Hochschule für Technik, Wirtschaft und Kultur Leipzig
Fakultät Informatik, Mathematik und Naturwissenschaften

Masterarbeit
zur Erlangung des akademischen Grades
Master of Science (M.Sc.)

Verteilte Verfahren für Privacy Preserving Record Linkage unter Verwendung von metrischen Räumen

Eingereicht von: Marcel Gladbach
Matrikel: 14 MIM
Leipzig, Mai 2017

Erstprüfer: Prof. Dr.-Ing. Thomas Kudraß, HTWK Leipzig
Zweitprüfer: Prof. Dr. Erhard Rahm, ScaDS Leipzig

ScaDS Leipzig - Competence Center for Scalable Data Services and Solutions

Hochschule für Technik, Wirtschaft und Kultur Leipzig

Marcel Gladbach:

Verteilte Verfahren für Privacy Preserving Record Linkage unter Verwendung von metrischen Räumen

Mai 2017

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich diese Arbeit selbstständig ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kenntlich gemacht.
Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden.
Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Leipzig, 23. Mai 2017

_____

Marcel Gladbach

Abstract

Privacy Preserving Record Linkage (PPRL) bezeichnet das Auffinden ähnlicher Datensätze aus zwei unterschiedlichen Datenquellen ohne Offenlegung der Daten. PPRL findet unter anderem bei der Verknüpfung sensibler Patientendaten in der medizinischen Forschung Anwendung und stellt besondere Herausforderungen an die Performance. Die Parallelisierung von PPRL-Verfahren mit Verwendung moderner Frameworks ist dabei ein offenes Forschungsthema. Mit dieser Arbeit ist ein solches paralleles Verfahren entwickelt und erfolgreich mit Apache Flink in ein lauffähiges Programm umgesetzt worden. Dabei wurde der bestehende Ansatz der metrischen Räume und Verwendung von Pivot-Elementen weiter verfolgt. Es konnten Lösungen für das parallele Finden der Pivot-Elemente und das parallele Matching gefunden werden. Hierzu wurden verschiedene Strategien implementiert, optimiert und evaluiert. Die Arbeit liefert ausführliche Analysen bezüglich Laufzeit und verschiedener Metriken zu den verwendenden Strategien und Parametern. Als Ergebnis wurden enorme Laufzeitverbesserungen gegenüber dem zugehörigen nicht verteilten Verfahren erzielt. Besonders bemerkenswert ist die hervorragende Skalierbarkeit des Verfahrens, welche trotz des zu Grunde liegenden Problems mit quadratischer Komplexität linear ausfällt.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Im Zeitalter von Big Data werden täglich Daten unfassbarer Größenordnungen in verschiedensten Datenquellen gesammelt. Um gewinnbringende Analysen auf diesen Daten durchzuführen, ist es oft notwendig mehrere Datenquellen miteinander zu verbinden. Häufig stammen die Daten von verschiedenen Organisationen und unterliegen somit keiner einheitlichen Struktur. Daher können die Daten nur über Ähnlichkeitsvergleiche anhand verschiedener Felder miteinander verknüpft werden. Dieses Verfahren wird als Record Linkage bezeichnet. Oft handelt es sich bei den zu verknüpfenden Daten allerdings um personenbezogene Information. Beispielsweise müssen zur Untersuchung des Ausbruchs von Infektionskrankheiten Patientendaten zusammen mit Reisedaten, Medikamentenkäufen und möglicherweise sogar Tierarztdaten analysiert werden. Auch eine effizientere Evaluation der Wirksamkeit und der Nebenwirkungen verschiedener Behandlungen und Arzneimittel ist so möglich. Diese sehr vertraulichen Daten bedürfen allerdings Schutz und können von den Quellen nicht offen freigegeben werden. Die persönlichen Daten wie Name und Adresse sind für die Analysen jedoch auch nicht relevant, sondern dienen nur zur Verknüpfung der Daten aus den verschiedenen Quellen.

Beim Privacy Preserving Record Linkage (PPRL) wird genau dieser Aspekt berücksichtigt. Hierbei werden personenbezogene Daten zwischen den Datenquellen nur in verschlüsselter Form ausgetauscht. Das Auffinden zusammenpassender Datensätze geschieht anhand der auf einheitliche Weise verschlüsselten Daten. Anschließend können die gewünschten Analysen mit den dann verknüpften nicht personenbezogenen Daten durchgeführt werden. Privacy Preserving Record Linkage findet Anwendung in den verschiedensten Bereichen wie Gesundheitsfürsorge und -überwachung, Verbrechensaufklärung, Demografieforschung, Marketinganalysen etc.

Privacy Preserving Record Linkage gilt als relativ junges Forschungsgebiet und bringt einige Herausforderungen mit sich, da klassische Methoden des Record Linkage nur bedingt anwendbar sind. Im Vordergrund stehen dabei unter anderem die Qualität der Ergebnisse des Linkage-Prozesses und der Datenschutz bzw. die Sicherheit der Verschlüsselungen. Ein weiterer Schwerpunkt liegt auf der Performance bzw. der Skalierbarkeit des Prozesses. Bei sehr großen Datenmengen ist es viel zu aufwendig, jedes mögliche Paar von Datensätzen aus den beiden Datenquellen miteinander zu vergleichen (Problem mit quadratischer Komplexität). Daher werden Filter- und Blocking-Verfahren eingesetzt, die es ermöglichen, bestimmte Mengen von Datensatz-Paaren vorab vom Vergleich auszuschließen. Einer dieser Ansätze ist die Ausnutzung der Eigenschaften von metrischen Räumen.[1] [2]

1.2 Zielstellung

Das Ziel dieser Arbeit ist es, aufbauend auf dem in [4] entwickelten Ansatz der Verwendung von metrischen Räumen und Pivot-Elementen, den Ablauf von PPRL zu parallelisieren. Dieses verteilte Verfahren soll die Skalierbarkeit sowie die grundlegende Performance des Prozesses verbessern. Dafür müssen für die verschiedenen Abschnitte des Ablaufs verteilte Verfahren entwickelt werden.

Bisher gibt es kaum Forschungsergebnisse für paralleles PPRL. Einige wenige Arbeiten [3] beschäftigen sich mit dem verteilten Locality Sensitive Hashing (LSH), einer anderen Blocking-Methode, in Bezug auf *Hadoop* und *MapReduce*. Die Einbeziehung modernerer Frameworks wie *Apache Flink* oder *Apache Spark* wird von den führenden Autoren auf dem Gebiet PPRL als offenes Forschungsziel angegeben [1].

Die praktische Umsetzung dieser Arbeit erfolgt daher mit dem Framework *Apache Flink*, welches weit mehr Möglichkeiten als *MapReduce* bietet und dennoch verteilt auf Hadoop-Clustern laufen kann. In Zusammenhang mit dem vielversprechenden Ansatz der metrischen Räume können hier Laufzeitverbesserungen erwartet werden. Dazu werden in dieser Arbeit unterschiedliche Strategien implementiert und getestet.

Eine abschließende Evaluation auf einem hoch parallelisierten Cluster soll die in dieser Arbeit erreichten Ergebnisse aufzeigen und Aufschluss über mögliche weitere Forschungsansätze geben. Dabei sollen die Unterschiede der verschiedenen Strategien herausgestellt werden. Im Mittelpunkt steht die Optimierung der Laufzeit des Gesamtprozesses unter Einbeziehung unterschiedlicher Metriken.

1.3 Aufbau der Arbeit

Einführend werden in Kapitel 2 die grundlegenden Konzepte von PPRL und metrischer Räumen erläutert. Außerdem wird der in [4] entwickelte Ansatz beschrieben und ein Überblick über das Framework Apache Flink gegeben. Kapitel 3 stellt anschließend die in dieser Arbeit entwickelten verteilten Verfahren vor und beschreibt deren Umsetzung mit Flink. Die erfassten Evaluationsergebnisse werden in Kapitel 4 ausgewertet. Abschließend fasst Kapitel 5 die Ergebnisse der Arbeit zusammen.

Alle verwendeten Zeichen können im Symbolverzeichnis am Ende der Arbeit nachgeschlagen werden.

2 Grundlagen

Das folgende Kapitel liefert die Grundlagen zu den in dieser Arbeit behandelten Themen. Dabei wird in Abschnitt 2.1 zunächst der allgemeine Ablauf von PPRL als Spezialfall des Record Linkage sowie Hintergründe zur Verschlüsselung der Daten beschrieben. Eine genaue Definition metrischer Räume ist in Abschnitt 2.2 zu finden, während Abschnitt 2.3 deren Nutzen für PPRL mit dem in [4] entwickelten Ansatz aufzeigt. Abschnitt 2.4 stellt das verwendete Framework Apache Flink vor.

2.1 Privacy Preserving Record Linkage

2.1.1 Record Linkage

Record Linkage (auch Entity Resolution genannt) bezeichnet den Prozess des Auffindens von ähnlichen Datensätzen aus unterschiedlichen und unabhängigen Datenquellen. Das Ziel dabei ist, diejenigen Datensätze miteinander zu verknüpfen, die dasselbe Realweltobjekt darstellen.

ID	Given_name	Surname	DOB	Gender	Address	Loan_type	Balance
6723	peter	robert	20.06.72	M	16 Main Street 2617	Mortgage	230,000
8345	smith	roberts	11.10.79	M	645 Reader Ave 2602	Personal	8,100
9241	amelia	millar	06.01.74	F	49E Applecross Rd 2415	Mortgage	320,750

Abbildung 2.1: Beispiel Bankkunden-Datenbank [1]

Beispielsweise könnte eine Bankkunden-Datenbank (Abbildung 2.1) mit einer Patienten-Datenbank (Abbildung 2.2) verknüpft werden. Die Schwierigkeit beim Record Linkage besteht darin, dass es in den verschiedenen Datenquellen keine eindeutigen Identifizierungsmerkmale (wie einheitliche IDs) gibt und daher die Anwendung eines einfachen SQL-Joins nicht möglich ist. Außerdem können die Datensätze zusätzlich

PID	Last_name	First_name	Age	Address	Sex	Pressure	Stress	Reason
P1209	roberts	peter	41	16 Main St 2617	m	140/90	high	chest pain
P4204	miller	amelia	39	49 Aplecross Road 2415	f	120/80	high	headache
P4894	sieman	jeff	30	123 Norcross Blvd 2602	m	110/80	normal	checkup

Abbildung 2.2: Beispiel Patienten-Datenbank [1]

verunreinigt sein. Es müssen also anhand bestimmter Felder (*Quasi-Identifiers* wie Name, Geburtsdatum und Addresse) String-Vergleiche ausgeführt werden. [1]

Record Linkage ist ein entscheidender Prozess in vielen Big-Data-Anwendungen, welcher außerdem dazu verwendet werden kann, Duplikate innerhalb einer Datenbank aufzufinden.

2.1.2 PPRL-Protokoll

Wie das Beispiel in Abschnitt 2.1.1 zeigt, sind die zu analysierenden Daten beim Record Linkage oft vertraulich und würden nie unverschlüsselt von der Datenquelle zur Verfügung gestellt werden. Ein Demografieforscher würde aus den genannten Datenquellen (Abbildung 2.1 und Abbildung 2.2) aber auch nur wenige Felder (hier z.B. *loan type, balance amount, blood pressure* and *stress level*) benötigen. Die Felder mit persönlichen Daten werden nur für die Verknüpfung der Datenquellen benutzt. Hier kommt das Privacy Preserving Record Linkage (PPRL) ins Spiel. [1]

Bei PPRL werden personenbezogene Daten ausschließlich verschlüsselt zwischen den Quellen ausgetauscht. Dazu wird derzeit meist mit einer vertrauenswürdigen Linkage Unit gearbeitet. Diese steht im Mittelpunkt des PPRL-Prozesses und führt das eigentliche Verknüpfen der Datenquellen anhand der verschlüsselten Daten aus.

Im angewandten Protokoll (Abbildung 2.3) tauschen zunächst die beiden Datenquellen genaue Parameter für die Verschlüsselung aus. Dies umfasst, welche Felder (Identifizierungsmerkmale wie z.B. Name, Geburtsdatum etc.) und Hash-Funktionen zu verwenden sind. Anschließend verschlüsselt jeder Datenbesitzer seine Datensätze um sie zu anonymisieren. Die Linkage Unit erhält dann die Datensätze beider Quellen in verschlüsselter Form und berechnet jeweils die Ähnlichkeit zueinander. Kommt der Linkage Algorithm zu dem Schluss, dass zwei Datensätze verknüpft werden sollen, werden ihre IDs einander zugeordnet. Diese Paare von IDs werden

abschließend an beide Datenquellen übergeben. Im Anschluss kann jeweils nur mit den forschungsrelevanten Daten in Verbindung mit den ID-Paaren weitergearbeitet werden ohne die personenbezogenen Daten verwenden zu müssen. Weiterhin werden oftmals zur weiteren Analyse nur Daten von zusammenpassenden Datensätzen verwendet, um Daten von Personen, welche in nur einer der Datenquellen vorkommen, nicht freizugeben. [2] [4]

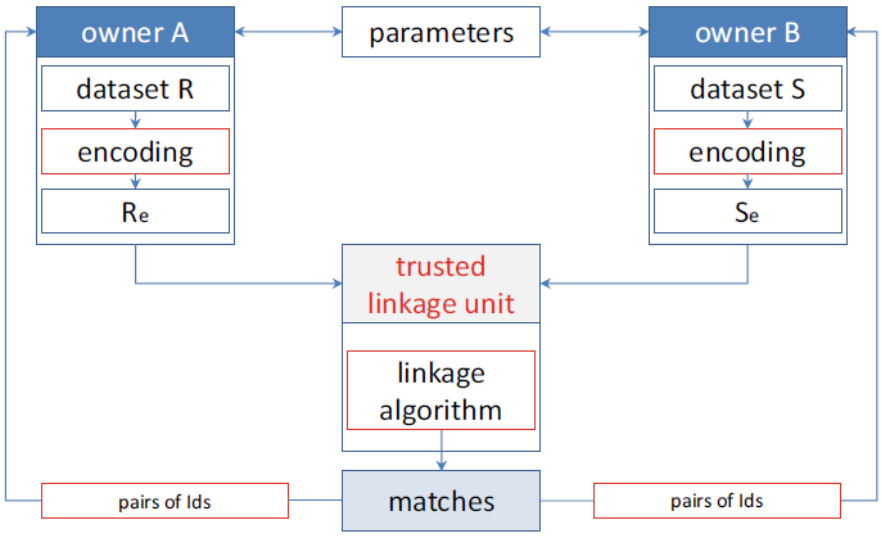

Abbildung 2.3: PPRL-Protokoll mit Linkage Unit [4]

Die besondere Anforderung an PPRL besteht in der Arbeit mit verschlüsselten Daten. Für das genaue Vorgehen (Art der Verschlüsselung etc.) gibt es verschiedene Ansätze. An vielen möglichen Lösungen wird aktuell geforscht. Bei den in dieser Arbeit verwendeten Daten findet die Verschlüsselung mit Bloom-Filtern statt.

2.1.3 Bloom-Filter

Bloom-Filter sind Bit-Arrays. Wie in Abbildung 2.4 dargestellt, werden auf alle Elemente eines Datensatzes (bei Strings oft q-Gramme) mehrere Hash-Funktionen angewendet. Die Ergebnisse der Hash-Funktionen bestimmen die Positionen der gesetzten Bits im Array. Die Bit-Arrays haben eine festgelegte Länge. Das Beispiel zeigt wie die Strings *"pete"* und *"peter"* in Bi-Gramme zerlegt werden und durch Anwendung von 2 Hash-Funktionen bestimmte Positionen des Bit-Arrays auf 1 setzen. Der Unterschied besteht nur in den Bits, die wegen *"er"* gesetzt wurden.

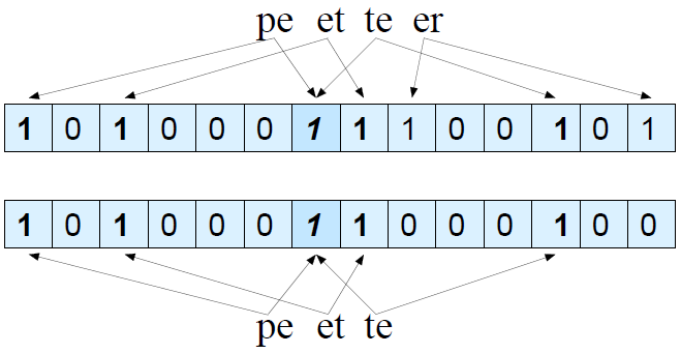

Abbildung 2.4: Bloom-Filter-Verschlüsselung [2]

Ähnliche Strings erzeugen in der Regel ähnliche Bit-Arrays. Allerdings muss für den PPRL-Kontext sowohl die Anzahl der Hash-Funktionen als auch die Länge der Bit-Arrays sorgfältig ausgewählt werden. Dies ist einerseits notwendig um eine Gleichverteilung der Bit-Arrays zu gewährleisten, um mögliche Entschlüsselungsangriffe zu verhindern. Weiterhin sollten zu viele Überschneidungen vermieden werden, um gute Ergebnisse der Ähnlichkeitsanalyse zu ermöglichen. In einem realistischen Anwendungsfall befüllen z.B. etwa 30 Hash-Funktionen ein Bit-Array der Länge 1.000. [2]

Die verwendeten Hash-Funktionen sind nur den Besitzern der Daten bekannt. So können die Daten von der Linkage Unit theoretisch nicht entschlüsselt werden. Die genaue Art der Verschlüsselung (d.h. welche Hash-Funktionen und Felder verwendet werden) wird in dieser Arbeit nicht behandelt. Es wird mit gegebenen verschlüsselten Bit-Arrays der Länge 1.000 gearbeitet (im Folgenden als Datensätze bezeichnet).

2.1.4 Herausforderung Performance

Im Zusammenhang mit Big-Data-Anwendungen besteht die Herausforderung bei PPRL in der Performance bzw. der Skalierbarkeit. Es handelt sich bei dem Linkage-Prozess grundsätzlich um ein Problem mit quadratischer Komplexität (vgl. Abbildung 2.5), wenn man alle Datensätze der beiden Datenquellen jeweils miteinander vergleicht. Dieser Ansatz ist jedoch bei enorm großen Datenmengen nicht praxistauglich.

Bei der Suche nach ähnlichen Datensätzen ist es für die Performance entscheidend, den Suchraum zu begrenzen. Es sind daher Filter-und Blocking-Verfahren nötig, um

bestimmte unähnliche Datensätze vorab vom Vergleich auszuschließen. Eine Möglichkeit, dies zu erreichen, ist die Anwendung bestimmter Eigenschaften von metrischen Räumen. Dieser Ansatz wird in dieser Arbeit verfolgt und in Abschnitt 2.3 genauer beschrieben.

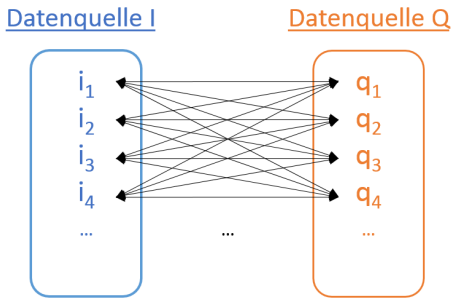

Abbildung 2.5: quadratische Komplexität des Linkage-Problems

2.2 Metrische Räume

2.2.1 Definition

Ein metrischer Raum M ist definiert durch das Paar (X, d), wobei X eine Menge von Elementen und d eine Distanz-Funktion ist.

Für alle Elemente $a, b, c \in X$ müssen die folgenden Eigenschaften gelten:

1.) $d(a, b) \geq 0$ (positive Definitheit)
Es gibt keine negativen Distanzen.

2.) $d(a, b) = 0 \Leftrightarrow a = b$ (Identität)
Ein Element hat zu sich selbst eine Distanz von 0. Ist die Distanz zwischen zwei Elementen 0, so handelt es sich um dasselbe Element.

3.) $d(a, b) = d(b, a)$ (Symmetrie)
Die Distanz ist richtungsunabhängig.

4.) $d(a, c) \leq d(a, b) + d(b, c)$ (Dreiecksungleichung)
Die Distanz eines Umwegs über ein drittes Element ist immer mindestens so groß, wie die direkte Distanz zwischen zwei Elementen.

[5]

2.2.2 Distanzen

Für metrische Räume können verschiedene Distanzfunktionen verwendet werden solange diese die in Abschnitt 2.2.1 beschriebenen Eigenschaften erfüllen. Zwei Beispiele für Distanzfunktionen, welche auch in dieser Arbeit verwendet werden sind die Hamming-Distanz und die Jaccard-Distanz.

Hamming-Distanz

Mit der Hamming-Distanz lassen sich Abstände von Zeichenketten berechnen. Sie ist definiert durch:

$$d_h(a, b) = |a \vee b| - |a \wedge b| \tag{2.1}$$

Im Falle von Bit-Vektoren ist $|a \wedge b|$ die Anzahl der übereinstimmenden Positionen gesetzter Bits in a *und* b. $|a \vee b|$ ist die Anzahl der Positionen mit gesetzten Bits in a *oder* b. Beispielsweise ergibt sich für den Abstand der beiden Bit-Vektoren aus Abbildung 2.4 $d_h(Alice, Bob) = 7 - 5 = 2$. Alternativ lässt sich d_h mit der Anwendung des exklusiven Oder (XOR) auf die einzelnen Positionen der beiden Bit-Vektoren berechnen. Dazu müssen alle Positionen gezählt werden, welche mit XOR wahr ergeben. [5]

Jaccard-Distanz

Die Jaccard-Distanz d_j ergibt sich aus der Jaccard-Ähnlichkeit:

$$d_j(a, b) = 1 - sim_j(a, b) \tag{2.2}$$

wobei die Jaccard-Ähnlichkeit $sim_j(a, b)$ definiert ist durch:

$$sim_j(a, b) = \frac{|a \wedge b|}{|a \vee b|} \tag{2.3}$$

Es gelten die gleichen Schlussfolgerungen für Bit-Vektoren wie bei der Hamming-Distanz, so dass sich für die beiden Bit-Vektoren aus Abbildung 2.4 ergibt:
$d_j(Alice, Bob) = 1 - \frac{5}{7} = \frac{2}{7}$. Die Jaccard-Distanz sowie die Jaccard-Ähnlichkeit sind auf $[0, 1]$ normiert. [5]

2.3 PPRL mit metrischen Räumen

Bei der Arbeit mit Bit-Arrays und der Untersuchung deren Ähnlichkeiten bietet sich die Verwendung von metrischen Räumen an, da diese aufgrund der Fokussierung auf die Distanz zwischen zwei Objekten eine einfache Handhabung versprechen. Außerdem bietet die Anwendung der Dreiecksungleichung einen hervorragenden Ansatzpunkt als Filter für den Linkage-Prozess bei PPRL, was in Abschnitt 2.3.2 beschrieben wird. Wie Abbildung 2.6 zeigt, sind die Laufzeiten dieses Ansatzes mit der Verwendung von Pivot-Elementen (*pivots_sample*, genaue Beschreibung folgt in Abschnitt 2.3.1 und Abschnitt 2.3.2) um einige Größenordnungen besser als andere PPRL-Techniken. Der Grund hierfür ist eine weitaus bessere Filterung (bzw. Blocking) als bei zuvor entwickelten Methoden. Somit wird die Anzahl der letztlich auszuführenden Vergleiche drastisch reduziert. Es besteht jedoch weiterhin ein nahezu quadratisches Wachstum der Laufzeiten bzw. Vergleiche. [4]

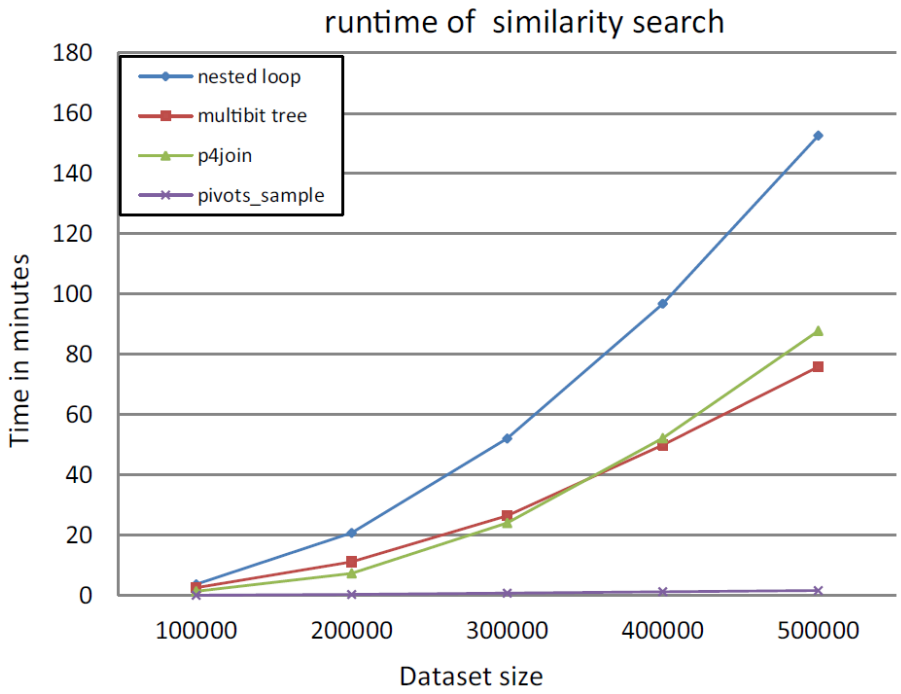

Abbildung 2.6: Laufzeit verschiedener PPRL-Techniken [4]

2.3.1 Ähnlichkeitsradius und Pivot-Elemente

Bei PPRL ist es üblich, einen bestimmten Threshold $t \in [0, 1]$ für die Ähnlichkeit zweier Datensätze anzugeben (üblicherweise 0,8). Als Resultat sollen all diejenigen Paare von Datensätzen ausgegeben werden, deren Ähnlichkeit mindestens t entspricht. Intern möchte man jedoch mit der nicht normierten Hamming-Distanz arbeiten, welche mittels *XOR* sehr einfach zu berechnen ist. Dazu muss für einen Datensatz q ein sogenannter Ähnlichkeitsradius $rad(q)$ berechnet werden, welcher den maximalen Hamming-Abstand zu einem anderen Datensatz angibt, sodass die Ähnlichkeit der beiden Datensätze mindestens t entspricht. Mit Hilfe der Relation zwischen Jaccard-Ähnlichkeit und Hamming-Distanz $sim_j(x, q) \geq t \Leftrightarrow d_h(x, q) \leq (|x| + |q|)(\frac{1-t}{1+t})$ und $|x| \leq \frac{|q|}{t}$ (Längenfilter) kann folgende Abschätzung für den Ähnlichkeitsradius angegeben werden [4]:

$$rad(q) = |q|\frac{1-t}{t} \tag{2.4}$$

Um die Eigenschaften metrischer Räume für PPRL nutzen zu können, müssen in der ersten Datenquelle (Index-Datenquelle I mit Index-Datensätzen $i \in I$) einige Datensätze als Pivot-Elemente p ausgewählt werden. Diesen Pivot-Elementen sind alle anderen Datensätze aus I zugeordnet. Die Distanzen der zugeordneten Datensätze zu jedem Pivot-Element werden beim Bestimmen der Pivot-Elemente vorberechnet. Jeder Datensatz wird demjenigen Pivot-Element zugeordnet, zu welchem es den geringsten Abstand $d_h(i, p)$ hat. Neben den zugeordneten Datensätzen (Menge $Set(p)$) hat jedes Pivot-Element als weitere wichtige Eigenschaft seinen Radius $rad(p)$. Dieser ergibt sich aus dem Abstand des am weitesten entfernten zugeordneten Datensatzes. Somit liegen alle Datensätze aus $Set(p)$ innerhalb von $rad(p)$. Als Index-Datenquelle wird üblicherweise die größere der beiden Datenquellen gewählt, da so der Nutzen für die Performance am größten ist. [4]

2.3.2 Nutzen der Dreiecksungleichung

Jeder Datensatz aus der zweiten Datenquelle (Query-Datenquelle Q mit Query-Datensätzen $q \in Q$) wird zunächst nur mit den Pivot-Elementen verglichen. Die Abstände der beiden Datensätze $x, y \in Set(p)$ zu p sind bekannt. Wie Abbildung 2.7 zeigt, kann der Datensatz y vom Vergleich zu q ausgeschlossen werden, da die Dreiecksungleichung der Form

$$d_h(p,q) \leq d_h(p,y) + d_h(q,y) \tag{2.5}$$

zeigt, dass y nicht innerhalb von $rad(q)$ liegen kann.

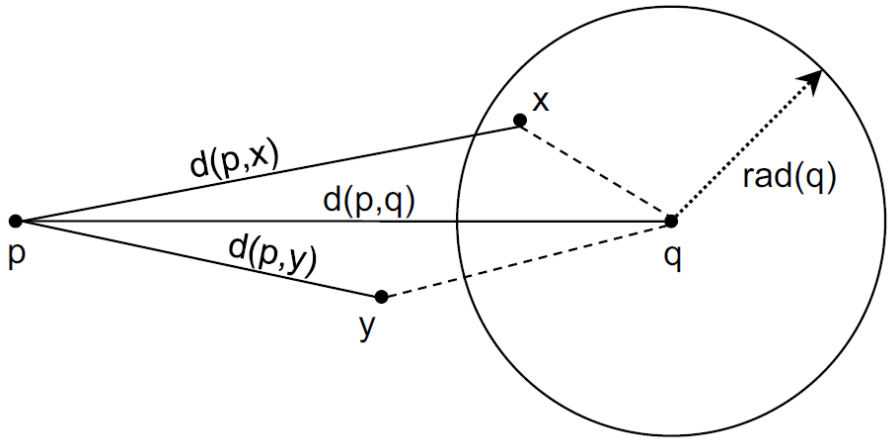

Abbildung 2.7: Anwendung der Dreiecksungleichung [4]

In der Praxis wird als erster Filter der Abstand $d_h(p,q)$ verwendet. Gilt

$$d_h(p,q) > rad(q) + rad(p) \tag{2.6}$$

so muss keiner der Datensätze aus $Set(p)$ mit q verglichen werden, da diese aufgrund der Dreiecksungleichung garantiert nicht innerhalb von $rad(q)$ liegen. Andernfalls werden die Datensätze aus $Set(p)$ mit ihren vorberechneten Distanzen zu p betrachtet. Gilt dabei

$$d_h(p,q) - d_h(p,y) > rad(q) \tag{2.7}$$

kann y ebenfalls als ähnlicher Datensatz von q ausgeschlossen werden. Die teure Berechnung des genauen Abstandes $d_h(x,q)$ muss nur für die übrigen Datensätze durchgeführt werden (vgl. Abbildung 2.7). [4]

2.3.3 Algorithmen zum Finden von Pivot-Elementen

Das Auswählen der Pivot-Elemente (Pivots) ist ein wichtiger Prozess. Die Verteilung der Index-Datensätze sowie die Lage der Pivots zueinander bestimmt maßgeblich den möglichen Anteil der geblockten Vergleiche. Auch die Anzahl der zu wählenden Pivots ist ein entscheidender Faktor. Diese sollte sich nach der intrinsischen Dimensionalität der Daten richten, d.h. nach der Länge der Bit-Arrays bzw. deren "Aussagekraft". Zu viele Pivots bringen keine weiteren Informationen, sondern nur Redundanz. [6]

Gute Pivot-Elemente

Generell gibt es für das Finden guter Pivots (d.h. solche, die später möglichst viele Vergleiche blocken) keine exakten Kenngrößen, sondern nur Heuristiken. Außerdem sind sie immer von der genauen Datenlage abhängig. Deshalb ist es auch nicht sinnvoll, vordefinierte Pivots zu verwenden.

Als eine der wichtigsten Eigenschaften guter Pivots gilt eine möglichst geringe Zahl der Überlappungen zwischen den Pivot-Radien. Dies reduziert die Zahl der Zuordnungen von Query-Datensätzen zu mehreren Pivots. Das bedeutet, dass Pivots am "Rand" des metrischen Raumes zu bevorzugen sind. Ein Datensatz soll möglichst verschiedene Abstände zu allen Pivots haben. [6] [7]

Im folgenden werden zwei Algorithmen zum Finden von Pivots vorgestellt, die diese Erkenntnisse berücksichtigen. Beide Algorithmen haben eine Laufzeit von $\mathbb{O}(n_I \cdot n_P)$, wobei n_I die Anzahl der Index-Datensätze und n_P die Anzahl der Pivots angibt.

Maximale Separation

Beim Maximum-Separation-Algorithmus (MS) wird versucht, die Summe der Distanzen zwischen den Pivots zu maximieren. Nachdem das erste Pivot-Element zufällig ausgewählt wurde, werden alle weiteren iterativ bestimmt. Dabei wird derjenige Datensatz gewählt, dessen Abstand zu allen bisher ausgewählten Pivots (in Summe) am größten ist (vgl. Abbildung 2.8). [6]

Input : Set of Index Records I with elements i_n
Input : Number of Pivots n_P
Output : Set of Pivots P

```
1  for i_n ∈ I:
2      distanceSum_n ← 0;
3  lastPivot ← i_1;
4  for (n_P):
5      maxDistance ← 0;
6      for i_n ∈ I:
7          distanceSum_n ← distanceSum_n + d_h(i_n, lastPivot);
8          if maxDistance < distanceSum_n:
9              nextPivot ← i_n;
10             maxDistance ← distanceSum_n;
11     I ← I \ {nextPivot};
12     P ← P ∪ {nextPivot};
13     lastPivot ← nextPivot;
```

Abbildung 2.8: Maximum-Separation (MS)

Farthest-First-Traversal

Der Farthest-First-Traversal-Algorithmus (FFT) ist ein Cluster-Algorithmus, der verwendet wird, um "Ecken" im metrischen Raum zu finden. Auch hier wird das erste Pivot-Element zufällig ausgewählt und die übrigen iterativ bestimmt. Es wird von jedem Datensatz der geringste Abstand zu allen bisher ausgewählten Pivots berechnet und anschließend derjenige ausgewählt, bei dem dieser minimale Abstand am größten ist (vgl. Abbildung 2.9). [6]

Input : Set of Index Records I with elements i_n
Input : Number of Pivots n_P
Output : Set of Pivots P

```
1  for i_n ∈ I:
2      minDistance_n ← ∞;
3  lastPivot ← i_1;
4  for (n_P):
5      maxDistance ← 0;
6      for i_n ∈ I:
7          distance ← d_h(i_n, lastPivot);
8          if distance < minDistance_n:
9              minDistance_n ← distance;
10         if maxDistance < minDistance_n:
11             nextPivot ← i_n;
12             maxDistance ← minDistance_n;
13     I ← I \ {nextPivot};
14     P ← P ∪ {nextPivot};
15     lastPivot ← nextPivot;
```

Abbildung 2.9: Farthest-First-Traversal (FFT)

2.4 Apache Flink

Abbildung 2.10: Flink-Logo [8]

Flink ist ein Open-Source-Framework und derzeit Top-Level-Projekt der Apache Software Foundation. Grundsätzlich ist Flink ein Distributed Stream Processor, verarbeitet also Daten-Streams parallel. Darauf aufbauend gibt es zwei unterschiedliche APIs:

- DataStream API: für Stream Processing
- DataSet API: für Batch Processing

Für diese Arbeit ist dabei nur die DataSet API interessent, mit welcher verteilte Stapelverarbeitung möglich ist. Jedoch ist zu beachten, dass Flink intern jegliche Prozesse (egal ob Batch oder Stream) mit der Streaming Engine verarbeitet. Batch-Verarbeitung wird als Spezialfall von Stream-Verarbeitung betrachtet und Daten-Sets werden intern als begrenzte Daten-Streams behandelt. Flink verspricht auf dieser Grundlage eine verteilte, hoch performante, stets verfügbare und genaue Datenverarbeitung sowie einen hohen Datendurchsatz bei gleichzeitig geringer Latenz auch bei einem hohen Grad der Parallelisierung. Prinzipiell bestehen Flink-Programme aus:

- Data Source (Datenquelle)
- Transformations (Verarbeitung der Daten)
- Data Sink (Datensenke)

Als Datenquelle bzw. Datensenke sind verschiedene Formate wie beispielsweise CSV verwendbar. Außerdem ist Flink kompatibel mit dem Hadoop Distributed File System (HDFS). Dieses Framework wurde gewählt, weil es für die parallele Verarbeitung der Daten eine große Auswahl an Operatoren bietet, weit mehr als nur *Map* und *Reduce*.

Die Ausführung von Flink kann lokal (Cluster wird simuliert) oder auf einem Cluster erfolgen. Ein Programm wird dabei als Job an den JobManager (Master Node) übermittelt. Von dort aus wird die Datenverarbeitung der TaskManager (Worker Nodes) gesteuert. Flink steht für die Programmiersprachen *Java* und *Scala* zur Verfügung. Für die Umsetzung dieser Arbeit wurde *Java* mit der Version 1.2 von Flink verwendet. [8] [9]

3 Verfahren

In diesem Kapitel werden die Verfahren beschrieben, welche im Rahmen dieser Arbeit entwickelt wurden. Abschnitt 3.1 erläutert zunächst abstrakt den allgemeinen Ablauf. Abschnitt 3.2 zeigt anschließend die konkrete Umsetzung mit Apache Flink sowie die verschiedenen Strategieansätze und Optimierungen.

3.1 Ablauf

3.1.1 Überblick

Der grundlegende Ablauf des in dieser Arbeit entwickelten verteilten PPRL-Verfahrens ist in Abbildung 3.1 dargestellt. Die einzelnen Teilschritte sind mit gelb unterlegten Buchstaben gekennzeichnet. Es geht dabei um den Linkage-Prozess innerhalb der Linkage Unit (vgl. Abschnitt 2.1.2). Daher werden die Verschlüsselungen (a und d) der Datensätze als gegeben betrachtet. Es liegen also zunächst die verschlüsselten Datensätze aus der Index-Datenquelle verteilt vor. Aus diesen müssen durch ein paralleles Verfahren (b) Pivot-Elemente ausgewählt werden (siehe Abschnitt 3.1.2). Anschließend werden die Datensätze den Pivot-Elementen zugeordnet und eventuell umverteilt (c). Dabei können durchaus mehrere Pivot-Elemente auf der gleichen Partition liegen. Zusätzlich wird ein *Distributed Cache* angelegt, welcher nur die Pivot-Elemente mitsamt ihrer Radien speichert und somit partitionsübergreifende Informationen liefert. Die Datensätze aus der Query-Datenquelle werden zunächst nur mit dem Distributed Cache verglichen. Greift der erste Filter (Gleichung 2.6) für alle Pivot-Elemente, so kann der aktuelle Datensatz als unähnlich ausgeschlossen werden (f). Andernfalls wird er zu einem oder mehreren Pivot-Elementen zugeordnet (g). Es folgt ein genauerer Vergleich mit den zugeordneten Index-Datensätzen des Pivot-Elements. Hierfür wird zunächst der zweite Filter (Gleichung 2.7) angewen-

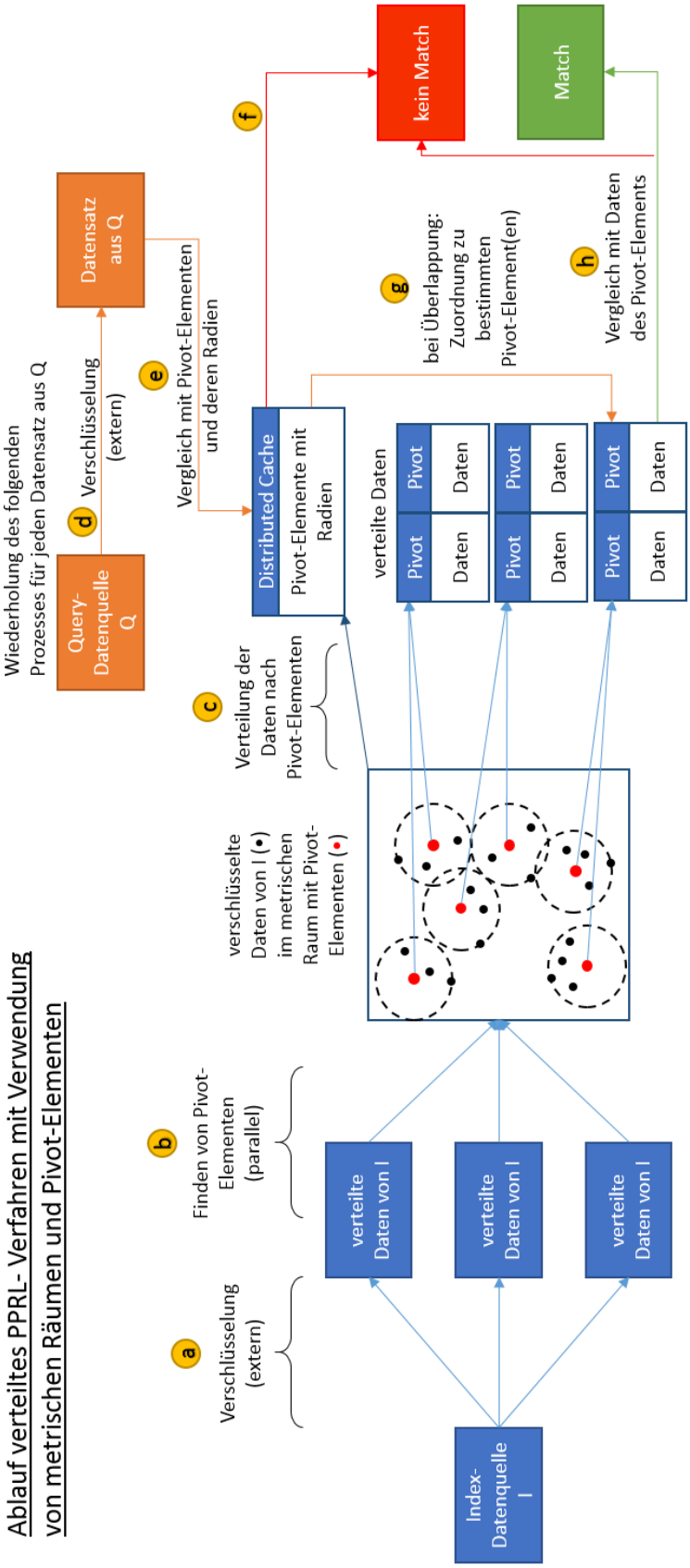

Abbildung 3.1: Ablauf verteiltes PPRL-Verfahren

det und anschließend eventuell die exakte Ähnlichkeit der Datensätze berechnet. Liegt diese überhalb des vorgegebenen Ähnlichkeits-Threshold t, wird das Paar von Datensätzen als Match in die Ergebnisliste aufgenommen (h). Das Verfahren garantiert, alle Paare von Datensätzen zu finden, deren Ähnlichkeit größer oder gleich t ist (vgl. auch Abschnitt 4.1.3).

3.1.2 Finden von Pivot-Elementen

Der Prozess des Findens der Pivot-Elemente (Abbildung 3.1, Schritt b) verläuft in zwei Teilschritten. Wie in Abbildung 3.2 zu sehen, werden durch einen ersten Verarbeitungsschritt ($b1$) zunächst auf jeder Partition lokal einige Datensätze (*lokale Pivots*) ausgewählt. Anschließend werden aus diesen die endgültigen *globalen Pivots* bestimmt ($b2$). Dieser Schritt erfolgt nicht parallelisiert. Mit der Zuordnung der übrigen Datensätze zu den Pivots (c) wird der Distributed Cache angelegt, welcher Informationen über die Pivots und deren Radien enthält. Für den lokalen und globalen Verarbeitungsschritt sind verschiedene Strategien denkbar.

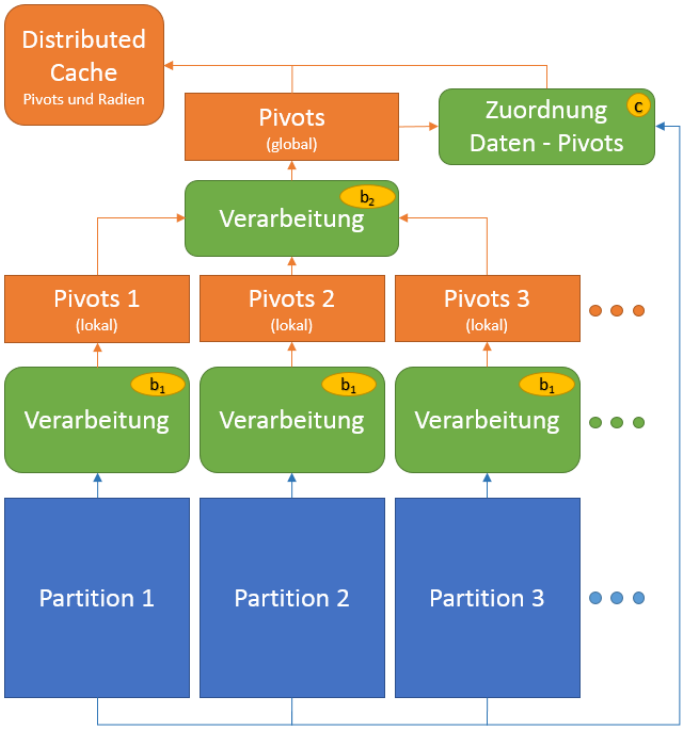

Abbildung 3.2: Ablauf verteiltes Finden von Pivot-Elementen

Lokale Strategien:

- zufällige Auswahl (*random*)
- Farthest-First-Traversal-Algorithmus (*fft*)
- Maximum-Separation-Algorithmus (*ms*)

Globale Strategien:

- keine Verarbeitung (*none*)
- Farthest-First-Traversal-Algorithmus (*fft*)
- Maximum-Separation-Algorithmus (*ms*)

Ein mögliches Vorgehen wäre zum Beispiel *random-ms*, also zufällige Datensätze pro Partition auswählen und aus diesen per MS die Pivots bestimmen. Dieses Vorgehen wird prinzipiell auch in nicht verteilten PPRL-Anwendungen genutzt. Dabei werden die Pivots nicht aus allen Index-Datensätzen ausgewählt, sondern nur aus einer kleineren Auswahl der Daten [4]. Allerdings gibt es noch viele weitere Kombinationsmöglichkeiten, die in dieser Arbeit evaluiert werden (siehe Abschnitt 4.2).

Auch die Anzahl der jeweils auszuwählenden Datensätze (lokal und global) spielt eine wichtige Rolle für die Performance und wird evaluiert. Es gilt dabei zwischen den benötigten Kosten zum Finden der Pivots und deren Vorteil für eine spätere Filterung beim Matching abzuwägen.

3.2 Umsetzung mit Flink

3.2.1 Ausführungsplan

Anhand des in Abschnitt 3.1 beschriebenen Konzeptes wurde im Rahmen dieser Arbeit mit Apache Flink ein Programm entwickelt. In diesem Abschnitt wird der genaue Ablauf dieses Programms erläutert. Als Ausgangspunkt hierfür wurde die Darstellung des Ausführungsplans mit dem *Flink Plan Visualizer* [8] verwendet. Mit diesem Web-Interface wird eine Möglichkeit angeboten, sich den Ausführungsplan grafisch darstellen zu lassen. Dabei werden die Umsetzung des Quellcodes und die vorgenommene Optimierung der einzelnen Schritte durch Flink aufgezeigt.

Ausgehend von dieser Ansicht wurde in Abbildung 3.3 ein noch detaillierterer Ausführungsplan des entwickelten Programms entworfen, um den genauen Ablauf zu veranschaulichen. Die großen Rechtecke stehen für Zwischenergebnisse (Mengen von Datensätzen) und sind wie die entsprechende Variable im Quellcode (vgl. Abschnitt A.1) bezeichnet. In der unteren Zeile ist dabei zu erkennen, wie die einzelnen Datensätze des jeweiligen Zwischenergebnisses aufgebaut sind. Die kleineren, nummerierten Rechtecke stellen die verschiedenen Verarbeitungsschritte (Datentransformationen) dar und sind mit den konkreten Operatoren von Flink bezeichnet.

Zunächst werden für jeden Datensatz aus der Index- und Query-Datenquelle jeweils die ID und das Bit-Array eingelesen (grüne Rechtecke *indexData* und *queryData*). Dafür wird die von Flink zur Verfügung gestellte Methode *readCsvFile* verwendet. Die Datensätze werden durch Flink intern verschiedenen Partitionen (bzw. Knoten) zugewiesen. Die Anzahl entspricht dem Grad der Parallelität.

Die Verarbeitung beginnt mit der Bestimmung der lokalen Pivots (*1*). Der verwendete Operator *MapPartition* erlaubt es, pro Partition einen Algorithmus auf allen Datensätzen der Partition auszuführen und eine beliebige Anzahl von Datensätzen als Ergebnis zu erhalten. Auf den Daten jeder Partition wird damit parallel der als lokale Strategie gewählte Algorithmus zum Finden der Pivots ausgeführt. Alle so erzeugten lokalen Pivots werden in *pivotsLocal* vereinigt.

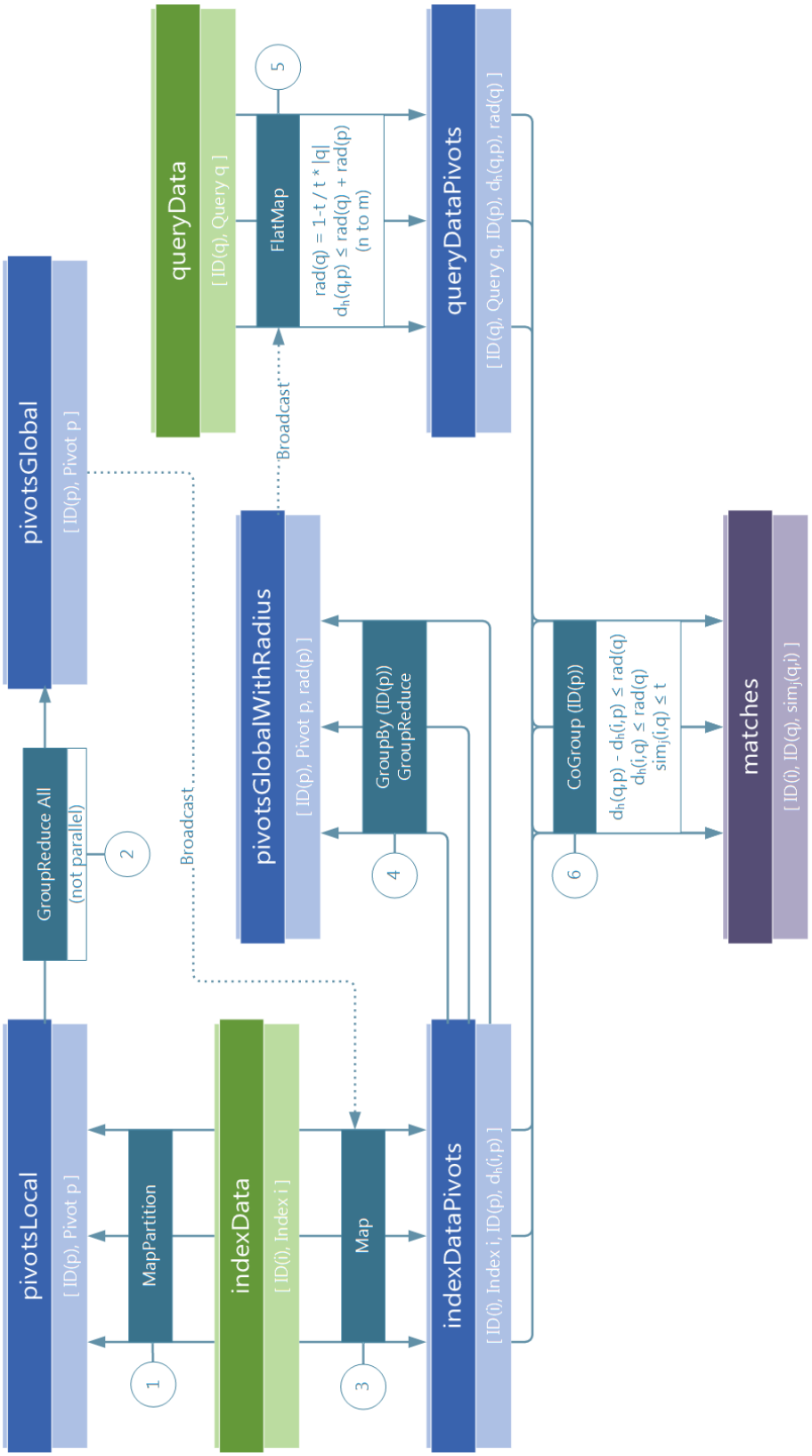

Abbildung 3.3: Ausführungsplan Flink-Programm

Anschließend werden daraus die globalen Pivots mit dem hierfür als globale Strategie gewählten Algorithmus bestimmt (*2*). Dieser Schritt verläuft als einziger im Programm nicht verteilt. Bei der Anwendung von *GroupReduce* auf die gesamte Menge von Datensätzen ist dies standardmäßig der Fall. *GroupReduce* funktioniert in diesem Fall ähnlich wie MapPartition (nur ohne Parallelität) und lässt eine beliebige Anzahl von Datensätzen als Ergebnis zu.

Der gewünschte Distributed Cache wird in Flink mit sogenannten Broadcast Variables umgesetzt. Wird eine Datenmenge als *Broadcast* deklariert, steht sie anschließend in allen parallelen Instanzen zur Verfügung [9]. Hierfür eignen sich vor allem kleinere Mengen, da diese mehrfach kopiert werden müssen. Dies ist bei den globalen Pivots der Fall. Es erfolgt ein Broadcasting der *pivotsGlobal* für die Zuordnung der Index-Datensätze zu den Pivots (*3*). Durch ein einfaches *Map* wird jedem Index-Datensatz genau ein Pivot (das mit dem geringsten Abstand) zugeordnet. Pivots werden dabei sich selbst zugeordnet. Danach stehen in *indexDataPivots* die Index-Datensätze jeweils mit ihrem zugeordneten Pivot und der entsprechenden Distanz zur Verfügung.

Als nächster Schritt folgt die Berechnung der Radien der Pivots (*4*). Die Index-Datensätze werden hierfür nach den Pivots gruppiert. Anschließend wird für jede Gruppe durch *GroupReduce* der Radius bestimmt. Die Wirkung von *GroupReduce* ist hier aufgrund der vorigen Gruppierung der Datensätze etwas anders. Die Verarbeitung erfolgt parallel pro Gruppe (ähnlich wie bei MapPartition pro Partition). Die so berechneten Radien werden mitsamt der Pivots in *pivotsGlobalWithRad* zwischengespeichert.

Für den ersten Verarbeitungsschritt der Query-Datensätze (*5*) erfolgt erneut ein Broadcasting der Pivots mit Radius. Per *FlatMap*, welches es erlaubt pro eingehendem Datensatz auch mehrere oder keinen ausgehenden Datensatz zu erzeugen, werden nun die Query-Datensätze den Pivots zugeordnet. Zunächst wird dafür pro Datensatz der Ähnlichkeitsradius ($rad(q)$) berechnet. Im Anschluss wird der erste Filter (vgl. Gleichung 2.6) angewandt und so bestimmt, welche Pivots mitsamt Daten für den jeweiligen Query-Datensatz näher zu betrachten sind. Die Ergebnisse werden in *queryDataPivots* als Paare von Query-Datensätzen und Pivots mit Abstand zueinander und dem Ähnlichkeitsradius zusammengefasst. Ein Query-Datensatz kann dabei mehrfach auftauchen (wenn er mehreren Pivots zugeordnet wurde) oder auch gar nicht (in diesem Fall wird er bereits hier als mögliches Match ausgeschlossen).

Der finale Schritt, das Matching (*6*), erfolgt mit *CoGroup*. Bei dieser Transformation werden die Datensätze der beiden eingehenden Mengen zuerst nach einem gemeinsamen Schlüssel (hier Pivot-Element) sortiert. Es folgt eine Verarbeitung pro Gruppe (ähnlich dem einfachen *Reduce*). Für jedes Pivot werden so alle zugeordneten Index-Datensätze paarweise mit den zugeordneten Query-Datensätzen verglichen. Dazu wird zuerst der zweite Filter (Dreiecksungleichung, vgl. Gleichung 2.7) verwendet. Für alle verbleibenden Kandidaten-Paare von Index- und Query-Datensatz wird deren genaue Distanz und Ähnlichkeit bestimmt. Als Ergebnis (violettes Rechteck *matches*) werden die Paare mit ihrer Ähnlichkeit ausgegeben, welche größer als der vorgegebene Threshold t ist.

Verknüpfung der Ablaufpläne

Aufgrund von Praktikabilität und einigen Besonderheiten der zur Verfügung stehenden Operatoren sind der konzeptionelle Ablaufplan (Abbildung 3.1 mit Abbildung 3.2) und der konkrete Ausführungsplan in Flink (Abbildung 3.3) nicht vollständig deckungsgleich. Die einzelnen Teilschritte können untereinander wie folgt zugeordnet werden:

Beschreibung	Flink Ausführungsplan	konzeptioneller Ablaufplan
Verschlüsselung	gegeben	*a*, *d*
Finden der lokalen Pivots	Schritt *1*	*b1*
Finden der globalen Pivots	Schritt *2*	*b2*
Zuordnung der Index-Datensätze	Schritt *3*	*c*
Bestimmung der Pivotradien	Schritt *4*	*c*
Zuordnung der Query-Datensätze	Schritt *5*	*e*, *f*, *g*
Matching	Schritt *6*	*h*

3.2.2 Parameter

Das entwickelte Programm berücksichtigt verschiedene Parameter, die den genauen Ablauf verändern können:

- Ähnlichkeits-Threshold t
- Grad der Parallelität N
- Pivot-Zuordnungsstrategie, Auswahlmöglichkeiten:
  - *global* (Ablauf wie zuvor beschrieben)
  - *lokal* (In diesem Fall werden die Schritte *1* bis *3* zusammengefasst. Nach dem Finden der Pivots innerhalb einer Partition werden die Index-Daten ausschließlich den entsprechenden Pivots dieser Partition zugeordnet. Dieser Verarbeitungsschritt erfolgt mit einer *MapPartition*-Operation. Es gibt keine globalen Pivots. Diese Auswahl ist nur in Verbinderung mit *none* als globale Strategie zum Finden der Pivots möglich.)
- lokale Strategie zum Finden der Pivots, Auswahlmöglichkeiten:
  - *random* (zufällige Auswahl)
  - *fft* (Auswahl mit FFT)
  - *ms* (Auswahl mit MS)
- globale Strategie zum Finden der Pivots, Auswahlmöglichkeiten:
  - *none* (keine Verarbeitung. Es werden die lokal bestimmten Pivots verwendet. Schritt *2* entfällt hierbei komplett.)
  - *fft* (Auswahl mit FFT)
  - *ms* (Auswahl mit MS)
- Anzahl der Pivots lokal n_L
- Anzahl der Pivots global (gesamt) n_P

3.2.3 Optimierungen

In diesem Abschnitt werden einige allgemeine Optimierungen beschrieben, das heißt solche, die unabhängig von den gewählten Parametern die Performance des Programms verbessern. Einige sind speziell auf das Framework Flink und seine Eigenschaften bezogen, andere auf den PPRL-Prozess.

Verwendung von Tuples

Tuples werden als Datenstruktur von Flink zur Verfügung gestellt und nativ unterstützt. Mit ihnen lassen sich Datensätze mit mehreren Feldern abbilden. Tuples wurden im Programm selbst erstellten Klassen in Java (*POJOs*) vorgezogen, um unnötigen Rechenaufwand zu vermeiden. [9]

Außerdem wurde darauf geachtet, dass jeweils nur die absolut benötigten Felder in die verschiedenen Tuples aufgenommen wurden. Zum Beispiel wird bei *indexDataPivots* nur die ID der Pivots und nicht deren Bit-Array abgespeichert. Da jedes Pivot als Index-Datensatz sich selbst zugeordnet ist und die Bit-Arrays für die Index-Datensätze mit transportiert werden, kann das Bit-Array des Pivots so dennoch benutzt werden.

Semantic Annotations

Mit *Semantic Annotations* (Anmerkungen in Java vor den einzelnen Funktionen) können Flink Hinweise über das Verhalten einer Funktion gegeben werden. Dabei werden diejenigen Felder der Tuples angegeben, welche von der Funktion nicht modifiziert werden. Damit kann das Programm durch Flink besser optimiert werden und es entstehen gegebenenfalls Perfomance-Vorteile. [9]

Verwendung von OpenBitSet und Distanzvergleiche

Als Datenstruktur zum Speichern der Bit-Arrays wurde die Klasse *OpenBitSet* von *Apache Lucene* gewählt. Diese schneidet bei ausgeführten Operationen (beispielsweise die Berechnung der Kardinalität) wesentlich besser ab als *java.util.BitSet*. [10]

Für Distanzvergleiche zwischen zwei Bit-Arrays bezüglich der Hamming-Distanz d_h kann so effizient die Kardinalität des Ergebnisses der Anwendung des *XOR*-Operators auf die beiden OpenBitSets errechnet werden.

Pivot-Algorithmen

Die beiden Algorithmen zum Finden von Pivot-Elementen wie in Abbildung 2.8 und Abbildung 2.9 (Abschnitt 2.3.3) gezeigt, sind bereits Verbesserungen der ursprünglichen Version dieser Algorithmen. Die Performance wird entscheidend erhöht, indem bereits berechnete Distanzen zwischengespeichert werden. Bei *MS* handelt es sich um die Summe der Distanzen zu allen Pivots pro Datensatz, welche jeweils nur iterativ ergänzt und nicht neu berechnet wird. Bei *FFT* geht es um die minimale Distanz eines Datensatzes zu allen bisherigen Pivots.

Verwendung von CoGroup beim Matching

Für das Matching (Schritt *6*), d.h. das Zusammenführen von Index- und Query-Datensätzen wurde der Operator *CoGroup* verwendet. Dieser wird auch als zweidimensionale Variante von *Reduce* bezeichnet. Dabei werden wie in Abbildung 3.4 dargestellt, Daten aus beiden Inputs nach einem Schlüssel (farbige Kästchen) sortiert und anschließend gruppenweise verarbeitet. [11] Im vorliegenden Fall sind die Pivots der Schlüssel. So können die genaueren Betrachtungen zwischen Index-und Query-Datensätzen verteilt pro Pivot-Element stattfinden. *CoGroup* ist hierfür wesentlich effizienter als ein *Join*.

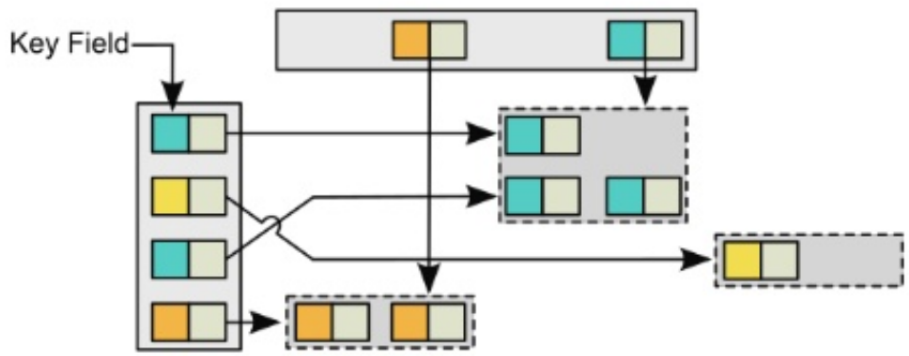

Abbildung 3.4: CoGroup [11]

Partitioning, Grouping und Combiner

Eine komplett neue Partitionierung der Datensätze ist eine sehr teure Transformation, da erheblich viele Daten über das Netzwerk geschickt werden müssen. [11] Versuche diesbezüglich brachten eine enorme Erhöhung der Laufzeiten und wurden daher eingestellt.

Manchmal ist aber eine Gruppierung der Daten nach einem Schlüssel unumgänglich (wie in Schritt *4* und *6*). Allerdings kann verhindert werden, dass diese Operation unnötig oft ausgeführt wird. Die Gruppierung von *indexDataPivots* nach Pivots findet direkt im Anschluss an Schritt *3* statt, die Datenmenge liegt dann als *UnsortedGrouping* vor und kann so für Schritt *4* und *6* verwendet werden.

Außerdem werden *Combiner* für Schritt *4* und *6* definiert. *Combiner* führen eine Voraggregation von Datenmengen aus und sollten daher stets vor *GroupReduce*- oder *CoGroup*-Operationen eingebaut werden. Sie sind zwar für die Semantik des Programms optional, können aber entscheidende Vorteile in der Performance bringen. [11]

Distanz- vor Ähnlichkeitsberechnung

Muss beim Matching (Schritt *6*) für ein Paar von Index-und Query-Datensatz die genaue Ähnlichkeit $sim_j(q, i)$ berechnet werden, so wird zuvor deren Distanz $d_h(q, i)$ bestimmt und mit $rad(q)$ verglichen (wie in Abbildung 3.3 bei Schritt *6* dargestellt). Die Berechnung von d_h ist schneller als jene von sim_j, da für die Berechnung von d_h der *XOR*-Operator verwendet werden kann. So können die allermeisten Paare aufgrund ihrer zu hohen Distanz ausgeschlossen werden. Da d_h und sim_j (normiert auf [0,1]) allerdings in unterschiedlichen Zahlenbereichen arbeiten und nicht direkt ineinander übertragbar sind, muss die exakte Ähnlichkeit für die wenigen übrigen Paare dennoch berechnet werden. Insgesamt ist diese Abfolge aber effizienter, als für jedes Datenpaar nur die Ähnlichkeit zu berechnen.

4 Evaluation

Dieses Kapitel stellt die in dieser Arbeit erzielten Ergebnisse vor. In Abschnitt 4.1 werden der experimentelle Aufbau sowie die gemessenen Metriken beschrieben, während in Abschnitt 4.2 die genauen Evaluationsergebnisse bezüglich der verschiedenen Parameter dargestellt und interpretiert werden.

4.1 Messungen

4.1.1 Aufbau

Die in diesem Kapitel beschriebenen Messungen wurden auf einem Cluster mit den folgenden Eigenschaften durchgeführt:

- Anzahl Knoten: 16
- pro Knoten: 12 CPU-Kerne, 6 Task Slots, 40 GB JVM Heap Memory
- Hadoop-Version: 2.6.0
- Flink-Version: 1.1.2

Die verwendeten Testdaten liegen als CSV-Datei im Hadoop Distributed File System (HDFS) vor. Sie entstammen einer realen Datenquelle. Dabei wurden als Query-Datensätze die originalen Daten kopiert und teilweise korrumpiert, d.h. mit Fehlern versehen. Anhand der IDs kann die Richtigkeit des Ergebnisses geprüft werden. (Gleiche IDs sollten einander zugeordnet werden.) Die Daten liegen in 10 verschiedenen Größenordnungen vor, die Anzahl Index-Datensätze n_I reicht von 80.000 bis 800.000. Das Verhältnis zur Anzahl der Query-Datensätze n_Q ist stets $n_I : n_Q = 80 : 20$. Im Folgenden wird als Anzahl der Datensätze die Summe aus n_I und n_Q bezeichnet. (Diese liegt demnach zwischen 100.000 und 1.000.000.)

4.1.2 Parameter

Wie in Abschnitt 3.2.2 beschrieben, berücksichtigt das entwickelte Programm verschiedene Parameter, von welchen die folgenden anhand unterschiedlicher Werte evaluiert wurden:

- Grad der Parallelität N
- Pivot-Zuordnungsstrategie (*lokal*/*global*)
- lokale Pivot-Strategie (*random*/*ms*/*fft*)
- globale Pivot-Strategie (*none*/*ms*/*fft*)
- Anzahl der Pivots lokal n_L
- Anzahl der Pivots global n_P

Der Ähnlichkeits-Threshold t wurde bei allen Messungen auf 0,8 gesetzt. Dieser Wert wird üblicherweise bei PPRL verwendet. Ein höherer t beschleunigt das Programm unabhängig von den anderen Parametern, da die Ähnlichkeitsradien kleiner werden und somit weniger Vergleiche ausgeführt werden müssen.

4.1.3 Metriken

Um einen detaillierten Einblick in den Ablauf des Programms mit unterschiedlichen Parametern zu bekommen, wurden für jeden Durchlauf verschiedenste Metriken gemessen, die im folgenden aufgelistet sind. Die Bestimmung der Metriken erfolgte mit den *Akkumulatoren* von Flink [9], welche es ermöglichen, innerhalb der einzelnen Funktionen globale Zähler zu steuern. Diese lassen sich nach Abschluss des Programms auslesen.

- **ausgeführte Vergleiche beim Finden der Pivots (V_P)**
  V_P zählt die Durchläufe der inneren *for*-Schleifen der Algorithmen zum Finden der Pivot-Elemente (vgl. Abbildung 2.8 und Abbildung 2.9) und somit die Anzahl der ausgeführten Distanzvergleiche zwischen Datensätzen innerhalb der Index-Datenquelle.
  V_L gibt die entsprechende Zahl für das Finden der lokalen Pivots (vgl. Schritt *1* in Abschnitt 3.2.1), V_G für das Finden der globalen Pivots (vgl. Schritt *2* in Abschnitt 3.2.1) an. Somit ist $V_P = V_L + V_G$.

- **ausgeführte Vergleiche mithilfe der Dreiecksungleichung zwischen I und Q (V_T)**
  V_T gibt an, wie oft der Vergleich mithilfe der Dreiecksungleichung (vgl. Gleichung 2.7) in Schritt *6* (Abschnitt 3.2.1) ausgeführt wurde.

- **ausgeführte Distanzvergleiche zwischen I und Q (V_D)**
  V_D gibt an, wie oft eine Distanzberechnung $d_h(i, q)$ in Schritt *6* (Abschnitt 3.2.1) ausgeführt wurde.

- **ausgeführte Ähnlichkeitsvergleiche zwischen I und Q (V_S)**
  V_S gibt an, wie oft eine Ähnlichkeitsberechnung $sim_j(i, q)$ in Schritt *6* (Abschnitt 3.2.1) ausgeführt wurde.

  V_T, V_D und V_S werden für Berechnungen bezüglich des Blockings benötigt. Damit lässt sich zeigen, wie effizient das Programm arbeitet, d.h. wie viele Vergleiche benötigt werden, um alle matchenden Datensatz-Paare zu finden.

- **durchschnittliche Anzahl von Kopien (Anzahl zugeordneter Pivots) pro Query-Datensatz (C_Q)**
  Die Größe C_Q gibt Aufschluss darüber, wie vielen Pivots ein Query-Datensatz in Schritt *5* (Abschnitt 3.2.1) durchschnittlich zugeordnet wurde, d.h. wie viele Kopien dieses Query-Datensatzes erzeugt werden mussten. Da das Erstellen und Senden von Kopien besonders bei verteilt liegenden Daten eine relativ teure Operation ist, wird versucht, diese Zahl möglichst gering zu halten.

- **durchschnittliche Anzahl von Kandidaten (Index-Datensätze) pro Query-Datensatz (K_Q)**
  K_Q ist proportional zu V_D ($K_Q = \frac{V_D}{n_Q}$) und gibt an, mit wie vielen Index-Datensätzen ein Query-Datensatz in Schritt *6* (Abschnitt 3.2.1) durchschnittlich bezüglich d_h verglichen wird.

- **Laufzeit**
  Die Laufzeit des Programms ist die zentrale Größe, welche in dieser Arbeit optimiert werden soll. Die angegebenen Laufzeiten wurden in der in Abschnitt 4.1.1 beschriebenen Umgebung gemessen und können unter anderen Voraussetzungen entsprechend abweichen.

4.1.4 Qualität der Ergebnisse

Alle bisher beschrieben Metriken sind Anhaltspunkte für die Performance des Programms. Für die Messung der Qualität werden für Record Linkage die folgenden Metriken verwendet:

- **Recall** $= \frac{gefundene\ echte\ Matches}{existierende\ Matches}$
- **Precision** $= \frac{gefundene\ echte\ Matches}{gefundene\ Matches\ gesamt}$
- **F1-Score** $= 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$

Hinsichtlich der bereits verschlüsselten Datensätze ist der Recall immer garantiert 100%. Das heißt, es werden alle Paare von Datensätzen gefunden, deren Ähnlichkeit größer oder gleich t ist. Zur Precision lässt sich hier keine Aussage treffen.

Betrachtet man die echten Datensätze, so lassen sich mit den verwendeten Testdaten genaue Zahlen für Recall, Precision und F1-Score berechen. Anhand der IDs der Daten kann überprüft werden, wie viele der gefundenen Matches wahre Matches sind (wenn die IDs übereinstimmen) und wie viele Paare der gleichen ID nicht als Match vom Programm gefunden worden. Es können anhand der Testdaten beispielhaft folgende Ergebnisse errechnet werden:

- Recall = 94,1%
- Precision = 96,4%
- F1-Score = 95,2%

Die Qualität wird in der Evaluation nicht weitergehend betrachtet, da sie unabhängig von den gewählten Parametern gleich ist und maßgeblich von der gegebenen Art der Verschlüsselung abhängt. Die Metriken unterscheiden sich außerdem grundsätzlich nicht von jenen bei nicht verteilten Verfahren.

Die Richtigkeit der Ergebnisse (d.h. ob tatsächlich alle Paare von verschlüsselten Datensätzen gefunden wurden, deren Ähnlichkeit größer oder gleich t ist) kann anhand der Testdaten mit Hilfe der Anzahl der gefunden Matches ermittelt werden. Diese Anzahl ist vorgegeben und wurde bei der Verwendung aller Parameter-Kombinationen überprüft.

4.2 Ergebnisse

Die Evaluation erfolgte in mehreren Teilschritten. Jeder Schritt konzentriert sich auf einen bestimmten Parameter, wobei die anderen Parameter konstant gehalten werden. Zunächst wurde so pro Strategie die ideale Anzahl von Pivots ermittelt (Abschnitt 4.2.1). Anschließend erfolgte ein genauer Vergleich zwischen den verschiedenen Strategien zum Finden der Pivots (Abschnitt 4.2.2). Mit den besten Strategien und der optimalen Anzahl von Pivots konnten schließlich die Skalierbarkeit (Abschnitt 4.2.3) und die Parallelität (Abschnitt 4.2.4) genauer untersucht werden. Zunächst sind in Abschnitt 4.2.1 und Abschnitt 4.2.2 der Grad der Parallelität mit $N = 8$ und die Anzahl der Datensätze mit $n_I + n_Q = 1.000.000$ festgehalten. Abschließend werden in Abschnitt 4.2.5 die Ergebnisse dieser Arbeit mit dem zu Grunde liegenden nicht verteilten Verfahren verglichen.

Strategie zum Zuordnen der Pivots

Versuche mit der *lokalen* Pivot-Zuordnungsstrategie (vgl. Abschnitt 3.2.2) ergaben mit allen getesteten Parameterkombinationen ein Vielfaches der Laufzeiten mit der **globalen** Pivot-Zuordnungsstrategie. Der Hauptgrund hierfür ist eine wesentlich höhere Anzahl von Vergleichen beim Matching, welche aus der lokalen Zuordnung resultiert. Die lokale Strategie wird daher nicht weiter betrachtet. Bei allen folgenden Ergebnissen ist die Pivot-Zuordnungsstrategie auf *global* festgesetzt.

4.2.1 Anzahl der Pivots

Sehr entscheidende Parameter für die Laufzeit sind die lokale und globale Anzahl der Pivots. Wie vorherige Arbeiten gezeigt haben, sollte die globale Anzahl der Pivots etwa der Dimensionalität der Bit-Arrays (im vorliegenden Fall 1.000) entsprechen [6]. Weiterhin geht aus Experimenten mit dem dieser Arbeit zu Grunde liegenden nicht verteilten Verfahren hervor, dass für eine Anzahl der Datensätze von 500.000 sehr stabile Laufzeiten mit Pivot-Anzahlen zwischen 1.000 und 4.000 erzielt werden können [4].

Aufgrund der Parallelisierung und der Vorauswahl von Pivots in den lokalen Partitionen entsteht hier aber eine neue Dynamik. Es ist zu erwarten, dass sich die

optimale Anzahl der globalen Pivots ebenfalls in den beschriebenen Größenordnungen bewegt. Für die Anzahl der auszuwählenden lokalen Pivots lässt sich aber keine Voraussage treffen.

Alle Messungen in diesem Abschnitt wurden mit der festgehaltenen Anzahl der Datensätze von $n_I + n_Q = 1.000.000$ und einer Parallelität von $N = 8$ durchgeführt.

Als erste Erkenntnis der Experimente lässt sich festhalten, dass die ideale Anzahl der Pivots (global und lokal) sehr stark von der gewählten Strategie zum Finden der Pivots abhängt. Für jede mögliche Strategie wurden daher verschiedene Anzahlen getestet. Die Kombinationen, welche die besten Laufzeiten pro Strategie erzielten, werden in Abschnitt 4.2.2 untereinander verglichen. Im folgenden werden beispielhaft die Strategien *fft-fft* und *random-fft* näher betrachtet.

Strategie *fft-fft*

Abbildung 4.1 zeigt verschiedene Metriken in Abhängigkeit von der lokalen Anzahl der Pivots n_L und der globalen Anzahl der Pivots n_P. Die farbliche Unterlegung zeigt bessere Werte in grün und schlechtere Werte in rot. In Tabelle A ist zu sehen, dass das Optimum der Laufzeit mit 120 s bei der Kombination $(n_L, n_P) = (2.000, 6.000)$ erzielt wurde. Es ist außerdem zu erkennen, dass je weiter man sich in der Tabelle von dem Optimum weg bewegt, die Laufzeit entsprechend schlechter wird und es somit keinen anderen Bereich gibt, der potentiell eine bessere Laufzeit erzielen kann. Die Gründe für diese Ergebnisse sind in den anderen Metriken zu finden.

Abbildung 4.1, Tabelle B zeigt die benötigten Vergleiche zum Finden der Pivots V_P in Millionen. Offensichtlich erhöht sich diese Zahl je mehr Pivots zu bestimmen sind. Für V_P ist vor allem n_L ausschlaggebend, wobei dies etwas zu relativieren ist, da die Vergleiche zum Finden der lokalen Pivots natürlich pro Partition und somit parallel stattfinden. Mit der Erhöhung von n_L erhöht sich aber nicht nur die Zahl der Vergleiche pro Partition, sondern auch maßgeblich die benötigte Zahl der Vergleiche beim globalen (nicht parallelen) Schritt des Findens des Pivots. Sollen beispielsweise $n_P = 6.000$ Pivots ausgewählt werden, so dauert dies entsprechend länger wenn insgesamt $n_L \cdot N = 4.000 \cdot 8 = 32.000$ lokale Pivots gefunden wurden, als wenn es nur $n_L \cdot N = 500 \cdot 8 = 4.000$ sind. Es hat sich außerdem herausgestellt, dass n_P Auswirkungen auf die Laufzeiten beim Zuordnen der Datensätze (Index und Query)

zu den Pivots hat. Bei mehr globalen Pivots nimmt dieser Schritt entsprechend mehr Zeit in Anspruch.

Strategie	fft-fft
Parallelität N	8
Anzahl Datensätze n_I+n_Q	1.000.000

A

Laufzeit [s]		Anzahl Pivots lokal n_L				
		500	**1.000**	**2.000**	**3.000**	**4.000**
Anzahl Pivots global n_P	**2.000**	195	182	196	210	217
	3.000	183	167	190	194	197
	4.000	180	149	179	190	189
	5.000	179	145	145	157	170
	6.000	180	146	120	140	177
	7.000	191	146	128	150	180

B

Mio. Vergleiche beim Finden von Pivots V_P		Anzahl Pivots lokal n_L				
		500	**1.000**	**2.000**	**3.000**	**4.000**
Anzahl Pivots global n_P	**2.000**	405	810	1.614	2.410	3.198
	3.000	407	816	1.628	2.432	3.228
	4.000	407	820	1.640	2.452	3.256
	5.000	407	824	1.652	2.472	3.284
	6.000	407	826	1.662	2.490	3.310
	7.000	407	828	1.672	2.508	3.336

C

Kopien pro Query-Datensatz C_Q		Anzahl Pivots lokal n_L				
		500	**1.000**	**2.000**	**3.000**	**4.000**
Anzahl Pivots global n_P	**2.000**	404	344	340	333	329
	3.000	429	316	294	281	278
	4.000	425	298	251	242	238
	5.000	424	285	222	213	208
	6.000	425	278	205	192	187
	7.000	424	264	194	179	172

D

Kandidaten pro Query-Datensatz K_Q		Anzahl Pivots lokal n_L				
		500	**1.000**	**2.000**	**3.000**	**4.000**
Anzahl Pivots global n_P	**2.000**	21.378	19.323	19.235	19.855	19.699
	3.000	15.586	12.611	12.895	12.437	12.335
	4.000	12.349	9.988	9.463	9.258	9.085
	5.000	12.357	8.345	7.506	7.518	7.451
	6.000	12.349	7.450	6.395	6.400	6.325
	7.000	12.349	6.675	5.743	5.620	5.521

Abbildung 4.1: Metriken für 1 Million Datensätze mit der Strategie *fft-fft* in Abhängigkeit von der Anzahl der Pivots (lokal und global)

Die Zahlen in Abbildung 4.1, Tabelle C und D verhalten sich gegenteilig zu V_P. Generell gilt, desto mehr globale Pivots ausgewählt werden, umso kleiner werden deren Radien $rad(p)$. Daher nimmt die Anzahl von Kopien pro Query-Datensatz C_Q (d.h. wie vielen Pivots ein Query-Datensatz zugeordnet wird) ab, obwohl die Zahl der Pivots steigt. Deshalb und weil jedem Pivot auch weniger Index-Datensätze zugeordnet sind, sinkt außerdem die Anzahl der Kandidaten pro Query-Datensatz K_Q (d.h. die Anzahl der letztlich ausgeführten Vergleiche). Weiterhin sind bei mehr Pivots bereits mehr Distanzen vorberechnet. Eine höhere Anzahl lokaler Pivots verbessert die Qualität der globalen Pivots, da die Menge, aus denen diese ausgewählt werden, größer ist.

Verdeutlicht werden die Erkenntnisse ebenfalls durch Abbildung 4.2. Hier sind die Laufzeiten der einzelnen Teilschritte (vgl. Nummerierung in Abschnitt 3.2.1) des Programms für beispielhafte Werte von n_L und n_P dargestellt. Diese Teillaufzeiten sind allerdings mit gewisser Vorsicht zu behandeln, da die einzelnen Programmabschnitte in Flink nicht nacheinander sondern sich überschneidend ablaufen und so keinesfalls in Summe die Gesamtlaufzeit ergeben. Einige Teilschritte werden bereits gestartet, wenn die ersten Ergebnisse vorliegen und müssen dann unter Umständen auf vorige Prozesse warten. Abbildung 4.3 veranschaulicht einen solchen Ablauf (hier für das Programm mit der optimalen Laufzeit). Die Teillaufzeit für Schritt *4* lag bei allen Experimenten unter 1 s und wird daher in Abbildung 4.2 nicht aufgeführt.

Strategie	fft-fft
Parallelität N	8
Anzahl Datensätze n_I+n_Q	1.000.000

Anzahl Pivots		Laufzeit [s]						
lokal n_L	global n_P	(1) Finden Pivots lokal (Map-Partition)	(2) Finden Pivots global (Group-Reduce)	(3) Zuordnung Pivots zu Index (Map)	(5) Zuordnung Pivots zu Query (Flat-Map)	(6) Matching (Co-Group)	Ausgabe der Resultate (DataSink)	**gesamt (nicht kumulativ)**
500	**5.000**	9	6	37	63	140	80	**179**
4.000	**5.000**	51	73	104	121	63	45	**170**
1.000	**6.000**	15	13	49	70	93	48	**146**
2.000	**6.000**	26	29	65	82	52	35	**120**

Abbildung 4.2: detaillierte Laufzeiten für 1 Million Datensätze mit der Strategie *fft-fft* in Abhängigkeit von der Anzahl der Pivots (lokal und global)

Die Teillaufzeiten aus Abbildung 4.2 geben dennoch einige Anhaltspunkte über den Ablauf des Programms. Wie bereits beschrieben, bedingt vor allem eine höhere Zahl lokaler Pivots längere Laufzeiten für das Finden von Pivots (Spalte *1* und *2*). Auch

der Gesamtprozess der Zuordnung der Pivots (Spalte *3* und *5*) wird so verlangsamt. Die Laufzeit des Matchings (Spalte *6*) ist vor allem abhängig von K_Q.

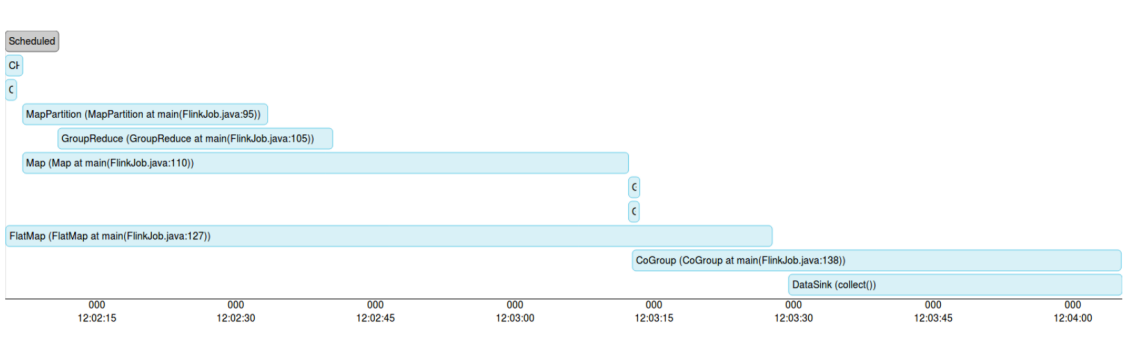

Abbildung 4.3: Zeitleiste für den Ablauf des Programms mit 1 Million Datensätzen, Strategie *fft-fft* und $(n_L, n_P) = (2.000, 6.000)$

Zusammenfassend ist festzuhalten, dass es auf das genaue Zusammenspiel mehrerer Faktoren ankommt, um eine optimale Laufzeit zu erzielen. Mehr Pivots bringen zweifelsfrei Vorteile beim Matching. Der Zugewinn (zum Beispiel an weniger K_Q) steigt aber nicht konstant und wird immer geringer je mehr Pivots verwendet werden. Die kontinuierlich steigenden Kosten beim Finden und Zuordnen der Pivots können daher irgendwann nicht mehr aufgewogen werden. Deshalb ist für die Strategie *fft-fft* das Optimum bei $(n_L, n_P) = (2.000, 6.000)$ zu finden. Größere Anzahlen von Pivots bringen keinen Laufzeitgewinn mehr mit sich.

Strategie *random-fft*

Ein Blick auf die Laufzeiten mit der Strategie *random-fft* (Abbildung 4.4, Tabelle A) zeigt, dass das Optimum stark von der gewählten Strategie zum Finden der Pivots abhängt. Es ist hier eine völlig andere Situation vorzufinden, als bei der Strategie *fft-fft*. Die Laufzeiten für $n_L \in \{3.000, 4.000, 5.000\}$ und $n_P \in \{3.000, 4.000, 5.000\}$ sind weitestgehend konstant. Das Optimum wurde experimentell bei $(n_L, n_P) = (4.000, 3.000)$ gefunden.

Da die lokalen Pivots bei dieser Strategie zufällig ausgewählt werden, sind auf lokaler Ebene keine Vergleiche zum Finden der Pivots nötig. Dementsprechend sind die Werte für V_P (Abbildung 4.4, Tabelle B) nur ein Bruchteil der Werte bei fft-fft. Die Zeit für Schritt *1* ist damit nahezu Null, was Abbildung 4.5 zeigt (Strich über GroupReduce).

Strategie	random-fft
Parallelität N	8
Anzahl Datensätze n_I+n_Q	1.000.000

A

Laufzeit [s]		Anzahl Pivots lokal n_L				
		1.000	**2.000**	**3.000**	**4.000**	**5.000**
Anzahl Pivots global n_P	**2.000**	248	193	174	185	176
	3.000	223	179	164	163	172
	4.000	225	176	170	165	165
	5.000	239	175	168	166	164
	6.000	250	184	183	167	173

B

Mio. Vergleiche beim Finden von Pivots V_P		Anzahl Pivots lokal n_L				
		1.000	**2.000**	**3.000**	**4.000**	**5.000**
Anzahl Pivots global n_P	**2.000**	14	30	46	62	78
	3.000	20	44	68	92	116
	4.000	24	56	88	120	152
	5.000	28	68	108	148	188
	6.000	30	78	126	174	222

C

Kopien pro Query-Datensatz C_Q		Anzahl Pivots lokal n_L				
		1.000	**2.000**	**3.000**	**4.000**	**5.000**
Anzahl Pivots global n_P	**2.000**	616	499	455	433	415
	3.000	695	511	464	425	400
	4.000	727	510	451	413	391
	5.000	775	514	451	403	376
	6.000	814	530	450	402	380

D

Kandidaten pro Query-Datensatz K_Q		Anzahl Pivots lokal n_L				
		1.000	**2.000**	**3.000**	**4.000**	**5.000**
Anzahl Pivots global n_P	**2.000**	16.239	14.352	14.403	14.415	14.486
	3.000	12.104	10.343	10.156	10.054	9.759
	4.000	10.249	8.408	8.105	7.937	7.817
	5.000	9.189	7.393	6.876	6.663	6.552
	6.000	8.530	6.611	6.157	5.922	5.838

Abbildung 4.4: Metriken für 1 Million Datensätze mit der Strategie *random-fft* in Abhängigkeit von der Anzahl der Pivots (lokal und global)

Auch bei *random-fft* ist festzustellen, dass C_Q und K_Q mit steigender Anzahl Pivots abnehmen (Abbildung 4.4, Tabelle C und D). Allerdings sind hier die Werte im Vergleich zu *fft-fft* um einiges höher (vor allem bei C_Q). Daraus folgt, dass die Pivots bei diesem Ansatz qualitativ schlechter sind. Demnach wird auch für das Matching eine längere Zeit benötigt.

Insgesamt gleichen sich bei dieser Strategie die Vor- und Nachteile von mehr Pivots in einem bestimmten Bereich aus, so dass es kein klares Optimum, sondern eher einen Bereich optimaler Laufzeiten gibt.

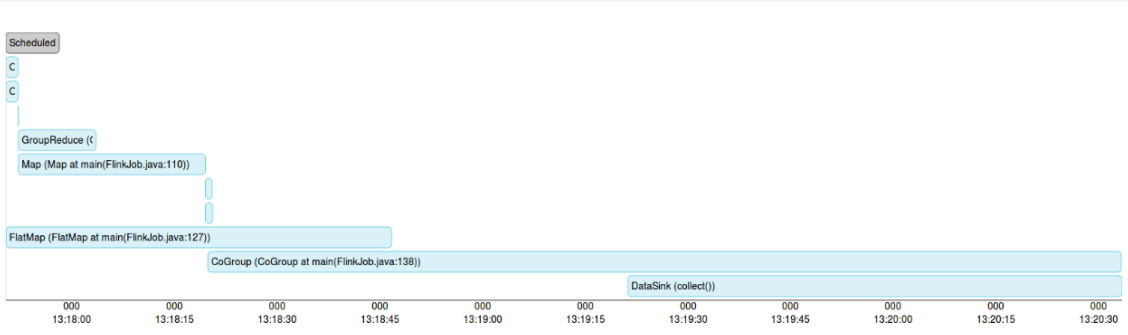

Abbildung 4.5: Zeitleiste für den Ablauf des Programms mit 1 Million Datensätzen, Strategie *random-fft* und $(n_L, n_P) = (4.000, 3.000)$

4.2.2 Strategie zum Finden der Pivots

Für jede mögliche Strategie zum Finden der Pivots wurde jeweils das Paar (n_L, n_P) bestimmt, welches die beste Gesamtlaufzeit liefert. Die Ergebnisse sind in Abbildung 4.6 (A,B,C und D stehen für die gleichen Metriken wie in Abschnitt 4.2.1) dargestellt, wobei die Legende Aufschluss über die genauen Anzahlen der Pivots gibt.

Grundsätzlich unterscheiden sich die Strategien insofern, dass jene mit *random-_* (blaue Balken) keine Vergleiche oder Zeit zum Finden der Pivots auf lokaler Ebene benötigen. Jene mit *_-none* (mittlere Balken bei rot, blau und grün) benötigen keine Vergleiche oder Zeit zum Finden der Pivots auf globaler Ebene. Unter Einbeziehung der genauen Anzahl der Pivots ist dies in Abbildung 4.6, Tabelle B zu erkennen. Noch deutlicher wird dies in Abbildung 4.7, wo erneut die Teillaufzeiten der einzelnen Funktionen des Programms abgebildet sind (Bemerkungen zur Darstellung siehe Abschnitt 4.2.1). Beispielsweise haben die Schritte *1* und *2* bei *random-none* eine

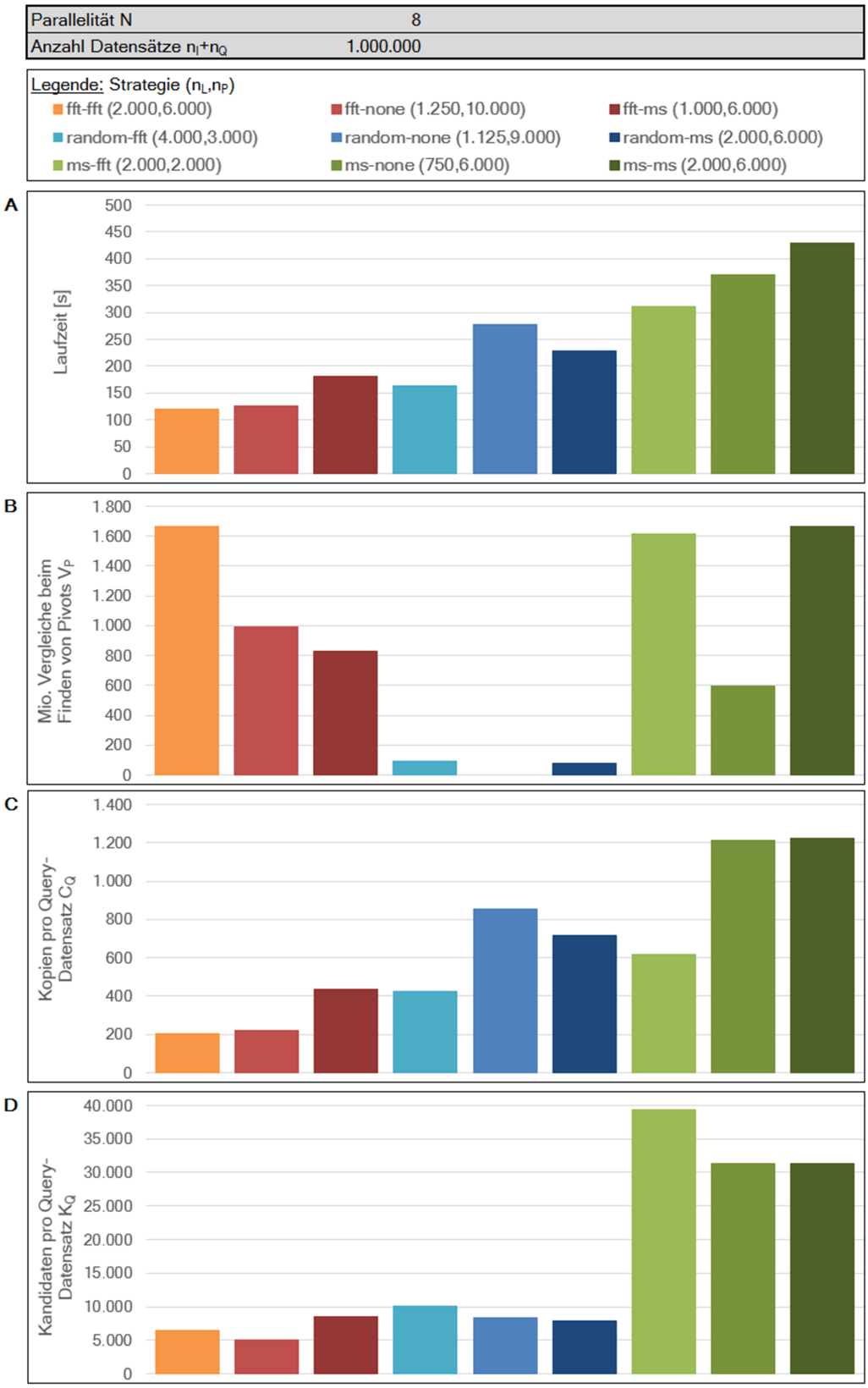

Abbildung 4.6: Metriken für bestes Ergebnis pro Strategie zum Finden der Pivots für 1 Million Datensätze

genäherte Laufzeit von 0 s. Diese Strategie kann gut als Referenz angesehen werden, denn jeder betriebene Aufwand beim Finden der Pivots sollte zumindest eine bessere Laufzeit hervorbringen als die komplett willkürliche Auswahl von Pivots.

Dass dies keinesfalls bei allen Experimenten der Fall war, zeigen die Strategien, welche auf lokaler Ebene *ms* verwenden (grüne Balken in Abbildung 4.6). Diese überraschenden Ergebnisse legen nahe, dass *ms* auf lokaler Ebene die Vorauswahl der Pivots negativ beeinflusst, was im Vergleich von *ms-none* mit *fft-none* zu erkennen ist. Besonders die Werte von K_Q (Abbildung 4.6 D) sind für die drei Strategien mit *ms-_* enorm hoch, was eine schlechte Verteilung der Pivots impliziert. Daher sind die Laufzeiten beim Matching hier extrem lang (vgl. Abbildung 4.7, Spalte *6*). Die genauen Ursachen für dieses Verhalten ließen sich nicht ausmachen, jedoch haben weitere Tests mit komplett anderen Datensätzen ein sehr ähnliches Ergebnis gezeigt.

Parallelität N		8								
Anzahl Datensätze n_I+n_Q		1.000.000								
Strategie		Anzahl Pivots		Laufzeit [s]						
lokal	global	lokal n_L	global n_P	(1) Finden Pivots lokal (Map-Partition)	(2) Finden Pivots global (Group-Reduce)	(3) Zuordnung Pivots zu Index (Map)	(5) Zuordnung Pivots zu Query (Flat-Map)	(6) Matching (Co-Group)	Ausgabe der Resultate (DataSink)	**gesamt (nicht kumulativ)**
fft	fft	**2.000**	**6.000**	26	29	65	82	52	35	**120**
fft	none	**1.250**	**10.000**	18	0	70	94	53	30	**126**
fft	ms	**1.000**	**6.000**	15	14	51	81	129	65	**182**
random	fft	**4.000**	**3.000**	0	11	27	56	133	72	**163**
random	none	**1.125**	**9.000**	0	0	46	150	228	117	**278**
random	ms	**2.000**	**6.000**	0	8	39	125	185	96	**228**
ms	fft	**2.000**	**2.000**	30	29	44	119	271	190	**312**
ms	none	**750**	**6.000**	13	0	44	174	324	181	**371**
ms	ms	**2.000**	**6.000**	31	36	71	230	354	188	**429**

Abbildung 4.7: detaillierte Laufzeiten für bestes Ergebnis pro Strategie zum Finden der Pivots für 1 Million Datensätze

Auf globaler Ebene schneidet *ms* etwas schlechter ab als fft (wie bereits in [6] festgestellt wurde). Vergleicht man die drei Strategien mit *random-_*, so sieht man, dass *ms* durchaus Vorteile auf globaler Ebene gegenüber einer komplett zufälligen Auswahl von Pivots bringt. *fft* ist hier aber noch besser und funktioniert weiterhin auch zur Vorauswahl auf lokaler Ebene sehr gut, was die drei Strategien mit *fft-_* zeigen. Man kann zusammenfassend sagen, dass der MS-Algorithmus in gewisser Weise globales Wissen voraussetzt, während der FFT-Algorithmus auch lokal gute Ergebnisse für die Weiterverarbeitung bringt.

Die eindeutig besten Laufzeiten erzielten die beiden Stratagien *fft-fft* (120 s) und *fft-none* (126 s). Die Optimierung der Pivotanzahl von *fft-fft* auf $(n_L, n_P) = (2.000, 6.000)$ wurde ausführlich in Abschnitt 4.2.1 beschrieben. Da bei *fft-none* kein globaler (nicht paralleler) Verarbeitungsschritt beim Finden der Pivots nötig ist, liegt hier die optimale Anzahl deutlich höher bei $n_P = 10.000$ (n_L ergibt sich bei dieser Strategie aus $n_L = \frac{n_P}{N} = \frac{10.000}{8} = 1.250$). Hier können die Vorteile, welche mehr Pivots für das Matching bringen, bis zu einer höheren Anzahl ausgenutzt werden. Daher ist der Wert für K_Q (Abbildung 4.6 D) sogar geringer als bei *fft-fft*. Allerdings zeigt Abbildung 4.7 hier erneut, dass mehr Pivots auch mehr Zeit bei der Zuordnung (Spalte *3* und *4*) in Anspruch nehmen, was letztlich ausschlaggebend für die höherer Gesamtlaufzeit ist. Beim Matching gibt es zwischen den beiden Strategien kaum Unterschiede.

Je nach Anwendungsfall und genauer Datenlage sind für das Verfahren die Strategien *fft-fft* und *fft-none* zu empfehlen. Eine genauere Betrachtung bei steigendem Grad der Parallelität mit diesen beiden Strategien ist in Abschnitt 4.2.4 zu finden.

Blocking

Abbildung 4.8 zeigt für die besten Laufzeitergebnisse mit den Strategien *fft-fft* und *fft-none* weitere detaillierte Metriken. Die Vergleiche zum Finden der Pivots (Zeilen 7-9) wurden bereits betrachtet.

Die Zeilen 10 und 11 geben Aufschluss darüber, wie die Verteilung der Pivots auf Seite der Index-Daten aussieht. Da es bei *fft-none* mehr Pivots gibt, sind durchschnittlich weniger Index-Datensätze einem Pivot zugeordnet. Als Folge ist auch der durchschnittliche Radius der Pivots kleiner. Interessant ist die Anzahl der komplett vom Vergleich mit der Index-Datenquelle ausgeschlossenen Query-Datensätze (Zeile 12, vgl. auch Gleichung 2.6). Aufgrund der geringeren Radien der Pivots sind dies bei *fft-none* etwas mehr Datensätze, jedoch ist die Anzahl verglichen mit der Mächtigkeit der gesamten Datenquelle verschwindend gering. Eine weitere Folge der geringeren Pivotradien ist, dass den Pivots weniger Query-Datensätze zugeordnet werden (Zeile 13). Wichtiger ist in diesem Zusammenhang jedoch die bereits zuvor betrachtete durchschnittliche Anzahl von Kopien pro Query-Datensatz (C_Q, Zeile 14), welche sich aus $C_Q = \frac{\textit{(Query-Datensätze pro Pivot)} \cdot n_P}{n_Q}$ ergibt und bei *fft-fft* geringer

ist. Diese Zahl ist letzlich ein Maß für die Qualität der Verteilung der Pivots auf Seite der Query-Daten.

	Strategie	fft-fft	fft-none
1	Parallelität N	8	
2	Anzahl Index-Datensätze n_I	800.000	
3	Anzahl Query-Datensätze n_Q	200.000	
4	maximal mögliche Vergleiche V_{max}	160.000.000.000	
5	Anzahl Pivots lokal n_L	2.000	1.250
6	Anzahl Pivots global n_P	6.000	10.000
7	Vergleiche zum Finden der Pivots lokal V_L	1.584.008.000	993.755.000
8	Vergleiche zum Finden der Pivots global V_G	78.003.000	0
9	Vergleiche zum Finden der Pivots gesamt V_P	1.662.011.000	993.755.000
10	Ø Index-Datensätze pro Pivot	133	80
11	Ø Pivotradius rad(p)	230	220
12	geblockte Query-Datensätze ohne Pivot	4	14
13	Ø Query-Datensätze pro Pivot	6.848	4.415
14	Ø Kopien pro Query-Datensatz C_Q	205	219
15	Vergleiche mit Dreiecksungleichung V_T	9.334.686.627	7.285.803.478
16	geblockte Vergleiche mit Dreiecksungleichung	94,17%	95,45%
17	Distanzvergleiche V_D	1.296.285.149	1.011.480.701
18	geblockte Distanzvergleiche gesamt	99,19%	99,37%
19	geblockte Distanzvergleiche durch Dreiecksungleichung	86,11%	86,12%
20	Ø Kandidaten pro Query-Datensatz K_Q	6.481	5.057
21	Ähnlichkeitsvergleiche V_S	1.801.158	1.801.158
22	geblockte Ähnlichkeitsvergleiche gesamt	99,9989%	99,9989%
23	geblockte Ähnlichkeitsvergleiche durch Distanzvergleiche	99,86%	99,82%
24	Matches	1.432.771	
25	Laufzeit	120s	126s

Abbildung 4.8: detaillierte Metriken für Strategien *fft-fft* und *fft-none*

Die erste wichtige Zahl für Aussagen über das Blocking ist die Anzahl der Vergleiche mit der Dreiecksungleichung (V_T, Zeile 15). Die maximal mögliche Zahl von Vergleichen zwischen der Index-Datenquelle I und der Query-Datenquelle Q lässt sich mit $V_{max} = n_I \cdot n_Q$ (Zeile 4) berechnen. Mit $1 - \frac{V_T}{V_{max}}$ (Zeile 16) kann errechnet werden, wie viele Vergleiche bereits durch die Verteilung der Query-Datensätze auf die Pivots geblockt werden konnten. Diese Zahlen sind mit rund 95% bei beiden Strategien ähnlich hoch, bei *fft-none* noch etwas besser aufgrund der höheren Anzahl der Pivots. Mit der Anwendung der Dreiecksungleichung findet die nächste Stufe des Blockings statt (vgl. auch Gleichung 2.7). Die Zahl der noch auszuführenden tatsächlichen Distanzvergleiche V_D zwischen I und Q ist in Zeile 17 angegeben. Dabei wurden

$1 - \frac{V_D}{V_T}$ (Zeile 19) Vergleiche durch die Dreiecksungleichung geblockt. Diese Zahl ist für beide Strategien mit 86% fast identisch. Insgesamt sind zu diesem Zeitpunkt bereits $1 - \frac{V_D}{V_{max}}$ (Zeile 18), d.h. mehr als 99% der möglichen Vergleiche geblockt worden. Für allgemeine Analysen ist die Zahl der Kandidaten pro Query-Datensatz ($K_Q = \frac{V_D}{n_Q}$, Zeile 20) leichter zu handhaben.

Mit der Berechnung der Distanz werden fast alle verbleibenden Paare als Match ausgeschlossen (Zeile 23). Somit werden nur relativ wenige Ähnlichkeitsvergleiche (Zeile 21) ausgeführt. Dies spart insgesamt Laufzeit, da die Ähnlichkeitsvergleiche etwas teurer sind als die Distanzvergleiche (siehe auch Abschnitt 3.2.3). Der Großteil der übrigen Paare erreichen den Threshold und werden als Matches (Zeile 24) ausgegeben. Die Anzahl der Matches muss bei beiden Strategien selbstverständlich gleich sein.

4.2.3 Skalierbarkeit

Ein wichtiger Aspekt bei Big-Data-Anwendungen ist die Skalierbarkeit. In diesem Abschnitt werden daher die Ergebnisse für die Strategie mit der besten Laufzeit (*fft-fft*) in Abhängigkeit von der Anzahl der Datensätze vorgestellt. Das zu Grunde liegende Problem des Auffindens ähnlicher Datensätze aus zwei Quellen hat grundsätzlich eine quadratische Komplexität (vgl. Abschnitt 2.1.4). Für die maximal mögliche Zahl von Vergleichen V_{max} ergeben sich für die verwendeten Testdaten Werte wie in Abbildung 4.9 dargestellt.

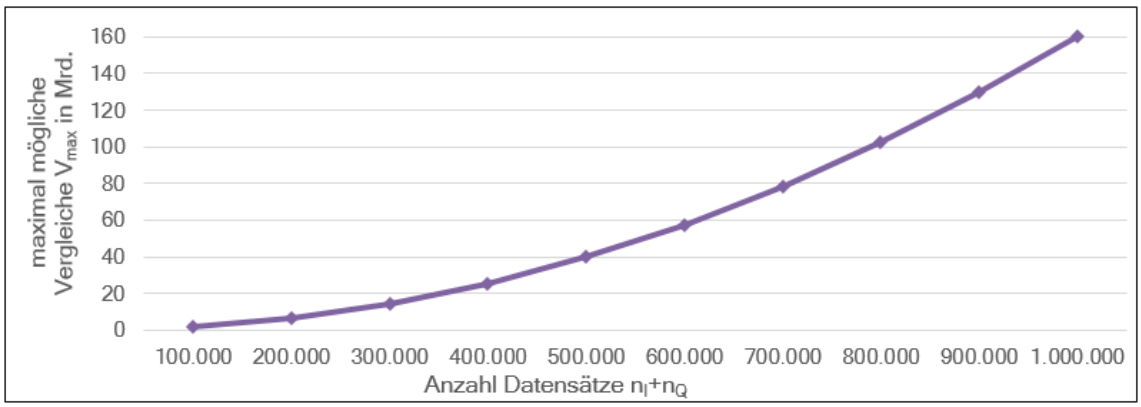

Abbildung 4.9: maximal mögliche Vergleiche V_{max} in Abhängigkeit von der Anzahl der Datensätze

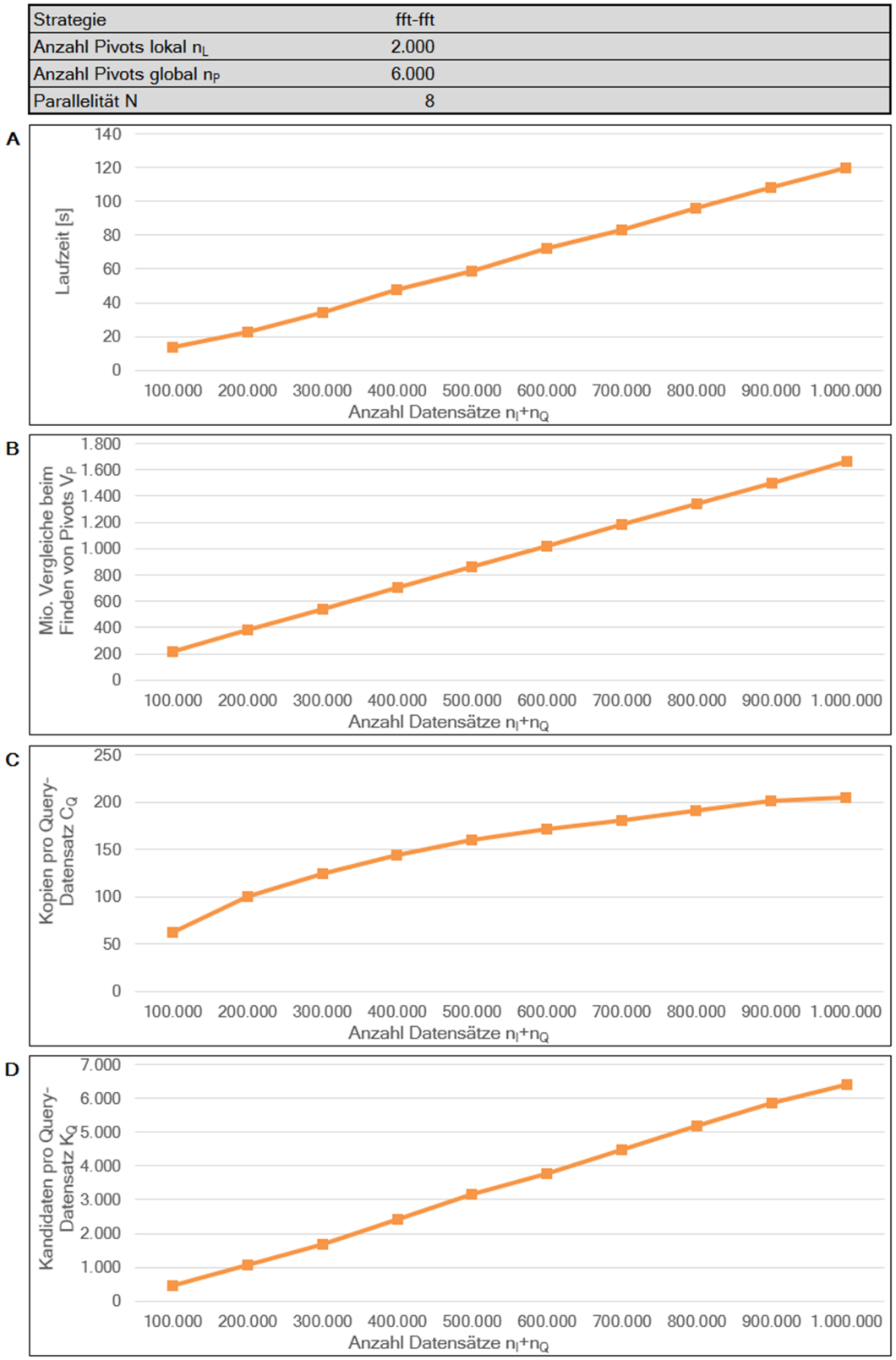

Abbildung 4.10: Metriken mit der Strategie *fft-fft* in Abhängigkeit von der Anzahl der Datensätze

Abbildung 4.10 zeigt die üblichen Metriken (wie in den Abschnitten zuvor) für die verschiedenen Datengrößen mit der Strategie *fft-fft*. Die Anzahl der Pivots ist fest bei $(n_L, n_P) = (2.000, 6.000)$. Es könnte daher vorkommen, dass es für die kleineren Datengrößen bessere Laufzeiten geben kann bei Verwendung anderer (n_L, n_P).

Die gemessenen Laufzeiten (Abbildung 4.10 A) zeigen einen nahezu idealen linearen Anstieg. Dies bedeutet, dass die Vorteile des Programms bei größer werdenden Datenmengen wachsen und die quadratische Komplexität des zu Grunde liegenden Problems entschärfen.

Der Hauptgrund hierfür liegt im Blocking von Vergleichen. K_Q (Abbildung 4.10 D) wächst im Vergleich zu V_{max} nur linear. C_Q (Abbildung 4.10 C) wächst sogar noch schwächer, was darauf hindeutet, dass die Verteilung der Pivots umso besser wird, desto größer die Anzahl der Datensätze ist.
Auch V_P (Abbildung 4.10 B) wächst wie erwartet nur linear, da die Laufzeit des FFT-Algorithmus bei fester Pivotanzahl nur von n_I abhängt (vgl. Abschnitt 2.3.3: Laufzeit FFT: $\mathbb{O}(n_I \cdot n_P)$ bzw. $\mathbb{O}(n_I \cdot n_L)$ auf lokaler Ebene).

4.2.4 Parallelität

Alle bisherigen Messungen bezogen sich auf eine Parallelität von $N = 8$. Abbildung 4.11 zeigt die Laufzeiten für die Strategie *fft-fft* in Abhängigkeit von verschiedenen Parallelitätsgraden. Dabei wurde in Bezug auf die vorigen Messungen n_P auf 6.000 gesetzt. n_L wurde so variiert, dass wie beim besten Ergebnis für $N = 8$ stets insgesamt 16.000 lokale Pivots erzeugt werden. Die genaue Anzahl richtet sich daher nach der Anzahl der Partitionen ($n_L = \frac{16.000}{N}$). Als Gegenprobe wurden Tests mit verschiedenen anderen Anzahlen von Pivots gemacht. Die dargestellten Ergebnisse verblieben aber als Bestwert für die jeweilige Parallelität.

Weiterhin wurde mit der Strategie *fft-none* getestet. Die gefundenen Bestwerte sind den Resultaten von *fft-fft* in Abbildung 4.12 gegenüber gestellt. Bei *fft-none* veränderte sich die ideale globale Pivotanzahl bei $N = 32$ und $N = 64$ auf $n_P = 16.000$.

Bei beiden Strategien ist offensichtlich zu erkennen, dass sich die Laufzeit der einzelnen Schritte bei höherer Parallelität verbessert. Dies ist in erster Linie auf die insgesamt erhöhte Rechenleistung bzw. die geringer werdende zu verarbeitende Datenmenge pro Partition zurückzuführen. Allerdings wird der Zugewinn an Laufzeit

pro Verdopplung von N immer geringer. Eine unbegrenzte Erhöhung der Parallelität scheint daher nicht vorteilhaft im Verhältnis zum erhöhten Aufwand.

Strategie	fft-fft
Anzahl Pivots lokal n_L	2.000 * 8 / N
Anzahl Pivots global n_P	6.000
Anzahl Datensätze n_I+n_Q	1.000.000

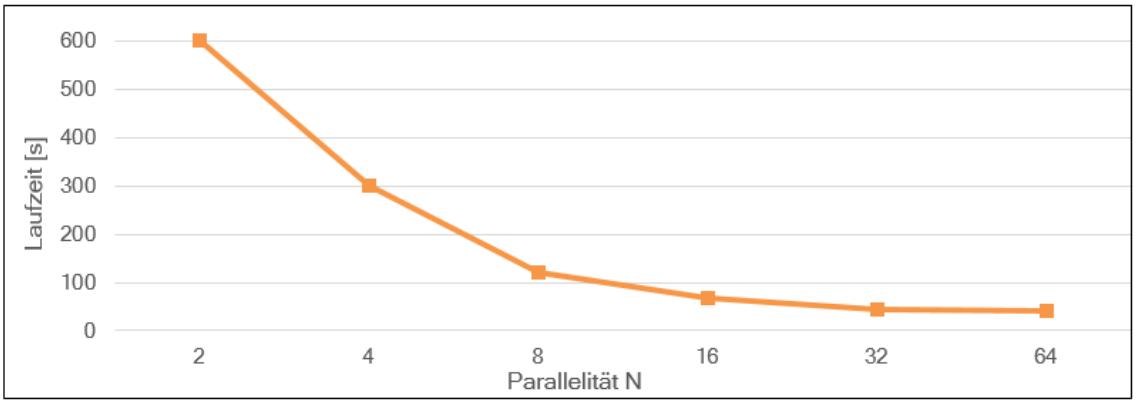

Abbildung 4.11: Laufzeit mit der Strategie *fft-fft* in Abhängigkeit vom Grad der Parallelität für 1 Million Datensätze

Anzahl Datensätze n_I+n_Q		1.000.000										
Strategie		Parallelität N	Anzahl Pivots		Laufzeit [s]						
lokal	global		lokal n_L	global n_P	(1) Finden Pivots lokal (Map-Partition)	(2) Finden Pivots global (Group-Reduce)	(3) Zuordnung Pivots zu Index (Map)	(5) Zuordnung Pivots zu Query (Flat-Map)	(6) Matching (Co-Group)	Ausgabe der Resultate (DataSink)	**gesamt (nicht kumulativ)**
fft	fft	**8**	**2.000**	**6.000**	26	29	65	82	52	35	120
fft	none	**8**	**1.250**	**10.000**	18	0	70	94	53	30	126
fft	fft	**16**	**1.000**	**6.000**	9	13	32	43	33	22	67
fft	none	**16**	**625**	**10.000**	9	0	36	49	34	18	71
fft	fft	**32**	**500**	**6.000**	3	8	18	25	25	16	45
fft	none	**32**	**500**	**16.000**	3	0	27	39	20	9	49
fft	fft	**64**	**250**	**6.000**	1	7	14	20	27	16	42
fft	none	**64**	**250**	**16.000**	1	0	15	24	20	10	37

Abbildung 4.12: detaillierte Laufzeiten mit verschiedenen Strategien und Parallelitätsgraden für 1 Million Datensätze

Dies ist besonders bei *fft-fft* der Fall. Beispielsweise ist hier der Zugewinn von $N = 32$ auf $N = 64$ nur noch marginal. Dafür gibt es mehrere Gründe. Zum einen stellt Schritt *2* (siehe Abbildung 4.12) einen Flaschenhals dar, da das Finden der globalen Pivots nicht parallel stattfindet. Außerdem führt eine niedrigere Anzahl lokaler Pivots zu einem Qualitätsverlust. Es ist zu beobachten, dass C_Q und K_Q für die Laufzeitbestwerte bei höheren N ansteigen. Wird jedoch n_L erhöht, so verschlechtert sich die Laufzeit für die Schritte *1* bis *5* (Finden und Zuordnung der Pivots) so weit, dass auch die Gesamtlaufzeit höher ist. Diese Tatsache ist zwar auch bei

fft-none zu beobachten, jedoch kann hier die Pivotanzahl weiter erhöht werden ohne große Laufzeiteinbuße beim Finden hinnehmen zu müssen, da es keinen globalen Findungsschritt gibt. Das komplett parallele Finden der Pivots wird hierbei ideal genutzt. Außerdem ist durch die insgesamt höhere Anzahl der Pivots die Laufzeit beim Matching geringer. Deswegen ist für $N = 64$ die Strategie *fft-none* erstmals insgesamt schneller als *fft-fft*.

Insgesamt ist festzuhalten, dass sich bei höherer Parallelität durchaus die ideale Anzahl (n_L, n_P) verändern kann. Die Qualität der Pivots nimmt eventuell sogar ab und C_Q und K_Q erhöhen sich. Durch die stärkere Verteilung der Rechenlast wird die Gesamtlaufzeit dennoch geringer. Wie auch in den Abschnitten zuvor zu erkennen, ist bezüglich der Laufzeit immer eine Abwägung zwischen dem Finden und Zuordnen der Pivots gegenüber dem Matching notwendig. Je nach konkreter Anwendung ist aber eine Parallelität von 8 bis 16 ausreichend. Eine weitere Erhöhung des Aufwandes führt nicht zu verhältnismäßigen Verbesserungen.

4.2.5 Vergleich mit nicht verteilten Verfahren

In [4], deren Ansatz die Grundlage dieser Arbeit bildet, werden einige Laufzeiten für das nicht verteilte Verfahren angegeben. Da es sich um exakt dieselben verwendeten Testdaten handelt, ist ein Vergleich gut möglich. Zum Finden der Pivots wird dort der MS-Algorithmus verwendet. Die Pivots werden aus einem zufälligen Sample (etwa dreifache Anzahl der Pivots) der Index-Datensätze bestimmt. Dies entspricht annähernd der hier beschriebenen Strategie *random-ms*. Die ausführlichen Experimente werden dort für die Datengröße $n_I + n_Q = 500.000$ durchgeführt. Dabei sind die Laufzeiten für $n_P = 1.500$ bis $n_P = 4.000$ stabil und erreichen im Bestfall 103 s. K_Q wird für diesen Fall mit etwa 15.000 angegeben.

Vergleicht man diese Resultate mit dem besten in dieser Arbeit erzielten Ergebnis für die entsprechende Datengröße bei $N = 8$ (Strategie *fft-fft*), erhält man für die Laufzeit 59 s und für $K_Q = 3.153$. Dies stellt eine signifikante Verbesserung dar.

Noch deutlicher wird der Vergleich der Laufzeiten für $n_I + n_Q = 1.000.000$. Hier gibt [4] eine Laufzeit von 5,2 Minuten (= 312 s) an. Bei $N = 8$ konnten in dieser Arbeit 120 s erreicht werden, was einer Laufzeitersparnis von 61% entspricht. Mit $N = 64$ sind sogar 88% Einsparung (mit 37 s) erzielbar.

5 Zusammenfassung

5.1 Ergebnisse der Arbeit

In dieser Arbeit wurde ein verteiltes PPRL-Verfahren entwickelt. Als Grundlage diente dazu der in [4] beschriebene Ansatz der Verwendung von metrischen Räumen und Pivot-Elementen. Dieser Ansatz nutzt zum Blocking von Vergleichen Eigenschaften der Dreiecksungleichung aus (Abschnitt 2.3). Das konzipierte Verfahren (Abschnitt 3.1) konnte anschließend erfolgreich mit dem Framework Apache Flink in ein lauffähiges Programm umgesetzt werden (Abschnitt 3.2).

Im Mittelpunkt des entwickelten Verfahrens stand das parallele Finden von Pivots und das parallele Matching. Für diese beiden Probleme konnten Lösungen gefunden werden. Das parallele Finden von Pivots erfolgt in einem lokalen und einem globalen Teilschritt, um die Rechenlast zu verteilen bzw. zu beschränken. Für beide Schritte wurden verschiedene Strategien vorgeschlagen und umgesetzt. Für das Matching ist entscheidend, dass sowohl Index- als auch Query-Datensätze zuvor parallel den Pivots zugeordnet werden. Dies ermöglicht den anschließenden Einsatz des effizienten *CoGroup*-Operators von Flink, um verteilt Matches zu finden. Weiterhin wurden erhebliche Optimierungen vorgenommen, um die allgemeine Laufzeit des Programms zu verbessern (Abschnitt 3.2.3). Dazu zählen unter anderem die Berücksichtigung der Besonderheiten von Flink sowie Verbesserungen des PPRL-Verfahrens.

Mit der Evaluation der Laufzeiten und verschiedener Metriken auf einem Cluster konnten vielseitige Erkenntnisse über den genauen Ablauf des Programms bei Anwendung verschiedener Parameter und Strategien gewonnen werden (Abschnitt 4.2). Als entscheidender Faktor stellte sich die Anzahl der Pivots heraus. Hierbei geht es um die Abwägung zwischen der Qualität der Pivots und dem Aufwand zum Finden. Es stellt sich stets die Frage, ob mehr verwendete Zeit zum Finden der Pivots

anschließend genügend zeitliche Vorteile beim Matching mit sich bringt. Je nach angewandter Strategie waren die Ergebnisse dazu unterschiedlich (Abschnitt 4.2.1).

Insgesamt brachten jedoch zwei Strategien die mit Abstand besten Laufzeiten hervor (*fft-fft* und *fft-none*, Abschnitt 4.2.2). So konnte eine enorme Verbesserung der Ergebnisse im Vergleich zum zu Grunde liegenden nicht verteilten Verfahren erzielt werden. Mit Erhöhung der Parallelität wurden diese Laufzeiten nochmals verbessert (Abschnitt 4.2.4) und ergaben so Vorteile von bis zu 88% (Abschnitt 4.2.5). Allerdings ist zu erwähnen, dass eine unbegrenzte Erhöhnung der Parallelität nicht unbedingt sinnvoll ist. Für die meisten Anwendungsfälle ist ein Grad der Parallelität von 8 bis 16 ausreichend.

Ein weitere wichtige Erkenntnis ist die sehr gute Skalierbarkeit des Verfahrens. Die Messungen zeigten einen linearen Anstieg der Laufzeiten bei Erhöhung der Datenmengen (Abschnitt 4.2.3). Dies ist besonders im Vergleich zur quadratischen Komplexität des zu Grunde liegenden Problems und im Vergleich zu früheren Arbeiten ein hervorragendes Ergebnis.

Die gestellten Ziele der Arbeit konnten erreicht werden. Mit der Verwendung des Frameworks Flink und der damit einhergehenden Parallelisierung konnte das bestehende PPRL-Verfahren mit dem vielversprechenden Ansatz der metrischen Räume erheblich verbessert werden.

5.2 Ausblick

Ein wichtiger Aspekt bei der Entwicklung neuer Verfahren ist deren anschließende praktische Anwendung. Hierbei besteht im vorliegenden Fall das Problem der Festsetzung der Parameter ohne Testergebnisse. Die genaue Datenlage beeinflusst die optimale Verteilung der Pivots sehr stark. Daher können für die Anwendung des Programms auf neue Daten nur Empfehlungen bezüglich der für die Laufzeit optimalen Kombination der Parameter gegeben werden. Zweifelsfrei sollte entweder die Strategie fft-fft oder (eher bei höherer Parallelität) fft-none verwendet werden. Schwieriger festzustellen ist die optimale Anzahl der Pivots. Als Richtlinie kann hierbei angegeben werden, dass die Anzahl der globalen Pivots eher größer sein sollte und die der lokalen Pivots eher niedriger (jedoch nicht kleiner als die Länge

der kodierten Bit-Arrays). Hierzu wäre weitere Forschungsarbeit nötig, um eventuell anhand einer kleinen Beispieldatenmenge die optimale Parameterkombination festzustellen.

Ein weiterer möglicher Anwendungsfall des Programms wäre, wenn die Query-Daten als Stream vorliegen. Eine Umstellung wäre ohne größeren Aufwand machbar, da die Daten intern von Flink ohnehin als Stream behandelt werden. Hierzu sollte die Strategie gewählt werden, welche die schnellsten Ergebnisse beim Matching bringt. Das Finden der Pivots könnte zuvor mit den Index-Daten erfolgen.

A Anhang

A.1 Quellcode des Flink-Programms

Dies ist eine gekürzte Version des Original-Quellcodes. Zur Vereinfachung wurden folgende Teile weggelassen: Imports, Akkumulatoren zur Erstellung der Metriken, das Schreiben der Ergebnisse in Dateien, Abschnitte im Zusammenhang mit der Zuordnungsstrategie "lokal".

```
1  public class PPRL{
2     public static void main(String[] args) throws Exception {
3        String indexDataPath = args[0];
4        String queryDataPath = args[1];
5        int numberOfPivotsGlobal = Integer.parseInt(args[5]);
6        int numberOfPivotsLocal = Integer.parseInt(args[3]);
7        int parallelism = Integer.parseInt(args[6]);
8        String pivotStrategyLocal = args[2];
9        String pivotStrategyGlobal = args[4];
10       double treshold = 0.8;
11       double tresholdFactor = ((1.0-treshold)/treshold);
12
13       ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
14       env.setParallelism(parallelism);
15
16       if(pivotStrategyGlobal.equals("none")){
17          numberOfPivotsLocal = numberOfPivotsGlobal / env.getParallelism();
18       }
19
20       //Index-Datensaetze (ID, value)
21       DataSet<Tuple2<Integer,OpenBitSet>> indexData
22          = readData(env,indexDataPath)
23          .map(new input())
24          .withForwardedFields("f0");
25
26       //lokale Pivots (ID, value)
27       DataSet<Tuple2<Integer,OpenBitSet>> pivotsLocal
28          = indexData
29          .mapPartition(new findPivotsLocal(pivotStrategyLocal,numberOfPivotsLocal))
30          .withForwardedFields("f0;f1");
31
32       //globale Pivots (ID, value)
33       DataSet<Tuple2<Integer,OpenBitSet>> pivotsGlobal;
34       if(pivotStrategyGlobal.equals("none")){
35          pivotsGlobal = pivotsLocal;
36          numberOfPivotsGlobal = numberOfPivotsLocal*env.getParallelism();
37       }else{
38          pivotsGlobal
39             = pivotsLocal
40             .reduceGroup(new findPivotsGlobal(pivotStrategyGlobal,numberOfPivotsGlobal))
41             .withForwardedFields("f0;f1");
42       }
```

```
43
44          //Index-Datensaetze mit zugeordneten Pivots (ID, value, PivotID, d(Pivot))
45          UnsortedGrouping<Tuple4<Integer,OpenBitSet,Integer,Long>> indexDataPivots
46             = indexData
47             .map(new pivotToIndex())
48             .withBroadcastSet(pivotsGlobal, "gpivots")
49             .withForwardedFields("f0;f1")
50             .groupBy(2);
51
52          //globale Pivots mit Radius (ID, value, PivotRadius)
53          DataSet<Tuple3<Integer,OpenBitSet,Long>> pivotsGlobalWithRad
54             = indexDataPivots
55             .reduceGroup(new assignPivot());
56
57          //Query-Datensaetze (ID, value)
58          DataSet<Tuple2<Integer,OpenBitSet>> queryData
59             = readData(env,queryDataPath)
60             .map(new input())
61             .withForwardedFields("f0");
62
63          //Query-Datensaetze mit zugeordneten Pivots (ID, value, PivotID, d(Pivot), rad(q))
64          DataSet<Tuple5<Integer,OpenBitSet,Integer,Long,Long>> queryDataPivots
65             = queryData
66             .flatMap(new pivotToQuery(tresholdFactor))
67             .withBroadcastSet(pivotsGlobalWithRad, "gpivotswr")
68             .withForwardedFields("f0;f1");
69
70          //Matches (IndexID, QueryID, Similarity)
71          DataSet<Tuple3<Integer,Integer,Double>> matches
72             = indexDataPivots
73             .combineGroup(new combineIndex())
74             .withForwardedFields("f0;f1;f2;f3")
75             .coGroup(queryDataPivots)
76             .where(2)
77             .equalTo(2)
78             .with(new matchData(treshold,tresholdFactor))
79             .withForwardedFieldsFirst("f0")
80             .withForwardedFieldsSecond("f0->f1");
81
82          matches.print();
83       }
84
85       //Einlesefunktion
86       public static DataSet<Tuple2<Integer,String>> readData(ExecutionEnvironment env, String dataPath) {
87          return  env.readCsvFile(dataPath)
88                 .fieldDelimiter(" ")
89                 .includeFields("11")
90                 .types(Integer.class, String.class);
91       }
92
93       //Formatierung des Inputs
94       public static final class input extends RichMapFunction<Tuple2<Integer,String>, Tuple2<Integer,OpenBitSet>> {
95          @Override
96          public Tuple2<Integer, OpenBitSet> map(Tuple2<Integer, String> in) throws Exception {
97             char[] c = in.f1.toCharArray();
98             OpenBitSet value = new OpenBitSet(c.length);
99             int i = 0;
100            for (char b : c) {
101               if (b == '1') {
102                  value.fastSet(i);
103               }
104               i++;
105            }
106            return new Tuple2<Integer, OpenBitSet>(in.f0, value);
107         }
108      }
109
110      //Schritt 1
111      public static final class findPivotsLocal extends RichMapPartitionFunction<Tuple2<Integer,OpenBitSet>, Tuple2
             <Integer,OpenBitSet>> {
112         String pivotStrategyLocal;
113         int numberOfPivotsLocal;
```

```
114
115         public findPivotsLocal(String pivotStrategyLocal, int numberOfPivotsLocal){
116            this.pivotStrategyLocal = pivotStrategyLocal;
117            this.numberOfPivotsLocal = numberOfPivotsLocal;
118         }
119
120         @Override
121         public void mapPartition(Iterable<Tuple2<Integer,OpenBitSet>> in, Collector<Tuple2<Integer,OpenBitSet>>
                out) throws Exception {
122            if(pivotStrategyLocal.equals("fft")){
123               findPivotsFFT(in, out, numberOfPivotsLocal);
124            }else if(pivotStrategyLocal.equals("ms")){
125               findPivotsMaxSep(in, out, numberOfPivotsLocal);
126            }else{
127               findPivotsRandom(in, out, numberOfPivotsLocal);
128            }
129         }
130      }
131
132      //Schritt 2
133      public static final class findPivotsGlobal extends RichGroupReduceFunction<Tuple2<Integer,OpenBitSet>, Tuple2
            <Integer,OpenBitSet>> {
134         String pivotStrategyGlobal;
135         int numberOfPivotsGlobal;
136
137         public findPivotsGlobal(String pivotStrategyGlobal, int numberOfPivotsGlobal){
138            this.pivotStrategyGlobal = pivotStrategyGlobal;
139            this.numberOfPivotsGlobal = numberOfPivotsGlobal;
140         }
141
142         @Override
143         public void reduce(Iterable<Tuple2<Integer,OpenBitSet>> in, Collector<Tuple2<Integer,OpenBitSet>> out)
                throws Exception {
144            if(pivotStrategyGlobal.equals("fft")){
145               findPivotsFFT(in, out, numberOfPivotsGlobal);
146            }else if(pivotStrategyGlobal.equals("ms")){
147               findPivotsMaxSep(in, out, numberOfPivotsGlobal);
148            }
149         }
150      }
151
152      //Schritt 3
153      public static final class pivotToIndex extends RichMapFunction<Tuple2<Integer,OpenBitSet>, Tuple4<Integer,
            OpenBitSet, Integer, Long>> {
154         Collection<Tuple2<Integer,OpenBitSet>> pivots;
155
156         @Override
157         public void open(Configuration parameters) throws Exception {
158            this.pivots = getRuntimeContext().getBroadcastVariable("gpivots");
159         }
160
161         @Override
162         public Tuple4<Integer, OpenBitSet, Integer, Long> map(Tuple2<Integer,OpenBitSet> in) throws Exception {
163            long distance, minDistance = Long.MAX_VALUE;
164            Tuple2<Integer,OpenBitSet> nearPivot = new Tuple2<Integer,OpenBitSet>(0,new OpenBitSet(1));
165            for(Tuple2<Integer,OpenBitSet> p : pivots){
166               if(p.f0.intValue() == in.f0.intValue()){
167                  nearPivot = in;
168                  minDistance = 0;
169                  break;
170               }else{
171                  distance = distance(in.f1,p.f1);
172                  if(minDistance > distance){
173                     minDistance = distance;
174                     nearPivot = p;
175                  }
176               }
177            }
178            return new Tuple4<Integer, OpenBitSet, Integer, Long>(in.f0,in.f1,nearPivot.f0,minDistance);
179         }
180      }
181
```

```
182     //Schritt 4
183     public static final class assignPivot extends RichGroupReduceFunction<Tuple4<Integer,OpenBitSet,Integer,Long
            >, Tuple3<Integer,OpenBitSet,Long>> {
184        @Override
185        public void reduce(Iterable<Tuple4<Integer,OpenBitSet,Integer,Long>> in, Collector<Tuple3<Integer,
               OpenBitSet,Long>> out) throws Exception {
186           Iterator<Tuple4<Integer,OpenBitSet,Integer,Long>> itr = in.iterator();
187           Tuple4<Integer,OpenBitSet,Integer,Long> pivot = new Tuple4<Integer,OpenBitSet,Integer,Long>(0,new
                  OpenBitSet(1),0,(long) 0);
188           long radius = 0;
189           while (itr.hasNext()) {
190              Tuple4<Integer,OpenBitSet,Integer,Long> next = itr.next();
191              if(next.f3.longValue() > radius){
192                 radius = next.f3.longValue();
193              }
194              if(next.f0.intValue() == next.f2.intValue()){
195                 pivot = next;
196              }
197           }
198           out.collect(new Tuple3<Integer,OpenBitSet,Long>(pivot.f0,pivot.f1,radius));
199        }
200     }
201
202     //Schritt 5
203     public static final class pivotToQuery extends RichFlatMapFunction<Tuple2<Integer,OpenBitSet>, Tuple5<Integer
            , OpenBitSet, Integer, Long, Long>> {
204        Collection<Tuple3<Integer,OpenBitSet,Long>> pivots;
205        double tresholdFactor;
206
207        public pivotToQuery(double tresholdFactor){
208           this.tresholdFactor = tresholdFactor;
209        }
210
211        @Override
212        public void open(Configuration parameters) throws Exception {
213           this.pivots = getRuntimeContext().getBroadcastVariable("gpivotswr");
214        }
215
216        @Override
217        public void flatMap(Tuple2<Integer,OpenBitSet> in, Collector<Tuple5<Integer, OpenBitSet, Integer, Long,
               Long>> out) throws Exception {
218           long queryRadius = (long) (Math.ceil(tresholdFactor*in.f1.cardinality()));
219           for(Tuple3<Integer,OpenBitSet,Long> p : pivots){
220              long distance = distance(p.f1,in.f1);
221              if(distance <= (p.f2.longValue()+queryRadius)){
222                 out.collect(new Tuple5<Integer, OpenBitSet, Integer, Long, Long>(in.f0,in.f1,p.f0,distance,
                        queryRadius));
223              }
224           }
225        }
226     }
227
228     //Combiner fuer Schritt 6
229     public static final class combineIndex extends RichGroupCombineFunction<Tuple4<Integer,OpenBitSet,Integer,
            Long>, Tuple4<Integer,OpenBitSet,Integer,Long>> {
230        @Override
231        public void combine(Iterable<Tuple4<Integer,OpenBitSet,Integer,Long>> in, Collector<Tuple4<Integer,
               OpenBitSet,Integer,Long>> out) {
232           Iterator<Tuple4<Integer, OpenBitSet, Integer, Long>> itr = in.iterator();
233           while(itr.hasNext()){
234              out.collect(itr.next());
235           }
236        }
237     }
238
239     //Schritt 6
240     public static final class matchData extends RichCoGroupFunction<Tuple4<Integer, OpenBitSet, Integer, Long>,
            Tuple5<Integer, OpenBitSet, Integer, Long, Long>, Tuple3<Integer, Integer, Double>> {
241        double treshold;
242        double tresholdFactor;
243
244        public matchData(double treshold, double tresholdFactor){
```

```
245           this.tresholdFactor = tresholdFactor;
246           this.treshold = treshold;
247        }
248
249        @Override
250        public void coGroup(Iterable<Tuple4<Integer, OpenBitSet, Integer, Long>> data, Iterable<Tuple5<Integer,
                 OpenBitSet, Integer, Long, Long>> query, Collector<Tuple3<Integer, Integer, Double>> out) throws
                 Exception {
251           List<Tuple4<Integer,OpenBitSet,Integer,Long>> index = new ArrayList<Tuple4<Integer,OpenBitSet,Integer,
                    Long>>();
252           for(Tuple4<Integer, OpenBitSet, Integer, Long> d : data){
253              index.add(d);
254           }
255           for(Tuple5<Integer, OpenBitSet, Integer, Long, Long> q : query){
256              for(Tuple4<Integer, OpenBitSet, Integer, Long> i : index){
257                 if((q.f3.longValue()-i.f3.longValue()) <= q.f4.longValue()){
258                    long distance = distance(q.f1,i.f1);
259                    if(distance <= q.f4.longValue()){
260                       double similarity = similarity(q.f1,i.f1);
261                       if(treshold <= similarity){
262                          out.collect(new Tuple3<Integer, Integer, Double>(i.f0,q.f0,similarity));
263                       }
264                    }
265                 }
266              }
267           }
268        }
269     }
270
271     //MS-Algorithmus
272     public static void findPivotsMaxSep(Iterable<Tuple2<Integer,OpenBitSet>> in, Collector<Tuple2<Integer,
              OpenBitSet>> out, int numberOfPivots) {
273        Iterator<Tuple2<Integer,OpenBitSet>> itr = in.iterator();
274        Map<Tuple2<Integer,OpenBitSet>,Long> data = new HashMap<Tuple2<Integer,OpenBitSet>,Long>();
275        Tuple2<Integer,OpenBitSet> lastPivot;
276        Tuple2<Integer,OpenBitSet> nextPivot = new Tuple2<Integer,OpenBitSet>(0,new OpenBitSet(1));
277        Tuple2<Integer,OpenBitSet> next = null;
278
279        while (itr.hasNext()) {
280           next = itr.next();
281           data.put(next,(long) 0);
282        }
283
284        lastPivot = next;
285        long distance, maxDistance;
286
287        for(int i = 0; i < numberOfPivots; i++){
288           maxDistance = 0;
289           for(Map.Entry<Tuple2<Integer,OpenBitSet>, Long> d : data.entrySet()){
290              distance = d.getValue() + distance(d.getKey().f1,lastPivot.f1);
291              if(i > 0){
292                 data.put(d.getKey(),distance);
293              }
294              if(maxDistance < distance){
295                 nextPivot = d.getKey();
296                 maxDistance = distance;
297              }
298           }
299           data.remove(nextPivot);
300           out.collect(nextPivot);
301           lastPivot = nextPivot;
302        }
303     }
304
305     //FFT-Algorithmus
306     public static void findPivotsFFT(Iterable<Tuple2<Integer,OpenBitSet>> in, Collector<Tuple2<Integer,OpenBitSet
              >> out, int numberOfPivots) {
307        Iterator<Tuple2<Integer,OpenBitSet>> itr = in.iterator();
308        Map<Tuple2<Integer,OpenBitSet>,Long> data = new HashMap<Tuple2<Integer,OpenBitSet>,Long>();
309        Tuple2<Integer,OpenBitSet> lastPivot;
310        Tuple2<Integer,OpenBitSet> nextPivot = new Tuple2<Integer,OpenBitSet>(0,new OpenBitSet(1));
311        Tuple2<Integer,OpenBitSet> next = null;
```

```
312
313         while (itr.hasNext()) {
314            next = itr.next();
315            data.put(next,(long) 1000);
316         }
317
318         lastPivot = next;
319         long distance, maxDistance, minDistance;
320
321         for(int i = 0; i < numberOfPivots; i++){
322            maxDistance = 0;
323            for(Map.Entry<Tuple2<Integer,OpenBitSet>, Long> d : data.entrySet()){
324               distance = distance(d.getKey().f1,lastPivot.f1);
325               if(i > 0){
326                  minDistance = d.getValue();
327                  if(distance < minDistance){
328                     data.put(d.getKey(),distance);
329                     minDistance = distance;
330                  }
331               }else{
332                  minDistance = distance;
333               }
334               if(maxDistance < minDistance){
335                  nextPivot = d.getKey();
336                  maxDistance = minDistance;
337               }
338            }
339            data.remove(nextPivot);
340            out.collect(nextPivot);
341            lastPivot = nextPivot;
342         }
343      }
344
345      //zufaelliges Finden von Pivots
346      public static void findPivotsRandom(Iterable<Tuple2<Integer,OpenBitSet>> in, Collector<Tuple2<Integer,
            OpenBitSet>> out, int numberOfPivots) {
347         Iterator<Tuple2<Integer,OpenBitSet>> itr = in.iterator();
348         for(int i = 0; i < numberOfPivots; i++){
349            if(itr.hasNext()) {
350               Tuple2<Integer,OpenBitSet> next = itr.next();
351               out.collect(next);
352            }
353         }
354      }
355
356      //Distanzvergleich
357      public static long distance(OpenBitSet x, OpenBitSet y){
358         OpenBitSet xor = x.clone();
359         xor.xor(y);
360         return xor.cardinality();
361      }
362
363      //Aehnlichkeitsvergleich
364      public static double similarity(OpenBitSet x, OpenBitSet y){
365         OpenBitSet and = x.clone();
366         OpenBitSet or = x.clone();
367         and.and(y);
368         or.or(y);
369         return ((double) and.cardinality()) / ((double) or.cardinality());
370      }
371   }
```

A.2 Inhalt der beiliegenden CD

- **Masterarbeit:** diese Arbeit als PDF-Datei
- **Messungen:** aufgenommene Messergebnisse
- **Quellcode:** ausführlicher Quellcode des Flink-Programms
- **Literatur:** Nachweise der aufgeführten Literatur

Symbolverzeichnis

d_h	Hamming-Distanz
d_j	Jaccard-Distanz
sim_j	Jaccard-Ähnlichkeit
t	Ähnlichkeits-Threshold
Q	Query-Datenquelle
q	Query-Datensatz (verschlüsseltes Bit-Array), $q \in Q$
$rad(q)$	Ähnlichkeits-Radius des Query-Datensatz q
n_Q	Anzahl der Query-Datensätze
I	Index-Datenquelle
i	Index-Datensatz (verschlüsseltes Bit-Array), $i \in I$
n_I	Anzahl der Index-Datensätze
P	Menge aller Pivot-Elemente, $P \subset I$
p	Pivot-Element, $p \in P$
$Set(p)$	Menge der p zugeordneten Index-Datensätze
$rad(p)$	Radius des Pivot-Elements p
n_P	Anzahl der Pivot-Elemente (allgemein bzw. global)
n_L	Anzahl der Pivot-Elemente (lokal pro Partition)
N	Grad der Parallelitat bzw. Anzahl der Partitionen
C_Q	durchschnittliche Anzahl von Kopien (Anzahl zugeordneter Pivots) pro Query-Datensatz
K_Q	durchschnittliche Anzahl von Kandidaten (Index-Datensätze) pro Query-Datensatz, $K_Q = \frac{V_D}{n_Q}$
V_L	Vergleiche beim Finden von Pivots (lokal)
V_G	Vergleiche beim Finden von Pivots (global)
V_P	Vergleiche beim Finden von Pivots (gesamt), $V_P = V_L + V_G$

V_T	ausgeführte Vergleiche mithilfe der Dreiecksungleichung zwischen I und Q
V_D	ausgeführte Distanzvergleiche zwischen I und Q
V_S	ausgeführte Ähnlichkeitsvergleiche zwischen I und Q
V_{max}	maximal mogliche Ähnlichkeitsvergleiche zwischen I und Q, $V_{max} = n_I \cdot n_Q$

Abbildungsverzeichnis

Literaturverzeichnis

[1] Dinusha Vatsalan, Ziad Sehili, Peter Christen, and Erhard Rahm. *Privacy-Preserving Record Linkage for Big Data: Current Approaches and Research Challenges [In: Handbook of Big Data Technologies (A. Zomaya, S. Sakr)].* Springer, 2017.

[2] Peter Christen. *Privacy-preserving Record Linkage.* Research School of Computer Science, ANU College of Engineering and Computer Science, The Australian National University, ScaDS Leipzig, Juli 2016.

[3] Dimitrios Karapiperis and Vassilios S. Verykios. *A Distributed Near-Optimal LSH-based Framework for Privacy-Preserving Record Linkage.* School of Science and Technology Hellenic Open University, 2014.

[4] Ziad Sehili and Erhard Rahm. *Speeding up Privacy Preserving Record Linkage for Metric Space Similarity Measures.* ScaDS Leipzig, Datenbank Spektrum, 2016.

[5] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search: The Metric Space Approach.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[6] Rui Mao, Peihan Zhang, Xingliang Li, Xi Liu, and Minhua Lu. *Pivot selection for metric-space indexing.* Datenbank Spektrum, 2016.

[7] Benjamin Bustos and Gonzalo Navarro. *Pivot Selection Techniques for Proximity Searching in Metric Spaces.* Universidad de Chile.

[8] Apache Flink. http://flink.apache.org/, 2017.

[9] Apache Flink. *Dokumentation.* https://ci.apache.org/projects/flink/flink-docs-release-1.2/, 2017.

[10] Apache Lucene OpenBitSet. https://lucene.apache.org/core/3_0_3/api/core/org/apache/lucene/util/OpenBitSet.html, 2017.

[11] Fabian Hueske. *Apache Flink Batch Advanced Training.* https://www.youtube.com/watch?v=1yWKZ26NQeU&index=3&list=PLaDktj9CFcS_dbgtlyYwGrXRYAWnlM_9i, Google Office Berlin, Juni 2015.

[12] Themis Palpanas and George Papadakis. *Blocking for BIG Data Integration.* University of Athens, Paris Descartes University, ScaDS Leipzig, Juli 2016.