Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Effiziente MapReduce-Parallelisierung von Entity Resolution-Workflows

Dissertation
zur Erlangung des akademischen Grades
Doctor Rerum Naturalium
(Dr. rer. nat.)
im Fachgebiet Informatik

vorgelegt von
Diplom-Informatiker Lars Kolb
geboren am 24. Mai 1984 in Leipzig

Leipzig, den 01. September 2014

Die Annahme der Dissertation haben empfohlen:

Prof. Dr. Erhard Rahm (Institut für Informatik, Universität Leipzig)
Prof. Dr. Felix Naumann (Hasso-Plattner-Institut für Softwaresystemtechnik, Universität Potsdam)

Die Verleihung des akademischen Grades erfolgt mit Bestehen der Verteidigung
am 08. Dezember 2014 mit dem Gesamtprädikat *magna cum laude*.

Zusammenfassung

In den vergangenen Jahren hat das neu entstandene Paradigma *Infrastructure as a Service* die IT-Welt massiv verändert. Die Bereitstellung von Recheninfrastruktur durch externe Dienstleister bietet die Möglichkeit, bei Bedarf in kurzer Zeit eine große Menge von Rechenleistung, Speicherplatz und Bandbreite ohne Vorabinvestitionen zu akquirieren. Gleichzeitig steigt sowohl die Menge der frei verfügbaren als auch der in Unternehmen zu verwaltenden Daten dramatisch an. Die Notwendigkeit zur effizienten Verwaltung und Auswertung dieser Datenmengen erforderte eine Weiterentwicklung bestehender IT-Technologien und führte zur Entstehung neuer Forschungsgebiete und einer Vielzahl innovativer Systeme. Ein typisches Merkmal dieser Systeme ist die verteilte Speicherung und Datenverarbeitung in großen Rechnerclustern bestehend aus Standard-Hardware. Besonders das *MapReduce*-Programmiermodell hat in den vergangenen zehn Jahren zunehmend an Bedeutung gewonnen. Es ermöglicht eine verteilte Verarbeitung großer Datenmengen und abstrahiert von den Details des verteilten Rechnens sowie der Behandlung von Hardwarefehlern. Innerhalb dieser Dissertation steht die Nutzung des MapReduce-Konzeptes zur automatischen Parallelisierung rechenintensiver Entity Resolution-Aufgaben im Mittelpunkt. Entity Resolution ist ein wichtiger Teilbereich der Informationsintegration, dessen Ziel die Entdeckung von Datensätzen einer oder mehrerer Datenquellen ist, die dasselbe Realweltobjekt beschreiben. Im Rahmen der Dissertation werden schrittweise Verfahren präsentiert, welche verschiedene Teilprobleme der MapReduce-basierten Ausführung von Entity Resolution-Workflows lösen.

Zur Erkennung von Duplikaten vergleichen Entity Resolution-Verfahren üblicherweise Paare von Datensätzen mithilfe mehrerer Ähnlichkeitsmaße. Die Auswertung des Kartesischen Produktes von n Datensätzen führt dabei zu einer quadratischen Komplexität von $O(n^2)$ und ist deswegen nur für kleine bis mittelgroße Datenquellen praktikabel. Für Datenquellen mit mehr als 100.000 Datensätzen entstehen selbst bei verteilter Ausführung Laufzeiten von mehreren Stunden. Deswegen kommen sogenannte *Blocking*-Techniken zum Einsatz, die zur Reduzierung des Suchraums dienen. Die zugrundeliegende Annahme ist, dass Datensätze, die eine gewisse Mindestähnlichkeit unterschreiten, nicht miteinander verglichen werden müssen. Die Arbeit stellt eine MapReduce-basierte Umsetzung der Auswertung des Kartesischen Produktes sowie einiger bekannter Blocking-Verfahren vor. Nach dem Vergleich der Datensätze erfolgt abschließend eine Klassifikation der verglichenen Kandidaten-Paare in *Match* bzw. *Non-Match*. Mit einer steigenden Anzahl verwendeter Attributwerte und Ähnlichkeitsmaße ist eine manuelle Festlegung einer qualitativ hochwertigen Strategie zur Kombination der resultierenden Ähnlichkeitswerte kaum mehr handhabbar. Aus diesem Grund untersucht die Arbeit die Integration maschineller Lernverfahren in MapReduce-basierte Entity Resolution-Workflows.

Eine Umsetzung von Blocking-Verfahren mit MapReduce bedingt eine Partitionierung der Menge der zu vergleichenden Paare sowie eine Zuweisung der Partitionen zu verfügbaren Prozessen. Die Zuweisung erfolgt auf Basis eines semantischen Schlüssels, der entsprechend der konkreten Blocking-Strategie aus den Attributwerten der Datensätze abgeleitet ist. Beispielsweise wäre es bei der Deduplizierung von Produktdatensätzen denkbar, lediglich Produkte des gleichen Herstellers miteinander zu vergleichen. Die Bearbeitung aller Datensätze desselben Schlüssels durch einen Prozess führt bei Datenungleichverteilung zu erheblichen Lastbalancierungsproblemen, die durch die inhärente quadratische Komplexität verschärft werden. Dies reduziert in drastischem Maße die Laufzeiteffizienz und Skalierbarkeit der entsprechenden MapReduce-Programme, da ein Großteil der Ressourcen eines Clusters nicht ausgelastet ist, wohingegen wenige Prozesse den Großteil der Arbeit verrichten müssen. Die Bereitstellung ver-

schiedener Verfahren zur gleichmäßigen Ausnutzung der zur Verfügung stehenden Ressourcen stellt einen weiteren Schwerpunkt der Arbeit dar.

Blocking-Strategien müssen stets zwischen Effizienz und Datenqualität abwägen. Eine große Reduktion des Suchraums verspricht zwar eine signifikante Beschleunigung, führt jedoch dazu, dass ähnliche Datensätze, z. B. aufgrund fehlerhafter Attributwerte, nicht miteinander verglichen werden. Aus diesem Grunde ist es hilfreich, für jeden Datensatz mehrere von verschiedenen Attributen abgeleitete semantische Schlüssel zu generieren. Dies führt jedoch dazu, dass ähnliche Datensätze unnötigerweise mehrfach bezüglich verschiedener Schlüssel miteinander verglichen werden. Innerhalb der Arbeit werden deswegen Algorithmen zur Vermeidung solch redundanter Ähnlichkeitsberechnungen präsentiert.

Als Ergebnis dieser Arbeit wird das Entity Resolution-Framework *Dedoop* präsentiert, welches von den entwickelten MapReduce-Algorithmen abstrahiert und eine High-Level-Spezifikation komplexer Entity Resolution-Workflows ermöglicht. Dedoop fasst alle in dieser Arbeit vorgestellten Techniken und Optimierungen in einem nutzerfreundlichen System zusammen. Der Prototyp überführt nutzerdefinierte Workflows automatisch in eine Menge von MapReduce-Jobs und verwaltet deren parallele Ausführung in MapReduce-Clustern. Durch die vollständige Integration der Cloud-Dienste Amazon EC2 und Amazon S3 in Dedoop sowie dessen Verfügbarmachung ist es für Endnutzer ohne MapReduce-Kenntnisse möglich, komplexe Entity Resolution-Workflows in privaten oder dynamisch erstellten externen MapReduce-Clustern zu berechnen.

Inhaltsverzeichnis

Teil I

Einführung

1 Einleitung

1.1 Motivation

Die Studie [89] schätzt, dass im Jahr 2011 weltweit ein Datenvolumen in Höhe von 1.8 Zettabyte erzeugt wurde. Darüber hinaus wird davon ausgegangen, dass sie sich die weltweit gespeicherte Datenmenge etwa alle zwei Jahre verdoppelt. Laut eines IBM-Berichtes desselben Jahres[1] sind 90% der globalen Datenmenge in den vergangenen zwei Jahren erzeugt worden. Getrieben wurde diese Entwicklung im Wesentlichen durch zwei Faktoren. Zum einen verringerten sich laut [89] die Kosten zur Datengenerierung, -speicherung und -verwaltung zwischen 2005 und 2011 um den Faktor 6, zum anderen erfolgte im gleichen Zeitraum weltweit eine Verdopplung der Investitionen in IT-Technologien.

Die Reduzierung der Kosten ging mit der Etablierung von *Infrastructure as a Service*-Diensten einher. Anbieter wie Amazon EC2[2] errichteten riesige Rechenzentren in Regionen mit Standortvorteilen hinsichtlich Lohnkosten, Steuerlast und Elektrizitätskosten. Charakteristisch für diese ist das Prinzip der horizontalen Skalierung. Dies bedeutet, dass der zusätzliche Bedarf bei steigenden Anforderungen nicht durch die Verwendung leistungsfähigerer High-End-Infrastruktur, sondern durch das Hinzufügen einer Menge von kostengünstigeren, lose gekoppelten Rechnern bestehend aus Standard-Hardware kompensiert wird [60]. Die Verwendung von Standard-Hardware ist durch den geringeren Stromverbrauch, die geringeren Anschaffungs- und Administrationskosten und die leichtere Erweiterbarkeit wesentlich kosteneffizienter als der Einsatz von High-End-Hardware [17]. Durch Virtualisierungstechniken besteht zudem die Möglichkeit, verfügbare Ressourcen auf verschiedene Kunden aufzuteilen und damit eine hohe Auslastung der Infrastruktur zu erreichen. Diese Einsparungen konnten zum großen Teil an die Kunden weitergegeben werden. Für diese bietet das beschriebene Geschäftsmodell zudem den Vorteil, dass auf das Betreiben und Administrieren eigener Rechenzentren weitgehend verzichtet werden kann. Statt Rechenkapazitäten für Lastspitzen vorhalten zu müssen, kann benötigte Infrastruktur bei Bedarf schnell akquiriert oder auch wieder freigegeben werden [14].

Gleichzeitig sahen sich Unternehmen mit Herausforderungen konfrontiert, die allgemein durch den Begriff *Big Data* sowie die *5Vs* beschrieben werden [144]. *Data Volume* bezeichnet dabei die bereits angesprochene stetig wachsende Menge an Daten, die mit herkömmlichen Datenverwaltungssystemen nicht mehr effizient gehandhabt werden kann. Beispielsweise musste die Social Media-Plattform Facebook nach eigenen Angaben aus dem Jahr 2012[3] jeden Tag 500 Terabyte neue Daten speichern. Der Begriff *Velocity* drückt die Geschwindigkeit, mit welcher die Daten generiert werden und von den Datenverwaltungssystemen verarbeitet werden müssen, aus. Der Mikroblogging-Dienst Twitter verzeichnete laut eigenen Angaben[4] im August 2013 eine Last von über 143.000 Nachrichten pro Sekunde. *Variety* bezeichnet die wachsende Heterogenität der Herkunft, der Art, des Formats und der Struktur der zu verarbeitenden Daten, was Werkzeuge zur Datenanalyse vor neue Herausforderungen stellt. *Veracity*

[1] http://bruceweed.files.wordpress.com/2011/10/ibm-big-data-success.pdf
[2] http://aws.amazon.com/ec2/
[3] http://techcrunch.com/2012/08/22/how-big-is-facebooks-data-2-5-billion-pieces-of-content-and-500-terabytes-ingested-every-day
[4] https://blog.twitter.com/2013/new-tweets-per-second-record-and-how

bezeichnet die Vertrauenswürdigkeit der zu untersuchenden Daten. Bei der Auswertung der riesigen, aus verschiedenen Quellen stammenden, Datenmengen muss davon ausgegangen werden, dass Daten unsauber oder gar fehlerhaft sind. *Value* bezeichnet die wichtigste Anforderung. Durch die Analyse aller zur Verfügung stehenden Daten unterschiedlicher Struktur, Herkunft und Vertrauenswürdigkeit soll ein wirtschaftlicher Nutzen für das jeweilige Unternehmen, die Organisation oder die Einrichtung abgeleitet werden. Durch diese Anforderungen rückten Technologien zur effizienten Speicherung, Verarbeitung und Analyse großer Datenmengen in den Fokus der Forschung.

Im Jahr 2003 stellte Google ein verteiltes Dateisystem zur fehlertoleranten Speicherung riesiger Datenmengen in Rechnerclustern bestehend aus Standard-Hardware [91] vor. Darauf aufbauend entwickelte Google für die Analyse der verteilt abgespeicherten Daten das MapReduce-Framework[5] [59]. Dabei handelt es sich um die Implementierung eines Programmiermodells, dessen Ursprünge im Bereich der funktionalen Programmierung liegen. Es ermöglicht unter Ausnutzung von Datenparallelität die parallele Ausführung von Berechnungen auf Tausenden von Rechnern und abstrahiert von den Details des verteilten Rechnens sowie der Behandlung von Hardwarefehlern. *Apache Hadoop*[6] ist eine quelloffene Implementierung des MapReduce-Programmiermodells sowie eines darunterliegenden verteilten Dateisystems und bildet die Grundlage zahlreicher Weiterentwicklungen und Forschungsprojekte. Die freie Verfügbarkeit von MapReduce und die Möglichkeit per Mausklick entsprechende Hardware-Ressourcen mieten zu können, machen es sehr attraktiv, verteilte Lösungsansätze für daten- und rechenintensive Probleme verschiedener Domänen zu entwickeln. Beispielsweise findet MapReduce im Bereich des Information Retrieval zum Clustering ähnlicher Dokumente [156] oder zur Parallelisierung von Read Mapping-Algorithmen bei der DNA-Sequenzierung [213] Verwendung.

Diese Arbeit widmet sich der effizienten Parallelisierung von Entity Resolution-Workflows[7] mit MapReduce. Das Entity Resolution-Problem wurde erstmals 1959 untersucht [180] und ist seit vielen Jahren Gegenstand zahlreicher Forschungsarbeiten. Es beschreibt den Prozess der Erkennung von Duplikaten in einer oder mehreren Datenquellen. Entity Resolution ist ein wichtiges Teilproblem der Informationsintegration das in vielen Realweltszenarien Anwendung findet. Typische Anwendungsbeispiele sind die Erkennung von Duplikaten in Personendatenbanken oder das Finden verschiedener Angebote für dasselbe Produkt durch Preisvergleichsportale. Aufgrund fehlender global eindeutiger und quellenübergreifender Identifikatoren ist zur Duplikaterkennung ein paarweiser Vergleich von Datensätzen hinsichtlich verschiedener Ähnlichkeitsmaße notwendig. Die Auswertung des Kartesischen Produktes von n Datensätzen führt zu einer quadratischen Komplexität von $O(n^2)$ und ist dementsprechend nur für kleine Datenquellen praktikabel. Aus diesem Grund bedient man sich sogenannter Blocking- (z. B. Standard Blocking) oder Windowing-Techniken (z. B. Sorted Neighborhood), um den Suchraum zu verkleinern. Dazu wird im ersten Fall ein einfaches Clustering der Datensätze vorgenommen, dessen Ziel es ist, Vergleiche von Datensätzen, die eine gewisse Mindestähnlichkeit unterschreiten, zu vermeiden. Beispielsweise wäre es bei der Deduplizierung von Produktdatensätzen denkbar, lediglich Produkte des gleichen Herstellers miteinander zu vergleichen. Im zweiten Fall werden Datensätze nicht in Cluster partitioniert, sondern anhand eines Sortierschlüssels sortiert, um ähnliche Datensätze "nah beieinander" anzuordnen und die Ähnlichkeitsberechnung auf Datensätze innerhalb eines Maximalabstandes zu beschränken. Trotz der resultierenden Reduktion der zu vergleichenden Datensatzpaare bleibt Entity Resolution ein rechen- und zeitaufwändiger Prozess der ohne Parallelisierung bereits für mittelgroße Datenquellen mehrere Stunden bis Tage in Anspruch nehmen kann [140].

Abbildung 1.1 zeigt einen vereinfachten generischen Entity Resolution-Workflow. Im Blocking-Schritt wird die Ausgangsdatenmenge in möglicherweise überlappende Cluster (Blöcke) partitioniert. Ein Block beschreibt dabei eine Teilmenge der Ausgangsdaten, deren Elemente miteinander verglichen werden sollen. Da die Ähnlichkeitsberechnung eines Datensatzpaares unabhängig von weiteren Datensätzen ist, bietet sich eine Parallelisierung unter Ausnutzung der gegebenen Datenparallelität an. Dazu

[5] *engl.* framework = Rahmenstruktur; bezeichnet in der Softwaretechnik ein Programmiergerüst
[6] `http://hadoop.apache.org`
[7] *engl.* workflow = Arbeitsfluss, Ablaufplan, Arbeitsschritte

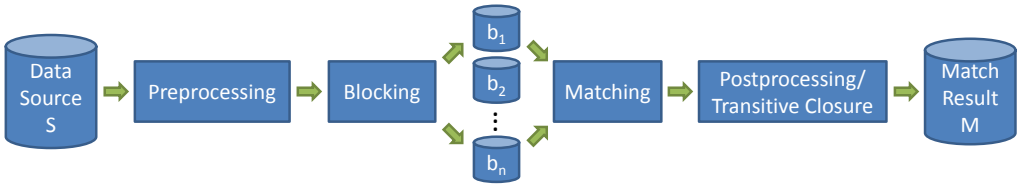


Abbildung 1.1: Vereinfachter Entity Resolution-Workflow

lesen mehrere Prozesse parallel die partitionierten Eingabedaten und verteilen diese entsprechend eines aus den Attributwerten abgeleiteten semantischen Schlüssels auf mehrere Prozesse um. Dabei werden alle Datensätze eines Blocks demselben Prozess zugewiesen, der die aufwändige Ähnlichkeitsberechnung durchführt. Verschiedene Blöcke können dabei parallel von verschiedenen Prozessen bearbeitet werden. Nach dem Vergleich der Datensätze muss basierend auf den berechneten Ähnlichkeiten eine Klassifikation der Kandidatenpaare in *Match* oder *Non-Match* vorgenommen werden. Dies kann für ein Datensatzpaar in den meisten Fällen ebenfalls unabhängig von weiteren Datensätzen erfolgen und ist damit ebenfalls einfach parallelisierbar.

1.2 Wissenschaftlicher Beitrag

Im Rahmen der vorliegenden Arbeit wird ein ganzheitliches Framework zur automatischen und effizienten MapReduce-Parallelisierung nutzerdefinierter Entity Resolution-Workflows vorgestellt, das vom Autor dieser Dissertation in den vergangenen Jahren konzipiert, implementiert und im Rahmen mehrerer Publikationen umfassend evaluiert wurde. Der wissenschaftliche Betrag dieser Dissertation besteht aus den folgenden Arbeiten:

Umsetzung von Blocking-Verfahren: Zur effizienten Berechnung von Entity Resolution-Workflows ist es notwendig, gängige Blocking-Methoden zu unterstützen. Die MapReduce-basierte Umsetzung von Verfahren ähnlich dem Standard-Blocking ist dabei vergleichsweise einfach realisierbar, da ein Datensatz lediglich auf Basis seines semantischen Schlüssels zu einem der p Prozesse umverteilt werden muss, wofür eine beliebige Partitionierungsfunktion $f : \text{Key} \mapsto [0, p)$ verwendet werden kann. Für andere Verfahren wie Sorted Neighborhood, muss f eine Ordnung der Datensätze bezüglich des Schlüssels gewährleisten. Zudem ist es u. U. erforderlich, dass Datensätze die durch f verschiedenen Prozessen zugewiesen werden, miteinander zu vergleichen. Neben der Umsetzung von Blocking-Verfahren werden ebenfalls Verfahren zur MapReduce-basierten Auswertung des Kartesischen Produktes vorgestellt. Um eine sequentielle Berechnung zu vermeiden, muss dabei vom skizzierten Schema abgewichen und eine Strategie zur effizienten Aufteilung aller Paare auf die zur Verfügung stehenden Prozesse gefunden werden.

Lastbalancierung und Behandlung von Speicherengpässen: Die Umverteilung der Datensätze auf Basis des semantischen Schlüssels bedingt, dass alle Datensätze des gleichen Blocks von einem einzelnem Prozess bearbeitet werden. Da Attributwerte von Realweltdaten typischerweise eine Datenungleichverteilung aufweisen, führt der Blocking-Schritt i. d. R. zu Blöcken variierender Größe. Aufgrund der quadratischen Komplexität der paarweisen Ähnlichkeitsberechnungen verursacht dies, dass ein Großteil der Ressourcen eines Rechnerclusters nicht ausgelastet ist, wohingegen wenige Prozesse den Großteil der Arbeit verrichten müssen. Ohne Lastbalancierung stellt die Zeit, die zur Bearbeitung des größten Blocks benötigt wird, eine untere Schranke der erreichbaren Laufzeit dar, was in drastischem Maße die Laufzeiteffizienz und Skalierbarkeit der entsprechenden MapReduce-Programme senkt. Trotz der prinzipiell geringeren Anfälligkeit gegenüber Datenungleichverteilung bestehen bei der MapReduce-Parallelisierung von Windowing-

Verfahren ähnliche Probleme. Aus diesem Grund ist die Konzeption und Umsetzung verschiedener Lastbalancierungsstrategien ein Hauptbestandteil der Dissertation. Ein verwandtes Problem ist die Gefahr von Speicherengpässen. Zum Vergleich aller Datensätze eines Blocks durch einen einzelnen Prozess ist es erforderlich, dass diese vollständig im Hauptspeicher gehalten werden. Da ein Clusterknoten üblicherweise mehrere solcher Prozesse beherbergt unter denen der zur Verfügung stehende Hauptspeicher aufgeteilt werden muss, kann dies für große Datenquellen zu Speicherengpässen führen und möglicherweise einen Rückgriff auf den wesentlich langsameren Externspeicher bedingen. Die in dieser Arbeit vorgestellten Methoden zur Lastbalancierung sind so parametrierbar, dass die Speicherprobleme ebenfalls behoben werden.

Vermeidung redundanter Ähnlichkeitsberechnungen: Bei der Verarbeitung von Realweltdaten ist davon auszugehen, dass Datenqualitätsprobleme wie veraltete, fehlerhafte, inkonsistente oder fehlende Attributwerte vorliegen. Um eine gewisse Robustheit gegen solche Probleme zu gewährleisten, ist es hilfreich, im Blocking-Schritt für jeden Datensatz mehrere von verschiedenen Attributen abgeleitete semantische Schlüssel zu generieren. Dies verspricht eine Erhöhung der resultierenden Match-Qualität, da einander entsprechende Datensätze bezüglich mehrerer Schlüssel einem gemeinsamen Block zugewiesen werden können. Diese Vorgehensweise führt jedoch dazu, dass ähnliche Datensätze unnötigerweise mehrfach bezüglich verschiedener Schlüssel miteinander verglichen werden. Hinzu kommt, dass ein Datensatzpaar für verschiedene Schlüssel i. Allg. von verschiedenen Prozessen bearbeitet wird und eine Inter-Prozess-Kommunikation aus Effizienzgründen vermieden werden soll. Innerhalb der Arbeit werden deswegen Algorithmen zur Vermeidung solch redundanter Ähnlichkeitsberechnungen präsentiert.

Integration maschineller Lernverfahren: Neben dem Mehrfachblocking ist es aufgrund der angesprochenen Datenqualitätsprobleme notwendig, mehrere Attribute der Datensätze in die Ähnlichkeitsberechnung einzubeziehen und verschiedene Ähnlichkeitsfunktionen anzuwenden. Auf Basis der berechneten Ähnlichkeiten muss für jedes Paar entschieden werden, ob die beiden Datensätze dasselbe Realweltobjekt beschreiben oder nicht. Ein typischer Ansatz ist die Bildung eines gewichteten Mittels der einzelnen Ähnlichkeitswerte und der nachfolgende Vergleich mit einem vorgegebenem Mindestähnlichkeitsschwellwert. Mit einer steigenden Anzahl verwendeter Attributwerte und Ähnlichkeitsmaße ist eine manuelle Festlegung einer qualitativ hochwertigen Strategie zur Kombination der resultierenden Ähnlichkeitswerte kaum mehr handhabbar. Aus diesem Grund soll das System neben schwellwert- und ähnlichen regelbasierten Klassifikationsmethoden auch maschinelle Lernverfahren unterstützen, die auf Basis einer vorgegebenen Trainingsdatenmenge automatisiert effektive Regeln zur Klassifikation lernen.

Iterative Berechnung der transitiven Hülle: Ein typischer Nachbearbeitungsschritt bei der Duplikaterkennung ist die Berechnung der transitiven Hülle des Match-Ergebnisses, um weitere, indirekt verbundene, *Matches* zu finden. Zu diesem Zweck wird ein existierender MapReduce-Algorithmus zur iterativen Bestimmung zusammenhängender Komponenten großer Graphen untersucht. Darauf aufbauend werden verschiedene Erweiterungen des Basisalgorithmus vorgestellt, die eine Reduktion des Datenvolumens der Zwischenergebnisse sowie der Anzahl der benötigten Iterationen ermöglichen und zu deutlichen Laufzeitverbesserungen bei der Verarbeitung großer Graphen führen.

MapReduce-basiertes Entity Resolution-Framework: Das Framework Dedoop (Deduplication with Hadoop) ermöglicht eine High-Level-Spezifikation komplexer Entity Resolution-Workflows, die automatisch in eine Menge von MapReduce-Jobs überführt werden. Die Spezifikation erfolgt mithilfe einer graphischen Benutzeroberfläche durch eine Web-Anwendung deren Verwendung keinerlei MapReduce-Kenntnisse voraussetzt. Dabei erlaubt Dedoop die simultane Bearbeitung verschiedener Entity Resolution-Workflows in mehreren MapReduce-Clustern durch mehrere Nutzer. Durch die vollständige Integration der Cloud-Dienste Amazon EC2 und Amazon S3[8] kann

[8] `http://aws.amazon.com/s3/`

zudem sehr einfach zur Laufzeit benötigte Recheninfrastruktur auf Amazons Cloud-Plattform bereitgestellt werden. Das Framework fasst alle in dieser Arbeit vorgestellten Techniken und Optimierungen in einem nutzerfreundlichen System zusammen, um eine effiziente Parallelisierung zu gewährleisten. Zudem unterstützt Dedoop eine parallele Ausführung typischer Nachbearbeitungsschritte, wie z. B. die Bestimmung der Güte der berechneten Duplikatmenge bezüglich der Maße *Precision*, *Recall* und *F-Measure*.

Die in der vorliegenden Arbeit dargestellten Ergebnisse wurden bereits als begutachtete Beiträge im Rahmen internationaler Konferenzen und Workshops sowie in Zeitschriften publiziert. Erste Vorarbeiten zur Parallelisierung von Entity Resolution-Workflows wurden 2010 auf dem internationalen *QDB*-Workshop veröffentlicht [126]. Die Umsetzung des Sorted Neighborhood-Blockings mit MapReduce wurde 2011 auf der deutschen Datenbank-Konferenz *BTW* vorgestellt [131]. Die Arbeit wurde als eine der besten fünf Konferenzbeiträge ausgewählt und erschien daraufhin 2012 in einer erweiterten Version im Journal *Computer Science – Research and Development* [134]. Die Präsentation der MapReduce-basierte Realisierung des Kartesischen Produktes und die Integration maschineller Lernverfahren erfolgte 2011 auf dem internationalen *CloudDB*-Workshop [127]. Ein erster Ansatz zur Lastbalancierung wurde ebenfalls 2011 bei der *CIKM*-Konferenz vorgestellt [130]. Aufbauend darauf erfolgte 2012 die Präsentation zweier Lastbalancierungsansätze für Entity Resolution mit MapReduce bei der *ICDE*-Konferenz [133]. Die Architektur und Funktionsweise des Dedoop-Frameworks wurden ebenfalls im Jahr 2012 im Rahmen der *VLDB*-Konferenz demonstriert [132]. Eine detaillierte Vorstellung des Frameworks sowie eine Evaluation verschiedener Entity Resolution-Strategien für ein komplexes Realweltproblem erfolgte 2013 im Artikel [128] des *Datenbank-Spektrums*. Ein erster Lösungsansatz zur Vermeidung redundanter Ähnlichkeitsberechnungen wurde 2013 auf dem internationalen Workshop *Data Analytics in the Cloud* präsentiert [135]. Im Rahmen einer 2013 auf der *ESWC*-Konferenz vorgestellten Arbeit wurde ein Verfahren zur Distanzberechnung in affinen Räumen mit MapReduce realisiert und mit einer GPU-basierten Implementierung verglichen [183]. Die Arbeit wurde mit dem *Best Paper Award* ausgezeichnet. Parallel dazu wurden weitere Methoden zur Parallelisierung paarweiser Ähnlichkeitsberechnungen unter Verwendung von GPUs untersucht und auf der *DILS*-Konferenz im Jahr 2013 vorgestellt [101]. Zuletzt wurde ein effizienter Algorithmus zur iterativen Berechnung zusammenhängender Komponenten ungerichteter Graphen mit MapReduce entwickelt und in diesem Jahr im Artikel [129] des *Datenbank-Spektrums* veröffentlicht.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in vier Hauptteile. Im Teil I – Einführung – werden verwandte Arbeiten diskutiert und Grundlagen beschrieben:

Kapitel 2 ordnet das Thema der Arbeit in das Forschungsgebiet der Informationsintegration ein, vermittelt Grundlagen zu den in der Arbeit verwendeten Modellen sowie Begriffen und gibt einen detaillierten Überblick über relevante Forschungsarbeiten.

Kapitel 3 stellt das Entity Resolution-Framework *Dedoop* vor, welches eine automatische Parallelisierung nutzerdefinierter Entity Resolution-Workflows in MapReduce-Clustern ermöglicht. Des Weiteren wird eine Übersicht über die wesentlichen, bei der Parallelisierung auftretenden, Probleme gegebenen, deren schrittweise Lösung Gegenstand der darauffolgenden Kapitel ist. Abschließend erfolgt eine Abgrenzung gegenüber bestehenden Ansätzen zur Parallelisierung von Entity Resolution-Workflows (Beiträge [128, 132]).

Der zweite Teil – Paralleles Blocking und Lastbalancierung – untersucht die Fragestellung, wie Verfahren zur Reduzierung des Suchraumes auf Grundlage des MapReduce-Programmiermodells effizient realisiert werden können.

Kapitel 4 und Kapitel 5 befassen sich mit der MapReduce-Umsetzung des Standard Blocking- bzw. des Sorted Neighborhood-Verfahrens. Ein wesentlicher Schwerpunkt dabei ist die Entwicklung und Erprobung von Lastbalancierungsalgorithmen, die eine gleichmäßige Auslastung der verfügbaren Rechenressourcen garantieren (Beiträge [130, 131, 133, 134]).

Kapitel 6 widmet sich der Fragestellung, wie die Berechnung aller Paare von Punkten eines Raumes, deren Distanz kleiner als ein vorgegebener Schwellwert ist, parallelisiert werden kann. Zu diesem Zweck wird eine MapReduce-Implementierung eines kürzlich veröffentlichten sequentiellen Algorithmus vorgeschlagen. Die vorgestellten Techniken können beispielsweise zur Umkreissuche oder zum Clustering von Datensätzen mit einem Maximalabstand verwendet werden (Beitrag [183]).

Der dritte Teil – Weitere Teilprobleme – widmet sich der MapReduce-Umsetzung weiterer Teilschritte des Entity Resolution-Prozesses.

Kapitel 7 betrachtet die Integration maschineller Lernverfahren zur automatischen Klassifikation in das Dedoop-Framework und zeigt, wie die Ähnlichkeitsberechnung aller Datensatzpaare des Kartesischen Produktes zweier Datenquellen mithilfe des MapReduce-Programmiermodells über mehrere Prozessoren und Rechner verteilt werden kann (Beitrag [127]).

Kapitel 8 zeigt, wie redundante Ähnlichkeitsberechnungen vermieden werden können, die entstehen, wenn Datensätze mehreren Clustern zugewiesen werden, um Robustheit gegenüber fehlenden, fehlerhaften oder inkonsistenten Attributwerten gewährleisten zu können (Beitrag [135]).

Kapitel 9 untersucht die Parallelisierung des Nachverarbeitungsschritts zur Berechnung der transitiven Hülle eines Match-Ergebnisses, um neben den direkt berechneten auch weitere, indirekt miteinander verbundene, *Matches* zu finden. Zur Lösung des Problems werden verschiedene Erweiterungen eines aktuellen MapReduce-Algorithmus zur Berechnung zusammenhängender Graphkomponenten vorgeschlagen, mit denen eine deutliche Laufzeitverbesserung im Vergleich zum Basisverfahren erreicht wird (Beitrag [129]).

Abschließend erfolgt im Teil IV eine Zusammenfassung der Ergebnisse der Dissertation sowie ein Ausblick auf zukünftige Arbeiten.

2
Verwandte Arbeiten

Dieses Kapitel gibt einen Überblick über relevante Literatur und Forschungsarbeiten. Zunächst wird im Abschnitt 2.1 eine Einordnung des in der vorliegenden Arbeit untersuchten Entity Resolution-Problems in die Thematik der Informationsintegration vorgenommen. Im Anschluss daran erfolgt im Abschnitt 2.2 eine Vorstellung der einzelnen Teilschritte und existierenden Lösungsstrategien des Entity Resolution-Prozesses. Abschnitt 2.3 stellt das MapReduce-Programmiermodell vor und gibt eine Übersicht über vorgeschlagene Verbesserungen und Erweiterungen. Abschließend wird im Abschnitt 2.4 ein Überblick über Ansätze zur Parallelisierung von Entity Resolution-Workflows und paarweisen Ähnlichkeitsberechnungen gegeben.

2.1 Informationsintegration und Entity Resolution

Entity Resolution ist ein wichtiger Bestandteil der *Informationsintegration*. Der Begriff Informationsintegration beschreibt nach [151] den Prozess der "korrekten, vollständigen und effizienten Zusammenführung" mehrere heterogener autonomer Datenquellen in ein integriertes Informationssystem. Ziel dieses Informationssystems ist die Bereitstellung einer einheitlichen Sicht bzw. Anfrageschnittstelle auf die verschiedenen Datenquellen sowie die Schaffung eines Informationsmehrwertes durch die Verknüpfung von Daten unterschiedlicher Herkunft. Typische Beispiele für integrierte Informationssysteme sind Data Warehouses, Metasuchmaschinen und Preisvergleichsportale.

Im vergangenen Jahrzehnt hat, bedingt durch die stetig wachsende Anzahl frei zugänglicher Datenquellen, die Verknüpfung von Informationen an Bedeutung gewonnen. Laut einer Netcraft-Studie[1] gab es im August 2013 ca. 188 Millionen aktive Websites, was eine Verachtfachung[2] gegenüber dem Jahr 2003 entspricht. Die Verfügbarkeit von interaktiven Web 2.0-Plattformen ermöglichte es auch Benutzern ohne fundierte technische Kenntnisse, Inhalte zu erstellen oder kollaborativ zu bearbeiten und trug damit entscheidend zum Wachstum der verfügbaren Informationsmenge bei. Insbesondere im wirtschaftlichen Kontext hat die Verknüpfung und Analyse verschiedener Informationsquellen eine enorme Bedeutung. Auch die Linked Open Data-Initiative[3], welche die Verknüpfung relevanter frei verfügbarer Datenquellen zum Ziel hat und diese Verknüpfungen der Allgemeinheit zur Weiterverwendung verfügbar macht, ist ein Beispiel für die steigende Relevanz der Informationsintegration. Das *PRISM*-Überwachungsprogramm des amerikanischen Auslandsgeheimdienstes *National Security Agency*, welches die Sammlung, Speicherung, Analyse und Verknüpfung riesiger Mengen an Telekommunikationsmetadaten, Webseiten, Bildern, Nachrichten und Chats zum Zwecke der Aufdeckung terroristischer Bedrohungen beinhaltet [84], zeigt ebenfalls die enorme Bedeutung der Informationsintegration.

Die wesentliche Problematik bei der Informationsintegration ist die unterschiedliche Repräsentation von Daten durch voneinander unabhängigen Datenquellen. [151] teilt die zu überwindenden Unterschiede in verschiedene Kategorien ein. Die *syntaktische Heterogenität* beschreibt Unterschiede in der

[1] `http://news.netcraft.com/archives/2013/08/09/august-2013-web-server-survey.html`
[2] `http://news.netcraft.com/archives/2012/04/04/april-2012-web-server-survey.html`
[3] `http://linkeddata.org`

Darstellung gleicher Dinge in verschiedenen Datenquellen. Typische Beispiele sind unterschiedliche Datums- oder Zahlenformate. Unter dem Begriff *strukturelle Heterogenität* bezeichnet [151] die unterschiedliche Modellierung gleicher Dinge, was sich meist in unterschiedlichen Schemata ausdrückt. Als Beispiel hierfür seien verschiedene Varianten zur Umsetzung einer *is-a*-Beziehung in ein relationales Datenbankschema genannt. Unter *semantische Heterogenität* versteht man eine unterschiedliche Möglichkeiten zur Interpretation der Bedeutung gleicher oder auch verschiedener Daten. Beispielsweise beschreiben Homonyme verschiedene Dinge mit dem gleichen Begriff. Darüber hinaus führt [151] weitere Merkmale auf, in denen sich zu integrierende Datenquellen unterschieden können. Dazu zählt die Schnittstelle für Anfragen an die Datenquellen (z. B. Webservices, HTML-Formulare, SQL-Anfragen), das Kommunikationsprotokoll zum Datenquellenzugriff (z. B. HTTP, JDBC, SOAP), das Datenaustauschformat (z. B. XML, HTML) sowie das verwendete Datenmodell (z. B. hierarchisch oder relational). Zur Überwindung der genannten Heterogenitäten existiert eine Vielzahl von Ansätzen, die sich hinsichtlich verschiedener Kriterien unterscheiden (siehe [67]). In [98] und [258] wird ein historischer Abriss der Ansätze zur Informationsintegration gegeben. Ein aktueller Überblick über die verschiedenen Teilprobleme und Lösungsansätze dieses komplexen Themengebietes findet sich in [65]. Ein zentrales und vielbeachtetes (siehe [203] für eine Übersicht) Teilproblem der Informationsintegration ist das *Schema Matching*, welches die Ermittlung von einander entsprechenden Elementen in den Quellschemata der einzelnen Datenquellen bezeichnet. Die Ermittlung dieser Korrespondenzen ist Voraussetzung für die Erzeugung eines integrierten Schemas des Informationssystems.

Um dem Kriterium der korrekten Zusammenführung von Datensätzen zu genügen und um eine hohe Datenqualität gewährleisten zu können, ist eine nachgelagerte Fehlererkennung und -behebung erforderlich. Dieser Schritt wird als *Data Cleaning* bezeichnet. Eine Datenbereinigung ist aufgrund von Datenqualitätsproblemen notwendig, die sich während des Integrationsprozesses akkumulieren können [204]. Datenqualitätsprobleme liegen entweder bereits in den einzelnen Datenquellen vor (z. B. Schreibfehler, Duplikate, fehlende Integritätsbedingungen) oder entstehen bei der Verknüpfung mehrerer Datenquellen (z. B. widersprüchliche Daten). Taxonomien von Datenqualitätsproblemen finden sich in [190] und [204]. In [18] wird ein Überblick über die Ansätze zur Lösung der verschiedenen Datenqualitätsprobleme gegeben. *Entity Resolution* ist ein Teilproblem des Data Cleanings. Es beschreibt den Prozess der Identifikation von Duplikaten in einer oder mehreren Datenquellen. Ein Duplikat bezeichnet dabei ein Paar von Datensätzen, die dasselbe Objekt der realen Welt beschreiben. Aufgrund von Datenqualitätsproblemen innerhalb einer Datenquelle, der Autonomie und Heterogenität verschiedener Datenquellen sowie des Nichtvorhandenseins von (datenquellenübergreifenden) eindeutigen Datensatzkennungen ist die Duplikaterkennung ein äußerst aufwändiger Prozess. In der Literatur wird Entity Resolution u. a. auch als Object Matching, Deduplication, Record Linkage und Reference Reconciliation bezeichnet. Nach dem Erkennen von Duplikaten müssen diese in einem finalen *Data Fusion*-Schritt zu einem einzelnen Datensatz des integrierten Informationssystems kombiniert werden. Dabei spielt die Erkennung und Korrektur von Fehlern sowie die Behebung von Widersprüchen eine wichtige Rolle. Ein Überblick über Lösungsansätze und Konfliktlösungsfunktionen findet sich in [33] und [69].

Das Entity Resolution-Problem wurde erstmals Ende der 1950er und Anfang der 1960er Jahre untersucht [179, 180]. Eine formale Beschreibung des Problems erfolgte in [81]: Seien A und B zwei Datenquellen. Ein Datensatz $a = \{(A_1, v_1), \ldots, (A_n, v_n)\}$ wird als Menge von Attribut-Wert-Paaren repräsentiert. Ziel der Entity Resolution ist die Aufteilung des kartesischen Produktes $A \times B$ in zwei disjunkte Teilmengen $M = \{(a, b) \in A \times B \mid a = b\}$ und $U = \{(a, b) \in A \times B \mid a \neq b\}$. Dabei bezeichnet M die Menge aller Duplikate (*Matches*[4]) und U die Menge aller Datensatzpaare, die einander nicht entsprechen (*Non-Matches*). Analog zum Zwei-Quellen-Fall kann die Erkennung von Duplikaten innerhalb einer Datenquelle A als die Ermittlung von $M = \{(a_i, a_j) \in A \times A \mid i < j \wedge a_i = a_j\}$ definiert werden. Übersichtsartikel zur Thematik der Entity Resolution finden sich in [51, 77, 94, 245]. Ein verwandtes Forschungsgebiet ist die Link Discovery, welche sich mit der Entdeckung von Beziehungen (z. B. vom Typ `owl:sameAs`) zwischen Instanzen verschiedener *Linked Open Data*-Datenquellen[5] beschäftigt.

[4] *engl.* to match = zusammenpassen, gleichkommen, übereinstimmen
[5] `http://linkeddata.org/data-sets`

2.2 Der Entity Resolution-Prozess

Im Folgenden wird zunächst ein grober Überblick über den Entity Resolution-Prozesses gegeben, bevor die relevante Literatur für die einzelnen Teilschritte in den Abschnitten 2.2.1 bis 2.2.5 vorgestellt wird. Im Abschnitt 2.2.6 erfolgt eine Aufzählung existierender Entity Resolution-Frameworks.

Die Abfolge der einzelnen Teilschritte des Entity Resolution-Prozesses ist in Abbildung 1.1 dargestellt:

- In einem Vorverarbeitungsschritt erfolgt zunächst eine Standardisierung der Struktur der Eingabedatensätze sowie eine Normalisierung der relevanten Attributwerte.
- Der Blocking-Schritt definiert eine Teilmenge $C \subseteq A \times B$ bzw. $C \subseteq A \times A$ des Kartesischen Produktes der Eingabedatenquellen, deren Elemente als Match-Kandidaten bezeichnet werden. Dieser Schritt hat die Eingrenzung potentieller Duplikate zum Ziel.
- Für jeden Match-Kandidaten $c \in C$ wird im Anschluss eine paarweise Ähnlichkeitsberechnung durchgeführt, wobei i. d. R. die Zeichenkettenähnlichkeiten verschiedener Attributwerte bestimmt werden. Die berechneten Ähnlichkeitswerte werden i. d. R. auf das Intervall $[0, 1]$ normalisiert, wobei 0 völlige Unähnlichkeit und 1 totale Übereinstimmung ausdrückt. Die Anwendung von n Ähnlichkeitsfunktionen erzeugt einen Ähnlichkeitsvektor s. Die Ausgabe des Matching-Schrittes besteht dementsprechend aus einer Menge von Tripeln der Form (a, b, s) mit $(a, b) \in C$ und $s \in [0, 1]^n$.
- Basierend auf den bestimmten Ähnlichkeitsvektoren wird anschließend eine Klassifikation der Match-Kandidaten in *Match* oder *Non-Match* vorgenommen. Das Ergebnis des Klassifikationsschrittes besteht üblicherweise nur aus der Menge M, also den *Matches*. M wird in der Literatur auch als Mapping oder Match-Ergebnis bezeichnet. Die Menge der *Non-Matches* wird i. d. R. nicht zurückgegeben oder gar materialisiert. Im Ein-Quellen-Fall kann M auch als eine Menge $C = \{C_1, \ldots, C_n\}$ von Clustern ausgedrückt werden, wobei alle k Datensätze des Clusters $C_i = \{a_{i,1}, \ldots, a_{i,k}\}$ dasselbe Realweltobjekt beschreiben. Die Clusterrepräsentation ist im Vergleich zur Tripelrepräsentation kompakter und benötigt weniger Speicherplatz, allerdings können dabei die im Matching-Schritt berechneten Ähnlichkeiten im Gegensatz zur Tripelrepräsentation nicht erhalten werden. Deswegen ist eine Ableitung der Cluster aus der Tripelrepräsentation möglich, was umgekehrt nicht der Fall ist. Ein Vorteil der Tripelrepräsentation ist die Möglichkeit, das Match-Ergebnis im Nachgang mit weiteren, auf eine andere Art und Weise berechneten, Match-Ergebnissen kombinieren zu können.
- Ein typischer Nachverarbeitungsschritt für den Ein-Quellen-Fall ist die Berechnung der transitiven Hülle des Match-Ergebnisses. Zur Quantifizierung der Güte des Ergebnisses eines Entity Resolution-Prozesses vergleicht man die ermittelte Duplikatmenge mit einem sogenannten, i. d. R. manuell erstellten, perfekten Match-Ergebnis und bestimmt die Maße *Precision* und *Recall*.

2.2.1 Vorverarbeitung

Ziel dieses Schrittes ist es, eine einheitliche Struktur der Datensätze sowie ein einheitliches Format der im Verlaufe des Entity Resolution-Prozesses verwendeten Attribute zu garantieren. Typische Beispiele sind die Konvertierung von Zahlen und Datumsformaten in ein einheitliches Format, die Korrektur von Tippfehlern und das Ersetzen von Abkürzungen durch entsprechende Langformen (z. B. mithilfe von Wörterbüchern) [51]. In einigen Anwendungsfällen ist die Extraktion einzelner Bestandteile eines Attributwertes (z. B. Straße, Hausnummer, PLZ, Ort bei Adressfeldern oder Name, Vorname bei Personennamen) sowie deren Aufteilung auf verschiedene Attribute hilfreich [111]. Auch das Entfernen von sehr häufig vorkommenden sogenannten Stoppwörtern [248], die kaum eine Relevanz für die Beschreibung eines Datensatzes haben, kann für manche Szenarien sinnvoll sein. Eine weitere Operation im Rahmen der Vorverarbeitung ist das Entfernen oder Ersetzen von bestimmten Zeichen wie Punkten, Bindestrichen, Kommata, aufeinanderfolgenden Whitespaces, Anführungsstrichen und Satzzeichen. Üblicher-

weise wird zusätzlich eine Konvertierung der Attributwerte in die Kleinschreibweise vorgenommen. Diese beiden Schritte dienen hauptsächlich dazu, eine Robustheit der später folgenden Ähnlichkeitsberechnung gegenüber kleineren Abweichungen zu gewährleisten [139]. Da die Ähnlichkeitsberechnung für jedes Kandidatenpaar durchzuführen ist, versprechen einfache algorithmische Optimierungen wie z. B. eine Konvertierung von Zeichenketten in numerische Werte oftmals eine erhebliche Beschleunigung. Um z. B. die Ähnlichkeit $dice(A,B) = \frac{2|A \cap B|}{|A|+|B|}$ zweier Token-Mengen A und B zu bestimmen, sind die tatsächlich in A und B enthaltenen Zeichenketten nicht von Interesse. Eine in solchen Fällen häufig vorgenommene Optimierung ist die Erstellung einer Liste aller Token und die Ersetzung jedes Elementes aus A und B durch den Index des jeweiligen Tokens in der Tokenliste [101, 250]. Das Kalkül dahinter ist, dass die Überprüfung, ob zwei Integer-Werte übereinstimmen, wesentlich effizienter ist, als zu prüfen, ob zwei Zeichenketten einander entsprechen.

2.2.2 Blocking

Zur Identifikation von Duplikaten werden Datensätze paarweise miteinander verglichen. Für zwei Datenquellen A und B erfordert dies $|A| \cdot |B|$ Paarvergleiche. Zur Deduplizierung einer einzelnen Datenquelle A müssen $\frac{1}{2} \cdot |A| \cdot (|A| - 1)$ Paare miteinander verglichen werden. Die Anzahl der benötigten Paarvergleiche ist in beiden Fällen quadratisch zur Anzahl der Eingabedatensätze. Zudem sind die pro Vergleich verwendeten Ähnlichkeitsfunktionen (siehe Abschnitt 2.2.3) selbst rechenintensive Operationen. Es ist offensichtlich, dass die Evaluation des Kartesischen Produktes nicht für große Datenquellen skaliert. Dies wurde eindrucksvoll durch eine Studie belegt [140], in der Laufzeiten verschiedener Entity Resolution-Ansätze von über 70 Stunden für die Lösung eines vergleichsweise kleinen Entity Resolution-Problems (ohne Parallelisierung) beobachtet wurden. Zur Effizienzsteigerung kommen sogenannte Blocking-Verfahren zum Einsatz. Diese haben zum Ziel, den Großteil der *non-matches* von der teuren Ähnlichkeitsberechnung auszuschließen ohne dabei fälschlicherweise tatsächliche *Matches* aus der Menge der verbleibenden Kandidatenpaare zu entfernen. Der Blocking-Schritt definiert also eine Menge von Match-Kandidaten, die in den nachfolgenden Schritten verifiziert werden. Die Güte eines Blocking-Verfahrens wird durch zwei Kennzahlen beschrieben. Die Effizienz eines Blocking-Verfahrens wird durch die *Reduction Ratio* ausgedrückt, welche die Reduktion der zu vergleichenden Paare im Vergleich zur Auswertung des Kartesischen Produktes quantifiziert. Die *Pairs Completeness* beschreibt den Anteil der tatsächlichen Duplikate, die nach dem Blocking in der Kandidatenmenge verbleiben, und quantifiziert damit die Qualität eines Blocking-Verfahrens. In der Literatur bezeichnet man Blocking auch als Indexing [51].

Während des Blockings erfolgt entweder eine Gruppierung oder eine Sortierung der vorverarbeiteten Datensätze. In beiden Fällen hat dies ein Clustering ähnlicher Datensätze zum Ziel. Da sich potentielle Duplikate eines einzelnen Datensatzes anschließend in dessen "Nähe" befinden, können Vergleiche mit weit entfernten (unähnlichen) Datensätze eingespart werden [178]. Die Gruppierung bzw. Sortierung erfolgt auf Basis sogenannter Block- bzw. Sortierschlüssel, die von den Attributwerten der Datensätze abgeleitet und leicht zu berechnen sind. Ein Schlüssel stellt dabei eine Signatur eines Datensatzes dar. Ein Überblick über die populärsten Blocking-Techniken wird in [20] gegeben. Eine systematische Evaluierung verschiedener Variationen mehrerer Blocking-Techniken bezüglich Effizienz und Skalierbarkeit erfolgte in [50]. Im Folgenden wird ein Überblick über die bekanntesten Blocking-Techniken gegeben.

Standard Blocking ist das bekannteste und am längsten verwendete Blocking-Verfahren [81]. Jedem Datensatz wird dabei ein Blockschlüssel zugewiesen. Die Blockschlüssel dienen konzeptionell als Schlüssel eines invertierten Indexes, der Datensätze anhand einer Signatur gruppiert. Eine Gruppe von Datensätze desselben Blockschlüssels bezeichnet man als Block. In der Folge werden ausschließlich Datensätze innerhalb eines Blocks miteinander verglichen. Zum Zwecke der Volkszählung wurde in der Vergangenheit beispielsweise der *Soundex*-Algorithmus [188] auf Familiennamen angewendet, um ähnlich klingende Namen mit einer identischen Zeichenfolge zu codieren (und zu gruppieren). Zur Deduplizierung von Produktdatensätzen wäre es hingegen denkbar das Herstellerattribut eines jeden

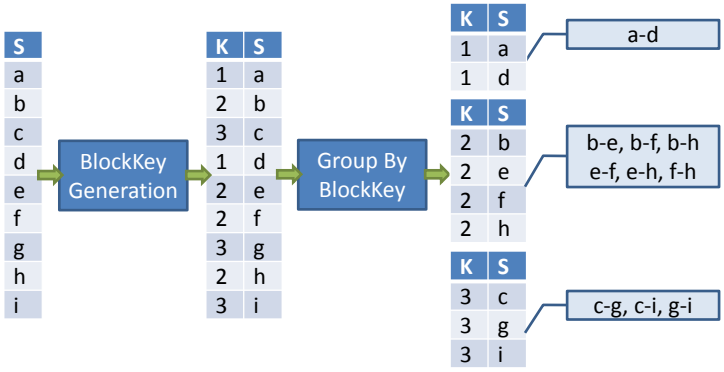


Abbildung 2.1: Beispielhafte Ausführung des Standard Blocking-Verfahrens.

Datensatzes als Blockschlüssel zu verwenden, um paarweise Vergleiche auf Produkte desselben Herstellers zu beschränken. Die Anzahl der verbleibenden Kandidatenpaare hängt von der Größe der einzelnen Blöcke und damit von der Häufigkeitsverteilung der generierten Blockschlüssel ab. Somit kann ohne eine Analyse der Datenquellen vorab keine Aussage über die *Reduction Ratio* getroffen werden. Abbildung 2.1 illustriert das Standard Blocking-Verfahren für eine Datenquelle S bestehend aus neun Datensätzen $a-i$. Zunächst wird für jeden Datensatz ein Blockschlüssel (hier 1, 2 oder 3) berechnet, anschließend wird die Ähnlichkeit aller Datensätze desselben Blocks berechnet. Bei der Generierung der Blockschlüssel können fehlende oder fehlerhafte Werte der verwendeten Attribute dazu führen, dass einander entsprechende Datensätze fälschlicherweise verschiedenen Blöcke zugewiesen und deswegen nicht als Duplikate erkannt werden. Diesem Nachteil kann entgegengegengewirkt werden, indem für einen Datensatz mehrere Blockschlüssel (z. B. unter Verwendung verschiedener Attribute und/oder verschiedener Berechnungsvorschriften) bestimmt werden (Multi-pass Blocking).

Q-gram Indexing wurde zuerst in Form des *Bigram Indexings* [20] im Entity Resolution-Framework *Febrl* [52] verwendet. Die grundlegende Idee besteht darin, im Gegensatz zum reinen Standard Blocking auch Datensätze mit unterschiedlichen aber ähnlichen Blockschlüsseln miteinander zu vergleichen. Dazu wird ein Blockschlüssel in eine Liste L von q-Grammen überführt. Ein q-Gramm entspricht einem Substring der Länge q des ursprünglichen Blockschlüssels. Beispielsweise würde mit $q = 2$ der Blockschlüssel *Apple* in eine Liste $L = [ap, pp, pl, le]$ zerlegt werden. Anschließend werden *alle* Sublisten von L mit einer Länge zwischen $\max\{1, \lfloor L.size() \cdot t \rfloor\}$ und $L.size()$ generiert, wobei $t \in [0, 1]$ ein nutzerdefinierter Parameter ist. Für das Beispiel und $t = 0.9$ werden demnach alle Sublisten von L der Länge 3 ($[pp, pl, le]$, $[ap, pl, le]$, $[ap, pp, le]$, $[ap, pp, pl]$) sowie der Länge 4 ($[ap, pp, pl, le]$) bestimmt. Anschließend werden die einzelnen q-Gramme einer jeden Subliste zu einem neuen Blockschlüssel verkettet. Für den Beispieldatendatz würden also die Blockschlüssel *ppplle*, *applle*, *appple*, *appppl* und *appppplle* generiert werden. Dies hat den Vorteil, dass er auch mit einem Datensatz verglichen werden würde, dem ursprünglich der Blockschlüssel *Aple* (fehlender Buchstabe) oder *Aplle* (Tippfehler) zugewiesen wurde. Der Nachteil dieser Methode ist der hohe Aufwand zur Berechnung aller möglichen Sublisten: Da ein Blockschlüssel bestehend aus n Zeichen in $k = n - q + 1$ q-Gramme zerlegt wird, müssen für den Datensatz insgesamt $\Sigma_{i=\max\{1, \lfloor k \cdot t \rfloor\}}^{k} \binom{k}{i}$ Sublisten berechnet und zu neuen Blockschlüsseln verkettet werden [50].

Suffix Array Indexing [5] ist ebenfalls ein Blocking-Verfahren, das aus dem ursprünglichen Blockschlüssel eines Datensatzes mehrere neue Schlüssel ableitet und den Datensatz anhand dieser neu generierten Schlüssel mehreren Blöcken zuweist. Die Grundidee ist, alle Suffixe des Ausgangsschlüssels mit einer Mindestlänge von l zu bestimmen und jeden dieser Suffixe als neuen Blockschlüssel zu verwenden. Ein Datensatz mit einem Blockschlüssel der Länge n wird demzufolge in $n-l+1$ verschiedene Blöcke eingeordnet. Sollte die Länge des Ausgangsschlüssels kleiner als l sein, wird der Ausgangs-

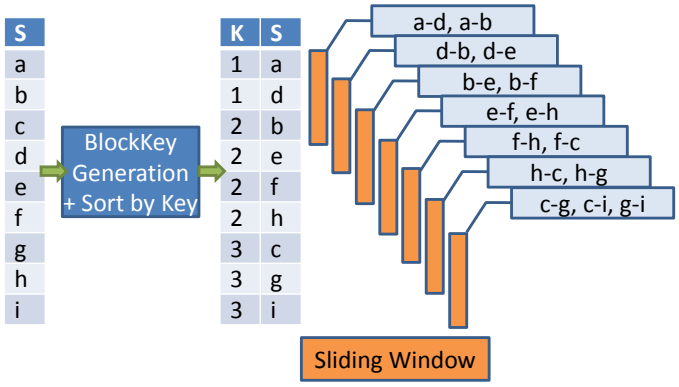


Abbildung 2.2: Beispielhafte Ausführung des Sorted Neighborhood-Verfahrens mit $w = 3$.

schlüssel als einziger Blockschlüssel verwendet. Für $l = 3$ würde ein Datensatz mit einem Ausgangsschlüssel *Apple* den Blöcken mit den Blockschlüsseln *apple*, *pple* und *ple* zugewiesen werden. Für kurze Schlüssel ist zu erwarten, dass die entsprechenden Blöcke viele Datensätze enthalten und sehr vielen Kandidatenpaare generieren. Zudem ist es wahrscheinlich, dass Duplikate auch einen weiteren (längeren) gemeinsamen Blockschlüssel besitzen. Aus diesem Grund werden aus Blöcken, deren Größe einen bestimmten Schwellwert überschreitet, alle Datensätze entfernt, die mindestens einen weiteren (längeren) Blockschlüssel haben. Gibt es beim Ein-Quellen-Fall n Blöcke der maximal erlaubten Größe s, so stellt $n \cdot \frac{s \cdot (s-1)}{2}$ eine obere Schranke für die Anzahl der generierten Kandidatenpaare dar [51]. Genau wie beim Q-gram Indexing ist die Anzahl der generierten Kandidatenpaare i. Allg. deutlich größer als beim strikten Standard Blocking mit Verwendung eines einzelnen Blockschlüssels pro Datensatz. Zudem ist es wahrscheinlich, dass zwei Datensätze unnötigerweise mehrfach (bezüglich verschiedener Blockschlüssel) miteinander verglichen werden, was Techniken zur Vermeidung redundanter Paarvergleiche erfordert. Diesen Nachtteilen steht jedoch eine höhere *Pairs Completeness* gegenüber, da die Wahrscheinlichkeit für die Eliminierung von tatsächlichen Duplikaten durch den Blocking-Schritt, aufgrund der Zuweisung eines Datensatz zu mehreren Blöcken, geringer ist. Im Vergleich zum Q-gram Indexing ist der Aufwand zur Berechnung aller Blockschlüssel deutlich geringer, allerdings erschwert die Notwendigkeit des Vorhandenseins von globalem Wissen (Gesamtanzahl von Datensätzen eines Blocks) eine verteilte Implementierung.

Sorted Neighborhood wurde 1995 zur effizienten Erkennung von Duplikaten in Datenbanktabellen vorgeschlagen [104]. Es handelt sich um ein Blocking-Verfahren, das Datensätze nicht in Blöcke partitioniert, sondern Datensätze anhand eines Sortierschlüssels sortiert, um ähnliche Datensätze "nah beieinander" anzuordnen. Analog zu den vorab vorgestellten Verfahren, wird für jeden Datensatz zunächst ein Schlüssel generiert. Nach der Sortierung aller n Datensätze bezüglich dieser Schlüssel wird ein Fenster der Größe $w \in [2, n]$ schrittweise über die Liste der sortierten Datensätze bewegt. Abbildung 2.2 illustriert das Sorted Neighborhood-Verfahren für eine Datenquelle S (Datenquelle und Blockschlüssel analog zu Abbildung 2.1). Die Datensätze werden zunächst anhand ihrer Blockschlüssel sortiert. Das Fenster der Größe $w = 3$ umfasst initial die Datensätze a, d, b und wird anschließend schrittweise um eine Position weitergeschoben. Insgesamt gibt es $n - w + 1$ verschiedene Fensterpositionen. Für die ersten $n - w$ Fensterpositionen wird der jeweils erste Datensatz mit den $w - 1$ nachfolgenden Datensätzen verglichen. Im letzten Schritt werden alle w Datensätze miteinander verglichen. Durch diese Berechnungsvorschrift wird jeder Datensatz genau einmal mit den $w-1$ vorherigen und den $w-1$ nachfolgenden Datensätzen verglichen. Insgesamt ergeben sich durch das Sorted Neighborhood-Verfahren $(n - \frac{w}{2}) \cdot (w - 1)$ Paarvergleiche. Im Vergleich zur Evaluierung des Kartesischen Produktes wird die Gesamtkomplexität von $O(n^2)$ auf $O(n) + O(n \cdot \log n) + O(n \cdot w)$ für die Blockschlüsselbestimmung, die Sortierung und die Ähnlichkeitsberechnung reduziert. Das Sorted Neighborhood-Verfahren eignet

sich hauptsächlich zur Deduplizierung einer einzelnen Datenquelle. Sollen Duplikate zwischen zwei Datenquellen erkannt werden, sind die sortierten Datensätze beider Datenquellen zu mischen. Dies kann jedoch dazu führen, dass sich innerhalb eines Fensters vorrangig Datensätze derselben Quelle befinden, die nicht miteinander verglichen werden sollen, und dafür ähnliche Datensätze verschiedener Quellen zu weit auseinander liegen. Der Vorteil des Sorted Neighborhood-Verfahrens gegenüber den vorigen Blocking-Techniken ist, dass die Anzahl der Kandidatenpaare alleinig von der Anzahl der Datensätze sowie der verwendeten Fenstergröße und *nicht* von der Häufigkeitsverteilung der Schlüsselattributwerte abhängt. Im Gegensatz zum Q-gram Indexing und Suffix Array Indexing sind die Präfixe der verwendeten Schlüssel beim Sorted Neighborhood-Verfahren von entscheidender Bedeutung. Zwei Duplikate, deren Blockschlüssel sich (z. B. aufgrund eines Tippfehlers) lediglich im ersten Zeichen unterscheiden, werden nicht als Duplikate erkannt, wenn sich dazwischen $w-1$ oder mehr Datensätze befinden.

In einer weiteren Studie [105] zeigten dieselben Autoren, dass eine mehrfache Ausführung des Basisverfahrens unter Verwendung verschiedener Sortierschlüssel (abgeleitet von verschiedenen Attributen) eine Erhöhung der *Pairs Completeness* verspricht. Parallel dazu kann die ursprüngliche verwendete Fenstergröße w und damit die Anzahl der Kandidatenpaare deutlich verringert werden ohne nennenswerte Qualitätseinbußen hinnehmen zu müssen. Ein solcher Multi-Pass-Ansatz ist insbesondere ratsam, um die Anfälligkeit gegenüber Fehlern in den Sortierschlüsselpräfixen zu verringern. Auch eine nachgelagerte Berechnung der transitiven Hülle aller gefundenen Paare kann die Verwendung einer kleineren Fenstergröße ohne wesentliche Verringerung der resultierenden *Pairs Completeness* erlauben [176].

Sowohl bei der klassischen Variante als auch beim Multi-Pass-Verfahren besteht das Problem, dass die Fenstergröße w größer als die Anzahl der Datensätze mit dem am häufigsten auftretenden Sortierschlüssel k sein sollte: Für n Datensätze mit dem Sortierschlüssel k muss $w = n + m$ gelten, damit der erste Datensatz sowohl mit den $n-1$ übrigen Datensätzen mit dem Schlüssel k, als auch mit den ersten m Datensätzen des nachfolgenden Sortierschlüssels verglichen wird. Die Ausrichtung der Fenstergröße anhand des am häufigsten auftretenden Sortierschlüssels kann jedoch dazu führen, dass Datensätze eines selten auftretenden Sortierschlüssels unnötigerweise mit "weit entfernten" Datensätzen verglichen werden. Aus diesem Grund wurde in [50] eine Anpassung des ursprünglichen Sorted Neighborhood-Algorithmus vorgeschlagen. Ähnlich wie beim Standard Blocking wird zunächst ein invertierter Index aufgebaut, der jedoch als sortierte Hashtabelle realisiert ist. Für jeden Sortierschlüssel wird dabei die Menge aller Datensätze, die diesen Schlüssel aufweisen, als Liste abgespeichert. Anschließend wird das Fenster der Größe w über die sortierten *Schlüssel* der Hashtabelle bewegt. In jedem Schritt werden dabei die Datensätze aller Sortierschlüssel, die sich innerhalb des Fensters befinden, miteinander verglichen. Wie beim Standardverfahren werden dabei zwei Datensätze maximal einmal miteinander verglichen. Für $w = 1$ entspricht diese Variante dem Standard Blocking. Der Nachteil dieser Variante gegenüber dem klassischen Sorted Neighborhood ist, dass (analog zum Standard Blocking, Q-gram Indexing und Suffix Array Indexing) der am häufigsten vorkommende Schlüssel die Anzahl der generierten Kandidatenpaare und damit die benötigte Zeit zur Ähnlichkeitsberechnung dominiert [50].

Das *Sorted Blocks*-Verfahren [70, 72] verfolgt eine ähnliche Idee. Es strebt eine Verallgemeinerung von Windowing- und klassischen Blocking-Verfahren an. Wie beim Sorted Neighborhood werden Datensätze zunächst anhand eines Sortierschlüssels sortiert. Anschließend erfolgt eine Gruppierung adjazenter Datensätze in disjunkte Partitionen. Die Partitionierung erfolgt auf Basis derselben Attribute, die zur Sortierung verwendet werden, die Autoren schlagen die Verwendung von Sortierschlüsselpräfixen vor. Analog zum Standard Blocking werden alle Datensätze einer Partition miteinander verglichen. *Zusätzlich* wird ein Fenster mit einer festen Größe über die Partitionsgrenzen geschoben, um die Erkennung von Duplikaten mit ähnlichen Sortierschlüsseln zu ermöglichen, die verschiedenen Partitionen zugewiesen wurden: Umfasst die i-te Partition n Datensätze, so wird das Fenster der Größe w beginnend vom $\max\{1,\ n-w+1\}$-ten Datensatz dieser Partition in $w-2$ Schritten um jeweils eine Position in Sortierreihenfolge verschoben. In jedem Schritt wird der erste Datensatz des Fensters, gehörend zur Partition i, mit allen anderen Datensätzen innerhalb des Fensters verglichen, die einer anderen Partiti-

on $j > i$ angehören. Dieses Vorgehen wird rekursiv für die Nachfolgepartitionen fortgesetzt. Für $w = 0$ entspricht das Verfahren dem Standard Blocking. Umfasst jede Partition nur einen Datensatz entspricht das Verfahren dem klassischen Sorted Neighborhood. Um zu vermeiden, dass die größte Partition die Berechnungszeit dominiert, wird u. a. die Aufteilung von Partitionen mit einer Größe oberhalb eines Schwellwertes in Subpartitionen vorgeschlagen.

Adaptive Sorted Neighborhood ist ein dritter Ansatz [253], um das Problem der Wahl einer optimalen Fenstergröße zu lösen. Hierbei wird das Fenster solange vergrößert bis der erste und der letzte Sortierschlüssel innerhalb des Fensters eine gewisse Mindestähnlichkeit bezüglich eines einfach zu berechnenden Ähnlichkeitsmaßes unterschreiten. Nach dem Vergleich aller enthaltenen Datensätze wird das Fenster auf die Ausgangsgröße zurückgesetzt und an die Position des Datensatzes verschoben, der zum Abbruch der Fenstervergrößerung führte. In [73] konnten jedoch, u. a. aufgrund des zusätzlichen Aufwandes zur Schlüsselähnlichkeitsberechnung keine wesentlichen Verbesserungen dieses Ansatzes gegenüber der klassischen Strategie beobachtet werden. Stattdessen schlugen die Autoren vor, die Fenstergröße beginnend von $w = 2$ solange zu erhöhen bis die Anzahl der durchschnittlich pro Vergleich gefundenen Duplikate einen Mindestschwellwert unterschreitet. In der Basisvariante dieses Ansatzes wird das Fenster dabei um jeweils einen Datensatz erweitert. In einer zweite Variante wird das Fenster nach jedem gefundenen Duplikat um $w-1$ ("momentane Größe minus 1") Datensätze erweitert. Werden dabei zwei Paare (a,b) und (a,c) als Duplikate identifiziert, wird das Paar (b,c) ebenso zum Match-Ergebnis hinzugefügt. Dieser Schritt erlaubt es, in der Folge Vergleiche des Datensatzes b mit den nachfolgenden $w-1$ Datensätzen einzusparen.

Canopy Clustering verfolgt die Idee, Datensätze mithilfe einer einfach zu berechnenden Abstandsfunktion in überlappende Cluster zu partitionieren, die als *Canopies* bezeichnet werden [57, 162]. Jedes Cluster entspricht einem Block, dessen Datensätze miteinander zu vergleichen sind. Zur Clustergenerierung wird aus der Menge der Datensätze (Kandidatenliste) zufällig ein Zentroid eines neuen Clusters gewählt. Anschließend werden dem Cluster alle Datensätze zugewiesen, deren Abstand zum Zentroiden einen Schwellwert d_1 nicht überschreiten. Zusätzlich werden alle Datensätze dieses Clusters deren Abstand zum Zentroiden unterhalb eines weiteren Schwellwertes $d_2 < d_1$ liegen, aus der Kandidatenliste entfernt, um deren Zuordnung zu einem anderen Cluster zu verhindern. Dieser Algorithmus wird solange fortgesetzt bis die Kandidatenliste leer ist. Die Anzahl der generierten Kandidatenpaare hängt von den beiden Schwellwerten, der verwendeten Distanzfunktion sowie der Werteverteilung der zur Abstandsberechnung verwendeten Attribute ab. Eine Erweiterung zur Begrenzung der Anzahl der Kandidatenpaare wurde in [50] vorgeschlagen.

Mapping-basiertes Blocking bildet Datensätze anhand von Blockschlüsseln in einen mehrdimensionalen Euklidischen Raum einer vorgegeben Dimensionalität ab [121]. Die Abbildung ist distanzerhaltend und erfolgt auf Basis verschiedener Erweiterungen des *FastMap*-Algorithmus [3, 79, 114]. Dieses Verfahren erfordern die Verwendung metrischer Abstandsmaße wie z. B. die *Levenshtein-Distanz*. Die Bildmenge wird anschließend, ähnlich wie beim Canopy Clustering, in eine Menge von überlappenden Blöcken geclustert. Auch im Bereich der Link Discovery wurden Verfahren entwickelt, die einen mehrdimensionalen Index generieren um alle Datensatzpaare, deren Abstand unterhalb eines vorgegebenen Schwellwertes liegt, effizient zu bestimmen [119, 181].

Die Art und Weise der Schlüsselgenerierung hat einen entscheidenden Einfluss auf die resultierende Effizienz und Qualität eines Entity Resolution-Workflows. Ist das Blocking-Kriterium zu "scharf" (z. B. vollständige Übereinstimmung eines Attributwertes) werden Duplikate möglicherweise nicht erkannt. Ist das Kriterium jedoch zu "lax" entstehen große Cluster, die zu einer hohen Anzahl an Match-Kandidaten führen und die Effizienz des Blocking-Verfahrens verringern. Bei der Auswahl der verwendeten Attribute ist insbesondere die Anzahl von Datensätze mit nicht-leeren Attributwerten sowie die Häufigkeitsverteilung der Attributwerte zu berücksichtigen [50]. Regeln zur Blockschlüsselgenerierung werden meist von Domänenexperten aufgestellt, die die Charakteristika der verwendeten Attribute kennen. Alternativ können Blockschlüssel auch automatisch durch maschinelle Lernverfahren erzeugt werden [27, 169]. Auf Basis von manuell gelabelten Trainingsdaten werden dabei sowohl die zu verwendenden Attribute

Verfahren	Ansatz/Grundidee	Vergleich von Datensätzen mit unterschiedlichen Ausgangsschlüsseln / Überlappende Cluster	Möglichkeit redundanter Paarvergleiche
Standard Blocking	Blocking (Gruppierung nach Blockschlüssel)	nur bei Multi-pass	nur bei Multi-pass
Q-gram Indexing	Ableitung mehrerer ähnlicher Schlüssel aus Ausgangsblockschlüssel & Blocking	✓	✓
Suffix Array Indexing	Ableitung mehrerer ähnlicher Schlüssel aus Ausgangsblockschlüssel & Blocking	✓	✓
Mapping-basiertes Blocking	Abbildung der Datensätze in mehrdimensionalen Euklidischen Raum & Clustering (distanzbasiert oder mittels Ähnlichkeitsschwellwert)	✓	✓
Canopy Clustering	Distanzbasiertes Clustering	✓	✓
Sorted Neighborhood	Windowing (Sortierung nach Sortierschlüssel & schrittweises Verschieben des Fensters)	✓	nur bei Multi-pass

Tabelle 2.1: Übersicht der vorgestellten Blocking-Verfahren zur Definition von Match-Kandidaten.

als auch Bildungsvorschriften für Blockschlüssel gelernt.

Tabelle 2.1 gibt einen zusammenfassenden Überblick über die vorgestellten Blocking-Verfahren. Die wichtigsten und am häufigsten eingesetzten Verfahren sind das Standard Blocking sowie das Sorted Neighborhood-Verfahren. Aus diesem Grund beschränkt sich die Arbeit im weiteren Verlauf auf die effiziente MapReduce-Parallelisierung dieser beiden Basisverfahren, aus denen sich weiteren Verfahren, wie das Q-gram Indexing, das Suffix Array Indexing und das Sorted Blocks-Verfahren ableiten lassen.

2.2.3 Ähnlichkeitsberechnung

Nach der Ermittlung der Match-Kandidaten im Blocking-Schritt erfolgt eine paarweise Ähnlichkeitsberechnung. Um die Ähnlichkeit zweier Datensätze zu quantifizieren, werden i. Allg. mehrere Ähnlichlichkeitsfunktionen auf Attributwerte der Datensätze angewendet. Neben den Attributwerten können auch Kontextinformationen der Datensätze, z. B. unter Ausnutzung der referenziellen Integrität in Datenbanksystemen zur Ähnlichkeitsbestimmung herangezogen werden. Statt Ähnlichkeitsfunktionen können auch Abstandsmaße verwendet werden; die errechneten Abstände müssen anschließend in Ähnlichkeitswerte umgewandelt werden [100].

Field Matching [172] ist ein Oberbegriff für Verfahren, die die Ähnlichkeit von Datensätzen alleinig aus der Ähnlichkeit von deren Attributwerten ableiten. Neben der Verwendung spezieller Ähnlichkeitsfunktionen, z. B. für Geokoordinaten, Zeitstempeln oder numerische Werte kommen dabei v. a. Zeichenkettenähnlichkeitsmaße zum Einsatz. Für viele Maße gibt es frei verfügbare Implementierungen, z. B. im Rahmen des *SimMetrics*[6] oder des *SecondString*[7]-Projektes. Ein Vergleich verschiedener Zeichenkettenähnlichkeitsmaße wurde in [56] und [177] durchgeführt. Die bestehenden Ähnlichkeitsmaße gliedern sich grob in editierdistanzbasierte und tokenbasierte Verfahren. Maße der ersten Gruppe bestimmen die benötigte Anzahl an Änderungsoperationen, um eine Zeichenkette in eine andere zu konvertieren und überführen den ermittelten Abstand in eine Zeichenkettenähnlichkeit. Bekannte Verfahren umfassen die *Levenshtein*-Distanz [152], die *Jaro-Winkler*-Distanz [244], die *Smith-Waterman*-Distanz [172, 221] sowie im weiteren Sinne auch den Soundex-Algorithmus [188, 243].

[6] `http://sourceforge.net/projects/simmetrics`
[7] `http://sourceforge.net/projects/secondstring`

Tokenbasierte Maße zerlegen die zu vergleichenden Zeichenketten in Tokenmengen[8] und berechnen anschließend deren Ähnlichkeit. Die Zerlegung kann anhand von Leerräumen, bestimmten Zeichen (z. B. Kommata) oder in q-Gramme erfolgen. Auch ein Entfernen von Stoppwörtern kann in diesem Schritt sinnvoll sein. Der *StandardAnalyzer* des *Lucene*[9]-Projektes ist eine beliebter Tokenizer, der Zeichenketten, unter Verwendung einer (erweiterbaren) komplexen Grammatik, in Wörter trennt. Zur Bestimmung der Ähnlichkeit zweier Tokenmengen existieren ebenfalls verschiedene Methoden. Sehr populär ist der *Jaccard-Koeffizient*, der die Größe der Schnittmenge mit der Größe der Vereinigungsmenge in Verhältnis setzt. Ein ähnliches Maß ist der *Dice*-Koeffizient, der die Ähnlichkeit zweier Mengen A, B mit $2 \cdot (|A \cap B|)/(|A| + |B|)$ definiert. Ein weiteres Beispiel ist die *TF-IDF*-Ähnlichkeit [210], welche ein einzelnes Token zunächst anhand dessen Auftretenshäufigkeit in der gesamten Datenquelle sowie anhand der Auftretenshäufigkeit in den zu vergleichenden Tokenmengen wichtet. Zwei Tokenmengen A, B werden in der Folge als Vektor eines $|A \cap B|$-dimensionalen Vektorraumes dargestellt. Die Komponenten eines Vektors entsprechen den TF-IDF-Gewichten der korrespondierenden Tokenmenge. Die Ähnlichkeit zweier Tokenmengen ergibt sich aus dem Winkel zwischen beiden Vektoren. Die verschiedenen Maße weisen eine unterschiedliche Eignung für bestimmte Attributtypen auf, beispielsweise eignen sich der Soundex-Algorithmus oder die Jaro-Winkler-Distanz besonders zum Vergleich von Personennamen. In [28, 30, 171] wurden Verfahren vorgeschlagen, die mithilfe maschineller Lernverfahren optimale Ähnlichkeitsmaße für die zu vergleichenden Attribute bestimmen sowie eine geeignete Parameterkonfiguration (z. B. Kosten für das Löschen, Ändern oder Einfügen eines Zeichens) der Ähnlichkeitsmaße ermitteln.

Kontextbasierte Verfahren nutzen zur Ähnlichkeitsbestimmung neben den Attributwerten auch assoziierte Daten, die mit den zu vergleichenden Datensätzen in Beziehung stehen. Ein einfacher Ansatz ist das Hinzufügen von Attributen assoziierter Datensätze zu den Datensätzen der Primärquelle. Zusätzlich zum Field Matching wurde in [9] die Ausnutzung von Dimensionshierarchien vorgeschlagen, um Duplikate in Datawarehouses aufzuspüren. In [149, 167, 168] wurde eine semiautomatische Anreicherung von Basisdatensätzen um Informationen aus Sekundärquellen, z. B. auf Basis anhand von gelernten Assoziationsregeln, untersucht. Um relevante Kontextattribute zu finden, kommen dabei domänenspezifische Konzepthierarchien zum Einsatz.

Zur Duplikaterkennung in hierarchischen Daten, wie z. B. XML, kann, ähnlich zum Schema Matching, neben den eigentlichen Knoteninformationen auch die vorhandene Struktur ausgenutzt werden [39, 150, 239, 240]. Typische Ansätze beim Vergleich zweier Knoten sind die Einbeziehung der jeweiligen Vater- und Kindknoten. Zusätzlich kann die strukturelle Ähnlichkeit zweier Knoten, wie z. B. die Ähnlichkeit der Pfade zur Wurzel des Dokumentes, zur Ähnlichkeitsbestimmung herangezogen werden.

Des Weiteren existieren Ansätze, die auf einer graphbasierten Darstellung der Datenquellen basieren [24, 46, 68, 123]. Datensätze werden dabei als attributierte Knoten in einem Graph dargestellt; assoziierte Datensätze sind mittels Kanten verbunden. In [46] wird zunächst eine attributwertbasierte Ähnlichkeitsberechnung vorgenommen, um die Menge möglicher Duplikatknoten einzugrenzen. Anschließend wird durch den Vergleich assoziierter Knoten ermittelt, wie stark zwei Knoten zusammenhängen. Auf Basis der Attributähnlichkeiten und der Verbindungsstärke wird der Graph abschließend in Cluster partitioniert, deren Knoten einander entsprechende Datensätze beschreiben. In [68] wird ein Abhängigkeitsgraph verwendet, indem ein Knoten einen Match-Kandidaten repräsentiert. Eine Kante zwischen zwei Knoten signalisiert, dass die Ähnlichkeitsberechnung zweier Datensätze von der Ähnlichkeit eines anderen Kandidatenpaars (eines anderen Typs) abhängt. In einem iterativen Prozess werden Knoten mittels Field Matching verglichen. Nach der Klassifikation eines Knotens als Duplikat wird diese Information an die abhängigen Nachbarknoten weiterpropagiert. Wurden z. B. zwei Publikationen als Duplikate erkannt, so können die assoziierten Autoren ebenfalls zusammengeführt werden. Die Verwendung von Attributwertkollektionen als Knoten wurde in [220] vorgeschlagen. Werden zwei Knoten als Duplikate erkannt, so kann die Entsprechung unterschiedlicher Attributwerte (bedingt durch eine

[8] *engl.* token = Zeichen, Zeichenfolge
[9] `http://lucene.apache.org/core`

unterschiedliche Repräsentation derselben Information) im Graph propagiert werden, um dieses Wissen zur Entdeckung weiterer Korrespondenzen zu nutzen. Diese Verfahren weichen vom klassischen Konzept, Datensätze unabhängig voneinander zu vergleichen und zu klassifizieren, ab und gehen, aufgrund des iterativen Charakters, typischerweise mit einem höheren Berechnungsaufwand einher [110, 207].

Das Entity Resolution-Framework *MOMA* [228] verfolgt einen *Mapping*[10]-basierten Ansatz und verwendet zwei verschiedene Mapping-Typen, um Beziehungen zwischen Datensätzen auszudrücken. Same-Mappings beschreiben Korrespondenzen; Assoziations-Mappings drücken semantische Beziehungen zwischen Datensätzen aus. Die Kombination und Komposition von Mappings erlaubt sowohl die Validierung ermittelter als auch die Entdeckung neuer Korrespondenzen. Für den Vergleich von Datensätzen mit sehr heterogenen Repräsentationen wird die Verwendung eines Neighborhood-Matchers vorgeschlagen, der die Korrespondenz assoziierter Datensätze (z. B. Publikationen) ausnutzt, um zwei heterogene Datensätze (z. B. Konferenzen) miteinander zu verknüpfen.

2.2.4 Klassifikation

Nach Bestimmung potentieller Duplikate und dem Vergleich der Kandidatenpaare muss die Menge der Duplikate bestimmt werden. Die geschieht auf Basis der berechneten Ähnlichkeitsvektoren.

Wahrscheinlichkeitsbasierte Verfahren zur Klassifikation haben ihren Ursprung in der grundsteinlegenden Arbeit von Felligi und Sunter [81]. Diese Methode geht von Ähnlichkeitsvektoren mit binären Ähnlichkeitswerten (0 =unterschiedliche Werte, 1 =Übereinstimmung) aus und basiert auf der Annahme, dass keine Abhängigkeiten zwischen den zur Ähnlichkeitsberechnung herangezogenen Attributen bestehen. Sei m_i die bedingte Wahrscheinlichkeit, dass ein Kandidatenpaar im i-ten Attribut übereinstimmt, wenn bekannt ist, dass die Datensätze tatsächlich Duplikate sind, und u_i die Wahrscheinlichkeit, dass die Werte des i-ten Attributes übereinstimmen, obwohl das Kandidatenpaar kein Duplikat ist. Für jedes Attribut wird anschließend ein Gewicht w_i bestimmt. Bei Übereinstimmung der Attributwerte beträgt $w_i = \log_2(m_i/u_i)$, sonst gilt $w_i = \log_2((1-m_i)/(1-u_i))$. In der Folge werden Schwellwerte angesetzt, um die Menge der Duplikate zu selektieren. Das Hauptproblem dieser Methode ist die Bestimmung der Wahrscheinlichkeiten m_i und u_i; eine Beschreibung möglicher Ansätze findet sich in [111]. Der Basisansatz wurde in mehreren Arbeiten erweitert, die u. a. die Verwendung normalisierter Ähnlichkeitsvektoren, die Einbeziehung der Häufigkeitsverteilung von Attributwerten und eine Berücksichtigung von Abhängigkeiten zwischen den Attributen zum Forschungsgegenstand haben [198, 244, 246, 247].

Schwellwertbasierte Verfahren aggregieren die Komponenten eines Ähnlichkeitsvektors $(s_1, ..., s_n)$, z. B. durch Berechnung einer einfachen oder gewichteten Summe in eine Gesamtähnlichkeit. Die Filterung der Duplikate erfolgt durch Ansetzen eines Schwellwertes für die aggregierte Gesamtähnlichkeit. In [51] wird die Verwendung zweier Schwellwerte diskutiert. Neben den sicheren Duplikaten wird dabei eine Menge wahrscheinlicher Duplikate generiert, die manuell klassifiziert werden müssen.

Regelbasierte Verfahren treffen die Match-Entscheidung anhand sogenannter Matching-Regeln. Eine Regel hat die Form einer konjunktiven oder disjunktiven Normalform und definiert eine Bedingung (z. B. Ähnlichkeit der Publikationstitel > 0.9 *und* im selben Jahr veröffentlicht *und* übereinstimmende Konferenz $\Rightarrow$ *Match*), die von einem Match-Kandidaten erfüllt sein muss, um als Duplikat klassifiziert zu werden (siehe z. B. [55, 100, 104, 158]). Die Aufstellung dieser Regeln erfolgt manuell durch Domänenexperten. Dies ist ein aufwändiger iterativer Prozess, bei dem nach manueller Beurteilung der resultierenden Duplikate die Regeln schrittweise verfeinert werden müssen [51]. In [66, 217] wurde vorgeschlagen, regelbasierte Verfahren um zusätzliche *Constraints* anzureichern. Diese prüfen, unabhängig von der Ähnlichkeitsberechnung, die Einhaltung domänenspezifischer Integritätsbedingungen und dienen der abschließenden Validierung klassifizierter Duplikate. Ein ähnlicher Ansatz ist die Verwendung

[10] *engl.* mapping = Zuordnung

sogenannter *negativer Regeln* zur Eliminierung inkonsistenter *Matches* aus dem Match-Ergebnis [241]. Mit ihnen werden Bedingungen modelliert, die für *kein* Duplikat im finalen Match-Ergebnis erfüllt sein dürfen.

In den vergangenen Jahren wurden viele Ansätze zur Beschleunigung sogenannter **(Set) Similarity Joins** veröffentlicht [12, 97, 166, 209, 212, 215, 237, 249, 250]. Ausgehend von zwei Objektmengen A, B berechnen Similarity Joins alle Paare $(a, b) \in A \times B$ mit $sim(a, b) \geq t$ bzw. $dist(a, b) \leq d$. Sie sind auf ein konkretes Ähnlichkeitsmaß (oder eine Klasse von Ähnlichkeitsmaßen) zugeschnitten und nutzen dessen Eigenschaften, um mithilfe verschiedener Filtertechniken alle Datensatzpaare, für die die Suchbedingung nicht gilt, zu eliminieren. Im Kontext der Entity Resolution können solche Verfahren beispielsweise verwendet werden, um die Menge aller Datensatzpaare (eines Blocks), deren Attributwerte eine bestimmte Mindestähnlichkeit aufweisen, zu bestimmen.

Maschinelle Lernverfahren können ebenfalls zur Klassifikation der Kandidatenpaare verwendet werden. Dabei kommen vorrangig überwachte Lernverfahren zum Einsatz, die auf Basis der Ähnlichkeitsvektoren manuell gelabelter Trainingsdaten ein Klassifikationsmodell generieren, welches anschließend auf die (in derselben Art und Weise ermittelten) Ähnlichkeitsvektoren der ungelabelten Kandidatenpaare angewendet wird. Aus dem Forschungsbereich des Data Minings sind verschiedene Algorithmen zum Lernen von Klassifikatoren bekannt; im Bereich der Entity Resolution sind Entscheidungsbäume (siehe z. B. [100]) und Support Vector Machines (SVM) (siehe [48]) sehr populär (z. B. [28, 53, 76]). Eine vergleichende Untersuchung verschiedener Entity Resolution-Frameworks mit Angabe der jeweils unterstützten Lernverfahren findet sich in [138]. Eine wichtige Fragestellung bei der Verwendung von maschinellen Lernverfahren ist, wie die manuell zu labelnden Trainingsdaten ausgewählt werden. In [137] wurde untersucht, was ein sinnvoller Mindestähnlichkeitsschwellwertes für Trainingsdaten ist und welchen Einfluß der Anteil der Duplikate/Nicht-Duplikate sowie die Anzahl der Trainingsdaten auf die Qualität der erzielten Match-Ergebnisse haben. Ein Vergleich verschiedener Lernverfahren und Methoden zur automatischen Auswahl von Trainingsdaten für unsaubere Webdaten erfolgte in [141]. Da der manuelle Aufwand zur Aufstellung optimaler Matching-Regeln sehr hoch ist, bietet sich die Verwendung maschineller Lernverfahren insbesondere an, wenn zur Ähnlichkeitsberechnung mehrere Attribute und Ähnlichkeitsmaße verwendet wurden. Um die Präzision des Klassifikationsschrittes weiter zu verbessern, wurde in [257] die Kombination der Ergebnisse mehrerer voneinander unabhängiger Lernverfahren untersucht. Mit der gleichen Motivation wurde in [146] die Partitionierung der Kandidatenpaare in Klassen untersucht, welche die Häufigkeitsverteilung der verglichenen Attributwerte widerspiegeln (z. B. “ungewöhnliche, gewöhnliche, und häufig auftretende Personennamen”). Anschließend wird für jede Klasse ein eigenes Modell gelernt und zur Klassifikation verwendet. Dem liegt die Annahme zugrunde, dass eine beinahe Übereinstimmung selten auftretender Attributwerte auf ein Duplikat hindeutet, wohingegen eine hohe Ähnlichkeit häufig auftretender Attributwerte kein sicheres Klassifikationskriterium darstellt.

Active Learning beschreibt eine Vorgehensweise, die ebenfalls auf maschinellem Lernen basiert, jedoch im Vergleich zu den im vorigen Abschnitt beschriebenen Verfahren mit wesentlich weniger Trainingsdaten auskommt. Ziel dieser Verfahren ist die Reduzierung des manuellen Aufwandes zur Selektion und Klassifikation von Trainingsdaten. Verfahren des Aktiven Lernens verwenden eine kleine Menge von Trainingsdaten, um ein initiales Klassifikationsmodell zu generieren, welches in einem iterativen Prozess schrittweise durch Nutzerfeedback verfeinert wird. Das initiale Modell nimmt dazu eine erste Trennung von Duplikaten und Nicht-Duplikaten vor. Anschließend werden dem Nutzer die Kandidatenpaare, die “am schwierigsten zu klassifizieren waren” zur manuellen Klassifikation vorgelegt. Bei Verwendung einer SVM wären dies beispielsweise die Kandidatenpaare, deren Ähnlichkeitsvektorabbilder im höherdimensionalen Raum den geringsten Abstand zur trennenden Hyperebene haben. Im Anschluss daran werden diese Paare zur initialen Trainingsmenge hinzugefügt und das Modell wird angepasst oder neu gelernt. Dieser Prozess wird solange wiederholt bis ein Stoppkriterium, wie z. B. eine Mindestqualität des generierten Modells bezüglich der Testdaten, erreicht ist. Techniken des Aktiven Lernens wurden u. a. im Rahmen der Entity Resolution-Frameworks *ALIAS* [211] und *Active Atlas*

[226, 227] sowie in [13, 58] verwendet. Auch im Bereich der Link Discovery wird Active Learning eingesetzt, um mit geringem manuellem Aufwand qualitativ hochwertige Link Spezifikationen zu erzeugen [184, 185, 186].

2.2.5 Nachverarbeitung

Die **Berechnung der transitive Hüllen und Clustering** sind Nachverarbeitungsschritte zur Korrektur des Match-Ergebnisses. Im Folgenden wird von einer Datenquelle und der klassischen Vorgehensweise, Match-Kandidaten unabhängig voneinander zu vergleichen und zu klassifizieren, ausgegangen. Auf Methoden, die auf einer "Ähnlichkeitsberechnung im Kollektiv" in Kombination mit Clustering-Techniken basieren, wurde bereits im Abschnitt 2.2.4 eingegangen. Einzige Eingabe ist die Menge der klassifizierten Duplikate, die als Kanten eines ungerichteten Graphen interpretiert werden.

Die Berechnung der transitiven Hülle dient zur Beseitigung von Kontradiktionen in der Menge der klassifizierten Duplikate. Die Paare (a, b) und (b, c) im Match-Ergebnis sagen aus, dass sowohl a und b als auch b und c dasselbe Realweltobjekt repräsentieren. Trifft dies tatsächlich zu, so ist wegen der Transitivität der Gleichheitsrelation das Paar (b, c), selbst wenn es nicht nicht im Match-Ergebnis enthalten ist, ebenfalls als Duplikat zu klassifizieren. Die ursprüngliche Einordnung von (b, c) in die *Non-Matches* kann beispielsweise durch den Blocking-Schritt verursacht sein. Da die Klassifikation jedoch auf einer *Ähnlichkeits*berechnung beruht, kann bei den generierten Duplikaten i. Allg. nicht von einer Gleichheit ausgegangen werden. Der transitive Schluss (insbesondere über mehrere Stufen hinweg) kann deswegen Datensätze verbinden, die nicht dasselbe Objekt beschreiben. Dies ist dadurch bedingt, dass nicht alle Ähnlichkeitsmetriken die Dreiecksungleichung erfüllen [176]. Ein möglicher Ansatz zur Behebung dieses Problems ist die Entfernung der Kante mit der niedrigsten Ähnlichkeit in jeder zusammenhängenden Komponente. Dies wird rekursiv fortgesetzt bis eine maximale Pfadlänge unterschritten wird oder die niedrigste Ähnlichkeit einen Mindestwert überschreitet [176].

Eine Clustering-basierte Strategie zur Auflösung großer zusammenhängender Komponenten wurde in [102] vorgeschlagen. Der Algorithmus verwendet Kantengewichte, welche den aggregierten Ähnlichkeiten der verbundenen Datensätze entsprechen. Für jede zusammenhängende Komponente werden dabei mehrere Zentroiden bestimmt. Anschließend wird jeder Knoten dem Zentroiden zugeordnet, zu dem er die geringste Entfernung aufweist. Dazu werden die Kanten einer zusammenhängenden Komponente absteigend nach Ähnlichkeit sortiert und nacheinander verarbeitet. Ein Knoten der ersten verarbeiteten Kante wird als Zentroid eines neuen Clusters bestimmt; der zweite Knoten wird ebenfalls zu diesem Cluster hinzugefügt. Bei der Bearbeitung weiterer Kanten wird geprüft, ob einer der beiden Knoten bereits Zentroid eines Clusters ist. Ist dies der Fall, wird der jeweils andere Knoten diesem Cluster hinzugefügt. Ist keiner der beiden Knoten Bestandteil eines bestehenden Clusters, so wird ein neues zweielementiges Cluster gebildet. Ist lediglich ein Knoten Bestandteil eines Clusters (aber nicht dessen Zentroid), so wird ein neues einelementiges Cluster, bestehend aus dem anderen Knoten, erstellt. Abschließend können Cluster, deren Zentroiden sehr ähnlich sind, verschmolzen werden.

Zusätzliche **Einschränkungen möglicher Korrespondenzen** können sich durch die Charakteristika der Datenquellen ergeben. Enthalten im Zwei-Quellen-Fall beide Datenquellen in sich keine Duplikate, so kann ein Datensatz der einen Quelle höchstens einem Datensatz der anderen Quelle entsprechen. Enthält eine der beiden Datenquellen Duplikate, so kann ein Datensatz der "sauberen" Datenquelle mit mehreren Datensätzen der duplikatbehafteten Datenquelle verbunden werden. Umgekehrt hat jeder Datensatz der "unsauberen" Quelle höchstens einen Partner in der duplikatfreien Datenquelle. Enthalten beide Duplikate, so bestehen keine Kardinalitätsrestriktionen. Da die bisher vorgestellten Klassifikationsstrategien derartige Einschränkungen nicht berücksichtigen, müssen diese Nebenbedingungen in einem Nachverarbeitungsschritt sichergestellt werden. Die Erstellung einer optimalen 1:1-Zuordnung ist äquivalent zur Lösung eines *Maximum Weighted Bipartite Graph Matching*-Problems. Ziel ist, ausgehend von einem Bipartiten Graphen, eine 1:1-Zuordnung mit maximaler Summe der Kanten-

gewichte (Ähnlichlichkeitswerte) zu finden. Für dieses Problem existieren verschiedene Algorithmen, wie z. B. der *Kuhn-Munkres*-Algorithmus [175]. Eine weitere bekannte Methode ist der *Stable Marriage*-Algorithmus [86], der jedoch keine optimale Lösung des Problems liefert, sondern eine "stabile" Zuordnung findet. Eine Zuordnung ist stabil, wenn für beliebige Datensätze a_1, a_2 der Datenquelle A und b_1, b_2 der Datenquelle B *niemals* gilt: $(a_1, b_1) \in M$, $(a_2, b_2) \in M$ aber $sim(a_1, b_2) > sim(a_1, b_1)$ und $sim(a_1, b_2) > sim(a_2, b_2)$, wobei M dem finalen Match-Ergebnis entspricht.

Zur **Evaluation der Qualität des Match-Ergebnisses** ist ein sogenanntes *perfektes Match-Ergebnis* (auch bezeichnet als *Gold Standard*) notwendig. Dieses muss entweder manuell erstellt oder, sofern vorhanden, anhand eindeutiger Datensatzidentifikatoren abgeleitet werden. Vom perfekten Match-Ergebnis wird angenommen, dass alle Duplikate enthalten sind und dass jedes Duplikat korrekt klassifiziert wurde. Bezüglich dieses perfekten Match-Ergebnisses kann die Qualität eines errechneten Match-Ergebnisses mittels verschiedener Maße ausgedrückt werden. Ein klassifiziertes Paar wird dazu in vier verschiedene Klassen eingeordnet. Die Klasse der *True Positives* beinhaltet alle korrekt als *Match* klassifizierten Datensatzpaare, wohingegen die Klasse der *False Positives* alle fälschlicherweise als *Match* klassifizierten Paare umfasst. In der Klasse der *True Negatives* sind alle korrekt als *Non-Match* klassifizierten Paare enthalten; die Klasse der *False Negatives* enthält die fälschlicherweise als *Non-Match* klassifizierten Paare. Die Anzahl der in den jeweiligen Klassen enthaltenen Paare wird mit TP, FP, TN bzw. FN bezeichnet. Die Qualität eines Match-Ergebnisses wird üblicherweise durch die Maße *Precision*$= TP/(TP + FP)$, *Recall*$= TP/(TP + FN)$ und derem harmomischen Mittel, dem *F-Measure*, ausgedrückt. Die Precision drückt aus, wie viele der berechneten Duplikate korrekt klassifiziert wurden; der Recall beschreibt, wie viele der tatsächlichen Duplikate auch gefunden wurden.

2.2.6 Entity Resolution-Frameworks

Es existiert eine Vielzahl frei verfügbarer Entity Resolution- und Link Discovery-Frameworks (z. B. [23, 26, 28, 29, 31, 41, 47, 52, 71, 76, 87, 88, 122, 150, 182, 205, 228, 226, 227, 235, 257]). Die Mehrzahl dieser Systeme sind Forschungsprototypen, die programmiert wurden, um neue Verfahren und Algorithmen zu entwickeln und zu evaluieren. Sie erlauben die Definition und Ausführung von Entity Resolution-Workflows für ein konkretes Entity Resolution-Problem. Dazu werden i. d. R. verschiedene Verfahren und Algorithmen für das Blocking, die Ähnlichkeitsberechnung und die Klassifikation bereitgestellt, auf die der Nutzer zurückgreifen kann. Eine Kurzbeschreibung dreizehn verschiedener Systeme findet sich in [51]. In [138] wurden elf verschiedene Frameworks hinsichtlich der unterstützten Techniken, der bei bestimmten Testfällen erzielten Qualität sowie bezüglich der benötigten Laufzeit verglichen. Um eine systemübergreifende Evaluierung zu ermöglichen, wurde zu diesem Zweck eine Evaluierungsplattform entwickelt, welche die Anbindung bestehender Systeme ermöglicht [139]. Eine Weiterentwicklung wurde für eine umfangreiche Evaluierung unterschiedlicher Systeme und Algorithmen für mehrere Entity Resolution-Probleme verwendet [140]. Eine Besonderheit dieser Studie ist die Berücksichtigung des Aufwandes zum Tuning der verwendeten Parameter und/oder die Anzahl der benötigten Trainingsdaten. Viele in der Literatur vorgeschlagene und evaluierte Verfahren sind durch eine oder mehrere Parameter konfigurierbar; veröffentlicht werden i. d. R. nur die Ergebnisse der "besten" Konfiguration, wobei der dazu betriebene manuelle Aufwand unerwähnt bleibt. Aus diesem Grund wurde in [140] unter Verwendung verschiedener Strategien zur automatischen Parameterkonfiguration allen Systemen der gleiche Konfigurationsaufwand zugestanden. Neben den Forschungsprototypen unterstützen auch zunehmend kommerzielle Systeme, insbesondere im Data Warehouse-Umfeld, die Bestimmung von Duplikaten in den zu integrierenden Datenquellen bzw. die Durchführung sogenannter *Fuzzy-Joins* [42, 43]. Eine detailliertere Diskussion bestehender Systemen, welche eine parallele oder verteilte Ausführung von Entity Resolution-Workflows ermöglichen, erfolgt im Abschnitt 2.4. Im Abschnitt 3.5 wird eine Abgrenzung der in dieser Dissertation vorgestellten Techniken zu den Entity Resolution-Ansätzen mit Parallelverarbeitung vorgenommen.

2.3 Das MapReduce-Programmiermodell

Im Jahr 2004 beschrieben Jeffrey Dean and Sanjay Ghemawat in ihrer grundsteinlegenden Arbeit [59] das MapReduce-Programmiermodell. Gleichzeitig stellten sie ein Framework vor, welches dieses Programmiermodell implementiert und eine automatische Parallelisierung von Berechnungen auf großen Datenmengen in Clusterumgebungen ermöglicht. MapReduce hat seine Ursprünge in der funktionalen Programmierung. MapReduce-Programme werden im Wesentlichen durch zwei Funktionen höherer Ordnung definiert, die vom MapReduce-Framework auf disjunkte Datenpartitionen angewendet werden.

$$map : (key_{in}, value_{in}) \rightarrow list(key_{tmp}, value_{tmp})$$
$$reduce : (key_{tmp}, list(value_{tmp})) \rightarrow list(key_{out}, value_{out})$$

Da die Funktionsaufrufe (weitesgehend) unabhängig voneinander sind und von einer verteilten Speicherung der Einagbedaten ausgegangen wird, kann die Parallelisierung eines beliebigen Programms automatisch erfolgen. Das MapReduce-Framework verbirgt dabei alle Details der verteilten Programmierung, wie Datenpartitionierung, Datenumverteilung, Scheduling[11], Netzwerkkommunikation und Fehlerbehandlung vor dem Nutzer, was die die Entwicklung von verteilten Berechnungen extrem vereinfacht. Darüber hinaus kann ein konkretes MapReduce-Programm unverändert in Clustern beliebiger Größe ausgeführt werden.

MapReduce wurde ursprünglich von *Google Inc.* entwickelt, um die Berechnung einfacher aber extrem datenintensiver Probleme, wie die Generierung eines invertierten Indexes zur Beantwortung von Suchanfragen oder die Generierung des Webgraphens, zu parallelisieren. Die in [59] beschriebene Implementierung ist proprietär und bis heute nicht frei verfügbar. Mit ähnlichen Herausforderungen konfrontiert, wurde ab 2004 im Rahmen der Entwicklung des Web Crawlers *Apache Nutch*[12] an einer Implementierung des MapReduce-Programmiermodells gearbeitet, aus der unter Mitwirkung von *Yahoo Inc.* im Jahr 2008 ein eigenes Top-Level-Projekt der Apache Software Foundation namens *Apache Hadoop* hervorging [242]. Aufgrund der freien Verfügbarkeit und einfachen Einrichtung [222] ist Hadoop die populärste Implementierung des MapReduce-Programmiermodells, die mittlerweile zum de facto-Industriestandard gereift ist [192]. Die Ausführungen im weiteren Verlauf der Arbeit beziehen sich auf die MapReduce-Implementierung Apache Hadoop.

2.3.1 Verteilte Dateisysteme

MapReduce-Frameworks, wie Hadoop, bauen auf einem verteilten Dateisystem auf. Hadoop verwendet das *Hadoop Distributed File System*[13] (siehe z. B. [216, 242]), welches dem *Google File System* [91] nachempfunden ist. Die Designziele des GFSs leiten sich durch die Anforderungen verteilter datenintensiver Anwendungen in Googles Rechenzentren ab. Die Aufgabe eines verteilten Dateisystems ist die Speicherung hunderter Terabytes auf Tausenden von Festplatten in Tausenden von lose gekoppelten Rechnern bestehend aus Standard-Hardware. Wichtige Merkmale sind Fehlertoleranz und Hochverfügbarkeit. Wegen des Verzichts auf teure Spezialhardware, wie RAID-Verbunde, muss die erforderliche Robustheit gegenüber *beständig auftretenden* Software- und Hardware*fehlern* durch das verteilte Dateisystem selber realisiert sein. Verteilte Dateisysteme für MapReduce sind im Gegensatz zu lokalen oder Netzwerkdateisystemen optimiert auf die sequentielle Verarbeitung weniger aber sehr großer Dateien; die Gewährleistung eines hohen Durchsatzes für Lese- und Schreiboperationen ist wichtiger als die Sicherstellung einer geringen Latenz. Weitere charakteristische Merkmale sind der weitesgehende Verzicht auf Dateimodifikationen sowie die fehlende Unterstützung von gleichzeitigen Schreibzugriffen auf dieselbe Datei. Einmal geschriebene Dateien werden nicht mehr geändert, aber in der Folge

[11] *engl.* scheduling = Koordination, Planung
[12] `http://nutch.apache.org/`
[13] `http://hadoop.apache.org/docs/stable1/hdfs_design.html`

mehrfach gelesen. Lesezugriffe folgen vorwiegend einem sequentiellen Zugriffsmuster; parallele Lesezugriffe durch mehrere Anwendungen sind möglich. Dieses Zugriffsmuster wird auch als *write-once, read-many-times pattern* bezeichnet [242].

Verteilte Dateisysteme ähnlich dem GFS folgen einer Master/Slave-Architektur. Die Slave-Knoten (im HDFS bezeichnet als *Datanodes*) speichern Datenblöcke in ihrem lokalen Dateisystem, während der Masterknoten (im HDFS bezeichnet als *Namenode*) den Dateibaum sowie zugehörige Metadaten verwaltet. Dateien werden in Blöcke einer festen Größe, typischerweise zwischen 64 und 512 MB, aufgeteilt. Sinn der vergleichsweise großen Blockgröße ist die Beschleunigung von sequentiellen Lesezugriffen auf große Dateien durch die Einsparung von Positionierungsbewegungen des Schreib-/Lesekopfes von Magnetplattenspeichern. Die Blöcke einer Datei werden zur Gewährleistung der Datensicherheit mehrfach repliziert und einer bestimmten Strategie folgend auf die einzelnen Datanodes aufgeteilt. Der Masterknoten ist für die Verwaltung der Zuordnung von Blöcken zu Dateien verantwortlich. Sowohl Schreib- als auch Lesezugriffe werden über den Masterknoten initiiert; der eigentliche Datentransfer erfolgt jedoch direkt zwischen Client und Datanodes, damit der Masterknoten nicht zum Engpass wird. Die Erstellung von Replikaten beim Schreiben von Daten erfolgt innerhalb des Cluster mittels einer sogenannten *Write Pipeline*. Nach dem Schreiben eines Blocks streamt der entsprechende Datanode die geschriebenen Daten zum nächsten Speicherknoten. Dieser Vorgang wird fortgesetzt bis der erforderliche Replikationsfaktor erreicht ist und eine umgekehrte Kette von Bestätigungsmeldungen und Prüfsummen beim Ausgangsdatanode eintrifft. Diese Vorgehensweise dient der Beschleunigung von Schreibzugriffen. Zum einen ist der Durchsatz innerhalb eines MapReduce-Clusters höher als bei Zugriffen von externen Clients, zum anderen kann der Client auf diese Art und Weise mehrere Blöcke hintereinander in das verteilte Dateisystem schreiben, deren erfolgreiche Replikation asynchron bestätigt wird. Jeder Datanode übermittelt periodisch die Identifikatoren seiner gespeicherten Blöcke an den Namenode. Anhand ausbleibender Benachrichtigungen erkennt der Namenode den Ausfall bzw. die Nichterreichbarkeit von Speicherknoten. Bei Unterreplikation eines Blocks veranlasst der Namenode das Anlegen eines neuen Replikats.

Aus Performanzgründen werden Dateibaum und Metadaten vom Namenode komplett im Hauptspeicher gehalten. Obwohl die Speicherkapazität des HDFSs durch Hinzufügen neuer Datanodes beliebig erweitert werden kann, ist die horizontale Skalierbarkeit durch den Hauptspeicher des Namenodes begrenzt. Da mit steigender Knotenanzahl auch immer mehr *Block Reports* und immer mehr Dateisystemzugriffe zu verarbeiten sind (MapReduce-Slaves sind selber HDFS-Clients, siehe Abschnitt 2.3.2), wird in sehr großen Clustern, bestehend aus mehreren tausend Knoten, der Namenode zunehmend zum Engpass, der den Durchsatz für Lese- und Schreiboperationen begrenzt. Um trotzdem eine echte horizontale Skalierbarkeit zu ermöglichen, wurde *HDFS Federation* entwickelt. Darunter verbirgt sich die Verwendung mehrerer, voneinander völlig unabhängiger Namenodes, die jeweils einen eigenen Namensraum verwalten, und die Speicherung von Blöcken mehrerer Namenodes durch die Datanodes. Ein weiterer Vorteil der Isolation verschiedener Namensräume ist die Möglichkeit, die Speicherkapazität eines großen Clusters unter mehreren Mandanten oder Applikationen mit unterschiedlichen Leistungsanforderungen aufzuteilen. Die Aufteilung des globalen Namespaces auf mehrere Namenodes erfolgt durch die Verwendung Client-seitiger *Mount Tables*, ähnlich der Datei */etc/fstab* in unixartigen Betriebssystemen.

Der Masterknoten ist sowohl im GFS als auch im HDFS ein *Single Point of Failure*, dessen Ausfall die Verfügbarkeit des ganzen Systems beeinflusst. Eine Lösung dieses Problems erfordert zumindest die Sicherung des Dateibaums, der Metadaten sowie eines *Write-Ahead-Logs* auf einem hochverfügbaren Speichermedium. Da aus Performanzgründen die Zuordnung von Blöcken zu Datanodes nicht durch den Namenode persistiert wird, stellen nach einem Failover lange Wartezeiten bis zum Eintreffen der Block Reports aller Clusterknoten ein erhebliches Problem dar. Um ein schnelles und automatisches Failover im Fehler- oder Wartungsfall zu ermöglichen, wurde HDFS um einen sogenannten *Standby Namenode*, der stets eine aktuelle Kopie der relevanten Datenstrukturen des *Active Namenodes* im Hauptspeicher hält, erweitert. Zu diesem Zwecke versenden die Datanodes ihre periodischen Nachrichten an beide

Namenodes. Diese Erweiterung wird als *HDFS High-Availability* bezeichnet. HDFS Federation und HDFS High-Availability können miteinander kombiniert werden.

2.3.2 Das MapReduce-Framework Apache Hadoop

Architektur: Hadoop implementiert das MapReduce-Programmiermodell zur Parallelisierung datenintensiver Berechnungen in Clustern, bestehend aus lose gekoppelten Rechnern. Es besteht aus zwei Schichten, dem verteilten Dateisystem und einer darauf aufbauenden MapReduce-Schicht. Das verteilte Dateisystem übernimmt die Aufgabe der fehlertoleranten Speicherung und Bereitstellung großer Datenmengen, während die MapReduce-Schicht die verteilte Ausführung von Programmen realisiert. Wie das verteilte Dateisystem, so ist auch die MapReduce-Schicht nach dem Master/Slave-Modell aufgebaut. Der Masterknoten (bezeichnet als *Jobtracker*) übernimmt im Wesentlichen die Aufgabe des Schedulings und der Überwachung von MapReduce-Programmen (auch bezeichnet als *MapReduce-Jobs*). Dazu gehört die Zerlegung der Berechnungen in sogenannte Map- und Reduce-Tasks[14] und die Zuweisung von Tasks zu Slave-Knoten (bezeichnet als *Tasktracker*), welche die eigentlichen Berechnungen durchführen. Für jeden Tasktracker ist statisch konfiguriert, wie viele Map- und Reduce-Tasks er maximal gleichzeitig bearbeiten kann; diese Anzahl richtet sich hauptsächlich nach den Hardwaremerkmalen (Größe des Hauptspeichers, Anzahl der Prozessorkerne) des Knotens. Die Gesamtzahl der im Cluster gleichzeitig ausführbaren Map- bzw. Reduce-Tasks wird als Map- bzw. Reduce- Kapazität des Clusters bezeichnet. Die Tasktracker-Knoten unterrichten den Jobtracker periodisch über den Fortgang aktuell laufender Berechnungen. Ist die Bearbeitung eines Tasks abgeschlossen, materialisiert der Tasktracker das Ergebnis im verteilten Dateisystem und wartet auf die Zuteilung eines neuen Tasks durch den Jobtracker. Der Jobtracker führt über den Fortschritt der einzelnen Tasks Buch. Bei Fehlschlag, z. B. aufgrund eines Knotenausfalls oder unverhältnismässig langsamen Fortschritts eines Tasks, erfolgt ein Neuvergabe an einen anderen Clusterknoten.

Zu Illustrationszwecken wurden die Clusterknoten bisher in eine von vier Klassen (Namenode, Datanode, Jobtracker oder Tasktracker) eingeordnet. Tatsächlich bezeichnen diese Klassen jedoch keine Knoten, sondern Prozesse, die auf den Clusterknoten laufen. Der Tasktracker-Prozess startet für jeden zu bearbeitenden Map- und Reduce-Task einen neuen Kindprozess. Dies dient zur Isolation der Tasks untereinander und verhindert, dass ein fehlerhaftes MapReduce-Programm die Funktionalität des Tasktrackers beeinträchtigt. Da die Initialisierung eines neuen Prozesses mit einem Overhead von ungefähr 1s einhergeht, erlaubt Hadoop die Wiederverwendung von Kindprozessen für die Verarbeitung unterschiedlicher Tasks [242]. Im Folgenden werden Kindprozesse zur Bearbeitung von Map- bzw. Reduce Tasks als Map- bzw. Reduce-Prozesse bezeichnet und es wird davon ausgegangen, dass einmal gestartete Kindprozesse beliebig oft wiederverwendet werden können. Der Zusammenhang zwischen Prozessen und Tasks ist in Abbildung 2.3 dargestellt. Im Beispiel gibt es einen Map-Prozess, der zwei Map-Tasks berechnet sowie zwei Reduce-Prozesse, die drei Reduce-Tasks ausführen. Üblicherweise beherbergt ein Slave-Knoten neben einem Tasktracker- auch einen Datanode-Prozess, d. h. er ist sowohl für die die Berechnung von Teilaufgaben als auch für die Speicherung von Blöcken zuständig. Die Eingabe eines MapReduce-Programms umfasst eine oder mehrere Dateien des zugrunde liegenden verteilten Dateisystems, die wiederum aus mehreren Blöcken einer festen Größe bestehen. Für jeden HDFS-Block wird vom Jobtracker ein Map-Task initialisiert. Bei der Zuweisung von Map-Tasks zu Tasktrackern versucht der Jobtracker Datenlokalität zu gewährleisten, d. h. unter den zur Verfügung stehenden Tasktrackern mit freien Kapazitäten wird einer ausgewählt, dessen beherbergender Knoten ein Replikat des HDFS-Blocks im lokalen Dateisystem vorhält. Die Bearbeitung der einzelnen Map-Tasks ist unabhängig voneinander und kann deswegen problemlos parallelisiert werden. Ein MapReduce-Programm wird durch zwei Funktionen, `map` und `reduce`, definiert. Zusätzlich muss der Nutzer die Anzahl r der Reduce-Tasks vorgeben. Ein- und Ausgabedatensätze beider Funktionen werden durch Schlüssel-Wert-Paare des jeweils gleichen Datentyps repräsentiert. Insbesondere gilt, dass alle `map`-Ausgabeschlüssel (-werte) den

[14] *engl.* task = Aufgabe, Auftrag, Arbeit

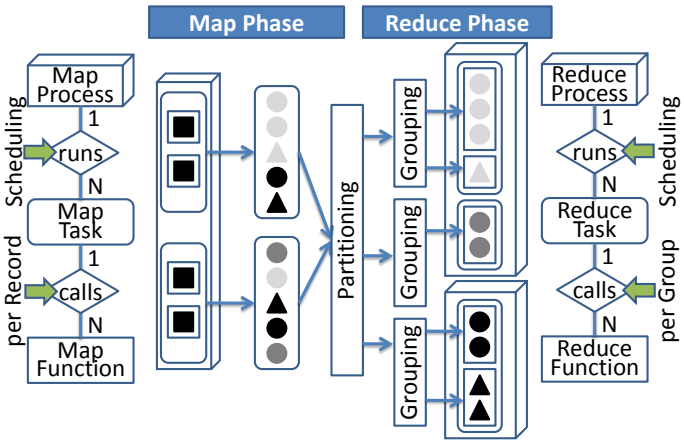


Abbildung 2.3: Schematischer Überblick über die Ausführung eines MapReduce-Programms unter Verwendung zusammengesetzter Schlüssel (die Werte sind nicht gezeigt). Es wird von einem Hadoop-Cluster ausgegangen, indem maximal ein Map- und zwei Reduce-Tasks parallel ausgeführt werden können. Die Eingabe besteht aus zwei HDFS-Blöcken, sodass m=2 Map-Tasks initialisiert werden. Die nutzerdefinierte Anzahl der Reduce-Tasks beträgt r=3. In diesem Beispiel wird von map-Ausgabeschlüsseln ausgegangen, die aus den zwei Komponenten Farbe und Form zusammengesetzt sind. Die Partitionierung und Reduce-Task-Zuweisung der map-Ausgabe erfolgt anhand der Farbe der Schlüssel. Die Reduce-Tasks sortieren und gruppieren die Eingabedaten anhand des vollständigen Schlüssels (also nach Farbe *und* Form) und rufen die reduce-Funktion für jede Gruppe auf.

gleichen Datentyp haben, der mit dem Datentyp der reduce-Eingabeschlüssel (-werte) übereinstimmt [59]. Eine detaillierte Beschreibung des physischen Ausführungsplans eines Hadoop-Programms findet sich in [62, 242].

Während der **Map-Phase** zerlegt jeder Map-Task seine Eingabedatenmenge in Schlüssel-Wert-Paare und wendet eine nutzerdefinierte map-Funktion sukzessiv auf jedes Paar an. Die Ausgabe der map-Funktion besteht aus einer Menge von Schlüssel-Paaren, die zunächst im Hauptspeicher gepuffert werden. Bei Erreichen einer bestimmten Puffergröße verarbeitet ein Hintergrundthread die Schlüssel-Wert-Paare. Durch Anwendung einer Partitionierungsfunktion part auf den Schlüssel der map-Ausgabe-Paare wird der Pufferinhalt in r Partitionen aufgeteilt. Anschließend wird für jede Partition eine Sortierung der Schlüssel-Wert-Paare vorgenommen. Hierzu kommt eine Funktion cmp zum Einsatz, die die Datensätze anhand ihrer Schlüssel vergleicht. Das Ergebnis wird in eine partitionierte und sortierte Datei in das lokale Dateisystem geschrieben. Nach Bearbeitung aller Eingabedaten eines Map-Tasks erfolgt ein abschliessendes Mischen der sortierten Partitionen verschiedener Pufferleerungen. Die Daten der i-ten Partition sind Teil der Eingabedatenmenge des i-ten Reduce-Tasks, d. h. insbesondere dass alle map-Ausgabe-Paare mit dem gleichen Schlüssel an denselben Reduce-Task umverteilt werden. Ein vom Tasktracker gestarteter HTTP-Serverthread stellt die sich im lokalen Dateisystem befindenden finalen Partitionen zur Abholung durch die Reduce-Tasks bereit. Sofern der Programmierer keine Anpassung vornimmt, erfolgt die Zuweisung von map-Ausgabe-Paaren zur Reduce-Tasks mittels der Funktion $part(key) = key.hashCode() \mod r$, was bei gleichverteilten Schlüsseln eine gleichmäßige Aufteilung auf die Reduce-Tasks gewährleistet. Die zur Sortierung verwendete Vergleichsfunktion cmp ergibt sich i. d. R. aus der natürlichen Sortierreihenfolge des verwendeten Schlüsseldatentyps; bei Bedarf kann durch deren Anpassung eine abweichende Sortierung erreicht werden. Im Beispiel von Abbildung 2.3 sind die map-Ausgabeschlüssel aus den Komponenten Farbe und Form zusammengesetzt. Die Zuweisung von map-Ausgabe-Paaren zu den r Reduce-Tasks erfolgt alleinig anhand der Farbkomponente, wohingegen zur Sortierung beide Komponenten herangezogen werden.

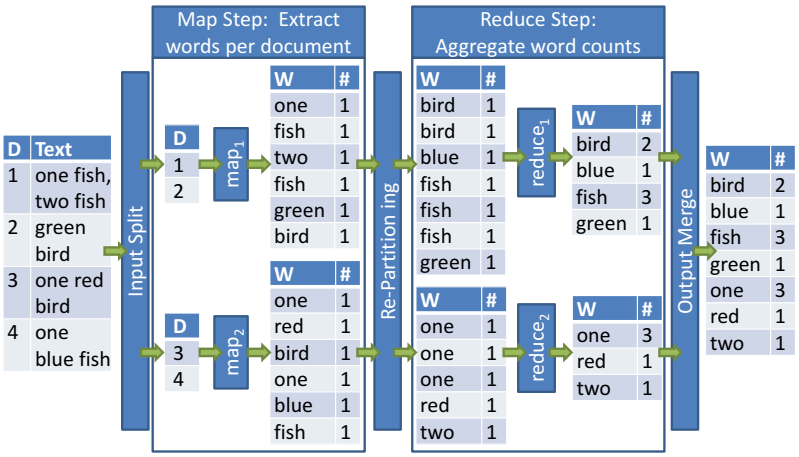


Abbildung 2.4: Exemplarischer Datenfluss für die Ermittlung von Termhäufigkeiten in einer Kollektion von Dokumenten mit MapReduce (adaptierte Version einer Abbildung aus [155]).

Zwischen der Map- und **Reduce-Phase** erfolgt eine Umverteilung der map-Ausgabedaten. Dazu kopiert der Reduce-Task i die i-te Ausgabepartitionen von allen bereits abgeschlossenen Map-Tasks in den Hauptspeicher. Nach Erreichen einer bestimmten Puffergröße werden die sortierten, von verschiedenen Map-Tasks stammenden Ausgabepartitionen im Hauptspeicher gemischt. Das Ergebnis wird als sortierte Datei in einem Job-spezifischen Arbeitsverzeichnis im lokalen Dateisystem abgespeichert. Nach dem Kopieren der Daten des letzten erfolgreich beendeten Map-Tasks werden die einzelnen sortierten Dateien im Arbeitsverzeichnis abschließend in mehreren Runden in eine global sortierte reduce-Eingabedatei gemischt. Die Anwendung einer Gruppierungsfunktion *group* auf die Schlüssel der sortierten Eingabedaten bewirkt eine Zusammenfassung adjazenter Schlüssel-Wert-Paare $(k_1, v_1), ..., (k_n, v_n)$ mit $group(k_1) = ... = group(k_n)$ zu einer Liste $[v_1, ..., v_n]$. Für jede Gruppe erfolgt ein Aufruf der nutzerdefinierten reduce-Funktion: reduce$(k_1, [v_1, ..., v_n])$. Im Beispiel von Abbildung 2.3 wird die reduce-Funktion für alle map-Ausgabe-Paare aufgerufen, deren Schlüssel sowohl in Farbe als auch in Form übereinstimmen. Da die Gesamtgrösse der Listenelemente v_i den einem Reduce-Tasks zur Verfügung stehenden Hauptspeicher überschreiten kann, wird, wie bei einem SQL-Forward-Only Cursor, lediglich ein sequentieller Zugriff auf die Werte mittels einer Iterator<Value>-Schnittstelle unterstützt. MapReduce-Frameworks, wie Apache Hadoop, initialisieren deswegen nur genau ein Objekt zur Repräsentation eines Wertes und setzen dessen Inhalt bei jedem $next()$-Aufruf des Iterators neu. Um beispielsweise alle Werte eines Schlüssel miteinander vergleichen zu können, muss deswegen nach jedem $next()$-Aufruf eine Kopie des aktuellen Wertes angelegt und im Hauptspeicher gepuffert werden. Die Ausgabe der reduce-Funktion besteht aus einer (möglicherweise leeren) Menge von Schlüssel-Paaren, die direkt in das verteilte Dateisystem materialisiert werden. Jeder Reduce-Task legt dazu eine Datei im Ausgabeverzeichnis des MapReduce-Programms an, deren Namen aus dem Index des Reduce-Tasks abgeleitet ist und schreibt alle reduce-Ausgabe-Paare in diese. Das finale Ergebnis eines MapReduce-Programms setzt sich aus der Menge aller reduce-Ausgabedateien zusammen. Zu Optimierungszwecken wird das erste Replikat jedes HDFS-Blockes jeder reduce-Ausgabedatei vom jeweiligen lokalen Datenode gespeichert.

Abbildung 2.4 verdeutlicht die Ausführung eines MapReduce-Programms am Beispiel der verteilten Berechnung von Termhäufigkeiten. Die Eingabedatenmenge besteht aus einer Menge von Dokumenten, die von zwei Map-Tasks bearbeitet werden. Die map-Funktion wird für jedes Dokument aufgerufen und gibt nach der Analyse des Dokumentes für jeden enthaltenen Term t ein Schlüssel-Wert-Paar $(t, freq)$ aus. Diese Paare werden zu r=2 Reduce-Tasks umverteilt, so dass alle Paare mit demselben Schlüssel vom selben Reduce-Task bearbeitet werden. Jeder Reduce-Task sortiert und gruppiert die eingehenden Schlüssel-Wert-Paare anhand des Schlüssels (des Terms) und ruft für jeden Term t die

Funktion reduce$(t, [freq_1, ..., freq_n])$ auf. Innerhalb der reduce-Funktion wird über die Termfrequenzen der einzelnen Dokumente, in denen t vorkommt, iteriert, um die Gesamthäufigkeit des Terms t in der Dokumentkollektion zu ermitteln. Abschließend erfolgt die Ausgabe eines Schlüssel-Wert-Paars $(t, \Sigma_{i=1}^{n} freq_i)$.

2.3.3 Forschungsansätze und Erweiterungen

Die Popularität und freie Verfügbarkeit von Apache Hadoop führte zu dessen Einsatz in verschiedenen Forschungs- und Anwendungsbereichen wie z. B. Data Warehousing [45], Graph-Verarbeitung [54], Information Retrieval [156], Data Mining [192], parallele Datenbanksysteme [194] oder Bioinformatik [213]. Im Zuge der Parallelisierung von Algorithmen unterschiedlicher Domänen wurden zahlreiche Verbesserungen und Erweiterungen des MapReduce-Programmiermodells und der Hadoop-Implementierung vorgeschlagen, implementiert und evaluiert [148]. Dieser Abschnitt gibt einen Überblick über die bestehenden Forschungsansätze.

Indexierung und Spaltenorientierung: MapReduce wurde in der Datenbank-Community kontrovers diskutiert und frühzeitig hinsichtlich einer Eignung als Alternative zu parallelen Datenbanksystemen untersucht [222]. Forschungsschwerpunkte waren die Umsetzung von Operationen der Relationenalgebra [62] sowie die effiziente Berechnung von Join-Operationen für große Datenmengen [32, 189]. Eine Laufzeitvergleich paralleler Datenbanksysteme und Apache Hadoop für verschiedene Anfragetypen erfolgte in [194]. Diese Studie zeigt eine deutliche Überlegenheit spaltenorientierter verteilter Datenbanksysteme gegenüber MapReduce-Implementierungen. Die Autoren führten den Unterschied auf langjährig erprobte Datenbanktechnologien wie Indexierung, Spaltenorientierung, Anfrageverarbeitung auf komprimierten Daten und hochoptimierte verteilte Auswertungspläne zurück. Ein wesentlicher Nachteil beim Einsatz von MapReduce für datenbankähnliche Aufgaben ist das Nichtvorhandensein von Indexen; es wird lediglich das Zugriffsmuster *Full Table Scan* unterstützt.

Um Ergebnisse von Punkt- und Bereichsanfragen frühzeitig eingrenzen zu können, wurde in [62] eine Anreicherung von HDFS-Blöcken um Indexe, die die enthaltenen Daten beschreiben, vorgeschlagen. Durch Anpassung von UDFs[15] des Hadoop-Frameworks wird zur Laufzeit der Index eines jeden Blocks gelesen, um das Enthaltensein von Datensätzen, die ein bestimmtes Selektionskriterium erfüllen, zu prüfen und die Verarbeitung eines Blocks auf ein bestimmten Byte-Intervall zu beschränken. Des Weiteren wurde die Kapselung von Datensätzen verschiedener Datenquellen, die denselben Wert eines bestimmten Attributes aufweisen, innerhalb eines HDFS-Blocks vorgeschlagen. Dieser Ansatz ermöglicht eine Berechnung von Join-Operationen ohne Datenumverteilung und resultiert in einer erheblichen Beschleunigung. Ein wesentlicher Nachteil der Ansätze aus [62] ist die Indexierung der gesamten Datenmenge bezüglich eines einzelnen Attributes. Dieser Nachteil wurde in weiterführenden Arbeiten [63, 64] behoben. Grundidee ist die Indexierung verschiedener Replikate eines HDFS-Blocks bezüglich mehrerer Attribute sowie die Konvertierung von Blöcken in ein binäres spaltenorientiertes Format. Die Indexierung und Umwandlung erfolgt automatisiert während des Uploads von Daten in das verteilte Dateisystem. Auch in anderen Arbeiten konnte eine deutliche Beschleunigung der Ausführung von MapReduce-Programmen durch spaltenorientierte Speicherung von Datensätzen gezeigt werden [82, 103, 157].

Fehlerbehandlung: Im Vergleich zu parallelen Datenbanksystemen unterstützt MapReduce eine feingranularere Fehlerbehandlung [194]. Parallele Datenbanksysteme verzichten zugunsten der Performanz weitesgehend auf die Materialisierung von Zwischenergebnissen innerhalb der Transaktionsgrenzen und verfolgen stattdessen den Ansatz des Pipelinings[16] von Zwischenergebnissen. Deswegen muss

[15] *engl.* UDF = User Definied Function = nutzerdefinierte Funktion zur Anpassung oder Erweiterung der Funktionalität eines Software-Systems

[16] Pipelining = Prinzip der überlappenden gleichzeitigen Ausführung verschiedener Operatoren eines Operatorbaums mit frühzeitiger Weitergabe von Datensätzen eines Erzeugeroperators zum Verbraucheroperator. Dies verhindert, dass die Datenverarbeitung im Verbraucheroperator verzögert wird, bis die komplette Eingabedatenmenge zur Verfügung steht (vgl. [201]).

im Falle eines Knotenausfalls während der Bearbeitung einer zeitintensiven Anfrage i. d. R. die komplette Anfrage neugestartet werden [194]. Fällt hingegen während der Ausführung eines MapReduce-Programms ein Knoten aus, so vergibt der Jobtracker alle deswegen fehlgeschlagenen Tasks an andere Tasktracker, die diese erneut berechnen. Dies wird durch die replizierte Speicherung von Zwischen- und Teilergebnissen im verteilten Dateisystem ermöglicht. Reduce-Tasks, die vom betroffenen Tasktracker erfolgreich beendet wurden, müssen i. d. R. nicht neu berechnet werden, da das Ergebnis im verteilten Dateisystem repliziert ist. Erfolgreich beendete Map-Tasks müssen hingegen neu berechnet werden, da Map-Ausgabe-Paare lediglich im lokalen Dateisystem des Tasktrackers gespeichert sind. Um die vollständigen Neuberechnung erfolgreich beendeter Map-Tasks zu vermeiden, wurden in [199, 200] verschiedene Optimierungen der Fehlerbehandlung von Apache Hadoop untersucht, die in Experimenten wesentliche Laufzeiteinsparungen zeigten. Der Ansatz basiert auf einer statischen Zuordnung von abzuarbeitenden Reduce-Tasks zu Tasktrackern, die durch den Jobtracker *zu Beginn der Ausführung* eines MapReduce-Programms festgelegt wird. Nach der Leerung des map-Ausgabepuffers durch einen Hintergrundthread wird jede resultierende Partition $0 \leq i < r$ zu dem Tasktracker kopiert, der für die Ausführung des i-ten Reduce-Tasks zuständig ist. Die periodische Übertragung von Zwischenergebnissen zu den zuständigen Tasktrackern stellt sicher, dass bei Ausfall eines Knotens der Großteil der Ausgabe erfolgreich beendeter Map-Tasks nicht verloren ist, sondern sich im lokalen Dateisystem anderer Knoten befindet. Lediglich sogenannte lokale Partitionen, also Eingabepartitionen für Reduce-Tasks, die vom ausgefallenen Knoten selber bearbeitet werden sollten, sind verloren. Um diese Partitionen wiederherstellen zu können, vermerkt jeder Map-Task für jede *lokale* Partition, für welche map-Eingabeschlüssel (üblicherweise eine Kombination aus Eingabedatei und Byte-Offset) die map-Funktion Schlüssel-Wert-Paare generiert, die in diese Partition fallen. Vor Ende der Berechnung eines Map-Tasks wird diese resultierende Datenstruktur an andere Knoten repliziert. Im Falle eines Knotenausfalls, sind die auf diese Weise gesammelten Informationen noch verfügbar und erlauben eine gezielte Neuberechnung einer verlorenen lokalen Partitionen i, indem die map-Funktion lediglich auf die für Partition i vermerkten map-Eingabeschlüssel angewendet wird.

Deklarative Anfragesprachen: Der zeitaufwändige Prozess der Erstellung eines MapReduce-Programms erfolgt auf einem sehr niedrigen Abstraktionslevel und führt i. d. R. zu schwer wartbaren Programmen, die auf einen konkreten Einsatzfall zugeschnitten sind und kaum Wiederverwendung erlauben [229]. Aus diesem Grunde wurden verschiedene Erweiterungen zur Unterstützung deklarativer Anfragesprachen, ähnlich der *Structured Query Language* für relationale Datenbanken, vorgestellt. Das frei verfügbare *Apache Hive*[17] erweitert Hadoop um Data Warehouse-Funktionalitäten. Kernfeature ist die Unterstützung einer SQL-ähnlichen Anfrageschnittstelle [229, 230]. In *HiveQL*, einem SQL-Dialekt, formulierte Nutzeranfragen werden automatisiert optimiert und in eine Menge von MapReduce-Jobs übersetzt. *Apache Pig*[18] ist ein System zur Verarbeitung großer, semi-strukturierter Daten auf Basis von Hadoop [90]. Es ist Googles *Sawzall*-System [196] nachempfunden und beinhaltet eine prozedurale Skriptsprache *Pig Latin* [191], welche durch eine Verkettung verschiedener Datentransformationsschritte eine einfache Definition komplexer MapReduce-Workflows erlaubt. Ähnlich zu HiveQL-Statements werden Pig Latin-Skripte analysiert, optimiert und in eine Menge von MapReduce-Programmen übersetzt, die in einen Hadoop-Cluster zu Ausführung gebracht werden können.

Da MapReduce auf die fehlertolerante Parallelverarbeitung großer Datenmengen ausgelegt ist, nicht aber auf geringe Latenzzeiten optimiert ist, sind die bisher vorgestellten Ansätze zur interaktiven Analyse großer Datenmengen ungeeignet. *Cloudera Impala*[19] verwendet die Metadatenverwaltung von Hive, um beispielsweise HDFS-(Unter)Verzeichnisse auf Tabellen(partitionen) abzubilden. SQL-Anfragen werden jedoch nicht in MapReduce-Jobs übersetzt, sondern mittels eines eignen Mechanismus zur verteilten Anfragebearbeitung umgesetzt. Ähnlich zur verteilten Anfragebearbeitung in parallelen Datenbanksystemen erstellt ein Koordinatorknoten für jede eingehende Anfrage einen verteilten Ausführungsplan, dessen einzelne Operatoren parallel durch verschiedene Impala-Prozesse bearbeitet wer-

[17] `http://hive.apache.org/`
[18] `http://pig.apache.org/`
[19] `http://impala.io/`

den. Üblicherweise wird dazu auf jedem Clusterknoten, zusätzlich zum Datanode-Prozess, ein Impala-Prozess gestartet, sodass bei der Anfragebearbeitung Datenlokalitätsaspekte berücksichtigt werden können. Eine weitere Gemeinsamkeit mit parallelen Datenbanksystemen ist die Teilaggregation und frühzeitige Weitergabe von Zwischenergebnissen nach dem Pipelining-Prinzip. Darüber hinaus wird in der zum Zeitpunkt der Erstellung der Arbeit aktuellen Version von Impala auf jegliche Materialisierung von Zwischenergebnissen verzichtet; nach dem initialen Lesen der Daten erfolgt die weitere Anfragebearbeitung vollständig im Hauptspeicher. Dies bedeutet eine Einschränkung der Verwendbarkeit von Impala (z. B. bei der Berechnung des Verbundes zweier Tabellen, dessen Ergebnis die aggregierte Hauptspeichergröße des Clusters überschreitet), soll aber in zukünftigen Versionen behoben werden. Impala unterstützt zudem mit *Parquet* ein spaltenorientiertes binäres Datenformat, das dem Datenmodell entspricht, welches in Googles Googles *Dremel*-System [164] verwendet wird. Google Dremel ist ebenfalls ein hochskalierbares System zur verteilten Bearbeitung von Leseanfragen auf *geschachtelten* Daten, dass auf einer hierarchischen Anfrageausführung basiert. Jeder Knoten schreibt eine eingehende Anfrage in mehrere Teilanfragen um und aggregiert die resultierenden Teilergebnisse. Die Bearbeitung der Teilanfragen erfolgt durch Knoten der nächstniedrigeren Hierarchieebene; der Externspeicherzugriff erfolgt ausschließlich durch Blattknoten. Auch hierfür existiert wiederum eine entsprechende Open Source-Implementierung namens *Apache Drill*[20], welche eine interaktive Datenanalyse für verschiedene Speicherplattformen, wie z. B. HDFS, *Apache Hive*[21], *MongoDB*[22] oder *Apache Cassandra*[23] zum Ziel hat.

HadoopDB[24] versucht, die Fehlertoleranz- und Skalierbarkeitseigenschaften mit den Performanzvorteilen klassischer Datenbanksysteme zu verbinden [1, 2]. Grundidee ist die zusätzliche Installation unabhängiger Datenbanksysteme (z. B. *MySQL* oder *PostgreSQL*) auf den einzelnen Knoten eines Hadoop-Clusters und die Verwendung einer Erweiterung von Hive zur Umwandlung von SQL-Anfragen in MapReduce-Programme. HadoopDB zielt darauf ab, einen Großteil der Arbeit in effizienter Art und Weise von den einzelnen Datenbanksystemen erledigen zu lassen und Teilergebnisse mittels Hadoop, das als Koordinations- und Kommunikationsschicht fungiert, zusammenzuführen und zu kombinieren. Auch kommerzielle verteilte Datenbanksysteme unterstützen in zunehmendem Maße MapReduce-Funktionalität, z. B. in Form von UDFs, die entsprechend dem MapReduce-Programmiermodell ausgeführt werden [85, 224, 252].

Erweiterung des Programmiermodells: Verschiedene Anwendungen erfordern für große Datenmengen ebenfalls eine verteilte Bearbeitung in Rechner-Clustern, eignen sich jedoch nur bedingt für eine Parallelisierung mit MapReduce. Typische Beispiele sind iterative Algorithmen, z. B. in den Bereichen Graph-Verarbeitung, Data Mining und Maschinelles Lernen, welche dieselbe Berechnung wiederholt ausführen bis ein Terminierungskriterium erreicht ist [255]. Eine MapReduce-Implementierung erfordert die wiederholte Ausführung eines MapReduce-Programms, was ein Ausschreiben der Ausgabedaten von Iteration i in das verteilte Dateisystem sowie ein Einlesen derselben Daten in der Iteration $i+1$ bedingt. Daten, die über die verschiedenen Iterationen hinweg unveränderlich sind, werden zudem unnötigerweise zwischen der Map- und Reduce-Phase im lokalen Dateisystem materialisiert und umverteilt. HaLoop [35, 36] und Twister [75] sind Erweiterungen, welche diesen Overhead durch das Caching von (unveränderlichen) Ein- und Ausgabedaten verringern und Hadoops Task-Scheduling Mechanismus anpassen, um die Ausführung von Tasks, die dieselbe Datenmenge bearbeiten, auf dem selben Knoten zu gewährleisten. In [49] wurde MapReduce um eine *Merge-Phase* zum Mischen der reduce-Ausgabedaten verschiedener MapReduce-Jobs erweitert, um die Berechnung von Operationen mit mehreren Eingabedatenquellen, wie z. B. Kreuzprodukt, Differenz, Schnittmengenbestimmung und Joins, zu erleichtern.

Nephele [238] ist ebenfalls ein Framework zur Ausführung verteilter Berechnungen in Clusterumgebun-

[20] `http://incubator.apache.org/drill`
[21] `http://hive.apache.org`
[22] `http://www.mongodb.org`
[23] `http://cassandra.apache.org/`
[24] `http://db.cs.yale.edu/hadoopdb/hadoopdb.html`

gen. Im Gegensatz zu MapReduce unterstützt es das Pipelining und Caching von Datensätzen zwischen verschiedenen Phasen der Berechnung. Programme werden mit *PACTs*, einer Verallgemeinerung des MapReduce-Programmiermodells, ausgedrückt [6, 8, 19]. Neben map und reduce werden weitere Funktionen höherer Ordnung unterstützt, die zudem nativ mehrere Datenquellen als Eingabe unterstützen. Nephele/PACT erlaubt im Vergleich zur starren Hintereinanderausführung von map und reduce eine wesentlich flexiblere Komposition verschiedener Datenverarbeitungsschritte und erleichtert die Umsetzung komplexer Workflows erheblich. Des Weiteren existiert eine Abstraktionsschicht namens *Sopremo*, welche die Definition von Workflows durch eine mächtige Skriptsprache mit der Bezeichnung *Meteor* ermöglicht. Meteor-Skripte werden automatisch in PACT-Programme übersetzt. Der Verbund aus Nephele, PACT und Sopremo wird als *Stratosphere*-Plattform [7] bezeichnet, die im Rahmen des Open Source-Projekts *Apache Flink*[25] weiterentwickelt wird. Ein ähnliches Framework namens *Dryad* wurde von Microsoft entwickelt [117]. Analog zu PACTS werden Berechnungen durch DryadLINQ-Programme ausgedrückt [118, 254], die automatisiert in einen physischen Ausführungsplan übersetzt werden.

Mit Beginn der Alpha-Version 0.23 wurde das Hadoop-Framework massiven Änderungen unterzogen. Ein wesentliches Designziel des resultierenden *NextGen MapReduce* (auch bezeichnet als *MapReduce 2.0*, *MRv2* oder *YARN*) war eine Reduzierung der Workload des Jobtrackers durch Dezentralisierung von Verwaltungsaufgaben. Dessen Aufgabenspektrum umfasste zuvor sowohl die Ressourcenverwaltung als auch die Ausführung und Überwachung von MapReduce-Jobs. In großen Hadoop-Clustern wurde beobachtet, dass der Jobtracker ab einer bestimmten Anzahl parallel ablaufender Tasks[26] zum Engpass wird und damit die horizontale Skalierbarkeit begrenzt. Mit YARN wurde die klassische Aufteilung der MapReduce-Schicht in Jobtracker und Tasktracker abgelöst und durch einen globalen *Ressource Manager*, einen anwendungsspezifischen *Application Master* sowie *Node Manager*-Prozessen auf jedem Clusterknoten ersetzt [233]. Aufgabe des Ressource Managers ist die Zuteilung von Cluster-Ressourcen (RAM, CPU, Bandbreite, Plattenspeicher) in Form abstrakter Container an Anwendungen sowie der Start eines Application Masters für jede entgegengenommene Anwendung. Ein Application Master ist ein leichtgewichtiger Prozess auf einem beliebigen Clusterknoten und fungiert als "Mini-Jobtracker" für die Ausführung einer konkreten Anwendung (z. B. eines MapReduce-Jobs). Er kommuniziert mit dem Ressource Manager, initiiert die Allokation zugewiesener Container auf verschiedenen Knoten und stößt die Ausführung von Teilaufgaben an. Die Ausführung von Teilaufgaben auf den einzelnen Knoten erfolgt durch die Node Manager, die ihrerseits den Application Master periodisch über den Berechnungsfortschritt unterrichten. YARN bildet oberhalb des HDFS eine Schicht zur Verwaltung abstrakter Cluster-Ressourcen und ermöglicht somit die Unterstützung *verschiedener Programmiermodelle* in Apache Hadoop. Neben MapReduce wurden zum Zeitpunkt der Erstellung der Arbeit auch *Apache Spark*[27], *Open MPI*[28], *Apache Hama*[29] und *Apache Giraph*[30], welches Googles Graphverarbeitungs-Framework *Pregel* [159] nachempfunden ist, unterstützt.

Lastbalancierung: Das MapReduce-Programmiermodell weist eine große Anfälligkeit für Datenungleichverteilung (auch bezeichnet als *Data Skew* oder *Partition Skew*) auf was zu erheblichen Lastbalancierungsproblemen führen kann. Das Data Skew-Problem wurde erstmals von den berühmten Datenbankforschern David DeWitt and Michael Stonebraker in einem vielbeachteten Blog-Post[31] thematisiert und in [154] formalisiert. Im Bereich paralleler Datenbanksysteme ist Lastbalancierung ein wohlbekanntes und gut erforschtes Problem (siehe z. B. [61, 161]). *Speculative Excecution* ist das einzige

[25] `http://flink.incubator.apache.org`

[26] In einem Vortrag des Spotify Mitarbeiters Adam Kawa anlässlich eines Treffens der Warsaw Hadoop User Group wird eine Grenze von ca. 40.000 parallel ausgeführten Tasks genannt (vgl. `http://de.slideshare.net/AdamKawa/apache-hadoop-yarn-simply-explained`).

[27] `http://spark.incubator.apache.org/`

[28] `http://www.open-mpi.org/`

[29] `http://hama.apache.org/`

[30] `http://giraph.apache.org/`

[31] Die Originalquelle ist zum Zeitpunkt der Erstellung dieser Arbeit nicht mehr verfügbar. Es existieren jedoch Archivversionen und Wiederveröffentlichungen, z. B. `http://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html` und `http://craig-henderson.blogspot.de/2009/11/dewitt-and-stonebrakers-mapreduce-major.html`

Instrument von MapReduce-Frameworks zur Lastbalancierung. Darunter versteht man die redundante Ausführung von "langsamen" Map- bzw. Reduce-Tasks gegen Ende der Map- bzw. Reduce-Phase an verschiedenen Knoten. Sobald eine "Kopie" eines redundant ausgeführten Tasks erfolgreich bearbeitet wurde, werden noch laufende Bearbeitungen weiterer Kopien vom Jobtracker gestoppt. Die Ausführungszeit eines MapReduce-Jobs wird durch den Zeitpunkt der erfolgreichen Beendigung des letzten Reduce-Tasks bestimmt. Zudem startet die Reduce-Phase erst nach der Beendigung des letzten Map-Tasks. Der Speculative Excecution-Mechanismus dient v. a. zur Beschleunigung von MapReduce-Programmen in *heterogenen* Rechner-Clustern, da dadurch vermieden wird, dass die Laufzeit eines MapReduce-Jobs vom langsamsten Clusterknoten bestimmt wird. Zudem wird dadurch eine Robustheit gegenüber (temporär) stark ausgelasteten oder fehlkonfigurierten Knoten erreicht. Ist eine deutlich variierende Ausführungszeit verschiedener Tasks jedoch nicht durch die Ressourcen der bearbeitenden Knoten, sondern durch eine unterschiedliche Größe der Eingabedatenmenge (*Data Skew*, *Partition Skew*) oder aber durch einen unterschiedlichen Zeitbedarf zur Verarbeitung gleichgroßer Eingabedatenmengen (*Computational Skew*) bedingt, ist das Instrument der redundanten Ausführung von Tasks wirkungslos [154]. Die Behandlung von Computational Skew-Effekten wurde in [142] untersucht. Insgesamt fand diese Problematik in der Forschung wenig Beachtung, da datenintensive MapReduce-Anwendungen sehr viel anfälliger für Datenungleichverteilung sind [10].

Bei der standardmäßigen Verwendung von (hinreichend großen) Dateien des verteilten Dateisystems als Eingabe für ein MapReduce-Programm sind Partition Skew-Effekte in der Map-Phase in den meisten Fällen ebenfalls kein Problem, da die Eingabedatenmenge in HDFS-Blöcke fester Größe (mit Ausnahme eines kleineren letzten Blocks) aufgeteilt wird und somit jeder Map-Tasks eine Eingabepartition derselben Größe bearbeitet. Ein MapReduce-inhärentes Problem ist jedoch die Tatsache, dass alle map-Ausgabe-Schlüssel-Wert-Paare, die denselben Schlüsselwert aufweisen, demselben Reduce-Task zur Bearbeitung zugewiesen werden. Da die Verteilung der map-Ausgabeschlüssel i. Allg. vorab unbekannt ist und die Partitionierung der map-Ausgabedaten vom Programmierer statisch *vor* Beginn der Berechnung festgelegt sein muss, sind die Möglichkeiten zur Lastbalancierung in der Reduce-Phase begrenzt. Insbesondere für reduce-Funktionen mit einer nicht-linearen Rechenkomplexität (z. B. paarweise Ähnlichkeitsberechnung aller reduce-Eingabe-Werte) können moderate Unterschiede in den Größen der von den Reduce-Tasks zu verarbeitenden Datenmengen zu signifikanten Laufzeitunterschieden bei der Bearbeitung führen. Allgemein gilt, dass durch die Zeit, die zur Bearbeitung der größten Schlüsselgruppe (also dem map-Ausgabe-Schlüssel mit den meisten Werten) benötigt wird, eine untere Schranke für die Gesamtausführungszeit eines MapReduce-Jobs gegeben ist. Dies bedeutet, dass durch eine Erhöhung des Parallelitätsgrades keine weitere Beschleunigung der Bearbeitung erzielt werden kann. Bei Nutzung von Infrastructure as a Service-Diensten führt dies zu einer deutlichen Senkung der Kosteneffizienz, da akquirierte, aber nicht ausgelastete Ressourcen, unnötige Kosten verursachen.

Ein erster Ansatz zur Behandlung des Partition Skews wurde in [115] vorgeschlagen. Der *LEEN*-Ansatz verlangt eine Modifizierung des Hadoop-Frameworks hinsichtlich verschiedener Punkte. Zunächst wird die Map-Phase von der Reduce-Phase entkoppelt, d.h. die erste Welle der Reduce-Tasks beginnt erst mit dem Kopieren der für sie relevanten Map-Ausgabepartitionen *nachdem* der letzte Map-Task erfolgreich beendet wurde[32]. Des Weiteren zählen die Map-Tasks die Auftretenshäufigkeit eines jeden map-Ausgabe-Schlüssels. Nach einer Aggregation der Teilstatistiken durch den Jobtracker am Ende der Map-Phase erfolgt eine Zuweisung von map-Ausgabe-Schlüsseln zu Reduce-Tasks. Dabei kommt ein heuristischer Algorithmus zum Einsatz, der zwei Ziele verfolgt. Zum einen soll die Varianz der von den einzelnen Reduce-Tasks zu verarbeitenden Eingabedaten minimiert werden. Um die zwischen der Map- und Reduce-Phase umzuverteilende Datenmenge zu reduzieren, soll zum anderen die Bearbeitung eines map-Ausgabe-Schlüssels durch einen Reduce-Task erfolgen, dessen beherbergender Knoten einen möglichst großen Anteil der Eingabedaten im lokalen Dateisystem vorhält. Da der Ansatz nur mit sehr großem Aufwand in das Hadoop zu integrieren ist, entwickelten die Autoren ein "Execution Framework", welches die Map- und Reduce-Phase eines MapReduce-Frameworks "nachahmt" und den

[32] In der Standard-Implementierung beginnen die Reduce-Tasks der ersten Welle bereits mit dem Kopieren von Map-Ausgabedaten nachdem ein vorgegebener Mindestprozentsatz an Map-Tasks erfolgreich beendet wurde.

Lastbalancierungsalgorithmus zwischen den beiden Phasen ausführt.

Ein ähnlicher Ansatz wurde in [95] vorgeschlagen. Im Vergleich zu [115] werden zudem unterschiedliche Rechenkomplexitätsklassen der reduce-Funktion berücksichtigt. Dazu geht der Ansatz von einer Funktion $w(|C(k)|, ||C(k)||)$ aus, welche die Zeit, die zur Ausführung von reduce$(k, [v_1, ..., v_n])$ benötigt wird, abschätzt. Dabei beschreibt $|C(k)|$ die Anzahl der Werte v_i mit dem Schlüssel k und $||C(k)||$ deren aggregierte Größe in Byte. Die geschätzte Zeit wird als Workload der Schlüsselgruppe k bezeichnet. Ähnlich zu [115] werden von den Map-Tasks Statistiken über Schlüsselverteilung generiert, die *nach* dem Ende der Map-Phase beim Jobtracker aggregiert und zur Lastbalancierung genutzt werden. Für jede map-Ausgabepartition i werden dabei folgende Kennzahlen aggregiert: die Anzahl $t(i)$ der Werte, die Gesamtgröße aller Werte in Bytes sowie die Anzahl $c(i)$ der Schlüssel in Partition i. Während die $t(i)$ und $s(i)$ durch Summierung der von den einzelnen Map-Tasks ermittelten Teilsummen bestimmt werden kann, muss $c(i)$ approximativ ermittelt werden. Darüber hinaus argumentieren die Autoren, dass für große Datenmengen eine exakte Ermittlung dieser Statistiken nicht praktikabel ist und schlagen stattdessen eine verteilte Generierung approximierter Statistiken vor [96]. Ausgehend von den aggregierten Statistiken wird für jede Partition i die durchschnittliche Anzahl $\overline{t}(i) = t(i)/c(i)$ der Werte eines enthaltenen Schlüssels sowie die durchschnittliche Gesamtgröße $\overline{s}(i) = s(i)/c(i)$ der Werte eines Schlüssels berechnet. Damit ergibt sich eine durchschnittliche Workload $W(i) = c(i) \cdot w(\overline{t}(i), \overline{s}(i))$ für die Bearbeitung der Partition i. In der Folge werden zwei Lastbalancierungsstrategien vorgeschlagen. Ähnlich dem Konzept des *Virtual Processor Partitionings* bei der Join-Berechnung in verteilten Datenbanksystemen (vgl. [61]) wird die Anzahl der Partitionen p als ganzzahliges Vielfaches der Anzahl der Reduce-Tasks gesetzt. Anschließend werden die Partitionen absteigend nach deren durchschnittlicher Workload sortiert und den r Reduce-Tasks mittels einer *Greedy*-Heuristik zugeordnet. Zusätzlich schlagen die Autoren eine Strategie zur Aufteilung von Partitionen, deren Workload einen Durchschnittswert überschreitet, in Fragmente vor, die verschiedenen Reduce-Tasks zugewiesen werden können. Da eine Umsetzung der vorgeschlagenen Strategien massive Eingriffe in eine MapReduce-Implementierung, wie Apache Hadoop, nach sich ziehen würden (Anzahl Map-Ausgabepartitionen $\neq$ r, verteilte Bestimmung approximierter Statistiken, zentrale Lastbalancierung, modifizierte Umverteilung von map-Ausgabedaten) wurden diese ebenfalls *nicht* in Hadoop implementiert. Die Evaluation erfolgte stattdessen auf Basis einer Simulation.

Der *SkewTune*-Ansatz [143] verfolgt die Strategie der *Late Skew Detection*. Dahinter verbirgt sich der Ansatz, eine Skew-Behandlung in der Map- oder Reduce-Phase eines MapReduce-Jobs durchzuführen wenn abzusehen ist, dass die Bearbeitung eines einzelnen Tasks wesentlich länger dauert als die Bearbeitung der übrigen Tasks, es keine ausstehenden Tasks mehr gibt und es freie Cluster-Ressourcen (Map- bzw. Reduce-Prozesse) gibt. Zu einem Zeitpunkt wird nur ein Task behandelt. Dazu wird der Task mit der größten erwarteten verbleibenden Rechenzeit t_{remain} (ermittelt aus dem Verhältnis der bisher bearbeiteten Datenmenge und der dafür benötigten Zeit) ausgewählt. Ausgehend von einem fixen Scheduling Overhead ω (ca. 30s) erfolgt eine Skew-Behandlung nur, wenn gilt: $t_{remain}/2 > \omega$. Dahinter verbirgt sich die Annahme, dass mindestens ein Map- bzw. Reduce-Prozess frei ist (also aktuell keinen Task bearbeitet) und die verbleibende Datenmenge des zu behandelnden Tasks gleichmäßig unter dem freien und dem Prozess, der den Task aktuell bearbeitet, aufgeteilt wird, sodass die verbleibende Ausführungszeit im Idealfall $t_{remain}/2$ beträgt. Nach erfolgter Identifizierung eines zu behandelnden *Straggler*[33]-Tasks wird dessen Bearbeitung gestoppt. Anschließend erfolgt ein (paralleles) Scannen der verbleibenden Eingabedatenmenge, um Statistiken über die verbleibende Eingabedatenmenge zu sammeln. Basierend auf diesen Statistiken wird die verbleibende Datenmenge auf die Map- bzw. Reduce-Prozesse aufgeteilt, die momentan frei sind *oder* bald *frei werden*. Dazu wird die verbleibende Datenmenge in adjazente Intervalle mit annähernd gleicher Größe aufgeteilt. Im Falle eines zu behandelnden Map-Tasks entspricht ein Intervall einem Fragment der Eingabedatenmenge. Im Falle eines Reduce-Tasks umfasst ein Intervall mehrere adjazente Schlüssel-Wert-Paare, wobei garantiert ist, dass alle Werte v_i eines map-Ausgabe-Schlüssels k demselben Intervall zugewiesen werden. Die Anzahl

[33] *engl.* straggler = Nachzügler

der Intervalle wird wiederum als ein ganzzahliges Vielfaches der Anzahl der freien Map- bzw. Reduce-Prozesse gesetzt. In der Folge werden die gebildeten Intervalle mittels eines heuristischen Algorithmus mit linearer Zeitkomplexität zu den freien und bald frei werdenden Map- bzw. Reduce-Prozessen zugewiesen. Eingabe dieses Algorithmus ist die nach verbleibender Ausführungszeit aufsteigend sortierte Liste der Map- bzw. Reduce-Prozesse sowie die Liste der Intervalle inkl. der erwarteten Ausführungszeiten. Unter der Annahme einer perfekten Aufteilung der verbleibenden Arbeit auf die optimale Anzahl von n Map- bzw. Reduce-Prozessen berechnet der Algorithmus zunächst die verbleibende Gesamtausführungszeit opt. Anschließend werden dem ersten Map- bzw. Reduce-Prozess solange Intervalle in sequentieller Reihenfolge zugewiesen bis die geschätzte dafür benötigte Bearbeitungszeit den Wert opt überschreitet. Dieser Prozess wird für die nächsten (in erwarteter Terminierungsreihenfolge) Map- bzw. Reduce-Tasks wiederholt bis alle Intervalle zugewiesen sind. Da der SkewTune-Ansatz potentiell die Eingabedatenmenge eines beliebigen Map- oder Reduce-Tasks zur Bearbeitung durch mehrere Map- bzw. Reduce-Prozesse aufteilen kann, ist dessen Einsetzbarkeit auf zustandslose MapReduce-Programme beschränkt, bei denen die Ausgabe eines Aufrufs der der `map`- und `reduce`-Funktion alleinig von den aktuellen Eingabedaten und *nicht* von der Eingabe früherer Funktionsaufrufe abhängt.

ClusterJoin ist ein Framework zur MapReduce-basierten Berechnung von Similarity Joins für beliebige Metriken [212]. Die Berechnung aller Punktpaare, deren Abstand unterhalb eines Schwellwertes liegt, erfolgt in drei Phasen. Im ersten Schritt wird mittels zufälligem Sampling eine Menge A sogenannter *Anchor Points* ermittelt, welche im späteren Verlauf die Rolle von Clusterzentroiden einnehmen, um die die Eingabedatensätze angeordnet werden. Jeder Anchor Point $C \in A$ formt ein Cluster, das alle Punkte umfasst, für die C derjenige Anchor Point mit dem kleinsten Abstand ist. Die Autoren argumentieren, dass durch das Sampling die vorliegende Datenverteilung abgebildet wird, da mehr Punkte aus dichten Regionen und wenige Punkte aus dünn besiedelten Regionen selektiert werden. Parallel zur Bestimmung von A wird eine Menge sogenannter *Query Points* zufällig gesamplet. Jeder Query Point Q wird dem Anchor Point mit dem geringsten Abstand zugeordnet. Da es nicht ausreichend ist, lediglich Punkte des gleichen Clusters zu vergleichen, muss Q zu weiteren Clustern zugeordnet werden, sofern in diesen ein Punkt P mit $\delta(Q,P) \leq \theta$ enthalten sein *kann*. Zur effizienten Bestimmung dieser Cluster schlagen die Autoren verschiedene allgemeine und metrikspezifische Filter vor. Die Anzahl der Query Points pro Anchor Point dient (in Kombination mit der Wahrscheinlichkeit, mit der ein Punkt der Eingabedatenquelle als Query Point selektiert wurde) zur Abschätzung der Größe eines jeden Clusters. Da die Parallelverarbeitung von Clustern unterschiedlicher Größe die Skalierbarkeit einschränkt, werden Cluster oberhalb eines vorgegebenen Größenschwellwertes mittels eines zweidimensionalen Hashverfahrens [189] in Subcluster unterteilt. Dabei wird das Cluster in k^2 Zellen unterteilt und jeder enthaltene Punkt auf k Zellen abgebildet. Für zwei Punkte des Ausgangsclusters ist garantiert, dass sie genau eine gemeinsame Zelle haben. In der zweiten Phase des Workflows wird, analog zur Bearbeitung der Query Points, jeder Punkt der Eingabedatenquelle zum Cluster mit dem nächsten Anchor Point und möglicherweise zu weiteren (Sub-) Clustern zugeordnet. Die paarweise Abstandsberechnung innerhalb der auf diese Weise definierten Cluster erfolgt unabhängig voneinander in der dritten Workflow-Phase.

Tabelle 2.2 vergleicht die existierenden Lastbalancierungsansätze bezüglich verschiedener Kriterien. Es fällt auf, dass die Mehrzahl der Ansätze nicht in die de-facto Standardimplementierung des MapReduce-Programmiermodells, Apache Hadoop, integriert wurden und somit *nicht* für die breite Community einsetzbar sind. Um dies zu gewährleisten, wären weitreichende Eingriffe in den Quellcode des Hadoop-Frameworks notwendig, die bei zukünftigen Weiterentwicklungen mit hohem Aufwand gewartet werden müssten. Die ersten drei Verfahren sind generische Ansätze für beliebige MapReduce-Programme, bei denen die Zeit zur Bearbeitung der größten `reduce`-Eingabe-Schlüsselgruppe eine unterste Schranke der mit einem beliebigen Parallelitätsgrad erreichbaren Gesamtausführungszeit ist. Der in [212] verfolgte Ansatz zur Lastbalancierung ist konzeptionell sehr ähnlich zu den in Dedoop verwendeten Techniken. Hierbei wird ausgenutzt, dass die Distanz zweier Datensätze unabhängig von anderen Datensätzen erfolgen kann. Dies erlaubt eine beliebig feingranulare Aufteilung der Datensatzpaare eines

[34] `http://research.microsoft.com/en-us/events/fs2011/helland_cosmos_big_data_and_big_challenges.pdf`

Verfahren	Anwendbarkeit/ Einsatzzweck	Ansatz	Zeitpunkt/Art der Last-balancierung	Aufteilung einzelner Schlüssel-gruppen	Zur Implementierung/Evaluation verwendete Plattform
LEEN [115], 2010	Beliebige MapReduce-Programme	Heuristische Zuweisung von Schlüsselgruppen zu Reduce-Tasks	Zwischen Map- u. Reduce-Phase/ statisch	✗	Eigenentwickeltes Execution Framework
[95, 96], 2011/12	Beliebige MapReduce-Programme	Heuristische Zuweisung von Schlüsselgruppen zu Reduce-Tasks	Zwischen Map- u. Reduce-Phase/ statisch	✗	Simulation
Skew Tune [143], 2012	Beliebige zustandslose MapReduce-Programme	Stopp des langsamsten Tasks und Aufteilung verbleibender Eingabe-daten auf freie Prozesse	Zur Laufzeit/ dynamisch	✗	Apache Hadoop
Cluster Join [212], 2014	Similarity Joins	2d-Hashing für Aufteilung von Paaren großer Cluster auf mehrere Tasks	Zwischen Map-Reduce-Jobs/ statisch	✓	Scope [40]/ Cosmos[34]/ Dryad [117]

Tabelle 2.2: Übersicht der vorgestellten Lastbalancierungsverfahren zur Behandlung des Partition Skews bei der Ausführung von MapReduce-Programmen.

Clusters (einer Schlüsselgruppe) auf mehrere Reduce-Tasks[35]. Der wesentliche Unterschied besteht in der Art und Weise, wie die Datenverteilung ermittelt (Sampling vs. exakte Bestimmung) und die Aufteilung großer Cluster (Blöcke) vorgenommen wird (Hashing vs. Aufteilung in gleichgroße Subcluster).

Weitere Forschungsarbeiten: Neben den vorgestellten MapReduce-Erweiterungen existiert eine Vielzahl weiterer Forschungsarbeiten, z. B. zu den Themen Energieeffizienz [80, 153, 145], automatisches Parameter Tuning [107, 109], Code Analyse und Optimierung [38, 106, 120], Datenallokation in heterogenen Clustern [11, 251], Fortschrittsberechnung und Laufzeitvorhersage [174, 256], Task Scheduling [187, 197] und *Cluster Sizing* (kosteneffiziente Bestimmung einer optimalen Clustergröße sowie sinnvoller Instanztypen bei Nutzung von Infrastructure as a Service-Diensten) [108].

2.4 Parallelisierung von Entity Resolution-Workflows

Entity Resolution ist seit vielen Jahren ein aktives Forschungsgebiet. Es existiert eine Vielzahl von Ansätzen zur Lösung unterschiedlicher Teilprobleme des Entity Resolution-Prozesses. Trotz der Verwendung von Blocking-Verfahren zur Reduzierung der Kandidatenpaare beansprucht die Duplikaterkennung aufgrund der inhärenten quadratischen Rechenkomplexität der paarweisen Ähnlichkeitsberechnung bereits für mittelgroße Datenquellen mit ca. 100.000 Datensätzen eine Rechenzeit von mehreren Stunden [140]. Um akzeptable Antwortzeiten bei der Deduplizierung großer Datenquellen mit Millionen von Datensätzen gewährleisten zu können, ist eine Parallelverarbeitung auf mehreren Prozessoren und/oder Rechnern unabdingbar. Im verbleibenden Teil dieses Abschnittes wird ein Überblick über bestehende Ansätze zur Parallelisierung von Entity Resolution-Workflows und paarweisen Ähnlichkeitsberechnungen gegeben. Eine Abgrenzung der in dieser Arbeit vorgestellten Techniken zu den existierenden Ansätzen erfolgt nach einer Vorstellung des Dedoop-Frameworks im Abschnitt 3.5.

2.4.1 Entity Resolution-Systeme mit Parallelverarbeitung

Febrl war eines der ersten Systeme, welches eine Parallelverarbeitung von Entity Resolution-Workflows unterstützte. In [52] beschreiben die Autoren erste Resultate für die Parallelisierung auf einem Mehr-

[35] Ein direkter Vergleich solch spezialisierter Verfahren mit generischen Lastbalancierungsverfahren ist somit nicht möglich.

prozessorsystem unter Verwendung des *Message Passing Interface*-Standards. Die Evaluierung zeigt insbesondere, dass der Teilschritt der paarweisen Ähnlichkeitsberechnung und Klassifizierung ungefähr 95% der Gesamtausführungszeit ausmacht und deswegen im Vergleich zum Blocking-Schritt besonders von der Parallelisierung profitiert. Nähere Details zur Parallelisierung werden jedoch nicht beschrieben.

Das *D-Swoosh*-Verfahren [22] betrachtet die Parallelisierung des *Swoosh*-Ansatzes [23] über mehrere Prozessoren und Knoten. Der Ansatz berücksichtigt nicht nur die Erkennung sondern auch die Fusion von Datensätzen, die dasselbe Realweltobjekt beschreiben. Die Autoren argumentieren, dass durch Verschmelzung entstandene Datensätze im Vergleich zu den Ausgangsdatensätzen mehr Informationen in sich tragen und deswegen zur Identifizierung weiterer Duplikate führen können, die durch den paarweisen Vergleich der Ausgangsdatensätze allein nicht gefunden werden können. Die Eingabe des Algorithmus besteht aus einer Funktion *Match*, die für zwei Datensätze entscheidet, ob sie dasselbe Realweltobjekt beschreiben sowie einer Funktion *merge*, die zwei Duplikate zu einem neuen Datensatz fusioniert. Mittels einer *scope*-Funktion wird jeder Eingabedatensatz initial mehreren Prozessoren zugewiesen und an diese gesendet. Dabei ist gewährleistet, dass jedes Datensatzpaar mindestens einen "gemeinsamen Prozessor hat". Die Autoren diskutieren mehrere dieser Zuweisungsverfahren. Eine Möglichkeit besteht darin, jeden Datensatz an die Mehrheit der zur Verfügung stehenden Prozessoren zu senden. In der Folge vergleicht jeder Prozessor die ihm zugewiesenen Datensatzpaare. Da zwei Datensätze mehrere "gemeinsame" Prozessoren haben können, besteht die Möglichkeit redundanter Paarvergleiche. Um dies zu vermeiden, überprüft jeder Prozessor mittels einer *responsible*-Funktion, ob er für den Vergleich zweier Datensätze zuständig ist. Nach der Fusion zweier Duplikate r_i und r_j wird der neu generierte Datensatz $r_{i,j}$ an alle Prozessoren aus $scope(r_{i,j})$ geschickt. Des Weiteren werden alle Prozessoren aus $scope(r_i)$ bzw. $scope(r_j)$ instruiert, ihre Kopien der Ausgangsdatensätze r_i bzw. r_j zu löschen. Ein Prozessor gleicht einen neu eingegangenen (fusionierten) Datensatz mit allen ihm zugewiesenen Datensätzen ab, für die bisher kein Duplikat gefunden wurde. Dieses Verfahren wird iterativ fortgesetzt bis jeder Prozessor alle Paare, für die er zuständig ist, verglichen hat und alle versendeten Nachrichten empfangen und bearbeitet wurden. Neben der bisher diskutieren Parallelisierung des kartesischen Produktes unterstützt *D-Swoosh* prinzipiell auch Blocking-Verfahren. Nachteile des generischen Verfahrens sind die exzessive Replikation der Eingabedaten sowie der sehr hohe Kommunikationsaufwand zur Propagierung von Datensatzfusionierungen. Beim Zusammenführen zweier Datensätze ergibt sich zudem das Problem der automatischen Behebung von Widersprüchen (siehe z. B. [33] und [69]). Der Algorithmus wurde mittels einer verteilten Java-Anwendung implementiert und in einem Cluster bestehend aus 15 Single-Core Rechnern evaluiert. Für 15 Knoten konnte bei der Evaluierung des Kartesischen Produktes lediglich ein Speedup von $5-6$ erreicht werden. Bei der Verwendung von Blocking-Verfahren beobachteten die Autoren zudem starke, durch variierende Blockgrößen verursachte, Lastbalancierungsprobleme. In [160] erfolgte eine Betrachtung verschiedener Möglichkeiten zur Parallelisierung des *Swoosh*-Ansatzes (mit Blocking) unter Verwendung des MapReduce- *und* des Pregel-Programmiermodells. Die vorgestellten Verfahren wurden jedoch nicht evaluiert.

In [219] wurde mit *Parallel Linkage* ebenfalls ein Ansatz vorgestellt, der gefundene Duplikate zu einem neuen Datensatz fusioniert und diesen in einem iterativen Prozess mit anderen Datensätzen der Ausgangsdatenquelle vergleicht. Die Autoren beschränken sich auf die Evaluierung des kartesischen Produktes zweier Eingabedatenquellen. Zusätzlich wird gegebenenfalls die "Sauberkeit" einer oder beider Datenquellen (also die Tatsache, dass eine Datenquelle in sich duplikatfrei ist) ausgenutzt, um die zu verrichtende Arbeit zu begrenzen. Der Ansatz verfolgt eine Replikation der kleineren Datenquelle zu jedem Prozessor. Die größere Eingabedatenquelle wird in mehrere Partitionen aufgeteilt, die den zur Verfügung stehenden Prozessoren zugewiesen werden. Jeder Prozessor wertet in der Folge das Kartesische Produkt der kleineren Datenmenge und der ihm zugewiesenen Partition der größeren Datenmenge aus. Anschließend erfolgt eine Propagierung fusionierter Datensätze zu anderen Prozessoren, um weitere Duplikate finden zu können. Dieser Schritt wird wiederholt bis keine neuen Duplikate mehr erkannt werden. Durch die Fusion von Datensätzen wird die einem Prozessor zugewiesene Datenmenge verringert. Diese Tatsache wird bei der Neuzuweisung fusionierter Datensätze berücksichtigt, um eine gleichmäßige Aufteilung der Arbeit auf die einzelnen Prozessoren zu gewährleisten. Die Evaluati-

on erfolgte für eine sehr kleine Datenquelle von 5.000 Datensätzen unter Verwendung einer verteilten MATLAB Umgebung mit 8 Prozessoren. Da die Evaluation des Kartesischen Produktes, im Gegensatz zur Parallelverarbeitung unterschiedlich großer Blöcke, keine Anfälligkeit gegenüber Datenungleichverteilung aufweist, konnten sehr gute Speedup-Werte von $7,5-8$ erzielt werden.

Eine erste Vorarbeit zu den in dieser Dissertation vorgestellten Techniken zur Parallelisierung von Entity Resolution-Workflows wurde 2010 in [126] vorgestellt. Der Ansatz basiert auf einer serviceorientierten verteilten Infrastruktur des *GOMMA*-Systems [125], die gewisse Ähnlichkeiten mit der Master-Slave-Architektur eines MapReduce-Clusters aufweist. Ein *Workflow Service* generiert auf Basis der Blocking-Konfiguration sogenannte *Match-Tasks*. Ein Match-Task ist eine logische Beschreibung einer Teilmenge der Ausgangsdaten, deren Ähnlichkeit zu berechnen ist. Zur Match-Task-Generierung kommuniziert der Workflow Service mit einen *Data Service*, welcher Zugang zu den Eingabedaten bietet und für die Speicherung von Teilergebnissen zuständig ist. Der Workflow Service verwaltet eine Liste zu bearbeitender Match-Tasks, die *Match Services* mit freien Kapazitäten zur Bearbeitung zugewiesen werden. In Abhängigkeit der Hardwareeigenschaften des beherbergenden Rechners kann ein Match Service mehrere Match-Tasks parallel bearbeiten. Dazu verwaltet er eine Menge von *Match Threads*, die jeweils die Eingabepartitionen der ihnen zugewiesenen Match-Tasks vom Data Service lesen. Um die zwischen Match Service und Data Service transferierten Datenmenge zu verringern, verwaltet jeder Match Service einen sogenannten *Partition Cache*, der eine feste Anzahl an (zuletzt verwendeten) Eingabepartitionen vorhält. Nach erfolgreicher Bearbeitung eines Match-Tasks informiert der Match Service den Workflow Service über den Erfolg und teilt ihm zusätzlich mit, welche Partitionen im Cache vorliegen. Der Workflow Service versucht in der Folge Datenlokalität zu gewährleisten, indem er diesem Match Service einen Match-Task zuweist, der eine der gecachten Partitionen umfasst. Der Ansatz unterstützt sowohl die parallele Auswertung des Kartesischen Produktes als auch die parallele Bearbeitung von Blöcken für eine oder zwei Eingabedatenquellen. Um Robustheit gegenüber Datenqualitätsproblemen zu gewährleisten wird im Gegensatz zu der in anderen Ansätzen verfolgten Zuweisung eines Datensatzes zu mehreren Blöcken die Definition eines (großen) sogenannten *Miscellaneous Blocks* verfolgt, der alle Datensätze mit fehlenden Attributwerten umfasst. Die Bearbeitung unterschiedlich großer Blöcke kann aufgrund der quadratischen Komplexität der paarweisen Ähnlichkeitsberechnung stark variierende Ausführungszeiten verursachen, was zu erheblichen Lastbalancierungsproblemen führt. Aus diesem Grund werden in einem *Partition Tuning*-Schritt große Blöcke in Subblöcke aufgeteilt, wobei für jedes Subblockpaar ein Match-Task generiert wird. Analog dazu werden mehrere kleinere Blöcke in einem einzelnen Match-Task zusammengefasst, um den Gesamtoverhead des Schedulings und Inter-Service-Kommunikation zu verringern. Zur Evaluation wurde eine ca. 114.000 Produktangebote umfassende Datenquelle mit verschiedenen Entity Resolution-Strategien dedupliziert. Die Experimente wurden auf einem lokalen Cluster bestehend aus vier Quad-Core-Rechnern ausgeführt. Auf jedem der Clusterknoten wurde ein Match Service gestartet, der vier Match Threads verwaltete, insgesamt standen 16 Prozessorkerne für die Ähnlichkeitsberechnung zur Verfügung. Für das beste Setup konnte ein nahezu perfekter Speedup von knapp 16 erreicht werden. Die Berücksichtigung der Datenlokalität bewirkte für eine Cache-Größe von 16 Partitionen eine Laufzeitverbesserung von ca. 15%. Die optimalen Blockgrößenschwellwerte wurden experimentell bestimmt, eine dynamische Bestimmung dieser Größen wurde nicht betrachtet. Der Schwachpunkt der verwendeten Infrastruktur ist der zentrale Data Service, der einerseits sämtliche Eingabedaten an die Match Services transferieren und andererseits die Resultate der Berechnungen speichern muss. Für Cluster bestehend aus hunderten von Knoten ist abzusehen, dass der den Data Service beherbergende Knoten zum Engpass wird und die horizontale Skalierbarkeit einschränkt. Diese Tatsache motivierte u. a. die in dieser Dissertation betrachtete Parallelisierung von Entity Resolution-Workflows mit MapReduce.

2.4.2 Entity Resolution mit MapReduce

Die Verwendung von MapReduce im Entity Resolution-Kontext wurde erstmals in [234] untersucht. Die Autoren betrachten die Bestimmung aller Paare $(a, b) \in A \times B$ mit $sim(a, b) \geq t$ zweier Tokenmengen A, B und präsentieren eine MapReduce-Implementierung des sequentiellen *PPJoin+*-Algorithmus [250]. Der Ansatz ist in drei Phasen unterteilt. Die erste Phase besteht aus zwei MapReduce-Jobs, die sequentiell ausgeführt werden. Im ersten Job wird der Join-Attributwert eines jeden Eingabedatensatzes tokenisiert. Für alle Token wird anschließend bestimmt, wie oft sie insgesamt in den Join-Attributwerten der Eingabedatensätze vorkommen. Der zweite MapReduce-Job konvertiert das Ergebnis des ersten Jobs in eine nach Auftretenshäufigkeit absteigend sortierte Liste aller Tokens. In der zweiten Phase, bestehend aus einem einzelnen MapReduce-Job, erfolgt die eigentliche Bestimmung aller Datensatzpaare, deren, aus den Join-Attributwerten abgeleiteten, Tokenmengen eine Mindestähnlichkeit von t aufweisen. Der sequentielle PPJoin+-Algorithmus verwendet verschiedene Filtertechniken zur effizienten Bestimmung aller Paare, die dem Ähnlichkeitskriterium genügen. Sei x die nach Auftretenshäufigkeit aufsteigend sortierte Tokenliste eines beliebigen Datensatzes. Die ersten $p = |x| - \lceil t \cdot |x| \rceil + 1$ Token dieser Tokenliste werden als Menge der Präfixtoken des Datensatzes bezeichnet. Damit zwei Datensätze dem Ähnlichkeitskriterium genügen, *müssen* sie in mindestens einem Präfixtoken übereinstimmen (vgl. [250]). Die Sortierreihenfolge der Token bewirkt, dass diese Überprüfung auf Basis der am seltensten in der Datenquelle vorkommenden Token vorgenommen wird, was die Wahrscheinlichkeit erhöht, dass zwei Datensätze bereits in diesem Schritt aus der Menge der Kandidatenpaare eliminiert werden. Die MapReduce-Implementierung nutzt diese notwendige Bedingung als Blocking-Kriterium. Für jeden Datensatz wird in der Map-Phase die Menge der Präfixtoken bestimmt. Für jedes der p Präfixtoken eines Datensatzes gibt die map-Funktion ein $(token, record)$-Paar aus, konzeptionell wird der Datensatz somit p Blöcken zugewiesen. In der Folge werden alle Datensätze, die ein bestimmtes Präfixtoken gemein haben, gruppiert und der reduce-Funktion übergeben. Innerhalb der reduce-Funktion wird der sequentielle PPJoin+-Algorithmus berechnet. Die Ausgabe besteht aus Paaren von Datensatzidentifikatoren. In einer aus zwei MapReduce-Jobs bestehenden optionalen dritten Phase können diese um alle Attribute der entsprechenden Ausgangsdatensätze angereichert werden. Die Evaluation erfolgt in einem Cluster bestehend aus 10 Knoten mit je einem Quad-Core-Prozessor für (künstlich vergrößerte) Datenquellen mit bis zu 30 Millionen Datensätzen. Die Autoren untersuchten u. a. den relativen Speedup für verschiedene Variationen ihres Ansatzes. Die Ergebnisse zeigen bereits für 10 Knoten suboptimale Speedup-Werte, was erneut durch die variierende Größe der reduce-Eingabepartitionen, also der Anzahl der Datensätze, die ein bestimmtes Präfixtoken gemein haben, verursacht wird. Aufgrund der sehr effizienten Arbeitsweise des sequentiellen PPJoin+-Algorithmus, der durch verschiedene Filtertechniken selber einen starke Einschränkung der Kandidatenmenge vornimmt, sind die beobachteten Lastbalancierungsprobleme weniger gravierend als z. B. beim D-Swoosh-Ansatz. Eine Schwäche des ursprünglichen Ansatzes ist die redundante Betrachtung von Datensätzen, die mehr als ein gemeinsames Präfixtoken haben, in verschiedenen Reduce-Tasks. Dies kann zu Duplikaten im Ergebnis der zweiten Phase führen, die in der dritten Phase des Verfahrens entfernt werden müssen. Darüber hinaus existiert eine Vielzahl weiterer Ansätze zur MapReduce-Parallelisierung von Similarity Joins [166, 209, 212, 215]. Auf [212] wurde bereits im Abschnitt 2.3.3 eingegangen. Der in [215] verfolgte Ansatz wird kurz im Abschnitt 6.2.1 thematisiert.

In [236] wird ein System namens *MapDupReducer* zur Bestimmung von ähnlichen Webdokumenten vorgestellt. Ein möglicher Anwendungsfall für dieses System ist die automatische Bestimmung von Dokumenten mit ähnlichem Inhalt, z. B. beim Browsing von News-Artikeln. Da dies für Datenmengen in der Größenordnung mehrerer hundert Terabytes aufgrund der begrenzten Rechen- und Speicherkapazität eines einzelnen Rechners eine verteilte Berechnung in Rechnerclustern erfordert, realisierten die Autoren das System auf Basis von MapReduce. In vier sequentiell ausgeführten MapReduce-Jobs werden, ähnlich zu [234], verschiedene Filterschritte (Präfix- und Positionsfilter) des PPJoin+-Algorithmus implementiert. Eine weitere eingesetzte Filtertechnik sind Signatur-Filter, die die "Fingerabdrücke" von Dokumenten vergleichen, um die Menge der Kandidatenpaare weiter einzuschränken (vgl. [136]). Ab-

schließend wird in einem Nachverabeitungsschritt die Dokumentenähnlichkeit aller nach Anwendung der Filterschritte verbleibenden Dokumentpaare berechnet, um die finale Menge an (Beinahe-) Duplikaten zu bestimmen. Die beschrieben Schritte können beispielsweise von Suchmaschinenbetreibern *vor* der weiteren Bearbeitung gecrawlter Dokumente durchgeführt werden, um den Aufwand zur Indexierung und Speicherung von Webdokumenten mit gleichem Inhalt zu verringern. Eine Evaluation des Systems wurde von den Autoren nicht präsentiert.

Der *MD-Approach* ist ein Ansatz zur MapReduce-basierten Parallelisierung von Entity Resolution-Workflows basierend auf einem zweistufigen Standard-Blocking [25]. Der erste Blocking-Schritt ist sehr grobgranular und hat v. a. das Ziel, möglichst keine tatsächlichen Duplikate zu eliminieren. Als Resultat entstehen sehr wahrscheinlich mehrere große Blöcke, deren parallele Bearbeitung zu Lastbalancierungsproblemen führen kann. Diese werden anschließend mittels eines feineren Blocking-Kriteriums in kleinere Subblöcke unterteilt. Die Autoren stellen einen entsprechenden Workflow bestehend aus 3 MapReduce-Jobs vor. Im ersten Job wird in der Map-Phase für jeden Datensatz eine Menge von Blockschlüsseln (abgeleitet von den Werten verschiedener Attributen) gebildet. Die Zuweisung eines Datensatzes zu mehreren Blöcken dient der Robustheit gegenüber fehlenden oder fehlerhaften Werten. Für jeden Blockschlüssel *b* eines Datensatzes wird durch die map-Funktion eine (b, record)-Paar ausgegeben. Alle Datensätze eines Blocks, dessen Größe unterhalb eines vorgegebenen Schwellwertes liegt, werden in einem Aufruf der reduce-Funktion des ersten MapReduce-Jobs miteinander verglichen. Alle Datensätze eines großen Blocks werden in einem zweiten MapReduce-Job in kleinere Subblöcke aufgeteilt, deren jeweilige Datensätze analog zum ersten Job in der Reduce-Phase miteinander verglichen werden. Die Aufteilung in Subblöcke kann beispielsweise durch Verkettung von Blockschlüsseln des ersten Blocking-Schritts mit den (feingranularen) Blockschlüsseln des zweiten Schritts erfolgen. Eine Besonderheit bei der Generierung von Subblöcken ist das Zusammenfassen der Datensätze von Blöcken, deren in Sortierreihenfolge benachbarte Blockschlüssel einen vorgegebenen Ähnlichkeitsschwellwert überschreiten. Für eine korrekte Implementierung dieser Idee müsste jeder Map-Task (Schreib-) Zugriff auf eine globale Hash-Tabelle haben, was mit einer skalierbaren MapReduce-Implementierung wie Apache Hadoop nicht möglich ist. Die Autoren verwenden daher das *Phoenix*-Framework [206], welches eine MapReduce-Implementierung für Multi-Core- und Mehrprozessorsysteme mit gemeinsamen (Haupt-) Speicher bereitstellt. Die Evaluation erfolgte deswegen auch lediglich auf einem einzelnen Rechner mit vier Prozessorkernen. Es wurden bis zu 4 Millionen synthetisch generierte Datensätze zur Evaluation herangezogen. Insgesamt konnte ein Speedup von ungefähr 3.5 erreicht werden. Die Autoren vergleichen zudem ihren Ansatz mit der einer Reimplementierung des Ansatzes aus [234] für das Phoenix-Framework hinsichtlich der resultierenden Match-Qualität und der benötigten Laufzeit. Die Aussagekraft dieses Vergleiches ist jedoch aufgrund der unterschiedlichen Arbeitsweise (und damit der Filterung der Kandidatenpaare) des Standard Blockings und des PPJoin+-Verfahrens fragwürdig. Dies gilt insbesondere, da in den verglichenen Implementierungen *unterschiedliche* Ähnlichkeitsmetriken und Schwellwerte verwendet wurden. Da die Menge der nach dem Blocking-Schritt verbliebenen Kandidatenpaare unterschiedlich ist und die Bestimmung von Ähnlichkeiten auf eine verschiedene Art und Weise erfolgt, ist weder ein Vergleich der resultierenden Qualität noch der Laufzeit sinnvoll. Darüber hinaus ist die Vorgehensweise, einen Entity Resolution-Workflow an die Probleme, die *durch* die Parallelisierung entstehen, anzupassen, kritisch zu betrachten. Die Autoren präsentieren keine Strategie zur Lösung der Lastbalancierungsprobleme, die bei der Bearbeitung eines gegebenen Entity Resolution-Workflows entstehen, sondern versuchen die Probleme durch Wahl der Blocking-Kriterien zur Reduktion der Kandidatenmenge zu vermeiden. Ein weiterer Kritikpunkt des Ansatzes ist die Vorgabe eines Schwellwertes zur Definition “großer” Blöcke. Da die Blockschlüsselverteilung i. Allg. vorab unbekannt ist, wäre eine dynamische Ermittlung dieser Schranke in Abhängigkeit von der Gesamtzahl der nach dem Blocking-Schritt verbleibenden Kandidatenpaare und des verwendeten Parallelitätsgrades sinnvoll. Darüber hinaus werden auch in diesem Ansatz Datensatzpaare mehrfach (bezüglich verschiedener Blockschlüssel) verglichen, was zu Duplikaten im finalen Ergebnis führen kann.

Dynamic Record Blocking ist eine Verallgemeinerung des zuletzt betrachteten zweistufigen Standard Blockings auf *n* Stufen [163]. Zur Deduplizierung von Personendatensätzen, die aus verschiedenen

heterogenen Quellen stammen, verwenden die Autoren eine Liste von *Top Level Attributen* wie z. B. Sozialversicherungsnummer und Name, deren Werte zur Definition von *Top Level-Blockschlüsseln* verwendet werden. Für alle Blöcke, deren Größe einen vorgegebenen Schwellwert übersteigt, werden schrittweise weitere Attribute, wie z. B. Geburtsjahr und Wohnort herangezogen, um die Blöcke in kleinere Subblöcke zu unterteilen. Dabei werden (wie in [25]) die Blockschlüssel verschiedener Stufen verkettet. Dieses Verfahren wird rekursiv fortgesetzt bis alle Subblöcke unterhalb des Größenschwellwertes liegen. Durch das Blocking-Schema wird ein Baum definiert, dessen Knoten einen (Sub-) Block repräsentieren. Die Bezeichnung eines jeden Knotens entspricht der Verkettung der Blockschlüssel auf dem Pfad von der Wurzel zum Knoten. Alle Kinder eines Knoten verfeinern das Blocking-Kriterium des Vaterknotens und definieren einen Subblock. Die Blöcke, die als Eingabe für die paarweise Ähnlichkeitsberechnung dienen, sind durch die Blätter des Baumes gegeben. Jede Ebene des Baumes wird durch einen einzelnen MapReduce-Job berechnet. In der map-Funktion des ersten MapReduce-Jobs werden für jeden Datensatz die Top Level-Blockschlüssel aus den Werten der Top Level-Attribute bestimmt und ein Paar (b,record) für jeden Top Level-Blockschlüssel *b* ausgegeben. Die reduce-Funktion bestimmt die Anzahl der Datensätze eines jeden Top Level-Blocks. Die Blöcke deren Größe oberhalb des Schwellwertes liegt, bilden die Eingabe für die nächste Iteration, in der die Blöcke durch Verfeinerung ihrer Blocking-Kriterien in kleinere Subblöcke unterteilt werden. Beträgt Höhe die des Baums *n*, so sind *n* MapReduce-Jobs sequentiell auszuführen, um die Eingabedatenmenge entsprechend des Blocking-Schemas zu unterteilen. Da in jeder Ebene mehrere Attribute zur Definition von Blockschlüsseln herangezogen werden, kann jeder Datensatz mehreren Blöcken zugewiesen sein. Um die redundante Ähnlichkeitsberechnung von Datensätzen zu vermeiden, die mehr als einen gemeinsamen Block haben, werden zwei weitere MapReduce-Jobs ausgeführt. Im ersten Job erfolgt die Bestimmung des größten gemeinsamen Blocks eines jeden Kandidatenpaares. Im zweiten MapReduce-Job wird für jeden Block eine Liste von Datensatzpaaren erstellt, die zu vergleichen sind (also deren größter gemeinsamer Block der aktuell betrachtete Block ist). Die Ähnlichkeitsberechnung der verbleibenden Kandidatenpaare erfolgt in einem finalen MapReduce-Job. Die Vorgehensweise zur Vermeidung redundanter Paarvergleiche ist der Hauptkritikpunkt des Verfahrens. Im ersten MapReduce-Job wird die Menge *aller Paare* eines Blockes materialisiert und umverteilt, was in Abhängigkeit des verwendeten Blockgrößenschwellwertes mit einem massiven I/O Overhead einhergeht. Dieselben Probleme bestehen bei der Materialisierung der Liste der zu vergleichenden Paare. Die Annotation eines jeden Datensatzes mit all seinen "Blatt-Blockschlüsseln" würde nur einen einzelnen leichtgewichtigen MapReduce-Job erfordern und eine Überprüfung der "größter gemeinsamer Block-Bedingung" zur Laufzeit ermöglichen. Die Evaluation erfolgte anhand von ca. 5,7 Mio. Datensätzen in einem Hadoop-Cluster bestehend aus 80 Knoten. Über die Hardwareeigenschaften der Knoten und die Anzahl der von jedem Knoten parallel ausgeführten Map- bzw. Reduce-Tasks wurden keine Angaben gemacht. Die Autoren vergleichen ihren Ansatz mit dem klassischen Standard Blocking, geben jedoch keine Auskunft wie die Generierung von Blöcken erfolgt, was die Aussagekraft des Laufzeitvergleiches fraglich erscheinen lässt. Ein aussagekräftiger Vergleich der erreichten Qualität fehlt ebenso – das verwendete Qualitätsmaß setzt die Anzahl der *True Positives* ins Verhältnis zur Anzahl der Kandidatenpaare. Hier wäre eine Untersuchung des resultierenden *Recalls* wünschenswert gewesen, insbesondere vor dem Hintergrund, dass sehr kleine Blockgrößenschwellwerte von maximal 50 verwendet wurden. Für verschiedene Blockgrößenschwellwerte wurden die resultierende Laufzeiterhöhung betrachtet, eine Angabe der Gesamtlaufzeit sowie deren Aufschlüsselung auf die verschiedenen Schritte erfolgte nicht. Für die Deduplizierung einer größeren Datenquelle bestehend aus 5 Mrd. Datensätze und eine maximale Blockgröße von 35 geben die Autoren eine Gesamtlaufzeit von 6,5 Tagen an, wobei 1,5 Tage auf das Blocking und 5 Tage auf die Ähnlichkeitsberechnung entfallen.

Auch im Bereich der Link Discovery wird das MapReduce-Programmiermodell verwendet, um den Prozess der Entdeckung von Links zwischen Instanzen verschiedener *Linked Open Data*-Datenquellen zu beschleunigen [34]. Darüber hinaus wurden populäre Link Discovery-Frameworks, wie *SILK* [235] und *LIMES* [182], zu SILK MapReduce[36] bzw. LIMESMR [112] erweitert.

[36] `https://www.assembla.com/spaces/silk/wiki/Silk_MapReduce`

2.4.3 Entity Resolution auf GPUs

Die Verwendung von Graphikprozessoren (GPUs) zur Beschleunigung von Entity Resolution-Workflows ist ein vergleichsweise neuer Ansatz und wurde erstmals in [83] betrachtet. Moderne GPUs besitzen tausende von Cores und erlauben die massiv-parallele Ausführung derselben Menge von Instruktionen auf disjunkten Datenpartitionen. Mit CUDA[37] und OpenCL[38] existieren zwei populäre *General Purpose Computation on Graphics Processing Unit*-Frameworks, welche die Verwendung von GPUs zur Parallelisierung beliebiger Algorithmen ermöglichen. In [83] wird OpenCL verwendet, welches GPUs (und CPUs) verschiedener Hersteller unterstützt, wohingegen CUDA auf NVIDIA GPUs zugeschnitten ist. Ein OpenCL-Programm wird als *Kernel* bezeichnet und in einem Dialekt der Programmiersprache *C* formuliert. Die Anzahl der Kernel, die gleichzeitig ausgeführt werden können, hängt von der Anzahl der Cores und des verfügbaren Speichers (aktuell bis zu 16GB) der verwendeten GPU, den Speicheranforderungen einer Kernel-Instanz sowie der Größe der Eingabedaten und der generierten Ausgabedaten ab. Bei der Portierung von Algorithmen auf Graphikprozessoren sind verschiedene Einschränkungen zu beachten. GPUs besitzen eine vierstufige Speicherhierarchie, wobei die Speicherkapzität in jeder Stufe abnimmt und die Zugriffszeit sinkt. Eingabedaten müssen vor Beginn jeder Berechnung vom Hauptspeicher des Host-Systems in den Speicher der GPU kopiert werden. Analog dazu müssen Ausgabedaten vom Speicher der GPU zur weiteren Bearbeitung in den Hauptspeicher des Host-Systems zurückkopiert werden. Auf der GPU verwendete Speicherbereiche müssen vom Programmierer zuvor manuell allokiert und später wieder freigegeben werden, eine dynamische Allokation von Speicherbereichen zur Laufzeit ist *nicht* möglich. Üblicherweise ist der verfügbare Speicher handelsüblicher GPUs deutlich kleiner als der zur Verfügung stehende Hauptspeicher. Dies erfordert i. d. R. eine Aufteilung der zur verrichteten Arbeit in mehrere Teilaufgaben, die hintereinander von der GPU bearbeitet werden. Dabei ist es entscheidend, die Abarbeitung der Teilaufgaben so zu organisieren, dass durch Wiederverwendung von Daten, die sich bereits im Speicher der GPU befinden, das zwischen Hauptspeicher und GPU transferierte Datenvolumen minimiert wird. Bei der Definition eines OpenCL-Programms sind weitere Einschränkungen, wie z. B. die Vermeidung von Fallunterscheidungen, zu berücksichtigen. GPUs gruppieren (üblicherweise 32) Kernel-Instanzen in sogenannte *Warps*. Alle Threads eines Warps führen zu einem Zeitpunkt dieselbe Operation auf verschiedenen Daten aus. Benötigt ein Thread dafür mehr Zeit als die übrigen Threads, so müssen diese auf den langsameren Thread warten. Darüber hinaus werden bedingte Anweisungen im Programmablauf serialisiert, jeder Thread wartet bis alle Threads des Warps eines *If*-Zweiges bearbeitet haben bis er mit der Bearbeitung des *Else*-Zweiges beginnt. Nach jeder Fallunterscheidung erfolgt ebenfalls eine Synchronisation der Threads (vgl. [83]). Des Weiteren können nur bestimmte Basisdatentypen wie, z. B. *int*, *float*, *double* und Arrays, verwendet werden. Zudem besteht die Notwendigkeit, benötigten Speicher zum Ablegen der Ergebnisse *vor* der Berechnung zu allokieren. Beispielsweise muss bei der Bestimmung von $\{(a,b) \subseteq A \times B \mid sim(a,b) > t\}$ prinzipiell vom ungünstigsten Fall ausgegangen werden, dass die Ähnlichkeit aller Paare oberhalb des Schwellwertes liegt, was die Reservierung eines Speicherbereiches für $|A| \cdot |B|$ Paare erfordert.

Die Autoren von [83] zeigen wie ein Entity Resolution-Workflow bestehend aus Blocking, Ähnlichkeitsberechnung, Klassifikation sowie der Berechnung der transitiven Hülle mittels GPUs parallelisiert werden kann. Die Autoren untersuchen zunächst die Evaluation des Kartesischen Produktes einer Eingabedatenquelle. Ein wesentliche Strategie dabei ist eine Aufteilung der Eingabedatenmenge in disjunkte Partitionen. In mehreren Runden wird die Ähnlichkeit aller Datensätze von je zwei Partitionen berechnet. Das Scheduling der Partitionspaare wird so organisiert, dass in jeder Runde nur eine Partition zur GPU kopiert werden muss und mit einer sich bereits im Speicher der GPU befindenden Partition abgeglichen wird. Die Autoren berücksichtigen dabei die Tatsache, dass die Größe der einzelnen Datensätze variiert. Da der auf der GPU zur Verfügung stehende Speicher möglichst voll ausgenutzt werden soll, ist die Wahl einer fixen Partitionsgröße suboptimal. Stattdessen schlagen die Autoren eine dynamische Bestimmung der Partitionsgrößen in Abhängigkeit der Gesamtgröße der enthaltenen Datensätze sowie

[37] `https://developer.nvidia.com/category/zone/cuda-zone`
[38] `https://www.khronos.org/opencl/`

des Speicherplatzes, der zur Ähnlichkeitsberechnung und zur Speicherung der Ergebnisse notwendig ist, vor. Des Weiteren werden eine GPU-basierte Umsetzung des Sorted Neighborhood-Verfahrens sowie GPU-Implementierungen verschiedener Ähnlichkeitsfunktionen beschrieben. Zur Aggregation der von n Ähnlichkeitsfunktionen berechneten Ähnlichkeitswerte eines Datensatzpaares in einen Gesamtähnlichkeitswert schlagen die Autoren eine GPU-seitige Sortierung der berechneten (id_i, id_j, sim)-Tupel vor, die durch die CPU gemischt und aggregiert werden. Abschließend beschreiben die Autoren einen Algorithmus, der unter Verwendung von Bitmasken zur Reduzierung des Speicherbedarfs, in mehreren Runden die transitive Hülle der ermittelten Duplikate bestimmt. Zur Evaluation wurde ein Datensatz bestehend aus 1,8 Millionen Datensätzen (Musik-CDs) unter Verwendung verschiedener CPUs und GPUs verwendet. Aufgrund der Komplexität $O(n^3)$ dominiert (bei der Verwendung des Sorted Neighborhood-Blockings) die Berechnung der transitiven Hülle mit steigender Datenmenge die Gesamtausführungszeit des Entity Resolution-Workflows. Aufgrund der relativ kleinen Fenstergröße von 20 ist dieser Schritt jedoch notwendig, um eine gute Match-Qualität gewährleisten zu können. Die Gesamtausführungszeit unter Verwendung der besten GPUs war um den Faktor 10 kleiner als die Ausführungszeit, die mit der besten CPU erreicht werden konnte. Abschließend setzen die Autoren die mit verschiedenen CPUs und GPUs benötigten Ausführungszeiten in das Verhältnis zu deren Kaufpreis. Die resultierenden Preis-Performance-Kennzahlen zeigen, dass (zumindest solange die zu Eingabedaten überhaupt durch einen einzelnen Rechner bearbeitet werden können) die Parallelisierung von Entity Resolution-Workflows mittels GPUs, auch unter Berücksichtigung finanzieller Gesichtspunkte, der Parallelisierung mittels Multi-Core CPUs überlegen ist.

Zuletzt wurde in einer weiteren Vorarbeit zu dieser Dissertation die Parallelisierung paarweiser Ähnlichkeitsberechnungen unter Verwendung von GPUs betrachtet [101]. Dies erfolgte im Kontext des *Ontology Matchings*, die untersuchten Methoden sind jedoch auf das Entity Resolution-Problem übertragbar. Im Mittelpunkt der Arbeit steht die GPU-basierte Bestimmung aller Paare von Konzepten zweier Ontologien, deren n-Gramm-Ähnlichkeit (bezüglich der Werte eines Attributes) einen vorgegebenen Mindestschwellwert überschreitet. Neben der unterschiedlichen Länge der Attributwerte kommt dabei erschwerend hinzu, dass ein Konzept für jedes Attribut mehrere Synonyme neben dem eigentlichen Attributwert aufweisen kann. Die GPU-Implementierung basiert auf einer algorithmischen Optimierung, die zuvor für CPUs evaluiert wurde. Nach einer Tokenisierung der zur Ähnlichkeit verwendeten Attributwerte und Synonyme wird jedes Konzept als Liste(n) von Tokens (n-Grammen) repräsentiert, wobei jedes Token durch einen numerischen Wert ersetzt wird, der der Position des (neu eingefügten) Tokens in einem globalen Wörterbuch entspricht. Dies erlaubt eine Reduzierung des Speicherbedarfs zur Repräsentation von Konzepten im Speicher der GPU und beschleunigt die Überprüfung der Gleichheit zweier Token (bestehend aus gleich langen Zeichenketten). Eine Sortierung der Tokenliste(n) eines jeden Konzepts ermöglicht zudem eine effiziente Bestimmung der Anzahl überlappender Token zweier Tokenlisten durch einen sequentiellen, schritthaltenden Durchlauf der sortierten Listen. Die Menge der Konzepte zweier Ontologien O, O' wird in Partitionen $P_i \subseteq O, Q_i \subseteq O'$ unterteilt, wobei jeder Partition die gleiche Anzahl an Konzepten zugewiesen wird. Anschließend werden in mehreren Runden Partitionspaare (P_i, Q_i) zur GPU transferiert und die Ähnlichkeiten der enthaltenen Konzepte berechnet. Eine Kernel-Instanz vergleicht ein Konzept $c \in P_i$ mit allen Konzepten $c \in Q_i$. Dabei werden alle Synonyme eines Attributwertes berücksichtigt – für jedes Konzeptpaar wird eine aggregierte Ähnlichkeit ermittelt. Um den effizienten Zugriff auf alle Tokens eines Attributwertes eines Konzeptes zu ermöglichen, wird jede Partition P_i bzw. Q_i durch eine Indexstruktur bestehend aus 3 Arrays repräsentiert, die alle im globalen Speicher (größte Kapazität, höchste Latenz) der GPU abgelegt werden. OpenCL weist jeder Kernel-Instanz eine global eindeutige Kennung zu, die verwendet wird, um die von einer Kernel-Instanz benötigten Eingabedaten aus dem globalen Speicher der GPU zu laden und die berechneten Ähnlichkeiten in einen bestimmten Bereich in ein Ergebnis-Array zu schreiben. Um in jeder Runde die Größe des vorab zu allokierenden Ergebnis-Arrays zu beschränken, werden für jedes Konzept $c \in O$ lediglich die k Konzepte $c' \in O'$ mit der höchsten Ähnlichkeit oberhalb des Ähnlichkeitsschwellwertes bestimmt. Da heutzutage selbst handelsübliche Desktop-PCs mit Multi-Core-Prozessoren ausgestattet sind, wurde zur weiteren Beschleunigung eine hybride Parallelbearbeitung von Partitionspaaren (P_i, Q_i) durch

mehrere CPU-Threads *und* die GPU vorgeschlagen. Dazu wurde das Ausführungsschema aus [83] angepasst, um das zwischen Hauptspeicher und GPU transferierten Datenvolumen auch unter Berücksichtigung von "CPU-Runden" zu minimieren. Zur Evaluation wurde ein Desktop-PC mit einem Quad-Core-Prozessor und einer handelsüblichen Graphikkarte verwendet. Zunächst wurde unter Verwendung eines CPU-Threads der Effekt der algorithmischen Optimierung am Beispiel eines eines Ontology Matching-Problems mit ca. 79.000 bzw. 67.000 Konzepten untersucht. Dazu wurde die Laufzeit der vorgestellten Implementierung mit der Laufzeit einer zeichenkettenbasierten *Nested Loop*-Implementierung sowie einer zeichenkettenbasierten *Hashtabellen*-Implementierung verglichen. Die Verwendung sortierter Tokenlisten und die Repräsentation von Token als primitive ganzzahlige Datentypen beschleunigte die Bearbeitung um den Faktor 13 gegenüber der Nested Loop-Implementierung und um den Faktor $4,3$ gegenüber der Hashtabellen-basierten Implementierung. Die Ausführung der besten Implementierung auf der GPU führte zu einer weiteren Reduktion der Ausführungszeit von 8 Minuten auf 100 Sekunden, was einer Beschleunigung um den Faktor $4,8$ entspricht. In einem zweiten Experiment wurde gezeigt, dass durch die Nutzung aller zur Verfügung stehenden Ressourcen des verwendeten Rechners die benötige Ausführungszeit noch weiter reduziert werden kann. Die gemeinsame Bearbeitung des Ontology Matching-Tasks durch drei CPU-Threads und die GPU resultierte in einer Ausführungszeit von 67 Sekunden. Die naheliegende Verwendung eines vierten CPU-Threads brachte für den verwendeten Quad-Core-Prozessor keine weitere Beschleunigung, da stets ein zusätzlicher dedizierter CPU-Thread benötigt wird, um die Eingabedaten zur GPU zu transferieren, auf das Ende der Berechnung zu warten und um die Ergebnisse aus dem Speicher der GPU in den Hauptspeicher zu transferieren.

3

Das Dedoop-Framework

Dedoop (Deduplication with Hadoop) ist ein MapReduce-basiertes Entity Resolution-Framework. Dieses Kapitel stellt im Abschnitt 3.1 das Dedoop-Framework mit seinen Funktionalitäten vor. Abschnitt 3.2 erläutert die Konfiguration und Ausführung von Entity Resolution-Workflows. Die Transformation eines vorgegebenen Workflows in eine Menge ausführbarer MapReduce-Jobs wird im Abschnitt 3.3 thematisiert. Im Abschnitt 3.4 wird auf die durch die Parallelisierung auftretenden Probleme und die in Dedoop verfolgten Strategien zu deren Lösung eingegangen. Abschließend erfolgt eine Zusammenfassung und eine Abgrenzung gegenüber bestehenden Arbeiten.

3.1 Überblick

Die Verwendung des MapReduce-Programmiermodells erlaubt eine parallele Ähnlichkeitsberechnung von Datensätzen in beliebig großen Rechner-Clustern und ist damit hervorragend geeignet, um die Ausführung rechenintensiver Entity Resolution-Workflows (z. B. als Bestandteil eines ETL-Prozesses[1]) zu beschleunigen. Abbildung 3.1 zeigt das grundlegende Schema der MapReduce-Umsetzung eines einfachen Entity Resolution-Workflows mit Standard Blocking. Zwei Map-Tasks lesen die im verteilten Dateisystem gespeicherten Datensätze ein und wenden die map-Funktion auf jeden Datensatz an. Die map-Funktion bestimmt für jeden Datensatz, entsprechend des vorgegebenen Blocking-Schemas, einen oder mehrere Blockschlüssel (hier den Produkttyp). Anschließend werden alle Datensätze zu den beiden Reduce-Tasks umverteilt, und zwar so, dass alle Datensätze desselben Blockschlüssels (Produkttyps) demselben Reduce-Task zugewiesen werden. Im Falle der Zuweisung eines Datensatzes zu n Blöcken werden n Replikate des ursprünglichen Datensatzes umverteilt. Die Reduce-Tasks gruppieren die eingehenden Datensätze und rufen für jede Gruppe (Produkttyp) die reduce-Funktion auf, in der die paarweise Ähnlichkeitsberechnung der Datensätze eines bestimmten Produkttyps erfolgt. Direkt im Anschluss an die Bestimmung der Ähnlichkeit zweier Datensätze wird, basierend auf dem ermittelten Ähnlichkeitswert, die Klassifikation des Datensatzpaares in *Match* oder *Non-Match* vorgenommen. Alle auf diese Weise ermittelten Duplikate werden in das finale Match-Ergebnis aufgenommen, das wiederum im verteilten Dateisystem gespeichert wird.

Obwohl der skizzierte Ansatz konzeptionell simpel erscheint, stellt sich heraus, dass die manuelle Spezifikation und Parametrisierung entsprechender MapReduce-Programme sowie deren Ausführung in MapReduce-Clustern ein sehr arbeits- und zeitwaufwändiger sowie fehleranfälliger Prozess ist. Darüber hinaus wird die MapReduce-Parallelisierung von Entity Resolution-Strategien dadurch erschwert, dass Blocking-Verfahren, wie z. B. Sorted Neighborhood, eine völlig andere Datenumverteilung erfordern und auch Datensätze mit unterschiedlichen Blockschlüsseln miteinander verglichen werden müssen. Des Weiteren ist der skizzierte Ansatz hochgradig anfällig für Lastbalancierungsprobleme bei Datenungleichverteilung (Anzahl der Produkte je Produkttyp). Im Falle der Mehrfachzuweisung von Datensätzen zu Blöcken (z. B. Produkttyp und Preisintervall) ist es zudem wahrscheinlich, dass Datensatzpaare

[1] ETL Prozess = Prozess der Extraktion und Transformation von Daten unterschiedlicher Datenquellen in eine zentrale Datenbank zum Zweck der quellenübergreifenden Datenanalyse.

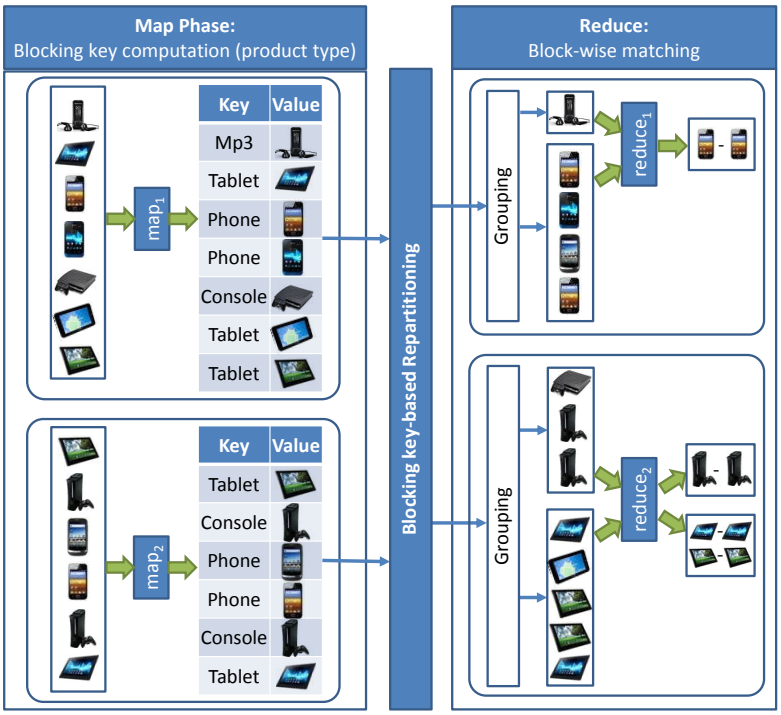


Abbildung 3.1: Vereinfachter Ablauf eines MapReduce-Programms zur Deduplizierung einer einzelnen Datenquelle von Produktdatensätzen unter Verwendung von zwei Map- und zwei Reduce-Tasks. Die Datensätze werden durch das MapReduce-Framework nach ihrem Produkttyp gruppiert und umverteilt. Anschließend erfolgt die Ähnlichkeitsberechnung aller Datensätze eines Blocks. Das Ergebnis umfasst alle Datensatzpaare, die als Duplikate klassifiziert wurden.

unnötigerweise mehrfach bezüglich unterschiedlicher Blockschlüssel verglichen werden. Dedoop bietet folgende Funktionalitäten, um die aufgeführten Probleme zu lösen:

- Dedoop erlaubt die einfache Spezifikation komplexer Entity Resolution-Workflows mittels einer graphischen Benutzeroberfläche. Nutzer können dabei auf eine umfangreiche Bibliothek von Entity Resolution-Techniken (Blocking-Verfahren, Methoden der Blockschlüsselgenerierung, Ähnlichkeitsmetriken, Klassifikationsmethoden) zurückgreifen und diese geeignet parametrisieren. Diese umfasst auch Methoden des maschinellen Lernens zur automatischen Generierung von Klassifikationsmodellen.

- Die nutzerdefinierten Entity Resolution-Workflows werden automatisch in einen ausführbaren MapReduce-Workflow, bestehend aus einer Menge von sequentiell ausgeführten MapReduce-Jobs, überführt und auf einem MapReduce-Cluster zur Ausführung gebracht. Die Ergebnisse können anschließend begutachtet und zur weiteren Verwendung aus dem verteilten Dateisystem kopiert werden.

- Dedoop unterstützt die simultane Bearbeitung und Überwachung mehrerer Entity Resolution-Workflows durch verschiedene Nutzer. Workflows können auf verschiedenen Clustern zur Ausführung gebracht werden.

- Um eine effiziente und gleichmäßige Ausnutzung der zur Verfügung stehenden Rechenressourcen zu gewährleisten, beinhaltet Dedoop Algorithmen zur Lastbalancierung, die für einen beliebigen

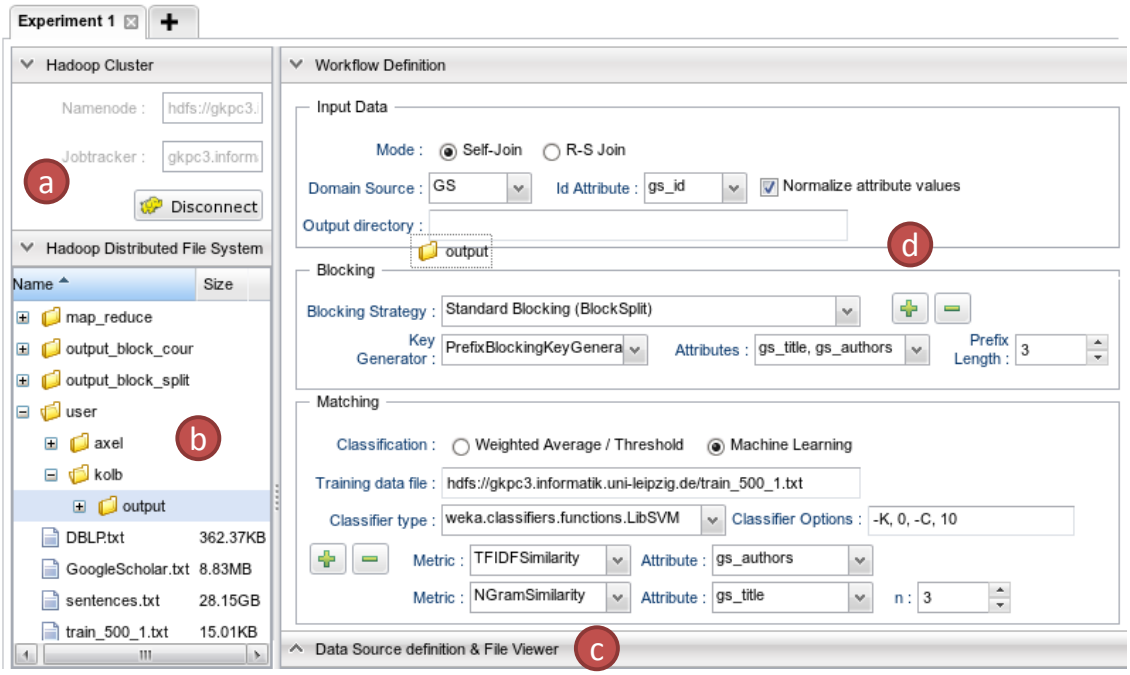


Abbildung 3.2: Graphische Benutzeroberfläche des Dedoop-Frameworks zur Spezifikation von Entity Resolution-Workflows.

nutzerdefinierten Entity Resolution-Workflow sicherstellen, dass jeder Reduce-Task die gleiche Menge an Paarvergleichen durchführt. Darüber hinaus wird sichergestellt, dass keine redundanten Paarvergleiche durchgeführt werden.

3.2 Workflow-Spezifikation und -Ausführung

Das Dedoop Framework stellt ein graphisches Webinterface (siehe Abbildung 3.2[2]) zur Konfiguration von Entity Resolution-Workflows bereit. Jeder Workflow wird in einem eigenen Tab spezifiziert ist mit einem laufenden Hadoop-Cluster assoziiert (Abbildung 3.2a). Zur Vereinfachung des Datenaustausches zwischen dem lokalen Dateisystem eines Nutzers und dem Hadoop-Cluster beinhaltet Dedoop einen komfortablen graphischen Dateimanager (Abbildung 3.2b), der u. a. den Upload von Eingabedaten in das verteilte Dateisystem bzw. den Download (komprimierter) Ausgabedaten ermöglicht. Zudem ist ein Client für den Zugriff auf den Speicherdienst Amazon S3 enthalten. Dessen Verwendung ist insbesondere bei der Nutzung des EC2-Dienstes sinnvoll, da der Datentransfer zwischen S3 und EC2-Instanzen sehr schnell und zudem kostenfrei ist. Ein Dateibetrachter (Abbildung 3.2c, eingeklappt) ermöglicht die Inspektion von HDFS-Dateien. Gleichzeitig dient er zur Spezifikation von (eine oder mehrere HDFS-Dateien umfassenden) Datenquellen sowie zur Festlegung und Benennung relevanter Attribute, die während der Workflow-Definition verwendet werden können.

Der Hauptbestandteil der Benutzeroberfläche (Abbildung 3.2d) dient der eigentlichen Workflow-Spezifikation. Dieser Bereich ist in drei Abschnitte aufgeteilt. Dedoop unterstützt sowohl die Duplikaterkennung innerhalb einer einzelnen als auch zwischen zwei Datenquellen. Im Abschnitt *Input Data* erfolgt die Angabe der Eingabedatenquellen, des einen Datensatz eindeutig kennzeichnenden Attribu-

[2] Weitere Screenshots sind unter `http://dbs.uni-leipzig.de/dedoop_slideshow/dedoop_slideshow.html` verfügbar.

tes sowie des HDFS-Ausgabeverzeichnisses. Im Falle zweier Eingabedatenquellen ist zudem das Mapping einander entsprechender Attribute zu konfigurieren. Der Abschnitt *Blocking* dient zur Festlegung der verwendeten Blocking-Strategie. Hier kann neben der Evaluierung des Kartesischen Produktes zwischen dem Sorted Neighborhood-Verfahren und dem Standard Blocking gewählt werden. Zur Generierung der verwendeten Blockschlüssel kommen ein oder mehrere Blockschlüsselgeneratoren zum Einsatz. Ein solcher Blockschlüsselgenerator bestimmt für jeden Datensatz basierend auf den Werten eines oder mehrerer Attribute eine Menge von Blockschlüsseln. Neben der Möglichkeit, einen Datensatz zur Gewährleistung der Robustheit gegenüber Datenqualitätsproblemen mehreren Blöcken zuzuordnen, erlaubt diese Vorgehensweise, die MapReduce-Umsetzung von Blocking-Verfahren, wie z. B. Q-gram Indexing oder Suffix Array Indexing, aus einer MapReduce-Implementierung des Standard Blockings abzuleiten. Die Konfiguration der Ähnlichkeitsberechnung (Ähnlichkeitsmetriken und Attribute) sowie der abschließenden Klassifikation erfolgt im Abschnitt *Matching*. Nutzer können prinzipiell zwischen einer regelbasierten Klassifikation (z. B. als Konjunktion von Ähnlichkeitswerten) und einer Klassifikation basierend auf einem gewichteten Durchschnitt und einem Gesamtähnlichkeitsschwellwert wählen. Zusätzlich wird eine automatische Klassifikation auf Basis eines mittels maschineller Lernverfahren berechneten Klassifikationsmodells (z. B. Entscheidungsbaum, Support Vector Machine, logistische Regression) unterstützt. In diesem Fall sind zusätzlich manuell gelabelte Trainingsdaten anzugeben, anhand derer das Modell gelernt wird. Optionale Nachverarbeitungsschritte sind die Berechnung der transitiven Hülle des Match-Ergebnisses, zur Bestimmung indirekter *matches* und die Evaluierung der resultierenden Qualität bezüglich eines perfekten Match-Ergebnisses.

Das Dedoop-Framework ist vollständig in Java implementiert. Ein Kernbestandteil ist eine auf dem *Google Web Toolkit*[3] und der *Smart GWT*-Bibliothek[4] basierende Webanwendung. Client-seitige Aktionen werden serverseitig von Java Servlets verarbeitet, die mit den Namenodes und Jobtrackern verbundener Hadoop-Cluster interagieren. Abbildung 3.3 zeigt die Architektur und die wesentlichen Funktionalitäten des Dedoop-Frameworks. Es ermöglicht die interaktive Spezifikation und Ausführung mehrerer Entity Resolution-Workflows durch verschiedene Benutzer mithilfe einer Browser-basierten graphischen Benutzeroberfläche. Jeder Workflow kann dabei prinzipiell auf einem anderen Hadoop-Cluster (z. B. einem lokalen Cluster und einem EC2-Cluster) ausgeführt werden. Für jedes Cluster verwaltet Dedoop einen sogenannten *Workflow Executer Thread* und eine Schlange abzuarbeitender Workflows. Diese werden nach dem FIFO-Prinzip aus der Schlange entfernt und in eine Sequenz von MapReduce-Jobs überführt, die nacheinander ausgeführt werden. Für die Überführung wird auf eine Bibliothek von MapReduce-Programmen zurückgegriffen, die entsprechend des nutzerdefinierten Workflows kombiniert und parametrisiert werden. Nach dem Scheduling eines Workflows durch den Nutzer setzt der Browser periodisch Ajax-Anfragen an den Server ab, um den Fortschritt des Workflows zu erfragen und die Benutzeroberfläche zu aktualisieren.

Durch die Integration einer *Amazon Web Services*-Bibliothek wird eine Automatisierung des wiederkehrenden, arbeitsaufwändigen Prozesses, mittels des Infrastructure as a Service-Dienstes Amazon EC2, eine Menge von virtuellen Maschinen zu instanziieren und darauf ein Hadoop-Cluster einzurichten und zu starten, ermöglicht. Dedoop stellt zu diesem Zweck umfangreiche Eingabemasken für die Konfiguration von EC2-Parametern (z. B. Anzahl virtueller Maschinen, Instanztypen, das verwendete AMI[5], geographische Region, privater Schlüssel für Zugriff) und Hadoop-Parametern (z. B. Map- und Reduce-Task-Kapazität eines Knotens, maximale Heap Size für Map- und Reduce-Tasks, verwendete Ports) bereit. Da die Virtuellen Maschinen eines EC2-Clusters untereinander über private IP-Adressen kommunizieren sollen (kostenfreier und schneller Datentransfer), diese jedoch von Clients außerhalb des EC2-Netzwerks nicht aufgelöst werden können, startet Dedoop vollautomatisiert einen SOCKS Proxy-Server für jedes verbundene, auf EC2 laufende, Hadoop-Cluster. Dies dient dazu, alle Verbindungen von außerhalb des EC2-Netzes über eine SSH-Verbindung zu einem Hadoop-Knoten innerhalb des EC2-Netzwerks,

[3] `http://www.gwtproject.org`

[4] `http://www.smartclient.com/product/smartgwt.jsp`

[5] AMI = Amazon Machine Image, entspricht dem Abbild eines virtuellen Rechners (inkl. Betriebssystem und benötigter Anwendungen, vgl. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html`).

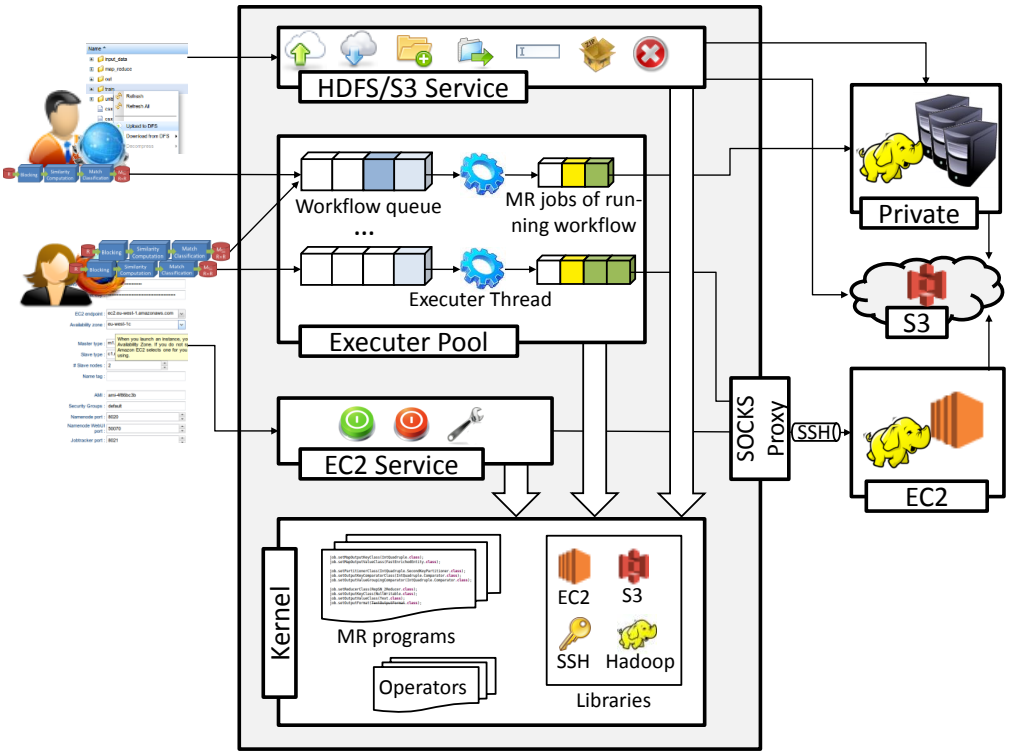


Abbildung 3.3: Architektur des Dedoop-Frameworks.

zu tunneln.

3.3 Transformation eines Entity Resolution-Workflows in einen ausführbaren MapReduce-Workflow

Der obere Teil der Abbildung 3.4 zeigt den allgemeinen, von Dedoop unterstützten, Entity Resolution-Workflow für zwei Datenquellen R und S. Die roten Bausteine repräsentieren Eingabe- oder Ausgabedaten wohingegen die blauen Bausteine die einzelnen Teilschritte[6] eines Entity Resolution-Workflows repräsentieren. Der erste Teilschritt ist lediglich bei der Verwendung eines maschinellen Lernverfahrens zur automatischen Generierung eines Klassifikationsmodells relevant. Er benötigt eine Menge T von manuell gelabelten Trainingsdaten als Eingabe. Der eigentliche Teil des Entity Resolution-Workflows besteht aus vier aufeinanderfolgenden Schritten: Blocking, Ähnlichkeitsberechnung, Klassifikation und Berechnung der transitiven Hülle. Das Ergebnis des Entity Resolution-Workflows besteht aus der Teilmenge des Kartesischen Produktes der Eingabedaten, die als *Match* klassifiziert wurden.

Dedoop bildet einen gegebenen Entity Resolution-Workflow in eine Sequenz von vier MapReduce-Jobs ab (unterer Teil der Abbildung 3.4), die auf einem Hadoop-Cluster ausgeführt werden können. Die ersten beiden Jobs sowie der letzte Job sind optional (gekennzeichnet durch die kursive Beschriftung). Der *Blocking-based Matching Job* implementiert, ähnlich zu der in Abbildung 3.1 skizzierten Grundidee, das Blocking, die Ähnlichkeitsberechnung und die Klassifikation. Dieser Job ist obligatorisch und der mit Abstand zeitaufwändigste Bestandteil der Berechnung. Im Folgenden wird ein kurzer Überblick

[6] Auf die Darstellung weiterer Vor- und Nachverarbeitungsschritte wurde zugunsten der Übersichtlichkeit verzichtet.

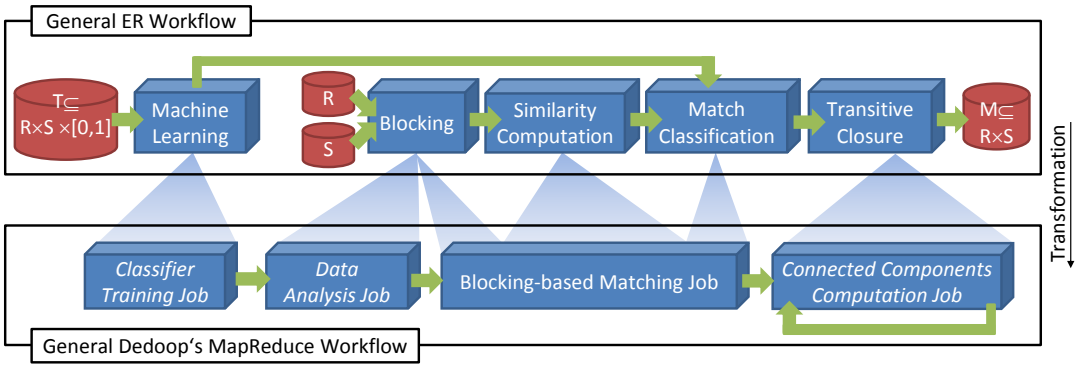


Abbildung 3.4: Transformation eines nutzerdefinierten Entity Resolution-Workflows (oben) in einen ausführbaren MapReduce-Workflow (unten).

über die vier MapReduce-Jobs gegeben.

Classifier Training Job: Im Falle einer auf maschinellen Lernverfahren basierenden Klassifikation wird von Dedoop ein separater MapReduce-Job ausgeführt, der entsprechend die Ähnlichkeitsberechnung auf den Trainingsbeispielen durchführt. Auf Basis der ermittelten Ähnlichkeitswerte und der für jedes Trainingsbeispiel vorgegeben Information, ob es sich hierbei um ein Duplikat handelt oder nicht, wird ein Klassifikationsmodell gelernt, das die *matches* bestmöglich von den *Non-Matches* trennt. Hierbei greift Dedoop auf die populäre *WEKA*-Bibliothek[7] zurück. Das resultierende Modell wird mithilfe Hadoops *Distributed Cache*[8]-Mechanismus allen Clusterknoten zur späteren Verwendung zugänglich gemacht.

Data Analysis Job: Wie in Abbildung 3.1 skizziert, erfolgt die Ähnlichkeitsberechnung bei der MapReduce-Umsetzung eines Blocking-basierten Entity Resolution-Workflows in der Reduce-Phase. Aufgrund der quadratischen Komplexität der Ähnlichkeitsberechnung pro Block ist die skizzierte Realisierung hochgradig anfällig gegenüber einer Datenungleichverteilung. Für Realweltprobleme führt dies dazu, dass die Gesamtausführungszeit von einem oder einigen wenigen Reduce-Tasks dominiert wird. Der Analyse-Job dient im Wesentlichen der Generierung von Statistiken über die Größenverteilung der einzelnen Blöcke. Diese Informationen werden durch den darauffolgenden Job ausgenutzt, um mittels eines auf das verwendete Blocking-Verfahren zugeschnittenen Lastbalancierungsalgorithmus die Datenumverteilung so anzupassen, dass alle Reduce-Tasks (ungefähr) dieselbe Menge an Paarvergleichen durchführen und damit gleich ausgelastet sind.

Blocking-based Matching Job: Die Arbeitsweise des dritten MapReduce-Jobs hängt vom verwendeten Blocking-Verfahren und dem gewählten Lastbalancierungsalgorithmus ab. Wie bereits erwähnt, erfolgt die Blockschlüsselberechnung innerhalb der Map-Phase. Im Falle des Standard Blockings (und ähnlicher Blocking-Verfahren) gibt die map-Funktion für jeden Eingabedatensatz ein oder mehrere (block_key, entity)-Paare aus, die anhand der Blockschlüssel zu den Reduce-Tasks umverteilt werden. Alle Datensätze eines Blocks werden demselben Reduce-Task zugewiesen und in einem Aufruf der reduce-Funktion miteinander verglichen. Das Sorted Neighborhood-Verfahren verlangt eine andere Datenumverteilung, da eine globale Sortierung der Datensätze bezüglich der Blockschlüssel über alle Reduce-Tasks hinweg gegeben sein muss. Zusätzlich müssen auch Datensätze (innerhalb eines Fensters der Größe w) über Reduce-Task-Grenzen hinweg miteinander verglichen werden. Dedoop verwendet den RepSN-Algorithmus [131, 134], um diese Anforderungen zu erfüllen. Dessen Grundidee ist die Verwendung einer Bereichspartitionierungsfunktion, um jeden Datensatz anhand seines Blockschlüssels einem Reduce-Task zuzuweisen und eine korrekte Sortierreihenfolge zu gewährleisten. Jeder Reduce-

[7] http://www.cs.waikato.ac.nz/ml/weka/index.html
[8] http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/filecache/DistributedCache.html

Task schiebt anschließend ein Fenster der Größe *w* über die ihm zugewiesenen und sortierten Datensätze und berechnet die Ähnlichkeit aller Datensätze innerhalb des Fensters. Um den Vergleich von Datensätzen mit einem Abstand kleiner *w* zu ermöglichen, die verschiedenen Reduce-Tasks zugewiesen wurden, repliziert die map-Funktion Datensätze nahe der Partitionsgrenze und weist die Replikate dem darauffolgenden Reduce-Task (oder sogar weiteren Reduce-Tasks) zu. Zu Lastbalancierungszwecken werden in beiden Fällen die während des Analyse-Jobs gesammelten Informationen ausgenutzt, um die Zuweisung von Daten zu Reduce-Tasks zu beeinflussen und trotzdem die Ähnlichkeit aller (und ausschließlich der) nach dem Blocking-Schritt verbleibenden Kandidatenpaare zu berechnen. Alle in der Arbeit vorgestellten Lastbalancierungsalgorithmen nutzten die Tatsache aus, dass Datensatzpaare völlig unabhängig voneinander verarbeitet werden können. Für jedes Kandidatenpaar wird direkt nach der Ähnlichkeitsberechnung eine Klassifikation in *Match* oder *Non-Match* vorgenommen; im ersten Fall wird das Kandidatenpaar vom bearbeitenden Reduce-Task in eine taskspezifische Datei in das Ausgabeverzeichnis geschrieben. Sofern keine nachgelagerte Berechnung der transitiven Hülle des Match-Ergebnisses erfolgt, bildet die (logische) Konkatenation der einzelnen Ausgabedateien im Ausgabeverzeichnis das finale Match-Ergebnis.

Connected Components Computation Job: Die Berechnung der transitiven Hülle des Match-Ergebnisses erlaubt die Entdeckung weiterer, indirekt verbundener, Matches. Zu diesem Zweck wird das Match-Ergebnis als ungerichteter Graph interpretiert und eine iterative Berechnung zusammenhängender Graphkomponenten vorgenommen. Dies resultiert in einer wiederholten Ausführung des MapReduce-Jobs, wobei die Ausgabe einer Iteration die Eingabe der nächsten Iteration bildet.

Weitere MapReduce-Jobs: In Abhängigkeit von der konkreten Konfiguration des Entity Resolution-Workflows können zu den im unteren Teil der Abbildung 3.4 dargestellten drei MapReduce-Jobs noch weitere, Vor- oder Nachverarbeitungsschritte realisierende MapReduce-Jobs hinzukommen. Ein typischer Vorverarbeitungsschritt ist die Ausführung eines MapReduce-Job zur Bestimmung globaler Datensatzcharakteristika, wie z. B. Termhäufigkeiten (TF-IDF-Ähnlichkeit) oder Blockgrößenverteilungen (Suffix Array Indexing). Die Resultate dieser Vorverarbeitungsschritte werden den nachfolgenden (regulären) MapReduce-Jobs mittels des Distributed Caches zur Verfügung gestellt. Ein typischer Nachverarbeitungsschritt ist die Bestimmung der Qualität des Match-Ergebnisses bezüglich eines perfekten Match-Ergebnisses.

3.4 Probleme und Lösungsansätze

Die Parallelisierung von Entity Resolution-Workflows auf Basis des MapReduce-Programmiermodells impliziert im Vergleich zu einer (sequentiellen) Verarbeitung auf einem Rechner wesentliche Probleme, die eine effiziente Verarbeitung erschweren.

Das größte Problem ist die Anfälligkeit einer MapReduce-Umsetzung Blocking-basierter Entity Resolution-Workflows gegenüber Datenungleichverteilungen. Da die Anzahl der Paarvergleiche innerhalb eines Blocks quadratisch mit dessen Größe wächst, sind stark variierende Ausführungszeiten bei der Bearbeitung der einzelnen Reduce-Tasks zu beobachten. Im Beispiel von Abbildung 3.1 hat der erste Reduce-Task 6 (=0+6) Paarvergleiche durchzuführen, während der zweite Reduce-Task mit 13 (=3+10) Paarvergleichen beinahe doppelt so viele Ähnlichkeitsberechnungen durchführen muss. Unter der Annahme, dass für jeden Datensatzvergleich eine annähernd konstante Zeitspanne benötigt wird, ist offensichtlich, dass zur Bearbeitung des zweiten Reduce-Tasks wesentlich mehr Zeit benötigt wird während nach erfolgreicher Bearbeitung des ersten Reduce-Tasks Rechenressourcen frei, aber ungenutzt sind. “Große” Blöcke verhindern deshalb, effektiv gesehen, die Auslastung von mehr als einer kleinen Anzahl an Reduce-Tasks und Knoten. Da Realweltdaten starke Datenungleichverteilungen aufweisen (vgl. [259]) und die aus den Attributwerten der Datensätze abgeleiteten Blockschlüssel diese Datenungleichverteilung reflektieren, sind Techniken zur Lastbalancierung unbedingt erforderlich, um die Skalierbarkeit und Laufzeiteffizienz entsprechender MapReduce-Implementierungen garantieren

ID	Entity	Blocking Keys
1		Console, Sony
2		Console, Sony
3		Console, Nintendo
4		Console, Nintendo
5		Phone, Sony
6		Phone, Sony
7		Phone, Samsung

Abbildung 3.5: Beispiel redundanter Paarvergleiche für ein zweistufiges Clustering von Produkt-Datensätzen (Standard-Blocking). Die Datensätze werden anhand ihres Produkttyps (grau schattiert) *und* ihres Herstellers (schwarz gerahmt) gruppiert.

zu können. Die im Abschnitt 2.3.3 vorgestellten Ansätze zur Behandlung des *Data Skew*-Problems sind genereller Natur und nicht auf das Entity Resolution-Problem zugeschnitten. Da diese Ansätze *keine* Aufteilung großer reduce-Eingabe-Schlüsselgruppen vorsehen, versprechen sie lediglich eine Abmilderung der Skew-Effekte, jedoch keine Lösung des zugrunde liegenden Problems. Da die Ähnlichkeitsberechnung zweier Datensätze unabhängig von anderen Datensätzen ist, erlaubt das Entity Resolution-Problem eine beliebige Aufteilung der Datensätze einer großen Schlüsselgruppe auf die zur Verfügung stehenden Ressourcen, solange sichergestellt ist, dass jedes Datensatzpaar der ursprünglichen Schlüsselgruppe bearbeitet wird. Dazu ist es u. U. erforderlich, einzelne Datensätze zu replizieren und mehreren Reduce-Tasks zuzuweisen. Dedoops Techniken zur Lastbalancierung basieren auf einem leichtgewichtigen zusätzlichen Analyse-Job, der vor dem Matching-Job ausgeführt wird und die zugrundeliegende Datenverteilung analysiert. In der Map-Phase des Matching-Jobs wird mithilfe gesammelten globalen Wissens die Umverteilung der Datensätze zu den Reduce-Tasks angepasst, um entsprechend des Blocking-Schemas die Ähnlichkeit aller Kandidatenpaare zu berechnen und gleichzeitig eine gleichmäßige Auslastung der Reduce-Tasks zu gewährleisten. Die für die einzelnen Blocking-Verfahren implementierten Lastbalancierungsalgorithmen werden in den Kapiteln 4 bis 6 im Detail vorgestellt.

Um auch für "schmutzige", automatisch extrahierte Webdaten eine hohe Match-Qualität gewährleisten zu können, verfolgen Entity Resolution-Ansätze die Strategie, für jeden Datensatz mehrere Blockschlüssel von verschiedenen Attributen abzuleiten und damit einen Datensatz logisch mehreren Blöcken zuzuordnen (vgl. [25, 163]). Dies führt dazu, dass sowohl *Matches* als auch *Non-Matches* mehrfach bezüglich verschiedener Blockschlüssel miteinander verglichen werden. Da hierbei stets *dieselben* Ähnlichkeitsmetriken berechnet werden und *dieselbe* Klassifikationsentscheidung getroffen wird, ist die mehrfache Ähnlichkeitsberechnung eines Datensatzpaares überflüssig und führt zu Duplikaten im Match-Ergebnis. Im Beispiel von Abbildung 3.5 werden Produktdatensätze zunächst nach ihrem Produkttyp und anschließend nach ihrem Hersteller gruppiert. Die Gruppierung anhand des Produkttyps ist nicht ausreichend, erst die Hinzunahme des Herstellerattributes ermöglicht, die Datensätze 2 und 6 als Duplikate zu klassifizieren. Hierbei handelt es sich um ein Smartphone für Computerspieler, das in verschiedenen Ausgangsdatenquellen einmal der Kategorie *Console* und einmal der Kategorie *Phone* zugeordnet wurde. Die Mehrfachzuweisung von Datensätzen zu Blöcken führt im Beispiel jedoch dazu, dass drei von insgesamt 16 Paarvergleichen mehrfach durchgeführt werden. Um dies zu vermeiden, ist es erforderlich, dass jedes Datensatzpaar nur ein einziges Mal bezüglich *eines* gemeinsamen Blockschlüssels verglichen wird. Die Gewährleistung dieser Bedingung in einer verteilten MapReduce-Umgebung ist schwierig, da die Datensätze verteilt vorliegen, Paarvergleiche unabhängig voneinander von verschiedenen Knoten durchgeführt werden und das MapReduce-Programmiermodell keine Kommunikation zwischen verschiedenen (Reduce-) Tasks vorsieht. Deswegen hat ein Reduce-Task keine Kenntnis darüber, ob ein anderer Reduce-Task dasselbe Datensatzpaar möglicherweise ebenfalls verarbeitet. Für eine Lösung dieser Problematik gibt es grundsätzlich zwei Lösungsansätze. In [163] und [193] wurde die Idee der Erzeugung einer Menge semantisch äquivalenter Blöcke, die keine redundanten Paarvergleiche mehr enthalten, verfolgt. Dabei bildet im Prinzip jedes nicht-redundante Datensatzpaar einen

eigenen Block, was zu einer sehr hohen Anzahl an Blöcken mit einer quadratischen Speicherkomplexität führt. Ein eleganterer (in Dedoop verfolgter) Ansatz ist die Annotation jedes Datensatzes mit *allen* seinen Blockschlüsseln in der Map-Phase und die Verwendung einer Funktion, die für ein beliebiges Datensatzpaar deterministisch einen Blockschlüssel aus der Schnittmenge der beiden Blockschlüsselmengen auswählt. Somit kann jeder Reduce-Task zur Laufzeit prüfen, ob ein Datensatzpaar bezüglich des aktuell betrachteten Blockschlüssels zu vergleichen ist oder ob ein anderer Reduce-Task dafür "zuständig" ist. Verschiedene Varianten zur effizienten Umsetzung dieser Idee und deren Einfluss auf die Lastbalancierung werden in Kapitel 8 betrachtet.

3.5 Zusammenfassung und Abgrenzung der eigenen Arbeit

Die Berechnung von Entity Resolution-Workflows ist ein sehr zeitintensiver Prozess. Um auch für große Datenquellen akzeptable Antwortzeiten gewährleisten zu können, ist neben dem traditionellen Ansatz, die zu vergleichenden Datensatzpaare durch die Verwendung von Blocking-Verfahren einzuschränken, *zusätzlich* eine Parallelverarbeitung erforderlich. In den vergangenen Jahren etablierte sich das Geschäftsmodell Infrastructure as a Service, welches eine bedarfsorientierte Akquirierung, Freigabe und Abrechnung von IT-Infrastruktur ermöglicht. Gleichzeitig entstand mit Apache Hadoop ein quelloffenes MapReduce-Framework, welches eine automatische Parallelisierung von daten- und rechenintensiven Berechnungen in Clusterumgebungen erlaubt. Die vorliegende Dissertation untersucht Methoden zur Berechnung aller Teilschritte eines Entity Resolution-Workflows mit MapReduce. Der Fokus liegt auf der MapReduce-Portierung existierender Entity Resolution-Techniken und der Vorstellung von Algorithmen zur Lösung von Problemen, die durch die Parallelverarbeitung entstehen. Als Ergebnis wird ein Framework zur automatischen und effizienten Parallelisierung nutzerdefinierter Entity Resolution-Workflows mit MapReduce vorgestellt, welches alle vorgestellten Verfahren in einem nutzerfreundlichen System namens *Dedoop* zusammenfasst.

Aufgrund des Stapelverarbeitungscharakters des MapReduce-Programmmierparadigmas eignet sich dieses nicht für interaktive Teilschritte eines Entity Resolution-Workflows mit Nutzerfeedback. Aus diesem Grund werden in dieser Arbeit keine interaktiven Techniken, wie z. B. Active Learning, betrachtet. Des Weiteren wird angenommen, dass die Ähnlichkeitsberechnung eines Datensatzpaares unabhängig von anderen Datensätzen erfolgen kann. Dies schliesst die Parallelisierung kontextbasierter Entity Resolution-Verfahren weitesgehend aus, sofern der relevante Kontext eines Datensatzes nicht direkt in diesem kodiert werden kann (dieser Ansatz wurde z. B. in [93] verfolgt). Aus der Fokussierung auf das MapReduce-Programmiermodell leitet sich *nicht* die Aussage ab, dass das MapReduce-Paradigma die *beste* Variante zur Parallelisierung von Entity Resolution-Workflows ist. Ähnliche Frameworks zur parallelen Datenverarbeitung, wie z. B. *Nephele* und *Dryad*, erlauben eine flexiblere Komposition von Datenverarbeitungsschritten, was eine Vereinfachung und Beschleunigung komplexer Workflows verspricht. Die Entscheidung, Apache Hadoop zu verwenden, gründet sich auf dessen freie Verfügbarkeit, den breiten Einsatz in verschiedenen Forschungsdomänen, die Bewährung im Produktivbetrieb vieler Unternehmen und die ständige Weiterentwicklung, die u. a. von großen Firmen, wie z. B. *Yahoo Inc.*, vorangetrieben wird. Es ist bekannt, dass das MapReduce-Modell zur Parallelisierung von Algorithmen bestimmter Problemklassen (z. B. iterative oder kommunikationsintensive Algorithmen) nur bedingt geeignet ist. Da jedoch die Ähnlichkeitsberechnung der die Gesamtausführungszeit dominierende Teil eines Entity Resolution-Workflows ist und die Ähnlichkeitsberechnung (abgesehen von kontextbasierten Verfahren) eines Datensatzpaares unabhängig von anderen Datensätzen erfolgen kann, bietet sich eine MapReduce-Parallelisierung unter Ausnutzung von Datenparallelität an. Die neuste Generation des Hadoop-Frameworks beinhaltet zudem mit *YARN* eine Abstraktionsschicht zur Verwaltung von Cluster-Ressourcen, die prinzipiell die Bearbeitung verschiedener Teilprobleme durch unterschiedliche Programmiermodelle innerhalb *desselben* Clusters erlaubt.

Tabelle 3.1 vergleicht Dedoop mit den wichtigsten bestehenden Ansätzen zur Parallelisierung von

Entity Resolution-Workflows (vgl. Abschnitt 2.4). Dabei wird entsprechend den aus den jeweiligen Veröffentlichungen zu entnehmenden Informationen eine Kategorisierung hinsichtlich folgender Kriterien vorgenommen:

- Auf Basis welcher Infrastruktur, welchen Programmmierparadigmas oder welchen Frameworks erfolgte die Parallelisierung?
- Unterstützt der Ansatz die neben Duplikaterkennung innerhalb einer einzelnen (möglicherweise integrierten) Datenquelle auch *direkt* die Duplikaterkennung zwischen zwei Datenquellen? Dies bedingt, dass *keine* Ähnlichkeitsberechnung zwischen Datensätzen derselben Datenquelle erfolgt.
- Unterstützt der Ansatz die parallele Auswertung des Kartesischen Produktes? Werden Blocking-Verfahren, wie Standard Blocking oder Sorted Neighborhood, unterstützt, um die Menge der Kandidatenpaare einzuschränken?
- Wird im Falle variierender Blockgrößen eine gleichmäßige Aufteilung der durchzuführenden Paarvergleiche auf die zur Verfügung stehenden Rechenressourcen zugesichert? Hierbei wird verlangt, dass die Algorithmen zur Lastbalancierung für beliebige nutzerdefinierte Entity Resolution-Workflows anwendbar sind und *nicht* die Anpassung der Blocking-Kriterien verfolgen.
- Garantiert der Ansatz, dass (z. B. bei Mehrfachzuweisung von Datensätzen zu Blöcken) keine redundanten Paarvergleiche durchgeführt werden?
- Unterstützt der Ansatz die Generierung von Modellen zur automatischen Klassifikation der Kandidatenpaare mittels maschineller Lernverfahren?
- Ist die Möglichkeit zur Parallelisierung der Berechnung der transitiven Hülle des ermittelten Match-Ergebnisses gegeben?
- Erlaubt eine graphische Benutzeroberfläche die komfortable Definition und Ausführung von Entity Resolution-Workflows?

Das in dieser Arbeit vorgestellte Entity Resolution-Framework ist der einzige bekannte Ansatz, der alle Kriterien vollständig erfüllt. Insbesondere die Techniken zur Lastbalancierung und zur Vermeidung redundanter Paarvergleiche sind (weitesgehend) Alleinstellungsmerkmale, die jedoch Grundvoraussetzung sind, um eine effiziente Parallelverarbeitung in verteilten Umgebungen und eine Skalierbarkeit für große Datenmengen zu ermöglichen. Im Gegensatz zu den im Abschnitt 2.3.3 vorgestellten allgemeinen Ansätzen zur Lastbalancierung beliebiger MapReduce-Programme wird sich der Umstand zu Nutze gemacht, dass die Ähnlichkeitsberechnung eines Datensatzpaares unabhängig von anderen Datensätzen erfolgen kann. Dies erlaubt eine beliebig feingranulare Aufteilung der Datensatzpaare eines Blockes (einer Schlüsselgruppe) auf Prozessoren (Reduce-Tasks), solange gewährleistet ist, dass jedes nach dem Blocking-Schritt verbleibende Kandidatenpaar *irgendwo* verglichen wird. Dadurch ist die unterste Schranke der Gesamtausführungszeit zur Bearbeitung des Entity Resolution-Workflows in einem beliebig großen MapReduce-Cluster *nicht* durch die Zeit, die zur Bearbeitung des größten Blocks benötigt wird, beschränkt. Eine weiteres Alleinstellungsmerkmal ist die Integration des Infrastructure as a Service-Dienstes Amazon EC2, die die vollautomatische Akquirierung und Freigabe benötigter Rechenressourcen sowie die automatische Einrichtung und Parametrisierung eines MapReduce-Clusters auf den gemieteten Instanzen erlaubt.

Da mit einer steigenden Anzahl verwendeter Attributwerte und Ähnlichkeitsmaße eine manuelle Festlegung einer qualitativ hochwertigen Strategie zur Aggregation der berechneten Ähnlichkeitswerte kaum mehr handhabbar ist, unterstützt Dedoop sowohl das maschinelle Lernen als auch die parallele Anwendung von Modellen zur qualitativ hochwertigen Klassifikation. Ein typischer Nachverarbeitungsschritt von Entity Resolution-Workflows ist die Berechnung der transitiven Hülle der ermittelten Duplikate zur Erkennung weiterer Duplikate, die (z. B. aufgrund des Blocking-Schrittes) nicht als *Match* klassifiziert wurden. Dazu wird ein Match-Ergebnis als ungerichteter Graph interpretiert und die Menge aller maximalen Subgraphen gesucht, in denen jedes Knotenpaar durch einen Pfad verbunden ist. Da für Graphen

Ansatz/Tool/ Framework	Infrastruktur	Ein-/Zwei-Quellen-Fall	Kart. Prod./ Blocking	Lastbalancierung	Vermeidung redund. Paarvergleiche	Maschinelle Lernverfahren	Trans. Hülle	GUI
D-Swoosh [22], 2007	Verteilte Java-Anwendung	✓/ ✗	✓/ ✓	✗	✗	✗	✗	✗
Parallel Linkage [219], 2007	MATLAB	✗/ ✓	✓/ ✗			✗	✗	✗
GOMMA [126], 2010	Java RMI	✓/ ✓	✓/ ✓	✓	✗	✓	✗	✗
MD-Approach [25], 2011	Phoenix	✓/ ✗	✗/ ✓	Anpassung der Blockingstrategie an Datenverteilung	✗	✗	✗	✗
PPJoin+@MR [234], 2010	MapReduce	✓/ ✓	✓/ Implizit	Implizit (seltene Präfixtoken)	✓[9]	✗		✗
MapDupReducer [236], 2010	MapReduce	✓/ ✗	✓/ Implizit	Implizit (seltene Präfixtoken)	✗	✗	✗	✓
Dynamic Record Blocking [163], 2012	MapReduce	✓/ ✗	✗/ ✓	Anpassung der Blockingstrategie an Datenverteilung	✓	✗	✗	✗
Dedoop [132], 2012	MapReduce	✓/ ✓	✓/ ✓	✓	✓	✓	✓	✓
Entity Resolution @GPU [83], 2013	GPU	✓/ ✗	✓/ ✓		✗	✗	✓	✗
Ontology Matching @GPU [101], 2013	GPU	✗/ ✓	✓/ ✗			✗	✗	✗

Tabelle 3.1: Zusammenfassende Klassifikation existierender Ansätze zur Parallelisierung von Entity Resolution-Workflows.

[9] Der ursprüngliche Quellcode wurde entsprechend dem in dieser Arbeit vorgestellten Ansatz angepasst. Die Autoren von [234] veröffentlichten einen bereitgestellten Patch auf deren Projektseite `http://asterix.ics.uci.edu/fuzzyjoin`.

mit Hunderttausenden oder gar Millionen von Kanten eine sequentielle Berechnung nicht handhabbar ist, unterstützt Dedoop als einziger Ansatz neben [83] die parallele Berechnung der transitiven Hülle der ermittelten Duplikatmenge.

Teil II

Paralleles Blocking und Lastbalancierung

4
Standard Blocking

Dieses Kapitel stellt nach einer kurzen Vorbetrachtung im Abschnitt 4.1 die Lastbalancierungsalgorithmen BlockSplit (Abschnitt 4.2) und PairRange (Abschnitt 4.3) für die effiziente Parallelisierung von Entity Resolution-Workflows vor. Beide Ansätze realisieren das Standard Blocking-Verfahren zur Reduzierung des Suchraums auf Grundlage des MapReduce-Programmiermodells. Dabei wird die Strategie verfolgt, die Datenverteilung zunächst in einem Vorverarbeitungsschritt zu analysieren, um anschliessend eine gleichmäßige Zuweisung von Datensatzpaaren zu Reduce-Tasks realisieren zu können. Die Notwendigkeit von Verfahren zur Lastbalancierung und die Effizienz der vorgeschlagenen Algorithmen wird im Abschnitt 4.4 anhand eines Realweltdatensatzes in einer echten Cloud-Umgebung bestehend aus bis zu 100 Knoten demonstriert. Abschnitt 4.5 betrachtet die Anpassung beider Verfahren für die Duplikaterkennung in zwei Datenquellen.

4.1 Überblick

Standard Blocking bezeichnet die Gruppierung von Datensätzen, die eine gewisse Mindestähnlichkeit aufweisen, anhand sogenannter Blockschlüssel. Anschliessend wird eine Reduzierung des Suchraums vorgenommen, indem die Ähnlichkeitsberechnung auf Datensätze derselben Gruppe (Block) beschränkt wird. Ziel dieses Verfahrens ist es, Vergleiche von unähnlichen Datensätzen, die höchstwahrscheinlich keine Duplikate sind, zu vermeiden und dadurch den Prozess der Ähnlichkeitsberechnung zu beschleunigen. Das MapReduce-Programmiermodell ist intuitiv gut geeignet, um diesen Blocking-Schritt und v. a. die nachfolgende Ähnlichkeitsberechnung zu parallelisieren. Map-Tasks können die verteilt gespeicherten Daten parallel einlesen und auf Basis berechneter Blockschlüssel zu den Reduce-Tasks umverteilen. Dabei werden alle Datensätze eines Blocks demselben Reduce-Tasks zugewiesen, sodass die verschiedenen Blöcke parallel durch verschiedene Reduce-Tasks bearbeitet werden können. Dieses Prinzip lässt sich problemlos auf Blocking-Verfahren, wie z. B. Q-gram Indexing oder Suffix Array Indexing, übertragen, die sich vom Standard Blocking lediglich in der Art und Weise der Schlüsselgenerierung unterscheiden. Im weiteren Verlauf dieses Kapitels wird der naive Basisansatz als Basic-Strategie bezeichnet. Zu Illustrationszwecken wird im Folgenden ein fortlaufendes Beispiel mit 14 Datensätzen und vier Blockschlüsseln verwendet, das in Abbildung 4.1 dargestellt ist.

Wie in Abbildung 4.2 dargestellt, ist die Basic-Strategie hochgradig anfällig gegenüber Datenungleichverteilungen. Da die Anzahl der Paarvergleiche innerhalb eines Blocks quadratisch mit dessen Größe wächst, sind stark variierende Ausführungszeiten bei der Bearbeitung der einzelnen Reduce-Tasks zu beobachten. Dies führt in der Praxis dazu, dass die Bearbeitung einiger weniger Reduce-Tasks die Gesamtausführungszeit des MapReduce-Jobs dominiert und im Falle gemieteter Infrastruktur keine Kosteneffizienz gewährleistet werden kann, da instanziierte, aber nicht ausgelastete Knoten trotzdem Kosten verursachen. Bei der Bearbeitung großer Blöcke kann es zudem zu Speicherengpässen kommen, da der bearbeitende Reduce-Tasks alle Werte (Datensätze) eines reduce-Aufrufs im Hauptspeicher halten muss oder auf Externspeicher zurückgreifen muss (vgl. [234] bzw. Abschnitt 2.3.2). Im Beispiel von Abbildung 4.2 muss der erste Reduce-Task 16 von insgesamt 20 Paarvergleichen durchführen, was 80% der kompletten Workload entspricht. Im günstigsten Fall hätte die verwendete Hashfunktion alle

Partition	Π_0							Π_1						
Entity	A	B	C	D	E	F	G	H	I	K	L	M	N	O
Blocking Key	w	w	x	x	x	z	z	w	w	y	y	z	z	z

Abbildung 4.1: Beispiel-Datenquelle bestehend aus 14 Datensätzen *A*–*O*, aufgeteilt in zwei Eingabepartitionen Π_0 und Π_1.

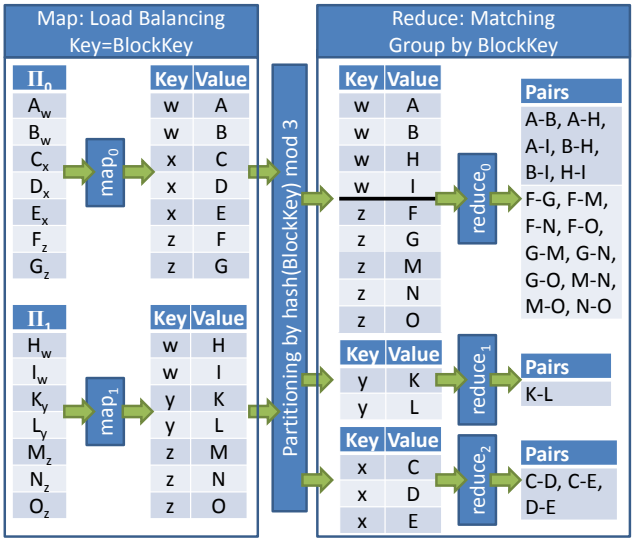


Abbildung 4.2: Vereinfachter Datenfluss der Basic-Strategie für die Beispiel-Datenquelle aus Abbildung 4.1 für und $r = 3$.

Datensätze mit dem Blockschlüssel *w* zum zweiten Reduce-Task (reduce_1) zugewiesen, was zu einer gleichmäßigeren Auslastung der drei Reduce-Tasks (10, 7 und 3) führt. Allerdings bestünde noch immer das grundsätzliche Problem, dass die zur Bearbeitung des größten Blocks benötigte Zeit eine untere Schranke für die (trotz eines beliebig hohen Parallelitätsgrades) erreichbare Gesamtausführungszeit darstellt. Ein Domänenexperte könnte prinzipiell durch Verfeinerung des Blocking-Kriteriums die Blockschlüsselgenerierung anpassen, um eine Generierung von Blöcken vergleichbarer Größe zu erzwingen. Dieser Ansatz wurde u. a. in [25] und [163] verfolgt, hat jedoch den Nachteil, dass Duplikate verschiedenen Blöcken zugewiesen werden können, was den Recall verringert. Darüber hinaus erfordert ein solches Vorgehen ebenfalls eine Vorab-Datenanalyse, um die Kenntnisse über die Blockgrößenverteilung zu erlangen. In diesem und nächsten Kapitel steht deswegen die effiziente Umsetzung einer gegebenen Blocking-Konfiguration im Mittelpunkt. Im Folgenden soll zunächst die Lastbalancierung für die Duplikaterkennung innerhalb einer Datenquelle *R* betrachtet werden. Des Weiteren wird angenommen, dass jedem Datensatz ein gültiger Blockschlüssel zugewiesen wird. Eine die Behandlung fehlender Attributwerte umfassende Verallgemeinerung analog zu [126] ist vergleichsweise einfach. Neben der Ähnlichkeitsberechnung aller Datensätze eines Blocks müssen die Datensätze $R_\emptyset \subseteq R$ ohne Blockschlüssel mit *allen* Datensätzen aus *R* verglichen werden. Dies ist gleichbedeutend mit der Auswertung des Kartesischen Produktes $R \times R_\emptyset$, was ein Spezialfall des Entity Resolution zwischen zwei Datenquellen ist. Die Erweiterung der Lastbalancierung für das Matching zweier Datenquellen wird im Abschnitt 4.5 thematisiert.

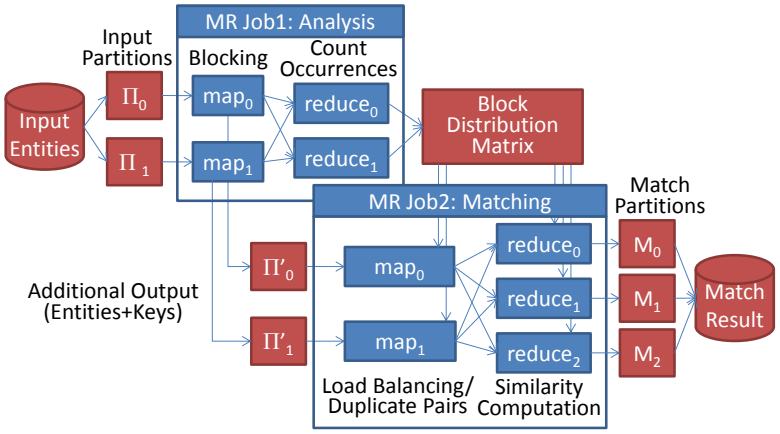


Abbildung 4.3: Allgemeines Lastbalancierungsschema für die Parallelisierung von Entity Resolution-Workflows mit MapReduce.

4.1.1 Allgemeines Lastbalancierungsschema

Die Realisierung beider in diesem Kapitel vorgestellten Lastbalancierungsstrategien basiert, wie in Abbildung 4.3 dargestellt, auf der sequentiellen Ausführung zweier MapReduce-Jobs mit einer identischen Partitionierung der Eingabedaten und derselben Anzahl an Map-Tasks. Der zuerst ausgeführte Datenanalyse-Job berechnet eine sogenannte *Block Distribution Matrix* (BDM), welche die Anzahl der Datensätze pro Block und Eingabepartition enthält und damit die zugrunde liegende Datenverteilung vollständig abbildet. Darüber hinaus ermöglichen die darin enthaltenen Informationen eine einfache Nummerierung der Blöcke, der Datensätze eines Blocks und sogar der zu vergleichenden Datensatzpaare über alle Tasks hinweg. Die BDM wird mithilfe des Distributed Caches allen Tasks des nachfolgenden MapReduce-Jobs als zusätzliche Eingabe verfügbar gemacht und von diesen während der Verarbeitung ausgewertet, um eine gleichmäßige Zuweisung von Paarvergleichen zu Reduce-Tasks zu garantieren. Prinzipiell stimmt die reguläre Eingabe des zweiten MapReduce-Jobs mit der des Analyse-Jobs überein. Um ein erneutes Parsen der Eingabedaten und eine redundante Berechnung der Blockschlüssel zu vermeiden, schreiben die Map-Tasks des Analyse-Jobs die Eingabedatensätze (mitsamt der berechneten Blockschlüssel) als zusätzliche Ausgabe in binärer Form in das verteilte Dateisystem. Dabei wird eine Projektion der Eingabedatensätze vorgenommen – es werden lediglich die zur Ähnlichkeitsberechnung benötigten Attribute ausgeschrieben. Diese zusätzliche Ausgabe des Analyse-Jobs dient als Eingabe des zweiten MapReduce-Jobs.

Die Lastbalancierung wird im Wesentlichen in der Map-Phase des Matching-Jobs realisiert. Ohne Eingriffe in den Quellcode des MapReduce-Frameworks beschränken sich Möglichkeiten zur Realisierung von Lastbalancierungsverfahren auf eine geeignete Konstruktion der map-Ausgabe-Schlüssel und -Werte sowie auf Definition nutzerdefinierter Funktionen zur Steuerung der Datenpartitionierung, -sortierung und -gruppierung. Aus diesem Grund generieren (im Gegensatz zur Basic-Strategie) beide Lastbalancierungsalgorithmen sorgfältig konstruierte, aus mehreren Komponenten *zusammengesetzte* Schlüssel, die im Zusammenspiel mit entsprechenden Partitionierungs- und Gruppierungsfunktionen (vgl. Abschnitt 2.3.2) eine gleichmäßige Lastverteilung realisieren. Der zusammengesetzte Schlüssel eines map-Ausgabe-Schlüssel-Wert-Paares kombiniert dabei Informationen über den Ziel-Reduce-Task eines Datensatzes, den Block, dem der Datensatz zugewiesen wurde, und den Datensatz selber. Dahinter verbirgt sich die Idee, dass die Partitionierungsfunktion lediglich einen bestimmten Teil der Schlüssel für das Routing zu den Reduce-Tasks berücksichtigt, wohingegen zur Gruppierung und Sortierung von Schlüssel-Wert-Paaren jeweils weitere Bestandteile des Schlüssels herangezogen werden. Des Weiteren kann die map-Funktion für einen einzelnen Datensatz mehrere Schlüssel-Wert-Paare generieren,

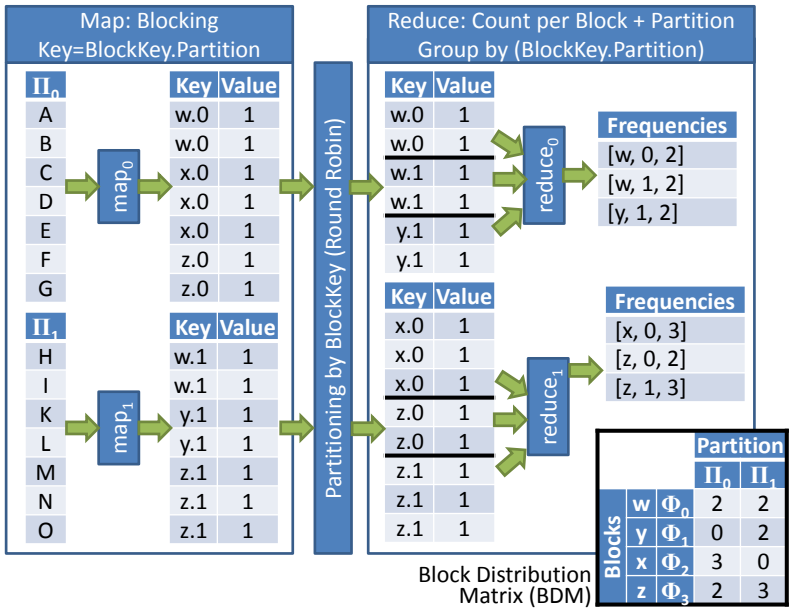


Abbildung 4.4: Datenfluss der Berechnung der Block Distribution Matrix für die Beispiel-Datenquelle aus Abbildung 4.1.

sofern dieser zu Lastbalancierungszwecken an mehrere Reduce-Tasks umverteilt werden muss. Da die Ähnlichkeitsberechnung in der Reduce-Phase unseren Experimenten zufolge über 95% der Gesamtausführungszeit eines Entity Resolution-Workflows ausmacht, betrachten beide Lastbalancierungsstrategien ausschließlich die Behandlung des Data Skews. Andere Optimierungstechniken für MapReduce-Programme wie, z. B. die Berücksichtigung von Datenlokalität [115], wurden nicht betrachtet, da kein nennenswerter Einfluss auf die Gesamtausführungszeit zu erwarten ist.

4.1.2 Berechnung der BDM

Sei b die Anzahl der Blöcke und m die Anzahl der Eingabepartitionen. Die BDM ist eine $b \times m$-Matrix, die die Anzahl an Datensätzen pro Block und Eingabepartition enthält. Zur Berechnung der BDM lesen m Map-Tasks die partitionierten Eingabedaten parallel ein und bestimmen den (oder die) Blockschlüssel eines jeden Datensatzes. Die map-Funktion gibt für jeden Datensatz ein Schlüssel-Wert-Paar mit einem zusammengesetzten Schlüssel (blocking_key⊙partition_index) und dem Wert 1 aus. Die map-Ausgabe-Paare werden durch Anwendung einer Partitionierungsfunktion part, die lediglich die Blockschlüssel-Komponente der zusammengesetzten Schlüssel berücksichtigt, zu den Reduce-Tasks umverteilt, sodass alle Datensätze eines Blocks vom selben Reduce-Task verarbeitet werden. Die Reduce-Tasks sortieren und gruppieren die eingehenden Schlüssel-Wert-Paare anhand des vollständigen Schlüssels und summieren in jedem Aufruf der reduce-Funktion die Anzahl der Blockschlüssel (Datensätze eines Blocks) pro Eingabepartition. Die reduce-Funktion gibt dementsprechend Tripel der Form [blocking_key, partition_index, num_entities] aus. Abbildung 4.4 zeigt den Datenfluss der BDM-Berechnung für das Beispiel aus Abbildung 4.1. Beispielsweise gibt die map-Funktion für den Datensatz M ein Paar mit dem Schlüssel $(z.1)$ aus, da M der Blockschlüssel z zugewiesen wurde und sich M in der zweiten Eingabepartition (partition_index=1) befindet. Dieses Paar wird dem zweiten Reduce-Task (task_index=1) zugewiesen, der drei Paare $(z.1, 1)$ gruppiert und der reduce-Funktion übergibt. Diese gibt $[z, 1, 3]$ aus, da sich insgesamt drei Datensätze mit dem Blockschlüssel z in der zweiten Eingabepartition befinden.

Die logisch konkatenierte Ausgabe der einzelnen Reduce-Tasks lässt sich als BDM interpretieren (siehe

Algorithmus 4.1: Berechnung der BDM

```
1  map_configure(m, r, partitionIndex)
2      // Store partitionIndex

3  map(k_in=unused, v_in=entity)
4      blockingKey = computeKey(entity);
5      additionalOutput (k=blockingKey, v=entity) ;                    // write to DFS
6      output (k_tmp =blockingKey.partitionIndex, v_tmp=1); ;      // regular map output

7  // part: repartition map output by blockingKey
8  // cmp: sort by blockingKey.partitionIndex
9  // group: group by blockingKey.partitionIndex
10 reduce(k_tmp=blockingKey.partitionIndex, list(v_tmp)=list(1))
11     sum ← 0;
12     foreach number in list(v_tmp) do
13         sum ← sum+number;
14     output (k_out=unused, v_out=blockingKey +","+partitionIndex +","+sum);
```

Abbildung 4.4, rechts unten). Eine beliebige Zeile der $r = 2$ Ausgabedateien entspricht einer Zelle der BDM-Matrix mit einem Wert größer 0. Adjazente Zeilen mit dem gleichen Blockschlüssel (z. B. $[w, 0, 2]$ und $[w, 1, 2]$) formen eine Zeile der BDM. Gibt es für einen Blockschlüssel in der Eingabedatenquelle nur einen einzigen Datensatz, so kann auf das Ausschreiben der entsprechenden BDM-Zelle verzichtet werden, da der Datensatz mit keinem anderen Datensatz bezüglich dieses Blockschlüssels abgeglichen werden muss. Dies dient zur Verringerung des Speicherbedarfs und zur Beschleunigung der Analyse der BDM im folgenden MapReduce-Job. Zusätzlich wird eine virtuelle Nummerierung der Blöcke entsprechend der Zeilennummern in der BDM vorgenommen, beispielsweise erhält Block w den Index 0. Im Beispiel variiert die Größe der Blöcke zwischen zwei und fünf Datensätzen für die Blöcke Φ_1 (Blockschlüssel y) bzw. Φ_3 (Blockschlüssel z). Dementsprechend variiert die Anzahl der Paarvergleiche zwischen 1 und 10; auf den größten Block Φ_3 entfallen 50% aller Paarvergleiche, obwohl er lediglich 35% aller Eingabedatensätze umfasst.

Algorithmus 4.1 zeigt den Pseudocode der BDM-Berechnung. Die Funktion map_configure wird vom Hadoop-Framework automatisch nach der Initialisierung eines Map-Tasks (aber vor dem ersten Aufruf der map-Funktion) aufgerufen. Innerhalb der map-Funktion wird auf eine Funktion additionalOutput zurückgegriffen, die jeden Datensatz mitsamt seines Blockschlüssel in binärer Form in das verteilte Dateisystem schreibt. Die resultierenden m zusätzlichen Ausgabedateien bilden die Eingabe des darauffolgenden Matching-Jobs (vgl. Abbildung 4.3, Partitionen Π'_i). Dabei wird sichergestellt, dass der zweite MapReduce-Job mit derselben Anzahl von Map-Tasks ausführt wird. Ein Map-Task des Matching-Jobs bearbeitet dabei eine zusätzliche Ausgabedatei des vorigen Jobs (vollständig) und extrahiert die ursprüngliche Partitionsnummer aus dem Dateinamen. Da Hadoop die Zuweisung von Map-Tasks zu Tasktrackern unter Berücksichtigung von Datenlokalitätsaspekten vornimmt, kann das Einlesen der Eingabedaten des Matching-Jobs weitestgehend lokal erfolgen. Ein Schlüssel $x.y$ bezeichnet einen aus den Komponenten x und y zusammengesetzten Schlüssel. Der Vergleich von Schlüsseln erfolgt komponentenweise. Die Kommentare geben an, welche Komponenten der Schlüssel zur Umverteilung, Sortierung und Gruppierung von Schlüssel-Wert-Paaren verwendet werden.

4.2 BlockSplit: Blockorientierte Lastbalancierung

Die Grundidee der blockorientierten Lastbalancierungsstrategie BlockSplit ist die Aufteilung der Paarvergleiche "großer" Blöcke auf mehrere Reduce-Tasks. Dazu generiert BlockSplit für jeden Block einen oder mehrere sogenannter *Match-Tasks* entsprechend des folgenden Schemas:

- BlockSplit verarbeitet "kleine" Blöcke innerhalb es einzelnen Match-Tasks, analog zur Basic-Strategie.
- Die Datensätze großer Blöcke werden anhand der Eingabepartition in m Subblöcke aufgeteilt. Die resultierenden Subblöcke werden anschließend durch zwei verschiedene Typen von Match-Task verarbeitet. Jeder der m Subblöcke wird (wie auch jeder nicht aufgeteilte "kleine" Block) durch einen einzelnen Match-Task bearbeitet der die Ähnlichkeit aller Datensätze des Blocks innerhalb einer Eingabepartition berechnet. Zusätzlich wird für jedes Paar von Subblöcken ein Match-Task generiert, der das Kartesische Produkt der enthaltenen Datensätze auswertet. Auf diese Art und Weise werden alle Paarvergleiche des Ausgangsblocks auf mehrere Match-Tasks aufgeteilt, die ihrerseits verschiedenen Reduce-Tasks zur Bearbeitung zugewiesen werden können.
- BlockSplit ermittelt für jeden so generierten Match-Task die Anzahl der enthaltenen Paarvergleiche. Anschließend werden die Match-Task anhand ihrer Workload sortiert und in absteigender Reihenfolge mittels einer Greedy-Heuristik auf die r Reduce-Tasks aufgeteilt. Neben der Aufteilung der Last großer Blöcke auf mehrere Reduce-Tasks bewirkt diese Vorgehensweise, dass die größten Match-Tasks zuerst bearbeitet werden, was die Wahrscheinlichkeit verringert, dass ihre Bearbeitung das Ende der Ausführung des MapReduce-Jobs signifikant verzögert.

Die Umsetzung obiger Idee basiert auf der Auswertung der BDM und der Konstruktion zusammengesetzter `map`-Ausgabe-Schlüssel. Die `map`-Funktion gibt Schlüssel-Wert-Paare der Form (reduce_index⊙ block_index⊙split, entity) aus. Der reduce_index ist ein Wert zwischen 0 und $r-1$, der von der Partitionierungsfunktion verwendet wird um die gewünschte Zuweisung der Datensätze zu Reduce-Tasks zu realisieren. Die Sortierung und Gruppierung der Schlüssel-Wert-Paare erfolgt anhand des vollständigen Schlüssels. Dies stellt sicher, dass die `reduce`-Funktion nur für Datensätze desselben Blocks aufgerufen wird. Der Wert der split-Komponente kennzeichnet einen konkreten Match-Task, der in einem Aufruf der `reduce`-Funktion bearbeitet werden soll. Gleichzeitig signalisiert er, ob in diesem Match-Task Datensätze eines Subblocks untereinander oder Datensätze zweier Subblöcke miteinander zu vergleichen sind. Im Folgenden wird die Match-Task- und Schlüssel-Generierung im Detail beschrieben.

4.2.1 Match-Task-Generierung

Während der Initialisierung wertet jeder der m Map-Tasks die BDM aus und bestimmt mithilfe der darin codierten Informationen die Anzahl resultierender Paarvergleiche pro Block. Die Gesamtanzahl der Paarvergleiche aller b Blöcke Φ_k beträgt $P = \frac{1}{2} \cdot \Sigma_{k=0}^{b-1} |\Phi_k| \cdot (|\Phi_k| - 1)$. Anschließend wird für jeden Block Φ_k geprüft, ob die Anzahl der Paarvergleiche des Blocks über der durchschnittlichen Workload eines Reduce-Tasks liegt, also ob gilt: $\frac{1}{2} \cdot |\Phi_k| \cdot (|\Phi_k| - 1) > P/r$. Ist dies *nicht* der Fall, so wird der Block innerhalb eines einzelnen Match-Tasks $k.*$ bearbeitet. Dies wird durch `map`-Ausgabe-Schlüssel mit den block_index k und der split-Komponente $*$ ausgedrückt. Anderenfalls handelt es sich um einen "großen" Block, dessen Bearbeitung durch einen einzelnen Reduce-Task eine ungleichmäßige Auslastung des MapReduce-Clusters zur Folge hätte. Deswegen wird der Block entsprechend der Eingabepartitionen in m Subblöcke unterteilt und folgende $\frac{1}{2} \cdot m \cdot (m-1) + m$ Match-Tasks gebildet[1]:

- m Match-Tasks, gekennzeichnet durch die Schlüssel-Komponenten $k.i$, zur Ähnlichkeitsberechnung der Datensätze des i-ten Subblocks ($i \in [0, m-1]$)
- $\frac{1}{2} \cdot m \cdot (m-1)$ Match-Tasks, gekennzeichnet durch die Schlüssel-Komponenten $k.i \times j$ ($i, j \in [0, m-1]$ und $i < j$), zur Auswertung des Kartesischen Produktes der Subblöcke i und j.

[1] Da die BDM die Anzahl an Datensätzen pro (Block, Partition)-Paar enthält, lässt sich leicht ermitteln, welche der m Eingabepartitionen tatsächlich Datensätze des Blocks Φ_k enthält, sodass der Block möglicherweise in weniger als m Subblöcke unterteilt wird und dementsprechend eine geringere Anzahl an Match-Tasks resultiert. Zugunsten der Lesbarkeit wird angenommen, dass jede der m Eingabepartitionen mindestens einen Datensatz des Blocks Φ_k enthält.

Die Verwendung der bestehenden Datenpartitionierung zur Aufteilung großer Blöcke ist durch die Annahme motiviert, dass m (entspricht defaultmäßig der Anzahl der zu bearbeitenden HDFS-Blöcke) mit zunehmender Größe der Eingabedatenmengen wächst, was in einer größeren Anzahl an Match-Tasks resultiert und ein höheres Lastbalancierungspotential verspricht. Bei einer konstanten Datenmenge reduziert sich zudem mit wachsendem m der Hauptspeicherbedarf der Reduce-Tasks, da sich dadurch die Anzahl an Match-Tasks erhöht und gleichzeitig die Anzahl der Datensätze pro Match-Task abnimmt. Nach der beschriebenen Generierung der Match-Tasks werden diese absteigend nach ihrer Anzahl an Paarvergleichen sortiert. Anschließend werden die Match-Tasks in Sortierreihenfolge zu dem Reduce-Task zugewiesen, der aktuell die geringste Gesamtanzahl zugewiesener Paarvergleiche hat. Im Folgenden wird der Ziel-Reduce-Task eines Match-Tasks $k.x$ mit $R(k.x)$ bezeichnet.

4.2.2 Schlüsselgenerierung und Datenumverteilung

Nach der Match-Task-Generierung wendet der i-te Map-Task ($0 \leq i < m$) die `map`-Funktion auf jeden Eingabedatensatz e seiner Eingabepartition an. Gehört der Datensatz zu einem Block Φ_k, der nicht in Subblöcke aufgeteilt werden muss, gibt die `map`-Funktion ein (R(k.*)⊙k⊙*, e)-Paar aus. Anderenfalls wird für diesen Datensatz zunächst ein (R(k.i)⊙k⊙i, e)-Paar ausgegeben. Dies entspricht der Zuweisung des Datensatzes zum Match-Task $k.i$, der die Ähnlichkeit aller Datensätze des Blocks Φ_k in der i-ten Eingabepartition berechnet. Zusätzlich gibt die `map`-Funktion für jedes der $m-1$ Paare (i, j) mit $j \in [0, m-1]$ und $j \neq i$ ein Schlüssel-Wert-Paar (R(k.a×b)⊙k⊙a×b, e) aus, wobei gilt: $a = \min(i, j)$ und $b = \max(i, j)$. Jedes dieser $m-1$ Paare bewirkt die Zuweisung des Datensatzes des Blocks zu einem Match-Task, der alle Datensätze des Blocks Φ_k der i-ten Eingabepartition mit Datensätzen einer anderen Eingabepartition vergleicht. Ein einzelner Datensatz eines "großen" Blocks wird also zum Zwecke der Lastbalancierung m-mal repliziert und durch eine geeignete Schlüsselkonstruktion m Match-Tasks zugewiesen. Dadurch erhöht sich zwar das zwischen der Map- und Reduce-Phase umverteilte Datenvolumen, allerdings wird dadurch eine Aufteilung der resultierenden Workload auf mehrere Reduce-Tasks ermöglicht. Ein Wert eines ausgegebenen Schlüssel-Wert-Paares besteht aus einer geeigneten Repräsentation eines Datensatzes; hierbei muss sichergestellt sein, dass ein eindeutiger Kennzeichner sowie alle zur Ähnlichkeitsberechnung benötigten Attribute in der Repräsentation enthalten sind.

Der Block Φ_3 (Blockschlüssel z) der Datenquelle aus Abbildung 4.1 resultiert in $10 > 20/3$ Paarvergleichen und wird deswegen in zwei Subblöcke $\Phi_{3.0}$ und $\Phi_{3.1}$ aufgeteilt. Aus der BDM (siehe Abbildung 4.4) ist ersichtlich, dass sich zwei bzw. drei Datensätze in den Subblöcken $\Phi_{3.0}$ und $\Phi_{3.1}$ befinden. Entsprechend des beschriebenen Schemas werden daraus die drei Match-Tasks 3.0 (1 Paar), 3.0×1 (6 Paare) und 3.1 (3 Paare) generiert. Die verbleibenden Blöcke Φ_0 (Blockschlüssel w), Φ_1 (Blockschlüssel y) und Φ_2 (Blockschlüssel x) werden nicht aufgeteilt und jeweils durch einen einzelnen Match-Task $0.*$ (6 Paare), $1.*$ (1 Paar) und $2.*$ (3 Paare) bearbeitet. Nach der Sortierung ergibt sich folgende Reihenfolge der Match-Tasks: $0.*$, 3.0×1, $2.*$, 3.1, $1.*$ und 3.0. Die Match-Tasks werden mittels einer Greedy-Heuristik anschließend in Sortierreihenfolge zu den $r = 3$ Reduce-Tasks zugewiesen, sodass sich die in Abbildung 4.5 gezeigte Aufteilung ergibt. Die Replikation der fünf Datensätze des Blocks Φ_3 führt dazu, dass für die 14 Eingabedatensätze insgesamt 19 Map-Ausgabe-Paare erzeugt und *gleichmäßig* zu den drei Reduce-Tasks (7, 7 bzw. 6 Paare) umverteilt werden.

Algorithmus 4.2 zeigt den Pseudocode der BlockSplit-Strategie. In der Funktion map_configure, die vom Hadoop-Framework automatisch zum Initialisierungszeitpunkt eines jeden Map-Tasks ausgeführt wird, erfolgt ein Einlesen der BDM. Nach der Generierung der Match-Tasks kann die im Hauptspeicher gepufferte BDM verworfen werden. Zudem können vom Map-Task $0 \leq i < m$ alle Match-Tasks $k.*$ und $k.a \times b$ aus dem Hauptspeicher entfernt werden, wenn Π_i keine Datensätze des Blocks Φ_k enthält oder wenn $a \neq i \wedge b \neq i$. Im Gegensatz zur bisherigen vereinfachten Darstellung verwendet die Implementierung fünfstellige Schlüssel der Form (reduce_index⊙block_index⊙partition_i⊙partition_j⊙cur_partition). Die split-Komponente wird zur vereinfachten Analyse in der Reduce-Phase durch zwei Integer-Werte repräsentiert. Für jeden Datensatz wird zudem die Herkunftspartition als fünfte Kom-

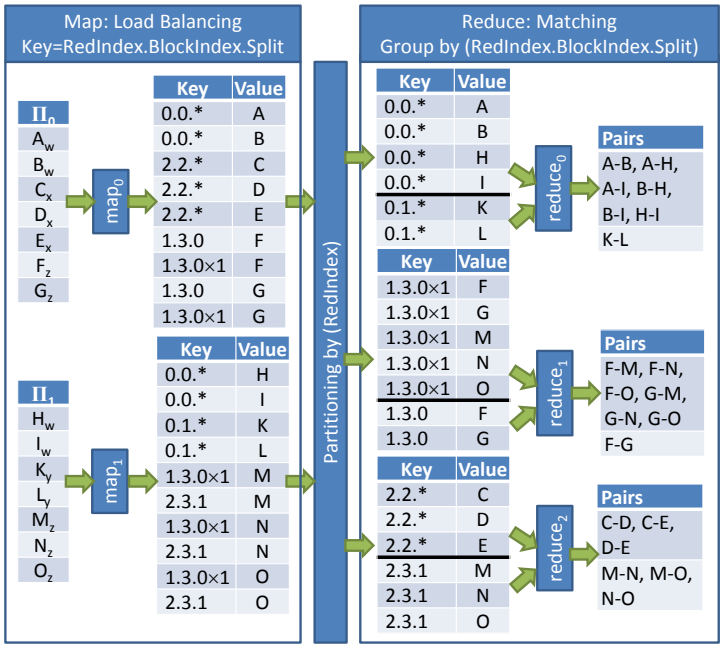


Abbildung 4.5: Vereinfachter Datenfluss der BlockSplit-Strategie für das Beispiel aus Abbildung 4.1 mit $r = 3$ Reduce-Tasks.

ponente an den `map`-Ausgabe-Schlüssel angehängt. Die Sortierung der Schlüssel-Wert-Paare erfolgt komponentenweise anhand des vollständigen Schlüssels, zur Gruppierung werden hingegen nur die ersten vier Komponenten herangezogen. Dies bewirkt, dass die Datensätze eines Match-Tasks $k.i \times j$ der `reduce`-Funktion in, nach Herkunftspartition, sortierter Reihenfolge zur Verfügung gestellt werden. Da die Information, zu welcher der beiden Partitionen ein Datensatz gehört, in der fünften Komponente des Schlüssel codiert ist, diese jedoch durch die Gruppierung anhand der ersten vier Komponenten "verloren geht", wird zusätzlich der Wert eines Schlüssel-Wert-Paares (also der Datensatz selber) um die aktuelle Eingabepartition i angereichert. Diese Modifikationen erlauben eine effizienter Verarbeitung der Datensätze eines Match-Tasks in der `reduce`-Funktion und verringern den Speicherbedarf der Reduce-Tasks.

Algorithmus 4.2: Implementierung der BlockSplit-Strategie

```
1  map_configure(m, r, partitionIndex)
2      compsPerReduceTask ← [];
3      for i ← 0 to r-1 do
4          compsPerReduceTask[i] ← 0;
5      matchTasks ← empty map;
6      compsPerReduceTask ← BDM.pairs()/r;
7      // Read BDM from reduce output of Algorithm 4.1
8      BDM ← readBDM();
9      // Match task creation
10     for k ← 0 to BDM.numBlocks()-1 do
11         comps ← ½ · BDM.size(k) · (BDM.size(k) − 1);
12         if 0 < comps ≤ compsPerReduceTask then
13             matchTasks.put((k, 0, 0), comps);
14         else if comps > 0 then
15             for i ← 0 to m-1 do
16                 |Φ_k^i| ← BDM.size(k, i);
17                 for j ← 0 to i do
18                     |Φ_k^j| ← BDM.size(k, j);
19                     if |Φ_k^i| · |Φ_k^j| > 0 then
20                         if i = j then
21                             matchTasks.put((k, i, i),
22                               ½ · |Φ_k^i| · (|Φ_k^i| − 1));
23                         else
24                             matchTasks.put((k, j, i),
                               |Φ_k^j| · |Φ_k^i|);
25     // Reduce task assignment
26     matchTasks.orderByValueDescending();
27     foreach ((k,i,j), comps) ∈ matchTasks) do
28         reduceTask ← getNextReduceTask();
29         matchTasks.put((k, i, j), reduceTask);
30         addCompsToReduceTask(reduceTask, comps);

31 // Process additional map output of Algorithm 4.1
32 map(k_in=blockingKey, v_in=entity)
33     k ← BDM.blockIndex(blockingKey);
34     comps ← ½ · BDM.size(k) · (BDM.size(k) − 1);
35     if comps ≤ compsPerReduceTask then
36         if comps>0 then
37             reduceTask ← matchTasks.get(k, 0, 0);
38             output(k_tmp=reduceTask.k.0.0.0,
39               v_tmp=(entitiy, partitionIndex));
40     else
41         for i ← 0 to m-1 do
42             min ← min{partitionIndex, i};
43             max ← max{partitionIndex, i};
44             reduceTask ← matchTasks.get(k, min, max);
45             if reduceTask ≠ null then
46                 output(k_tmp=reduceTask.k.max.min.partitionIndex,
47                   v_tmp=(entitiy, partitionIndex));

48 // part: repartition map output by reduceTask
49 // cmp: sort by entire map output key
50 // group: group by blockIndex.i.j (k.i.j)
51 reduce(k_tmp=reduceTask.k.i.j,
52   list(v_tmp)=list((entity, partitionIndex)))
53     buf ← {};
54     if i = j then
55         foreach (e_2, partitionIndex) ∈ list(v_tmp) do
56             foreach e_1 ∈ buf do
57                 match(e_1, e_2) ;      // Comparison + output
58             buf ← buf ∪ {e_2};
59     else
60         pair ← list(v_tmp).firstElement();
61         buf ← buf ∪ {pair.first()};
62         firstPartitionIndex ← pair.second();
63         foreach (e_2, partitionIndex) ∈ list(v_tmp) do
64             if partitionIndex = firstPartitionIndex then
65                 buf ← buf ∪ {e_2};
66             else
67                 foreach e_1 ∈ buf do
68                     // Comparison + output
69                     match(e_1, e_2);

70 getNextReduceTask()
71     index ← 0;
72     minValue ← compsPerReduceTask[0];
73     for i ← 1 to r-1 do
74         if compsPerReduceTask[i] <minValue then
75             index ← i;
76             minValue ← compsPerReduceTask[i];
77     return index;

78 addCompsToReduceTask(reduceTask, comparisons)
79     compsPerReduceTask[reduceTask] ← comparisons +
80       compsPerReduceTask[reduceTask];
```

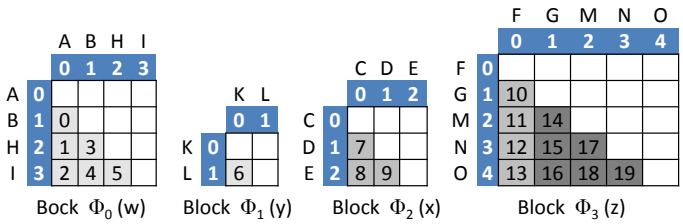


Abbildung 4.6: Globales Nummerierungsschema für das Beispiel aus Abbildung 4.1. Die drei grau schattierten Intervalle (Bereiche) illustrieren die Aufteilung der Paarvergleiche auf die $r = 3$ Reduce-Tasks.

4.3 PairRange: Paarorientierte Lastbalancierung

Die im vorigen Abschnitt vorgestellte BlockSplit-Strategie teilt große Blöcke anhand der m Eingabepartitionen in Subblöcke, die einzeln oder paarweise durch Match-Tasks verarbeitet werden. Dieser Ansatz kann bei variierenden Subblockgrößen noch immer zu unbalancierten Match- und Reduce-Tasks führen. In diesem Abschnitt wird die PairRange-Strategie vorgestellt, die diesen Schwachpunkt beseitigt und eine nahezu perfekt gleichmäßige Aufteilung aller Paarvergleiche auf die Reduce-Tasks garantiert. Die PairRange-Strategie basiert auf den folgenden beiden Ideen:

- PairRange implementiert auf Grundlage der BDM eine *virtuelle* Nummerierung aller Datensätze und deren Vergleiche (Paare). Dieses Nummerierungsschema wird verwendet, um Datensätze zu einem oder mehreren Reduce-Tasks umzuverteilen und festzulegen, welche Paare von jedem Reduce-Task bearbeitet werden.
- Um eine gleichmäßige Auslastung der Reduce-Tasks zu gewährleisten, teilt die PairRange-Strategie die Menge aller zu vergleichenden Datensatzpaare unter Verwendung des Nummerierungsschemas in r gleich große Bereiche (Intervalle) auf und weist den k-ten Bereich $\Re_k$ dem k-ten Reduce-Task zur Bearbeitung zu.

4.3.1 Schema zur virtuellen Nummerierung von Datensätze und Paaren

Jeder Map-Task verarbeitet zeilenweise die Datensätze seiner Eingabepartition. Eine Nummerierung der Datensätze pro Block und Partition ist somit einfach möglich. Ein Map-Task $0 < i < m$ ist prinzipiell jedoch nicht in der Lage, den globalen Index eines Datensatzes innerhalb seines Blocks zu bestimmen, da er kein Wissen über die Anzahl an Datensätzen des Blocks in Eingabepartitionen Π_j $(j < i)$ hat und es keine Kommunikation zwischen den Map-Tasks gibt. Erst die BDM erlaubt die lokale Berechnung des globalen blockspezifischen Indexes eines Datensatzes durch einen Map-Task. Sei Π_i die i-te Eingabepartition $(0 \leq i < m)$ und Φ_k der durch die k-ten Zeile der BDM beschriebene Block $(k \geq 0)$. Zur Bestimmung des globalen blockspezifischen Indexes eines Datensatzes des Blocks Φ_k in der Partition Π_i muss lediglich der durch Zählen ermittelte partitionsspezifische Index zur Gesamtanzahl der Datensätze des Blocks Φ_k in den Partitionen Π_0 bis Π_{i-1} addiert werden. Der Datensatz M des Beispiels aus Abbildung 4.1 ist der erste Datensatz des Blocks Φ_3 der Partition Π_1. Der BDM ist zu entnehmen, dass in den "Vorgänger-Partitionen" (hier nur Π_0) zwei weitere Datensätze aus Φ_3 enthalten sind (siehe Abbildung 4.4). Somit ist M der dritte Datensatz des Blocks Φ_3 und erhält den blockspezifischen Datensatzindex 2. Abbildung 4.6 illustriert die resultierende Nummerierung (weiße Ziffern) der Datensätze des fortlaufenden Beispiels.

Die blockspezifische Nummerierung aller Datensätze erlaubt eine Nummerierung aller Paarvergleiche. Dabei wird ein Paar (x, y) mit Datensatzindexen x und y nur nummeriert, wenn $x < y$. Paare (x, x)

werden also nicht berücksichtigt, ebenso wenig wie Paare (y,x), wenn bereits (x,y) nummeriert wurde. Dies dient der Vermeidung unnötiger Berechnungen. Die Nummerierung der Paare wird spaltenweise, fortlaufend über alle Blöcke vorgenommen. Der Paarindex $p_i(x,y)$ zweier Datensätze mit den Datensatzindexen x und y $(x<y)$ im Block Φ_i ist definiert als:

$$p_i(x,y) = c(x,y,|\Phi_i|) + o(i) \tag{4.1}$$

$$c(x,y,N) = \frac{x}{2}(2\cdot N - x - 3) + y - 1 \tag{4.2}$$

$$o(i) = \frac{1}{2}\cdot \Sigma_{k=0}^{i-1}(|\Phi_k|\cdot(|\Phi_k|-1)) \tag{4.3}$$

Hierbei bezeichnet $c(x,y,N)$ den Index der Zelle (x,y) in einer $N\times N$-Matrix und $o(i)$ die Anzahl der Paare in allen "Vorgänger-Blocks" Φ_0 bis Φ_{i-1}. Des Weiteren bezeichnet $|\Phi_i|$ die Anzahl der Datensätze des Blocks Φ_i. Abbildung 4.6 zeigt die Paarnummerierung für das fortlaufende Beispiel. Der Paarindex des Paares $p_i(x,y)$ entspricht dem Wert der durch die Spalte x und die Zeile y gegebenen Zelle des Blocks Φ_i – z. B. hat das Paar $(2,3)$ des Blocks Φ_0 den Paarindex 5.

Die PairRange-Strategie teilt die Menge aller Paare in r nahezu gleich große Bereiche (*Pair Ranges*) und weist den k-ten Bereich dem k-ten Reduce-Task zur Bearbeitung zu. Bei einer Gesamtanzahl von P Paaren, r Bereichen und r Reduce Tasks fällt ein Paar mit dem Paarindex $0\leq p<P$ in den Bereich $\mathfrak{R}_k$, wenn gilt:

$$p\in\mathfrak{R}_k \;\Leftrightarrow\; k = \lfloor r\cdot\frac{p}{P}\rfloor \tag{4.4}$$

Die ersten $r-1$ Reduce-Tasks bearbeiten demzufolge genau $\lceil\frac{P}{r}\rceil$ Paare. Der letzte Reduce-Task ist für die Bearbeitung der verbleibenden $P-(r-1)\cdot\lceil\frac{P}{r}\rceil$ zuständig. Im Beispiel von Abbildung 4.6 ergibt sich für $r=3$ eine Aufteilung von 20 Paaren in die drei Bereiche $\mathfrak{R}_0=[0,6]$, $\mathfrak{R}_1=[7,13]$ und $\mathfrak{R}_2=[14,19]$, die durch unterschiedliche Graustufen gekennzeichnet sind. Da die Anzahl der Reduce-Tasks die Anzahl der Bereiche vorgibt, kann der Parameter r verwendet werden, um die Anzahl an Datensätzen pro Bereich bzw. Reduce-Task und damit den Hauptspeicherbedarf der Reduce-Tasks zu kontrollieren.

4.3.2 Bestimmung der relevanten Bereiche eines Datensatzes

Während der Initialisierung liest jeder Map-Task $0\leq i<m$ die BDM ein, berechnet die Gesamtanzahl aller Paarvergleiche P und bestimmt die Intervallgrenzen der r Bereiche. Des Weiteren wird für jeden Block, von dem Datensätze in der Partition Π_i enthalten sind, ein Zähler initialisiert und dessen Wert auf die Anzahl der Datensätze des Blocks in Partitionen Π_j $(j<i)$ gesetzt. Nach der Initialisierung wird die `map`-Funktion auf jeden Datensatz e angewendet. Sei Φ_k der durch den Blockschlüssel des Datensatzes e definierte Block. Zunächst wird mithilfe des Zählers für Φ_k der globale Index x des Datensatzes innerhalb von Φ_k bestimmt. Anschließend wird dieser Zähler inkrementiert und die Menge aller Bereiche berechnet, die mindestens ein Paar umfassen in dem e enthalten ist.

Zur Bestimmung dieser Bereiche ist es *nicht* erforderlich, alle Paare, an denen e beteiligt ist, auszuwerten – in vielen Fällen ist es ausreichend, nur zwei Paare zu betrachten. Sei N die Anzahl der Datensätze in Φ_k. Der Datensatz e ist an den Paaren $(0,x)$, …, $(x-1,x)$, $(x,x+1)$, …, $(x,N-1)$ beteiligt. Das Nummerierungsschema erlaubt eine schnelle Identifizierung des kleinsten und größten Indexes dieser Paare. Dazu berechnet die `map`-Funktion $p_{min}=p_k(0,\ \max\{x,1\})$ und $p_{max}=p_k(\min\{x,N-2\},\ N-1)$ mithilfe der Formeln 4.1–4.3. Der Datensatz M des fortlaufenden Beispiels hat beispielsweise den Index 2 innerhalb des Blocks Φ_3 der Größe 5 – die entsprechenden Paarindexe sind $p_{min}=p_3(0,2)=11$ und $p_{max}=p_3(2,4)=18$.

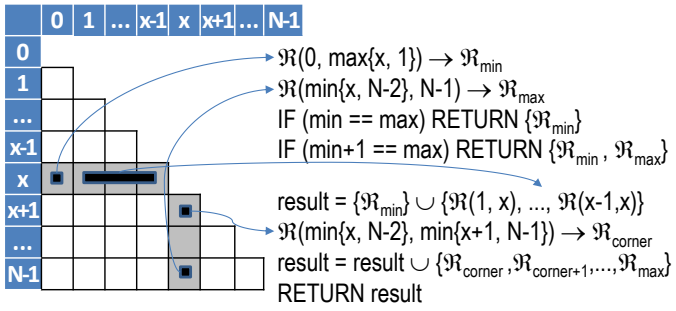


Abbildung 4.7: Schema zur Bestimmung der Bereiche, die Paare umfassen, an denen ein Datensatz mit dem blockspezifischen Index x partizipiert.

Anschließend lassen sich mithilfe der Formel 4.4 die Bereiche $\mathfrak{R}_{min} \ni p_{min}$ und $\mathfrak{R}_{max} \ni p_{max}$ der beiden Paare bestimmen. Im Falle von $\mathfrak{R}_{min} = \mathfrak{R}_{max}$ fallen alle Paare, an denen e beteiligt ist, in genau einen Bereich ($\mathfrak{R}_{min}$). Wenn $min + 1 = max$ gilt, also wenn $\mathfrak{R}_{max}$ der direkte Nachfolger von $\mathfrak{R}_{min}$ ist, partizipiert e an genau diesen beiden Bereichen, d. h. die Menge der für e relevanten Bereiche ist $\{\mathfrak{R}_{min}, \mathfrak{R}_{max}\}$. Dies trifft für den Datensatz M zu, der in den $\mathfrak{R}_2$ und $\mathfrak{R}_3$ fällt. Die Menge der Paare großer Blöcke erstrecken sich i. Allg. über mehr als zwei Bereiche ($min+1 < max$). In diesem Fall können zwischen $\mathfrak{R}_{min}$ und $\mathfrak{R}_{max}$ Bereiche liegen, in denen sich kein Paar befindet, an dem e partizipiert. Wie in Abbildung 4.7 illustriert, berechnet die `map`-Funktion in diesem Fall zunächst den Bereich eines jeden Paares $(0, \max\{x, 1\}), \ldots, (x-1, x)$ der Zeile x der $N \times N$-Matrix. Die Spalte x dieser Matrix umfasst die Paarindexe $p_{corner} = p_k(\min\{x, N-2\}, \min\{x+1, N-1\}), \ldots, p_{max}$. Da e in allen Paaren des Intervalls $[p_{corner}, p_{max}]$ enthalten ist und die entsprechenden Bereiche fortlaufende Indexe haben, ist es ausreichend $\mathfrak{R}_{corner} \in p_{corder}$ zu bestimmen und alle Bereiche zwischen $\mathfrak{R}_{corner}$ und $\mathfrak{R}_{max}$ zur Menge relevanter Bereiche hinzuzufügen.

4.3.3 Schlüsselkonstruktion und Reduce-Phase

Für jeden auf diese Weise ermittelten relevanten Bereich des Datensatzes e gibt die `map`-Funktion ein Schlüssel-Wert-Paar (range_index⊙block_index⊙entity_index, e) aus. Die Partitionierung der Daten erfolgt auf Basis des Bereichsindexes, was alle Datensätze des Bereiches $\mathfrak{R}_k$ zum k-ten Reduce-Task umverteilt. Wie auch bei der BlockSplit-Strategie werden Datensätze repliziert und durch geeignete Schlüsselkonstruktion mehreren Reduce-Tasks zugewiesen, um eine balancierte Reduce-Phase gewährleisten zu können. Die Sortierung der Schlüssel-Wert-Paare erfolgt komponentenweise anhand des vollständigen Schlüssels, zur Gruppierung werden nur die ersten beiden Komponenten des dreistelligen Schlüssels herangezogen. Ein Reduce-Task erhält nicht notwendigerweise alle Datensätze eines Blocks, sondern nur diejenigen, die an einem Paar des abgedeckten Bereiches partizipieren. Die `reduce`-Funktion wird also für eine Teilmenge der Datensätze eines Blockes aufgerufen. Sie überprüft für jedes Paar (x, y) mit Datensatzindexen $x < y$, ob es innerhalb des durch den Reduce-Task abzudeckenden Bereiches liegt. Ist dies der Fall, so wird die Ähnlichkeitsberechnung und Klassifikation des Paares vorgenommen. Da der blockspezifische Index eines jeden Datensatzes in der Schlüsselkomponente codiert ist, diese Information jedoch durch die Gruppierung nach den ersten beiden Komponenten verloren geht, wird der Wert (also der Datensatz selber) eines jeden Schlüssel-Wert-Paares um diesen Index angereichert. Dies ist notwendig, um die Paarindexe in der `reduce`-Funktion berechnen zu können.

Abbildung 4.8 illustriert die PairRange für das fortlaufende Beispiel aus Abbildung 4.1. Der Datensatz M des Blocks Φ_3 hat den blockspezifischen Datensatzindex 2 und ist an den vier Paaren (F, M), (G, M), (M, N), (M, O) mit den Paarindexen 11, 14, 17, und 18 beteiligt (siehe Nummerierungsschema in Abbildung 4.6). Die Menge aller 20 Paare ist in die drei Bereiche $[0, 6]$, $[7, 13]$ und

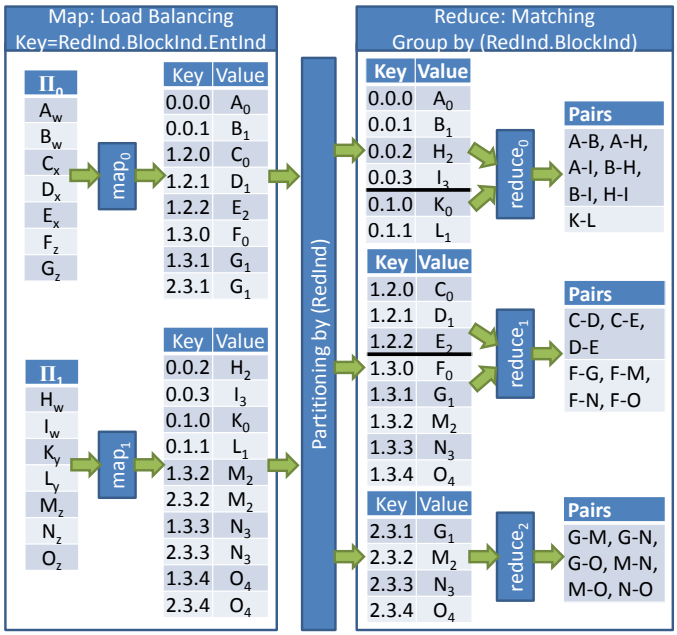


Abbildung 4.8: Vereinfachter Datenfluss der PairRange-Strategie für das Beispiel aus Abbildung 4.1 und $r = 3$ Reduce-Tasks.

[14, 19] aufgeteilt. Datensatz M muss zum zweiten Reduce-Task range_index=1 zur Evaluierung des Paars mit dem Index 11 und zum dritten Reduce-Task range_index=2 zur Evaluierung der übrigen drei Paare zugewiesen werden. Die `map`-Funktion gibt dementsprechend die beiden Schlüssel-Wert-Paare (1.3.2, M_2) und (2.3.2, M_2) aus. Neben M werden auch alle weiteren Datensätze des Blocks Φ_3 zum zweiten Reduce-Task umverteilt (F, G, N und O). Aufgrund des ihm zugewiesenen Bereiches [7, 13] bearbeitet der zweite Reduce-Tasks lediglich die Paare 7 bis 9 (des Blocks Φ_2) und 10 bis 13 (des Blocks Φ_3). Die verbleibenden Paare des Blocks Φ_3 werden vom dritten Reduce-Task bearbeitet, der bis auf F alle Datensätze von Φ_3 als Eingabe erhält. Der Datensatz F wird nicht benötigt, da er an keinem der Paare mit einem Index zwischen 14 und 19 partizipiert.

Algorithmus 4.3 zeigt den Pseudocode der PairRange-Strategie. In den Funktionen map_configure und reduce_configure, die vom Hadoop-Framework automatisch zum Initialisierungszeitpunkt eines jeden Map- bzw. Reduce-Tasks ausgeführt werden, erfolgt ein Einlesen der BDM. Im Gegensatz zur BlockSplit-Strategie ist eine temporäre Speicherung der kompletten BDM im Hauptspeicher nicht erforderlich. Es ist ausreichend, für jeden in der betreffenden Eingabepartition auftretenden Blockschlüssel die Anzahl an Datensätzen mit demselben Blockschlüssel in "Vorgänger-Partitionen" zu speichern (Algorithmus 4.3 Zeile 6–10).

In der Praxis hat es sich zudem bewährt, eine Bestimmung des Indexes eines jeden Paares in der Reduce-Phase zu vermeiden. Der Zweck dieser Indexberechnung ist die Prüfung, ob sich ein Paar innerhalb des zu bearbeitenden Bereiches befindet (Algorithmus 4.3 Zeile 42–51). Dies lässt sich auch durch zwei technische Änderungen des beschriebenen Verfahrens realisieren, die gleichzeitig ein Einlesen der BDM in der Reduce-Phase überflüssig machen. Ein ausgewählter Map-Task (z. B. derjenige, der die Eingabepartition Π_0 bearbeitet) sendet ein spezielles Schlüssel-Wert-Paar (i.-1.0, $\lfloor P/r \rfloor$) an jeden Reduce-Task $0 \leq i < r$, bevor er mit der Anwendung der `map`-Funktion auf die Datensätze der Eingabepartition beginnt. Durch die Sortierung und Gruppierung der Schlüssel-Wert-Paare ist sichergestellt, dass dieses Schlüssel-Wert-Paar die Eingabe des ersten Aufrufs der `reduce`-Funktion eines jeden Reduce-Tasks ist. Somit kennt jeder Reduce-Task die (maximale) Anzahl der über alle `reduce`-Aufrufe hinweg durchzu-

Algorithmus 4.3: Implementierung der PairRange-Strategie

```
1  map_configure(m, r, partitionIndex)
2      // Read BDM from reduce output of Algorithm 4.1
3      BDM ← readBDM();
4      compsPerReduceTask ← ⌈BDM.pairs()/r⌉;
5      // Index of next entity for each block
6      entityIndex ← [];
7      for i ← 0 to BDM.numBlocks()-1 do
8          entityIndex[i] ← 0;
9          for j ← 0 to partitionIndex-1 do
10             entityIndex[i] ← entityIndex[i]+ BDM.size(i, j)
11
12 // Process additional map output of Algorithm 4.1
13 map(k_in=blockingKey, v_in=entity)
14     ranges ← ∅;
15     i ← BDM.blockIndex(blockingKey);
16     x ← entityIndex[i];
17     N ← BDM.size(i);
18     ℜ_min ← rangeIndex(0, max{x, 1}, N, i);
19     ℜ_max ← rangeIndex(min{x, N-2}), N-1, N, i);
20     ranges ← {ℜ_min} ∪ {ℜ_max};
21     if ranges.size>2 then
22         for k ← 1 to x-1 do
23             ranges ← ranges ∪{k};
24         ℜ_corner ← rangeIndex(min{x, N-2}, min{x+1, N-1}, N, i);
25         for k ← ℜ_corner to ℜ_max-1 do
26             ranges ← ranges ∪ {k};
27     foreach r ∈ ranges do
28         output (k_tmp=r.i.x, v_tmp=(entity, x));
29     entityIndex[i] ← entityIndex[i]+1;
30
31 reduce_configure(m, r)
32     BDM ← readBDM();
33     compsPerReduceTask ← ⌈BDM.pairs()/r⌉;
34
35 // part: repartition map output by range index (r)
36 // cmp: sort by blockIndex.entityIndex (i.x)
37 // group: group by blockIndex (i)
38 reduce(k_tmp=r.i.x, list(v_tmp)=list((entity, x)))
39     N ← BDM.size(i);
40     buf ← [];
41     foreach (e, x) ∈ list(v_tmp) do
42         buf.append((e, x));
43     for j ← 0 to buf.size()-2 do
44         for k ← 1 to buf.size()-1 do
45             range ← rangeIndex (
46               buf[j].second(), buf[k].second(), N, i
47             );
48             if range=r then
49                 // Comparison + output
50                 match(buf[j].first(), buf[k].first());
51             else if range>r then
52                 return;
53             else
54                 continue;
55
56 rangeIndex(col, row, blockSize, blockIndex)
57     cellIndex ← 0.5 · col · (2·blockSize-col-3)+row-1;
58     pairIndex ← cellIndex + pairIndexOffset(blockIndex);
59     return ⌊pairIndex/compsPerReduceTask⌋;
60
61 pairIndexOffset(blockIndex)
62     sum ← 0;
63     for k ← 0 to blockIndex-1 do
64         sum ← BDM.size(k)·(BDM.size(k)-1) + sum;
65     return 0.5 · sum;
```

führenden Paarvergleiche. Wird nun bei jedem Paarvergleich ein Zähler inkrementiert, lässt sich leicht feststellen, ob das Ende des zu bearbeitenden Bereiches erreicht ist und die Ähnlichkeitsberechnung weiterer Paare vom nachfolgenden Reduce-Task durchzuführen ist (entspricht Algorithmus 4.3 Zeile 48–49). Um die Ähnlichkeitsberechnung von Datensatzpaaren zu verhindern, die in den Bereich des vorigen Reduce-Tasks fallen (entspricht Algorithmus 4.3 Zeile 50–51), muss bekannt sein, wie viele Paare am Beginn eines Bereiches zu "überspringen" sind. Dies lässt sich auf eine ähnliche Art und Weise realisieren.

4.4 Evaluation

In diesem Abschnitt werden die vorgestellten Lastbalancierungsstrategien hinsichtlich ihrer Robustheit gegenüber verschiedenen Graden der Datenungleichverteilung (Abschnitt 4.4.1), ihrer Eignung, Berechnungen auf mehrere Reduce-Tasks zu verteilen (Abschnitt 4.4.2), und hinsichtlich der Fähigkeit, mit einer wachsenden Anzahl von Knoten des MapReduce-Clusters zu skalieren (Abschnitt 4.4.3), untersucht. Dabei wird in jedem Experiment ein sinnvoller Wertebereich einer dieser drei Faktoren evaluiert und die übrigen beiden Faktoren konstant gehalten.

Die Experimente wurden in einer Amazon EC2-Umgebung mit bis zu 100 virtuellen Maschinen des Typs *c1.medium*[2] durchgeführt. Auf jedem Knoten wurde Hadoop 0.20.2 installiert und analog zur Evaluation in [234] konfiguriert. Die Master-Prozesse (Namenode und Jobtracker) wurden auf einen dedizierten

[2] `https://aws.amazon.com/ec2/purchasing-options/reserved-instances`

Dataset	#Entities	#Blocks	#Pairs	Largest Block	
				#Entities	#Pairs
DS1	$1.1 \cdot 10^5$	1,483	$3 \cdot 10^8$	18%	71%
DS2	$13.9 \cdot 10^5$	14,659	$6.7 \cdot 10^9$	4%	26%

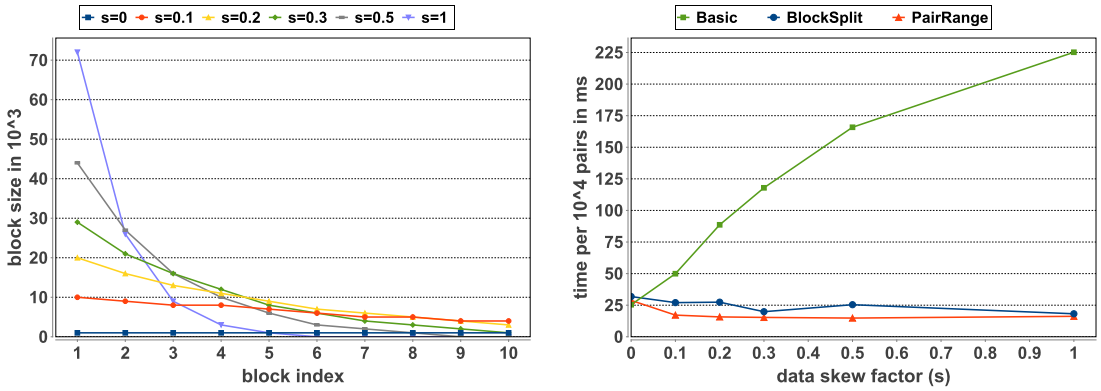
Abbildung 4.9: Zur Evaluation verwendete Datenquellen.

Knoten ausgelagert. Da jede c1.medium-Instanz über zwei virtuelle Prozessorkerne verfügt, wurde die Anzahl der gleichzeitig ausführbaren Map- bzw. Reduce-Tasks eines jeden Knotens auf 2 gesetzt. Zur Evaluierung wurden zwei verschiedene Datenquellen verwendet (siehe Abbildung 4.9). Die erste Datenquelle DS1 enthält ungefähr 114.000 Produktdatensätze. Die zweite Datenquelle DS2 (entspricht der in [234] verwendeten CiteseerX-Datenquelle) ist um eine Größenordnung größer und enthält ca. 1,4 Millionen Publikationsdatensätze. Um unterschiedliche Grade an Datenungleichverteilungen zu erzeugen, wurde im ersten Experiment eine künstliche Blockschlüsselgenerierung vorgenommen. In den beiden übrigen Experimenten fungierte der Präfix der Länge 3 des Produkt- bzw. Publikationstitels als Blockschlüssel eines jeden Datensatzes. Die resultierende Anzahl an Blöcken sowie die relative Größe des jeweils größten Blocks sind in Abbildung 4.9 aufgeschlüsselt. Für die Ähnlichkeitsberechnung der Datensätze wurde die Levenshtein-Distanz der Produkt- bzw. Publikationstitel bestimmt. Die Klassifikation in *Match* und *Non-Match* erfolgte anhand eines Ähnlichkeitsschwellertes von $t = 0,8$.

4.4.1 Grad der Datenungleichverteilung

Im Rahmen dieses Experimentes wird die Robustheit der Lastbalancierungsstrategien gegenüber verschiedenen Graden der Datenungleichverteilung untersucht und mit der Basic-Strategie verglichen. Zu diesem Zweck wird die Blockschlüsselverteilung künstlich angepasst, um für die Datenquelle DS1 Blöcke zu erzeugen, deren Größenverteilung einer Exponentialverteilung entspricht: Bei einer festen Anzahl von $b = 100$ Blöcken ist die Anzahl der Datensätze des k-ten Blocks proportional zu $e^{-s \cdot k}$. Dabei ist $s \in [0, 1]$ ein Parameter, um den Grad der Datenungleichverteilung zu justieren. Bei $s = 0$ enthalten alle $b = 100$ Blöcke dieselbe Anzahl an Datensätzen, wohingegen bei $s = 1$ 63% aller Datensätze dem ersten Block zugewiesen werden. Abbildung 4.10(a) illustriert die Blockgrößenverteilung für die ersten 10 von 100 Blöcken. Für die untersuchten Verteilungen ergeben sich Unterschiede in der Anzahl durchzuführender Paarvergleiche. Beispielsweise resultiert eine gleichmäßige Aufteilung von 50 Datensätzen auf zwei Blöcke in $2 \cdot 25 \cdot 24/2 = 600$ Paaren, wohingegen eine 45-zu-5-Aufteilung $45 \cdot 44/2 + 5 \cdot 4/2 = 1.000$ Paare definiert. Aus diesem Grund wird in diesem Experiment die durchschnittliche Ausführungszeit pro Datensatzpaar betrachtet.

Abbildung 4.10(b) vergleicht die durchschnittliche Ausführungszeit pro 10^4 Paaren der Basic, BlockSplit- und PairRange-Strategien für sechs verschiedene Blockgrößenverteilungen ($s \in \{0, 0.1, 0.2, 0.3, 0.5, 1\}$). Die Evaluierung erfolgte in einem MapReduce-Cluster bestehend aus $n = 10$ Knoten. Zur Ausführung des Matching-Jobs wurden $m = 20$ Map- und $r = 100$ Reduce-Tasks verwendet, sodass jeder Reduce-Task genau einen Block bearbeitet. Im Falle einer gleichmäßigen Blockgrößenverteilung ($s = 0$) ist die Basic-Strategie etwas schneller als beide Lastbalancierungsstrategien, da die Berechnung der BDM und der Overhead zur Lastbalancierung in der map-Phase entfällt und keine Replikation von Datensätzen erfolgt. Dieser zusätzliche Aufwand der Lastbalancierungsstrategien wird jedoch mit einem zunehmenden Grad an Datenungleichverteilung (also mit wachsendem s) vernachlässigbar, da die Zeit zur Berechnung der BDM konstant bleibt, aber die Gesamtanzahl der Paarvergleiche sowie der Anteil der Paare, die auf den größten Block entfallen, steigt. Bereits für eine moderate Datenungleichverteilung von $s = 0.1$ schneiden beide Lastbalancierungsstrategien um den Faktor 2 besser ab. Die Ergebnisse zeigen, dass die Basic-Strategie nicht robust gegenüber Datenungleichverteilung ist. Da die Anzahl der Paare des größten Blocks mit einer wachsenden Datenungleichverteilung zunimmt, dieser jedoch alleinig von einem Reduce-Task bearbeitet wird, ist die Basic-Strategie nicht in der Lage, die



(a) Blockgrößenverteilung

(b) Laufzeitvergleich

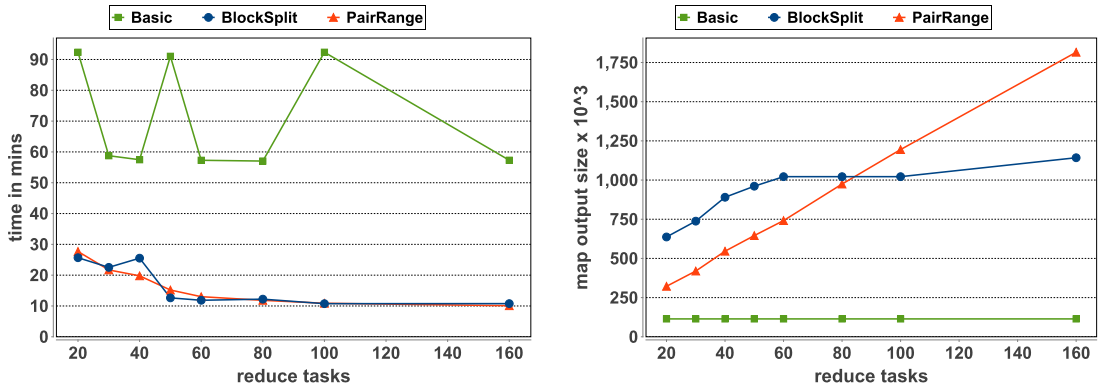
Abbildung 4.10: Aus der künstlichen Blockschlüsselgenerierung resultierende Blockgrößenverteilung für die Datenquelle DS1; gezeigt sind die Größen der ersten 10 von insgesamt 100 Blöcken (links). Laufzeiten für verschiedene Grade an Datenungleichverteilung unter Verwendung von $n = 10$ Knoten und $r = 100$ Reduce-Tasks (rechts).

Workload dieses Blocks auf mehrere Prozessoren zu verteilen. Für $s = 1$ benötigt die Basic-Strategie 225 ms pro 10^4 Paarvergleiche. Die ist im Vergleich zur BlockSplit und PairRange-Strategie um den Faktor 12 langsamer. Im Gegensatz dazu konnte sowohl die BlockSplit- als auch die PairRange-Strategie für alle untersuchten Blockgrößenverteilungen eine nahezu konstante durchschnittliche Ausführungszeit pro 10^4 Paare beobachtet werden. Aufgrund der gleichmäßigeren Aufteilung der Paarvergleiche auf die 100 Reduce-Tasks schneidet die PairRange-Strategie im Vergleich zur BlockSplit-Strategie etwas besser ab.

4.4.2 Anzahl der Reduce-Tasks

Das zweite Experiment untersucht den Einfluss der Anzahl ausgeführter Reduce-Tasks r in einem MapReduce-Cluster bestehend aus $n = 10$ Knoten. Dabei wird die Anzahl der Map-Tasks konstant auf $m = 20$ belassen und die Anzahl der Reduce-Tasks zwischen $r = 20$ und $r = 160$ variiert. Die Konfiguration der Map-Tasks sowie die untere Schranke der untersuchten Reduce-Task-Konfiguration ergibt sich auch der Tatsache, dass jeder der 10 MapReduce-Knoten mit je zwei virtuellen Prozessorkernen so konfiguriert ist, dass gleichzeitig zwei Map- und Reduce-Tasks ausgeführt werden können. Mit $m < 20$ oder $r < 20$ würden somit nicht alle Ressourcen des Clusters genutzt werden. Da die Ausführungszeit der Map-Phase im Vergleich zur der Reduce-Phase nahezu vernachlässigbar ist und die Map-Phase kaum anfällig gegenüber Datenungleichverteilungen ist, wurde lediglich die Anzahl der Reduce-Tasks variiert, da diese ein verbessertes Lastbalancierungspotential verspricht. Abbildung 4.11(a) zeigt die resultierenden Ausführungszeiten der Basic-, BlockSplit- und PairRange-Strategien für die Datenquelle DS1. Die abgetragenen Ausführungszeiten der BlockSplit- und PairRange-Strategien beinhalten den Overhead zur Berechnung der BDM in Höhe von ca. 35s.

Erneut ist zu beobachten, dass beide Lastbalancierungsstrategien wesentlich besser abschneiden als die Basic-Strategie, beispielsweise konnte die Ausführungszeit für $r = 160$ um den Faktor 6 verbessert werden. Die Basic-Strategie profitiert offenbar nicht von einer hohen Anzahl ausgeführter Reduce-Tasks, da sie nicht in der Lage ist, die Workload großer Blöcke auf mehrere Reduce-Tasks zu verteilen. Infolgedessen stellt die Zeit, die für die Bearbeitung des größten Blocks (auf den mehr als 70% aller Paarvergleiche entfallen, siehe Abbildung 4.9) benötigt wird, eine untere Schranke der Gesamtausführungszeit dar. Da die Zuweisung der Blöcke zu den Reduce-Tasks (unabhängig von der Blockgröße)



(a) Laufzeitvergleich für verschiedene Werte von r

(b) Anzahl generierter map-Ausgabe-Paare

Abbildung 4.11: Laufzeitvergleich aller Strategien für die Datenquelle DS1 in einem MapReduce-Cluster bestehend aus $n = 10$ Knoten und einer variierenden Anzahl an Reduce-Tasks (links). Anzahl der von der map-Funktion ausgegebenen Schlüssel-Wert-Paare für den Datensatz DS1 (rechts).

lediglich auf Basis einer Hashfunktion $h(key) \mod r$ erfolgt, kann es in Abhängigkeit von r sogar dazu kommen, dass mehrere große Blöcke einem Reduce-Task zugewiesen werden. Die ist beispielsweise für $r \in \{20, 50, 100\}$ der Fall. Im Gegensatz zur Basic-Strategie profitiert sowohl die BlockSplit- als auch die PairRange-Strategie deutlich von einer Erhöhung der Anzahl ausgeführter Reduce-Tasks. Aufgrund der nahezu perfekt gleichmäßigen Aufteilung aller Paarvergleiche schneidet die PairRange-Strategie für die maximale Anzahl von $r = 160$ Reduce-Tasks im Vergleich zur BlockSplit-Strategie um 7% besser ab. Für kleinere Werte von r kann PairRange jedoch langsamer als die BlockSplit-Strategie sein, da die Dauer der Ähnlichkeitsberechnung eines Datensatzpaares nicht konstant ist, sondern von den Längen der verglichenen Attributwerte abhängt. Der Einfluss dieser *Computational Skew*-Effekte nimmt jedoch mit zunehmenden r ab, da die Anzahl der Datensätze und Paarvergleiche pro Reduce-Task sinkt.

Abbildung 4.11(b) illustriert die Anzahl der in der Map-Phase generierten Schlüssel-Wert-Paare in Abhängigkeit von der Anzahl der Reduce-Tasks. Für die Basic-Strategie stimmt die Anzahl der von der map-Funktion ausgegebenen Paare mit der Anzahl der Eingabedatensätze überein, da, unabhängig von r, jeder Datensatz genau einem Reduce-Task zugewiesen wird und somit keine Replikation von Datensätzen erfolgt. Die BlockSplit-Strategie generiert für $r = 20$ die meisten map-Ausgabe-Paare aller untersuchten Strategien. Im weiteren Verlauf zeigt sie ein Verhalten ähnlich einer Treppenfunktion. Dies wird dadurch verursacht, dass die Anzahl der Reduce-Tasks bestimmt, ab welchem Größenschwellwert eine Aufteilung von Blöcken in Subblöcke erfolgt, aber nicht *wie* dies geschieht; die Generierung der Match-Tasks ist alleinig abhängig von der initialen Partitionierung der Eingabedaten. Eine Erhöhung von r führt dazu, dass mehr Blöcke in Subblöcke aufgeteilt werden, was eine Replikation von Datensätzen erfordert, da ein Datensatz an mehreren Paaren partizipiert, die i. Allg. von verschiedenen Match-Tasks bearbeitet werden. Für $r \geq 60$ erhöht sich die Anzahl der von der map-Funktion ausgegebenen Schlüssel-Wert-Paare nur noch in begrenztem Umfang, da die Blöcke, auf die die Mehrheit der Paarvergleiche entfällt, bereits für kleinere Werte von r in Subblöcke aufgeteilt wurden. Im Gegensatz dazu ist der Lastbalancierungsansatz der PairRange-Strategie unabhängig von der Größe einzelner Blöcke. Entscheidend ist lediglich die fixe Anzahl der Paarvergleiche P sowie die Anzahl der Reduce-Tasks (=Anzahl der Bereiche). Da jedem Reduce-Task die gleiche Anzahl an Paarvergleichen zugewiesen wird und deren Aufteilung (unabhängig von Blockgrenzen) lediglich anhand der Paarindexe vorgenommen wird, führt eine Erhöhung von r zwangsläufig dazu, dass immer weniger Blöcke vollständig von einem Reduce-Task bearbeitet werden. Aus diesem Grund wächst die Anzahl der map-Ausgabe-Paare nahezu linear mit der Anzahl der Reduce-Tasks. Demzufolge werden durch die PairRange-Strategie für $r = 160$ bei weitem die meisten Schlüssel-Wert-Paare von der map-Funktion ausgegeben. Der damit

Algorithm	Dataset	Execution Time in mins
BlockSplit	DS1	10.30
	DS1 (sorted)	18.30
PairRange	DS1	10.38
	DS1 (sorted)	11.78

Abbildung 4.12: Laufzeitvergleich der BlockSplit- und PairRange-Strategie für die nach Blockschlüssel sortierte/unsortierte Datenquelle DS1 unter Verwendung von $n = 10$ Knoten und $r = 100$.

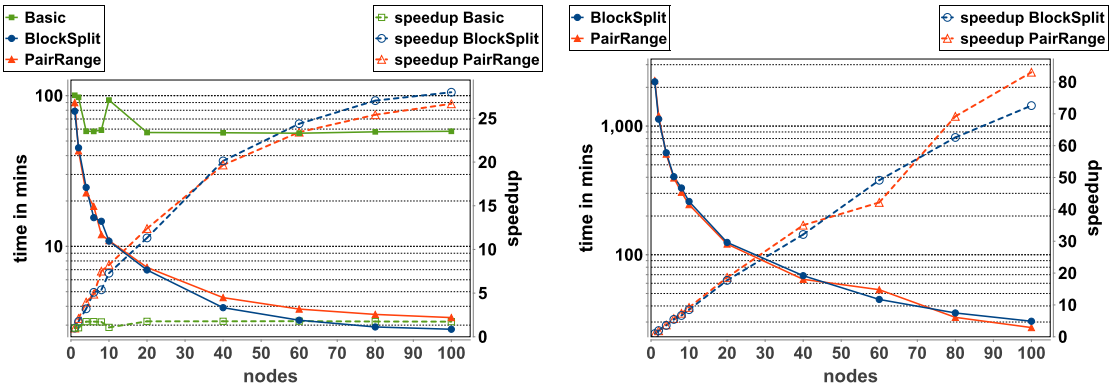
einhergehende Overhead zur Sortierung und Umverteilung schlägt sich interessanterweise *nicht* in der resultierenden Ausführungszeit nieder (vgl. Abbildung 4.11(a)), da die Ähnlichkeitsberechnung in der Reduce-Phase die Gesamtausführungszeit dominiert und die PairRange-Strategie eine gleichmäßigere Auslastung der Reduce-Tasks gewährleistet. Inwieweit dieses Verhalten auch für größere Datenquellen und größere MapReduce-Cluster zutrifft, wird im folgenden Experiment untersucht.

Das Experiment aus Abschnitt 4.4.1 zeigte, dass beide Lastbalancierungsstrategien *nicht* anfällig gegenüber Datenungleichverteilung sind. Es sei jedoch angemerkt, dass die Wirksamkeit der BlockSplit-Strategie von der gegebenen Partitionierung der Eingabedaten abhängt. Um diesen Effekt zu untersuchen, wurden die Datensätze der Datenquelle DS1 aufsteigend nach dem Produkttitel sortiert. Abbildung 4.12 zeigt die Ausführungszeiten der BlockSplit- und PairRange-Strategie für die unsortierte bzw. sortierte Datenquelle DS1. Die Laufzeitmessung erfolgte erneut in einem Cluster bestehend aus $n = 10$ Knoten mit $m = 20$ Map- und $r = 100$ Reduce-Tasks. Da sich der Blockschlüssel eines Datensatzes aus den ersten drei Zeichen des Produkttitels zusammensetzt, führt eine Sortierung der Datenquelle dazu, dass Datensätze großer Blöcke ungleichmäßig über die m Eingabepartitionen verteilt sind. Dies verhindert das Splitten großer Blöcke anhand der Eingabepartitionen sowie die Aufteilung ihrer Workload auf mehrere Match-Tasks. Das Experiment bestätigt dies. Im Falle nach Blockschlüssel sortierter Datensätze wurde eine Erhöhung der Ausführungszeit der BlockSplit-Strategie um ca. 80% beobachtet.

4.4.3 Skalierbarkeit

Die Gewährleistung der Skalierbarkeit von MapReduce-Algorithmen ist auch aus monetären Gesichtspunkten entscheidend. Anbieter von Infrastructure as a Service-Diensten berechnen die zu entrichtenden Nutzungsgebühren üblicherweise auf Basis (angebrochener) Maschinenstunden, egal ob diese ausgelastet sind oder nicht. Zur Evaluierung der Skalierbarkeit der Basic-, BlockSplit- und PairRange-Strategien wird die Größe des MapReduce-Clusters zwischen $n = 1$ und $n = 100$ Knoten variiert. Dabei wird entsprechend der Beobachtungen aus Abschnitt 4.4.2 die Anzahl der Map-Tasks auf $m = 2 \cdot n$ und die Anzahl der Reduce-Tasks auf $r = 10 \cdot n$ gesetzt – für jeden hinzukommenden Knoten (mit zwei virtuellen Kernen) wird also m um 2 und r um $5 \cdot 2$ erhöht. Die Abbildungen 4.13(a) und 4.13(b) zeigen die Ausführungszeiten der einzelnen Strategien für die Datensätze DS1 bzw. DS2.

Wie erwartet, skaliert das Basic-Verfahren aufgrund der Tatsache, dass alle Datensätze eines Blocks durch einen Reduce-Task verglichen werden, nicht für mehr als zwei Knoten. Die Gesamtausführungszeit wird durch den Reduce-Task dominiert, der den größten Block bearbeitet und damit 70% aller Paarvergleiche durchführt (DS1). Ein Hinzunahme weiterer Knoten reduziert lediglich die zusätzliche Workload des Reduce-Tasks, der den größten Block bearbeitet, verringert die Gesamtausführungszeit jedoch nur geringfügig. Auf die Evaluation der Basic-Strategie für die um eine Größenordnung größere Datenquelle DS2 wurde angesichts dieses Resultats aus Kostengründen verzichtet. Im Gegensatz dazu wird deutlich, dass sowohl BlockSplit als auch PairRange in der Lage sind, die zu verrichtende Arbeit gleichmäßig über die zu Verfügung stehenden Reduce-Tasks und Knoten zu verteilen. Sie skalieren nahezu linear bis zu $n = 10$ Knoten für die kleinere Datenquelle DS1 und bis zu $n = 40$ Knoten für



(a) DS1 (b) DS2.

Abbildung 4.13: Ausführungszeiten und Speedup-Werte der einzelnen Strategien.

die größere Datenquelle DS2. Für große n weisen beide Lastbalancierungsverfahren bessere Speedup-Werte für DS2 als für DS1 auf. Dies ist durch die höhere Gesamtlast begründet, die für eine effiziente Ausnutzung der verfügbaren Prozessoren entscheidend ist. Für DS1 hingegen ist die Bearbeitungszeit eines einzelnen Reduce-Tasks so gering, dass aufgrund des Task Scheduling Overheads eine Verdopplung der Clustergröße von 10 Knoten (100 Reduce-Tasks) auf 20 Knoten (200 Reduce-Tasks) die Ausführungszeit lediglich um den Faktor 1.5 reduziert. Aufgrund der vergleichsweise geringen Anzahl an Paarvergleichen pro Reduce-Task und der deutlich höheren Anzahl an `map`-Ausgabe-Paaren (vgl. Abbildung 4.11(b)) schneidet für DS1 und $n = 100$ die BlockSplit-Strategie (Speedup 28) besser ab als die PairRange-Strategie (Speedup 26.7). Da bei der Bearbeitung von DS2 die durchschnittliche Anzahl von Vergleichen pro Reduce-Task 2.000 mal höher als bei der Bearbeitung von DS1 ist (vgl. Abbildung 4.9), wird dieser Mehraufwand der PairRange-Strategie durch die optimal balancierte Reduce-Task-Workload aufgewogen. Allgemein gilt, dass die BlockSplit-Strategie für kleine, teilbare[3] Datenquellen bevorzugt werden kann, solange die enthaltenen Datensätze nicht nach dem zur Blockschlüsselgenerierung verwendetem Attribut sortiert sind. In den übrigen Fällen ist die PairRange-Strategie zu bevorzugen.

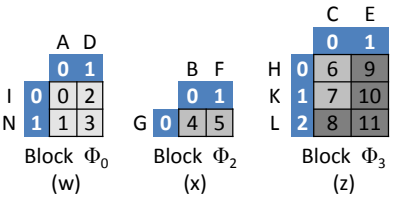
Abschließend wurde das Verhältnis der Ausführungszeit zur Bearbeitung der Map-Phase (inkl. Sortierung in der Map-Phase, Datenumverteilung und Sortierung der eingehenden Datensätze durch die Reduce-Tasks) und zur Ähnlichkeitsberechnung in der Reduce-Phase gemessen. Für PairRange und n=10 Knoten beträgt das Verhältnis ungefähr 1:66 für DS1 und 1:1.200 für DS2. Die Ausführungszeit wird also in beiden Fällen deutlich durch die Reduce-Phase dominiert. Bei einem höheren Parallelitätsgrad ($n = 100$) ändert sich das Verhältnis auf 1:4 (DS1) bzw. 1:50 (DS2). Für BlockSplit konnte ein ähnliches Verhalten beobachtet werden. Offenbar profitiert die rechenintensive Reduce-Phase deutlich mehr von einem höheren Parallelitätsgrad als die I/O-lastige Map-Phase. Tatsächlich erhöht sich ab einer gewissen Knotenanzahl sogar die Ausführungszeit der Map-Phase. Dies liegt daran, dass die Daten in Blöcken fester Größe im verteilten Dateisystem (repliziert) gespeichert sind und die Anzahl der Blöcke ausschließlich von der Größe der Datenquelle und der HDFS-Blockgröße, nicht jedoch von der Clustergröße n, abhängt. Zum Cluster neu hinzugefügte Knoten speichern i. Allg. keine Blöcke bestehender Daten. Map-Tasks dieser Knoten müssen somit die ihnen zugewiesene Eingabedatenpartition von Datanodes anderer Knoten über das Netzwerk lesen, was zu einer Erhöhung der Gesamtausführungszeit führen kann.

[3] In Abhängigkeit vom verwendeten Komprimierungsalgorithmus ist es möglich, dass eine komprimierte Datenquelle nicht durch mehrere Map-Tasks bearbeitet werden kann (vgl. [242]).

R	S	
Π_0	Π_1	Π_2
A_w	G_x	L_z
B_x	H_z	M_y
C_z	I_w	N_w
D_w	J_y	
E_z	K_z	
F_x		

Blocks				Partition Π_0	Partition Π_1	Partition Π_2	Overall #E	Overall #P
w	Φ_0	R		2	-	-	2	4
		S		-	1	1	2	
y	Φ_1	R		0	-	-	2	0
		S		-	2	0	0	
x	Φ_2	R		2	-	-	2	2
		S		-	0	1	1	
z	Φ_3	R		2	-	-	2	6
		S		-	2	1	3	

(a) Beispieldaten (links) und BDM (rechts).



(b) Nummerierungsschema und Bereichsaufteilung.

Abbildung 4.14: Entity Resolution für zwei Datenquellen R und S.

4.5 Erweiterung der Lastbalancierungsalgorithmen für das Matchen zweier Eingabedatenquellen

Dieser Abschnitt beschreibt die Anpassung der in den vergangenen Abschnitten vorgestellten Lastbalancierungsstrategien zur Duplikaterkennung in zwei Datenquellen R und S. Erneut wird im Folgenden angenommen, dass jeder Datensatz einen gültigen Blockschlüssel hat. Zur Behandlung ungültiger oder fehlender Blockschlüssel (vgl. Behandlung des *misc*-Blocks in [126]) kann, wie folgt, vorgegangen werden:

$$\text{match}_B(R,S) = \text{match}_B(R \setminus R_\emptyset,\ S \setminus S_\emptyset) \cup \text{match}_\perp(R,\ S_\emptyset) \cup \text{match}_\perp(R_\emptyset,\ S \setminus S_\emptyset) \tag{4.5}$$

Dabei beinhalten $R_\emptyset \subseteq R$ und $S_\emptyset \subseteq S$ alle Datensätze ohne gültigen Blockschlüssel. Die Berechnung aller Duplikate $(a,b) \subseteq R \times S$ mithilfe einer Blockschlüsselgenerierungsfunktion B wird mit $\text{match}_B(R,S)$ bezeichnet. Das Ergebnis dieses Schrittes kann durch die Vereinigung der Ergebnisse dreier Teilschritte ermittelt werden. Zunächst wird die Duplikaterkennung für alle Datensätze mit gültigem Blockschlüssel durchgeführt ($R \setminus R_\emptyset$ und $S \setminus S_\emptyset$). Das Ergebnis dieses Schrittes wird anschließend um die Ergebnisse der Evaluation des Kartesischen Produktes zwischen R und $S_\emptyset$ sowie zwischen $R_\emptyset$ und $S \setminus S_\emptyset$ angereichert. Die letzteren beiden Teilschritte können beispielsweise durch Verwendung eines konstanten Blockschlüssel $\perp$ realisiert werden. Eine Betrachtung effizienterer Methoden zur Auswertung des Kartesischen Produktes zweier Datenquellen erfolgt im Kapitel 7.

Zu vereinfachten Darstellung wird im Folgenden weiterhin angenommen, dass jede map-Eingabepartition ausschließlich Datensätze *einer* Datenquelle enthält[4]. Zu Illustration der Algorithmen werden die Datensätze $A-N$ mit den Blockschlüsseln $w-z$ aus Abbildung 4.14(a) verwendet. Jeder Datensatz gehört zu einer der beiden Datenquellen R und S. Die Anzahl der Partitionen (HDFS-Blöcke) der betrachteten Datenquelle kann sich unterschieden – R erstreckt sich lediglich über eine Partition Π_0, wohingegen S auf die beiden Partitionen Π_1 und Π_2 verteilt ist. Der Datenanalyse-Job zur Berechnung der BDM ist weitesgehend identisch zum Ein-Quellen-Fall (vgl. Abschnitt 4.1.2). Der einzige Unterschied ist, dass die `map`-Funktion den `map`-Ausgabe-Schlüssel (blocking_key⊙ partition_index⊙source), um eine dritte Komponente erweitert, sodass in der Reduce-Phase eine separate Bestimmung der Anzahl der Datensätze des Blocks Φ_i für die Datenquellen R und S erfolgen kann ($\Phi_{i,R}, \Phi_{i,S}$). Die Sortierung der `map`-Ausgabe-Paare erfolgt alleinig auf Basis der Blockschlüsselkomponente. Zur Sortierung und Gruppierung wird der vollständige Schlüssel verwendet. Die BDM hat im Wesentlichen dieselbe Struktur wie im Ein-Quellen-Fall – der einzige Unterschied besteht in der logischen Aufteilung eines (Block, Partition)-Paares in zwei Teilmengen, die jeweils die entsprechenden Datensätze aus R bzw. S enthalten (siehe Abbildung 4.14(a)).

[4] Dies kann problemlos mithilfe des *MultipleInputs*-Funktionalität des Hadoop-Frameworks sichergestellt werden

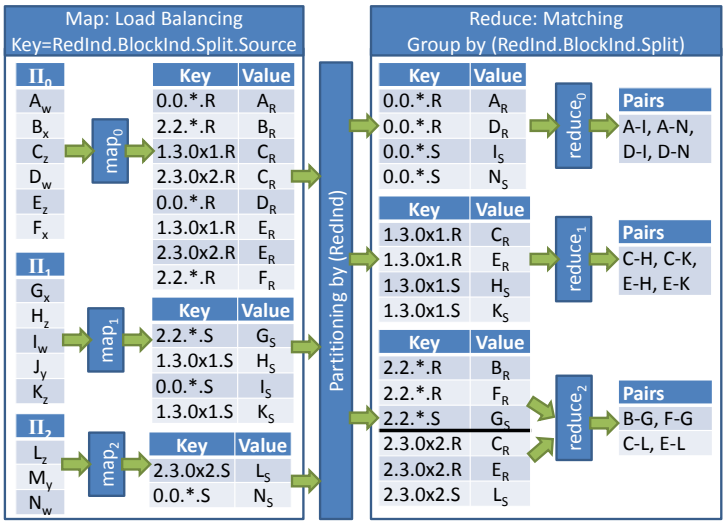


Abbildung 4.15: Datenfluss des BlockSplit-Algorithmus für die Datenquellen R und S aus Abbildung 4.14(a).

4.5.1 Blockorientierte Lastbalancierung

Zur Unterstützung des Zwei-Quellen-Falls muss der beschriebene BlockSplit-Algorithmus nur leicht angepasst werden. Der Hauptunterschied besteht in der Erweiterung der `map`-Ausgabe-Schlüssel (und -Werte) um eine Komponente zur Unterscheidung aus welcher Datenquelle die einzelnen Datensätze stammen. Die `map`-Funktion gibt somit Schlüssel-Wert-Paare mit Schlüsseln der Form (reduce_index⊙block_index⊙split⊙source) und Werten der Form (entity⊙source) aus. Wie auch im Ein-Quellen-Fall werden große Blöcke Φ_k in Subblöcke aufgeteilt. Für einen Match-Task $k.i \times j$ werden dabei allerdings nur Datensätze aus $\Phi_{k,R}$ der Partition Π_i sowie Datensätze aus $\Phi_{k,S}$ der Partition Π_j betrachtet.

Abbildung 4.15 zeigt den Datenfluss der BlockSplit-Strategie für die Beispieldaten aus Abbildung 4.14(a). Der BDM ist zu entnehmen, dass insgesamt $P = 12$ Paarvergleiche durchzuführen sind. Demnach muss der größte Block Φ_3 behandelt werden, da seine Workload von sechs Paaren die durchschnittliche Workload eines Reduce-Tasks in Höhe von $P/r = 4$ Paare übersteigt. Dies Aufteilung der Paarvergleiche des Blocks resultiert in zwei Match-Tasks *3.0×1* und *3.0×2*. Nach der Sortierung der Match-Tasks ergibt sich folgende Reihenfolge und Reduce-Task-Zuordnung: *0.** (4 Paare, reduce_0), *3.0×1* (4 Paare, reduce_1), *2.** (2 Paare, reduce_2), *3.0×2* (2 Paare, reduce_2). Die Partitionierung der Schlüssel-Wert-Paare erfolgt auf Basis der ersten Komponente des Schlüssels, die Sortierung erfolgt anhand des vollständigen Schlüssels. Zur Gruppierung werden die ersten drei Komponenten der Schlüssel herangezogen – die `reduce`-Funktion wird für jeden Match-Task $k.i \times j$ aufgerufen . Die Sortierung ermöglicht eine Verarbeitung von nach Datenquelle sortierten Datensätzen: zunächst werden alle Datensätze aus R gepuffert, anschließend wird jeder weitere Datensatz (aus S) mit allen gepufferten Datensätzen verglichen.

4.5.2 Paarorientierte Lastbalancierung

Der PairRange-Algorithmus für zwei Eingabedatenquellen basiert auf den gleichen Ideen wie die Variante für eine einzelne Datenquelle. Die Nummerierung der Datensätze erfolgt pro Block *und* Datenquelle. Ebenso werden die Paare (a, b) eines Blockes Φ_i nummeriert, wobei gilt: $a \in \Phi_{i,R}$ und $b \in \Phi_{i,S}$. Das Nummerierungsschema ordnet die Datensätze von R als Spalten und die Datensätze aus S als Zeilen

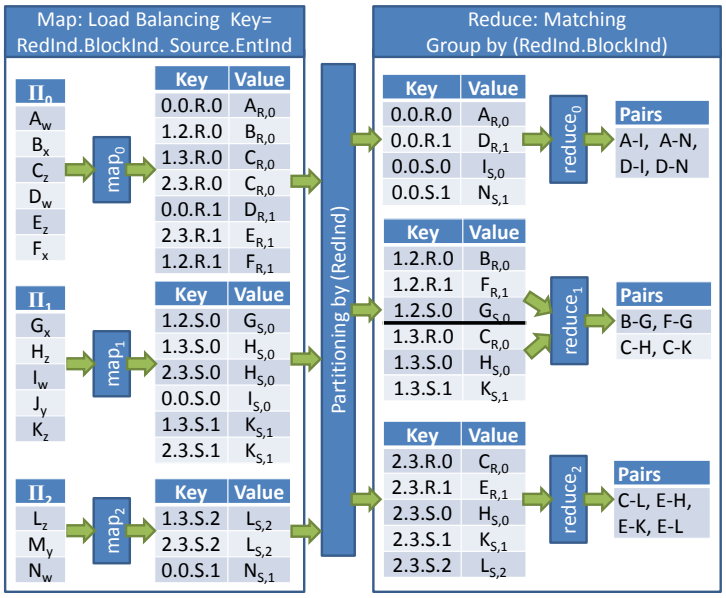


Abbildung 4.16: Datenfluss des PairRange-Algorithmus für die Datenquellen R und S aus Abbildung 4.14(a).

der $|\Phi_{i,R}| \times |\Phi_{i,S}|$-Matrix an. Für zwei Datensätze $e_R \in \Phi_{i,R}$ und $e_S \in \Phi_{i,S}$ mit den block-und quellenspezifischen Datensatzindexen x und y ist der globale Paar-Index, wie folgt, definiert:

$$p_i(x,y) = c(x,y,|\Phi_{i,S}|) + o(i) \tag{4.6}$$

$$c(x,y,N) = x \cdot N + y \tag{4.7}$$

$$o(i) = \Sigma_{k=0}^{i-1}(|\Phi_{k,R}| \cdot |\Phi_{k,S}|) - 1 \tag{4.8}$$

Abbildung 4.14(b) zeigt die resultierende Paarnummerierung für die Beispieldatenquellen aus Abbildung 4.14(a). Mit $r = 3$ werden die 12 Paarvergleiche auf drei, jeweils vier Paare umfassende, Bereiche aufgeteilt. Block Φ_1 muss nicht berücksichtigt werden, da kein Datensatz aus R den Blockschlüssel y hat. Die `map`-Funktion bestimmt für jeden Datensatz, in welche Bereiche dieser fällt. Für einen Datensatz $e_R \in \Phi_{i,R}$ mit Index x müssen dabei die Bereiche der Paare $p_i(x,0)$ bis $p_i(x,|\Phi_{i,S}|)$ betrachtet werden. Für einen Datensatz $e_S \in \Phi_{i,S}$ mit Index y sind die Bereiche der Paare $p_i(0,y)$ bis $p_i(|\Phi_{i,R}|,y)$ relevant. Für jeden der auf die Weise bestimmten Bereiche gibt die `map`-Funktion ein Schlüssel-Wert-Paar mit dem Schlüssel (range_index⊙block_index⊙source⊙entity_index) und dem Wert (entity⊙source⊙entity_index) aus. Im Vergleich zur Variante für eine Datenquelle wird der Wert neben dem Index des Datensatzes auch mit einem Flag für die Datenquelle (R bzw. S) annotiert. Die Partitionierung der `map`-Ausgabe-Paare erfolgt anhand des Bereichsindexes. Die Sortierung der Datensätze erfolgt anhand des vollständigen Schlüssels. Die `reduce`-Funktion wird für jeden Block aufgerufen (Gruppierung anhand der ersten beiden Komponenten). Erneut ermöglicht die Sortierung der Schlüssel-Wert-Paare eine Verarbeitung der nach Datenquelle sortierten Datensätze.

Abbildung 4.16 illustriert den Ansatz für die Datenquellen aus Abbildung 4.14(a). Der Datensatz $C \in R$ hat beispielsweise den Index 0 innerhalb des Blocks $\Phi_{3,R}$. Er partizipiert an Paaren, die in die Bereiche $\mathfrak{R}_1$ und $\mathfrak{R}_2$ fallen. Demzufolge gibt die `map`-Funktion zwei Schlüssel-Wert-Paare mit den Schlüsseln (1.3.R.0) und (2.3.R.0) aus, um C zum zweiten und zum dritten Reduce-Task zu senden.

4.6 Zusammenfassung

In diesem Kapitel wurden mit BlockSplit und PairRange zwei Algorithmen zur effizienten MapReduce-Parallelisierung Blocking-basierter Entity Resolution-Workflows vorgestellt. Durch die Aufteilung der Paarvergleiche großer Blöcke auf mehrere Reduce Tasks garantieren beide Verfahren eine gleichmäßige Auslastung der eingesetzten Ressourcen. Die Evaluierung in einer Cloud-Umgebung zeigte, dass beide Strategien robust gegenüber tatsächlich vorliegender Datenungleichverteilung sind und zudem mit der Anzahl der Clusterknoten skalieren. Der BlockSplit-Algorithmus ist im Vergleich zum PairRange-Ansatz konzeptionell einfacher und verursacht einen geringeres zwischen der Map- und Reduce-Phase umverteiltes Datenvolumen. PairRange ist hingegen völlig unabhängig von der gegebenen Partitionierung der Eingabedaten, hat einen geringeren Speicherbedarf und weist für große Datenquellen aufgrund der nahezu perfekten Lastbalancierung ein geringfügig besseres Skalierbarkeitsverhalten als die BlockSplit-Strategie auf.

5
Sorted Neighborhood

Dieses Kapitel widmet sich der Verwendung des MapReduce-Programmiermodells zur Parallelisierung des populären Sorted Neighborhood-Verfahrens. Nach einer kurzen Einführung erfolgt eine Analyse der wesentlichen Probleme, die bei der Umsetzung des Sorted Neighborhood-Blockings entstehen (Abschnitt 5.1). Anschließend werden im Abschnitt 5.2 zwei MapReduce-Algorithmen (JobSN und RepSN) vorgestellt, die das Sorted Neighborhood-Verfahren korrekt implementieren. Abschnitt 5.3 beschreibt, wie die RepSN-Strategie angepasst werden kann, um eine simultane Mehrfachausführung des Basisverfahrens unter Verwendung verschiedener, von verschiedenen Attributen abgeleiteter, Sortierschlüssel zu unterstützen (vgl. Abschnitt 2.2.2). Abschnitt 5.4 widmet sich der automatischen Datenpartitionierung und -umverteilung zur Sicherstellung einer gleichmäßigen Auslastung der verwendeten Clusterknoten. Eine umfangreiche Evaluierung der vorgestellten Algorithmen erfolgt im Abschnitt 5.5.

5.1 Einführung

Das bereits im Abschnitt 2.2.2 beschriebene Sorted Neighborhood-Verfahren ist ein Blocking-Verfahren, das Datensätze anhand ihrer Blockschlüssel sortiert, um ähnliche Datensätze nah beieinander anzuordnen. Anschließend wird ein Fenster einer festen Größe w schrittweise über die n sortierten Datensätze geschoben und eine Ähnlichkeitsberechnung der Datensätze innerhalb des Fensters vorgenommen (siehe Abbildung 2.2). Das Sorted Neighborhood-Verfahren reduziert die Komplexität der Duplikaterkennung von $O(n^2)$ für die Evaluierung des Kartesischen Produktes von n Datensätzen auf $O(n)+O(n\cdot\log n)+O(n\cdot w)$ für die Blockschlüsselgenerierung, Sortierung und Ähnlichkeitsberechnung. Die Gesamtzahl der Paarvergleiche $(n-w/2)\cdot(w-1)$ ist im Gegensatz zum Standard Blocking unabhängig von der zugrunde liegenden Blockschlüsselverteilung, sondern alleinig von der Anzahl der Datensätze n sowie der Fenstergröße w abhängig. Die Wahl der Fenstergröße w erlaubt es somit, zwischen der Anzahl der durchgeführten Paarvergleiche (die wiederum die Laufzeit bestimmt) und der resultierenden Qualität abzuwägen. Des Weiteren ist das Sorted Neighborhood-Verfahren im Gegensatz zum Standard Blocking weniger anfällig gegenüber einer suboptimalen Wahl des Blockschlüssels, da auch Datensätze mit unterschiedlichen (aber ähnlichen) Blockschlüsseln verglichen werden. Die mehrfache Ausführung des Basisverfahrens unter Verwendung unterschiedlicher Blockschlüssel (im Folgenden bezeichnet als Multi-pass Sorted Neighborhood) verringert den Einfluss "schlechter" Blockschlüssel (z. B. aufgrund unsauberer Daten) und ermöglicht i. Allg. die Verwendung kleinerer Fenstergrößen. Die *lineare* Komplexität der Ähnlichkeitsberechnung resultiert zudem im Vergleich zum Standard Blocking in einer geringeren Anfälligkeit für Lastbalancierungsprobleme.

Der wesentliche Unterschied des Sorted Neighborhood-Blockings im Vergleich zu anderen Blocking-Verfahren ist die Tatsache, dass die Ähnlichkeitsberechnung nicht ausschließlich auf Datensätze beschränkt ist, die denselben Blockschlüssel besitzen. Im Beispiel von Abbildung 2.2 haben die Datensätze d und b verschiedene Blockschlüssel (1 und 2). Da ihre Distanz in der nach Blockschlüssel sortierten Datensatzliste kleiner w ist, werden sie trotzdem (in der zweiten Fensterposition) miteinander verglichen. Diese Eigenschaft steht prinzipiell im Widerspruch mit dem MapReduce-Konzept, das eine unabhängige Bearbeitung der Eingabepartitionen durch mehrere Map-Tasks und eine unabhängige Bearbeitung

(nach Blockschlüssel) gruppierter Datensätze durch mehrere Reduce-Tasks vorsieht. Das Kernproblem ist also die Frage, wie eine Gruppierung von Datensätzen realisiert werden kann, deren Distanz kleiner als w ist. Unter der Annahme, dass die Position eines Datensatzes in der sortierten Liste bekannt ist, wäre eine Realisierung ähnlich der im Abschnitt 5.1 vorgestellten Basic-Strategie grundsätzlich möglich. Hierzu könnte beispielsweise die Fensterpostion als Kriterium zur Umverteilung und Gruppierung der Datensätze verwendet werden. Neben der Tatsache, dass die Daten weder sortiert vorliegen noch die Position in der sortierten Datensatzliste bekannt ist, hat ein solches Vorgehen den Nachteil, dass aufgrund der hochgradig überlappenden Blöcke (z. B. $\{a, d, b\}$ und $\{d, b, e\}$ in Abbildung 2.2) nahezu jeder Datensatz in w verschiedene Blöcke fällt und somit w-mal von der `map`-Funktion ausgegeben werden müsste.

Um eine effiziente MapReduce-basierte Parallelisierung des Sorted Neighborhood-Verfahrens zu gewährleisten, muss die Basic-Strategie angepasst werden. In der `map`-Funktion erfolgt nach wie vor die Bestimmung des Blockschlüssels eines jeden Datensatzes. Anschließend werden die Datensätze zu den Reduce-Tasks umverteilt – für das Beispiel aus Abbildung 2.2 wäre es bei der Verwendung von zwei Reduce-Tasks denkbar, alle Datensätze mit einem Blockschlüssel ≤ 2 dem ersten Reduce-Task und alle übrigen Datensätze dem zweiten Reduce-Task zuzuweisen. Die Reduce-Tasks sortieren die eingehenden Datensätze nach dem Blockschlüssel und implementieren anschließend den *Sliding Window*-Ansatz. Dazu muss ein Reduce-Task entweder alle Datensätze in eine einzige Gruppe gruppieren (sodass in der `reduce`-Funktion über alle Datensätze iteriert werden kann) oder eine Liste der letzten $w-1$ verarbeiteten Datensätze im Hauptspeicher verwalten, auf die, über mehrere Aufrufe der `reduce`-Funktion hinweg, zugegriffen werden kann. In Dedoop wurde die zweite Variante verwendet. Eine MapReduce-Umsetzung dieser Idee muss im Wesentlichen die nachfolgend beschriebenen Probleme lösen:

Sorted Reduce Partitions: Der Sorted Neighborhood-Ansatz verlangt eine Sortierung der Datensätze anhand ihrer Blockschlüssel. Um eine Implementierung des Sliding Window-Ansatzes durch die Reduce-Tasks zu ermöglichen, muss bei der Umverteilung der `map`-Ausgabedaten sicherstellt sein, dass *kein* Datensatz, der dem Reduce-Task R_i zugewiesen wird, einen größeren Blockschlüssel als *irgendein* Datensatz des Reduce-Tasks R_{i+1} hat. Die im folgenden Abschnitt vorgestellten Ansätze zur Umsetzung des Sorted Neighborhood-Verfahrens lösen dieses Problem durch die Verwendung einer Bereichspartitionierung und der Konstruktion zusammengesetzter `map`-Ausgabeschlüssel.

Boundary Entities: Die Idee des fortlaufenden Verschiebens des Fensters bedingt, dass nicht ausschließlich Daten eines Reduce-Tasks miteinander verglichen werden, sondern erfordert den Vergleich von Datensätzen über Reduce-Task-Grenzen hinweg. Im Detail müssen die letzten $v < w$ Datensätze des Reduce-Tasks R_i mit den w—v ersten Datensätzen des nachfolgenden Reduce-Tasks R_{i+1} verglichen werden. Diese Datensätze werden im weiteren Verlauf des Kapitels als *Boundary Entities* bezeichnet. Zur vereinfachten Darstellung wird angenommen, dass jedem Reduce-Task mindestens w Datensätze zugewiesen werden. Unter dieser Annahme ist es ausreichend, lediglich Datensätze aufeinanderfolgender Reduce-Tasks miteinander zu vergleichen[1]. Im folgenden Abschnitt werden zwei Ansätze namens JobSN und RepSN vorgestellt, welche mehrere MapReduce-Jobs ausführen bzw. Datensätze replizieren, um eine korrekte Verarbeitung der Boundary Entities zu gewährleisten (Abschnitt 5.2.1 und 5.2.2).

Multi-pass Sorted Neighborhood: Das Basis-Sorted Neighborhood-Verfahren kann mehrfach für verschiedene Blockschlüssel ausgeführt werden. Ein solcher Multi-pass Ansatz verspricht sowohl eine verbesserte Match-Qualität als auch eine höhere Effizienz des Blocking-Verfahrens [105]. Im Abschnitt 5.3 wird deswegen eine effiziente Implementierung des Multi-pass Sorted Neighborhood-Blockings vorgestellt, die das mehrfache Lesen der Eingabedatenmenge vermeidet. Stattdessen werden sämtliche Blockschlüssel in einer Map-Phase berechnet und die Paarvergleiche aller Durchgänge in einer Reduce-Phase durchgeführt.

[1] Die in Dedoop verwendete Implementierung ist in der Lage, Boundary Entities über mehrere Reduce-Tasks hinweg zu vergleichen, was bei großen Fenstergrößen und einem hohen Parallelitätsgrad notwendig sein kann.

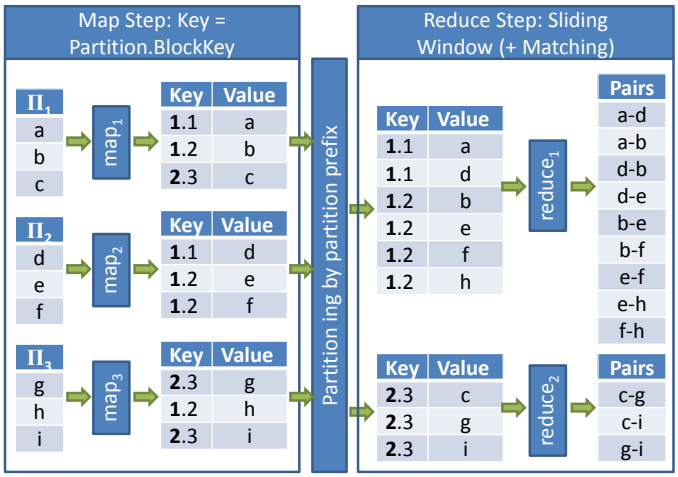


Abbildung 5.1: Verwendung einer Bereichspartitionierungsfunktion zur Sicherstellung der Reduce-Task-übergreifenden Sortierung der Datensätze aus Abbildung 2.2. Die map-Ausgabe-Schlüssel sind aus einem Partitionspräfix und dem eigentlichen Blockschlüssel zusammengesetzt. Die Partitionierung anhand des Partitionspräfixes und die komponentenweise Sortierung der zusammengesetzten Schlüssel sichert die gewünschte Sortierreihenfolge der Datensätze. Wenn beide Reduce-Tasks ein Fenster der Größe $w = 3$ über ihre Eingabedaten schieben, werden lediglich 12 von 15 Vergleichen durchgeführt (vgl. Abbildung 2.2). Die Datensatzpaare (f, c), (h, c) und (h, g) können nicht verarbeitet werden, da die Komponenten eines Paares jeweils zu verschiedenen Reduce-Tasks umverteilt wurden.

Lastbalancierung: Da die Zuweisung von map-Ausgabe-Schlüssel-Wert-Paaren anhand der Blockschlüssel der Datensätze erfolgt, führt eine ungleichmäßige Häufigkeitsverteilung der auftretenden Blockschlüssel (trotz der nicht-quadratischen Komplexität des Sorted Neighborhood-Verfahrens) ebenfalls zu Lastbalancierungsproblemen. Im Abschnitt 5.4 wird deshalb eine Erweiterung der Implementierung des Multi-pass Sorted Neighborhood-Blockings vorgestellt, die eine gleichmäßige Aufteilung aller Paarvergleiche auf die Reduce-Tasks garantiert.

5.2 Umsetzung des Sorted Neighborhood-Verfahrens

Die globale Reduce-Task-übergreifende Sortierung der Datensätze wird im Wesentlichen durch die Verwendung einer geeigneten Partitionierungsfunktion part erreicht. Dabei weist part: $k \rightarrow [1, r]$ jeden Datensatz anhand seines Blockschlüssels k zu einem der r Reduce-Tasks zu. Die Verwendung einer monoton wachsenden Partitionierungsfunktion ($k_1 \geq k_2 \Rightarrow \text{part}(k_1) \geq \text{part}(k_2)$) stellt dabei sicher, dass kein Datensatz, der dem Reduce-Task i zugewiesen wird, einen größeren Blockschlüssel als irgendein Datensatz des Reduce-Tasks $i + 1$ hat. Da der Wertebereich möglicher Blockschlüssel i. d. R. vorab bekannt ist, kann eine einfache Bereichspartitionierungsfunktion verwendet werden, um dies sicherzustellen. Zunächst wird davon ausgegangen, dass diese Funktion nutzerdefiniert ist, die automatische Bestimmung der optimalen Partitionierungsfunktion wird im Abschnitt 5.4 betrachtet.

Abbildung 5.1 illustriert die Verwendung einer Bereichspartitionierungsfunktion für das Beispiel aus Abbildung 2.2 mit $m = 3$ Map- und $r = 2$ Reduce-Tasks. In diesem Beispiel ist part, wie folgt, definiert: $\text{part}(k) = 1$ wenn $k \leq 2$, sonst $\text{part}(k) = 2$. Die map-Funktion generiert zunächst den Blockschlüssel k eines jeden Datensatzes und konstruiert einen aus zwei Komponenten zusammengesetzten Schlüssel (part(k)⊙k). Im Beispiel der Abbildung 5.1 hat der Datensatz c den Blockschlüssel 3. Da $\text{part}(3) = 2$ ist, gibt die map-Funktion ein Schlüssel-Wert-Paar (2.3, c) aus, das zum zweiten Reduce-Task gesen-

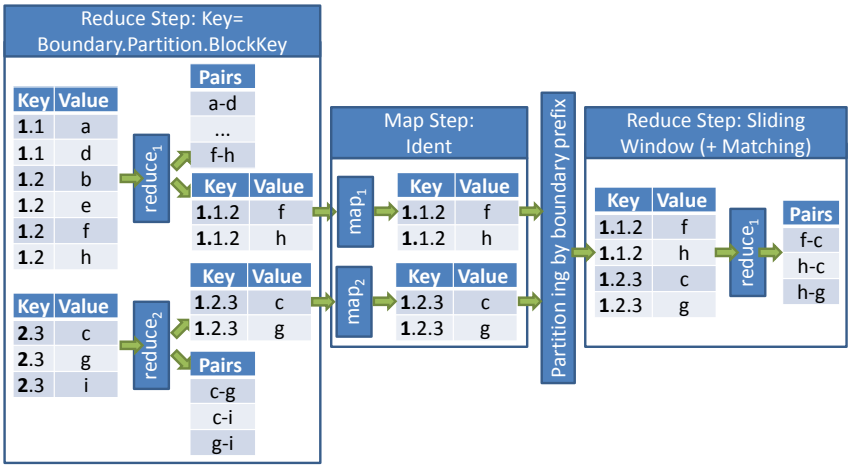


Abbildung 5.2: JobSN: Erweiterung der Strategie aus Abbildung 5.1 um einen zusätzlichen MapReduce-Job zum Vergleich korrespondierender Boundary Entities ($w = 3$). Die linke Box entspricht der Reduce-Phase des ersten MapReduce-Jobs. Die ausgegebenen Boundary Entities bilden die Eingabe des zweiten MapReduce-Jobs.

det wird. Jeder Reduce-Task sortiert die eingehenden Schlüssel-Wert-Paare komponentenweise anhand des vollständigen Schlüssels. Da alle Schlüssel denselben Partitionspräfix aufweisen, werden alle Datensätze entsprechend der Sortierreihenfolge ihrer Blockschlüssel sortiert. Anschließend kann jeder Reduce-Task das Fenster über die sortierten Datensätze seiner Eingabedatenmenge schieben und den Vergleich der Datensätze innerhalb des Fensters vornehmen. Die alleinige Verwendung einer Bereichspartitionierungsfunktion ist nicht ausreichend, um das Sorted Neighborhood-Verfahren korrekt zu implementieren, da Datensätze mit einer Distanz $< w$, die verschiedenen Reduce-Tasks zugeordnet sind, nicht miteinander verglichen werden. Im Beispiel von Abbildung 5.1 betrifft dies die Paare $(f, c), (h, c)$ und (h, g). Allgemein werden bei einer Fenstergröße von w und r Reduce-Tasks $(r-1) \cdot w \cdot (w-1)/2$ Paare nicht verarbeitet. Aus diesem Grund werden nachfolgend zwei Ansätze namens JobSN und RepSN vorgestellt, die bisher skizzierte Idee um die Fähigkeit zur korrekten Behandlung der Boundary Entities erweitert.

5.2.1 JobSN: Ausführung eines zusätzlichen MapReduce-Jobs

Die JobSN-Strategie verwendet ebenfalls eine Bereichspartitionierungsfunktion, um die globale Sortierung der Datensätze sicherzustellen. Zusätzlich wird dieser Basis-Ansatz um einen zweiten Schritt erweitert. In einem nachgelagerten MapReduce-Job werden die Vergleiche der Boundary Entities mit einer Distanz $< w$ durchgeführt. Dabei wird die Sortierung der Datensätze ausgenutzt; jeder Reduce-Task des ersten MapReduce-Jobs bestimmt die ersten und letzten $w-1$ Datensätze seiner Eingabedatenmenge. Die letzten $w-1$ Datensätze des Reduce-Tasks i sind mit den ersten $w-1$ Datensätzen des Reduce-Tasks $i+1$ in Beziehung zu setzen. Um dies zu erreichen, schreibt jeder Reduce-Tasks des ersten MapReduce-Jobs diese $2 \cdot (w-1)$ Boundary Entities in eine zusätzliche (von der regulären Ausgabe getrennte) Datei in das verteilte Dateisystem[2]. Der erste Reduce-Tasks schreibt dabei lediglich die letzten $w-1$, der letzte Reduce-Task lediglich die ersten $w-1$ Datensätze aus.

Abbildung 5.2 illustriert den JobSN-Ansatz für das gleiche Beispiel wie in Abbildung 5.1 ($w = 3$). Die Map-Phase des ersten MapReduce-Jobs ist identisch zu Abbildung 5.1 und aus Platzgründen weggelassen. In der Reduce-Phase erfolgt neben der Ähnlichkeitsberechnung eine zusätzliche Ausgabe der

[2] Dies kann beispielsweise durch Verwendung von Hadoops *MultipleOutputs*-Bibliothek implementiert werden.

Boundary Entities. Damit im zweiten MapReduce-Job korrespondierende Boundary Entities zusammengeführt und verglichen werden können, werden die Schlüssel der zusätzlich ausgegebenen Schlüssel-Wert-Paare um eine weitere Komponente angereichert, welche die Reduce-Task-Grenze identifiziert. Die letzten $w-1$ Datensätze des Reduce-Tasks $i < r$ gehören zur Grenze zwischen den Reduce-Tasks i und $i+1$. Gleiches gilt für die ersten $w-1$ Datensätze des Reduce-Tasks $i+1$. In der Folge wird der Reduce-Eingabe-Schlüssel dieser Datensätze um den Präfix i erweitert und ein Schlüssel-Wert-Paar (boundary⊙part(k)⊙k, entitiy) für jede Boundary Entity ausgegeben. Im Beispiel präfigiert der erste Reduce-Task die Schlüssel der letzten Datensätze f, h mit der Komponente 1 und schreibt die Schlüssel-Wert-Paare (1.1.2, f) und (1.1.2, h) in eine gesonderte Datei. Der zweite Reduce-Task führt denselben Schritt für die ersten beiden Datensätze c, g seiner Eingabepartition durch und gibt die Paare (1.2.3, c) und (1.2.3, g) aus. Der dreistellige Schlüssel kodiert somit die Datenabstammung der Datensätze – Datensatz g hat den Blockschlüssel 3, wurde dem zweiten Reduce-Task zugewiesen und ist eine Boundary Entity der Grenze zwischen dem ersten und dem zweiten Reduce-Task.

Die Map-Tasks des zusätzlichen MapReduce-Jobs lesen die vom vorigen Job ausgegebenen zusätzlichen Schlüssel-Wert-Paare (Boundary Entities) und geben diese unverändert aus. Die Partitionierung und Umverteilung der Schlüssel-Wert-Paare erfolgt anhand der ersten Komponente. Die Schlüssel-Wert-Paare werden komponentenweise anhand des vollständigen Schlüssels sortiert. Auf diese Weise kann in der `reduce`-Funktion das Fenster über alle korrespondierenden Boundary Entities geschoben werden. Dabei werden ausschließlich Datensätze miteinander verglichen, die von verschiedenen Reduce-Tasks des ersten MapReduce-Jobs ausgegeben wurden, beispielsweise wird das Paar (f, h) nicht erneut bearbeitet, da dies bereits im ersten Job erfolgte.

Algorithmus 5.1 zeigt den Pseudo-Code[3] der JobSN-Strategie. Die Funktionen map_configure, reduce_configure, map_close und reduce_close werden vom Hadoop-Framework automatisch nach der Initialisierung bzw. vor Beendigung eines Map- bzw. Reduce-Tasks aufgerufen. Ein programmatischer Aufruf ist ebenso möglich (Zeile 24 und 71). Reduce-Tasks können Daten im Hauptspeicher puffern und über verschiedene `reduce`-Aufrufe hinweg darauf zugreifen. Die Partition zu der ein Datensatz mit dem Blockschlüssel k (im i-ten Durchgang) zugeordnet wird, wird durch eine getPartition(i, k, r) bestimmt. Diese kann entweder nutzerdefiniert oder automatisch bestimmt sein (siehe Abschnitt 5.4). Die Funktion additionalOut$(key, value)$ schreibt ein Schlüssel-Wert-Paar in binärer Form in eine gesonderte (Reduce-Task-spezifische) Datei in das verteilte Dateisystem. Diese zusätzliche Ausgabe bildet die Eingabe des zweiten MapReduce-Jobs. Die Verarbeitungsreihenfolge von Schlüssel-Wert-Paaren, die denselben Schlüssel aufweisen, in der `reduce`-Funktion ist undefiniert. Um zu gewährleisten, dass die Reihenfolge der in der Reduce-Phase des zweiten MapReduce-Jobs verarbeiteten Boundary Entities der Verarbeitungsreihenfolge des ersten MapReduce-Jobs entspricht, wird der Schlüssel einer jeden ausgegebenen Boundary Entity um einen Index erweitert, der den Suffix des zusammengesetzten Schlüssels bildet (Zeile 45 und 50). Ansonsten wäre es im Beispiel von Abbildung 5.2 möglich, dass in der Reduce-Phase des zweiten MapReduce-Jobs der Datensatz h vor dem Datensatz f oder der Datensatz g vor dem Datensatz c bearbeitet wird.

5.2.2 RepSN: Replikation von Datensätzen

Die RepSN-Strategie implementiert das Sorted Neighborhood mittels eines einzigen MapReduce-Jobs. Das Verfahren zielt darauf ab, der Datenpartition jedes Reduce-Tasks $i > 1$ die letzten $w-1$ Datensätze des Reduce-Tasks $i-1$ voranzustellen. Sofern dies gelingt, kann ein korrekter Vergleich der Boundary Entities gewährleistet werden. Da die $w-1$ Boundary Entities auch in der Partition i enthalten sein müssen, das MapReduce-Paradigma jedoch keinen Datenaustausch zwischen Reduce-Tasks vorsieht, müssen (potentielle) Boundary Entities bereits in der Map-Phase repliziert und durch eine geeignete Schlüsselkonstruktion an mehrere Reduce-Tasks gesendet werden.

[3] Der Algorithmus unterstützt das Multi-pass Sorted Neighborhood, das für $p = 1$ dem beschriebenen Verfahren entspricht.

Algorithmus 5.1: Implementierung der JobSN-Strategie (Multi-pass)

```
1  // -- Phase 1 --
2  map_configure(jobConf)
3      p ← getNumberOfPasses(jobConf);
4      reduceTasks ← jobConf.numReduceTasks();

5  map(key_in=unused, value_in=entity)
6      for pass ← 1 to p do
7          blockKey ← generateBlockingKeyForPass(pass, entity);
8          partition ← getPartition(pass, blockKey, reduceTasks);
9          output(key_tmp=pass.partition.blockKey, value_tmp=entity);

10 reduce_configure(jobConf)
11     p ← getNumberOfPasses(jobConf);
12     lastPass ← -1;
13     windowSize ← getWindowSize(jobConf);
14     reduceTasks ← jobConf.numReduceTasks();
15     queue ← [];
16     top ← [];
17     bottom ← [];

18 // part: repartition by partition component
19 // cmp: sort by entire map output key
20 // group: group by by entire map output key
21 reduce(key_tmp=pass.partition.blockKey, list(value_tmp)=list(entity))
22     if pass ≠lastPass then
23         if lastPass ≠ -1 then
24             reduce_close();
25         queue ← [];
26         lastPass ← pass;
27     foreach entity ∈ list(value_tmp) do
28         if partition>1 and top.size()<windowSize-1 then
29             top.addLast((partition, blockKey, entity));
30         if partition<reduceTasks then
31             if bottom.size() = windowSize-1 then
32                 bottom.removeFirst();
33             bottom.addLast((partition, blockKey, entity));
34         foreach e ∈ queue do
35             // Comparison + output
36             match(e, entity);
37         queue.addLast(entity);
38         if queue.size() = windowSize then
39             queue.removeFirst();

40 reduce_close()
41     i ← 0; j ← 0;
42     foreach (partition, blockKey, entity) ∈ top do
43         i ← i + 1;
44         additionalOut(
45             key_out=lastPass.(partition-1).partition.blockKey.i,
46             value_out = entity);
47     foreach (partition, blockKey, entity) ∈ bottom do
48         j ← j + 1;
49         additionalOut(
50             key_out=lastPass.partition.partition.blockKey.j,
51             value_out = entity);
52     top ← []; bottom ← [];

53 // -- Phase 2 --
54 // Read additional reduce output of phase 1
55 map(key_in=pass.boundary.partition.blockKey.entityIndex,
56   value_in=entity)
57     output(key_tmp=pass.boundary.partition.blockKey.entityIndex,
58       value_tmp=entity);

59 reduce_configure(jobConf)
60     lastPass ← -1;
61     lastBoundary ← -1;
62     windowSize ← getWindowSize(jobConf);
63     queue ← [];

64 // part(pass.boundary.partition.blockKey.entityIndex)=
65 //     hash(pass, partition) mod reduceTasks
66 // cmp: Sort by entire key
67 // group: Group by pass.boundary.partition
68 reduce(key_tmp=pass.boundary.partition.blockKey.entityIndex,
69   list(value_tmp)=list(entity))
70     if pass ≠ lastPass or boundary ≠ lastBoundary then
71         reduce_close();
72         queue ← [];
73         lastPass ← pass;
74         lastBoundary ← boundary;
75     foreach entity ∈ list(value_tmp) do
76         queue.addLast((partition, entity));

77 reduce_close()
78     // Match entites in queue with a distance of at
79     // most windowSize-1 from different partitions
```

Die RepSN-Strategie erweitert den Ansatz aus Abbildung 5.1. Durch die Anwendung einer Bereichspartitionierungsfunktion auf den Blockschlüssel eines jeden Datensatzes bestimmt die map-Funktion zunächst den Reduce Task, zudem der Datensatz gesendet werden soll. Für diesen Datensatz wird ein Schlüssel-Wert-Paar mit dem zusammengesetzten Schlüssel (part(k)⊙part(k)⊙k) ausgegeben. Die im Vergleich zum in Abbildung 5.1 skizzierten Ansatz neu hinzugekommene erste Schlüsselkomponente wird als Boundary-Präfix bezeichnet. Für regulär ausgegebene Schlüssel-Wert-Paare entspricht der Boundary-Präfix dem Partitionspräfix. Während der Anwendung der map-Funktion auf die Datensätze seiner Eingabepartition vermerkt jeder Map-Task für jeden Ziel-Reduce-Task $1 \leq i < r$ die $w-1$ verarbeiteten Datensätze mit dem größten Blockschlüssel k. Nach der vollständigen Bearbeitung seiner Eingabepartition repliziert jeder Map-Task die $w-1$ Datensätze jeder Reduce-Partition $1 \leq i < r$, die den größten Blockschlüssel aufweisen, zum Nachfolger-Reduce-Task $i+1$. Dazu wird für jeden replizierten Datensatz ein Schlüssel-Wert-Paar mit einem Schlüssel der Form (part(k)+1⊙part(k)⊙k) ausgegeben. Die Umverteilung der ausgegebenen Schlüssel-Wert-Paare erfolgt anhand des Boundary-Präfixes, also der ersten Komponente der dreistelligen Schlüssel. Die Sortierung der eingehenden Datensätze erfolgt komponentenweise anhand des vollständigen Schlüssels. Durch die Struktur der Schlüs-

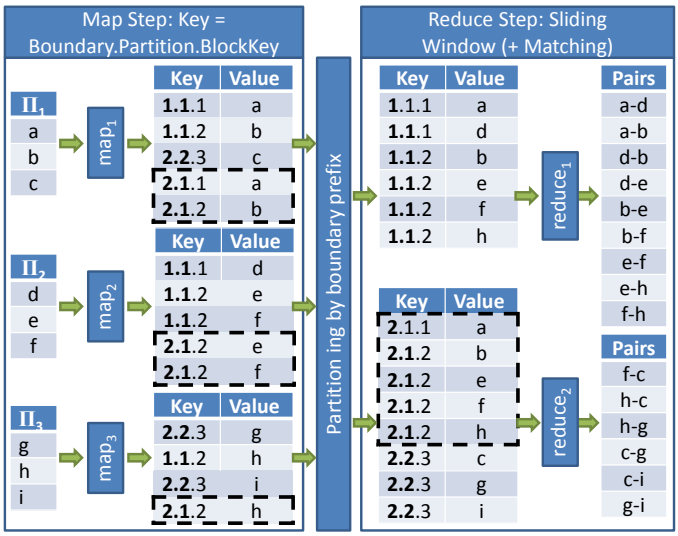


Abbildung 5.3: Ausführung der RepSN-Strategie für das Beispiel aus Abbildung 5.1 ($w = 3$). Potentielle Boundary Entities (gekennzeichnet durch einen gestrichelter Rahmen) werden durch die map-Funktion repliziert und zwei adjazenten Reduce-Tasks zugewiesen.

sel (die zweite Schlüsselkomponente ist kleiner als bei den regulären Datensätzen) ist sichergestellt, dass die zum Reduce-Task $i > 1$ replizierten Datensätze vor den regulären Datensätzen in der Reduce-Eingabepartition erscheinen. In der Folge kann der Reduce-Task das Fenster über die Boundary Entities und die regulären Datensätze schieben und alle Datensätze mit einer Distanz kleiner w vergleichen. Algorithmus 5.2 zeigt den Pseudo-Code[4] der RepSN-Strategie.

Abbildung 5.3 illustriert die RepSN-Strategie für das Beispiel aus Abbildung 5.1 für $w = 3$ und $r = 2$. Jeder der $m = 3$ Map-Tasks bestimmt die $w - 1 = 2$ Datensätze des Ziel-Reduce-Tasks 1 mit dem größten Blockschlüssel. Die Ausgabe der map-Funktion ist zweigeteilt. Der erste Teil entspricht dem Beispiel aus Abbildung 5.1. Der einzige (technische) Unterschied ist die Duplizierung des Partitionspräfixes. Der zweite Teil (gekennzeichnet durch einen gestrichelten Rahmen) besteht aus den Datensätzen, die zum zweiten Reduce-Task repliziert werden müssen. Dies soll am Beispiel des zweiten Map-Tasks verdeutlicht werden. Die Datensätze d, e und f mit den Blockschlüsseln 1, 2 und 2 werden alle dem ersten Reduce-Task zugewiesen ($\mathsf{part}(k) = 1$, wenn $k \leq 2$, sonst $\mathsf{part}(k) = 2$). Da e und f die beiden Datensätze mit dem größten Blockschlüssel sind, werden sie repliziert und zusätzlich zum zweiten Reduce-Task gesendet. Da keine Boundary Entities zum ersten Reduce-Task repliziert werden, stimmt dessen Eingabe und Arbeitsweise mit der des ersten Reduce-Tasks aus Abbildung 5.1 überein. Der zweite Reduce-Task hat im Vergleich zum Beispiel aus Abbildung 5.1 eine größere Eingabepartition zu bearbeiten. Er kann jedoch alle bis auf die $w - 1 = 2$ letzten replizierten Datensätze (f und h) verwerfen.

Der RepSN-Ansatz ermöglicht im Gegensatz zur JobSN-Strategie die Umsetzung des Sorted Neighborhood-Verfahrens in einem einzigen MapReduce-Job. Der Einsparung des zusätzlichen MapReduce-Jobs steht eine Replikation und Mehrfachausgabe von Datensätzen in der Map-Phase gegenüber. Da keine Kommunikation zwischen verschiedenen Map-Tasks erfolgt, ist eine globale Bestimmung der $w-1$ zum Reduce-Task $i > 1$ zu replizierenden Datensätze mit dem größten Blockschlüssel nicht möglich, stattdessen bestimmt jeder Map-Task die potentiell relevanten $w-1$ letzten Datensätze seiner Eingabepartition. Unabhängig von der Gesamtanzahl der Datensätze werden insgesamt $m \cdot (r-1) \cdot (w-1)$ Datensätze repliziert. Ein Vergleich des Overheads zur Datenreplikation des RepSN-Verfahrens mit dem Overhead der Ausführung eines zusätzlichen MapReduce-Jobs der JobSN-Strategie erfolgt in Abschnitt 5.5.

[4] Der Algorithmus unterstützt das Multi-pass Sorted Neighborhood, dass für $p = 1$ dem beschriebenen Verfahren entspricht.

Algorithmus 5.2: Implementierung der RepSN-Strategie (Multi-pass)

```
1  map_configure(jobConf)
2      p ← getNumberOfPasses(jobConf);
3      reduceTasks ← jobConf.numReduceTasks();
4      windowSize ← getWindowSize(jobConf);
5      // list of the entities with the w-1 highest
6      // blocking keys for each pass and reduce task<r
7      for i ← 1 to p do
8          for j ← 1 to reduceTasks-1 do
9              rep^i_j ← [];

10 map(key_in=unused, value_in=entity)
11     for i ← 1 to p do
12         blockKey ← generateBlockingKeyForPass(i, entity);
13         j ← getPartition(i, blockKey, reduceTasks);
14         if j < reduceTasks then
15             if rep^i_j.size()<windowSize-1 then
16                 rep^i_j.add((blockKey, entity));
17                 // Sort by blocking key
18                 rep^i_j.sort();
19             else if blockKey > rep^i_j.getFirst().getBlockKey() then
20                 rep^i_j.removeFirst();
21                 rep^i_j.add((blockKey, entity));
22                 // Sort by blocking key
23                 rep^i_j.sort();
24         // Regular output with key structure
25         //    pass.boundary.partition.blockKey
26         output(key_tmp=i.j.j.blockKey, value_tmp=entity);

27 map_close
28     for i ← 1 to p do
29         for j ← 1 to reduceTasks-1 do
30             foreach (blockKey, entity) ∈ rep^i_j do
31                 // Replicate boundary entities
32                 // to next reduce task
33                 output(key_tmp=i.(j+1).j.blockKey,
34                   value_tmp=entity);

35 reduce_configure(jobConf)
36     p ← getNumberOfPasses(jobConf);
37     windowSize ← getWindowSize(jobConf);
38     lastPass ← -1;
39     queue ← [];

40 // part(pass.boundary.partition.blockKey)
41 //    = boundary
42 // cmp: Sort by entire key
43 // group: Group by entire key
44 reduce(key_tmp=pass.boundary.partition.blockKey,
45   list(value_tmp)=list(entity))
46     if i ≠ lastPass then
47         lastPass ← i;
48         queue ← [];
49     foreach entity ∈ list(value_tmp) do
50         if boundary = partition then
51             // regular entity
52             foreach e ∈ queue do
53                 // Comparison + output
54                 match(e, entity);
55         queue.addLast(entity);
56         if queue.size() = windowSize then
57             queue.removeFirst();
```

5.3 Multi-pass Sorted Neighborhood

Die Verwendung eines einzelnen Blockschlüssels pro Datensatz kann unter Umständen nicht ausreichend sein, um alle Datensätze die ohne den Blocking-Schritt als Duplikat erkannt worden wären, zu identifizieren. Des Weiteren kann es erforderlich sein, relativ große Fenstergrößen zu verwenden. Bei der einmaligen Anwendung des Basisverfahrens mit einem Blockschlüssel pro Datensatz sollte die Fenstergröße mindestens dem größten Abstand zweier Datensätze mit demselben Blockschlüssel entsprechen. Eine große Fenstergröße zieht jedoch eine höhere Anzahl an Paarvergleichen nach sich und führt zu einer Erhöhung der Ausführungszeit. Die mehrfache Anwendung des Basisverfahrens unter Verwendung mehrerer, von verschiedenen Attributen abgeleiteten, Blockschlüssel erlaubt die Verwendung kleinerer Fenstergrößen und verspricht somit eine deutliche Verbesserung der Laufzeit und der Qualität des Match-Ergebnisses [105]. Eine naive Umsetzung dieses Multi-Pass-Ansatzes wäre die p-fache Ausführung des JobSN- oder RepSN-Verfahrens unter Verwendung p verschiedener Blockschlüssel pro Datensatz. Dieser Ansatz verursacht jedoch einen unnötigen Overhead für die p-fache Ausführung eines MapReduce-Jobs und das p-faches Lesen der Eingabedaten vom verteilten Dateisystem.

Im Folgenden wird eine Erweiterung der RepSN-Strategie (namens MultiRepSN) beschrieben, die das Multi-pass Sorted Neighborhood mittels eines einzigen MapReduce-Jobs realisiert (vgl. Algorithmus 5.2). Die Anpassung der JobSN-Strategie verläuft weitestgehend ähnlich (vgl. Algorithmus 5.1), auf eine nähere Beschreibung wird an dieser Stelle verzichtet. Die wesentliche Änderung der Idee des MultiRepSN-Verfahrens gegenüber der RepSN-Strategie ist das Anfügen eines Durchgangspräfixes zu jedem map-Ausgabe Schlüssel: Für jeden Eingabedatensatz und jeden Durchgang $i \in [1, p]$

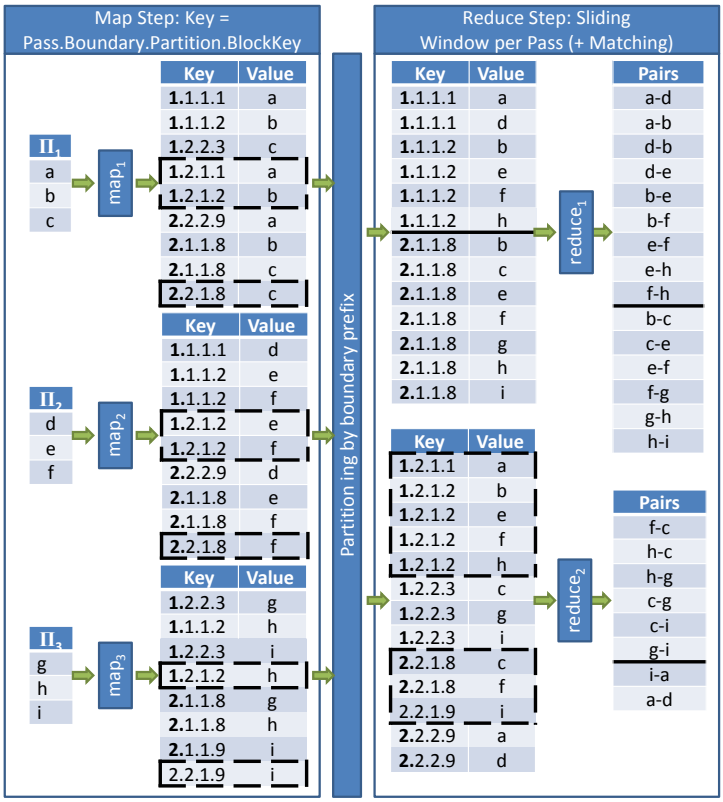


Abbildung 5.4: Erweiterung der RepSN-Strategie zur Unterstützung des Multi-pass Sorted Neighborhoods. Der erste Durchgang ($w_1 = 3$) entspricht dem Beispiel aus Abbildung 5.3. Im zweiten Durchgang wird eine Fenstergröße von $w_2 = 2$ verwendet.

gibt die `map`-Funktion ein Schlüssel-Wert-Paar mit einem zusammengesetzten Schlüssel der Form (i⊙part$_i$(k)⊙part$_i$(k)⊙k) aus, wobei k den Blockschlüssel des i-ten Durchgangs bezeichnet. Für jeden Durchgang i kommt dabei eine eigene Partitionierungsfunktion part$_i$ zum Einsatz, um auf Basis des Wertebereichs der in diesem Durchgang verwendeten Blockschlüsselgenerierungsfunktion eine möglichst gleichmäßige Aufteilung der Datensätze auf die Reduce-Tasks gewährleisten zu können. Analog zum Basis-RepSN bestimmt jeder Map-Task die potentiellen Boundary Entities *für jeden Durchgang* und gibt nach der Bearbeitung aller Datensätze seiner Eingabepartition entsprechende Schlüssel-Wert-Paare mit Schlüsseln der Form (i⊙part$_i$(k)+1⊙part$_i$(k)⊙k) aus. Die ausgegebenen Schlüssel-Wert Paare werden auf Basis der Boundary-Komponente (2. Komponente des vierstelligen Schlüssels) zu den Reduce-Tasks umverteilt. Die Sortierung und Gruppierung der Datensätze erfolgt komponentenweise anhand des vollständigen Schlüssels. Da die Nummer des Durchgangs die erste Schlüsselkomponente ist, werden die Datensätze (inkl. Boundary Entities) zunächst nach Durchgang und pro Durchgang nach Blockschlüssel sortiert. Somit kann jeder Reduce-Task das Fenster über seine Eingabedaten schieben und alle Datensätze mit derselben Durchgangsnummer miteinander vergleichen.

Abbildung 5.4 erweitert das Beispiel aus Abbildung 5.3 um einen zweiten Durchgang ($p = 2$, $w_1 = 3$, $w_2 = 2$). Die Blockschlüsselgenerierungsfunktion, die `map`-Ausgabe, die Partitionierungsfunktion und die Fenstergröße des ersten Durchgangs entsprechen Abbildung 5.3. Die im zweiten Durchgang verwendete Blockschlüsselgenerierungsfunktion weist den Datensätzen a und d den Blockschlüssel 9 zu, die übrigen Datensätze erhalten den Blockschlüssel 8. Die Partitionierungsfunktion part$_2$ weist die Datensätze den $r = 2$ Reduce-Tasks, wie folgt, zu: $part_2(k) = 1$, wenn $k = 8$ und $part_2(k) = 2$ sonst.

Die für den zweiten Durchgang ausgegebenen Paare haben einen Durchgangspräfix von 2 und führen zu (neuen) Paarvergleichen, die die Erkennung weiterer Korrespondenzen ermöglichen.

5.4 Automatische Bereichspartitionierung

Im Gegensatz zum Standard Blocking ist das Sorted Neighborhood-Verfahren weniger anfällig für Lastbalancierungsprobleme. Dies ist dadurch bedingt, dass sich die zu vergleichenden Datensätze ausschließlich aus der Sortierreihenfolge der Blockschlüssel und der verwendeten Fenstergröße ergeben. Die Anzahl der Paarvergleiche hängt lediglich von der Anzahl der Datensätze und der Fenstergröße ab, ist jedoch völlig unabhängig von der konkreten Häufigkeitsverteilung der Blockschlüssel. Somit ist es ausgeschlossen, dass die Bearbeitung der Datensätze eines sehr häufig auftretenden Blockschlüssels die Ausführungszeit der Entity Resolution-Workflows dominiert. Die im bisherigen Verlauf dieses Kapitels vorgestellten Ansätze zur MapReduce-Umsetzung des Sorted Neighborhood-Verfahrens verwenden eine Bereichspartitionierungsfunktion, um jeden Datensatz anhand seines Blockschlüssels zu einem der r Reduce-Tasks umzuverteilen und dabei eine Reduce-Task-übergreifende Sortierreihenfolge zu gewährleisten. Da die Partitionierungsfunktion deterministisch ist, werden infolgedessen alle Datensätze (bis auf die replizierten Boundary Entities) mit demselben Blockschlüssel demselben Reduce-Task zugewiesen. Da von Realwelt-Daten abgeleitete Blockschlüssel ungleich verteilt sind, verursacht dies eine variierende Anzahl von zu verarbeitenden Datensätzen und damit eine unterschiedliche Anzahl von Paarvergleichen pro Reduce-Task. Aufgrund der nicht-quadratischen Komplexität des Sorted Neighborhood-Verfahrens sind die Auswirkungen dieser ungleichmäßigen Lastverteilung zwar weniger drastisch als beim Standard-Blocking, trotzdem verspricht eine gleichmäßige Aufteilung der n Datensätze auf die r Reduce-Tasks eine deutliche Verbesserung der Laufzeit- und Skalierbarkeitseigenschaften. Darüber hinaus ist es selbst unter der Annahme, dass alle Blockschlüssel ungefähr gleich oft vorkommen, sehr schwierig eine gute Bereichspartitionierungsfunktion zu definieren, die die Datensätze gleichmäßig auf die Reduce-Tasks aufteilt. Dies gilt insbesondere für eine große Anzahl an Reduce Tasks (hoher Parallelitätsgrad). Durch die Mehrfachausführung des Sorted Neighborhood-Verfahrens ($p > 1$) wird der manuelle Aufwand zusätzlich erhöht, da in den verschiedenen Durchläufen unterschiedliche Blockschlüsselgenerierungsfunktionen mit unterschiedlichen Wertebereichen und Blockschlüsselverteilungen zum Einsatz kommen. Eine manuelle Bestimmung "guter" Partitionierungsfunktionen erfordert eine Vorabanalyse der Daten und geht mit einem hohen Konfigurationsaufwand einher. Das Beispiel aus Abbildung 5.4 illustriert bereits die Notwendigkeit einer automatischen gleichmäßigen Partitionierung der Datensätze. Obwohl beide Partitionierungsfunktionen gut gewählt sind, variiert die Anzahl der Paarvergleiche beider Reduce-Tasks zwischen 15 und 8 Paaren.

Im verbleibenden Teil dieses Abschnitts wird eine Erweiterung des MultiRepSN-Ansatzes zur automatischen Bereichspartitionierung vorgestellt. Dazu wird, wie bei der Lastbalancierung für das Standard Blocking, in einem Vorverarbeitungsschritt ein zusätzlicher MapReduce-Job zur Datenanalyse ausgeführt. Das Ergebnis dieses Vorverarbeitungsschrittes ist eine sogenannte *Key Partitioning Matrix* (KPM), die im Wesentlichen der BDM des Standard Blockings entspricht und die Anzahl der Datensätze pro Durchgang, Blockschlüssel und Eingabepartition enthält. Die KPM wird von den Map-Tasks des eigentlichen MR-Jobs zum Initialisierungszeitpunkt eingelesen, um eine gleichmäßige Zuweisung der n Datensätze zu den r Reduce-Task zu gewährleisten. Der präsentierte Ansatz kann ebenfalls (unverändert) für die JobSN-Strategie angewandt werden.

5.4.1 Berechnung der KPM

Die KPM enthält für jeden Durchgang $i \in [1, p]$ und jeden Blockschlüssel k die Anzahl der Datensätze pro Eingabepartition $j \in [1, m]$. Die MapReduce-Berechnung der KPM ähnelt der Berechnung der BDM (vgl. Abschnitt 4.1.2). Algorithmus 5.3 zeigt den Pseudo-Code der KPM-Berechnung. Bei der Anwen-

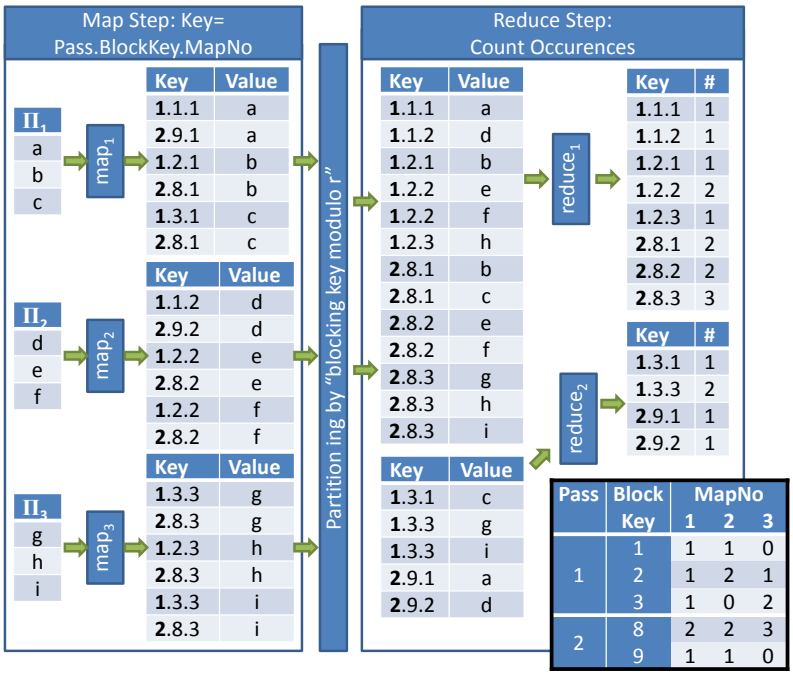


Abbildung 5.5: Datenfluss der Berechnung der Key Partitioning Matrix für das Beispiel aus Abbildung 5.4.

dung der `map`-Funktion auf einen Datensatz der j-ten Eingabepartition werden zunächst die p Blockschlüssel des Datensatzes bestimmt. Anschließend wird für Durchgang i ein Schlüssel-Wert Paar der Form (i⊙k$_i$⊙j, 1) ausgegeben, wobei k_i den Blockschlüssel des Durchgangs i bezeichnet. Die Partitionierung der Schlüssel-Wert-Paare erfolgt auf Basis der ersten beiden Komponenten der zusammengesetzten Schlüssel. Zur Sortierung und Gruppierung wird der vollständige Schlüssel herangezogen. Die Reduce-Tasks summieren die Anzahl der Datensätze pro `map`-Ausgabe-Schlüssel und geben die KPM zeilenweise aus. Analog zur Berechnung der BDM schreibt jeder Map-Task die Datensätze seiner Eingabepartition mitsamt der berechneten Blockschlüssel in eine zusätzliche Datei in das verteilte Dateisystem. Diese zusätzliche Ausgabe dient als Eingabe des darauffolgenden MapReduce-Jobs, der mit derselben Anzahl an Map-Tasks ausgeführt wird und dieselbe Partitionierung der Eingabedaten vorfindet.

Abbildung 5.5 illustriert die Berechnung der BDM für das Beispiel aus Abbildung 5.4 mit $p = 2$ Durchgängen. Für die Datensätze c, g und i wird im ersten Durchgang der Blockschlüssel 3 ermittelt. Da sich c in der ersten Eingabepartition befindet und sich die Datensätze g und i in der dritten Eingabepartition befinden, enthält die BDM die Zellen 1, 0 und 2, um die Aufteilung der Datensätze mit dem Blockschlüssel $k_1 = 3$ über die $m = 3$ Eingabepartitionen auszudrücken. Zu Darstellungszwecken sind die Zeilen der KPM in sortierter Reihenfolge angeordnet, die konkrete Reihenfolge ist jedoch irrelevant.

5.4.2 Lastbalancierung

Die Lastbalancierung basiert auf einer virtuellen Nummerierung aller in der Reduce-Phase zu verarbeitenden Schlüssel-Wert-Paare. Die der Nummerierung zugrunde liegende Reihenfolge ergibt sich aus der Sortierung der Schlüssel-Wert-Paare (in dieser Reihenfolge) anhand des Durchgangs, des Blockschlüssels, der `map`-Eingabepartition und der Position des entsprechenden Datensatzes innerhalb der Eingabepartition. Mithilfe der KPM kann jeder Map-Task aus der Kombination aus dem Durchgang $i \in [1, p]$, dem Blockschlüssel k_i, der Eingabepartition $j \in [1, m]$ und dem Index x eindeutig die globale Positi-

Algorithmus 5.3: Berechnung der KPM

```
1  map_configure(jobConf)
2      p ← getNumberOfPasses(jobConf);
3      partition ← getMapTaskIndex(jobConf);

4  map(key_in=unused, value_in=entity)
5      blockingKeys ← ;
6      for i ← 1 to p do
7          blockKey ← generateBlockingKeyForPass(i, entity);
8          blockingKeys ← blockingKeys + "\t"+ blockKey;
9          output(key_tmp=i.blockKey.partition, value_tmp=1);
10     additionalOut(key_out=blockingKeys, value_out=entity);

11 // part(pass.blockingKey.partition)=
12 //     hash(pass, blockingKey) % reduceTasks
13 // cmp: Sort by entire key
14 // grpup: Group by entire key
15 reduce(key_tmp=pass.blockKey.partition, list(value_tmp)=list(number))
16     sum ← 0;
17     foreach number in list(value_tmp) do
18         sum ← sum + number;
19     out ← pass + "\t" + blockKey + "\t" + partition + "\t" + sum;
20     output (key_out=unused, value_out=out);
```

on $pos \in [1,\ p \cdot n]$ eines auszugebenden Schlüssel-Wert-Paares in der für den Sliding Window-Ansatz benötigten Sortierreihenfolge bestimmen. Dabei bezeichnet x den x-ten Datensatz innerhalb der Eingabepartition j mit dem Blockschlüssel k_i.

Jeder Map-Task liest zum Initialisierungszeitpunkt die KPM (reguläre Ausgabe des vorigen MapReduce-Jobs) ein und bestimmt die Gesamtanzahl durchzuführender Paarvergleiche $P = \Sigma_{i=1}^{p}(n - \frac{w_i}{2}) \cdot (w_i - 1)$ sowie die durchschnittliche Anzahl an Paaren pro Reduce-Task $P_\emptyset = \frac{P}{r}$. Anschließend teilt jeder Map-Task die Liste der sortierten Schlüssel-Wert-Paare in r Intervalle, sodass sich beim Schieben des Fensters über die Datensätze der Intervalle eine (nahezu) gleiche Anzahl an Paarvergleichen ergibt[5]. Zu diesem Zweck ermittelt jeder Map-Task für jeden Reduce-Task $j \in [1, r]$ den kleinsten Index $pos_j \in [1,\ p \cdot n]$ eines Schlüssel-Wert-Paars, sodass die Anzahl der vom Reduce-Task j durchzuführenden Paarvergleiche größer oder gleich $P_\emptyset$ ist.

Nach der Initialisierung wenden die Map-Tasks die `map`-Funktion auf jeden Datensatz (zusätzliche Ausgabe des vorigen MapReduce-Jobs) an. Für jeden Datensatz e werden wie, im vorigen Abschnitt 5.3 beschrieben, p Schlüssel-Wert-Paare ausgegeben. Zur Bestimmung des Ziel-Reduce-Tasks $\mathsf{part}_i(\mathsf{k})$ eines jeden Schlüssel-Wert-Paares wird zunächst mithilfe des in der KPM kodierten Wissens dessen Position pos in der Sortierreihenfolge bestimmt. Anschließend wird der Ziel-Reduce-Task j durch Vergleich der Position des Schlüssel-Wert-Paares mit den während der Initialisierung bestimmten Intervallgrenzen ermittelt: $part(pos) = j \Leftrightarrow pos_{j-1} < pos \leq pos_j$ wobei gilt $pos_0 = 0$. Mithilfe der Intervallgrenzen kann ebenfalls ermittelt werden, ob das Schlüssel-Wert-Paar zum nachfolgenden Reduce-Task $(j+1) < r$ repliziert werden muss. Dies ist der Fall, wenn gilt: $pos_j - (w-1) < pos \leq pos_j$. Im Gegensatz zur manuellen Festlegung einer Bereichspartitionierungsfunktion kann somit die unnötige Replikation potentieller Boundary Entities (z. B. die Datensätze a, b und e des zweiten Reduce-Tasks in Abbildung 5.3) vermieden werden.

Abbildung 5.6 illustriert die automatische Bereichspartitionierung für das Beispiel aus Abbildung 5.5. Aus den $p = 2$ Durchgängen und den Fenstergrößen $w_1 = 3$ und $w_2 = 2$ ergeben sich für die $b = 9$ Datensätze insgesamt $P = (9-\frac{3}{2})\cdot(3-1)+(9-\frac{2}{2})\cdot(2-1) = 23$ Paarvergleiche. Bei gleichmäßiger Auslastung der Reduce-Tasks beträgt für $r = 2$ die durchschnittliche Anzahl an Paaren pro Reduce-Task $P_\emptyset = 11.5$.

[5] Dabei ist zu berücksichtigen, dass durch die unterschiedlichen Fenstergrößen w_i die Anzahl der Paarvergleiche der einzelnen Durchgänge variieren kann.

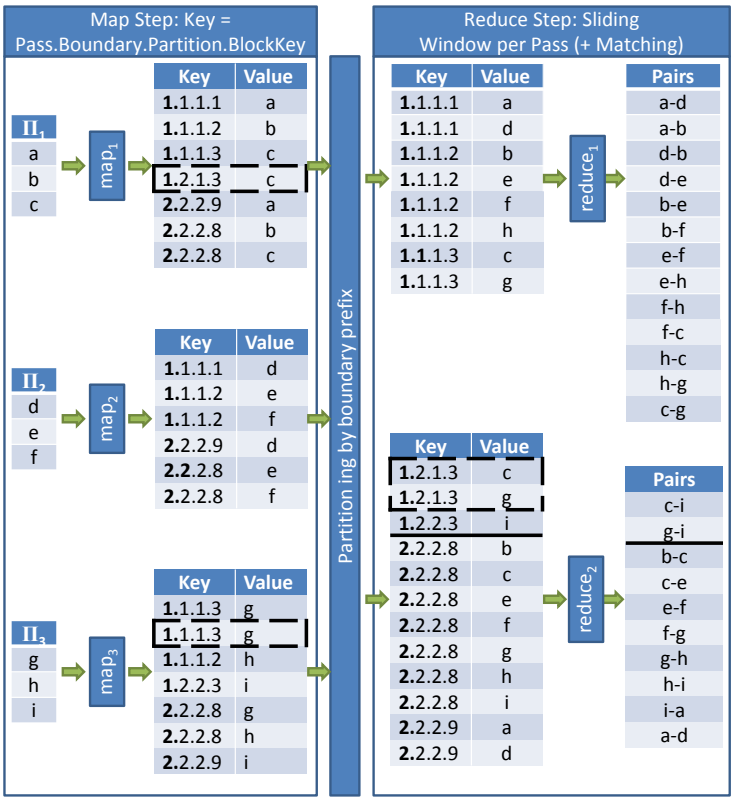


Abbildung 5.6: Anpassung des Beispiels aus Abbildung 5.4. Durch den Ansatz zur automatischen Bereichspartitionierung kann eine gleichmäßige Auslastung der Reduce-Tasks sichergestellt werden ($p = 2$, $w_1 = 3$, $w_2 = 2$).

Da im ersten Durchgang ein größeres Fenster als im zweiten Durchgang verwendet wird, verarbeiten beide Reduce-Tasks Schlüssel-Wert-Paare des ersten Durchgangs, wohingegen Schlüssel-Wert-Paare des zweiten Durchgangs ausschließlich durch den zweiten Reduce-Task bearbeitet werden. Für den ersten Durchgang ist g der letzte Datensatz, der vom ersten Reduce-Task zu bearbeiten ist. Dies ergibt sich aus der Tatsache, dass sich aus den Vorgängern von g (die Datensätze a, d, b, e, f, h, c) lediglich $11 < P_\emptyset$ Paarvergleiche ergeben, wohingegen g der erste Datensatz ist, ab dem die optimale Anzahl an Paarvergleichen pro Reduce-Task erreicht ist ($13 \geq P_\emptyset$). Die verbleibenden Schlüssel-Wert-Paare des ersten sowie des zweiten Durchgangs werden zum zweiten Reduce-Task gesendet. Um die Auswertung aller Paare des ersten Durchgangs, an denen der Datensatz i beteiligt ist, abzudecken, werden die Boundary Entities c und g des ersten Durchgangs ebenfalls zum zweiten Reduce-Task umverteilt. Insgesamt konnte im Vergleich zur in Abbildung 5.4 verwendeten manuellen Bereichspartitionierung eine gleichmäßigere Auslastung der Reduce-Tasks von 13 zu 10 Paaren (gegenüber 15 zu 8) erreicht werden. Die kleine Differenz zur optimalen Lastverteilung von 12 zu 11 ergibt sich aus der Tatsache, dass der letzte Datensatz die Anzahl der vom ersten Reduce-Task durchzuführenden Paarvergleiche von 11 auf 13 erhöht, da er in $w_1 - 1 = 2$ Paaren partizipiert. Der Ansatz garantiert, dass die Anzahl der Paare, die einem Reduce-Task zugewiesen wird, zwischen $P_\emptyset - (\max(w_i) - 1)$ und $P_\emptyset + (\max(w_i) - 1)$ liegt und sich somit die Last zweier beliebiger Reduce-Tasks um maximal $2 \cdot (\max(w_i) - 1)$ Paare unterscheidet.

Algorithmus 5.4 zeigt den Pseudo-Code des vorgestellten Ansatzes. Dieser Algorithmus ersetzt die *manuelle* Bereichspartitionierung der (Multi-pass) JobSN bzw. RepSN-Strategie (Algorithmus 5.1, Zeile 8 bzw. Algorithmus 5.2, Zeile 13) durch eine *automatische* Bereichspartitionierung. Während der Initiali-

Algorithmus 5.4: Automatische Bereichspartitionierung

```
1  configure(jobConf, mapTasks, reduceTasks, p, partition)
2      KPM ← readKPM(jobConf);
3      // Count number of entities
4      n ← 0;
5      foreach blockKey ∈ KPM.getPass(1) do
6          for i ← 1 to mapTasks do
7              n ← n +
8                KPM.getPass(1).getKey(blockKey).getPart(i);
9      // Global index of next entity of pass i and
10     // block j for this map task. This map contains
11     // at most one entry per (pass, blockKey) pair.
12     entityIndexes ← empty map;
13     for i ← 1 to p do
14         entityIndex ← 1;
15         foreach blockKey ∈ KPM.getPass(i).getKeysSorted() do
16             for j ← 1 to mapTasks do
17                 if j = partition then
18                     entityIndexes.put((i, blockKey),
19                       (i-1)·n + entityIndex);
20                 entityIndex ← entityIndex +
21                   KPM.getPass(i).getBlock(blockKey).getPart(j);
22     P ← Σ_{i=1}^{p} (n − w_i/2)·(w_i − 1); // Number of pairs
23     P_∅ ← ⌈P/reduceTasks⌉; // Avg. pairs per reduce task
24     splitPoints ← computeSplitPoints(p, P, P_∅);

25 computeSplitPoints(p, P, P_∅)
26     splitPoints ← [];
27     pairsLeft ← P_∅;
28     offset ← 0;
29     for i ← 1 to p do
30         entitiesLeft ← n;
31         while entitiesLeft > 0 do
32             entitiesThatFit ← min{pairsLeft/w_i + w_i/2, entitiesLeft};
33             offset ← offset + entitiesThatFit;
34             splitPoints.addLast(offset);
35             entitiesLeft ← entitiesLeft - entitiesThatFit;
36             pairsThatFit ← (entitiesThatFit − w_i/2)·(w_i − 1);
37             pairsLeft ← pairsLeft - pairsThatFit;
38             if pairsLeft ≤ 0 then
39                 pairsLeft ← P_∅;
40     return splitPoints;

41 getPartition(pass, blockKey, reduceTasks)
42     entityIndex ← entityIndexes.get((pass, blockKey));
43     // adjust entity index for next call
44     entityIndexes.get((pass, blockKey)).put(entityIndex + 1);
45     for i ← 1 to splitPoints.size() do
46         if entityIndex ≤ splitPoints.get(i) then
47             return i;
```

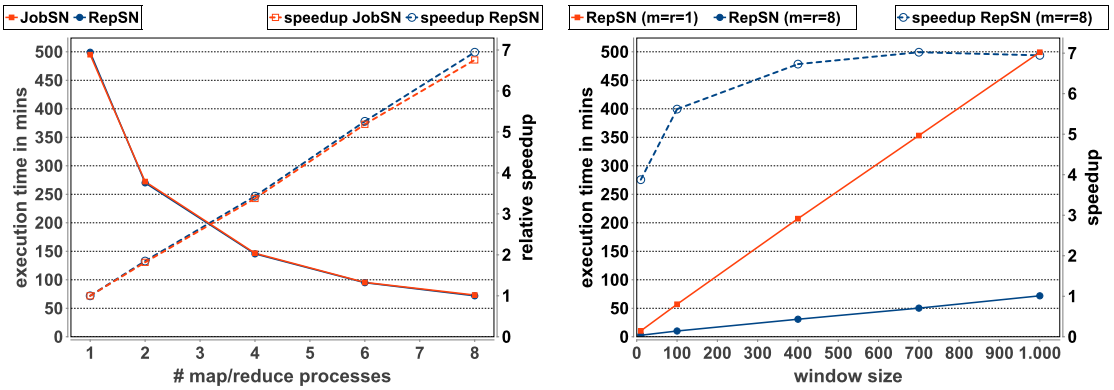
sierung der Map-Tasks des JobSN bzw. RepSN-Algorithmus wird in map_configure zusätzlich die Methode configure des Algorithmus 5.4 aufgerufen. Darin erfolgt in den Zeilen 22 bis 39 die Berechnung der Intervallgrenzen pos_j. Für die Bestimmung der globalen Position $pos \in [1,\ p \cdot n]$ eines jeden Schlüssel-Wert-Paares sind die Zeilen 12 bis 21 sowie Zeile 44 relevant.

5.5 Evaluation

In diesem Abschnitt wird eine experimentelle Evaluierung der vorgestellten Methoden vorgenommen. Dazu wird die JobSN- hinsichtlich Laufzeit und Skalierbarkeit mit der RepSN-Strategie verglichen und das Skalierbarkeitsverhalten für verschiedene Fenstergrößen untersucht (Abschnitt 5.5.1). Die Evaluierung des Ansatzes zur automatischen Bereichspartitionierung erfolgt im Abschnitt 5.5.2. Im Abschnitt 5.5.3 wird die vorgeschlagene Implementierung des Multi-pass Sorted Neighborhood mit der p-fachen Ausführung des Basisverfahrens verglichen. Anschließend wird im Abschnitt 5.5.4 anhand eines beispielhaften Match-Problems gezeigt, dass ein Multi-pass-Ansatz im Vergleich zum klassischen Verfahren zu einer Verbesserung der Qualität des Match-Ergebnisses *und* der Laufzeit führen kann.

Die Experimente wurden in einem lokalen Cluster bestehend aus vier homogenen Knoten mit je einer Intel(R) Core(TM)2 Duo E6750 2x2.66 GHz CPU, 4GB Hauptspeicher, einem 64-bit Debian Linux OS und einer Java 1.6 64-bit Server JVM durchgeführt. Die Demonstration der Skalierbarkeit bei Verwendung einer automatischen Bereichspartitionierung (Abschnitt 5.5.2) kam zudem ein Hadoop-Cluster in einer Amazon EC2-Umgebung mit 100 virtuellen Maschinen des Typs *c1.medium* zum Einsatz. Auf jedem Knoten wurde Hadoop 0.20.2 installiert und analog zur Evaluation in [234] konfiguriert. Die beiden Master-Prozesse (Namenode und Jobtracker) wurden auf einen dedizierten Knoten ausgelagert. Da jeder Clusterknoten über zwei (virtuelle) Prozessorkerne verfügt, wurde die Anzahl der Map- und Reduce-Tasks, die ein Knoten maximal gleichzeitig ausführen kann, auf 2 gesetzt.

Zur Evaluation wurde erneut der Datensatz DS2 aus Abbildung 4.9 mit ca. 1,4 Millionen Publikati-



(a) Vergleich der JobSN- und RepSN-Strategie.

(b) Laufzeit des RepSN-Verfahrens in Abhängigkeit der verwendeten Fenstergröße.

Abbildung 5.7: Vergleich der JobSN- und RepSN-Strategie für $p = 1$ und $w = 1.000$ unter Verwendung einer manuellen Bereichspartitionierung (links). Laufzeit und Speedup des RepSN-Verfahrens für verschiedene Fenstergrößen unter Verwendung von $\hat{r} = 1$ bzw. $\hat{r} = 8$ Reduce-Prozessen (rechts).

onsdatensätzen herangezogen. Zur Ähnlichkeitsbestimmung der zu vergleichenden Publikationsdatensätze wurde die Trigrammähnlichkeit des Abstracts und der Levenshtein-Ähnlichkeit berechnet. Paare mit einer durchschnittlichen Ähnlichkeit von mindestens $t = 0,7$ wurden als Duplikate betrachtet. Der Default-Blockschlüssel eines jeden Datensatzes für das Single-pass Sorted Neighborhood ergibt sich aus den ersten beiden Kleinbuchstaben des Publikationstitels.

5.5.1 Vergeich der JobSN- und RepSN-Strategie

In diesem Abschnitt wird zunächst ein Vergleich der beiden Implementierungen des Basisverfahren vorgenommen. Für eine zwischen einem und vier Knoten variierende Clustergröße und eine Fenstergröße von $w = 1.000$ werden dazu die benötigten Laufzeiten sowie die (relativen) Speedup-Werte beider Ansätze verglichen. Um eine Vergleichbarkeit für eine verschiedene Anzahl an Map- und Reduce-Tasks gewährleisten zu können, wird für jede Clusterergröße dieselbe manuelle Bereichspartitionierungsfunktion[6] verwendet, die die Menge aller Datensätze anhand ihrer Blockschlüssel in 10 möglichst gleich große Partitionen aufteilt. Aufgrund der Tatsache, dass alle Datensätze desselben Blockschlüssels demselben Reduce-Task zugewiesen werden, variiert, trotz der sorgfältigen Konstruktion der Blockschlüsselgenerierungsfunktion, die Anzahl der Datensätze pro Partition. Da der Titel vieler Publikation mit *"A "* oder *"The "* beginnt, sind für den konkreten Entity Resolution-Workflow die erste Partition (Blockschlüssel $<$ *"a!"*) sowie die neunte Partition (*"t"* $\leq$ Blockschlüssel $<$ *"ti"*) merklich größer als die übrigen Partitionen. Unabhängig von der Clustergröße werden stets 10 Reduce-Tasks verwendet, die von maximal $\hat{r} = 8$ Reduce-Prozessen auf vier Knoten bearbeitet werden. Dabei entspricht $\hat{r}$ der maximalen Anzahl an Reduce-Tasks, die gleichzeitig im Cluster ausführt werden können (vgl. Abschnitt 2.3.2).

Abbildung 4.13(a) zeigt die resultierenden Ausführungszeiten und Speedup-Werte beider Implementierungen. Die Konfiguration mit $\hat{m} = \hat{r} = 1$ entspricht der sequentiellen Ausführung des MapReduce-Programms durch einen Reduce-Prozess auf einem Knoten. Das Experiment mit der Konfiguration $\hat{m} = \hat{r} = 2$ wurde ebenfalls auf einem Knoten durchgeführt, allerdings wurden hier zwei Reduce-Prozesse verwendet, sodass beide Prozessorkerne genutzt werden konnten. Für die betrachteten Clus-

[6] Die nach Blockschlüssel sortierten Datensätze werden unter Verwendung der Intervallgrenzen $a!$, b, d, f, k, p, s, t und ti in 10 Partitionen aufgeteilt. Ein Datensatz mit einem Blockschlüssel, der lexikographisch kleiner als $a!$ ist (z. B. *"a "*), wird demnach zum ersten Reduce-Task gesendet.

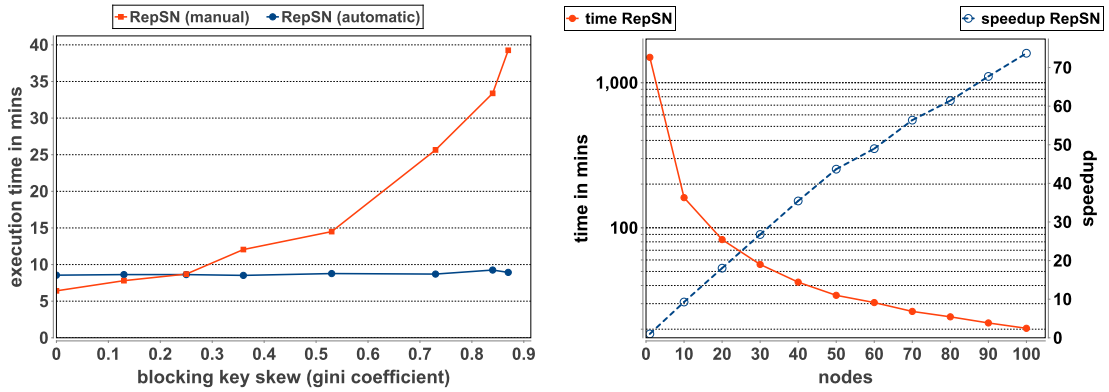
tergrößen von bis zu vier Knoten und acht Prozessorkernen skalieren beide Verfahren nahezu linear, beispielsweise konnte die Ausführungszeit des RepSN-Verfahrens von 8,3h auf ungefähr 1,2h reduziert werden. Die Laufzeit beider Verfahren variiert nur unmerklich. Unterschiede konnten nur für einen kleinen Parallelitätsgrad beobachtet werden, so benötigte das JobSN-Verfahren bei der sequentiellen Bearbeitung der Entity Resolution-Workflows fünf Minuten weniger als das RepSN-Verfahren. Für alle anderen Konfigurationen terminierte das RepSN-Verfahren etwas vor dem JobSN-Verfahren. Dies resultiert v. a. aus der Einsparung des zusätzlichen MapReduce-Jobs zur Evaluierung der Boundary Entities. Die nicht perfekt linearen Speedup-Werte von ungefähr 7 für acht Prozessorkerne lassen sich auf die Tatsache zurückführen, dass 10 Partitionen (Reduce-Tasks) variierender Größe (Last) von acht Reduce-Prozessen verarbeitet werden müssen, was keine optimale Auslastung der verwendeten Ressourcen erlaubt. Aufgrund des weitesgehend ähnlichen Laufzeitvehaltens beider Strategien wird in den folgenden Experimenten lediglich die RepSN-Strategie betrachtet.

In einem zweiten Schritt wurde das Skalierbarkeitsverhalten der (Single-pass) RepSN-Strategie in Abhängigkeit der verwendeten Fenstergröße betrachtet. Dazu wurden die Laufzeiten für zwischen $w = 10$ und $w = 1.000$ variierende Fenstergrößen unter Verwendung von $\hat{r} = 1$ (sequentiell) bzw. $\hat{r} = 8$ Reduce-Prozessen (vier Knoten) untersucht. Hierbei kam erneut die manuell definierte Funktion zur Partitionierung der Datensätze zum Einsatz. Abbildung 5.7(b) zeigt, dass in beiden Fällen die Ausführungszeiten linear mit der Fenstergröße (und damit der Anzahl der zu vergleichenden Paare) steigen. Der Speedup der parallelen Ausführung auf vier Knoten und acht Prozessorkernen verbessert sich von 4 für die kleinste betrachtete Fenstergröße $w = 10$ auf den Wert 7 für Fenster mit einer Größe $w \geq 400$. Diese Verbesserung ergibt sich aus dem günstigeren Verhältnis zwischen dem (nicht parallelisierbaren) MapReduce-Overhead und der Zeit, die für die paarweise Ähnlichkeitsberechnung benötigt wird.

5.5.2 Lastbalancierung durch automatische Bereichspartitionierung

In diesem Experiment wird der Einfluss der Datenungleichverteilung auf die Ausführungszeit untersucht. Dazu wird für verschiedene Grade an Datenungleichverteilung die Laufzeit der RepSN-Strategie mit und ohne Lastbalancierung ermittelt. Durch Manipulation der Blockschlüssel wird dabei der Grad der Datenungleichverteilung systematisch variiert, um (ähnlich zu Abschnitt 4.4.1) Blockgrößenverteilungen zu erzeugen, die einer Exponentialverteilung folgen. Bei einer festen Anzahl von $b = 8$ Blöcken ist die Anzahl der Datensätze des k-ten Blocks proportional zu $e^{-s \cdot k}$. Dabei ist $s \geq 0$ ein Parameter, um den Grad der Datenungleichverteilung zu justieren. Zur Quantifizierung der Ungleichverteilung der Blockschlüssel wird der *Gini*-Koeffizient $g = \frac{2 \cdot \sum_{i=1}^{b} i \cdot K_i}{b \cdot \sum_{i=1}^{b} K_i} - \frac{b+1}{b}$ verwendet. Dabei bezeichnet K_i die Anzahl der Datensätze des i-ten Blocks, wobei $i \in [1, b]$ und $K_i \leq K_{i+1}$. Ein Gini-Koeffizient von $g = 0$ drückt perfekte Gleichverteilung aus ($s = 0$), ein Wert von $g = 1$ drückt eine maximale Ungleichverteilung aus ($s \to \infty$). Das Experiment wurde in einem 4-Knoten Cluster (mit acht Map- und acht Reduce-Prozessen) mit einer Fenstergröße von $w = 100$ durchgeführt. Für die Konfigurationen mit manueller Bereichspartitionierung wurde eine simple Partitionierungsfunktion gewählt, die den i-ten von $b = 8$ Blöcken dem i-ten von $r = 8$ Reduce Tasks zuweist.

Abbildung 5.8(a) zeigt die resultierenden Ausführungszeiten. Erwartungsgemäß ist die manuelle Bereichspartitionierung anfällig für ungleichmäßig verteilte Blockschlüssel. Große Unterschiede in der Häuigkeitsverteilung der Blockschlüssel implizieren eine ungleichmäßige Auslastung der Reduce Tasks und erhöhen die Gesamtausführungszeit des MapReduce-Programms, die durch die Rechenzeit des zuletzt endenden Reduce-Tasks bestimmt wird. Im Gegensatz dazu ist bei Verwendung der automatischen Bereichspartitionierung eine gleichmäßige Auslastung der Reduce-Tasks garantiert. Da die Anzahl der Paarvergleiche nicht durch die Datenverteilung beeinflusst wird, konnte daher für alle untersuchten Grade an Datenungleichverteilung eine nahezu konstante Ausführungszeit beobachtet werden. Der zusätzliche Aufwand der KPM-Berechnung beträgt etwa 2,5 Minuten. Aus diesem Grund ist bei gleichverteilten Daten die manuelle Festlegung einer Partitionierungsfunktion effizienter als der vorgestellte



(a) Laufzeitvergleich bei manueller und automatischer Bereichspartitionierung ($p = 1$)

(b) Skalierbarkeitsverhalten bei automatischer Bereichspartitionierung ($p = 1$)

Abbildung 5.8: Laufzeiten für verschiedene Grade an Datenungleichverteilung unter Verwendung von $n = 4$ Knoten, $r = 8$ Reduce-Tasks und einer Fenstergröße von $w = 100$ (links). Laufzeit und Speedup-Werte des RepSN-Verfahrens bei automatischer Bereichspartitionierung und einer Fenstergröße von $w = 5.000$ (rechts).

Ansatz zur Lastbalancierung. Abbildung 5.8(a) zeigt, dass eine automatische Bereichspartitionierung für das untersuchte Beispiel ab einem Gini-Koeffizienten von $g > 0.25$[7] zu favorisieren ist. Bei der Verarbeitung von Realweltdaten mit nicht künstlichen generierten Blockschlüsseln ist aufgrund der Spracheigenschaften von einer deutlichen Datenungleichverteilung auszugehen. Vor diesem Hintergrund und angesichts der Tatsache, dass dem vergleichsweise geringen Aufwand zur Vorabdatenanalyse der Daten ein großes Potential zur Verbesserung der Laufzeit gegenübersteht, ist eine automatische, datengetriebene Bestimmung der Partitionierungsfunktion stets einer aufwändigen manuellen Festlegung vorzuziehen.

Zur Demonstration der Skalierbarkeit der RepSN-Strategie mit automatischer Bereichspartitionierung wurde die Fenstergröße auf $w = 5.000$ erhöht ($6,91 \cdot 10^9$ Kandidatenpaare) und die Laufzeit für eine zwischen 1 und 100 variierende Anzahl an Clusterknoten untersucht. Für dieses Experiment bildeten die ersten drei Buchstaben einer Publikation den Blockschlüssel, zur Quantifizierung der Ähnlichkeit wurde die Trigrammähnlichkeit der Publikationstitel bestimmt. Abbildung 5.8(b) zeigt die gemessenen Ausführungszeiten und Speedup-Werte. Die Ergebnisse zeigen, dass die zeitintensiven Paarvergleiche gleichmäßig auf die einzelnen Reduce-Tasks und Knoten verteilt werden kann. Die Gesamtausführungszeit konnte von 25h bei Verwendung eines einzigen Knotens auf 20 Minuten bei der Verwendung von 100 Knoten reduziert werden. Die entspricht einem Speedup von ≈74. Bis ca. 40 Knoten konnte ein nahezu lineares Speedup-Verhalten beobachtet werden. Für einen noch höheren Parallelitätsgrad (größere Knotenanzahl) wächst der Speedup nicht mehr linear, dies ist durch ein verringertes Lastbalancierungspotential für viele "kleine" Tasks zurückzuführen. Zudem führt die große Fenstergröße in Verbindung mit einer hohen Knotenanzahl (und damit einer hohen Anzahl von Reduce-Tasks) dazu, dass sehr viele Boundary Entities eines Reduce-Tasks i zum nachfolgenden Reduce-Task $i + 1$ und sogar zu weiteren Reduce-Tasks $j > i + 1$ repliziert werden müssen.

[7] Zum Vergleich: Die sorgfältig konstruierte Partitionierungsfunktion des Experimentes aus Abschnitt 5.5.1 hat einen Gini-Koeffizienten von $g = 0.13$. Eine einfache Aufteilung der sortierten Blockschlüssel in $r = 8$ Intervalle mit den Intervallgrenzen c, f, i, l, o, r und u resultiert in einem Gini-Koeffizienten von $g = 0.32$.

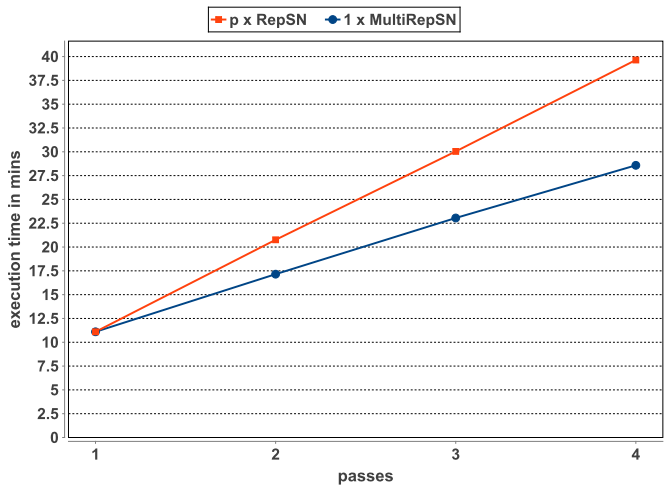


Abbildung 5.9: Vergleich der p-fachen Ausführung des Single-pass RepSN-Verfahrens mit der einmaligen Ausführung des MultiRepSN-Verfahrens ($w_i = 100$, automatische Bereichspartitionierung).

5.5.3 Multi-pass Sorted Neighborhood

Im Rahmen dieses Experiments wird die im Abschnitt 5.9 beschriebene Multi-pass-Implementierung mit der mehrfachen Ausführung des RepSN-Basisverfahrens verglichen. Dazu wird die Laufzeit beider Implementierungen für $p = 1,\ldots,4$ und $w_i = 100$ in einem Cluster bestehend aus $n = 4$ Knoten gemessen. Um den Einfluss von Datenungleichverteilung zu eliminieren, wird für beide Verfahren die im Abschnitt 5.5.2 beschriebene automatische Bereichspartitionierung eingesetzt. Zusätzlich zum Default-Blockschlüssel für den ersten Durchgang wurden drei weitere Blockschlüsselgenerierungsfunktionen eingesetzt: Nachname des Erstautors (zweiter Durchgang), Verkettung des ersten Buchstabens der Nachnamen aller Autoren (dritter Durchgang) sowie die Verkettung des ersten Buchstabens des Publikationstitels und des Nachnamens des Erstautors (vierter Durchgang).

Abbildung 5.9 zeigt die gemessenen Ausführungszeiten. Da eine Verdopplung von p zu einer Verdopplung der Anzahl der Paarvergleiche P führt, kann in beiden Fällen ein zur Anzahl der Durchgänge proportionales Wachstum der Ausführungszeit beobachtet werden. Durch die automatische Bereichspartitionierung ist zudem in beiden Fällen sichergestellt, dass alle Reduce-Tasks gleich ausgelastet sind. Die größere Ausführungszeit der mehrfachen Ausführung des Basisverfahrens ($p\times$ RepSN) ergibt sich dadurch, dass die gleiche Datenmenge p-mal aus dem verteilten Dateisystem eingelesen und geparst werden muss. Dem steht ein vergleichsweise geringer Overhead (vier- statt dreistellige `map`-Ausgabe-Schlüssel) der MultiRepSN-Strategie gegenüber. Für $p > 1$ schneidet die Multi-pass Implementierung deutlich besser ab. Diese Laufzeiteinsparung ist umso größer, je mehr Blockschlüsselgenerierungsfunktionen verwendet werden. Für das untersuchte Beispiel konnte eine Einsparung von 17% ($p = 2$), 23% ($p = 3$) bzw. 28% ($p = 4$) erreicht werden.

5.5.4 Datenqualität vs. Ausführungszeit

In diesem Abschnitt wird demonstriert, dass ein Multi-pass-Ansatz sowohl zur Verbesserung der Match-Qualität beitragen als auch zur einer Laufzeitverringerung führen kann. Da für die bisher betrachtete Datenquelle kein perfektes Match-Ergebnisses existiert, musste auf eine andere Datenquelle zurückgegriffen werden, um die Match-Qualität verschiedener Konfigurationen evaluieren zu können. Die

	Single-pass		Multi-pass (p=2)					
Window size	w=1,000		w_1=w_2=500		w_1=w_2=200		w_1=w_2=100	
#Comparisons	≈7.5·10^6		≈7.7·10^6		≈3.1·10^6		≈1.6·10^6	
Reduction Ratio	76.5%		75.7%		90.3%		95.1%	
Execution time [in s]	110		109		81		69	
	Pairs	TC	Pairs	TC	Pairs	TC	Pairs	TC
Precision	91.7%	90.7%	88.8%	82.9%	88.1%	82.7%	87.1%	81.8%
Recall	62.0%	71.0%	75.5%	88.5%	68.7%	85.9%	58.5%	77.2%
F-Measure	74.0%	79.6%	81.6%	85.6%	77.2%	84.3%	70.0%	79.5%

Abbildung 5.10: Vergleich der Match-Qualität und der Ausführungszeiten einer Single-pass- und dreier Multi-pass Konfigurationen mit verschiedenen Fenstergrößen. *Pairs* gibt die von der jeweiligen Konfiguration erzielte Match-Qualität alleinig auf Basis der als *Match* klassifizierten Paare an. *TC* gibt die Qualität nach Berechnung der transitiven Hülle an.

Datenquelle wurde aus der in [140] verwendeten Menge von Publikationsdatensätzen[8] abgeleitet, für die ein perfektes Match-Ergebnis vorliegt. Die resultierende Eingabedatenmenge besteht aus 5.343 Publikationsdatensätzen der Datenquelle *Scholar* für die jeweils eine korrespondierende Publikation in der duplikatfreien Datenquelle *DBLP* enthalten ist. Zunächst wurde das Single-pass RepSN-Verfahren mit einer Fenstergröße von $w = 1.000$ durchgeführt und die Ergebnisse mit denen dreier verschiedener Multi-pass Konfigurationen ($p = 2$) mit den Fenstergrößen $w_1 = w_2 = 500$, $w_1 = w_2 = 200$ und $w_1 = w_2 = 100$ verglichen. Im ersten Durchgang diente der erste Buchstabe des Namens des Erstautors als Blockschlüssel, im zweiten Durchgang wurde der erste Buchstabe des Publikationstitels verwendet. Datensätze mit einer Trigrammähnlichkeit des Titels von mindestens $0,75$ wurden als Duplikate klassifiziert. Die Evaluierung erfolgte in einem Cluster bestehend aus vier Knoten ($\hat{m} = \hat{r} = 8$) unter Verwendung der automatischen Bereichspartitionierung.

Der obere Bereich der Abbildung 5.10 zeigt die beobachteten Ausführungszeiten sowie die Anzahl der durchgeführten Paarvergleiche. Die *Reduction Ratio* gibt den Anteil der eingesparten Vergleiche zu der Anzahl aller Paare ($31,7 \cdot 10^6$) an. Der untere Teil der Abbildung 5.10 listet die Qualität (Precision, Recall und F-Measure) des Match-Ergebnisses für alle vier Konfigurationen an. Zusätzlich wird die Qualität nach Berechnung der transitiven Hülle der Match-Ergebnisse angegeben.

Die Single-pass-Konfiguration ($w = 1.000$) und die Multi-pass-Konfiguration mit $w = 500$ resultieren in einer vergleichbaren Anzahl an Paarvergleichen und demzufolge einer ähnlichen Laufzeit. In beiden Fällen führt die Berechnung der transitiven Hülle zur Entdeckung weiterer Duplikate und damit zu einer Verbesserung des Recalls. Der Multi-pass-Ansatz erreicht trotz der kleineren Fenstergöße einen deutlich höheren Recall und ein höheres F-Measure, da durch die Verwendung mehrerer Blockschlüssel auch Datensätze miteinander verglichen werden, die zuvor nicht innerhalb eines Fensters lagen. Eine weitere Verkleinerung der Fenstergröße auf $w = 200$ führt im Vergleich zur Single-pass Konfiguration mit $w = 1.000$ zu einer um ca. 25% verringerten Ausführungszeit. Auch für $w = 200$ resultiert die Berechnung der transitiven Hülle in einer deutlichen Verbesserung des F-Measures. Selbst mit der kleinsten verwendeten Fenstergröße von $w = 100$ lässt sich nach Anwendung der transitiven Hülle ein ähnliches F-Measure wie bei der Single-pass Konfiguration mit $w = 1.000$ erzielen. Dem steht gleichzeitig eine Verringerung der Ausführungszeit um 40% gegenüber. Die Ergebnisse zeigen, dass mittels eines Multi-pass Sorted Neighborhood-Ansatzes im Vergleich zum klassischen Sorted Neighborhood selbst für deutlich kleinere Fenstergrößen eine bessere Match-Qualität erzielt werden kann.

[8] `http://dbs.uni-leipzig.de/file/DBLP-Scholar.zip`

5.6 Zusammenfassung

In diesem Kapitel wurde untersucht, wie sich das Sorted Neighborhood-Verfahren mit MapReduce umsetzen lässt, um die Ähnlichkeitsberechnung der Kandidatenpaare über mehrere Prozessoren und Rechner parallelisieren zu können. Zu diesem Zweck wurden zwei MapReduce-Implementierungen namens JobSN und RepSN vorgestellt und deren Effizienz und Skalierbarkeit anhand einer Realweltdatenquelle in einem lokalen Cluster sowie einem Cluster in einer Cloud-Umgebung evaluiert. Um eine Skalierbarkeit der Verfahren gewährleisten zu können, wurde zusätzlich ein Ansatz zur datengetriebenen Bereichspartitionierung vorgestellt, der eine gleichmäßige Auslastung der verwendeten Ressourcen gewährleistet. Der Lastbalancierungsansatz erweitert die beiden Basis-Algorithmen und unterstützt gleichzeitig Multi-pass-Strategien mit unterschiedlichen Fenstergrößen pro Durchgang. Dies ermöglicht die Kapselung mehrerer Durchgänge des Basisverfahrens in einem MapReduce-Job und vermeidet somit im Vergleich zu einer mehrfachen Anwendung des Basis-Algorithmus das redundante Einlesen und Parsen der Eingabedaten.

6

Distanzberechnung in affinen Räumen

Dieses Kapitel widmet sich der MapReduce-Parallelisierung des kürzlich veröffentlichten HR³-Verfahrens [181] zur Bestimmung aller Paare von Punkten eines affinen Raumes, deren Minkowski-Distanz kleiner als ein vorgegebener Schwellwert ist. Dazu wird eine Diskretisierung des affinen Raumes vorgenommen und entsprechend der vorgegebenen Granularität der Diskretisierung eine effektive Einschränkung der Kandidatenmenge vorgenommen. Das HR³-Verfahren wird im Abschnitt 6.1 näher erläutert. Im darauffolgenden Abschnitt 6.2 wird zunächst eine direkte MapReduce-Umsetzung des Ausgangsverfahrens beschrieben. Aufbauend darauf wird eine Erweiterung vorgestellt, die auch im Falle ungleichmäßig im Raum verteilter Punkte, eine gleichmäßige Aufteilung der finalen Abstandsberechnungen auf die Cluster-Ressourcen gewährleistet und unnötige Replikation von Daten vermeidet. Eine Evaluation beider Verfahren erfolgt im Abschnitt 6.3.

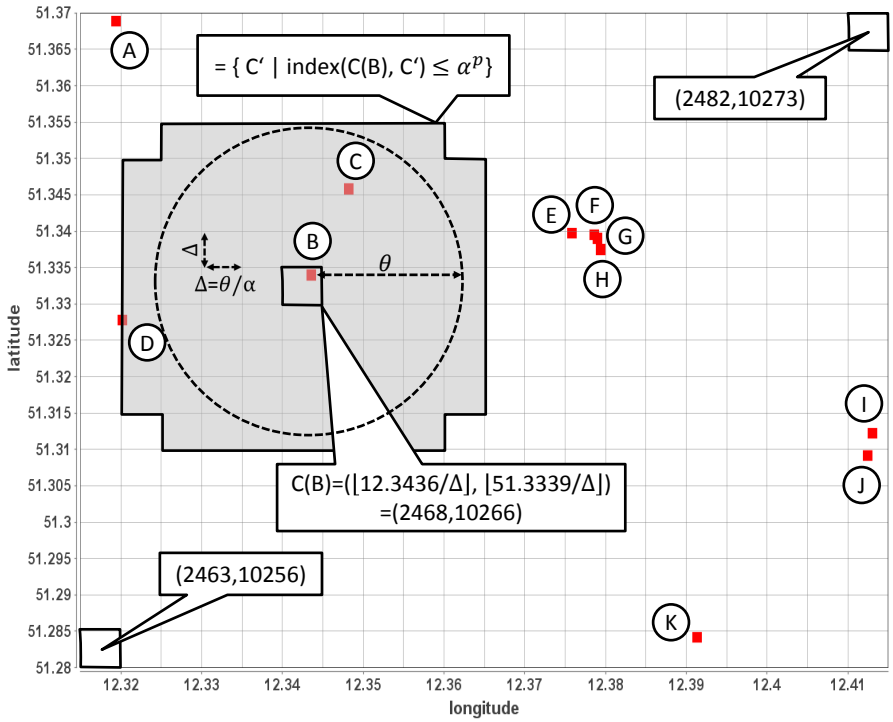
6.1 Einführung

Der HR³-Algorithmus [181] erlaubt die effiziente Bestimmung aller Paare von Punkten eines affinen Raumes deren Abstand einen vorgegebenen Schwellwert $\theta \in \mathbb{R}^+$ nicht überschreitet. Als Abstand zweier Punkte $x, y \in \mathbb{R}^n$ wird die Minkowski-Distanz der Ordnung $p \geq 1$ betrachtet, die als $\delta(x,y) = \sqrt[p]{\Sigma_{i=1}^{n} |x_i - y_i|^p}$ definiert ist. Für $p = 1$ bzw. $p = 2$ entspricht δ der Manhattan-Distanz bzw. Euklidischen Distanz. Techniken zur Beantwortung solcher Fragestellungen können beispielsweise bei der Umkreissuche zum Einsatz kommen. Darüber hinaus kann mithilfe von Abständen ein Blocking realisiert werden. Weisen die Datensätze einer Datenquelle u. a. $n \geq 1$ numerische Attributen desselben Wertebereiches auf, so kann jeder Datensatz in einen n-dimensionalen Raum abgebildet werden. Die Menge aller Datensatzpaare, deren Abstand unterhalb einen vorgegebenen Schwellwerts liegt, bildet die Kandidatenmenge, die zur weiteren Ähnlichkeitsberechnung herangezogen wird. Denkbare Einsatzszenarien sind die Gruppierung von Produkt- oder Personendatensätzen, deren Verkaufspreise bzw. Altersangaben um einen maximalen Betrag divergieren oder die Gruppierung von Orten anhand ihrer geographischen Koordinaten bei der Integration von Ortsdatensätzen aus heterogenen Quellen.

Die Grundidee des HR³-Algorithmus ist es, den Raum in Hypercubes[1] derselben Kantenlänge $\Delta = \theta/\alpha$ aufzuteilen. Die Granularität $\alpha \in \mathbb{N}\backslash\{0\}$ muss in Abhängigkeit der Dimensionalität n dabei so gewählt werden, dass $n(\alpha-1)^p > \alpha^p$ gilt. Für jeden nichtleeren Hypercube wird anschließend vermerkt, welche Punkte in ihm enthalten sind. Dazu wird jeder Datensatz $x = (x_1, \ldots, x_n)$ der Datenquelle R zu den diskreten Koordinaten $(\lfloor x_1/\Delta \rfloor, \ldots, \lfloor x_n/\Delta \rfloor)$ abgebildet. Die Menge aller möglichen Punkte mit denselben diskreten Koordinaten formt einen Hypercube (im Folgenden bezeichnet als Cube). Der Cube, welcher den Punkt x enthält, wird mit $C(x)$ bezeichnet. Des Weiteren wird der Vektor $(\lfloor x_1/\Delta \rfloor, \ldots, \lfloor x_n/\Delta \rfloor) \in \mathbb{N}^n$ als Koordinaten von $C(x)$ bezeichnet. Abbildung 6.1 illustriert die Diskretisierung anhand 11 verschiedener Ortsdatensätze. Der Punkt B mit den Koordinaten (12,3436; 51,3339) wird für $\theta = 0,02$ und $\alpha = 4$ auf die diskreten Koordinaten (2468, 10226) abgebildet.

Um die Menge der Punkte einzuschränken, mit denen ein Punkt x verglichen werden muss, werden

[1] *engl.* Hypercube = Hyperwürfel



Point	Place	Longitude	Latitude	C(Point)	
A	Auensee	12.3194	51.3688	(2463,10273)	Π_0
B	Palmgarten	12.3436	51.3339	(2468,10226)	
C	Red Bull Arena	12.3482	51.3457	(2469,10269)	
D	Baumwollspinnerei	12.3201	51.3277	(2464,10265)	
E	Zeitgeschichtliches Forum Leipzig	12.3758	51.3397	(2475,10267)	
F	St. Nicholas Church	12.3786	51.3395	(2475,10267)	
G	University of Leipzig	12.3790	51.3390	(2475,10267)	Π_1
H	City-Hochhaus Leipzig	12.3794	51.3375	(2475,10267)	
I	Monument to the Battle of the Nations	12.4130	51.3122	(2482,10262)	
J	Südfriedhof	12.4125	51.3091	(2482,10261)	
K	Agra Park	12.3913	51.2841	(2478,10256)	

Abbildung 6.1: Anwendung des HR³-Algorithmus für eine Datenquelle bestehend aus 11 Orten in Leipzig. Zur Bestimmung aller Punkte, deren Euklidische Distanz maximal $\theta = 0.02$ beträgt, wird zunächst eine virtuelle Aufteilung des Raumes in eine Menge von Hypercubes (hier Quadrate) mit einer Kantenlänge von $\Delta = \theta/4$ vorgenommen. Ein Hypercube wird durch seine Koordinaten $(c_1, ..., c_n)$ identifiziert. Die grau schattierten Zellen kennzeichnen $\{C' \mid index(C(B), C') \leq \alpha^p\}$, also die Menge aller Cubes, die potentiell Punkte enthalten, die mit dem Punkt B verglichen werden *müssen*.

Algorithmus 6.1: Der HR³-Algorithmus

Input : Dataset R, order p of Minkowski distance δ, distance threshold θ, space tiling granularity factor α, dimensionality n of the affine space
Output : $M = \{(x,y) \in R \times R \mid \delta(x,y) \leq \theta\}$

```
1  cubeMap ← empty map;
2  M ← {};
3  Δ ← θ/α;
4  foreach r ∈ R do
5      // Build Map<Cube,Set<Point>>
6      cubeMap(⌊r_1/Δ⌋,...,⌊r_n/Δ⌋) ←
7        cubeMap(⌊r_1/Δ⌋,...,⌊r_n/Δ⌋) ∪ {r};
8  foreach (c_1,...,c_n) ∈ cubeMap.keySet() do
9      foreach (c'_1,...,c'_n) ∈ getRelatedCubes((c_1,...,c_n)) do
10         cmp ← compareCubes((c_1,...,c_n), (c'_1,...,c'_n));
11         if cmp < 0 then
12             foreach x ∈ cubeMap.get((c_1,...,c_n)) do
13                 foreach y ∈ cubeMap.get((c'_1,...,c'_n)) do
14                     if computeDistance(x, y) ≤ θ then
15                         M ← M ∪ {(x,y)};
16         else if cmp = 0 then
17             buf ← {};
18             foreach y ∈ cubeMap.get((c_1,...,c_n)) do
19                 foreach x ∈ buf do
20                     if computeDistance(x, y) ≤ θ then
21                         M ← M ∪ {(x,y)};
22                 buf ← buf ∪ {y};
23 return M;

24 getRelatedCubes(C)
25     RC ← {C' | index(C,C') ≤ α^p} with
26         index(C,C') = { 0, if ∃i : |c_i − c'_i| ≤ 1 with i ∈ {1,...,n},
                         { Σ_{i=1}^{n} (|c_i − c'_i| − 1)^p else.        ;
27     return RC;

28 compareCubes(C, C')
29     for i ← 1 to n do
30         if c_i ≠ c'_i then
31             return c_i − c_j;
32     return 0;

33 computeDistance(x, y)
34     return ∛(Σ_{i=1}^{n} |x_i − y_i|^p);
```

mittels der Funktion in Formel 6.1 alle Cubes C' bestimmt, für die gilt: $index(C(x), C') \leq \alpha^p$.

$$index(C,C') = \begin{cases} 0, \text{ wenn } \exists i : |c_i - c'_i| \leq 1 \text{ mit } i \in \{1,...,n\}, \\ \sum_{i=1}^{n} (|c_i - c'_i| - 1)^p \text{ sonst.} \end{cases} \tag{6.1}$$

Anschließend müssen alle Punkte $y \neq x$ eines jeden Cubes C' mit x verglichen werden. In [181] wurde gezeigt, dass auf diese Weise alle Punkte y mit $\delta(x,y) \leq \theta$ gefunden werden und durch Erhöhung der Granularität α jede mögliche *Reduction Ratio* erreicht werden kann. Im Beispiel von Abbildung 6.1 kennzeichnet der grau schattierte Bereich alle Cubes, deren Punkte (hier C, D) mit B verglichen werden müssten, da Ihre Distanz zu B kleiner oder gleich θ sein *kann*. Insgesamt kann für das Beispiel eine Reduction Ratio von ca. 82% erreicht werden, da nur 10 statt 55 Distanzberechnungen durchgeführt werden müssen. Algorithmus 6.1 zeigt den Pseudocode des HR³-Algorithmus für eine einzelne Datenquelle. Eine Anpassung für zwei Datenquellen ist leicht möglich, wird im weiteren Verlauf dieses Kapitels jedoch nicht betrachtet.

6.2 MapReduce-Umsetzung des HR³-Algorithmus

Für große Datenquellen kann sowohl die wiederholte Berechnung der Funktion getRelatedCubes (Algorithmus 6.1 Zeile 24–27) als auch die große Anzahl an Distanzberechnungen (Algorithmus 6.1 Zeile 33–34) zu hohen Laufzeiten führen und eine Parallelverarbeitung erfordern. Im Falle, dass der Punktabstand lediglich als Kriterium zum Clustern der Datensätze verwendet wird, erhöht sich die Ausführungszeit zusätzlich um die Zeit, die für die eigentliche Ähnlichkeitsberechnung benötigt wird. In diesem Abschnitt wird zunächst eine MapReduce-Basisimplementierung des HR³-Algorithmus vorgestellt.

Algorithmus 6.2: MapReduce-Implementierung des HR^3-Algorithmus

```
1  map(k_in=unused, v_in = x)
2      Δ ← θ/α;
3      C(x) ← (⌊x_1/Δ⌋, ..., ⌊x_n/Δ⌋);
4      RC ← getRelatedCubes(C(x));
5      cid_1 ← getCubeId(C(x));
6      foreach C' ∈ RC do
7          cid_2 ← getCubeId(C');
8          if cid_1 ≤ cid_2 then
9              output(cid_1.cid_2.0, (x, 0));
10         else
11             output(cid_2.cid_1.1, (x, 1));

12 getCubeId((c_1, ..., c_n))
13     str ← empty string;
14     for i ← 1 to n do
15         if i > 0 then
16             str ← str.append("_");
17         str ← str.append(c_i);

18 // part = hash(cid_1,cid_2) mod r
19 // sort component-wise by entire key
20 // group by cid_1, cid_2
21 reduce(k_tmp=cid_1,cid_2.flag,
       v_tmp=list<(x, flag)>)
22     buf ← {};
23     if cid_1 = cid_2 then
24         foreach (x, flag) ∈ v_tmp do
25             foreach y ∈ buf do
26                 compare(x, y);
27             buf ← buf ∪ {x};
28     else
29         foreach (x, flag) ∈ v_tmp do
30             if flag=0 then
31                 buf ← buf ∪ {x};
32             else
33                 foreach y ∈ buf do
34                     compare(x, y);
```

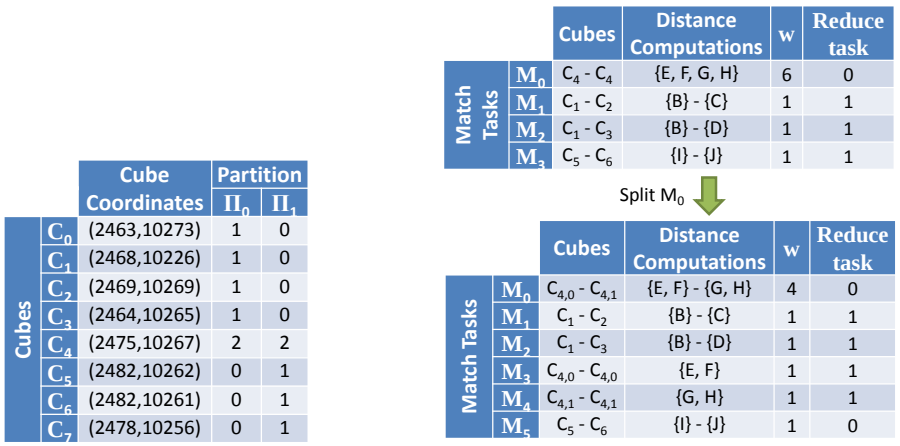
Anschließend wird eine Erweiterung des Verfahrens beschrieben, das Lastbalancierung garantiert und unnötige Datenreplikation vermeidet.

6.2.1 Basisimplementierung

HR^3 kann mittels eines einzigen MapReduce-Jobs implementiert werden. Die Grundidee ist, den Abstand zwischen Punkte zweier *verwandter* Cubes in einem Aufruf der `reduce`-Funktion zu berechnen. Zwei Cubes C, C' werden als verwandt bezeichnet, wenn $index(C, C') \leq \alpha^p$ gilt. Dazu bestimmt die `map`-Funktion für jeden Punkt x die Koordinaten des umgebenden Cubes $C(x)$ sowie die Menge der verwandten Cubes RC, die Punkte innerhalb der Maximaldistanz enthalten könnten (Algorithmus 6.2 Zeile 3–4). Für jeden Cube $C' \in RC$ gibt die `map`-Funktion ein Schlüssel-Wert-Paar mit dem zusammengesetzten Schlüssel ($\mathsf{cid}_1 \odot \mathsf{cid}_2 \odot \mathsf{flag}$) und dem Wert ($x$, flag) aus (Algorithmus 6.2 Zeile 6–11). Die ersten beiden (textuellen) Schlüsselkomponenten $cid_1 = \min\{C(x).id, C'.id\}$ und $cid_2 = \max\{C(x).id, C'.id\}$ identifizieren die beiden betroffen Cubes C und C' anhand ihrer Koordinaten. Das flag kennzeichnet, ob x zum ersten (flag $= 0$) oder zum zweiten Cube gehört (flag $= 1$).

Die Umverteilung der Schlüssel-Wert-Paare zu den Reduce-Tasks erfolgt durch Anwendung einer Hashfunktion auf die ersten beiden Schlüsselkomponenten cid_1 und cid_2. Dies weist alle Punkte aus $C(x) \cup C'$ demselben Reduce-Task zu. Die Sortierung der Schlüssel-Wert-Paare erfolgt komponentenweise anhand des vollständigen Schlüssels. Jeder Reduce-Task ruft die `reduce`-Funktion für alle eingehenden Schlüssel-Wert-Paare auf, deren erste beiden Schlüsselkomponenten übereinstimmen. In der `reduce`-Funktion erfolgt die Distanzberechnung der übergebenen Punkte. Durch die Sortierung der Schlüssel-Wert-Paare ist sichergestellt, dass alle Punkte des ersten Cubes vor den Punkten des zweiten Cubes von der `reduce`-Funktion verarbeitet werden. Dies ermöglicht einen effizienten Vergleich von Punkten verschiedener Cubes, da nur die Punkte des ersten Cubes im Hauptspeicher gepuffert werden müssen.

Der beschriebene Ansatz hat zwei fundamentale Schwächen. Da jeder Map-Task nur die Daten seiner Eingabepartition bearbeitet, hat er keine Kenntnis über die globale Datenverteilung. Dementsprechend wird jeder Punkt $|RC|$ mal repliziert, unabhängig davon, ob in jedem der verwandten Cubes ein Punkt enthalten ist oder nicht. Darüber hinaus ist der Ansatz, ähnlich wie die naive Umsetzung des Standard Blockings, anfällig gegenüber ungleich verteilten Punkten. Durch die inhärente quadratische Komplexität der paarweisen Distanzberechnung können moderate Schwankungen in der Anzahl der Punkte



Cubes		Cube Coordinates	Partition Π_0	Partition Π_1
Cubes	C_0	(2463,10273)	1	0
	C_1	(2468,10226)	1	0
	C_2	(2469,10269)	1	0
	C_3	(2464,10265)	1	0
	C_4	(2475,10267)	2	2
	C_5	(2482,10262)	0	1
	C_6	(2482,10261)	0	1
	C_7	(2478,10256)	0	1

(a) Cube Population Matrix

Match Tasks	Cubes	Distance Computations	w	Reduce task
M_0	C_4 - C_4	{E, F, G, H}	6	0
M_1	C_1 - C_2	{B} - {C}	1	1
M_2	C_1 - C_3	{B} - {D}	1	1
M_3	C_5 - C_6	{I} - {J}	1	1

Split M_0

Match Tasks	Cubes	Distance Computations	w	Reduce task
M_0	$C_{4,0}$ - $C_{4,1}$	{E, F} - {G, H}	4	0
M_1	C_1 - C_2	{B} - {C}	1	1
M_2	C_1 - C_3	{B} - {D}	1	1
M_3	$C_{4,0}$ - $C_{4,0}$	{E, F}	1	1
M_4	$C_{4,1}$ - $C_{4,1}$	{G, H}	1	1
M_5	C_5 - C_6	{I} - {J}	1	0

(b) Match-Task-Generierung

Abbildung 6.2: Cube Population Matrix für die Beispieldatenquelle aus Abbildung 6.1 unter Verwendung von $m = 2$ Map-Tasks (links). Match-Task-Generierung und Zuweisung zu $r = 2$ Reduce-Tasks mit/ohne Splitten von "großen" Match-Tasks (rechts).

der einzelnen Cubes zu stark variierenden Ausführungszeiten der bearbeitenden Reduce-Tasks führen. In Abhängigkeit von der Problemgröße, der Granularität der Aufteilung des Raumes und der Datenverteilung kann die Skalierbarkeit des beschriebenen Ansatzes auf einige wenige Knoten beschränkt sein. Aus diesem Grund wird im folgenden Abschnitt eine Erweiterung der Basisimplementierung vorgestellt, welche die angesprochenen Probleme behebt.

In [215] wurde kürzlich ein MapReduce-Algorithmus vorgestellt, der eine ähnliche Strategie verfolgt, um die Menge aller Punkte zu bestimmen, deren Abstand einen vorgegebenen Schwellwert nicht überschreitet. Der Ansatz nimmt eine (grobgranularere) Aufteilung des Raumes in Hypercubes der Kantenlänge θ vor und weist ein *benachbartes* Cube-Paar einem Reduce-Task zu, der die Distanzberechnung der enthaltenen Punkte vornimmt. Zusätzlich werden Filtertechniken eingesetzt, um die Anzahl der Distanzberechnungen von Punkten benachbarter Cubes, deren Abstand größer als der vorgegebene Schwellwert ist, zu reduzieren. Eine balancierte Aufteilung der Distanzberechnungen auf die zur Verfügung stehenden Reduce-Tasks wurde von den Autoren nicht betrachtet.

6.2.2 Lastbalancierung

Der im Folgenden vorgestellte Ansatz ist eine Variation des BlockSplit-Algorithmus zur effizienten Umsetzung des Standard Blockings (siehe Abschnitt 4.2). Erneut wird vorab ein Analyse-Job ausgeführt, der die Eingabedaten parallel scannt und die Anzahl der Punkte aller Cubes bestimmt. Ein zweiter anschließend ausgeführter MapReduce-Job nutzt diese Statistiken, um die Zuweisung von Punkten zu Reduce-Tasks so anzupassen, dass alle gesuchten Punktpaare ermittelt werden und eine gleichmäßige Auslastung der Reduce-Tasks gewährleistet ist.

Data Analysis Job: Der erste MapReduce-Job ermittelt in der Map-Phase für jeden Punkt die Koordinaten des umgebenden Cubes und summiert in der Reduce-Phase die Anzahl der Punkte pro Cube und Eingabepartition. Zu diesem Zweck gibt jeder Map-Task $0 \leq i < m$ ein Schlüssel-Wert-Paar (cid⊙i, 1) für jeden Punkt seiner Eingabepartition aus. Die Zuweisung der `map`-Ausgabe-Paare zu den Reduce-Tasks erfolgt auf Basis der ersten Schlüsselkomponente, zur Sortierung und Gruppierung wird der komplette Schlüssel verwendet. Die Ausgabe des Analyse-Jobs ist eine Cube Population Matrix (CPM) der Dimension $c \times m$, die die Anzahl der Punkte von c Cubes über m Eingabepartitionen spezifiziert. Für das

Beispiel aus Abbildung 6.1 und $m = 2$ Map-Tasks, die auf den Eingabepartitionen Π_0 und Π_1 arbeiten, würde die in Abbildung 6.2(a) gezeigte CPM berechnet werden.

Distance Computation Job: Der zweite MapReduce-Job verwendet dieselbe Anzahl an Map-Tasks und dieselbe Partitionierung der Eingabedaten. Zum Initialisierungszeitpunkt liest jeder Map-Task die im vorigen Analyse-Job berechnete CPM ein. Ähnlich zum Algorithmus 6.2 werden in der `reduce`-Funktion Paare verwandter Cubes verarbeitet. Zu diesem Zweck wird im Basisalgorithmus jeder Punkt von der `map`-Funktion $|RC|$-mal ausgegeben. Im Falle des Beispiels aus Abbildung 6.2(a) würden dementsprechend für den Punkt B 77-Schlüssel-Wert-Paare ausgegeben werden, was der Anzahl der grau schraffierten Cubes entspricht. Mithilfe der CPM kann hingegen sehr schnell festgestellt werden, dass nur drei der 77 Cubes besetzt sind. Dementsprechend kann die ausgegebene Datenmenge signifikant verringert werden. Insgesamt sind für den Datensatz B des Cubes $C(B) = C_1$ nur die Paare ($C_1 \odot C_2 \odot 0$, $B \odot 0$) sowie ($C_1 \odot C_3 \odot 0$, $B \odot 0$) auszugeben, um in der Reduce-Phase den Abstand zwischen B und C bzw. B und D berechnen zu können. Da B der einzige Punkt des Cubes C_1 ist, kann auf die Ausgabe des Paars ($C_1 \odot C_1.0$, $B \odot 0$) verzichtet werden. Des Weiteren können im Vergleich zur Basisimplementierung die länglichen Koordinaten der Cubes durch ganzzahlige Werte (Zeilennummern der CPM) ersetzt werden, was das umzuverteilende Datenvolumen sowie den Aufwand zur Sortierung und Gruppierung deutlich reduziert.

Vor Anwendung der `map`-Funktion auf den ersten Punkt der Eingabepartition erstellt jeder Map-Task eine Liste von Match-Tasks. Ein Match-Task ist ein Tripel (C_i, C_j, w), wobei C_i, C_j zwei verwandte Cubes sind und $w = |C_i| \cdot |C_j|$ (für $i \neq j$) bzw. $w = |C_i| \cdot (|C_i| - 1)/2$ (für $i = j$) die korrespondierende Workload, also die Anzahl der durchzuführenden Distanzberechnungen, der beiden Cubes ist. Die Gesamtworkload W entspricht der Summe der Workloads aller Match-Tasks. Um die Aufteilung der Match-Tasks auf die r Reduce-Tasks festzulegen, sortiert jeder Map-Task die Liste der Match-Tasks zunächst in absteigender Reihenfolge nach der Workload. Anschließend werden die Match-Tasks in dieser Reihenfolge den Reduce-Tasks nach einer Greedy-Heuristik zugeteilt. Dazu wird der aktuell betrachtete Match-Task dem Reduce-Task zugewiesen, der momentan die geringste Gesamtlast zu verrichten hat. Die für das Beispiel aus Abbildung 6.1 resultierenden Match-Tasks sind im oberen Teil der Abbildung 6.2(b) gezeigt ($r = 2$). Offenbar führt diese Strategie noch immer zu einer ungleichmäßigen Auslastung der beiden Reduce-Tasks (sechs zu drei Distanzberechnungen), da der Großteil der Gesamtworkload vom Match-Task $C_4 - C_4$ verursacht wird. Um dem zu begegnen, werden für jeden "großen" Match-Task $M = (C_i, C_j, w)$ mit $w > W/r$ beide Cubes entsprechend der Partitionierung der Eingabedaten in m Subcubes unterteilt. Infolgedessen wird (vor der Sortierung und Reduce-Task-Zuweisung) jeder große Match-Task M in eine Menge kleinerer Match-Tasks zerlegt, die jeweils ein Paar von Subcubes verarbeiten. Der untere Teil der Abbildung 6.2(b) zeigt die Aufteilung des großen Match-Tasks $(C_4, C_4, 6)$. Da dessen Workload $w = 6$ die durchschnittliche Workload eine Reduce-Tasks in Höhe von $9/2 = 4,5$ überschreitet, wird C_4 in zwei Subcubes $C_{4,0}$ (enthält E, F) und $C_{4,1}$ (enthält G, H) aufgeteilt. Daraus ergeben sich die drei Subtasks $(C_{4,0}, C_{4,0}, 1)$, $(C_{4,1}, C_{4,1}, 1)$ und $(C_{4,0}, C_{4,1}, 4)$, auf die sich alle Distanzberechnungen des originalen Match-Tasks aufteilen. Dies bewirkt für das Beispiel eine balancierte Workload beider Reduce-Tasks (fünf zu vier Distanzberechnungen).

Nach der initialen Match-Task-Generierung, erstellt jeder Map-Task i einen Index, der jeden Cube auf die Menge der entsprechenden Match-Tasks abbildet. Dabei müssen lediglich Cubes berücksichtigt werden, von denen Punkte in der i-ten Eingabepartition enthalten sind. Für jeden Punkt x bestimmt die `map`-Funktion den umgebenen Cube $C_j := C(x)$ und schlägt im Index nach, an welchen Match-Tasks C_j bzw. der Subcube $C_{j,i}$ beteiligt ist. Für jeden dieser Match-Tasks, gibt die `map`-Funktion ein Schlüssel-Wert-Paar (red_task⊙match_task⊙flag, x⊙flag) aus. Erneut kennzeichnet das flag, zu welchem der beiden an einem Match-Task beteiligten (Sub)Cubes der Punkt x gehört. Zum Routing der Schlüssel-Wert-Paare zu den Reduce-Tasks wird die erste Komponente des zusammengesetzten Schlüssels verwendet. Die Sortierung erfolgt komponentenweise anhand des vollständigen Schlüssels, die Gruppierung erfolgt auf Basis der ersten beiden Schlüsselkomponenten, sodass die `reduce`-Funktion für alle Punkte, deren Distanz im Rahmen eines Match-Tasks berechnet werden soll, aufgerufen wird. Abbildung 6.3 illustriert

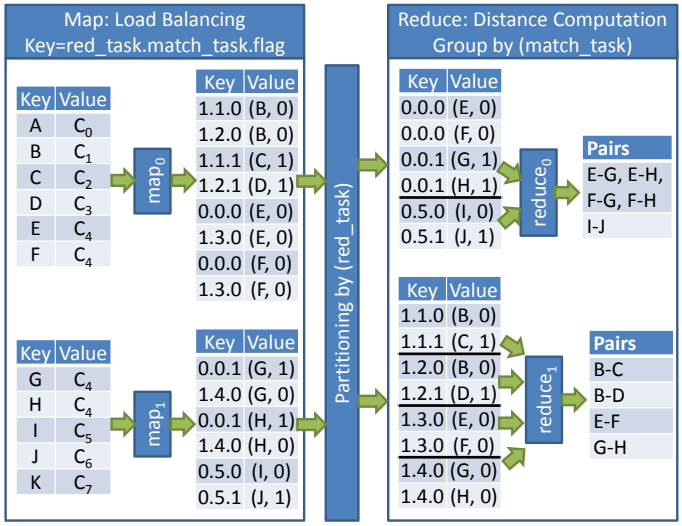


Abbildung 6.3: Datenfluss der MapReduce-Umsetzung des HR³-Algorithmus für das Beispiel aus Abbildung 6.1 mit Lastbalancierung entsprechend des Schemas aus Abbildung 6.2(b).

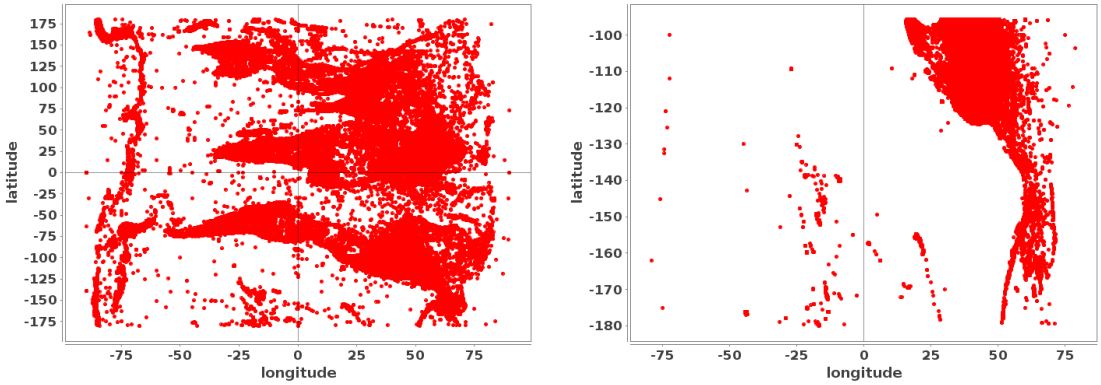
Dataset	Source	Size	Features
DS1	DBPedia	475,000	latitude, longitude
DS2	Linked Geo Data	500,000	latitude, longitude
DS3	Linked Geo Data	6,000,000	latitude, longitude

Abbildung 6.4: Zur Evaluation verwendete Datenquellen.

den Datenfluss der vorgestellten Implementierung für das fortlaufende Beispiel aus Abbildung 6.1.

6.3 Evaluation

In diesem Abschnitt erfolgt eine Evaluation der beiden vorgestellten MapReduce-Implementierungen des HR³-Algorithmus. Dazu werden in einem ersten Experiment die Laufzeiten beider vorgestellten Implementierungen für zwei Datensätze ähnlicher Größe aber unterschiedlicher Datenverteilung verglichen. In einem zweiten Experiment wird die Größe des Datensatzes systematisch von $0{,}5 \cdot 10^6$ bis $6 \cdot 10^6$ erhöht, um die Ausführungszeiten der Basisimplementierung und der Erweiterung mit Lastbalancierung in Abhängigkeit der Datenmenge untersuchen zu können. In allen Experimenten wurde die Euklidische Distanz zwischen den Punkten der Eingabedatenmenge berechnet. Abbildung 6.4 zeigt die verwendeten Datenquellen. Alle Datenquellen enthalten Ortsdatensätze (z. B. Einrichtungen, Gebäude, Parks und Orte), die durch den jeweiligen Ortsnamen und dessen Geokoordinaten gegeben sind. In [183] wurden noch weitere Datenquellen mit anderen numerischen Eigenschaften, wie z. B. Höheninformationen, verwendet. Zusätzlich wurde die MapReduce-Implementierung mit einer hybriden GPU/CPU-Implementierung sowie einer nebenläufigen Implementierung auf einem einzelnen Knoten unter Verwendung mehrerer Prozessorkerne verglichen. Dabei kamen jedoch verschiedene Rechner, Berechnungsstrategien und Programmiersprachen zum Einsatz, was einen objektiven Vergleich verhindert. Aus diesem Grund wird hier lediglich auf die MapReduce-Implementierung eingegangen.



(a) DS1 (b) DS2

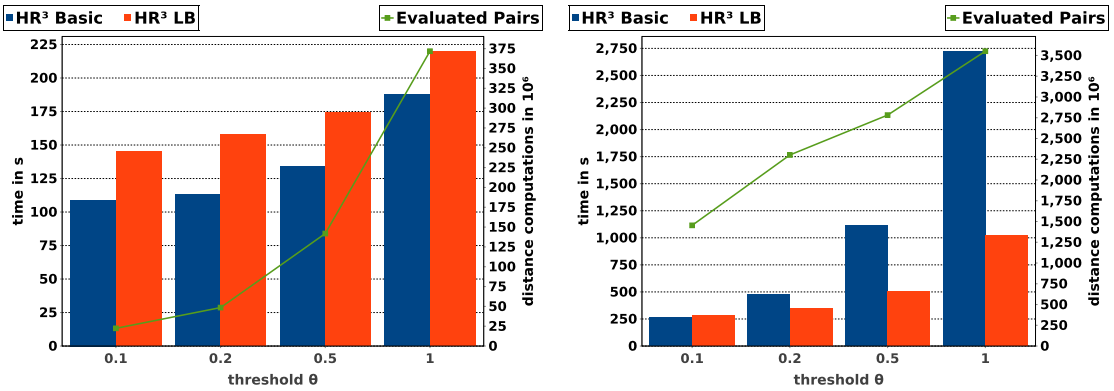
Abbildung 6.5: Datenverteilung der Datenquellen DS1 und DS2.

6.3.1 Laufzeitverhalten in Abhängigkeit der Datenverteilung

Zunächst wird die Laufzeit beider Implementierungen für die Datenquellen DS1 und DS2 und verschiedene Distanzschwellwerte $\theta \in \{0{,}1; 0{,}2; 0{,}5; 1\}$ und eine konstante Granularität $\alpha = 4$ untersucht. Beide Datenquellen enthalten 500.000 Datensätze. Abbildung 6.5 zeigt, dass sich bei DS2 die gleiche Anzahl an Punkten auf eine sehr viel kleinere Fläche verteilt. Diese höhere Punktdichte resultiert in einer sehr viel größeren Anzahl an Punkten pro Cube, was mit deutlich größeren Laufzeiten einhergeht. Aufgrund des höheren Optimierungspotentials ist also zu erwarten, dass die Implementierung mit Lastbalancierung für die Datenquelle DS2 zu deutlich besseren Ergebnissen führt. Die Experimente wurden erneut in einer Amazon-EC2 Umgebung mit 10 virtuellen Maschinen des Typs c1.medium mit je zwei virtuellen Kernen durchgeführt. Jeder Knoten wurde dahingehend konfiguriert, zwei Map- und Reduce-Tasks parallel ausführen zu können.

Abbildung 6.6 zeigt die resultierenden Laufzeiten. Mit steigendem Distanzschwellwert θ wächst die Anzahl der durchzuführenden Distanzberechnungen sowie das zwischen Map- und Reduce-Phase umzuverteilende Datenvolumen. Bedingt durch die im Vergleich zu DS2 geringe Punktdichte und damit deutlich kleinere Anzahl an Distanzberechnungen ist für die Datenquelle DS1 dabei nur ein leichter Anstieg der Ausführungszeit beider Verfahren zu beobachten. So verdoppelt sich bei der Erhöhung der Maximaldistanz von $\theta = 0{,}1$ auf $\theta = 0{,}2$ zwar die Anzahl der Paarvergleiche, jedoch wächst die Ausführungszeit des Basisverfahrens (HR^3 Basic) bzw. Lastbalancierungsverfahrens (HR^3 LB) lediglich um 4% bzw. 9%. Eine Verzehnfachung der Maximaldistanz auf $\theta = 1$ erhöht die Anzahl der Distanzberechnungen um den Faktor 17, die Ausführungszeiten wachsen jedoch lediglich um 72% bzw. 52%. Offenbar dominiert für DS1 die Zeit, die zum Einlesen, Umverteilen und zum Sortieren der Daten benötigt wird, die Gesamtausführungszeit. Aufgrund dieser Tatsache sowie den geringen absoluten Laufzeiten des Basisverfahrens in Höhe von maximal drei Minuten für $\theta = 1$ schneidet die Lastbalancierungsvariante für alle Distanzschwellwerte schlechter ab als das Basisverfahren.

Für die Datenquelle DS2 ist ein anderes Verhalten zu beobachten. Schon für $\theta = 0{,}1$ sind über 1,4 Millionen Distanzberechnungen durchzuführen, dies entspricht der 65-fachen Anzahl an Distanzberechnungen, die bei der Bearbeitung von DS1 mit $\theta = 1$ durchzuführen sind. Die Ausführungszeit beider Implementierungen beträgt im Vergleich zu DS1 jedoch jeweils nur etwa das Zweifache. Dies unterstreicht, dass die Gesamtausführungszeit bei der Bearbeitung von DS1 nicht durch die Distanzberechnungen dominiert wird. Für DS2 und $\theta = 0{,}1$ schneidet die Basisimplementierung noch etwa 7% besser ab als die Erweiterung mit Lastbalancierung. Mit steigendem Distanzschwellwert θ wächst die Anzahl der Distanzberechnungen so stark, dass durch die Lastbalancierung eine deutliche Beschleuni-



(a) DS1 (b) DS2

Abbildung 6.6: Laufzeitvergleich für die Datenquellen DS1 und DS2 für verschiedene Distanzschwellwerte θ und einer Granularität von $\alpha = 4$.

gung gegenüber der Basisimplementierung zu beobachten ist. Mit Lastbalancierung konnte eine Verbesserung der Laufzeit um 27%, 55% und 62% für $\theta = 0{,}2$, $\theta = 0{,}5$ und $\theta = 1$ erreicht werden. Erwartungsgemäß zahlt sich der Mehraufwand (Datenanalyse, Generierung von Match-Tasks, erhöhtes Datenvolumen) der Lastbalancierungsvariante für ungleichmäßig verteilte Daten ab einer hinreichend großen Anzahl an Distanzberechnungen aus, da dieser durch eine gleichmäßigere Aufteilung der Distanzberechnungen zu Reduce-Tasks amortisiert werden kann.

6.3.2 Laufzeitverhalten in Abhängigkeit der Datenmenge

In einem zweiten Experiment wurde das Laufzeitverhalten beider Implementierungen für eine wachsende Datenmenge untersucht. Dazu wurden Teilmengen der Datenquelle DS3 mit $0{,}5 \cdot 10^6$ bis $6 \cdot 10^6$ Datensätzen gebildet und die Menge aller Punktepaare bestimmt, deren Maximalabstand $\theta = 0{,}5$ beträgt. Erneut wurde eine Granularität von $\alpha = 4$ gewählt. Aufgrund des Hauptspeicherbedarfs bei der Generierung von Match-Task, des Speicherbedarfs zur Materialisierung der `map`-Ausgabe und der zu erwartenden langen Laufzeiten wurde für alle Konfigurationen eine fixe Cloud-Umgebung mit 20 Instanzen des Typs c1.xlarge mit acht virtuellen Prozessorkernen und 7 GB Hauptspeicher verwendet.

Abbildung 6.7 vergleicht die Laufzeiten des Basisverfahrens (HR³ Basic) und der Erweiterung mit Lastbalancierung (HR³ LB). Die Lastbalancierungsvariante schneidet für alle untersuchten Konfigurationen besser als das Basisverfahren ab. Zudem ist deutlich zu erkennen, das mit steigender Datenmenge und damit einer steigenden Anzahl von Distanzberechnungen die Differenz der Laufzeiten beider Implementierungen wächst. Konnte für $0{,}5 \cdot 10^6$ Datensätze lediglich eine Verbesserung von 3% erreicht werden, so beträgt für $6 \cdot 10^6$ Datensätze die Laufzeit der Lastbalancierungsvariante lediglich ein Viertel der Laufzeit des Basisverfahrens. Die Ergebnisse untermauern eindrucksvoll die Notwendigkeit des Einsatzes von Techniken zur Lastbalancierung bei der MapReduce-Parallelisierung von Distanzberechnungen zwischen Punkten großer Datenquellen. Es ist zu erwarten, dass sich der Performanzvorteil der Lastbalancierungsvariante mit steigender Dimensionalität der Daten oder einer geringeren Granularität der Raumaufteilung weiter verstärkt.

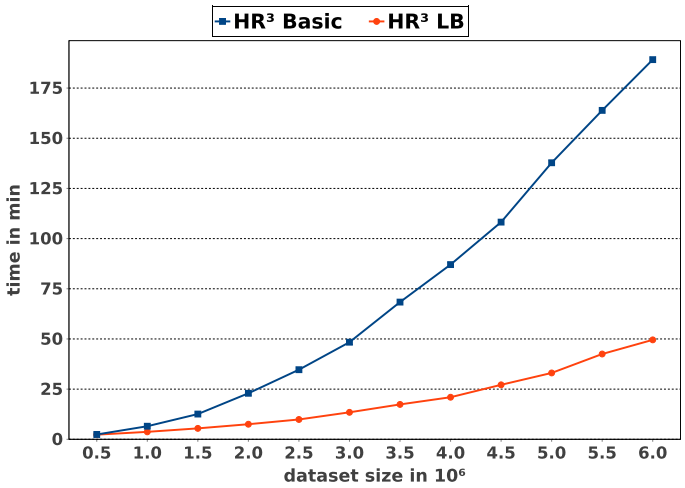


Abbildung 6.7: Laufzeitvergleich in Abhängigkeit der Datenmenge ($\theta = 0{,}5, \alpha = 4$).

6.4 Zusammenfassung

In diesem Kapitel wurde die Parallelisierung des HR^3-Verfahrens zur Bestimmung von Punkten eines mehrdimensionalen affinen Raumes, deren Minkowski-Distanz kleiner als ein vorgegebener Schwellwert ist, untersucht. Nach einer Vorstellung der Arbeitsweise des HR^3-Algorithmus wurde eine einfache MapReduce-Implementierung vorgestellt, die das Verfahren in einem einzigen MapReduce-Job umsetzt. Darauf aufbauend wurde eine Erweiterung vorgestellt, die eine gleichmäßige Aufteilung der Distanzberechnungen auf die genutzten Rechenressourcen gewährleistet. Dabei wurde eine Variation des im Abschnitt 4.2 vorgestellten BlockSplit-Algorithmus verwendet, die, basierend auf einer Vorabdatenanalyse, sogenannte Match-Tasks generiert und diese den zur Verfügung stehenden Reduce-Tasks so zuweist, dass deren gleichmäßige Auslastung gewährleistet ist. Die Evaluation der Verfahren in einer Cloud-Umgebung zeigte, dass beim Einsatz von Techniken zur Lastbalancierung für große Datenquellen und/oder ungleichmäßig verteilte Daten eine deutliche Reduktion der benötigten Rechenzeit erreicht werden kann.

Teil III

Weitere Teilprobleme

7

Integration maschineller Lernverfahren und Auswertung des Kartesischen Produktes

Nachdem in den vergangen Kapiteln Methoden zur effizienten und skalierbaren Umsetzung typischer Blocking-Verfahren betrachtet wurden, befasst sich dieses Kapitel mit der Fragestellung, wie eine Auswertung des Kartesischen Produktes zweier Datenquellen mithilfe des MapReduce-Programmiermodells über mehrere Prozessoren und Rechner verteilt werden kann. Parallel dazu wird gezeigt, wie die Integration maschineller Lernverfahren zur automatischen Klassifikation in den von Dedoop unterstützten MapReduce-Workflow erfolgt.

7.1 Maschinelle Lernverfahren

Um qualitativ hochwertige Ergebnisse erzielen zu können, ist bei bei der Duplikaterkennung in Realweltdatenquellen erforderlich, bei der Ähnlichkeitsberechnung mehrere Attribute und Ähnlichkeitsmetriken zu betrachten. Je mehr Attribute und/oder Ähnlichkeitsmetriken verwendet werden, umso schwieriger ist es, eine sinnvolle Strategie festzulegen, wie diese Ähnlichkeitswerte in eine Gesamtähnlichkeit kombinieren werden und daraus eine Klassifikationsentscheidung für ein Kandidatenpaar abgeleitet werden kann. Aus diesem Grund verwenden aktuelle Entity Resolution-Frameworks Techniken des maschinellen Lernens. Die grundlegende Idee ist, das Entity Resolution-Problem als ein Klassifikationsproblem zu betrachten bei dem jedes Kandidatenpaar anhand der zuvor ermittelten Ähnlichkeitswerte als *Match* oder *Non-Match* klassifiziert werden muss. Zu diesem Zweck wird zunächst ein Klassifikationsmodell gelernt, dass die *Matches* der vorgegebenen manuell annotierten Trainingsdaten bestmöglich von den *Non-Matches* trennt.

Abbildung 7.1 illustriert die Verwendung maschineller Lernverfahren für einen Entity Resolution-Workflow mit zwei Eingabedatenquellen R und S. Für die Trainingsphase werden Trainingsbeispiele $(a, b) \in R \times S$ als Eingabe benötigt, die mit der Information annotiert sein müssen, ob es sich um Duplikat handelt oder nicht. Aufgrund der interaktiven Einbindung des Nutzers eignet sich dieser Schritt nicht für eine Parallelisierung. Die manuelle Annotation von Trainingsbeispielen ist sehr zeitaufwändig und erfolgt in der Regel in einem Vorverarbeitungsschritt. Zunächst wird Ähnlichkeitsberechnung für die Trainingsdaten durchgeführt. Für jedes annotierte Trainingsbeispiel werden dabei, entsprechend der Workflow-Definition, k Ähnlichkeitsmetriken berechnet. Anhand der resultierenden Ähnlichkeitswerte und der Vorgabe, ob es sich bei diesem Datensatzpaar um einen *Match* handelt oder nicht, wird ein Klassifikationsmodell (z. B. Entscheidungsbaum, Support Vector Machine) generiert. Das gelernte Klassifikationsmodell wird anschließend auf alle Paare des Kartesischen Produktes $R \times S$ angewendet, um eine Klassifikationsentscheidung zu treffen. Wie in der Trainingsphase werden dazu dieselben k Ähnlichkeitsmetriken auf denselben Attributen berechnet. Die beschriebene Technik lässt sich problemlos auf Entity Resolution-Workflows mit Blocking-Schritt anwenden, dabei wird die Anwendung des Klassifikationsmodells jedoch auf eine Teilmenge von $R \times S$ beschränkt. Da beim Einsatz eines Blocking-Verfahren meist auch tatsächliche *Matches* aus der Menge der Kandidatenpaare entfernt werden, steht in diesem Kapitel die parallele Auswertung des Kartesischen Produktes im Vordergrund.

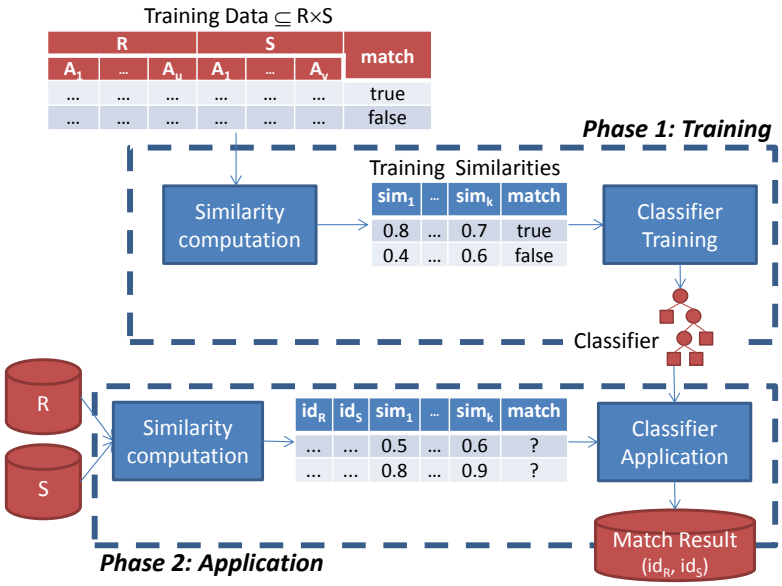


Abbildung 7.1: Schematischer Überblick eines Entity Resolution-Workflows mit maschinellem Lernverfahren zur automatischen Klassifikation der Kandidatenpaare.

Der hohen Effektivität lernbasierter Ansätze steht i. d. R. eine geringe Effizienz gegenüber. Eine Evaluation verschiedener Entity Resolution-Frameworks zeigte, dass die Laufzeiten lernbasierter Ansätze i. Allg. deutlich höher als die nicht lernbasierter Ansätze sind [140]. Nahezu alle untersuchten Verfahren waren nicht in der Lage, das Kartesische Produkt des größten untersuchten Problems (ca. 168 Mio Paarvergleiche) in einer annehmbaren Zeit auszuwerten und mussten nach einer Laufzeit von mehreren Tagen abgebrochen werden [140]. Wie später gezeigt wird, ist die Ursache dafür die extrem zeitaufwändige Berechnung *aller* k Ähnlichkeitswerte aller Datensatzpaare. Der Aufwand zum Training und zur Anwendung des Klassifikationsmodells ist demgegenüber nahezu vernachlässigbar. Im Gegensatz dazu, kann die Ähnlichkeitsberechnung bei schwellwert- oder regelbasierter Klassifikation oftmals bereits nach der Berechnung von $k' < k$ Ähnlichkeitswerten abgebrochen werden, z. B. wenn klar ist, dass der Mindestähnlichkeitsschwellwert nicht mehr erreicht werden kann. Da die Schnittstellen populärer Data Mining-Bibliotheken, wie *Weka* [99] oder *RapidMiner* [170], die vollständige Berechnung aller Ähnlichkeitswerte erfordern, können solche Optimierungen bei der Verwendung lernbasierter Klassifikationsverfahren i. d. R. nicht vorgenommen werden. Um trotzdem akzeptable Antwortzeiten erreichen zu können, ist eine Verteilung der Paarvergleiche und Klassifikationsentscheidungen auf mehrere Prozessoren und Rechner erforderlich.

Algorithmus 7.1 beschreibt Dedoops Classifier Training Job zum Lernen des Klassifikationsmodells (vgl. Abschnitt 3.3). Die Trainingsbeispiele haben die Form $(id_1, id_2, match)$. Dabei ist id_1 ein eindeutiger Identifikator eines Datensatzes der Datenquelle R, wohingegen id_2 einen Datensatz der Quelle S beschreibt. Jeder Map-Task liest diese Trainingsbeispiele während der Initialisierung ein und puffert diese im Hauptspeicher. Während der Map-Phase werden die durch id_1 bzw. id_2 beschriebenen Datensätze um die Attribute angereichert, die für die Berechnung der k Ähnlichkeitswerte benötigt werden. Dazu wird die `map`-Funktion auf jeden Datensatz von $R \cup S$ angewendet und geprüft, ob der Datensatz Bestandteil eines Trainingsbeispiels ist. Ist dies der Fall, so wird für jedes Trainingsbeispiel ein Schlüssel-Wert-Paar ($id_1 \odot id_2 \odot match$, entity) ausgegeben. Da zur Berechnung des Klassifikationsmodells alle Trainingsdaten als Eingabe benötigt werden, wird nur ein einziger Reduce-Task verwendet. Die `reduce`-Funktion wird für jedes Trainingsbeispiel aufgerufen. Innerhalb von `reduce` wird der Ähnlichkeitsvektor eines jeden Trainingsbeispiels berechnet und zu einer Datenstruktur im Hauptspeicher

Algorithmus 7.1: Training des Klassifikationsmodells

```
1  map_configure(JobConf jobConf)
2      labeledExamples ← empty map;
3      // Format: id1, id2, match
4      foreach line ∈ getTrainingDataPath(jobConf) do
5          labeledPair ← parseLine(line);
6          id1 ← labeledPair.getFirstId();
7          id2 ← labeledPair.getSecondId();
8          match ← labeledPair.getMatch();
9          if labeledExamples.containsKey(id1) then
10             labeledExamples.get(id1).add((id2, match));
11         else
12             list ← [];
13             list.add((id2, match));
14             labeledExamples.put(id1, list);
15         if labeledExamples.containsKey(id2) then
16             labeledExamples.get(id2).add((id1, match));
17         else
18             list ← [];
19             list.add((id1, match));
20             labeledExamples.put(id2, list);

21 map(k_in=unused, v_in=entity)
22     id1 ← entity.getId();
23     if labeledExamples.containsKey(id1) then
24         foreach (id2, match) ∈ labeledExamples.get(id1) do
25             if entity ∈ R then
26                 output(k_tmp=id1.id2.match, v_tmp=entity);
27             else
28                 output(k_tmp=id2.id1.match, v_tmp=entity);

29 reduce_configure(JobConf jobConf)
30     trainingData ← [];
31     classifier ← getClassifierClass(jobConf).newInstance();
32     classifier.setOptions(getClassifierOptions(jobConf));

33 // part: single reduce task only
34 // cmp: sort by entire key
35 // group: group by entire key
36 reduce(k_tmp=id1.id2.match, list(v_tmp)=list(entity))
37     e1 ← list(entity).first();
38     e2 ← list(entity).second();
39     simValues ← computeSimilarity(e1, e2);
40     trainingData.add((simValues, match));

41 reduce_close()
42     model ← classifier.buildClassificationModel(trainingData);
43     // Write model to DFS
```

hinzugefügt. Nach der Bearbeitung aller Eingabe-Daten, erfolgt die Berechnung des Klassifikationsmodells auf Basis der im Hauptspeicher gepufferten Ähnlichkeitsvektoren. Das Modell wird abschließend serialisiert, im verteilten Dateisystem gespeichert und mittels Hadoops Distributed Cache-Mechanismus in das lokale Arbeitsverzeichnis der Reduce-Tasks des Blocking-based Matching Jobs (vgl. Abschnitt 3.3) kopiert.

7.2 Parallele Auswertung des Kartesischen Produktes

Da die Anzahl der Trainingspaare i. Allg. sehr viel kleiner als die Menge der Paarvergleiche ist, ist die Anwendungsphase im Vergleich zur Lernphase sehr viel zeitaufwändiger. In den durchgeführten Experimenten betrug der Anteil des MapReduce-Jobs zum Lernen des Klassifikationsmodells weniger als 5% der Gesamtausführungszeit. Des Weiteren wird die Ausführungszeit der Anwendungsphase (Blocking-based Matching-Job) durch die Ähnlichkeitsberechnung dominiert, die Zeit zur Anwendung des Klassifikationsmodells ist nahezu vernachlässigbar (siehe Abschnitt 7.3.1). Der verbleibende Teil dieses Abschnitts widmet sich daher der Fragestellung, wie die paarweise Ähnlichkeitsberechnung aller Paare des Kartesischen Produktes $R \times S$ mittels MapReduce parallelisiert werden kann. Dafür werden zwei Ansätze vorgestellt, die die Menge aller Paare auf mehrere Tasks aufteilen und eine parallele Ähnlichkeitsberechnung und Klassifikation ermöglichen. Die MapSide-Strategie nutzt die bestehende Datenpartitionierung und realisiert die Ähnlichkeitsberechnung und Klassifikation vollständig während der Map-Phase. Die ReduceSplit-Strategie verteilt die auszuwertenden Paare gleichmäßig auf alle Reduce-Tasks und führt die Ähnlichkeitsberechnung und Klassifikation in der Reduce-Phase durch.

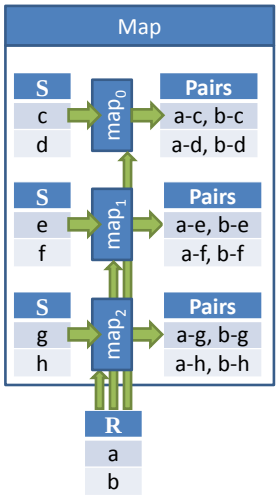


Abbildung 7.2: Ausführung der MapSide-Strategie für ein Beispiel mit $m = 3$ Map-Tasks. Die kleinere Datenquelle $R = \{a, b\}$ ist zusätzliche Eingabe eines jeden Map-Tasks.

7.2.1 MapSide: Replikation und Pufferung einer Datenquelle

Die MapSide-Strategie basiert auf der Idee des *Broadcast Joins* [32]. Die Grundidee ist es, die kleinere Relation (hier R) zur allen Map-Tasks zu broadcasten[1]. Dazu liest jeder Map-Task zum Initialisierungszeitpunkt die kleinere Datenquelle als zusätzliche Eingabe aus dem verteilten Dateisystem und puffert sie vollständig im Hauptspeicher. Die reguläre Eingabe des MapReduce-Jobs besteht ausschließlich aus der größeren Datenquelle (hier S). Innerhalb der `map`-Funktion wird ein Datensatz aus S mit *allen* (im Hauptspeicher gepufferten) Datensätzen aus R verglichen. Die MapSide-Strategie benötigt keine Reduce-Phase, die Anzahl der Reduce-Tasks wird statisch auf 0 gesetzt. In diesem Fall schreibt Hadoop die im lokalen Dateisystem der Tasktracker abgelegten `map`-Ausgabe-Schlüssel-Wert-Paare (die nun das finale Ergebnis des MapReduce-Jobs darstellen) automatisch in das verteilte Dateisystem und bricht die Ausführung des MapReduce-Jobs nach der Map-Phase ab. Dadurch kann der Overhead zur Datenumverteilung, Sortierung und Gruppierung sowie der Overhead zum Starten und Beenden der Reduce-Tasks eingespart werden. Kann die kleinere Datenquelle R nicht von allen Map-Tasks vollständig im Hauptspeicher gepuffert werden, so muss R in x Blöcke unterteilt und das beschriebene Basisverfahren x-mal wiederholt werden.

Abbildung 7.7(a) illustriert die MapSide-Strategie für ein Beispiel mit einer kleinen Datenquelle R (zwei Datensätze) und eine größere Datenquelle S (sechs Datensätze). Die Datensätze $a, b \in R$ werden von allen Map-Tasks während ihrer Initialisierung eingelesen und im Hauptspeicher gepuffert. Anschließend wird jeder Datensatz aus S (bestehend aus den Partitionen $\{c, d\}$, $\{e, f\}$ und $\{g, h\}$) mit allen gepufferten Datensätzen aus R verglichen. Da jeder Map-Task einen HDFS-Block bearbeitet und die Größe aller Blöcke (mit Ausnahme des letzten Blocks) übereinstimmt, entstehen keine Lastbalancierungsprobleme. Im Beispiel führt jeder der $m = 3$ Map-Tasks 4 von insgesamt 12 Vergleichen durch.

7.2.2 ReduceSplit: Partitionierung und Replikation beider Datenquellen

Die ReduceSplit-Strategie verlangt nicht, dass eine Datenquelle vollständig im Hauptspeicher gepuffert werden kann und ist deshalb für beliebig große Datenquellen einsetzbar. Die Ähnlichkeitsberechnung und Klassifikation erfolgt im Gegensatz zur MapSide-Strategie in der Reduce-Phase. Ziel ist dabei eine

[1] engl. to broadcast sth. = etwas übertragen, verbreiten, senden

Algorithmus 7.2: Implementierung der ReduceSplit-Strategie

```
1  map_configure(JobConf jobConf)
2      x ← getNumBlocksForR(jobConf);
3      y ← getNumBlocksForS(jobConf);
4      counter ← 0;

5  map(k_in=unused, v_in=entity)
6      if entity ∈ R then
7          i ← counter mod x;
8          for j ← 0 to y-1 do
9              output (k_tmp=i.j.R, v_tmp=(entity, R));
10     else
11         j ← counter mod y;
12         for i ← 0 to x-1 do
13             output (k_tmp=i.j.S, v_tmp=(entity, S));
14     counter ← counter + 1;

15 // part(key) = i + j · x mod r
16 // cmp: sort by entire key
17 // group: group by i.j
18 reduce(k_tmp=i.j.source, list(v_tmp)=list(entity, source))
19     buf ← {};
20     foreach (e_2, source) ∈ list(v_tmp) do
21         if e_2 ∈ R then
22             buf ← buf ∪ {e_2};
23         else
24             foreach e_1 ∈ buf do
25                 match(e_1, e_2);
```

gleichmäßige Aufteilung aller Paare auf die r Reduce-Tasks. Der ReduceSplit-Ansatz verfolgt die Strategie, beide Datenquellen in disjunkte Blöcke aufzuteilen – die Datenquelle R bzw. S wird dazu in x bzw. y Blöcke partitioniert. Anschließend werden alle x Blöcke von R mit allen y Blöcken von S abgeglichen. Das Gesamtproblem wird somit in $x \cdot y$ kleinere Teilprobleme zerlegt, die unabhängig voneinander parallel durch mehrere Reduce-Tasks bearbeitet werden können.

Jeder Datensatz aus R muss mit allen Datensätzen von insgesamt y Blöcken der Datenquelle S verglichen werden. Umgekehrt muss jeder Datensatz aus S mit allen Datensätzen von insgesamt x Blöcken der Datenquelle R verglichen werden. Dementsprechend werden während der Map-Phase y Schlüssel-Wert-Paare für jeden Datensatz der Datenquelle R und x Schlüssel-Wert-Paare für jeden Datensatz aus S ausgegeben. Die Schlüssel haben jeweils die Form (i⊙j⊙source), die Werte haben die Form (entity⊙source). Dabei bezeichnet $i \in [0, x-1]$ einen Blockindex der Datenquelle R, $j \in [0, y-1]$ einen Blockindex der Datenquelle S und source die Datenquelle des aktuell betrachteten Datensatzes. Bei der Anwendung der `map`-Funktion auf einen Datensatz der Datenquelle R wird zunächst bestimmt, in welchem Block sich der Datensatz befindet. Dazu wird zufällig ein Blockindex $i \in [0, x-1]$ des Datensatzes bestimmt. Alternativ kann der Map-Task einen Zähler führen, der bei jedem Aufruf der `map`-Funktion inkrementiert wird und den Blockindex mittels Zähler $\bmod x$ bestimmen. Anschließend werden y Schlüssel-Wert-Paare (i⊙j⊙R, entity⊙R) mit $j \in [0, y-1]$ ausgegeben. Analog dazu wird für jeden Datensatz der Quelle S zunächst der Blockindex j bestimmt. Anschließend werden x Schlüssel-Wert-Paare (i⊙j⊙S, entity⊙S) mit $i \in [0, x-1]$ ausgegeben.

Algorithmus 7.2 zeigt den Pseudocode der ReduceSplit-Strategie. Da die Anzahl der Blockpaare üblicherweise größer als die Anzahl der Reduce Tasks ist, kommt eine Partitionierungsfunktion $\mathsf{part}(i, j) = (i + j \cdot x) \mod r$ zum Einsatz, um die $x \cdot y$ Blockpaare gleichmäßig auf die r Reduce-Tasks aufzuteilen. Dabei werden alle Blockpaare nummeriert $(i + j \cdot x)$ und mithilfe der Modulo-Funktion fortlaufend den Reduce-Tasks zugewiesen. Alle Datensätze werden anhand des vollständigen Schlüssels sortiert. Zur Gruppierung werden lediglich die ersten beiden Komponenten der zusammengesetzten Schlüssel herangezogen, sodass die `reduce`-Funktion für alle Datensätze eines Blockpaars aufgerufen wird. Durch die dritte Schlüsselkomponente und die Sortierung der Datensätze ist zusätzlich sichergestellt, dass während der Verarbeitung der Datensätze eines Blockpaars alle Datensätze von R vor den Datensätzen aus S verarbeitet werden. Dies erlaubt es, alle $|R|/x$ Datensätze der Datenquelle R im Hauptspeicher zu puffern und anschließend jeden Datensatz aus S mit allen gepufferten Datensätzen abzugleichen.

Abbildung 7.3 illustriert die Ausführung der ReduceSplit-Strategie für dieselben Datenquellen, die bereits in Abbildung 7.7(a) verwendet wurden. Die Datenquelle R ist in $x = 2$ und die Datenquelle S in $y = 3$ Blöcke aufgeteilt. Die Vorgehensweise soll anhand des Datensatzes $a \in R$ verdeutlicht werden. Dieser wird dem ersten Block mit dem Index $i = 0$ zugewiesen und $y = 3$ mal mit den Schlüsseln

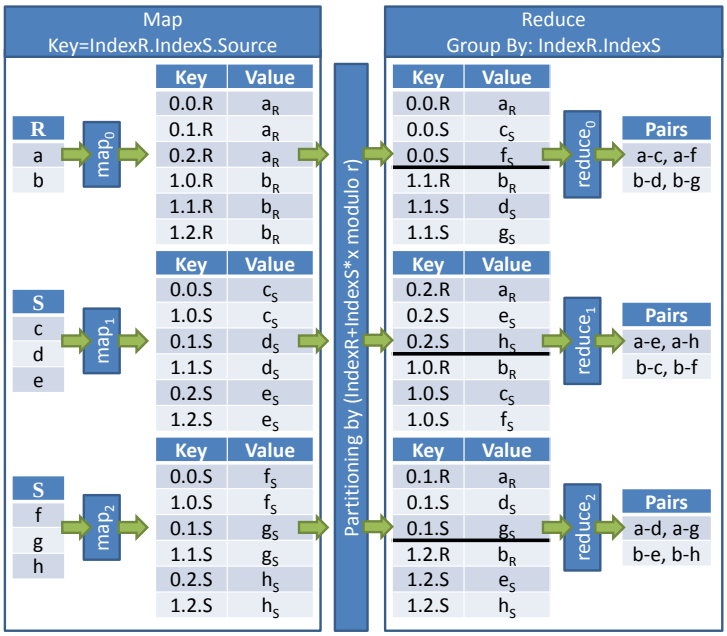


Abbildung 7.3: Ausführung der ReduceSplit-Strategie für die Beispieldatenquellen aus Abbildung 7.7(a) mit $m = 3$ Map- und $r = 3$ Reduce-Tasks. Die Datenquelle R ist in die $x = 2$ Blöcke $\{a\}$ und $\{b\}$ aufgeteilt, die Datenquelle S ist in die $y = 3$ Blöcke $\{c, f\}$, $\{d, g\}$ und $\{e, h\}$ aufgeteilt.

$(0.0.R)$, $(0.1.R)$, $(0.2.R)$ ausgegeben. Durch die Partitionierungsfunktion wird a zu allen $r = 3$ Reduce-Tasks gesendet und mit den Datensätzen der $y = 3$ Blöcke $\{c, f\}$, $\{d, g\}$ und $\{e, h\}$ der Datenquelle S verglichen.

Der wesentliche Vorteil der ReduceSplit-Strategie gegenüber dem MapSide-Ansatz ist, dass keine der Datenquellen vollständig im Hauptspeicher gepuffert werden muss. Ein Reduce-Task muss lediglich $|R|/x$ Datensätze puffern. Diesem Vorteil stehen die Kosten einer höheren Datenreplikation gegenüber. Während der Map-Phase werden insgesamt $|R| \cdot y + |S| \cdot x$ Schlüssel-Wert-Paare ausgegeben. Die MapSide-Strategie erfordert lediglich die Replikation der kleineren Datenquelle, dort werden nur $|R| \cdot m$ Datensätze repliziert. Dementsprechend sollte bei der Verwendung der ReduceSplit-Strategie die Anzahl x einerseits möglichst klein gewählt werden, um unnötige Datenreplikation zu vermeiden, anderseits jedoch so groß gewählt sein, dass jeder Reduce-Task $|R|/x$ Datensätze puffern kann. Des Weiteren sollte y ebenfalls möglichst klein gewählt werden, wobei $x \cdot y$ ein ganzzahliges Vielfaches von r sein sollte, um eine gleichmäßige Aufteilung aller Blockpaare auf die Reduce-Tasks gewährleisten zu können.

7.2.3 Self-Join: Auswertung aller Paare einer Datenquelle

Bisher wurde die Auswertung des Kartesischen Produktes zweier Datenquellen R und S betrachtet. Soll zur Duplikaterkennung die Ähnlichkeit aller Datensätze einer einzigen Datenquelle berechnet werden, ist eine Anpassung der vorgestellten Strategien erforderlich, um keine Paare mehrfach auszuwerten und keinen Datensatz mit sich selbst zu vergleichen (siehe Ausführungsschema in Abbildung 7.4).

MapSide: Nach wie vor liest jeder Map-Task die Datenquelle R ein und puffert sie im Hauptspeicher. Die eigentliche Eingabe des MapReduce-Jobs besteht ebenfalls aus R – jeder Map-Task wendet die `map`-Funktion auf jeden Datensatz einer Eingabepartition (HDFS-Block) von R an. In der `map`-Funktion wird der aktuell betrachtete Datensatz e mit allen gepufferten Datensätzen e' verglichen, deren Identifikator

$$
\begin{array}{c}
 \\ e_1 \\ e_2 \\ e_3 \\ \vdots \\ e_{|R|-1} \\ e_{|R|}
\end{array}
\begin{array}{c}
\begin{array}{cccccc} e_1 & e_2 & e_3 & \cdots & e_{|R|-1} & e_{|R|} \end{array} \\
\left(\begin{array}{cccccc}
 & & & & & \\
\times & & & & & \\
\times & \times & & & & \\
\vdots & \vdots & \ddots & & & \\
\times & \times & \times & \times & & \\
\times & \times & \times & \times & \times &
\end{array}\right)
\end{array}
$$

Abbildung 7.4: Schema der Ähnlichkeitsberechnung aller Paare einer Datenquelle R. Jedes "×" kennzeichnet einen durchzuführenden Paarvergleich. Insgesamt sind $(|R| \cdot (|R|-1))/2$ Paare auszuwerten.

lexikographisch kleiner als der Identifikator von e ist. Zur effizienten Bearbeitung ist es daher hilfreich, die gepufferten Datensätze nach dem Identifikator sortiert zu speichern.

ReduceSplit: Die Datenquelle R wird wiederum in x disjunkte Blöcke zerlegt. Für jeden Datensatz wird dazu in der `map`-Funktion ein Blockindex $i \in [0, x-1]$ bestimmt. Anschließend wird ein Schlüssel-Wert-Paar (i⊙i⊙⊥, entity) ausgegeben. Dies führt zur einer Gruppierung aller Datensätze eines Blocks in der Reduce-Phase. In der `reduce`-Funktion können die einzelnen Datensätze mittels der von Hadoop angebotenen Iterator-Schnittstelle schrittweise gelesen und im Hauptspeicher gepuffert werden. Nachdem ein Datensatz gelesen wurde und bevor er zu der Datenstruktur im Hauptspeicher hinzugefügt wird, muss er mit allen gepufferten Datensätzen verglichen werden. Auf diese Weise ist sichergestellt, dass alle Datensätze eines Blocks miteinander verglichen werden und unnötige Vergleiche vermieden werden. Neben dem erwähnten Schlüssel-Wert-Paar werden des Weiteren $x-1$ Schlüssel-Wert-Paare der Form (min{i,j}⊙max{i,j}⊙i, entity⊙i) ausgegeben, wobei gilt: $j \in \{0, \ldots, x-1\} \setminus \{i\}$. Dies erlaubt die Berechnung des Kartesischen Produktes der Blöcke i und j in der Reduce-Phase.

7.3 Evaluation

In diesem Abschnitt werden die vorgestellten Methoden zur automatischen Klassifikation mittels maschineller Lernverfahren sowie zur Auswertung des Kartesischen Produktes evaluiert. In einem ersten Experiment (Abschnitt 7.3.1) wird die Anzahl der berechneten Ähnlichkeitsmetriken variiert und deren Einfluss auf die resultierende Match-Qualität untersucht. Des Weiteren wird der Anteil der Ähnlichkeitsberechnung an der Gesamtausführungszeit des MapReduce-Jobs ermittelt. Im Abschnitt 7.3.2 erfolgt ein Vergleich der Ausführungszeiten des MapSide- und der ReduceSplit-Strategie für einen fixen Entity Resolution-Workflow. Abschließend wird im Abschnitt 7.3.3 die Skalierbarkeit der MapSide-Strategie untersucht.

Die Experimente wurden vollständig in einer Amazon EC2-Umgebung mit bis zu 50 virtuellen Maschinen des Typs *c1.medium* durchgeführt. Jede dieser VMs stellt 1.7GB und zwei virtuelle Prozessorkerne bereit. Jeder Knoten wurde so konfiguriert, dass maximal zwei Map- bzw. Reduce-Tasks gleichzeitig ausführen zu können. Die Master-Prozesse zur Verwaltung des verteilten Dateisystems und zur Orchestrierung der MapReduce-Jobs wurden auf eine weitere VM ausgelagert. Im Rahmen der Experimente wurde das Kartesische Produkt zweier bibliographischer Datenquellen DBLP (ca. 2.600 Datensätze) und Scholar (ca. 64.000 Datensätze) berechnet, für die ein perfektes Match-Ergebnis verfügbar ist [140]. Die Auswertung des Kartesischen Produktes beider Datenquellen resultiert in 168,1 Millionen Paaren, auf die das Klassifikationsmodell anzuwenden ist. Zum Lernen des Klassifikationsmodells wurde ein Entscheidungsbaum[2] sowie eine Support Vector Machine[3] der Weka 3.6.4-Bibliothek verwendet.

[2] weka.classifiers.trees.LMT (Default-Konfiguration)
[3] weka.classifiers.functions.LibSVM (Optionen -K 0 -C 10)

		Decision Tree				SVM			
		Time in mins	Time breakdown			Time in mins	Time breakdown		
			Sim	Apply	MR		Sim	Apply	MR
Matcher	①	11.6	88.4%	3.4%	8.2%	11.6	88.9%	2.7%	8.4%
	① - ②	19.0	92.7%	2.3%	5.0%	19.1	93.0%	1.8%	5.2%
	① - ③	25.1	94.1%	1.8%	4.1%	24.4	94.5%	1.5%	4.0%
	① - ④	42.4	96.5%	1.1%	2.4%	42.9	96.8%	0.9%	2.4%
	① - ⑤	46.4	96.8%	1.1%	2.1%	46.2	97.0%	0.9%	2.1%
	① - ⑥	51.3	97.1%	1.0%	1.9%	51.7	97.2%	0.8%	2.0%

Abbildung 7.5: Ausführungszeit der MapSide-Strategie ($n = 10$ Knoten, $m = 100$ Map-Tasks) und Aufschlüsselung in drei Kategorien. Die Anzahl der berechneten Ähnlichkeitswerte variiert zwischen 1 und 6 (siehe auch Abbildung 7.6).

7.3.1 Analyse der Ausführungszeit und der Match-Qualität

Im ersten Experiment wird für eine variierende Anzahl berechneter Ähnlichkeitsmetriken die Ausführungszeit des MapSide-Verfahrens sowie die resultierende Match-Qualität evaluiert. Dazu wurden drei Ähnlichkeitsmetriken (TFIDF-Ähnlichkeit, Trigram-Ähnlichkeit, Jaccard-Ähnlichkeit) auf zwei verschiedene Attribute (Publikationstitel und Autoren) angewendet. Zur automatischen Klassifikation kamen zwei verschiedene Verfahren des maschinellen Lernens (Entscheidungsbaum und SVM) zum Einsatz. Die Experimente wurden in einem Cluster bestehend aus $n = 10$ Knoten durchgeführt. Zur Berechnung wurden $m = 100$ Map-Tasks gebildet, um Computational Skew-Effekte abzuschwächen.

Abbildung 7.5 schlüsselt die Ausführungszeit für jede der sechs betrachteten Kombinationen in die drei Kategorien Ähnlichkeitsberechnung, Anwendung des Klassifikationsmodells und verbleibender MapReduce-Overhead auf. Die Ergebnisse zeigen, dass die Ausführungszeit sowohl für den Entscheidungsbaum als auch für die SVM eindeutig durch die Ähnlichkeitsberechnung dominiert wird. In Abhängigkeit der Anzahl k berechneter Ähnlichkeitswerte nimmt die Ähnlichkeitsberechnung zwischen 88% ($k = 1$) und 97% ($k = 6$) der Gesamtausführungszeit des MapReduce-Jobs ein. Für das untersuchte Beispiel ist die Anwendung des durch die SVM gelernten Klassifikationsmodells geringfügig schneller als die Auswertung eines gelernten Entscheidungsbaums. Um die inverse Dokumenthäufigkeit eines Tokens bestimmen zu können, muss zur Berechnung des ersten und des vierten Ähnlichkeitswerts (siehe Abbildung 7.6) auf einen materialisierten IDF-Index[4] zugegriffen werden. Die Berechnung des vierten Ähnlichkeitswerts (zusätzlich zu den ersten drei) führt deswegen zu einem deutlichen Anstieg der Ausführungszeit. Die Werte des Autor-Attributs sind i. Allg. deutlich kürzer als die Publikationstitel, zusätzlich ist für viele Datensätze der Scholar-Datenquelle keine Autor-Information vorhanden. Aus diesen Gründen führt die Hinzunahme des fünften und sechsten Ähnlichkeitswerts im Vergleich zum zweiten und dritten Ähnlichkeitswert nur zu einer leichten Erhöhung der Ausführungszeit.

Neben der Laufzeit wurde ebenso die Match-Qualität der einzelnen Konfigurationen untersucht (siehe Abbildung 7.6). Da die Match-Qualität massiv von der Qualität der Trainingsdaten abhängt, wurde die in [140] vorgeschlagene Ration-Strategie zur Selektion der Trainingsdaten verwendet. Dazu wurden für jede Konfiguration (k Ähnlichkeitswerte und Entscheidungsbaum/SVM) 500 Paare mit einem Mindestähnlichkeitsschwellwert (TFIDF-Ähnlichkeit) von $0,4$ selektiert, wobei mindestens 40% *Matches* und *Non-Matches* darstellten. Diese Vorgehensweise wurde 10-mal wiederholt, Abbildung 7.6 weist die gemittelten Werte aus. Die Ergebnisse zeigen, dass i. Allg. die Verwendung mehrerer Ähnlichkeitsmetriken und deren Anwendung auf verschiedene Attribute zu einer höheren Match-Qualität führt. So konnte durch Hinzunahme des Autor-Attributs (vierter Ähnlichkeitswert) eine Erhöhung des F-Measures um

[4] Der IDF-Index wurde in einem Vorverarbeitungsschritt durch einen separaten MapReduce-Job erzeugt. Die dafür benötigte Ausführungszeit ist nicht in den ausgewiesenen Ausführungszeiten enthalten. Der Index liegt im *MapFileOutputFormat* vor und wird jedem Knoten mittels Hadoops Distributed Cache-Mechanismus zugänglich gemacht.

		Decision Tree	SVM
Matcher	①	75.4%	75.4%
	① - ②	82.1%	82.6%
	① - ③	82.0%	82.4%
	① - ④	85.9%	85.2%
	① - ⑤	87.1%	85.2%
	① - ⑥	86.2%	85.8%

① - TFIDF (title)
② - Trigram(title)
③ - Jaccard(title)
④ - TFIDF(authors)
⑤ - Trigram(authors)
⑥ - Jaccard(authors)

Abbildung 7.6: Vergleich der durchschnittlichen Match-Qualität (F-Measure) für zwei Verfahren des maschinellen Lernens und eine variierende Anzahl berechneter Ähnlichkeitswerte.

3-4% erreicht werden. Andererseits kann eine zu hohe Zahl berechneter Ähnlichkeitsmaße aufgrund von Überanpassung zu einer Verringerung der Match-Qualität führen. Dies kann am Beispiel des sechsten Ähnlichkeitswertes, der als Eingabe für die Erzeugung des Entscheidungsbaums dient, beobachtet werden. Trotzdem kann i. Allg. festgestellt werden, dass sich die Verwendung mehrerer Ähnlichkeitsmetriken und die Betrachtung verschiedener Attribute positiv auf die erzielte Match-Qualität auswirkt. Da die Anzahl der berechneten Ähnlichkeitswerte die Gesamtausführungszeit dominiert, unterstreicht dies die Bedeutung einer effizienten Parallelisierung der Ähnlichkeitsberechnung. Für das untersuchte Beispiel konnte mit dem Entscheidungsbaum ein um etwa 2% besseres F-Measure erreicht werden. Andererseits scheint die SVM weniger sensitiv gegenüber der Wahl der Ähnlichkeitsmetriken und Attribute zu sein. Für die SVM wurden für die gleichen Datenquellen bereits etwas bessere Ergebnisse berichtet [140]. Es ist zu erwarten, dass durch Tuning der internen SVM-Parameter eine weitere Qualitätsverbesserung erreicht werden kann.

7.3.2 Vergleich der MapSide- und ReduceSplit-Strategie

Im zweiten Experiment wird die Laufzeit der MapSide- und ReduceSplit-Strategie in einer fixen Cloud-Umgebung bestehend aus $n = 10$ Knoten verglichen. Als Eingabe für die Klassifikation durch einen analog zum vorigen Abschnitt gelernten Entscheidungsbaum dienen alle sechs Ähnlichkeitswerte (siehe Abbildung 7.6).

Abbildung 7.7(a) zeigt die Ausführungszeiten der MapSide-Strategie für eine variierende Anzahl $m = k \cdot 20$ an Map-Tasks mit $1 \leq k \leq 5$. In dem verwendeten MapReduce-Cluster, bestehend aus 10 Knoten können, maximal 20 Map-Tasks parallel bearbeitet werden. Durch eine Erhöhung der Anzahl der Map-Tasks steigt zwar der MapReduce-Overhead zur Verwaltung der einzelnen Tasks, jedoch können Computational Skew-Effekte reduziert werden. Diese werden durch die Ähnlichkeitsberechnung von Attributwerten unterschiedlicher Länge sowie durch heterogene Hardware in Cloud-Infrastrukturen verursacht. Aufgrund des hohen Anteils der Ähnlichkeitsberechnung an der Gesamtausführungszeit können Zeitunterschiede bei der Bearbeitung von Datenmengen derselben Größe einen wesentlichen Einfluss auf die Gesamtausführungszeit haben. So konnte die Gesamtausführungszeit bei der Verwendung von $m = 100$ statt $m = 20$ Map-Tasks um ca. 20% reduziert werden.

Zur Evaluation der ReduceSplit-Strategie wurde hingegen eine fixe Anzahl an $m = 20$ Map- und $r = 20$ Reduce-Tasks verwendet, da $x \cdot y$ Blockpaare definiert werden, die nach dem Round Robin-Prinzip durch 20 Reduce-Tasks bearbeitet werden. Durch Wahl von x und y können problemlos mehr als 20 Blockpaare gebildet werden, was der Abschwächung der Computational Skew-Effekte durch die Verwendung einer höheren Anzahl an Map-Tasks bei der MapSide-Strategie gleichkommt. Im Experiment wurden x und y zwischen 1 und 20 variiert. Abbildung 7.7(b) zeigt die Ausführungszeiten aller evaluierten Kombinationen. Zusammenfassend konnte für den betrachteten Fall Folgendes festgestellt werden. Für $x \cdot y < r$ ist die Ausführungszeit deutlich höher als für die übrigen Kombinationen, da nicht alle

m	time in mins
20	63.5
40	54.9
60	52.5
80	52.0
100	50.8

(a) MapSide

		Scholar | y				
		1	5	10	15	20
DBLP | x	1	871.4	167.8	91.9	70.5	54.0
	5	186.5	75.8	59.9	53.7	49.0
	10	101.4	64.0	55.6	57.5	54.3
	15	71.9	54.4	54.6	56.7	53.5
	20	57.3	56.7	59.2	60.6	58.0

(b) ReduceSplit

Abbildung 7.7: Ausführungszeiten des MapSide-Verfahrens für eine variierende Anzahl m an Map-Tasks (links). Ausführungszeiten (in Minuten) der ReduceSplit-Strategie für verschiedene Kombinationen von x und y. Dabei gibt x bzw. y an, in wie viele Blöcke die Datenquelle DBLP bzw. Scholar aufgeteilt wird (rechts).

Cluster-Ressourcen genutzt werden. Des Weiteren zeigen die Ergebnisse, dass die Anzahl der Blöcke der kleineren Datenquelle kleiner als die Anzahl der Blöcke der größeren Datenquelle sein sollte ($x < y$). So schneidet die Konfiguration ($x = 5$, $y = 20$) besser ab als die Konfiguration ($x = 20$, $y = 5$), da im letzteren Fall die größere Datenquelle Scholar 20-mal repliziert werden muss. Die dritte Beobachtung ist, dass eine Erhöhung von y bei konstanten x i. Allg. zu einer Reduzierung der Ausführungszeit führt, da Computational Skew-Effekte abgeschwächt werden. Die einzige beobachtete Ausnahme ist der Übergang von ($x = 10$, $y = 10$) zu ($x = 10$, $y = 15$). Ursache dafür ist, dass im ersten Fall $10 \cdot 10 = 100$ Blockpaare erzeugt werden, die gleichmäßig auf die $r = 20$ Reduce-Tasks aufgeteilt werden können. Im zweiten Fall werden hingegen 150 Blockpaare generiert, bei deren Bearbeitung durch 20 Reduce-Tasks gegen Ende der Berechnung nur noch 10 von 20 Reduce-Tasks ein Blockpaar zu bearbeiten haben. Daraus kann die letzte Beobachtung abgeleitet werden, dass $x \cdot y$ stets ein ganzzahliges Vielfaches von r sein sollte. Zusätzlich zu den ausgewiesenen Ausführungszeiten wurden Konfigurationen mit $x, y \in [1, 100]$ evaluiert. Für größere Werte von x und y konnten trotz des höheren MapReduce-Overheads und der erhöhten Datenreplikation relativ konstante Ausführungszeiten, ähnlich zu den besten Ergebnissen, beobachtet werden. Dies zeigt, dass eine balancierte Zuweisung von Paaren zu Reduce-Tasks und eine Behandlung von Computational Skew-Effekten wichtiger sind als eine Minimierung des umverteilten Datenvolumens.

Mit der besten Konfiguration ($x = 5$, $y = 20$) konnte eine im Vergleich zur MapSide-Strategie geringfügig bessere Ausführungszeit erreicht werden. Eine optimale Wahl von x und y ist schwierig, da sie vom Parallelitätsgrad (Anzahl der Reduce-Tasks) und den Charakteristika der Datensätze abhängig ist. Sofern die kleinere Datenquelle von jedem Map-Task im Hauptspeicher gepuffert werden kann, ist die MapSide-Strategie der ReduceSplit-Strategie vorzuziehen.

7.3.3 Skalierbarkeit

Im letzten Experiment wurde das Skalierbarkeitsverhalten der Bearbeitung eines lernbasierten Entity Resolution-Workflows untersucht. Da eine systematische Evaluierung der ReduceSplit-Strategie aufgrund der Abhängigkeit von x und y zum verwendeten Parallelitätsgrad schwierig ist, wurden lediglich die Laufzeiten der MapSide-Strategie für eine zwischen $n = 1$ und $n = 50$ variierende Knotenanzahl betrachtet. Die Anzahl der Map-Tasks wurde dabei auf $m = 10 \cdot n$ gesetzt, um den Einfluss von Computational Skew-Effekten abzuschwächen. Erneut wurde ein Entscheidungsbaum verwendet, um alle Paare anhand der $k = 6$ berechneten Ähnlichkeitswerte zu klassifizieren.

Abbildung 7.8 zeigt die resultierenden Ausführungszeiten und Speedup-Werte. Bis zu $n = 10$ Knoten konnte ein nahezu linearer Speedup beobachtet werden. Für mehr als 10 Knoten wächst der Speedup

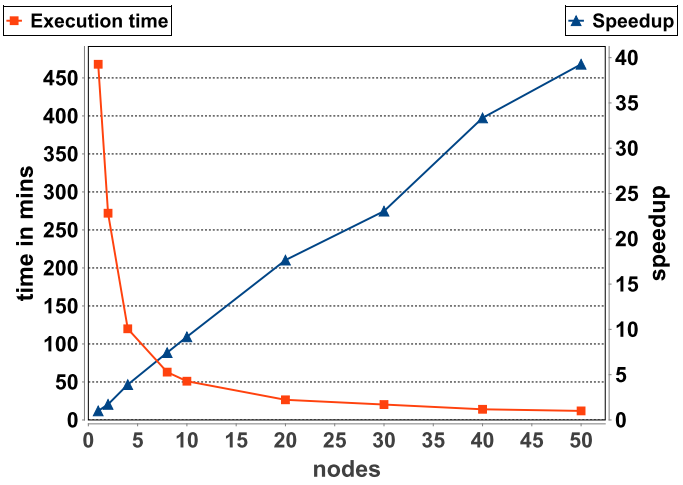


Abbildung 7.8: Ausführungszeiten und Speedup-Werte der MapSide-Strategie bei Berechnung aller $k = 6$ Ähnlichkeitswerte und einer Knotenanzahl von $n = 1$ bis $n = 50$ Knoten.

nicht mehr linear, da durch die damit einhergehende höhere Anzahl von Map-Tasks die pro Task zu verrichtende Arbeit sinkt. Gleichzeitig steigt der relative Anteil des MapReduce-Overheads zum Starten und Beenden von Tasks. Beispielsweise beträgt die durchschnittliche Zeit zur Bearbeitung eines Blockpaares unter Verwendung von $n = 10$ Knoten ungefähr 10 Minuten, für $n = 50$ Knoten werden durchschnittlich nur noch zwei Minuten benötigt. Der absolute Overhead eines einzelnen Tasks ist jedoch in beiden Fällen konstant. Trotzdem konnte ein Speedup von 39 für 50 Knoten erreicht werden. Mit einer wachsenden Problemgröße sind bessere Speedup-Werte zu erwarten.

7.4 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie Entity Resolution-Workflows mit maschinellen Lernverfahren zur automatischen Klassifikation mit MapReduce umgesetzt werden können. Die Evaluierung verdeutlichte, dass es notwendig ist, mehrere Ähnlichkeitsmetriken und Attribute zu verwenden, um eine hohe Match-Qualität erzielen zu können. Da die Ähnlichkeitsberechnung i. Allg. und die Anzahl der für alle Paare zu berechneten Ähnlichkeitswerte im Speziellen die Gesamtausführungszeit dominieren, ist eine Parallelisierung der Paarvergleiche erforderlich. Zu diesem Zweck wurden zwei Ansätze für die Parallelisierung der Ähnlichkeitsberechnung und Klassifikation aller Paare des Kartesischen Produktes zweier Datenquellen vorgestellt. Beide Strategien sind in der Lage, die zu verrichtende Arbeit gleichmäßig auf die zu Verfügung stehenden Cluster-Ressourcen zu verteilen und mit der Anzahl der Knoten zu skalieren.

8

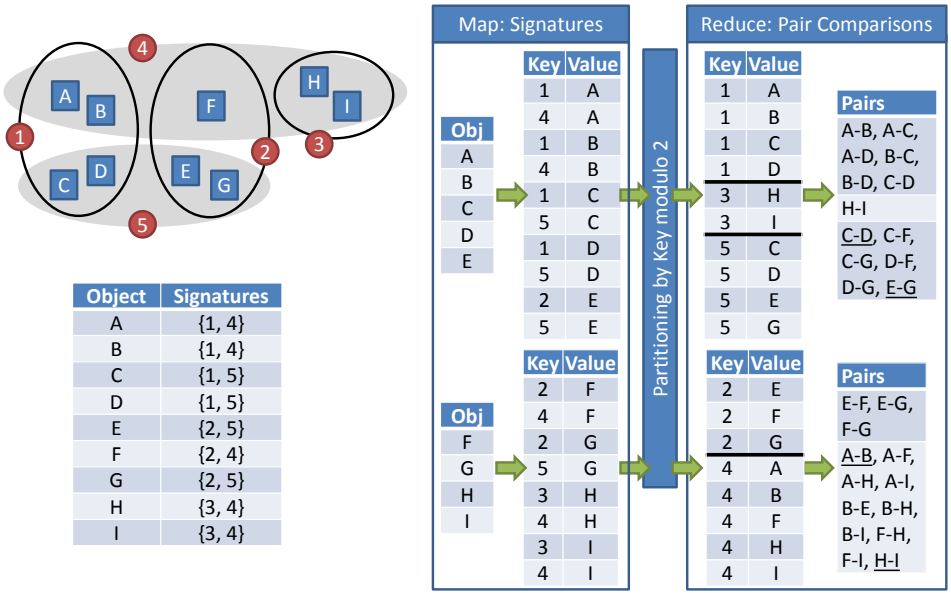
Vermeidung redundanter Ähnlichkeitsberechnungen

Um ein qualitativ hochwertiges Match-Ergebnis auch für "schmutzige", aus verschiedenen Quellen semi-automatisch extrahierte Webdaten gewährleisten zu können, ist es erforderlich, Datensätze mehreren überlappenden Clustern (Blöcken) zuzuweisen. Dies führt dazu, dass die Ähnlichkeit zweier Datensätze unnötigerweise mehrfach berechnet wird, sofern diese mehr als ein Cluster teilen. Dieses Kapitel untersucht die Fragestellung, wie der bisher betrachtete MapReduce-Workflow (vgl. Abbildung 3.4) angepasst werden muss, um diese unnötigen Vergleiche zu vermeiden. Dazu wird nach einer Einführung in die betrachtete Problematik (Abschnitt 8.1) eine einfach zu integrierende Strategie vorgestellt, die diese redundanten Paarvergleiche eliminiert (Abschnitt 8.2). Anschließend wird im Kapitel 8.3 der Einfluss der Vermeidung redundanter Ähnlichkeitsberechnungen auf die in den Kapiteln 4 und 5 vorgestellten Algorithmen zur Lastbalancierung diskutiert. Im Zuge dessen werden zwei weitere Strategien zur Vermeidung redundanter Paarvergleiche vorgestellt, welche die Nachteile des ersten Verfahrens beheben. Die Evaluation im Abschnitt 8.4 demonstriert die Wirksamkeit der Vermeidungsstrategien und vergleicht die vorgestellten Ansätze miteinander.

8.1 Einführung

Die Problematik der redundanten Paarvergleiche ist nicht spezifisch für Entity Resolution, sondern tritt in ähnlicher Form bei weiteren daten- und/oder rechenintensiven Problemstellungen, wie z. B. beim Clustering semantisch ähnlicher Dokumente [165] oder bei der effizienten Berechnung von Similarity Joins in Datenbanken [250], auf. Charakteristisch für diese Anwendungen ist, dass eine komplexe paarweise Ähnlichkeitsberechnung (PSC[1]) erfolgt, die aufgrund der inhärenten quadratischen Komplexität für große Datenmengen sowohl eine Reduzierung des Suchraums als auch eine Parallelisierung erfordert. Für solche Problemstellungen bietet sich eine MapReduce-Parallelisierung an, was sich durch eine Vielzahl von auf MapReduce basierenden PSC-Implementierungen widerspiegelt [74, 78, 173, 213, 234]. Ein typischer Ansatz zur Einschränkung der Kandidatenmenge ist es, die Ähnlichkeitsberechnung von Datensätzen mit mutmaßlich geringer Ähnlichkeit zu vermeiden. Dies wird analog zur Idee der Blocking-Verfahren dadurch erreicht, dass Datensätze in Cluster gruppiert werden und die Ähnlichkeitsberechnung auf Datensätze desselben Clusters beschränkt wird. Zu diesem Zweck verfolgen die erwähnten Implementierungen die Strategie, für jedes Objekt (Dokument, Zeichenkette, Datensatz) eine *Signatur* (Term, Token, Blockschlüssel) zu generieren. Eine Signatur identifiziert dabei ein Cluster, das alle Objekte beinhaltet, die diese Signatur aufweisen. Zur Umsetzung dieser Idee, wird in der Map-Phase ein (`signature`, `object`)-Paar für jedes Objekt generiert. Das MapReduce-Framework gruppiert alle Objekte anhand deren Signatur und berechnet die Ähnlichkeit aller Objekte eines Clusters in der Reduce-Phase.

[1] aus dem Englischen – Pair-wise Similarity Computation



Object	Signatures
A	{1, 4}
B	{1, 4}
C	{1, 5}
D	{1, 5}
E	{2, 5}
F	{2, 4}
G	{2, 5}
H	{3, 4}
I	{3, 4}

Abbildung 8.1: PSC unter Verwendung eines 2-pass Blocking von neun Objekten (*A*-*I*) in drei (schwarz umrandete) bzw. zwei (grau schattierte) Cluster. Die verwendete Signaturfunktion ist in der Tabelle aufgeschlüsselt. Der Datenfluss der MapReduce-Berechnung auf der rechten Seite zeigt, dass vier (unterstrichene) Paare von insgesamt 26 Paaren mehrfach ausgewertet werden.

Die Erzeugung "guter" Signaturen ist eine schwierige Aufgabe, da stets zwischen Effizienz und resultierender Qualität abgewogen werden muss. Einerseits sollten die Cluster möglichst klein sein, um die Menge der Paarvergleiche zu reduzieren und die Effizienz zu erhöhen. Andererseits führt die Verwendung kleiner Clustergrößen dazu, dass fälschlicherweise auch Paare von ähnlichen Objekten eliminiert werden. Dies gilt insbesondere für semi-automatisch extrahierte Webdaten, die i. Allg. sehr "schmutzig" sind, also fehlerhafte, widersprüchliche oder fehlende Informationen enthalten. Um dem zu begegnen, setzen PSC-Algorithmen (z. B. Entity Resolution [138], Document Clustering [165] , Sequenzalignment [213]) mehrere Signaturen pro Objekt ein. Sei O eine Menge von Objekten und S eine Menge möglicher Signaturen. Eine Signaturfunktion $\sigma : O \rightarrow \mathscr{P}(S)$ weist jedem Objekt $o \in O$ eine (nicht-leere) Menge von Signaturen $s \subseteq S$ zu. Anschließend wird die Ähnlichkeit aller Objektpaare $(o_1, o_2) \in O \times O$ mit $o_1 \neq o_2 \wedge \sigma(o_1) \cap \sigma(o_2) \neq \emptyset$ berechnet. Speziell für Entity Resolution ist es empfehlenswert, die Signaturen von verschiedenen Attributen der Datensätze abzuleiten. Dies soll garantieren, dass ähnliche Objekte auch im Falle von Datenqualitätsproblemen zusammen gruppiert werden und eine hohe Pairs Completeness erreicht wird.

Abbildung 8.1 zeigt ein Beispiel mit neun Objekten (*A*-*I*) und fünf verschiedenen Signaturen (1 bis 5). Die in der Tabelle gezeigte Signaturfunktion könnte das Ergebnis eines 2-pass Blockings sein. Der erste Durchgang erzeugt drei Cluster mit den Signaturen 1, 2 und 3, im zweiten Durchgang werden zwei weitere Cluster mit den Signaturen 4 und 5 generiert. Im Beispiel erzeugt die Signaturfunktion pro Objekt dieselbe Anzahl an Signaturen, dies ist jedoch nicht zwingend notwendig. Die `map`-Funktion wird für jedes Objekt aufgerufen und gibt für *jede* Signatur des Objektes ein (signature, object)-Paar aus. Beispielsweise werden für das Objekt A mit $\sigma(A) = \{1, 4\}$ die Paare $(1, A)$ und $(4, A)$ ausgegeben. Zur Umverteilung der `map`-Ausgabe-Paare zu den Reduce-Tasks wird hier eine einfache Modulo-Funktion verwendet, die die Schlüsselgruppen 1, 3 und 5 dem ersten und die übrigen Schlüsselgruppen dem zweiten Reduce-Task zuweist. Die `reduce`-Funktion wird für jede Signatur aufgerufen, in `reduce` erfolgt die Ähnlichkeitsberechnung der assoziierten Objekte. So werden z. B. alle sechs Paare *A*-*B*, *A*-*C*, ..., *C*-*D* von Objekten mit der Signatur 1 vom ersten Reduce-Task bearbeitet.

Sobald zwei Objekte mehr als eine gemeinsames Cluster aufweisen, werden sie mehrfach, bezüglich verschiedener Signaturen, verglichen. Dies erhöht unnötigerweise die Laufzeiteffizienz und führt i. Allg. dazu, dass im finalen Match-Ergebnis Duplikate enthalten sind, die in einem aufwändigen Nachverarbeitungsschritt entfernt werden müssen (siehe z. B. [163]). Im Beispiel von Abbildung 8.1 werden die vier unterstrichenen Paare *A-B*, *C-D*, *E-G*, und *H-I* zweimal verglichen, da sie jeweils zwei gemeinsame Signaturen aufweisen. Das Vermeiden dieser redundanten Paarvergleiche in einer MapReduce-Umgebung ist schwierig, da Cluster i. Allg. von verschiedenen Reduce-Tasks bearbeitet werden, die jeweils nur die ihnen zugewiesenen Eingabedaten "sehen" und deswegen keine Kenntnis darüber haben, ob möglicherweise ein anderer Reduce-Task das gleiche Objektpaar bearbeitet. In den folgenden beiden Abschnitten werden Modifikationen des MapReduce-Ausführungsschemas vorgestellt, die es den Reduce-Tasks erlauben, die Existenz redundanter (von anderen Reduce-Tasks/Knoten bearbeiteter) Paare zu erschließen und somit die unnötigen Kosten wiederholter Paarvergleiche einzusparen.

8.2 Redundanzfreie Ähnlichkeitsberechnung

In der Literatur fand die Problematik der redundanten Paarvergleiche bisher kaum Beachtung. Die erste bekannte Arbeit untersuchte die Problematik in einer nicht-verteilten Umgebung [193]. Die Autoren schlugen vor, ein Objektpaar für eine bestimmte Signatur nur dann zu vergleichen, wenn diese mit der kleinsten gemeinsamen Signatur beider Objekte übereinstimmt (*Least Common Signature*-Bedingung). Dies stellt sicher, dass jedes Objektpaar mit überlappenden Signaturmengen genau einmal verglichen wird. Zu diesem Zweck wurden die Ausgangscluster in eine Menge kleinerer semantisch äquivalenter Cluster überführt, wobei im Prinzip jedes nicht-redundante Paar ein eigenes Cluster bildet. Diese Herangehensweise führt jedoch zu einer sehr großen Anzahl an Blöcken, was mit einer quadratischen Speicherkomplexität einhergeht. Um dieses Problem zu beheben, schlugen die Autoren einen komplexen Algorithmus zur Reorganisation und zum Verschmelzen von Clustern vor. In einer MapReduce-Umgebung müsste diese Vorverarbeitung mit mehreren MapReduce-Jobs umgesetzt werden, da die Daten über mehrere Knoten verteilt und in einem verteilten Dateisystem abgelegt sind. Um dies zu vermeiden, wurde in Dedoop die Strategie verfolgt, jedem Datensatz zusätzliche Metadaten hinzuzufügen und in der Reduce-Phase *zur Laufzeit* mithilfe dieser Metadaten zu überprüfen, ob die Least Common Signature-Bedingung erfüllt ist.

Die dabei verfolgte Strategie kann, wie folgt, verallgemeinert werden. Redundanzfreie PSC verlangt, dass jedes Objektpaar genau einmal verglichen wird, auch wenn es mehr als eine gemeinsame Signatur hat. Die wesentliche Problematik bei MapReduce-basierter PSC ist, dass ein Reduce-Task keine Kenntnis darüber hat, ob ein anderer Reduce-Task das gleiche Objektpaar bearbeitet. Zur Lösung dieses Problems muss ein beliebiger Reduce-Task für ein beliebiges Objektpaar (o_1, o_2) deterministisch eine Signatur $l \in \sigma(o_1) \cap \sigma(o_2)$ auswählen, für die beide Objekte zu vergleichen sind. Entsprechend der Least Common Signature-Bedingung würde dies die kleinste Signatur $l = min(\sigma(o_1) \cap \sigma(o_2))$ der Schnittmenge beider Signaturmengen sein. In der Folge würde das Paar (o_1, o_2) nur von dem Reduce-Task bearbeitet werden, der die Schlüsselgruppe l bearbeitet, alle anderen Reduce-Tasks würden den Vergleich überspringen[2]. Dieser einfache aber effektive Ansatz stellt sicher, dass alle relevanten Objektpaare ausgewertet und keine redundanten Paarvergleiche durchgeführt werden.

Bei der Bearbeitung eines Clusters mit der Signatur k in der `reduce`-Funktion ist es also ausreichend, für jedes Objektpaar (o_1, o_2) zu überprüfen, ob es eine *kleinere* gemeinsame Signatur $k' < k$ gibt. Ist dies der Fall, so kann das Paar (o_1, o_2) übersprungen werden, ansonsten ist k die kleinste gemeinsame Signatur beider Objekte und der aktuelle Reduce-Task ist "zuständig" die Ähnlichkeit von o_1 und o_2 zu bestimmen. Um dies zu realisieren, bietet sich folgende Anpassung des Basisverfahrens an:

[2] Für die Ähnlichkeitsberechnung mit anderen Objekten müssen o_1 und o_2 trotzdem für jede Signatur $l' \in \sigma(o_1) \cap \sigma(o_2)$ mit $l' \neq l$ von der `map`-Funktion ausgegeben werden.

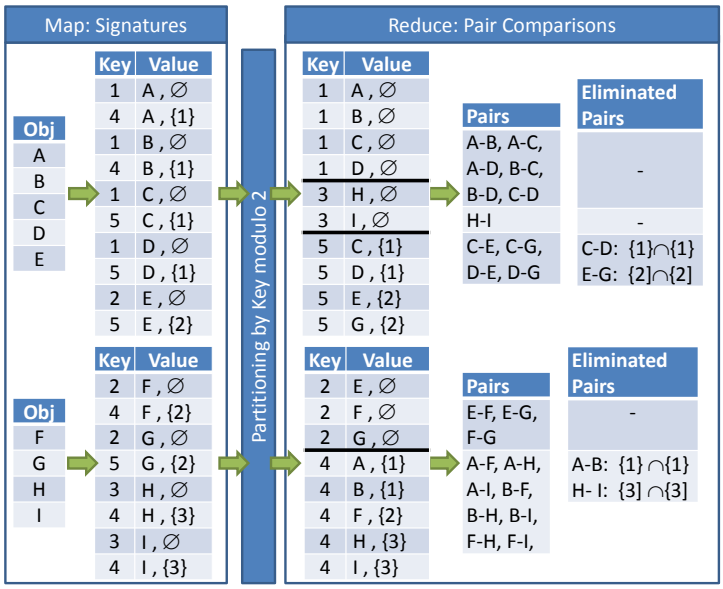


Abbildung 8.2: Redundanzfreie PSC für das Beispiel aus Abbildung 8.1. In der Reduce-Phase sind sowohl die durchgeführten Ähnlichkeitsberechnungen als auch die eliminierten Paare gezeigt.

map: Für jedes Objekt o bestimmt die `map`-Funktion dessen Signaturmenge $\sigma(o) = \{s_1, s_2, ..., s_n\}$. Die `map`-Funktion gibt nicht nur n Schlüssel-Wert-Paare (s_i, o) aus, sondern annotiert o für jedes $1 \leq i \leq n$ mit dessen Signaturen, die kleiner als s_i sind. Dementsprechend gibt `map` für jedes $1 \leq i \leq n$ ein Schlüssel-Wert-Paar $(s_i, [o, \sigma_{s_i}(o)])$ mit $\sigma_{s_i}(o) = \{s \in \sigma(o) \mid s < s_i\}$ aus.

reduce: Beim Aufruf der `reduce`-Funktion für die Signatur k wird für jedes Wert-Paar $([o_1, \sigma_k(o_1)], [o_2, \sigma_k(o_2)])$ überprüft, ob die Signaturmengen disjunkt sind, also ob $\sigma_k(o_1) \cap \sigma_k(o_2) = \emptyset$ gilt. Sind $\sigma_k(o_1)$ und $\sigma_k(o_2)$ disjunkt, so ist k die kleinste Signatur und o_1, o_2 werden verglichen. Ansonsten erfolgt keine Ähnlichkeitsberechnung des Objektpaars, da es eine kleinere gemeinsame Signatur $k' < k$ beider Objekte gibt.

Abbildung 8.2 zeigt den MapReduce-Datenfluss dieses Verfahrens für das Beispiel aus Abbildung 8.1. Die Arbeitsweise soll am Beispiel des Objektes A mit den Signaturen $\sigma(A) = \{1, 4\}$ erläutert werden. Zunächst gibt die `map`-Funktion ein Schlüssel-Wert-Paar $(1, [A, \emptyset])$ aus, da A keine kleinere Signatur als 1 hat. Zusätzlich wird ein Paar $(4, [A, \{1\}])$ ausgegeben, da $\sigma_4(A) = \{1\}$. Das erste Schlüssel-Wert-Paar $(1, [A, \emptyset])$ wird dem ersten Reduce-Task zugewiesen, der A mit den Objekten B, C, D, die ebenfalls die Signatur 1 haben, vergleicht. Eine kleinere gemeinsame Signatur kann nicht existieren, da 1 die kleinste Signatur des Objekts A ist. Das zweite Schlüssel-Wert-Paar $(4, [A, \{1\}])$ wird zum zweiten Reduce-Task umverteilt. Dort muss überprüft werden, ob A mit den Objekten B, F, H, I verglichen werden muss, die auch die Signatur 4 aufweisen. Letztlich wird A mit F, H und I verglichen, da 4 für diese drei Paare die jeweils kleinste gemeinsame Signatur ist ($\sigma_4(A) = \{1\}$, aber $\sigma_4(F) = \{2\}$, $\sigma_4(H) = \{3\}$, $\sigma_4(I) = \{3\}$). Die Ähnlichkeitsberechnung $A - B$ wird übersprungen, da beide Datensätze eine kleinere gemeinsame Signatur ($\sigma_4(A) \cap \sigma_4(B) = \{1\} \neq \emptyset$) haben. Insgesamt werden vier von 26 Paaren eingespart, was einer Einsparung von $\approx 15\%$ entspricht.

Algorithmus 8.1 zeigt den Pseudo-Code der MapReduce-Implementierung des Least Common Signature-Verfahrens (LCS). Die Implementierung verwendet sortierte Signaturlisten anstatt (unsortierter) Signaturmengen (Zeile 3–7). Diese Vorgehensweise hat zwei Vorteile. Zum Einen entspricht $\sigma_{s_i}(o)$ einfach dem Präfix der Signaturliste $\sigma(o)$ bis zur aktuellen Signatur s_i. Des Weiteren ermöglicht dieser Ansatz die Verwendung eines schritthaltenden linearen Durchlaufs zweier sortierter Listen zur effiziente Über-

Algorithmus 8.1: MapReduce-Umsetzung des Least Common Signature-Verfahrens zur Vermeidung redundanter Ähnlichkeitsberechnungen

```
1  map(k_in=unused, v_in=o)
2      S ← σ(o).distinct();
3      S.sort();
4      SS ← [] // smaller signature list
5      foreach s_i ∈ S do
6          output(k_tmp = s_i, v_tmp = (o, SS));
7          SS.append(s_i);
8  reduce(k_tmp=s, list(v_tmp)=list(object, SS))
9      buf ← {};
10     foreach (o_1, SS_1) ∈ list(object, SS) do
11         foreach (o_2, SS_2) ∈ buf do
12             if ¬doOverlap (SS_1, SS_2) then
13                 compare(o_1, o_2);
14         buf ← buf ∪ {(o_1, SS_1)};
15 doOverlap(SS_1, SS_2)
16     i ← 0; j ← 0;
17     l_1 ← SS_1.length();
18     l_2 ← SS_2.length();
19     while (i < l_1) ∧ (j < l_2) do
20         s_1 ← SS_1.get(i);
21         s_2 ← SS_2.get(j);
22         cmp ← s_1.compareTo(s_2);
23         if cmp = 0 then
24             return true;
25         else if cmp < 0 then
26             i++;
27         else
28             j++;
29     return false;
```

prüfung der Least Common Signature-Bedingung (Zeile 12). Eine mögliche Alternative zum Einsatz sortierter Signaturlisten ist die Neuberechnung der Signaturen aus den Attributwerten der Datensätze in der Reduce-Phase. Dies verspricht zwar eine Verringerung des zwischen der Map- und Reduce-Phase umzuverteilenden Datenvolumens, hat jedoch mehrere entscheidende Nachteile. Zunächst erfordert ein solches Vorgehen, dass die ursprünglichen Attributwerte "erhalten" werden. Der Dedoop-Prototyp führt jedoch eine Projektion in der Map-Phase durch, sodass jeder Datensatz ausschließlich durch seinen Identifikator, die zur Ähnlichkeitsberechnung benötigten (normalisierten) Attributwerte und die nutzerdefinierten Attributwerte, die im finalen Match-Ergebnis enthalten sein sollen, verbleiben. Eine Neuberechnung der Signaturen in der Reduce-Phase würde somit das Datenvolumen möglicherweise sogar erhöhen. Ein zweiter Nachteil ist die Tatsache, dass die Neuberechnung in der Reduce-Phase für jedes von der `map`-Funktion ausgegebene Schlüssel-Wert-Paar erfolgen muss. Da ein Objekt mehrfach von der `map`-Funktion ausgegeben wird (möglicherweise auch zu Lastbalancierungszwecken), ist die Anzahl der Signaturberechnungen dieser Variante insgesamt deutlich höher. Der größte Nachteil ist jedoch, dass die Überprüfung, ob zwei Signaturlisten überlappen, wesentlich teurer ist, da keine Sortierung ausgenutzt werden kann.

Das vorgestellte Verfahren Least Common Signature-Verfahren zur Vermeidung redundanter Paarvergleiche bei der Verwendung mehrerer Signaturen pro Datensatz lässt sich relativ leicht in den allgemeinen MapReduce-Workflow aus Abbildung 3.4 integrieren. Dazu muss der Blocking-based Matching Job nur minimal angepasst werden. Bei der Ausgabe eines Schlüssel-Wert-Paares in der Map-Phase wird jeder Datensatz um die Menge seiner Signaturen, die kleiner als die aktuelle Signatur sind, angereichert. Der Aufbau der Schlüssel sowie deren Partitionierung, Sortierung und Gruppierung bleiben unverändert. In der Reduce-Phase werden die zusätzlichen Metadaten herangezogen, um redundante Paarvergleiche zu eliminieren. Diese Eigenschaften ermöglichen eine einfache Integration in bestehende PSC-Anwendungen. So konnte die Optimierungstechnik mit nur wenigen Änderungen am Quellcode in die Arbeit [234] zur Berechnung von Similarity Joins eingebracht werden. Die Autoren veröffentlichten daraufhin auf Ihrer Projektseite eine vom Autor der vorliegenden Dissertation bereitgestellte Patch-Datei[3].

[3] `http://asterix.ics.uci.edu/fuzzyjoin`

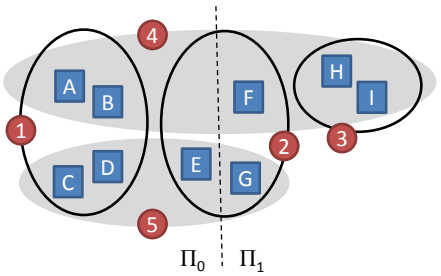
8.3 Redundanzfreiheit und Lastbalancierung

Die Parallelisierung von Entity Resolution-Workflows mit MapReduce ist aufgrund der inhärenten quadratischen Komplexität der paarweisen Ähnlichkeitsberechnung hochgradig anfällig gegenüber einer ungleichmäßigen Blockschlüsselverteilung. Vor diesem Hintergrund wurden in den Kapiteln 4 und 5 Lastbalancierungstechniken diskutiert, deren Einsatz essenziell für skalierbare MapReduce-Parallelisierung Blocking-basierter Entity Resolution-Workflows ist.

8.3.1 Überblick

Eine Kombination der Lastbalancierungstechniken für das Standard Blocking mit dem LCS-Ansatz ist ohne weitere Anpassung möglich. Die Ersetzung jeder Signatur durch den korrespondierenden numerischen Blockindex führt zudem zu einer beschleunigten Überprüfung der Least Common Signature-Bedingung, da die Gleichheit von Integer-Werten wesentlich effizienter feststellbar ist als die Gleichheit von Zeichenketten. Für das Sorted Neighborhood-Verfahren ist eine kleine Modifikation des Algorithmus 8.1 notwendig, da Kandidatenpaare nicht auf Basis eines gemeinsamen Blockschlüssels sondern in Abhängigkeit der Entfernung zueinander (in der Sortierreihenfolge der Blockschlüssel) gebildet werden. Für die Anwendung der LCS-Strategie auf das Multi-pass Sorted Neighborhood-Verfahren muss jeder Datensatz in der Map-Phase nicht mit seinen (kleineren) Blockschlüsseln, sondern mit seinen *Positionen* in der sortierten Datensatzliste aller vorhergehenden Durchgänge angereichert werden. In der `reduce`-Funktion müssen die zwei Listen (derselben Länge) eines Datensatzpaares dahingehend geprüft werden, ob ein Index existiert, an dem die die absolute Differenz der Positionsnummern kleiner als die Fenstergröße w ist. Ist dies der Fall, so ist die Least Common Signature-Bedingung *nicht* erfüllt, da die Ähnlichkeit beider Datensätze bereits in einem vorigen Durchgang berechnet wurde.

Leider werden durch den LCS-Ansatz die den Lastbalancierungsalgorithmen zugrunde liegenden Annahmen invalidiert. Nunmehr ist es nicht mehr möglich, die Gesamtzahl tatsächlich durchzuführender Paarvergleiche aus der Anzahl der Datensätze pro Block abzuleiten. Bei der Lastbalancierung wird i. Allg. eine größere Anzahl an Paarvergleichen angenommen als tatsächlich durchzuführen ist. Am Beispiel des BlockSplit-Algorithmus führt dies dazu, dass Blöcke unnötigerweise gesplittet werden. Ebenso wenig kann die genaue Workload eines Match-Tasks anhand der Anzahl der Datensätze eines Blocks in den entsprechenden Eingabepartitionen berechnet werden, was eine fehlerhafte Zuweisung von Match-Tasks zu Reduce-Tasks nach dem Greedy-Prinzip bewirkt. Diese Falschannahmen sind dadurch bedingt, dass ein beliebiges Datensatzpaar n gemeinsame Signaturen ($= n$ Paarvergleiche) haben kann, aber nur bezüglich einer Signatur miteinander verglichen wird. Allgemein ist es wahrscheinlicher, dass ein Datensatzpaar für eine "große" Signatur (entsprechend der lexikographischen Sortierreihenfolge) als redundant eingestuft wird als es für eine "kleine" Signatur der Fall ist. Die Vermeidung redundanter Paarvergleiche nach dem Least Common Signature-Prinzip führt also zu einem gewissen Grad selber zu einer ungleichmäßigen Aufteilung von Datensatzpaaren auf die zur Verfügung stehenden Reduce-Tasks und damit zu einer ungleichmäßigen Auslastung der Rechenressourcen. Um dieses Problem zu beheben, bestehen grundsätzlich zwei verschiedene Möglichkeiten, die am Beispiel des BlockSplit-Verfahrens erläutert werden. Eine Möglichkeit ist es, die Falschannahmen zu korrigieren indem im Analyse-Job nicht die Datenverteilung, sondern, unter Berücksichtigung der eliminierten Paare, die tatsächliche Workload aller Match-Tasks ermittelt wird (Exact-Strategie). Ein leichtgewichtigerer Ansatz ist es, die redundanten Paarvergleiche gleichmäßig über alle Signaturen zu verteilen. Dieser Ansatz wird durch die Heuristic-Strategie verfolgt, deren Grundidee es ist, ein Datensatzpaar nicht für deren kleinste gemeinsame, sondern für *irgendeine* deterministisch bestimmbare gemeinsame Signatur durchzuführen. Dabei wird die Lastbalancierung unverändert auf Basis inkorrekter Annahmen vorgenommen, aber versucht die Fehler gleichmäßig über alle Blöcke "zu verteilen".



Signatures	Match Task Workload 0x0	0x1	1x1	∗
1	6	0	0	6
2	0	2	1	3
3	0	0	1	1
4	**1-1=0**	6	**3-1=2**	8
5	**3-1=2**	**3-1=2**	0	4

Abbildung 8.3: Erweiterte BDM für die Lastbalancierung mit dem BlockSplit-Verfahren in Kombination mit der Eliminierung redundater Paarvergleiche nach dem LCS-Prinzip. Statt der Anzahl von Datensätzen pro Block und Eingabepartition enthält die BDM die tatsächliche Anzahl an durchzuführenden Paarvergleichen pro Block und Match-Task.

8.3.2 Exakte Lastbalancierung durch Erweiterung des Analyse-Jobs

Die Exact-Strategie kombiniert den LCS-Ansatz zur Eliminierung redundanter Paarvergleiche und das BlockSplit-Lastbalancierungsverfahren. Dabei liegt der Fokus auf einer *korrekten* Bestimmung der Workload aller Blöcke und Match-Tasks. Dazu ist es notwendig, die im Abschnitt 4.1.2 vorgestellte Berechnungsvorschrift sowie den Aufbau der BDM anzupassen. Im Folgenden wird dies nur für den Ein-Quellen-Fall diskutiert, das Grundprinzip ist jedoch auch für die Duplikaterkennung in zwei Datenquellen anwendbar.

Abbildung 8.3 zeigt den Aufbau der BDM für das initiale Beispiel aus Abbildung 8.1. Sei b die Anzahl der Blöcke (Signaturen) und m die Anzahl der Eingabepartitionen. Die BDM entspricht einer $b \times m \cdot (m-1)$-Matrix, die die Anzahl der *tatsächlich durchzuführenden Paarvergleiche* pro Block und Match-Task enthält. Bei der Berechnung der BDM wird also für jeden Block zunächst davon ausgegangen, dass es sich um einen großen Block handelt, dessen Paarvergleiche auf mehrere Match-Tasks aufgeteilt werden müssen. Die fett markierten Zellen kennzeichnen die vier eliminierten Paarvergleiche *A-B*, *C-D*, *E-G* und *H-I*. Die in der BDM kodierten Informationen erlauben es, während der Lastbalancierung in der Map-Phase des Blocking-based Matching Jobs die Gesamtzahl der tatsächlich durchzuführenden Ähnlichkeitsberechnungen zu ermitteln ($P = 22$). Basierend auf diesem Wert kann in der Folge für jeden Block Φ_k entschieden werden, ob es sich um einen "großen" Block handelt oder nicht. Sollte ersteres der Fall sein, so kann die korrekte Workload eines jeden zu bildenden Match-Tasks direkt aus der BDM entnommen werden. Sollte sich herausstellen, dass Φ_k nicht gesplittet werden muss, so ergibt sich die Workload des subsummierenden Match-Tasks $k.*$ aus der Summe der Paarvergleiche seiner $m \cdot (m-1)$ Subtasks. Dies entspricht der letzten Spalte der BDM in Abbildung 8.3, die hier nur zu Illustrationszwecken gezeigt ist und *nicht* materialisiert wird. Insgesamt wird durch die Modifikation der BDM eine korrekte Zuweisung von Match-Tasks zu den Reduce-Tasks nach dem Greedy-Prinzip ermöglicht.

Ein wesentlicher Nachteil dieser Vorgehensweise ist die im Vergleich zur bisherigen BDM-Berechnung (siehe Algorithmus 4.1) gestiegene Komplexität des Analyse-Jobs. Algorithmus 8.2 zeigt den Pseudo-Code der angepassten BDM-Berechnung. Die Map-Phase bleibt dabei weitesgehend unverändert, die einzige Anpassung besteht darin, dass der Wert eines jeden ausgegebenen Schlüssel-Wert-Paars nicht aus der Zahl 1, sondern aus einer sortierten Liste aller Blockschlüssel des Datensatzes, die kleiner als der aktuell betrachtete Blockschlüssel sind, besteht. Die Reduce-Phase des Analyse-Jobs hat jedoch eine quadratische Komplexität, da für jedes Kandidatenpaar die Least Common Signature-Bedingung überprüft werden muss[4], um für jeden Match-Task die korrekte Anzahl an Paarvergleichen ermitteln zu können. Dabei ist zu beachten, dass diese Überprüfung ausschließlich zur Gewährleistung der Korrekt-

[4] In der Reduce-Phase des Analyse-Jobs findet *keine* Ähnlichkeitsberechnung statt.

Algorithmus 8.2: BDM-Berechnung mit Berücksichtigung eliminierter Paarvergleiche

```
1  map_configure(m, r, partitionIndex)
2      // Store partitionIndex

3  map(k_in=unused, v_in=o)
4      S ← σ(o).distinct();
5      S.sort();
6      SS ← [] // smaller signature list
7      additionalOutput (k=blockingKey, v=(o, S))
8      foreach s_i ∈ S do
9          // regular map output
10         output(k_tmp = s_i.partitionIndex, v_tmp = SS);
11         SS.append(s_i);

12 reduce_configure(m)
13     lastSignature ← null;
14     inputPartitions ← [];
15     reset();

16 // part: repartition map output by s_i
17 // cmp: sort by s_i.partitionIndex
18 // group: group by s_i.partitionIndex
19 reduce(k_tmp=s_i.partitionIndex, list(v_tmp)=list(SS))
20     if lastSignature≠null ∧ lastSignature≠ s_i then
21         writeBDMLine();
22     foreach SS ∈ list(v_tmp) do
23         inputPartitions[partitionIndex ].append(SS);
24     lastSignature ← s_i;

25 reduce_close()
26     writeBDMLine();

27 reset()
28     for i ← 0 to m-1 do
29         // smaller signature lists of objects having
30         // a common signature separated by input
31         // partition
32         inputPartitions[i] ← [];

33 writeBDMLine()
34     str ← lastSignature;
35     for i ← 0 to m-1 do
36         for j ← i to m-1 do
37             w ← 0;
38             if i=j then
39                 for k ← 0 to inputPartitions[i].size()-2 do
40                     for l ← k + 1 to inputPartitions[i].size()-1 do
41                         if ¬doOverlap(
42                           inputPartitions[i].get(k),
43                           inputPartitions[i].get(l)) then
44                             w ← w + 1;
45             else
46                 foreach SS_1 ∈ inputPartitions[i] do
47                     foreach SS_2 ∈ inputPartitions[j] do
48                         if ¬doOverlap(SS_1, SS_2) then
49                             w ← w + 1;
50             str ← str + "," + w;
51     output (k_out=unused, v_out=str);
52     reset();
```

heit der Lastbalancierung dient. In der Reduce-Phase des darauffolgenden Blocking-based Matching Jobs muss vor jeder Ähnlichkeitsberechnung *erneut* überprüft werden, ob der Vergleich übersprungen werden kann.

Die Exact-Strategie passt also das BlockSplit-Verfahren dahingehend an, dass eine gleichmäßige Aufteilung von Ähnlichkeitsberechnungen auf die Reduce-Tasks auch bei der Eliminierung redundanter Paarvergleiche nach dem LCS-Ansatz garantiert wird. Dem steht jedoch eine längere Rechenzeit des Analyse-Jobs sowie ein höherer Hauptspeicherbedarf während der Lastbalancierung in der Map-Phase des Blocking-based Matching Jobs gegenüber. Die Evaluation im Abschnitt 8.4 wird an einem konkreten Beispiel untersuchen, ob die gestiegene Komplexität durch eine gleichmäßige Auslastung der zur Verfügung stehenden Ressourcen amortisiert werden kann.

8.3.3 Heuristische Verteilung der redundanten Paarvergleiche

Die Heuristic-Strategie verfolgt im Gegensatz zur Exact-Strategie nicht das Ziel, die bei der Lastbalancierung getroffenen Falschannahmen zu korrigieren, sondern versucht, die durch das Least Common Signature-Prinzip verursachte ungleichmäßige Auslastung der Reduce-Tasks zu vermeiden, indem die Ähnlichkeit eines Datensatzpaars nicht für deren kleinste gemeinsame, sondern für *irgendeine* gemeinsame Signatur berechnet wird. Dadurch soll erreicht werden, dass die eliminierten Paarvergleiche sowie deren Effekte auf die Lastbalancierung gleichmäßig über alle Blöcke verteilt werden.

Um zu entscheiden, ob ein Reduce-Task für den Vergleich eines Objektpaars (o_1, o_2) zuständig ist, muss deterministisch eine eine Signatur $l \in \sigma(o_1) \cap \sigma(o_2)$ bestimmt werden, für die beide Objekte zu vergleichen sind. Die Heuristic-Strategie verwendet zur Bestimmung dieser Signatur l eine kommutative zweistellige Funktion, die jedes Objektpaar eindeutig auf eine Signatur der Schnittmenge abbildet. Ei-

ne einfache Umsetzung der obigen Idee ist die Wahl einer Signatur $s_i \in \{s_1, \ldots, s_n\} = \sigma(o_1) \cap \sigma(o_2)$ mit $i = p \cdot (h(o_1.id) + h(o_2.id)) \mod |\sigma(o_1) \cap \sigma(o_2)|$, wobei h eine Hashfunktion und p eine Primzahl bezeichnet.

Die Heuristic-Strategie ist im Vergleich zur Exact-Strategie ein sehr leichtgewichtigerer Ansatz. Bis auf die Tatsache, dass in der Map-Phase für jede Signatur eines Datensatz ein separates Schlüssel-Wert-Paar ausgegeben wird, bleibt der MapReduce-Job zur Analyse der Datenverteilung gegenüber Algorithmus 4.1 unverändert. In der Map-Phase des Blocking-based Matching Jobs muss der Wert eines ausgegebenen Schlüssel-Wert-Paars im Vergleich zur LCS- und Exact-Strategie mit der Menge *aller* Signaturen des Objektes annotiert werden. In der Reduce-Phase ist vor der Ähnlichkeitsberechnung eines Paars (o_1, o_2) zu prüfen, ob die auf die oben beschriebene Weise bestimmte Signatur l mit der aktuellen, im `reduce`-Eingabe-Schlüssel kodierten, Signatur übereinstimmt. Ist dies nicht der Fall, so kann der Vergleich des Objektpaars übersprungen werden, da die Ähnlichkeitsberechnung für das Paar bezüglich einer anderen Signatur zu erfolgen hat. Aufgrund der längeren Signaturlisten muss beim Heuristic-Verfahren im Vergleich zur Exact-Strategie ein geringfügig größeres Datenvolumen zwischen der Map- und Reduce-Phase umverteilt werden. Des Weiteren ist die Überprüfung, ob ein Paarvergleich übersprungen werden kann, etwas aufwändiger. Dem steht jedoch eine wesentlich einfachere Berechnung des Analyse-Jobs gegenüber. Ein weiterer wesentlicher Vorteil der Heuristic- gegenüber der Exact-Strategie ist die Möglichkeit, sie auch in Kombination mit dem PairRange-Lastbalancierungsalgorithmus einzusetzen.

8.4 Evaluation

Die vorgestellten Verfahren wurden in drei Experimenten evaluiert. Das erste Experiment demonstriert die Wichtigkeit der Verwendung mehrerer Signaturen bzw. Blockschlüssel pro Datensatz für das Erzielen einer guten Match-Qualität. Zudem wird die Laufzeit des herkömmlichen MapReduce-Workflows mit der MapReduce-Implementierung des LCS-Ansatzes verglichen. Anschließend wird untersucht, wie sich die Laufzeiten mit und ohne Eliminierung der redundanten Ähnlichkeitsberechnungen für einen systematisch variierten Grad an Redundanz verhalten. Im dritten Experiment wird untersucht, inwieweit die Auswirkungen, die die LCS-Strategie auf die Lastbalancierung hat, bei Verwendung der Exact- bzw. Heuristic-Strategie vermieden werden können. Zur Evaluation wurde die bereits im Abschnitt 4.4 verwendete Datenquelle DS1 herangezogen, die ca. 114.000 Produktdatensätze umfasst. Für alle Konfigurationen wurde das im Abschnitt 4.2 vorgestellte BlockSplit-Verfahren zur Lastbalancierung eingesetzt. Alle Experimente wurden in einer fixen EC2-Umgebung bestehend aus 20 virtuellen Maschinen des Typs c1.medium sowie einer dedizierten Master-Instanz des Typs m1.small (Namenode und Tasktracker-Prozess) durchgeführt. Erneut wurde Hadoop 0.20.2 auf jeder virtuellen Maschine eingerichtet, die Map- und Reduce-Task-Kapazität der 20 Slave-Knoten wurde jeweils auf 2 gesetzt. In allen Experimenten wurde der Blocking-based Matching-Job mit $r = 5 \cdot 20 \cdot 2$ Reduce-Tasks ausgeführt.

8.4.1 Anzahl der Signaturen pro Datensatz

Um die Qualität verschiedener Blocking-Strategien vergleichen zu können, ist ein Referenzergebnis notwendig. In Ermangelung eines perfekten Match-Ergebnisses wurde ein sogenannter *Silver Standard* generiert, indem die Trigrammähnlichkeit der Produkttitel *aller* Paare der Eingabedatenquelle DS1 berechnet wurde und alle Paare mit einer Ähnlichkeit $\geq 0{,}75$ als Duplikate betrachtet wurden. Anschließend wurden vier verschiedene Blocking-Strategien unter Verwendung von einem bis vier Blockschlüsseln pro Datensatz evaluiert. Abbildung 8.4 zeigt für jede der vier Blocking-Strategien, welcher Anteil der *Matches* des Silver Standards trotz des Blockings im jeweiligen Match-Ergebnis enthalten ist (unter Verwendung derselben Vorschrift zur Ähnlichkeitsberechnung und Klassifikation). Die alleinige Verwendung des Preis-Attributs zum Clustering der Datensätze resultiert lediglich in einer Pairs Completeness

Strategy	Pairs	Redundancy	Pairs completeness
1	$\approx 0.81 \cdot 10^9$	0%	0.75
1-2	$\approx 1.11 \cdot 10^9$	7.61%	0.90
1-3	$\approx 1.97 \cdot 10^9$	16.82%	0.98
1-4	$\approx 2.31 \cdot 10^9$	28.09%	≈ 1

Pass	Blocking keys
1	$\lceil \log_2(\text{price}) \rceil$
2	title.substr(0, 3)
3	category.substr(0, 10)
4	manufacturer

Abbildung 8.4: Pairs Completeness und Anteil der redundanten Vergleiche für verschiedene Blocking-Strategien unter Verwendung von einem bis vier Blockschlüsseln pro Datensatz (links). Vorschrift zur Ableitung der Blockschlüssel aus den Attributen der Datensätze (rechts).

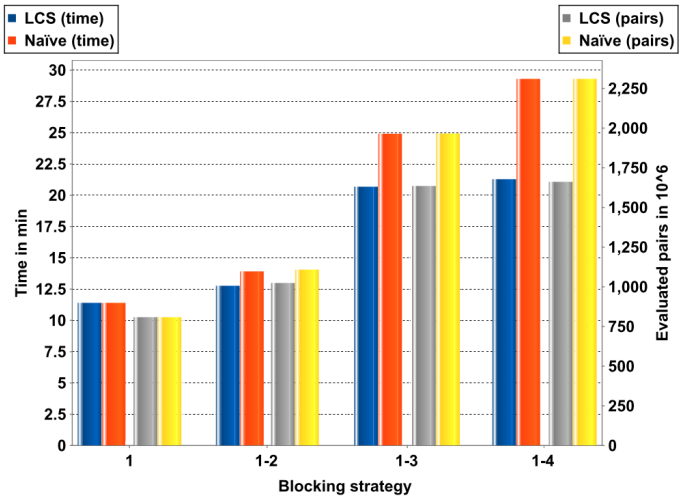


Abbildung 8.5: Ausführungszeit und Anzahl durchgeführter Ähnlichkeitsberechnungen für alle Blocking-Strategien mit und ohne Vermeidung redundanter Paarvergleiche.

von 75%, ein Viertel aller *Matches* konnte also nicht gefunden werden, da die jeweiligen Datensätze nicht in einem gemeinsamen Cluster liegen. Durch Hinzunahme weiterer (von anderen Attributen abgeleiteter) Blockschlüssel kann die Match-Qualität deutlich erhöht werden, bei Verwendung aller vier Blockschlüssel wird eine nahezu perfekte Pairs Completeness erreicht.

Gleichzeitig führt die Verwendung mehrerer Blockschlüssel pro Datensatz zu einem beträchtlichen Anteil redundanter Ähnlichkeitsberechnungen von bis zu 28%. Abbildung 8.5 vergleicht die gemessenen Ausführungszeiten mit und ohne Eliminierung der redundanten Paarvergleiche nach dem LCS-Prinzip. Die LCS-Strategie schneidet für alle drei Konfigurationen mit mehr als einem Blockschlüssel besser ab, als die naive Variante ohne Berücksichtigung der redundanten Paarvergleiche. Insgesamt konnte beobachtet werden, dass die Einsparung der Laufzeit der LCS-Implementierung proportional zum Grad der Redundanz ist, so konnte eine Laufzeiteinsparung von 8% für zwei Blockschlüssel und sogar von 30% für vier Blockschlüssel pro Datensatz erzielt werden.

8.4.2 Grad der Redundanz

Im zweiten Experiment wird die Robustheit der LCS-Strategie für einen systematisch variierten Grad an Redundanz untersucht. Der Grad der Redundanz wird durch Verwendung einer fixen Anzahl an Clustern, deren Größe schrittweise erhöht wird, kontrolliert. Bei einer Gesamtanzahl von n Datensätzen wird initial eine Aufteilung der Datenquelle in c gleich große Cluster der Größe $a = n/c$ vorgenommen. Dazu wird der Datensatz $0 \leq j < n$ dem Cluster $\lfloor j/c \rfloor$ zugewiesen, sodass Cluster $0 \leq i < c$ alle

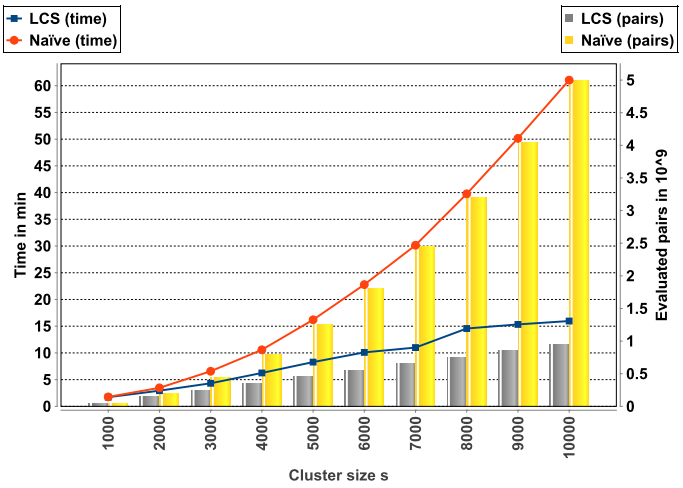


Abbildung 8.6: Ausführungszeiten und Anzahl durchgeführter Ähnlichkeitsberechnungen für verschiedene Grade an Redundanz.

Datensätze des Intervalls $[i \cdot a,\ (i+1) \cdot a - 1]$ enthält. In der Folge bleiben die Anzahl der Cluster c sowie deren untere Intervallgrenzen konstant. Anschließend wird jedes Cluster vergrößert, indem die obere Intervallgrenze nach rechts verschoben wird, sodass jedes Cluster $a \leq s < n/2$ Datensätze umfasst. Dementsprechend enthält das Cluster $0 \leq i < c$ alle Datensätze des Intervalls $[i \cdot a, (i \cdot a + s - 1) \bmod n]$. Bei dieser Vorgehensweise bestimmt die Clustergröße s die Überlappung der Cluster und damit den Grad der redundanten Paarvergleiche. Die Anzahl redundanter Ähnlichkeitsberechnungen beträgt insgesamt $c \cdot \frac{s(s-1)}{2} - c \cdot a \cdot (\frac{a-1}{2} + s - a)$.

Abbildung 8.6 vergleicht die Laufzeiten der LCS-Strategie und der naiven Ähnlichkeitsberechnung aller resultierenden Paare. Dazu wurde eine Teilmenge von $n = 100.000$ Datensätzen der Datenquelle DS1 in $c = 100$ Cluster aufgeteilt. Anschließend wurde die initiale Clustergröße von $a = 1.000$ schrittweise bis zu einer Clustergröße von $s = 10.000$ Datensätzen erhöht. Dies resultiert in einer Paarüberlappung zwischen 25% für $s = 2.000$ und 81% für $s = 10.000$. Während die Ausführungszeit der naiven Strategie proportional zur Anzahl der Kandidatenpaare $c \cdot \frac{s(s-1)}{2}$ steigt, profitiert die LCS-Implementierung stark von der Vermeidung redundanter Paarvergleiche und terminiert deutlich schneller als die naive Strategie. So konnte für die größte Clustergröße von $s = 10.000$ eine Reduktion der Ausführungszeit um den Faktor 4 erreicht werden. Trotzdem wird für alle Konfigurationen dieselbe Pairs Completeness erreicht. In Abbildung 8.6 ist zu erkennen, dass die Ausführungszeit der LCS-Implementierung etwas stärker als die Anzahl tatsächlich durchzuführender Ähnlichkeitsberechnungen steigt. Dies wird dadurch verursacht, dass trotzdem für *jedes* Kandidatenpaar überprüft werden muss, ob die Ähnlichkeitsberechnung übersprungen werden kann. Zudem steigt mit dem Grad der Redundanz die Anzahl der Signaturen eines Datensatzes, was die durchschnittliche Zeit zur Überprüfung der Signaturlisten in der Reduce-Phase erhöht. Die Ergebnisse zeigen jedoch, dass der Overhead zur Erkennung redundanter Paarvergleiche im Vergleich zu den erzielbaren Laufzeiteinsparungen sehr klein ist.

8.4.3 Einfluss auf Lastbalancierung

Im dritten Experiment wird die LCS-Implementierung mit der Exact- und Heuristic-Strategie für verschiedene Grade an Redundanz verglichen. Zu diesem Zweck wird dieselbe Konfiguration ($n = 100.000$ Datensätze, $c = 100$ Cluster, Clustergröße $s \in [1.000, 10.000]$) wie im vorigen Experimente verwendet. Alle drei Ansätze garantieren, dass die Ähnlichkeit eines jeden Kandidatenpaars genau einmal berech-

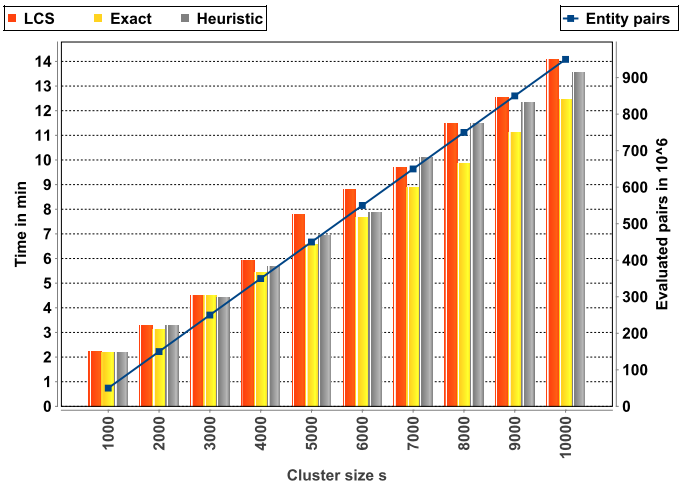


Abbildung 8.7: Vergleich der vorgestellten Strategien zur Eliminierung redundanter Paarvergleiche für verschiedene Grade an Redundanz (siehe Konfiguration aus Abschnitt 8.4.2).

net wird, und liefern das gleiche Ergebnis. Die Exact- und Heuristic-Strategien versuchen jedoch zu gewährleisten, dass die Lastbalancierung nicht durch die Eliminierung redundanter Paarvergleiche in negativer Weise beeinträchtigt wird.

Abbildung 8.7 zeigt die Laufzeiten der drei Verfahren zur Eliminierung redundanter Paarvergleiche. Die Anzahl der eliminierten Paare steigt aufgrund der fixen linken Intervallgrenzen mit der Clustergröße s. Für geringe Grade an Redundanz sind kaum Unterschiede der drei Implementierungen zu beobachten. Zwar basiert die Lastbalancierung bei der LCS-Strategie auf falschen Annahmen, jedoch hat die verhältnismässig geringe Anzahl eliminierter Paarvergleiche nur unwesentliche Auswirkungen auf die Workload der Reduce-Tasks. Mit steigender Clustergröße ist jedoch zu beobachten, dass die LCS-Strategie den anderen Strategien unterlegen ist. Mit einem zunehmenden Überlappungsgrad steigt die Anzahl der eliminierten Paarvergleiche und damit die Auswirkungen der Falschannahmen auf die Lastbalancierung. Insbesondere die Exact-Strategie schneidet für $s \geq 4.000$ im Vergleich zur LCS-Strategie jeweils um ca. 10% besser ab. Der zusätzliche Aufwand zur Überprüfung eines jeden Kandidatenpaares im Analyse-Job konnte durch eine gleichmäßigere Aufteilung aller durchzuführenden Paarvergleiche auf die $r = 200$ Reduce-Tasks amortisiert werden. Für die Heuristic-Strategie gilt dies nur bedingt, im Schnitt konnte lediglich eine Verbesserung von ca. 3% erreicht werden. Offenbar kann der zusätzliche Overhead zur Bestimmung der *Schnittmenge* der Signaturlisten eines Kandidatenpaares in der Reduce-Phase nur in begrenztem Maße durch eine gleichmäßige Aufteilung der eliminierten Paare auf alle Blöcke aufgewogen werden. Die Ergebnisse demonstrieren erneut, dass eine möglichst perfekte Lastbalancierung entscheidend für die effiziente Ausführung von MapReduce-Programmen ist.

8.5 Zusammenfassung

Effiziente PSC erfordert eine Einschränkung der Kandidatenmenge sowie eine Parallelisierung der teuren Ähnlichkeitsberechnungen. Um eine Robustheit gegenüber fehlenden, fehlerhaften und inkonsistenten Attributwerten gewährleisten zu können, ist es notwendig jeden Datensatz mehreren Clustern zuzuweisen. Ein Nebeneffekt dieser Vorgehensweise ist die unnötige mehrfache Ähnlichkeitsberechnung derselben Datensatzpaare. Am Beispiel der MapReduce-basierten Entity Resolution wurde zunächst ein einfacher Ansatz vorgeschlagen, der redundante Paarvergleiche vermeidet. Die Evaluation zeigte, dass die LCS-Strategie einer naiven Auswertung aller Kandidatenpaare für alle Redundanzgrade

deutlich überlegen ist. Abschließend wurden die Auswirkungen der Eliminierung von Paarvergleichen auf die Lastbalancierung untersucht. Im Rahmen eines Experimentes konnte gezeigt werden, dass die durch die Eliminierung redundanter Ähnlichkeitsberechnungen verursachten Falschannahmen für ein hohes Maß an Überlappung nicht vernachlässigbar sind. So konnte bei Verwendung der Exact-Strategie, trotz des erheblichen Mehraufwandes während der Datenanalyse, eine Verbesserung um ca. 10% gegenüber der LCS-Strategie erreicht werden.

9 Iterative Berechnung zusammenhängender Graphkomponenten

Ein typischer Nachbearbeitungsschritt des Entity Resolution-Prozesses ist die Berechnung der transitiven Hülle des Match-Ergebnisses. Dies erlaubt z. B. die Verwendung kleinerer Fenstergrößen beim Einsatz des Sorted Neighborhood-Verfahrens. Ein thematisch verwandtes Problem ist die Berechnung zusammenhängender Komponenten eines Graphen. In diesem Kapitel wird untersucht, wie Berechnungen dieser Art effizient mit MapReduce parallelisiert werden können. Die Betrachtung erfolgt losgelöst vom Entity Resolution-Problem, die vorgeschlagenen Algorithmen und Strategien sind jedoch problemlos auf das Entity Resolution-Problem übertragbar und wurden dementsprechend in das Dedoop-Framework integriert (vgl. Abschnitt 3.3). Nach einer Einführung und einer Betrachtung der verwandten Arbeiten im Abschnitt 9.1 wird zu diesem Zweck ein kürzlich veröffentlichter Algorithmus namens CC-MR untersucht (Abschnitt 9.2). Basierend auf dieser Analyse werden im Abschnitt 9.3 verschiedene Erweiterungen des CC-MR-Algorithmus vorgestellt. Die Evaluation im Abschnitt 9.4 zeigt, dass mit den vorgenommenen Optimierungen eine deutliche Beschleunigung des ursprünglichen Algorithmus erreicht werden kann.

9.1 Einführung und verwandte Arbeiten

Eine Vielzahl von Big Data-Anwendungen erfordert die effizienter Verarbeitung großer Graphstrukturen, z. B. zur Verwaltung sozialer Netze. Auch Anwendungen im Unternehmenskontext müssen eine Vielzahl miteinander verbundener Entitäten, wie Kunden, Produkte, Angestellte und assoziierte Geschäftsprozesse, wie Angebote und Rechnungen verarbeiten, die zu Analysezwecken als Graph repräsentiert werden können [195]. Die Erkennung zusammenhängender Graphkomponenten ist ein fundamentaler Schritt zur Erkennung von Mustern und zum Clustering von Entitäten, die miteinander in Beziehung stehen. Eine zusammenhängende Graphkomponente eines ungerichteten Graphen ist ein maximaler Subgraph, in dem alle Knoten durch einen Pfad miteinander verbunden sind. Abbildung 9.1 zeigt einen Graphen, bestehend aus zwei zusammenhängenden Komponenten.

9.1.1 Zielstellung

Eine effiziente Verarbeitung von Graphen mit Millionen von Knoten und Kanten verlangt eine Parallelverarbeitung. Das MapReduce-Framework wurde bereits frühzeitig zur Parallelisierung von Berechnungen auf in Rechnerclustern verteilt gespeicherten Graphen eingesetzt [54, 59, 155]. Die Berechnung zusammenhängender Graphkomponenten ist ein inhärent iterativer Prozess. Eine MapReduce-Implementierung resultiert in der wiederholten Ausführung eines MapReduce-Jobs, wobei die Ausgabe einer Iteration die Eingabe der nächsten Iteration bildet. Für eine effiziente Berechnung ist es dementsprechend von entscheidender Bedeutung, das Datenvolumen der Zwischenergebnisse sowie die Anzahl der Iterationen zu minimieren, da jede Iteration mit einem signifikanten Overhead I/O- und Task

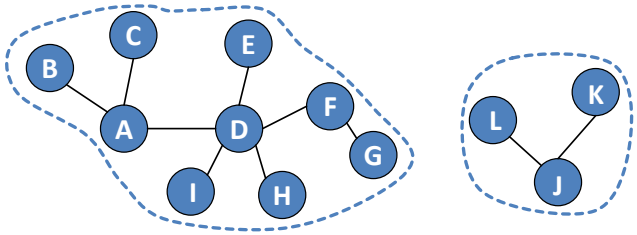


Abbildung 9.1: Beispiel eines Graphen bestehend aus zwei zusammenhängenden Komponenten.

Scheduling-Overhead einhergeht. Obwohl zuletzt eine Vielzahl frei verfügbarer Frameworks, wie z. B. *Apache Giraph*, *GraphX*[1], *GraphLab PowerGraph*[2]), entwickelt wurden, die direkt auf die Parallelisierung von Graphalgorithmen zugeschnitten sind, erfordert die große Popularität und Verbreitung des Hadoop-Frameworks das Finden effizienter MapReduce-Implementierungen. In diesem Kapitel wird ein aktueller MapReduce-Algorithmus zur Berechnung von zusammenhängenden Graphkomponenten namens CC-MR analysiert und durch verschiedene Optimierungen deutlich verbessert.

Die Berechnung der transitiven Hülle einer binären Relation, repräsentiert durch einen Graphen, ist ein thematisch verwandtes Problem. Im Zusammenhang mit Entity Resolution kommt die Berechnung der transitiven Hülle der berechneten *Matches* zum Einsatz, um weitere, indirekt verbundene, *Matches* zu finden. Aufgrund des enormen Datenvolumens der Zwischenergebnisse, die zwischen den Iterationen materialisiert werden müssen, sollte bei der MapReduce-Berechnung der transitiven Hülle unbedingt von einer expliziten Bestimmung aller (indirekt) miteinander verbundenen Paare abgesehen werden. So entspricht die transitive Hülle der linken, neun Datensätze umfassenden, Komponente aus Abbildung 9.1, einer Menge von 36 Paaren $(A,B),(A,C),\ldots,(H,I)$. Effizienter ist hingegen die Bestimmung des Clusters $\{A,B,C,D,E,F,G,H,I\}$, aus dem im Nachgang bei Bedarf alle Paare abgeleitet werden können. Die Bestimmung dieses Cluster ist äquivalent zur Bestimmung einer zusammenhängenden Graphkomponente mit einer minimalen Anzahl an Kanten. Dazu wird ein Knoten (hier A) als Repräsentant ausgewählt und mit jedem anderen Knoten der Komponente verbunden, alle übrigen Kanten werden verworfen. Die linke Komponente des Beispiels aus Abbildung 9.1 wird dementsprechend durch die acht Paare $(A,B),(A,C),\ldots,(A,I)$ ausgedrückt. Die in diesem Kapitel betrachteten Algorithmen haben das Ziel, einen gegebenen ungerichteten Eingabegraphen in eine Menge solcher sternförmiger Komponenten mit minimaler Kantenanzahl zu transformieren.

9.1.2 Verwandte Arbeiten

Da die in diesem Kapitel betrachtete Problematik weitesgehend losgelöst von den in den vorigen Kapiteln betrachteten Themen ist und die erarbeiteten Lösungsstrategien nicht spezifisch für das Entity Resolution-Problem sind, wurde auf eine Diskussion der relevanten Literatur im Kapitel 2 verzichtet. Im Folgenden wird daher zunächst ein Überblick über die relevanten Forschungsarbeiten gegeben.

Parallele Berechnung der transitiven Hülle: Einer der ersten Algorithmen zur parallelen Berechnung der transitiven Hülle einer binären Relation $R(a,b)$ wurde 1988 im Kontext paralleler Datenbanksysteme vorgeschlagen [231]. Der TCPO-Algorithmus basiert auf einer verteilten Berechnung von Verbund- und Vereinigungsoperationen. Die Anwendung einer Hashfunktion h auf das Attribut b erzeugt n Partitionen $R_1,\ldots,R_n$ die auf n Knoten (Prozessoren) verteilt werden. Anschließend wird dieselbe Hashfunktion auf das Attribut a der Ausgangsrelation R angewendet. Die resultierenden Partitionen $D_1^1,\ldots,D_n^1$ werden ebenfalls den n Knoten zugewiesen. In der Folge berechnet jeder Knoten i die

[1] `http://spark.apache.org/graphx`
[2] `http://graphlab.org/projects/source.html`

Relation $D_i^2 = \pi_{a,b}(R_i \bowtie_{b=a} D_i^1)$ und fügt sie zum Teilergebnis $T_i = D_i^1 \cup D_i^2$ hinzu. Die Paare der neu berechneten Relation D_i^2 werden im nächsten Schritt durch Anwendung von h auf das Attribut a neu über die n Knoten verteilt, sodass jeder Knoten i die Relationen $D_i^3 = \pi_{a,b}(R_i \bowtie_{b=a} D_i^2)$ und $T_i = T_i \cup D_i^3$ berechnen kann. Dieser Prozess wird solange fortgeführt bis in Iteration j keine neuen Tupel generiert werden, also bis $\bigcup_{i=1}^{n} D_i^j = \emptyset$ gilt. Das finale Ergebnis ist durch die Vereinigung der Teilergebnisse T_i gegeben.

In [44] wurde eine Verbesserung des *TCPO*-Algorithmus vorgeschlagen. Dabei wird eine Double Hashing-Technik verwendet, um das in jeder Iteration umzuverteilende Datenvolumen zu begrenzen. Beide Ansätze sind parallele Implementierungen eines sequentiellen iterativen Algorithmus, der in [16] vorgeschlagen wurde und nach d Iterationen terminiert, wobei d die Tiefe des Graphen bezeichnet. Der *Smart*-Algorithmus [116] verbessert dieses Verfahren, indem die Anzahl der benötigten Iterationen auf $\log d + 1$ beschränkt wird. Eine mögliche MapReduce-Implementierung des Smart-Algorithmus wurde in [4] diskutiert, der Ansatz wurde jedoch nicht evaluiert. Die vorgeschlagene Umsetzung übersetzt *jede* Iteration des Smart-Algorithmus in *mehrere* MapReduce-Jobs, die sequentiell ausgeführt werden müssen. Da MapReduce eine Materialisierung der (Zwischen-) Ergebnisse eines jeden MapReduce-Jobs erfordert, ist der Ansatz für große Graphen nicht praktikabel.

Bestimmung zusammenhängender Graphkomponenten: Die Bestimmung zusammenhängender Graphkomponenten ist ein gut erforschtes Problem. Traditionelle Ansätze traversieren dazu den Graphen mittels Breiten- oder Tiefensuche und weisen eine lineare Laufzeitkomplexität auf [225]. Zur effizienten Bearbeitung großer Graphen wurden später parallele Algorithmen mit logarithmischer Laufzeitkomplexität entwickelt [15, 113, 218]. Eine vergleichende Evaluation dieser Algorithmen erfolgte in [92]. Diese Ansätze basieren auf dem *Parallel Random Access Machine*-Modell und sind deswegen nicht für das MapReduce-Programmiermodell, das von einer Shared Nothing-Architektur ausgeht, geeignet. Die Autoren von [37] schlugen einen Algorithmus zur Berechnung in Shared Memory-Clustern vor, in denen Knoten über entfernte Hauptspeicherzugriffe miteinander kommunizieren können. Da das MapReduce-Framework für die vollständig unabhängige Bearbeitung disjunkter Datenpartitionen vorsieht, ist dieser Ansatz ebenfalls nicht anwendbar.

Ein erster MapReduce-Algorithmus zur Berechnung zusammenhängender Graphkomponenten wurde in [54] vorgeschlagen. Der wesentliche Nachteil dieses Ansatzes ist, dass pro Iteration drei MapReduce-Jobs sequentiell auszuführen sind. Zudem existieren weitere Ansätze, die darauf abzielen, die Anzahl der benötigten MapReduce-Jobs zu minimieren [124, 147]. Diese Verfahren sind jedoch dem CC-MR-Algorithmus, der in [214] vorgeschlagen wurde, deutlich unterlegen. Für einen Graphen der Tiefe d benötigt der CC-MR-Algorithmus im schlimmsten Fall d Iterationen, in der Praxis zeigt sich nach Aussage der Autoren jedoch ein logarithmisches Laufzeitverhalten. Der CC-MR-Algorithmus wird im folgenden Abschnitt im Detail vorgestellt. Ein unabhängig entwickelter, sehr ähnlicher Ansatz wurde nahezu gleichzeitig in [208] veröffentlicht. Die Autoren schlugen vier verschiedene Algorithmen vor, wobei der mit dem besten Laufzeitverhalten im Wesentlichen dem CC-MR-Algorithmus entspricht.

Neuere Frameworks zur verteilten Graphverarbeitung, wie Google Pregel [159], basieren auf dem *Bulk Synchronous Parallel*-Paradigma (BSP), welches die verteilte Berechnung in eine Sequenz sogenannter Supersteps aufteilt. Jeder Superstep besteht aus einer Phase der parallelen Berechnung, die von einer Phase des Datenaustauschs und einer Synchronisationsbarriere gefolgt wird. BSP-Algorithmen sind i. Allg., aufgrund des deutlich kleineren Overheads pro Iteration, besser für die Berechnung iterativer Graphalgorithmen geeignet als das MapReduce-Programmiermodell. In [208] wurde jedoch gezeigt, dass MapReduce-Algorithmen in stark ausgelasteten Clustern vergleichbaren BSP-Algorithmen überlegen sein können.

9.2 Der CCMR-Algorithmus

Die Eingabe des CC-MR-Algorithmus [214] besteht aus einem Graphen (V, E) mit einer Knotenmenge V und einer Menge $E \subseteq V \times V$. Das Ziel des CC-MR-Algorithmus ist die Transformation des Ausgangsgraphen in eine Menge sternförmiger Subgraphen. Unter Ausnutzung einer totalen Ordnung der Knoten (z. B. die lexikographische Sortierung der Knotenlabel) wird zu diesem Zweck in einem iterativen Prozess jeder Knoten mit seinem kleinsten Nachbarn verbunden. Während der Berechnung der CCs wird dazu für jeden Knoten v und seine (aktuellen) Nachbarknoten $adj(v)$ geprüft, ob v der kleinste dieser Knoten ist. Ist dies der Fall (*Local Max*-Zustand), werden alle $u \in adj(v)$ dem Knoten v zugewiesen. Dies entspricht dem Einfügen einer Kante zwischen v und jedem $u \in adj(v)$ und dem Verwerfen aller Kanten zwischen zwei Knoten $u_i, u_j \in adj(v)$. Ein Subgraph im *Local Max*-Zustand bildet bereits eine zusammenhängende Komponente, jedoch können weitere Knoten existieren, die zu dieser Komponente gehören, aber noch entdeckt werden müssen. Ist v hingegen nicht kleiner als alle seiner Nachbarknoten, liegt der sogenannte *Merge*-Zustand vor. Sei $u \in adj(v)$ mit $u < v$ der kleinste Nachbarknoten von v. Im *Merge*-Zustand werden v und $adj(v) \backslash \{u\}$ dem Knoten u zugewiesen. Dies entspricht dem Verschmelzen der Subgraphen mit den Zentren v und u zu einem neuen sternförmigen Subgraphen mit dem Zentrum u. Diese Schritte werden iterativ solange angewendet bis keine Subgraphen mehr verschmolzen werden.

Im Folgenden wird die MapReduce-Implementierung dieser Idee erläutert. Des Weiteren wird der vom CC-MR-Algorithmus verwendete Lastbalancierungsansatz für die effiziente Handhabung ungleichmäßig großer Komponenten diskutiert. Abbildung 9.2 illustriert die Berechnung der zusammenhängenden Graphkomponenten für den Beispielgraphen aus Abbildung 9.1. Insgesamt sind drei Iterationen notwendig, um die beiden im unteren rechten Teil der Abbildung 9.2 gezeigten Komponenten mit den Zentren A und J zu berechnen.

9.2.1 MapReduce-Implementierung

Eine Kante $v-u$ wird durch ein Schlüssel-Wert-Paar (v, u) repräsentiert. Um für jeden Knoten v zu entscheiden, ob er sich im *Local Max*-Zustand oder im *Merge*-Zustand befindet, wird v und jeder $u \in adj(v)$ zum selben Reduce-Task umverteilt. Zusätzlich wird gewährleistet, dass auf alle $u \in adj(v)$ in sortierter Reihenfolge zugegriffen werden kann. Dies erlaubt es dem Reduce-Task, allein durch den Vergleich von v und dem ersten $u \in adj(v)$ zu entscheiden, welcher der beiden Fälle vorliegt. Wie in Abbildung 9.2 dargestellt ist, gibt die `map`-Funktion dazu für jede Kante (v, u) des Eingabegraphen die Schlüssel-Wert-Paare (v⊙u, u) und (u⊙v, v) aus. Die Zuweisung der `map`-Ausgabepaare zu den Reduce-Tasks erfolgt durch Anwendung einer Partitionierungsfunktion, die lediglich die erste Schlüsselkomponente der zusammengesetzten Schlüssel verwendet. Die bewirkt, dass alle Knoten, die mit v verbunden sind, zum selben Reduce-Task umverteilt werden. Analog dazu werden alle Knoten, die mit u verbunden sind, demselben Reduce-Task zugewiesen. Die Reduce-Tasks sortieren die Schlüssel-Wert-Paare ihrer Eingabepartitionen komponentenweise anhand der zusammengesetzten Schlüssel. Anschließend erfolgt eine Gruppierung benachbarter Schlüssel-Wert-Paare, deren erste Schlüsselkomponente übereinstimmt. Eine Gruppe $v : [val_1, \ldots, val_n]$ besteht dabei aus dem Schlüssel für den Knoten v (der Wert der zweiten Schlüsselkomponente ist, aufgrund der Gruppierung nach der ersten Komponente, undefiniert) und einer *sortierten* Wertliste $val_1, \ldots, val_n$, welche den Nachbarn $adj(v)$ des Knotens v entspricht. So bearbeitet der erste Reduce-Task des ersten MapReduce-Jobs in Abbildung 9.2 die Gruppe $A : [B, C, D]$. Anschließend wird für jede Gruppe die `reduce`-Funktion aufgerufen.

Algorithmus 9.1 zeigt den Pseudo-Code der `reduce`-Funktion. Diese vergleicht jeden durch den `reduce`-Eingabe-Schlüssel gegebenen Knoten v mit seinem kleinsten Nachbarn $first \in adj(v)$. Falls $v < first$ gilt (*Local Max*-Zustand), ist v bereits der kleinste Knoten der (Sub-) Komponente. In diesem Fall wird ein Schlüssel-Wert-Paar (v, u) für jeden Knoten $u \in adj(v)$ ausgegeben (Zeile 15 bis 17 und 23

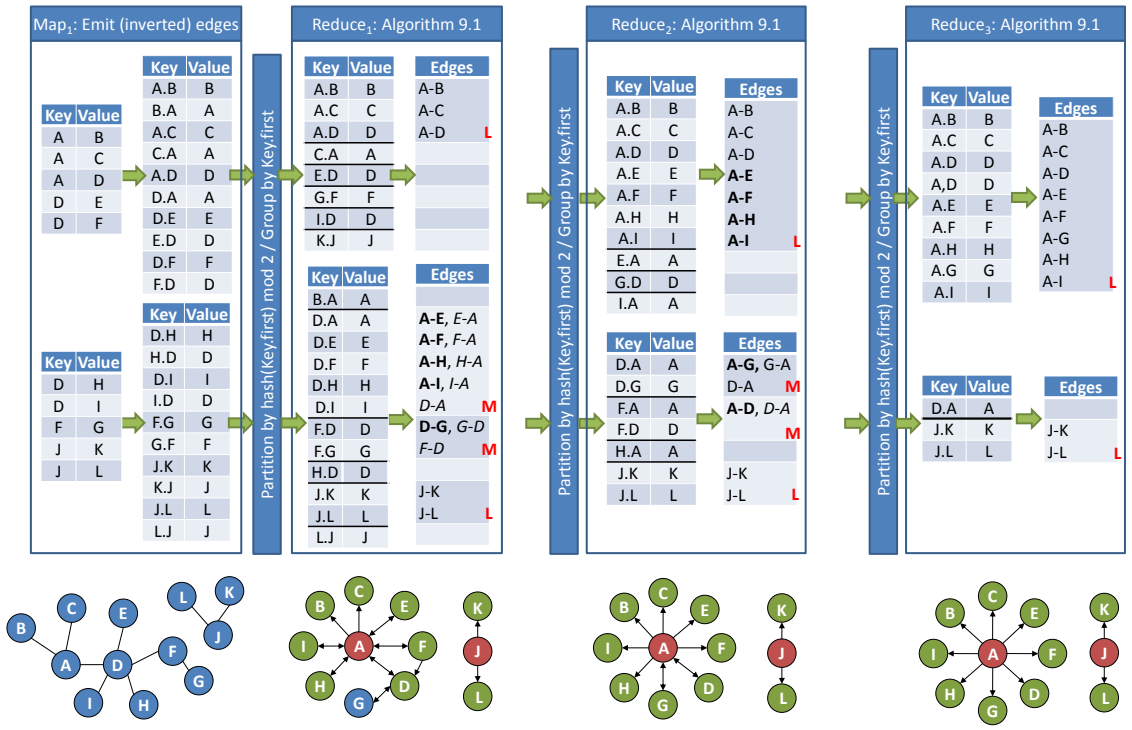


Abbildung 9.2: Der obere Teil der Abbildung illustriert den Datenfluss des CC-MR-Algorithmus für den Graphen aus Abbildung 9.1. Ein rotes L kennzeichnet einen *Local Max*-Zustand, relevante *Merge*-Zustände sind durch ein rotes M gekennzeichnet. Neu entdeckte Kanten sind fett gedruckt dargestellt. Der Algorithmus terminiert nach drei Iterationen, da keine neuen Rückwärtskanten (kursiv gedruckt) generiert wurden. Die Map-Phase jeder Iteration $i > 1$ gibt die Ausgabe der vorigen Iteration i unverändert aus und ist hier aus Platzgründen weggelassen. Der untere Teil der Abbildung zeigt den nach jeder Iteration resultierenden Graphen. Rote Knoten kennzeichnen die kleinsten Knoten der Komponenten. Korrekt zugewiesene Knoten sind grün markiert, blaue Knoten müssen noch einem anderen Knoten zugewiesen werden.

bis 24). So befinden sich z. B. die (Sub-) Komponenten $A : [B, C, D]$ und $J : [K, L]$ der ersten Iteration in Abbildung 9.2 im *Local Max*-Zustand, was durch ein rotes L gekennzeichnet ist.

Im *Merge*-Zustand, also wenn $v > first$ gilt, werden alle $u \in adj(v) \setminus \{first\}$ dem Knoten $first$ zugewiesen. Dazu gibt die `reduce`-Funktion ein Schlüssel-Wert-Paar (first, u) für jeden dieser Knoten u aus (siehe Algorithmus 9.1 Zeile 26). Zusätzlich gibt die `reduce`-Funktion für jeden Knoten u das invertierte Paar (u, first) aus (siehe Zeile 27). Sollte v kleiner als der größte Knoten der Menge $adj(first)$ sein, wird abschließend ein Schlüssel-Wert-Paar (v, first) ausgegeben (Zeile 32). Die letzten beiden Typen von Ausgabepaaren repräsentieren sogenannte *Rückwärtskanten*, die temporär zum Graphen hinzugefügt werden. Sie dienen als Brücken, um $first$ in späteren Iterationen mit weiteren Nachbarn von $\{v\} \cup adj(v) \setminus \{first\}$ verbinden zu können, die möglicherweise noch kleiner als $first$ sind.

Die *Merge*-Zustände in Abbildung 9.2, in denen eine Verschmelzung zweier Subgraphen stattfindet (die Liste der `reduce`-Eingabe-Werte umfasst mehr als ein Element), sind mit einem roten M markiert. Beispielsweise werden für die Gruppe $F : [D, G]$ in der ersten Iteration eine neue (Vorwärts-) Kante (D, G) sowie zwei Rückwärtskanten (G, D) und (F, D) ausgegeben. Die Gruppe $D : [A, E, F, H, I]$ befindet sich ebenfalls im *Merge*-Zustand, u. a. wird dabei die Rückwärtskante (D, A) ausgegeben. In der zweiten Iteration erweitert der erste Reduce-Task die Komponente des Knotens A um die in der vorigen Iteration

Algorithmus 9.1: Implementierung des CC-MR-Algorithmus

```
1  map_configure(jobConf)
2      iteration ← getIteration(jobConf);

3  map(Vertex v, Vertex u)
4      output(v.u, u);
5      if iteration==1 then
6          // Output reversed edge
7          // in first iteration
8          output(u.v, v);

9  // part: repartition map first key component
10 // cmp: sort by entire key
11 // group: group first key component

12 reduce(Vertex source, Iterator<Vertex> values)
13     locMaxState ← false;
14     first ← values.next();
15     if source.id < first.id then
16         locMaxState ← true;
17         output(source.first, first);      // Forward edge
18     last ← first;
19     while values.hasNext() do
20         cur ← values.next();
21         if cur.id = last.id then
22             continue ;          // Remove duplicates
23         if locMaxState then
24             output(source.cur, cur);  // Forward edge
25         else
26             output(first.cur, cur);    // Forward edge
27             output(cur.first, first);  // Backward edge
28             // Increment counter to signalize
29             // that a new iteration is required
30         lastId ← cur.id;
31     if ¬locMaxState ∧ (source.id < lastId) then
32         output(source.first, first);    // Backward edge
```

entdeckten Nachbarknoten E, F, H und I. Der zweite Reduce-Task verbindet bei der Verarbeitung der Gruppe $D : [A, G]$ die Knoten A und G. Ebenso wird die Kante (A, D) bei der Verarbeitung der Gruppe $F : [A, D]$ generiert. Es ist zu erkennen, dass Kanten mehrfach entdeckt werden können, so wird z. B. die Kante (A, D) von beiden Reduce-Tasks der zweiten Iteration erzeugt. Diese Duplikate werden entsprechend der Zeile 22 des Algorithmus 9.1 in der nächsten Iteration entfernt. Für (A, D) erfolgt dies durch den ersten Reduce-Task der dritten Iteration.

Die Ausgabe der i-ten Iteration bildet die Eingabe der Iteration $i + 1$. Die Map-Phase der folgenden Iterationen gibt für jede Eingabekante (v, u) ein Schlüssel-Wert-Paar (v.u⊙, u) aus. Im Gegensatz zur ersten Iteration werden *keine* invertierten Kanten erzeugt. Der Algorithmus terminiert, wenn während einer Iteration *keine* Rückwärtskanten ausgegeben wurden. Dies kann durch ein sogenanntes *Driver*-Programm, das wiederholt den gleichen MapReduce-Job ausführt, mithilfe eines Zählers festgestellt werden. Zu diesem Zweck kann auf die *Counters*-Bibliothek[3] des Hadoop-Frameworks zurückgegriffen werden: Jeder Tasktracker erhöht bei Ausgabe einer Rückwärtskante einen bestimmten Zähler, der Jobtracker aggregiert diese Werte automatisch gegen Ende der Ausführung eines MapReduce-Jobs. Da die Iteration $i > 1$ ausschließlich vom Ergebnis der vorhergehenden Iteration $i - 1$ abhängt, kann vor dem Start der Iteration i das HDFS-Eingabeverzeichnis der $i - 1$-ten Iteration durch das Ausgabeverzeichnis der Iteration $i - 1$ ersetzt werden. Dies erlaubt einen Neustart des MapReduce-Jobs mit den gleichen Parametern und minimiert das im HDFS abgelegte Datenvolumen.

9.2.2 Lastbalancierung

Im bisher beschriebenen Verfahren werden alle Knoten einer Komponente durch einen einzigen Reduce-Task bearbeitet. Dies bewirkt eine Eingrenzung der Skalierbarkeit und Laufzeiteffizienz, wenn der Graph aus vielen kleineren, aber einer großen Komponente besteht, die die Mehrzahl der Knoten des Graphen umfasst. Zur Behandlung solcher großen Komponenten beinhaltet der CC-MR-Algorithmus einfachen Lastbalancierungsansatz.

[3] `http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/Counters.html`

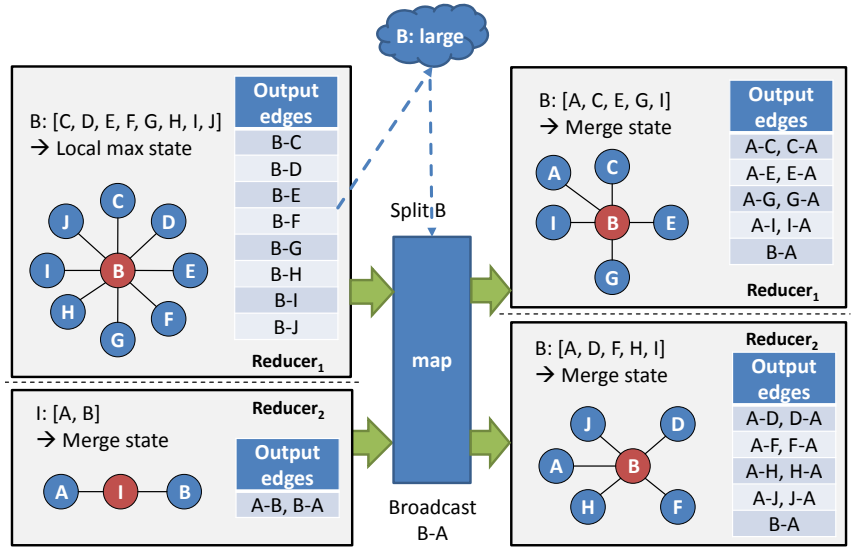


Abbildung 9.3: Lastbalancierungsmechanismus des CC-MR-Algorithmus.

Die Erkennung großer Komponenten erfolgt in der Reduce-Phase durch Zählen der Knoten einer Gruppe im *Local Max*-Zustand. Für Gruppen, deren Größe einen bestimmten Schwellwert überschreitet, wird der kleinste Knoten $source$ vom bearbeitenden Reduce-Task in einer *separaten* Ausgabedatei vermerkt, die von den Map-Tasks der folgenden Iterationen ausgewertet wird. Diese separaten Ausgabedateien (möglicherweise eine pro Iteration und Reduce-Task) sind jedoch nicht Bestandteil der regulären Eingabe der folgenden Iterationen. Vor Beginn der nächsten Iteration kopiert das externe *Driver*-Programm die separaten Ausgabedateien mittels des *DistributedCache*-Mechanismus in das lokale Dateisystem aller Tasktracker.

Während der folgenden Iteration liest jeder Map-Task diese Informationen zur Initialisierungszeit ein. Dies bewirkt, dass jeder Map-Task das Zentrum (den kleinsten Knoten) einer jeden großen Komponente "kennt". Bei Anwendung der `map`-Funktion auf eine *Vorwärtskante* (v,u) mit $v < u$ wird überprüft, ob v das Zentrum einer großen Komponente ist. Ist dies der Fall, so wird eine abweichende Umverteilung des `map`-Ausgabe-Paars (v⊙u, u) vorgenommen. Im Gegensatz zur Bearbeitung einer "normalen" Kante erfolgt die Zuweisung des Paars zu einem Reduce-Task durch Anwendung einer Partitionierungsfunktion auf die zweite (statt auf die erste) Komponente des zusammengesetzten Schlüssels. Dies bewirkt eine gleichmäßige Aufteilung aller $u \in adj(v)$ auf die Reduce-Tasks. Aus diesem Grund sollten Komponenten, die einmal als groß markiert wurden, in diesem Zustand verbleiben. Da jeder Reduce-Task in der Folge nur noch einen Teil der Komponente bearbeitet, könnte sonst fälschlicherweise eine Klassifikation in "nicht-groß" erfolgen. Rückwärtskanten (v,u) mit $v > u$, bei denen v das Zentrum einer großen Komponente ist, werden an *alle* Reduce-Tasks gesendet. Dies ist notwendig, um sicherzustellen, dass alle $w \in adj(v)$, die über alle Reduce-Tasks verteilt sind, mit u verbunden werden können. Abgesehen von diesen Modifikationen bleibt der Algorithmus 9.1 unverändert.

Die Arbeitsweise des Lastbalancierungsansatzes ist in Abbildung 9.3 dargestellt. Im diesem fiktiven Beispiel wird die Komponente $B : [C, D, E, F, G, H]$ als Kandidat für die Lastbalancierung identifiziert. In der darauffolgenden Map-Phase werden alle Nachbarn von B anhand ihres Labels gleichmäßig über alle (hier zwei) Reduce-Tasks aufgeteilt. Die Rückwärtskante (B,A) wird zu allen Reduce-Tasks gesendet, sodass in der darauffolgenden Iteration alle Nachbarn von B mit dem Knoten A verbunden werden können.

9.3 Optimierung des CCMR-Algorithmus

Trotz seiner Effizienz im Vergleich zu bestehenden MapReduce-Implementierungen hat der CC-MR-Algorithmus erhebliches Verbesserungspotential. Im Folgenden werden drei Erweiterungen vorgeschlagen und evaluiert, die die Anzahl der Iterationen und das Datenvolumen der Zwischenergebnisse reduzieren. Zunächst wird die Wahl des Zentrums einer zusammenhängenden Graphkomponente vorgeschlagen, die sich an der Anzahl der Knotennachbarn anstatt den Knotenlabels orientiert. Des Weiteren wird der CC-MR-Algorithmus dahingehend erweitert, dass bereits in der Map-Phase eine Verknüpfung zusammenhängender Knoten einer Eingabepartition erfolgt, um die verbleibende Arbeit für die Reduce-Tasks und die darauffolgenden Iterationen zu verringern. Die dritte Optimierung besteht in der Identifikation *stabiler* Komponenten, wie z. B. $J, [K, L]$, die in späteren Iterationen nicht weiterwachsen. Eine Separation von der regulären Ausgabe einer Iteration vermeidet eine unnötige Weiterverarbeitung stabiler Komponenten in den Folgeiterationen. Die vorgeschlagenen Änderungen haben keinen Einfluss auf die Komplexitätseigenschaften des CC-MR-Algorithmus – nach wie vor ist von einem logarithmischen Verhalten bezüglich der Tiefe des Eingabegraphen auszugehen.

9.3.1 Wahl der Komponentenzentren

Der CC-MR-Algorithmus weist alle (indirekt) miteinander verbundenen Knoten einer zusammenhängenden Graphkomponente dem Knoten der Komponente zu, der das lexikographisch kleinste Label[4] aufweist. Dies ist ein naheliegender und einfach zu implementierender Ansatz, der in Kombination mit den konstruierten zweistelligen `map`-Ausgabe-Schlüsseln die automatisch vom Hadoop-Framework vorgenommene Sortierung der `reduce`-Eingabedaten ausnutzt. Die Wahl des Knotens mit dem kleinsten Label als Zentrum einer Komponente berücksichtigt jedoch nicht die Struktur des Graphen, was möglicherweise zu einer unnötig hohen Anzahl benötigter Iterationen führen kann. So müssen z. B. für die linke Komponente aus Abbildung 9.1 die fünf Knoten E, F, G, H, I dem Knoten A zugewiesen werden (B, C, D sind bereits Nachbarn von A). Würde stattdessen der Knoten D als Zentrum der Komponente deklariert werden, so müssten lediglich die drei Knoten B, C, G umgehängt werden. Im schlimmsten Fall befindet sich der Knoten mit dem kleinsten Label am Anfang einer langen Kette von Knoten. In diesem Falle wäre ein Knoten in der Mitte der Kette ein geeigneterer Kandidat für das Zentrum einer Komponente.

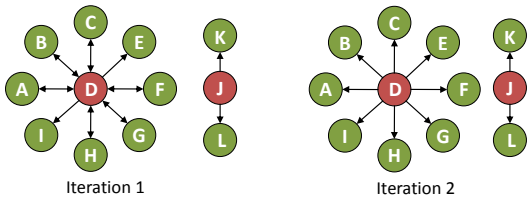
Die CC-MR-VD-Strategie verfolgt eine ähnliche Idee. Statt dem Knoten mit dem kleinsten Label wird derjenige Knoten zum Zentrum bestimmt, der die höchste Anzahl an Nachbarknoten (Knotengrad) im Eingabegraphen hat. Obwohl dies keine optimale Lösung garantiert, verspricht dieser Ansatz eine Verringerung der Gesamtanzahl an Knotenneuzuordnungen und damit der Anzahl an Kanten, die pro Iteration ausgegeben werden. Zudem ist es möglich, dass die Anzahl der benötigten Iterationen reduziert werden kann. Ist der Grad eines jeden Knotens bekannt, so kann eine weitere Optimierung gegenüber dem CC-MR-Algorithmus vorgenommen werden. Die Ausgabe von Rückwärtskanten (v, u) mit $v > u$ in der Reduce-Phase kann eingespart werden, sofern v den Knotengrad 1 hat. Diese Reduktion des ausgegebenen (und in späteren Iterationen zu verarbeitenden) Datenvolumens ist möglich, da v keine weiteren Nachbarn hat, die in der Folgeiteration mit u verbunden werden müssen.

Zur Bestimmung der Knotengrade benötigt die CC-MR-VD-Strategie einen zusätzlichen leichtgewichtigen MapReduce-Job. Die Ausgabe dieses Jobs besteht aus nach Knotenlabel indexierten Dateien, die unter Verwendung von Hadoops *MapFileOutputFormat*[5] erzeugt werden und ein externspeicherbasiertes Nachschlagen des Knotengrades für ein gegebenes Knotenlabel ermöglichen. Das Ergebnis dieses Vorverarbeitungsschritts kann je nach Datenvolumen entweder mittels des Distributed Cache-Mechanismus zu allen Knoten kopiert oder von diesen, bei Bedarf, aus dem HDFS eingelesen werden.

[4] Bei der Berechnung der transitiven Hülle eines Match-Ergebnisses entspricht das Knotenlabel dem Datensatzidentifikator.
[5] `http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/MapFileOutputFormat.html`

Vertex	D	A	F	J	B	C	E	G	H	I	K	L
Degree	5	3	2	2	1	1	1	1	1	1	1	1

Modified sort order



		CC-MR		CC-MR-VD	
		Forward	Backward	Forward	Backward
Iteration	1	9	7	8	5
	2	10	5	8	0
	3	8	0		

Generated Edges

Abbildung 9.4: Aus dem initialen Knotengrad resultierende Knotenordnung und Ausgabe jeder Iterationen bei Verwendung der CC-MR-VD-Strategie (oberer Teil). Vergleich der CC-MR- und CC-MR-VD-Strategie bezüglich der Anzahl ausgegebener Kanten (unterer Teil).

In der Map-Phase der ersten Iteration werden diese Informationen verwendet, um jeden Knoten mit seinem Knotengrad anzureichern. Im weiteren Verlauf werden Knoten nicht mehr alleinig anhand der Knotenlabel, sondern zuerst nach dem Knotengrad (in absteigender Sortierreihenfolge) und anschließend nach Knotenlabel (in aufsteigender Sortierreihenfolge) sortiert. Abbildung 9.4 zeigt den nach jeder Iteration resultierenden Graphen bei Anwendung der CC-MR-VD-Strategie auf den Beispielgraphen aus Abbildung 9.1. Entsprechend der modifizierten Knotensortierung wird Knoten *D* als Zentrum der linken Komponente gewählt. Dadurch kann eine Iteration eingespart und die Anzahl der insgesamt ausgegebenen Kanten beinahe halbiert werden.

Diesen Vorteilen stehen jedoch einige Nachteile gegenüber. Neben dem zusätzlichen Aufwand zur Berechnung der Knotengrade erhöht sich die Größe der einzelnen Datensätze, da ein Datensatz neben dem Label auch durch seinen Knotengrad beschrieben wird. Zusätzlich geht der externspeicherbasierte wahlfreie Punktzugriff auf den Knotengradindex in der Map-Phase der ersten Iterationen mit einem zusätzlichen Overhead einher, der proportional zur Größe des Eingabegraphen ist. Die Evaluation im Abschnitt 9.4 wird zeigen, dass diese Nachteile für große Graphen nicht durch die erzielten Einsparungen amortisiert werden können.

9.3.2 Lokale Berechnung von Teilkomponenten in der Map-Phase

Die Verarbeitung der Eingabedaten in der Map-Phase des CC-MR-Algorithmus ist zustandslos. Ein Map-Task wendet die `map`-Funktion auf jede Kante seiner Eingabepartition an und gibt diese (abgesehen von der Erzeugung des zusammengesetzten Schlüssels) unverändert aus. Das Ergebnis eines Aufrufs der `map`-Funktion ist dabei unabhängig von anderen Daten der Eingabepartition. Die CC-MR-Mem-Strategie verfolgt hingegen den Ansatz, eine fixe Anzahl aufeinanderfolgender Eingabekanten im Hauptspeicher zu puffern und diese bereits in der Map-Phase zu Teilkomponenten zu verbinden. Bei der Bestimmung dieser Teilkomponenten werden zusammenhängende Knoten nach derselben Strategie, wie zuvor, einem ausgewählten Zentrumsknoten der Komponente zugewiesen. Nach der Ausgabe dieser Teilkomponenten wird der Puffer geleert, um die nächste Partition der Eingabekanten puffern und verbinden zu können. Überlappende Teilkomponenten, die in verschiedenen Runden oder von verschiedenen Reduce-Tasks berechnet wurden, werden, wie zuvor, in der Reduce-Phase verschmolzen.

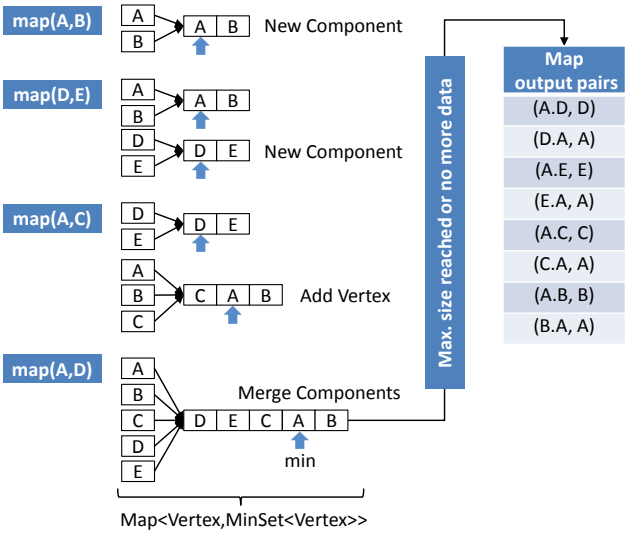


Abbildung 9.5: Lokale Berechnung von Teilkomponenten in der Map-Phase.

Der effizienten Berechnung von Teilkomponenten in der Map-Phase kommt bei Anwendung der CC-MR-Mem-Strategie eine entscheidende Bedeutung bei. Zu diesem Zweck bietet sich die Repräsentation von Komponenten als Menge zusammenhängender Knoten an, für die der kleinste Knoten vermerkt wird (*MinSet*). Wie in Abbildung 9.5 gezeigt, wird mittels einer Hashtabelle jeder Knoten auf die Komponente abgebildet, zu der er gehört. Eine Komponente und ihr kleinster Knoten werden aktualisiert, wenn ein neuer Knoten hinzugefügt wird oder eine Verschmelzung mit einer anderen Komponente erfolgt. Im Beispiel von Abbildung 9.5 formen die ersten beiden Kanten die zwei Mengen $\{A, B\}$ und $\{D, E\}$. Die dritte Kante (A, C) führt zu einer Erweiterung der ersten Menge um den Knoten C. Die vierte Eingabekante (A, D) vereinigt beide Mengen zur Teilkomponente $\{A, B, C, D, E\}$ mit dem kleinsten Knoten $\min\{A, D\} = A$. Diese Verschmelzung ist so implementiert, dass alle Knoten der Menge mit der geringeren Kardinalität zur Menge mit der größeren Kardinalität hinzugefügt werden. Zusätzlich müssen deren Einträge in der Hashtabelle auf die größere Menge aktualisiert werden. Nachdem die vorgegebene Maximalanzahl von Eingabekanten auf die Weise verarbeitet wurde, werden die generierten Komponenten verwendet, um `map`-Ausgabe-Paare entsprechend des Schemas im rechten Teil der Abbildung 9.5 zu erzeugen.

Algorithmus 9.2 zeigt den Pseudo-Code der `map`-Funktion der CC-MR-Mem-Strategie. Eingabekanten werden, wie beschrieben, zur components-Hashtabelle hinzugefügt. Wenn deren Größe einen Schwellwert übersteigt oder sich keine weiteren Eingabedaten in der Eingabepartition des Map-Tasks befinden, werden die Teilkomponenten, wie folgt, ausgegeben. Für jeden Knoten $v \neq c.min$ einer Teilkomponente c werden die Vorwärtskante $(min.v, v)$ sowie die Rückwärtskante $(v.min, min)$ ausgegeben. Die Partitionierung, Sortierung und Gruppierung bleiben gegenüber dem CC-MR-Algorithmus unverändert. Dies gilt ebenso für die `reduce`-Funktion, die einzige Ausnahme ist, dass in Zeile 27 und 32 des Algorithmus 9.1 keine Rückwärtskanten ausgegeben werden, da dies bereits in der Map-Phase erfolgt. Genau wie beim CC-MR-Algorithmus erfolgt eine Aufteilung von Vorwärtskanten großer Komponenten auf alle Reduce-Tasks sowie ein Broadcasting von Rückwärtskanten großer Komponenten zur allen Reduce-Tasks, um eine balancierte Reduce-Phase gewährleisten zu können.

Die CC-MR-Mem-Strategie findet bereits Teilkomponenten in der Map-Phase, sodass die verbleibende Arbeit für die Reduce-Phase reduziert wird. Zudem wird i. Allg. eine Reduktion der Anzahl benötigter Iterationen sowie des Datenvolumens der Zwischenergebnisse, die zwischen den Iterationen im verteilten Dateisystem materialisiert werden, erreicht. Dem steht ein erhöhter Speicherbedarf sowie eine

Algorithmus 9.2: CC-MR-Mem (Map-Phase)

```
1  map_configure(JobConf job)
2      max ← job.getBufferSize();
3      components ← new HashMap<Vertex,MinSet>(max);
4  map(Vertex v, Vertex u)
5      if components.size() ≥ max then
6          generateOutput();
7      comp1 ← components.get(v);
8      comp2 ← components.get(u);
9      if (comp1 ≠null) ∧ (comp2 ≠null) then
10         if comp1 ≠comp2 then                          // Merge
11             if comp1.size() ≥ comp2.size() then
12                 comp1.addAll(comp2);
13                 foreach Vertex w ∈ comp2 do
14                     components.put(w, comp1);
15             else
16                 comp2.addAll(comp1);
17                 foreach Vertex w ∈ comp1 do
18                     components.put(w, comp2);
19     else if comp1 ≠null then                          // Add Vertex
20         comp1.add(u);
21         components.put(u, comp1);
22     else if comp2 ≠null then                          // Add Vertex
23         comp2.add(v);
24         components.put(v, comp2);
25     else                                              // New component
26         MinSet component= new MinSet(v, u);
27         components.put(v, component);
28         components.put(u, component);
29 map_close()
30     generateOutput();
31 generateOutput()
32     foreach component ∈ components.values() do
33         if ¬component.isMarkedAsProcessed() then
34             component.markAsProcessed();
35             min ← component.min;
36             foreach Vertex v ∈ component do
37                 if v≠min then
38                     output(min.v, v);      // Forward edge
39                     output(v.min, min); // Backward edge
40     components.clear();
```

gestiegene Komplexität der Map-Phase gegenüber. Die Größe des Eingabepuffers ist dabei ein Konfigurationsparameter, der es erlaubt, zwischen zusätzlichem Overhead und möglichen Einsparungen abzuwägen. Eine Kombination des CC-MR-Mem-Ansatzes mit dem CC-MR-VD-Ansatz ist problemlos möglich.

9.3.3 Erkennung und Separation stabiler Komponenten

Große Graphen können aus vielen Komponenten mit stark unterschiedlicher Größe (Knotenanzahl) bestehen. Üblicherweise werden kleine und mittelgroße Komponenten nach deutlich weniger Iterationen vollständig entdeckt als große Komponenten. Wenn eine Komponente während einer Iteration weder von einer anderen Komponente "verschluckt" wird, noch wächst, wird sie auch in allen Folgeiterationen unverändert bleiben und kann deshalb als stabil betrachtet werden. So wird z. B. die Komponente $J : [K, L]$ in Abbildung 9.2 bereits während der ersten Iteration erkannt. Da in der zweiten Iteration weder weitere Knoten hinzukommen, noch eine Zuweisung der Knoten J, K, L zu einem Zentrumsknoten erfolgt, bleibt die Komponente bis zur Terminierung des Algorithmus unverändert. Das Ziel der dritten Optimierung des CC-MR-Algorithmus ist es, solche stabilen Komponenten effizient zu erkennen und von der regulären Ausgabe, die Eingabe der Folgeiteration ist, zu trennen. Zu diesem Zweck werden stabile Komponenten in separate Ausgabedateien geschrieben, die nicht von den Map-Tasks der Folgeiterationen gelesen werden. Dies vermeidet den unnötigen Aufwand, diese Knoten in den Folgeiterationen aus dem verteilten Dateisystem zu lesen, umzuverteilen und am Ende jeder Iteration unverändert wieder auszuschreiben.

Die Optimierung wird zunächst für den CC-MR-Algorithmus bzw. für die CC-MR-VD-Strategie beschrieben. Abbildung 9.6 illustriert den Ansatz für das fortlaufende Beispiel. Eine Komponente mit

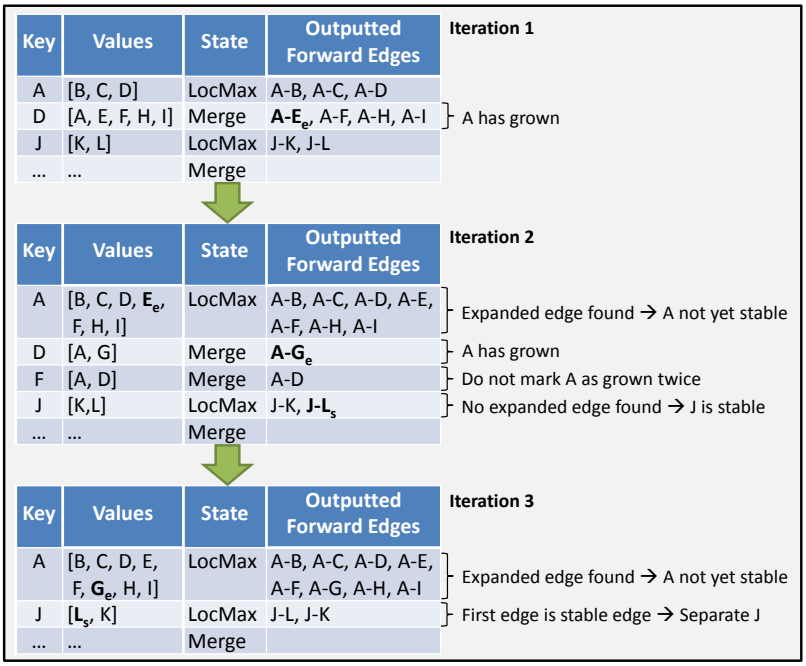


Abbildung 9.6: Erkennung und Separation stabiler Komponenten in der Reduce-Phase des CC-MR-Algorithmus (vgl. Abbildung 9.2).

dem (kleinsten) Zentrumsknoten v wächst, wenn es mindestens einen *Merge*-Zustand $u : [v, w, \ldots]$ mit $v < u$ gibt. In diesem Fall gibt die `reduce`-Funktion neue Vorwärtskanten $(v, w), \ldots$ sowie entsprechende Rückwärtskanten $(w, v), \ldots$ aus. Aufgrund der Rückwärtskanten kann die Komponente mit dem Zentrum v in der Folgeiteration weiterwachsen. Um dem Reduce-Task, der diese Komponente in der Folgeiteration bearbeitet, darüber zu informieren, dass diese noch nicht stabil ist, wird die *erste* Vorwärtskante (v, w) mit einem speziellen *Expanded*-Flag erweitert, was durch ein Schlüssel-Wert-Paar (v, w_e) ausgedrückt wird. In Abbildung 9.6 wird dementsprechend die erste Vorwärtskante (A, E) des *Merge*-Zustands $D : [A, E, F, H, I]$ zur Kante (A, E_e) erweitert. Sollte ein Reduce-Task mehrere Komponenten im *Merge*-Zustand mit demselben kleinsten Knoten v bearbeiten, so wird das *Expanded*-Flag nur einmal gesetzt, um den weiteren Aufwand bei der Verarbeitung gewachsener Komponenten (insbesondere im Zusammenhang mit der Lastbalancierung) zu begrenzen. So wird z. B. nur die erste Vorwärtskante (A, G_e) der beiden *Merge*-Zustände $D : [A, G]$ und $F : [A, D]$, die in der zweiten Iteration der Abbildung 9.6 vom selben Reduce-Task bearbeitet werden (vgl. Abbildung 9.2), mit einem Flag versehen.

In der `reduce`-Funktion jeder Iteration $i > 1$ wird jede Komponente mit dem Zentrum v im *Local Max*-Zustand geprüft, ein Knoten w_e der Komponente mit dem *Expanded*-Flag versehen ist, was bedeutet, dass die Komponente in der letzten Iteration um mindestens einen Knoten gewachsen ist. Wenn kein solcher Knoten w_e gefunden wird, so kann die Komponente als stabil betrachtet werden. In der zweiten Iteration der Abbildung 9.6 wird beispielsweise der markierte Knoten E_e gefunden, sodass die Komponente mit dem Zentrum A nicht von der regulären Ausgabe separiert werden kann. Die Frage, ob eine Komponente stabil ist oder nicht, kann erst beantwortet werden, nachdem die letzte Eingabekante von der `reduce`-Funktion bearbeitet wurde. Da Komponenten sehr groß sein können, ist es i. Allg. nicht möglich, dass der bearbeitende Reduce-Task alle `reduce`-Eingabekanten im Hauptspeicher puffert. Aus diesem Grund gibt jeder Reduce-Task bei der iterativen Verarbeitung der Werteliste kontinuierlich jede Ausgabekante aus. Lediglich die *letzte* Kante (v, z_s) wird mit einem sogenannten *Stable*-Flag versehen, sofern die Komponente mit dem Zentrum v in der vergangenen Iteration nicht gewachsen ist.

Eine stabile Komponente wird in der Folgeiteration von der regulären Ausgabe getrennt. Zu diesem Zweck gibt ein Map-Task, der eine Kante (v, z_s) liest, ein $(v.\perp, z_s)$-Schlüssel-Wert-Paar[6] statt des üblichen $(v.z, z_s)$ Paars aus. Dies bewirkt, dass z_s der erste Knoten in der Liste der `reduce`-Eingabewerte des Schlüssels v ist, sodass bereits bei der Verarbeitung des ersten Knotens bekannt ist, dass die Komponente stabil ist und alle Kanten problemlos in eine separate Reduce-Task-spezifische Ausgabedatei in das verteilte Dateisystem geschrieben werden können. In Abbildung 9.6 gibt es für die Komponente $J : [K, L]$ in der zweiten Iteration keine Knoten mit einem *Expanded*-Flag. Dementsprechend wird die Komponente als stabil markiert, indem die letzte ausgegebene Kante (J, L_s) mit einem *Stable*-Flag annotiert wird. In der dritten Iteration (was auch die letzte Iteration für das kleine fortlaufende Beispiel ist) wird die Komponente mit dem Zentrum J schließlich von der regulären Ausgabe separiert. Das finale Ergebnis der Berechnung besteht aus den regulären Ausgabedateien der letzten Iteration und den zusätzlichen Ausgabedateien aller Iterationen (potentiell eine Datei pro Reduce-Task für jede Iteration $i > 2$). Es sei angemerkt, dass der beschriebene Ansatz nahezu keinen zusätzlichen Overhead in Bezug auf das Datenvolumen oder den benötigten Hauptspeicher verursacht.

Für große Komponenten, deren Knoten zu Lastbalancierungszwecken von mehreren Reduce-Tasks bearbeitet werden, müssen Kanten, die ein *Expanded*- oder *Stable*-Flag aufweisen zu allen Reduce-Tasks gesendet werden. Um Duplikate im finalen Ergebnis zu vermeiden, werden diese Kanten jedoch nur von genau einem Reduce-Task (z. B. dem Reduce-Task mit dem Index 0) ausgegeben.

Die beschriebene Optimierung kann ebenfalls mit der CC-MR-Mem-Strategie kombiniert werden. Wenn eine Komponente als stabil markiert ist, kann dabei auf die Ausgabe von Rückwärtskanten in der Zeile 39 des Algorithmus 9.2 verzichtet werden. Ein wesentlicher Unterschied im Vergleich zum CC-MR-Algorithmus ist, dass Komponenten auch in der Map-Phase wachsen können[7], sodass auch in der Map-Phase Kanten $(v.w, w_{e\_map})$ mit *Expanded*-Flag generiert werden müssen. In der `reduce`-Funktion wird eine Komponente im *Local Max*-Zustand dementsprechend als stabil betrachtet, wenn sich weder ein Knoten w_e, noch ein w_{e_map} in der Liste der `reduce`-Eingabe-Werte befindet. Im Gegensatz zu den Kanten (v, w_e), die in der Reduce-Phase der vorigen Iteration generiert wurden, müssen Kanten (v, w_{e_map}), die in der Map-Phase der aktuellen Iteration erzeugt wurden, in der Reduce-Phase ausgegeben werden, um die Information, dass die Komponente mit dem Zentrum v gewachsen ist, in die nächste Iteration transportieren zu können.

9.4 Evaluation

In diesem Abschnitt wird das Laufzeit- und Skalierbarkeitsverhalten der drei Erweiterungen mit dem CC-MR-Algorithmus verglichen. Zur Evaluation werden hinreichend große und frei verfügbare Standardgraphen aus der *Stanford Large Network Dataset Collection*[8] verwendet, die häufig zur Evaluierung von Graph-Algorithmen herangezogen werden (siehe Abbildung 9.7). Google Web ist dabei die kleinste Datenquelle und enthält Links zwischen Webseiten. Die Datenquelle Patent citations ist ca. vier-mal größer und beinhaltet Zitierungen zwischen gewährten Patenten. Der Live Journal-Graph repräsentiert Freundschaftsbeziehungen zwischen Nutzern einer Online Community. Im Vergleich zum Patent citations-Graph hat er eine kleinere Knoten- aber ungefähr vier-mal größere Kantenanzahl.

Im ersten Experiment werden zunächst nur die CC-MR-VD- und CC-MR-Mem-Erweiterungen ohne Behandlung stabiler Komponenten betrachtet. Die Experimente wurden in einem Amazon EC2-Cluster bestehend aus 20 Instanzen des Typs c1.medium mit je zwei virtuellen Prozessorkernen sowie einer dedizierten Master-Instanz des Typs m1.small durchgeführt. Auf jedem Knoten wurde Hadoop 0.20.2 mit

[6] Je nachdem durch welchen Datentyp ein Knoten repräsentiert wird, kann $\perp$ als leere Zeichenkette oder Integer.MIN_VALUE gesetzt werden.

[7] Dies ist der Fall, wenn $u \neq min$ in Zeile 20, $v \neq min$ in Zeile 23 oder wenn in den Zeilen 11 bis 18 des Algorithmus 9.2 zwei Komponenten verschmolzen werden.

[8] `http://snap.stanford.edu/data`

Dataset	Nodes	Input edges	#CCs
Google Web	875,713	5,105,039	2,746
Patent citations	3,774,768	16,518,948	3,627
Live Journal	4,847,571	68.993.773	1,876
Memetracker	185,050,250	453,192,771	47,021,622

Abbildung 9.7: Charakteristika der zur Evaluation verwendeten Eingabegraphen.

einer Map- und Reduce-Task-Kapazität von zwei eingerichtet. Die Gesamtanzahl von Reduce-Task pro Iteration wurde dementsprechend auf 40 gesetzt. In einem zweiten Experiment wurde der Effekt der Separation stabiler Komponenten untersucht. Zu diesem Zweck wurde ein vierter Graph aus der Memetracker [9]-Datenquelle erzeugt, die Webdokumente, die bestimmte Phrasen oder Aussagen enthalten, mitsamt ihrer Linkstruktur umfasst. Im Gegensatz zu den anderen drei Graphen handelt es sich beim Memetracker-Graphen nicht um einen dichten Graphen, er besteht vielmehr aus vielen kleinen Komponenten und isolierten Knoten. Wegen der großen Größe des Graphen wurde das Cluster für das zweite Experiment auf 40 Knoten des Typs m1.xlarge mit einer Map- und Reduce-Task-Kapazität von zwei erhöht. Die Gesamtanzahl der Reduce-Tasks wurde folglich auf 80 erhöht. Im dritten Experiment wurde die Skalierbarkeit des CC-MR- und des CC-MR-Mem-Algorithmus verglichen. Zu diesem Zweck wurde erneut der Memetracker-Graph verwendet und die Clustergröße schrittweise von $n = 1$ auf $n = 100$ Knoten des Typs m1.xlarge erhöht. Für n Knoten wurde die Anzahl der Reduce-Tasks auf $2 \cdot n$ gesetzt.

Alle Experimente wurden mit dem im Abschnitt 9.2.2 vorgestellten Lastbalancierungsverfahren durchgeführt. Wie in [214] wurde dabei eine Komponente als groß betrachtet, wenn ihre Größe 1% der Anzahl der in der vorigen Iteration ausgegebenen Vorwärtskanten überschreitet. Beim Einsatz der CC-MR-Mem-Strategie wurde die maximale Anzahl an Kanten, die ein Map-Task im Hauptspeicher puffert, auf 20.000 gesetzt.

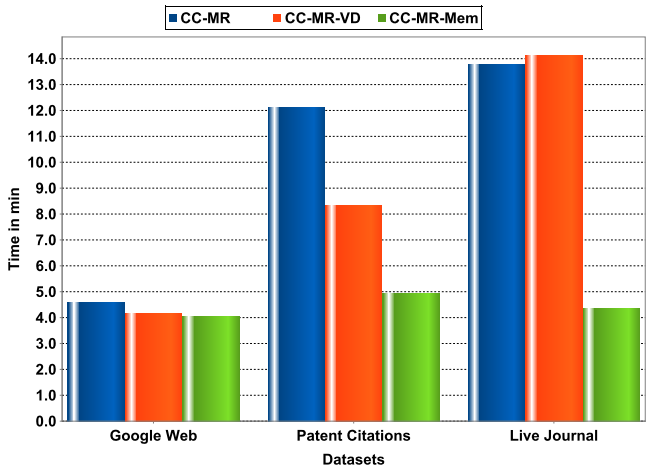
9.4.1 Vergleich mit dem CCMR-Basisalgorithmus

Zunächst werden die drei Algorithmen CC-MR, CC-MR-VD (Berücksichtigung der Knotengrade) und CC-MR-Mem (lokale Berechnung von Teilkomponenten in der Map-Phase) für die ersten drei Datenquellen evaluiert. Dabei werden folgende vier Kriterien berücksichtigt: Anzahl benötigter Iterationen, Ausführungszeit, Anzahl in der Reduce-Phase ausgegebener Kanten (über alle Iterationen hinweg) sowie das korrespondierende Datenvolumen. Die Ergebnisse sind in Abbildung 9.8 dargestellt.

Die Ergebnisse zeigen, dass die CC-MR-Mem-Strategie für alle untersuchten Fälle besser als der CC-MR-Algorithmus abschneidet. Mit zunehmender Problemgröße wächst der erzielte Geschwindigkeitsvorteil bis zum Faktor 3,2 für den Live Journal-Graphen. Zusätzlich konnte das in das verteilte Dateisystem ausgeschriebene Datenvolumen bis zu einem Faktor von 8.8 für den Live Journal-Graphen verringert werden. Die CC-MR-Mem-Strategie profitiert offenbar von der hohen Dichte der Eingabegraphen und ist in der Lage, überlappende Teilkomponenten frühzeitig in der Map-Phase zu verbinden, die sonst erst in der Reduce-Phase späterer Iterationen verbunden worden wären. Dies bewirkt, dass weniger Vorwärts- und Rückwärtskanten generiert werden, was wiederum die Wahrscheinlichkeit einer früheren Terminierung des Algorithmus erhöht. Für jedes betrachtete Problem benötigt die CC-MR-Mem-Strategie zwei Iteration weniger als das CC-MR-Verfahren.

Für alle Graphen führt die CC-MR-VD-Strategie die Berücksichtigung der Knotengrade zu einer deutlichen Reduktion der Anzahl ausgegebener Kanten. Die Anzahl der Verschmelzungen von Teilkomponenten im *Merge*-Zustand konnte vermindert werden, was für die Datenquellen Google Web und Patent Citations zu kürzeren Ausführungszeiten im Vergleich zum CC-MR-Algorithmus führte. Das insgesamt

[9] `http://snap.stanford.edu/data/memetracker9.html`



(a) Gesamtausführungszeit aller Verfahren für die drei betrachteten Datenquellen.

	CC-MR				CC-MR-VD				CC-MR-MEM			
	Iterations	Overall edges	HDFS Out (GB)	Time (min)	Iterations	Overall edges	HDFS Out (GB)	Time (min)	Iterations	Overall edges	HDFS Out (GB)	Time (min)
Google Web	8	$\approx 41 \cdot 10^6$	0.64	4.6	7	$\approx 27 \cdot 10^6$	0.78	4.2	6	$\approx 9 \cdot 10^6$	0.12	4.0
Patent Citations	8	$\approx 648 \cdot 10^6$	11.25	12.1	8	$\approx 311 \cdot 10^6$	8.98	8.3	6	$\approx 112 \cdot 10^6$	1.87	4.9
Live Journal	6	$\approx 640.10^6$	9.97	13.8	6	$\approx 441 \cdot 10^6$	12.33	14.1	4	$\approx 87 \cdot 10^6$	1.13	4.3

(b) Zusammenfassung der Ergebnisse unter Berücksichtigung verschiedener Kriterien.

Abbildung 9.8: Vergleich der CC-MR-, CC-MR-VD- und CC-MR-Mem Algorithmen für die Datenquellen Google Web, Patent Citations und Live Journal. Die für den CC-MR-VD-Algorithmus angegebene Anzahl benötigter Iterationen, die Ausführungszeit und das ausgegebene Datenvolumen beinhalten den Overhead des zusätzlichen MapReduce-Jobs zur Berechnung der Knotengrade.

ausgegebene Datenvolumen ist jedoch i. Allg. höher als beim CC-MR-Algorithmus, da alle Knoten, die in den Eingabegraphen durch einen Integer-Wert dargestellt sind, um einen weiteren ganzzahligen Wert zur Speicherung des Knotengrades angereichert werden müssen. Für andere denkbare Knotenrepräsentationen (z. B. Zeichenketten für URLs von Webseiten) kann dieses Verhältnis jedoch abweichend sein. Bei der Verarbeitung des größten Eingabegraphen Live Journal konnte der Overhead des zusätzlichen MapReduce-Jobs und der Annotation aller Knoten mit ihrem Knotengrad in der Map-Phase der ersten Iteration nicht durch die erzielten Einsparungen amortisiert werden. Das Ergebnis verdeutlicht, dass der externspeicherbasierte wahlfreie Zugriff auf den Knotengradindex der Schwachpunkt des CC-MR-VD-Verfahrens ist, sodass in der momentanen Form von dessen Einsatz für große Graphen abgeraten werden muss.

9.4.2 Berücksichtigung stabiler Komponenten

Im zweiten Experiment wird der Effekt der frühzeitigen Separation stabiler Komponenten, die in den Folgeiterationen unverändert bleiben, betrachtet. Im Rahmen des Experimentes wurde sowohl der CC-MR-Algorithmus als auch die CC-MR-Mem-Strategie mit und ohne Berücksichtigung stabiler Komponenten für den Memetracker-Graphen ausgeführt. Ohne die Berücksichtigung stabiler Komponenten konnte für die CC-MR-Mem-Strategie bereits eine um den Faktor 2,6 schnellere Ausführungszeit gegenüber dem Basisverfahren beobachtet werden (siehe Abbildung 9.9).

	CC-MR		CC-MR-Mem	
	regular	stable	regular	stable
Iterations	11	11	**9**	**8**
Overall Edges	$\approx 4.71 \cdot 10^9$	$\approx 4.15 \cdot 10^9$	$\approx 1.87 \cdot 10^9$	$\approx 1.30 \cdot 10^9$
HDFS out (GB)	552.4	467.3	212.0	133.7
Time (min)	74.7	71.6	28.8	21.6

Abbildung 9.9: Vergleich des CC-MR- und des CC-MR-Mem-Algorithmus für die Memetracker-Datenquelle mit und ohne Separation stabiler Komponenten.

Durch die frühzeitige Trennung stabiler Komponenten konnte für den CC-MR-Algorithmus lediglich eine Verbesserung um 4% erzielt werden. Offenbar haben, aufgrund des gleichbleibenden MapReduce-Overheads zur Job- und Task-Verwaltung, die über 11 Iterationen hinweg eingesparten 85 GB Datenvolumen nur einen geringen Einfluss auf die Gesamtausführungszeit. Im Gegensatz dazu profitiert die CC-MR-Mem-Strategie deutlich von der Separation stabiler Komponenten – die Ausführungszeit konnte um etwa 25% verringert werden. Verglichen mit dem CC-MR-Basis-Algorithmus konnte die Gesamtausführungszeit um den Faktor 3,5 von 74,7 auf 21,6 Minuten reduziert werden. Eine interessante Beobachtung in diesem Zusammenhang ist die Verringerung der Anzahl der für die CC-MR-Mem-Strategie benötigten Iterationen. Aufgrund der Trennung der stabilen Komponenten von des regulären Ausgabe sind in der Eingabemenge eines Map-Tasks keine stabilen Komponenten mehr enthalten. Dies erhöht die Wahrscheinlichkeit, dass zwei Komponenten innerhalb einer Eingabepartition, die sonst durch eine stabile Komponente getrennt wären, bereits in der Map-Phase verschmolzen werden, was normalerweise erst in der Reduce-Phase erfolgen würde. Im Vergleich zur CC-MR-Mem-Strategie ohne Betrachtung stabiler Komponenten konnte das insgesamt ausgeschriebene Datenvolumen um 37% reduziert werden was zusammen mit der eingesparten Iteration zu einer Verbesserung der Ausführungszeit um 25% führte.

9.4.3 Skalierbarkeit

Das dritte Experiment untersucht das Skalierbarkeitsverhalten des CC-MR-Algorithmus und der CC-MR-Mem-Erweiterung. Zu diesem Zweck werden erneut die zusammenhängenden Komponenten des größten betrachteten Memetracker-Graphen berechnet und die Clustergröße zwischen $n = 1$ und $n = 100$ Knoten variiert. Zur besseren Vergleichbarkeit wurde auf die Behandlung stabiler Komponenten verzichtet. Abbildung 9.10 illustriert die gemessenen Ausführungszeiten und Speedup-Werte.

Die Ausführung des CC-MR-Algorithmus auf einem einzigen Knoten schlug fehl, da während einer Iteration die Größe des Eingabegraphen, die Ausgabe der vorigen Iteration, die Ausgabe der Map-Phase der aktuellen Iteration (inkl. für Lastbalancierungszwecke replizierter Rückwärtskanten) und die Reduce-Ausgabe der aktuellen Iteration die verfügbare Plattenkapazität des Knotens überstieg. Aus diesem Grund wurde die Ausführungszeit des CC-MR-Algorithmus für $n = 1$ in Abbildung 9.10 als das Doppelte der für $n = 2$ benötigten Ausführungszeit angenommen. Im Gegensatz dazu war die CC-MR-Mem-Strategie, aufgrund des deutlich kleineren ausgegebenen Datenvolumens, in der Lage, alle zusammenhängenden Komponenten auf einem Knoten zu berechnen. Dabei war die Berechnung sogar schneller als die Ausführung des CC-MR-Algorithmus auf $n = 2$ Knoten.

Insgesamt zeigte sich die CC-MR-Mem-Strategie dem CC-MR-Algorithmus für alle untersuchten Clustergrößen überlegen. Die superlinearen Speedup-Werte in Abbildung 9.10 sind durch die Ausführungszeiten für $n = 1$ (2) mit nur zwei (vier) parallel ablaufenden Map- und Reduce-Tasks bedingt. In diesen Fällen unterliegen beide Ansätze massiven Lastbalancierungsproblemen, die zunächst erkannt werden müssen, bevor der Lastbalancierungsmechanismus in den Folgeiterationen greift. Für größere Clusterkonfigurationen werden diese Effekte abgemildert, sodass eine gleichmäßigere Auslastung der ver-

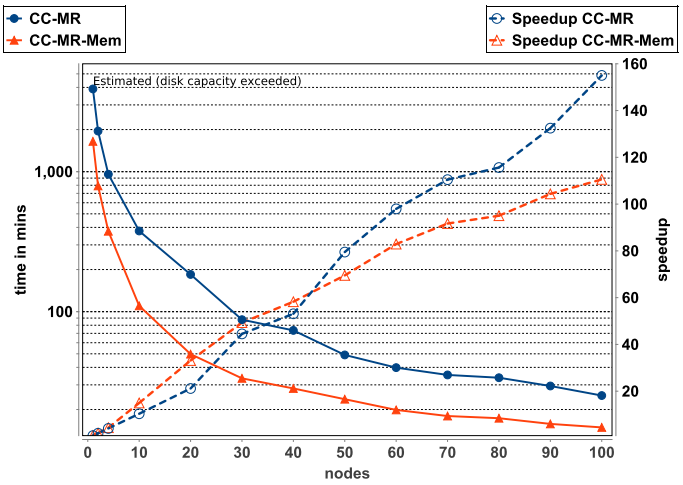


Abbildung 9.10: Ausführungszeiten und Speedup-Werte der CC-MR- und CC-MR-Mem-Algorithmen für den Memetracker Graphen (ohne Berücksichtigung stabiler Komponenten).

fügbaren Knoten erreicht werden kann. Bis zu $n = 60$ Knoten schneidet die CC-MR-Mem-Strategie um den Faktor 2 bis 4 besser als der CC-MR-Algorithmus ab. So sind zur Lösung des Problems in ca. 100 Minuten mit der CC-MR-Mem-Strategie lediglich zehn Knoten notwendig, wohingegen für den CC-MR-Algorithmus 30 Knoten eingesetzt werden müssen. Die Resultate zeigen, dass die CC-MR-Mem-Strategie ihre Effektivität, trotz des sinkenden Optimierungspotentials der lokalen Berechnung von Teilkomponenten, in der Map-Phase auch für eine wachsende Anzahl an Map-Tasks beibehält. Dies ist darin begründet, dass die Berechnung von Teilkomponenten innerhalb von Gruppen einer festen Größe (20.000 Kanten) und nicht innerhalb vollständiger `map`-Eingabepartitionen erfolgt. Trotz alledem sinkt der relative Vorteil der CC-MR-Mem-Strategie gegenüber dem CC-MR-Algorithmus für eine steigende Knotenanzahl. So entsprach die Verbesserung für $n = 60$ Knoten einem Faktor von 2, wohingegen für $n = 100$ "nur" noch eine Verbesserung um den Faktor 1,7 (14,9 zu 25,2 Minuten) beobachtet werden konnte.

9.5 Zusammenfassung

Die iterative Bestimmung zusammenhängender Komponenten großer Graphen erfordert eine Parallelisierung der Berechnung. In diesem Kapitel wurden drei Erweiterungen eines entsprechenden kürzlich veröffentlichten MapReduce-Algorithmus vorgestellt und evaluiert, die darauf abzielen, das Datenvolumen der Zwischenergebnisse und die Anzahl der benötigten Iterationen zu verringern. Die besten Resultate konnten dabei durch die CC-MR-Mem-Strategie erzielt werden, die eine Verbindung von Teilkomponenten sowohl in der Map- als auch in der Reduce-Phase jeder Iteration vorsieht. Die Evaluation zeigte, dass die CC-MR-Mem-Strategie dem CC-MR-Basis-Algorithmus, insbesondere für große Graphen, überlegen ist. Des Weiteren wurde vorgeschlagen, stabile Komponenten frühzeitig von der regulären Ausgabe zu separieren, um eine unnötige Verarbeitung in den Folgeiterationen zu vermeiden. Diese Erweiterung verursacht nahezu keine zusätzlichen Kosten, kann aber in Kombination mit der CC-MR-Mem-Strategie zu deutlichen Laufzeitverbesserungen für Graphen mit geringer Konnektivität führen.

Teil IV

Zusammenfassung und Ausblick

10 Zusammenfassung und Ausblick

10.1 Anwendungsfall: Vergleichende Evaluation verschiedener Entity Resolution-Strategien

In diesem Abschnitt erfolgt eine zusammenfassende Demonstration der Vielseitigkeit und Effizienz des Dedoop-Frameworks. Dazu wird eine vergleichende Evaluation verschiedener Blocking- und Klassifikationsstrategien für ein bibliographisches Entity Resolution-Problem vorgenommen. Nach der Dedoop-gestützten Einrichtung eines MapReduce-Clusters mit 20 Knoten in der Cloud-Umgebung Amazon EC2 wurden alle zu vergleichenden Entity Resolution-Workflows mittels Dedoops graphischer Benutzeroberfläche konfiguriert und anschließend ohne weitere Nutzerinteraktion automatisiert nacheinander ausgeführt.

Für dieses Experiment wurde erneut die bibliographische Datenquelle Scholar aus [140] verwendet, welche 64.263 Datensätze enthält, die aus Anfrageergebnissen an die *Google Scholar*-Suchmaschine für wissenschaftliche Publikationen extrahiert wurden. Aufgrund der Heterogenität der in der wissenschaftlichen Literatur verwendeten Zitierungsformate und der automatischen Extraktion bibliographischer Informationen aus gecrawlten PDF-Dateien durch den Google Scholar-Dienst sind die Datensätze generell sehr unsauber. Zur Bestimmung der Match-Qualität wurde ein Gold Standard aus dem manuell konstruierten, perfekten Match-Ergebnis aus [140] abgeleitet, der auf der Zuordnung von Google Scholar-Publikationen zu Publikationen der sauberen Datenquelle DBLP basiert. Da nicht alle Publikationsdatensätze der Scholar-Datenquelle einen korrespondierenden Datensatz in der Datenquelle DBLP haben (DBLP enthält lediglich Publikationen aus dem Bereich der Informatik), wurden berechnete *Matches* (a, b), von denen weder a noch b von DBLP abgedeckt wird, bei der Beurteilung der Match-Qualität ignoriert. Im Gegensatz zum Experiment im Abschnitt 5.5.4 besteht die Eingabe der Entity Resolution-Workflows jedoch aus allen 64.263 Datensätzen der Scholar-Datenquelle.

Insgesamt wurden fünf verschiedene Blocking-Strategien (Kartesisches Produkt sowie Standard Blocking und Sorted Neighborhood mit je einem bzw. zwei Blockschlüsseln pro Datensatz) mit den vier in Abbildung 10.1 dargestellten Klassifikationsstrategien kombiniert, sodass insgesamt 20 Entity Resolution-Workflows evaluiert wurden. Als Eingabe für den Klassifikationsschritt dienten jeweils die Dice-Koeffizienten der aus dem Publikationstitel bzw. dem Autor-Attribut gebildeten Trigram-Mengen. In einem Vorverarbeitungsschritt wurde vorab eine Konvertierung der Attributwerte in die Kleinschreibweise sowie eine Entfernung von Sonderzeichen und aufeinanderfolgenden Whitespaces vorgenommen. Die betrachteten Klassifikationsstrategien umfassen zwei manuell spezifizierte regelbasierte Strategien, die Mindestähnlichkeitsschwellwerte für den Publikationstitel (rule_1) bzw. das Titel- und Autor-Attribut (rule_2) zweier Datensätze definieren. Des Weiteren wurden mit einer Support Vector Machine-Implementierung sowie einem Entscheidungsbaumverfahren der WEKA-Bibliothek zwei lernbasierte Ansätze verwendet, um eine Klassifikation der Kandidatenpaare auf Basis der errechneten Ähnlichkeitswerte vorzunehmen. Für die Generierung des Klassifikationsmodells wurden bei den lernbasierten Ansätzen jeweils 500 manuell gelabelte Trainingsdaten bereitgestellt, die zufällig ausgewählt wurden, wobei *Matches* und *Non-Matches* zu je mindestens 40% vertreten sind (vgl. *Ratio*-Strategie [141]). Da

Match Classification	Input Similarity Features	Match Criterion
$Rule_1$	3-gram(title)	sim $\geq$ 0.8
$Rule_2$	3-gram(title), 3-gram(authors)	sim(title)$\geq$ 0.6 and sim(author)$\geq$ 0.4
SVM	3-gram(title), 3-gram(authors)	SVM (WEKA LibSVM, -K 0 -C 10)
DT	3-gram(title), 3-gram(authors)	Decision Tree (WEKA LMT, default config)

Abbildung 10.1: Evaluierte Klassifikationsstrategien.

	Standard Blocking								Sorted Neighborhood								Cartesian			
	Single pass (author)				Multi-pass (author, title)				Single pass (author/w=1000)				Multi-pass (author/w_1=500, (title/w_2=500)							
	$Rule_1$	$Rule_2$	SVM	DT	$Rule_1$	$Rule_2$	SVM	DT	$Rule_1$	$Rule_2$	SVM	DT	$Rule_1$	$Rule_2$	SVM	DT	$Rule_1$	$Rule_2$	SVM	DT
Comparisons	$\approx 3{,}7 \cdot 10^6$				$\approx 23{,}2 \cdot 10^6$				$\approx 63.7 \cdot 10^6$				$\approx 63.9 \cdot 10^6$				$\approx 2.06 \cdot 10^9$			
Time [in min]	1.73	1.77	2.21	2.14	1.97	2.03	2.57	2.58	2.48	2.48	3.34	2.73	2.68	2.67	3.35	2.74	24.00	25.08	44.61	35.94
Recall	25.5%	45.2%	48.6%	55.2%	45.7%	57.2%	73.0%	79.6%	28.2%	50.4%	52.7%	59.3%	44.8%	62.1%	72.4%	78.9%	52.2%	69.6%	93.5%	92.6%
Precision	96.1%	88.4%	70.4%	70.4%	86.8%	89.4%	73.3%	74.5%	95.6%	87.5%	85.8%	75.3%	87.0%	88.5%	74.1%	74.4%	84.7%	87.0%	66.0%	71.2%
F-Measure	40.3%	59.8%	57.5%	61.9%	59.9%	69.7%	73.2%	77.0%	43.6%	64.0%	65.3%	66.4%	59.1%	73.0%	72.2%	76.5%	64.6%	77.4%	77.4%	80.5%

Abbildung 10.2: Ausführungszeiten und Match-Qualität verschiedener Entity Resolution-Strategien für die Deduplizierung der Scholar-Datenquelle.

die Ergebnisse der lernbasierten Ansätze zufallsbehaftet sind, wurden die präsentierten Ergebnisse über 10 Durchgänge mit verschiedenen Trainingsdaten gemittelt.

Abbildung 10.2 zeigt die ermittelten Recall-, Precision- und F-Measure-Werte für die 20 untersuchten Entity Resolution-Strategien. Zusätzlich ist jeweils die Gesamtzahl der durchgeführten Paarvergleiche sowie die benötigte Ausführungszeit angegeben. Naturgemäß konnte die beste Match-Qualität ohne Blocking-Schritt, also bei Evaluierung des Kartesischen Produktes von mehr als zwei Milliarden Kandidatenpaaren, erreicht werden. Das beste erreichte F-Measure beträgt nur ca. 80%, was die Schwierigkeit des betrachteten Entity Resolution-Problems unterstreicht (bibliographische Entity Resolution-Probleme mit mindestens einer sauberen Datenquelle, wie z. B. DBLP, können typischerweise mit F-Measure-Werten von 90-95% gelöst werden [140]). Beim Einsatz von Blocking-Verfahren konnte mit dem Entscheidungsbaumverfahren die beste Match-Qualität erreicht werden, gefolgt von der SVM und der Verwendung der Regel $rule_2$. Die Regel $rule_1$, die alle Publikationen mit einer Titelähnlichkeit von mindestens 80% als *Match* klassifiziert, stellt sich als zu einfach und restriktiv heraus, da lediglich ein Recall von 52% erreicht wird. Dies bedeutet im Umkehrschluss, dass ungefähr die Hälfte aller *Matches* eine Titelähnlichkeit von kleiner als 80% aufweist, was erneut die schlechte Datenqualität unterstreicht.

Die zur Auswertung des Kartesischen Produkts benötigte Ausführungszeit variiert zwischen 25 Minuten für die regelbasierten und 36-45 Minuten für die lernbasierten Klassifikationsstrategien. Durch den Blocking-Schritt kann die Zahl der Kandidatenpaare um einen Faktor von $100-300$ und die Ausführungszeit auf $1{,}7-3{,}3$ Minuten reduziert werden. Diese Ausführungszeiten beinhalten die Ausführung des MapReduce-Jobs zum Lernen des Klassifikationsmodells (sofern notwendig) und des Analyse-Jobs zur Ermittlung der Datenverteilung. Als Blockschlüssel dienten die ersten drei Buchstaben des Autor-Attributes bzw. des Publikationstitels. Beim Sorted Neighborhood-Blocking mit einem bzw. zwei Durchgängen wurde eine Fenstergröße von $w = 1000$ (1-pass) bzw. $w_1 = w_2 = 500$ (2-pass) verwendet, was in einer nahezu gleichen Anzahl von Paarvergleichen resultiert. Im Gegensatz dazu ist die Anzahl der Paarvergleiche beim Standard Blocking mit zwei Blockschlüsseln pro Datensatz sehr viel größer als bei der Verwendung von einem Blockschlüssel pro Datensatz. Trotzdem unterscheiden sich die benötigten Ausführungszeiten nur geringfügig, da die 20 Knoten insgesamt nur leicht ausgelastet sind.

Die mit den verschiedenen Klassifikationsstrategien erreichte Match-Qualität verhält sich beim Einsatz von Blocking-Techniken ähnlich wie bei der Auswertung des Kartesischen Produktes – erneut konnten mit dem Entscheidungsbaumverfahren die besten Ergebnisse erreicht werden. Die 1-pass Blocking-

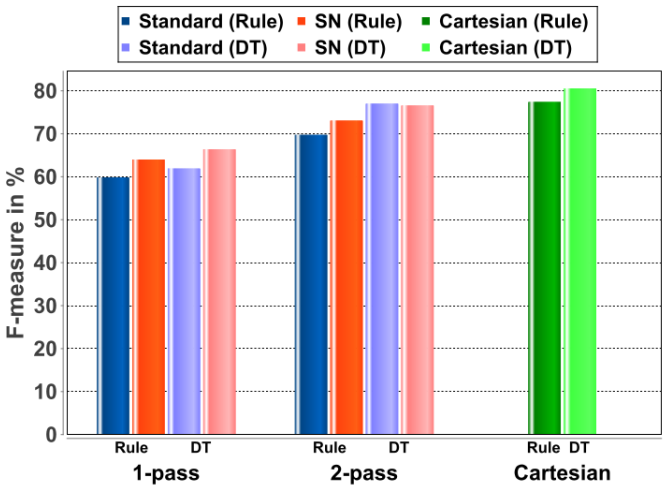


Abbildung 10.3: Vergleich der besten, mit unterschiedlichen Blocking-Verfahren erzielten, F-Measure-Werte bei Verwendung einer regelbasierten Klassifikation ($rule_2$) und des Entscheidungsbaums (DT).

Strategie mit allein vom Autor-Attribut abgeleiteten Blockschlüsseln zeigte sich als zu restriktiv und führte zu einem geringen Recall und einem F-Measure-Wert von 66% und weniger. Mit Hinzunahme des Publikationstitel-Attributs konnte sowohl für das Standard Blocking als auch für das Sorted Neighborhood-Verfahren die Match-Qualität deutlich auf bis zu 77% F-Measure verbessert werden.

Abbildung 10.3 fasst die besten, mit regelbasierter ($rule_2$) bzw. lernbasierter Klassifikation (Entscheidungsbaum) erreichten, F-Measure-Werte zusammen. Es ist zu erkennen, dass der lernbasierte Ansatz dem regelbasierten Ansatz für alle untersuchten Blocking-Konfigurationen überlegen ist. Das Sorted Neighborhood-Verfahren schneidet in drei von vier Fällen etwas besser ab als das Standard Blocking. Allgemein konnten beim Blocking mit mehreren, von unterschiedlichen Attributen abgeleiteten, Blockschlüsseln pro Datensatz deutlich bessere Ergebnisse als beim 1-pass Blocking erreicht werden. Zusätzlich sind die erreichten Ergebnisse mit den Ergebnissen des Kartesischen Produkts vergleichbar, dessen Auswertung jedoch für größere Entity Resolution-Probleme nicht praktikabel ist.

Das Experiment erlaubt sicherlich kein Ziehen genereller Schlüsse über die relative Güte verschiedener Entity Resolution-Ansätze, da nur ein einzelnes Entity Resolution-Problem betrachtet wurde und keine breite systematische Evaluation gesetzter Parameter (z. B. weitere Klassifikationsregeln, variierende Fenstergrößen, andere Blockschlüssel, verschiedene Größen der Trainingsdatenmenge usw.) oder alternativer Techniken stattfand. Trotzdem unterstreicht es den praktischen Nutzen des Dedoop-Frameworks für die umfangreiche Evaluierung mehrerer Entity Resolution-Strategien, da die Konfiguration der Workflows sehr komfortabel über eine graphische Benutzeroberfläche erfolgt und durch deren automatische Parallelisierung eine deutliche Reduzierung der Antwortzeiten erreicht wird.

10.2 Zusammenfassung

Die vorliegende Arbeit untersuchte das Problem der effizienten MapReduce-Parallelisierung laufzeitintensiver Entity Resolution-Workflows.

10.2.1 Teil I – Einführung

Im ersten Teil der Arbeit wurde ein Überlick über die Forschungsarbeiten zu den Themen Entity Resolution und MapReduce gegeben sowie eine Übersicht bestehender Ansätze zur Parallelisierung von Entity Resolution-Workflows präsentiert. Im Anschluss an diese Literaturstudie erfolgte die überblicksartige Vorstellung des MapReduce-basierten Entity Resolution-Frameworks *Dedoop*, welches eine High-Level-Spezifikation komplexer Entity Resolution-Workflows ermöglicht, die automatisch in eine Menge von MapReduce-Jobs transformiert werden und in einem MapReduce-Cluster ausgeführt werden können. Des Weiteren wurden die grundsätzlichen bei der MapReduce-Parallelisierung bestehenden Probleme erläutert, deren Lösung Gegenstand des weiteren Verlaufs der Dissertation ist.

10.2.2 Teil II – Paralleles Blocking und Lastbalancierung

Der zweite Teil der Arbeit beschäftigte sich mit der Fragestellung, wie eine effiziente Umsetzung Blocking-basierte Entity Resolution-Workflows auf Grundlage des MapReduce-Programmiermodells erreicht werden kann. Dazu wurde mit dem Standard Blocking und dem Sorted Neighborhood-Verfahren eine repräsentative Auswahl existierender Blocking-Verfahren betrachtet. Zunächst wurden grundlegende MapReduce-Basis-Implementierungen beider Blocking-Verfahren erläutert, die eine korrekte Umverteilung der verteilt vorliegenden Eingabedatensätze sicherstellen, sodass eine parallele Ähnlichkeitsberechnung aller durch den Blocking-Schritt definierten Kandidatenpaare erfolgen kann. Für das Sorted Neighborhood-Verfahren wurde zudem eine Erweiterung vorgestellt, die eine Kapselung mehrerer Durchgänge des Basis-Verfahrens (Multi-pass Sorted Neighborhood) in einen einzelnen MapReduce-Job ermöglicht.

Des Weiteren wurde gezeigt, dass Techniken zur Lastbalancierung für eine *effiziente* Umsetzung Blocking-basierter MapReduce-Workflows unverzichtbar sind. Um Skalierbarkeit auch für große Entity Resolution-Probleme gewährleisten zu können, ist es notwendig, alle zur Verfügung stehenden Ressourcen gleichmäßig auszulasten, um zu vermeiden, dass bei variierenden Blockgrößen einige wenige Prozessoren einen Großteil der Rechenlast tragen, während die Mehrzahl der Rechenressourcen nur wenig ausgelastet ist. Zu diesem Zweck wurden verschiedene Lastbalancierungsverfahren für beide Blocking-Verfahren vorgestellt, die auf Basis einer leichtgewichtigen Vorab-Datenanalyse die Zuweisung der Kandidatenpaare zu den einzelnen Reduce-Tasks modifizieren und eine gleichmäßige Aufteilung der gesamten Workload gewährleisten. Die umfangreiche Evaluierung zeigte, dass die vorgestellten Ansätze robust gegenüber Datenungleichverteilung sind und mit der Anzahl der Clusterknoten skalieren.

Neben der Betrachtung klassischer Blocking-basierter Entity Resolution-Workflows wurde ebenfalls die verwandte Fragestellung untersucht, wie die Berechnung von Punkten eines Raumes, deren Abstand kleiner als ein vorgegebener Schwellwert ist, mit MapReduce parallelisiert werden kann. Dazu wurde zunächst ein kürzlich veröffentlichter sequentieller Algorithmus mit MapReduce implementiert. Auch hierbei zeigte sich, dass durch den Einsatz ähnlicher Lastbalancierungstechniken eine deutliche Beschleunigung erzielt werden kann.

10.2.3 Teil III – Weitere Teilprobleme

Der dritte Teil der Arbeit widmete sich der Umsetzung und Optimierung weiterer Teilschritte des allgemeinen Entity Resolution-Workflows. Zunächst wurde gezeigt, wie maschinelle Lernverfahren, die die berechneten Ähnlichkeitswerte eines jeden Kandidatenpaares zu einer qualitativ hochwertigen Match-Entscheidung kombinieren, in den von Dedoop unterstützten Entity Resolution-Workflow integriert werden können. Dabei wurde auf die populäre Standardbibliothek WEKA zurückgegriffen, die eine Vielzahl komplexer Klassifikatoren kapselt, die ohne Einschränkung über Dedoops Benutzeroberfläche parametrisiert und zur Klassifikation eingesetzt werden können. Zusätzlich wurde in diesem Zusam-

menhang gezeigt, wie die Ähnlichkeitsberechnung und Klassifikation auch ohne einen Blocking-Schritt parallelisiert werden kann. Die vorgestellten Strategien zur Evaluierung aller Paare des Kartesischen Produktes sind in der Lage, die zu verrichtende Arbeit gleichmäßig auf die zu Verfügung stehenden Cluster-Ressourcen aufzuteilen und skalieren mit der Anzahl der Knoten.

Im Anschluss daran wurde ein Verfahren zur Vermeidung redundanter Paarvergleiche bei der Verwendung mehrerer Blockschlüssel pro Datensatz präsentiert. Dies ist ein lohnenswertes Optimierungsziel, da bei überlappenden Blöcken Datensätze unnötigerweise mehrfach bezüglich unterschiedlicher Blockschlüssel (i. Allg. sogar von verschiedenen Knoten) verglichen werden. Der vorgestellte Basisansatz ermöglicht jedem Reduce-Task ohne jegliche Kommunikation für ein Datensatzpaar zu entscheiden, ob es für den aktuell bearbeiteten Blockschlüssel zu vergleichen ist oder nicht. Die Evaluation für einen systematisch variierten Überlappungsgrad zeigte, dass dieser Ansatz einer naiven Auswertung aller Kandidatenpaare deutlich überlegen ist. In der Folge wurde erläutert, dass es beim Einsatz dieses Verfahrens nicht mehr möglich ist, die Gesamtzahl tatsächlich durchzuführender Paarvergleiche aus den während der Datenanalyse gesammelten Statistiken abzuleiten. Vor diesem Hintergrund wurden Erweiterungen vorgeschlagen und evaluiert, die dem Problem entgegenwirken und mit denen für das betrachtete Problem eine weitere Laufzeitverbesserung von bis zu 10% erzielt werden konnte.

Der letzte Beitrag der Arbeit besteht aus der effizienten iterativen Berechnung der transitiven Hülle eines Match-Ergebnisses. Dazu erfolgte zunächst eine detaillierte Vorstellung eines kürzlich veröffentlichten Algorithmus zur MapReduce-basierten Berechnung zusammenhängender Graphkomponenten. Nach einer Diskussion möglicher Optimierungen wurden verschiedene Weiterentwicklungen vorgestellt, die Teilkomponenten sowohl in der Map- als auch in der Reduce-Phase jeder Iteration verbinden und die wiederholte Verarbeitung stabiler Komponenten vermeiden. Eine Evaluierung dieser Erweiterungen zeigte eine signifikante Verringerung des über alle Iterationen hinweg ausgegebenen Datenvolumens sowie der Anzahl benötigter Iterationen und führte zur einer Beschleunigung um den Faktor $2-4$ gegenüber dem Basisverfahren.

10.3 Ausblick

Die in dieser Arbeit vorgestellten Methoden und Algorithmen wurden im Dedoop-Prototypen zusammengeführt, welcher der Community zum Download zur Verfügung steht[1] und als Ausgangspunkt für weitere Forschungsarbeiten dienen kann. Dieser Abschnitt beschreibt drei mögliche Forschungsrichtungen genauer.

Erweiterung des Dedoop-Frameworks: Trotz des erheblichen Funktionsumfangs des Dedoop-Prototypens ist die Umsetzung und Einbindung weiterer Entity Resolution-Techniken sinnvoll. Die Konzeption und Evaluierung von Strategien zur automatischen Erkennung von Fälschungen und Plagiaten in Online-Shops und auf Auktionsplattformen ist ein interessantes zukünftiges Forschungsthema [202]. Um solche Studien für große Mengen an Webdaten durchführen zu können, erscheint die Integration bestehender MapReduce-Implementierungen iterativer Clustering-Verfahren aus der *Apache Mahout*-Bibliothek[2] in den Dedoop-Prototypen hilfreich. Ein weitere sinnvolle Erweiterung des Dedoop-Frameworks ist die Unterstützung des Nutzers bei der Auswahl guter Trainingsdaten. Dabei ist v. a. die Integration von Active Learning-Techniken vielversprechend, die ein initiales Klassifikationsmodell in einem iterativen Prozess schrittweise mittels Nutzerfeedback verfeinern und den manuellen Aufwand im Vergleich zu traditionellen maschinellen Lernverfahren deutlich verringern. Hierfür könnte auf die vorhandene graphische Benutzerschnittstelle zurückgegriffen und diese entsprechend erweitert werden.

[1] `http://dbs.uni-leipzig.de/howto_dedoop`
[2] `http://hortonworks.com/hadoop/mahout/`

Privacy-preserving Entity Resolution: Traditionell wird bei der Bearbeitung von Entity Resolution-Problemen angenommen, dass alle Daten unverschlüsselt und nicht anonymisiert vorliegen. Wenn personenbezogene Daten (z. B. Patientendaten) aus verschiedenen Quellsystemen einer dritten Partei (z. B. Universitäten und anderen Forschungseinrichtungen) für die systematische Analyse zur Verfügung gestellt werden, ist u. a. aus gesetzlichen Datenschutzrichtlinien eine vorherige Anonymisierung und Verschlüsselung dieser Personendaten erforderlich. Privacy-preserving Entity Resolution bezeichnet die Duplikaterkennung in einer Kollektion von Datensätzen, deren Attributwerte so verschlüsselt (z. B. Tokenisierung, Mehrfach-Hashing und Abbildung auf Bitvektoren) wurden, dass eine Wiederherstellung der Originaldaten unmöglich ist. Zur Duplikaterkennung kann somit nicht auf traditionelle Entity Resolution-Techniken, wie Field Matching, zurückgegriffen werden. Stattdessen ist die Quantifizierung der Ähnlichkeit von Datensätzen allein auf Basis der verschlüsselten Daten (z. B. Bitvektoren) erforderlich. Hierfür existieren bereits erste Ansätze, die aufgrund der quadratischen Komplexität der paarweisen Ähnlichkeitsberechnung noch nicht für große Entity Resolution-Probleme anwendbar sind. Ein Überblick über die bisherigen Arbeiten findet sich in [51, 232]. In ersten Vorarbeiten wurde bereits begonnen, Dedoop um neue Techniken zu erweitern, die eine automatische, skalierbare und qualitativ-hochwertige Duplikaterkennung in verschlüsselten Datenquellen ermöglichen sollen.

Einsatz von GPUs: Zusätzlich zur Parallelisierung mit MapReduce bietet sich die Nutzung von Graphical Processing Units (GPUs) auf den einzelnen Clusterknoten an, um die rechenintensiven Ähnlichkeitsberechnungen zu beschleunigen. Frei verfügbare Frameworks, wie *OpenCL* oder *CUDA*, ermöglichen den Einsatz von massiv-parallelen Graphikprozessoren für Berechnungen in verschiedenen Anwendungsgebieten. Cloud-Dienstleister, wie Amazon EC2, haben auf den steigenden Bedarf reagiert und bieten die Nutzung virtueller Maschinen mit exklusivem Zugriff auf leistungsfähige GPUs an. Um neben der gleichmäßigen Nutzung der Clusterknoten auch eine optimale Ausnutzung der Ressourcen eines jeden Knotens zu gewährleisten, ist die die Kombination einer MapReduce-basierten Parallelisierung von Entity Resolution-Workflows mit der Beschleunigung von Ähnlichkeitsberechnungen durch GPUs sinnvoll. Dabei würde MapReduce die Rolle einer Koordinations- und Persistenzschicht einnehmen. Beim Einsatz von GPUs kann (u. a. auf eigene) Vorarbeiten zurückgegriffen werden [83, 101]. Nach ersten Experimenten ist insbesondere im Bereich der Privacy-preserving Entity Resolution der Einsatz von GPUs vielversprechend, da sich die uniforme Repräsentation von Datensätzen (z. B. durch Bitvektoren gleicher Länge) gut auf die von GPUs unterstützen Datenstrukturen abbilden lässt und die Ähnlichkeitsberechnung auf einfache Bitoperationen zurückgeführt werden kann.

Kombination verschiedener paralleler Programmierparadigma: Der kombinierte Einsatz verschiedener Programmierparadigma und Systeme zur Bearbeitung ist als sehr vielversprechend anzusehen. So eignet sich die Verwendung des MapReduce-Programmiermodells sehr gut für Batch-artige Aufgaben, wohingegen Systeme, die auf dem Bulk Synchronous Parallel-Paradigma basieren, für iterative Algorithmen (z. B. Clustering, Transitive Hülle) zu bevorzugen sind. Auch hierfür existieren bereits erste Ansätze [160]. Mit der Öffnung des Hadoop-Frameworks für weitere Programmiermodelle im Rahmen der *YARN*-Abstraktionsschicht scheinen mittlerweile die notwendigen technischen Voraussetzungen gegeben, um verschiedene Teilprobleme des Entity Resolution-Prozesses durch unterschiedliche Programmiermodelle innerhalb eines Clusters bearbeiten zu können. Gleichzeitig bieten alternative Systeme zur parallelen Datenverarbeitung HDFS-Schnittstellen für die Weiterverarbeitung von MapReduce-Ausgabedaten an.

Mit dieser Arbeit wurden wichtige Beiträge im Bereich der Parallelisierung von Entity Resolution-Workflows geleistet. Die Diskussion möglicher, zukünftiger Forschungsarbeiten zeigt, dass in den untersuchten Bereichen noch Potential zur Erweiterung und Verbesserung besteht. Dass unter Mitwirkung großer Unternehmen mit *Data Tamer* [223] und *WOO* [21] bereits Plattformen entstanden sind, die eine (MapReduce-) Parallelisierung des vollständigen Datenintegrationsworkflows verfolgen, unterstreicht die Relevanz der in dieser Arbeit untersuchten Themen.

Teil V

Anhang

Literaturverzeichnis

[1] ABOUZEID, A., BAJDA-PAWLIKOWSKI, K., ABADI, D. J., RASIN, A., SILBERSCHATZ, A. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proceedings of the VLDB Endowment 2*, 1 (2009).

[2] ABOUZIED, A., BAJDA-PAWLIKOWSKI, K., HUANG, J., ABADI, D. J., SILBERSCHATZ, A. HadoopDB in Action: Building Real World Applications. In *Proc. of the International Conference on Management of Data* (2010).

[3] ADLY, N. Efficient Record Linkage using a Double Embedding Scheme. In *Proc. of the International Conference on Data Mining* (2009).

[4] AFRATI, F. N., BORKAR, V. R., CAREY, M. J., POLYZOTIS, N., ULLMAN, J. D. Map-Reduce Extensions and Recursive Queries. In *Proc. of 14th International Conference on Extending Database Technology* (2011).

[5] AIZAWA, A. N., OYAMA, K. A Fast Linkage Detection Scheme for Multi-Source Information Integration. In *Proc. of the International Workshop on Challenges in Web Information Retrieval and Integration* (2005).

[6] ALEXANDROV, A., BATTRÉ, D., EWEN, S., HEIMEL, M., HUESKE, F., KAO, O., MARKL, V., NIJKAMP, E., WARNEKE, D. Massively Parallel Data Analysis with PACTs on Nephele. *Proceedings of the VLDB Endowment 3*, 2 (2010).

[7] ALEXANDROV, A., BERGMANN, R., EWEN, S., FREYTAG, J.-C., HUESKE, F., HEISE, A., KAO, O., LEICH, M., LESER, U., MARKL, V., NAUMANN, F., PETERS, M., RHEINLÄNDER, A., SAX, M. J., SCHELTER, S., HÖGER, M., TZOUMAS, K., WARNEKE, D. The stratosphere platform for big data analytics. *VLDB Journal* (2014, zur Veröffentlichung angenommen).

[8] ALEXANDROV, A., EWEN, S., HEIMEL, M., HUESKE, F., KAO, O., MARKL, V., NIJKAMP, E., WARNEKE, D. MapReduce and PACT - Comparing Data Parallel Programming Models. In *Proc. of the 14th Conference on Database Systems for Business, Technology, and Web* (2011).

[9] ANANTHAKRISHNA, R., CHAUDHURI, S., GANTI, V. Eliminating Fuzzy Duplicates in Data Warehouses. In *Proc. of the 28th International Conference on Very Large Data Bases* (2002).

[10] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A. G., STOICA, I., LU, Y., SAHA, B., HARRIS, E. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. of the 9th Symposium on Operating Systems Design and Implementation* (2010).

[11] ARASANAL, R. M., RUMANI, D. U. Improving MapReduce Performance through Complexity and Performance Based Data Placement in Heterogeneous Hadoop Clusters. In *Proc. of the 9th International Conference on Distributed Computing and Internet Technology* (2013).

[12] ARASU, A., GANTI, V., KAUSHIK, R. Efficient Exact Set-Similarity Joins. In *Proc. of the 32nd International Conference on Very Large Data Bases* (2006).

[13] ARASU, A., GÖTZ, M., KAUSHIK, R. On Active Learning of Record Matching Packages. In *Proc. the International Conference on Management of Data* (2010).

[14] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R. H., KONWINSKI, A., LEE, G., PATTERSON, D. A., RABKIN, A., STOICA, I., ZAHARIA, M. A view of cloud computing. *Communications of the ACM 53*, 4 (2010).

[15] AWERBUCH, B., SHILOACH, Y. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. *IEEE Transactions on Computers 36*, 10 (1987).

[16] BANCILHON, F., MAIER, D., SAGIV, Y., ULLMAN, J. D. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. of Symposium on Principles of Database Systems* (1986).

[17] BARROSO, L. A., HÖLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[18] BATINI, C., SCANNAPIECO, M. *Data Quality: Concepts, Methodologies and Techniques*. Data-Centric Systems and Applications. Springer, 2006.

[19] BATTRÉ, D., EWEN, S., HUESKE, F., KAO, O., MARKL, V., WARNEKE, D. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *Proc. of the 1st ACM Symposium on Cloud Computing* (2010).

[20] BAXTER, R., CHRISTEN, P., CHURCHES, T. A Comparison of Fast Blocking Methods for Record Linkage. In *Proc. of the International Workshop on Data Cleaning, Record Linkage and Object Consolidation* (2003).

[21] BELLARE, K., CURINO, C., MACHANAVAJIHALA, A., MIKA, P., RAHURKAR, M., SANE, A. WOO: A Scalable and Multi-tenant Platform for Continuous Knowledge Base Synthesis. *Proceedings of the VLDB Endowment 6*, 11 (2013).

[22] BENJELLOUN, O., GARCIA-MOLINA, H., GONG, H., KAWAI, H., LARSON, T. E., MENESTRINA, D., THAVISOMBOON, S. D-Swoosh: A Family of Algorithms for Generic, Distributed Entity Resolution. In *Proc. of the 27th International Conference on Distributed Computing Systems* (2007).

[23] BENJELLOUN, O., GARCIA-MOLINA, H., MENESTRINA, D., SU, Q., WHANG, S. E., WIDOM, J. Swoosh: A Generic Approach to Entity Resolution. *VLDB Journal 18*, 1 (2009).

[24] BHATTACHARYA, I., GETOOR, L. Collective Entity Resolution In Relational Data. *IEEE Data Engineering Bulletin 29*, 2 (2006).

[25] BIANCO, G. D., DE MATOS GALANTE, R., HEUSER, C. A. A fast approach for parallel deduplication on multicore processors. In *Proc. of the Symposium on Applied Computing* (2011).

[26] BIANCO, G. D., GALANTE, R., HEUSER, C. A., GONÇALVES, M. A. Tuning large scale deduplication with reduced effort. In *Proc. of the Conference on Scientific and Statistical Database Management* (2013).

[27] BILENKO, M., KAMATH, B., MOONEY, R. J. Adaptive blocking: Learning to scale up record linkage. In *Proc. of the 6th International Conference on Data Mining* (2006).

[28] BILENKO, M., MOONEY, R. J. Adaptive Duplicate Detection Using Learnable String Similarity Measures. In *Proc. of the 9th International Conference on Knowledge Discovery and Data Mining* (2003).

[29] BILENKO, M., MOONEY, R. J. On Evaluation and Training-Set Construction for Duplicate Detection. In *Proc. of the International Workshop on Data Cleaning, Record Linkage and Object Consolidation* (2003).

[30] BILENKO, M., MOONEY, R. J., COHEN, W. W., RAVIKUMAR, P., FIENBERG, S. E. Adaptive Name Matching in Information Integration. *IEEE Intelligent Systems 18*, 5 (2003).

[31] BILGIC, M., LICAMELE, L., GETOOR, L., SHNEIDERMAN, B. D-Dupe: An Interactive Tool for Entity Resolution in Social Networks. In *Proc. of the 13th International Symposium on Graph Drawing* (2005).

[32] BLANAS, S., PATEL, J. M., ERCEGOVAC, V., RAO, J., SHEKITA, E. J., TIAN, Y. A Comparison of Join Algorithms for Log Processing in MapReduce. In *Proc. of the International Conference on Management of Data* (2010).

[33] BLEIHOLDER, J., NAUMANN, F. Data Fusion. *ACM Computation Surveys 41*, 1 (2008).

[34] BÖHM, C., DE MELO, G., NAUMANN, F., WEIKUM, G. LINDA: Distributed Web-of-Data-Scale Entity Matching. In *CIKM* (Proc. of the 21st International Conference on Information and Knowledge Management).

[35] BU, Y., HOWE, B., BALAZINSKA, M., ERNST, M. D. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proceedings of the VLDB Endowment 3*, 1 (2010).

[36] BU, Y., HOWE, B., BALAZINSKA, M., ERNST, M. D. The HaLoop Approach to Large-Scale Iterative Data Analysis. *VLDB Journal 21*, 2 (2012).

[37] BUS, L., TVRDÍK, P. A Parallel Algorithm for Connected Components on Distributed Memory Machines. In *Proc. of European PVM/MPI Users' Group Meeting* (2001).

[38] CAFARELLA, M. J., RÉ, C. Manimal: Relational Optimization for Data-Intensive Programs. In *Proc. of the 13th International Workshop on the Web and Databases* (2010).

[39] CARVALHO, J. C. P., DA SILVA, A. S. Finding Similar Identities among Objects from Multiple Web Sources. In *Proc. of the 5th International Workshop on Web Information and Data Management* (2003).

[40] CHAIKEN, R., JENKINS, B., LARSON, P.-Å., RAMSEY, B., SHAKIB, D., WEAVER, S., ZHOU, J. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proceedings of the VLDB Endowment 1*, 2 (2008).

[41] CHAUDHURI, S., CHEN, B.-C., GANTI, V., KAUSHIK, R. Example-driven Design of Efficient Record Matching Queries. In *Proc. of the 33rd International Conference on Very Large Data Bases* (2007).

[42] CHAUDHURI, S., GANJAM, K., GANTI, V., KAPOOR, R., NARASAYYA, V. R., VASSILAKIS, T. Data Cleaning in Microsoft SQL Server 2005. In *Proc. of the International Conference on Management of Data* (2005).

[43] CHAUDHURI, S., GANTI, V., MOTWANI, R. Robust Identification of Fuzzy Duplicates. In *Proc. of the 21st International Conference on Data Engineering* (2005).

[44] CHEINEY, J.-P., DE MAINDREVILLE, C. A Parallel Transitive Closure Algorithm Using Hash-Based Clustering. In *Proc. of International Workshop on Database Machines* (1989).

[45] CHEN, S. Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce. *Proceedings of the VLDB Endowment 3*, 2 (2010).

[46] CHEN, Z., KALASHNIKOV, D. V., MEHROTRA, S. Exploiting Relationships for Object Consolidation. In *Proc. of the International Workshop on Information Quality in Information Systems* (2005).

[47] CHEN, Z., KALASHNIKOV, D. V., MEHROTRA, S. Exploiting Context Analysis for Combining Multiple Entity Resolution Systems. In *Proc. of the International Conference on Management of Data* (2009).

[48] CHERKASSKY, V. The Nature Of Statistical Learning Theory. *IEEE Transactions on Neural Networks 8*, 6 (1997).

[49] CHIH YANG, H., DASDAN, A., HSIAO, R.-L., JR., D. S. P. Map-Reduce-Merge: Simplified Relationalon Large Clusters Data Processing. In *Proc. of the International Conference on Management of Data* (2007).

[50] CHRISTEN, P. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE Transactions on Knowledge and Data Engineering 24*, 9 (2012).

[51] CHRISTEN, P. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Data-centric systems and applications. Springer, 2012.

[52] CHRISTEN, P., CHURCHES, T., HEGLAND, M. Febrl - A Parallel Open Source Data Linkage System. In *Proc. of the 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining* (2004).

[53] COCHINWALA, M., KURIEN, V., LALK, G., SHASHA, D. Efficient data reconciliation. *Information Sciences 137*, 1-4 (2001).

[54] COHEN, J. Graph Twiddling in a MapReduce World. *Computing in Science and Engineering 11*, 4 (2009).

[55] COHEN, W. W. Data Integration Using Similarity Joins and a Word-Based Information Representation Language. *ACM Transactions on Information Systems 18*, 3 (2000).

[56] COHEN, W. W., RAVIKUMAR, P. D., FIENBERG, S. E. A Comparison of String Distance Metrics for Name-Matching Tasks. In *Proc. of the IJCAI Workshop on Information Integration on the Web* (2003).

[57] COHEN, W. W., RICHMAN, J. Learning to Match and Cluster Large High-Dimensional Data Sets For Data Integration. In *Proc. of the 8th International Conference on Knowledge Discovery and Data Mining* (2002).

[58] DE FREITAS, J., PAPPA, G. L., DA SILVA, A. S., GONÇALVES, M. A., DE MOURA, E. S., VELOSO, A., LAENDER, A. H. F., DE CARVALHO, M. G. Active Learning Genetic Programming for Record Deduplication. In *Proc. of the IEEE Congress on Evolutionary Computation* (2010).

[59] DEAN, J., GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation* (2004).

[60] DEVLIN, B., GRAY, J., LAING, B., SPIX, G. Scalability terminology: Farms, clones, partitions, and packs: RACS and RAPS. Tech. rep., Microsoft Research, 1999.

[61] DEWITT, D. J., NAUGHTON, J. F., SCHNEIDER, D. A., SESHADRI, S. Practical Skew Handling in Parallel Joins. In *Proc. of the 18th International Conference on Very Large Data Bases* (1992).

[62] DITTRICH, J., QUIANÉ-RUIZ, J.-A., JINDAL, A., KARGIN, Y., SETTY, V., SCHAD, J. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *Proceedings of the VLDB Endowment 3*, 1 (2010).

[63] DITTRICH, J., QUIANÉ-RUIZ, J.-A., RICHTER, S., SCHUH, S., JINDAL, A., SCHAD, J. Only Aggressive Elephants are Fast Elephants. *Proceedings of the VLDB Endowment 5*, 11 (2012).

[64] DITTRICH, J., RICHTER, S., SCHUH, S., QUIANÉ-RUIZ, J.-A. Efficient OR Hadoop: Why not both? *IEEE Data Engineering Bulletin 36*, 1 (2013).

[65] DOAN, A., HALEVY, A. Y., IVES, Z. G. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[66] DOAN, A., LU, Y., LEE, Y., HAN, J. Object Matching for Information Integration: A Profiler-Based Approach. In *Proc. of the Workshop on Information Integration on the Web* (2003).

[67] DOMENIG, R., DITTRICH, K. R. An Overview and Classification of Mediated Query Systems. *SIGMOD Record 28*, 3 (1999).

[68] DONG, X., HALEVY, A. Y., MADHAVAN, J. Reference Reconciliation in Complex Information Spaces. In *Proc. of the International Conference on Management of Data* (2005).

[69] DONG, X. L., NAUMANN, F. Data Fusion - Resolving Data Conflicts for Integration. *Proceedings of the VLDB Endowment 2*, 2 (2009).

[70] DRAISBACH, U., NAUMANN, F. A Comparison and Generalization of Blocking and Windowing Algorithms for Duplicate Detection. In *Proc. of the 7th International Workshop on Quality in Databases* (2009).

[71] DRAISBACH, U., NAUMANN, F. DuDe: The Duplicate Detection Toolkit. In *Proc. of the 8th International Workshop on Quality in Databases* (2010).

[72] DRAISBACH, U., NAUMANN, F. A Generalization of Blocking and Windowing Algorithms for Duplicate Detection. In *Proc. of the International Conference on Data and Knowledge Engineering* (2011).

[73] DRAISBACH, U., NAUMANN, F., SZOTT, S., WONNEBERG, O. Adaptive Windows for Duplicate Detection. In *Proc. of the 28th International Conference on Data Engineering* (2012).

[74] EKANAYAKE, J., GUNARATHNE, T., QIU, J. Cloud Technologies for Bioinformatics Applications. *IEEE Transactions on Parallel and Distributed Systems 22*, 6 (2011).

[75] EKANAYAKE, J., LI, H., ZHANG, B., GUNARATHNE, T., BAE, S.-H., QIU, J., FOX, G. Twister: A Runtime

for Iterative MapReduce. In *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing* (2010).

[76] ELFEKY, M. G., ELMAGARMID, A. K., VERYKIOS, V. S. TAILOR: A Record Linkage Tool Box. In *Proc. of the 18th International Conference on Data Engineering* (2002).

[77] ELMAGARMID, A. K., IPEIROTIS, P. G., VERYKIOS, V. S. Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering 19*, 1 (2007).

[78] ELSAYED, T., LIN, J. J., OARD, D. W. Pairwise Document Similarity in Large Collections with MapReduce. In *Proc. of the 46th Annual Meeting of the Association for Computational Linguistics* (2008).

[79] FALOUTSOS, C., LIN, K.-I. FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets. In *Proc. of the International Conference on Management of Data* (1995).

[80] FELLER, E., RAMAKRISHNAN, L., MORIN, C. On the performance and energy efficiency of hadoop deployment models. In *Proc. of International Conference on Big Data* (2013).

[81] FELLIGI, I. P., SUNTER, A. B. A Theory of Record Linkage. *Journal of the American Statistical Association 64* (1969).

[82] FLORATOU, A., PATEL, J. M., SHEKITA, E. J., TATA, S. Column-Oriented Storage Techniques for MapReduce. *Proceedings of the VLDB Endowment 4*, 7 (2011).

[83] FORCHHAMMER, B., PAPENBROCK, T., STENING, T., VIEHMEIER, S., DRAISBACH, U., NAUMANN, F. Duplicate Detection on GPUs. In *Proc. of the 15th Conference on Database Systems for Business, Technology, and Web* (2013).

[84] FOX, D. PRISM und TEMPORA. *Datenschutz und Datensicherheit 37*, 9 (2013).

[85] FRIEDMAN, E., PAWLOWSKI, P. M., CIESLEWICZ, J. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB Endowment 2*, 2 (2009).

[86] GALE, D., SHAPLEY, L. S. College admissions and the stability of marriage. *The American Mathematical Monthly 69*, 1 (1962).

[87] GALHARDAS, H., FLORESCU, D., SHASHA, D., SIMON, E. AJAX: An Extensible Data Cleaning Tool. In *Proc. of the International Conference on Management of Data* (2000).

[88] GALHARDAS, H., FLORESCU, D., SHASHA, D., SIMON, E., SAITA, C.-A. Declarative Data Cleaning: Language, Model, and Algorithms. In *Proc. of the 27th International Conference on Very Large Data Bases* (2001).

[89] GANTZ, J., REINSEL, D. Extracting value from Chaos. *IDC iView* (2011).

[90] GATES, A., NATKOVICH, O., CHOPRA, S., KAMATH, P., NARAYANAM, S., OLSTON, C., REED, B., SRINIVASAN, S., SRIVASTAVA, U. Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience. *Proceedings of the VLDB Endowment 2*, 2 (2009).

[91] GHEMAWAT, S., GOBIOFF, H., LEUNG, S.-T. The Google File System. In *Proc. of the 19th ACM Symposium on Operating Systems Principles* (2003).

[92] GREINER, J. A Comparison of Parallel Algorithms for Connected Components. In *Proc. of Symposium on Parallelism in Algorithms and Architectures* (1994).

[93] GROSS, A., HARTUNG, M., KIRSTEN, T., RAHM, E. On Matching Large Life Science Ontologies in Parallel. In *Proc. of the 7th International Conference on Data Integration in the Life Sciences* (2010).

[94] GU, L., BAXTER, R., VICKERS, D., RAINSFORD, C. Record Linkage: Current Practice and Future

Directions. Tech. rep., CSIRO Mathematical and Information Sciences, 2003.

[95] GUFLER, B., AUGSTEN, N., REISER, A., KEMPER, A. Handling Data Skew in MapReduce. In *Proc. of the 1st International Conference on Cloud Computing and Services Science* (2011).

[96] GUFLER, B., AUGSTEN, N., REISER, A., KEMPER, A. Load Balancing in MapReduce Based on Scalable Cardinality Estimates. In *Proc. of the 28th International Conference on Data Engineering* (2012).

[97] HADJIELEFTHERIOU, M., CHANDEL, A., KOUDAS, N., SRIVASTAVA, D. Fast Indexes and Algorithms for Set Similarity Selection Queries. In *Proc. of the 24th International Conference on Data Engineering* (2008).

[98] HALEVY, A. Y., RAJARAMAN, A., ORDILLE, J. J. Data Integration: The Teenage Years. In *Proc. of the 32nd International Conference on Very Large Data Bases* (2006).

[99] HALL, M., FRANK, E., HOLMES, G., PFAHRINGER, B., REUTEMANN, P., WITTEN, I. H. The WEKA Data Mining Software: An Update. *SIGKDD Explorations 11*, 1 (2009).

[100] HAN, J., KAMBER, M. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[101] HARTUNG, M., KOLB, L., GROSS, A., RAHM, E. Optimizing Similarity Computations for Ontology Matching - Experiences from GOMMA. In *Proc. of the 9th International Conference on Data Integration in the Life Sciences* (2013).

[102] HASSANZADEH, O., MILLER, R. J. Creating probabilistic databases from duplicated data. *VLDB Journal 18*, 5 (2009).

[103] HE, Y., LEE, R., HUAI, Y., SHAO, Z., JAIN, N., ZHANG, X., XU, Z. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *Proc. of 27th International Conference on Data Engineering* (2011).

[104] HERNÁNDEZ, M. A., STOLFO, S. J. The merge/purge problem for large databases. In *Proc. of the International Conference on Management of Data* (1995).

[105] HERNÁNDEZ, M. A., STOLFO, S. J. Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem. *Data Mining and Knowledge Discovery 2*, 1 (1998).

[106] HERODOTOU, H., BABU, S. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *Proceedings of the VLDB Endowment 4*, 11 (2011).

[107] HERODOTOU, H., DONG, F., BABU, S. MapReduce Programming and Cost-based Optimization? Crossing this Chasm with Starfish. *Proceedings of the VLDB Endowment 4*, 12 (2011).

[108] HERODOTOU, H., DONG, F., BABU, S. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *Proc. of the 2nd ACM Symposium on Cloud Computing* (2011).

[109] HERODOTOU, H., LIM, H., LUO, G., BORISOV, N., DONG, L., CETIN, F. B., BABU, S. Starfish: A Self-tuning System for Big Data Analytics. In *Proc. of the 5th Biennial Conference on Innovative Data Systems Research* (2011).

[110] HERSCHEL, M., NAUMANN, F., SZOTT, S., TAUBERT, M. Scalable Iterative Graph Duplicate Detection. *IEEE Transactions on Knowledge and Data Engineering 24*, 11 (2012).

[111] HERZOG, T. N., SCHEUREN, F. J., WINKLER, W. E. *Data Quality and Record Linkage Techniques*. Springer, 2007.

[112] HILLNER, S., NGOMO, A.-C. N. Parallelizing LIMES for Large-Scale Link Discovery. In *Proc. of the 7th International Conference on Semantic Systems* (2011).

[113] HIRSCHBERG, D. S., CHANDRA, A. K., SARWATE, D. V. Computing Connected Components on Parallel Computers. *Communications of the ACM 22*, 8 (1979).

[114] HJALTASON, G. R., SAMET, H. Properties of Embedding Methods for Similarity Searching in Metric Spaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence 25*, 5 (2003).

[115] IBRAHIM, S., JIN, H., LU, L., WU, S., HE, B., QI, L. LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud. In *Proc. of 2nd International Conderence on Cloud Computing Technology and Science* (2010).

[116] IOANNIDIS, Y. E. On the Computation of the Transitive Closure of Relational Operators. In *Proc. of 12th International Conference on Very Large Databases* (1986).

[117] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., FETTERLY, D. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proc. of the European Conference on Computer Systems* (2007).

[118] ISARD, M., YU, Y. Distributed Data-Parallel Computing Using a High-Level Programming Language. In *Proc. of the International Conference on Management of Data* (2009).

[119] ISELE, R., JENTZSCH, A., BIZER, C. Efficient Multidimensional Blocking for Link Discovery without losing Recall. In *Proc. of the 14th International Workshop on the Web and Databases* (2011).

[120] JAHANI, E., CAFARELLA, M. J., RÉ, C. Automatic Optimization for MapReduce Programs. *Proceedings of the VLDB Endowment 4*, 6 (2011).

[121] JIN, L., LI, C., MEHROTRA, S. Efficient Record Linkage in Large Data Sets. In *Proc. of the 8th International Conference on Database Systems for Advanced Applications* (2003).

[122] JURCZYK, P, LU, J. J., XIONG, L., CRAGAN, J. D., CORREA, A. FRIL: A Tool for Comparative Record Linkage. In *AMIA Annual Symposium Proceedings* (2008).

[123] KALASHNIKOV, D. V., MEHROTRA, S. Domain-Independent Data Cleaning via Analysis of Entity-Relationship Graph. *ACM Transactions on Database Systems 31*, 2 (2006).

[124] KANG, U., TSOURAKAKIS, C. E., FALOUTSOS, C. PEGASUS: A Peta-Scale Graph Mining System. In *Proc. of Intl. Conference on Data Mining* (2009).

[125] KIRSTEN, T., GROSS, A., HARTUNG, M., RAHM, E. GOMMA: a component-based infrastructure for managing and analyzing life science ontologies and their evolution. *Journal of Biomedical Semantics 2* (2011).

[126] KIRSTEN, T., KOLB, L., HARTUNG, M., GROSS, A., KÖPCKE, H., RAHM, E. Data Partitioning for Parallel Entity Matching. In *Proc. of the 8th International Workshop on Quality in Databases* (2010).

[127] KOLB, L., KÖPCKE, H., THOR, A., RAHM, E. Learning-based Entity Resolution with MapReduce. In *Proc. of the 3rd International Workshop on Cloud Data Management* (2011).

[128] KOLB, L., RAHM, E. Parallel Entity Resolution with Dedoop. *Datenbank-Spektrum 13*, 1 (2013).

[129] KOLB, L., SEHILI, Z., RAHM, E. Iterative Computation of Connected Graph Components with MapReduce. *Datenbank-Spektrum 14*, 2 (2014).

[130] KOLB, L., THOR, A., RAHM, E. Block-based Load Balancing for Entity Resolution with MapReduce. In *Proc. of the 20th International Conference on Information and Knowledge Management* (2011).

[131] KOLB, L., THOR, A., RAHM, E. Parallel Sorted Neighborhood Blocking with MapReduce. In *Proc. of the 14th Conference on Database Systems for Business, Technology, and Web* (2011).

[132] KOLB, L., THOR, A., RAHM, E. Dedoop: Efficient Deduplication with Hadoop. *Proceedings of the VLDB Endowment 5*, 12 (2012).

[133] KOLB, L., THOR, A., RAHM, E. Load Balancing for MapReduce-based Entity Resolution. In *Proc. of the 28th International Conference on Data Engineering* (2012).

[134] KOLB, L., THOR, A., RAHM, E. Multi-pass Sorted Neighborhood Blocking with MapReduce. *Computer Science - Research and Development 27* (2012).

[135] KOLB, L., THOR, A., RAHM, E. Don't Match Twice: Redundancy-free Similarity Computation with MapReduce. In *Proc. of the 2nd International Workshop on Data Analytics in the Cloud* (2013).

[136] KOLCZ, A., CHOWDHURY, A., ALSPECTOR, J. Improved RobustnessDetection viaof Signature-Based Near-Replica Lexicon Randomization. In *Proc. of the 10th International Conference on Knowledge Discovery and Data Mining* (2004).

[137] KÖPCKE, H., RAHM, E. Training Selection for Tuning Entity Matching. In *Proc. of the International Workshop on Quality in Databases and Management of Uncertain Data* (2008).

[138] KÖPCKE, H., RAHM, E. Frameworks for entity matching: A comparison. *Data & Knowledge Engineering 69*, 2 (2010).

[139] KÖPCKE, H., THOR, A., RAHM, E. Comparative evaluation of entity resolution approaches with FEVER. *Proceedings of the VLDB Endowment 2*, 2 (2009).

[140] KÖPCKE, H., THOR, A., RAHM, E. Evaluation of entity resolution approaches on real-world match problems. *Proceedings of the VLDB Endowment 3*, 1 (2010).

[141] KÖPCKE, H., THOR, A., RAHM, E. Evaluation of Learning-Based Approaches for Matching Web Data Entities. *IEEE Internet Computing 14*, 4 (2010).

[142] KWON, Y., BALAZINSKA, M., HOWE, B., ROLIA, J. A. Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In *Proc. of the 1st ACM Symposium on Cloud Computing* (2010).

[143] KWON, Y., BALAZINSKA, M., HOWE, B., ROLIA, J. A. SkewTune: Mitigating Skew in MapReduce Applications. In *Proc. of the International Conference on Management of Data* (2012).

[144] LANEY, D. 3D Data Management: Controlling Data Volume Velocity, and Variety. *Application Delivery Strategies* (2001).

[145] LANG, W., PATEL, J. M. Energy Management for MapReduce Clusters. *Proceedings of the VLDB Endowment 3*, 1 (2010).

[146] LANGE, D., NAUMANN, F. Frequency-aware Similarity Measures: Why Arnold Schwarzenegger is Always a Duplicate. In *Proc. of the 20th International Conference on Information and Knowledge Management* (2011).

[147] LATTANZI, S., MOSELEY, B., SURI, S., VASSILVITSKII, S. Filtering: A Method for Solving Graph Problems in MapReduce. In *Proc. of Symposium on Parallelism in Algorithms and Architectures* (2011).

[148] LEE, K.-H., LEE, Y.-J., CHOI, H., CHUNG, Y. D., MOON, B. Parallel Data Processing with MapReduce: A Survey. *SIGMOD Record 40*, 4 (2011).

[149] LEE, M.-L., HSU, W., KOTHARI, V. Cleaning the Spurious Links in Data. *IEEE Intelligent Systems 19*, 2 (2004).

[150] LEITÃO, L., CALADO, P., WEIS, M. Structure-Based Inference of XML Similarity for Fuzzy Duplicate Detection. In *Proc. of the 16th International Conference on Information and Knowledge Management* (2007).

[151] LESER, U., NAUMANN, F. *Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen*. dpunkt, 2007.

[152] LEVENSHTEIN, V. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission 1*, 1 (1965).

[153] LEVERICH, J., KOZYRAKIS, C. On the Energy (In)efficiency of Hadoop Clusters. *Operating Systems*

Review 44, 1 (2010).

[154] LIN, J. The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce. In *Proc. of the 7th International Workshop on Large-Scale Distributed Systems for Information Retrieval* (2009).

[155] LIN, J., DYER, C. *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010.

[156] LIN, J. J. Brute Force and Indexed Approaches to Pairwise Document Similarity Comparisons with MapReduce. In *Proc. of the 32nd International Conference on Research and Development in Information Retrieval* (2009).

[157] LIN, Y., AGRAWAL, D., CHEN, C., OOI, B. C., WU, S. Llama: Leveraging Columnar Storage for Scalable Join Processing in the MapReduce Framework. In *Proc. of the International Conference on Management of Data* (2011).

[158] LOW, W. L., LEE, M.-L., LING, T. W. A Knowledge-based Approach for Duplicate Elimination in Data Cleaning. *Information Systems 26*, 8 (2001).

[159] MALEWICZ, G., AUSTERN, M. H., BIK, A. J. C., DEHNERT, J. C., HORN, I., LEISER, N., CZAJKOWSKI, G. Pregel: A System for Large-Scale Graph Processing. In *Proc. of the International Conference on Management of Data* (2010).

[160] MALHOTRA, P., AGARWAL, P., SHROFF, G. Graph-Parallel Entity Resolution using LSH & IMM. In *Proc. of the 1st International Workshop on Algorithms for MapReduce and Beyond* (2014).

[161] MÄRTENS, H. A Classification of Skew Effects in Parallel Database Systems. In *Proc. of the 7th International Euro-Par Conference* (2001).

[162] MCCALLUM, A., NIGAM, K., UNGAR, L. H. Efficient Clustering of high-dimensional Data Sets with Application to Reference Matching. In *Proc. of the 6th International Conference on Knowledge Discovery and Data Mining* (2000).

[163] MCNEILL, N., KARDES, H., BORTHWICK, A. Dynamic Record Blocking: Efficient Linking of Massive Databases in MapReduce. In *Proc. of the 10th International Workshop on Quality in Databases* (2012).

[164] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., VASSILAKIS, T. Dremel: Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment 3*, 1 (2010).

[165] MENDES, M. E. S., SACKS, L. Evaluating Fuzzy Clustering for Relevance-based Information Access. In *Proc. of IEEE International Conference on Fuzzy Systems* (2003).

[166] METWALLY, A., FALOUTSOS, C. V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *Proceedings of the VLDB Endowment 5*, 8 (2012).

[167] MICHALOWSKI, M., THAKKAR, S., KNOBLOCK, C. A. Exploiting secondary Sources for unsupervised Record Linkage. In *Proc. of the International Workshop on Data Cleaning, Record Linkage, and Object Consolidation* (2003).

[168] MICHALOWSKI, M., THAKKAR, S., KNOBLOCK, C. A. Automatically Utilizing Secondary Sources to Align Information Across Sources. *AI Magazine 26*, 1 (2005).

[169] MICHELSON, M., KNOBLOCK, C. A. Learning Blocking Schemes for Record Linkage. In *Proc. of the 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference* (2006).

[170] MIERSWA, I., WURST, M., KLINKENBERG, R., SCHOLZ, M., EULER, T. YALE: Rapid Prototyping for Complex Data Mining Tasks. In *Proc. of the 12th International Conference on Knowledge Discovery and Data Mining* (2006).

[171] MINTON, S., NANJO, C., KNOBLOCK, C. A., MICHALOWSKI, M., MICHELSON, M. A Heterogeneous Field Matching Method for Record Linkage. In *Proc. of the 5th International Conference on Data Mining* (2005).

[172] MONGE, A. E., ELKAN, C. The Field Matching Problem: Algorithms and Applications. In *Proc. of the 2nd International Conference on Knowledge Discovery and Data Mining* (1996).

[173] MORETTI, C., BUI, H., HOLLINGSWORTH, K., RICH, B., FLYNN, P. J., THAIN, D. All-Pairs: An Abstraction for Data-Intensive Computing on Campus Grids. *IEEE Transactions on Parallel and Distributed Systems 21*, 1 (2010).

[174] MORTON, K., FRIESEN, A. L., BALAZINSKA, M., GROSSMAN, D. Estimating the Progress of MapReduce Pipelines. In *Proc. of the 26th International Conference on Data Engineering* (2010).

[175] MUNKRES, J. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial & Applied Mathematics 5*, 1 (1957).

[176] NAUMANN, F., HERSCHEL, M. *An Introduction to Duplicate Detection*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[177] NAVARRO, G. A Guided Tour to Approximate String Matching. *ACM Computing Surveys 33*, 1 (2001).

[178] NEWCOMBE, H. Record Linking: The Design of Efficient Systems for Linking Records into Individual and Family Histories. *American Journal of Human Genetics 19*, 3 (1967).

[179] NEWCOMBE, H. B., KENNEDY, J. M. Record Linkage: Making Maximum Use of the Discriminating Power of Identifying Information. *Communications of the ACM 5*, 11 (1962).

[180] NEWCOMBE, H. B., KENNEDY, J. M., AXFORD, S. J., JAMES, A. P. Automatic Linkage of vital Records. *Science 130*, 3381 (1959).

[181] NGOMO, A.-C. N. Link Discovery with Guaranteed Reduction Ratio in Affine Spaces with Minkowski Measures. In *Proc. of 11th International Semantic Web Conference* (2012).

[182] NGOMO, A.-C. N., AUER, S. LIMES - A Time-Efficient Approach for Large-Scale Link Discovery on the Web of Data. In *Proc. of the 22nd International Joint Conference on Artificial Intelligence* (2011).

[183] NGOMO, A.-C. N., KOLB, L., HEINO, N., HARTUNG, M., AUER, S., RAHM, E. When to Reach for the Cloud: Using Parallel Hardware for Link Discovery. In *Proc. of the 10th International Extended Semantic Web Conference* (2013).

[184] NGOMO, A.-C. N., LEHMANN, J., AUER, S., HÖFFNER, K. RAVEN - Active Learning of Link Specifications. In *Proc. of the 6th International Workshop on Ontology Matching* (2011).

[185] NGOMO, A.-C. N., LYKO, K. EAGLE: Efficient Active Learning of Link Specifications Using Genetic Programming. In *Proc. of the 9th International Extended Semantic Web Conference* (2012).

[186] NGOMO, A.-C. N., LYKO, K., CHRISTEN, V. COALA - Correlation-Aware Active Learning of Link Specifications. In *Proc. of the 10th International Extended Semantic Web Conference* (2013).

[187] NYKIEL, T., POTAMIAS, M., MISHRA, C., KOLLIOS, G., KOUDAS, N. MRShare: Sharing Across Multiple Queries in MapReduce. *Proceedings of the VLDB Endowment 3*, 1 (2010).

[188] ODELL, M., RUSSELL, R. C. The soundex coding system. *US Patents*, 1261167 (1918).

[189] OKCAN, A., RIEDEWALD, M. Processing Theta-Joins using MapReduce. In *Proc. of the International Conference on Management of Data* (2011).

[190] OLIVEIRA, P., RODRIGUES, F., HENRIQUES, P., GALHARDAS, H. A Taxonomy of Data Quality Problems. In *Proc. of the International Workshop on Data and Information Quality* (2005).

[191] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., TOMKINS, A. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proc. of the International Conference on Management of Data* (2008).

[192] PANDA, B., HERBACH, J., BASU, S., BAYARDO, R. J. PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce. *Proceedings of the VLDB Endowment 2*, 2 (2009).

[193] PAPADAKIS, G., IOANNOU, E., NIEDERÉE, C., PALPANAS, T., NEJDL, W. Eliminating the Redundancy in Blocking-based Entity Resolution Methods. In *Proc. of the Joint International Conference on Digital Libraries* (2011).

[194] PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., STONEBRAKER, M. A Comparison of Approaches to Large-Scale Data Analysis. In *Proc. of the International Conference on Management of Data* (2009).

[195] PETERMANN, A., JUNGHANNS, M., MÜLLER, R., RAHM, E. Graph-based Data Integration and Business Intelligence with BIIIG. *Proceedings of the VLDB Endowment 7*, 13 (2014).

[196] PIKE, R., DORWARD, S., GRIESEMER, R., QUINLAN, S. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming 13*, 4 (2005).

[197] POLO, J., CASTILLO, C., CARRERA, D., BECERRA, Y., WHALLEY, I., STEINDER, M., TORRES, J., AYGUADÉ, E. Resource-Aware Adaptive Scheduling for MapReduce Clusters. In *Proc. of the 12th International Middleware Conference* (2011).

[198] PORTER, E. H., WINKLER, W. E. Approximate String Comparison and its Effect on an Advanced Record Linkage System. Tech. rep., US Bureau of the Census, 1997.

[199] QUIANÉ-RUIZ, J.-A., PINKEL, C., SCHAD, J., DITTRICH, J. RAFT at Work: Speeding-Up MapReduce Applications under Task and Node Failures. In *Proc. of International Conference on Management of Data* (2011).

[200] QUIANÉ-RUIZ, J.-A., PINKEL, C., SCHAD, J., DITTRICH, J. Rafting mapreduce: Fast recovery on the raft. In *Proc. of the 27th International Conference on Data Engineering* (2011).

[201] RAHM, E. *Mehrrechner-Datenbanksysteme - Grundlagen der verteilten und parallelen Datenbankverarbeitung*. Addison-Wesley, 1994.

[202] RAHM, E. Discovering product counterfeits in online shops: a big data integration challenge. *Journal of Data and Information Quality* (2014, zur Veröffentlichung angenommen).

[203] RAHM, E., BERNSTEIN, P. A. A survey of approaches to automatic schema matching. *VLDB Journal 10*, 4 (2001).

[204] RAHM, E., DO, H. H. Data Cleaning: Problems and Current Approaches. *IEEE Data Engineering Bulletin 23*, 4 (2000).

[205] RAMAN, V., HELLERSTEIN, J. M. Potter's Wheel: An Interactive Data Cleaning System. In *Proc. of the 27th International Conference on Very Large Data Bases* (2001).

[206] RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G. R., KOZYRAKIS, C. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proc. of 13th International Conference on High-Performance ComputerArchitecture* (2007).

[207] RASTOGI, V., DALVI, N. N., GAROFALAKIS, M. N. Large-Scale Collective Entity Matching. *Proceedings of the VLDB Endowment 4*, 4 (2011).

[208] RASTOGI, V., MACHANAVAJJHALA, A., CHITNIS, L., SARMA, A. D. Finding connected components in map-reduce in logarithmic rounds. In *Proc. of 29th International Conference on Data Engineering* (2013).

[209] RONG, C., LU, W., WANG, X., DU, X., CHEN, Y., TUNG, A. K. H. Efficient and Scalable Processing

of String Similarity Join. *IEEE Transactions on Knowledge and Data Engineering 25*, 10 (2013).

[210] SALTON, G., WONG, A., YANG, C. S. A Vector Space Model for Automatic Indexing. *Communications of the ACM 18*, 11 (1975).

[211] SARAWAGI, S., BHAMIDIPATY, A. Interactive Deduplication using Active Learning. In *Proc. of the 8th International Conference on Knowledge Discovery and Data Mining* (2002).

[212] SARMA, A. D., HE, Y., CHAUDHURI, S. ClusterJoin: A Similarity Joins Framework using Map-Reduce. *Proceedings of the VLDB Endowment 7*, 12 (2014).

[213] SCHATZ, M. C. CloudBurst: Highly Sensitive Short Read Mapping with MapReduce. *Bioinformatics 25*, 11 (2009).

[214] SEIDL, T., BODEN, B., FRIES, S. CC-MR - Finding Connected Components in Huge Graphs with MapReduce. In *Proc. of European Conference Machine Learning and Knowledge Discovery in Databases* (2012).

[215] SEIDL, T., FRIES, S., BODEN, B. MR-DSJ: Distance-Based Self-Join for Large-Scale Vector Data Analysis with MapReduce. In *Proc. of the 15th Conference on Database Systems for Business, Technology, and Web* (2013).

[216] SHAFER, J., RIXNER, S., COX, A. L. The Hadoop Distributed Filesystem: Balancing Portability and Performance. In *Proc. of the International Symposium on Performance Analysis of Systems and Software* (2010).

[217] SHEN, W., LI, X., DOAN, A. Constraint-Based Entity Matching. In *Proc. of the 20th National Conference on Artificial Intelligence and the 17th Innovative Applications of Artificial Intelligence Conference* (2005).

[218] SHILOACH, Y., VISHKIN, U. An O(log n) Parallel Connectivity Algorithm. *Journal of Algorithms 3*, 1 (1982).

[219] SIK KIM, H., LEE, D. Parallel Linkage. In *Proc. of the 16th International Conference on Information and Knowledge Management* (2007).

[220] SINGLA, P., DOMINGOS, P. Multi-Relational Record Linkage. In *Proc. of the Workshop on Multi-Relational Data Mining* (2004).

[221] SMITH, T. F., WATERMAN, M. S. Identification of Common Molecular Subsequences. *Journal of Molecular Biology 147*, 1 (1981).

[222] STONEBRAKER, M., ABADI, D. J., DEWITT, D. J., MADDEN, S., PAULSON, E., PAVLO, A., RASIN, A. MapReduce and Parallel DBMSs: Friends or Foes? *Communications of the ACM 53*, 1 (2010).

[223] STONEBRAKER, M., BRUCKNER, D., ILYAS, I. F., BESKALES, G., CHERNIACK, M., ZDONIK, S. B., PAGAN, A., XU, S. Data Curation at Scale: The Data Tamer System. In *Proc. of the 6th Biennial Conference on Innovative Data Systems Research* (2013).

[224] SU, X., SWART, G. Oracle In-Database Hadoop: When MapReduce Meets RDBMS. In *Proc. of the International Conference on Management of Data* (2012).

[225] TARJAN, R. E. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing 1*, 2 (1972).

[226] TEJADA, S., KNOBLOCK, C. A., MINTON, S. Learning Object Identification Rules for Information Integration. *Information Systems 26*, 8 (2001).

[227] TEJADA, S., KNOBLOCK, C. A., MINTON, S. Learning Domain-Independent String Transformation Weights for High Accuracy Object Identification. In *Proc. of the 8th International Conference on Knowledge Discovery and Data Mining* (2002).

[228] THOR, A., RAHM, E. MOMA - A Mapping-based Object Matching System. In *Proc. of the 3rd*

Biennial Conference on Innovative Data Systems Research (2007).

[229] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., MURTHY, R. Hive - A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment 2*, 2 (2009).

[230] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ZHANG, N., ANTHONY, S., LIU, H., MURTHY, R. Hive - A Petabyte Scale Data Warehouse Using Hadoop. In *Proc. of the 26th International Conference on Data Engineering* (2010).

[231] VALDURIEZ, P., KHOSHAFIAN, S. Parallel Evaluation of the Transitive Closure of a Database Relation. *International Journal of Parallel Programming 17*, 1 (1988).

[232] VATSALAN, D., CHRISTEN, P., VERYKIOS, V. S. A taxonomy of privacy-preserving record linkage techniques. *Information Systems 38*, 6 (2013).

[233] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., SAHA, B., CURINO, C., O'MALLEY, O., RADIA, S., REED, B., BALDESCHWIELER, E. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. of the 3rd ACM Symposium on Cloud Computing* (2013).

[234] VERNICA, R., CAREY, M. J., LI, C. Efficient Parallel Set-Similarity Joins Using MapReduce. In *Proc. of the International Conference on Management of Data* (2010).

[235] VOLZ, J., BIZER, C., GAEDKE, M., KOBILAROV, G. Discovering and Maintaining Links on the Web of Data. In *Proc. of 8th International Semantic Web Conference* (2009).

[236] WANG, C., WANG, J., LIN, X., WANG, W., WANG, H., LI, H., TIAN, W., XU, J., LI, R. MapDupReducer: Detecting Near Duplicates over Massive Datasets. In *Proc. of the International Conference on Management of Data* (2010).

[237] WANG, J., LI, G., FENG, J. Trie-Join: Efficient Trie-based String Similarity Joins with Edit-Distance Constraints. *Proceedings of the VLDB Endowment 3*, 1 (2010).

[238] WARNEKE, D., KAO, O. Nephele: Efficient Parallel Data Processing in the Cloud. In *Proc. of 2nd Workshop on Many-Task Computing on Grids and Supercomputers* (2009).

[239] WEIS, M., NAUMANN, F. DogmatiX Tracks down Duplicates in XML. In *Proc. of the International Conference on Management of Data* (2005).

[240] WEIS, M., NAUMANN, F. Detecting Duplicates in Complex XML Data. In *Proc. of the 22nd International Conference on Data Engineering* (2006).

[241] WHANG, S. E., BENJELLOUN, O., GARCIA-MOLINA, H. Generic entity resolution with negative rules. *VLDB Journal 18*, 6 (2009).

[242] WHITE, T. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale, Third Edition*. O'Reilly, 2012.

[243] WILLIAM PHILLIPS, J., BAHN, A. K., MIYASAKI, M. Person-matching by electronic Methods. *Communications of the ACM 5*, 7 (1962).

[244] WINKLER, W. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. In *Proc. of the Section on Survey Research Methods, American Statistical Association* (1990).

[245] WINKLER, W. The State of Record Linkage and Current Research Problems. Tech. rep., US Bureau of the Census, 1999.

[246] WINKLER, W. E. Methods for Record Linkage and Bayesian Networks. Tech. rep., US Bureau of the Census, 2002.

[247] WINKLER, W. E., THIBAUDEAU, Y. An application of the fellegi-sunter model of record linkage to

the 1990 us decennial census. Tech. rep., US Bureau of the Census, 1991.

[248] WITTEN, I. H., MOFFAT, A., BELL, T. C. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann, 1999.

[249] XIAO, C., WANG, W., LIN, X. Ed-Join: An Efficient Algorithm for Similarity Joins With Edit Distance Constraints. *Proceedings of the VLDB Endowment 1*, 1 (2008).

[250] XIAO, C., WANG, W., LIN, X., YU, J. X. Efficient Similarity Joins for Near Duplicate Detection. In *Proc. of the 17th International Conference on World Wide Web* (2008).

[251] XIE, J., YIN, S., RUAN, X., DING, Z., TIAN, Y., MAJORS, J., MANZANARES, A., QIN, X. Improving MapReduce Performance through Data Placement in Heterogeneous Hadoop Clusters. In *Proc. of the International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum* (2010).

[252] XU, Y., KOSTAMAA, P., GAO, L. Integrating Hadoop and Parallel DBMS. In *Proc. of the International Conference on Management of Data* (2010).

[253] YAN, S., LEE, D., KAN, M.-Y., GILES, C. L. Adaptive Sorted Neighborhood Methods for Efficient Record Linkage. In *Proc. of the International ACM/IEEE Joint Conference on Digital Libraries* (2007).

[254] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P. K., CURREY, J. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proc. of the 8th Symposium on Operating Systems Design and Implementation* (2008).

[255] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., STOICA, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. of the 9th USENIX conference on Networked Systems Design and Implementation* (2012).

[256] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R. H., STOICA, I. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. of the 8th Symposium on Operating Systems Design and Implementation* (2008).

[257] ZHAO, H., RAM, S. Entity identification for heterogeneous database integration - a multiple classifier system approach and empirical evaluation. *Information Systems 30*, 2 (2005).

[258] ZIEGLER, P., DITTRICH, K. R. Three decades of data integration - All problems solved? In *Proc. of the 18th IFIP World Computer Congress* (2004).

[259] ZIPF, G. K. The Psycho-Biology of Language: An Introduction to Dynamic Philology. *M.I.T. Press.* (1935).