ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

Konzeption und Entwicklung eines effizienten Prozess Mining Ansatzes auf Basis von MapReduce

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Corvin Frey
corvin.frey@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Klaus Kammerer

2015

Fassung 29. März 2016

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Um Geschäftsprozesse optimieren zu können, müssen diese erst einmal erhoben und definiert werden. Process Mining ist ein Technik, mit deren Hilfe sich Prozessmodelle aus gespeicherten Logdaten vorhandener Informationssysteme erheben lassen. Unternehmen bieten sich oft große Datenmengen an bereits vorhandener Logdaten für solche Prozessanalysen an. Ziel dieser Arbeit ist es ein Framework für große Logdaten zu entwickeln, welches Process Mining effizient auf Basis des MapReduce Programmierparadigmas durchführen kann. Hierfür werden zuerst Grundkonzepte eingeführt, die Entwicklung eines Heuristic Mining Algorithmus auf Basis von MapReduce beschrieben und dieser prototypisch im sogenannten ProDoop Framework implementiert.

Um die Leistungsfähigkeit des vorgestellten Ansatzes zu überprüfen, wird anschließend die Geschwindigkeit von ProDoop ermittelt. Das durchgeführte Experiment zeigt hierbei, dass der implementierte Heuristic Mining Algorithmus vor allem bei großen Datenmengen effizient angewendet werden kann. Abschließend werden weitere Möglichkeiten zur Effizienzsteigerung vorgestellt.

Inhaltsverzeichnis

1

Einleitung

In der heutigen, globalisierten Welt sind Unternehmen darum bestrebt, international erfolgreich und konkurrenzfähig agieren zu können. Die jeweiligen Geschäftsprozesse zu kennen und zu optimieren ist eine Möglichkeit. Oftmals sind Geschäftsprozesse nur implizit definiert und nicht formal beschrieben (z.B. mit Hilfe einer Geschäftsprozess-Modellierungssprache, wie BPMN [1]). Bevor Geschäftsprozesse angepasst und optimiert werden können, müssen diese deshalb erst einmal erhoben und definiert werden.

Viele Unternehmen besitzen bereits Informationssysteme, die einzelne Aktionen innerhalb eines implizit definierten Geschäftsprozesses dokumentieren. Die dabei entstehenden Logdaten können als Grundlage für die Erhebung von Prozessmodellen dienen. *Process Mining* nennt sich eine Technik, mit der man aus solchen Logdaten Prozessmodelle über den tatsächlichen Ablauf eines Geschäftsprozesses erstellen kann (siehe Abbildung 1.1 [2]). Des Weiteren lassen sich mit Process Mining z.B. soziale Beziehungen zwischen einzelnen Prozessbeteiligten ermitteln [3].

Informationssysteme in Unternehmen erzeugen und speichern bereits große Logdatenmengen, die sich mit klassischen Datenverarbeitungsmethoden nicht mehr effizient verarbeiten und auswerten lassen [4]. Aus diesem Grund werden Methoden benötigt, mit denen man Process Mining trotz großer Logdatenmengen effizient durchführen kann. *MapReduce* ist ein Programmiermodell, das speziell dafür entwickelt wurde, große Datenmengen effizient zu verarbeiten. Die Grundidee ist hierbei, die Bearbeitung von Daten auf einem verteilten System in mehreren parallelen Strömen zu verarbeiten.

Im Rahmen dieser Arbeit wird ein Algorithmus vorgestellt, mit dem sich Process Mining auf Basis von MapReduce effizient durchführen lässt. Dieser Algorithmus wurde mit Hilfe des Apache Hadoop Frameworks im “ProDoop”-Prototypen implementiert und anschließend evaluiert.

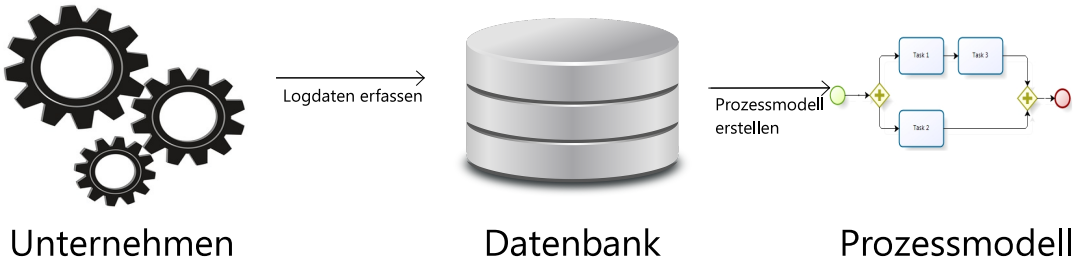

Abbildung 1.1: Grundlegende Funktionsweise von Process Mining

Kapitel 2 führt grundlegende Techniken und Methoden, wie Business Process Management, Process Mining und MapReduce ein. Kapitel 3 beschreibt die Funktionsweise von Apache Hadoop [5]. Kapitel 4 dokumentiert die technische Umsetzung des ProDoop Prototypen. Kapitel 5 beschreibt ein durchgeführtes Experiment, das die Rechengeschwindigkeit der technischen Implementierung des Prototypen untersucht. In Kapitel 6 werden die Ergebnisse des Experiments und die Bedeutung von MapReduce in Verbindung mit Process Mining diskutiert. Dabei wird auch ein Vergleich zu anderen Forschungsansätzen gezogen. Kapitel 7 fasst diese Arbeit zusammen.

2

Grundlagen

In diesem Kapitel werden grundlegende Konzepte vorgestellt, welche für die praktische Umsetzung von Process Mining und MapReduce benötigt werden. Im Folgenden wird das sogenannte *Business Process Management (BPM)* eingeführt.

2.1 Business Process Management

Ein Geschäftsprozess (englisch: *business process*) besteht aus mehreren Aktivitäten, die koordiniert ausgeführt werden. Ihre gemeinsame Ausführung soll dazu führen, ein vorher definiertes Geschäftsziel zu erreichen. Jeder Geschäftsprozess kann hierbei sowohl die Interaktionen von Prozessbeteiligten innerhalb eines Unternehmens beschreiben, aber auch zwischen mehreren Unternehmen. Ein Geschäftsprozess definiert nicht

nur, welche Aktivitäten ausgeführt werden sollen, sondern auch, von wem sie ausgeführt werden, zu welchem Zeitpunkt und an welchem Ort [6].

Das *Business Process Management (BPM)* beschreibt Vorgehensweisen, wie Geschäftsprozesse erhoben, beschrieben, dargestellt und ausgeführt werden können. BPM beschreibt außerdem Methoden zur Konfiguration, Überwachung und Analyse von Geschäftsprozessen mit dem Ziel, einen Prozess zu verbessern. Dies umfasst nicht nur den genauen Ablauf von Aktivitäten, sondern auch die Organisationsstruktur. Diese beschreibt die hierarchische Struktur der Mitarbeiter eines Unternehmens.

Eine Übersicht über die Anwendung von BPM liefert der BPM Lifecycle, siehe Abbildung 2.1. Der *BPM Lifecycle* beschreibt den fortlaufenden Lebenszyklus eines Geschäftspro-

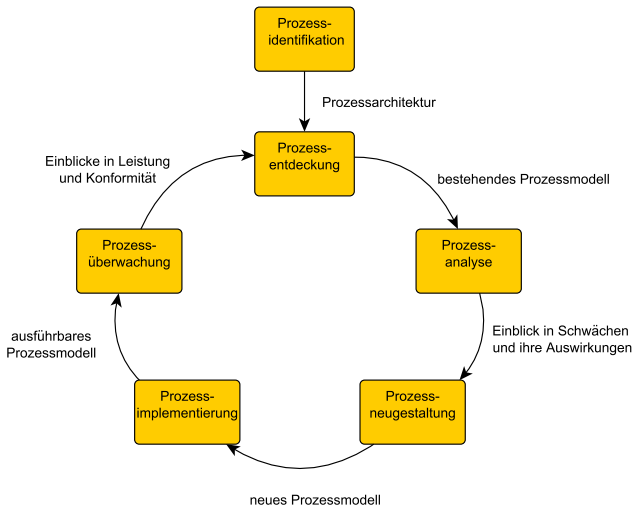

Abbildung 2.1: BPM Lifecycle (angelehnt an [7])

zesses. Der BPM Lifecycle beginnt mit der *Identifikation* von Prozessen (Processidentifikation) [7]. Hier werden bestehende Geschäftsprozesse erkannt. In der Prozessidentifikation werden Geschäftsprozesse im Ist-Zustand modelliert. Hilfreich sind hierbei Befragungen von Prozessbeteiligten und Beobachtern, sowie die Durchführung von Process Mining (siehe Kapitel 2.2). Daraufhin folgt die Phase der *Prozessanalyse*. Aus den im Schritt Prozessidentifikation erhobenen Informationen (z.B. Logdaten eines Informationssystems) wird nun ermittelt, welche Probleme und Schwachstellen bei den bestehenden Geschäftsprozessen vorhanden sind. Aus den entdeckten Schwachstel-

len werden in der folgenden Phase *Prozessneugestaltung* Optimierungsmöglichkeiten ermittelt. Darauf aufbauend wird das zuvor erstellte Ist-Prozessmodell neu modelliert. Anschließend folgt ein Vergleich zwischen der Leistung des neuen Soll- und des alten Ist-Geschäftsprozesses. In der nächsten Phase wird der verbesserte Prozess in einem Informationssystem technisch implementiert. Hier können organisatorische Änderungen erfolgen (z.B. Mitarbeiter A führt Aktivität B und nicht mehr Aktivität C aus). In der folgenden Phase wird der neu implementierte Geschäftsprozess kontrolliert und überwacht. Die Leistung des Prozesses wird anhand von *Key Performance Indicators (KPIs)* gemessen. Aus den gemessenen Leistungsdaten können im Folgenden weitere Prozessverbesserungen abgeleitet werden.

Die einzelnen Schritte des BPM Lifecycle werden hierbei ständig wiederholt.

Die *Business Process Model and Notation 2.0 (BPMN 2.0)* ist eine grafische Prozessbeschreibungssprache zur Darstellung von Prozessmodellen, die verbreitet Anwendung findet. Die wichtigsten Elemente des BPMN 2.0 Kollaborationsdiagramms sind in Abbildung 2.2 zu sehen. Dazu zählen folgende Elemente [1]:

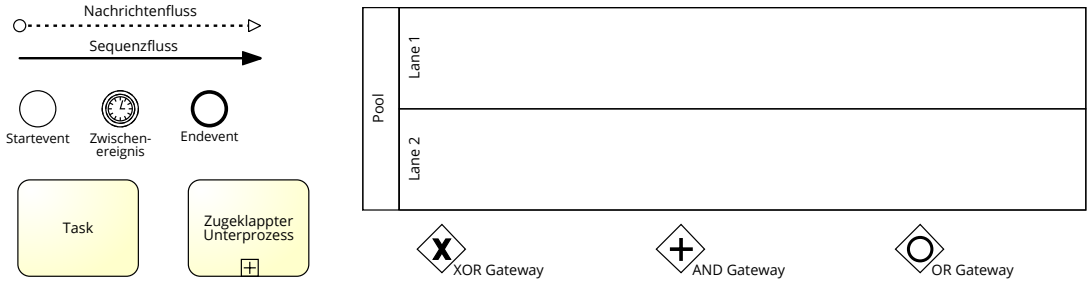

Abbildung 2.2: BPMN 2.0: Die wichtigsten Elemente

Pools zeigen die Zugehörigkeit zu einer Organisationseinheit (z.B. ein Unternehmen). Alle Elemente innerhalb der Grenzen eines Pools werden innerhalb der Organisationseinheit ausgeführt. Eine Kommunikation zwischen mehreren Pools kann über Nachrichtenflüsse abgebildet werden.

Eine *Lane* kennzeichnet eine Ressource innerhalb der Organisationseinheit. Eine Ressource kann beispielsweise ein Mitarbeiter oder ein System sein, das eine Aktivität ausführt.

Events beschreiben auftretende Ereignisse innerhalb eines Prozessmodells. Sie werden in BPMN 2.0 als Kreise dargestellt. In jedem BPMN-Prozessmodell muss ein Startevent und ein Endevent vorhanden sein. Ein Prozess startet mit Eintreten des Startevents und endet mit dem Endevent. Es existieren auch Zwischenereignisse, diese werden durch einen doppelten Rand dargestellt. Der Geschäftsprozess kann beim Auftreten eines Zwischenereignisses angehalten werden oder auch warten, bis ein bestimmtes Ereignis eintritt.

Tasks werden als Rechtecke mit abgerundeten Ecken dargestellt. Sie beschreiben einzelne Aktivitäten.

Events, Aktivitäten und Gateways werden durch *Sequenzflüsse* miteinander verbunden. Dies sind durchgezogene Verbindungslinien mit Pfeil, die anzeigen, in welcher Reihenfolge der Prozess abläuft.

Gateways können einen Prozessablauf in mehrere Pfade aufteilen oder vereinigen. Mit einem *Split-Gateway* wird ein Sequenzfluss aufgeteilt und mit einem *Join-Gateway* kann dieser wieder zusammengefügt werden. Ein AND-Gateway symbolisiert die parallele Ausführung mehrerer Pfade. Jeder dieser Pfade muss hierbei ausgeführt werden. Das parallele *Join-Gateway* wartet bis alle Pfade ihre Ausführung beendet haben und synchronisiert somit den Sequenzfluss. Das XOR-Gateway bietet die Möglichkeit einen Pfad aus mehreren auszuwählen. Bei einem OR-Gateway wird mindestens ein Pfad ausgeführt, eine Ausführung mehrerer Pfade ist ebenfalls möglich.

In einem Prozessmodell wird der Ablauf der Ausführung von Aktivitäten beschrieben. Jede Aktivität benötigt Ressourcen zur Ausführung.

Ein Beispiel für ein Prozessmodell ist in Abbildung 2.3 dargestellt. Dieses beschreibt die Registrierung eines Kunden im Callcenter.

Der Prozess startet, wenn ein Kunde im Callcenter anruft und sich als Neukunde registrieren möchte. Zuerst soll der Mitarbeiter die Kundendaten aufnehmen. Dann soll er die Daten prüfen und anschließend einen Bonitätscheck durchführen. Je nachdem, wie die Bonität bewertet wird, bietet der Mitarbeiter des Callcenters verschiedene Zahlungsarten an. Danach nimmt der Mitarbeiter den Zahlungswunsch des Kunden entgegen und speichert diesen. Anschließend ist der Prozess beendet.

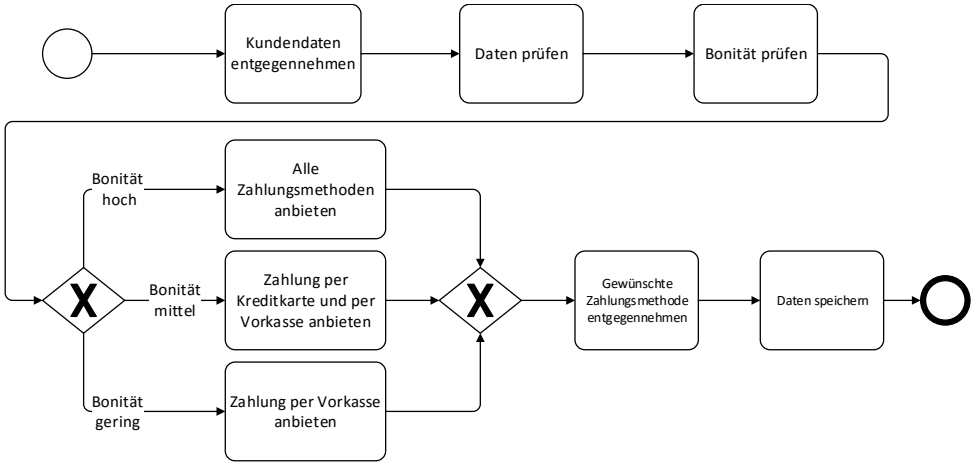

Abbildung 2.3: BPMN 2.0 Prozessmodell der Neukundenregistrierung in einem Callcenter

Ein Prozessmodell dient als Vorlage für sogenannte *Prozessinstanzen*. Eine Prozessinstanz beschreibt eine konkrete Ausführung eines Prozessmodells. Diese kann sich noch in der Ausführung befinden (*laufende Prozessinstanz*) oder bereits beendet sein (*historische Prozessinstanz*).

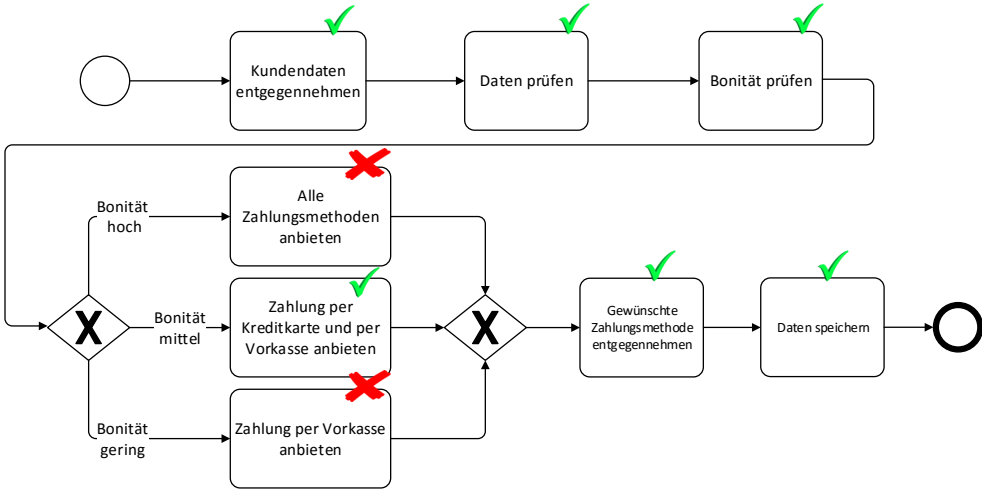

Abbildung 2.4: Beispiel einer Prozessinstanz

Abbildung 2.4 zeigt eine Prozessinstanz des Prozessmodells aus Abbildung 2.3: Ein Mitarbeiter des Callcenters nimmt die Kundendaten entgegen, und prüft diese und an-

schließend die Bonität des anfragenden Kunden. Aufgrund der ermittelten Bonität bietet er dem Kunden zwei Zahlungsmethoden an. Der Mitarbeiter nimmt den Zahlungswunsch entgegen und speichert diesen.

Die Symbole über den Aktivitäten in Abbildung 2.3 signalisieren, dass die jeweilige Aktivität erfolgreich beendet wurde (*grüner Haken*) oder nicht ausgeführt wurde (*rotes Kreuz*).

Der sogenannte *Trace* beschreibt den Ablauf der Ausführung von Aktivitäten einer Prozessinstanz. Der Trace der Prozessinstanz von Abbildung 2.4 ist: <Kundendaten entgegen nehmen, Daten prüfen, Bonität prüfen, Zahlung per Kreditkarte und per Vorkasse anbieten, gewünschte Zahlungsmethode entgegen nehmen, Daten speichern>

Ein Geschäftsprozess kann mit Hilfe eines *Business Process Management Systems (BPMS)* ausgeführt und überwacht werden. Ein BPMS ist ein prozess-zentriertes Informationssystem, das auf Grundlage von Prozessmodellen die Ausführung von Geschäftsprozessen koordiniert [6].

Viele Unternehmen setzen jedoch kein BPMS ein. Außerdem existieren viele Geschäftsprozesse, die nicht so leicht von einem BPMS gesteuert werden können. Dies sind vor allem Geschäftsprozesse, die nur in geringem Maß maschinelle Aktivitäten enthalten und zum größten Teil manuelle Aktivitäten enthalten.

In Unternehmen kommen jedoch häufig Informationssysteme zum Einsatz, die Geschäftsprozesse implizit abbilden. Die Ausführung von Aktivitäten wird hier oft in *Event Logs* dokumentiert. Für jede Aktivität (z.B. der Erzeugung einer Rechnung) wird ein Eintrag im Event Log gespeichert.

Ein Event Log ist meist so aufgebaut, dass er eine eindeutige Zuordnung zu einer Aktivität, zu einer ausführenden Ressource sowie zu einer Prozessinstanz enthält. Zudem wird jedem Eintrag eines Event Logs ein Zeitstempel zugeordnet. Ein Event Log Eintrag kann weitere Informationen enthalten, die für weitere Analyse hilfreich sein können (z.B. ein zweiter Zeitstempel für die Beendigung der Aktivität; die Rolle, die einem Prozessbeteiligten zugeordnet ist; die einer Ressource zugehörige Gruppe; eine Identifikationsnummer eines Kunden oder Patienten usw.)

Ein Event Log kann beispielsweise in einer Datei oder einer Datenbank gespeichert sein und enthält Informationen zu einem bestimmten Geschäftsprozess. Als *Instanzlog* bezeichnet man hierbei eine Menge von Event Logs, die zu einer bestimmten Prozessinstanz gehören.

2.2 Process Mining

Die Grundidee von Process Mining besteht darin, Informationen aus Event Logs zu gewinnen, die von einem Informationssystem gespeichert wurden [8]. Process Mining bietet Techniken, um Informationen über den Ablauf von Prozessen oder soziale Abhängigkeiten zu ermitteln.

Eine Liste mit Event Logs könnte so aussehen wie in Tabelle 2.1. Aus den darin vorhandenen Daten kann das Prozessmodell aus Bild 2.5 abgeleitet werden. Die Aktivitäten im erstellten Modell sind so koordiniert, dass fast jeder Instanzlog damit ausführbar ist.

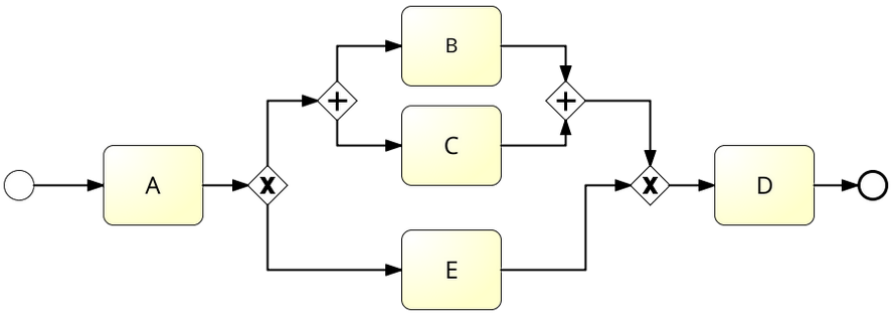

Abbildung 2.5: Ermitteltes Prozessmodell aus Tabelle 2.1 (angelehnt an [8])

Das sogenannte Social Mining ist ein Teilbereich des Process Minings. Es zielt darauf ab, die Abhängigkeiten der Mitarbeiter zu untersuchen. Damit kann eine Analyse des sozialen Netzwerks durchgeführt werden [9]. In Bild 2.6 sind die Ergebnisse einer solchen Untersuchung grafisch dargestellt. Dabei wird, geordnet nach einzelnen Instanzen, überprüft, an welchen Mitarbeiter jemand seine Arbeit übergibt. In den Logdaten von Tabelle 2.1 erkennt man beispielsweise, dass *John* in *case 1* seine Arbeit an *Mike* übergibt. Dieser übergibt dann an *John*.

CaseID	Activity	Ressource	Timestamp
case 1	A	John	9-3-2014 15:01
case 2	A	John	9-3-2014 15:12
case 3	A	Sue	9-3-2014 16:03
case 3	B	Carol	9-3-2014 16:07
case 1	B	Mike	9-3-2014 18:25
case 1	C	John	10-3-2014 9:23
case 2	C	Mike	10-3-2014 10:34
case 4	A	Sue	10-3-2014 10:35
case 2	B	John	10-3-2014 12:34
case 2	D	Pete	10-3-2014 12:51
case 5	A	Sue	10-3-2014 13:05
case 4	C	Carol	11-3-2014 10:12
case 1	D	Pete	11-3-2014 10:14
case 3	C	Sue	11-3-2014 10:44
case 3	D	Pete	11-3-2014 11:03
case 4	B	Sue	11-3-2014 11:18
case 5	E	Clare	11-3-2014 12:22
case 6	D	John	11-3-2014 12:34
case 5	D	Clare	11-3-2014 14:34
case 6	A	Sue	11-3-2014 15:05
case 4	D	Pete	11-3-2004 15:56

Tabelle 2.1: Beispiel für ein Event Log (angelehnt an [8])

Im Folgenden wird ein Process Mining Algorithmus vorgestellt, mit dem Informationen über den Ablauf von Prozessen berechnet werden können.

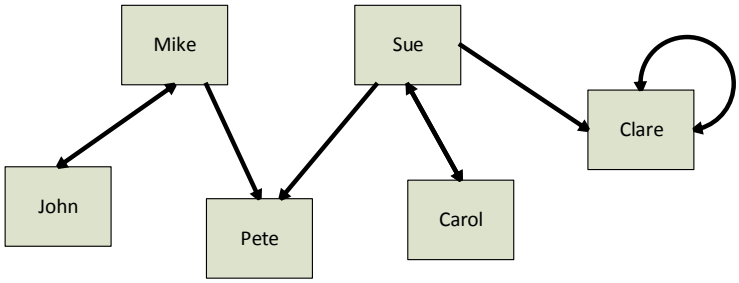

Abbildung 2.6: Ermittelte soziale Abhängigkeiten aus Tabelle 2.1 (angelehnt an [8])

2.2.1 Heuristic Miner

Der Process Mining Algorithmus namens Heuristic Miner ist in der Lage, Logdaten zu verarbeiten. Dabei erlaubt er eine sehr effiziente Bearbeitung, um qualitativ hochwertige Prozessmodelle zu ermitteln [10].

Process Mining Algorithmen haben viele Herausforderungen zu meistern. Eine davon besteht darin, falsch zugeordnete Event Logs auszufiltern. Beispiel dafür ist *Case 6* in Tabelle 2.1. Die Daten aus *Case 6* wurden nicht richtig gespeichert und sollen daher nicht im Ergebnis erscheinen.

Darüber hinaus gibt es Instanzlogs, die zwar korrekt aufgezeichnet sind, aber Einträge von Aktivitäten enthalten, die sehr selten in der Ausführung einer Prozessinstanz vorkommen [11]. Diese Einträge werden als *"Noise"* bezeichnet. Ein Process Mining Algorithmus sollte in der Lage sein diese herauszufiltern, da sonst das erzeugte Prozessmodell schnell zu komplex werden kann. Noise kann auch dann in einem Event Log auftreten, wenn bei der Ausführung einer Prozessinstanz Fehler auftreten.

Eine weitere Herausforderung von Process Mining Algorithmen besteht darin, parallele Ausführungen in einem Event Log zu erkennen und als Gateways im resultierenden Prozessmodell darzustellen.

Im Rahmen dieser Arbeit wird darauf verzichtet, zwischen XOR-,AND- und OR-Gateways zu unterscheiden. Jedes auftretende Gateway wird als ein OR-Gateway behandelt. Dies vermeidet den hohen Rechenaufwand beim Erkennen der verschiedenen Typen von Gateways. Wie später zu sehen ist, werden trotzdem sehr aussagekräftige Prozessmodelle erstellt.

Um die vom *Heuristic Miner* erzeugten Prozessmodelle darstellen zu können, werden im folgenden *C*-Netze* eingeführt (angelehnt an C-Netze aus [8]). Jedes C*-Netz besteht aus mehreren Knoten und Kanten. Es existiert je ein Start- und ein Endknoten. Diese werden durch Kreise dargestellt. Alle anderen Knoten (Rechtecke mit abgerundeten Kanten) stellen Aktivitäten dar. Gerichtete Kanten zeigen Abhängigkeiten zwischen den Knoten. Falls Knoten mehrere Ausgangskanten haben, dann handelt es sich um einen OR-Split. Abbildung 2.7 zeigt das Beispiel eines C*-Netzes. Knoten *A* stellt einen OR-Split-Knoten dar: die auf Knoten A folgenden Pfade können in beliebiger Kombination, auch parallel, ausgeführt werden. Entsprechend gilt, dass ein Knoten mit mehreren eingehende Kanten einen OR-Join-Knoten darstellt.

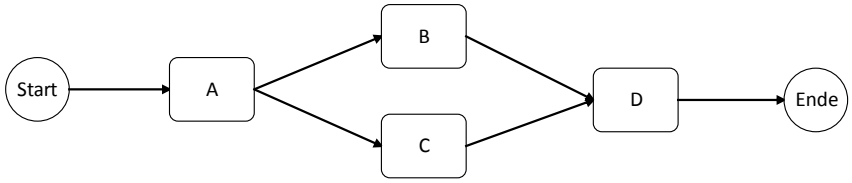

Abbildung 2.7: Beispiel eines C*-Netzes zur Darstellung eines Prozessmodells

Im Folgenden wird dargestellt, wie der Heuristic Miner funktioniert. Voraussetzung dafür sind vorhandene Event Logs. Bei diesen wird die zeitliche Abfolge von Aktivitäten geordnet nach Prozessinstanzen betrachtet. Als Beispiel dienen folgende Traces, die einfach aus einem Event Log errechnet werden können:

$$L_1 = [\langle a, c, d\rangle^5, \langle a, c, c, d\rangle^2, \langle a, b, c, d\rangle^{10}, \langle a, c, b, d\rangle^{10}, \langle a, c, c, c, d\rangle^1, \langle a, d\rangle^5, \langle a, d, b\rangle^1]$$

Der Exponent der einzelnen Traces gibt an, wie oft die angegebene Reihenfolge im gesamten Event Log vorkommt. Das Ergebnismodell beginnt mit einem Startknoten (S) und endet mit einem Endknoten (E). Der Heuristic Mining Algorithmus wandelt L_1 in folgende Traces um:

$\|a >_L b\|$	Start	a	b	c	d	Ende
Start	0	34	0	0	0	0
a	0	0	10	16	6	0
b	0	0	0	10	10	1
c	0	0	10	4	18	0
d	0	0	1	0	0	33
Ende	0	0	0	0	0	0

Tabelle 2.2: $|a >_L b|$ Tabelle des Heuristic Miners

$L = [\langle S, a, c, d, E\rangle^5, \langle S, a, c, c, d, E\rangle^2, \langle S, a, b, c, d, E\rangle^{10}, \langle S, a, c, b, d, E\rangle^{10},$
$\langle S, a, c, c, c, d, E\rangle^1, \langle S, a, d, E\rangle^5, \langle S, a, d, b, E\rangle^1]$

Daraufhin wird auf L folgende Gleichung angewendet (aus [8]):

$$|a >_L b| = \sum_{\sigma \in L} L(\sigma) \times |\{1 \leq i < |\sigma| | \sigma(i) = \alpha \wedge \sigma(i+1) = b\}| \tag{2.1}$$

Die Gleichung betrachtet jede Aktivität, die im Event Log vorkommt (im Beispiel: Start, a, b, c, d, Ende) und berechnet die Anzahl der Vorkommnisse der Aktivitäten. Angewendet auf das Event Log aus Tabelle 2.1 erhält man folgende Ergebnisse, die in Tabelle 2.2 dargestellt sind.

Die einzelnen Werte in Tabelle 2.2 sind wie folgt zu interpretieren: Die Zahl 10 in der dritten Zeile und der vierten Spalte bedeutet, dass in der Logdatei Aktivität *b* 10 mal direkt nach Ausführung der Aktivität *a* ausgeführt wurde. Die Tabelle gibt also die Häufigkeiten an, welche Aktvitäten-Ausführung von welcher anderen Aktvitäten-Ausführung gefolgt wird.

Um ein Prozessmodell zu berechnen, muss zusätzlich noch folgende Formel angewendet werden (aus [8]):

$$|a \Rightarrow_L b| = \begin{cases} \frac{|a>_L b| - |b>_L a|}{|a>_L b| + |b>_L a| + 1} & if a \neq b \\ \frac{|a>_L a|}{|a>_L a| + 1} & if a = b \end{cases}$$

Diese Formel berechnet für jedes Paar, auf das es angewendet wird, welches Abhängigkeitsmaß es besitzt. Für jedes Paar (x, y) von Aktivitäten ergeben sich Werte zwischen -1 und $+1$. Der Wert für $|x \Rightarrow_L y|$ ist nahe bei $+1$, wenn es eine stark positive Abhän-

$\lvert a \Rightarrow_L b \rvert$	Start	a	b	c	d	Ende
Start	$\frac{0}{0+1} = 0$	$\frac{34-0}{34+0+1}$ =0.97	0	0	0	0
a	$\frac{0-34}{0+34+1}$ =-0.97	0	$\frac{10-0}{10+0+1}$ =0.91	$\frac{16-0}{16+0+1}$ =0.94	$\frac{6-0}{6+0+1}$ =0.86	0
b	0	$\frac{0-10}{0+10+1}$ =-0.91	0	$\frac{10-10}{10+10+1}$ =0	$\frac{10-1}{10+1+1}$ =0.75	$\frac{1-0}{1+0+1}$ =0.5
c	0	$\frac{0-16}{0+16+1}$ =-0.94	$\frac{10-10}{10+10+1}$ =0	$\frac{4}{4+1}$ =0.8	$\frac{18-0}{18+0+1}$ =0.95	0
d	0	$\frac{0-6}{0+6+1}$ =0.86	$\frac{1-10}{1+10+1}$ =-0.75	$\frac{0-18}{0+18+1}$ =-0.95	0	$\frac{33-0}{33+0+1}$ =0.97
Ende	0	0	$\frac{0-1}{0+1+1}$ =-0.50	0	$\frac{0-33}{0+33+1}$ =-0.97	0

Tabelle 2.3: $\lvert a \Rightarrow_L b \rvert$ Tabelle des Heuristic Miners

gigkeit zwischen x und y gibt. Dann wird x oft von y gefolgt, x kommt aber selten direkt nach y vor. Man kann also daraus schließen, dass x die Ursache dafür ist, dass y erfolgt. Ein Wert nahe an -1 bedeutet eine stark negative Abhängigkeit: y folgt selten auf x, x wird jedoch oft nach y ausgeführt.

Für $\lvert x \Rightarrow_L x \rvert$ wird eine leicht veränderte Formel verwendet. Ein Wert nahe $+1$ bedeutet eine stark reflexive Abhängigkeit. Dies ist ein Hinweis darauf, dass an dieser Stelle ein Loop erfolgen kann.

Tabelle 2.3 zeigt die Ergebnisse der Anwendung von Abbildung $\lvert a \Rightarrow_L b \rvert$ auf Tabelle 2.2. Sie zeigen beispielsweise, dass eine stark positive Abhängigkeit zwischen a und b existiert. Außerdem besteht für c eine stark reflexive Abhängigkeit. Die Tabellen 2.2 und 2.3 bilden die Ausgangsbasis für den Heuristic Miner. Das resultierende Modell, ein gerichteter Graph, ist in Grafik 2.8 zu sehen. Für die Konstruktion des Ergebnismodells geht man wie folgt vor: Die Knoten sind Ereignisse und Aktivitäten aus den Event Logs. Die Kanten werden aus den Einträgen der Ergebnistabellen berechnet. Für jeden Eintrag (x, y) erhält man aus Tabelle 2.2 eine Zahl für die absolute Häufigkeit und aus Tabelle 2.3 entnimmt man ebenfalls für (x, y) einen Wert für das Abhängigkeitsmaß zwischen -1 und 1. Zur Erstellung der Ergebnismodells muss ein sogenannter *Threshold* angegeben werden. Dieser besteht aus zwei Werten, einem für die Häufigkeit und einem für das

Abhängigkeitsmaß. Diese geben an, welche Werte aus den Tabellen überschritten werden müssen, sodass diese dann als Kante in das resultierende Ergebnismodell aufgenommen werden. Das interessante am Threshold besteht darin, dass Anomalien wie Noise oder falsche Event Logs aus dem Ergebnis ausgeklammert werden können (außer es werden zu kleine Thresholds verwendet). Außerdem besteht das grundlegende Problem jedes Ergebnismodells darin, eine gute Balance zwischen Einfachheit (ein möglichst übersichtliches Modell mit wenigen Kanten) und *"Fitness"* (ein Modell, das möglichst jedes in den Event Logs aufgezeichnete Verhalten berücksichtigt) zu finden [11]. Durch Angabe eines Thresholds kann gewählt werden, wie einfach oder komplex das resultierende Ergebnismodell sein soll.

Abbildung 2.8 zeigt das berechnete Ergebnismodell für einen Threshold von 4 für die Häufigkeit sowie 0,6 für das Abhängigkeitsmaß.

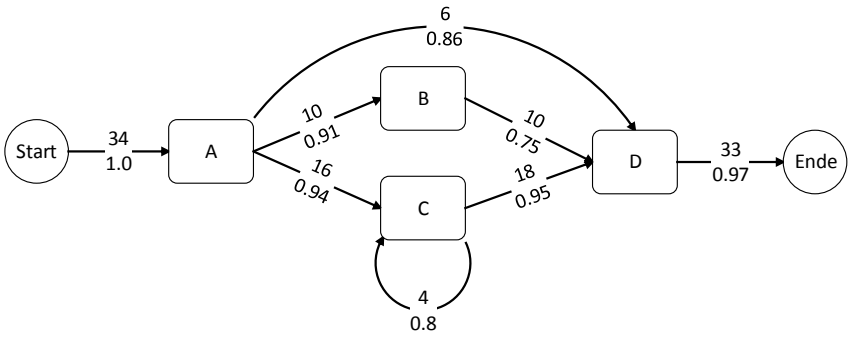

Abbildung 2.8: Ergebnisgrafik des Heuristic Miner Beispiels mit Threshold 4 und 0.6

In Abbildung 2.9 ist das Ergebnismodell mit Threshold 6 für die Häufigkeit und 0,7 für das Abhängigkeitsmaß dargestellt. Der Loop bei Aktivität C verschwindet hierbei, es ergibt sich somit ein noch übersichtlicheres Prozessmodell. Soll also nur das Standard-Verhalten abgebildet werden, so empfiehlt es sich, höhere Werte für den Threshold zu wählen. Für ein detailliertes Prozessmodell wird entsprechend ein kleinerer Threshold benötigt.

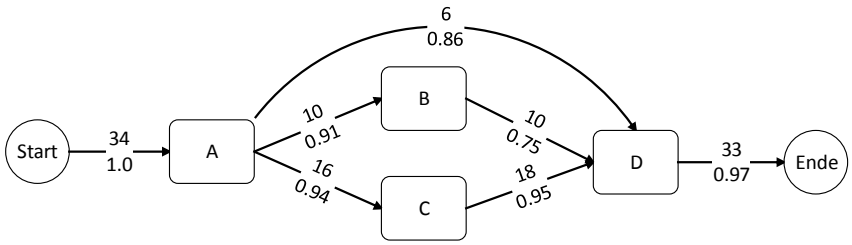

Abbildung 2.9: Ergebnisgrafik des Heuristic Miner Beispiels mit Threshold 6 und 0.7

2.2.2 Social Network Analysis

Aus Event Logs können nicht nur kausale Abhängigkeiten zwischen Aktivitäten berechnet werden, sondern auch Informationen über organisatorische Strukturen. In diesem Abschnitt wird eine *soziale Netzwerkanalyse* vorgestellt. Diese errechnet aus einem Event Log ein Soziogramm in Form eines Graphen, dessen Knoten Personen repräsentieren und dessen Kanten Abhängigkeiten zwischen den einzelnen Personen repräsentieren [9]. Voraussetzung der Anwendung einer sozialen Netzwerkanalyse ist, dass Event Logs zur Verfügung stehen, in denen zu jeder Aktivität die ausführende Ressource (also die jeweilige Person) protokolliert wurde. Als Darstellungsform wird wieder ein C*-Netz verwendet, das einen Start- und einen Endknoten enthält.

Die Vorgehensweise ist ähnlich wie beim Heuristic Miner. Aus den Event Logs werden Traces erstellt. In den Traces werden aber nicht die Aktivitäten angegeben, sondern die ausführenden Personen. Außerdem wird zu jedem Eintrag noch ein Start- (S) und ein Endknoten (E) eingefügt.

Als Beispiel dient Folgendes: die Mitarbeiter heißen Heinz (H), Peter (P), Klaus (K) und Franziska (F). Der zugrunde liegende Event Log sieht wie folgt aus:

$L = [\langle S, H, K, E\rangle^{10}, \langle S, P, K, E\rangle^{10}, \langle S, H, K, F, P, K, E\rangle^{5}, \langle S, P, K, F, H, K, E\rangle^{6},$
$\langle S, P, K, F, K, E\rangle^{4}, \langle S, K, H, F, P, K, E\rangle^{1}]$

Sei L der Event Log und $p_1, p_2 \in P$, wobei P die Menge der Personen ist, dann wird folgende Formel für die Berechnung der sogenannten *"Handover of Work Matrix"* ange-

$\lvert a >_L b \rvert$	Start	Heinz	Peter	Klaus	Franziska	Ende
Start	0	$\frac{15}{36} = 0.42$	$\frac{20}{36} = 0.56$	$\frac{1}{36} = 0.03$	0	0
Heinz	0	0	0	$\frac{21}{36} = 0.58$	$\frac{1}{36} = 0.03$	0
Peter	0	0	0	$\frac{25}{36} = 0.69$	0	0
Klaus	0	$\frac{1}{36} = 0.03$	0	0	$\frac{15}{36} = 0.42$	$\frac{36}{36} = 1.0$
Franziska	0	$\frac{6}{36} = 0.17$	$\frac{6}{36} = 0.17$	$\frac{4}{36} = 0.11$	0	0
Ende	0	0	0	0	0	0

Tabelle 2.4: Handover of Work Matrix

wendet (angelehnt an [3]):

$$p_1 \triangleright_L p_2 = \left(\sum_{c \in L} |p_1 \triangleright_L p_2| \right) / |L| \tag{2.2}$$

Das heißt, für jede Person $p1$ wird berechnet, wer in welcher Häufigkeit direkt danach seine Tätigkeit aufnimmt. Man rechnet also aus, wie oft Person $p1$ der Person $p2$ ihre Arbeit übergibt. Aus diesem Grund heißt der Algorithmus Handover of Work. Das Ergebnis wird durch die Anzahl der Prozessinstanzen geteilt, die im Event Log abgebildet werden.

Das Ergebnis für L ist in Tabelle 2.4 zu sehen. Beispielsweise hat Heinz 21 mal seine Arbeit an Klaus übergeben. 36 Instanzen wurden in der Logdatei aufgezeichnet. Daher lautet der Eintrag $\frac{21}{36} = 0.58$.

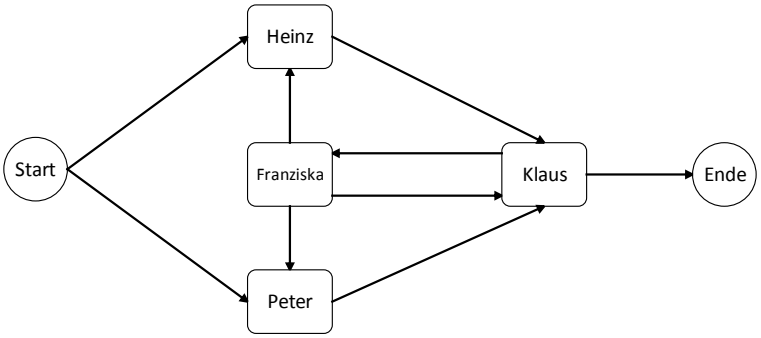

Abbildung 2.10: Handover of Work C*-Netz

Aus dieser Tabelle kann anschließend ein C*-Netz erstellt werden. Jede Person bildet einen Knoten, außerdem enthält der Graph einen Start- und Endknoten. Die Kanten

werden aus der Tabelle abgeleitet. Jedes Paar (x, y), das einen Wert in der Tabelle hält (der größer ist als der vorgegebene Threshold) bildet eine Kante.

Für das Beispiel wird ein Threshold von 0.1 verwendet. Das Ergebnis ist in Abbildung 2.10 dargestellt. Die Kanten zeigen an, welche Person welcher anderen die Arbeit übergibt. Zu beobachten ist, dass sehr kleine Werte aus der Tabelle in der Grafik nicht dargestellt werden. Der Threshold könnte auch noch höher gesetzt werden, um nur noch die wichtigsten Abhängigkeiten anzuzeigen und ein noch übersichtlicheres Modell zu erhalten. Zusätzlich zu einem klassischen Soziogramm erhält man mit dem Startknoten und dem Endknoten weitere Informationen: Man sieht, wer zu Beginn eines Falls arbeitet (im Beispiel betrifft dies Heinz und Peter) und wer den Fall beendet (Klaus). Insgesamt erhält man mit dem erstellten Soziogramm, das über die Handover of work Matrix erstellt wurde, einen Einblick in die Arbeitsstruktur zwischen Personen. Eventuell kann man daraus eine Organisationsstruktur eines Unternehmens ableiten, oder aber man sieht, wer gerne mit wem arbeitet. Beides sind Informationen, die zusammen mit dem vom Heuristic Miner erstellten Prozessmodell dabei helfen können, Geschäftsprozesse zu verstehen und zu analysieren.

2.3 MapReduce

In Kapitel 2.2 wurden Algorithmen vorgestellt, die es ermöglichen, Process Mining auf Event Logs anzuwenden. Um diese effizient auf großen Datenmengen anzuwenden, werden weitere Technologien benötigt, die nachfolgend eingeführt werden.

MapReduce ist ein Programmiermodell, das große Datensätze effizient parallel verarbeiten kann. Eine Implementierung in MapReduce bietet sehr gute Leistungen im Vergleich zu alternativen Systemen [12]. Auch im Vergleich zur Programmierung in einer maschinennahen Programmiersprache kann MapReduce Geschwindigkeitsvorteile bieten [13].

MapReduce wurde 2004 von Google eingeführt. Seitdem wurde es kontinuierlich weiterentwickelt und von vielen anderen großen Unternehmen verwendet [5]. Die Grundidee von MapReduce ist es, große Datensätze in kleinere Teilaufgaben aufzutrennen ("Split"),

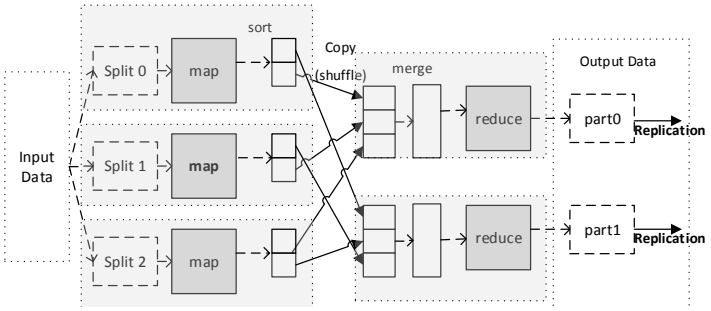

Abbildung 2.11: Schematische Ausführung von MapReduce (angelehnt an [14])

diese parallel zu verarbeiten ("Map") und dann wieder zusammenzufügen ("Reduce"). Kennzeichen moderner MapReduce Frameworks, wie Apache Hadoop, ist die einfache Handhabung: Programmierer müssen lediglich eine Map-Funktion und eine Reduce-Funktion bereitstellen, um weitere Aspekte wie Verteilung der einzelnen Datenpakete auf verteilte Hardware kümmert sich in der Regel das entsprechende Framework transparent [15].

In der Regel werden die einzelnen Teilaufgaben bei MapReduce auf verteilten Systemen (*Clustern*) verarbeitet. Diese setzt sich aus mehreren sogenannten *Nodes* zusammen. Eine Node kann entweder ein physikalischer oder virtualisierter Computer sein.

Der Ablauf eines MapReduce Jobs wird in Abbildung 2.11 dargestellt. Die zu verarbeitenden Datensätze werden zunächst in Blöcke geteilt ("Split"). Diese Blöcke haben in der Regel eine Größe von etwa 128MB [14]. In einem Cluster befinden sich die Blöcke hierbei auf vielen verschiedenen Knoten. Bei einer realen Ausführung sind auf jedem Knoten viele Blöcke. Jeder Knoten führt nach dem Aufteilvorgang mehrere Map-Fuktionen aus, wobei eine Map-Funktion einen Teil der Kernlogik der anzuwendenden Algorithmen darstellt. Map-Funktionen werden über einen kompletten Cluster parallel ausgeführt. Die Ergebnisse der Map-Funktionen werden danach über ein Netzwerk innerhalb des Clusters auf andere Knoten kopiert, auf denen wiederum Reduce-Funktionen ausgeführt werden. Dies ist die sogenannte Shuffle Phase, in der die Ergebnisse der einzelnen Map-Funktionen wieder zusammengefügt und anschließend in ein Dateisystem geschrieben werden. Je nach Implementierung eines MapReduce Frameworks werden diese

Ergebnisdaten innerhalb eines verteilen Dateisystems repliziert, das heißt auf mehrere Knoten im Cluster geschrieben. Dadurch kann einem möglichen Datenverlust bei Defekt eines Knotens vorgebeugt werden.

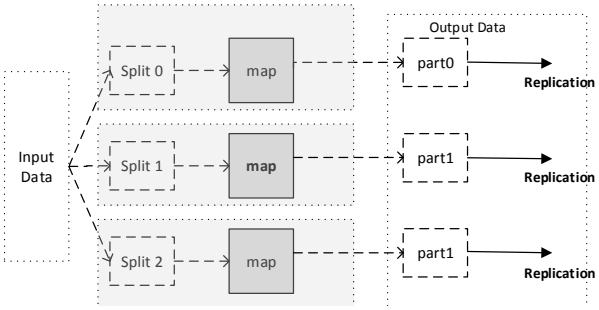

Abbildung 2.12: MapReduce: Ablauf bei mehreren Mappern ohne Reducer (angelehnt an [14]

Die Anzahl der verwendeten Map- und Reduce-Funktionen kann variieren. Es besteht auch die Möglichkeit, dass keine Reduce-Funktion angewendet wird (siehe Abbildung 2.12). Dies ist dann der Fall, wenn die Map-Funktion ausreicht, um das gesuchte Ergebnis zu erhalten: auf eine Reduce-Funktion kann beispielsweise verzichtet werden, wenn man aus einem Text nur diejenigen Zeilen herausfiltern möchte, die ein bestimmtes Wort enthalten. Auch bei einer Addition von Zahlen ist keine Reduce-Funktion notwendig.

Zum leichteren Verständnis von MapReduce dient im Folgenden ein einfaches Word-Count Beispiel. Der Ausgangsdatensatz ist ein Text und für jedes darin vorkommende Wort soll ermittelt werden, wie oft es darin vorkommt. Die zugehörige Map- und Reduce-Funktion ist nachfolgend in Pseudo-Code dargestellt [15]:

```
1  map(String key, String value):
2          //key: document name
3          // value: document contents
4          for each word w in value:
5                  EmitIntermediate(w, "1");
6
7  reduce(String key, Iterator values):
8          // key: a word
9          // values: a list of counts
10         int result = 0;
```

```
11          for each v in values:
12                  result += ParseInt(v);
13          Emit(AsString(result));
```

Listing 2.1: Map und Reduce Funktion in Pseudo Code

Eine Menge von Schlüssel/Wert Paaren (k1/v1) dient hierbei mehreren Map-Funktionen als Input (Zeile 1). Aus jedem einzelnen Schlüssel/Wert Paar gibt die Map-Funktion eine Menge von Intermediate Schlüssel/Wert Paaren aus (Zeile 5). Es entsteht folglich eine Liste von Intermediate Schlüssel/Wert Paaren (list(k2,v2)). Ein MapReduce Framework gruppiert dann alle Intermediate Schlüssel/Wert Paare nach gleichen Schlüsseln (k2,list(v2)). Diese werden an die Reduce-Funktion übergeben (Zeile 7). Nach Beendigung der Reduce-Funktion soll eine möglichst kleine Menge von Ausgabewerten entstehen (list(v2)). Die Anwendung des MapReduce Konzepts kann schematisch wie folgt zusammengefasst werden [15]:

map	(k1,v1)	$\rightarrow$	list	(k2,v2)
reduce	(k2,list(v2))	$\rightarrow$	list	(v2)

Der Ablauf des WordCount Beispiels ist in Abbildung 2.13 dargestellt. Die Eingangsdatenmenge wird am Anfang aufgeteilt. Die Splitgröße beträgt im Beispiel eine Zeile. Jede Zeile dient einem separaten Mapper als Eingangsdatum. Jeder Mapper betrachtet seine Eingabedaten als Schlüssel-Wert Paare (k1,v1). Für den obersten Mapper bedeutet dies: (k1=Zeile1, v1= "Business Process Int."). Der Mapper verarbeitet diese Eingabe und erzeugt eine Liste von Schlüssel/Wert Paaren: list(k2,v2). Im Beispiel ist dies: [("Business",1), ("Process",1), ("Int.",1)]. Diese Zwischenergebnisse werden über die Shufflephase auf die einzelnen Reducer verteilt. Als Eingangsdaten für den Reducer dienen ein Schlüssel mit einer Liste von Werten (k2,list(v2)). Für den zweitobersten Reducer bedeutet dies ("Process", (1,1,1)) als Eingang. Das Ergebnis des Reducers wird als Liste von Werten dargestellt: list(v2). Für den zweitobersten Reducer hat diese Liste nur einen Eintrag: ("Process", 3). Von allen Reducern gehen die Ergebnisse an den Output. Dort werden sie gespeichert.

Durch die vielen Verbindungen der einzelnen Datenpakete in der Shuffle Phase ist im Beispiel gut zu erkennen, dass die Kommunikation zwischen den Nodes innerhalb eines MapReduce Clusters sehr groß werden kann. Um dieses Verhalten zu optimieren, kann nach Anwendung der Map-Funktionen und vor der Shuffle-Phase ein sogenannter *Combiner* eingefügt werden. Dieser arbeitet wie ein lokaler Reducer, wird also auf dem gleichen Node wie die zugehörige Map-Funktion ausgeführt [15]. Aufgabe des Combiners ist es, die Ergebnisliste der Map-Funktionen zu verkleinern, bevor sie während der Shuffle-Phase zu den Reducer-Nodes kopiert wird. Der Einsatz eines Combiners ist möglich bei Berechnung von Summen, Maximum oder Minimum. Der zweitoberste Mapper in Abbildung 2.13 beispielsweise liefert als Ausgabe: [("Big",1),("Big",1),("Data",1)]. Der Combiner kann die beiden gleichen Datensätze addieren und liefert als Ergebnis: [("Big",2),("Data",1)]. Der Effekt auf den Netzwerkverkehr ist hier zwar gering, bei großen Datenmengen ergeben sich aber positive Effekte, die sich auch auf die Ausführungsgeschwindigkeit auswirken.

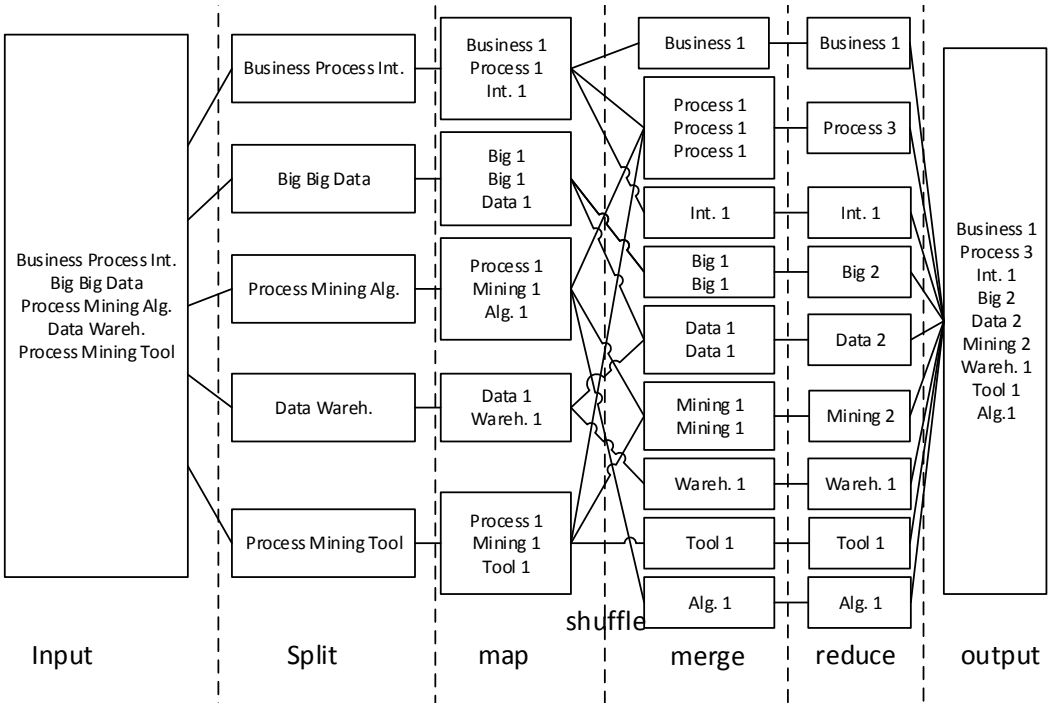

Abbildung 2.13: Darstellung der Phasen von MapReduce anhand eines WordCount Beispiels

3

Apache Hadoop

Die Funktionsweise von MapReduce ist in unterschiedlichen Frameworks implementiert, z.B. Apache Hadoop, Twister und Disco [5, 16, 17]. Vor allem bei sehr großen Realwelt-Datenmengen arbeitet hierbei Apache Hadoop effizienter als andere Frameworks [18]. Momentan ist Apache Hadoop eines der beliebtesten und am häufigsten eingesetzte MapReduce Framework auf dem Markt [19]. Aufgrund des guten Supports kann ein Hadoop-Cluster schnell aufgesetzt werden. Zudem können vorkonfigurierte Hadoop-Cluster in vielen Cloud-Umgebungen, z.B. Amazon AWS oder Microsoft Azure, gemietet werden.

In diesem Kapitel werden Aufbau und Funktionsweise von Apache Hadoop ("Hadoop") erklärt.

Hadoop ist ein Open Source Software Framework, das hauptsächlich in Java geschrieben wurde. Hauptziel des Hadoop-Projekts ist es, große Datenmengen in einem Cluster

von vielen Knoten verteilt zu speichern und Programme zur Datenverarbeitung und Analyse verteilt auf viele Nodes parallel zu verarbeiten. Zur Datenverarbeitung wurde das von Google geprägte MapReduce Framework in Hadoop implementiert. Hadoop nutzt als verteiltes Dateisystem eine Weiterentwicklung des Googles Distributed Filesystem (GFS), nachfolgend als HDFS (Hadoop Distributed File System) bezeichnet. HDFS kann große Datenmengen verteilt und fehlertolerant speichern (z.B. bei Defekten von Speichermedien).

Die Softwarearchitektur von Hadoop ist in Abbildung 3.3 dargestellt. Hierbei verwaltet YARN (Yet Another Resource Negotiator) alle Hardwareressourcen und steuert den Ablauf von Berechnungen ("Jobs"). Auf YARN aufbauend können unterschiedliche Anwendungen gleichzeitig in einem Hadoop Cluster laufen. Seit Hadoop 2.0 und der Einführung von YARN ist es möglich, nicht mehr nur MapReduce-basierte Anwendungen auf dem HDFS auszuführen. Beispielsweise können Frameworks für Echtzeit-Analyse (Apache Tez) und Graphen-Analysen (Apache Giraph) parallel zur Ausführung von MapReduce-Operationen ausgeführt werden [20] [21].

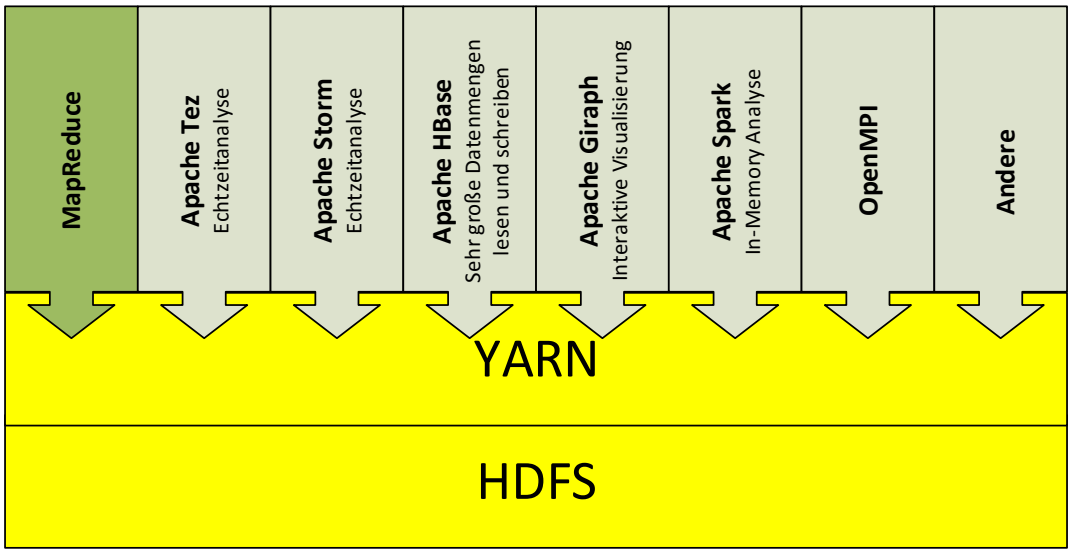

Abbildung 3.1: Hadoop Softwarearchitektur [22]

3.1 YARN

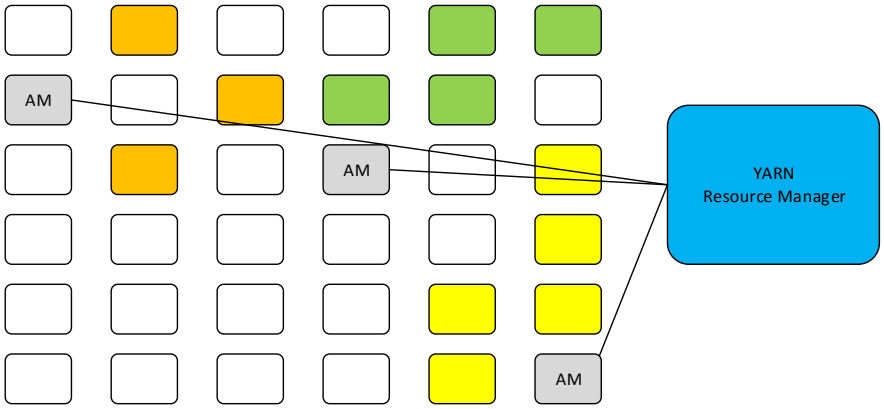

Abbildung 3.2: Beispiel für ein YARN Cluster

Abbildung 3.2 zeigt als Beispiel ein Hadoop Cluster mit 37 Knoten. Auf diesem Cluster werden parallel drei verschiedene Jobs ausgeführt. Die gelb dargestellten Nodes führen Map-Jobs aus, orange dargestellte Nodes einen Reduce-Job. Gleichzeitig führen grün dargestellte Nodes eine Echtzeit-Analyse mit Apache Tez durch. Die einzelnen Jobs können von verschiedenen Clients gestartet worden sein. Dafür muss ein Client Kontakt mit dem *ResourceManager* aufnehmen. Im gesamten Cluster existiert eine Node, die den ResourceManager ausführt. Dieser verwaltet wiederum die Hardwareressourcen des gesamten Clusters. Jeder Job benötigt zur Verwaltung einen eigenen *ApplicationMaster (AM)*. Auf drei Nodes im Beispiel-Cluster läuft ein ApplicationMaster. 21 weitere Nodes befinden sich im Ruhezustand.

Nachfolgend werden die wichtigsten Komponenten von YARN detaillierter betrachtet ([23]). Ein Hadoop Cluster besteht aus *MasterNodes* und *SlaveNodes*. Eine *MasterNode* hat die Aufgabe, die Hardwareressourcen des Clusters zu verwalten. SlaveNodes bezeichnen Nodes, in denen Daten gespeichert und Datenberechnungen stattfinden. Auf jeder *SlaveNode* läuft ein NodeManager und mehrere Container, von denen einer von einem ApplicationMaster belegt sein kann, siehe Abbildung 3.3. Container sind logische

Einheiten von Ressourcen (z.B. 1 CPU-Kern, 2GB RAM), die jeweils einem Knoten zugeordnet sind. Ein NodeManager ist ein Programm, das die Hardwareressourcen innerhalb des Knotens verwaltet. Es kann Container starten und beenden, hat Überblick über die zur Verfügung stehenden Kapazitäten und die auftretenden Fehler.

Auf einer MasterNode läuft ein ResourceManager. Der ResourceManager verwaltet die Hardwareressourcen des Clusters, darunter CPU-Kapazität und Speicherplatz. Der ResourceManager implementiert zwei Schnittstellen: zum Einen zu Clients, die Applikationen übermitteln; zum Anderen zu ApplicationMasters, die über Hardwareressourcen für ihre Jobs verhandeln. Einzelne Jobs, nachfolgend *Applikationen* genannt, sollen so effizient wie möglich ausgenutzt werden.

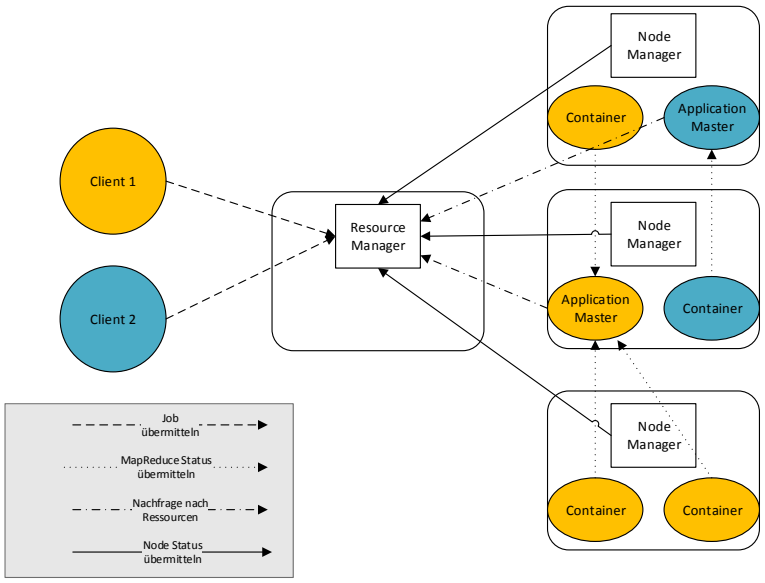

Abbildung 3.3: YARN Software-Architektur mit ResourceManager und NodeManager [5]

Wenn eine Applikation von einem Client beauftragt wird, wird zuerst abgewartet, bis der Scheduler des ResourceManagers genügend freie Hardwareressourcen zur Verfügung hat. Erst dann wird die Applikation gestartet. Als Erstes wird der ResourceManager den Auftrag geben, einen Container zu starten, in dem der ApplicationMaster arbeiten soll. Der ApplicationMaster ist für den kompletten Job verantwortlich, dies umfasst z.B. den Umgang mit dynamisch ändernden Anforderungen an Hardwareressourcen

sowie Fehlerbehandlungen. Um Container zu erhalten, stellt der ApplicationMaster eine Anfrage an den ResourceManager. In dieser ist definiert, wie viele Container der ApplicationMaster benötigt, an welchem Ort sie sich physikalisch befinden sollen und wie viele Prozessorkerne und RAM pro Container benötigt werden. Der ResourceManager entscheidet nun nach verfügbarer Kapazität und definierter Scheduling-Strategie, ob die angeforderten Hardwareressourcen zugeteilt werden können [23]. Bei Zuteilung wird dem ApplicationMaster mitgeteilt, dass bestimmte Hardwareressourcen zugeteilt werden. Diese Zuteilung wird an die betreffenden NodeManager weitergeleitet, welche wiederum die Container einrichten. Anschließend kann der ApplicationMaster einen Task innerhalb eines Containers starten. Dies ist im Fall einer MapReduce-Applikation entweder ein Map-Task oder ein Reduce-Task, da jede MapReduce-Applikation aus mehreren Map-Tasks und Reduce-Tasks besteht. Der Container kommuniziert direkt mit dem ApplicationMaster, um den eigenen Status mitzuteilen oder Befehle entgegenzunehmen. Während der Ausführung einer Applikation erhält der Client direkt vom ApplicationMaster Informationen über Status und Fortschritt des jeweiligen Jobs. Wenn die Arbeit erledigt ist und eine Applikation abgeschlossen ist, so teilt dies der ApplicationMaster dem ResourceManager mit. Darauf wird der Container, in dem der ApplicationMaster registriert war, aufgelöst und kann für andere Zwecke wiederverwendet werden. Der ResourceManager kann daraufhin die frei gewordenen Ressourcen weiteren Applikationen zuweisen.

YARN bietet eine robuste Struktur für die Verwaltung und Überwachung von Containern, wobei die Implementierung der einzelnen Applikationen dem jeweiligen Framework überlassen wird.

3.2 Hadoop Distributed Filesystem (HDFS)

Im letzten Kapitel wurde behandelt, wie YARN die Ausführung von Jobs in einem Hadoop-Cluster regelt und die damit verbundene Verteilung von Prozessorleistung und Hauptspeicherkapazität kontrolliert. Im Folgenden wird das verteilte Dateisystem HDFS betrachtet [5, 24].

Eine Anforderung an die Entwicklung von HDFS bestand darin, möglichst große Datenmengen verlässlich zu speichern. HDFS ist in der Lage, Datenmengen von bis zu 200 PB in einem einzigen Cluster mit mehreren tausend Knoten mit über einer Milliarde Dateien zu verwalten. Um Hardwarefehlern vorzubeugen, werden Dateien in Pakete aufgeteilt und verteilt im Cluster gespeichert. Auftretende Hardwarefehler können hierbei selbstständig erkannt und behoben werden. HDFS ist effizient implementiert, die Ausführung von Jobs innerhalb eines HDFS-Clusters benötigt nur einen sehr geringen Verwaltungsaufwand.

Wichtiger Teil der HDFS Architektur sind *NameNodes*. Sie verwalten Informationen über Position und Zustand aller Dateien innerhalb eines HDFS-Clusters (Namespace) und entscheiden, in welche physikalischen Nodes Daten gespeichert werden. Es existiert nur eine NameNode pro Cluster. Die NameNode befindet sich auf einer MasterNode. Jede SlaveNode besitzt eine sogenannte *DataNode*: diese speichert Daten in Blöcken. Außerdem existiert eine *CheckpointNode*, die der NameNode dabei hilft, ihre Verwaltungsdaten aktuell zu halten.

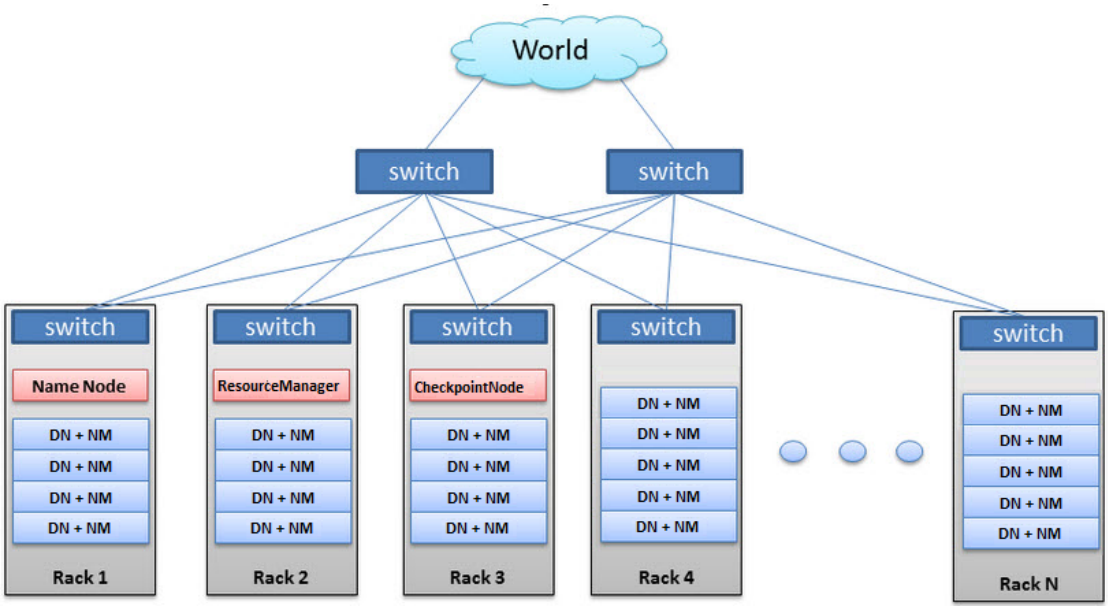

Abbildung 3.4: Beispiel für ein HDFS Cluster (angelehnt an [25])

Abbildung 3.4 zeigt ein weiteres Beispiel für ein Hadoop-Cluster. Das Cluster besteht aus N verschiedenen *Racks*, wobei jedes Rack aus 5 Nodes besteht. Große Cluster

werden in der Regel mit drei MasterNodes konzipiert (rosa dargestellt). Auf diesen sind die NameNode, der ResourceManager und die CheckpointNode instantiiert. Jeder Slaveknoten beherbergt eine DataNode (DN) und einen NodeManager (NM). Je nach Konfiguration kann ein MasterNode auch über DataNode und NodeManager verfügen, dies wird jedoch vermieden, um der NameNode die kompletten Hardwareressourcen eines Knotens zur Verfügung zu stellen [5].

Eine NameNode verwaltet den kompletten Namespace des Clusters [24]. Eine Datei wird in Blöcke von 128 MB aufgeteilt und dabei dreimal repliziert (Blockgröße und Replikationsfaktor können konfiguriert werden). Aus diesem Grund hält die NameNode zu jeder Datei fest, in welche Blöcke sie aufgeteilt ist und wo sich die Blöcke (und ihre Replikationen) physikalisch im Cluster befinden.

Wenn ein Client eine Datei aus dem HDFS lesen möchte, so muss er erst Kontakt mit der NameNode aufnehmen. Diese teilt ihm mit, wo sich die gesuchten Blöcke mit allen Replikationen befinden. Der Client wählt dann die Blöcke aus, die für ihn über den Netzwerkweg am Schnellsten zu erreichen sind und liest die Daten direkt von der jeweiligen DataNode. Wenn ein Client eine Datei ins HDFS schreiben möchte, so muss er für jeden Block, den er erzeugen möchte, bei der NameNode anfragen. Diese nominiert drei DataNodes, die alle den gleichen Block speichern sollen. Der Client schreibt dann seine Daten in die erste DataNode. Daraufhin gibt diese die Daten an die zweite weiter. Diese wiederum nimmt Kontakt mit der dritten DataNode auf. Wenn der letzte Schreibvorgang beendet ist, bestätigen die DataNodes der NameNode, dass sie den Block erfolgreich geschrieben haben (siehe Abbildung 3.5).

Die NameNode speichert die Metadaten des Dateisystems in eine Datei namens `fsimage`, welche im lokalen Dateisystem gespeichert wird. `fsimage` wird in anderen Quellen auch `checkpoint` genannt. Alle Änderungen am Dateisystem (Lösch- oder Einfüge-Operationen von Blöcken) werden nicht in `fsimage` gespeichert, sondern in einer Datei namens `journal`. `fsimage` wird dabei von der NameNode nicht geändert. Nur bei einem NameNode-Neustart liest die NamenNode `fsimage` erneut, führt alle Aktualisierungen, die in `journal` festgehalten sind durch und speichert `fsimage`. Wenn eine NameNode eine Woche ohne Unterbrechung ausgeführt wurde, so kann das

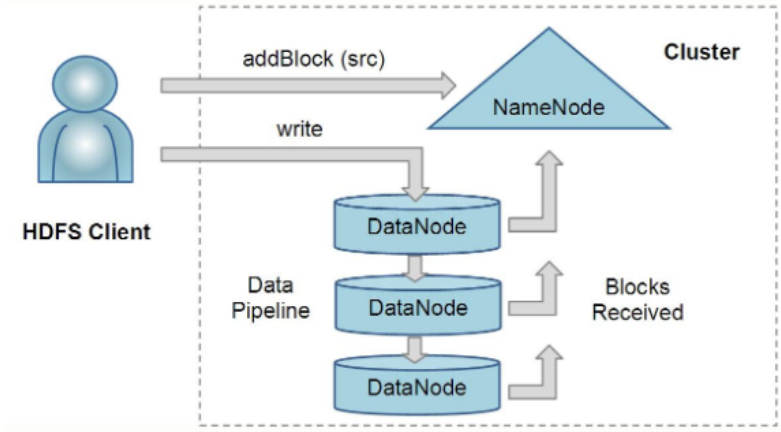

Abbildung 3.5: Ablauf eines Schreibvorgangs eines Clients ins HDFS [24]

Einspielen des `journals` bei Neustart bis zu einer Stunde dauern. Aus diesem Grund wurden *CheckpointNodes* eingeführt. Diese lesen in periodischen Abständen `fsimage` und `journal` und migrieren alle Änderungen aus dem Journal in `fsimage`. Alternativ zu einer CheckpointNode kann auch eine zweite NameNode oder eine BackupNode verwendet werden. Beide haben die gleiche Funktionalität wie eine CheckpointNode.

Jede DataNode prüft die Blöcke in ihrem Cluster regelmäßig auf Integrität und sendet regelmäßig sogenannte Heartbeats an die NameNode. Diese erkennt somit, welche DataNodes verfügbar sind. Über sogenannte Blockreports der DataNodes wird die NameNode zusätzlich über den Zustand der einzelnen Blöcke unterrichtet. Die NameNode kann hierbei berechnen, ob der Replikationsfaktor jedes Blocks mit dem Soll-Wert übereinstimmt. Falls Replikationen fehlen, wählt die NameNode einen oder mehrere DataNodes aus, die Blockreplikationen erstellen sollen. Falls zu einem Block zu viele Kopien bestehen, wird die NameNode die überzähligen löschen lassen. Die DataNodes überwachen sich selbst, melden sich selbstständig bei der DataNode, die immer für einen korrekten Zustand der Daten sorgt [24].

3.3 MapReduce in Hadoop

In Kapitel 3.1 wurde erläutert, wie Hadoop Applikationen über ein Cluster verteilt ausführen kann. Im Folgenden geht es um den Inhalt der Applikationen, hierbei beschränken wir uns auf MapReduce Jobs. Die Umsetzung von MapReduce in Hadoop folgt dem MapReduce Framework, das in Kapitel 2.3 behandelt wurde. Die erforderlichen Map- und Reduce- Funktionen können in verschiedenen Programmiersprachen geschrieben werden, wie Java, Python oder C++. Als Beispiel wird das WordCount Beispiel aus Kapitel 2.3 betrachtet, das in Java Code umgesetzt wurde [5](siehe Anhang).

Die Map Funktion bekommt ein Key/Value Paar als Argument übergeben und gibt als Ausgabe eine Liste von Key/Value Paaren aus. Die Argumente der Reduce-Funktion sind: einen Key und eine Liste von Values. Die Ausgabe des Reducers ist ein Key und ein Value. In der Main Funktion erkennt man, dass ein Combiner verwendet wird (Zeile 61).

In Kapitel 2.3 wurde erwähnt, dass Programmierern mit dem MapReduce Framework viel Arbeit abgenommen wird. Diese müssen nur Map- und Reduce-Funktionen programmieren. Um alle anderen Dinge, wie die verteilte Speicherung der Daten oder der Umgang mit Fehlern, müssen sich Anwender keine Gedanken machen, dies wird vom jeweiligen System übernommen. Trotzdem werden für ein simples WordCount Beispiel über 60 Zeilen Code benötigt. Bei komplexeren Datenabfragen, wie sie in dieser Arbeit durchgeführt werden sollen, müssten viele Mapper und Reducer programmiert werden. Um diesem Problem zu begegnen, wurden Tools entwickelt, die Datenabfragen für den Anwender viel einfacher machen [26]. Diese Tools heißen *Hive* und *Pig*. Ziel dieser Tools ist es, dass Anwender weniger Zeit für die Programmierung benötigen und mehr Zeit für die Analyse zur Verfügung haben. Programme, die in Hive und Pig geschrieben wurden, werden bei ihrer Ausführung in MapReduce-Funktionen übersetzt. Voraussetzung für Hive ist, dass strukturierte Daten zur Verfügung stehen.

Bei Hive wird SQL-ähnlicher Programm-Code geschrieben, diese deklarative Abfragesprache nennt sich *HiveQL*. Das Wordcount Beispiel sieht in HiveQL folgendermaßen aus:

```
1 SELECT wordtable.word, count *
2 FROM wordtable
3 GROUP BY wordtable.word;
```

Listing 3.1: Umsetzung des Wordcount Beispiels in Hive

Im Jahr 2006 hat Yahoo! ein weiteres Tool entwickelt, das der vereinfachten Ausführung von MapReduce Abfragen dient [26]. Dieses nennt sich Pig und funktioniert im Stile einer imperativen Skriptsprache, die ebenfalls Ähnlichkeiten zu SQL aufweist. Der Programmierer kann hierbei genau angeben, mit welchen Operationen sich die Daten Schritt für Schritt ändern sollen. Die Umsetzung des Wordcount Beispiels in Pig sieht folgendermaßen aus:

```
1 b= GROUP a BY word;
2 c= FOREACH b GENERATE FLATTEN group, COUNT(a);
```

Listing 3.2: Umsetzung des Wordcount Beispiels in Pig

Das Beispiel zeigt, dass man den ursprünglichen MapReduce-Code stark minimieren kann. Pig hat gegenüber Hive einen großen Vorteil: Es kann auch mit unstrukturierten Daten umgehen. In dieser Arbeit soll Process Mining mit unstrukturierten Ausgangsdaten im .csv Format durchgeführt werden. Pig kann diese problemlos einlesen. Deswegen wird im weiteren Verlauf dieser Arbeit Pig für die Entwicklung von MapReduce-Abfragen verwendet.

3.4 Apache Pig

Der Name des Tools wurde bewusst gewählt, so soll Pig laut Hadoop-Entwicklern ähnliche Eigenschaften wie Schweine haben [27]: Schweine fressen alles (Pig kann mit strukturierten und unstrukturierten Daten umgehen), Schweine leben überall (Pig läuft nicht nur in Verbindung mit MapReduce, sondern auch mit anderen parallelen Frameworks) und Schweine sind gut an den Menschen angepasst (Pig ist einfach in der Bedienung und kann über vom Benutzer selbst definierte Funktionen leicht erweitert werden).

Die Programmiersprache selbst nennt sich Pig Latin. Es gibt drei verschiedene Möglichkeiten, Pig anzuwenden: entweder man führt Pig in einer Kommandozeile (Shell) aus, die sich *grunt* nennt. Anweisungen können hier einfach nacheinander eingegeben werden. Es können auch Dateien erstellt werden, die Pig Anweisungen enthalten, sogenannte Pig-Skripte. Der Vorteil hierbei ist, dass diese abgespeichert und wiederverwendet werden können, was vor allem für größere Abfragen nützlich ist. Eine weitere Möglichkeit ist es, Pig Latin in den Code anderer Programmiersprachen einzubetten, sogenanntes *Embedded Pig Latin*. Es ist beispielsweise möglich, ein Java-Programm zu schreiben und darin Pig ausführen zu lassen. Das übermittelte Pig Latin wird vom sogenannten *Pig Latin Compiler* optimiert und anschließend in MapReduce-Code umgewandelt.

Pig Latin ist eine Datenflusssprache [26]. Am Anfang werden einer oder mehrere Datensätze mittels einer *LOAD*-Operation geladen, dann werden die gewünschten Operationen auf die Datenmenge angewandt. Die Ausgabe der Ergebnisdaten erfolgt mittels *DUMP*- oder *STORE*-Operation. Die Befehle werden nacheinander in der angegebenen Reihenfolge ausgeführt. Schleifen oder bedingte Anweisungen sind nicht möglich. Der Pig Latin Compiler kann aber Änderungen in der Ausführungsreihenfolge vornehmen. Folgender Beispiel-Code zeigt die Funktionsweise von Pig:

```
1 a = LOAD 'file.txt';
2 ...
3 b = FILTER a BY ...;
4 ...
5 c = GROUP b BY ...;
6 ...
7 DUMP c;
8 ...
9 STORE b INTO out1;
```

Listing 3.3: Beispiel für die Funktionsweise von Pig

Im Folgenden wird das Datenmodell von Apache Pig vorgestellt.

Datentypen Pig unterscheidet zwischen *skalaren Datentypen*, die einzelne Werte enthalten und *komplexen Datentypen*, die andere Datentypen enthalten [28]. Die skalaren Datentypen sind identisch mit denen, die in den meisten Programmiersprachen vorkommen. Die wichtigsten sind *int* und *long* für Ganzzahlen, *float* und *double* für Fließkommazahlen und *chararray* für Zeichenketten.

Pig kennt drei komplexe Datentypen: *maps*, *tuples* und *bags*. Jeder dieser komplexen Datentypen kann andere (skalare oder komplexe) Datentypen enthalten, siehe Abbildung 3.6. Sie haben folgende Charakteristiken [28]:

- *Map* besteht aus einem chararray und einem Datenelement. Das Datenelement kann aus jedem skalaren oder komplexen Datentyp bestehen. Das chararray ist ein Schlüssel, und dient als Index, um das Datenelement zu finden. Eine Map-Konstante kann beispielsweise folgendermaßen aussehen: **['name'#'peter',**

'schuhgroesse'#'46']. Dies stellt ein Map mit zwei Schlüsseln dar (*name* und *schuhgroesse*). Das erste Datenelement ist ein chararray, das zweite ein int.

- *Tuple* hat eine festgelegte Länge und besteht aus einer geordneten Sammlung von Datenelementen. Diese Datenelemente können von jedem beliebigen Typ sein. Ein Tuple wird in *Fields* unterteilt, jedes Field enthält ein Datenelement. Im Vergleich mit einer Tabelle kann ein Tuple als eine Tabellenzeile angesehen werden. Ein Beispiel für eine Tuplekonstante mit zwei Fields ist: **('peter',46)**
- *Bag* ist eine ungeordnete Sammlung von Tuples, daher können einzelne Tuples nicht referenziert werden. Eine Bagkonstante mit drei Tuples mit je zwei Fields kann folgendermaßen aussehen: **('peter',46),('paul',42),('anna',35)**. Bags erhält man, wenn man Daten gruppiert, wobei jede einzelne Gruppe einen bag ergibt.

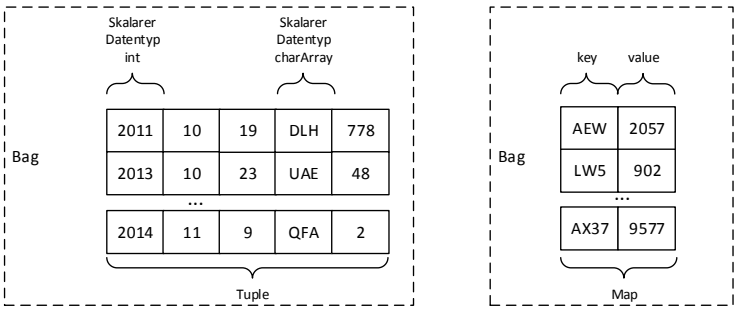

Abbildung 3.6: Verschiedene Datentypen in Pig Latin

Input und Output Im Folgenden wird die Funktionsweise der bereits eingeführten Load-, Store- und Dump-Anweisungen betrachtet, die für die Ein- und Ausgabe von Daten sorgen.

```
1 a= LOAD 'data.csv' USING PigStorage(',');
2 STORE z INTO 'result.csv' USING PigStorage(',');
3 DUMP z;
```

Listing 3.4: Beispiel für ein LOAD, STORE und DUMP-Anweisungen

- Mit der *Load*-Anweisung können Daten aus dem HDFS geladen werden. Dabei kann optional eine Funktion angegeben werden, mit der die Daten geladen werden sollen. Diese Anweisung in Zeile 1 lädt die Datei data.csv aus dem aktuellen Verzeichnis und kann dabei ein Komma als Separator interpretieren. Hier können beliebige Separatoren und Dateipfade angegeben werden.
- *Store* Dies ist das Äquivalent zur Load Anweisung, ist aber für die Datenspeicherung zuständig. Zeile 2 speichert z mittels PigStorage als .csv Datei ab.
- *Dump* Diese Anweisung dient dazu, Daten nicht zu speichern, sondern nur auf dem Bildschirm auszugeben. Dies kann zu Testzwecken sehr hilfreich sein. Eine Ausgabe von z auf dem Bildschirm würde so erfolgen wie in Zeile 3.

Relationale Operatoren Relationale Operatoren sind die wichtigsten Werkzeuge, die Pig Latin bereitstellt, um Daten zu transformieren. Im Folgenden werden einige vorgestellt [28]:

- *Foreach* Diese Anweisung bekommt eine Menge von Records übergeben. Für jeden einzelnen Record nimmt er gewisse Änderungen vor und produziert neue Records. Mit folgender Anweisung werden die Records in a so umgeformt, dass in b nur noch die ersten beiden Fields übrig bleiben.

```
1 a= LOAD 'input' AS (name, vorname, alter, schuhgroesse);
2 b= FOREACH a GENERATE name, vorname;
```

Listing 3.5: Beispiel für FOREACH

Referenzen auf Fields können entweder über den Namen, wie oben gesehen erfolgen, oder aber über die Position. Angeführt vom $ Zeichen wird die Positionsnummer angegeben (beginnend bei 0). Dann könnte die zweite Zeile so aussehen:

```
1 a= LOAD 'input' AS (name, vorname, alter, schuhgroesse);
2 b= FOREACH a GENERATE $0,$1;
```

Listing 3.6: Beispiel für Referenz auf Fields

- *Filter* Diese Anweisung erlaubt es festzulegen, welche Records weiterverwendet werden und welche verworfen werden. Bei jedem Record wird geprüft, ob es eine angegebene Bedingung erfüllt oder nicht. Die Vergleichsoperatoren $==, != , > , >=, <, <=$ können auf jeden skalaren Datentyp angewendet werden. $==$ und $!=$ sogar auf Maps und Tuples. Mit *and* und *or* können mehrere Bedingungen zu einer kombiniert werden. Mit *not* kann das Ergebnis des Vergleichs invertiert werden.
  Bei chararrays kann mittels *matches* ein Vergleich zu einem regulären Ausdruck vorgenommen werden.

```
1 a= LOAD 'input' AS (name, vorname, alter, schuhgroesse);
2 b= FILTER a BY NOT name MATCHES 'Pe.*';
```

Listing 3.7: Beispiel für FILTER

In diesem Beispiel werden in b nur Records gespeichert, bei denen das Field name nicht mit “Pe“ beginnt.

- *Group* Hier gibt der Benutzer einen key an. Alle Records, die über den gleichen Wert eines keys verfügen, werden in einen Bag gespeichert.

```
1 personen= LOAD 'input' AS (name, vorname, alter, schuhgroesse);
2 grpd= GROUP personen BY alter;
3 cnt= FOREACH grpd GENERATE group, COUNT(personen);
```

Listing 3.8: Beispiel für GROUP

In diesem Beispiel enthält grpd Records, die aus zwei Fields bestehen. Das erste Field ist der key und heißt group. Das zweite Field ist ein Bag mit den gesammelten Records. Mit dem FOREACH Statement in Zeile 3 werden mit COUNT alle Records im Bag gezählt.
Das Group Statement erzwingt in der Regel eine Reduce Phase.

- *Order By* Dieser Befehl sortiert die Ausgabe der Daten.

```
1 personen= LOAD 'input' AS (name, vorname, alter, schuhgroesse);
2 grpd= ORDER personen BY alter;
```

Listing 3.9: Beispiel für ORDER BY

Auch die ORDER BY Anweisung erfordert eine Reduce Phase, da hier wieder alle Records zusammen gruppiert werden müssen.

- *Distinct* Diese Anweisung entfernt alle mehrfach gleich vorhandenen Records.

```
1 personen= LOAD 'input' AS (name, vorname, alter, schuhgroesse);
2 grpd= DISTINCT personen;
```

Listing 3.10: Beispiel für DISTINCT

Distinct hat eine Reduce Phase zur Folge.

- *Join* Dies ist ein sehr wichtiges Arbeitsinstrument in Pig, da es häufig verwendet wird. Dabei werden Records von einem Input mit Records eines anderen Inputs zusammengefügt. Jeder Input erhält einen oder mehrere keys. Wenn die keys bei beiden Inputs gleich sind, dann werden die beiden betreffenden Records zusammengefügt. Die nicht passenden Records werden verworfen.

```
1 schueler= LOAD 'input1' AS (name, vorname, lehrername, klasse);
2 lehrer= LOAD 'input2' AS (name, vorname, klasse);
3 jnd= JOIN schueler BY lehrername, lehrer BY name;
4 res= FILTER jnd BY schueler.name MATCHES 'P.*';
```

Listing 3.11: Beispiel für JOIN

Bei der join Anweisung werden die Fieldnamen beibehalten. Falls der Fieldname nicht mehr eindeutig ist, so wie beispielsweise name, so muss zur Referenzierung der ursprüngliche Input mit angegeben werden, also z.B. *schueler.name*. In der Relation *res* sind also aus *jnd* nur noch die Records, bei denen der Schülername mit “P“ beginnt.

Man kann auch joins mit mehreren keys vornehmen. Diese müssen bei beiden Inputs aber vom gleichen Typ oder von kompatiblen Typen sein.

```
1 schueler= LOAD 'input1' AS (name, vorname, lehrername, klasse);
2 lehrer= LOAD 'input2' AS (name, vorname, klasse);
3 jnd= JOIN schueler BY (lehrername,klasse), lehrer BY (name,klasse);
```

Listing 3.12: Beispiel für JOIN mit mehreren keys

Die Join Anweisung erzwingt eine neue Reduce Phase.

- *Flatten* Diese Anweisung wird verwendet, um einen Bag aufzulösen. Dies kommt dann vor, wenn GROUP verwendet wurde, und die Daten weiterverarbeitet werden sollen, z.B.:

```
1 personen= LOAD 'input' AS (name, alter);
2 grpd= GROUP personen BY alter;
3 cnt= FOREACH grpd GENERATE FLATTEN (group);
4 grpd2= GROUP cnt BY name;
```

Listing 3.13: Beispiel für FLATTEN

Ein Record aus grpd kann folgendermaßen aussehen:

(28,{(Maier),(Schneider),(Krumm)})

Da nachher nach Namen gruppiert werden soll, wird FLATTEN angewendet. Die drei Records, die sich aus dem obigen Record bei Ausgabe von cnt ergeben, sehen so aus:

(28,Maier)

(28,Schneider)

(28,Krumm)

Jetzt besteht die Möglichkeit, nach dem zweiten Field zu gruppieren.

- *Nested Foreach* Die Foreach Anweisung ist noch leistungsfähiger als das, was oben bei Foreach vorgestellt wurde. Es kann mehrere relationale Operatoren anwenden auf jeden Record, der ihm übergeben wird.

```
1 schueler= LOAD 'input' AS (name:chararray, notendurchschnitt:float,
      klasse:chararray);
2 grpd= GROUP personen BY klasse;
3 top3= FOREACH grpd {
4                 sorted= ORDER schueler BY notendurchschnitt;
5                 top= LIMIT sorted 3;
6                 GENERATE group, FLATTEN(top);
7 };
```

Listing 3.14: Beispiel für NESTED FOREACH

Obiges Beispiel gruppiert erst alle Schüler nach Klassen. Innerhalb des Nested-Foreach werden dann mehrere Operatoren angewandt: zuerst wird nach Notendurchschnitt geordnet und die besten drei werden herausgesucht. Dann werden die Bags mit FLATTEN aufgelöst. Im Endeffekt werden für jede Klasse die drei besten Schüler ausgegeben. Wie im Beispiel zu sehen ist, muss in der letzten Zeile eines Nested Foreach immer ein GENERATE stehen.

Anzumerken ist hier, dass im Beispiel trotz eines GROUP und eines ORDER BY lediglich eine Reducephase erfolgt. MapReduce kann mit einem primären key gruppieren und einen sekundären key benutzen, um die Daten zu sortieren.

- *Union* Wenn zwei Datensätze aneinander angehängt werden sollen, dann wird das Union Statement verwendet. Wenn beide Datensätze das gleiche Schema haben, dann wird dieses Schema für den resultierenden Datensatz übernommen. In einem Schema werden die Datentypen der einzelnen Fields angegeben. Ist dies nicht der Fall, so prüft der Compiler, ob die Datentypen des einen Schemas in die Datentypen des anderen Schemas überführbar sind. Dieser “Kompromiss“ ist dann das neue Schema. Ist eine Überführung der Datentypen nicht möglich, so erhält der neue Datensatz kein Schema.
  Mit der DESCRIBE Anweisung werden die Schemas der Datensätze ausgegeben.

```
1 a= LOAD 'input1' AS (x:chararray,y:float);
2 b= LOAD 'input2' AS (x:chararray,y:double);
3 c= UNION a,b;
4 DESCRIBE c;
5
6 C:{x:chararray,y:double}
```

Listing 3.15: Beispiel für resultierende Datentypen bei Verwendung von UNION

User Defined Functions (UDF) Zunächst werden die wichtigsten Funktionen vorgestellt, die in Pig schon implementiert sind. Dabei handelt es sich um Aggregatfunktionen, das heißt, dass sie jeweils ein Bag von Werten übergeben bekommen und ein einzelnes Resultat ausgeben.

- *AVG* Bekommt ein Bag von int, long, float oder double Werten übergeben und berechnet den Durchschnittswert. Das Ausgabeformat ist gleich, wie der Eingabetyp.
- *COUNT* bekommt ein Bag mit Werten im beliebigen Format und gibt aus, wie viele Records im Bag sind (Nullwerte ausgenommen).
- *MAX* Eingabe: Bag mit int, long, float, double Werten. Liefert als Ausgabe das Maximum.
- *MIN* Gleich wie bei MAX. Berechnet aber das Minimum.
- *SUM* Eingabewerte wie bei MAX. Berechnet die Summe aller Werte. Der Ausgabetyp ist wieder der gleiche Typ wie in der Eingabe.

Dies ist nur ein Auszug aus allen in Pig eingebauten UDFs.
Es gibt noch eine Sammlung weiterer nützlicher Funktionen, die in der piggybank.jar sind. Diese wird bei einer Pig Installation mitgeliefert. Wenn man eine UDF aus der piggybank.jar verwenden möchte, muss man sie im Pigskript erst registrieren:

```
1 register 'your_path_to_piggybank/piggybank.jar';
2 schueler= load 'input1' AS (name:chararray, vorname:chararray);
3 rueckwaerts= FOREACH schueler GENERATE org.apache.pig.piggybank.
    evaluation.string.Reverse(name);
```

Listing 3.16: Beispiel für die Verwendung der piggybank.jar

Das obige Beispiel wendet die Reverse Funktion aus der piggybank an. Diese sorgt dafür, dass alle Namen rückwärts ausgegeben werden.

Eine der großen Stärken von Pig liegt darin, dass der Nutzer seine eigenen User Defined Functions schreiben kann und in Pig verwenden kann. Diese können mit Java oder Python programmiert werden. Folgender Code zeigt als Beispiel, wie Pig seine integrierte COUNT Funktion in Java umgesetzt hat [28]:

```
1  // src/org/apache/pig/builtin/COUNT.java
2  public Long exec(Tuple input) throws IOException {
3      try {
4          // Erstes Element des Tuples ist ein Bag, dessen
5          //Anzahl an Elementen COUNT zaehlen soll
6          DataBag bag = (DataBag)input.get(0);
7          Iterator it = bag.iterator();
8          long cnt = 0;
9          while (it.hasNext()){
10             Tuple t = (Tuple)it.next();
11             // NULL Werte und leere Tuples nicht mitzaehlen
12             if (t != null && t.size() > 0 &&
13                 t.get(0) != null) {
14               cnt++;
15             }
16         }
17         return cnt;
18     } catch (Exception e) {
19         ...
20     }
21 }
```

Listing 3.17: Beispiel für den Java Code einer UDF

Wenn man eine selbst erstellte Funktion verwenden möchte, muss man sie in Hadoops lokales Dateisystem laden und im Pig Skript registrieren.

MapReduce Plan Für die Ausführung jedes Pig Skripts erstellt Hadoop einen Plan. Dieser Plan legt fest, welche Jobs und in welcher Abfolge sie ausgeführt werden sollen. Jedem Reduce-Job geht ein Map-Job voraus. Wie oben bei den jeweiligen Operationen schon erwähnt, werden Reduce-Jobs in der Regel durch folgende Statements erzwungen: JOIN, CROSS, GROUP, COGROUP, DISTINCT, ORDER BY. Damit hat der Compiler bei der Erstellung des MapReduce Plans in Map- und Reduce-Jobs zwei

Möglichkeiten: die Operationen, die beispielsweise zwischen zwei GROUP-Anweisungen stehen, können entweder im Reduce-Job, der zum ersten GROUP gehört, ausgeführt werden oder im Map-Job, der zum zweiten GROUP gehört [29]. Pig entscheidet sich hierbei immer für die erste Variante. Den Reduce-Jobs werden dadurch mehr Aufgaben zugeschoben. Map-Jobs sollen möglichst nur die JOIN, (CO)GROUP, DISTINCT und ORDER BY Anweisungen umsetzen. Auf diese Art können die Jobs effizienter abgewickelt werden.

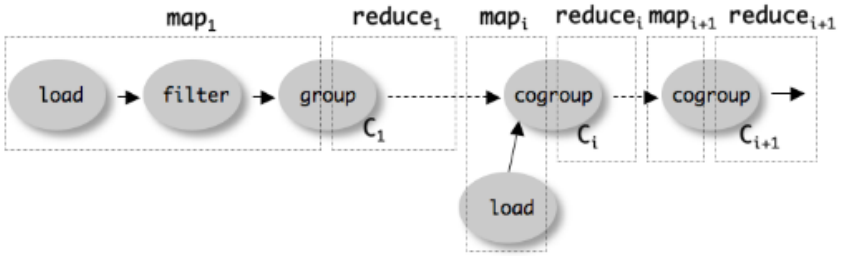

Abbildung 3.7: Beispiel für die Überführung eines Pigskripts in MapReduce [29]

Abbildung 3.7 zeigt, dass die Reduce-Jobs möglichst alle Aufgaben zwischen zwei GROUP Statements übernehmen. Anstelle von GROUP bzw. COGROUP könnten natürlich auch JOIN, DISTINCT, CROSS oder ORDER BY stehen.

In diesem Kapitel wurde mit Apache Hadoop ein Tool vorgestellt, welches das MapReduce Framework umsetzt. Pig ist ein Werkzeug, das dabei hilft, auch komplexe Datenanalyse-Algorithmen wie einen Heuristic Miner kompakt zu programmieren. Trotzdem ist der zu programmierende Code gut nachvollziehbar und leicht erlernbar. Die Übersetzung des Pig Skripts in MapReduce erledigt der Compiler.

Apache Hadoop und Pig werden für die WebApplikation benötigt, die im nächsten Kapitel beschrieben wird.

4

Prototypische Implementierung

In diesem Kapitel wird *ProDoop* vorgestellt, eine WebApplikation, mit der sich auf Basis von MapReduce Process Mining durchführen lässt.

4.1 Problemstellung

Die Grundbeobachtung für die Entwicklung von ProDoop bestand darin, dass es viele Unternehmen gibt, die Event Logs von Prozessen speichern [11]. Zudem verfolgen sie den BPM Lifecycle und möchten ihre Prozessmodelle regelmäßig verbessern. Obwohl Process Mining eine geeignete Möglichkeit zur Prozessentdeckung ist, fehlt es den Unternehmen am nötigen Wissen, wie dieses angewendet wird. ProDoop löst dieses Problem, es erfordert lediglich die Übermittlung von Logdaten und kann darauf Process Mining Algorithmen “per Mausklick” anwenden. Für die Verarbeitung von Daten verwendet es

MapReduce und das Apache Hadoop Framework. ProDoop ist anwenderfreundlich über den Browser zu bedienen, Ergebnisse werden optisch ansprechend in verschiedenen Grafiken dargestellt.

4.2 Softwarearchitektur

ProDoop wird über den Browser bedient. Außerdem wird ein Hadoop Cluster verwendet, dessen HDFS als Datenbank dient und das MapReduce Programme ausführt. ProDoop ist eine Web Applikation. Da Hadoop fast vollständig in Java programmiert wurde [19] und eine umfassende Java API enthält, wird Java als Programmiersprache für ProDoop verwendet.

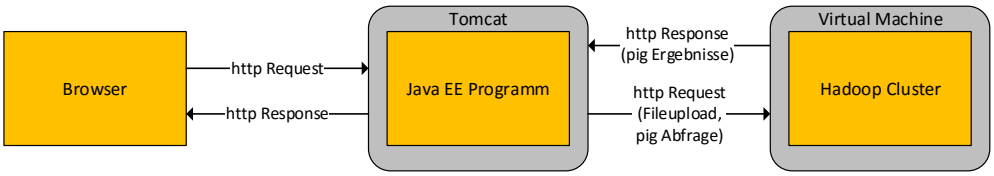

Abbildung 4.1: Architektur der erstellten Softwareapplikation

Als Hadoop-Implementierung wird die Hortonworks Sandbox verwendet: diese stellt eine vorkonfigurierte Hadoop-Umgebung bereit, die speziell dafür entwickelt wurde, Hadoop auf einer VM auszuführen [30]. Ein Webbrowser stellt die Webseiten im HTML-Format dar. Über den Browser kann der Nutzer mit der JavaEE-Applikation interagieren. Die Verbindung erfolgt über HTTP-Request und -Responses.

Außerdem wird ein *Apache Tomcat* Webserver verwendet, auf dem ProDoop läuft. Darüber wird der Kontakt zum Browser aber auch die Verbindung zum Hadoop Cluster hergestellt. Im Folgenden wird der Aufbau von Tomcat betrachtet.

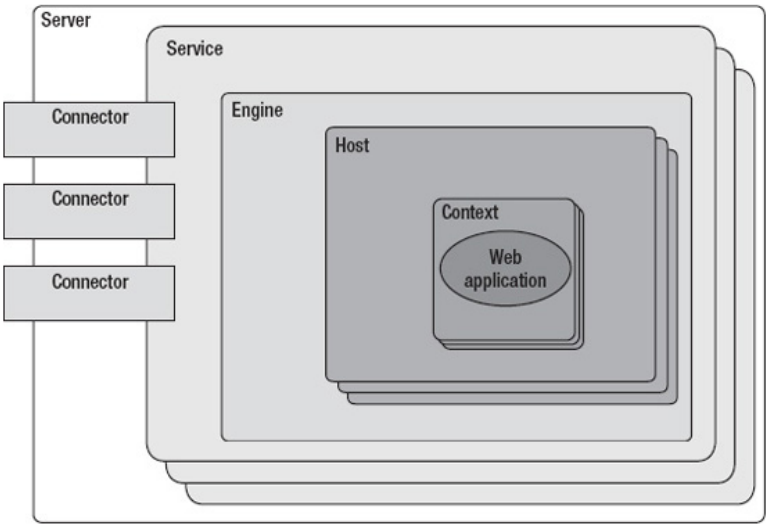

Abbildung 4.2: Aufbau eines Tomcat Webservers [31]

Tomcat-Server

Apache Tomcat ist ein Open-Source Java-Anwendungsserver [32]. Die Architektur von Tomcat besteht aus folgenden Elementen [31]:
Context ist die innerste Komponente. Jedes Context-Element enthält eine einzelne Webapplikation.

Ein *Connector* sorgt mit Hilfe eines TCP-Ports dafür, dass Verbindungen zwischen Applikationen und Clients hergestellt werden. Im vorliegenden Fall ermöglicht er die HTTP-Verbindungen zwischen Webapplikation und Browser sowie zwischen Webapplikation und Hadoop Cluster.

Jede *Host-Komponente* enthält einen *Virtual Host*: dieser ist die Verbindung eines DNS-Namens, wie z.B. www.meineDomain.de mit dem Server. Jeder Server kann mehrere Hosts enthalten. Gleichzeitig kann ein Host mehrere Webapplikationen enthalten. Standardmäßig wird der Host *localhost* verwendet. Dieser wird für die vorliegende Anwendung verwendet.

Eine *Engine* enthält einen oder mehrere Hosts. In der entwickelten Anwendung wird eine Engine namens *Catalina* verwendet [33]. Catalina verarbeitet alle über den Connector

eingehenden HTTP-Requests, leitet sie an den zugehörigen Host weiter und versendet die Responses zurück zum Client.

Die *Service-Komponente* verbindet einen oder mehrere Connectoren mit den zugehörigen Engines. Jede Tomcat-Instanz enthält ein einziges Server-Element. Dieses wiederum beinhaltet eine oder mehrere Service-Komponenten.

Die vorgestellte Hierarchie bietet große Flexibilität für verschiedene Anwendungen. Eine Tomcat-Instanz sowie eine Catalina-Engine reichen für die Anforderungen von ProDoop aus.

Um Process Mining durchführen zu können, wendet ProDoop auf Logdaten Process Mining Algorithmen an. Unter anderem wird der Heuristic Miner aus Kapitel 2.2.1 dazu verwendet.

4.3 Heuristic Mining mit Pig

Im Folgenden wird die Implementierung eines Heuristic Miners in Pig, das in Kapitel 3.4 vorgestellt wird, beschrieben. Hierbei wurden ausschließlich Operationen und Funktionen, die das Pig-Framework zur Verfügung stellt, sowie solche, die in der Piggybank enthalten sind, verwendet.

Die Implementierung als Pig-Skript kann in 3 Teile aufgeteilt werden.

```
1 REGISTER piggybank.jar;
2 callcenter=load '$INPUT' using org.apache.pig.piggybank.
3         storage.CSVExcelStorage(',','NO_MULTILINE', 'UNIX',
4         'SKIP_INPUT_HEADER');
5 /* Diese Anweisung laedt die Inputdatei. Der erste Parameter von
      CSVExcelStorage gibt an, dass KOMMA als Separator verwendet wird,
      damit wird eine .csv Datei geladen. SKIP_INPUT_HEADER gibt an, dass
      die erste Zeile (Ueberschrift) nicht eingelesen wird. */
6 b= FOREACH callcenter GENERATE $0 AS serviceID, $3 AS operation, $1 AS
      startdate, $2 AS enddate;
7 /*in b werden 4 Fields von a uebernommen und benannt: 1.Field serviceID;
      4.Field operation; 2.Field startdate; 3.Field enddate  */
8 start3= GROUP b BY operation;/* b wird nach dem Field operation gruppiert
      ; alle Records einer Gruppe werden in einem bag gespeichert */
9 start4= FOREACH start3 GENERATE group, COUNT(b);/*Generiert operation als
       1.Field in start4; 2.Field: COUNT zaehlt alle Records, die in einer
      Gruppe sind */
```

```
10 STORE start4 AS out1;
11 /* start4 wird als out1 gespeichert. start4 gibt an, welche Operationen
       insgesamt wie oft in der Datenbasis vorkommen */
```

Listing 4.1: Umsetzung des Heuristic Miners in Pig: Teil 1

Der erste Skript-Teil beginnt damit, dass in den Alias `callcenter` die Ausgangsdaten geladen werden. Das Ergebnis wird als `start4` in HDFS gespeichert.

```
1  bWithEpoch= FOREACH b GENERATE serviceID, operation, ToDate(startdate, '
       yyyy/MM/dd H:mm:ss.SSS', 'Europe/Berlin') as epoch1, ToDate(enddate, '
       yyyy/MM/dd H:mm:ss.SSS', 'Europe/Berlin')
2   as epoch2;
3   /* bWithEpoch uebernimmt von b die Fields serviceID und operation; das
        Field startdate wird ins Date-Format umgewandelt und als epoch1
        gespeichert. yyyy/MM/dd H:mm:ss.SSS gibt das Format an, in dem
        startdate gespeichert ist. Das Field enddate wird ins Date-Format als
         epoch2 umgewandelt.*/
4  bWithEpoch2= FOREACH bWithEpoch GENERATE $0 AS serviceID2,
5  $1 AS operation2, $2 AS epoch12, $3 AS epoch22;
6  /* bWithEpoch2 speichert eine Kopie von bWithEpoch, benennt die Fields
       aber um. Dies wird spaeter benoetigt, da Pig keine Selbst-Joins
       durchfuehren kann. */
7  jn= GROUP bWithEpoch BY $0;/* gruppiert bWithEpoch nach serviceID*/
8  B= FOREACH jn{
9          sorted= ORDER bWithEpoch by epoch1 ASC,
10         lim = LIMIT sorted1;
11         GENERATE FLATTEN (lim);
12 };
13 /* NESTED FOREACH: mehrere Anweisungen werden auf jede Gruppe angewandt:
       Erst werden alle Records aufsteigend nach epoch1 geordnet; dann wird
       nur der erste Record uebernommen, alle anderen werden verworfen. Mit
       GENERATE FLATTEN wird der bag aufgeloest und in Tuples umgewandelt.
14 B gibt fuer jede Instanz an, welche Operation als erstes durchgefuehrt
       wurde */
15 C= GROUP B BY operation;/* gruppiert B nach dem Field operation*/
16 D= FOREACH C GENERATE FLATTEN(group) AS (operation),
17         COUNT(B) AS operationcount;
18 /* GENERATE FLATTEN loest den bag wieder auf und wandelt die Datensaetze
       in Tuples um. C enthaelt als 1. Field den Gruppenname: operation; als
       2.Field die Anzahl der Records, die sich im jeweiligen bag befunden
       hat.*/
19 STORE D AS out2;
20 /* D gibt aus, wie oft jede einzelne Operation als Erstes in einer
       Instanz erfolgt ist */
```

Listing 4.2: Umsetzung des Heuristic Miners in Pig: Teil 2

Der 2. Skript-Teil erzeugt eine Ausgabe (`out2`), die für jede einzelne Operation angibt, wie oft sie als erstes in einer Instanz erfolgt ist.

```
1  d= JOIN bWithEpoch BY serviceID, bWithEpoch2 BY serviceID2;
2  /* JOIN von bWithEpoch mit bWithEpoch2 nach serviceID bzw. serviceID2.
       Dies ist im Endeffekt ein JOIN aller Daten mit sich selbst, gruppiert
       nach Instanz. */
3  e= FOREACH d GENERATE $0,$1,$2,$3,$4,$5,$6, MinutesBetween($6,$2) AS
       minutes, $7;
4  /* e uebernimmt alle Fields aus d; Zusaetzlich wird minutes als 8.Field
       aufgenommen. minutes gibt die zeitliche Distanz in Minuten zwischen
       epoch1 (von bWithEpoch) und epoch12 (von bWithEpoch2) an. epoch1 und
       epoch12 sind die jeweiligen Startzeitpunkte der Operationen*/
5  f= FILTER e BY (minutes>=0) AND ($3!=$8);
6  /* f uebernimmt nur diejenigen Records von e, bei denen minutes>=0 ist
       und die Endzeitpunkte der Operationen nicht gleich sind ($3!=$8).
       Damit werden in f Paare von Operationen gespeichert, von denen die
       zweite nach der ersten startet (oder gleichzeitig). Dass die erste und
        die zweite Operation gleich sind, wird ausgeschlossen */
7  g= GROUP f BY ($0,$1,$2);/* f wird gruppiert nach serviceID, operation
       und epoch1 (Startzeitpunkt)*/
8  h= FOREACH g{ sorted= ORDER f BY $7 ASC;
9           lim= LIMIT sorted 1;
10          GENERATE FLATTEN (lim);
11 };
12 /* in jeder Gruppe werden die Records aufsteigend geordnet nach $7 (
       minutes). lim waehlt in jeder Gruppe nur den obersten Record aus.
       GENERATE FLATTEN loest den bag auf und verwandelt den Datensatz in ein
        Tuple.*/
13 /* in h wird somit in jeder Instanz fuer jede darin vorkommende Aktion
       berechnet, welche Operation direkt danach kommt */
14 i= GROUP h BY ($1,$5);/* hier wird h gruppiert nach operation der ersten
       Operation ($1) und nach operation der zweiten Operation ($5)  */
15 j= FOREACH i GENERATE FLATTEN (group), COUNT(h) AS operationcount;
16 /* GENERATE FLATTEN wandelt den bag in einen Tuple um. 1.Field: operation
        der ersten Operation; 2.Field: operation der zweiten Operation; 3.
       Field: Anzahl der Records, die im bag waren */
17 STORE j AS out3;
18 /* out3 gibt fuer jede Kombination von 2 Operationen aus, wie oft
       Operation 1 von Operation 2 gefolgt wird */
```

Listing 4.3: Umsetzung des Heuristic Miners in Pig: Teil 3

Die Ausgabedatei `out3` von Programmteil 3 gibt für jede mögliche Kombination von 2 Operationen aus, wie oft Operation 1 von Operation 2 innerhalb einer Prozessinstanz gefolgt wird.

Somit enthalten out1,out2 und out3 die für den Heuristic Miner benötigten Daten (vgl. Tabelle 2.2 und Tabelle 2.3). Zur Erstellung der Tabellen sind weitere Berechnungen nötig, die von ProDoop durchgeführt werden.

Abbildung 4.3 zeigt den Ablaufplan des Pig-Skriptes für den Heuristic Miner. Die Kreise in Abbildung 4.3 enthalten jeweils einen Alias, unter dem ein Datensatz zwischengespeichert wird. Die Pfeile zeigen an, in welche Richtung eine Transformation der Daten stattfindet. Sie geben außerdem an, mit welcher Anweisung die Transformation erfolgt (z.B. FILTER, FOREACH, GROUP etc.).

`callcenter` ist der einzige Alias, der keine eingehende Kante hat, denn `callcenter` lädt den Ausgangsdatensatz. `start4`, `D` und `j` besitzen keine ausgehenden Kanten. Sie dienen als Ergebnis und werden als `out1`, `out2` und `out3` gespeichert.

Die Programmlogik kann, wie in Abbildung 4.3 zu sehen ist, in 3 Teile unterteilt werden. Jeder Programmteil endet mit der Abspeicherung eines Datensatzes (`start4`, `D` und `j`). In den Transformationen sind hierbei Muster zu erkennen: zu Beginn eines Programmteils wird häufig `FILTER` und `FOREACH` verwendet, damit wird die Datenmenge verringert. Mit `FILTER` werden unnötige Log-Einträge aussortiert und mit `FOREACH` werden unnötige Fields aussortiert (linker und rechter Teil des Skriptes). Die Verringerung der Datenmenge zu Beginn eines Skript-Teils ist nötig, um die Ausführung des Programms effizient zu gestalten

Jeder Programmteil endet mit den 2 aufeinanderfolgenden Anweisungen `GROUP` und `FOREACH,COUNT`. Mit `GROUP` wird nach einem Field gruppiert. Daraus resultieren viele Gruppen. Jede Gruppe enthält den Gruppennamen und einen Bag, in dem alle Records dieser Gruppe enthalten sind. Mit der anschließenden `COUNT` Anweisung werden alle Records gezählt, die in dieser Gruppe gespeichert sind.

Nach Ablauf des linken Skript-Teils erfolgt eine Gruppierung nach *Operation*. Mit `COUNT` wird gezählt, wie viele Records in einer Gruppe sind. Als Ergebnis erhält man, wie oft jede Operation insgesamt in der Datenbasis vorkommt.

Ein weiteres Muster, das im mittleren und rechten Skript-Teil verwendet wird ist die Kombination einer `GROUP` Anweisung mit anschließendem `FOREACH, ORDER`. Damit wird nach einem Field gruppiert. Die resultierenden Bags werden anschließend geordnet.

Im Programm werden die Records innerhalb der Bags zeitlich aufsteigend geordnet. Mit anschließender Verwendung von `LIMIT 1` wird der zeitlich erste Record ausgewählt. Damit wird im mittleren Skript-Teil ermittelt, welche Operation am Anfang einer Instanz erfolgt.

Die erwähnte Zwischenspeicherung von Datensätzen in Aliase und die jeweilige Datentransformation gelten lediglich für die Pig Programmlogik und nicht für die Umsetzung in MapReduce. Das Pig-Skript wird in 6 verschiedene MapReduce Jobs übersetzt. In Abbildung 4.3 sind diese durch gestrichelte, rote Linien dargestellt. In jedem MapReduce Job werden mehrere Pig Anweisungen zusammengefasst. Bei der Aufteilung des Pig-Skripts in MapReduce Jobs berücksichtigt der Compiler, welche Datensätze ausgegeben werden sollen. Diese müssen in HDFS geschrieben werden und somit am Ende eine MapReduce Jobs zur Verfügung stehen. Außerdem erkennt der Compiler, dass im Pig-Skript 5 `GROUP` Anweisungen und eine `JOIN` Anweisung verwendet werden. Da jedes `GROUP` und jeder `JOIN` eine eigene Reduce Phase zur Folge hat (siehe Kapitel 3.4), erstellt der Compiler 6 MapReduce Jobs. Wie in Abbildung 4.3 zu erkennen ist, erfolgen in 3 verschiedenen MapReduce Jobs `COUNT` Anweisungen. Diese haben zur Folge, dass in den betreffenden MapReduce Jobs zwischen der Map Phase und der Shuffle Phase ein Combiner zum Einsatz kommt (vgl. Kapitel 2.3 und Kapitel 3.4).

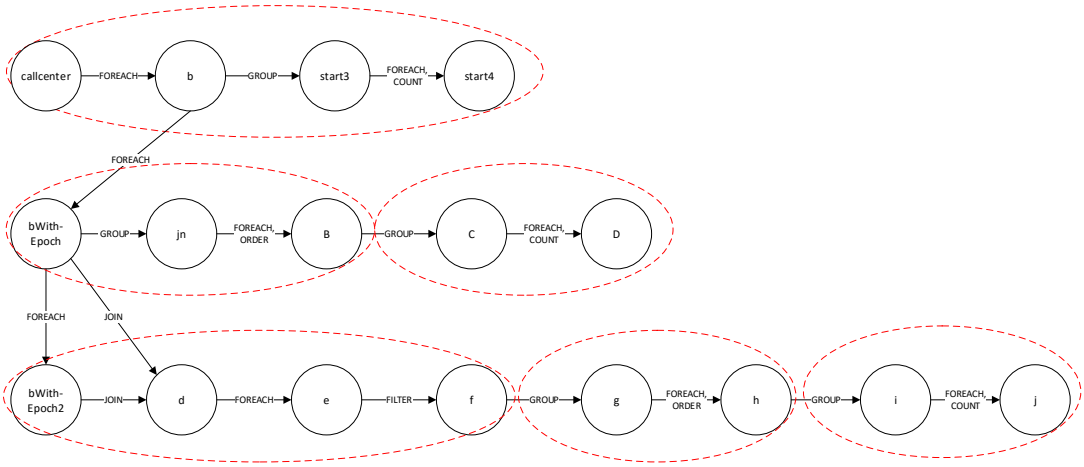

Abbildung 4.3: Ablaufplan des Heuristic Miners in Pig

4.4 Umsetzung in JavaEE

Im Folgenden wird erläutert, wie ProDoop in JavaEE umgesetzt wird. JavaEE erlaubt es, dynamische Webseiten zu erstellen, wie sie für ProDoop benötigt werden. Der Anwender kann über den Browser Auswahlentscheidungen treffen, die an den Server per HTTP-Request übermittelt werden. Abhängig von den Auswahlentscheidungen des Nutzers und von den Ergebnissen des Hadoop Clusters werden die resultierenden HTML-Seiten jedes Mal neu erzeugt und per HTTP-Response an den Browser gesendet.

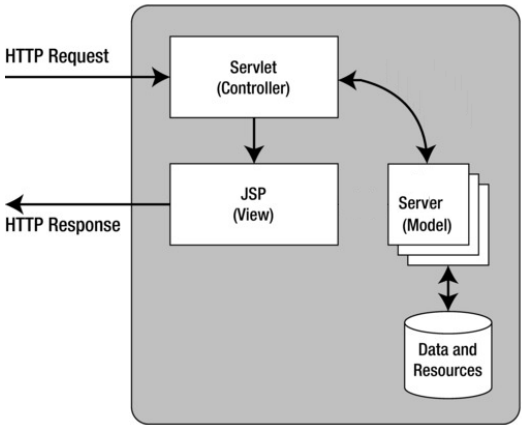

Abbildung 4.4: Das MVC Pattern umgesetzt mit JavaEE (agelehnt an [32])

Bei ProDoop wurde das *Model-View-Controller Pattern (MVC Pattern)* wie in Abbildung 4.4 dargestellt umgesetzt. Ein HTTP-Request wird vom *Servlet* entgegengenommen. Dieses dient als Controller und ist ein Vermittler zwischen Model und View. Je nachdem, welche Informationen das Servlet vom Browser erhält, wird es entweder gleich eine Antwort via JSP-Datei zurücksenden oder eine Funktion aus dem Model aufrufen. Eine JSP-Datei besteht aus HTML-Code und JSP-Elementen, es enthält keinerlei Logik. Als JSP Code kann jeglicher Java Code eingebettet werden. Die JSP Datei wird vom Tomcat Server in HTML Code übersetzt und an den Browser gesendet. Der dynamische JSP Code kann also nur vom Server übersetzt werden, nicht vom Webclient, der nur das Ergebnis darstellt.

Im Controller ist der größte Teil der Programmlogik enthalten. Das Model und der Controller bestehen aus normalen Java Klassen. In ProDoop ist der Controller dafür

zuständig, Kontakt mit dem Hadoop Cluster aufzunehmen. Der Controller kann Daten ins HDFS hochladen, Daten aus dem HDFS laden, Pig-Anfragen an Hadoop senden und Ergebnisse entgegennehmen. Der Controller kann Funktionen aus dem Model aufrufen, die Berechnungen durchführen. Die Berechnungen werden anschließend wieder vom Controller entgegengenommen. Das Servlet kann dann wieder eine JSP-Datei aufrufen.

Die Kommunikation zwischen Browser und Webapplikation erfolgt über HTTP-Requests und -Responses. Das HTTP-Protokoll ist *statuslos*, das heißt, dass Informationen über die vorherige Kommunikation nicht gespeichert werden [31]. In der Applikation müssen aber Informationen, die der User durchführt (z.B. über die ausgewählte Datei) auch zu einem späteren Zeitpunkt verfügbar sein. Dieses Problem wird durch die sogenannte *session-Variable* gelöst: dabei hält der Server Informationen in einem *session-Objekt* gespeichert.

Im Folgenden wird der genaue Ablauf von ProDoop vorgestellt. Ein User beginnt mit einem Loginformular. Bei erfolgreichem Login wird der User auf eine Seite geführt, auf der er wählen muss, auf welcher Datei er eine Datenanalyse durchführen möchte.

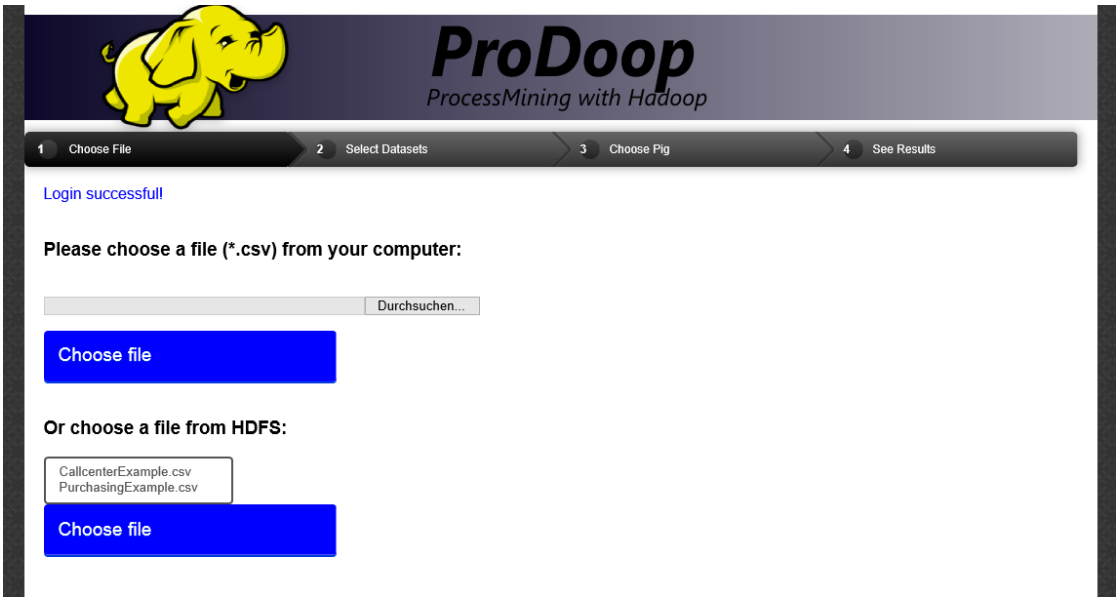

Abbildung 4.5: Auswahl einer Datei in ProDoop

Möglich sind nur .csv Dateien. Der User hat zwei Optionen: entweder er lädt eine Datei von seiner Festplatte ins HDFS hoch, oder er wählt eine Datei aus, die bereits hochgeladen ist. Die JSP-Datei *FileUploader.jsp* setzt die Darstellung im Browser um und nimmt die Wahl des Users entgegen. Wenn der User auf den *Choose File* Button klickt, dann wird die Auswahl des Users an das Servlet namens *UploadServlet* übermittelt. *UploadServlet* kann die Auswahl des Users mit der *doPost* Methode abgreifen. Dieses instanziiert dann ein neues *HadoopAnalyzerService*. Dies ist Teil des ProDoop-Controllers und wird benötigt, um eine Datei ins HDFS hochzuladen.

HDFS Java API Um mit dem Hadoop-HDFS zu interagieren, wird die Klasse *FileSystem* benötigt [14]. In ProDoop ruft *HadoopAnalyzerService* eine weitere Klasse *uploadFileToHDFS* auf. Diese enthält eine gleichnamige Methode, die für einen Upload von *.csv-Dateien ins HDFS zuständig ist.

Wichtige Funktionen aus der Klasse `Filesystem` sind [14]:

- public void `copyFromLocalFile`(Path src, Path dst)
  Diese Methode kopiert eine Datei vom lokalen Dateisystem ins HDFS.
- public FSDataInputStream `open`(Path f)
  Mit dieser Methode kann eine Datei vom HDFS gelesen werden.
- public FileStatus[] `listStatus`(Path f)
  Diese Methode liefert alle Dateien, die sich im angegebenen Verzeichnis f befinden.
- public boolean `delete`(Path f, boolean recursive)
  Diese löscht eine Datei oder ein Verzeichnis. Falls für die Variable `recursive` gilt: recursive==true, so wird ein Verzeichnis mit seinem gesamten Inhalt gelöscht.

Wenn ein User eine Datei im HDFS ausgewählt hat, dann erfolgt der nächste Schritt. Das Programm benötigt noch Information, welche Bedeutung den einzelnen Spalten der zuvor hochgeladenen .csv-Datei zukommt.

Die Darstellung der betreffenden JSP-Datei im Browser ist in Abbildung 4.6 dargestellt. Auf der linken Seite wird dem User ein Auszug aus der ausgewählten .csv-Datei gezeigt.

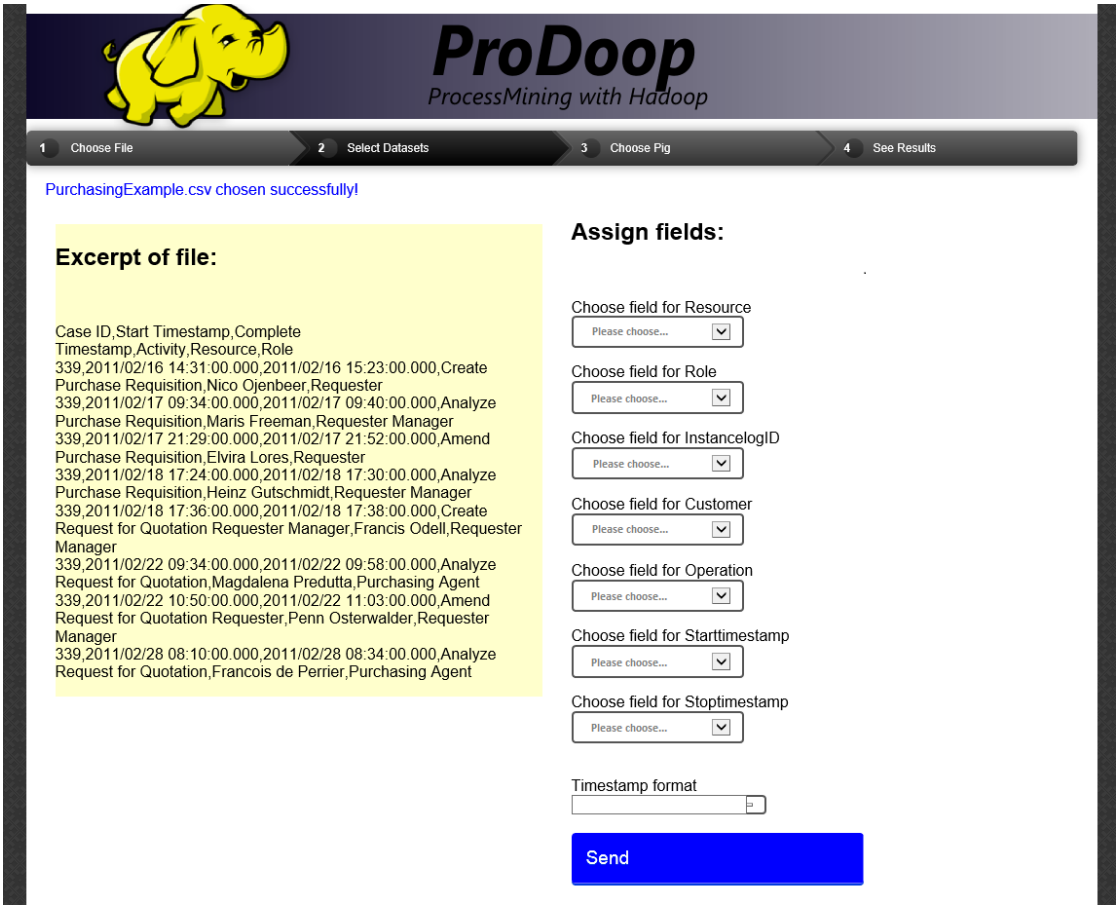

Abbildung 4.6: Zuordnung der einzelnen Fields in ProDoop

Dieser soll dabei behilflich sein, die Zuordnungen auf der rechten Seite zu erstellen. ProDoop arbeitet intern mit folgenden Feldbezeichnungen:

- *Resource*: Eine Ressource eines Business Processes
- *Role*: Zuordnung der Resource zu einer Rolle
- *InstancelogID*: Bezeichner einzelner Instanzen
- *Customer*: Kunde (optional, nur verwendet, falls eine entsprechende Auswertung ausgewählt wurde)
- Eine *Operation*: Name einer Aktivität
- Die *Starttimestamp*: Beginn einer Aktivität

- *Stoptimestamp*: Ende einer Aktivität

Per Dropdown-Menü ordnet der User zu, welche Felder aus seiner gewählten Datei zu den von ProDoop verwendeten Feldbezeichnungen passen. Er kann auch Feldzuordnungen leer lassen, die nicht zugeordnet werden sollen oder können. Außerdem benötigt ProDoop die Formatierung der beiden Timestamps. Diese wird in Form eines *regulären Ausdrucks* angegeben [34].

Auf Grundlage dieser Zuordnungen bietet ProDoop in der nächsten Ansicht mehrere vorimplementierte Pig-Analysefunktionen an. Zur Auswahl stehen folgende Analysen:

- *Frequency of starting operations* untersucht den Beginn aller Instanzen; gibt aus, mit welcher Häufigkeit die Aktivitäten am Anfang einer Instanz auftreten.
- *Handover of work for a single person*: Der User wählt einen Mitarbeiter aus, von diesem wird die Häufigkeit und Art der Übergabe und Übernahme von anderen Mitarbeitern ausgewertet und grafisch dargestellt.
- *Handover of work for a single role* Ähnlich zur Handover of work-Analyse, jedoch wird hier eine Rolle ausgewählt.
- *Social Network Analysis* führt eine Social Network Analyse wie in Kapitel 2.2.2 vorgestellt durch.
- *Social Network Analysis Rolelevel*: eine Auswertung erfolgt wie bei der Social Network Analysis, jedoch in Bezug auf die Role.
- *Resource Frequency and total time* berechnet, wie oft jede einzelne Resource insgesamt tätig war und wie lange deren Arbeitszeit (ohne Wartezeit) war.
- *Activity Frequency and total time*: wie oben, jedoch werden die Häufigkeit und Gesamtzeit jeder einzelnen Aktivität berechnet.
- *Role Frequency and total time*: hier wird auf die Rolle Bezug genommen.
- *Heuristic Miner* wendet den in Kapitel 4.3 vorgestellten Heuristic Miner an.

Mit der Auswahl einer Möglichkeit, führt ProDoop eine Pig-Analysefunktion aus und stellt das Ergebnis anschließend im Browser dar.

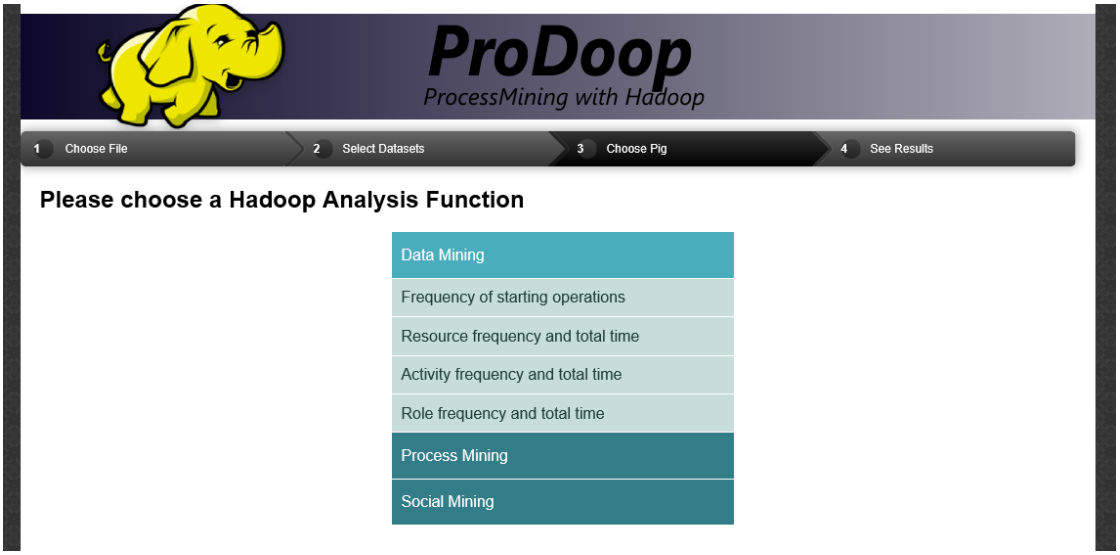

Abbildung 4.7: Hadoop Analysefunktion auswählen in ProDoop

Pig Java API Um eine Pig Abfrage durchzuführen, muss ProDoop Kontakt mit dem Hadoop Cluster aufnehmen. Dies wird vom Controller von ProDoop übernommen. ProDoop berücksichtigt außerdem die von einem User durchgeführten Feldzuordnungen. Der Controller sendet die Pig Abfrage ans Hadoop Cluster und empfängt anschließend die Resultate von diesem. Die Kommunikation zwischen ProDoop und Hadoop Cluster wird durch die *Pig Java API* ermöglicht.

Um eine Pig Abfrage an Hadoop zu übermitteln, kann die Klasse `PigServer` verwendet werden [27]. Diese bietet zwei Methoden:

- public void `registerScript`(java.lang.String fileName, java.util.Map<java.lang.String, java.lang.String> params) throws java.io.IOException
  Als *fileName* wird ein Pig-Skript angegeben, darin können Variablen eingebaut werden. Mit *params* wird eine Map angegeben, die festlegt, wie die Variablen ersetzt werden sollen.
- public void `registerQuery`(java.lang.String query) throws java.io.IOException
  übergibt einen Pig Latin Ausdruck mit der Variablen *query* als String und registriert diesen. Das Resultat der Abfrage kann mittels `openIterator()` empfangen werden.

In ProDoop wird die Methode `registerQuery` verwendet, da die Verwendung von Variablen im vorliegenden Anwendungsfall wesentlich einfacher ist als mit der Methode `registerScript`. Folgender Programmauszug zeigt die Verwendung:

```
1  //...
2  import org.apache.pig.ExecType;
3  import org.apache.pig.PigServer;
4  import org.apache.pig.data.Tuple;
5
6  public class ersteAktionVergleichen {
7  public int ersteAktionVergleichen(HttpSession session, String str) throws
       IOException{
8          //...
9          String s="Startoperationen: <br>";//der Ergebnisstring s wird
               instantiiert
10         PigServer pigServer = new PigServer(ExecType.MAPREDUCE);
11         //eine neue Instanz der PigServer Klasse wird gestartet;
               MapReduce wird zur Ausfuehrung von Pig gewaehlt
12         pigServer.registerQuery("file = load '/user/hue/filesforjava/"+(
               String)session.getAttribute("chosenFile")+"' USING PigStorage
               (',');");
13         //mit registerQuery wird Zeile fuer Zeile das Pig-Skript
               uebermittelt
14         pigServer.registerQuery("a= FILTER file BY NOT ($0 MATCHES '"+(
               String)session.getAttribute("firstLineFirstColumn")+"');");
15         pigServer.registerQuery("b = FOREACH a GENERATE $"+session.
               getAttribute("instancelogRow").toString()+" as serviceID, $"+
               session.getAttribute(str+"Row").toString()+" as operation , $"
               +session.getAttribute("timestamp1Row").toString()+" as date;")
               ;
16         //hier steht das restliche Pig-Skript mit registerQuery
17         pigServer.registerQuery("D= FOREACH C GENERATE FLATTEN(group) AS
               (operation), COUNT(B) AS operationcount;");
18         //eine Store oder Dump Anweisung am Ende ist nicht noetig
19         Iterator<Tuple> iterator = pigServer.openIterator("D");
20         //openIterator fuer Alias D wird angewandt. Damit wird das
               gewuenschte Ergebnis des Pigskripts abgerufen
21         while(iterator.hasNext()){
22                 Tuple tuple = iterator.next();
23                 //das Ergebnis wird in einzelne Tuple aufgeteilt
24                 if(tuple.get(0).toString()!=null){
25                 //...
26                 s=s+tuple.get(0).toString()+"  "+tuple.get(1).toString()+
                       "<br>";
27                 }//jedes Tuple wird als String an den Ergebnissting
                       angehaengt
28         }
29         //...
30         session.setAttribute("resultString",s);
31         //der Ergebnisstring wird als Sessionvariable gespeichert
32 }
33 }
```

Listing 4.4: Anwendung von registerQuery und open Iterator aus der PigServer Klasse

Mit mehreren Aufrufen von Methode `registerQuery` wird nach und nach die jeweilige Pig Abfrage abgearbeitet. Variablen können als String eingefügt werden. Mit der Methode `pigServer.openIterator("x")` kann das Ergebnis abgerufen werden [27]. Der übergebene Parameter *x* gibt an, für welchen Alias der Iterator geöffnet werden soll. Zurückgegeben wird ein Iterator von Tupeln. Im Programm werden alle Tupel durchiteriert. Der Inhalt jedes Tupels wird als String in die Variable *s* gespeichert. Diese wird am Ende als Session Variable gespeichert. Diesen String ruft die JSP-Datei, die für die Darstellung des Ergebnisses zuständig ist, wieder ab. Damit ist eine textuelle Ausgabe des Resultats möglich. Um eine optische Darstellung des Resultats zu ermöglichen werden von der Modelseite die nötigen Daten als Arrays in einer Session Variable gespeichert und können dann von der JSP-Datei weiterverarbeitet werden. Zur Darstellung des Ergebnisses als Diagramm oder als Graph wird Javascript Code in die JSP-Datei eingebettet. Sieht man von der Darstellung von Graphen ab (Heuristic Miner und Social Network), so kann das Chart.js-Framework, das im Folgenden vorgestellt wird, die jeweiligen Ergebnisse ansprechend darstellen.

Chart.js Die Javascript Bibliothek *Chart.js* bietet die Möglichkeit, verschiedene Darstellungsformen für die Resultate zu wählen [35]. In ProDoop werden der *Doughnut-*, *Radar-* und der *Bar-Chart* verwendet. Der Doughnut Chart ist dann interessant, wenn man relative Anteile von mehreren Größen zueinander vergleichen möchte.

Das Beispiel in Abbildung 4.8 zeigt, welche Startoperationen anteilig in einem Event Log vorkommen (Datenbasis: CallcenterExample.csv). Es ist also zu sehen, dass in dem untersuchten Callcenter ein Fall überwiegend durch eingehende Anrufe gestartet wird (Inbound Call, 3291). Emails (Inbound Email, 385 (blau) und Handle Email, 163 (gelb)) stehen viel seltener am Anfang eines Falls.

Eine andere Möglichkeit, Daten darzustellen, bietet der Radar-Chart.

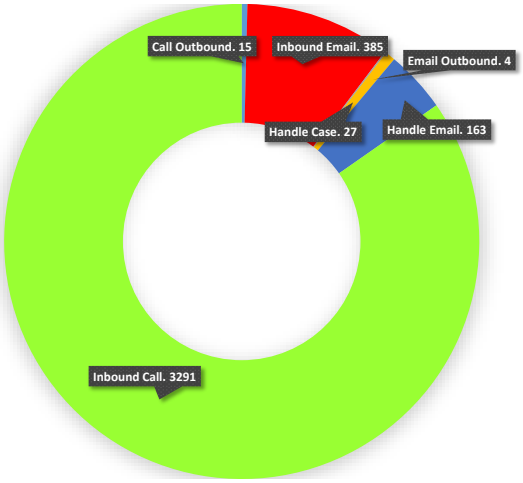

Abbildung 4.8: Beispiel für einen Doughnut Chart in ProDoop

Abbildung 4.9: Beispiel für einen Radar Chart in ProDoop

Wird als Pig Abfrage *Handover of work for a single person* gewählt und dann als Person beispielweise *Maris Freeman* ausgewählt (bei Verwendung der Datenbasis Purchasing-Example.csv), dann wird der Radar-Chart aus Abbildung 4.9 ausgegeben. Hier wird dargestellt, an welche Mitarbeiter Maris Freeman ihre Arbeit übergibt. Es ist auch zu erkennen, dass sie einen Teil der Fälle selbst beendet.

Falls als Resultat zwei verschiedene Datensätze verglichen werden sollen, bietet sich der Bar-Chart an.

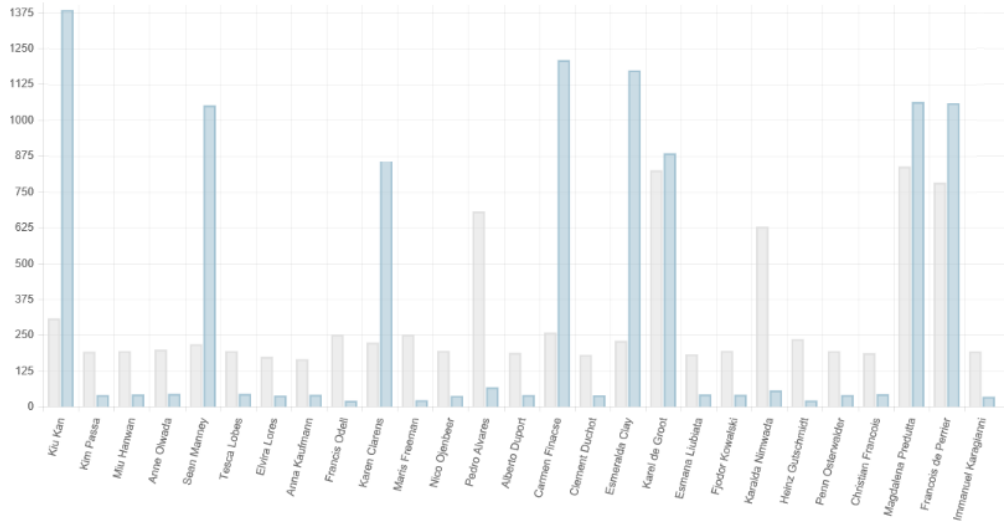

Abbildung 4.10: Beispiel für einen Bar Chart in ProDoop

Der graue Balken in Abbildung 4.10 zeigt, wie viele Aktionen ein Mitarbeiter insgesamt durchgeführt hat (Datenbasis: PurchasingExample.csv). Der blaue Balken steht für die gesamte Zeit in Minuten, die ein Mitarbeiter dafür benötigt hat (Nettoarbeitszeit). In der Grafik ist unter anderem zu erkennen, dass *Karel de Groot*, *Magdalena Predutta* und *Francois de Perrier* die meisten Fälle bearbeitet haben.

Um die Ergebnisse von Process Mining Abfragen darzustellen, wird eine Javascript Bibliothek benötigt, die Graphen darstellen kann. Cytoscape.js ist eine solche Bibliothek, die im Folgenden eingeführt wird.

Cytoscape.js *Cytoscape.js* ist eine Open-Source Graph-Bibliothek, die an der Universität von Toronto entwickelt wurde [36]. Cytoscape.js kann Graphen mit gerichteten und ungerichteten Kanten verschiedener Größe darstellen und ist somit in der Lage, Prozessmodelle und Social Networks zu verarbeiten und darzustellen. Um einen Graph ausgeben zu können, benötigt Cytoscape.js die Angabe von *Knoten* und *Kanten*. Knoten können als Kreise dargestellt werden. Besonders hilfreich ist die Möglichkeit, die Knotengröße und die Kantendicke individuell zu konfigurieren. Dies nutzt ProDoop beispielsweise, um oft frequentierte Knoten größer anzuzeigen. Außerdem wird die Liniendicke in Abhängigkeit von der Häufigkeit gesetzt, mit der die Kante genutzt wird.

Cytoscape.js bietet eine Layoutvorlage an, die einen gerichteten Graphen mit einem Startknoten als Wurzelelement erstellt. Dabei werden die Positionen der restlichen Knoten selbstständig berechnet (automatisches Layouting). Cytoscape.js bietet außerdem einige Interaktionsmöglichkeiten für Benutzer. Diese können beispielsweise einen Graphen innerhalb der Darstellungsfläche eines Browsers verschieben, sowie einzelne Knoten verschieben und in den Graph hinein- und herauszoomen. Dies ist vor allem bei Graphen mit vielen Elementen nützlich, um einzelne Abhängigkeiten zu erkennen.

Abbildung 4.11 zeigt das Ergebnis des Heuristic Miners (Datenbasis: PurchasingExample.csv) bei einem Threshold von 207 und 0.74. Der Threshold kann von Benutzern manuell gesetzt werden. Die Knoten neben dem Startknoten sind zwar Teil des Prozesses, sie besitzen aber keine Kanten zum restlichen Graphen, die den Threshold übertreffen würden.

Einen Graphen gleicher Bauart benötigt man für die soziale Netzwerkanalyse. Abbildung 4.12 zeigt, wie das Ergebnis dafür von Cytoscape.js dargestellt wird. Als Threshold wurde 0.1 gewählt (Datenbasis: PurchasingExample.csv). Da der Graph ziemlich viele Elemente enthält, empfiehlt es sich, mit dem Threshold zu variieren. Die Zoomfunktion und das Verschieben von Knoten kann ebenfalls die Übersicht erhöhen.

In diesem Kapitel wurde mit ProDoop eine WebApplikation vorgestellt, die Process Mining Algorithmen mit Hilfe des Mapreduce Programmiermodells implementiert und verschiedene grafische Darstellungsformen der Ergebnisse bietet. ProDoop kann mit wenig Aufwand verändert oder erweitert werden. Denkbar sind dabei die Implementierung weiterer Mining Algorithmen, die mit Pig eingebettet werden können. Möglich ist auch die Anbindung an ein größeres Hadoop Cluster, um große Datenmengen zu verarbeiten. Denkbar wäre zudem, die WebApplikation über das Internet bereitzustellen, damit könnte ProDoop von Kunden weltweit verwendet werden.

Die Wahl von MapReduce als Programmiermodell und für Apache Hadoop als Framework wurde unter Anderem mit der guten Effizienz bei großen Datenmengen begründet. Um dies nachvollziehen zu können, soll die Leistungsfähigkeit von ProDoop im folgenden Kapitel evaluiert werden.

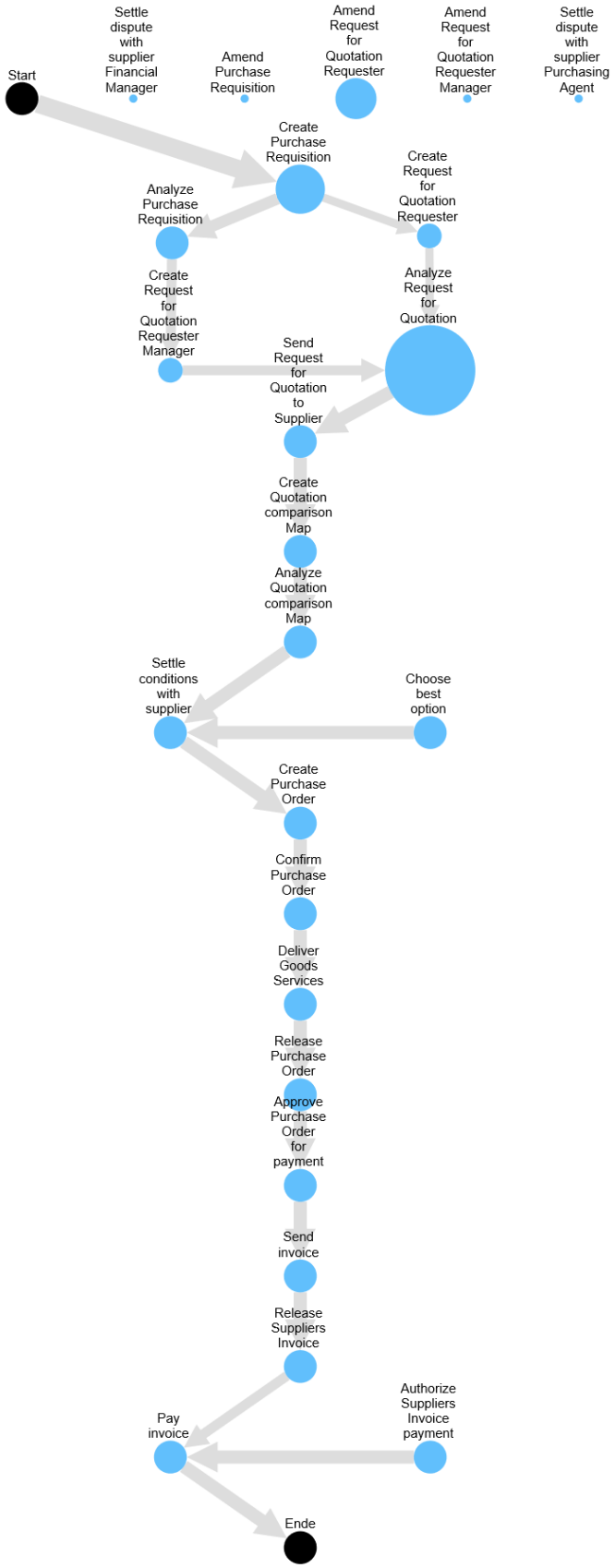

Abbildung 4.11: Das Ergebnis des Heuristic Miners mit Cytoscape.js

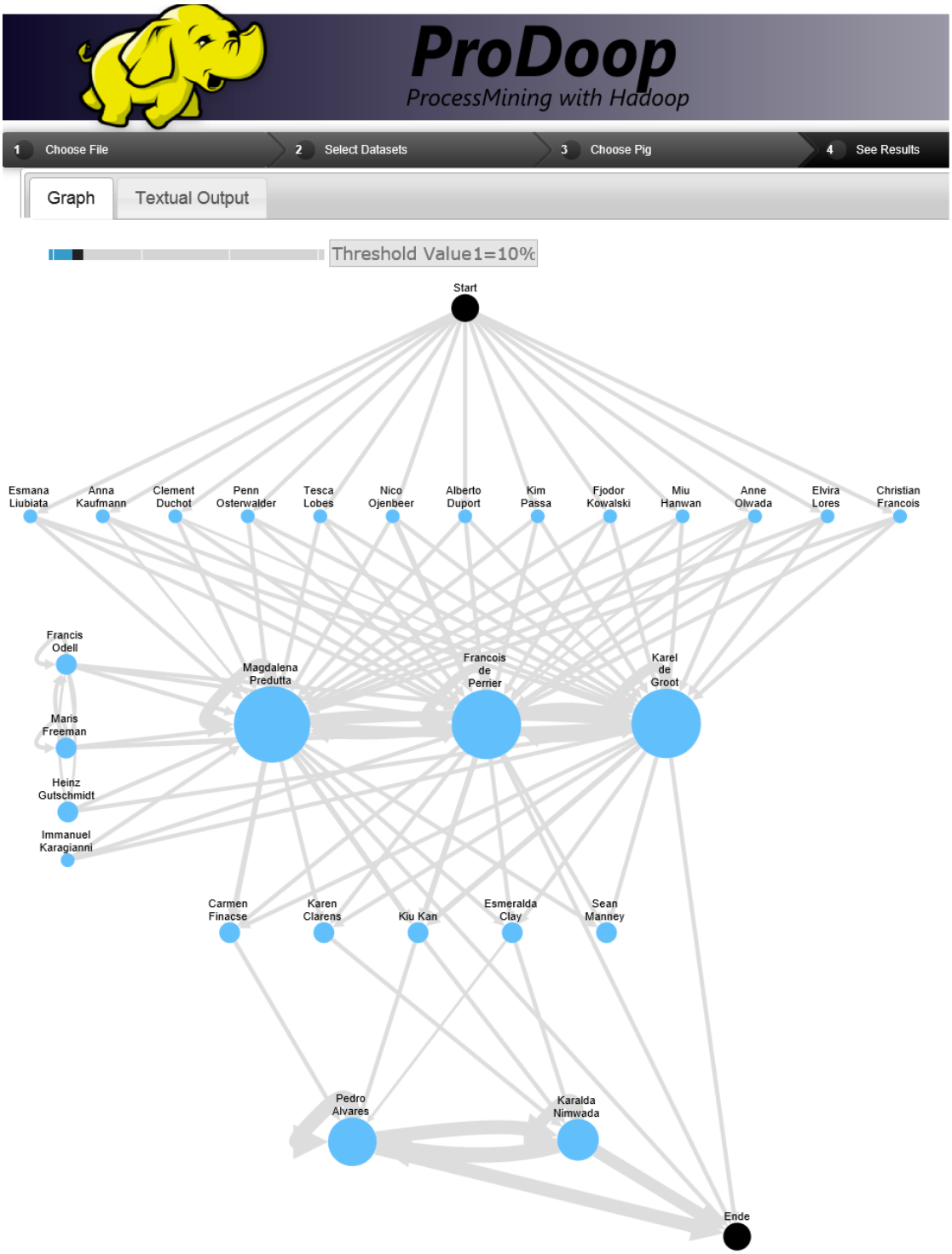

Abbildung 4.12: Das Ergebnis der Social Network Analysis mit Cytoscape.js

5

Evaluierung

In Kapitel 4 wurde die ProDoop WebApplikation vorgestellt, mit deren Hilfe Process Mining auf Basis von MapReduce durchgeführt werden kann. Die Wahl von MapReduce als Framework für die Datenanalyse wurde mit dessen guter Effizienz begründet. Diese wurde von anderen Quellen ermittelt. In diesem Kapitel soll in einem Experiment festgestellt werden, wie leistungsfähig MapReduce bei der Umsetzung eines Process Mining Algorithmus ist.

5.1 Fragestellung

Ziel des Experiments ist es, die Effizienz von MapReduce zu ermitteln. Dafür werden die Ausführungszeiten eines Process Mining Algorithmus auf Basis von MapReduce

gemessen. Dabei soll die Größe des MapReduce Clusters variiert werden und die Größe des untersuchten Datensatzes unterschiedlich gewählt werden.

Als Framework für MapReduce wird Apache Hadoop verwendet. Als Process Mining Algorithmus dient der Heuristic Miner, der in Kapitel 2.2.1 vorgestellt wurde. Dieser wird mit Pig, wie in Kapitel 3.4 vorgestellt, implementiert.

5.2 Definition der Metriken

Im Experiment sollen die Laufzeiten ermittelt werden, die verschiedene Hadoop Cluster für eine Pig Abfrage von unterschiedlichen Datenmengen benötigen. Am Ende jeder Pig Abfrage gibt Pig eine Datei mit statistischen Daten aus. Darin wird auch die Startzeit und Endzeit einer Pig Abfrage angegeben. Die Differenz beider Zeitpunkte wird als Laufzeit angegeben. Der Startzeitpunkt gibt den Zeitpunkt an, an dem Pig den Job übermittelt, nicht den Zeitpunkt, an dem der erste Map Job startet [28]. Falls das Hadoop Cluster stark ausgelastet ist, können beide Zeitpunkte stark voneinander abweichen. Der Endzeitpunkt ist der Zeitpunkt, an dem die Ausführung des Pig Jobs endet. Die Laufzeit wird in Millisekunden ausgegeben. Für die Auswertung wird die gemessene Zeit auf volle Sekunden aufgerundet, um eine bessere Übersichtlichkeit der Ergebnisse zu erhalten.

5.3 Experimentaufbau

Für das Experiment wird ein Hadoop Cluster mit einer Größe von zwei bis 11 Knoten verwendet. Da keine ausreichenden Hardwarekapazitäten zur Verfügung stehen, werden Cloud Computing Ressourcen gemietet. Cloud Computing bedeutet, dass ein Anbieter einen Teil seiner IT-Infrastruktur (Rechenkapazität, Festplattenplatz, Netzwerkkapazität und Arbeitsspeicher) einem Benutzer nach Bedarf über einen Netzwerkzugang zur Verfügung stellt [37]. Dies kommt dem Experiment entgegen, da die Hardwarekapazität variiert werden kann (Clustergröße). Es gibt einige Anbieter von Cloud Computing Services, für das Experiment wird *Amazon Web Services (AWS)* verwendet. *Amazon*

Elastic MapReduce (EMR) nennt sich der Service von AWS, der einen verwalteten Hadoop Cluster auf virtuellen AWS-Servern erzeugt und verwaltet. Die virtuellen Server nennen sich *Amazon Elastic Compute Cloud (EC2)-Instanzen* [38]. EC2 bietet anpassbare Rechenkapazität in der Cloud. Wenn ein Hadoop Cluster in EMR gestartet wird, kann die Anzahl der gewünschten EC2 Server Instanzen konfiguriert werden.

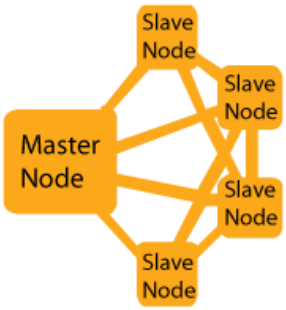

Abbildung 5.1: Aufbau eines Hadoop Clusters in EMR [39]

In EMR ist Apache Hadoop so konfiguriert, dass stets eine MasterNode und beliebig viele SlaveNodes zur Verfügung stehen. Die MasterNode enthält den Resource Manager und die Namenode. Sie enthält keine DataNode und ist damit nur zur Verwaltung nötig. Bei einem Defekt kann die MasterNode nicht wiederhergestellt werden. Laufende Jobs gehen damit in EMR verloren. Die SlaveNodes enthalten je eine DataNode und einen NodeManager (siehe Kapitel 3).

Bei Erstellung eines Hadoop Clusters mit EMR kann festlegt werden, welche Rechenkapazität die einzelnen EC2 Server Instanzen haben sollen. Für das Experiment wird die Variante *m3.xlarge* verwendet. Diese bietet ausbalancierte Datenverarbeitungs-, Arbeitsspeicher- und Netzwerkressourcen. *m3.xlarge* bietet folgende Eigenschaften [39]:

- Prozessoren: Intel Xeon E5-2670 v2
- Virtuelle CPUs: 4
- Arbeitsspeicher: 15 GB
- SSD-Speicher: 2 x 40 GB

Zur Durchführung des Tests wird ein weiterer AWS Service, A*mazon Simple Storage Service (S3)*, verwendet. Dabei handelt es sich um einen beständigen und skalierbaren Objektspeicher [39].

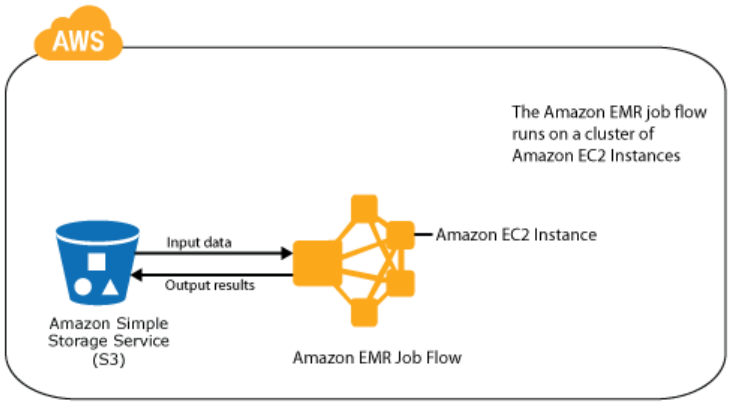

Abbildung 5.2: Zusammenspiel der verschiedenen AWS Services in EMR (angelehnt an [39])

Wie Abbildung 5.2 zeigt, werden die Ausgangsdaten in S3 gespeichert. EMR startet mehrere EC2 Instances und nutzt diese als Hadoop Cluster. Das Cluster lädt die Ausgangsdaten vom S3 herunter, führt die entsprechenden Analysen durch und speichert die Ergebnisdaten wieder in S3.

Zur Anwendung des Heuristic Miners in Amazon EMR wird das Pig-Skript, das in Kapitel 3.4 vorgestellt wurde, benötigt. Dieses wird in S3 hochgeladen. Als Nächstes werden Logdaten in verschiedener Größe benötigt. Es sollen Daten in einer Größe von 100KB, 1MB, 10MB, 100MB, 1GB und 10 GB verwendet werden. Dafür stehen keine real erstellten Logdaten in dieser Größenordnung zur Verfügung. Als Ausgangsdaten waren lediglich Logdaten von 1MB vorhanden. Das heißt, die anderen Datengrößen wurden manuell erstellt. Für die Datei mit 100KB wurde einfach der vordere Teil der Ausgangsdatenmenge abgeschnitten. Für die größeren Dateien wurde ein Programm erstellt, das die Ausgangsdaten mehrfach aneinander hängt. Dabei werden aber die InstanzIDs in jedem Durchlauf geändert. Außerdem werden zufällig Zeilen gelöscht und andere Zeilen mit Zufallsdaten eingefügt. Damit sollen die erzeugten Daten so nah wie möglich an der Realität liegen. In einem Megabyte Logdaten sind ca. 9100 Zeilen

gespeichert. Dies entspricht ca. 600 Fällen. Ein Auszug aus den verwendeten Daten sieht folgendermaßen aus:

```
1 1417,2011/07/23 12:53:00.000,2011/07/23 13:31:00.000,
  Create Purchase Requisition,Christian Francois,Requester
2 1417,2011/07/23 17:51:00.000,2011/07/23 17:59:00.000,
  Create Request for Quotation Requester,Immanuel Karagianni, Requester
3 1417,2011/08/02 07:02:00.000,2011/08/02 07:24:00.000,
  Analyze Request for Quotation,Karel de Groot,Purchasing Agent
4 1417,2011/08/02 08:17:00.000,2011/08/02 08:27:00.000,
  Amend Request for Quotation Requester,Anna Kaufmann,
  Requester Manager
5 1417,2011/08/08 04:03:00.000,2011/08/08 04:23:00.000,
  Analyze Request for Quotation,Magdalena Predutta, Purchasing Agent
6 159,2011/01/21 15:02:00.000,2011/01/21 15:42:00.000,
  Create Purchase Requisition,Elvira Lores,Requester
7 159,2011/01/22 02:58:00.000,2011/01/22 03:05:00.000,
  Analyze Purchase Requisition,Francis Odell,Requester Manager
8 159,2011/01/22 03:11:00.000,2011/01/22 03:14:00.000,
  Create Request for Quotation Requester Manager,Maris Freeman,
  Requester Manager
9 159,2011/01/23 03:45:00.000,2011/01/23 03:59:00.000,
  Analyze Request for Quotation,Francois de Perrier,Purchasing Agent
```

Listing 5.1: Auszug aus der im Experiment verwendeten .csv Datei

Die einzelnen Felder sind mit Kommas getrennt. Das erste Feld identifiziert die jeweilige Instanz. Alle Zeilen mit der gleichen Nummer gehören zur gleichen Instanz. Das zweite Feld steht für das Startdatum, das dritte für das Enddatum. Anschließend folgt ein Feld, das einen Bezeichner einer Aktivität kennzeichnet. Im fünften Feld steht die Ressource, die die Operation ausführt. Das letzte Feld bezeichnet die Rolle, die die Ressource ausfüllt.

Alle erstellten .csv Dateien werden in S3 hochgeladen, damit sie nachher im Hadoop Cluster verwendet werden können.

5.4 Experimentdurchführung

Für die Versuchsdurchführung werden im AWS Webclient nacheinander verschiedene EMR Cluster gestartet. Die Anzahl der ClientNodes werden variiert (1,2,4,6 und 8 CoreNodes). Wenn ein Cluster erstellt ist, ist es möglich, Pig Skripte darauf ausführen zu

lassen. In das Pig Skript, das den Heuristic Miner abbildet, werden weitere Parameter gesetzt. Diese sind Platzhalter für den Inputdatensatz und für die Outputs. Es wird nacheinander das gleiche Pig Skript mehrmals ausgeführt, hierbei wird der Inputdatensatz variiert, sodass jeder Datensatz von 100KB bis 10GB Größe einmalig als Datenbasis verwendet wird. Die Pig Skripte werden hierbei nacheinander ausgeführt. Die komplette Rechenkapazität steht also jeweils immer einem Pig Skript zur Verfügung. Ansonsten würde dies das Ergebnis verfälschen.

Als Resultat jedes Experiments ist interessant, welche Zeit jedes Cluster für eine Pig Abfrage bei verschiedenen Ausgangsdatenmengen benötigt hat. Diese kann aus den Logdaten ausgelesen werden, die Hadoop für jede beendete Pig Abfrage erstellt.

5.5 Datenanalyse des Experiments

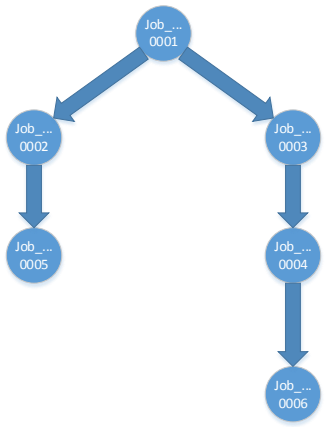

Abbildung 5.3: Aufteilung des Pig Skripts in Jobs

Nachdem alle Ergebnisse des Experiments ermittelt wurden, kann im Folgenden ein Diagramm betrachtet werden, das die Zeiten in Abhängigkeit von Clustergröße und Datengröße darstellt (siehe Abbildung 5.4).

Die vertikale Achse steht für die Laufzeit und die horizontale Achse zeigt die Clustergröße. Bei der Clustergröße ist allerdings nur die Anzahl von ClientNodes angegeben. Die Masternode wurde außen vor gelassen, da sie nur zu Verwaltungszwecken da ist. Zum

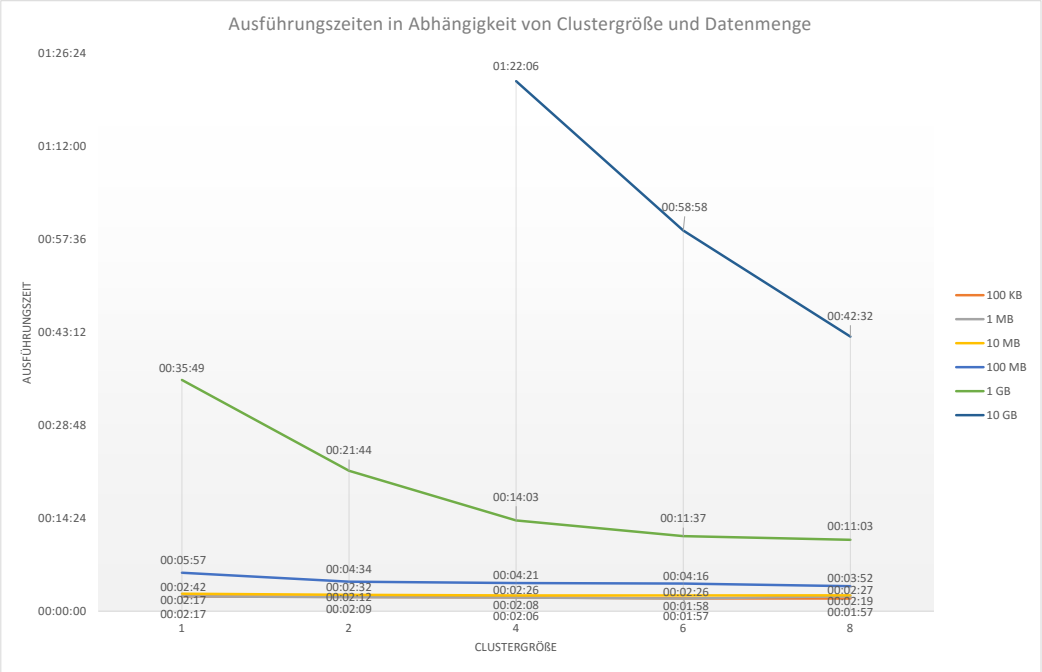

Abbildung 5.4: Ergebnisse des Experiments

Verständnis der Ergebnisse muss beachtet werden, dass Hadoop die Ausführung von Pig-Skripten in mehrere Jobs unterteilt (siehe Abbildung 3.7). Dies gilt auch für das hier verwendete Pig-Skript, das den Heuristic Miner abbildet. Dieser wird in 6 Jobs unterteilt. Die einzelnen Jobs werden hierbei nach Möglichkeit parallel ausgeführt.

Die Aufteilung des Pig Skripts in Jobs und deren Ablauf ist in Abbildung 5.3 dargestellt. Hier ist zu erkennen, dass zu Beginn `job0001` ausgeführt wird. Danach starten parallel die beiden Jobs `job0002` und `job0003`. Diese benötigen Daten, die von `job0001` bereitgestellt werden. Die Jobs werden dann, wie in Abbildung 5.3 zu sehen ist, in zwei parallelen Strängen abgearbeitet. Jeder Job muss in einen oder mehrere Mapper und Reducer aufgeteilt werden (siehe Kapitel 2.3). Für jeden belegten HDFS Block, der eine Größe von 128MB hat, wird ein Mapper benötigt [14]. Die Anzahl der Reducer richtet sich ebenfalls nach der Datenmenge. Jedem Reducer werden maximal 1 GB Datenvolumen zugewiesen. Das heißt, dass bei jedem vollen GB ein neuer Reducer

zugefügt wird. Insgesamt gilt also: je größer die Datenmenge, desto mehr Mapper und Reducer werden benötigt. Dies muss für die Bewertung der Ergebnisse des Experiments berücksichtigt werden.

Zunächst erkennt man in Abbildung 5.4, dass die Bearbeitungszeit nie deutlich unter zwei Minuten sinkt. Auch wenn die Dateigröße noch so klein gewählt wird, bleibt eine untere Grenze von etwa 120 Sekunden. Ein Grund hierfür ist, dass Hadoop einen ziemlich hohen Verwaltungsaufwand hat. Die Zeit, die Hadoop benötigt, um Mapper und Reducer zu starten und Ergebnisse zusammenzufügen, ist bei kleinen Datenmengen im Vergleich zur tatsächlichen Rechenzeit relativ hoch. Bei großen Datenmengen fällt dies nicht so sehr ins Gewicht. Dies zeigt, dass Hadoop für kleine Datenmengen weniger geeignet ist.

Abgesehen von ganz kleinen Datenmengen bis 10 MB ist zu beobachten, dass bei steigender Clustergröße die Bearbeitungszeit abnimmt. Dies ist nachvollziehbar, da jeder Job in mehrere Mapper und Reducer aufgeteilt wird. Wenn mehr Nodes zur Verfügung stehen, teilt Hadoop jedem Node Mapper und Reducer zu, die abzuarbeiten sind, d.h., bei steigender Clustergröße das Maß an Parallelität zunimmt. Bei kleinen Datenmengen tritt dieser Effekt weniger stark ein, da hier die meisten Jobs nur aus einem Mapper und einem Reducer bestehen.

Zu erkennen ist außerdem, dass die Steigerung der Clustergröße unterschiedliche Auswirkungen auf die Bearbeitungszeiten hat. Je größer die Datenmenge, desto besser ist der Effekt der Skalierung.

Die Geschwindigkeit der Berechnung des Clusters kann als $1/t$ bestimmt werden, wobei t die Bearbeitungszeit des Clusters ist. Diese Geschwindigkeit kann abhängig von der Clustergröße dargestellt werden, wie in Abbildung 6.2 zu sehen ist. Als Vergleichswerte zeigt die gelbe Linie den Fall eines linearen Anstiegs bei steigender Clustergröße. Dieser kann als Idealfall angesehen werden. Das Diagramm zeigt, dass der Geschwindigkeitsanstieg im Experiment bei einer Datenmenge von 10 GB fast linear ist (blaue Linie). Das heißt, dass bei einer Verdoppelung der Clustergröße nur etwa die halbe Berechnungszeit benötigt wird.

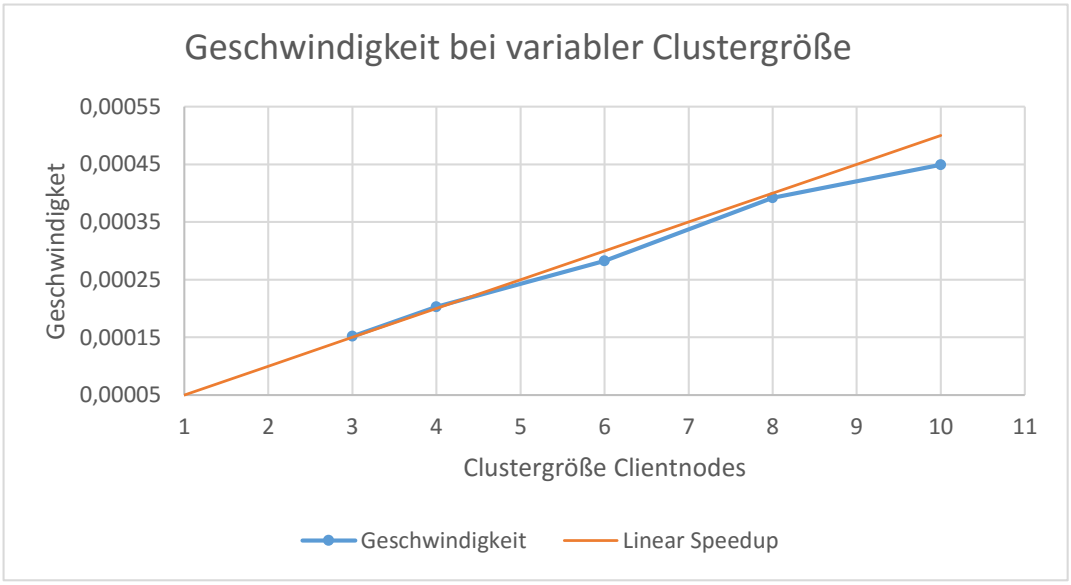

Abbildung 5.5: Geschwindigkeit der Berechnungen in Abhängigkeit der Clustergröße

5.6 Bewertung des Experiments

Das Experiment hat gezeigt, dass MapReduce einen gewissen Verwaltungsaufwand benötigt. Die gemessene Bearbeitungszeit kleiner Datensätze zeigt, dass MapReduce hier Laufzeitnachteile mit sich bringt. Eine klassische Berechnung ohne MapReduce ist hier zu bevorzugen. Bei großen Datenmengen fällt die Bearbeitungszeit für Verwaltungsaufgaben weniger ins Gewicht. Es konnte sogar gezeigt werden, dass bei einer Datenmenge von 10GB schon eine nahezu lineare Skalierbarkeit von MapReduce erreicht wird. Falls der Anwender die Rechenzeit verkürzen möchte, so genügt eine Erhöhung der Clustergröße. Besonders interessant ist der beobachtete Effekt bei 10GB Dateigröße aus Abbildung 6.2, bei der eine Verdoppelung der Clustergröße nahezu zu einer Halbierung der Rechenzeit geführt hat. Bei Nutzung der Amazon AWS stehen den doppelten Kosten für das Cluster die Hälfte der Gebühren für die Zeit gegenüber. Die Kosten bleiben also etwa gleich, was aus wirtschaftlicher Sicht zu begrüßen ist.

6

Diskussion

In Kapitel 5 wurde ein Experiment vorgestellt, das interessante Ergebnisse geliefert hat. Dabei muss allerdings berücksichtigt werden, dass keine realen Ausgangsdaten verwendet werden konnten. Die reale Datenbasis betrug lediglich 1MB. Diese wurde dann auf eine Größe von bis zu 10GB aufgebläht, dabei aber per Zufallsmodus modifiziert. Der ermittelte Effekt hat dennoch Aussagekraft. Es ist zu erwarten, dass der Effekt bei realen Daten ähnlich ist, die absolut gemessenen Berechnungszeiten können sich allerdings ändern. Die ermittelte Skalierbarkeit von MapReduce konnte auch von anderen Quellen, wie etwa in Abbildung 6.1, gezeigt werden [18, 40]. Dies ist ein Anzeichen dafür, dass MapReduce für BigData Anwendungen gut geeignet ist.

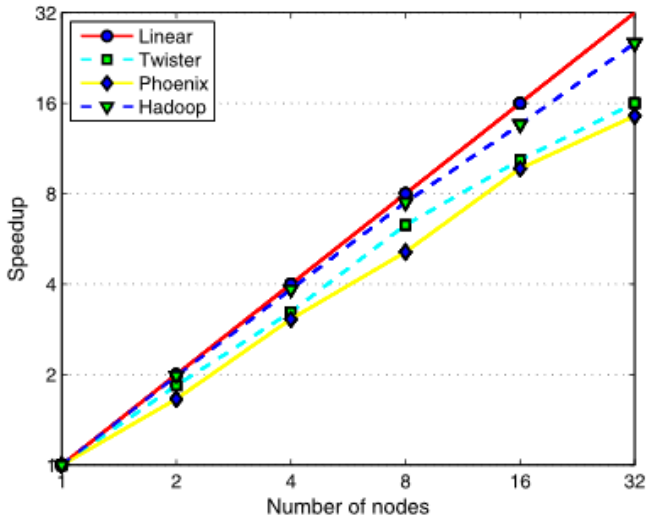

Abbildung 6.1: Geschwindigkeit von verschiedenen MapReduce Implementierungen bei steigender Clustergröße [18]

6.1 Jobtuning

Wenn man die Ergebnisse des Experiments betrachtet, stellt sich die Frage, ob eine Vergrößerung des Clusters die einzige Möglichkeit ist, um die Bearbeitungszeit zu verkürzen. Immerhin bietet Hadoop viele Möglichkeiten das Cluster und die Ausführung von Jobs zu spezifizieren. Zur weiteren Optimierung bieten sich folgende Maßnahmen an [14]:

- *Pig-Skript optimieren* Es gibt unter anderem folgende Möglichkeiten, wie Pig-Skripte verbessert werden können [27]:
  - STORE anstelle von DUMP verwenden. Bei Verwendung von DUMP müssen unter Umständen mehr Jobs durchgeführt werden, was wiederum die Ausführung verlangsamt.
  - FILTER-Anweisungen so früh und so oft wie möglich verwenden. Damit wird die Anzahl der Records verringert, die bearbeitet werden müssen.

  - Nur solche Fields speichern, die zur weiteren Berechnung benötigt werden (z.B.: *E = foreach D generate $0, $1;*). Damit kann die Datenmenge verringert werden.

- *Anzahl Reducer optimieren* In den Hadoop Grundeinstellungen wird pro GB Datengröße ein Reducer verwendet. Eine Veränderung der Anzahl verwendeter Reducer kann hier sinnvoll sein. Ein höherer Wert kann die Parallelität erhöhen, zu viele Reducer führen jedoch zu erhöhtem Netzwerkverkehr, der die Shuffle Phase ausbremsen kann [41]. Als Idealfall beträgt die Dauer eines Reducers mehrere Minuten. Dies hängt aber auch davon ab, wie CPU-intensiv die Jobs sind.
  Im Experiment aus Kapitel 5 wurden einige Reducephasen beobachtet, die viel Zeit benötigt haben (siehe Tabelle A.3). Zu Testzwecken wurde die Anzahl Reducer bei 1GB Datenmenge und 8 Knoten verändert. Die Ergebnisse sind in Abbildung 6.1 zu sehen.

- *Anzahl Mapper optimieren* Die Anzahl der verwendeten Mapper richtet sich standardmäßig nach der Anzahl der verwendeten HDFS Blöcke. Sie kann durch die Blockgröße verändert werden, oder durch Angabe von *mapred.min.split.size* im Pig-Skript konfiguriert werden. Da das Setup eines Mappers eine gewisse Zeit benötigt, sollte die Anzahl Mapper so gewählt werden, dass jeder Mapper mindestens eine Minute läuft [41]. Im Experiment wurde mit unterschiedlicher Zahl Mapper getestet (siehe auch Abbildung 6.1).

- *Intermediate Results komprimieren* Die Outputs der Map-Phase werden übers Netzwerk zu den Reducern kopiert. Eine Komprimierung dieser Daten kann zu einer Verringerung des Netzwerkverkehrs führen. Diese Methode wurde bei einer Datenmenge von 1GB getestet. Es ergaben sich allerdings keine signifikanten Unterschiede in der Bearbeitungszeit. Vermutlich tritt der erwartete Effekt erst bei größeren Datenmengen auf.

In den Ergebnisdaten des Experiments aus Kapitel 5 ist für die Variante mit 1GB Datengröße und einem Cluster mit 9 Knoten Verbesserungspotential zu sehen (siehe Tabelle A.3). Die Ausführungszeiten der dritten und vierten Reducephase sind deutlich länger als die anderen Reducephasen. Daher wird in diesem Experiment bei den

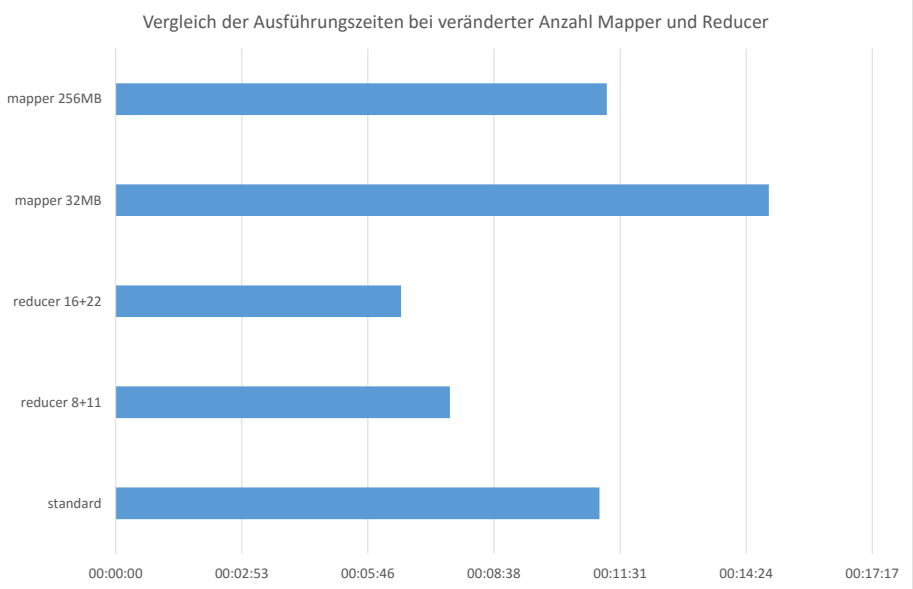

Abbildung 6.2: Dauer der Berechnungen bei Veränderung der Anzahl Mapper und Reducer

Varianten *reduce 8+11* und *reduce 16+22* mit einer Erhöhung der verwendeten Reducer getestet. In der dritten Reducephase werden 8 bzw. 16 Reducer verwendet und in der vierten Reducephase werden 11 bzw. 22 Reducer benutzt. Wie in den Ergebnisdaten in Tabelle A.31 und Tabelle A.30 sowie in Abbildung 6.1 zu sehen ist, hat die Erhöhung der Reducerzahl zu einer deutlichen Verringerung der Zeit geführt.

Außerdem wurde im Jobtuning Experiment mit einer Veränderung der Mapperzahl getestet. Im Standardfall beträgt die maximale Datenmenge, die pro Mapper verarbeitet wird 128MB. Diese wurde in der Variante *mapper 32MB* auf 32MB verringert und für *mapper 256MB* auf 256MB vergrößert. Wie in Abbildung 6.1 zu sehen ist, wirken sich diese Veränderungen negativ auf die erzielten Bearbeitungszeiten aus. Auch dieses Ergebnis ist nachvollziehbar, da die Bearbeitungszeiten der Mapper im Standardfall höchstens 62 Sekunden betragen (siehe Tabelle A.3).

Betrachtet man die Reducer, so bietet sich ein anderes Bild. Die Logdaten des Standardfalls zeigen durchschnittliche Bearbeitungszeiten von 348 und 95 Sekunden für

die 3. bzw. die 4. Reducephase. Das Experiment hat gezeigt, dass eine Erhöhung der Parallelität bei den betroffenen Reducern eine effizientere Ausführung ermöglicht.

Die vorgestellten Möglichkeiten zum Tuning von Jobs sollten in der Praxis angewendet werden, da sie zu einer Verringerung der Ausführungszeiten und somit zu einer Kostensenkung führen können. Dabei sollten insbesondere die Ergebnisdaten von abgeschlossenen Abfragen untersucht werden.

6.2 Vergleich zu Datenbank Management Systemen (DBMS)

In Kapitel 5 wurde aufgezeigt, dass MapReduce eine gute Effizienz erzielen kann. Trotzdem muss berücksichtigt werden, dass MapReduce in vielen Fällen nicht die schnellsten Ausführungszeiten erreicht. In einem Experiment konnten parallele DBMS bei Datenanalysen mit bis zu 100 Knoten deutliche Leistungsvorteile gegenüber MapReduce verzeichnen [40]. Dabei wurden aber lediglich einzelne SQL Statements umgesetzt und getestet. Bei einem Test mit komplexeren Abfragen können sich die Ergebnisse anders darstellen [42]. Gegenüber DBMS hat MapReduce außerdem einen eindeutigen Kostenvorteil, denn es stellt keine hohen Anforderungen an die Hardware, die einzelnen Nodes können heterogen sein. Die Installation eines Hadoop Clusters ist zudem unkompliziert [42]. Die Kostenvorteile verstärken sich, je größer die zu untersuchende Datenmenge ist.

In den letzten Jahren wurden einige Datenbanksysteme für Echtzeitanalyse (auch In-Memory Datenbanken genannt) entwickelt, wie z.B. *Apache Spark*, *Apache Tez* oder *SAP HANA* [20, 43, 44]. Das Konzept besteht hierbei, möglichst viele Daten im Hauptspeicher zu halten und möglichst wenige Lese-/Schreiboperationen auf klassischen HDDs durchzuführen, da diese nur einen unzureichenden Datendurchsatz bieten. Damit können die In-Memory Datenbanken schneller arbeiten als MapReduce. Wegen der hohen Anforderungen an den Hauptspeicher sind sie jedoch nicht für so große Datenmengen geeignet wie MapReduce. Deswegen sind die beiden Systeme weniger als Konkurrenz zu betrachten, sondern eher als Ergänzung. So nutzt beispielsweise

SAP HANA eine Schnittstelle zu Hadoop um dort mit MapReduce große Datenmengen vorzusortieren. Damit können sie anschließend in HANA analysiert werden.

Apache Tez ist ein Framework, das innerhalb von Apache Hadoop auf YARN ausgeführt werden kann. Interessant an Tez ist, dass es Pig-Skripte interpretieren kann. Damit könnten Pig-Skripte zur Untersuchung von kleineren Datenmengen mit Tez angewandt werden und ab einer bestimmten Datengröße mit MapReduce umgesetzt werden.

7

Zusammenfassung

Die Erstellung von Prozessmodellen ist für die Verbesserung von Geschäftsprozessen in Unternehmen von Vorteil. Ziel dieser Arbeit war es, ein Framework auf der Basis von Process Mining und MapReduce zu entwickeln, mit dessen Hilfe verschiedene Prozessperspektiven, wie beispielsweise die Organisationsstruktur, aus vorhandenen Logdaten ermittelt werden können. Zu diesem Zweck wurde ein heuristischer Mining-Algorithmus auf Basis der Skriptsprache Apache Pig vorgestellt, der effizientes Process Mining auf einem MapReduce-basierten Cluster ermöglicht.

Mit der Vorstellung der prototypischen Implementierung *ProDoop* konnte in dieser Arbeit gezeigt werden, wie Process Mining auf Basis von MapReduce implementiert werden kann. ProDoop nutzt hierbei Schnittstellen eines Hadoop Clusters, um Daten hochzuladen und Abfragen darauf auszuführen. Nach Eingabe von Logdaten ist ProDoop in der Lage Prozessmodelle zu berechnen und im Browser grafisch ansprechend darzustellen.

Des Weiteren hat sich in einem Experiment gezeigt, dass Apache Hadoop ein nützliches MapReduce-basiertes Framework ist. Es kann Datenanalysen mittels einer verteilten Ausführung schnell und zuverlässig durchführen. Außerdem enthält es mit Apache Pig eine Skriptsprache, mit der auch komplexe Abfragen relativ einfach zu programmieren sind.

Besonders bei großen Datenmengen ist Hadoop wegen dessen guter Skalierbarkeit eine gute Wahl. Es erreicht schnelle Ausführungszeiten bei geringen Kosten. Wenn die in dieser Arbeit ermittelten Möglichkeiten zum Jobtuning angewandt werden, so lässt sich die Effizienz des vorgestellten heuristischen Miners weiter steigern.

Für die nächsten Jahre ist davon auszugehen, dass die Datenmenge, die in Unternehmen erzeugt und gespeichert wird, stark ansteigen wird. Für die Auswertung dieser Daten ist Hadoop ein nützliches Werkzeug. Daher wird Hadoop in Verbindung mit MapReduce in Zukunft von steigender Bedeutung sein.

Literaturverzeichnis

[1] Freund, J., Rücker, B.: Praxishandbuch BPMN 2.0. Carl Hanser Verlag GmbH & Company KG (2014)

[2] fluxicon: (www.fluxicon.com) besucht am 5.Oktober 2015.

[3] Van Der Aalst, W.M., Reijers, H.A., Song, M.: Discovering Social Networks from Event Logs. Computer Supported Cooperative Work (CSCW) **14** (2005) 549–593

[4] 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: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, ACM (2010) 975–986

[5] hadoop.apache.org: (http://hadoop.apache.org) besucht am 12.Oktober 2015.

[6] Weske, M.: Business Process Management: Concepts, Methods. Technology. Springer, Berlin (2007)

[7] Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer (2013)

[8] Weijters, A., van Der Aalst, W.M., De Medeiros, A.A.: (Process Mining with the Heuristics Miner-Algorithm)

[9] Van der Aalst, W.M., Song, M.: Mining Social Networks: Uncovering Interaction Patterns in Business Processes. In: Business Process Management. Springer (2004) 244–260

[10] De Weerdt, J., De Backer, M., Vanthienen, J., Baesens, B.: A Multi-Dimensional Quality Assessment of State-of-the-art Process Discovery Algorithms using Real-Life Event Logs. Information Systems **37** (2012) 654–676

[11] Van Der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer Science & Business Media (2011)

[12] Jiang, D., Ooi, B.C., Shi, L., Wu, S.: The Performance of MapReduce: An in-depth Study. Proceedings of the VLDB Endowment **3** (2010) 472–483

[13] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for Multi-Core and Multiprocessor Systems. In: High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on, IEEE (2007) 13–24

[14] White, T.: Hadoop: The Definitive Guide: [Storage and Analysis at Internet Scale]. O'Reilly, Beijing; Köln[u.a.] (2015)

[15] Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM **51** (2008) 107–113

[16] iterativemapreduce.org/: (http://www.iterativemapreduce.org/) besucht am 19.März 2016.

[17] discoproject.org: (http://discoproject.org) besucht am 19.März 2016.

[18] Zhang, J., Wong, J.S., Li, T., Pan, Y.: A Comparison of Parallel Large-Scale Knowledge Acquisition Using Rough Set Theory on Different MapReduce Runtime Systems. International Journal of Approximate Reasoning **55** (2014) 896–907

[19] Wadkar, S.: Pro Apache Hadoop. 2nd ed edn. APress, New York (c2014)

[20] tez.apache.org: (http://tez.apache.org) besucht am 12.Februar 2016.

[21] giraph.apache.org: (http://giraph.apache.org) besucht am 12.Februar 2016.

[22] www.admin magazine.com: (http://www.admin-magazine.com) besucht am 13.Oktober 2015.

[23] Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al.: Apache Hadoop YARN: Yet Another Resource Negotiator. In: Proceedings of the 4th annual Symposium on Cloud Computing, ACM (2013) 5

[24] Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop Distributed File System. In: Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, IEEE (2010) 1–10

[25] Hedlund, B.: (www.bradhedlund.com) besucht am 14.Oktober 2015.

[26] deRoos, D.: Hadoop for dummies. Online-ausg. edn. John Wiley Sons, Hoboken, NJ (2014)

[27] pig.apache.org: (http://pig.apache.org) besucht am 19.Oktober 2015.

[28] Gates, A.: Programming Pig. 1st ed edn. O'Reilly, Sebastopol, CA (2011)

[29] Olston, B. Reed, U.S.R.K., Tomkins, A.: A Not-so-Foreign Language for Data Processing. In: ACM SIGMOD 2008 International Conference on Management of Data, Vancouver, Canada. (2008)

[30] Hortonworks: (http://hortonworks.com) besucht am 28.Oktober 2015.

[31] Vukotic, A., Goodwill, J.: Apache Tomcat 7. APress, [S.l.] (c2011)

[32] Zambon, G.: Beginning JSP, JSF and Tomcat: Java Web Development. APress, [S.l.] (2012)

[33] tomcat.apache.org: (http://tomcat.apache.org) besucht am 22.März 2016.

[34] Friedl, J.: Mastering Regular Expressions. A nutshell handbook. O'Reilly Media, Incorporated (2006)

[35] chartjs.org: (http://www.chartjs.org) besucht am 10.November 2015.

[36] js.cytoscape.org: (http://js.cytoscape.org) besucht am 10.November 2015.

[37] Mell, P., Grance, T.: The NIST Definition of Cloud Computing. (2011)

[38] Deyhim, P.: Best Practices for Amazon EMR. Technical report, Amazon Web Services Inc (2013)

[39] Amazon Web Services: (http://docs.aws.amazon.com) besucht am 17.November 2015.

[40] 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: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, ACM (2009) 165–178

[41] www.wiki.apache.org: (http://www.wiki.apache.org/hadoop/poweredby) besucht am 10.Oktober 2015.

[42] McClean, A., Conceicao, R., O'Halloran, M.: A Comparison of MapReduce and Parallel Database Management Systems. In: The Eighth International Conference on Systems. (2013) 64–68

[43] spark.apache.org: (http://spark.apache.org) besucht am 25.Februar 2016.

[44] sap.com: (http://www.sap.com) besucht am 25.Februar 2016.

A

Anhang

```
1  import java.io.IOException;
2  import java.util.StringTokenizer;
3
4  import org.apache.hadoop.conf.Configuration;
5  import org.apache.hadoop.fs.Path;
6  import org.apache.hadoop.io.IntWritable;
7  import org.apache.hadoop.io.Text;
8  import org.apache.hadoop.mapreduce.Job;
9  import org.apache.hadoop.mapreduce.Mapper;
10 import org.apache.hadoop.mapreduce.Reducer;
11 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
12 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
13
14 public class WordCount {
15
16   public static class TokenizerMapper
17        extends Mapper<Object, Text, Text, IntWritable>{
18
19     private final static IntWritable one = new IntWritable(1);
20     private Text word = new Text();
21
```

```
22          // die Map-Funktion bekommt als Parameter
23          //einen Text als value uebergeben
24      public void map(Object key, Text value, Context context
25                      ) throws IOException, InterruptedException {
26        StringTokenizer itr = new StringTokenizer(value.toString());
27        // der Text wird aufgeteilt in einzelne Woerter
28        while (itr.hasMoreTokens()) {
29          word.set(itr.nextToken());
30          context.write(word, one);
31          //Ausgabe fuer jedes Wort: jeweiliges Wort als key
32          //und one bzw. 1 als value
33        }
34      }
35    }
36
37    public static class IntSumReducer
38         extends Reducer<Text,IntWritable,Text,IntWritable> {
39      private IntWritable result = new IntWritable();
40
41          //die Reduce-Funktion bekommt ein Wort als key und eine Liste von
42          //Zahlen (z.B.: 1,1,1) als values uebergeben
43      public void reduce(Text key, Iterable<IntWritable> values,
44                         Context context
45                         ) throws IOException, InterruptedException {
46        int sum = 0;
47        for (IntWritable val : values) {
48          sum += val.get();//die einzelnen Zahlen werden addiert
49        }
50        result.set(sum);
51        context.write(key, result);// Ausgabe: ein Wort als key und eine
52        //Zahl als value bzw. result
53      }
54    }
55
56    public static void main(String[] args) throws Exception {
57      Configuration conf = new Configuration();
58      Job job = Job.getInstance(conf, "word count");
59      job.setJarByClass(WordCount.class);
60      job.setMapperClass(TokenizerMapper.class);
61      job.setCombinerClass(IntSumReducer.class);
62      //hier wird ein Combiner verwendet
63      job.setReducerClass(IntSumReducer.class);
64      job.setOutputKeyClass(Text.class);
65      job.setOutputValueClass(IntWritable.class);
66      FileInputFormat.addInputPath(job, new Path(args[0]));
67      FileOutputFormat.setOutputPath(job, new Path(args[1]));
68      System.exit(job.waitForCompletion(true) ? 0 : 1);
69    }
70  }
```

Listing A.1: Umsetzung des Wordcount MapReduce Beispiels in Java

Tabelle A.1: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 10 GB und Clustergröße 1+10

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 8561_ 0037	159	11	57	13	40	86	78	82	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /290
job_1447 8561_ 0038	80	9	75	29	67	150	116	132	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 8561_ 0039	160	18	74	15	42	675	445	544	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 8561_ 0040	813	110	54	12	33	812	57	171	g,h,lim	GROUP_BY	
job_1447 8561_ 0041	4	1	51	36	43	19	19	19	C,D	GROUP_BY, COMBINER	s3:// output /291
job_1447 8561_ 0042	110	12	47	18	40	47	43	45	i,j	GROUP_BY, COMBINER	s3:// output/ 292

Pig script completed in 37 minutes, 6 seconds and 415 milliseconds (2226415 ms)

Tabelle A.2: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 10 GB und Clustergröße 1+8

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1442 4806_ 0025	159	11	54	12	38	121	82	110	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /149
job_1442 4806_ 0026	80	9	75	21	62	206	141	178	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1442 4806_ 0027	160	18	73	12	44	779	393	612	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1442 4806_ 0028	814	110	51	13	32	811	57	151	g,h,lim	GROUP_BY	
job_1442 4806_ 0029	4	1	42	33	37	20	20	20	C,D	GROUP_BY, COMBINER	s3:// output /150
job_1442 4806_ 0030	110	12	46	10	39	75	37	54	i,j	GROUP_BY, COMBINER	s3:// output/ 151

Pig script completed in 42 minutes, 32 seconds and 346 milliseconds (2552346 ms)

Tabelle A.3: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 1 GB und Clustergröße 1+8

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1440 0202_0001	16	2	17	14	16	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /82
job_1440 0202_0002	8	1	50	38	45	38	38	38	B, bWith-Epoch, jn,lim	GROUP_BY	
job_1440 0202_0003	16	2	62	24	47	350	347	348	bWith-Epoch, bWith-Epoch2, d,e,f	HASH_JOIN	
job_1440 0202_0004	80	11	54	17	44	107	82	95	g,h,lim	GROUP_BY	
job_1440 0202_0005	1	1	25	25	25	7	7	7	C,D	GROUP_BY, COMBINER	s3:// output /83
job_1440 0202_0006	11	2	32	11	24	20	20	20	i,j	GROUP_BY, COMBINER	s3:// output/ 84

Pig script completed in 11 minutes, 2 seconds and 994 milliseconds (662994 ms)

Tabelle A.4: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 100 MB und Clustergröße 1+8

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1442 4806_0013	2	1	12	10	11	6	6	6	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /143
job_1442 4806_0014	1	1	17	17	17	9	9	9	B, bWith-Epoch, jn,lim	GROUP_BY	
job_1442 4806_0015	2	1	20	19	20	68	68	68	bWith-Epoch, bWith-Epoch2, d,e,f	HASH_JOIN	
job_1442 4806_0016	8	2	21	15	18	27	27	27	g,h,lim	GROUP_BY	
job_1442 4806_0017	1	1	6	6	6	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /144
job_1442 4806_0018	1	1	11	11	11	6	6	6	i,j	GROUP_BY, COMBINER	s3:// output/ 145

Pig script completed in 3 minutes, 51 seconds and 910 milliseconds (231910 ms)

Tabelle A.5: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 10 MB und Clustergröße 1+8

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1442 4806_0007	1	1	8	8	8	6	6	6	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /140
job_1442 4806_0008	1	1	9	9	9	5	5	5	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1442 4806_0009	2	1	9	6	7	12	12	12	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1442 4806_0010	1	1	14	14	14	10	10	10	g,h,lim	GROUP_BY	
job_1442 4806_0011	1	1	6	6	6	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /141
job_1442 4806_0012	1	1	7	7	7	6	6	6	i,j	GROUP_BY, COMBINER	s3:// output/ 142

Pig script completed in 2 minutes, 26 seconds and 509 milliseconds (146509 ms)

Tabelle A.6: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 1 MB und Clustergröße 1+8

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1442 4806_0001	1	1	8	8	8	7	7	7	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /137
job_1442 4806_0002	1	1	6	6	6	4	4	4	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1442 4806_0003	2	1	8	8	8	7	7	7	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1442 4806_0004	1	1	6	6	6	5	5	5	g,h,lim	GROUP_BY	
job_1442 4806_0005	1	1	6	6	6	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /138
job_1442 4806_0006	1	1	6	6	6	6	6	6	i,j	GROUP_BY, COMBINER	s3:// output/ 139

Pig script completed in 2 minutes, 18 seconds and 764 milliseconds (138764 ms)

Tabelle A.7: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 100 KB und Clustergröße 1+8

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_ 1447 8561_ 0025	1	1	6	6	6	8	8	8	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /283
job_ 1447 8561_ 0026	1	1	4	4	4	4	4	4	B, bWith- Epoch, jn,lim	GROUP_BY	
job_ 1447 8561_ 0027	2	1	5	4	5	4	4	4	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_ 1447 8561_ 0028	1	1	5	5	5	4	4	4	g,h,lim	GROUP_BY	
job_ 1447 8561_ 0029	1	1	6	6	6	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /284
job_ 1447 8561_ 0030	1	1	6	6	6	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 139

Pig script completed in 1 minute, 56 seconds and 518 milliseconds (116518 ms)

Tabelle A.8: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 10 GB und Clustergröße 1+6

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_ 1442 4099_ 0049	159	11	55	12	35	179	22	135	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /128
job_ 1442 4099_ 0050	80	9	75	24	58	208	76	160	B, bWith- Epoch, jn,lim	GROUP_BY	
job_ 1442 4099_ 0051	160	18	73	13	43	795	439	632	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_ 1442 4099_ 0052	814	110	51	12	33	1110	56	156	g,h,lim	GROUP_BY	
job_ 1442 4099_ 0053	3	1	46	41	44	12	12	12	C,D	GROUP_BY, COMBINER	s3:// output /129
job_ 1442 4099_ 0054	110	12	47	11	34	103	12	75	i,j	GROUP_BY, COMBINER	s3:// output/ 130

Pig script completed in 58 minutes, 57 seconds and 763 milliseconds (3537763 ms)

Tabelle A.9: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 1 GB und Clustergröße 1+6

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1442 4099_0043	16	2	24	15	20	8	7	8	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /125
job_1442 4099_0044	8	1	70	51	58	52	52	52	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1442 4099_0045	16	2	75	51	65	337	332	334	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1442 4099_0046	80	11	54	14	39	135	71	110	g,h,lim	GROUP_BY	
job_1442 4099_0047	1	1	25	25	254	11	11	11	C,D	GROUP_BY, COMBINER	s3:// output /126
job_1442 4099_0048	11	2	28	19	25	9	8	8	i,j	GROUP_BY, COMBINER	s3:// output/ 127

Pig script completed in 11 minutes, 37 seconds and 309 milliseconds (697309 ms)

Tabelle A.10: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 100 MB und Clustergröße 1+6

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1442 4099_0025	2	1	13	10	12	6	6	6	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /116
job_1442 4099_0026	1	1	17	17	17	12	12	12	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1442 4099_0027	2	1	17	16	17	71	71	71	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1442 4099_0028	8	2	43	16	31	48	48	48	g,h,lim	GROUP_BY	
job_1442 4099_0029	1	1	6	6	6	6	6	6	C,D	GROUP_BY, COMBINER	s3:// output /117
job_1442 4099_0030	1	1	11	11	11	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 118

Pig script completed in 4 minutes, 16 seconds and 315 milliseconds (256315 ms)

Tabelle A.11: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 10 MB und Clustergröße 1+6

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_ 1442 4099_ 0019)	1	1	8	8	8	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /113
job_ 1442 4099_ 0020	1	1	7	7	7	5	5	5	B, bWith- Epoch, jn,lim	GROUP_BY	
job_ 1442 4099_ 0021	2	1	7	7	7	12	12	12	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_ 1442 4099_ 0022	1	1	15	15	15	10	10	10	g,h,lim	GROUP_BY	
job_ 1442 4099_ 0023	1	1	8	8	8	6	6	6	C,D	GROUP_BY, COMBINER	s3:// output /114
job_ 1442 4099_ 0024	1	1	7	7	7	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 115

Pig script completed in 2 minutes, 26 seconds and 269 milliseconds (146269 ms)

Tabelle A.12: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 1 MB und Clustergröße 1+6

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_ 1447 8561_ 0019)	1	1	7	7	7	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /280
job_ 1447 8561_ 0020	1	1	5	5	5	4	4	4	B, bWith- Epoch, jn,lim	GROUP_BY	
job_ 1447 8561_ 0021	2	1	5	5	5	6	6	6	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_ 1447 8561_ 0022	1	1	6	6	6	5	5	5	g,h,lim	GROUP_BY	
job_ 1447 8561_ 0023	1	1	6	6	6	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /281
job_ 1447 8561_ 0024	1	1	6	6	6	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 282

Pig script completed in 1 minute, 56 seconds and 606 milliseconds (116606 ms)

Tabelle A.13: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 100 KB und Clustergröße 1+6

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 8561_ 0013)	1	1	7	7	7	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /277
job_1447 8561_ 0014	1	1	6	6	6	5	5	5	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 8561_ 0015	2	1	6	5	6	6	6	6	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 8561_ 0016	1	1	5	5	5	4	4	4	g,h,lim	GROUP_BY	
job_1447 8561_ 0017	1	1	6	6	6	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /278
job_1447 8561_ 0018	1	1	6	6	6	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 279

Pig script completed in 1 minute, 57 seconds and 536 milliseconds (117536 ms)

Tabelle A.14: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 10 GB und Clustergröße 1+4

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 7696_ 0036	159	11	55	18	34	266	6	119	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /268
job_1447 7696_ 0037	80	9	79	21	50	328	72	156	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 7696_ 0038	160	18	76	13	46	898	423	592	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 7696_ 0039	813	110	53	13	29	1552	55	140	g,h,lim	GROUP_BY	
job_1447 7696_ 0040	3	1	33	29	31	6	6	6	C,D	GROUP_BY, COMBINER	s3:// output /269
job_1447 7696_ 0041	110	12	48	11	33	168	5	75	i,j	GROUP_BY, COMBINER	s3:// output/ 270

Pig script completed in 1 hour, 22 minutes, 5 seconds and 851 milliseconds (4925851 ms)

Tabelle A.15: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 1 GB und Clustergröße 1+4

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 7696_ 0025	16	2	29	26	27	7	6	6	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /262
job_1447 7696_ 0026	8	1	73	56	62	43	43	43	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 7696_ 0027	16	2	74	49	66	400	394	397	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 7696_ 0028	82	11	57	14	34	187	58	119	g,h,lim	GROUP_BY	
job_1447 7696_ 0029	1	1	18	18	18	6	6	6	C,D	GROUP_BY, COMBINER	s3:// output /263
job_1447 7696_ 0030	11	2	47	25	39	19	19	19	i,j	GROUP_BY, COMBINER	s3:// output/ 264

Pig script completed in 14 minutes, 2 seconds and 736 milliseconds (842736 ms)

Tabelle A.16: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 100 MB und Clustergröße 1+4

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 7696_ 0019	2	1	12	10	11	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /259
job_1447 7696_ 0020	1	1	17	17	17	8	8	8	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 7696_ 0021	2	1	21	20	21	68	68	68	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 7696_ 0022	8	2	50	23	39	48	48	48	g,h,lim	GROUP_BY	
job_1447 7696_ 0023	1	1	25	25	25	6	6	6	C,D	GROUP_BY, COMBINER	s3:// output /260
job_1447 7696_ 0024	1	1	11	11	11	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 261

Pig script completed in 4 minutes, 21 seconds and 8 milliseconds (261008 ms)

Tabelle A.17: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 10 MB und Clustergröße 1+4

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 7696_ 0013	1	1	8	8	8	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /256
job_1447 7696_ 0014	1	1	6	6	6	7	7	7	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 7696_ 0015	2	1	7	6	6	14	14	14	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 7696_ 0016	1	1	15	15	15	10	10	10	g,h,lim	GROUP_BY	
job_1447 7696_ 0017	1	1	6	6	6	6	6	6	C,D	GROUP_BY, COMBINER	s3:// output /257
job_1447 7696_ 0018	1	1	7	7	7	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 258

Pig script completed in 2 minutes, 26 seconds and 311 milliseconds (146311 ms)

Tabelle A.18: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 1 MB und Clustergröße 1+4

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 7696_ 0007	1	1	7	7	7	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /253
job_1447 7696_ 0008	1	1	5	5	5	5	5	5	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 7696_ 0009	2	1	7	7	7	6	6	6	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 7696_ 0010	1	1	7	7	7	5	5	5	g,h,lim	GROUP_BY	
job_1447 7696_ 0011	1	1	7	7	7	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /254
job_1447 7696_ 0012	1	1	6	6	6	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 255

Pig script completed in 2 minutes, 6 seconds and 137 milliseconds (126137 ms)

Tabelle A.19: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 100 KB und Clustergröße 1+4

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 7696_ 0001	1	1	8	8	8	7	7	7	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /250
job_1447 7696_ 0002	1	1	4	4	4	4	4	4	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 7696_ 0003	2	1	7	7	7	5	5	5	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 7696_ 0004	1	1	5	5	5	4	4	4	g,h,lim	GROUP_BY	
job_1447 7696_ 0005	1	1	5	5	5	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /251
job_1447 7696_ 0006	1	1	6	6	6	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 252

Pig script completed in 2 minutes, 7 seconds and 824 milliseconds (127824 ms)

Tabelle A.20: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 1 GB und Clustergröße 1+2

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 3371_ 0019	16	2	53	31	41	19	14	16	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /239
job_1447 3371_ 0020	8	1	55	49	53	67	67	67	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 3371_ 0021	16	2	55	28	45	422	419	420	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 3371_ 0022	82	11	55	14	35	377	59	138	g,h,lim	GROUP_BY	
job_1447 3371_ 0023	1	1	22	22	22	8	8	8	C,D	GROUP_BY, COMBINER	s3:// output /240
job_1447 3371_ 0024	11	2	44	33	39	10	9	9	i,j	GROUP_BY, COMBINER	s3:// output/ 241

Pig script completed in 21 minutes, 43 seconds and 649 milliseconds (1303649 ms)

Tabelle A.21: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 100 MB und Clustergröße 1+2

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 3371_ 0013	2	1	13	10	11	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /236
job_1447 3371_ 0014	1	1	26	26	26	10	10	10	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 3371_ 0015	2	1	28	25	26	71	71	71	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 3371_ 0016	8	2	42	30	36	37	36	36	g,h,lim	GROUP_BY	
job_1447 3371_ 0017	1	1	15	15	15	11	11	11	C,D	GROUP_BY, COMBINER	s3:// output /237
job_1447 3371_ 0018	1	1	11	11	11	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 238

Pig script completed in 4 minutes, 33 seconds and 758 milliseconds (273758 ms)

Tabelle A.22: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 10 MB und Clustergröße 1+2

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 3371_ 0007	1	1	8	8	8	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /233
job_1447 3371_ 0008	1	1	12	12	12	5	5	5	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 3371_ 0009	2	1	12	12	12	12	12	12	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 3371_ 0010	1	1	16	16	16	9	9	9	g,h,lim	GROUP_BY	
job_1447 3371_ 0011	1	1	6	6	6	6	6	6	C,D	GROUP_BY, COMBINER	s3:// output /234
job_1447 3371_ 0012	1	1	7	7	7	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 235

Pig script completed in 2 minutes, 31 seconds and 659 milliseconds (151659 ms)

Tabelle A.23: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 1 MB und Clustergröße 1+2

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 3371_ 0001	1	1	7	7	7	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /230
job_1447 3371_ 0002	1	1	8	8	8	4	4	4	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 3371_ 0003	2	1	9	9	9	6	6	6	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 3371_ 0004	1	1	6	6	6	5	5	5	g,h,lim	GROUP_BY	
job_1447 3371_ 0005	1	1	5	5	5	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /231
job_1447 3371_ 0006	1	1	7	7	7	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 232

Pig script completed in 2 minutes, 8 seconds and 879 milliseconds (128879 ms)

Tabelle A.24: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 100 KB und Clustergröße 1+2

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 8561_ 0007	1	1	6	6	6	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /274
job_1447 8561_ 0008	1	1	7	7	7	5	5	5	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 8561_ 0009	2	1	7	7	7	6	6	6	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 8561_ 0010	1	1	6	6	6	5	5	5	g,h,lim	GROUP_BY	
job_1447 8561_ 0011	1	1	6	6	6	6	6	6	C,D	GROUP_BY, COMBINER	s3:// output /275
job_1447 8561_ 0012	1	1	6	6	6	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 276

Pig script completed in 2 minutes, 12 seconds and 60 milliseconds (132060 ms)

Tabelle A.25: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 1 GB und Clustergröße 1+1

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 2528_0025	16	2	26	19	30	48	5	27	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /206
job_1447 2528_0026	8	1	33	22	30	40	40	40	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 2528_0027	16	2	51	21	38	422	369	396	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 2528_0028	82	11	28	15	22	51	47	48	g,h,lim	GROUP_BY	
job_1447 2528_0029	1	1	12	12	12	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /216
job_1447 2528_0030	11	2	34	14	31	17	14	16	i,j	GROUP_BY, COMBINER	s3:// output/ 217

Pig script completed in 35 minutes, 48 seconds and 781 milliseconds (2148781 ms)

Tabelle A.26: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 100 MB und Clustergröße 1+1

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 2528_0007	2	1	16	14	15	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /206
job_1447 2528_0008	1	1	27	27	27	8	8	8	B, bWith- Epoch, jn,lim	GROUP_BY	
job_2528_0009_0015	2	1	27	26	26	68	68	68	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 2528_0010	8	2	26	17	22	31	28	30	g,h,lim	GROUP_BY	
job_1447 2528_0011	1	1	9	9	9	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /207
job_1447 2528_0012	1	1	11	11	11	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 208

Pig script completed in 5 minutes, 56 seconds and 836 milliseconds (356836 ms)

Tabelle A.27: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 10 MB und Clustergröße 1+1

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 2528_ 0019	1	1	8	8	8	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /212
job_1447 2528_ 0020	1	1	11	11	11	5	5	5	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 2528_ 0021	2	1	11	11	11	11	11	11	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 2528_ 0022	1	1	16	16	16	9	9	9	g,h,lim	GROUP_BY	
job_1447 2528_ 0023	1	1	7	7	7	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /213
job_1447 2528_ 0024	1	1	7	7	7	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 214

Pig script completed in 2 minutes, 41 seconds and 742 milliseconds (161742 ms)

Tabelle A.28: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 1 MB und Clustergröße 1+1

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 2528_ 0013	1	1	7	7	7	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /209
job_1447 2528_ 0014	1	1	8	8	8	4	4	4	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 2528_ 0015	2	1	9	9	9	5	5	5	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 2528_ 0016	1	1	7	7	7	5	5	5	g,h,lim	GROUP_BY	
job_1447 2528_ 0017	1	1	5	5	5	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /210
job_1447 2528_ 0018	1	1	6	6	6	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 211

Pig script completed in 2 minutes, 16 seconds and 657 milliseconds (136657 ms)

Tabelle A.29: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 100 KB und Clustergröße 1+1

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 8561_ 0001	1	1	6	6	6	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /271
job_1447 8561_ 0002	1	1	7	7	7	4	4	4	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 8561_ 0003	2	1	7	7	7	4	4	4	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 8561_ 0004	1	1	6	6	6	5	5	5	g,h,lim	GROUP_BY	
job_1447 8561_ 0005	1	1	7	7	7	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /272
job_1447 8561_ 0006	1	1	6	6	6	5	5	5	i,j	GROUP_BY, COMBINER	s3:// output/ 273

Pig script completed in 2 minutes, 17 seconds and 585 milliseconds (137585 ms)

Tabelle A.30: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 1 GB und Clustergröße 1+8; Reducerzahl 8+11

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1442 9455_ 0007	16	2	17	15	16	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /155
job_1442 9455_ 0008	8	1	60	28	48	74	74	74	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1442 9455_ 0009	16	8	73	31	58	145	118	133	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1442 9455_ 0010	80	11	54	20	44	113	88	102	g,h,lim	GROUP_BY	
job_1442 9455_ 0011	1	1	19	19	19	7	7	7	C,D	GROUP_BY, COMBINER	s3:// output /156
job_1442 9455_ 0012	11	2	19	10	17	7	7	7	i,j	GROUP_BY, COMBINER	s3:// output/ 157

Pig script completed in 7 minutes, 38 seconds and 669 milliseconds (458669 ms)

Tabelle A.31: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 1 GB und Clustergröße 1+8; Reducerzahl 16+22

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_ 1442 9455_ 0019	16	2	17	14	16	6	5	6	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /161
job_ 1442 9455_ 0020	8	1	66	51	62	58	58	58	B, bWith- Epoch, jn,lim	GROUP_BY	
job_ 1442 9455_ 0021	16	16	74	49	64	86	48	76	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_ 1442 9455_ 0022	80	22	57	21	43	81	37	57	g,h,lim	GROUP_BY	
job_ 1442 9455_ 0023	1	1	19	19	19	5	5	5	C,D	GROUP_BY, COMBINER	s3:// output /162
job_ 1442 9455_ 0024	11	2	36	15	30	20	20	20	i,j	GROUP_BY, COMBINER	s3:// output/ 163

Pig script completed in 6 minutes, 31 seconds and 849 milliseconds (391849 ms)

Tabelle A.32: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 1 GB und Clustergröße 1+8; 1 Mapper pro 32 MB

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_ 1442 9455_ 0001	16	2	17	15	16	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /152
job_ 1442 9455_ 0002	32	1	38	27	34	56	56	56	B, bWith- Epoch, jn,lim	GROUP_BY	
job_ 1442 9455_ 0003	64	2	39	14	34	423	412	417	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_ 1442 9455_ 0004	320	11	33	7	23	256	177	224	g,h,lim	GROUP_BY	
job_ 1442 9455_ 0005	2	1	24	19	22	13	13	13	C,D	GROUP_BY, COMBINER	s3:// output /153
job_ 1442 9455_ 0006	33	2	37	15	31	21	21	21	i,j	GROUP_BY, COMBINER	s3:// output/ 154

Pig script completed in 14 minutes, 55 seconds and 473 milliseconds (895473 ms)

Tabelle A.33: Ausführungs-Logdaten des Heuristic Miners mit Eingabedateigröße 1 GB und Clustergröße 1+8; 1 Mapper pro 256 MB

JobID	Maps	Reds	Max Map Time	Min Map Time	Avg Map Time	Max Red Time	Min Red Time	Avg Red Time	Alias	Feature	Output
job_1447 8561_ 0031	4	2	27	26	27	5	5	5	a,b, callcenter, start3, start4	MULTI _QUERY, COMBINER	s3:// output /286
job_1447 8561_ 0032	4	1	81	39	60	75	75	75	B, bWith- Epoch, jn,lim	GROUP_BY	
job_1447 8561_ 0033	8	2	83	31	55	364	357	360	bWith- Epoch, bWith- Epoch2, d,e,f	HASH_JOIN	
job_1447 8561_ 0034	40	11	92	29	66	105	65	91	g,h,lim	GROUP_BY	
job_1447 8561_ 0035	1	1	16	16	16	8	8	8	C,D	GROUP_BY, COMBINER	s3:// output /287
job_1447 8561_ 0036	11	2	22	11	18	9	9	9	i,j	GROUP_BY, COMBINER	s3:// output/ 288

Pig script completed in 11 minutes, 12 seconds and 803 milliseconds (672803 ms)

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Name: Corvin Frey Matrikelnummer: 613235

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Corvin Frey