sein können. Da zur Bildung der Bloom-Filter später die Q-Gramme der eingelesenen Attribute bestimmt werden müssen, werden alle Attribute als Datentyp `String` behandelt. Prinzipiell ist das Einlesen der Daten der erste Schritt im Flink-Job und stellt damit einen Source-Operator dar.

Nach dem Einlesen der Daten wird im Rahmen der Vorverarbeitung zunächst ein Data-Cleaning durchgeführt. Da das Data-Cleaning sehr stark von den am PPRL teilnehmenden Parteien und deren Absprachen untereinander abhängt, wurden bei diesem Schritt nur wenige Operationen implementiert. Hierzu gehört die Konvertierung aller Großbuchstaben in Kleinbuchstaben sowie die Entfernung von Weißraum (Leerzeichen, Tabulatoren und Zeilenumbrüchen) innerhalb der einzelnen Attribute.

Schließlich beinhaltet die Vorverarbeitung noch die Maskierung der Daten. Als Privacy-Technik kommen in dieser Arbeit ausschließlich Bloom-Filter zum Einsatz. Zu diesem Zweck wurde eine eigene Bloom-Filter-Implementierung erstellt. Zwar gibt es bereits einige öffentlich verfügbare Implementierungen, jedoch sind diese meist entweder für Anwendungsgebiete außerhalb des PPRL entworfen oder sie sind nicht kompatibel mit Flink, da sie die von Flink geforderten Eigenschaften für Datentypen nicht erfüllen (siehe Unterabschnitt 2.4.4). Aus diesem Grund wurde auf die Klasse `java.util.BitSet` zurückgegriffen. Ein solches BitSet dient dabei zur Repräsentation des Bit-Arrays, welches dem Bloom-Filter zugrunde liegt. Darüber hinaus wurden die in Abschnitt 2.2 vorgestellten Bloom-Filter-Operationen mit Hilfe dieser Klasse realisiert. Bezüglich der Hashfunktionen wurde das *Double-Hashing-Scheme* aus [SBR11] umgesetzt. Hierbei werden die k Hashfunktionen $H_1, \ldots, H_k$ des Bloom-Filters basierend auf zwei Hashfunktionen G_1 und G_2 wie folgt erzeugt [SBR11]:

$$H_i(x) = (G_1(x) + (i-1) \cdot G_2(x)) \mod m. \tag{3.1}$$

Durch dieses Schema ist es möglich, beliebig viele Hashfunktionen durch zwei Ausgangsfunktionen zu erzeugen. Zwar sollte dieses Vorgehen in der Praxis aus Sicherheitsgründen

nicht genutzt werden [NSKS14], jedoch ist es zur Evaluierung von Verfahren komfortabler in der Nutzung. Als Ausgangsfunktionen wurden MD5 und SHA [KCB97] gewählt. Generell ist es natürlich möglich, die beiden Bloom-Filter-Parameter k und m je nach Bedarf festzulegen.

Zur Repräsentation eines Records durch einen Bloom-Filter muss zunächst die Menge aller Q-Gramme $\bar{E}$ erzeugt werden. Hierfür kann die Länge Q der Q-Gramme flexibel bestimmt werden. Weiterhin ist es möglich, zusätzliche Füllzeichen bei der Erstellung der Q-Gramme zu berücksichtigen (vgl. Unterabschnitt 2.2.3).

Die beiden Verfahren PB und SPB, welche zum phonetischen Blocking eingesetzt werden, benötigen bei der Vorverarbeitung noch einen zusätzlichen Schritt. In diesem wird für jeden Datensatz der Bloom-Filter erstellt, welcher zum Blocking genutzt werden soll. Für die phonetische Codierung stehen mehrere Verfahren zur Verfügung. Umgesetzt wurden Soundex, Metaphone sowie Double-Metaphone (siehe Unterabschnitt 2.3.2). Außerdem muss diesbezüglich ausgewählt werden, welches Attribut bzw. welche Attribute als Blocking-Key genutzt werden sollen. Beim SPB ist es dabei möglich, auf die gewählten Attribute unterschiedliche phonetische Codierungen anzuwenden. Für den Bloom-Filter, der dann zur Repräsentation des phonetischen Codes eingesetzt wird, sind ebenfalls die Länge und die Anzahl der Hashfunktionen frei wählbar.

Die Umsetzung der Vorverarbeitung mit Flink ist ausschließlich durch Map-Funktionen realisiert, da keine Gruppierung oder Umverteilung der Daten notwendig ist. Nach dem Einlesen der Daten, wird zunächst in einer Map-Phase das Data-Cleaning gemeinsam mit der Erzeugung der einzelnen Q-Gramme durchgeführt. In einer zweiten Map-Phase werden diese dann in den Bloom-Filter aufgenommen. Schließlich wird in einer optionalen dritten Map-Phase der Blocking-Key für die Verfahren zum phonetischen Blocking erstellt. Das Flink-Programm zur Durchführung der Vorverarbeitung endet damit, dass die nun codierten Datensätze wiederum im CSV-Format gespeichert werden. Hierbei können die Daten beispielsweise ins HDFS geschrieben werden.

3.5 Linkage

Die Durchführung des Linkages mit den Schritten Blocking, Vergleich und Klassifikation ist abhängig von dem gewählten Verfahren für das P3RL. Um das entwickelte Framework möglichst flexibel und erweiterbar zu gestalten, wird ein `JobRunner` eingesetzt, welcher zur Ausführung eines `P3RL-Jobs` dient. Ein solcher P3RL-Job ist dabei als abstrakte Klasse realisiert, welche von konkreten Verfahren zur Durchführung des Linkages abstrahiert. Weiterhin enthält ein P3RL-Job bereits bestimmte Attribute und Methoden,

welche unabhängig vom konkreten Verfahren notwendig sind. Um ein konkretes Verfahren umzusetzen, muss diese Klasse implementiert werden. Dazu müssen die notwendigen Operationen mit Hilfe der Flink-Operatoren realisiert werden.

Um das Linkage durchführen zu können, müssen zunächst die am PPRL teilnehmenden Parteien ihre codierten Daten an die Linkage-Unit senden. Nach Erhalt der Daten speichert die Linkage-Unit die Daten verteilt und möglicherweise repliziert im HDFS ab. Das eigentliche Linkage beginnt daraufhin damit, dass die Daten aus dem HDFS parallel von den Rechnern im Flink-Cluster gelesen werden. Hierbei werden die codierten und im CSV-Format gespeicherten Datensätze zunächst in Java-Objekte überführt (Deserialisierung). Nach dem Lesen der Daten wird in einer Map-Phase jedem Eingabe-Datensatz ein Identifikator zugewiesen. Dieser gibt die Zuordnung des Datensatzes zu der Partei an, von welcher der Datensatz stammt. Prinzipiell könnte dieser Identifikator auch schon bei der Vorverarbeitung durch die einzelnen Parteien hinzugefügt werden. Einerseits ist diese Zuordnung wichtig, um später die übereinstimmenden Datensätze zu identifizieren. Andererseits werden so Vergleiche zwischen Datensätzen derselben Partei verhindert.

Da die Klasse `BitSet`, auf welcher die Bloom-Filter-Implementierung basiert, nicht die Eigenschaften eines POJOs erfüllt, können die codierten Daten nicht automatisch in Bloom-Filter-Objekte umgewandelt werden. Aus diesem Grund ist nach dem Lesen der Daten eine weitere Map-Phase notwendig, in der die gelesenen Daten vom Typ `String` umgewandelt werden. Das Resultat der Lese-Phase sind schließlich Tupel der Form '(DataSetId, Id, Bloom-Filter)', welche als `LinkageTuple` bezeichnet werden. Dies entspricht einem Bloom-Filter, welcher den Datensatz mit dem Primärschlüssel 'Id' repräsentiert und aus der Datenquelle mit dem Identifikator 'DataSetId' stammt. Beispielsweise entspricht das Tupel '(A, A2, [1101101101000010])' dem Bloom-Filter $Bf^A_{A2} = [1101101101000010]$, der den Datensatz $r^A_{A2} \in D_A$ repräsentiert.

Das weitere Vorgehen ist abhängig von dem gewählten Verfahren zur Durchführung des Blockings. In den nachfolgenden Unterabschnitten werden die einzelnen Verfahren und ihre Realisierung mit Flink näher betrachtet.

3.5.1 Nested-Loop

Das Nested-Loop-Verfahren kann in Flink sehr einfach mit Hilfe einer Cross-Funktion realisiert werden. Die Cross-Funktion bildet dabei jedes Datensatzpaar der beiden Eingabe-Datenmengen, was dem kartesischen Produkt zwischen den beiden Eingabe-Datenmengen entspricht.

Für jedes von der Cross-Funktion erzeugte Kandidaten-Record-Paar müssen schließlich

die Ähnlichkeitsberechnung und die Klassifikation durchgeführt werden. Dies wird durch eine FlatMap-Operation realisiert. Die FlatMap-Operation berechnet dazu zunächst die Ähnlichkeit der beiden Datensätze. Hierfür wurde sowohl die Dice- als auch die Jaccard-Similarity (siehe Unterabschnitt 2.2.1) umgesetzt. Als Ausgabe erzeugt die FlatMap-Operation nur die Kandidaten-Record-Paare, deren Ähnlichkeit über dem Schwellwert t liegt. Die resultierenden Datensatzpaare werden dann als Ergebnis im HDFS abgespeichert.

3.5.2 LSH

Beim LSH erfolgt zunächst innerhalb einer FlatMap-Funktion die Berechnung der LSH- bzw. Blocking-Keys $BK_1, \ldots, BK_\Lambda$ für jeden Record. Für jeden LSH-Key werden die Hashfunktionen $f_{\lambda_1}, \ldots, f_{\lambda_\Psi}$ zur Bildung des Keys zufällig aus $\mathcal{F}_{Hamming}$ gewählt und berechnet (siehe Unterabschnitt 2.3.1). Hierbei wird jedoch eine Funktion $f_i \in \mathcal{F}_{Hamming}$ nur höchstens einmal pro Key benutzt, sodass letztlich jeder LSH-Key aus verschiedenen Bitpositionen gebildet wird. Dieses Vorgehen wird Λ-mal wiederholt, sodass am Ende für einen Datensatz Λ LSH-Keys erzeugt wurden. Die einzelnen LSH-Keys werden dabei nummeriert, sodass eine eindeutige Zuordnung zu den Hashfunktionen, welche den entsprechenden Key erzeugt haben, gewährleistet wird. Die FlatMap-Funktion erzeugt für jeden berechneten LSH-Key des Datensatzes als Ausgabe ein Tupel der Form '(KeyID, Key, LinkageTuple, KeyList)'. Damit wird jeder Datensatz Λ-mal repliziert. Das Problem beim LSH besteht darin, dass durch Erzeugung mehrerer LSH-Keys auch Kandidaten-Record-Paare mehrfach gebildet werden können. Diese führen letztendlich zu überflüssigen Vergleichen und sollten möglichst vermieden werden. Aus diesem Grund enthält das Ausgabe-Tupel der FlatMap-Funktion zusätzlich eine Liste ('KeyList') aller erzeugten LSH-Keys. Die Liste der LSH-Keys kann später dazu genutzt werden, um festzustellen, ob ein Kandidaten-Record-Paar bereits in einem anderen Block gebildet wurde.

Das Vorgehen zur Bildung der LSH-Keys ist in Abbildung 3.3 veranschaulicht. Hierbei wurden die Datensätze aus dem laufenden Beispiel von Unterabschnitt 2.1.1 beispielhaft in Bloom-Filter der Länge 8 überführt. Weiterhin sei die Parallelität 2, wobei initial je ein Rechner die Datensätze von Partei A und ein Rechner die Datensätze der Partei B liest. Für LSH wurden die Parameter $\Lambda = 2$ und $\Psi = 3$ gewählt. Der LSH-Key BK_1 bzw. BK_2 berechnet sich dabei für einen Record wie folgt: $BK_1(Bf_i^j) = f_3(Bf_i^j) \odot f_1(Bf_i^j) \odot f_7(Bf_i^j)$ und $BK_2(Bf_i^j) = f_4(Bf_i^j) \odot f_2(Bf_i^j) \odot f_6(Bf_i^j)$, wobei $f_i \in \mathcal{F}_{Hamming}$. In der Abbildung ist dies durch die Notation $BK_1 = <[3],[1],[7]>$ bzw. $BK_2 = <[4],[2],[6]>$ dargestellt, was bedeutet, dass der LSH-Key BK_1 (bzw. BK_2) gebildet wird, indem die Bitwerte an Position 3, 1 und 7 (bzw. 4, 2 und 6) des entsprechenden Bloom-Filters

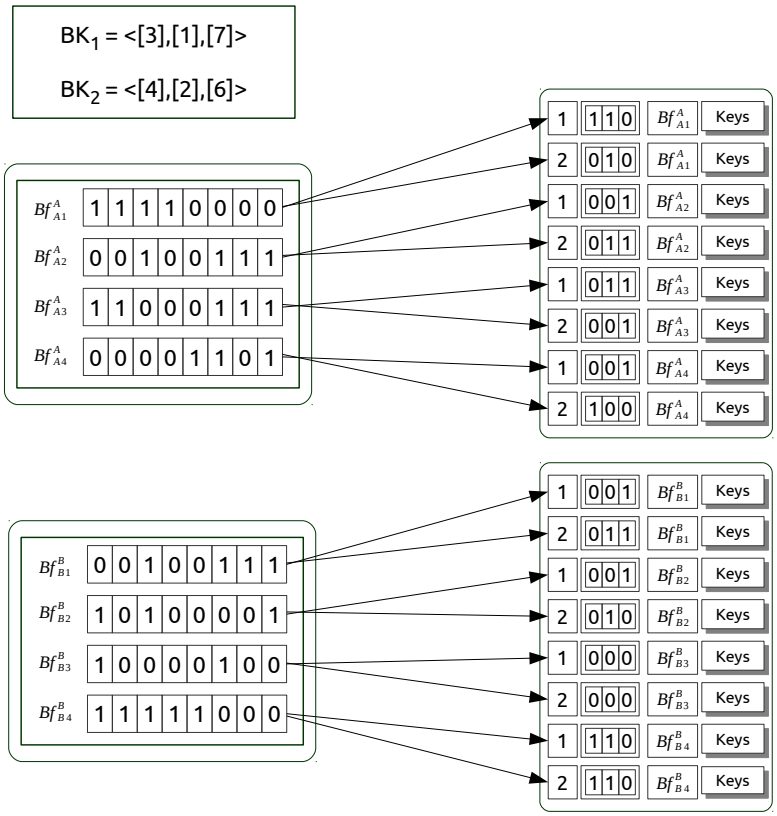


Abbildung 3.3: Beispiel: FlatMap-Operation zur Bildung der LSH-Keys.

konkateniert werden. Beispielsweise werden für den Bloom-Filter $Bf^A_{A2} = [00100111]$ die Keys $BK_1(Bf^A_{A2}) = [001]$ und $BK_2(Bf^A_{A2}) = [011]$ erzeugt.

Nach der Erzeugung der LSH-Keys erfolgt die Gruppierung der Datensätze durch eine GroupBy-Funktion, welche auf die beiden Felder 'KeyId' und 'Key' angewandt wird. Die beiden Felder entsprechen dabei dem Wert des LSH-Keys mit dessen Zuordnung (ID). Es ist wichtig, dass die Gruppierung auf beiden Feldern erfolgt, da auch LSH-Keys, die von verschiedenen Hashfunktionen erzeugt wurden, denselben Wert liefern können. Die Gruppierung der Daten führt zu einer Umverteilung der Datensätze, sodass alle Datensätze, die denselben Wert für einen LSH-Key $BK_1, \ldots, BK_\Lambda$ aufweisen, demselben Block und damit Rechner zugeordnet werden. Durch eine GroupReduce-Funktion werden innerhalb eines Blocks alle Paare von Datensätzen erzeugt. Bei der Erzeugung dieser Kandidaten-Record-Paare ist darauf zu achten, dass nicht zwei Records aus derselben Datenquelle miteinander verglichen werden.

Abbildung 3.4 zeigt für das obige Beispiel, wie die Gruppierung der Datensätze und ihre anschließende Aggregation erfolgt. Wie in der Abbildung ersichtlich, werden beispielsweise die Bloom-Filter Bf^A_{A2}, Bf^A_{A4}, Bf^B_{B1} und Bf^B_{B2} demselben Block zugeordnet, da sie den gleichen Wert für BK_1 erzeugen. Demnach erzeugt dieser Block insgesamt vier Kandidaten-Record-Paare.

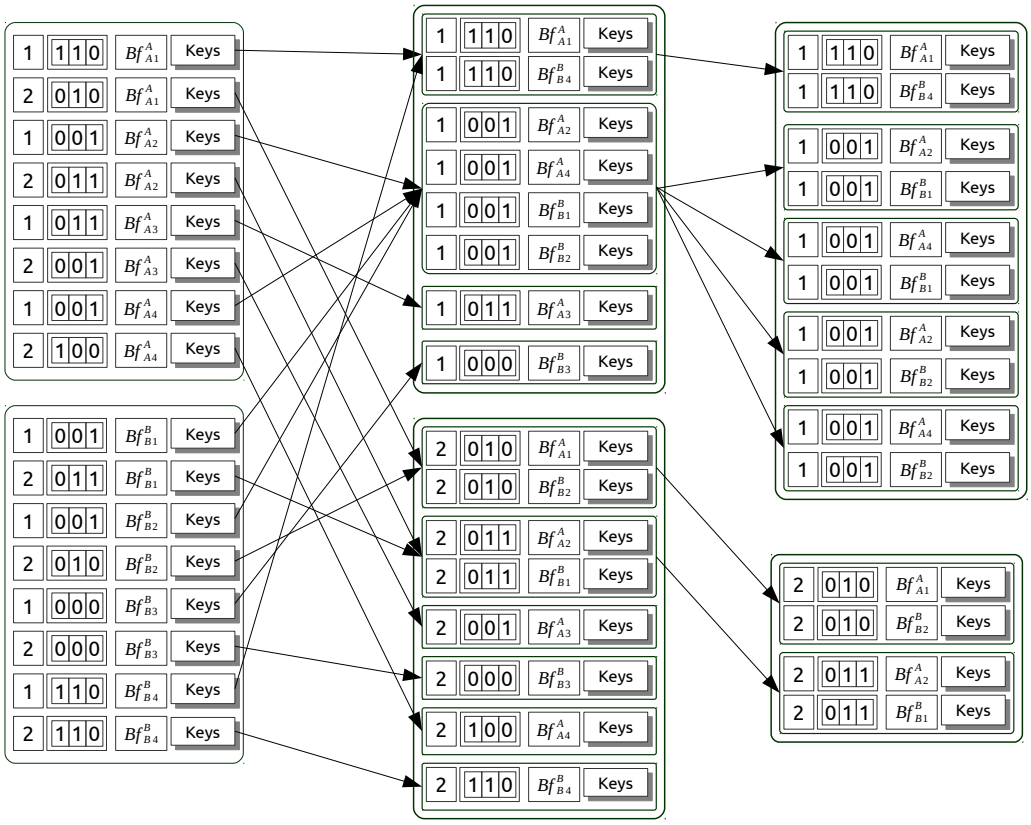


Abbildung 3.4: Beispiel: Gruppierung der Datensätze anhand der LSH-Keys mit anschließender Bildung der Kandidaten-Record-Paare.

Nachdem alle Kandidaten-Record-Paare erzeugt wurden, wird eine Filter-Funktion auf jeden Datensatz angewandt. Diese Filter-Funktion dient dazu, doppelte Kandidaten-Record-Paare zu entfernen und damit doppelte Vergleiche zu vermeiden. Hierzu wird die Liste aller erzeugten LSH-Keys betrachtet, welche den Tupeln beigefügt ist. Für die beiden Datensätze des Kandidaten-Paars wird diese Liste durchlaufen und es wird überprüft, ob die beiden Datensätze zusätzlich in einem anderen LSH-Key übereinstimmen und damit bereits für den Vergleich vorgemerkt sind. Seien Bf_x und Bf_y die beiden Bloom-Filter eines Kandidaten-Record-Paars und BK_i der (betrachtete) Blocking-Key, welcher zur Erzeugung des Kandidaten-Paars geführt hat. Falls ein Blocking-Key BK_j mit $j < i$ gefunden werden kann, für den gilt, dass $BK_j(Bf_x) = BK_j(Bf_y)$, so wird das Kandidaten-Paar bereits in einem anderen Block verglichen und muss daher nicht weiter berücksichtigt werden. Abbildung 3.5 zeigt die entsprechende Filter-Operation für das obige Beispiel. Hierbei wird das doppelte Kandidaten-Record-Paar (Bf^A_{A2}, Bf^B_{B1}) entfernt. Alle anderen Kandidaten-Record-Paare werden nur einmal betrachtet und müssen daher nicht entfernt werden.

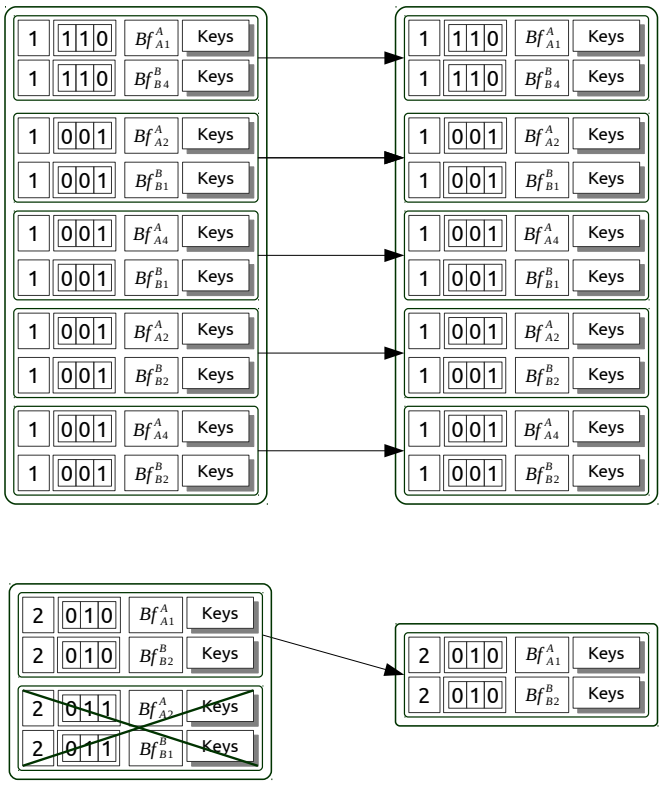


Abbildung 3.5: Beispiel: Filter-Operation zur Entfernung von doppelten Kandidaten-Record-Paaren.

Der vorgestellte Ansatz zur Entfernung von doppelten Kandidaten-Record-Paaren führt zu zusätzlichem Berechnungsaufwand. So müssen maximal $O(\Lambda - 1)$ LSH-Keys miteinander verglichen werden. Da jeder LSH-Key aus Ψ Bits zusammengesetzt ist, sind demzufolge

insgesamt höchstens $O((\Lambda - 1) \cdot \Psi)$ Bit-Vergleiche notwendig. Vor allem für große Werte für Ψ und Λ kann es daher sein, dass der Test auf doppelte Kandidaten-Record-Paare aufwändiger ist, als den Vergleich eines Kandidaten-Record-Paars doppelt auszuführen.

Für alle übrigen Kandidaten-Record-Paare, welche in der Filter-Phase nicht ausgeschlossen wurden, erfolgt schließlich die Ähnlichkeitsberechnung mit anschließender Speicherung der Ergebnisse. Die Ähnlichkeitsberechnung erfolgt dabei, wie beim Nested-Loop, unter Verwendung einer FlatMap-Operation.

3.5.3 Phonetisches Blocking

Die beiden Verfahren zum phonetischen Blocking PB und SPB unterscheiden sich nicht in der Art ihrer Ausführung. Ihr Unterschied besteht darin, dass der als Blocking-Key verwendete Bloom-Filter unterschiedlich erzeugt wird (vgl. Abschnitt 3.2). Da das phonetische Blocking ebenso wie das Standard-Blocking abläuft, ist die Umsetzung mit Flink sehr einfach. Zunächst erfolgt die Gruppierung (groupBy) der Datensätze anhand des phonetischen Codes, welcher als Bloom-Filter vorliegt. Dies führt dazu, dass alle Datensätze, bei denen der Blocking-Bloom-Filter dieselben gesetzten Bits aufweist, demselben Block zugeordnet werden. Die Datensätze eines Blocks befinden sich dann auf einem Rechner des Computer-Clusters. Innerhalb eines Blocks werden wieder alle Paare von Datensätzen, die Kandidaten-Record-Paare, erzeugt und es erfolgt erneut die Ähnlichkeitsberechnung und Klassifikation durch die FlatMap-Operation.

Das Vorgehen beim phonetischen Blocking ist in Abbildung 3.6 zusammengefasst. Hierbei wurden erneut die Datensätze aus dem laufenden Beispiel von Unterabschnitt 2.1.1 zur Veranschaulichung genutzt. In der ersten Phase erfolgt die Gruppierung anhand des Bloom-Filters, welcher den phonetischen Code repräsentiert. Dieser hat im Beispiel die Länge 4. Insgesamt werden beim Blocking sechs Blöcke erzeugt, wobei vier Blöcke nur jeweils einen Datensatz enthalten und damit keine Kandidaten-Paare erzeugen. Die übrigen zwei Blöcke enthalten genau zwei Datensätze, weswegen insgesamt wiederum zwei Kandidaten-Record-Paare erzeugt werden.

3.6 Evaluierung

Das entwickelte Framework zur Durchführung des P3RL soll eine einfache und weitgehend automatische Evaluation der umgesetzten Verfahren ermöglichen. Im Allgemeinen soll damit der Vergleich von verschiedenen Verfahren oder eines Verfahrens mit verschiedenen Parametern erleichtert werden. Zur Umsetzung dieses Ziels ist es notwendig, dass ein

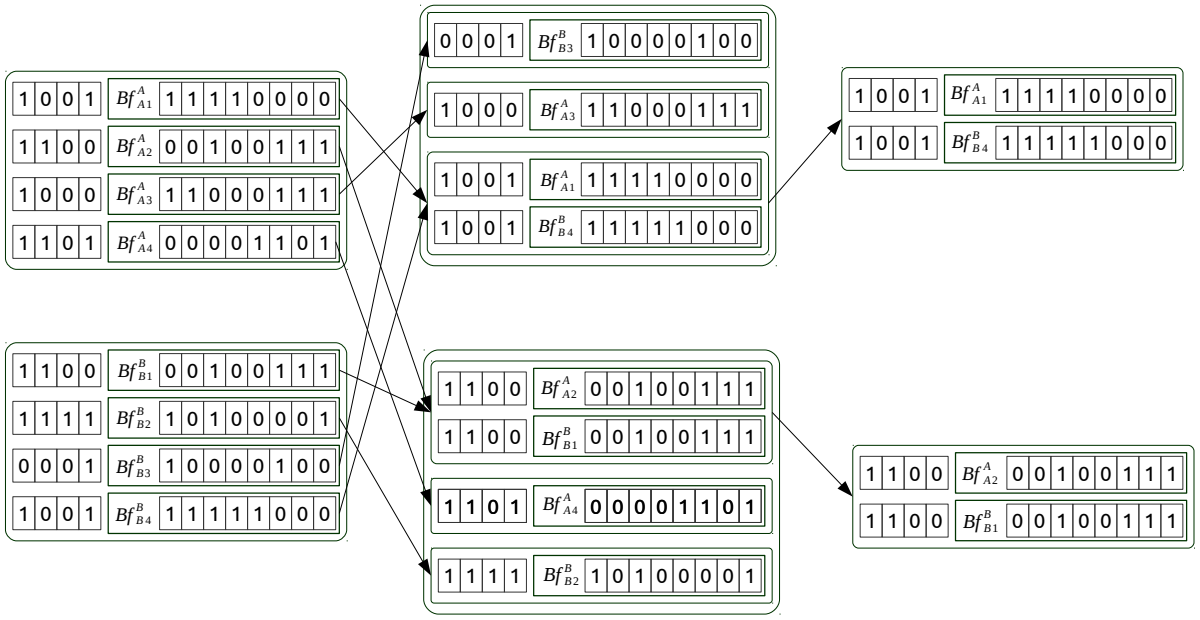


Abbildung 3.6: Beispiel: Phonetisches Blocking mit Flink.

P3RL-Job verschiedene Metriken berechnet und bereitstellt. Wie in Unterabschnitt 2.4.5 beschrieben, bietet Flink unterschiedliche Möglichkeiten an, um die Ausführung eines Jobs zu überwachen. Besonders von Interesse ist hierbei die REST-API von Flink, welche eine Abfrage der berechneten Metriken erlaubt.

In Unterabschnitt 2.1.3 wurde beschrieben, welche Metriken zur Evaluierung des PPRL relevant sind. Zur Bestimmung der Reduction-Rate ist es notwendig, die Anzahl der Eingabe-Datensätze sowie die daraus generierten Kandidaten-Record-Paare zu erfassen. Zu diesem Zweck bieten sich zwei Akkumulatoren an, die zum Zählen der Datensätze bzw. der Kandidaten-Record-Paare benutzt werden können. Für die Qualitätsmetriken (Pairs-Completeness, Pairs-Quality und F-Measure) muss für die berechneten Matches, also die als übereinstimmend klassifizierten Kandidaten-Record-Paare, feststellbar sein, ob diese auch in der Realität übereinstimmen. Dieser True-Match-Status kann in realen Anwendungen normalerweise nicht bestimmt werden. Bei der Evaluierung von Verfahren für das PPRL werden jedoch meist Testdaten benutzt. Diese Testdaten sind dabei so aufgebaut, dass sie über eine eindeutige Kennung (ID) verfügen, welche für übereinstimmende Datensätze denselben Wert hat. Referenzieren die Datensätze hingegen verschiedene Realwelt-Objekte, so ist auch die Kennung der beiden Datensätze verschieden voneinander. Um festzustellen, wie viele Kandidaten-Paare korrekt klassifiziert wurden, können die umgesetzten Verfahren durch eine zusätzliche Filter-Funktion erweitert werden. Innerhalb der Filter-Funktion wird für jedes Kandidaten-Record-Paar bestimmt, ob die Kennung der beiden Datensätze übereinstimmt. Falls die Kennung übereinstimmt, so kann ein Zähler (Akkumulator) zur Erfassung der True-Matches erhöht werden. Ist die Kennung

der beiden Datensätze stattdessen verschieden, so wird der entsprechende Zähler für die False-Matches erhöht. Zur Berechnung der Qualitätsmetriken ist es außerdem notwendig, die Anzahl der False-non-Matches zu bestimmen. Die Menge der False-non-Matches kann nur sehr umständlich mit dem zuvor verwendeten Ansatz bestimmt werden, da hierzu alle möglichen Kandidaten-Record-Paare, welche bisher nicht berücksichtigt wurden, betrachtet werden müssten. Aus diesem Grund wird einem Flink-Job die Anzahl der insgesamt vorhanden Matches direkt als Parameter übergeben. Die Anzahl der False-non-Matches kann dann als Differenz zwischen der Anzahl aller Matches und der Anzahl der gefundenen True-Matches berechnet werden.

Die berechneten Kennzahlen werden von dem P3RL-Job gesammelt und sind nach seiner Beendigung über das Flink-REST-Interface abrufbar. Die Ausführungszeit des Jobs sowie dessen Start- und Endzeitpunkt werden standardmäßig von Flink registriert und müssen daher nicht explizit gemessen werden. Weitere Informationen, wie beispielsweise der Name oder die Parameter des Jobs, können zur Ausführungskonfiguration des Jobs hinzugefügt werden und sind damit ebenfalls über das REST-Interface verfügbar.

Zusammenfassend erfolgt die Erfassung der verschiedenen Kennzahlen bzw. Metriken während der Ausführung des P3RL-Jobs. Dies hat den Nachteil, dass zusätzlicher Berechnungsaufwand erforderlich ist, welcher die Ausführungszeit eines Jobs verlängert. Vor allem die Filter-Funktion zur Bestimmung der True-Matches und False-Matches verursacht in diesem Zusammenhang den größten Mehraufwand, da für jedes Kanidaten-Record-Paar ein zusätzlicher `String`-Vergleich durchgeführt werden muss. Aus diesem Grund ist letztlich die tatsächlich erreichbare Ausführungszeit geringer. Jedoch steht in dieser Arbeit der Vergleich von verschiedenen Verfahren mit unterschiedlichen Parametern im Vordergrund, weshalb diese Unterschiede vernachlässigbar sind. Außerdem sind bei allen umgesetzten Verfahren die gleichen Schritte zur Bestimmung der Metriken notwendig, womit alle Verfahren durch einen ähnlichen Mehraufwand belastet werden.

Wie zuvor beschrieben, werden die relevanten Kennzahlen und Metriken durch den P3RL-Job bestimmt und sind anschließend über das REST-Interface des Job-Managers abrufbar. Das weitere Ziel ist es, die berechneten Metriken auszuwerten, die Ergebnisse graphisch aufzubereiten und sie in Form von Diagrammen (Charts) darzustellen. Dies hat den Vorteil, dass die verschiedenen Verfahren direkt miteinander verglichen und analysiert werden können, ohne mühsam manuell verschiedene Resultate zusammentragen zu müssen. Problematisch hierbei ist jedoch einerseits, dass Jobs teilweise mehrfach ausgeführt werden, vor allem um Abweichungen in der Laufzeit zu messen. Beim LSH hängt die Qualität der Ergebnisse außerdem von der Wahl der Hashfunktionen bzw. Keys ab. Werden diese zufällig bestimmt, so ist es möglich, dass unterschiedliche Ergebnisse berechnet werden. Solche mehrfach ausgeführten Jobs sollten daher zusammengefasst bzw. aggregiert wer-

den, beispielsweise indem die Standardabweichung und das arithmetische Mittel für die relevanten Metriken berechnet wird. Andererseits sind die Resultate, welche über das REST-Interface abrufbar sind, nicht persistent gespeichert. Falls beispielsweise Flink neu gestartet wird, so sind nach dem Neustart die Ergebnisse nicht mehr verfügbar. Aus diesen Gründen wurde die Architektur um eine Datenbank erweitert, die zur Speicherung und Verwaltung der Ergebnisse dient. Der Vorteil hierbei ist, dass Anfragen auf die Testdaten ermöglicht werden, wodurch umfangreiche Analysemöglichkeiten entstehen. Um die Metriken in der Datenbank abzuspeichern, wurde daher eine weitere Komponente, der `ResultSaver`, umgesetzt. Dieser dient dazu, die Metriken und ausgewählte Informationen über den Job, von der REST-Schnittstelle abzufragen und danach zu persistieren. Hierzu werden die Daten, welche im JSON-Format vorliegen, zunächst in Java-Objekte überführt. Aus den gesammelten Kennzahlen werden dann die Metriken Reduction-Rate (RR), Pairs-Completeness (PC), Pairs-Quality (PQ) sowie F-Measure (FM) berechnet (siehe Unterabschnitt 2.1.3). Mit Hilfe von Hibernate[10] werden die Java-Objekte in der Datenbank persistiert. Die Verwendung von Hibernate hat den Vorteil, dass das verwendete Datenbank-Management-System leichter ausgetauscht werden kann. Standardmäßig wird als Datenbank-Management-System PostgreSQL[11] genutzt.

Schließlich können die Daten aus der Datenbank abgefragt werden und zur Erzeugung für verschiedene Diagramme genutzt werden. Diese Aufgabe übernimmt der `ChartGenerator` (siehe Abbildung 3.2). Die Erzeugung der Diagramme erfolgt dabei durch Verwendung von JFreeChart[12]. JFreeChart ist eine quelloffene Java-Bibliothek zur Erzeugung von Diagrammen verschiedenen Typs.

[10] `http://hibernate.org/orm/`, Zugriff: 04.04.2017
[11] `https://www.postgresql.org/`, Zugriff: 04.04.2017
[12] `http://www.jfree.org/jfreechart/`, Zugriff: 04.04.2017

Kapitel 4

Evaluierung

4.1 Datengrundlage und verwendete Parameter

Zur Evaluation der implementierten P3RL-Verfahren sollen verschiedene Datenmengen genutzt werden. Aufgrund von Sicherheits- und Datenschutz-Bedenken existieren nur wenige reale Datenquellen, die Datensätze mit personenbezogenen Informationen enthalten und damit zur Evaluierung des PPRL eingesetzt werden können [VCV13]. Eine der wenigen frei zugänglichen Datenquellen mit personenbezogenen Daten ist die Datenbank der North-Carolina-Voter-Registration (NCVR)[13]. Die NCVR-Datenbank enthält Datensätze zu wahlberechtigten Personen aus dem Bundesstaat North Carolina (USA). Hierbei werden verschiedene Daten zu den Personen gespeichert, unter anderem der Name, die Adresse und das Alter. Insgesamt werden ungefähr 8 Millionen Personen in der Datenbank erfasst. Die NCVR-Datenbank wird häufig zur Evaluierung des PPRL eingesetzt [VCV13] und soll auch im Rahmen dieser Arbeit benutzt werden. Darüber hinaus sollen auch vollständig synthetische (generierte) Datenmengen genutzt werden. Zur Erzeugung solcher Daten existieren verschiedene Daten-Generatoren, die personenbezogene Daten zu Testzwecken erzeugen können [CC02, Chr05, Chr08, CP09, Chr09]. Der Vorteil synthetischer Daten ist, dass die Charakteristiken der Daten genau festgelegt werden können. So lässt sich beispielsweise die Anzahl der enthaltenen Datensätze und die Anzahl ihrer Duplikate kontrollieren. Außerdem können die Daten auf gewünschte Weise modifiziert werden, sodass typische Fehler und die entsprechenden Fehlerhäufigkeiten simuliert werden können. Zudem ist bei synthetischen Daten bekannt, welche Datensätze ein Duplikat darstellen, womit die Qualität des PPRL untersucht werden kann. Schließlich ist ein weiterer Vorteil von synthetischen Daten, dass die verwendeten Testdaten veröffentlicht werden können, sodass Experimente wiederholbar und verifizierbar sind [Vat14].

[13] `http://dl.ncsbe.gov/`, Zugriff: 04.04.2017

Nachfolgend werden die Charakteristiken der Datenmengen, die im Rahmen dieser Arbeit verwendet werden, näher beschrieben. Darüber hinaus wird erläutert, wie die Maskierung der Daten erfolgt.

Synthetische Datenmengen Die synthetischen Testdaten wurden mit Hilfe des Daten-Generators und Daten-Korruptors des *Febrl*-Toolkits[14] [CC02, Chr08] erzeugt. Die Generierung der Daten basiert dabei auf verschiedenen Look-up-Tabellen, die beispielsweise zahlreiche Namen und Adressfragmente sowie Altershäufigkeiten enthalten. Außerdem können die Ausgangsdaten durch verschiedene Typen von Modifikationen verändert werden, wodurch sich unsaubere Daten simulieren lassen. Unter anderem ist es möglich, die Datensätze durch das Einfügen von typographischen und phonetischen Modifikationen abzuändern. Die Generierung der Daten erfolgt dabei so, dass zuerst die Original-Daten generiert werden. In einem zweiten Schritt werden dann die Duplikate erzeugt, auf welche die Modifikationen mit definierten Wahrscheinlichkeiten und Häufigkeiten angewandt werden [CC02, Chr08]. In Tabelle 4.1 ist beispielhaft ein Ausschnitt der generierten Testdaten dargestellt. Wie in der Tabelle ersichtlich, werden verschiedene Attribute, wie zum Beispiel Name, Adresse, Postleitzahl oder das Geburtsdatum (dob), erzeugt.

rec_id	given_name	surname	address	suburb	postcode	state	dob
rec-110-org	gabrielle	thorpe	blackburn street	wendouree	2264	nsw	19280928
rec-369-org	noah	brooker	pinkerton circuit	tweed heads	2251	nsw	19760219
rec-116-dup-0	alexnader	nevdin	wrest street	leichhardt	6069	nsw	19710508
rec-116-org	alexander	nevin	wrest street	leichhardt	6069	nsw	19710508

Tabelle 4.1: Beispiel: Synthetische Daten.

Mit Hilfe des Datengenerators wurden zur Evaluation zwei synthetische Datenkorpora[15], welche mit *GEN-L* und *GEN-M* bezeichnet werden, generiert. Die beiden Datenkorpora enthalten Datenmengen bestehend aus 1.000, 5.000, 10.000, 50.000, 100.000, 500.000, 1.000.000 und 5.000.000 Datensätzen. Die einzelnen Datenmengen sind dabei so aufgebaut, dass 20 % der Datensätze Duplikate sind, wobei jedes Duplikat genau einem Original-Record entspricht. Dadurch kann jede Datenmenge in zwei Dateien aufgeteilt werden: Eine Datei A, welche die Original-Datensätze enthält und eine Datei B, welche die Duplikate enthält. Diese beiden Dateien dienen als Eingabe für die zu evaluierenden P3RL-Verfahren. Zur Unterscheidung, welcher Datensatz einem Original-Record bzw. einem Duplikat entspricht, dienen die Identifikator-Suffixe 'org' bzw. 'dup'.

Die beiden Datenkorpora GEN-L und GEN-M unterscheiden sich hinsichtlich der Qualität ihrer Daten. Der Datenkorpus GEN-L dient zur Simulation von Daten, welche leicht

[14] `https://sourceforge.net/projects/febrl/`, Zugriff: 05.04.2017.

[15] Aufgrund der Mehr- bzw. Uneindeutigkeit des Wortes *Datensatz* wird im Rahmen dieser Arbeit der Begriff *Datenkorpus* benutzt. Ein Datenkorpus ist eine Sammlung von Datenmengen mit ähnlichen Charakteristiken. Eine Datenmenge (engl. *dataset*) besteht wiederum aus einzelnen Datensätzen (engl. *records*).

verschmutzt sind, also nur wenige Fehler enthalten. Der Datenkorpus GEN-M hingegen dient zur Simulation von Daten mit mittlerer Qualität. Konkret bedeutet dies, dass Datensätze aus GEN-L insgesamt maximal zwei Modifikationen (Fehler) aufweisen, wobei je Feld (Attribut) maximal eine Modifikation erlaubt ist. Die Datensätze aus GEN-M können insgesamt drei Modifikationen aufweisen, wobei maximal zwei Modifikationen je Feld möglich sind. Die Art der Modifikation wird über Wahrscheinlichkeitswerte bestimmt, welche angeben mit welcher Wahrscheinlichkeit welche Operation auf einem Feld durchgeführt wird. Die einzelnen Wahrscheinlichkeiten sind dabei abhängig von dem gewählten Feld, welches ebenfalls zufällig aus allen vorhanden Feldern ausgewählt wird. Insgesamt wurden die Wahrscheinlichkeiten so gewählt, dass phonetische und typographische Fehler überwiegen. Nur mit sehr geringer Wahrscheinlichkeit (0-5 %) werden Wörter innerhalb von Feldern vertauscht (falls vorhanden) oder ganze Felder ersetzt bzw. gelöscht.

Weiterhin wird bei beiden Datenkorpora ausgeschlossen, dass fehlende Werte für Vor- und Nachnamen auftreten, da diese für das phonetische Blocking benutzt werden. In der Realität können diese Attribute jedoch fehlen bzw. unbekannt sein. Das Fehlen von Werten, welche als Blocking-Key benutzt werden, stellt eine Herausforderung für Blocking-Verfahren dar, da diese Datensätze keinem Block zugeordnet werden können. Die Berücksichtigung derartiger Fehler ist daher schwierig und wird im Rahmen dieser Arbeit nicht näher untersucht.

Anzumerken ist, dass sich bei der derzeitigen Implementierung des benutzten Febrl-Datengenerators eingefügte Modifikationen auch gegenseitig neutralisieren können, indem eine Modifikation eine zuvor eingefügte Modifikation rückgängig macht (beispielsweise bei der Vertauschung eines Zeichens).

NCVR-Datenmengen Die Datensätze der NCVR-Datenbank dienen zur Erzeugung eines dritten Datenkorpus, welcher durch *NCVR* gekennzeichnet wird. In diesem sind die in [Vat14] verwendeten Datenmengen, bestehend aus 10.976, 109.772 und 1.097.720 Datensätzen, enthalten. Die einzelnen Datensätze wurden dabei aus der realen NCVR-Datenbank entnommen, wobei als Attribute der Vorname, der Nachname, die Stadt und die Postleitzahl extrahiert wurden. Die Datenmengen sind hier so aufgebaut, dass insgesamt 25 % der Datensätze Duplikate sind, wobei wiederum jedes Duplikat genau einem Original-Record entspricht. Die Aufteilung der Datensätze auf die zwei Eingabe-Dateien erfolgt so, dass jede Datei genau die Hälfte der Datensätze enthält, das heißt 5.488, 54.886 bzw. 548.860 Datensätze. Hiervon sind jeweils 50 % die Duplikate, das heißt insgesamt 2.744, 27.443 bzw. 274.430. Diese Eigenschaften führen zu einer anderen Ausgangslage als bei den synthetischen Datenmengen. Einerseits haben die beiden Eingabe-Datenmengen die gleiche Größe, wodurch die Anzahl aller möglichen Record-Paare höher ist als bei Da-

tenmengen mit unterschiedlicher Größe, aber derselben Gesamtanzahl von Datensätzen.[16] Andererseits besteht bei den synthetischen Daten eine Eingabe-Datenmenge (Datei B) vollständig aus Duplikaten, wohingegen bei den NCVR-Datenmengen neben den Duplikaten noch weitere Datensätze (in Datei B) enthalten sind. Zur Simulation von fehlerhaften (schmutzigen) Daten wurde das Tool *GeCo*[17] [CV13, TVC13] benutzt, um die Datensätze zu manipulieren [Vat14]. Hierbei wurde für diese Arbeit die Datenmenge ausgewählt, bei der für jeden Datensatz eine Modifikation durchgeführt wurde. Jedes Attribut wird dabei mit einer Wahrscheinlichkeit von 25 % gewählt. Die Fehlertypen sind Zeichenänderungen (Einfügung, Löschung, Ersetzung, Vertauschung), Eingabefehler (Simulation von Fehlern aufgrund von Vertippen auf der Tastatur), phonetische Fehler sowie Fehler aufgrund von optischen Fehlinterpretationen (Ersetzung von Zeichen durch ein anderes Zeichen mit ähnlicher Form). Jede dieser Modifikationen wird dabei mit einer Wahrscheinlichkeit von 25 % gewählt. Die Ausnahme bildet das Feld Postleitzahl, hier werden keine Eingabefehler und keine phonetischen Fehler eingefügt, weshalb die zwei übrigen Modifikationen mit einer Wahrscheinlichkeit von 50 % gewählt werden [Vat14].

Maskierung der Daten Die Maskierung der Daten erfolgt durch die Verwendung von Bloom-Filtern. Die grundlegenden Überlegungen zur Wahl der Parameter der Bloom-Filter wurden in Unterabschnitt 2.2.2 beschrieben. Im Rahmen dieser Arbeit wird die Anzahl der Hashfunktionen für die Bloom-Filter auf $k = 20$ festgelegt. Dies entspricht einer Falsch-positiv-Rate von $fpr \approx 10^{-6}$. Die notwendige Länge m der Bloom-Filter ist abhängig von der Anzahl der aufzunehmenden Elemente ω. Um diese Anzahl abzuschätzen, muss zunächst festgelegt werden, welche Attribute für das Linkage benutzt bzw. berücksichtigt werden sollen. Für die synthetischen Datenmengen sind dies der Vorname, der Nachname, das Geburtsdatum, die Postleitzahl und das Bundesstaat-Kürzel. Für die NCVR-Daten werden die vier vorhandenen Attribute Vorname, Nachname, Stadt und Postleitzahl benutzt. Nachfolgend muss abgeschätzt werden, wie viele Q-Gramme von den einzelnen gewählten Attributen durchschnittlich erzeugt werden. Hierfür ist zum einen wichtig, welche Länge Γ die entsprechenden Attributwerte im Durchschnitt aufweisen. Zum anderen muss die Länge Q der Q-Gramme festgelegt werden. Außerdem muss bestimmt werden, ob zusätzliche Füllzeichen benutzt werden. Im Rahmen dieser Arbeit werden ausschließlich Trigramme ($Q = 3$) berücksichtigt und die Attribute werden in $Q - 1 = 2$ Füllzeichen am Anfang sowie am Ende eingebettet. Insgesamt ergeben sich damit $2 \cdot (Q-1) = 4$ zusätzliche Füllzeichen. Für die durchschnittliche Anzahl an Zeichen für ein Attribut mit durchschnittlicher Länge Γ (inklusive Füllzeichen) ergibt sich die Anzahl der resultierenden Q-Gramme als $(\Gamma - Q) + 1$. Die Abschätzung der durchschnitt-

[16] Angenommen beide Eingabe-Datenmengen bestehen aus jeweils 5 Datensätzen. Die Anzahl aller möglichen Record-Paare ist $5 \cdot 5 = 25$. Enthält eine Eingabe-Datenmenge jedoch 3 Datensätze und die andere 7 Datensätze, so existieren nur $3 \cdot 7 = 21$ mögliche Record-Paare.

[17] `http://dmm.anu.edu.au/geco`, Zugriff: 05.04.2017.

lichen Länge der Attribute und der daraus resultierenden durchschnittlichen Anzahl an Q-Grammen für die Testdatensätze ist in Tabelle 4.2 dargestellt. Die Abschätzung beruht dabei auf [DKX+14] sowie einer manuellen Analyse der einzelnen Datensätze.[18] Für die Postleitzahl (PLZ) wurden zwei verschiedene Werte benutzt, da bei den synthetischen Datensätzen aus GEN-L bzw. GEN-M die Postleitzahl stets die Länge 4 hat und bei den NCVR-Datensätzen die Länge 5. Insgesamt ergeben sich damit durchschnittlich 48 Trigramme für einen synthetischen Datensatz und 34 Trigramme für einen NCVR-Datensatz. Unter Verwendung von Gleichung 2.30 lässt sich damit bestimmen, wie die Länge m der Bloom-Filter gewählt werden sollte. Für die synthetischen Datenmengen ergibt sich die Länge der Bloom-Filter als $m_{\text{GEN}} = \frac{20 \cdot 48}{\ln(2)} = 1385$. Für die NCVR-Daten hingegen ergibt sich eine Länge von $m_{\text{NCVR}} = \frac{20 \cdot 34}{\ln(2)} = 981$.

	Vorname	Nachname	GebDat	PLZ	Staat	City
Feldlänge (ohne Füllzeichen)	6	7	8	4 | 5	3	8
Feldlänge (mit Füllzeichen)	10	11	12	8 | 9	7	12
Q-Gramme ($Q = 3$)	8	9	10	6 | 7	5	10

Tabelle 4.2: Abschätzung der Feldlängen der Attribute und der Anzahl der Q-Gramme für die genutzten Testdatensätze.

Schließlich muss noch der Ähnlichkeitsschwellwert t festgelegt werden, welcher zur Klassifikation der Kandidaten-Record-Paare benötigt wird. Für die synthetischen Datensätze wurde dieser Schwellwert auf $t_{\text{GEN}} = 0{,}8$ und für die NCVR-Datensätze auf $t_{\text{NCVR}} = 0{,}9$ gesetzt. Die Entscheidung, zwei verschiedene Schwellwerte für die beiden Datenkorpora zu wählen, hat mehrere Ursachen. Einerseits kann ein synthetischer Datensatz bis zu drei Fehler (Modifikationen) enthalten, wohingegen in einem NCVR-Datensatz nur ein Fehler enthalten ist. Daher sollte der Schwellwert für die synthetischen Datenmengen etwas niedriger gewählt sein, um auch eine Toleranz gegenüber mehreren Fehlern zu gewährleisten. Andererseits erzeugen die synthetischen Datensätze mehr Q-Gramme, da mehr Attribute berücksichtigt werden. Werden mehr Attribute berücksichtigt, erhöht das die Wahrscheinlichkeit dafür, dass sich die Bloom-Filter von zwei nicht übereinstimmenden Datensätzen in mehr Bit-Positionen unterscheiden. Zur Veranschaulichung eignet sich die Betrachtung von Familienhaushalten. Leben mehrere Familienmitglieder in einem gemeinsamen Haushalt, so haben alle Personen die gleichen Adressdaten (Straße, Postleitzahl, Stadt, Bundesland) und häufig auch den gleichen Nachnamen. Obwohl die einzelnen Datensätze zu den entsprechenden Personen nicht übereinstimmen, das heißt nicht die gleiche Person referenzieren, sind sie sich doch sehr ähnlich. Attribute wie der Vorname und das

[18] In der Praxis können solche Abschätzungen durch Analyse öffentlich verfügbarer Datensätze erfolgen. Außerdem existieren statistische Häufigkeitstabellen, welche beispielsweise die Häufigkeiten von Personen- und Straßennamen erfassen. Solche Häufigkeitstabellen können aus Census-Daten abgeleitet werden [Com]. Manche Felder (z. B. PLZ) sind zudem standardisiert und daher einfach abzuschätzen.

Geburtsdatum sind in diesem Zusammenhang wichtig, um die Datensätze voneinander abzugrenzen. Ähnlich ist es bei den verwendeten NCVR-Datensätzen. Da hier nur vier Attribute berücksichtigt werden, unterscheiden sich die Datensätze insgesamt weniger voneinander. Aus diesem Grund muss der Schwellwert für die NCVR-Datensätze höher gewählt werden, um False-Matches zu vermeiden.

Parameter der Blocking-Verfahren Für die Verfahren zum phonetischen Blocking müssen die Parameter für die Blocking-Bloom-Filter festgelegt werden. Diesbezüglich wurde einheitlich festgelegt, dass die Anzahl der Hashfunktionen auf 15 gesetzt wird, wodurch sich eine Länge von $\frac{15 \cdot 1}{\ln(2)} = 22$ ergibt.[19] Außerdem soll bei den Verfahren zum phonetischen Blocking der Nachname als Blocking-Key dienen. Beim SPB wird zusätzlich der Vorname als Salt benutzt. Die phonetische Codierung erfolgt jeweils unter Verwendung von Soundex.

Für das LSH ist die Wahl der Parameter Ψ und Λ nicht trivial. Einerseits beeinflussen sich die Parameter wechselseitig (vgl. Unterabschnitt 2.3.1) und andererseits hängt die Wahl von der Qualität der Ausgangsdatensätze ab. Ein Ziel dieser Arbeit ist, verschiedene LSH-Parameter zu evaluieren und dabei zu berücksichtigen, welche Besonderheiten sich bei der parallelen Ausführung von LSH mit Flink ergeben. In Unterunterabschnitt 2.3.1.3 wurde dargestellt, welche grundsätzlichen Überlegungen bei der Wahl der Parameter für das LSH entscheidend sind. Zur Bestimmung, welche Parameterkonstellationen im Rahmen dieser Arbeit untersucht werden sollen, wurde Gleichung 2.35 näher betrachtet. Die Gleichung dient zur Abschätzung der Wahrscheinlichkeit, mit der die Bloom-Filter x und y, die eine Ähnlichkeit von $Sim(x, y)$ aufweisen, in wenigstens einem LSH-Key übereinstimmen und damit demselben Block zugeordnet werden. Wenn x und y eine Ähnlichkeit von $Sim(x, y) = 0{,}8$ aufweisen und die Länge des LSH-Keys auf $\Psi = 5$ festgelegt wird, beträgt die Wahrscheinlichkeit, dass x und y in einem LSH-Key übereinstimmen, nur ungefähr 33 % (vgl. Unterabschnitt 2.3.1). Für die synthetischen Datenmengen wurde der Ähnlichkeitsschwellwert auf $t_{\text{GEN}} = 0{,}8$ und für die NCVR-Datenmengen auf $t_{\text{NCVR}} = 0{,}9$ gesetzt. Folglich sollten möglichst alle Bloom-Filter-Paare, welche eine Ähnlichkeit von 80 % bzw. 90 % aufweisen, als Kandidaten-Record-Paar durch das LSH gebildet werden. Um dies zu Erreichen wurde zu Beginn der Evaluation der Parameter δ, welcher die tolerierbare Fehlerwahrscheinlichkeit für False-non-Matches beim LSH angibt, sehr restriktiv mit $\delta_{\text{GEN}} = \delta_{\text{NCVR}} = 0{,}00001 = 10^{-5}$ gewählt. Außerdem wurde der Wert τ auf $\tau_{\text{GEN}} = t_{\text{GEN}}$ bzw. $\tau_{\text{NCVR}} = t_{\text{NCVR}}$ festgelegt. Mit diesen Überlegungen berechnet sich nach Gleichung 2.36 die optimale Anzahl an LSH-Keys (Iterationen) als $\Lambda_{\text{GEN}} = \lceil \ln(0{,}00001) / \ln(1 - (0{,}8)^5) \rceil = 29$ bzw.

[19] In den Bloom-Filter, welcher zum Blocking genutzt wird, wird nur ein Wert aufgenommen, nämlich die phonetische Codierung des Blocking-Attributs.

$\Lambda_{\text{NCVR}} = \lceil \ln(0{,}00001) / \ln(1 - (0{,}9)^5) \rceil = 13$. Wie an den Werten ersichtlich ist, ist die Anzahl der notwendigen LSH-Keys für die NCVR-Datenmengen deutlich geringer als für die synthetischen Datenmengen. Dies macht deutlich, dass sich die Charakteristiken der synthetischen Datenmengen, vor allem die höhere Anzahl von Fehlern bzw. Modifikationen, negativ auf das LSH auswirken, da mehr LSH-Keys zu einem höheren Berechnungsaufwand führen.

Für den Parameter δ wurden zudem weniger restriktive Werte mit $\delta_{\text{GEN}} \in \{0{,}01; 0{,}25; 0{,}5\}$ und $\delta_{\text{NCVR}} \in \{0{,}01; 0{,}25\}$ gewählt. Insgesamt werden hierdurch für die synthetischen Datenmengen folgende zusätzliche LSH-Parameterkonstellationen der Form (Ψ, Λ) berücksichtigt: $(5, 12)$, $(5, 4)$ und $(5, 2)$. Für die NCVR-Daten ergeben sich hingegen zusätzlich $(5, 6)$ sowie $(5, 2)$ als LSH-Parameter.

	GEN-L	**GEN-M**	**NCVR**
#Fehler (Feld)	1	2	1
#Fehler (gesamt)	2	3	1
Aufteilung der Datensätze	80/20		50/50
#Duplikate (gesamt)	20 %		25 %
Maximale #Datensätze	5.000.000		1.097.720
Länge der Q-Gramme	$Q = 3$		
#Hashfunktionen Bloom-Filter	$k = 20$		
Länge Bloom-Filter	$m_{\text{GEN}} = 1385$		$m_{\text{NCVR}} = 981$
Ähnlichkeitsschwellwert	$t_{\text{GEN}} = 0{,}8$		$t_{\text{NCVR}} = 0{,}9$
Blocking-Attribut (PB, SPB)	Soundex(Nachname)		
Salt-Attribut (SPB)	Soundex(Vorname)		
#Hashfkt. Blocking-Bloom-Filter	15		
Länge Blocking-Bloom-Filter	22		
LSH-Parameter (Ψ, Λ)	$(5, 2), (5, 4), (5, 12),$ $(5, 29), (20, 6), (20, 18)$		$(5, 2), (5, 6),$ $(5, 13)$

Tabelle 4.3: Charakteristiken der verwendeten Datenkorpora und gewählte Parameter.

Während der Evaluation der Verfahren wurde festgestellt, dass $\Psi = 5$ im Zusammenhang mit großen Datenmengen zu Bildung von wenigen Blöcken führt (siehe Abschnitt 4.4). Dies liegt darin begründet, dass durch $\Psi = 5$ nur maximal $2^5 = 32$ Blöcke pro LSH-Key entstehen können. Daher wurde die Länge der LSH-Keys schließlich auf $\Psi = 20$ erhöht und für die synthetischen Datenmengen evaluiert. Außerdem wurde für die synthetischen Datenmengen deutlich, dass von den Bloom-Filtern, die übereinstimmende Datensätze (Matches) maskieren, oft Ähnlichkeiten erreicht werden, welche deutlich über dem gewählten Schwellwert t_{GEN} liegen. Dies wurde dadurch ersichtlich, dass deutlich weniger False-non-Matches auftraten, als zuvor mit dem Parameter δ festgelegt wur-

den. Aus diesem Grund wurde der Parameter τ_{GEN} auf $\tau'_{\text{GEN}} = 0{,}93$ erhöht. Die nun berücksichtigten Fehlerwerte δ'_{GEN} wurden auf $\{0{,}01; 0{,}25\}$ gesetzt. Für die Anzahl der LSH-Keys Λ ergibt sich dadurch einerseits $\Lambda = \lceil \ln(0{,}01) / \ln(1-(0{,}93)^{20}) \rceil = 18$ und andererseits $\Lambda = \lceil \ln(0{,}25) / \ln(1-(0{,}93)^{20}) \rceil = 6$. Insgesamt ergeben sich damit die weiteren LSH-Parameter $(20, 6)$ und $(20, 18)$.

In Tabelle 4.3 sind zusammenfassend die wesentlichen Charakteristiken der benutzten Datenkorpora sowie die verwendeten Parameter dargestellt.

4.2 Systemkonfiguration

Die verwendete Testumgebung setzt sich wie folgt zusammen:

- Computer-Cluster mit 5 Servern, jeweils mit folgenden Spezifikationen:
  - 64 GB RAM
  - 7 TB HDD
  - Intel Core i7-6700 (4×3,4 GHz mit jeweils 2 Threads)
  - Interne und externe Netzwerkschnittstelle mit jeweils 1 Gbit/s
  - CentOS Linux 7.2 (1511)
  - Java(TM) SE Runtime Environment (build 1.8.0_73-b02)
  - Hadoop 2.7.1.2.4.0.0-169
- Flink 1.1.4
  - 1 Job-Manager mit 8 GB JVM-Heap
  - 4 Task-Manager mit jeweils 58 GB JVM-Heap und 8 Task-Slots

4.3 Methodik

Die Evaluierung der umgesetzten P3RL-Verfahren Nested-Loop, Phonetic-Blocking (PB), Salted-phonetic-Blocking (SPB) sowie Locality-sensitive-Hashing (LSH) erfolgt durch die Durchführung der Verfahren auf allen Datenmengen der vorgestellten Datenkorpora. Dazu wurden die einzelnen Datenmengen vorab im HDFS gespeichert. Für jedes Verfahren wird die Ausführungszeit erfasst sowie die Metriken Reduction-Rate, Pairs-Quality, Pairs-Completeness und F-Measure berechnet. Während die Messung der Ausführungszeit di-

rekt durch Flink erfolgt, findet die Berechnung der Kennzahlen, welche zur Bestimmung der genannten Metriken notwendig sind, innerhalb der umgesetzten Verfahren statt. Damit ist die Berechnung der Kennzahlen in der Ausführungszeit der Verfahren mit eingeschlossen. Darüber hinaus werden die Linkage-Ergebnisse in das HDFS geschrieben, was ebenfalls die Ausführungszeiten beeinflusst.

Insgesamt verfügt die verwendete Testumgebung über 32 Task-Slots, wodurch ein Flink-Job mit einer maximalen Parallelität von $\Upsilon = 32$ ausgeführt werden kann. Zur Untersuchung des erreichbaren Speedups [RSS15] werden die Verfahren für verschiedene Flink-Job-Parallelitätsgrade $\Upsilon \in \{1, 2, 4, 8, 16, 32\}$ durchgeführt. Somit lässt sich bestimmen, wie der verwendete Parallelitätsgrad die Laufzeit der verschiedenen Verfahren beeinflusst und sich damit auf die Skalierbarkeit auswirkt.

Um möglichst statistisch signifikante Ergebnisse zu erhalten, wird jeder Testlauf bis zu zehnmal wiederholt. Die genaue Anzahl an Wiederholungen schwankt dabei je nach Verfahren, Anzahl der Datensätze in der betrachteten Datenmenge und der gesetzten Flink-Job-Parallelität Υ. Im Allgemeinen wurde die Anzahl der Wiederholungen für große Datenmengen und Verfahren mit langen Ausführungszeiten ($> 6\,\mathrm{h}$) stark reduziert, da ansonsten ein sehr hoher Zeitaufwand notwendig gewesen wäre. Zur Auswertung werden die Ergebnisse der einzelnen Testläufe aggregiert, indem das arithmetische Mittel und die Standardabweichung für die gesammelten Metriken berechnet werden. Dieses Vorgehen ist wichtig für die Qualität der Testergebnisse, vor allem in Bezug auf das LSH. Dieses wurde so umgesetzt, dass die Wahl der genutzten Hashfunktionen zufällig erfolgt, wodurch sowohl die Ausführungszeiten als auch die Qualität der Linkage-Ergebnisse schwanken können.

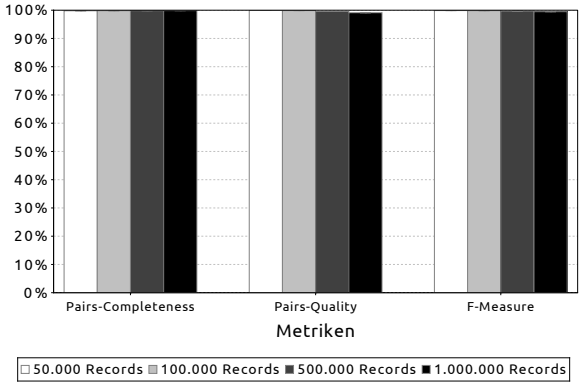
Die Durchführung der Testläufe sowie die Berechnung der Metriken erfolgt durch das im Rahmen der Arbeit entwickelte Framework zur Durchführung des P3RL mit Flink (siehe Kapitel 3). Die erhaltenen Ergebnisse werden in einer Datenbank gespeichert und aggregiert. Anschließend werden die Ergebnisse vom entwickelten Framework zur Erzeugung von Diagrammen weiterverarbeitet.

4.4 Ergebnisse

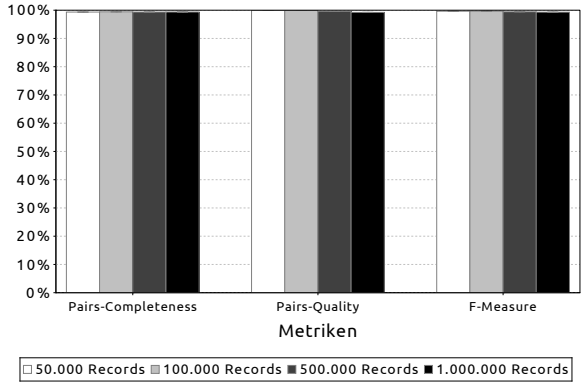
Im nachfolgenden Abschnitt werden die Ergebnisse der Testläufe vorgestellt und diskutiert. Zunächst wird dazu betrachtet, welche Ergebnis-Qualität die umgesetzten Verfahren erreichen. Anschließend werden die Ausführungszeiten der Verfahren für verschieden große Datenmengen sowie unterschiedliche Flink-Job-Parallelitätsgrade ausgewertet.

4.4.1 Qualität

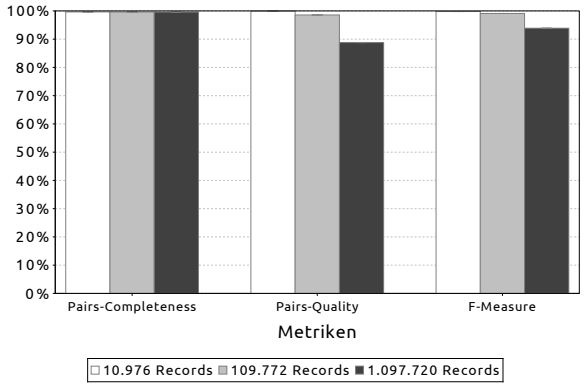
Qualität der Bloom-Filter-Codierung Zur Bestimmung der erreichbaren Linkage-Qualität eignet sich die Betrachtung der Ergebnisse des Nested-Loop-Verfahrens. Hierbei werden alle möglichen Record-Paare miteinander verglichen, weshalb die Ergebnisse des Nested-Loop-Verfahrens als Vergleichsgrundlage für die anderen Verfahren dienen. Die Ergebnisse für die genutzten Datenkorpora mit verschiedenen Datenmengen sind in Abbildung 4.1 dargestellt. Für die Datenkorpora GEN-L sowie GEN-M wurden die Ergebnisse auf die Datenmengen mit 50.000 bis 1.000.000 Datensätze beschränkt. Der Grund hierfür ist, dass sich einerseits die Ergebnisse für kleinere Datenmengen nicht signifikant von den betrachteten Datenmengen unterscheiden. Andererseits war eine Durchführung des Nested-Loop-Verfahrens für die Datenmengen mit 5.000.000 Datensätzen aufgrund der quadratischen Komplexität des Verfahrens nicht mehr in einem akzeptablen Zeitrahmen möglich.



(a) GEN-L



(b) GEN-M



(c) NCVR

Abbildung 4.1: Linkage-Qualität für das Nested-Loop-Verfahren.

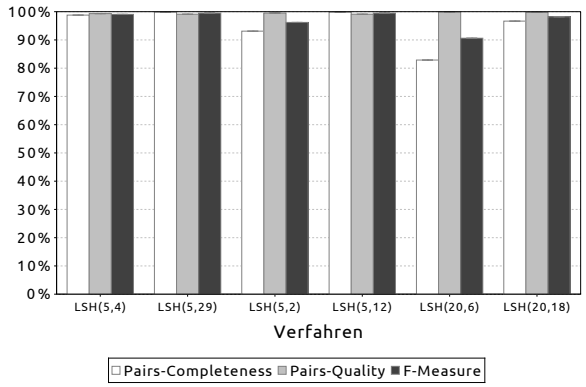
Wie aus den dargestellten Diagrammen deutlich wird, wird bei allen genutzten Datenmengen eine nahezu optimale Pairs-Completeness erreicht. Einzig für die Datenmengen

aus dem Datenkorpus GEN-M nimmt die Pairs-Completeness etwas ab, jedoch können hier immer noch ungefähr 98 % aller übereinstimmenden Datensätze aufgefunden werden. Insgesamt wird die Pairs-Completeness kaum durch eine Erhöhung der Anzahl der Datensätze beeinflusst. Auch für die Pairs-Quality können sehr gute Werte ($\geq 97\,\%$) für die synthetischen Datenmengen erzielt werden. Anders als die Pairs-Completeness verschlechtert sich jedoch die Pairs-Quality bei größeren Datenmengen. Besonders wird dies bei den NCVR-Datenmengen deutlich, wo die Pairs-Quality für die größte Datenmenge mit 1.097.720 Datensätzen auf unter 90 % fällt. Demzufolge werden für diese Datenmenge vergleichsweise viele False-Matches berechnet. Die Ursache hierfür liegt in der geringen Anzahl an Attributen, welche für die NCVR-Datenmengen berücksichtigt werden (vgl. Abschnitt 4.1). Bei der manuellen Analyse der False-Matches wurde deutlich, dass ein Großteil der False-Matches Personen betrifft, welche dieselben Adressdaten und einen ähnlichen Vor- oder Nachnamen aufweisen. Um derartige False-Matches zu vermeiden, könnte der Ähnlichkeitsschwellwert t erhöht werden. Dies würde sich jedoch negativ auf die Pairs-Completeness auswirken, da dann einige der fehlerhaften Datensätze nicht mehr korrekt klassifiziert werden. Insgesamt zeigen die Ergebnisse für das Nested-Loop-Verfahren, dass durch die verwendete Bloom-Filter-Codierung qualitativ hochwertige Linkage-Ergebnisse berechnet werden können.

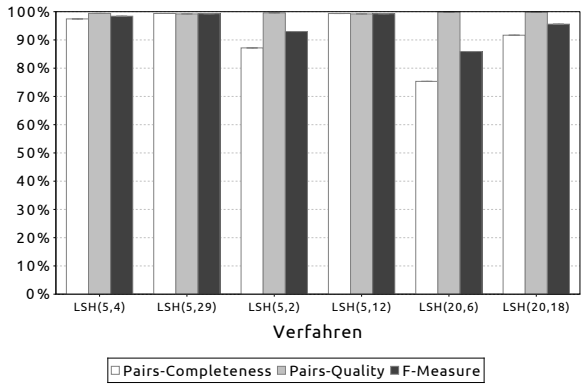
LSH-Verfahren Zur Evaluierung des LSH wurden verschiedene Werte für die beiden Parameter Ψ und Λ gewählt (siehe Abschnitt 4.1). Zur Bestimmung, welche Linkage-Qualität von den einzelnen Verfahren erreicht wird, werden nachfolgend die Datenmengen mit 1.000.000 (GEN-L, GEN-M) bzw. 1.097.720 (NCVR) Datensätzen betrachtet. Die von den einzelnen Verfahren erzielten Ergebnisse sind in Abbildung 4.2 dargestellt. Die Ergebnisse sind dabei nach dem entsprechenden Datenkorpus unterteilt. Die jeweiligen LSH-Verfahren sind durch die verwendeten Parameter bestimmt und durch die Notation 'LSH(Ψ,Λ)' gekennzeichnet.

Wie erwartet, schwankt die Qualität der Linkage-Ergebnisse je nach Auswahl der beiden LSH-Parameter. Es wird deutlich, dass sich die Pairs-Quality-Werte im Vergleich zum Nested-Loop-Verfahren teilweise verbessern können. Dies liegt daran, dass durch das Blocking mit LSH einige nicht übereinstimmende Datensätze, welche jedoch einen Ähnlichkeitswert über dem Schwellwert t aufweisen, von einem genauen Vergleich ausgeschlossen wurden. Bezüglich der Pairs-Completeness ist erkennbar, dass die LSH-Verfahren 'LSH(5,29)' sowie 'LSH(5,12)' für die synthetischen Daten die besten Werte von ungefähr 99 % erreichen. Auch für die stärker modifizierten Datensätze aus dem Datenkorpus GEN-M verringert sich dabei die Pairs-Completeness für diese beiden Verfahren nur leicht. Gleichzeitig wird anhand der beiden Verfahren 'LSH(5,29)' und 'LSH(5,12)' deutlich, dass die Erhöhung der Anzahl der Iterationen bzw. LSH-Keys auf $\Lambda > 12$ nur noch

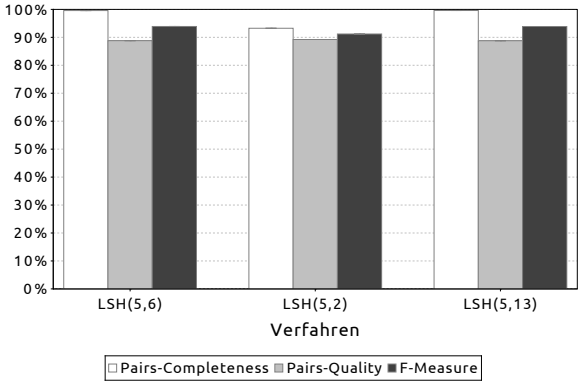
zu geringfügigen Gewinnen führt. Das bedeutet, dass bereits mit 12 LSH-Keys nahezu alle übereinstimmenden Datensatzpaare identifiziert werden. Die zusätzlichen Iterationen finden nur noch sehr wenige (unter 100) bis dahin unentdeckte Matches. Sogar eine weitere Reduzierung der Anzahl der LSH-Keys auf $\Lambda = 4$ bei 'LSH(5,4)' führt nur zu einem Verlust von ungefähr 2 %. Erst für 'LSH(5,2)', also bei Verwendung von $\Lambda = 2$ LSH-Keys, fällt die Pairs-Completeness merklich ab: Während bei GEN-L die Pairs-Completeness noch ungefähr 93% beträgt, fällt sie bei GEN-M auf unter 88 %. Bei der Betrachtung der NCVR-Ergebnisse fällt auf, dass 'LSH(5,6)' eine höhere Pairs-Completeness auf den NCVR-Datensätzen erreicht als 'LSH(5,4)' auf den Datensätzen aus GEN-L. Die Ursache hierfür ist, dass sich übereinstimmende Datensätze aus dem NCVR-Korpus ähnlicher sind, als es übereinstimmende Datensätze aus den synthetischen Datenkorpora sind.



(a) GEN-L



(b) GEN-M



(c) NCVR

Abbildung 4.2: Linkage-Qualität der verschiedenen LSH-Verfahren für die Datenmengen mit 1.000.000 (GEN-L, GEN-M) bzw. 1.097.720 (NCVR) Datensätzen.

Grundsätzlich fällt bei der Betrachtung der erreichten Pairs-Completeness-Werte auf, dass die bei der Parameterwahl angenommenen Fehlerwahrscheinlichkeiten δ nicht erreicht werden (vgl. Abschnitt 4.1). Beispielsweise wurde für das Verfahren 'LSH(5,2)' eine Fehlerwahrscheinlichkeit von bis zu 50% durch $\delta = 0.5$ angenommen. Tatsächlich werden für

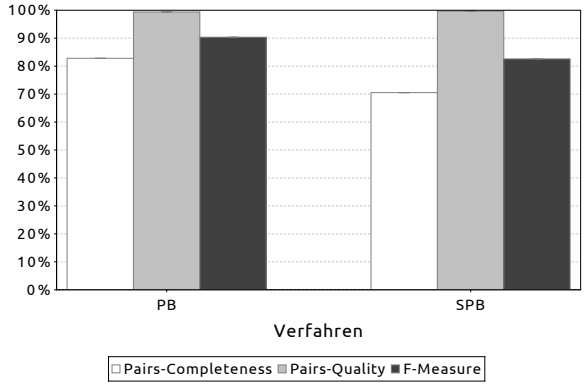
'LSH(5,2)' jedoch Pairs-Completeness-Werte von wenigstens 87 % erreicht, was deutlich unter der angenommenen maximalen Fehlerwahrscheinlichkeit liegt. Hieraus ist ableitbar, dass der Ähnlichkeitsschwellwert τ, welcher zur Bestimmung der LSH-Parameter benutzt wird, zu klein für die verwendeten Testdatenmengen gewählt wurde. Die Wahl eines geeigneten Wertes für τ ist wichtig, um möglichst optimale LSH-Parameter auszuwählen. In der Realität stellt dies ein Problem dar, da die Charakteristiken der jeweiligen Datensätze nur schwer abgeschätzt werden können.

Neben den LSH-Verfahren, bei denen die Länge der LSH-Keys auf $\Psi = 5$ gesetzt wurde, wurden außerdem noch die Verfahren 'LSH(20,6)' und 'LSH(20,18)' mit $\Psi = 20$ für GEN-L und GEN-M evaluiert. Es ist erkennbar, dass 'LSH(20,6)' von den umgesetzten LSH-Verfahren die geringsten Pairs-Completeness-Werte erzeugt: Für die Datenmenge aus GEN-M beträgt die Pairs-Completeness nur noch ungefähr 75 %. Der Grund für die vergleichsweise schlechten Werte ist, dass zu wenig LSH-Keys genutzt werden. Die Erhöhung der LSH-Key-Länge auf $\Psi = 20$ führt dazu, dass es für unähnliche Datensätze unwahrscheinlicher wird, denselben Wert für einen LSH-Key zu erzeugen. Ebenso können dadurch in der Realität übereinstimmende Datensätze aufgrund von fehlerhaften Daten unterschiedliche LSH-Keys erzeugen. Um die Chance zu erhöhen, dass zwei Datensätze in einem LSH-Key übereinstimmen, muss die Anzahl der LSH-Keys erhöht werden. Bei der Betrachtung der Ergebnisse von 'LSH(20,18)' wird deutlich, dass die Verwendung von 18 anstatt von 6 Keys zu einer Verbesserung der Pairs-Completeness von über 15 % im Vergleich zum 'LSH(20,6)' führt.

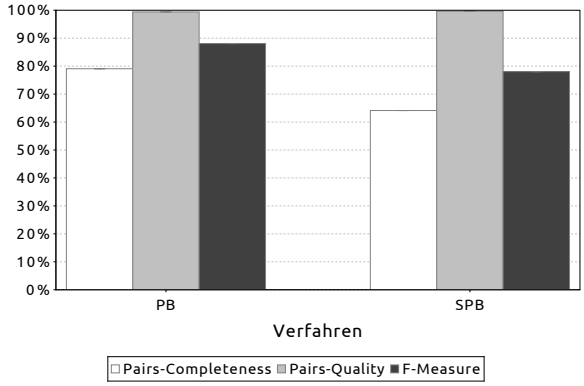
PB und SPB Die Ergebnisse der beiden Verfahren PB und SPB zum phonetischen Blocking sind in Abbildung 4.3 dargestellt. Wie bei den LSH-Verfahren werden auch hier die Ergebnisse für die Datenmengen mit 1.000.000 (GEN-L, GEN-M) bzw. 1.097.720 (NCVR) Datensätzen ausgewertet.

Es lässt sich feststellen, dass beim PB circa 80 % der Matches, welche in den synthetischen Datenmengen vorhanden sind, gefunden werden. Bei den stärker modifizierten Datensätzen aus GEN-M fällt dabei die Pairs-Completeness nur etwa um 3 % im Vergleich zu den leicht modifizierten Datensätzen aus GEN-L ab. Für die NCVR-Datenmenge liegt die Pairs-Completeness bei rund 87 %, was eine leichte Verbesserung gegenüber der GEN-L-Datenmenge darstellt. Daraus lässt sich schließen, dass die in die NCVR-Datensätze eingefügten Modifikationen besser durch die phonetische Codierung kompensiert werden können, als es bei den Datensätzen aus GEN-L der Fall ist. Die Ursache hierfür könnte sein, dass bei den NCVR-Datensätzen das Blocking-Attribut (Nachname) weniger häufig modifiziert wurde. Bei der Betrachtung der Ergebnisse des SPB ist erkennbar, dass beim SPB deutlich weniger Matches als beim PB erkannt werden können. So liegt die Pairs-Completeness beim SPB um ungefähr 10 – 15 % unter den beim PB erreichten Werten.

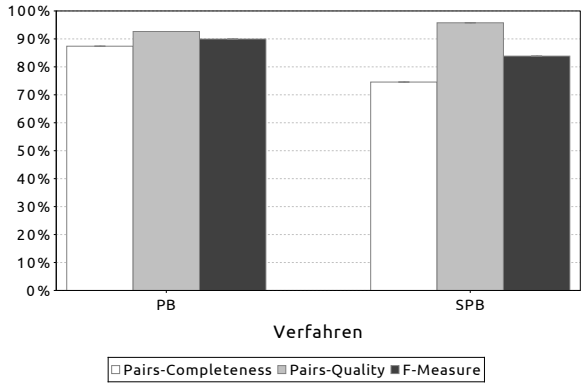
Damit führt die Nutzung eines weiteren Blocking-Attributs auch bei nur leicht verschmutzen Daten zu signifikanten Einbußen.



(a) GEN-L



(b) GEN-M



(c) NCVR

Abbildung 4.3: Linkage-Qualität der Verfahren PB und SPB für die Datenmengen mit 1.000.000 (GEN-L, GEN-M) bzw. 1.097.720 (NCVR) Datensätzen.

Insgesamt wird dennoch deutlich, dass durch die Nutzung phonetischer Codierungen viele Fehler kompensiert werden können. Die Datensätze der NCVR-Datenmenge wurden so modifiziert, dass ein Fehler in eines der vier benutzten Attribute eingefügt wurde. Die Wahrscheinlichkeit für die Modifikation ausgewählt zu werden, lag für jedes der vier Attribute bei 25 %. Angenommen jedes der Attribute wurde gleich häufig modifiziert, so liegt die Wahrscheinlichkeit, dass ein Blocking-Attribut einen Fehler aufweist, ebenfalls bei 25 %. Würde das Blocking direkt auf dem Blocking-Attribut durchgeführt, so könnten wiederum 25 % der Matches aufgrund dieser Modifikation nicht gefunden werden. Daraus folgt, dass ohne Einsatz einer phonetischen Codierung nur eine Pairs-Completeness von maximal 75 % beim PB möglich wäre. Beim SPB werden zwei Attribute für das Blocking benutzt, weshalb die Wahrscheinlichkeit dafür, dass genau eines der beiden Attribute eine Modifikation aufweist, bei 50 % liegt. Daher wäre ohne Nutzung einer phonetischen Codierung für das SPB nur eine Pairs-Completeness von 50 % erreichbar. Aus Abbildung 4.3

wird ersichtlich, dass beim PB und SPB eine Pairs-Completeness von 87 % bzw. 74 % erreicht wird. Damit können ungefähr die Hälfte aller in ein Blocking-Attribut eingefügten Modifikationen durch Einsatz der phonetischen Codierungen kompensiert werden.

Gesamtergebnis Die Evaluation der Verfahren zeigt, dass grundsätzlich qualitativ hochwertige Linkage-Ergebnisse unter Verwendung der Bloom-Filter-Codierung berechnet werden können. Insgesamt hat die Anzahl der betrachteten Datensätze nur einen geringen Einfluss auf die erzielte Qualität: Bei allen getesteten Verfahren bleibt die Pairs-Completeness für verschieden große Datenmengen nahezu gleich. Zwar weichen die Werte für einige Datenmengen um bis zu 4 % voneinander ab, jedoch sind diese Abweichungen unabhängig von der Größe der Datenmenge. Die Ursache der Abweichungen liegt stattdessen in den individuellen Charakteristiken der verschiedenen Datenmengen, welche sich durch die Einfügung von zufällig gewählten Modifikationen ergeben. Ebenso kann von den Verfahren eine hohe Pairs-Quality erzielt werden. Hierbei hat die Anzahl der betrachteten Datensätze einen höheren Einfluss auf die erreichbaren Werte. Besonders für die NCVR-Datenmengen nimmt die Pairs-Quality mit steigender Anzahl an Datensätzen ab.

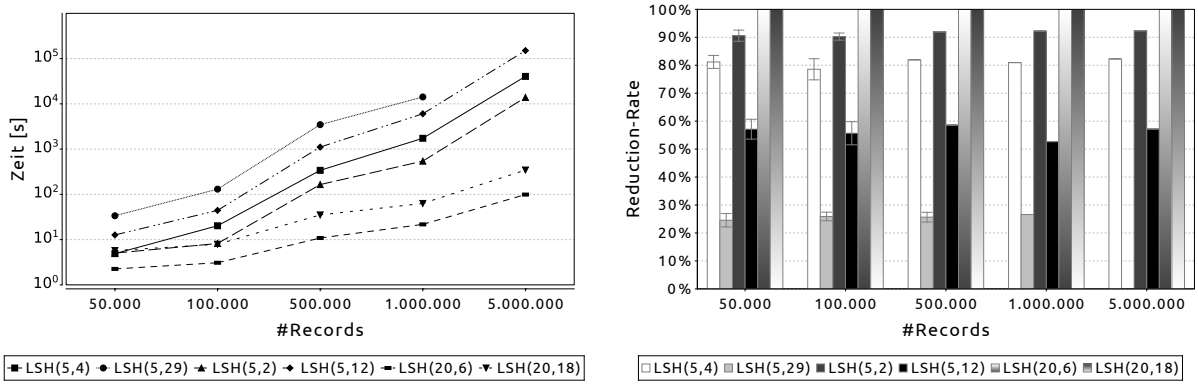
Der Vergleich der LSH-Verfahren mit den Verfahren zum phonetischen Blocking zeigt, dass durch Verwendung des LSH bessere Ergebnisse hinsichtlich der Pairs-Completeness erreicht werden können. Demnach können mit LSH mehr Matches im Vergleich zum phonetischen Blocking gefunden werden. Nur 'LSH(20,6)' erzielt ähnliche Ergebnisse hinsichtlich der Pairs-Completeness wie das Phonetic-Blocking-Verfahren. Insgesamt konnten durch Einsatz des Salted-phonetic-Blockings die wenigsten Matches gefunden werden. Dadurch wird deutlich, dass die Verfahren zum phonetischen Blocking nur dann zum Einsatz kommen sollten, wenn davon ausgegangen werden kann, dass die für das Blocking benutzten Attribute nur selten Fehler aufweisen.

4.4.2 Skalierbarkeit und Speedup

Die Untersuchung der umgesetzten Verfahren hinsichtlich ihrer Skalierbarkeit erfolgt durch die Analyse der jeweils erreichten Ausführungszeiten. Die nachfolgende Auswertung beschränkt sich dabei auf die Ergebnisse, welche auf den Datenmengen aus GEN-L bestehend aus 50.000 – 5.000.000 Datensätzen unter Ausnutzung der maximalen Flink-Job-Parallelität von $\Upsilon = 32$ erzielt wurden. In Tabelle 4.4 (Seite 90) sind die von den verschiedenen getesteten Verfahren erreichten Ausführungszeiten zusammengefasst. Hierbei ist die kürzeste benötigte Ausführungszeit je Datenmenge optisch hervorgehoben. Die Ergebnisse der LSH-Verfahren sowie der Verfahren zum phonetischen Blocking werden nachfolgend genauer betrachtet. Dabei wird auch darauf eingegangen, welcher Speedup durch Nutzung eines höheren Flink-Job-Parallelitätsgrades von den Verfahren erreicht werden kann.

LSH-Verfahren Die von den getesteten LSH-Verfahren erreichten Ausführungszeiten sind in Abbildung 4.4(a) gegenübergestellt. Außerdem sind in Abbildung 4.4(b) die erreichten Reduction-Rate-Werte der jeweiligen Verfahren einsehbar.

Es wird deutlich, dass die LSH-Verfahren je nach gewählten Parametern sehr unterschiedliche Ausführungszeiten erreichen. Grundsätzlich liegen die Ausführungszeiten der Verfahren, bei denen LSH-Keys der Länge 5 gewählt wurden, deutlich höher als bei den Verfahren mit einer LSH-Key-Länge von 20. Entsprechend der Abbildung 4.4(b) ist zu erkennen, dass auch die Reduction-Rate für Verfahren mit $\Psi = 5$ deutlich geringer ist als für die Verfahren mit $\Psi = 20$.



(a) Ausführungszeit (b) Reduction-Rate

Abbildung 4.4: Ausführungszeit und Reduction-Rate der verschiedenen LSH-Verfahren ($\Upsilon = 32$) für unterschiedlich große Datenmengen aus GEN-L.

Die beiden Verfahren 'LSH(20,6)' und 'LSH(20,18)' erreichen hinsichtlich der Reduction-Rate nahezu optimale Werte von über 99 %. Damit wird der Suchraum sehr stark eingeschränkt und es werden nahezu alle überflüssigen Record-Paar-Vergleiche vermieden. Dieser Umstand hat direkten Einfluss auf die Laufzeit der beiden Verfahren, was die Ursache für die geringeren Ausführungszeiten darstellt. Außerdem steigt die Ausführungszeit von 'LSH(20,6)' und 'LSH(20,18)' bei einer Erhöhung des Datenumfangs im Vergleich zu den anderen Verfahren weniger stark an. Demnach skalieren 'LSH(20,6)' und 'LSH(20,18)' im Vergleich zu den übrigen Verfahren besser. Insgesamt die kürzesten Ausführungszeiten werden mit $\Psi = 20$ und $\Lambda = 6$ erreicht. Dementsprechend benötigt das Verfahren 'LSH(20,6)' zur Durchführung des Linkages auf insgesamt 5.000.000 Datensätzen nur ungefähr 99 Sekunden (vgl. Tabelle 4.4). Weiterhin ist erkennbar, wie die Erhöhung der Anzahl der LSH-Keys von $\Lambda = 6$ auf $\Lambda = 18$ zu einer Erhöhung der Ausführungszeit von 'LSH(20,18)' im Gegensatz zu 'LSH(20,6)' führt. Diesbezüglich ist auffällig, dass die Verdreifachung der LSH-Keys von $\Lambda = 6$ auf $\Lambda = 18$ ungefähr zu einer Verdreifachung der Ausführungszeit führt.

Des Weiteren ist erkennbar, dass die Verfahren 'LSH(5,29)' und 'LSH(5,12)' die höchsten Laufzeiten erreichen. Gleichzeitig wird von diesen Verfahren nur eine geringe Reduction-Rate von ungefähr 25 % bzw. 55 % erzielt. Hierdurch zeigt sich erneut, dass die Anzahl der LSH-Keys bei 'LSH(5,29)' zu hoch gewählt wurde. Wie in Unterabschnitt 4.4.1 gezeigt, kann die Pairs-Completeness hierdurch ebenfalls nur unwesentlich verbessert werden.

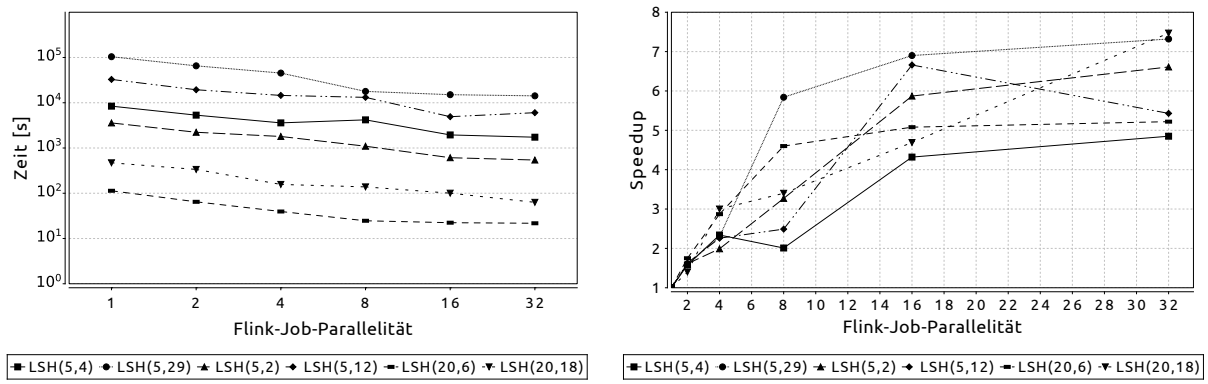
Zusammenfassend kann festgestellt werden, dass eine HLSH-Key-Länge von $\Psi = 5$ zu niedrig ist. Vor allem bei der gleichzeitigen Verwendung einer hohen Anzahl an LSH-Keys kann hierdurch der Suchraum nicht stark genug eingeschränkt werden. Die Ursache hierfür ist, dass unter Verwendung von $\Psi = 5$ nur maximal $2^5 = 32$ mögliche Werte für einen LSH-Key existieren. Angenommen es wird nur ein LSH-Key benutzt ($\Lambda = 1$), dann kann die Eingabe-Datenmenge nur in maximal 32 Blöcke aufgeteilt werden. Sind die Werte der LSH-Keys gleichmäßig verteilt, so würden sich in jedem Block $\frac{\bar{n}}{32}$ Datensätze befinden. Für eine Gesamtanzahl von $\bar{n} = 1.000.000$ Datensätzen wären dies $\frac{1.000.000}{32} = 31.250$ Datensätze je Block. Damit werden zu viele Datensätze demselben Block zugeordnet, was zu einer hohen Anzahl an notwendigen Vergleichen je Block führt. Wird gleichzeitig Λ höher gewählt, so werden je Iteration bzw. LSH-Key wiederum nur 32 Blöcke mit jeweils 31.250 Datensätzen gebildet. Außerdem wurde eine Mehrheit der Kandidaten-Paare bereits in der ersten Iteration erzeugt, weshalb diese Kandidaten-Paare später herausgefiltert werden müssen. Wird hingegen $\Psi = 20$ gesetzt, so kann ein HLSH-Key maximal $2^{20} = 1.048.576$ Werte annehmen, was zu wesentlich kleineren Blöcken führt. Hierdurch können zwar bei geringer Anzahl an LSH-Keys weniger Matches identifiziert werden (vgl. Unterabschnitt 4.4.1), jedoch wirkt sich eine Erhöhung der Anzahl der LSH-Keys weniger stark auf die erreichten Laufzeiten aus als bei geringerer Key-Länge. Der Grund dafür ist, dass je Iteration bzw. LSH-Key kleinere Blöcke entstehen.

Durch die zuvor beschriebenen negativen Effekte bei der Nutzung von $\Psi = 5$, war eine Ausführung von 'LSH(5,29)' auf der Datenmenge mit 5.000.000 Records nicht mehr möglich: Bei der Ausführung trat eine `OutOfMemoryException` aufgrund einer Überschreitung des Garbage-Collection-Overhead-Limits auf. Dies bedeutet, dass aufgrund der hohen Anzahl an LSH-Keys und der geringen Anzahl von Blöcken, zu viele Java-Objekte erzeugt werden. Dies führt dazu, dass das System nahezu vollständig ($\approx 98\,\%$) mit der Garbage-Collection ausgelastet ist und dass hierbei nur wenig ($\approx 2\,\%$) Speicher freigeräumt werden kann [Ora].

Die Auswertung der LSH-Verfahren macht einen Nachteil des im Rahmen der Arbeit eingesetzten HLSH gegenüber dem JLSH deutlich: Das HLSH nutzt im Gegensatz zum JLSH Bitwerte anstatt Positionen zur Erzeugung der LSH-Keys. Während eine Hashfunktion beim HLSH nur zwei mögliche Werte, nämlich 0 oder 1, als Ausgabe erzeugt, kann eine Hashfunktion beim JLSH m mögliche Ausgabewerte erzeugen. Foglich entstehen bei der

Verwendung der gleichen LSH-Key-Länge Ψ unterschiedlich viele Blöcke.

Schließlich soll untersucht werden, inwieweit die Ausführungszeit durch die Flink-Job-Parallelität beeinflusst werden kann. In Abbildung 4.5 ist diesbezüglich dargestellt, wie sich die Ausführungszeiten mit steigender Flink-Job-Parallelität verbessern können. Die Ergebnisse beziehen sich dabei auf die Datenmenge mit 1.000.000 Datensätzen aus dem Datenkorpus GEN-L. Wie in Abbildung 4.5(a) ersichtlich ist, können die Laufzeiten dabei schrittweise verkürzt werden. Eine Ausnahme bilden die Verfahren 'LSH(5,4)' sowie 'LSH(5,12)' bei denen die Ausführungszeit von $\Upsilon = 4$ zu $\Upsilon = 8$ leicht zunimmt. Allerdings setzt sich dies nicht fort, sodass bei einer Flink-Job-Parallelität von $\Upsilon = 16$ kürzere Ausführungszeiten als bei $\Upsilon = 4$ erreicht werden können. Demnach ist die kurzzeitige Erhöhung wahrscheinlich eher auf schlecht gewählte LSH-Keys (Hashfunktionen) oder auf Latenzen zurückzuführen. Außerdem wurden diese beiden Verfahren für jede getestete Flink-Job-Parallelität nur zweimal ausgeführt. Für 'LSH(5,12)' erhöht sich außerdem die Laufzeit bei einer Flink-Job-Parallelität von $\Upsilon = 16$ im Vergleich zu $\Upsilon = 32$. Auch hier wird vermutet, dass die Abweichung auf die zuvor genannten Ursachen zurückzuführen ist.



(a) Ausführungszeit (b) Speedup

Abbildung 4.5: Ausführungszeitgewinne der LSH-Verfahren durch Erhöhung der Flink-Job-Parallelität für 1.000.000 Datensätze (GEN-L).

In Abbildung 4.5(b) ist dargestellt, welchen Speedup die LSH-Verfahren bei Steigerung der Flink-Job-Parallelität erreichen. Der maximal erreichbare Speedup variiert dabei je nach Verfahren zwischen 4,9 und 7,5. Den höchsten Speedup erreichen die Verfahren 'LSH(20,18)' und 'LSH(5,29)', bei denen eine Verkürzung der Ausführungszeit um mehr als einen Faktor von 7 möglich ist. Grundsätzlich wird ersichtlich, dass der Speedup ab einer Flink-Job-Parallelität von $\Upsilon = 16$ weniger stark zunimmt. Während der Ausführung der Testläufe konnte festgestellt werden, dass häufig einige der Task-Slots bereits nach wenigen Sekunden oder Minuten freigegeben wurden. Deshalb waren gegen Ende der Ausführungszeit nur noch sehr wenige $(1-3)$ Task-Slots belegt. Diese Situation führt

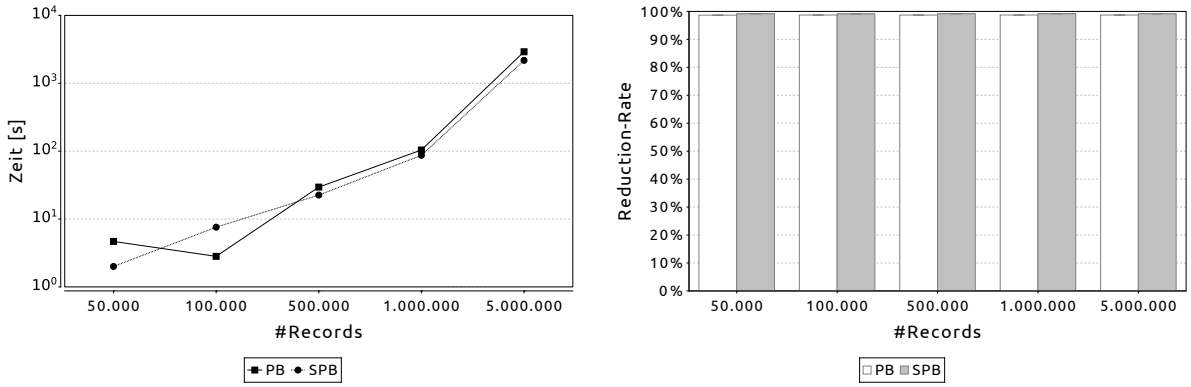
dazu, dass eine kleine Anzahl von Task-Slots die Ausführungszeit dominiert, während die Mehrzahl der Task-Slots unbelegt ist. Flink verfügt diesbezüglich aktuell über keinen Mechanismus, welcher den Workload eines Task-Slots dynamisch aufspalten und an ungenutzte Task-Slots weiterleiten kann. Hierdurch führt die weitere Erhöhung der Flink-Job-Parallelität nur noch zu relativ kleinen Zeiteinsparungen. Die Ursache dieses Problems liegt darin, dass sich die durch die LSH-Verfahren gebildeten Blöcke in ihrer Größe unterscheiden. Da in jedem der Task-Slots ein oder mehrere Blöcke abgearbeitet werden, hat dies zur Folge, dass die Task-Slots, welche nur kleine Blöcke zu verarbeiten haben, wesentlich kürzere Ausführungszeiten benötigen. Hierbei kann es beispielsweise auch vorkommen, dass in einem Block nur die Datensätze aus einer Datenquelle bzw. von einer Partei enthalten sind. Damit erzeugt dieser Block keine Kandidaten-Record-Paare, womit keine Ähnlichkeitsvergleiche für Datensätze dieses Blockes durchgeführt werden müssen. Zum anderen benötigen die Task-Slots, welche einen Block mit vielen zugeordneten Datensätzen verarbeiten müssen, eine wesentlich höhere Ausführungszeit, da mehr Kandidaten-Record-Paare erzeugt werden. Dies macht eine hohe Anzahl an Ähnlichkeitsvergleichen für diesen Block notwendig, wodurch die Bearbeitungszeit für diesen Block die gesamte Ausführungszeit bestimmen kann.

Das zuvor beschriebene Problem wird als Daten-Skew bezeichnet und ist ein typisches Problem beim verteilten und parallelen Datenmanagement [RSS15]. Da derartige Skew-Effekte die Skalierbarkeit und den Speedup von parallelen Verfahren beeinträchtigen, wurden verschiedene Ansätze zur Lastbalancierung und zur Behandlung des Daten-Skews entwickelt. Hierbei existieren bereits einige Ansätze, welche für das Blocking im Rahmen des Record-Linkages unter Verwendung von MapReduce untersucht wurden [KTR11, KTR12b]. Die Grundidee bei diesen Ansätzen ist, große Blöcke aufzuspalten, sodass deren Verarbeitung parallel erfolgen kann. Derartige Verfahren können auch für das PPRL unter Verwendung von LSH umgesetzt werden, wodurch sich die Skalierbarkeit und der Speedup der umgesetzten Verfahren weiter verbessern könnte.

Daneben könnten weitere Verfahren zur Vermeidung von Skew-Effekten im Rahmen des LSH untersucht werden. Einerseits könnten beispielsweise zu große Blöcke verworfen werden, sodass für diese keine Kandidaten-Record-Paare erzeugt werden. Da beim LSH mehrere LSH-Keys verwendet werden können, besteht für die Datensätze, welche dem zu verwerfenden Block zugeordnet sind, weiterhin die Möglichkeit ein Kandidaten-Record-Paar zu bilden. Außerdem ist in sehr großen Blöcken die Wahrscheinlichkeit hoch, dass eine Vielzahl der enthaltenen Datensätze unähnlich sind und somit nur wenige von ihnen tatsächlich übereinstimmen. Dennoch kann dieser Ansatz dazu führen, dass weniger übereinstimmende Datensätze identifiziert werden und damit die Pairs-Completeness sinkt. Eine andere Möglichkeit besteht darin, die LSH-Keys zu optimieren. Zunächst

könnten hierzu häufig gesetzte bzw. ungesetzte Bit-Positionen innerhalb der Bloom-Filter identifiziert werden. Die Hashfunktionen zur Erzeugung der LSH-Keys könnten dann so gewählt werden, dass kein Bit an einer der identifizierten Positionen zum LSH-Key beiträgt. Hierdurch wird vermieden, dass zwei Datensätze nur aufgrund einer häufig auftretenden Charakteristik als ähnlich betrachtet werden. Da die gesetzten Bits in einem Bloom-Filter ein oder mehrere Q-Gramme repräsentieren, entsprechen die häufig gesetzten Bit-Positionen häufig vorkommenden Q-Grammen. Beispielsweise tritt das Trigramm 'sch' in häufig vorkommenden deutschen Familiennamen, wie zum Beispiel 'Schmidt', 'Schulze', 'Schneider', 'Schröder' und 'Schäfer', auf. Damit weisen auch verhältnismäßig viele Bloom-Filter die gleichen gesetzten Bit-Positionen auf, welche Teil des LSH-Keys sein können. Ein anderes Beispiel hierfür ist das Trigramm 'str', welches als Teil des Wortes 'Straße' bzw. 'street' oder der Abkürzung 'Str.' bzw. 'str.' häufig in Adressangaben zu finden ist [Uni]. Wird diese Information in den Bloom-Filter aufgenommen, so führt dies dazu, dass bei sehr vielen Bloom-Filtern die gleichen Bits für diese Adressinformation gesetzt werden. Zur Umgehung dieser Probleme könnten auch bei der Erzeugung von Bloom-Filtern solche häufig vorkommende Q-Gramme entweder unberücksichtigt bleiben oder weniger stark gewichtet werden. Hierdurch würde die Wahrscheinlichkeit verringert werden, dass ein LSH-Key aus Bits zusammengesetzt wird, welche nur unzureichende Unterscheidungsmerkmale repräsentieren.

PB und SPB Im Folgenden werden die Ausführungszeiten der beiden Verfahren zum phonetischen Blocking untersucht. Dazu sind in Abbildung 4.6 die von den Verfahren PB und SPB erreichten Ausführungszeiten sowie die erzielte Reduction-Rate gegenübergestellt.



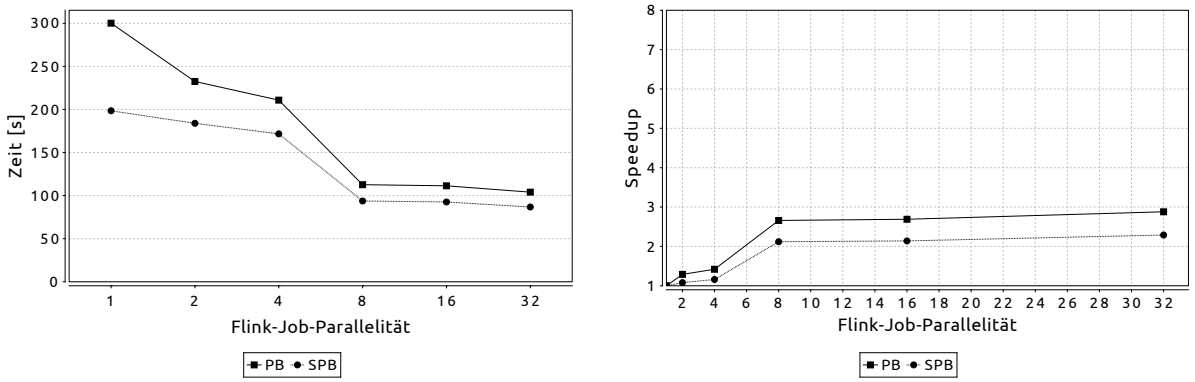
(a) Ausführungszeit (b) Reduction-Rate

Abbildung 4.6: Ausführungszeit und Reduction-Rate der Verfahren PB und SPB ($\Upsilon = 32$) für verschieden große Datenmengen aus GEN-L.

Es zeigt sich, dass beide Verfahren eine sehr hohe Reduction-Rate erreichen. Hierbei liegt die Reduction-Rate beim SPB bei 99 % und damit leicht über der Reduction-Rate des PB, welche ungefähr 98 % beträgt. Wie auch bei den LSH-Verfahren hat eine Ver-

größerung des Datenvolumens keine Auswirkungen auf die erreichten Werte. Bei Betrachtung der Aufführungszeiten lässt sich erkennen, dass das SPB grundsätzlich kürzere Ausführungszeiten erreicht. Eine Ausnahme zeigt sich bei 100.000 Records, wo stattdessen das PB eine kürzere Ausführungszeit erreicht. Es wird vermutet, dass diese Abweichung auf Latenzen zurückzuführen ist, da auch hier die Reduction-Rate beim SPB höher liegt als beim PB. Außerdem tritt diese Abweichung nicht bei den Ergebnissen für den Datenkorpus GEN-M auf. Insgesamt ist der Aufführungszeitgewinn beim SPB im Vergleich zum PB jedoch nur relativ gering. Den größten Gewinn erzielt SPB für 5.000.000 Datensätze, wo es eine Ausführungszeit von ungefähr 2175 Sekunden erreicht. Im Gegensatz dazu benötigt das PB hierfür 2913 Sekunden. Dies entspricht einem Ausführungszeitgewinn bei Verwendung des SPB um den Faktor 1,3.

Auch für die Verfahren zum phonetischen Blocking soll betrachtet werden, inwieweit die Ausführungszeiten durch die Flink-Job-Parallelität beeinflusst werden. Zu diesem Zweck sind in Abbildung 4.7(a) die Laufzeiten für verschiedene Flink-Job-Parallelitätsgrade bezogen auf die Datenmenge mit 1.000.000 Datensätzen aus dem Datenkorpus GEN-L dargestellt. Demgegenüber ist in Abbildung 4.7(b) der erreichbare Speedup dargestellt, welcher von den Verfahren PB und SPB bei Steigerung der Flink-Job-Parallelität erreicht wird.



(a) Ausführungszeit (b) Speedup

Abbildung 4.7: Ausführungszeitgewinne der Verfahren PB und SPB durch Erhöhung der Flink-Job-Parallelität für 1.000.000 Datensätze (GEN-L).

Aus den Ergebnissen wird deutlich, dass die Laufzeiten beider Verfahren durch Erhöhung der Flink-Job-Parallelität um mehr als die Hälfte reduziert werden können. Insgesamt profitiert jedoch das PB etwas mehr durch die Erhöhung der Flink-Job-Parallelität. Des Weiteren wird deutlich, dass der Ausführungszeitgewinn beim Übergang von $\Upsilon = 4$ zu $\Upsilon = 8$ am größten ist. Der genaue Grund für diese vergleichsweise starke Verkürzung der Ausführungszeit ist unklar, könnte jedoch möglicherweise auf eine bessere Verteilung der Blöcke auf die Task-Slots zurückzuführen sein. Beispielsweise könnte es sein, dass mehrere große Blöcke bis $\Upsilon = 4$ in denselben Task-Slots verarbeitet werden. Erst ab $\Upsilon = 8$ werden

diese Blöcke verschiedenen Task-Slots zugeordnet und können dadurch parallel verarbeitet werden, wodurch sich eine höhere Zeiteinsparung ergibt. Weiterhin steigt ab einer Flink-Job-Parallelität von $\Upsilon = 8$ der erreichbare Speedup nur noch geringfügig. Auch bei den phonetischen Verfahren ist die Ursache hierfür Daten-Skew. Durch die Nutzung von phonetischen Codierungen werden mehrere Werte auf denselben phonetischen Code abgebildet. Da die Namen, für welche die phonetische Codierung erzeugt wird, ebenfalls ungleichmäßig verteilt sind, führt dies zu unterschiedlich großen Blöcken. Damit wird auch hier die gesamte Ausführungszeit durch die Verarbeitung einiger Blöcke dominiert, weshalb durch die Erhöhung der Flink-Job-Parallelität nur noch wenig Zeiteinsparungen möglich sind. Auffällig hierbei ist, dass dieses Problem auch beim SPB auftritt, obwohl beim SPB zusätzlich ein Salt-Attribut zur Erzeugung des Blocking-Keys genutzt wird. Theoretisch sollte die Nutzung eines Salts dazu führen, dass der Blocking-Key von zwei Attributen abhängig ist, wodurch letztlich kleinere Blöcke entstehen. Möglicherweise wurde der Blocking-Bloom-Filter, welcher den Blocking-Key repräsentiert, zu klein gewählt, sodass dieser die Anzahl der möglichen Blöcke für das SPB begrenzt. Außerdem ist es möglich, dass mehrere phonetische Codierungen denselben Blocking-Bloom-Filter erzeugen, wodurch ebenfalls große Blöcke entstehen können. Derartige Probleme könnten durch Anpassung der Parameter des Blocking-Bloom-Filters abgemildert werden. Dazu sollte die Anzahl der genutzten Hashfunktionen vergrößert werden und die Länge des Blocking-Bloom-Filters entsprechend angepasst werden.

Verfahren	**#Records**				
	50.000	100.000	500.000	1.000.000	5.000.000
Nested-Loop	570,39 s	2453,32 s	39.162,81 s	150.015,27 s	–
LSH(5,4)	4,9 s	20,22 s	339,29 s	1732,51 s	40.430,91 s
LSH(5,29)	33,66 s	130,46 s	3467,97 s	14.206,78 s	–
LSH(5,2)	5,03 s	8,21 s	164,45 s	542,18 s	13.876,87 s
LSH(5,12)	12,68 s	44,22 s	1108,57 s	6031,23 s	150.372,91 s
LSH(20,6)	2,25 s	3,08 s	**10,85 s**	**21,71 s**	**98,9 s**
LSH(20,18)	5,73 s	7,9 s	35,13 s	63,04 s	344,87 s
PB	4,68 s	**2,82 s**	29,53 s	104,03 s	2913,07 s
SPB	**2,0 s**	7,59 s	22,53 s	86,80 s	2175,13 s

Tabelle 4.4: Laufzeiten aller getesteten Verfahren ($\Upsilon = 32$) für unterschiedlich große Datenmengen aus GEN-L.

Gesamtergebnis Die Analyse der umgesetzten Verfahren zeigt, dass die Verwendung von LSH zu kürzeren Ausführungszeiten und einem höheren Speedup führen kann. Hierzu ist jedoch die Wahl der Parameter entscheidend: Während die getesteten LSH-Verfahren

mit $\Psi = 20$ erheblich kürzere Ausführungszeiten für große Datenmengen im Vergleich zu den Verfahren PB und SPB erreichen, benötigen die LSH-Verfahren mit $\Psi = 5$ wesentlich mehr Zeit. Die Verfahren zum phonetischen Blocking erreichen nur für die Datenmengen mit 50.000 und 100.000 Datensätzen kürzere Ausführungszeiten als die Verfahren 'LSH(20,6)' bzw. 'LSH(20,18)'. Die insgesamt kürzesten Laufzeiten werden ab einer Anzahl von 500.000 Records von 'LSH(20,6)' erreicht. Dieses Verfahren benötigt für die größte getestete Datenmenge mit insgesamt 5.000.000 Datensätzen nur ungefähr 1,5 Minuten. Für 1.000.000 Datensätze benötigt 'LSH(20,6)' nur rund 22 Sekunden, womit es um einen Faktor von etwa 6910 schneller ist als das Nested-Loop-Verfahren. Im Vergleich zum SPB zeigt sich, dass SPB für 1.000.000 Datensätze um ca. einen Faktor von 4 langsamer ist als 'LSH(20,6)'. Für 5.000.000 Datensätze zeigt sich der Unterschied noch deutlicher: Hier ist das SPB um etwa einen Faktor von 22 langsamer als 'LSH(20,6)'. Wie zu erwarten war, werden vom Nested-Loop-Verfahren stets die längsten Ausführungszeiten benötigt. Trotz der Parallelisierung bleibt das Verfahren praktisch nicht einsetzbar.

Das grundlegende Problem, welches sowohl bei den LSH-Verfahen als auch bei den Verfahren zum phonetischen Blocking auftritt, sind Skew-Effekte, die durch verschieden große Blöcke entstehen. Diese haben zur Folge, dass die Skalierbarkeit und der erreichbare Speedup beeinträchtigt werden. Um weitere Zeiteinsparungen zu ermöglichen, ist daher eine Behandlung des Daten-Skews unerlässlich. Die Übertragung und Nutzung vorhandener Verfahren zur Lastbalancierung stellt daher eine sinnvolle Erweiterung für die umgesetzten Verfahren dar.

4.5 Fazit

Die Auswertung der von den umgesetzten Verfahren erzielten Ergebnisse zeigt, dass mit Hilfe von LSH ein besserer Trade-off zwischen der Linkage-Qualität und der Performanz bzw. der Skalierbarkeit erreicht werden kann als beim phonetischen Blocking. Hierbei wird deutlich, dass die Wahl der LSH-Parameter großen Einfluss auf die Ergebnisse hat: Selbst geringfügige Änderungen der beiden Parameter Ψ und Λ können zu sehr unterschiedlichen Resultaten führen. Diesbezüglich wurde deutlich, dass die Länge Ψ der LSH-Keys unter Nutzung des HLSH nicht zu klein gewählt werden darf, da ansonsten zu wenig Blöcke gebildet werden. Dies führt dazu, dass bei steigenden Datenvolumen sehr viele Datensätze demselben Block zugeordnet werden, wodurch der Suchraum nicht stark genug eingeschränkt werden kann.

Für die verwendeten Datenmengen konnte das LSH-Verfahren mit $\Psi = 20$ und $\Lambda = 18$ den besten Trade-off hinsichtlich der Ausführungszeit und der Pairs-Completeness errei-

chen. Das Verfahren erzielt auch für die stärker modifizierten Daten aus dem Datenkorpus GEN-M eine Pairs-Completeness von über 90 %. Gleichzeitig ist die Ausführung von 'LSH(20,18)' um mehr als einen Faktor von 8 schneller als die des PB bei einem Datenvolumen von 5.000.000 Records. Zwar benötigt 'LSH(20,6)' wiederum nur etwa $\frac{1}{3}$ der Ausführungszeit von 'LSH(20,18)', jedoch erreicht 'LSH(20,6)' nur noch eine Pairs-Completeness von etwa 75 % für die Datenmengen aus GEN-M.

Die beiden Verfahren zum phonetischen Blocking erreichen kürzere Laufzeiten als die LSH-Verfahren mit $\Psi = 5$. Durch die Nutzung des SPB kann im Vergleich zum PB weniger Zeit als erwartet eingespart werden. Bei der Untersuchung der Pairs-Completeness wird jedoch deutlich, dass die Pairs-Completeness-Werte beim SPB ungefähr um $10 - 15\,\%$ niedriger sind als bei Verwendung des PB. Damit wird der Vorteil der etwas kürzeren Ausführungszeit des SPB ausgeglichen.

Grundsätzlich zeigt die Auswertung, dass sowohl die LSH-Verfahren als auch die Verfahren zum phonetischen Blocking von einer Parallelisierung profitieren. Allerdings konnten die LSH-Verfahren einen wesentlich höheren Speedup erreichen. Durch vollständige Ausnutzung der Rechenleistung des Computer-Clusters ist es möglich, dass unter Verwendung von LSH das PPRL auf mehreren Millionen Datensätzen in wenigen Minuten durchführbar ist. Hierbei können auch fehlerhafte Daten kompensiert werden, sodass eine gute Linkage-Qualität mit einer hohen Pairs-Completeness erreicht werden kann.

Kapitel 5

Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es, die Skalierbarkeit und die Performanz des Privacy-preserving-Record-Linkages (PPRL) zu verbessern, indem parallele Verfahren zur Durchführung des Linkage-Prozesses untersucht, umgesetzt und evaluiert werden. Da die Leistungsfähigkeit des PPRL wesentlich von den eingesetzten Blocking- bzw. Filter-Verfahren abhängig ist, lag auch der Schwerpunkt der Arbeit in der Anpassung und Umsetzung verschiedener Blocking-Verfahren für das Parallel-Privacy-preserving-Record-Linkage (P3RL). Im Fokus stand hierbei die Untersuchung und Implementierung des Locality-sensitive-Hashings (LSH), welches bisher vielversprechende Ergebnisse liefern konnte [Dur12, KV15, KV16]. Zum Vergleich wurden daneben Verfahren zum phonetischen Blocking betrachtet und umgesetzt. Die Realisierung der Verfahren erfolgte mit Hilfe von Apache Flink, welches als Cluster-Computing-Framework die Entwicklung und Ausführung von parallelen Programmen vereinfacht. Im Gegensatz zu Hadoop MapReduce bietet Flink einen größeren Funktionsumfang mit zahlreichen vordefinierten Operationen.

Im Rahmen der Arbeit wurden zunächst die Grundlagen des PPRL betrachtet. Zudem wurde beschrieben, wie Bloom-Filter als Privacy-Technik für das PPRL genutzt werden und die Wahl ihrer Parameter erfolgen kann. Außerdem wurden die Grundlagen der umgesetzten Blocking-Verfahren erläutert und es wurde auf die wesentlichen Konzepte und die Funktionsweise von Flink eingegangen.

Zur Realisierung des P3RL wurde im Rahmen der Arbeit ein Framework entwickelt, welches die Ausführung und den Vergleich der umgesetzten Verfahren vereinfacht. Während der Ausführung werden dazu verschiedene Metriken gesammelt, welche zur Erstellung von Diagrammen weiterverarbeitet werden. Die gesammelten Metriken und die Diagramme können dann zur Evaluierung der Verfahren genutzt werden. Außerdem wurde das Framework so realisiert, dass vorhandene Komponenten wiederverwendbar sind, sodass das Framework möglichst einfach um weitere Verfahren erweitert werden kann.

Die Evaluierung der umgesetzten Verfahren erfolgte auf verschiedenen Datenmengen, welche unterschiedliche Charakteristiken hinsichtlich der Daten-Qualität und der Anzahl der enthaltenen Datensätze aufweisen. Im Vordergrund der Evaluierung stand, wie sich verschieden gewählte LSH-Parameter auf die Skalierbarkeit, den Speedup und die Linkage-Qualität auswirken. Es wurde festgestellt, dass sowohl durch LSH als auch durch phonetisches Blocking die Ausführungszeiten im Vergleich zum vollständigen paarweisen Vergleich drastisch reduziert werden können. Grundsätzlich konnten unter Verwendung von LSH kürzere Ausführungszeiten sowie bessere Linkage-Ergebnisse erzielt werden als beim phonetischen Blocking. Es wurde deutlich, dass bereits kleine Änderungen in den LSH-Parametern große Auswirkungen hinsichtlich der benötigten Ausführungszeit und der erreichten Linkage-Qualität haben. Weiterhin konnten durch Steigerung der Parallelität, mit welcher ein Flink-Programm ausgeführt wird, die Ausführungszeiten um einen Faktor von bis zu ca. 7,5 verringert werden. Hierbei konnten die Verfahren zum phonetischen Blocking insgesamt weniger stark von der Erhöhung der Parallelität profitieren. Es wurde erkannt, dass sowohl beim LSH als auch beim phonetischen Blocking Skew-Effekte dazu führen, dass der erreichbare Speedup vermindert wird. Die Ursache hierfür sind unterschiedlich große Blöcke, welche durch die Blocking-Verfahren gebildet werden. Dies führt dazu, dass einige Blöcke die gesamte Ausführungszeit dominieren, sodass trotz Erhöhung der Parallelität nur noch geringe Zeitgewinne möglich sind. Dennoch zeigte die Evaluierung, dass durch Parallelisierung des PPRL-Prozesses und durch Wahl geeigneter Parameter für die Blocking-Verfahren die Skalierbarkeit des PPRL verbessert und gleichzeitig eine hohe Linkage-Qualität erzielt werden kann.

Ausblick Das im Rahmen dieser Arbeit entwickelte Framework zur Durchführung des P3RL bietet verschiedene Erweiterungsmöglichkeiten. Zum einen können die vorhandenen Verfahren erweitert werden, indem Mechanismen zur Behandlung von Daten-Skew umgesetzt werden. Hierfür existieren bereits einige vielversprechende Ansätze, welche für das Blocking im Rahmen des Record-Linkages und unter Verwendung von MapReduce untersucht wurden [KTR11, KTR12b]. Zum anderen können noch weitere Blocking-Verfahren umgesetzt und mit den bereits vorhandenen Verfahren verglichen werden. Besonders von Interesse erscheint hierbei die Parallelisierung weiterer LSH-basierter Verfahren, wie beispielsweise in [KV16] vorgestellt. Des Weiteren können die Analysemöglichkeiten erweitert werden, indem weitere von Flink bereitgestellte Metriken berücksichtigt werden. Beispielsweise könnte die genaue Speicher- und CPU-Auslastung oder die für die Garbage-Collection aufgewendete Zeit für verschiedene Verfahren analysiert werden. Weiterhin wäre es wünschenswert, dass weitere Testläufe für die umgesetzten Verfahren durchgeführt werden. Einerseits könnten noch größere Datenmengen mit mehr als 5.000.000 Datensätzen berücksichtigt werden, da solche Datenmengen in der Realität ebenfalls auftreten. Andererseits könnte ein größeres Computer-Cluster genutzt werden, um eine erwei-

terte Analyse des Speedups zu ermöglichen. Außerdem ist es sinnvoll zu analysieren, wie sich Änderungen der verschiedenen Bloom-Filter-Parameter auf die eingesetzten Blocking-Verfahren, insbesondere auf das LSH, auswirken.

Abschließend gibt es im gesamten Bereich des PPRL noch zahlreiche offene Probleme und Untersuchungsmöglichkeiten [VCV13, VSCR17]. Besonders die Durchführung des Linkages zwischen mehr als zwei Parteien stellt eine große Herausforderung beim PPRL dar [VSCR17]. Hierbei müssen wesentlich mehr Datensätze sowie Record-Paare berücksichtigt werden, wodurch die vorhandenen Blocking-Verfahren noch stärker gefordert sind. Auch zur Realisierung derartiger Anwendungsszenarien ist es unerlässlich, dass das PPRL parallel auf einem hochperformanten Computer-Cluster unter Einsatz von sehr effizienten Blocking- oder Filter-Verfahren ausgeführt wird.

Abkürzungsverzeichnis

API . . . . . Application-Programming-Interface

CLK . . . . . Cryptographic-Longterm-Key

CSV . . . . . Character-separated-Values

DAG . . . . . Directed-acyclic-Graph

FBF . . . . . Field-Level-Bloom-Filter

HDFS . . . . . Hadoop-distributed-File-System

HLSH . . . . . Hamming-LSH

HMAC . . . . . Keyed-Hash-Message-Authentication-Code

JLSH . . . . . Jaccard-LSH

JM . . . . . Job-Manager

JSON . . . . . JavaScript-Object-Notation

JVM . . . . . Java-virtual-Machine

LSH . . . . . Locality-sensitive-Hashing

NCVR . . . . . North-Carolina-Voter-Registration

P3RL . . . . . Parallel-Privacy-preserving-Record-Linkage

PB . . . . . Phonetic-Blocking

POJO . . . . . Plain-old-Java-Object

PPRL . . . . . Privacy-preserving-Record-Linkage

QID . . . . . Quasi-Identifikator

RBF . . . . . Record-Level-Bloom-Filter

REST . . . . . Representational-State-Transfer

RL . . . . . Record-Linkage

SPB . . . . . Salted-phonetic-Blocking

TM . . . . . Task-Manager

TS . . . . . Task-Slot

Symbolverzeichnis

B . . . . . Block

b . . . . . Anzahl Blöcke

BK . . . . . Blocking-Key

Bf . . . . . Bloom-Filter

C . . . . . Menge aller Record-Paare (bzw. -Mengen)

C_{Kand} . . . . . Menge aller Kandidaten-Record-Paare (bzw. -Mengen)

D . . . . . Datenbank

$\tilde{D}$ . . . . . Datenbank mit maskierten Records

δ . . . . . Wahrscheinlichkeit eines False-non-Matches (LSH)

d . . . . . Distanzmaß

E . . . . . Menge zu repräsentierender Elemente (Bloom-Filter)

e . . . . . Element aus E

$\bar{E}$ . . . . . Menge aller Q-Gramme

ε . . . . . Entscheidungsmodell

η . . . . . Anzahl zu nutzender QID-Attribute

$\mathcal{F}$ . . . . . Funktionsfamilie

FM . . . . . F-Measure

FN . . . . . Menge der False-non-Matches

FP . . . . . Menge der False-Matches

fpr . . . . . Falsch-positiv-Rate

Γ . . . . . Durchschnittliche Länge eines Attributs

H . . . . . Hashfunktion (Bloom-Filter)

k . . . . . Anzahl Hashfunktionen (Bloom-Filter)

Λ . . . . . Anzahl der LSH-Keys

M . . . . . Menge der klassifizierten Matches

m . . . . . Länge Bloom-Filter

$\hat{M}$ . . . . . Menge der wahren Matches

μ . . . . . Anteil der 0-Bits im Bloom-Filter

n . . . . . Anzahl Records (Partei)

$\bar{n}$ . . . . . Anzahl Records aller Parteien

$\mathring{n}$ . . . . . Durchschnittliche Anzahl Records einer Partei

ω . . . . . Anzahl Elemente (Bloom-Filter)

P . . . . . Partei

p . . . . . Anzahl Parteien

PC . . . . . Pairs-Completeness

π . . . . . Permutation

PQ . . . . . Pairs-Quality

Ψ . . . . . Länge der LSH-Keys

Q . . . . . Länge Q der Q-Gramme

R . . . . . Menge aller Records

r . . . . . Record

$\tilde{r}$ . . . . . Maskierter Record

RR . . . . . Reduction-Rate

t . . . . . Ähnlichkeitsschwellwert

τ . . . . . Ähnlichkeitsschwellwert (LSH)

TN . . . . . Menge der True-non-Matches

TP . . . . . Menge der True-Matches

U . . . . . Menge der klassifizierten Non-Matches

$\hat{U}$ . . . . . Menge der wahren Non-Matches

Υ . . . . . Flink-Job-Parallelität

ξ . . . . . Phonetischer Code

Z . . . . . Menge von Elementen (LSH)

ζ . . . . . Phonetischer Code (Salt)

Abbildungsverzeichnis

1.1 Herausforderungen beim PPRL. 3

2.1 Grundlegender PPRL-Prozess für zwei Parteien. 11
2.2 Beispiel: Erstellung eines Bloom-Filters beim PPRL. 30
2.3 Beispiel: Streaming-Dataflow-Graph eines einfachen Flink-Programms. . . . 41
2.4 Beispiel: Paralleler Streaming-Dataflow-Graph eines einfachen Flink-Programms. 43
2.5 Beispiel: Zuordnung von Subtasks zu Task-Slots. 44
2.6 Architektur der Flink-Laufzeitumgebung. 45

3.1 Beispiel: Bildung des Blocking-Keys beim Phonetic-Blocking bzw. Salted-phonetic-Blocking. 56
3.2 Architektur des entwickelten Frameworks für das P3RL mit Flink. 57
3.3 Beispiel: FlatMap-Operation zur Bildung der LSH-Keys. 62
3.4 Beispiel: Gruppierung der Datensätze anhand der LSH-Keys mit anschließender Bildung der Kandidaten-Record-Paare. 63
3.5 Beispiel: Filter-Operation zur Entfernung von doppelten Kandidaten-Record-Paaren. 64
3.6 Beispiel: Phonetisches Blocking mit Flink. 66

4.1 Linkage-Qualität für das Nested-Loop-Verfahren. 78
4.2 Linkage-Qualität der verschiedenen LSH-Verfahren für die Datenmengen mit 1.000.000 (GEN-L, GEN-M) bzw. 1.097.720 (NCVR) Datensätzen. . . 80
4.3 Linkage-Qualität der Verfahren PB und SPB für die Datenmengen mit 1.000.000 (GEN-L, GEN-M) bzw. 1.097.720 (NCVR) Datensätzen. 82
4.4 Ausführungszeit und Reduction-Rate der verschiedenen LSH-Verfahren ($\Upsilon = 32$) für unterschiedlich große Datenmengen aus GEN-L. 84
4.5 Ausführungszeitgewinne der LSH-Verfahren durch Erhöhung der Flink-Job-Parallelität für 1.000.000 Datensätze (GEN-L). 86

4.6 Ausführungszeit und Reduction-Rate der Verfahren PB und SPB ($\Upsilon = 32$) für verschieden große Datenmengen aus GEN-L. 88
4.7 Ausführungszeitgewinne der Verfahren PB und SPB durch Erhöhung der Flink-Job-Parallelität für 1.000.000 Datensätze (GEN-L). 89

Tabellenverzeichnis

2.1 Beispiel: Datenbank D_A der Partei A (Mobilfunkanbieter). 10
2.2 Beispiel: Datenbank D_B der Partei B (Kfz-Versicherung). 10
2.3 Konfusionsmatrix bezüglich der Klassifikation von Record-Paaren. 19
2.4 Protokolltypen von PPRL-Verfahren. 22
2.5 Soundex Regeltabelle. 38
2.6 Auswahl einiger Transformationen der DataSet-API. 47
2.7 Flink-Metriken. 49

4.1 Beispiel: Synthetische Daten. 70
4.2 Abschätzung der Feldlängen der Attribute und der Anzahl der Q-Gramme für die genutzten Testdatensätze. 73
4.3 Charakteristiken der verwendeten Datenkorpora und gewählte Parameter. . 75
4.4 Laufzeiten aller getesteten Verfahren ($\Upsilon = 32$) für unterschiedlich große Datenmengen aus GEN-L. 90

Literaturverzeichnis

[AI08] Andoni, Alexandr und Piotr Indyk: *Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions.* Communications of the ACM, 51(1):117–122, Januar 2008.

[All] Allianz Deutschland AG: *Zahlen, Daten und Fakten zur Allianz Deutschland AG.* `https://www.allianzdeutschland.de/zahlen-daten-und-fakten-zur-allianz-deutschland-ag-/id_73439330/index`. [Zugriff: 29.03.2017].

[BBR11] Bellahsene, Zohra, Angela Bonifati und Erhard Rahm: *Schema Matching and Mapping.* Springer, 2011.

[BCC+03] Baxter, Rohan, Peter Christen, Tim Churches et al.: *A Comparison of Fast Blocking Methods for Record Linkage.* In: *ACM SIGKDD*, Band 3, Seiten 25–27, 2003.

[Blo70] Bloom, Burton: *Space/Time Trade-offs in Hash Coding with Allowable Errors.* Communications of the ACM, 13(7):422–426, 1970.

[BM04] Broder, Andrei und Michael Mitzenmacher: *Network Applications of Bloom Filters: A survey.* Internet Mathematics, 1(4):485–509, 2004.

[BMSB09] Barone, Daniele, Andrea Maurino, Fabio Stella und Carlo Batini: *A Privacy-Preserving Framework for Accuracy and Completeness Quality Assessment.* Emerging Paradigms in Informatics, Systems and Communication, 83, 2009.

[Bro97] Broder, Andrei Z.: *On the resemblance and containment of documents.* In: *Compression and Complexity of Sequences 1997. Proceedings*, Seiten 21–29. IEEE, 1997.

[Bru96] Bruce, Schneier: *Applied Cryptography: Protocols, Algorithms, and Source code in C.* John Wiley & Sons, Inc., New York, 1996.

[BS92] Borgman, Christine L. und Susan L. Siegfried: *Getty's synoname TM and its cousins: A survey of applications of personal name-matching algorithms.* Journal of the American Society for Information Science, 43(7):459, 1992.

[BS06] Batini, Carlo und Monica Scannapieco: *Data Quality: Concepts, Methodologies and Techniques.* Springer-Verlag, 2006.

[Buc12] Buchmann, Johannes: *Internet Privacy: Eine multidisziplinäre Bestandsaufnahme / A multidisciplinary analysis.* Springer-Verlag, 2012.

[Bun] Bundesministerium für Gesundheit: *Gesetzliche Krankenversicherung: Mitglieder, mitversicherte Angehörige und Krankenstand. Monatswerte Januar 2017.* `http://www.bundesgesundheitsministerium.de/fileadmin/Dateien/3_Downloads/Statistiken/GKV/Mitglieder_Versicherte/KM1_Januar_2017.pdf`. [Zugriff: 29.03.2017].

[BYRN99] Baeza-Yates, Ricardo A. und Berthier Ribeiro-Neto: *Modern Information Retrieval.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[CC02] Christen, Peter und Tim Churches: *Febrl - Freely extensible biomedical record linkage.* Australian National University, Department of Computer Science, 2002.

[CC04] Churches, Tim und Peter Christen: *Some methods for blindfolded record linkage.* BMC Medical Informatics and Decision Making, 4(1):9, 2004.

[CG07] Christen, Peter und Karl Goiser: *Quality and Complexity Measures for Data Linkage and Deduplication.* In: *Quality Measures in Data Mining*, Seiten 127–151. Springer, 2007.

[Chr05] Christen, Peter: *Probabilistic Data Generation for Deduplication and Data Linkage.* In: *International Conference on Intelligent Data Engineering and Automated Learning*, Seiten 109–116. Springer, 2005.

[Chr06] Christen, Peter: *A Comparison of Personal Name Matching: Techniques and Practical Issues.* In: *Data Mining Workshops, 2006. ICDM Workshops 2006. Sixth IEEE International Conference on*, Seiten 290–294. IEEE, 2006.

[Chr08] Christen, Peter: *Febrl – A Freely Available Record Linkage System with a Graphical User Interface.* In: *Proceedings of the second Australasian workshop on Health data and knowledge management*, Band 80, Seiten 17–25. Australian Computer Society, Inc., 2008.

[Chr09] Christen, Peter: *Development and User Experiences of an Open Source Data Cleaning, Deduplication and Record Linkage System.* ACM SIGKDD Explorations Newsletter, 11(1):39–48, 2009.

[Chr12a] Christen, Peter: *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer Science & Business Media, 2012.

[Chr12b] Christen, Peter: *A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication.* IEEE transactions on knowledge and data engineering, 24(9):1537–1555, 2012.

[Com] Comenetz, Joshua: *Frequently Occurring Surnames from the 2010 Census.* `https://www2.census.gov/topics/genealogy/2010surnames/surnames.pdf`. [Zugriff: 06.04.2017].

[CP09] Christen, Peter und Agus Pudjijono: *Accurate synthetic generation of realistic personal information.* In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Seiten 507–514. Springer, 2009.

[CV13] Christen, Peter und Dinusha Vatsalan: *Flexible and extensible generation and corruption of personal data.* In: *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, Seiten 1165–1168. ACM, 2013.

[DBGH11] Dal Bianco, Guilherme, Renata Galante und Carlos A. Heuser: *A fast approach for parallel deduplication on multicore processors.* In: *Proceedings of the 2011 ACM Symposium on Applied Computing*, Seiten 1027–1032. ACM, 2011.

[Deu] Deutsche Telekom AG: *Deutsche Telekom: Unternehmenspräsentation.* `https://www.telekom.com/resource/blob/312750/310f6a9efe02666460f8ebbdc26aabe7/dl-unternehmenspraesentation-data.pdf`. [Zugriff: 29.03.2017].

[DGDLM13] Demchenko, Yuri, Paola Grosso, Cees De Laat und Peter Membrey: *Addressing Big Data Issues in Scientific Data Infrastructure.* In: *Collaboration Technologies and Systems (CTS), 2013 International Conference on*, Seiten 48–55. IEEE, 2013.

[Dic45] Dice, Lee R.: *Measures of the Amount of Ecologic Association Between Species.* Ecology, 26(3):297–302, 1945.

[DKX+14] Durham, Elizabeth A., Murat Kantarcioglu, Yuan Xue, Csaba Toth, Mehmet Kuzu und Bradley Malin: *Composite Bloom Filters for Secure Record Linkage*. IEEE transactions on knowledge and data engineering, 26(12):2956–2968, 2014.

[DQB95] Dusserre, L., C. Quantin und H. Bouzelat: *A One Way Public Key Cryptosystem for the Linkage of Nominal Files in Epidemiological Studies*. Medinfo, 8:644–647, 1995.

[Dun46] Dunn, Halbert L: *Record linkage*. American Journal of Public Health and the Nations Health, 36(12):1412–1416, 1946.

[Dur12] Durham, Elizabeth Ashley: *A framework for accurate, efficient private record linkage*. Doktorarbeit, Vanderbilt University, 2012.

[EIV07] Elmagarmid, Ahmed K., Panagiotis G. Ipeirotis und Vassilios S. Verykios: *Duplicate Record Detection: A survey*. IEEE Transactions on knowledge and data engineering, 19(1), 2007.

[EVE02] Elfeky, Mohamed G., Vassilios S. Verykios und Ahmed K. Elmagarmid: *TAILOR: A Record Linkage Toolbox*. In: *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, Seiten 17–28. IEEE, 2002.

[Faw04] Fawcett, Tom: *ROC Graphs: Notes and Practical Considerations for Researchers*. Machine learning, 31(1):1–38, 2004.

[Fie05] Fienberg, Stephen E.: *Confidentiality and Disclosure Limitation*. Encyclopedia of Social Measurement, 1:463–69, 2005.

[FPS+13] Forchhammer, Benedikt, Thorsten Papenbrock, Thomas Stening, Sven Viehmeier, Uwe Draisbach und Felix Naumann: *Duplicate Detection on GPUs*. HPI Future SOC Lab: proceedings 2011, 70:59, 2013.

[FS69] Fellegi, Ivan P. und Alan B. Sunter: *A Theory for Record Linkage*. Journal of the American Statistical Association, 64(328):1183–1210, 1969.

[GIM+99] Gionis, Aristides, Piotr Indyk, Rajeev Motwani et al.: *Similarity Search in High Dimensions via Hashing*. In: *Proceedings of the 25th VLDB Conference*, Band 99, Seiten 518–529, 1999.

[GKS08] Ganta, Srivatsava Ranjit, Shiva Prasad Kasiviswanathan und Adam Smith: *Composition Attacks and Auxiliary Information in Data Pri-*

vacy. In: *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, Seiten 265–273. ACM, 2008.

[HF10] Hall, Rob und Stephen E. Fienberg: *Privacy-Preserving Record Linkage.* In: *International Conference on Privacy in Statistical Databases*, Seiten 269–283. Springer, 2010.

[HPIM12] Har-Peled, Sariel, Piotr Indyk und Rajeev Motwani: *Approximate Nearest Neighbor: Towards Removing the Curse of Dimensionality.* Theory of computing, 8(1):321–350, 2012.

[HS98] Hernández, Mauricio A. und Salvatore J. Stolfo: *Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem.* Data Mining and Knowledge Discovery, 2(1):9–37, 1998.

[HSW07] Herzog, Thomas N., Fritz J. Scheuren und William E. Winkler: *Data Quality and Record Linkage Techniques.* Springer Publishing Company, Incorporated, 1st Auflage, 2007.

[IM98] Indyk, Piotr und Rajeev Motwani: *Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality.* In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, Seiten 604–613. ACM, 1998.

[Ind04] Indyk, Piotr: *Nearest neighbors in high-dimensional spaces.* 2004.

[Jac12] Jaccard, Paul: *The distribution of the flora in the alpine zone.* New phytologist, 11(2):37–50, 1912.

[KCB97] Krawczyk, Hugo, Ran Canetti und Mihir Bellare: *HMAC: Keyed-Hashing for Message Authentication.* IETF, 1997.

[KKD+13] Kuzu, Mehmet, Murat Kantarcioglu, Elizabeth Ashley Durham, Csaba Toth und Bradley Malin: *A practical approach to achieve private medical record linkage in light of public resources.* Journal of the American Medical Informatics Association, 20(2):285–292, 2013.

[KKDM11] Kuzu, Mehmet, Murat Kantarcioglu, Elizabeth Durham und Bradley Malin: *A Constraint Satisfaction Cryptanalysis of Bloom Filters in Private Record Linkage.* In: *International Symposium on Privacy Enhancing Technologies Symposium*, Seiten 226–245. Springer, 2011.

[KKH+10] Kirsten, Toralf, Lars Kolb, Michael Hartung, Anika Gross, Hanna Köpcke und Erhard Rahm: *Data Partitioning for Parallel Entity Matching.* In: *8th International Workshop on Quality in Databases*, 2010.

[KL07] Kim, Hung-sik und Dongwon Lee: *Parallel linkage.* In: *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, Seiten 283–292. ACM, 2007.

[KL10] Kim, Hung-sik und Dongwon Lee: *HARRA: Fast Iterative Hashed Record Linkage for Large-Scale Data Collections.* In: *Proceedings of the 13th International Conference on Extending Database Technology*, Seiten 525–536. ACM, 2010.

[KR10] Köpcke, Hanna und Erhard Rahm: *Frameworks for entity matching: A comparison.* Data & Knowledge Engineering, 69(2):197–210, 2010.

[KRM+12] Kuehni, Claudia E., Corina S. Rueegg, Gisela Michel, Cornelia E. Rebholz, Marie-Pierre F. Strippoli, Felix K. Niggli, Matthias Egger, Nicolas X. von der Weid, Swiss Paediatric Oncology Group (SPOG) et al.: *Cohort Profile: The Swiss Childhood Cancer Survivor Study.* International journal of epidemiology, 41(6):1553–1564, 2012.

[KS14] Kroll, Martin und Simone Steinmetzer: *Automated Cryptanalysis of Bloom Filter Encryptions of Health Records.* 8th International Conference on Health Informatics, 2014.

[KTR11] Kolb, Lars, Andreas Thor und Erhard Rahm: *Block-based Load Balancing for Entity Resolution with MapReduce.* In: *CIKM*, Seiten 2397–2400, 2011.

[KTR12a] Kolb, Lars, Andreas Thor und Erhard Rahm: *Dedoop: Efficient Deduplication with Hadoop.* PVLDB, 5(12):1878–1881, 2012.

[KTR12b] Kolb, Lars, Andreas Thor und Erhard Rahm: *Load Balancing for MapReduce-based Entity Resolution.* In: *ICDE*, Seiten 618–629, 2012.

[Kuk92] Kukich, Karen: *Techniques for Automatically Correcting Words in Text.* ACM Computing Surveys (CSUR), 24(4):377–439, 1992.

[KV09] Karakasidis, Alexandros und Vassilios S. Verykios: *Privacy Preserving Record Linkage Using Phonetic Codes.* In: *Informatics, 2009. BCI'09. Fourth Balkan Conference in*, Seiten 101–106. IEEE, 2009.

[KV13] Karapiperis, Dimitrios und Vassilios S. Verykios: *A Distributed Framework For Scaling Up LSH-Based Computations in Privacy Preserving Record Linkage.* In: *Proceedings of the 6th Balkan Conference in Informatics*, Seiten 102–109. ACM, 2013.

[KV14] Karapiperis, Dimitrios und Vassilios S. Verykios: *A Distributed Near-Optimal LSH-based Framework for Privacy-Preserving Record Linkage*. Computer Science and Information Systems, 11(2):745–763, 2014.

[KV15] Karapiperis, Dimitrios und Vassilios S. Verykios: *An LSH-Based Blocking Approach with a Homomorphic Matching Technique for Privacy-Preserving Record Linkage*. IEEE Transactions on Knowledge and Data Engineering, 27(4):909–921, 2015.

[KV16] Karapiperis, Dimitrios und Vassilios S. Verykios: *A fast and efficient Hamming LSH-based scheme for accurate linkage*. Knowledge and Information Systems, 49(3):861–884, 2016.

[KVC12] Karakasidis, Alexandros, Vassilios S. Verykios und Peter Christen: *Fake Injection Strategies for Private Phonetic Matching*. In: *Data Privacy Management and Autonomous Spontaneus Security*, Seiten 9–24. Springer, 2012.

[LP07] Lindell, Yehuda und Benny Pinkas: *An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries*. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Seiten 52–78. Springer, 2007.

[LP09] Lindell, Yehuda und Benny Pinkas: *Secure Multiparty Computation for Privacy-Preserving Data Mining*. Journal of Privacy and Confidentiality, 1(1):59–98, 2009.

[LRU14] Leskovec, Jure, Anand Rajaraman und Jeffrey David Ullman: *Mining of massive datasets*. Cambridge University Press, 2014.

[LYC+06] Lai, P. K. Y., S. M. Yiu, K. P. Chow, C. F. Yong und L. C. K. Hui: *An Efficient Bloom Filter Based Solution for Multiparty Private Matching*. In: *Security and Management*, Seiten 286–292, 2006.

[MU05] Mitzenmacher, Michael und Eli Upfal: *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[MX07] Motwani, Rajeev und Ying Xu: *Efficient Algorithms for Masking and Finding Quasi-Identifiers*. In: *Proceedings of the Conference on Very Large Data Bases (VLDB)*, Seiten 83–93, 2007.

[New67] Newcombe, Howard B.: *Record Linking: The Design of Efficient Systems for Linking Records into Individual and Family Histories.* American Journal of Human Genetics, 19(3 Pt 1):335, 1967.

[NK62] Newcombe, Howard B. und James M. Kennedy: *Record Linkage: Making Maximum Use of the Discriminating Power of Identifying Information.* Commun. ACM, 5(11):563–566, 1962.

[NKAJ59] Newcombe, Howard B., James M. Kennedy, S. J. Axford und Allison P. James: *Automatic Linkage of Vital Records.* Science, 130(3381):954–959, 1959.

[NKH+13] Ngomo, Axel-Cyrille Ngonga, Lars Kolb, Norman Heino, Michael Hartung, Sören Auer und Erhard Rahm: *When to Reach for the Cloud: Using Parallel Hardware for Link Discovery.* In: *Extended Semantic Web Conference*, Seiten 275–289. Springer, 2013.

[NSKS14] Niedermeyer, Frank, Simone Steinmetzer, Martin Kroll und Rainer Schnell: *Cryptanalysis of Basic Bloom Filters Used for Privacy Preserving Record Linkage.* Journal of Privacy and Condentiality, 6(2):59–79, 2014.

[OR18] Odell, M und R Russell: *The soundex coding system.* US Patents, 1261167, 1918.

[Ora] Oracle Corporation: *Understand the OutOfMemoryError Exception.* `https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/memleaks002.html`. [Zugriff: 16.04.2017].

[Phi00] Philips, Lawrence: *The double metaphone search algorithm.* C/C++ users journal, 18(6):38–43, 2000.

[Pos69] Postel, Hans Joachim: *Die Kölner Phonetik. Ein Verfahren zur Identifizierung von Personennamen auf der Grundlage der Gestaltanalyse.* IBM-Nachrichten, 19:925–931, 1969.

[QBD96] Quantin, C., H. Bouzelat und L. Dusserre: *Irreversible encryption method by generation of polynomials.* Medical Informatics, 21(2):113–121, 1996.

[RD00] Rahm, Erhard und Hong Hai Do: *Data Cleaning: Problems and Current Approaches.* Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 23(4):3–13, 2000.

[RFB+14] Randall, Sean M., Anna M. Ferrante, James H. Boyd, Jacqueline K. Bauer und James B. Semmens: *Privacy-preserving record linkage on large real world datasets.* Journal of biomedical informatics, 50:205–212, 2014.

[RFBS13] Randall, Sean M., Anna M. Ferrante, James H. Boyd und James B. Semmens: *The effect of data cleaning on record linkage quality.* BMC medical informatics and decision making, 13(1):64, 2013.

[Roc13] Rocha, Margarida Cristiana Napoleão: *Vigilância dos óbitos registrados com causa básica hanseníase: caracterização no Brasil (2004-2009) e investigação em Fortaleza, Ceará (2006-2011).* 2013.

[RSS15] Rahm, Erhard, Gunter Saake und Kai-Uwe Sattler: *Verteiltes und Paralleles Datenmanagement.* Springer, 2015.

[SB16] Schnell, Rainer 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)*, Seiten 218–224. IEEE, 2016.

[SBB04] Schnell, Rainer, Tobias Bachteler und Stefan Bender: *A Toolbox for record linkage.* Austrian Journal of Statistics, 33(1-2):125–133, 2004.

[SBR09] Schnell, Rainer, Tobias Bachteler und Jörg Reiher: *Privacy-preserving record linkage using Bloom filters.* BMC Medical Informatics and Decision Making, 9(1):41, 2009.

[SBR11] Schnell, Rainer, Tobias Bachteler und Jörg Reiher: *A Novel Error-Tolerant Anonymous Linking Code.* German Record Linkage Center, Working Paper Series No. WP-GRLC-2011-02, 2011.

[SKB+15] Sehili, Ziad, Lars Kolb, Christian Borgs, Rainer Schnell und Erhard Rahm: *Privacy Preserving Record Linkage with PPJoin.* In: *BTW*, Seiten 85–104, 2015.

[SRB14] Schnell, R., A. Richter und C. Borgs: *Performance of different methods for privacy preserving record linkage with large scale medical data sets.* In: *Presentation at International Health Data Linkage Conference, Vancouver*, 2014.

[Thea] The Apache Software Foundation: *Apache Flink 1.2.0. Documentation: Apache Flink Documentation.* `https://ci.apache.org/projects/flink/flink-docs-release-1.2/`. [Zugriff: 07.03.2017].

[Theb] The Apache Software Foundation: *Apache Flink 1.2.0. Documentation: Basic API Concepts*. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/api_concepts.html. [Zugriff: 07.03.2017].

[Thec] The Apache Software Foundation: *Apache Flink 1.2.0. Documentation: Component Stack*. https://ci.apache.org/projects/flink/flink-docs-release-1.2/internals/components.html. [Zugriff: 07.03.2017].

[Thed] The Apache Software Foundation: *Apache Flink 1.2.0. Documentation: Configuration*. https://ci.apache.org/projects/flink/flink-docs-release-1.2/setup/config.html. [Zugriff: 07.03.2017].

[Thee] The Apache Software Foundation: *Apache Flink 1.2.0. Documentation: Dataflow Programming Model*. https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/programming-model.html. [Zugriff: 07.03.2017].

[Thef] The Apache Software Foundation: *Apache Flink 1.2.0. Documentation: Distributed Runtime Environment*. https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/runtime.html. [Zugriff: 07.03.2017].

[Theg] The Apache Software Foundation: *Apache Flink 1.2.0. Documentation: Flink DataSet API Programming Guide*. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/index.html. [Zugriff: 07.03.2017].

[Theh] The Apache Software Foundation: *Apache Flink 1.2.0. Documentation: Flink DataStream API Programming Guide*. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/datastream_api.html. [Zugriff: 07.03.2017].

[Thei] The Apache Software Foundation: *Apache Flink 1.2.0. Documentation: Google Compute Engine Setup*. https://ci.apache.org/projects/flink/flink-docs-release-1.2/setup/gce_setup.html. [Zugriff: 07.03.2017].

[Thej] The Apache Software Foundation: *Apache Flink 1.2.0. Documentation: Jobs and Scheduling*. https://ci.apache.org/projects/flink/flink-docs-release-1.2/internals/job_scheduling.html. [Zugriff: 07.03.2017].

[Thek] The Apache Software Foundation: *Apache Flink 1.2.0. Documentation: Metrics*. https://ci.apache.org/projects/flink/flink-docs-release-1.2/monitoring/metrics.html. [Zugriff: 07.03.2017].

[Thel] The Apache Software Foundation: *Apache Flink 1.2.0. Documentation: Monitoring REST API*. https://ci.apache.org/projects/flink/flink-docs-release-1.2/monitoring/rest_api.html. [Zugriff: 07.03.2017].

[Them] The Apache Software Foundation: *Apache Flink 1.2.0. Documentation: Standalone Cluster*. https://ci.apache.org/projects/flink/flink-docs-release-1.2/setup/cluster_setup.html. [Zugriff: 07.03.2017].

[Then] The Apache Software Foundation: *Apache Flink: Introduction to Apache Flink*. http://flink.apache.org/introduction.html. [Zugriff: 07.03.2017].

[Theo] The Apache Software Foundation: *Apache Flink: Scalable Stream and Batch Data Processing*. https://flink.apache.org/. [Zugriff: 07.03.2017].

[TRH+15] Traub, Jonas, Tilmann Rabl, Fabian Hueske, Till Rohrmann und Volker Markl: *Die Apache Flink Plattform zur parallelen Analyse von Datenströmen und Stapeldaten*. Proceedings of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB, 2015.

[TVC13] Tran, Khoi-Nguyen, Dinusha Vatsalan und Peter Christen: *GeCo – An online personal data Generator and Corruptor*. In: *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, Seiten 2473–2476. ACM, 2013.

[Uni] United States Census Bureau: *Most Common Street Names*. http://www.usd116.org/profdev/ahtc/lessons/PlautFel09/scans/2009_07_09/StreetNamesCensus.pdf. [Zugriff: 16.04.2017].

[Vat14] Vatsalan, Dinusha: *Scalable and Approximate Privacy-Preserving Record Linkage*. Doktorarbeit, The Australian National University, 2014.

[vBDS88] Berkel, Brigitte van und Koenraad De Smedt: *Triphone analysis: a combined method for the correction of orthographical and typographical errors*. In: *Proceedings of the second conference on Applied natural language processing*, Seiten 77–83. Association for Computational Linguistics, 1988.

[VC14] Vatsalan, Dinusha und Peter Christen: *Scalable Privacy-Preserving Record Linkage for Multiple Databases*. In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, Seiten 1795–1798, New York, NY, USA, 2014. ACM.

[VC16a] Vatsalan, Dinusha und Peter Christen: *Multi-Party Privacy-Preserving Record Linkage using Bloom Filters*. 2016.

[VC16b] Vatsalan, Dinusha und Peter Christen: *Privacy-preserving matching of similar patients*. Journal of Biomedical Informatics, 59:285 – 298, 2016.

[VCL10] Vernica, Rares, Michael J. Carey und Chen Li: *Efficient parallel set-similarity joins using MapReduce*. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, Seiten 495–506. ACM, 2010.

[VCOV14] Vatsalan, Dinusha, Peter Christen, Christine M. O'Keefe und Vassilios S. Verykios: *An Evaluation Framework for Privacy-Preserving Record Linkage*. Journal of Privacy and Confidentiality, 6(1):3, 2014.

[VCV13] Vatsalan, Dinusha, P. Christen und V. S. Verykios: *A taxonomy of privacy-preserving record linkage techniques*. Information Systems, 38(6):946–969, 2013.

[VSCR17] Vatsalan, Dinusha, Ziad Sehili, Peter Christen und Erhard Rahm: *Privacy-Preserving Record Linkage for Big Data: Current Approaches and Research Challenges*. Handbook of Big Data Technologies, 2017.

[WWL+10] Wang, Chaokun, Jianmin Wang, Xuemin Lin, Wei Wang, Haixun Wang, Hongsong Li, Wanpeng Tian, Jun Xu und Rui Li: *MapDupReducer: Detecting Near Duplicates over Massiv Datasets*. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, Seiten 1119–1122. ACM, 2010.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Weiter wird versichert, dass die elektronische Version der eingereichten Arbeit in Inhalt und Formatierung mit den gedruckten und gebundenen Exemplaren übereinstimmt. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Ort, Datum Unterschrift