**Universität Leipzig**

**Fakultät für Mathematik und Informatik**

**Institut für Informatik**

# Bachelorarbeit

Wiederverwendung berechneter Matchergebnisse für MapReduce-basiertes Object Matching

Leipzig, Juli 2013

vorgelegt von
Sintschilin, Sergej
Studiengang Informatik

Betreuender Hochschullehrer:

Prof. Dr. Erhard Rahm
Fakultät für Mathematik und Informatik
Angewandte Informatik

# Inhaltsverzeichnis

# 1 Einleitung

Im Rahmen der Bachelorarbeit soll ein existierendes Projekt, namens Dedoop, um zusätzliche Features erweitert werden. Mit Dedoop lassen sich Ähnlichkeiten zwischen den Paaren von Entitäten aus großen Datenmengen feststellen. Die Verarbeitung der Daten erfolgt auf der Grundlage von MapReduce-basierten Verfahren. Das Finden von Ähnlichkeiten ist eine Berechnung, die sehr viel Rechenzeit in Anspruch nimmt. Falls eine Änderung von einer oder mehreren Entitäten in der ursprünglichen Menge erfolgte, muss die Berechnung neu gestartet werden. Die Aufgabe der Bachelorarbeit ist, auf die vollständige Wiederholung der Vergleiche zu verzichten und einen Weg zu finden, unter der Verwendung von bereits berechneter Matchergebnisse sich nur auf bestimmte Teilmengen zu beschränken, die die Neuberechnung benötigen.

Es wurde eine Begrenzung festgelegt, was den Beschäftigungsumfang an der Aufgabe der Bachelorarbeit betrifft. Es gibt mehrere Verfahren, mit denen Dedoop das Object Matching durchführt. In manchen Fällen ist das Endergebnis von der Reihenfolge der Entitäten abhängig. Es gibt keinen natürlichen Weg, der mit dem gewöhnlichen MapReduce-Ansatz die gleichen Ergebnisse aus den Partitionen erzielt, wo die gesamte Menge nötig ist. Das Beschränken auf die geänderten Datensätze führt immer zu einem fälschlichen Ergebnis. Deshalb werden solche Match-Verfahren, wie Sorted Neighborhood, in dieser Arbeit nicht betrachtet.

# 2 Grundlagen

Bevor man auf die Funktionsweise des Bachelorprojektes eingehen darf, wird einiges Wissen über die Elemente, aus denen sich Dedoop zusammensetzt, abverlangt. Dedoop ist vollständig in Java geschrieben. Deswegen ist das Beherrschen dieser Programmiersprache für die Erweiterung des Tools erforderlich. Die Berechnungen für das Object Matching sind mit dem MapReduce-Algorithmus realisiert und das dafür benutzte Framework ist Hadoop. Dabei arbeitet man mit der alten Java-MapReduce-Schnittstelle, weil Dedoop um diese Schnittstelle aufgebaut ist. In seinen kommenden Versionen unterstützt Hadoop weiterhin die alte Schnittstelle, weil viele Projekte immer noch nicht auf die neue umgestiegen sind [8]. Zusätzlich soll die Bedienung der neu eingebauten Features in die vorhandene grafische Benutzerschnittstelle integriert werden, für dessen Aufbau das Toolkit GWT eingesetzt wurde.

## 2.1 MapReduce

MapReduce ist ein Programmiermodell für nebenläufige, verteilte Berechnungen über riesige Datenmengen unter der Benutzung von mehreren Rechnern. MapReduce wurde von Google im Jahr 2004 vorgestellt. Das Implementierungskonzept gewann schnell an Popularität und kann heute nahezu überall eingesetzt werden; die Umsetzung ist bereits in C++ [10], C# [11], Ruby [12], Java [13], Erlang [14] und Python [15] realisiert. Es entstand aus der Notwendigkeit, die großen Mengen an Daten, die für den Hauptspeicher zu groß sind, zu analysieren und zu bearbeiten. Im Allgemeinen konzentriert sich das MapReduce-Konzept auf die Verarbeitung von im Petabyte-Bereich befindlichen Daten. Demzufolge wird der Ansatz oft auf Computerclustern angewendet. [2s1]

Egal in welchem Framework der MapReduce-Algorithmus implementiert ist, es wird stets bis auf wenige Details dem gleichen Konzept gefolgt. Eine MapReduce-Berechnung nimmt eine Menge von Schlüssel-Wert-Paaren entgegen als Eingabe und gibt eine Menge von Schlüssel-Wert-Paaren als Ausgabe zurück. Jede Berechnung besteht aus zwei Phasen; zunächst erfolgt Map und danach Reduce. Die Arbeitsweise innerhalb jeder Phase wird durch die vom Programmierer geschriebenen Funktionen bestimmt. In der Map-Phase erfolgt eine Iteration über die Eingabepaare. Die Map-Funktion wird für jedes Eingabepaar aufgerufen, um eine Menge von zwischenliegenden Schlüssel-Wert-Paaren zu generieren. Die Funktion kann ebenso als ein Filter dienen, der nichts zurückgibt, falls für die aktuelle Eingabe keine Ergebnisse zu generieren sind. Sobald die Map-Phase abgeschlossen ist, erfolgt die automatische Vereinigung aller zwischenliegenden Werte mit dem gleichen Schlüssel zu einer Gruppe. In der Reduce-Phase iteriert man über die resultierenden Gruppen. Die Reduce-Funktion nimmt jeden Schlüssel und die dazugehörige Werteliste und reduziert diese Liste zu einem finalen Wert. Weil die beiden Funktionen zustandslos sind und die einzelnen Instanzen nicht voneinander abhängen, lassen sich alle Berechnungen auf Clustern aus einer beliebigen Anzahl von Rechnern problemlos parallelisieren. Demzufolge kann man theoretisch eine der Eingabepaaren entsprechende Anzahl von Map-Funktion-Instanzen und eine der zwischenliegenden Gruppen entsprechende Anzahl von Reduce-Funktion-Instanzen konstruieren und die Instanzen ihre einfließenden Daten in der jeweiligen Phase gleichzeitig bearbeiten lassen. [2s1-2, 3s18-19]

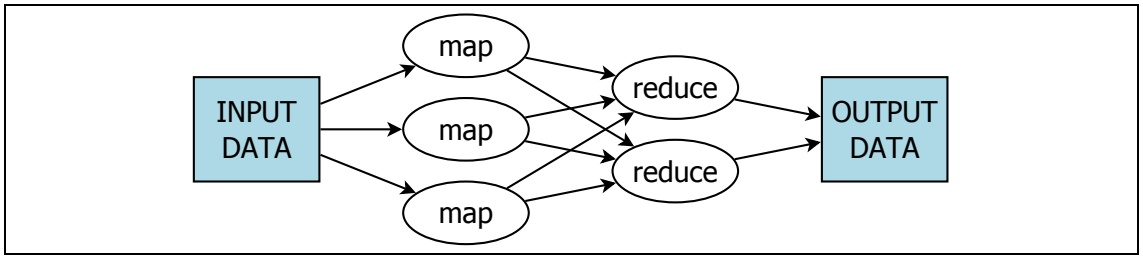


**Abbildung 2.1-1.** Vereinfachte Darstellung des Datenflusses im MapReduce-Modell.

**Beispiel 2.1-2.** Eines der klassischen Beispiele für die Verwendung von MapReduce ist das Zählen der Wortvorkommen. Dafür teilt man den Text in unabhängige Blöcke, die im Map-Schritt verarbeitet werden. Beim Ausgang einer Map-Funktion bekommt man ein Paar, bestehend aus dem Wort und seiner Häufigkeit der Verwendung. Im einfachsten Fall analysiert die Map-Funktion den Text nicht, sondern teilt ihn nur nach den Leerzeichen und erzeugt für jedes Wort eine Rückgabe. Als Folge erhält man aus einen Textblock, in dem das Wort „Hallo“ fünf Mal vorkommt, kein Hallo-5-Paar, sondern fünf Hallo-1-Paare. Innerhalb der Reduce-Funktion summiert man die einzelnen Werte eines Schlüssels. In beiden Fällen ergibt sich daraus die Anzahl der Vorkommen eines Wortes im gesamten Text. Diese nicht ganz einfache Methode zum Ermitteln der Anzahl der Wörter ist nützlich, wenn die Größe des zu analysierenden Textes in Gigabyte (oder mehr) gemessen wird.

Der Ansatz von MapReduce zur Datenverarbeitung bietet zwei wesentliche Vorteile gegenüber den herkömmlichen Lösungen. Der erste und wichtigste Vorteil ist die Leistung. MapReduce kann parallelisiert werden, was erlaubt, die volle Ausnutzung der zur Verfügung stehenden Ressourcen zu erreichen und somit große Mengen an Daten auf mehreren Kernen, Prozessoren und Rechnern zu verarbeiten. Der zweite Vorteil ist die Möglichkeit, ein MapReduce-Programm mit dem üblichen Code zu schreiben, auf dem das Framework implementiert ist. Im Vergleich zu dem, was man mit SQL durchführen kann, sind die Möglichkeiten innerhalb eines in einer Programmiersprache geschriebenen Codes für MapReduce auch ohne den Einsatz von spezialisierten Lösungen viel größer.

Das Programmiermodell von MapReduce kommt bei den datenintensiven Anwendungen zum Einsatz, wo eine Parallelisierung der benutzten Algorithmen zum Lösen der gestellten Aufgaben unverzichtbar ist [7]. Es findet eine sehr breite Anwendung in solchen komplexen Problemgebieten, wie Bildung von verschiedenen Statistiken für Webseiten, maschinelles Lernen, Data-Mining, Erstellung von verschiedenen Datenstrukturen (z.B. Graphen), Analyse von sehr großen Textkorpora (z.B. Konstruktion von Term-Frequenz-Matrizen) und Bearbeitung von digitalen Daten (z.B. Satellitenbilder) [2s2-3].

## 2.2 Hadoop

Hadoop ist eine Plattform für die Erstellung von Anwendungen, die große Datenmengen verarbeiten können. Das System beruht auf einem verteilten Ansatz zur Datenverarbeitung und zur Speicherung von Informationen. Diese beiden Ansätze wurden aus den Konzepten für den MapReduce-Algorithmus und für Google-Dateisystem übernommen, als Google sie publizierte. Die Entwicklung von Hadoop begann im Rahmen des Projektes Nutch – eine kostenlose Crawler-basierte Internet-Suchmaschine, geschrieben in Java. Im Frühjahr 2006 wurde Hadoop als ein vollwertiges Projekt unter die Obhut von Yahoo! genommen. Im Februar 2008 nahm Yahoo! die weltgrößte auf Hadoop basierende Applikation in Betrieb. Sie läuft auf mehr als zehntausend Prozessorkernen und verarbeitet Daten, die in allen Yahoo!-Suchmaschinen eingesetzt werden. Im Januar 2008 wurde Hadoop ein Top-Level-Projekt von Apache Software Foundation. Ein Jahr später stellte Yahoo! den Quelltext von Hadoop der Öffentlichkeit zur Verfügung. Von diesem Moment an begann die weit verbreitete Nutzung von Hadoop außerhalb von Yahoo!; die Technologie kommt in vielen großen Organisationen zum Einsatz, unter anderem eBay, Google, Facebook, IBM, ImageShack, Last.fm, Microsoft und Twitter [7]. Im April 2008 stellte Hadoop mit der kürzesten Laufzeit zum Sortieren von einem Terabyte an Daten einen neuen Weltrekord auf. [3sXV, 3s10-12]

### 2.2.2 HDFS

Der Hauptgrund für die Verwendung des MapReduce-Modells ist die Verarbeitung großer Datenmengen. Deshalb ist es notwendig, diese Daten zuerst in einem bestimmten Speichermedium abzulagern und für den MapReduce-Cluster zugänglich zu machen. Allerdings ist es unpraktisch, solche großen Datenmengen auf lokalen Dateisystemen zu halten und es ist noch unpraktischer, die Daten zwischen mehreren Knoten im Cluster zu synchronisieren. Um dieses Problem anzugehen, liegt im Kern des gesamten Frameworks ein verteiltes Dateisystem, genannt Hadoop Distributed File System (HDFS), welches leicht über mehrere Knoten in einem Cluster skaliert. Die wichtigsten Merkmale von HDFS sind Skalierbarkeit, Sparsamkeit, Zuverlässigkeit, Effizienz und Plattformunabhängigkeit. Mit HDFS ist eine sichere Aufbewahrung und Verarbeitung von großen Datenmengen, die bis in den Petabyte-Bereich reichen können, möglich. Daten und Berechnungen werden über einen Cluster verteilt, der aus Hunderten oder sogar Tausenden von Rechnern bestehen kann. Diese Rechner können auch aus kostengünstigen und unzuverlässigen Serverhardware aufgebaut sein.

Trotzdem bleibt HDFS sehr robust. Bei der Entwicklung des Systems wurden die Hardwareausfälle eher als eine Norm und nicht als eine Ausnahme betrachtet. Der Replikationsmechanismus von HDFS stellt sicher, dass kein Verlust von Informationen im Falle von Systemausfällen auftritt. Bei der Aufbewahrung der Daten ist es möglich, mehrere Kopien einer Datei auf dem Cluster abzuspeichern. Die Verteilung der Daten ermöglicht ihre Verarbeitung parallel auf mehreren Knoten durchzuführen, was den Prozess erheblich beschleunigt. Außerdem laufen die MapReduce-Berechnungen möglichst nah an den Daten, um den Datenverkehr zwischen den Knoten gering zu halten. Des Weiteren soll HDFS portabel und plattformunabhängig sein, um eine große Vielfalt von Anwendungen zu unterstützen. Weil HDFS grundlegend in der Programmiersprache Java geschrieben ist, lässt es sich auf jedem Betriebssystem einsetzen, welches eine JVM (Java Virtual Machine) besitzt. [2s3, 3s41-46, 5]

### 2.2.3 Hadoop MapReduce

In Hadoop ist eine Implementation des MapReduce-Frameworks für Java vorhanden. Das implementierte MapReduce-Modell weist eine Master-Slave-Architektur [3s28-30] auf. Der Koordinator (Master) soll nur auf einem Knoten im Cluster ausgeführt sein. Alle anderen Knoten sind als Arbeiter (Slaves) definiert. Der Koordinator erfüllt die ganze Arbeit, die mit den Metadaten zusammenhängt, und steuert den laufenden Betrieb im System. Die Arbeiter erledigen die lokalen Aufgaben auf Anweisung des Koordinators oder eines Clients. Ein Client ist eine einfache Anwendung oder ein Benutzer, der mit dem System arbeitet. In die Rolle des Clients kann praktisch alles treten, was auf die Hadoop-Schnittstelle zurückgreift. Die gesamte Kommunikation zwischen den Komponenten des Systems verläuft durch die speziellen Protokolle, die auf TCP/IP [5] basiert sind. Eine MapReduce-Berechnung wird als ein Job bezeichnet. Der Koordinator akzeptiert Job-Anfragen von den Clients und startet die Berechnungen auf den Arbeitern [3s146]. Die MapReduce-Programme sind in den JAR-Dateien gepackt. Das Ausführen eines Jobs verschiebt diese Dateien auf das HDFS und benachrichtigt den Koordinator, wo das Programm zu finden ist, damit er die Dateien über den gesamten Cluster verteilen kann. Die Programme sind konfigurierbar. Die für einen Job relevanten Parameter werden als Schlüssel-Wert-Paare in einem `JobConf`-Objekt [3s22-23] gehalten. Bei der Ausführung serialisiert ein Client-Programm das Objekt und verschickt es an den Koordinator. Die `JobConf` soll die Mapper- und die Reducer-Klasse identifizieren und die Eingabe- und Ausgabeverzeichnisse angeben. Optionale Parameter sind unter anderem die Anzahl der Prozesse innerhalb der Map- und Reduce-Phase und das Format der Eingabe- und Ausgabedateien. Zusätzlich kann man eigene

Parameter mit der `JobConf` versenden, falls sie von den benutzerdefinierten Map- und Reduce-Funktion benötigt werden. Nach dem Empfang der für eine MapReduce-Berechnung benötigten Konfigurationsparameter vergibt der Koordinator die Aufgaben an die verfügbaren Arbeiter und stellt dem Client umfassende Informationen über den Fortschritt innerhalb jedes Arbeiters bereit [3s170-171]. Auf ihren Knoten können die Arbeiter mehrere Prozesse für die aktuelle Aufgabe laufen lassen [3s169]. In jedem Prozess ist ein Mapper oder ein Reducer instanziiert. Die einzelnen Instanzen tauschen weder Informationen miteinander aus, noch sind sie über die Existenz voneinander bewusst. Der Benutzer soll nie explizit in die Kommunikation zwischen den Mappern und den Reducern eingreifen, weil die gesamte Datenübertragung durch die Hadoop-MapReduce-Plattform selbstständig behandelt wird. Die einzelnen Prozesse sind absichtlich mit keinem Mechanismus für den Informationenaustausch miteinander in irgendeiner Weise ausgestattet. Dadurch wird die Zuverlässigkeit jedes Prozesses nur durch die Zuverlässigkeit des lokalen Knotens bestimmt. Im Falle eines Knotenausfalls ist der Koordinator in der Lage die Prozesse umzuverteilen, ohne dabei die Arbeit auf den anderen Knoten zu beeinträchtigen [3s175]. In diesem Fall liegt der Unterschied zwischen den Map- und den Reduce-Prozessen darin, dass ein Mapper die Zwischenergebnisse lokal abspeichert, was beim einem Fehler zu ihrem Verlust führt und einen Neustart der Berechnung erfordert, während ein Reducer seine Ausgabe in das globale Repository verlagert [2s4].

Die Quelle der Eingabedaten für eine MapReduce-Berechnung kann alles Mögliche sein. Deshalb bietet Hadoop den Nutzern die Möglichkeit, den Datenfluss mit dem benutzerdefinierten Code zu steuern. Wie die Eingabedaten separiert und gelesen werden, wird durch eine `InputFormat`-Klasse [3s198-200] spezifiziert. Man kann auswählen, welches `InputFormat` man für die Eingabedaten eines Jobs einsetzt. Einige `InputFormat`-Klassen sind in der Hadoop-Bibliothek vorab vorhanden. In der Regel stammen die Eingabedaten für einen Job aus den Dateien, die auf einen laufenden Cluster in HDFS gespeichert sind. Wenn man Dateien als Datenquelle benutzt, soll man das `InputFormat` wählen, das die Funktionalitäten und die Eigenschaften von der abstrakten Klasse `FileInputFormat` [3s200-202] erbt. Wenn man einen Hadoop-Job startet, wird das `FileInputFormat` mit einem Pfad versorgt, der die Dateien zum Lesen beinhaltet. Danach beginnt das `FileInputFormat`, alle Dateien in diesem Verzeichnis zu lesen. Da für ein MapReduce-Problem typisch ist, dass die Größe der Eingabe bis zu mehreren Gigabytes reicht, bricht das `FileInputFormat` die Eingabedaten in mehrere disjunkte Blöcke, die standardmäßig 64 MB groß sind. Hadoop erstellt einen Prozess für jeden Block. Basierend darauf, wo die Eingabebestandteile sich physikalisch befinden, werden die Prozesse zu den Knoten im System zugewiesen. Falls eine Datei über mehrere Knoten im Cluster verteilt ist, besteht keine Notwendigkeit, die Blöcke

zwischen einander zu übertragen, stattdessen werden diese lokal verarbeitet. Einem einzelnen Knoten können mehrere Prozesse zugewiesenen sein. Jeder Knoten versucht, möglichst viele Prozesse parallel auszuführen. Wenn die Eingabemenge sehr groß ist, kann die Leistung durch Parallelität erheblich verbessert werden. Nachdem die Prozesse verteilt sind, definiert das `InputFormat`, wie man darauf zugreift. Man lädt die Daten von ihrer Quelle und wandelt sie in Schlüssel-Wert-Paare um, die als Eingabeparameter für den Mapper geeignet sind. Für beliebige Dateiformate existieren verschiedene Implementierungen der Klasse `FileInputFormat`. Wenn man die Klasse im `JobConf`-Objekt nicht spezifiziert, wird standartmäßig das `TextInputFormat` [3s209-210] genommen. Es behandelt jede Zeile einer Datei als einen einzelnen Eintrag und führt kein Parsen durch. Dies ist nützlich für zeilenorientierte Daten wie CSV-Dateien. Es gibt weitere wichtige `FileInputFormats`. Das `KeyValueTextInputFormat` [3s211] ist dem `TextInputFormat` ähnlich, jedoch mit dem Unterschied, dass es die Zeilen als Schlüssel-Wert-Paare einliest. Das `SequenceFileInputFormat` [3s213] versteht die speziellen binären Dateien, die für Hadoop spezifiziert sind, was die Einlesegeschwindigkeit deutlich steigert. Dieses `InputFormat` wird bei der Definition eines Workflows eingesetzt. Ein Workflow [3s163] ist eine geordnete Menge von Jobs, die nacheinander ausgeführt werden; die Ausgabe des ersten Jobs ist die Eingabe des zweiten, die Ausgabe des zweiten Jobs ist die Eingabe des dritten usw. Die Sequenzdateien können als Ausgabe eines Jobs dienen, um sofort von einem anderen Job verarbeitet zu werden.

Hadoop instanziiert für jeden Block, den das `InputFormat` generiert, einen separaten Prozess mit einem Mapper. Die `Mapper`-Klasse [4] enthält den benutzerdefinierten Code der Map-Methode. Zum jeden gegebenen Schlüssel-Wert-Paar liefert die Methode eine Menge von Schlüssel-Wert-Paaren. Diese Paare werden im Speicher gepuffert und regelmäßig auf die lokale Festplatte geschrieben. Wenn die Map-Phase beendet ist, müssen die Schlüssel-Wert-Paare zwischen den Knoten ausgetauscht werden, um alle Werte mit dem gleichen Schlüssel zu einem Reducer zu senden. Dieser Vorgang zwischen den zwei Phasen wird als „Shuffle“ [2s3-4, 3s177-180] bezeichnet. Der Austausch der Zwischenergebnisse beginnt schon, nachdem die ersten Map-Prozesse erfolgreich abgeschlossen sind, während noch etliche laufen können. Ein fertiger Mapper leitet die Information über den Standort seiner Zwischenergebnisse an den Koordinator, der diese Information wiederrum an die Arbeiter mit einem Reduce-Prozess sendet. Wenn ein Reducer vom Koordinator über diese Standorte benachrichtigt wird, benutzt der Arbeiter entfernte Prozeduraufrufe, um diese Zwischenergebnisse vom Knoten, wo der Mapper sie ablagerte, zu lesen. Alle Werte mit dem gleichen Schlüssel werden immer der gleichen Reducer-Instanz zugewiesen, unabhängig davon, aus welchem Mapper die Ergebnisse stammen. Deswegen müssen sich alle Knoten darauf einigen, an welchen Reducer die Zwischenergebnisse zu senden

sind. Dafür teilt der Prozess die Ausgabe seines Mappers in Partitionen, deren Anzahl der vorab initialisierten Anzahl der Reduce-Prozesse entspricht. Die `Partitioner`-Klasse [4] bestimmt, zu welcher Partition ein bestimmtes Schlüssel-Wert-Paar gehört. Der standartmäßige `Partitioner` berechnet einen Hash-Wert aus dem Schlüssel und ordnet das Paar basierend auf diesem Ergebnis einer Reducer-Instanz zu. Im Reduce-Prozess erfolgt die lokale Sortierung der Eingabepaare nach dem Schlüssel, so dass die Paare mit dem gleichen Schlüssel gruppiert werden. Die Reihenfolge der Werte innerhalb einer Gruppe ergibt sich zufällig. Die `Reducer`-Klasse [4] enthält den benutzerdefinierten Code der Reduce-Methode. Die Methode wird für jeden Schlüssel einmal aufgerufen. Die zwischenliegenden Werte, die mit dem Schlüssel assoziiert sind, werden über einen Iterator an die Reduce-Funktion weitergeleitet. Dieser Ansatz ermöglicht die Listen von Werten zu behandelt, die zu groß für den Speicher sind. Wie die Mapper geben auch die Reducer eine Menge von Schlüssel-Wert-Paaren zurück, die jedoch an das endgültige Ergebnis des Jobs angehängt werden. Neben dem Schlüssel und dem Wert bzw. dem Iterator über die Werte erhalten die Map- und die Reduce-Methoden zwei weitere Parameter: `OutputCollector` und `Reporter` [4]. Das `OutputCollector`-Objekt besitzt eine Methode, die ein Schlüssel-Wert-Paar annimmt und weiterleitet. Das `Reporter`-Objekt liefert Informationen über den aktuellen Prozess, sowie über den Gesamtfortschritt im System. Des Weiteren stellt es einige asynchrone Funktionen für das Feedback bereit. Damit kann man Statusmeldungen an den Benutzer schicken und etliche Zähler definieren, dessen Stand innerhalb jedes Prozesses erhöht werden kann, um zum Ende des Jobs eine Summe für spätere Auswertungen daraus zu bilden.

Die Schlüssel-Wert-Paare, die der `OutputCollector` in der Reduce-Phase sammelt, werden in die Ausgabedateien geschrieben. Die Art, wie sie geschrieben werden, wird durch die `OutputFormat`-Klasse geregelt. Das `OutputFormat` funktioniert in ähnlicher Weise wie das `InputFormat`. Die `OutputFormats`, die von der `FileOutputFormat`-Klasse erben, schreiben die Ausgabe in die Dateien auf dem HDFS. Jeder Reducer schreibt eine separate Datei in das gemeinsame Verzeichnis. Diese Dateien werden in der Regel mit dem Namen `part-nnnnn` versehen, wobei es sich bei `nnnnn` um die Partitions-ID handelt, die dem Reduce-Prozess zugeordnet ist [3s151]. Hadoop stellt einige `FileOutputFormat`-Klassen [3s216] zur Verfügung. Die standartmäßige Klasse ist das `TextOutputFormat` [3s216]. Es schreibt Schlüssel-Wert-Paare in die einzelnen Zeilen einer Textdatei. Dieses Format ist von Menschen lesbar und kann durch die `KeyValueTextInputFormat`-Klasse wieder gelesen werden. Ein besseres Zwischenformat für die Nutzung zwischen zwei MapReduce-Jobs ist das `SequenceFileOutputFormat` [3s216-217]. Es serialisiert schnell beliebige Datentypen in die Datei. Das dazugehörige `SequenceFileInputFormat` wird diese Dateien in die ursprüngliche Typen deserialisieren und die Daten dem nächsten Mapper darlegen, in der gleichen Weise, wie der vorhergehende Reducer sie ausgab.

```
public class WordCount {
  public class WordCountMapper
      extends MapReduceBase
      implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      // iterate over all words in the line and form (key, value) pairs
      while (tokenizer.hasMoreTokens()) {
        String word = tokenizer.nextToken();
        output.collect(new Text(word), one); // send the pair to reducer
      }
    }
  }
  public class WordCountReducer
      extends MapReduceBase
      implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
      int sum = 0;
      // iterate over all values and add them up
      while (values.hasNext())
        sum += values.next().get();
      output.collect(key, new IntWritable(sum)); // send the pair to output
    }
  }
  public static void main(String[] args) {
    JobConf conf = new JobConf(WordCount.class); // create a JobConf object
    conf.setJobName("WordCount"); // assign a identification name for the job
    // set the output data type classes for key and value
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    // set the class names for mapper and reducer
    conf.setMapperClass(WordCountMapper.class);
    conf.setReducerClass(WordCountReducer.class);
    // set the HDFS input and output directories
    FileInputFormat.addInputPath(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    JobClient.runJob(conf); // launch the job
  }
}
```

**Code 2.2.3-1.** Implantation des MapReduce-Jobs aus dem Beispiel 2.1-2.

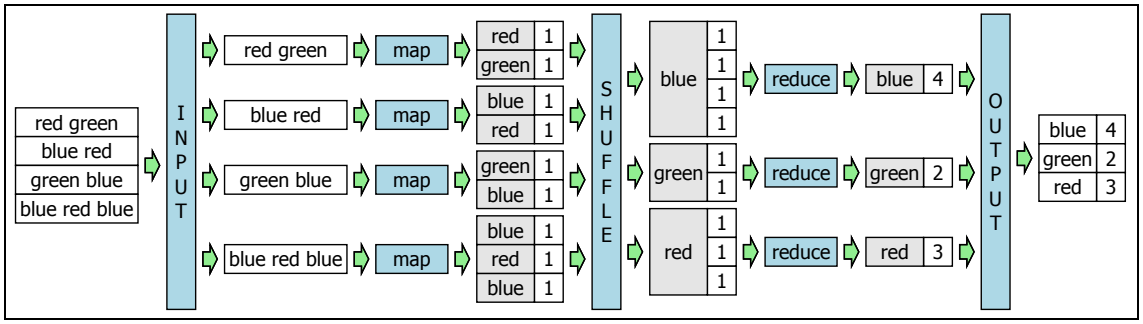


**Abbildung 2.2.3-2.** Visualisierung des Datenflusses bei der Ausführung des Jobs mit dem Code 2.2.3-1. Als Beispiel besteht die Eingabe aus 4 Textzeilen; die Map-Methode wird insgesamt 4 Mal aufgerufen. Am Ende der Map-Phase werden 3 verschiedene Schlüssel erzeugt; die Reduce-Methode wird insgesamt 3 Mal aufgerufen. Der Job ist mit 1 bis 4 Map-Prozessen und mit 1 bis 3 Reduce-Prozessen durchführbar.

# 2.3 GWT

Google Web Toolkit (GWT) ist eine Entwicklungsumgebung für den Aufbau und die Optimierung der komplexen Browser-basierten Anwendungen. Sein Ziel ist es, den Entwicklern zu erlauben, dynamische und leistungsstarke Webanwendungen zu schrieben, ohne, dass sie über fachliches Wissen von Browser-Besonderheiten, XMLHttpRequest (XHR), JavaScript und Document Object Model (DOM) verfügen.

GWT erreicht diese Ziele mit Hilfe des GWT-Compilers [19], der den JavaScript-Code aus dem Java-Code generiert. Der Compiler unterstützt die Klassen aus den Paketen `java.lang`, `java.util` und `java.io` [18] und versteht die von GWT zur Verfügung gestellte API-Schnittstelle, mit deren Hilfe man die komplexen grafischen Benutzeroberflächen erstellt, die in einem Web-Browser eines Benutzers angezeigt und ausgeführt werden. GWT erlaubt, AJAX-Anwendungen in Java zu schreiben, deren Quelltext dann zum hochoptimierten JavaScript-Code kompiliert wird. Die Konstruktion von AJAX-Anwendungen auf diese Weise ist produktiver dank einer hohen Ebene der Abstraktion von den gemeinsamen Konzepten wie DOM-Manipulation und XHR-Kommunikation. Die Nutzung von GWT ist viel ähnlicher einer GUI-Entwicklung mit Swing oder SWT als eine normale Entwicklung von Web-basierten Anwendungen [20]. Man ist auch nicht an die in GWT eingebauten Widgets eingeschränkt. Alles, was man mit HTML und JavaScript tun kann, kann in GWT durchgeführt werden, einschließlich der Interaktion mit dem von Hand geschriebenen JavaScript. Wenn eine Anwendung bereit ist, implementiert zu werden, wandelt GWT den Java-Quelltext in eigenständige JavaScript-Dateien um, die für alle gängigen Browser, als auch für die mobilen Browser auf Android und iPhone

optimiert sind [17]. Eine kompilierte GWT-Anwendung besteht aus den Fragmenten von HTML, XML und JavaScript. Allerdings ist der generierte Inhalt aufgrund seiner Komplexität fast unmöglich zu lesen, deswegen soll man die kompilierte Anwendung am besten als eine Black Box betrachten.

GWT folgt der modernen Webanwendung und setzt auf das „Single-Page“-Prinzip. Das heißt, dass der Browser nicht mehr als ein gewöhnliches HTML-Terminal, sondern als eine Anwendungsplattform verwendet wird. Eine Single-Page-Applikation (SPA) wird beim ersten Abruf der Seite komplett geladen. Allerdings muss der Anwender damit rechnen, dass im Gegensatz zum klassischen Ansatz die initiale Downloadgröße der Anwendung mit ihrer Komplexität wächst. Danach muss die Webseite nicht mehr neugeladen werden, weil der Browser die Seite im Cache behält und alle Benutzerinteraktionen intern abarbeitet. Beim Bedarf erfolgt eine Client-Server-Kommunikation. SPA lädt ausschließlich die benötigten Daten und keine vollständige Seite. Die Anwendung verarbeitet die eingehenden Daten, mit denen bestimmte Seitenregionen aktualisiert oder erweitert werden. Dieser Single-Page-Ansatz hilft das übermäßige Herunterladen vom ungenutzten Inhalt zu vermeiden, was die serverseitigen Ressourcen spart und die Latenz in der Applikation vermindert. [24]

Zwar benutzt die GWT-Umgebung eine normale Trennung des Server- und des Client-Codes, verfügt sie jedoch über die JVM-Methoden, mit denen man die Lücke zwischen dem Java-Bytecode im Debugger und dem JavaScript des Browsers überbrücken kann. Wenn man die Anwendung im Development-Modus startet, kann man somit den Code auf dem Client-Rechner in der bevorzugten Java-IDE wie eine normale Desktop-Applikation debuggen. Es findet keine Kompilierung des Codes statt. Stattdessen startet GWT intern einen Java-basierten HTTP-Server als Web-Container. Man kann mit der grafischen Web-Oberfläche in verschiedenen Browsern arbeiten und gleichzeitig den Java-Code betrachten, der den entsprechenden clientseitigen JavaScript-Code darstellt. Alle Änderungen in der Entwicklungsumgebung werden durch das Aktualisieren der Seite im Browser sofort sichtbar. Ein spezieller Mechanismus ermöglicht auch die Kontrolle von Variablen, das Setzen von Haltepunkten im Java-Quelltext und die Nutzung aller anderen in Java zur Verfügung stehenden Debugger-Tools. Dies ist ein wichtiger Punkt, weil das Debugging vom generierten JavaScript-Code eine nahezu unmögliche Aufgabe ist. [19]

GWT wird in vielen Produkte von Google verwendet, einschließlich Google Wave und der neuen Version von AdWords [16]. Google hat ein großes Interesse daran, dass immer mehr Entwickler Web-2.0-Anwendungen schreiben, und bietet deshalb GWT als Open-Source-Projekt an [23]. Es wird bereits von Tausenden von Entwicklern auf der ganzen Welt verwendet. Die GWT-Laufzeitbibliotheken sind unter Apache License 2.0

[22] lizensiert und man kann GWT frei dazu benutzen, kommerzielle Anwendungen zu entwickeln. Allerdings liegt die GWT-Toolbar nur in binärer Form dar und keine Änderungen an ihr sind erlaubt. Zu beachten ist, dass dies auch für den Java-nach-JavaScript-Compiler gilt, was bedeutet, dass etwaige Fehler im generierten JavaScript-Code nicht kontrollierbar sind. Eine neue Version des Browsers erfordert eventuell die Aktualisierung von den GWT-Komponenten, um sich an die geänderten Standards anzupassen [21].

## 2.4 Dedoop

Dedoop [1] ist ein mächtiges und ein leicht zu bedienendes Tool für MapReduce-basiertes Object Matching von großen Datenmengen. Über eine Weboberfläche lassen sich komplexe Workflows zur Lösung von Object Matching-Problemen definieren. Optional kann man weitere Jobs für zusätzliche Aufgaben, wie maschinelles Lernen, an den Workflows anhängen. Die definierten Workflows werden automatisch in ausführbare Workflows übersetzt und auf Hadoop-Clustern laufen gelassen. Der Name Dedoop stammt aus dem Wörterspiel „Deduplication with Hadoop“. Die Arbeit an Dedoop begann Ende Januar 2012 und war Ende März weitestgehend fertig gestellt. Die Entwicklung von Hadoop fand innerhalb der Abteilung Datenbanken am Institut für Informatik in der Universität Leipzig statt. Das Projekt wurde allein von Lars Kolb implementiert. Bei der Konzeption von Dedoop waren Dr. Andreas Thor und Prof. Dr. Erhard Rahm beteiligt. Bis heute wird Dedoop weiterentwickelt. Später erweiterten zwei Studenten, Axel Fischer und Ziad Sehili, im Rahmen ihrer Abschlussarbeiten und Hilfskrafttätigkeiten einige Funktionalitäten des Tools.

Unter dem Object Matching (oder Deduplizierung oder Entity Resolution) ist ein Prozess gemeint, wo eine Suche nach Entitäten, die auf das gleiche Objekt aus der realen Welt verweisen, sattfindet, um sie schließlich in eine gemeinsame Gruppe zu fassen. Das kritische Problem, mit dem sich das Object Matching befasst, ist das Identifizieren von redundanten Daten. Die Redundanz kann bei der Informationsintegration oft der Fall sein. Wenn man zwei Datenquellen zusammenführt, die ein unterschiedliches Format des Inhaltes aufweisen, kann es das Finden von möglichen Duplikaten erschweren. Ein ähnlicher Fall kann eintreten, wenn man – als Beispiel vorgeführt – Informationen zu den Personen aus unterschiedlichen sozialen Netzwerken in eine gemeinsame Datenbank laden will, ohne dabei eine Person, die sich auf den beiden Portalen registrierte, doppelt in der Datenbank vorzufinden. Das Object Matching ist ein sehr rechenintensiver Prozess, da die Laufzeit mit der Anzahl der Entitäten exponentiell wächst. Beim gewöhnlichen Ansatz müssen alle Entitäten miteinander vergleichen

werden, um alle Matches zu finden. Diese Vorgehensweise ist bei sehr großen Mengen undenkbar, da die Anzahl der Vergleiche dramatisch steigt. Wenn man die zehntausend Entitäten aus einer Menge mit den zehntausend Entitäten aus einer anderen Menge vergleichen will, so ist das Produkt der beiden Mengen, also insgesamt hundert Millionen paarweise Vergleiche notwendig, um das Object Matching zwischen allen möglichen Entitätenpaaren abzuschließen. Bei Dedoop wird dieses Problem durch ein Cluster-Verfahren eingegangen. Die Lösung liegt darin, die unnötigen Vergleiche frühzeitig zu identifizieren und aus dem weiteren Verlauf wegzulassen, um die Anzahl der durchzuführenden Berechnungen zu minimieren. Ein Object Matching-Workflow besteht aus zwei Schritten: Blocking und Matching. Im Blocking-Schritt werden die zu vergleichenden Daten semantisch in Blöcke zerlegt. Der paarweise Vergleich der Entitäten wird nur auf die Entitäten eingeschränkt, die sich im gleichen Block befinden. Die Aufteilung in Blöcke geschieht mit Hilfe von Blocking-Schlüsseln, die sich aus den Werten eines oder mehrerer Attribute von Entitäten zusammensetzen. Im Matching-Schritt stellt man alle möglichen Entitätenpaare innerhalb jedes Blocks auf und quantifiziert anhand eines Algorithmus den Ähnlichkeitsgrad zwischen ihnen. Für solche Vorgehensweise bietet sich auch an, mehrere Algorithmen heranzuziehen, um ihre Ergebnisse zu kombinieren. Wenn man das Beispielproblem, welches sich beim Zusammenführen von zwei großen Datenquellen mit Personeninformationen ergibt, nun erneut betrachtet, sieht man eventuell eine Lösung durch das Cluster-Verfahrens. Das Hindernis soll die unterschiedliche Schreibweise der Namen sein (z.B. Reihenfolge des Vor- und Nachnamens, Satzzeichen, zweiter Vorname). Wenn man davon ausgeht, dass in den beiden Quellen die Informationen zum Geburtsdatum und -ort vorhanden sind und sie ein gleiches Format aufweisen, kann man die beiden Attribute kombinieren, um sie als Blocking-Schlüssel zu verwenden. Nachdem man die Kandidaten eingrenzte, kann ein Match-Verfahren alle Namen, die in einem Block vorkommen, überprüfen, ob sie zur selben Person gehören.

Das Object Matching-Workflow ist relativ einfach durch das MapReduce-Modell realisierbar. Die Umsetzung von Blocking erfolgt in der Map-Funktion und die Umsetzung von Matching erfolgt in der Reduce-Funktion. Die Map-Funktion berechnet den Blocking-Schlüssel zu jeder Entität. Die Ausgabe von Map ist somit ein Paar, bestehend aus dem generierten Schlüssel und dem Wert, der die gesamte Entität umfasst. Hadoop gruppiert die Entitäten mit demselben Schlüssel zu einem Block. In der Reduce-Phase werden die Entitäten innerhalb eines Blocks paarweise verglichen. Falls der berechnete Ähnlichkeitsgrad den angegebenen Schwellenwert erreicht, wird das Entitätenpaar als eine Übereinstimmung klassifiziert und in die Ergebnismenge aufgenommen. Neben der vollständigen Übernahme eines Attributwertes, sind auch weitere Verfahren, wie Präfix, Suffix und Soundex, implementiert. Die Soundex-Funktion

erkennt ähnlich klingelnde Wörter, indem es ein Kodierungsschema für die Buchstaben benutzt. Zum Beispiel werden B, P, F und V als ein gleiches Zeichen kodiert. Um den Ähnlichkeitsgrad zu bestimmen, stehen dafür ebenfalls mehrere Algorithmen, wie Levenshtein und N-Gramme, bereit. Die Levenshtein-Distanz (auch Editierdistanz) zwischen zwei Zeichenketten ist die minimale Anzahl von Einfüge-, Lösch- und Ersetz-Operationen, um die erste Zeichenkette in die zweite umzuwandeln. Mit der Bildung von Fragmenten eingegebener Größe (N-Gramme) aus einer Zeichenkette, kann man die Anzahl der gemeinsamen Vorkommen dieser N-Gramme zwischen zwei Attributwerten zählen und somit die Ähnlichkeit dazwischen messen.

Für das Object Matching kann man entweder eine Quelle (Single-Source) oder zwei Quellen (Dual-Source) als Eingabe benutzen. Bei der Single-Source werden alle Entitäten miteinander verglichen. Dabei ist sichergestellt, dass keine Entität sich selbst matcht. In einem Block mit $N$ Entitäten sind $1/2 \cdot (N^2 - N)$ Vergleiche durchzuführen. Bei der Dual-Source vergleicht man jede Entität aus der ersten Quelle (Domain) mit jeder Entität aus der zweiten Quelle (Range). Hier geht man davon aus, dass alle Entitäten innerhalb der jeweiligen Quelle paarweise verschieden sind und die Duplikate nur in der gegenüberstehenden Quelle zu finden sind. In einem Block mit $M$ Entitäten aus Domain und mit $N$ Entitäten aus Source sind $M \cdot N$ Vergleiche durchzuführen.

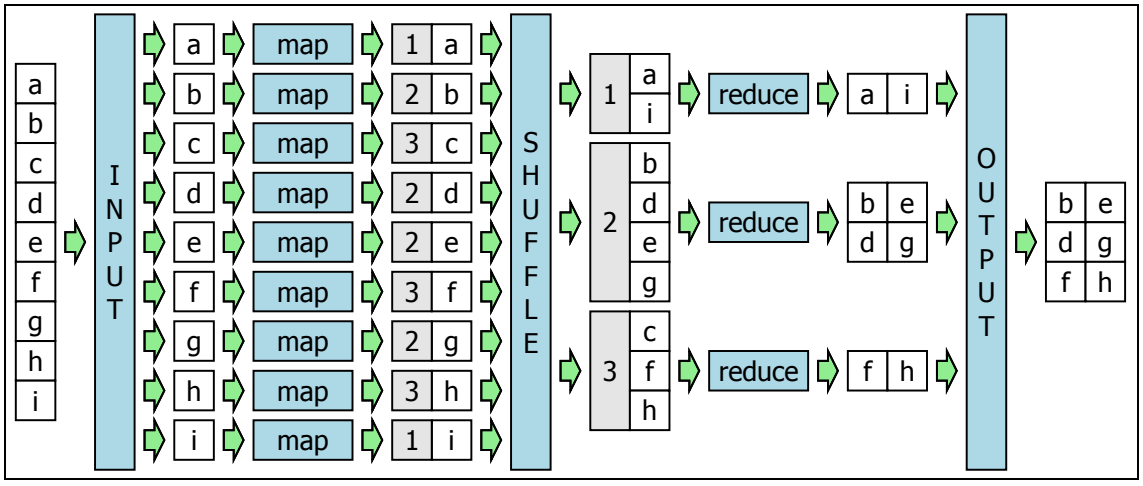


**Abbildung 2.4-1.** Visualisierung des Datenflusses bei einem Single-Source-Matching. Als Beispiel besteht die Eingabe aus 9 Zeilen. Am Ende der Map-Phase werden 3 verschiedene Blocking-Schlüssel erzeugt. Zu vergleichen ergeben sich folgende Entitätenpaare: $\{a,i\},\{b,d\},\{b,e\},\{b,g\}\{d,e\},\{d,g\},\{e,g\},\{c,f\},\{c,h\},\{f,h\}$. Durch das Blocking wird die Anzahl der Vergleiche von 36 $(1/2 \cdot (9 \cdot 8))$ auf 10 $(1/2 \cdot (2 \cdot 1 + 4 \cdot 3 + 3 \cdot 2))$ reduziert.

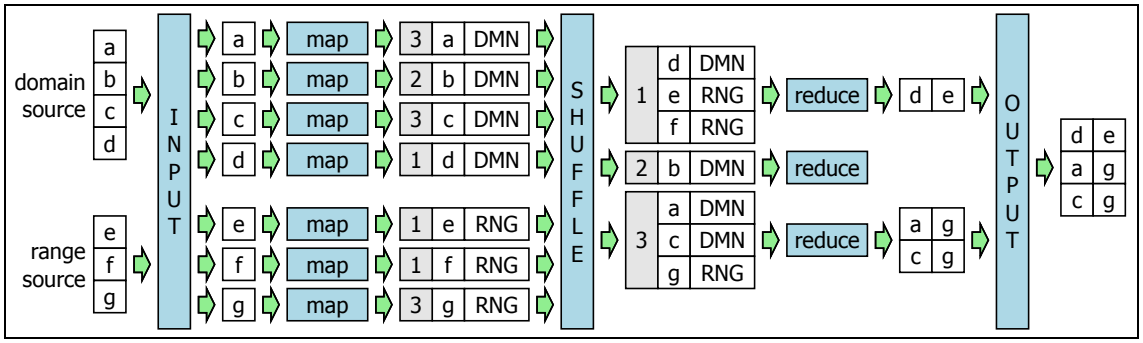


**Abbildung 2.4-2.** Visualisierung des Datenflusses bei einem Dual-Source-Matching. Als Beispiel besteht die Domain-Source aus 4 Zeilen und Range-Source aus 3 Zeilen. Am Ende der Map-Phase werden 3 verschiedene Blocking-Schlüssel erzeugt. Zu vergleichen ergeben sich folgende Entitätenpaare: $\{d, e\}, \{d, f\}, \{a, g\}, \{c, g\}$. Durch das Blocking wird die Anzahl der Vergleiche von 12 $(4 \cdot 3)$ auf 4 $(1 \cdot 2 + 1 \cdot 0 + 2 \cdot 1)$ reduziert.

Was das `InputFormat` und das `OutputFormat` bei den implementieren Jobs in Dedoop betrifft, so einigte man sich auf ein Standard, in welcher Form die Daten vorliegen müssen. Dedoop erwartet die Eingabedaten im CSV-Format – die Zeilen in einer Textdatei sind Datensätze und die Datenfelder innerhalb einer Zeile sind durch ein Komma getrennt [9]. Somit können Tabellen in CSV-Dateien abgebildet werden. Eine Spalte muss immer den Identifikator (ID) darstellen. Auf diese Weise kann die ID bei einer MapReduce-Berechnung als Schlüssel benutzt werden, um den Eintrag aus der Datei eindeutig zu identifizieren. Die Ergebnismenge besteht ebenfalls aus CSV-Dateien. In den ersten zwei Spalten sind die IDs der beiden Entitäten, die miteinander verglichen wurden. In der dritten Spalte ist das berechnete Ähnlichkeitsmaß. Für das Parsen der CSV-Dateien ist im Kernel von Dedoop die Bibliothek opencsv importiert.

Bei der Dedoop-Installationen ist eine allumfassende Bedienungsanwendung integriert (siehe Abbildung 2.4-3). Es ist eine grafische JavaScript-Anwendung, die man unter einer generierten URL im Webbrowser erreichen kann. Sie wurde mit Hilfe von GTW entwickelt und ist deshalb leicht mit den neu dazukommenden Funktionalitäten von Dedoop erweiterbar. Die Benutzerschnittstelle bietet den vollen Zugang zu sämtlichen Funktionen von Dedoop und macht es einfach mit den komplexeren Features des Frameworks zu experimentieren. Mit einfachen Mausklicks lassen sich die unterschiedlichen Object Matching-Workflows definieren und gegen einen Hadoop-Cluster ausführen. Dedoop wandelt die eingebenden Konfigurationsparameter automatisch in eine ausführbare MapReduce-Workflow um. Des Weiteren kann ein Benutzer auf das Dateisystem von HDFS über eine Baum-Ansicht zugreifen und die üblichen Dateioperationen ausführen, unter anderem Upload, Download, Löschen und Verschieben von Dateien und Verzeichnissen. In einem Manager lässt sich der Inhalt von Textdateien, die ein CSV-Format aufweisen, betrachten. Mit dem Zugriff auf das

Dateisystem kann man die Quellen und das Ziel für ein Object Matching schnell auswählen. Die Wertparameter sind über die Standardkomponenten von GWT festlegbar. Nach dem Abschicken des Workflows wird der Fortschritt der einzelnen MapReduce-Jobs in der Webanwendung visualisiert.

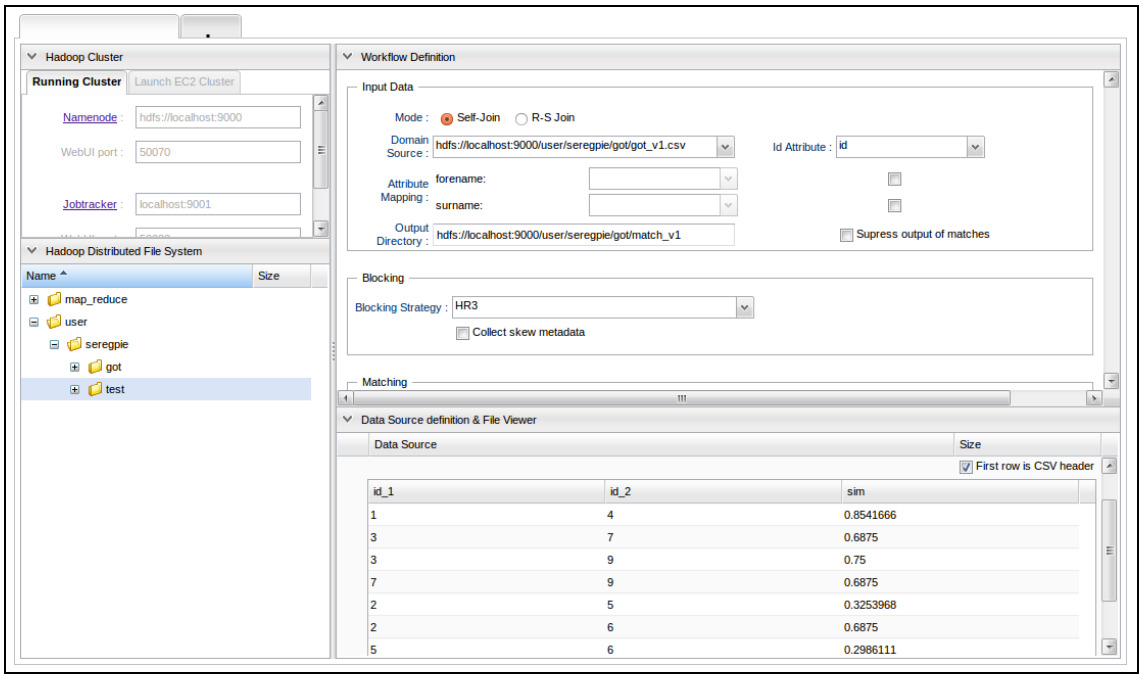


**Abbildung 2.4-3.** Screenshot der Weboberfläche von Dedoop.

# 3 Implementierung

Der gewählte Ansatz zur Lösung des gestellten Problems zur Wiederverwendung berechneter Matchergebnisse für MapReduce-basiertes Object Matching kann in zwei Schritte unterteilt werden. Im ersten Schritt stellt man fest, welche Änderungen zwischen der alten und der neuen Eingabequelle aufgetreten sind. Im zweiten Schritt sind mehrere Aufgaben zu bewerkstelligen, die zwar unabhängig voneinander sind, jedoch demselben Zweck dienen, die alten Matchergebnisse auf den neuesten Stand zu bringen. Unter der Nutzung der im ersten Schritt gewonnen Informationen wiederholt man die Match-Workflows für die hinzugekommenen Datenteile und beseitigt aus den alten Ausgabedaten die nicht mehr brauchbaren Matchergebnisse. Des Weiteren sind alle einzugebenden Parameter für die schnelle Umsetzung der beiden in einer Workflow zusammengefassten Schritte als ein Dialogfenster in die vorhandene Weboberfläche von Dedoop integriert.

## 3.1 Vordefinition

Zum Starten eines Workflows werden mehrere Pfade benötigt, deren Anzahl von der Auswahl der zu bearbeitenden Quellen im vorherigen Object Matching abhängt. Um das Dialogfenster aufzurufen, soll man die benötigten Pfade in der HDFS-Baumansicht markieren und auf den entsprechenden Knopf im Popupmenü klicken. Im erscheinenden Fenster sind die gewählten Pfade unter den Listenfeldern zu verteilen. Als Erstes gibt man die Pfade zur ursprünglichen Quellen noch einmal an – eine Quelle beim Single-Source und zwei Quellen beim Dual-Source. Ergänzend erfordert jede dieser Quellen ihre modifizierte Version, um Änderungen zwischen den beiden festzustellen. Bei Dual-Source kann man auch eine der beiden alten Versionen der Quelle weglassen, falls keine Änderungen vorliegen. Weitere Daten, die man aus dem früheren Object Matching braucht, sind seine Matchergebnisse und die dafür eingesetzten Konfigurationsparameter. Damit man sich an die alten Parameter nicht zu erinnern braucht und sie nicht noch einmal per Hand eingeben muss, erwartet ein Eingabefeld eine Datei, wo diese Parameter gespeichert sind. Nach der Auswahl der Datei wird deren Inhalt geladen und für die Wiederverwendung vorbereitet. Beim erfolgreichen Abschließen jeder Workflow serialisiert Dedoop das Konfigurationsobjekt in eine Datei unter demselben Pfad, welcher das Arbeitsverzeichnis des Workflows repräsentierte. Schließlich bleibt nur die Angabe des Speicherortes, wohin die auszuführenden Jobs ihre Ergebnisse ablegen. Das Absenden des Workflows bleibt unzulässig, bis alle Angaben gemacht sind und das gültige Konfigurationsobjekt erfolgreich deserialisiert ist.

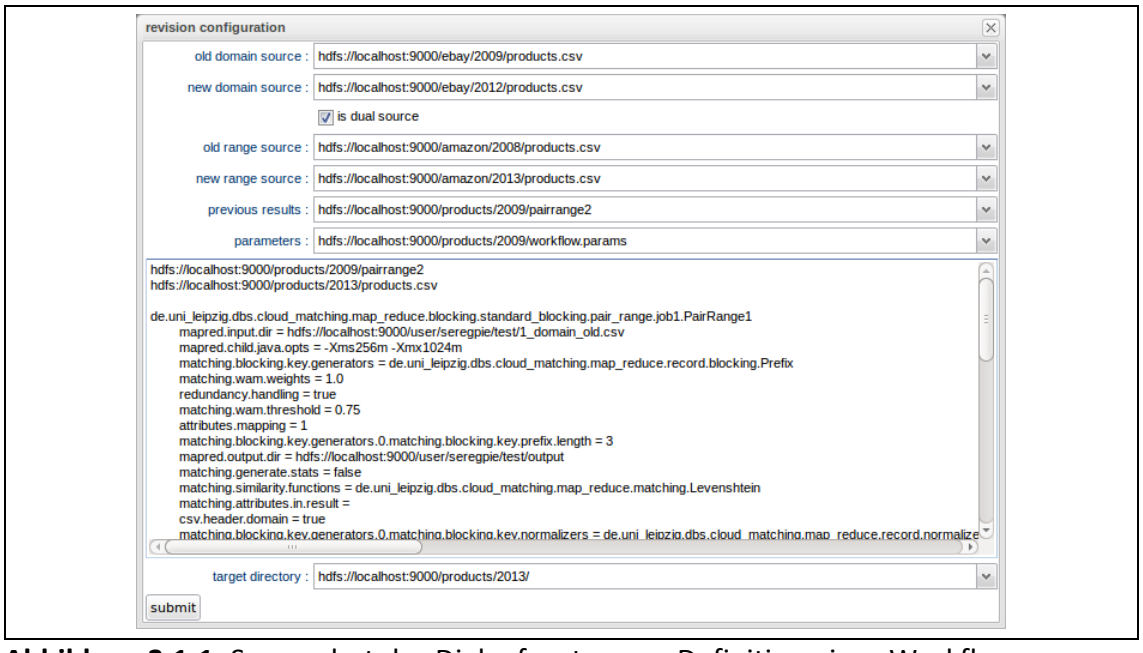


**Abbildung 3.1-1.** Screenshot des Dialogfensters zur Definition eines Workflows.

## 3.2 Auffindung von Änderungen

Um festzulegen welche CSV-Zeilen sich geändert haben und somit welche Einträge eine Aktualisierung benötigen, muss die alte Quelle gegen die neue vergleichen werden. Diese Aufgabe wird von einem einzigen MapReduce-Job übernommen. Es beinhaltet zwei Implementierungen des Mappers und eine des Reducers. Der Job nimmt als Eingabe zwei Quellen und erstellt als Ausgabe drei Ordner: `rmn`, `dlt` und `add`. Im Ordner `rmn` sind die unverändert gebliebenen Entitäten. Im Ordner `dlt` sind die alten, nicht mehr aktuellen Entitäten. Im Ordner `add` sind die neuen, hinzugekommenen Entitäten. Bei allen drei Datenmengen wird die ursprüngliche Kopfzeile miteingespeichert, falls sie vorher vorhanden war. Von den gelöschten Entitäten werden nur die Werte aus der Spalte, die als ID gekennzeichnet wurde, in die Ausgabe geschrieben, weil die restlichen Attribute für spätere Aufgaben unwichtig sind. Für die Speicherung der gelieferten Ergebnisse, werden statt einem `OutputFormat` drei unterschiedlichen benutzt. Alle drei `OutputFormats` erben von der Klasse `MultipleTextOutputFormat` und überschreiben ihre Methode `generateFileNameForKeyValue` für die Generierung des Namens der Ausgabedatei. Dieser Methode wird dadurch erweitert, dass sie lediglich den Ordnernamen mit einem Schrägstrich vor dem generierten Dateinamen eingefügt.

Folgende Parameter werden für die Job-Konfiguration benötigt:

- `mapred.input.dir.old`
  Das Verzeichnis mit der alten Version der Daten im CSV-Format.
- `mapred.input.dir.new`
  Das Verzeichnis mit der neuen Version der Daten im CSV-Format.
- `attribute.id`
  Der Index des ID-Attributen. Der Parameter muss für die beiden Quellen gültig sein.
- `attributes.comparing`
  Die Liste der Indizien der Attribute, deren Werte relevant für das Object Matching sind. Der Parameter muss für die beiden Quellen gültig sein.
- `csv.header`
  Der Wahrheitswert, ob der Kopfdatensatz vorhanden ist. Der Parameter muss für die beiden Quellen gültig sein.
- `mapred.output.dir`
  Das Ausgabeverzeichnis des Jobs.

Zwei Mapper bearbeiten die Eingabedaten. Ein Mapper nimmt die Zeilen aus der alten Quelle und markiert sie als „alt“ und der andere Mapper nimmt die Zeilen aus der neuen Quelle und markiert sie als „neu“. Die beiden Mapper erben von derselben Klasse und führen die weiteren Aufgaben identisch aus. Der Ausgabeschlüssel des Mappers ist ein

Tupel bestehend aus zwei Werten. Der erste Wert ist die ID des Datensatzes. Die ID wird aus der Zeile mit Hilfe des CSV-Parsers extrahiert. Der zweite Wert des Ausgabeschlüssels ist der Hashwert des Datensatzes. Allerdings werden nicht alle Attribute einer Zeile für die Generierung des Hash-Codes verwendet, da einige irrelevant für das Matchergebnis sein können und somit in keinen weiteren Jobs berücksichtigt werden. Der CSV-Parser legt die notwendigen Attribute als ein Array ab. Der Hash-Code ergibt sich aus der standartmäßigen Hashfunktion, die von der Java-Bibliothek zur Verfügung gestellt wird. `Arrays.hashCode(Object[] a)` gibt den Hash-Code basierend auf die Inhalte des angegebenen Arrays [25]. Der Kopfdatensatz, falls vorhanden, wird im Mapper sofort durch das `MultipleOutputs`-Objekt in die Endergebnismenge geschrieben und bei weiteren Operationen übersprungen. Das `OutputCollector` für die Ausgabe der gelöschten Entitäten bekommt nur den Teil des Kopfdatensatzes, den Namen der Spalte, die die ID kennzeichnet.

```
private final static NullWritable nil = NullWritable.get();

public void map(LongWritable key, Text value,
    OutputCollector<TextIntPair, IntTextPair> collector, Reporter reporter)
    throws IOException {
  // check if current record is header
  if (<RECORD IS HEADER>) {
    if (<RECORD COMES FROM OLD SOURCE>) {
      mos.getCollector("rmn", reporter).collect(nil, value);
      // extract id attribute
      mos.getCollector("dlt", reporter).collect(nil, new Text(<ID>));
      mos.getCollector("add", reporter).collect(nil, value);
    }
  } else {
    // extract id attribute
    // extract relevant attributes and make hash sum
    TextIntPair outputKey = new TextIntPair(<ID>, <HASH>);
    // current label depends of input source (old or new)
    IntTextPair outputValue = new IntTextPair(<LABEL>, value);
    collector.collect(outputKey, outputValue);
  }
}
```

**Code 3.2-1**: Implementation des Mappers für die Auffindung von Änderungen.

Man vergleicht die von den beiden Mappern gelieferten Ergebnisse nach dem erzeugten Schlüssel-Paar miteinander; zuerst nach dem ersten Wert, der die ID der Zeile repräsentiert und dann, falls die IDs übereinstimmen, nach dem zweiten Wert, der den Hash-Code der relevanten Attribute beinhaltet. Bei den übereinstimmenden IDs heißt es, dass der dazugehörige Eintrag in beiden Quellen – in der alten und in der neuen

Version – weiterhin vorhanden ist. Sind aber die Hash-Werte verschieden, so deutet das darauf hin, dass im gleichen Eintrag aus der neuen Quelle ein oder mehrere relevanten Attribute sich geändert haben. Kurz gesagt, zwei Einträge bilden nur dann eine Gruppe, wenn sie für die jeweilige Matching-Konfiguration als identisch anzusehen sind. Bekommt der Reducer ein aus zwei Einträgen bestehende Gruppe, dann ist der Eintrag aus der neuen Version unverändert geblieben. Der Reducer behält den alten Eintrag und schreibt ihn in den Ordner `rmn`. Kriegt der Reducer nur einen einzigen Eintrag zu einem Schlüssel, so heißt es, dass hier eine Änderung stattfand. Durch die in den Mappern gesetzten Markierung, kann der Reducer unterscheiden, vom welchen Mapper der Eintrag kommt und somit aus welcher Quelle diese Zeile gelesen wurde. Stammt die eingehende CSV-Zeile aus der alten Quelle, so bedeutet es, dass sie in der neuen nicht mehr zu finden ist oder die relevanten Attribute mit den alten nicht mehr übereinstimmen. Der ID-Wert dieses Eintrags wird im Ordner `dlt` gespeichert, um ihn später aus den vorherigen Matchergebnissen, die diese IDs beinhalten, zu entfernen. Stammt die eingehende CSV-Zeile aus der neuen Quelle, so bedeutet es, dass sie neueingefügt wurde oder — wie im vorhergehenden Fall — beim Vergleich mit der alten Zeile, die dieselbe ID aufweist, eine Änderung der relevanten Attribute festgesellt wurde. Der Eintrag wird vollständig in den Ordner `add` gespeichert. Die geänderten CSV-Zeilen werden demzufolge zweimal ausgegeben; einmal als „gelöscht“ und einmal als „neueingefügt“. Eine neue Kategorie ist nicht notwendig, da man solche Zeilen genauso wie die eingefügten und die gelöschten Zeilen in den späteren Schritten behandelt.

```
private final static NullWritable nil = NullWritable.get();

public void reduce(TextIntPair key, Iterator<IntTextPair> values,
    OutputCollector<NullWritable, Text> collector, Reporter reporter)
    throws IOException {
  String oldValue = null;
  String newValue = null;
  // extract old value from values if available
  // extract new value from values if available
  if (oldValue != null && newValue != null)
    mos.getCollector("rmn", reporter).collect(nil, new Text(oldValue));
  else if (oldValue != null && newValue == null)
    mos.getCollector("dlt", reporter).collect(nil, key.getFirst());
  else if (oldValue == null && newValue != null)
    mos.getCollector("add", reporter).collect(nil, new Text(newValue));
}
```

**Code 3.2-2**. Implementation des Reducers für die Auffindung von Änderungen.

Der Job für die Auffindung von Änderungen wird bei Single-Source immer einmal ausgeführt. Beim Dual-Source muss für mindestens eine der Quellen – Domain oder Range – ihre neue Version vorliegen, damit der komplette Workflow erfolgreich

vollzogen werden kann. Dabei sind eventuell zwei Abläufe notwendig; ein Job für Domain-Source oder ein Job für Range-Source oder zwei Jobs für die beiden Eingangsquellen.

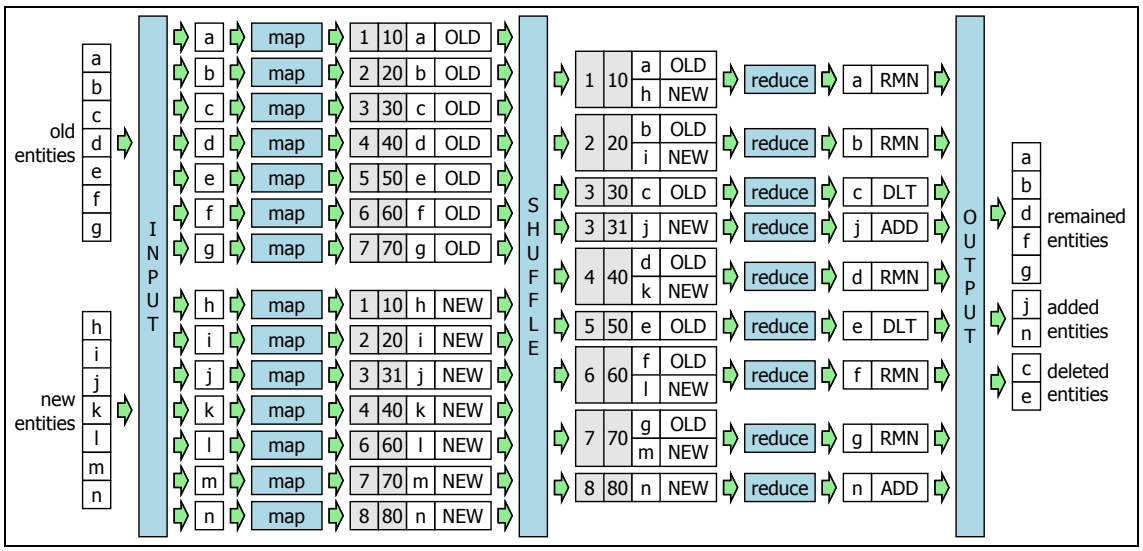


**Abbildung 3.2-3.** Visualisierung des Datenflusses beim Finden von Änderungen zwischen zwei CSV-Dateien. Um alle Fälle abzudecken ist ein Eintrag gelöscht, ein Eintrag ist eingefügt und ein Eintrag ist geändert.

## 3.3 Reduktion von Matchergebnissen

Nachdem die Änderungen zwischen den neuen und den alten Inhalten gefunden sind, kann man die alten Matchergebnisse heranziehen, um die nicht mehr aktuellen Matches daraus zu löschen. Solche Matches sind die, wo mindestens eine Entität aus dem Entitätenpaar nach dem Finden der Änderungen als ein gelöschter Eintrag markiert ist. Zu diesem Zweck ist eine Klasse vorbereitet, die zwei Mapper und einen Reducer zur Verfügung stellt. Die Klasse ist eher allgemein gestaltet und ist nicht gezielt auf die Eingabe von Matchergebnissen und die zu löschenden IDs spezifiziert. Wie der Name schon vermuten lässt, wird mit Hilfe dieses Jobs die Differenz von zwei Datenmengen ermittelt. Als Eingabe erwartet der Job zwei Quellen: Minuend und Subtrahend. Die beiden ausgewählten Namen entsprechen der vertrauten Definition aus der relationalen Algebra: der Minuend ist die Menge, von der abgezogen wird, und der Subtrahend ist die Menge, die abgezogen wird. Dieser Implementation unterscheidet allerdings von der relationalen Algebra. Es wird nicht nach der ganzen Zeile nachgeschaut, sondern nach einem gewählten Attribut. Die Ausgabe des Jobs ist somit eine reduzierte Minuend-Menge. Ein Eintrag aus der Ursprungsmenge von Minuend

wird dann gelöscht, wenn er das gleiche gewählte Attribut besitzt, wie ein Eintrag aus der Subtrahend-Menge.

Folgende Parameter werden für die Job-Konfiguration benötigt:

- `mapred.input.dir.minuend`
  Das Verzeichnis mit den Minuend-Daten im CSV-Format.
- `mapred.input.dir.subtrahend`
  Das Verzeichnis mit den Subtrahend-Daten im CSV-Format.
- `attribute.join.minuend`
  Der Index des Attributs in den Minuend-Daten, dessen Wert als Join-Attribut benutzt wird.
- `attribute.join.subtrahend`
  Der Index des Attributs in den Subtrahend-Daten, dessen Wert als Join-Attribut benutzt wird.
- `csv.header.minuend`
  Der Wahrheitswert, ob der Kopfdatensatz in der Minuend-Quelle vorhanden ist.
- `csv.header.subtrahend`
  Der Wahrheitswert, ob der Kopfdatensatz in der Subtrahend-Quelle vorhanden ist.
- `mapred.output.dir`
  Das Ausgabeverzeichnis des Jobs.

Zwei Mapper bearbeiten die Eingabequellen. Ein Mapper nimmt die Zeilen aus der einen Quelle und markiert sie als „Minuend“ und der andere Mapper nimmt die Zeilen aus der anderen Quelle und markiert sie als „Subtrahend“. Die beiden Mapper erben von derselben Klasse und führen die weiteren Aufgaben identisch aus. Der Ausgabe-Schlüssel der Mapper ist das gewählte Attribut, der durch den CSV-Parser extrahiert wird. Der Index des Attributes kann zwischen den beiden Quellen variieren; der Parameter für jede Quelle wird separat angegeben. Der Ausgabewert des Mappers ist der gleiche Eingabewert bloß mit einer Markierung, um zu erkennen aus welcher Quelle der Eintrag stammt. Da der Job nur Dateien im CSV-Format versteht, ist ein Mechanismus implementiert, um die CSV-Kopfzeile – falls solche vorhanden ist – individuell zu behandeln; die Kopfzeile von Minuend wird direkt in das endgültige Ergebnis übernommen und die Kopfzeile von Subtrahend wird übersprungen. Den Wahrheitswert dafür lässt sich auch separat für jede Quelle eingeben.

```
private final static NullWritable nil = NullWritable.get();
public void map(LongWritable key, Text value,
    OutputCollector<Text, IntTextPair> collector, Reporter reporter)
    throws IOException {
  // check if current record is header
  if (<RECORD IS HEADER>) {
    if (<RECORD COMES FROM MINUEND SOURCE>)
      mos.getCollector("difference", reporter).collect(nil, value);
  } else {
    // extract join attribute
    Text outputKey = new Text(<JOIN ATTRIBUTE>);
    // current label depends of input source (minuend or subtrahend)
    IntTextPair outputValue = new IntTextPair(<LABEL>, value);
    collector.collect(outputKey, outputValue);
  }
}
```

**Code 3.3-1**. Implementation des Mappers für die Reduktion von Matchergebnissen.

Innerhalb der Reduce-Methode läuft man über die einzelnen Werte und speichert sie in eine Liste ab. Sobald man auf einen Wert stößt, der aus der Subtrahend-Quelle kommt, heißt es, dass die aktuellen Einträge, die mit dem Attribut aus dieser Quelle assoziiert sind, nicht erwünscht sind. In diesem Fall wird die Funktion einfach komplett übersprungen. Wenn es nicht passiert und alle Einträge aus der Minuend-Quelle stammen, wird die Schleife irgendwann verlassen und die zwischengespeicherten Werte in der Liste werden erneut iteriert, um in die finale Ausgabe geschrieben zu werden.

```
private final static NullWritable nil = NullWritable.get();
public void reduce(Text key, Iterator<IntTextPair> values,
    OutputCollector<NullWritable, Text> collector, Reporter reporter)
    throws IOException {
  List<Text> outputValues = new ArrayList<Text>();
  while (values.hasNext()) {   // iterate over all values
    IntTextPair value = values.next();
    if (<RECORD COMES FROM SUBTRAHEND SOURCE>) {
      return; // stop the execution of method
    } else {
      // add value to list
      outputValues.add(new Text(value.getSecond().toString()));
    }
  }
  // iterate over all in the list and sent them to output
  for (Text outputValue : outputValues)
    mos.getCollector("output", reporter).collect(nil, outputValue);
}
```

**Code 3.3-2**. Implementation des Reducers für die Reduktion von Matchergebnissen.

Die Ausführung des Jobs hängt davon ab, um welche Eingabedatei es sich handelt. Im Falle, dass die Eingabe eine Single-Source ist, gehören die zwei ID-Spalten aus den Matchergebnissen alle zu dieser Quelle. Das Join-Attribut von Subtrahend ist der Index des ID-Attributes der Quelle. Der Job wird zwei Mal gestartet. Beim ersten Start ist das Join-Attribut von Minuend die erste Spalte. Beim wiederholten Ausführen des Jobs werden nur zwei Konfigurationsparameter gewechselt; als Minuend dient die Ausgabe des ersten Jobs und das Join-Attribut von Minuend ist die nun zweite Spalte. Die ID einer Entität kann sowohl in der ersten als auch in der zweiten Spalte zu finden sein, deshalb muss man über die Ergebnisse zwei Mal laufen. Ist die Eingabe eine Dual-Source, werden auch zwei Jobs benötig. Im ersten Lauf werden alle Matchergebnisse entfernt, die mit einer Entität aus der Domain-Source assoziiert sind. Dabei ist das Join-Attribut von Minuend die erste Spalte und das Join-Attribut von Subtrahend ist die ID von der Domain-Quelle. Im zweiten Lauf werden alle Matchergebnisse entfernt, die mit einer Entität aus der Range-Source assoziiert sind. Man nimmt wieder die Ausgabe vom ersten Job als Minuend. Hier sind schon alle Zeilen gelöscht, die eine ID eines geänderten Eintrags aus der Domain-Quelle beinhalteten. Das Join-Attribut von Minuend ist jetzt die zweite Spalte und das Join-Attribut von Subtrahend ist die ID der Range-Source.

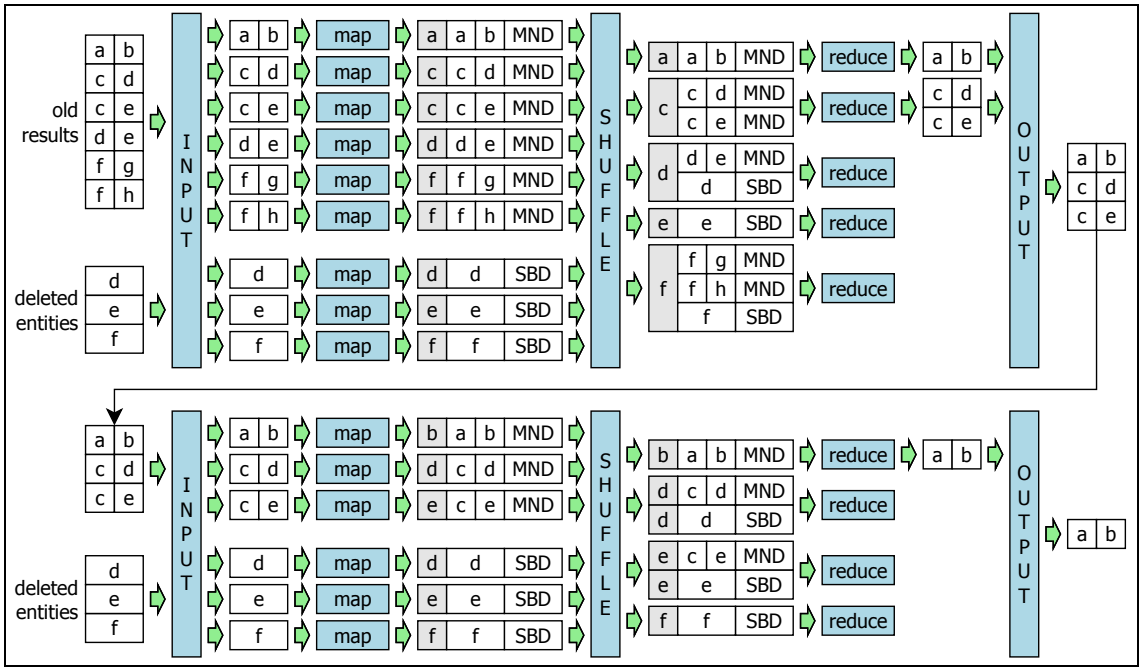


**Abbildung 3.3-3.** Visualisierung des Datenflusses beim Löschen von nicht mehr aktuellen Matchergebnissen. Die Entitäten $d$, $e$ und $f$ wurden in der Ursprungsquelle entweder gelöscht oder geändert. Der erste Ablauf entfernt 3 Zeilen aus den alten Ergebnissen und der zweite Ablauf entfernt weitere 2 Zeilen.

## 3.4 Erweiterung von Matchergebnissen

Für den weiteren Ablauf werden die Match-Aufgaben mit den gleichen Konfigurationen aus den vorherigen resultierten Ergebnissen wiederholt. Dabei beschränkt man sich nur auf eine Teilmenge der neuen Datensätze, die der erste Schritt erzeugt. Bevor man die neuen Einträge zum Matching weiterleitet, ist zwischen der Single-Source und der Dual-Source zu unterscheiden. Wurde bei der Eingabe eine einzelne Quelle benutzt, so sind zwei Matching-Aufgaben durchzuführen. Man führt ein Matching der eingefügten Entitäten gegen die unveränderten und ein Matching zwischen den eingefügten Entitäten untereinander durch. Bei der Dual-Source gibt es noch drei Fälle. Liegt nur bei einer Quelle eine Änderung vor, ist nur ein Matching notwendig. Falls die Domain-Source neu ist, matcht man die eingefügten Entitäten aus der Domain-Source gegen alle Entitäten aus der Range-Source. Falls die Range-Source neu ist, matcht man die eingefügten Entitäten aus der Range-Source gegen alle Entitäten aus der Domain-Source. Falls aber die beiden Quellen neu sind, sind die Matchings aus den vorhergehenden zwei Fällen beide zu verwirklichen, darüber hinaus mit dem Zusatz, dass die eingefügten Entitäten aus der Domain-Source und die eingefügten Entitäten aus der Range-Source gegeneinander zu matchen sind. In allen Fällen benutzt man also alle Entitäten, die der erste Schritt als neu eingefügt erkannte, und matcht sie gegen die verbliebenden. Das Matching der verbliebenden Entitäten untereinander ist unnötig, weil es die gleiche Ergebnismenge erzeugen würde, die schon nach dem Löschen der nicht mehr aktuellen Vergleiche aus der alten Ergebnismenge vorliegt.

# 4 Hindernisse während der Entwicklung

Bei der Erweiterung von Dedoop ergaben sich einige Probleme, die die Entwicklung behinderten. Ihre Lösung hat zum Teil eine tiefe Änderung des ursprünglichen Quelltextes bewirkt, anstatt nur dessen Ausbau. Das frühere Modell von Dedoop hinterließ keine Informationen über die durchgeführten Workflows. Die benutzten Konfigurationsparameter sind jedoch essentiell, um das Object Matching unter den alten Bedienungen zu wiederholten. Im Kern von Dedoop wurde die Klasse, die asynchrone Methoden für das Arbeiten mit dem HDFS bereitstellt, um neue Methoden zu diesem Zweck erweitert. Es wurden zwei neue Methoden implementiert. Eine Methode speichert die Konfigurationsparameter unter dem gegebenen Pfad. Die andere Methode liest die Konfigurationsparameter aus dem gegebenen Pfad. Dafür wurde die Klasse, die als Übermittlungseinheit der Konfigurationsparameter von der Weboberfläche an den Server benutzt wurde, serialisierbar gemacht. Außerdem wurde

diese Klasse noch komplett überarbeitet. Am Anfang lag `LinkedHashMap` als die grundlegende Datenstruktur der Klasse dar. Die Schlüssel waren die Klassennamen des Jobs und der Wert war eine weitere Map mit den Konfigurationsparametern für diesen Job. Diese Implementationsweise eignete sich für den gewählten Ansatz nicht, denn die Map-Struktur lässt keine Duplikate als Schlüssel zu [26]. Man könnte keine Jobs mit demselben Klassennamen – auch wenn sie sich in ihrer Konfiguration komplett unterscheiden – in der gemeinsamen Workflow laufen lassen. Deswegen wurde `LinkedHashMap` durch eine Liste ersetzt, wo für die Elemente eine neue Klasse deklariert wurde, die zwei Variablen annimmt: den Klassennamen des Jobs und eine Map als Konfigurationsparameter.

Eine geplante Feature, die unimplementiert geblieben ist, ist das Zusammenführen von allen Matchergebnissen unter einen Pfad. Da jeder Job seine eigene Ausgabe im separaten Ordner erzeugt, ist es wünschenswert alle Ausgangsdaten zu einem Gesamtergebnis zu vereinen. Diese Aufgabe ist durch das einfache Verschieden der Dateien nicht lösbar, denn es liegen mehrere Quellen vor und in jeder Quelle eine CSV-Kopfzeile vorhanden ist. Ein MapReduce-Job eignet sich dafür genauso wenig. Da die Anzahl der Quellen variiert und die Reihenfolge der Dateien innerhalb ihres Verzeichnisses beliebig ist, können die Mapper auf keinem sicheren Weg entschieden, ob es sich um eine Kopfzeile oder um eine Entität handelt.

# 5 Auswertung

Die gegebene Lösung für das Problem funktioniert besser je kleiner die Anzahl der Änderungen ist. Bei 10% der Änderungen spart man 80% der Vergleiche, bei 25% spart man 55% und bei 50% spart man 25%. Unabhängig von den durch das Blocking-Verfahren erzeugten Blöcke bleibt das Verhältnis logarithmisch. Ab einer bestimmten Anzahl lohnt sich diese Abarbeitung der alte und der neuen Versionen nicht, da man die Laufzeit für die Auffindung von Änderungen und für das Löschen der nicht mehr aktuellen Ergebnisse dazurechnen soll. Wie man in der Abbildung 5-1 sieht, erreicht eine Kurve ihre Spitze dort, wo die Anzahl der neueingefügten Entitäten maximal ist und es keine ungeänderten Entitäten gibt. In diesem Punkt entspricht die Anzahl der Vergleiche der kompletten Neuberechnung. Weil die Initialisierung aller Jobs auch eine bestimmte Zeit in Anspruch nimmt, soll man das Object Matching mit der Wiederverwendung berechneter Matchergebnisse besonders bei wenigen Änderungen und bei großen Datenmengen dem herkömmlichen Ansatz vorziehen.

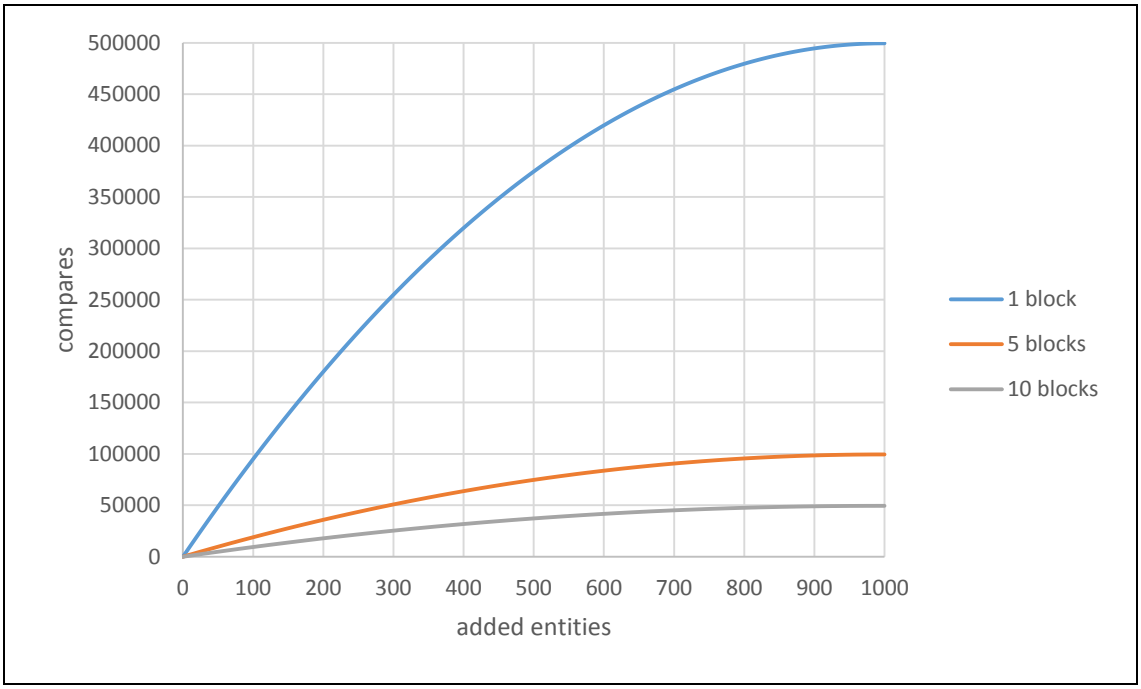


**Abbildung 5-1**. Grafik mit den benötigten Vergleichen unter verschiedenen Bedingungen für ein Object Matchin. Y-Achse zeigt die Anzahl der Vergleiche. X-Achse zeigt die Anzahl der neueingefügten Entitäten, die im linearen Verhältnis zu den ungeänderten Entitäten stehen. Das heißt die Anzahl der beiden Mengen bleibt bei jeder Koordinate konstant 1000. Für die drei gezeigten Kurven wird die folgende Formel benutzt:

$$B \cdot \left(\frac{N}{B} \cdot \frac{O}{B}\right) + B \cdot \left(\frac{(N/B)^2 - N/B}{2}\right)$$

Die Formal gilt nur für die Single-Source. $N$ ist die Anzahl der neueingefügten Entitäten. $O$ ist die Anzahl der ungeänderten Entitäten. Dabei gilt $O = 1000 - N$. $B$ ist die Anzahl der durch das Blocking erstellten Partitionen. Linke Seite repräsentiert die Anzahl der Vergleiche zwischen den neueingefügten und ungeänderten Entitäten. Rechte Seite repräsentiert die Anzahl der Vergleiche bei den neueingefügten Entitäten untereinander.

# 6 Kurzzusammenfassung

Die Bachelorarbeit umfasst die Erweiterung des Projektes Dedoop. Dedoop stellt eine Reihe von Werkzeugen zur Verfügung, die das Finden von Duplikaten durch Object Matching-Ansätze in einer Datenmenge automatisieren. Das Object Matching geschieht auf der MapReduce-Plattform Hadoop. Mit Hilfe der entwickelten Erweiterung, ist es

möglich das vollständige Neuberechnen an den Daten bei ihrer Änderung zu vermeiden. Das Verfahren geschieht in zwei Phasen. In der ersten Phase stellt man die Änderungen fest, die zwischen der alten Datenmenge und der neuen Datenmenge stattfanden. Die dabei gewonnenen Informationen werden in drei Kategorien unterteilt: Datensätze, die in der alten und in der neuen Datenmenge unverändert zu finden sind, Datensätze aus der neuen Quelle, die die Neuberechnung benötigen, und Datensätze aus der alten Quelle, die aus der Neuberechnung ausgeschlossen werden sollen. In der zweiten Phase wird das alte Object Matching, angewendet auf die aus der ersten Phase gewonnenen Teilmengen, wiederholt. Die für die Neuberechnung benötigten Datensätze sind die, die aktualisiert oder neueingefügt wurden. Deshalb liegen für sie noch keine Ergebnisse aus dem alten Object Matching vor. Diese Datensätze werden in der zweiten Phase gegeneinander und gegen die unverändert gebliebenen Datensätze gematcht. Die aus der Neuberechnung ausgeschlossen Datensätze sind die, die aktualisiert oder gelöscht wurden. Für sie liegen bereits Matchergebnisse vor, und deshalb müssen diese Ergebnisse von diesen Datensätzen bereinigt werden. Der Vorteil dieses Verfahrens liegt darin, dass man die unverändert gebliebenen Datensätze nicht noch einmal gegeneinander zu matchen braucht.

# Literaturverzeichnis


[1] Lars Kolb. Andreas Thor. Erhard Rahm. Dedoop: Efficient Deduplication with Hadoop. 2012.
[2] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplfied Data Processing on Large Clusters. OSDI'04. 2004.
[3] Tom White. Hadoop: The Definitive Guide, 2nd Edition. O'Reilly. 2010.
[4] Tutorial zu Hadoop MapReduce
http://hadoop.apache.org/docs/stable/mapred_tutorial.html
[5] Tutorial zu HDFS
http://hadoop.apache.org/docs/stable/hdfs_design.html
[6] Liste von MapReduce-basierten Anwendungen
http://www.dbms2.com/2008/08/26/known-applications-of-mapreduce/
[7] Liste von mit Hadoop arbeitenden Organisationen
http://wiki.apache.org/hadoop/PoweredBy
[8] Stellungnahme zur alten und neuen Hadoop-Java-API
http://mail-archive.com/mapreduce-dev@hadoop.apache.org/msg01833.html
[9] CSV-Spezifikation
http://www.creativyst.com/Doc/Articles/CSV/CSV01.htm
[10] Projekthomepage zu Qt
http://qt-project.org/
[11] Projektüberblick zu DryadLINQ
http://research.microsoft.com/en-us/projects/dryadlinq/
[12] Projekthomepage zu Skynet
http://skynet.rubyforge.org/
[13] Projekthomepage zu Hadoop
http://hadoop.apache.org/
[14] Quellcode für einfache MapReduce-Implementation in Erlang
http://github.com/tvcutsem/erlang-mapreduce
[15] Projekthomepage zu Disco
http://discoproject.org/
[16] Projektüberblick zu GWT
http://developers.google.com/web-toolkit/overview
[17] http://developers.google.com/web-toolkit/doc/latest/FAQ_GettingStarted
[18] http://developers.google.com/web-toolkit/doc/latest/RefJreEmulation
[19] http://developers.google.com/web-toolkit/doc/latest/DevGuideCompilingAndDebugging
[20] http://developers.google.com/web-toolkit/doc/latest/RefWidgetGallery
[21] http://developers.google.com/web-toolkit/doc/latest/DevGuideUiBrowser
[22] http://developers.google.com/web-toolkit/terms
[23] http://developers.google.com/web-toolkit/makinggwtbetter
[24] Artikel über SPA
http://net.tutsplus.com/tutorials/javascript-ajax/important-considerations-when-building-single-page-web-apps/
[25] http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html
[26] http://docs.oracle.com/javase/6/docs/api/java/util/LinkedHashMap.html

# Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.