Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Tim Horgas

Performance-Analyse von Apache Spark und Apache Hadoop

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Tim Horgas

Performance-Analyse von Apache Spark und Apache Hadoop

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Wirtschaftsinformatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Zukunft
Zweitgutachter: Prof. Dr. Steffens

Eingereicht am: 24.09.15

Tim Horgas

Thema der Arbeit
Performance-Analyse von Apache Spark und Apache Hadoop

Stichworte
Apache Spark, Apache Hadoop, Big Data, Benchmarking, Performance-Analyse

Kurzzusammenfassung
Diese Bachelorarbeit beschäftigt sich im Kontext Big Data mit der Analyse der Performance von Apache Spark im Vergleich zu Apache Hadoop. Dabei werden Apache Hadoop und Apache Spark in Form eines Benchmarks verglichen und anschließend durch eine Nutzwertanalyse bewertet.
Tim Horgas

Title of the paper
Performance-Analysis of Apache Spark and Apache Hadoop

Keywords
Apache Spark, Apache Hadoop, Big Data, Benchmarking, Performance-Analysis

Abstract
This Bachelor-Thesis describes a Performance-Analyses of the frameworks Apache Spark and Apache Hadoop in context of Big Data. The Performance-Analyses contains a benchmark of Apache Hadoop and Apache Spark with an evaluation achieved as value benefit analysis of both frameworks.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Code

1 Einleitung

In diesem Kapitel wird zuerst der Kontext Big Data und die sich daraus ergebende Entwicklung von neuen Technologien dargestellt. Im Anschluss werden in diesem Zusammenhang kurz die Technologien Apache Hadoop und Apache Spark vorgestellt und zum Abschluss des Kapitels die Zielsetzung dieser Bachelorarbeit erläutert.

1.1 Big Data als Grundlage für neue Technologien

Big Data ist einer der primären IT-Trends der letzten Jahre. Auf Veranstaltungen wie z.B. der CeBIT 2015, der weltgrößten Messe der Informationstechnik, gehört Big Data zu den Markttrends und ist sowohl in der Forschung als auch in der Wirtschaft eines der meistbeachteten Themen in der Informatik. Der steigende Wert von Informationen in Verbindung mit einer immer größer werdenden Datenmenge stellt immer höhere Anforderungen an die Speicherung und Verarbeitung von Daten. Dieser Zusammenhang treibt die Entwicklung von Technologien im Bereich Big Data voran und wirft Fragen nach Performance und Anwendungsfällen auf. Um Fragestellungen aufzustellen und Anwendungsfälle in Bezug auf Big Data herauszuarbeiten, ist es notwendig, den Begriff Big Data zu analysieren. Es gibt bisher noch keine einheitliche Definition für den Begriff Big Data, jedoch haben sich in den letzten Jahren zwei Ansätze herausgestellt, die eine derzeitige Betrachtungsweise des Begriffes zulassen.

Der erste Ansatz zur Definition von Big Data ist wie folgt beschrieben: Big Data sind Datenmengen, die durch ihre Größe nicht durch traditionelle Datenbanksysteme in einer angemessen Zeit verarbeitet werden können (vgl. (Fre14), S.9). Als traditionelle Datenbanksysteme werden relationale Datenbankmanagementsysteme (DBMS) bezeichnet, die ACID-Eigenschaften besitzen und in der Praxis seit vielen Jahren für den Produktivbetrieb von klassischen Anwendungen wie zum Beispiel CRM-Systemen in Verwendung sind, mit der Abfragesprache SQL als Industriestandard. Traditionelle Datenbankmanagementsysteme sind für den Betrieb mit Datenmengen im Terrabyte-Bereich geeignet. Jedoch führt zum Beispiel die rasante Entwicklung des Internets in den letzten Jahren besonders durch Mobile Web oder die Verwendung von Sensordaten zu Datenmengen in der Größenordnung von Petabytes, bei denen traditionelle Systeme an ihre Grenzen stoßen. Ein Beispiel für die Anwendung solcher Datenmengen ist das

CMS-System der Europäischen Organisation für Kernforschung (CERN), dessen Datacenter nach eigenen Angaben ca. 1 Petabyte Daten pro Tag verarbeitet ((CER15)) und damit nicht mehr für traditionelle DBMS geeignet ist.

Der zweite Ansatz zur Begriffsbestimmung von Big Data wurde ursprünglich von ((Lan01), S. 1f) in Bezug auf E-Commerce vorgestellt. Mittlerweile hat sich dieser Ansatz zu einer der am häufigsten verwendeten Definitionen von Big Data entwickelt. Der Ansatz liefert folgende Definition der drei V's: Big Data weist in der Regel die 3 Eigenschaften *Volume* (sehr große Datenmengen), *Velocity* (hohe Geschwindigkeit der Datenerzeugung und -verarbeitung) und *Variety* (stark verschiedenartige Daten) auf. Diese Definition lässt sich auf das bereits erwähnte Beispiel des CMS-Sytems vom CERN anwenden: *Volume* sind hier eine Datenmenge von ca. 30 Petabyte, was in der aktuellen Zeit von der Größenordnung her eine riesige Datenmenge darstellt. Der bereits erwähnte Datendurchsatz von 1 Petabyte pro Tag basiert auf einer der aktuell am höchst möglichen Geschwindigkeiten der Datenerzeugung und -verarbeitung (*Velocity*). Die gesamte Datenmenge besteht aus Echtzeitdaten, Simulationsdaten und Metadaten, wobei diese stark verschiedenartig sind (*Veracity*). Damit erfüllt das CERN-CMS-System die aktuellen Ansätze zur Definition von Big Data.

Im Gegensatz zu anderen Markttrends und vergangenen Entwicklungen in der Informatik, sind Unternehmen die treibende Kraft bei der Entwicklung von Werkzeugen zur Verarbeitung von Big Data. Internet-Unternehmen wie zum Beispiel Facebook haben mit ihrem aktuellen Geschäftsmodell die Herausforderung, Datenmengen zu verarbeiten, die in traditionellen Unternehmensbranchen in der Regel noch nicht vorhanden sind. Zusätzlich sind die Daten dabei in vielen unterschiedlichen Formaten und weisen eine hohe Heterogenität auf. Die zunehmende Möglichkeit zur Verwendung von Sensordaten, Social-Media-Daten und bereits vorhandenen Daten aus heterogenen Quellen ist sowohl für Unternehmen als auch für die Forschung der primäre Antrieb zur Enwicklung von Technologien für den Kontext Big Data. Die Aufgabe der Forschung ist es dabei, für Big Data geeignete Werkzeuge auf ihren Anwendungsfall zu testen und geeignete andere und neue Anwendungsfälle zu finden und diese zu validieren.

1.2 Technologien zur Umsetzung von Big Data Analysen

Die Anfänge von Analysen im Kontext Big Data sind eng mit der Entwicklung des Internets verknüpft. Während dieser Enwicklung kam es zu vorher nie dagewesenen Datenmengen, die gespeichert und analysiert werden müssen. Die Datenmengen sind dabei in der Praxis so groß, dass die Speicherung nur verteilt umgesetzt werden kann und die Verarbeitung nur parallel in einem Cluster aus Computern möglich ist. Diese Tatsache führt zwangsmäßig zur

Entwicklung von neuen Technologien, die in der Lage, sind die Herausforderungen der drei V's gerecht zu werden. Google hat in dieser Entwicklung eine zentrale Rolle eingenommen und 2003 ein Konzept für ein verteiltes Dateisystem zur Speicherung von großen Datenmengen vorgestellt ((GGL03)). Weitere Ziele dieses Konzeptes sind eine hohe horizontale Skalierbarkeit und Fehlertoleranz in Clustern mit verhältnismaßig günstiger Standardhardware zu erreichen.

Horizontale Skalierung bezeichnet eine Steigerung der Rechenleistung durch das Hinzufügen von Rechnern zu einem Netzwerk aus tendenziell kleinen und miteinander verbundenen Servern (vgl. (MMSW07), S. 1). Im Gegensatz dazu steht die vertikale Skalierung, bei der die Steigerung der Rechenleistung durch einen kompletten Austausch von Servern durch noch leistungsfähigere Server erreicht wird. Alternativ kann auch der Austausch von Hardware-Bestandteilen durch leistungsfähigere Bestandteile diesen Effekt erzielen. Horizontale Skalierung ist durch die Möglichkeit der Verwendung von Standardhardware in vielen Fällen die kostengünstigere Variante der Skalierung und ist deshalb besonders für die Verwendung in der Wirtschaft interessant. Das in diesem Kontext entwickelte Google File System (GFS) war das Vorbild für die Implementation des Hadoop File Systems (HDFS), eine von ehemaligen Yahoo!-Mitarbeitern entwickelte Open-Source-Variante eines verteilten Dateisystems. HDFS ist Bestandteil des in Java geschriebenen Hadoop-Frameworks, das außerdem noch das Programmiermodell MapReduce beinhaltet. Das ursprünglich ebenfalls von Google Mitarbeitern vorgestellte MapReduce ist ein Programmiermodell mit dazugehöriger Implementation für die Verarbeitung von großen Datenmengen (vgl. (DG08), S. 1). Technologien im Bereich Big Data haben in der Regel eine „shared nothing"-Architektur. Diese ist verteilt, leicht skalierbar und oft in Verwendung mit nicht-relationalen Datenbanken. Dabei ist aber auch die Nutzung von verteilten relationalen Datenbanken möglich (vgl. (DK13), S. 2). Kennzeichnend für diese „shared nothing"-Architektur ist, dass jeder Computer im Cluster nur seine lokalen Ressourcen wie zum Beispiel CPU und Hauptspeicher nutzt (vgl. (KKW+13), S. 1).

Aufbauend auf Speicherung und Verarbeitung ist besonders die Analyse von Big Data, sowohl in der Forschung als auch in der Wirtschaft für neue Geschäftsfelder ein interessantes Themenfeld. Technologien für Big Data ermöglichen zum Beispiel die Entwicklung von Anwendungen im Bereich der personalisierten Services, Internet Sicherheit, personalisierten Medizin und digitalen Archiven (vgl. (FHL14), S. 5). Mit den Änderungen der Speicherung und Verarbeitung von Big Data durch neue Technologien sind auch für die Analyse von Big Data grundlegende Änderungen im Vergleich zu den Analyse-Werkzeugen der traditionellen Datenbanksystemen nötig. Die riesigen Datenmengen haben oft eine sehr große Anzahl an Dimensionen und starke Anhäufungen von Störwerten und statistischen Ausreißern, die wiederum zu zweifelhaften Korrelationen und unlogischen Zusammenhängen führen (vgl.

(FHL14), S. 5). Die Herausforderung in der Analyse von Big Data ist hier neben der Datenmenge, eine angemessene Verarbeitung und Fehlertoleranz für unzureichend oder gar nicht normalisierte Daten aus heterogen Quellen zu erreichen. Hinzu kommt die Notwendigkeit der Echtzeitverarbeitung als weiterer Trend, der besonders in der Industrie als Vision unter dem Begriff „Industrie 4.0" vorgestellt wird. Durch Big Data entstehen neue Chancen, wie beispielsweise die Umsetzung von Predictive Analysis mit dem Ziel, neue und verborgene Erkenntnisse durch Mustererkennung zu gewinnen. Big Data kann die Geschäftsmodelle durch solche Anwendungen erweitern oder komplett verändern und neue Geschäftsmodelle zum Beispiel durch Echtzeitverabeitung schaffen.

1.3 Apache Hadoop und Apache Spark

Das Open Source Framework Apache Hadoop ist eine Technologie, die den Anforderungen von Big Data gerecht wird. Apache Hadoop ist horizontal skalierbar und für die Verarbeitung von sehr großen Datenmengen konzipiert (vgl. (Had14b)). Apache Spark ist ein neues Framework, das nach Einschätzung der Entwickler formal die gleichen Anforderungen wie Apache Hadoop beherrscht. Allerdings soll mit Apache Spark die Geschwindigkeit von Anwendungen, durch die Verwendung eines neuen Programmiermodells und dem Fokus auf In-Memory-Verarbeitung, deutlich schneller sein als vergleichbare Anwendungen von Apache Hadoop (vgl. (Apa15a)).

1.4 Zielsetzung

In dieser Bachelorarbeit werden die Frameworks Apache Hadoop und Apache Spark anhand einer Performance-Analyse für die Benutzung im Kontext Big Data verglichen. Die Performance-Analyse basiert auf Benchmark-Tests, die auf Daten aus dem Google Books N-Gramm Korpus ausgeführt sind (verfügbar unter (Goo12)). Die Benchmark-Tests gehen dabei auf verschiedene Datenmengen und deren jeweilige Verarbeitungsgeschwindigkeit ein. Daraus werden Auswirkungen für die praktische Anwendung der beiden Frameworks gefolgert. Die Ergebnisse der Performance-Analyse sind in Form einer Nutzwertanalyse zusammengefasst und ausgewertet. Aus der Nutzwertanalyse werden abschließend Empfehlungen für die praktische Anwendung im Kontext Big Data abgeleitet.

2 MapReduce

Dieses Kapitel beschreibt das Programmiermodell MapReduce. Dabei wird am Anfang auf die Verwendung des Programmiermodells eingegangen und dann der grundsätzlichen Aufbau von MapReduce dargestellt. Im Anschluss werden die zwei zentralen Komponenten von MapReduce mit der Bezeichnung *map*- und *reduce*-Funktion vorgestellt. Zum Abschluss des Kapitels wird ein Fazit in Form von Vor- und Nachteilen gezogen.

2.1 Verwendung und Aufbau

MapReduce ist ein von Google-Mitarbeitern entwickeltes, stark skalierbares Programmiermodell für die Verarbeitung von großen Datenmengen (vgl. (DG08), S. 1). MapReduce wird von verschiedenen Unternehmen zum Beispiel für Web-Indizierung, statistische Auswertungen, Reporting, Data Mining und Machine Learning verwendet (Siehe (Had15b)). Wie viele Ansätze in der Informatik kann auch MapReduce zu den Divide and Conquer Verfahren gezählt werden. Divide and Conquer Algorithmen zeichnet aus, dass ein Problem in mehrere kleine Teilprobleme unterteilt wird und diese dann parallel bearbeitet werden (vgl. (LD10), S. 22). Für das Verfahren MapReduce sind die Daten verteilt im Cluster gespeichert. Die Daten werden während der Ausführung von MapReduce lokal von den einzelnen Knoten, auf denen die Daten vorhanden sind, verarbeitet.

Das Programmiermodell MapReduce basiert logisch auf den Funktionen *map* und *reduce* (letztere wird häufig auch als *fold* bezeichnet), die bereits aus funktionalen Programmiersprachen bekannt sind (vgl. (DG08), S. 2). In der funktionalen Programmierung wird der Funktion *map* eine Liste und eine Funktion übergeben. Die *map* Funktion berechnet dann als Ergebnis eine neue Liste, wobei die übergebene Funktion auf die übergebene Liste angewandt wird. Im Unterschied dazu nimmt im MapReduce-Modell die Map-Funktion ein Key-Value-Paar entgegen und liefert als Ergebnis eine Menge von neuen Key-Value-Paaren (vgl. (DG08), S. 2). Die *reduce* Funktion bekommt in der funktionalen Programmierung als Inputparameter eine Liste, eine Funktion und einen Inititalwert. Das Ergebnis ist dann ein Wert, wobei aufbauend auf dem Initialwert die übergebene Funktion auf die übergebene Liste angewendet wird und das reduzierte Ergebnis zurückgegeben wird. Die Reduce-Funktion im MapReduce-Modell

bekommt als Input eine Menge mit Key-Value-Paaren und verkleinert diese zu einer reduzierten Menge von Key-Value-Paaren oder zu einem Key-Value-Paar als Ergebnis (vgl. (DG08), S. 2).

Im Gegensatz zur traditionellen Verarbeitung von Daten im Cluster werden die Map-Funktionen und die Reduce-Funktionen als einzelne Einheiten automatisiert zu den Daten gesendet und dort lokal ausgeführt. Dabei müssen die Map-Funktionen und die Reduce-Funktionen bei den Daten zwingend hintereinander ausgeführt werden. Das Programmiermodell ist auf Cluster mit vielen Standardrechnern ausgerichtet, es bietet deshalb eine hohe Fehlertoleranz bei möglichen Hardwarefehlern und hohe Verfügbarkeit ohne zusätzlichen individuellen Programmieraufwand (vgl. (DG08), S. 1).

Die Beschaffenheit des Key-Value-Paares hängt dabei vom Anwendungsfall ab. Zur Verarbeitung einer Webindizierung als Key-Value-Paar eignet sich beispielsweise als Key die URL und als Value der HTML Inhalt (vgl. (LD10), S. 22). Ein möglicher Anwendungsfall zur Webindizierung ist unter der Bezeichnung URLCount bekannt. Bei diesem werden Weblogs gescannt, um herauszufinden, wie oft eine URL aufgerufen wurde.

2.1.1 Map-Funktion

Die Map-Funktion nimmt vereinfacht ein (Key, Value)-Paar entgegen, führt an diesem funktionsspezifische Operationen aus und liefert eine Liste von Paaren als Zwischenergebnisse in der Form: Liste(Key1...n, Value1...n). Am Beispiel des URL-Counts erzeugt die Map-Funktion für jede URL in einem Weblog ein (URL, "1")-Paar. Folgender Pseudo-Code veranschaulicht das Verfahren am Beispiel des URL-Counts:

Data: (Key, Value); //Key = URL, Value = "1"
Result: Menge von (Key, Value)-Paaren
map(String Key, String Value);
foreach *Key* **do**
  erzeuge (Key, “1“);
end

Algorithm 1: Map-Funktion
(Quelle: vgl. (DG08), S. 2)

Das MapReduce-Framework verteilt Key-Value-Paare mit gleichem Key zu den passenden Reduce-Fuktionen, dieser Prozess wird von einer Partitionierungsfunktion umgesetzt. Diese ist häufig als eine Hash-Funktion implementiert, wie zum Beispiel *(hash(key))* **mod** *R)* (vgl. (DG08), S. 4).

2.1.2 Reduce-Funktion

Der Reduce-Funktion werden alle Zwischenergebnisse mit demselben Schlüssel zugeordnet (Key1, Liste(Value1)) und die Werte funktionsspezifisch zu (Key1,ValuesReduced)) reduziert. Am Beispiel des URL-Counts wird für jede URL die Gesamtanzahl beziehungsweise deren Häufigkeit gezählt, indem der Value von den (URL, "1")-Paaren für jede URL addiert wird. Folgender Pseudo-Code veranschaulicht das Verfahren:

```
Data: (Key, list(Value); //Key = URL, Value = URL count
Result: Menge von (URL, total count)-Paaren
reduce(String Key, Iterator Values);
int result = 0;
foreach v in Values do
    result += ParseInt(v);
end
erzeuge(AsString(result))
```

Algorithm 2: Reduce-Funktion
(Quelle: vgl. (DG08), S. 2)

2.2 Bewertung des MapReduce-Verfahrens

Um das das MapReduce-Verfahren zu bewerten und mit anderen Verfahren vergleichen zu können, werden in den folgenden zwei Abschnitten die theoretischen Vor- und Nachteile des MapReduce-Verfahrens gezeigt.

2.2.1 Vorteile

Ein entscheidender Vorteil von MapReduce gegenüber anderen verteilten Methoden zur Datenverarbeitung ist die geringe Auslastung des Netzwerkes, da nicht die Daten sondern die Programme über das Netzwerk gesendet werden (vgl. (LD10), S.19). Ein weiterer Vorteil im Vergleich zu anderen Methoden der Datenverarbeitung im Kontext Big Data ist, dass die Daten nicht vorher nach vordefinierten Kriterien partitioniert werden müssen (vgl. (WSV14), S. 5). Die beiden Funktionen sind relativ einfach zu programmieren, außerdem ist der Änderungsaufwand für andere passende Anwendungsfälle relativ gering. MapReduce ist besonders für horizontale Skalierung geeignet und kann dadurch mit Anwendungsfällen umgehen, die stark wachsende Datenbestände aufweisen. Zusätzlich ist sowohl die Fehlerbehandlung bei Ausfall eines Knotens im Cluster, als auch die Verteilung der Programme automatisiert (vgl. (BB13), S.

1). Die parallele Verarbeitung und Synchronisation wird dabei von Frameworks automatisch umgesetzt und muss nicht extra vom Benutzer implementiert werden (vgl. (FDGR11), S. 1).

2.2.2 Nachteile

MapReduce ist tendenziell ein eher einschränkendes Programmiermodell. Die Reihenfolge der zwei Funktionen ist zwingend einzuhalten, wobei auch die Funktionen an sich für die Implementation einschränkendend sind. Zwar lassen sich viele Anwendungsfälle auf das Modell übertragen, jedoch sind besonders iterative und interaktive Vorgänge teilweise nicht abbildbar (vgl. (ZCF+10), S. 1). Auch Graph-Algorithmen lassen sich mit anderen Verfahren besser ausführen (Siehe (VMD+13), S. 3f). Durch die Verarbeitung im Cluster ist das Debugging der *Map*-Funktionen und *Reduce*-Funktionen generell schwierig umzusetzen (vgl. (DG08), S. 7). Die Echtzeitverarbeitung ist mit MapReduce nur sehr schwer möglich, was einer der größten Nachteile des Programmiermodells ist. Dies führt generell zu der Problematik, dass MapReduce nicht für zustandsbehaftete Berechnungen geeignet ist, die zum Beispiel in der objektorientierten Programmierung umgesetzt werden können (vgl. (Gra03), S. 8). Zustandsbehaftete Berechnungen sind aber oft nötig, um Probleme der realen Welt abzubilden. Bei komplexen Anforderungen an die Verarbeitung ist es oft notwendig, das 2-stufige MapReduce Verfahren mehrmals hintereinander mit unterschiedlichen Parametern auszuführen, um zu dem gewünschtem Ergebnis zu kommen (vgl. (ZCF+10), S. 1). Solche iterativen Vorgänge führen oft zu einer Verschlechterung der Performance.

3 HDFS

Dieses Kapitel beschreibt das verteilte Dateisystem HDFS, das ursprünglich für die Verwendung von MapReduce entwickelt wurde. Am Anfang des Kapitels wird dabei auf die Anwendungsmöglichkeiten und Ziele von HDFS eingegangen. Danach wird das theoretische Konzept der Master-Slave-Replikation beschrieben, welches im HDFS implementiert ist und für dessen Verständnis notwendig ist. Im Anschluss wird der Hardware- und Softwaretechnische Aufbau vom HDFS dargestellt, abschließend wird das HDFS im Form von Vor- und Nachteilen bewertet.

3.1 Anwendungsmöglichkeiten und Ziele von HDFS

Das Hadoop Distributed File System (HDFS) ist ein in der objektorientierten Programmiersprache Java geschriebenes verteiltes Dateisystem (vgl. (GGP14)), mit dem Ziel, auf einem Cluster aus Standardhardware zu laufen (vgl. (Had14a)). Vorbild für die Entwicklung von HDFS war das Programmiermodell MapReduce von Google, wobei HDFS im Vergleich zu anderen verteilten Dateisystemen die meiste Ähnlichkeit mit dem zum MapReduce Framework entwickelten Google File System (GFS) hat (Siehe (SKRC10), S. 1). Im Gegensatz zum GFS ist HDFS eine Open-Source-Implementation. Weitere von den Entwicklern genannte Ziele von HDFS sind Toleranz gegenüber Hardwarefehlern, ein hoher Datendurchsatz, eine geringe Netzwerkauslastung und die Speicherung von sehr großen Dateien, die allerdings nach der Speicherung nicht mehr viel verändert werden. Geeignete Anwendungsfälle sind aufbauend auf diese Ziele aus Sichtweise der IT zum Beispiel die Speicherung von Daten für Anwendungen wie MapReduce und Webcrawling (Siehe (Had14a)).

Aus wirtschaftlicher Sicht ist es auf Basis dieser Technologie ein Ziel, neue Anwendungen wie zum Beispiel personalisierte Services anzubieten, mit denen Kunden als Individuelle Entitäten behandelt werden sollen. Für solche Anwendungen ist die datenschutzkonforme Speicherung von persönlichen Preferenzen von Kunden in Bezug auf Produkte nötig, wobei diese Datensammlung zu großen Datenmengen entsprechend der drei V‘s führen kann. Als Speichersystem eignet sich dann zum Beispiel ein verteiltes Dateisystem wie HDFS. Aufbauend auf die Speicherung können Informationen generiert werden, um anschließend umfassende Kundenprofile zu erstellen. Dazu müssen vorhandene Daten aus unterschiedlichen Quellen

gesammelt und analysiert werden. Zusätzlich ist dann oft eine Integration von neuen Daten notwendig, um für das Geschäftsmodell nützliche Informationen zu generieren. In lange am Markt existierenden Unternehmen sind über Jahre oder teilweise Jahrzehnte Datensilos entstanden, von denen nur ein geringer Anteil analysiert wird. Die zentrale Anwendung für HDFS ist dann die Speicherung von Daten, an denen Operationen der Muster- und Modellerkennung aus dem Wissenschaftsgebiet Data Science gemacht werden können. Data Science umfasst dabei die Umformung von Daten in Infomationen zur Enwicklung von Vorhersagen, Produkten oder Lösungen (vgl. (CLS14)). Ein weiterer Trend in der Wirtschaft ist die Echtzeitverarbeitung, wobei diese in oben genannten Zielen von HDFS nicht explizit aufgelistet ist. Dies wirft Fragen in Bezug auf die Performance von HDFS in solchen Szenarien auf.

Eine Grundlage für die Funktionsweise von HDFS ist die im Folgenden vorgestellte Master-Slave-Replikation. Diese ist die theoretische Basis für den Aufbau des HDFS-Dateisystems und setzt sich aus den Begriffen Replikation und Master-Slave-Architektur zusammen.

3.2 Master-Slave-Replikation

Replikation ist die absichtlich redundante Speicherung von Daten zur Erhöhung der Verfügbarkeit und der Performance in Bezug auf die Fehlertoleranz und die Geschwindigkeit der Lesezugriffe (vgl. (RSS15), S. 285). In der Regel werden dabei die Daten in einem Cluster verteilt und mehrfach gespeichert. Zur Erhöhung der Verfügbarkeit wird dabei die redundante Speicherung eingesetzt. Diese erhöht die Fehlertoleranz in einem Cluster, da bei Ausfall eines Knotens im Idealfall mehrere Replikate vorhanden sind und der Zugriff auf die benötigten Daten bei einem Replikat geschehen kann. Die redundante Speicherung führt in Kombination mit der Datenlokalität zu einer erhöhten Geschwindigkeit, da immer vom lokal am nächsten gelegenen Knoten im Cluster gelesen wird. Entsprechend ist die Geschwindigkeit beim Lesen maximal, denn der Lesezugriff geht über den kürzestmöglichen Weg, um auf die im Cluster zu lesenden Daten zuzugreifen.

Die Replikation wird im HDFS über eine Master-Slave-Architektur realisiert (vgl. (RSS15), S. 66). In der theoretischen Definition einer Master-Slave-Architektur gibt es einen Master-Knoten mit mehreren Slave-Knoten (vgl. (SF12)). Der Master übernimmt die Verteilung und Koordination der Speicherungs- und Änderungsoperationen an den Daten. Die Slaves sind Speichereinheiten für die Daten, alle Operationen werden durch den Master koordiniert.

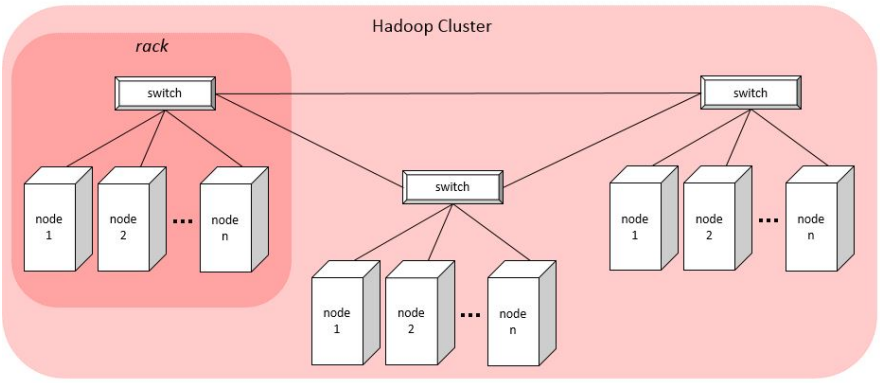


Abbildung 3.1: Beispielhaftes Hadoop Cluster mit Zusammensetzung aus mehreren *racks* (Quelle: Eigene Darstellung)

3.3 Aufbau von HDFS

Der Aufbau von HDFS ist in den folgenden zwei Abschnitten in den Hardware-technischen Aufbau und den Software-technischen Aufbau aufgeteilt. Der Schwerpunkt liegt dabei auf dem Software-technischen Aufbau, dieser wird deshalb ausführlich beschrieben.

3.3.1 Hardware

Das HDFS benötigt folgende grundlegende Hardware-Komponenten: *nodes* (engl. für Knoten) können beliebige Rechner mit einer für HDFS benötigten Java-Installation sein. Ab einer hohen Anzahl von *nodes* sind diese zu *racks* verbunden. Ein *rack* ist ein Cluster aus mehreren *nodes*, die physisch nah beienander liegen und über den gleichen Netzwerk-Switch miteinander verbunden sind. Durch diese Eigenschaft eines *racks* ist die Bandbreite des Netzwerkes zwischen *nodes* in einem *rack* höher als die Bandbreite zwischen *nodes* aus verschiedenen *racks*. Mehrere *racks* bilden ein HDFS-Cluster (Siehe Abbildung 3.1).

3.3.2 Software

HDFS ist vom Interface her wie ein UNIX-Dateisystem aufgebaut, jedoch sind in der Entwicklung Abstriche bei der Einhaltung von Standards aufgrund der Performance gemacht worden (Siehe (SKRC10), S. 1). HDFS basiert in seiner ursprünglichen Definition auf einer Single-Namespace-Architektur, dabei ist der Namensraum der Dateien von den Dateien selber entkoppelt (vgl. (Shv10), S. 7). Diese ist in Bezug auf die Datenverteilung eine Erweiterung der Master-Slave-Architektur. Der Namensraum mit den Metadaten des Dateisystems liegt

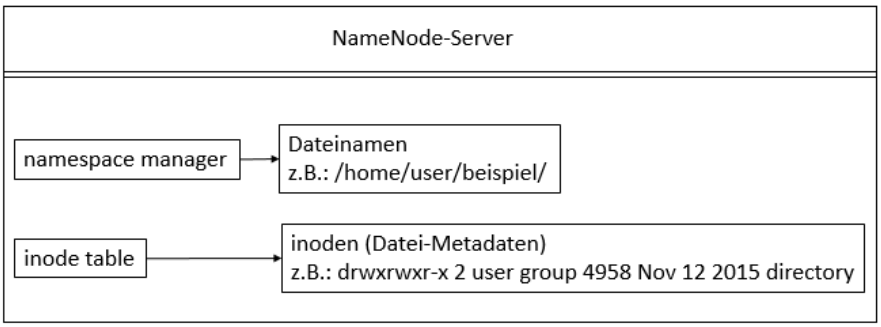


Abbildung 3.2: Komponenten eines NameNodes
(Quelle: Eigene Darstellung nach (Apae) und (Wik15))

auf einem als Java-Software laufenden Server mit der Bezeichnung *NameNode*. Die Dateien selber sind ebenfalls auf Servern, die als Java-Software laufen, mit der Bezeichnung *DataNodes* gespeichert. Der Hauptvorteil dieser Architektur ist die Skalierbarkeit des Dateisystems (vgl. (SKRC10), S. 7). Die Single-Namespace-Architektur erlaubt die vertikale Skalierung des *NameNodes* und die horizontale Skalierung der *DataNodes*. Insgesamt kann HDFS durch die Verwendung von Java als sehr portable Programmiersprache auf einer sehr großen Auswahl an Rechnern installiert werden, welche typischerweise mit GNU/Linux-Betriebssystemen ausgestattet sind (vgl. (Had14a)).

Der *NameNode*-Server übernimmt dabei neben der Speicherung von allen HDFS Metadaten die Vermittlung und Verteilung der Daten über das gesamte Cluster (vgl. (Had14a)). In Abbildung 3.2 sind die zentralen Komponenten eines *NameNodes* abgebildet. Die *namespace manager* Komponente verwaltet dabei die Dateinamen in einer Tabelle, die *inode table* Komponente speichert die Datei-Metadaten ebenfalls in einer Tabelle (Siehe Codekommentar: (Apae), Zeilen 135-145). Zu den Datei-Metadaten gehören zum Beispiel Zugriffsrechte, Benutzer, Gruppen und der letzte Zugriff auf die Datei (Siehe (Apac)).

Die *DataNodes* sind für die physische Speicherung der Daten zuständig. Die Dateien werden im HDFS je nach Dateigröße in einen oder mehrere Blöcke (*blocks*) aufgeteilt und auf die *DataNodes* verteilt (vgl. (Had14a)). Jede Datei wird im HDFS als eine Sequenz von Blöcken gespeichert, außer dem letzten Block haben alle Blöcke die gleiche Größe. Die Standardgröße für einen *block* ist dabei 64 MB. Das unterliegende Betriebssystem kann die *blocks* dann betriebssystemspezifisch in weitere Datenblöcke unterteilen. Die *DataNodes* bieten neben Lese- und Schreiboperationen auf *blocks* noch weitere Operationen, wie zum Beispiel das Erstellen und Löschen von *blocks* (vgl. (Apad), Zeilen 212-222) und Einstellungen zu der Replikation an. Ein *DataNode* Server verwaltet die *blocks* mit Hilfe einer Tabelle. Die *blocks* bestehen wiederum

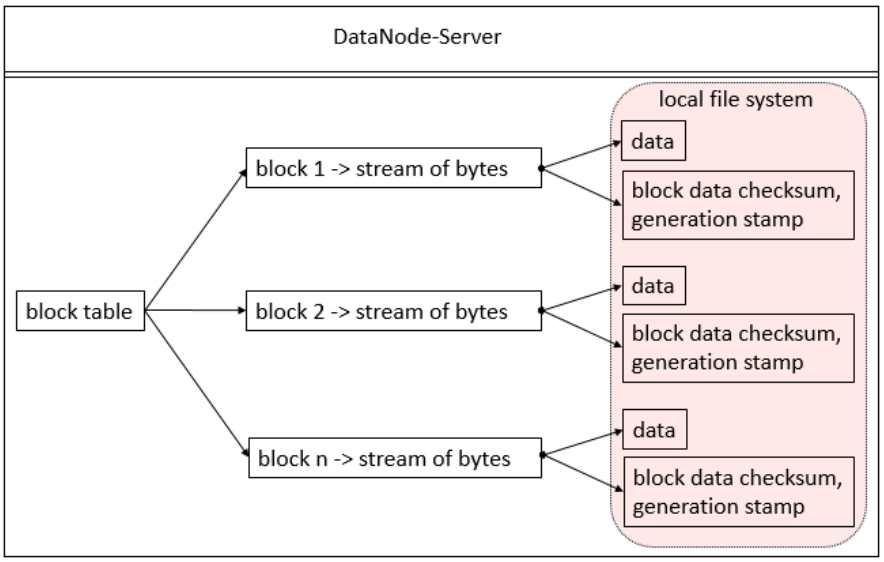


Abbildung 3.3: Komponenten eines DataNodes
(Quelle: Eigene Darstellung nach (Apad) und (SKRC10))

aus jeweils zwei Dateien, die im lokalen Dateisystem des *DataNodes* gespeichert werden. Die erste Datei beinhaltet die Daten, die zweite die dazugehörigen Metadaten. Die Metadaten beinhalten eine Prüfsumme und einen Versionsstempel (vgl. (SKRC10)). In Abbildung 3.3 sind die Komponenten eines *DataNodes* abgebildet.

Die Replikation ist in HDFS standardmäßig auf 3 Kopien (Replikate) pro *block* eingestellt. Die Anzahl der Replikate pro *block* im HDFS wird als Replikationsfaktor bezeichnet (Siehe (Had14a)). Bei der Standardeinstellung von 3 Replikaten pro *block* werden zwei Replikate auf jeweils unterschiedliche *nodes* im lokalen *rack* verteilt, das dritte Replikat wird auf einen *node* in einem anderen *rack* gespeichert. Ein entscheidender Baustein für die Replikation ist der *block report*. Dieser wird von jedem *data node* beim Startup generiert und beinhaltet eine Liste mit allen *data blocks*, die auf dem jeweiligen *Datanode* gespeichert sind. Der *block report* wird in regelmäßigen Abständen von den *DataNodes* an den *NameNode* gesendet, damit der *NameNode* Entscheidungen bezüglich der Replikation treffen kann. Zusätzlich senden die *DataNodes* standardmäßig alle 3 Sekunden *heartbeats*, um anzuzeigen dass der *DataNode* als Knoten im Cluster verfügbar ist (vgl. (SKRC10), S. 11).

HDFS-Clients bilden die Schnittstelle zwischen Benutzerprogrammen und dem HDFS (vgl. (SKRC10)). HDFS-Clients sind eine Code Bibliothek (vgl. (SKRC10)), die für den Datenzugriff zuerst den *NameNode* für die Datenposition kontaktieren und dann eine Datenpipeline zu dem bestimmten *DataNode* für Datenoperationen herstellen (vgl. (Shv10), S. 7). Die Daten-

position wird vom *NameNode* mit Hilfe der *block reports* ermittelt. Bei Schreiboperationen von einer Datei, die in mehreren *blocks* gespeichert werden muss, werden entsprechend der Datenlokalität zuerst der erste *block* der Datei gespeichert und im Anschluss der zweite und dann alle folgenden *blocks*. Für jeden einzelnen *block* müssen dabei der *block* selber und alle seine Replikate gespeichert werden, wobei für jeden *block* jedesmal vom *NameNode* der *block report* zur Adressierung abgefragt werden muss (vgl. (SKRC10), S. 2). Entsprechend werden alle zu der Datei gehörenden *blocks* des positionsmäßig am nächsten gelegenem *DataNodes* gespeichert. Im Anschluss werden alle Replikate gespeichert. Dieser Vorgang macht besonders Schreiboperationen sehr aufwändig und ist exemplarisch für den ersten *block* einer Datei in Abbildung 3.4 dargestellt.

Im ersten Schritt kontaktiert der HDFS-Client den *NameNode*, um das Schreiben der Datei file1 zu anzukündigen, die aus 3 *blocks* besteht. Der *NameNode* bestimmt die *DataNodes*, auf denen Block1 gespeichert werden soll und liefert den *block report*. In Abbildung 3.4 sind das DataNode1, DataNode2, DataNode3 und DataNode11. Unter der Annahme dass DataNode1 im Hadoop Cluster am nähesten am Client liegt, wird Block1 auf diesem als erstes gespeichert. Im Anschluss wird Block1 auf DataNode2 und auf DataNode3, gespeichert, da diese im selben *rack* wie DataNode1 liegen. Als letztes wird Block1 auf DataNode11 gespeichert, da sich dieser in einem anderen *rack* befindet. Die Speicherung der Replikate wird über eine direkte *pipeline* zwischen den *DataNodes* durchgeführt. Da file1 aus insgesamt 3 *blocks* besteht, wird der Prozess im Anschluss mit Block2 und Block3 wiederholt, was zur Vereinfachung in Abbildung 3.4 nicht mehr abgebildet ist.

3.4 Bewertung von Hadoop HDFS

Für die Bewertung des HDFS sind in den folgenden zwei Abschnitten sind die Vor- und Nachteile des verteilten Dateisystems erklärt. Zuerst werden dabei die Vorteile genannt und anschließend die Nachteile einschließlich eines Beispiels erläutert.

3.4.1 Vorteile

HDFS ist für sehr große und unstrukturierte Datenmengen geeignet, wobei gleichzeitig die Dateien an sich sehr groß sein können. Die Replikation ist automatisch integriert und sorgt neben der Backupfunktion für eine automatische Fehlertoleranz in bei Hardwareausfällen und für sehr schnelle Lesezugriffe im Cluster. Die Master-Slave-Architektur ist beliebig skalierbar und ist gut geeignet für stark wachsende Datenbestände. Diese Skalierbarkeit macht ein HDFS-Cluster durch den möglichen Einsatz von Standardhardware sehr kostengünstig. In

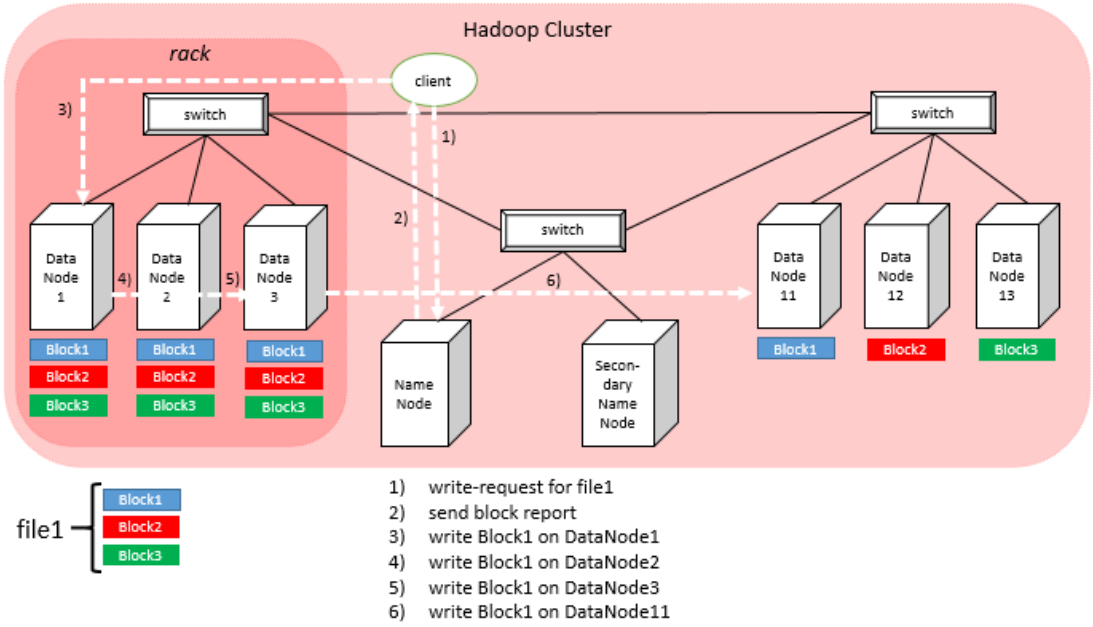


Abbildung 3.4: Beispielhaftes Hadoop Cluster mit Schreibvorgang an der Datei file1 (Quelle: Eigene Darstellung)

Kombination mit MapReduce verringert die Methode der Programmverteilung zusätzlich die Netzwerklast, da dabei die Programme zu den Daten gebracht werden und durch Replikation weniger Daten durch das Netzwerk gesendet werden (Siehe (Had14a)).

3.4.2 Nachteile

HDFS ist nur für große Datenmengen und große Dateien entwickelt, die Verarbeitung von zu kleinen Datenmengen und Dateiengrößen ist im Vergleich zu traditionellen Datenbanksystemen ineffizient (Siehe (PPR+09), S. 13f). Der Grund für diese Ineffizienz ist die Replikation. Die Speicherung von einer großen Datei braucht weniger *blocks* als die Speicherung von vielen kleinen Dateien, die in der Gesamtmenge an benötigten Speicherplatz genauso viel verbrauchen wie die große Datei (Siehe (WSV14), S. 18). Eine Datei mit der Größe von 2 Gigabyte braucht bei einer *block*-Größe von 64 Megabyte im HDFS inklusive Replikas 96 *blocks* zur Speicherung (2048 MB / 64 MB = 32 MB * 3 Replikas = 96 *blocks*). Die Vergleichsrechnung von 2000 kleinen Dateien mit jeweils 1 Megabyte Speicherplatzverbrauch illustriert die Problematik. In jedem *block* im HDFS kann genau eine Datei gespeichert werden, also inklusive Replikas 6000 *blocks* (2000 Dateien * 1 MB * 3 Replikas = 6000 *blocks*). Jedoch sehen auch die Entwickler HDFS auch nur als eine Ergänzung zu einer Speicherung in traditionellen Datenbanksystemen, die eben

genannte Situation mit vielen kleinen Dateien ist daher kein geeigneter Anwendungsfall. Im Gegensatz zu den schnellen Lesezugriffen sind Schreiboperationen durch die Master-Slave-Replikation sehr aufwendig. HDFS ist für einen *write-once-read-many*-Zugriff konzipiert (Siehe (Had14a)). Bei einem Schreibzugriff auf einer Datei, die aus mehreren *blocks* besteht, wird für jeden *block* der *NameNode* vom HDFS-Client aufgerufen, um das Schreiben des *blocks* auf einen *DataNode* zu adressieren und zusätzlich das Schreiben des *blocks* auf die vom *NameNode* bestimmten Replikate einzuleiten. Für jeden einzelnen dieser Schreibzugriffe muss eine *pipeline* zum jeweiligen *DataNode* hergestellt werden.

Grundsätzlich ist in der Standardimplementation vom HDFS nur ein Master-Knoten in Form eines *NameNodes* vorhanden, dieser stellt einen „Single Point of Failure" dar. Zwar haben die Entwickler diese Problematik erkannt und einen *SecondaryNameNode* eingeführt, jedoch ist dieser ausdrücklich kein Backup. Die Funktionalität des *SecondaryNameNodes* beschränkt sich stattdessen auf die periodische Aktualisierung der Metadaten (Siehe (WSV14), S. 22). Es besteht dennoch aufgrund der Architektur die Gefahr eines Clusterausfalls und in diesem Fall die Notwendigkeit eines manuellen Eingriffs (Siehe (Had14a)). Für einen Vergleich von Apache Spark und Apache Hadoop in einem HDFS Cluster kann es also in den Benchmarks in Kapitel 6 interessant sein, auch die Performance bei Schreibzugriffen zu testen. Jedoch ist darauf zu achten, große Dateien zu verwenden, die zum Beispiel für den Bechmark in dem Google Books N-Gramm Korpus vorhanden sind.

4 Apache Hadoop

In diesem Kapitel wird das Framework Apache Hadoop vorgestellt. Dazu werden am Anfang des Kapitels die Anwendungsmöglichkeiten und Ziele von Apache Hadoop beschrieben, gefolgt von dem Aufbau des Frameworks. Der Aufbau ist in die Komponenten YARN mit dessen Architektur, Hadoop MapReduce und die Erweiterung Apache Hive aufgeteilt. Zum Abschluss des Kapitels wird Apache Hadoop in Form von Vor- und Nachteilen bewertet.

4.1 Anwendungsmöglichkeiten und Ziele von Apache Hadoop

Apache Hadoop ist ein in der objektorientierten Programmiersprache Java geschriebenes Framework für die Speicherung und Verarbeitung von Daten im Cluster (vgl. (Had14b)). Dabei ist Apache Hadoop für so große Datenmengen entwickelt, dass die Möglichkeiten zur Verarbeitung in traditionellen Datenbanksystemen überschritten sind. Ursprüngliche Ziele von Hadoop sind die Bereitstellung von Open-Source-Software für zuverlässige, skalierbare und verteilte Computerberechnungen (Siehe (Had14b)). Das Hadoop Framework wird nach dem Vorbild von Google für die Verarbeitung von großen Datenmengen benutzt. Als erster Schritt müssen die Daten aus Quellen extrahiert und gespeichert werden (ETL), der zweite Schritt ist dann die Verarbeitung der Daten zu Informationen. Hadoop bietet sowohl Funktionalität zur Speicherung von Daten durch die Komponente HDFS, als auch zur Verarbeitung der Daten durch die Komponente Hadoop MapReduce. Die für Apache Hadoop geeigneten Daten können dabei dem Begriff Big Data entsprechend der drei V's zugeordnet werden. Jedoch ist es notwendig, den Anwendungsfall für die Nutzung des Hadoop-Frameworks genau auf die Eigenschaften der drei V's zu untersuchen. Die Entwickler von Apache Hadoop nennen als geeignete Anwendungsfälle zum Beispiel Webcrawling- und Map/Reduce Anwendungen (Siehe (Had14a)).

Besonders die Generierung von Informationen aus Big Data ist eine entscheidene Funktion für die wirtschaftliche Entwicklung von Unternehmen. Die Nachteile von MapReduce (beschrieben im Abschnitt 2.2.2) führten zu der Entwicklung von weiteren Frameworks, um mehr Funktionalität als das eher einschränkende Programmiermodell von MapReduce zu bieten.

4.2 Aufbau von Hadoop

Hadoop (Version 2.7.1, Veröffentlicht am 06.07.2015) besteht grundsätzlich aus den 4 Komponenten Hadoop Common, HDFS, Hadoop MapReduce und YARN. Hadoop Common ist ein Modul zur Unterstützung der anderen Komponenten (Had14b). Die theoretischen Konzepte zu Hadoop MapReduce sind in Kapitel 2 erklärt, auf produktspezifische Eigenschaften des Frameworks wird in diesem Kapitel in Abschnitt 4.2.2 eingegangen. Die Grundlage von Hadoop bildet das bereits in Kapitel 3 beschriebene verteilte Dateisystem HDFS. Da Hadoop ursprünglich nur für MapReduce Anwendungen entwickelt worden ist, wurde für zusätzliche Erweiterungen die Komponente *YARN: Yet Another Ressource Negotiator* entwickelt. YARN ermöglicht es, mehr Funktionalität für das Hadoop-Framework durch neue Anwendungen wie zum Beispiel Apache Hive und Apache Spark bereitzustellen. Es erlaubt ebenfalls unterschiedliche Programme wie zum Beispiel MapReduce-Anwendungen und Spark-Anwendungen zu derselben Zeit parallel laufen zu lassen (vgl. (Whi12), S. 196). Auf YARN wird in diesem Kapitel in folgendem Abschnitt genauer eingegangen.

4.2.1 Architektur von YARN

Durch YARN wird das Ressourcen Management vom Programmiermodell getrennt, mit dem Ziel die Clusterauslastung zu verbessern und andere Programmiermodelle zu ermöglichen (vgl. (VMD$^+$13), S. 2). Die Trennung des Resourcen Managements vom Programmiermodell bei der Einführung von YARN in Hadoop Version 0.23 wird primär durch die neu eingeführten Komponenten *ResourceManager* und *ApplicationMaster* umgesetzt. Es gibt es dabei einen *ResourceManager* pro Cluster, der die globale Verteilung der Ressourcen im Cluster regelt (Siehe (Had15c)). Der *ResourceManager* überprüft durch Scheduling den Status und den Resourcenverbrauch der textitNodes und erzwingt Invarianten im Cluster (vgl. (VMD$^+$13), S. 4). Für das Management und die Koordination der Programme gibt es *ApplicationMaster*, die lokal auf ihrem *Node* die Ausführung und das Monitoring der Programme übernehmen (Siehe (Had15a)). Scheduling und Monitoring sind Begriffe aus der Betriebssystemtheorie. Scheduling kann als Planung der Ausführung von zeitlich parallel laufenden Prozessen bezeichnet werden (vgl. (Man13), S. 3), Monitoring ist ein Konzept zur Synchronisation dieser Prozesse (vgl.(Man13), S.168). Programme, die durch *ApplicationMaster* gesteuert werden, können dabei klassische MapReduce-Jobs oder mehrere Jobs in Ausführungsreihenfolge eines gerichteten azyklischen Graphs (DAG = Directed Acyclic Graph) sein. Einen gerichteten azyklischen Graphen zeichnet in diesem Zusammenhang aus, dass kein Knoten während der Traversierung mehrmals besucht

werden darf (vgl. (WSV14), S. 24). Weitere Komponenten innerhalb von YARN sind *Container* und *NodeManager*.

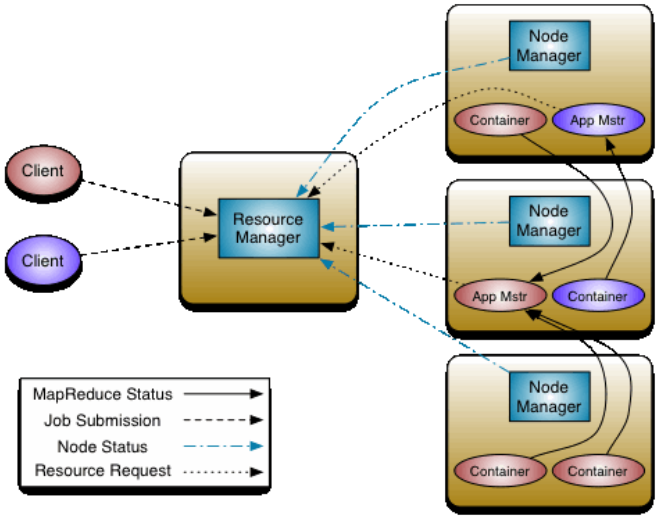


Abbildung 4.1: Architektur von YARN
(Quelle: (Had15a))

Container sind ein abgegrenzter Bereich mit zugewiesenen Systemressourcen (vgl. (WSV14), S. 26), sie bilden die Basiseinheiten im YARN Framework. Die zugewiesenen Sytemressourcen können dabei CPU-Kerne, Hauptspeicher, Festplattenspeicher und Netzwerkdienste sein (Siehe (Had15a)). *Container* befinden sich auf einzelnen *Nodes* im Cluster und sind fest an einen *Node* gebunden, es kann also keine Überschneidung eines *Containers* über mehrere *Nodes* geben (vgl. (WSV14), S. 26). Programme werden in einem oder mehreren *Containern* ausgeführt, die physische Zuweisung der *Container* wird durch den *NodeManager* geregelt (vgl. (WSV14), S. 26).

NodeManager sind auf allen *Nodes* im Cluster vorhanden und können als Slave-Services bezeichnet werden (vgl. (WSV14), S. 26), deren Tätigkeit auf jeden einzelnen *Node* beschränkt ist. Sie verwalten den kompletten Lebenszyklus von allen *Containern*, die auf dem jeweiligen *Node* vorhanden sind. Die Aufgaben eines *NodeManagers* sind die Identifikation von *Containern*, das Monitoring der *Container* und das Management von Abhängigkeiten der *Container* (vgl. (VMD+13), S. 7). Außerdem sind die *NodeManager* für das Aufräumen des Speichers im lokalen Dateisystem des *Nodes* und für das Löschen von *Containern* zuständig (vgl. (VMD+13), S. 7). Die Informationen über die Ressourcenauslastung der *Container* werden zuerst beim Start eines *Nodes* und danach in regelmäßigen Abständen von den *NodeManagern* an den *RessourceManager* gesendet.

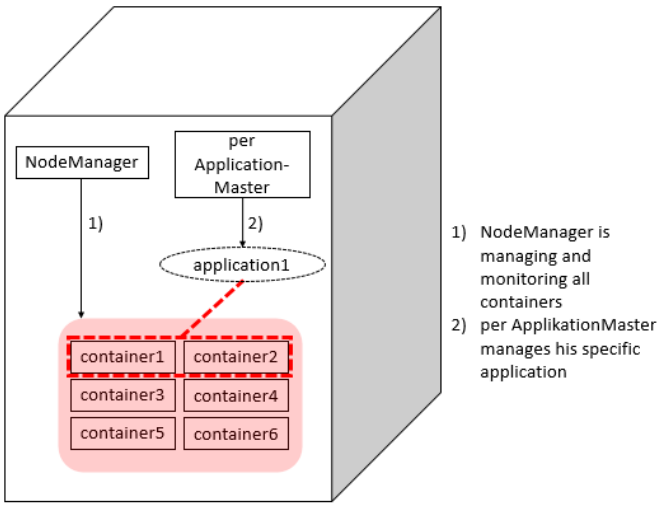


Abbildung 4.2: Slave-*Node* in einem YARN Cluster
(Quelle: Eigene Darstellung)

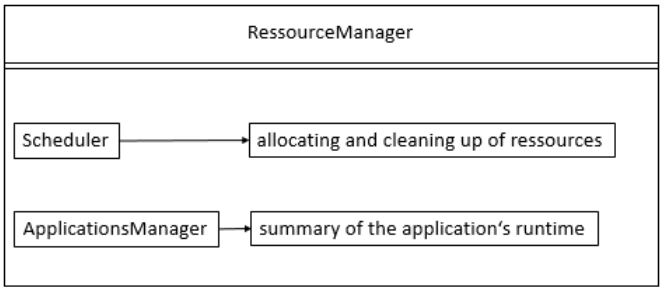


Abbildung 4.3: Wichtige Komponenten des *RessourceManagers*
(Quelle: Eigene Darstellung nach (Apag) und (Apai))

Der *ResourceManager* gleicht einem globalen Modell eines Status des gesamten Clusters, dessen Basis die benötigten Ressourcen von den laufenden Programmen ist (vgl. (VMD+13), S. 5f). Der *ResourceManager* hat zwei Hauptkomponenten, den *Scheduler* und den *ApplicationsManager* (Siehe (Had15a)). Die primäre Funktion des *ResourceManagers* ist der *Scheduler* (Siehe (WSV14), S. 27), wobei dieser für eine optimale Clusterauslastung sorgt. Dafür braucht der *Resource-Manager* sehr genaue Informationen über die Ressourcen, die von den Programme benötigt werden (vgl. (VMD+13), S. 6). Der *Scheduler* erhält Eigenschaften in Form von Ressourcen durch die Annahme von Statusberichten der *Container*, die über die *NodeManager* an den *RessourceManager* gesendet werden. Dabei bezieht der *ResourceManager* auch Kriterien wie zum Beispiel Kapazitäten und Warteschlangen mit ein (Siehe (Fre14), S. 26). Allerdings arbeitet der *ResourceManager* nur mit übergreifenden Ressourcen-Profilen für jedes der laufenden

Programme. Lokale Optimierungen und die interne Ausführung der Programme werden nicht vom *ResourceManager* optimiert. Der *ResourceManager* ist nur für das Echtzeit-Scheduling von Ressourcen verantwortlich (vgl. (VMD+13), S. 6). Die *ApplicationsManager* Komponente akzeptiert Programmanfragen und initialisiert auf einem *Node* den zu dem Programm zugehörigen *ApplicationMaster* (vgl. (Had15a). Im Fehlerfall kann die *ApplicationsManager* Komponente den *ApplicationMaster* neustarten.

ApplicationMaster sind für die Ausführung und das Monitoring der Programme zuständig (vgl. (Had15c)). Dazu nimmt ein *ApplicationMaster* in Abstimmung mit dem *Scheduler* des *RessourceManagers* geeignete *Container*, tracked deren Status und erledigt das Monitoring für den Ausführungsfortschritt des Programmes (vgl (Had15a)). Zu den Aufgaben des *ApplicationMasters* gehört außerdem die Generierung eines Ausführungsplans für die physische Ausführung eines Programmes sowie die Koordination der Fehlerbehandlung (VMD+13), S. 4). Für die Generierung des Ausführungsplans ist die Abstimmung mit dem *ResourceManager* über die Ressourcen notwendig. Auch die Aufgabe, den Status von laufenden Programmen zur Verfügung zu stellen ist dem jeweiligen *ApplicationMaster* zu zuordnen (vgl. (VMD+13), S. 6).

4.2.2 Hadoop MapReduce

Neben dem Hardware- und Softwaretechnischen Aufbau eines Hadoop-Clusters braucht eine lauffähige MapReduce- bei der Verwendung im Hadoop-Framework an sich nur 4 Komponenten. Die 4 Komponenten sind eine Mapper-Klasse, eine Reducer-Klasse, eine Klasse für die Inputputformatierung und eine Klasse für die Outputformatierung (Siehe (WSV14), S. 13). Eine MapReduce Anwendung wird dabei als MapReduce-Job bezeichnet, dieser wird wiederum in einzelne Tasks unterteilt. Im Folgenden ist der Ablauf eines MapReduce-Jobs in einem HDFS Cluster mit YARN beschrieben:

Ein *Client*, der in seiner eigenen Java Virtual Machine läuft, initialisiert einen MapReduce-Job und sendet für diesen eine Anfrage an den *ResourceManager* (vgl. (Apaa)). Der *ResourceManager* bestätigt diese Anfrage und vergibt eine *applicationID* (vgl. (Whi12), S. 197). Der *Client* spezifiziert den Output des MapReduce-Jobs, generiert Inputsplits und kopiert die Ressourcen des Jobs in das HDFS (vgl. (Whi12), S. 197). Zu den Ressourcen gehören die JAR und Konfigurations- und Splitinformationen (vgl. (Whi12), S. 197). Im Anschluss teilt der *Client* dem *RessourceManager* mit, dass der MapReduce-Job bereit für die Ausführung ist.

Die Ausführungsbestätigung wird im *RessourceManager* intern von der *ApplicationsManager*-Komponente an die *Scheduler* Komponente weitergeleitet, wobei diese dann einen geeigneten *Container* sucht. Wenn ein geeigneter *Container* bestimmt ist, wird der *Container* vom *NodeManager* initialisiert und es wird in diesem *Container* der *ApplicationMaster* gestartet. Der

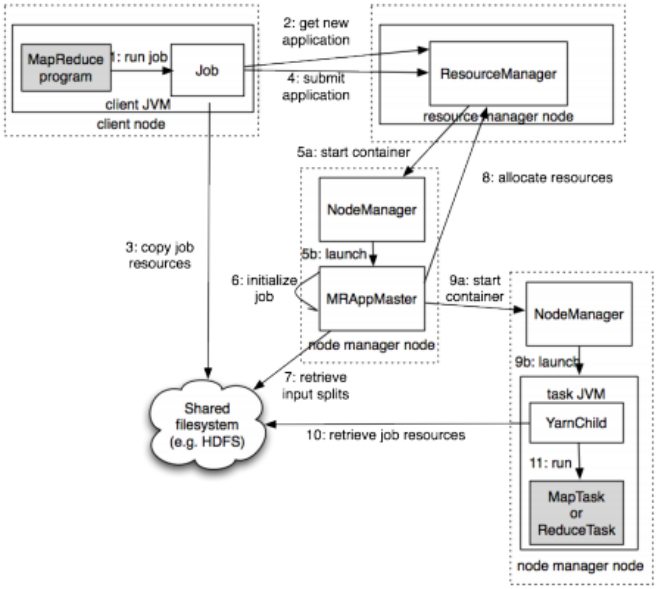


Abbildung 4.4: Ausführung einer MapReduce-Anwendung mit YARN
(Quelle: (Whi12), S. 197)

ApplicationMaster registriert sich beim *RessourceManager* und wird dann vom *NodeManager* mit allen anderen *Containern* auf dem *Node* überwacht (vgl. (Whi12), S. 197). Der *ApplicationMaster* initialisiert Objekte, die unter anderem Informationen über den Job-Fortschritt während der Ausführung erhalten (vgl. (Whi12), S. 197). Im Anschluss muss der *ApplicationMaster* die Inputsplits abfragen, die vom *Client* im HDFS erstellt wurden und erstellt für jeden Inputsplit einen Map-Task und entsprechende Reduce-Tasks (vgl. (Whi12), S. 197). An bestimmten Kriterien, unter anderem die Anzahl der *mapper* und *reducer*, entscheidet der *ApplicationMaster* über die Ausführung des MapReduce-Jobs. Dazu gehören Enscheidungen wie zum Beispiel eine parallele Ausführung oder die Notwendigkeit zur Initialisierung von weiteren *Containern* (vgl. (Whi12), S. 197f). Diese Entscheidungen können zur weiteren Beschaffung von Resourcen in Abstimmung mit dem *RessourceManager* führen, wobei dabei vom *ApplicationsManager* Informationen für das *Scheduling* des *RessourceManagers* übertragen werden. Dazu gehören zum Beispiel *heartbeats* und Informationen über die Position der Inputsplits auf *Nodes* und in *Racks*. Wenn alle *Container*, die für die Ausführung der Tasks nötig sind, vom *Scheduler* des *RessourceManagers* zugeteilt wurden, bestimmt der *ApplicationMaster* das Outputverzeichnis und startet die *Container* (vgl. (Whi12), S. 198f). Für das Starten der *Container* kontaktiert der *ApplicationMaster* die *NodeManager*, die dann ein *YarnChild* starten. Das *YarnChild* läuft als eigene Java Virtual Machine und lokalisiert als erstes alle benötigten Resourcen wie zum

Beispiel die JAR-Datei und Job-Konfiguration aus dem *HDFS*, die dem jeweils bestimmten Task zugeordnet sind. Anschließend führt das *YarnChild* den jeweiligen Task aus.

4.2.3 Apache Hive

Apache Hive ist ein in den Anfängen von Facebook entwickeltes Framework, das die Datenabfrage aus einem HDFS-Cluster mit einem SQL-Dialekt, der sogenannten *Hive Query Language* (HiveQL) ermöglicht (vgl. (CWR12), S.1). Apache Hive ist mittlerweile ein eigenes Projekt der Apache Software Foundation und kann deshalb nicht mehr als Hadoop-Komponente bezeichnet werden. Jedoch ist Apache Hive für die Verwendung im HDFS entwickelt und kann deshalb als Erweiterung von Apache Hadoop gewertet werden. Apache Hive ist für Data-Warehouse Anwendungen mit sehr großen Datenmengen entwickelt, es erlaubt auf Zeilenebene ACID Transaktionen (Siehe (Gat15)). Die HiveQL Queries werden durch das Framework intern in MapReduce-Jobs konvertiert und dann über YARN im Cluster ausgeführt. Für die Benutzung von Apache Hive ist im Gegensatz zu MapReduce keine Kenntnis der Programmiersprache Java nötig. Apache Hive erweitert daher die Anzahl der möglichen Benutzer von Apache Hadoop um IT-Entwickler mit Kenntnissen in SQL. Dies führt besonders auf Basis der insgesamt weiten Verbreitung von SQL zu einer starken Verbreitung von Apache Hadoop. Apache Hive bietet insgesamt eine ähnliche Funktionaltät wie SQL-Anwendungen, dennoch ist Apache Hive aufgrund der Batch-Verarbeitung nicht für OLTP geeignet und erlaubt keine Echtzeitanfragen (Siehe (Con15)). Die Stärken von Apache Hive sind Skalierbarkeit, die Erweiterbarkeit mit dem Hadoop-Framework, Fehlertoleranz und eine lose Kopplung mit den Daten und deren möglichen Input-Formaten (Siehe (Con15)). Queries in HiveQL werden über die Hive-Shell gestartet. Ähnlich wie Apache Hadoop im Bereich Batch-Processing ist Apache Hive im Bereich SQL nicht für kleine Datenmengen geeignet, die Performance liegt bei solchen Datenmengen deutlich unter der Performance von klassischen relationalen Datenbanken (vgl. (PPR$^+$09), S.12).

4.3 Bewertung von Apache Hadoop

In den folgenden zwei Abschnitten werden die Vor- und Nachteile des Hadoop-Frameworks erläutert. Dabei wird noch einmal auf die im Hadoop-Framework integrierten Komponenten MapReduce und HDFS eingegangen und zusätzlich die Komponente YARN und das Framework Apache Hive mit einbezogen. Obwohl Apache Hive eigentlich ein extern zu bewertendes Framework ist, wird es aufgrund der intern vollzogenen Umwandlung von Hive-Queries in MapReduce-Code in dieser Bewertung ebenfalls im Kontext von Apache Hadoop berücksichtigt.

4.3.1 Vorteile

Apache Hadoop ist in Java geschrieben und kann deshalb auf einer großen Anzahl von verschiedenen Rechnern laufen. Es ist für den Einsatz auf Standardhardware entwickelt und kann deshalb beim Einsatz in Unternehmen zu Kostensenkungen führen. Außerdem ist Hadoop schemafrei und für die Analyse von großen und unstrukturierten Daten geeignet. Zusätzlich kennzeichnet das Hadoop-Framework eine sehr gute Skalierbarkeit und Fehlertoleranz durch das integrierte HDFS. Das Hadoop-Framework ist modular aufgebaut und hat, besonders durch die Integration von YARN, mehr Möglichkeiten der Datenverarbeitung bereitgestellt und die Abhängigkeit vom MapReduce-Modell verringert. Komponenten wie zum Beispiel HDFS können durch YARN unabhängig installiert und mit anderen Frameworks verbunden werden. Es sind neue Datenverabeitungsparadigmen für Dateien im HDFS möglich und damit auch die Integration von weiteren Geschäftsmodellen und Prozessen, die in der Wirtschaft bisher von anderen Systemen verarbeitet wurden. Dazu zählen zum Beispiel Data-Warehouse-Anwendungen mit Apache Hive oder Möglichkeiten zum ETL durch die Verwendung von Apache Pig (vgl. (WSV14), S. 11).

4.3.2 Nachteile

Die Entwicklung und die Performance von Apache Hadoop ist von der Programmiersprache Java abhängig. An dieser Stelle soll keine Grundsatzdiskussion über Java eröffnet werden, jedoch muss man sich bewusst sein, dass die Abhängigkeit zu Java die Entwicklung von Hadoop beeinflussen kann. Die Wiederverwendung von Daten während mehrerer paralleler Arbeitsvorgänge durch das Hadoop-Framework und iterative Programmabläufe sind mit Apache Hadoop nur schwer und vor allem eher unperformant möglich (vgl. (ZCF+10), S. 1). Apache Hadoop ist grundsätzlich nur für große Datenmengen entwickelt und ist deshalb auch nur für Anwendungen mit entsprechenden Datenmengen nützlich. Es gibt keine in Apache Hadoop integrierte Möglichkeit zu Datenbereinigung, diese Aufgaben müssen vom Anwendungsentwickler übernommen werden. Apache Hadoop ist tendenziell für Offline-Analysen entwickelt und bietet deshalb keine wirkliche Integration von Echzeitverarbeitung, die durch Schlagworte wie zum Beispiel Industrie 4.0 immer mehr an Bedeutung gewinnt.

5 Apache Spark

In diesem Kapitel wird das Framework Apache Spark vorgestellt. Am Anfang des Kapitels werden die Anwendungsmöglichkeiten und Ziele von Apache Spark vorgestellt, danach wird der Aufbau des Frameworks beschrieben. Zu dem Aufbau gehören das Programmiermodell mit dem neuartigen Konzept der *RDDs*, die Architektur eines Clusters von Apache Spark und die Komponente Spark SQL. Zum Abschluss des Kapitels wird Apache Spark mit Hilfe von Vor- und Nachteilen bewertet.

5.1 Anwendungsmöglichkeiten und Ziele von Apache Spark

Apache Spark ist ein in der funktionalen Programmiersprache Scala geschriebenes Framework für Datenverarbeitung im Cluster mit APIs in den Programmiersprachen Java, Scala und Python (vgl. (AXL$^+$15), S.2). Es wird für die Analyse von Daten im Kontext Big Data genutzt und zählt zum aktuellen Zeitpunkt zu den am stärksten bearbeiteten Projekten der Apache Software Foundation (vgl. (Apa15a)). Dabei soll Apache Spark das MapReduce-Modell erweitern und neue Verarbeitungmethoden für Daten aus dem Kontext Big Data bereitstellen (vgl. (KKW15), S. 1). Der Schwerpunkt der Entwickler von Apache Spark liegt dabei auf der Geschwindigkeit, wobei die Hauptziele die interaktive Analyse von Dateien und Echtzeitanwendungen sind (vgl. (KKW15), S. 1). Die Daten werden dabei verteilt im Cluster verarbeitet. Im Unterschied zu Apache Hadoop liegt der Fokus von Apache Spark auf der Verarbeitung der Daten im Hauptspeicher des jeweiligen Knotens im Cluster (vgl. (RLOW15), S. 4).

Das Spark-Framework stellt dabei Fehlertoleranz und Skalierbarkeit ähnlich wie Hadoop MapReduce zur Verfügung, es erlaubt aber im Gegensatz zu MapReduce eine effektive Verarbeitung von iterativen Machine-Learning-Algorithmen und Tools für interaktive Datenanalyse und interaktives Data Mining (vgl. (ZCF$^+$10), S. 1). Nach ((ZCDD12), S. 14) ist die Performance von Apache Spark 20-mal so schnell wie die Performance von Apache Hadoop bei iterativen Programmen, außerdem kann Apache Spark eine Menge von mehreren hundert Gigabytes an Daten interaktiv durchsuchen (vgl. (RLOW15), S. 6).

Apache Spark bietet die Bibliotheken Spark Streaming für Echtzeitverabeitung, Spark SQL für strukturierte Daten, MLib für Maschinelles Lernen und GraphX für Graphenverarbeitung

an und kann auf NoSQL Datenbanken wie HBase und Cassandra zugreifen (Siehe (RLOW15), S. 6). In HDFS-Clustern kann Apache Spark mit YARN integriert werden, dabei stellt Apache Spark für diesen Anwendungsfall eine Alternative für die MapReduce-Komponente des Hadoop-Frameworks dar. Anwendungsbeispiele aus der Wirtschaft von Apache Spark sind unter anderem Produktempfehlungen eines Webshops aus dem Themengebiet Data Science (Siehe (RLOW15), S. 5) und Vorhersagen über Verkehrsaufkommen aus dem Themengebiet der Datenverarbeitungprogramme (vgl. (ZCDD12), S. 11). Auch die für Industrie 4.0 benötigte Sensordaten-Verarbeitung ist mit Apache Spark möglich.

5.2 Aufbau von Apache Spark

Im Folgendem ist der Aufbau von Apache Spark anhand der für den Performance-Test in Kapitel 6 relevanten Bestandteile des Spark-Frameworks beschrieben. Der Aufbau ist dabei in die Themen Programmiermodell, Aufbau eines Spark-Clusters und die Komponente Spark SQL aufgeteilt.

5.2.1 Programmiermodell

Der Schwerpunkt des Programmiermodells liegt auf dem Konzept der *RDDs*, diese sind im folgendem Abschnitt zuerst erklärt. Danach sind die möglichen Variablen des Programmiermodells erläutert.

Resilient Distributed Datasets (RDDs)

Die grundlegende Datentruktur im Framework Apache Spark sind *resilient distributed datasets (RDDs)*, jedes *RDD* ist ein Objekt der Programmiersprache Scala (vgl. (ZCF+10), S. 2). *RDDs* sind eine read-only Sammlung von Objekten, die über ein Cluster partitioniert sind und bei Verlust einer Partition neu berechnet werden können (vgl. (ZCF+10), S. 1). Ein *RDD* kann explizit im Hauptspeicher von mehreren Rechnern im Cluster gecached werden und durch mehrere Operationen parallel wiederverwendet werden (vgl. (ZCF+10), S. 1).

Jedes *RDD* hat die in Tabelle 5.1 beschriebenen grundlegenden fünf Eigenschaften von denen zwei optional sind. Zu den nicht optionalen Eigenschaften gehören eine Liste von Partitionen des Datensatzes, eine Liste für Abhängigkeiten (*Dependencies*) zu anderen *RDDs* und eine Funktion für die Berechnung von jeder einzelnen Partition des Datensatzes. Optionale Eigenschaften sind ein *Partitioner* mit Metadaten über die Art der Partitionierung (beispielsweise hash- oder bereichsbasiert) und eine Liste mit Positionen der Daten (Siehe (Apaf)). Das Interface eines *RDDs* bietet grundsätzlich zwei Arten von Operationen. Die erste Art von Operationen

sind *Transformations* wie zum Beispiel *map*, *filter* und *join*, die einen neuen Datensatz aus einem existierenden Datensatz erstellen. Die zweite Art von Operationen sind *Actions* wie zum Beispiel *count*, die nach einer Berechnung an einem Datensatz einen Wert zurückgeben (vgl. (Apa15c), Abschnitt "RDD Operations").

Operation	**Beschreibung**
partitions()	@return eine Liste mit Partitionsobjekten
preferredLocations(p)	Liste von *Nodes*, auf die durch Datenlokalität schneller zugegriffen werden kann
dependencies()	@return Liste von *Dependencies*
iterator(p, parentIters)	Brechnung der Partionen von p mit einem Iterator für die Elternpartitionen
partitioner()	@return Metadaten über die Partitionierung (hash- oder bereichsbasiert)

Tabelle 5.1: Interface eines *RDDs* in Spark
(Quelle: (ZCDD12), S. 6)

Transformations an *RDDs* werden *lazy* berechnet, die Auswertung wird erst bei der Ausführung einer *Action* gestartet. Bei der Anwendung von *Transformations* an *RDDs* wird jedes der *RDD*-Objekte, das während der *Transformations* benutzt wird, in einem gerichteten azyklischen Graphen (in englisch: DAG = directed acyclic Graph) gespeichert. Dieser ist ein logischer DAG aus Operationen und wird als *Lineage* bezeichnet (vgl. (KKW15), S. 146). Der implizit bestimmte *Lineage*-Graph besteht aus Knoten und gerichteten Kanten mit Attributen, die *RDDs* repräsentieren dabei die Knoten des Graphen. Die gerichteten Kanten des Graphen sind Pointer, die die Verbindungen (*Dependencies*) zwischen den *RDDs* darstellen und auf ein oder mehrere Elternknoten zeigen. Die Metadaten der *Dependencies* sind die Attribute der gerichteten Kanten (vgl. (KKW15), S. 146). Es gibt zwei Typen von *Dependencies*, *Narrow Dependencies* und *Wide Dependencies*. Bei *Narrow Dependencies* benutzt jede Partition des Eltern-*RDDs* maximal ein Kind-*RDD*, was in der Datenbanktheorie eine 1:1 Verbindung ist. Im Gegensatz dazu haben Eltern-*RDDs* bei *Wide Dependencies* mehrere Kind-*RDDs*, was als eine 1:n Verbindung in der Datenbanktheorie bezeichnet werden kann (vgl. (ZCDD12), S. 6).

Die logische Repräsentation eines *RDD* ist eine Sammlung von Objekten (Siehe (KKW15), S. 155). Die physiche Ausführung von Operationen an einem *RDD* wird durch eine *Action* gestartet, wobei während der physischen Ausführung das *RDD* in mehrere Partitionen aufgeteilt wird (Siehe (KKW15), S. 155). Die Gesamtmenge der Daten des *RDD* wird dabei auf die Partitionen verteilt, die Operationen werden verteilt über die Datenpartitionen ausgeführt. Wenn zum Beispiel *RDDs* aus Inputdateien aus dem HDFS erstellt werden, haben diese eine Partition für

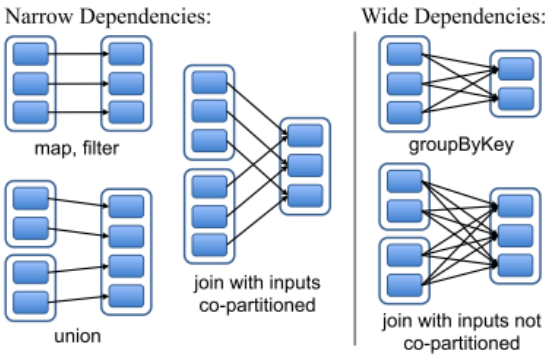


Abbildung 5.1: *Dependencies* in Spark
Jedes Rechteck ist ein *RDD*, die jeweils Blauen Rechtecke innerhalb der *RDDs* sind Partitionen (Quelle: (ZCDD12), S. 7)

jeden *Block* der Datei im HDFS (vgl. (ZCDD12), S. 6). Folgender Spark-Code erstellt zwei *RDDs* (Codebeispiel 5.1). Es werden dabei (URL, Total Count)-Tupel erzeugt, wobei in einer Textdatei mit dem Namen „weblog.txt"nach URLs mit der Top-Level-Domain „.com"gesucht wird.

```
1  val weblog = sc.textFile("/home/tim/Dokumente/Spark/weblog.
       txt")
2  val urlCounts = weblog.flatMap(line => line.split(" ")).
       filter(word => word.contains(".com")).map(url => (url, 1)).
       reduceByKey(_ + _)
```

Code 5.1: Sparkshell

Grundsätzlich werden zwei *RDDs* erstellt, ein *RDD* mit dem Bezeichner `weblog` und ein *RDD* mit dem Bezeichner `urlCounts`. Für diese wird der *Lineage* als DAG mit weiteren *RDD*-Objekten erstellt. Die Methode `toDebugString` zeigt den *Lineage* der *RDDs* an (vgl. (Apaf)). In folgender Ausgabe 5.2 ist der *Lineage* des `weblog`-*RDDs* als Ergebnis des Methodenaufrufes `toDebugString` abgebildet. Die Leserichtung entsprechend der getätigten Eingaben ist dabei von unten nach oben. Die letzte Eingabe wird dabei beim Aufruf einer *Action* auch als letztes berechnet. Es wird im Beispiel 5.2 zuerst ein *HadoppRDD* erstellt (Siehe 5.2, Zeile 4) und dann implizit in der `textFile`-Methode die *map*-Operation ausgeführt, um die Strings in der Datei weblog.txt zu lesen (Siehe 5.2, Zeile 3). Im Anschluss wird das partitionierte *RDD* als *MapPartitonsRDD* mit Strings als Inhalt zurückzugeben. Bis hierhin wurde allerdings nur ein DAG von *RDDs* mit Metadaten über deren Verbindungen erzeugt, es wurde noch keine *Action* durchgeführt.

```
1 scala> weblog.toDebugString
2 res2: String =
3 (2) C:/Spark/SparkData/weblog.txt MapPartitionsRDD[1] at
     textFile at <console>:21 []
4  | C:/Spark/SparkData/weblog.txt HadoopRDD[0] at textFile at <
      console>:21 []
```

Code 5.2: *Lineage* des `weblog`-*RDDs*

Im folgendem Code 5.3 ist der *Lineage* des `urlCounts`-*RDDs* abgebildet. Die ersten beiden Operationen in Zeilen 8 und 7 sind identisch zu Beispiel 5.2, dann werden 3 *MapPartitionsRDDs* durch die Operationen *flatMap*, *filter* und *map* in Zeilen 6 bis 4 erzeugt. Durch die *reduceByKey* wird in Zeile 3 ein *ShuffledRDD* erstellt. Der *Lineage* besteht also insgesamt aus 5 *RDDs*. Die Ausführung einer *Action* wie zum Beispiel `count` oder die Speicherung des *RDDs* in einer Liste mit (URL, TotalCount)-Tupeln führt zu einer rekursiven Berechnung beginnend beim letzten erzeugten *RDD* (im Beispiel 5.3 das *ShuffledRDD* in Zeile 3) entlang des *Lineage* zu allen Eltern-*RDDs* (im Beispiel 5.3 bis zum *HadoopRDD* in Zeile 8).

```
1 scala> urlCounts.toDebugString
2 res1: String =
3 (2) ShuffledRDD[5] at reduceByKey at <console>:23 []
4  +-(2) MapPartitionsRDD[4] at map at <console>:23 []
5     | MapPartitionsRDD[3] at filter at <console>:23 []
6     | MapPartitionsRDD[2] at flatMap at <console>:23 []
7     | C:/Spark/SparkData/weblog.txt MapPartitionsRDD[1] at
        textFile at <console>:21 []
8     | C:/Spark/SparkData/weblog.txt HadoopRDD[0] at textFile at <
        console>:21 []
```

Code 5.3: *Lineage* des `urlCounts`-*RDDs*

RDDs können auf zwei verschiedene Arten persistiert werden. Standardmäßig sind *RDDs* materialisiert, wenn sie in Operationen verwendet werden und werden gelöscht wenn die Operationen im Hauptspeicher beendet sind (vgl. (ZCF+10), S. 2). Die zwei *Actions* mit der Möglichkeit zur Persistierung sind *cache* und *save*. Die Methode `cache()` eignet sich, wenn ein Datensatz mehrfach benutzt werden soll (vgl. (ZCF+10), S. 2). Der Datensatz wird nach der ersten Benutzung nach Möglichkeit im Hauptspeicher gelassen und kann anschließend performant wiederbenutzt werden. Die Methode `save()` speichert Datensätze im lokalen Dateisystem oder im verteilten Dateisystem wie zum Beispiel HDFS (vgl. (ZCF+10), S. 2).

Variablen

Spark bietet zwei Typen von Variablen an, *Broadcast Variables* und *Accumulators*. Die beiden Typen unterscheiden sich in Bezug auf die Serialisierung und in der Art des Zugriffes (vgl. (ZCF+10), S. 4). *Broadcast Variables* sind read-only Variablen, die vorzugsweise im Hauptspeicher gehalten werden und zum Beispiel für die Verteilung von großen Datensätzen unter *Nodes* im Cluster geeignet sind (vgl. (Apa15c), Abschnitt "Broadcast Variables"). Der Wert einer *Broadcast Variable* wird in das verteilte Dateisystem der Spark-Installation gespeichert, dabei ist die serialisierte Form der Variablen der Pfad zu der Datei (Siehe (ZCF+10), S. 4). Wenn der Wert benötigt wird, wird zuerst der lokale Hauptspeicher nach dem Wert abgefragt. Wenn sich der Wert nicht im lokalen Hauptspeicher befindet, wird aus dem Dateisystem gelesen (vgl. (ZCF+10), S. 5).

Accumulators sind write-only Variablen (vgl. (KKW15), S. 102) und werden zum Beispiel für Counter und parallele Summen eingesetzt (vgl. (ZCF+10), S. 3). Sie bekommen bei ihrer Erzeugung eine eindeutige ID, die serialisierte Form ist die ID und ein "Zero"-Wert für den Variablentyp (vgl. (ZCF+10), S. 5). Für jeden Thread, der auf den Worker-*Nodes* läuft, wird eine seperate Kopie der *Accumulator*-Variablen erzeugt. Außerdem werden während der Operationen an der Variablen Updateinformationen veröffentlicht (vgl. (Apa15c), Abschnitt "Accumulators"). *Accumulators* aggregieren Informationen und können durch eine assoziative Operation von Worker-*Nodes* verändert werden (vgl. (KKW15), S. 100).

5.2.2 Architektur eines Spark-Clusters

Apache Spark hat im Cluster Modus eine Master/Slave-Architektur, der Master-*Node* ist im Framework ein Java-Prozess mit dem Namen *Master*, die als Java-Prozesse laufenden Slave-*Nodes* werden *Worker* genannt (vgl. (KKW15), S. 117). Als *Master* kann alternativ auch ein Cluster Manager verwendet werden, dessen Umgebung dann die Rolle von *Master* und *Workern* übernimmt (Siehe (Clo15)). Der *Master* koordiniert die *Worker* und startet auf den *Workern* *Executors*, in denen die Programme ausgeführt werden. Außerdem startet der *Master* einen *Driver*, der RDDs erzeugt und verarbeitet (Siehe (KKW15) S. 118). Ein *Driver* konvertiert Programme des Benutzers für die physische Ausführung in *Tasks* und koordiniert das Scheduling der einzelnen *Tasks*, die in den *Executors* laufen (vgl. (KKW15), S. 119). Sowohl der *Driver* als auch die *Executors* sind eigene Java Prozesse und bilden zusammen eine Spark *Application* (Siehe (KKW15), S. 118).

Der *Driver* ist durch den Aufruf der Spark-Shell implizit erstellt und enthält die main()-Funktion des Benutzercodes (vgl. (KKW15), S. 118). Der *Driver* erhält *Jobs* und erstellt anhand

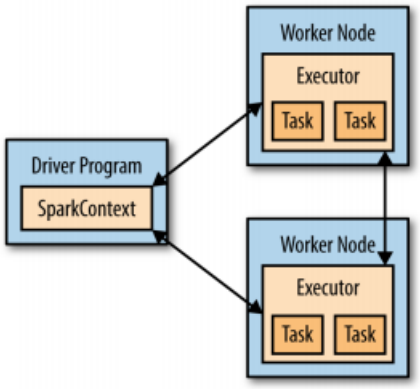


Abbildung 5.2: Master/Slave-Architektur eines Spark Clusters
(Quelle: (KKW15), S. 14)

des *Lineage* einen Ausführungsplan, wobei er einen *Job* in mehrere *Stages* unterteilt (vgl. (RLOW15), S. 227). Eine *Stage* ist eine Menge von *Tasks*, wobei die *Tasks* parallel ausgeführt werden und voneinander abhängen (vgl. (Apab)). Die *Tasks* in einer *Stage* führen alle den gleichen Code aus, wobei jeder *Task* auf einer unterschiedlichen Partition der Daten arbeitet (vgl. (RLOW15), S. 227. Der *Driver* greift auf die Daten im Cluster über ein *SparkContext* Objekt zu (vgl. (KKW15), S. 14). Ein *SparkContext* Objekt repräsentiert die Verbindung zu einem Spark-Cluster und ist der zentrale Zugang für die Funktionalitäten von Apache Spark (vgl. (Apah)). Die Ausführung einer Spark-*Application* wird über einen Cluster Manager koordiniert, für diesen kann der im Spark-Framework eingebaute *Standalone Cluster Manager*, Hadoop YARN oder der verteilte System-Kernel Apache Mesos verwendet werden (Siehe (Apab)). Der Cluster Manager verteilt die Resourcen und startet *Executors* auf den *Workern* in Abstimmung mit dem *Driver* (vgl. (KKW15), S. 120). Die *Executors* sind für die Ausführung der *Tasks* und die Bereitstellung von Hauptspeicher für die Speicherung der Daten zuständig (vgl. (KKW15), S. 119). Sie registrieren sich beim *Driver* und senden Ergebnisse der *Tasks* an den *Driver* (vgl. (KKW15), S. 119). Eine Datei kann über mehrere *Executors* im Cluster verteilt gespeichert und parallel analysiert werden (vgl. (KKW15), S. 15).

Bei der Verwendung von YARN übernimmt der *RessourceManager* die Rolle des *Masters* und die *NodeManager* die Aufgaben der *Worker* (Siehe (Clo15)).

5.2.3 Spark SQL

Spark SQL ist ein Modul im Apache Spark Framework, das die relationale Verarbeitung mit der funktionalen Orientierung von Apache Spark integriert (Siehe (AXL+15), S.1). Die grundlegen-

de Datenstruktur von Spark SQL sind *Dataframes*, die äquivalent zu Tabellen von relationalen Datenbanksystemen sind. Ein *Dataframe* ist eine verteilte Sammlung von Zeilen mit demselben Schema (vgl. (AXL+15), S.2). Auf *Dataframes* können entweder direkt SQL-Statements über die `sql()`-Methode angewandt werden oder es kann die interne Domain Spezific Language der *Dataframes* benutzt werden, die Operationen wie *select*, *groupBy* und *join* bereitstellt. Spark SQL unterstützt die SQL Standard-Datentypen wie *boolean*, *integer*, *double*, *decimal*, *string*, *date*, *timestamp* und komplexe Datentypen wie zum Beispiel *structs*, *arrays*, *maps* und *unions* (Siehe (AXL+15), S.2). Die Kombination von funktionalen Programmcode mit relationalen Abfragen in Form von Queries kann als eine Erweiterung der Programmlogik im Vergleich zu reinen SQL Systemen oder zu Apache Hive betrachtet werden. Es besteht also die theoretische Möglichkeit, zum Beispiel durch Schleifen im Code eines Spark Programms SQL Queries zu verkürzen. Es kann aus Gründen der Performance eine komplette Änderung der Queries vorgenommen werden, da bestimmte Vorgänge der Queries bereits vorher zum Beispiel durch Schleifen erledigt werden können. Abschließend muss jedoch erwähnt werden, dass weder die Domain Spezific Language, noch die Queries über die `sql()`-Methode eine vollständige Unterstützung für die SQL-Sprachdefinition liefern.

5.3 Bewertung von Apache Spark

Für die Bewertung von Apache Spark sind in den folgenden zwei Abschnitten die Vor- und Nachteile des Spark-Frameworks zusammengefasst. Einige von den genannten Vorteilen sind durch die Entwickler von Apache Spark veröffentlicht und sollten deshalb in weiteren Performance-Tests überprüft werden.

5.3.1 Vorteile

Das Framework Apache Spark erlaubt durch *RDDs* eine ähnliche Skalierbarkeit und Fehlertoleranz wie Hadoop-MapReduce und kann zusätzlich noch zu bearbeitende Datensätze über mehrere parallele Operationen performant wiederverwenden (vgl. (ZCF+10), S. 1). Auf ihrer Website behaupten die Entwickler, dass Programme im Spark Framework bis zu 100-mal schneller im Hauptspeicher und bis zu 10-mal schneller auf HDD-Festplatten laufen als im Hadoop MapReduce Framework (Siehe (Apa15a)). Nach Tests der Entwickler sind iterative Anwendungen 20-mal schneller, die Erstellung von analytischen Echtzeit Reports sind 40-mal schneller und ein 1 Terrabyte Datenset kann mit einer Latenz von 5-7 Sekunden interaktiv durchsucht werden (Siehe (ZCDD12), S. 2). In 2014 hat Apache Spark 100 Terrabyte in 23 Minuten sortiert und den Daytona GraySort Contest gewonnen. Apache Spark war in diesem

Fall 3-mal schneller als Hadoop-MapReduce und hat dafür 10-mal weniger Rechner benötigt (Siehe (Xin14)). Die Wiederverwendung von Zwischenergebnissen ist besonders für interaktive Machine Learning- und Graphalgorithmen wie zum Beispiel PageRank, K-means Clustering, logistische Regression und interaktives Data Mining geeignet (vgl. (ZCDD12), S. 1).

5.3.2 Nachteile

Das Prinzip des *Lineage* mit der Wiederherstellung von Daten im Fehlerfall kann bei langen *Lineage*-Ketten sehr aufwändig sein, da das *RDD* in diesem Fall neu berechnet werden muss (vgl. (ZCDD12), S. 8). Es gibt die Möglichkeit zur Erstellung von *Checkpoints*, jedoch müssen diese vom Programmierer explizit an die Daten angepasst und implementiert werden. *RDDs* sind read-only Dateien, was zu Einschränkungen für einige Anwendungsfälle führt und zusätzlich einen hohen Aufwand bei vielen Änderungsoperationen mit sich bringt. Wenn eine Änderung gemacht wird, muss dafür ein neues Objekt in Form eines *RDDs* erstellt werden. Besonders bei der Verwendung von HDFS in Kombination mit einem einem hohen Replikationsfaktor können schreibintensive Anwendungen Probleme mit der Performance in Bezug auf die Geschwindigkeit bekommen. Deshalb eignen sich *RDDs* nicht gut zur Speicherung von Daten, auf die feingranulare Updates ausgeführt werden sollen. Dafür werden Systeme mit Update-Logging und Checkpointing wie zum Beispiel traditionelle Datenbanksysteme benötigt (vgl. (ZCDD12), S. 4). Nicht für *RDDs* geeignete Anwendungen sind damit Speichersysteme von Web-Applikationen und inkrementelle Web-Crawler (vgl. (ZCDD12), S. 4).

Spark SQL liefert keine vollständige Unterstützung für den Sprachkern von SQL (vgl. (Apa15d)). Es ist also davon auszugehen, dass bei einer Migration zu Spark SQL in den meisten Fällen Anpassungen und Änderungen an vorhandenen SQL Queries gemacht werden müssen.

6 Performance-Test

In diesem Kapitel wird die Performance der beiden Framworks Apache Hadoop und Apache Spark in Form eines Benchmarks getestet. Da es aktuell noch keinen Standard-Benchmark für den Test von Frameworks mit Fokus auf Big Data gibt, wurde ein eigener Benchmark entwickelt. Im Folgendem wird zuerst wird der Aufbau des Benchmarks und die herausgekommenen Ergebnisse präsentiert. Am Ende des Kapitels wird auf die Korrektheit des Benchmarks eingegangen und eine abschließende Ergebnisbesprechung des gesamten Performance-Tests vorgestellt.

6.1 Aufbau des Benchmarks

Für den Aufbau des Benchmarks sind zuerst die Abschnitte Cluster, Daten und Hypothesen angegeben. Darin sind der Hardware-technische Aufbau des Cluster, die benutzten Daten und die zu analysierenden Hypothesen beschrieben. Im Anschluss sind die für den Bechmark durchgeführten Operationen genannt, danach wird das Ausführungsmodell des Benchmarks erklärt.

6.1.1 Cluster

Der Benchmark wird in einem HDFS-Cluster aus 5 virtuellen Maschinen ausgeführt. Alle virtuellen Maschinen laufen mit dem Betriebssystem Ubuntu 14.04, es sind Apache Hadoop Version 2.7.0 mit Java 1.8 und Apache Spark in der Version 1.4.0 installiert. Für die Analyse der System-Ressourcen im Cluster ist Ganglia als System-Monitor in der Version 3.6.0 installiert. YARN ist als *RessourceManager* konfiguriert. Ein Rechner ist der Master-Knoten, die 4 restlichen Rechner sind Slave-Knoten. Die virtuellen Machinen haben folgende Hardware-Konfiguration:

- Master-Knoten mit 8 Gigabyte RAM, 2 Prozessorkernen und 500 Gigabyte Festplattenspeicher
- 4 Slave-Knoten mit jeweils 8 Gigabyte Ram, 1 Prozessorkern und 500 Gigabyte Festplattenspeicher

6.1.2 Daten

Die Daten für den Benchmark sind n-Gramme aus dem Google N-Gram Corpus (verfügbar unter (Goo12)). Die n-Gramme sind zeilenweise in Textdateien oder als CSV-Dateien gespeichert, die Dateigroße der einzelnen Dateien variiert zwischen weniger als 1 Megabyte und mehreren Gigabyte. Die Daten sind in einem der folgenden zwei Formate gespeichert (vgl. (Goo12)):

1. ngram TAB year TAB match_count TAB volume_count NEWLINE
   Zum Beispiel das 1-Gramm: circumvallate 1978 335 91
2. ngram TAB year TAB match_count TAB page_count TAB volume_count NEWLINE
   Zum Beispiel das 5-Gramm: analysis is often described as 1991 1 1 1

Begründung für die Wahl des Datensets

Der Google Books N-Gramm Korpus wurde als Test-Datenset ausgewähl, da dieser frei verfügbar ist und dem in Kapitel 1.1 vorgestellten zweiten Ansatz zur Definition von Big Data zugeordnet werden kann. Der Google Books N-Gramm Korpus ist zwar mit einer Datenmenge von 2,2 Terrabyte um mehr als ein 10.000 faches kleiner als beispielsweise der praktische Anwendungsfall des CERN-CMS-Systems, jedoch ist auch das Cluster für die Performance-Analyse verhältnismäßig kleiner. Die Daten sind in unterschiedlichen Formaten (Variety) vorhanden und werden auf Geschwindigkeit (Velocity) getestet, die zur Verarbeitung gerbraucht wird. Deshalb ist auch eine anschließende Performance-Analyse angemessen, um Rückschlüsse auf mögliche Anwendungsfälle ziehen zu können.

6.1.3 Hypothesen

Der Benchmark umfasst fünf Hypothesen, um Apache Spark mit Apache Hadoop in Bezug auf die Performance zu vergleichen. Die fünf Hypothesen sind:

1. Apache Spark ist schneller als Apache Hadoop, wenn die Daten in den Hauptspeicher passen.
2. Apache Spark ist sogar schneller als Apache Hadoop, wenn die Daten nicht in den Hauptspeicher passen.
3. Apache Spark und Apache Hadoop brauchen für die x-fache Datenmenge eine x-mal so lange Zeit für die Verarbeitung.
4. Apache Spark und Apache Hadoop sind schneller, wenn der Replikationsgrad erhöht wird.

5. Apache Spark und Apache Hadoop sind linear schneller, wenn das Cluster horizontal skaliert wird .

Begründung für die Wahl der Hypothesen

Auf der Website von Apache Spark (Apa15a) wird behauptet, dass Programme im Spark-Framework durch die effektivere Ausnutzung des Hauptspeichers eine bis zu 100-mal kürzere Laufzeit als mit dem Hadoop-Framework haben. In den Tests der 1. Hypothese müsste Apache Spark deshalb in jedem Fall schnellere Ausführungszeiten der Programme als Apache Hadoop erreichen. Jedoch ist es zumindest fraglich, ob dabei der Faktor 100 wirklich erreicht werden kann.

Die Entwickler von Apache Spark behaupten ebenfalls, dass Apache Spark selbst auf externen Speicher (HDD und SSD) 10-mal schnellere Laufzeiten als Apache Hadoop erreicht. Deshalb umfassen die Tests der 2. Hypothese Datenmengen, die nicht zu derselben Zeit in den Hauptspeicher passen.

In der Performance-Analyse in (LRI+14) führt eine Verdopplung der Inputdaten-Größe beim Spark-Framework zu einer ungefähr 2-mal so langen Laufzeit der Anwendung (Siehe (LRI+14), S.6). Deshalb wird mit der 3. Hypothese getestet, ob dieser Faktor auch bei einer deutlichen Erhöhung immer noch gilt.

Nach (Had14a) führt ein Replikationsfaktor mit einem größeren Wert als 1 bei Programmen im Hadoop-Framework zu einer Reduzierung der Lesezugriffe. Damit müsste sich die Geschwindigkeit der Ausführung von Programmen mit vielen Lesezugriffen im Hadoop-Framework durch die Erhöhung des Replikationsfaktors verringern. Es ist also unter Annahme der 4. Hypothese zu analysieren, ob Apache Spark ein ähnliches Verhalten wie Apache Hadoop aufweist.

In Bezug auf die horizontale Skalierbarkeit ist der Anspruch an verteilte Systeme, dass eine lineare Performance-Steigerung erreicht wird (vgl. (Shv10), S.10). Einige Anwendungen im Benchmark der Enwickler von Apache Spark haben dabei auch eine lineare Skalierung erreicht (Siehe (ZCDD12), S.10). Apache Hadoop und Apache Spark müssten also in diesem Benchmark durch die Hinzunahme von Knoten zum Cluster linear schneller sein.

6.1.4 Operationen

Folgende Operationen sind im Benchmark mit Apache Spark und Apache Hadoop ausgeführt:

- WordCount
- URLCount

- SumYear
- StartsWith (Buchstabe)
- GroupBy
- Join

Der Code für die Operationen befindet sich auf der CD, die in der ausgedruckten Variante dieser Bachelorabeit enthalten ist. Die Operationen WordCount und URLCount sind bereits aus Kapitel 2 bekannt und sind für Apache Hadoop und für Apache Spark mit den jeweiligen Standardbibliotheken implementiert. Die Operation SumYear berechnet die Summe aller Jahreszahlen der N-Gramme, dafür werden Apache Hive und Spark SQL verwendet. Die Operation StartsWith gibt alle N-Gramme an, die mit einem zu spezifizierenden Buchstaben anfangen. StartsWith, Join und GroupBy sind ebenfalls mit Apache Hive und Spark SQL implementiert, wobei bei der Operation Join mit Apache Spark die Domain Specific Language verwendet wird.

Begründung für die Wahl der Operationen

Die beschriebenen Operationen sind für den Benchmark ausgewählt, da diese allgemeine Anwendungsfälle darstellen und eine gute Vergleichbarkeit der Ergebnisse erwartet wird. WordCount ist der Standard-Anwendungsfall für das MapReduce-Modell, URLCount ist davon eine modifizierte Variante. Deshalb sind diese beiden Operationen für einen Standard-Benchmark geeignet. Die Operationen GroupBy und Join sind die Standard-Anwendungsfälle aus dem Kontext SQL, die Operationen SumYear und StartsWith sind auf Basis der verwendeten Daten interessante Anwendungsfälle.

6.1.5 Ausführungsmodell des Benchmarks

Insgesamt sind im Cluster nach Spezifikation der virtuellen Maschinen 40 Gigabyte an Hauptspeicher verfügbar. Die Angabe von Ganglia unterscheidet sich dabei um 1 Gigabyte. Der Hauptspeicher wird von Ganglia mit insgesamt 39 Gigabyte angegeben, davon sind im Leerlauf insgesamt ca. 11 Gigabyte verbraucht. Jeder Slave-Knoten benötigt dabei im Leerlauf ca. 1,8 Gigabyte Hauptspeicher, der Master-Knoten verbraucht im Leerlauf ca. 4,5 Gigabyte Hauptspeicher. Während der Ausführung von Hadoop MapReduce-Programmen gibt YARN 32 Gigabyte zur Verfügung stehenden Hauptspeicher und 32 Gigabyte genutzten Hauptspeicher an. Es laufen dabei 31 Container und 31 v-Cores. Im Unterschied dazu gibt YARN während der Ausführung von Spark-Programmen 32 Gigabyte zur Verfügung stehenden Hauptspeicher

und 29 Gigabyte genutzten Hauptspeicher an. Es laufen 5 Container und 5 v-Cores. Beim Performance-Test werden die Ausführungszeiten verglichen, die von YARN angegeben werden.

6.2 Benchmark

Der Benchmark wird anhand der 5 Hypothesen mit Daten aus dem Google Books N-Gramm Korpus durchgeführt. Für jede Hypothese werden dabei die Daten genannt, die für den Test der Hypothese verwendet sind. Im Anschluss an diese Beschreibung sind die Ergebnisse der jeweiligen Hypothese vorgestellt. Alle benutzen Daten sind für den Benchmark im HDFS gespeichert und werden von dort aus von den beiden Frameworks verarbeitet.

6.2.1 Hypothese 1

> *„Apache Spark ist schneller als Apache Hadoop, wenn die Daten in den Hauptspeicher passen."*

Für den Test der Hypothese 1 eignet sich ein relativ kleiner Datensatz, da dieser vollständig in den Hauptspeicher passen soll. Der Testdatensatz für die Operationen WordCount, URLCount, SumYear, StartsWith und GroupBy ist 1grams British English, der entpackt eine Größe von 16348265765 Bytes (ca. 16 Gigabyte) im HDFS hat und damit vollständig in den Hauptstpeicher passt. Für die Operation Join werden die Datensätze 1grams Chinese mit einer Größe von 490047773 Bytes (ca. 450 Megabyte) und 1grams English Fiction mit einer Größe von 3579368126 Bytes (ca. 3,3 Gigabyte) verwendet.

Operation	Zeit von Hadoop	Zeit von Spark	Zeitunterschied Spark
WordCount	00:34:38	00:45:43	+00:11:05
URLCount	00:14:15	00:02:16	-00:11:59
SumYear	00:06:38	00:05:08	-00:01:30
StartsWith	00:10:14	00:08:54	-00:01:20
GroupBy	00:10:40	00:07:08	-00:03:32
Join	00:12:05	00:02:47	-00:09:18

Tabelle 6.1: Ergebnisse zu Hypothese 1
(Quelle: Eigene Darstellung)

In Tabelle 6.1 sind die Laufzeiten der Operationen mit der Zeitangabe hh:mm:ss abgebildet. Alle Operationen außer WordCount sind entsprechend der Hypothese 1 von Apache Spark schneller ausgeführt wurden als von Apache Hadoop. Der Grund für die längere Laufzeit der Operation WordCount von Apache Spark liegt in der Ausgabe des Standard-Algorithmus.

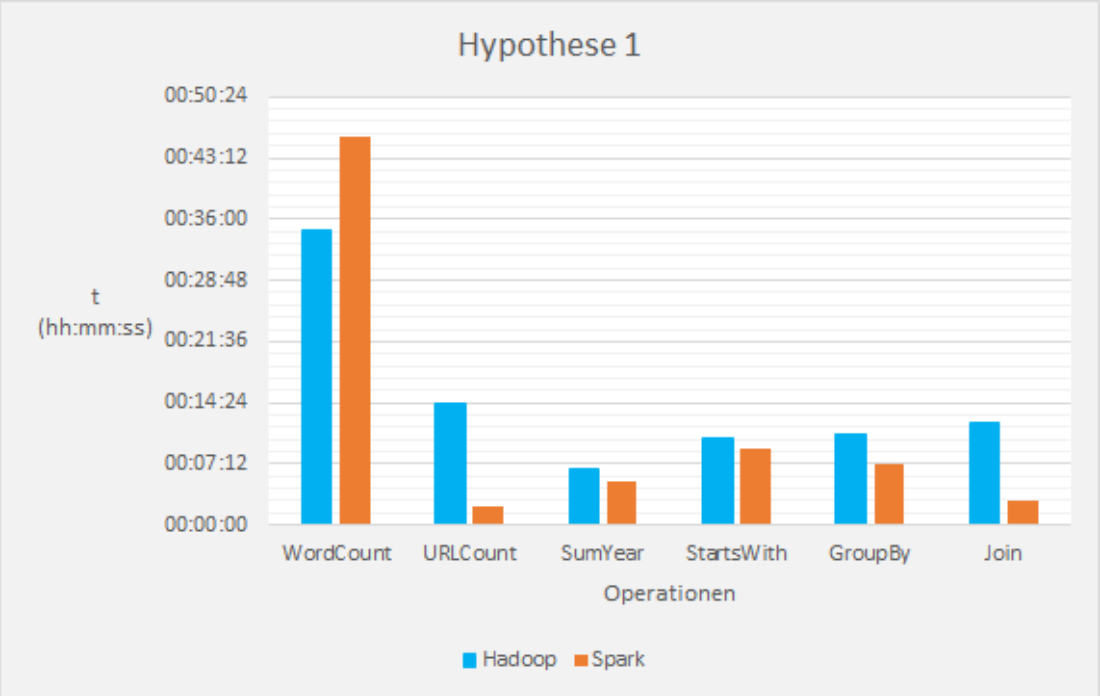
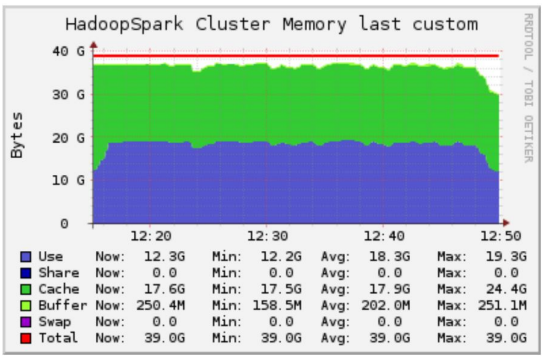


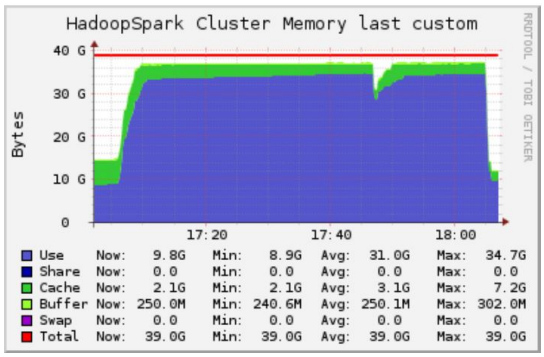
Abbildung 6.1: Ergebnisse der Hypothese 1: Diagramm
(Quelle: Eigene Darstellung)

Der Standard-Algorithmus für einen WordCount mit Apache Spark ist von der Spark-Website (Apa15b) entnommen und speichert 138 Dateien als Ergebnis in das HDFS. Im Unterschied dazu speichert Apache Hadoop nur eine Ergebnisdatei in das HDFS. Für einen direkten Vergleich muss also der Algorithmus für Apache Spark so geändert werden, dass wie mit Apache Hadoop nur eine Ergebnisdatei ins HDFS geschrieben wird. Nach dieser Änderung dauert ein WordCount des 1grams British English Datensets mit Apache Spark 8 Minuten und 9 Sekunden. In den folgenden Tests wird nur noch der optimierte WordCount-Algorithmus verwendet. In Abbildung 6.2 ist die Hauptspeicherauslastung von Apache Hadoop im Vergleich zu Apache Spark während der Operation WordCount in 6.2a und 6.2b abgebildet. Abbildung 6.2c zeigt die Hauptspeicherauslastung für den optimierten WordCount-Algorithmus.

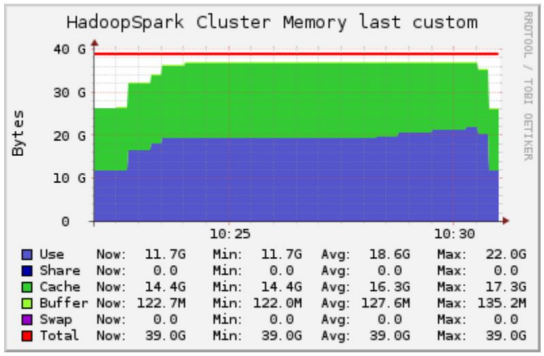
Unerwarteterweise ist die Operation mit der höchsten Hauptspeicherauslastung von der gemessenen Zeit her die langsamste Operation. Der nicht optimierte WordCount-Algorithmus nutzt zwar wie in Abbildung 6.2b abgebildet am meisten Hauptspeicher, ist aber 37 Minuten und 34 Sekunden langsamer als der optimierte WordCount-Algorithmus. Dies liegt vermutlich daran, dass die Daten vor der Speicherung im HDFS in den Hauptspeicher geladen und im Anschluss von dort aus in das HDFS gespeichert werden. In diesen Fall werden wie bereits



(a) Apache Hadoop



(b) Apache Spark nicht optimierter WordCount

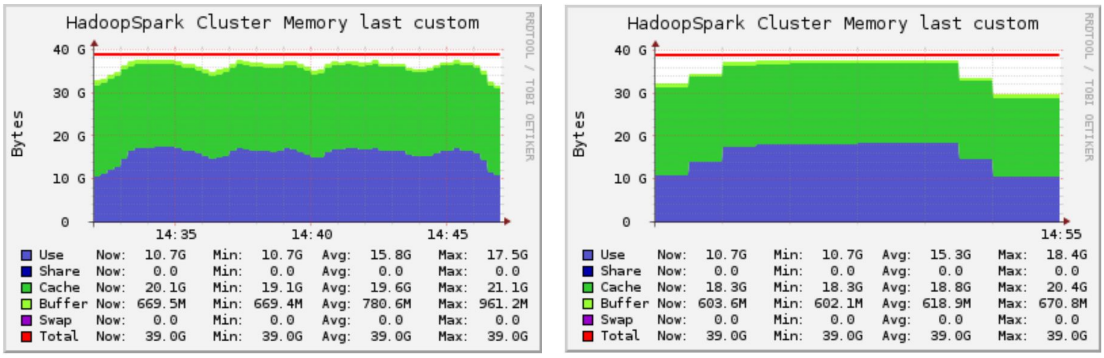


(c) Apache Spark optimierter WordCount

Abbildung 6.2: Auslastung des Hauptspeichers von Apache Hadoop und Apache Spark während der Operation WordCount
(Quelle: Vom Ganglia-Monitor erstellte Clustermetriken)

beschrieben 138 Ergebnis-Dateien in das HDFS geschrieben. Auch in Abbildung 6.2c lässt sich im späteren Verlauf ein Anstieg der Hauptspeicherauslatung erkennen. An dieser Stelle wird die Ergebnisdatei vermutlich komplett in den Hauptspeicher geladen und dann in das HDFS gespeichert.

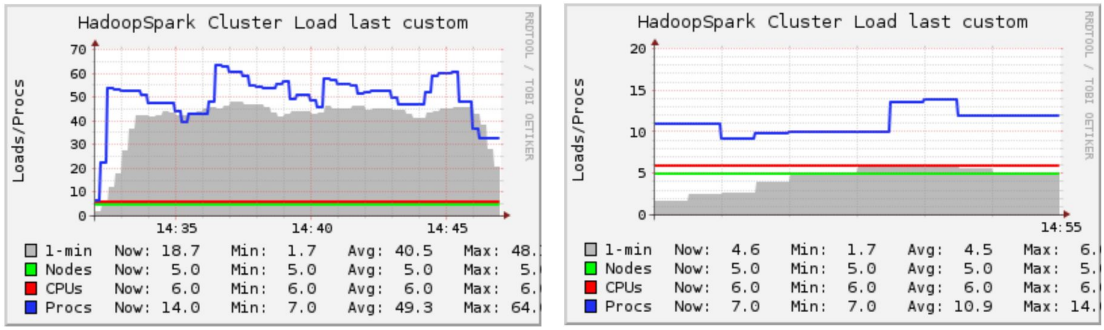
Der zweitgrößte Zeitunterschied ist bei der Operation URLCount vorhanden, die dabei gemessene Hauptspeicherauslastung ist in Abbildung 6.3 abgebildet. Die Hauptspeicherauslastung von Apache Spark ist im Vergleich zu Apache Hadoop trotz des großen Zeitunterschiedes nicht so unterschiedlich wie erwartet. Demnach ist die deutlich schnellere Performance von Apache Spark nicht primär auf die erhöhte Nutzung des Hauptspeichers zurückzuführen. Besonders auffällig ist dabei die durchschnittliche Anzahl laufender Prozesse im Cluster, welche in Abbildung 6.4 abgebildet ist. Apache Hadoop hat während der Ausführung der Operation URLCount im Durchschnitt fast 5-mal so viele Prozesse wie Apache Spark laufen. Diese im Vergleich große Anzahl von durchschnittlich 49,3 Prozessen ist anscheinend langsamer und



(a) Apache Hadoop (b) Apache Spark

Abbildung 6.3: Auslastung des Hauptspeichers von Apache Hadoop und Apache Spark während der Operation URLCount
(Quelle: Vom Ganglia-Monitor erstellte Clustermetriken)

weniger effektiv als die kleinere Anzahl von durchschnittlich 10,9 Prozessen von Apache Spark. Bei der SumYear Operation ist der Zeitunterschied von 1 Minute und 30 Sekunden von Apache Spark und Apache Hadoop relativ gering, sowohl die Hauptspeicherauslastung als auch die CPU-Auslastung sind mit Rücksicht auf Rundungsfehler etc. fast übereinstimmend. Jedoch hat Apache Spark eine geringere Netzwerkauslastung und eine deutlich geringere Anzahl an durchschnittlich laufenden Prozessen. Die laufenden Prozesse scheinen jedoch nicht so effektiv zu sein wie zum Beispiel bei der URLCount Operation.



(a) Apache Hadoop (b) Apache Spark

Abbildung 6.4: Durchschnittliche Anzahl laufender Prozesse von Apache Hadoop und Apache Spark während der Operation URLCount
(Quelle: Vom Ganglia-Monitor erstellte Clustermetriken)

6.2.2 Hypothese 2

> *„Apache Spark ist sogar schneller als Apache Hadoop, wenn die Daten nicht in den Hauptspeicher passen."*

Für die Operationen WordCount, URLCount, SumYear, StartsWith und GroupBy werden alle englischen 1-Gramme verwendet. Diese haben entpackt im HDFS eine Größe von 73726450693 Bytes (ca. 69 Gigabyte). Bei der Operation Join werden die Datensätze 1grams British English mit einer Größe von 16348265765 Bytes (ca. 16 Gigabyte) und 1grams English mit einer Größe von 29235186521 Byte (ca. 27 Gigabyte) verwendet. Für alle Operationen sind damit die Datensätze jeweils zu groß, um zu derselben Zeit im Hauptspeicher gehalten zu werden.

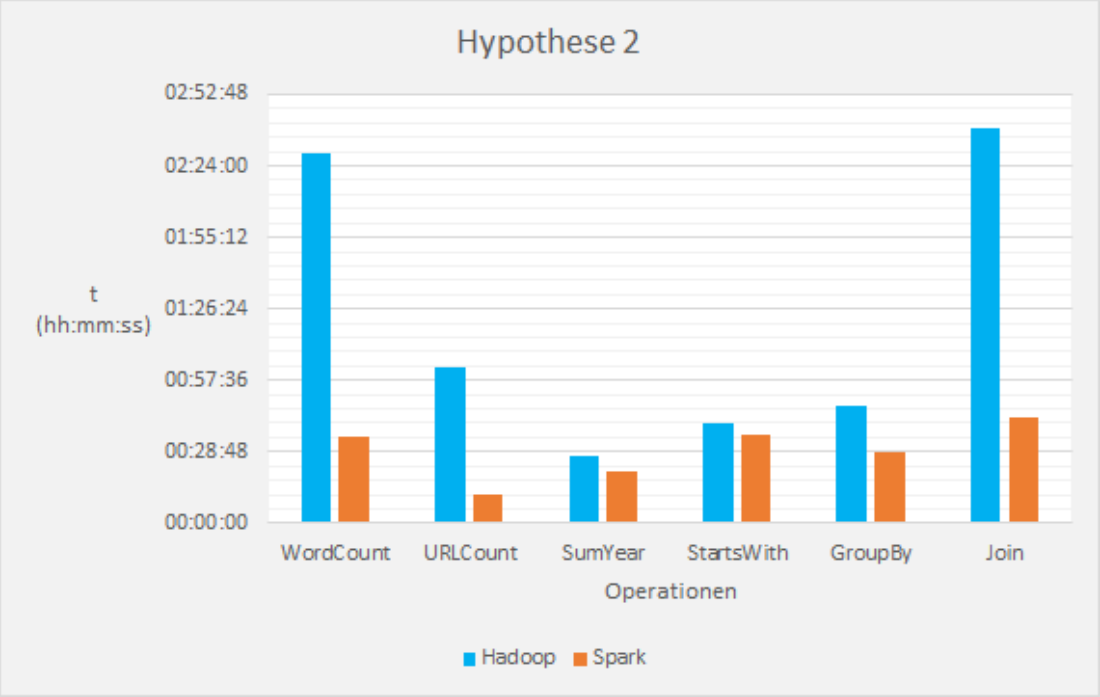


Abbildung 6.5: Ergebnisse der Hypothese 2: Diagramm
(Quelle: Eigene Darstellung)

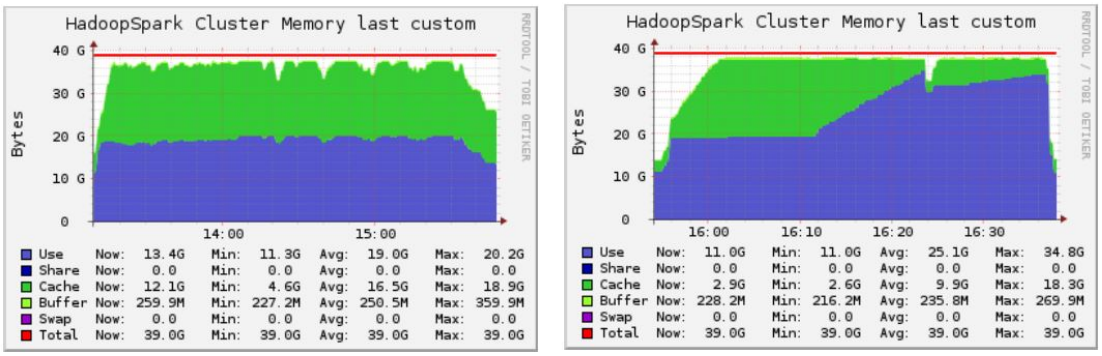
Wie zu erwarten, ist Apache Spark in der Ausführung von allen Operationen schneller als Apache Hadoop. Die größten Zeitunterschiede gibt es bei den Operationen WordCount, URLCount und Join. Bei der Operation Join gibt es deutliche Unterschiede zwischen Apache Hadoop und Apache Spark in Bezug auf die Auslastung des Hauptspeichers. Im direkten Vergleich in Abbildung 6.6 nutzt Apache Spark im Durchschnitt ca. 6 Gigabyte mehr Hauptspeicher als Apache Hadoop. Ähnlich wie in Hypothese 1 laufen bei der Ausführung der Operation

Operation	Zeit von Hadoop	Zeit von Spark	Zeitunterschied Spark
WordCount	02:28:54	00:35:01	-01:53:53
URLCount	01:02:46	00:11:54	-00:50:52
SumYear	00:26:57	00:20:39	-00:06:18
StartsWith	00:40:02	00:35:38	-00:04:24
GroupBy	00:47:02	00:28:33	-00:18:29
Join	02:39:03	00:42:23	-01:56:40

Tabelle 6.2: Ergebnisse zu Hypothese 2
(Quelle: Eigene Darstellung)

Join mit Apache Hadoop fast 4-mal so viele Prozesse wie bei Apache Spark, alle anderen Clustermetriken unterscheiden sich nur im geringen Maße und sollten deshalb keinen großen Einfluss auf die Geschwindigkeit der Ausführung haben.

Bei den Operationen SumYear und StartsWith sind die Zeitunterschiede zwischen Apache Hadoop und Apache Spark im Verhältnis zu den anderen Operationen deutlich geringer. Auch die Messung der Clustermetriken zeigt außer bei der durschnittlichen Anzahl laufender Prozesse keine deutlichen Unterschiede, wobei die Werte der durschnittlichen Anzahl laufender Prozesse mit den Werten der Join Operation vergleichbar sind. Im Durchschnitt nutzt Apache Spark bei der Operation SumYear fast 2 Gigabyte Hauptspeicher mehr als Apache Hadoop. Anscheinend führt dies zu einer besseren Geschwindigkeit bei der Ausführung der Operation.



(a) Apache Hadoop (b) Apache Spark

Abbildung 6.6: Auslastung des Hauptspeichers von Apache Hadoop und Apache Spark während der Operation Join
(Quelle: Vom Ganglia-Monitor erstellte Clustermetriken)

6.2.3 Hypothese 3

> *„Apache Spark und Apache Hadoop brauchen für die x-fache Datenmenge eine x-mal so lange Zeit für die Verarbeitung."*

Für die Operationen WordCount, URLCount, SumYear, StartsWith und GroupBy werden alle English Fiction 2-Gramme verwendet. Diese haben entpackt im HDFS eine Größe von 158816679510 Bytes (ca. 147 Gigabyte) und entsprechen damit ungefähr der 9-fachen Datenmenge des Datensatzes 1grams British English aus Hypothese 1. Für die Operation Join werden die Datensätze 1grams Chinese mit einer Größe von 490047773 Bytes (ca. 450 Megabyte) und 1grams English mit einer Größe von 29235186521 Byte (ca. 27 Gigabyte) verwendet. Die Verarbeitung der Operationen sollte nach Hypothese 3 ungefähr 9-mal länger dauern als bei den Tests aus Hypothese 1. Für Apache Spark wird ausschließlich der optimierte WordCount-Algorithmus verwendet.

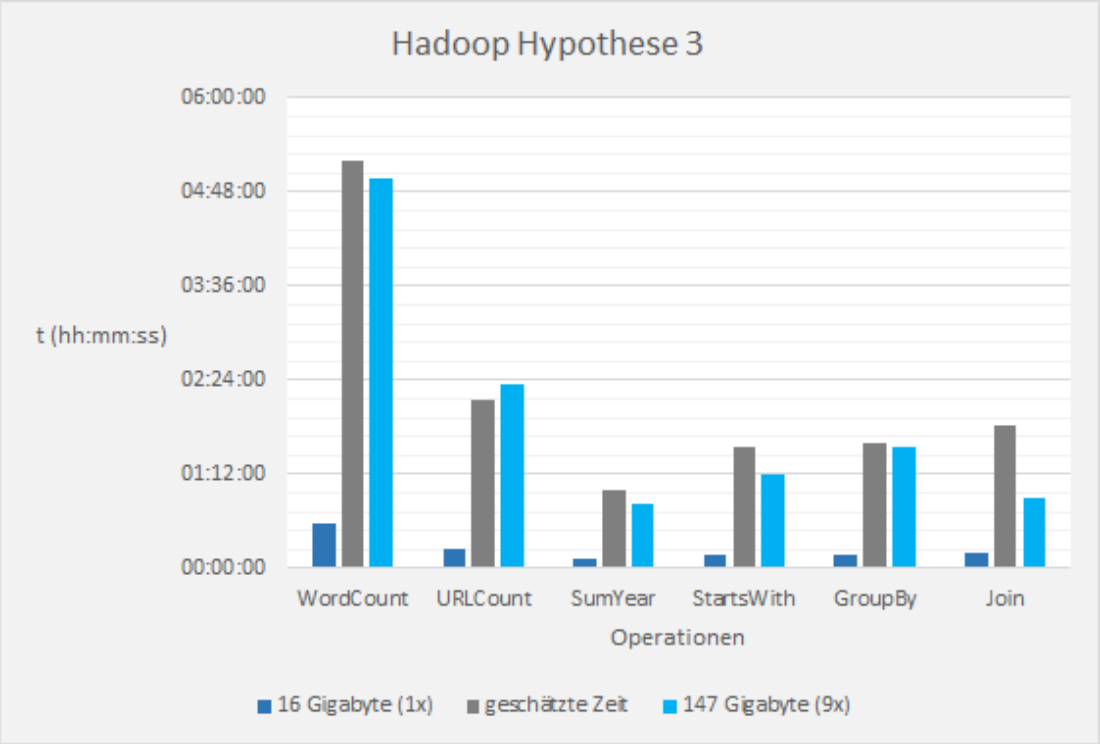


Abbildung 6.7: Ergebnisse der Hypothese 3 von Apache Hadoop: Diagramm (Quelle: Eigene Darstellung)

In Tabelle 6.3 sind die durch Hypothese 3 geschätzten Zeiten und die tatsächlich gemessenen Zeiten von Apache Hadoop abgebildet. Die Operation GroupBy hat die beste Zeitabschät-

Operation	Zeit von Hadoop	geschätzte Zeit	Zeitunterschied
WordCount	04:58:03	05:11:42	-00:13:39
URLCount	02:21:02	02:08:15	+00:12:47
SumYear	00:48:47	00:59:42	-00:11:05
StartsWith	01:11:30	01:32:06	-00:20:36
GroupBy	01:33:05	01:36:00	-00:02:55
Join	00:53:53	01:48:45	-00:54:52

Tabelle 6.3: Ergebnisse zu Hypothese 3 von Hadoop
(Quelle: Eigene Darstellung)

zung als Ergebnis, Apache Hadoop war 2 Minuten und 55 Sekunden schneller als der vorher geschätzte Wert. Apache Hadoop ist bei allen Operationen außer URLCount um ca. 11-38 Minuten schneller als die geschätzten Werte. Die größten Abweichungen von den geschätzten Werten sind bei den Operationen StartsWith und Join aufgetreten. Bei der Operation Join könnte die Abweichung durch die Größe der Datensätze begründet sein. Die Operation Join verwendet 2 Datensätze, wobei im Test der Hypothese 3 nur der 2. Datensatz ungefähr die 9-fache Datenmenge im Vergleich zum 2. Datensatz aus der Hypothese 1 aufweist. Der 1. Datensatz aus Hypothese 3 ist der gleiche Datensatz, der auch in Hypothese 1 verwendet wird. Anscheinend müssen für einen solchen Vergleich andere Größenverhältnisse der Datensätze verwendet werden.

Operation	Zeit von Spark	geschätzte Zeit	Zeitunterschied
WordCount	03:33:15	01:13:21	+02:19:54
URLCount	00:18:55	00:20:24	-00:01:29
SumYear	00:35:17	00:46:12	-00:10:55
StartsWith	00:57:39	01:20:06	-00:22:27
GroupBy	00:49:06	01:04:12	-00:15:06
Join	00:13:08	00:25:03	-00:11:55

Tabelle 6.4: Ergebnisse zu Hypothese 3 von Spark
(Quelle: Eigene Darstellung)

In Tabelle 6.4 sind die durch Hypothese 3 geschätzten Zeiten und die tatsächlich gemessenen Zeiten von Apache Spark abgebildet. Die beste Zeitabschätzung für eine Operation hat sich bei URLCount ergeben, Apache Spark war 1 Minute und 29 Sekunden schneller als der vorher geschätzte Wert. Apache Spark ist bei allen Operationen außer WordCount ca. 11-22 Minuten schneller als die geschätzten Werte.

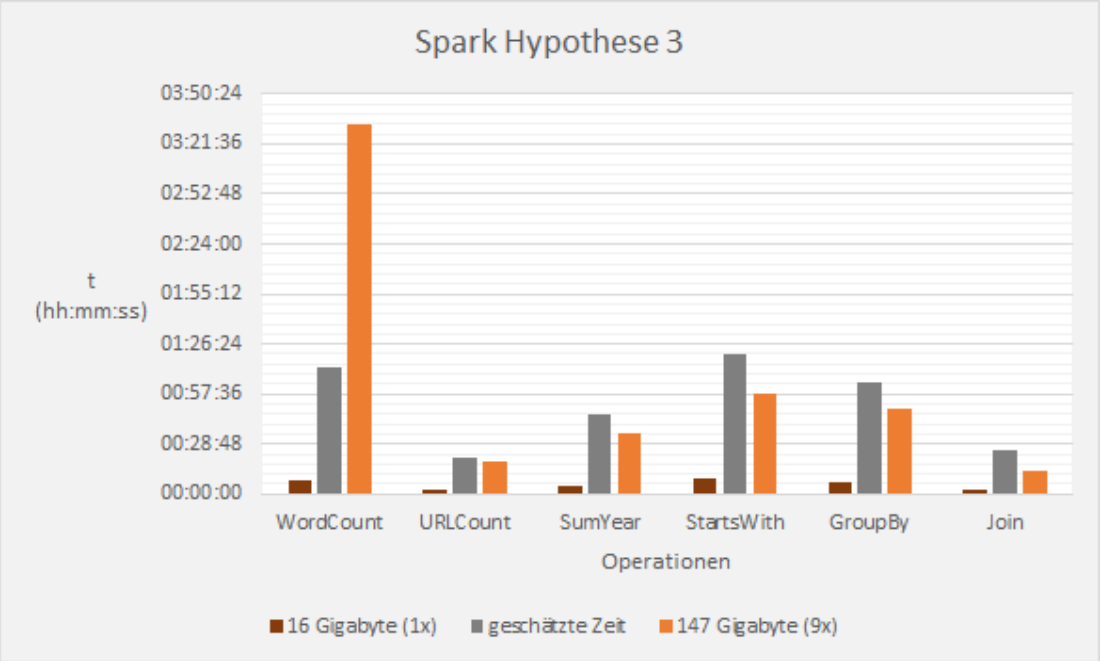


Abbildung 6.8: Ergebnisse der Hypothese 3 von Apache Spark: Diagramm
(Quelle: Vom Ganglia-Monitor erstellte Clustermetriken)

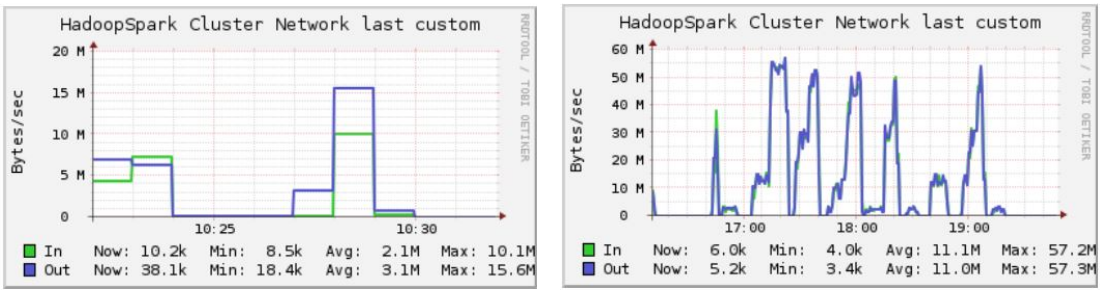
Für die Operation WordCount braucht Apache Spark in diesem Test 2 Stunden, 19 Minuten und 54 Sekunden länger als die vorher geschätzte Zeit. Vermutlich liegt diese deutlich größere Abweichung an dem im Vergleich zu Hypothese 1 stark erhöhten Speicheraufwand und der damit verbundenen erhöhten Belastung des Netzwerks wie in Abbildung 6.9 dargestellt. Die durchschnittliche Belastung des Netzwerks ist bei ca 160 Gigabyte großen Inputdaten ca. 3,5 mal so hoch wie bei ca. 16 Gigabyte großen Inputdaten.

6.2.4 Hypothese 4

> *„Apache Spark und Apache Hadoop sind schneller, wenn der Replikationsgrad erhöht wird."*

Für alle Operationen werden die gleichen Datensätze wie in Hypothese 1 verwendet. Jedoch ist der Replikationsfaktor im HDFS von 1 auf 4 geändert, damit sind die Daten für den Test der Hypothese 4 im HDFS 4-mal repliziert. Nach Hypothese 4 müsste die erhöhte Datenlokalität die Geschwindigkeit der Operationen im Vergleich zu Hypothese 1 erhöhen.

In den Tabellen 6.5 und 6.6 sind die gemessenen Zeiten der 6 Operationen abgebildet, wobei der Replikationsgrad der von den Operationen verwendeten Daten im HDFS auf 4 umgestellt



(a) ca. 16 Gigabyte große Inputdaten

(b) ca. 160 Gigabyte große Inputdaten

Abbildung 6.9: Vergleich der Netzwerkauslastung von Apache Spark während der Operation WordCount
(Quelle: Vom Ganglia-Monitor erstellte Clustermetriken)

Operation	Zeit von Hadoop	Zeitunterschied zu Hypothese 1
WordCount	00:34:10	+00:00:28
URLCount	00:14:51	-00:00:36
SumYear	00:06:16	-00:00:22
StartsWith	00:10:35	+00:00:21
GroupBy	00:11:34	-00:00:54
Join	00:08:53	-00:03:12

Tabelle 6.5: Ergebnisse zu Hypothese 4 von Apache Hadoop
(Quelle: Eigene Darstellung)

ist. Die Zeitunterschiede sind außer bei der Operation Join von Hadoop alle geringer als 60 Sekunden. In Abhängigkeit von der Größe der verwendeten Daten sind solche Schwankungen bei der mehrfachen Ausführung von Operationen oft aufgetreten. Deshalb lässt sich bei diesen Test keine für Apache Spark deutliche Verbesserung der Performance durch die Erhöhung des Repliaktionsgrades feststellen.

Für Apache Hadoop lässt sich eine Verbesserung der Zeit für die Operation Join um 3 Minuten und 12 Sekunden feststellen. Ein möglicher Grund für die Verbesserung der Performance könnte die in Abbildung 6.12 dargestellte verringerte Netzwerklast sein, die während der Operation Join bedingt durch Datenlokalität vorhanden sein müsste. Alle Knoten im Cluster befinden sich während der Tests in demselben *Rack*, die Netzwerkgeschwindigkeit zwischen den *Nodes* ist dementsprechend hoch. Es ist davon auszugehen, dass ein Test mit erhöhtem Replikationsgrad in einem Cluster mit unterschiedlichen *Racks* und damit verbundenen unterschiedlichen Netzwerkleistungen andere Ergebnisse liefern würde.

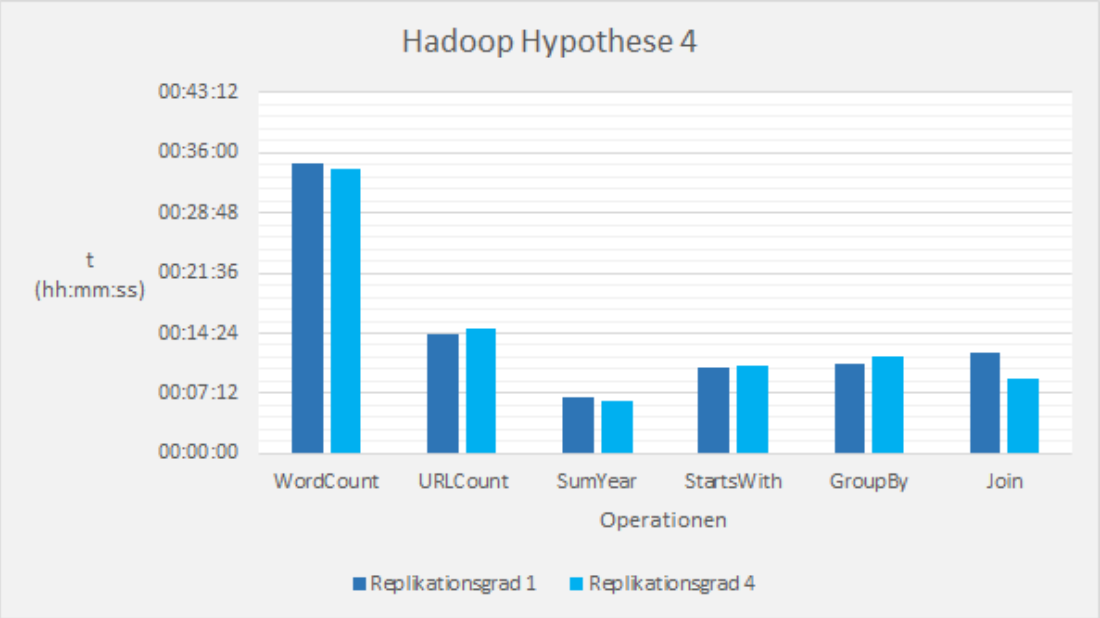


Abbildung 6.10: Ergebnisse der Hypothese 4 von Apache Hadoop: Diagramm (Quelle: Vom Ganglia-Monitor erstellte Clustermetriken)

6.2.5 Hypothese 5

> *„Apache Spark und Apache Hadoop sind linear schneller, wenn das Cluster horizontal skaliert wird."*

Für alle Operationen werden die gleichen Datensätze wie in Hypothese 1 und 4 verwendet, der Replikationsfaktor ist für die Ausführung der Spark-Programme weiterhin 4. Dadurch ist gewährleistet, dass alle Daten auf allen *DataNodes* im Cluster vorhanden sind, auch wenn nicht alle *DataNodes* während der Ausführung einer Operation benutzt werden. Im Gegensatz zu Apache Hadoop kann mit Apache Spark die Anzahl der zu benutzenden *DataNodes* pro auszuführenden Programm durch die Angabe der Anzahl von *Executors* dynamisch bestimmt werden. Dies geht allerdings nur beim Starten eines Programms und nicht während der Laufzeit. Für Apache Hadoop müssen die *DataNodes* jeweils über Properties in Form von XML-Dateien aus dem Cluster entfernt werden. In diesem Test werden die *DataNodes* nacheinander zuerst *decommissioned* und dann beendet, damit diese von Hadoop nicht mehr zur Ausführung von Jobs benutzt werden können. Es muss also im Gegensatz zu Apache Spark direkt das HDFS-Cluster skaliert werden, es gibt keine Möglichkeit zum dynamischen Laden von *DataNodes*. Entsprechend wird für Apache Hadoop bei der Entfernung eines *DataNodes* aus dem Cluster

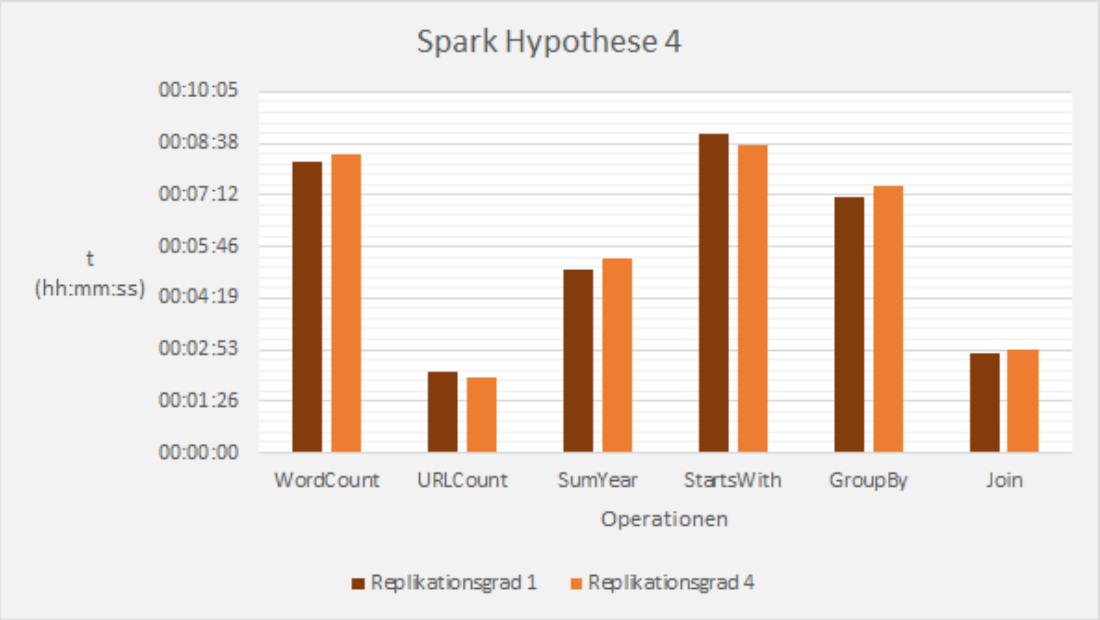


Abbildung 6.11: Ergebnisse der Hypothese 4 von Apache Spark: Diagramm
(Quelle: Vom Ganglia-Monitor erstellte Clustermetriken)

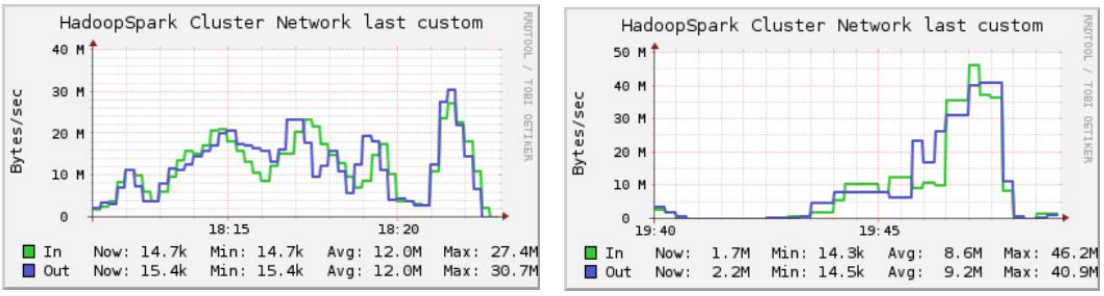
der Replikationsfaktor um 1 reduziert. Das Cluster wird mit jeweils 1-4 vorhandenen *DataNodes* skaliert und es wird die Geschwindigkeit von jeder Operation gemessen.

Die Tabellen 6.7 und 6.8 zeigen zusammenfassend die deutlichen Geschwindigkeitsvorteile von Apache Spark gegenüber Apache Hadoop. Die Operationen WordCount, URLCount und Join können von Apache Spark besonders schnell verarbeitet werden. Bei diesen Operationen genügt ein Spark-Cluster bestehend aus einem Master und einem *Worker*, um ein Hadoop-Cluster mit einem Master und 4 *DataNodes* in Bezug auf die Geschwindigkeit zu schlagen (Siehe Diagramm 6.13). Die Unterschiede der Geschwindigkeiten der Operationen SumYear, StartsWith und GroupBy sind nicht so stark wie bei den oben genannten Operationen, dennoch ist Apache Spark in diesen Tests in einem äquivalenten Cluster immer schneller als Apache Hadoop.

Wie die Abbildung 6.14 exemplarisch an der Operation WordCount zeigt, skalieren die Frameworks Apache Hadoop und Apache Spark nicht linear. In den Diagrammen 6.14a und 6.14b ist die durchgezogene Linie die tatsächlich gemessene Zeit, die gepunktete Linie stellt die Annahme der Hypothese 5 dar. Die Annahme ist eine mit Microsoft Excel mit der Methode der kleinsten Quadrate erstellte Trendlinie, die eine lineare Gerade auf Basis der tatsächlich gemessenen Zeit und der Anzahl *Nodes* darstellt. Außerdem ist das mit Microsoft Excel berechnete Bestimmtheitsmaß für die Trendlinie abgebildet, das als Maß für den linearen Zusammen-

Operation	Zeit von Spark	Zeitunterschied zu Hypothese 1
WordCount	00:08:21	+00:00:12
URLCount	00:02:05	-00:00:11
SumYear	00:05:26	+00:00:18
StartsWith	00:08:37	-00:00:47
GroupBy	00:07:28	+00:00:20
Join	00:02:52	-00:00:05

Tabelle 6.6: Ergebnisse zu Hypothese 4 von Apache Spark
(Quelle: Eigene Darstellung)



(a) Replikationsfaktor 1

(b) Replikationsfaktor 4

Abbildung 6.12: Vergleich der Netzwerkauslastung von Apache Spark während der Operation WordCount
(Quelle: Vom Ganglia-Monitor erstellte Clustermetriken)

hang zwischen der gemessenen Zeit und der Anzahl *Nodes* dient. Der großte Unterschied zwischen Apache Hadoop und Apache Spark in Bezug auf das Bestimmtheitsmaß ist dabei bei der Operation WordCount aufgetreten. Apache Spark skaliert hier fast linear mit einem Bestimmtheitsmaß von ca. 92%. Apache Hadoop hat hingegen ein Bestimmtheitsmaß von ca. 85%. Das durchschnittliche Bestimmtheitsmaß aller Operationen hat für Apache Hadoop einen Wert von ca. 84% und für Apache Spark einen Wert von ca. 87%. Damit skalieren die Frameworks Apache Hadoop und Apache Spark mit allen getesteten Operationen durch das Hinzufügen von *Nodes* zum Cluster im Durchschnitt nur fast linear schneller.

Die mit Microsoft Excel berechneten Trendlinien von Apache Hadoop weisen bei allen getesteten Operationen eine stärkere Steigung als die für Apache Spark erstellten Trendlinien auf. In diesem Performance-Test kann also die Behauptung aufgestellt werden, dass Apache Hadoop effektiver als Apache Spark skaliert. An dieser Stelle wären weitere Tests in Bezug auf die Effektivität der Skalierung möglich.

Operation	4 Nodes	3 Nodes	2 Nodes	1 Node
WordCount	00:34:10	00:43:36	01:07:59	02:18:24
URLCount	00:14:51	00:20:12	00:29:19	01:00:11
SumYear	00:06:16	00:08:34	00:11:37	00:26:31
StartsWith	00:10:35	00:12:26	00:18:14	00:38:35
GroupBy	00:11:34	00:14:33	00:20:39	00:44:44
Join	00:10:14	00:14:58	00:17:28	00:33:55

Tabelle 6.7: Ergebnisse zu Hypothese 5 von Apache Hadoop
(Quelle: Eigene Darstellung)

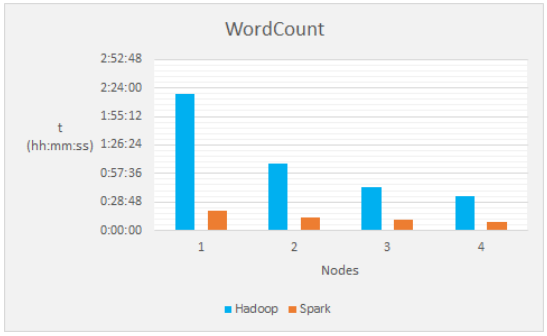
Operation	4 Nodes	3 Nodes	2 Nodes	1 Node
WordCount	00:08:21	00:11:21	00:13:10	00:20:14
URLCount	00:02:05	00:03:49	00:05:17	00:10:40
SumYear	00:05:18	00:06:41	00:09:21	00:17:41
StartsWith	00:08:37	00:11:12	00:16:23	00:32:40
GroupBy	00:07:28	00:09:30	00:13:09	00:27:13
Join	00:02:52	00:03:23	00:04:33	00:08:17

Tabelle 6.8: Ergebnisse zu Hypothese 5 von Apache Spark
(Quelle: Eigene Darstellung)

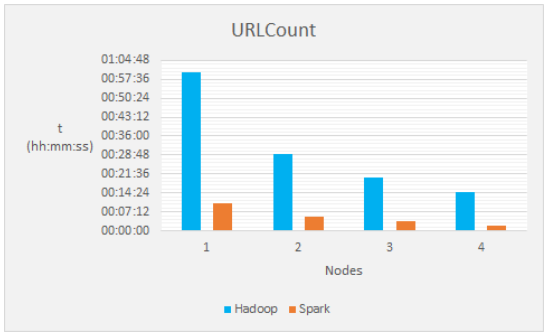
6.3 Korrektheit des Benchmarks

Es ist zu beachten, dass die meisten Operationen für die entsprechende Hypothese nicht mehrmals ausgeführt wurden. Für einen kompletten Benchmark sollten die Operationen für jede Hypothese mehrfach ausgeführt (zum Beispiel 5-10 mal) und dann ein Durchschnittswert von den gemessenen Zeiten gebildet werden. Darauf wird in dieser Bachelorarbeit verzichtet, da die Zeiten in diesem Performance-Test in der richtigen Größenordnung sind. Zufällig wiederholte Ausführungen der Operationen unter gleichen Bedingungen haben keine signifikant unterschiedlichen Zeiten im Vergleich zu den aufgeführten Zeiten ergeben. Ein Beispiel für so eine zufällig wiederholte Ausführung ist in den Ergebnissen zu Hypothese 5 bei dem SumYear-Test für 4 Datanodes vorhanden. Bei diesem Test unterscheidet sich die gemessene Zeit in Hypothese 5 um nur acht Sekunden von dem vorherigen Durchlauf in Hypothese 4.

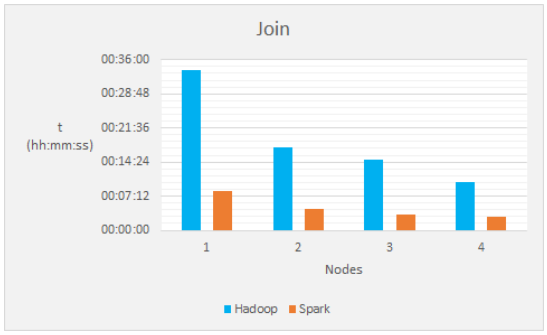
Bei den abgebildeten Diagrammen ist zu beachten, dass die gezeigten Durchschnittswerte nicht immer exakt sein können. Der Grund dafür ist, das die Start- und Endzeiten im Ganglia-Monitors nur minutenweise eingegeben sind, aber die laufenden Programme von Apache Spark und Apache Hadoop sekundengenaue Start- und Endzeiten haben. Im schlechtesten Fall ist es also möglich, dass 59 Sekunden Leerlauf in die Berechnung der Durchschnittswerte



(a) WordCount



(b) URLCount

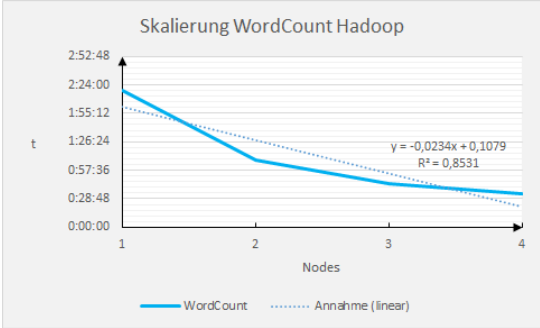


(c) Join

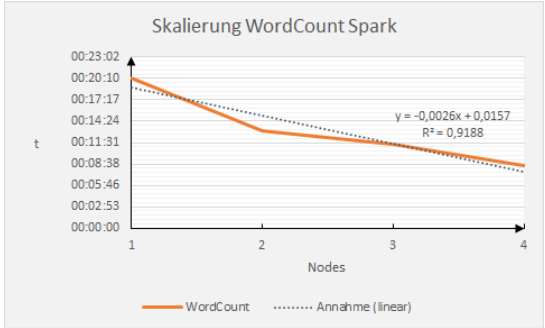
Abbildung 6.13: Vergleich der Ausführungszeiten von Apache Hadoop und Apache Spark bei Skalierung der *Nodes* mit den Operationen WordCount, URLCount und Join (Quelle: Vom Ganglia-Monitor erstellte Clustermetriken)

mit einfließen können. Je länger dabei die Ausführung des Programms dauert, desto weniger wirken sich die Werte aus Leerlaufzeiten auf die Durchschnittswerte aus. Bei sehr kurzen Ausführungszeiten sollte diese Problematik auf jeden Fall in der Analyse beachtet werden.

Die benutzten Operationen sind für einen konkreten Benchmark nicht alle gleich gut geeignet. Bei einigen Tests haben die Ergebnisdateien oder Ergebnisausgaben der Operationen WordCount, StartsWith und GroupBy zwischen Apache Spark und Apache Hadoop Unterschiede aufgewiesen. Der größte Unterschied ist bei den Ergebnisdateien des WordCount-Algorithmus in Hypothese 3 vorgekommen. Apache Hadoop liefert eine ca. 34,5 Megabyte große Ergebnisdatei, Apache Spark hingegen eine 2,36 Gigabyte große Ergbenisdatei. Damit ist dieses Ergebnis für einen Benchmark, der die Geschwindigkeit der Operationen misst, eigentlich nicht mehr verwertbar. Für eine bessere Vergleichbarkeit müssten vermutlich die verwendeten Parsing-Methoden im Code der Programme für Apache Hadoop und Apach Spark weiter angepasst werden.



(a) Apache Hadoop



(b) Apache Spark

Abbildung 6.14: Vergleich der tatsächlichen Skalierung mit der angenommenen linearen Skalierung bei der Operation WordCount
(Quelle: Eigene Darstellung)

Die Operationen URLCount, StartsWith und Join haben bei Stichproben immer dieselben Ergebnisse geliefert. Ein Beispiel dafür ist die beim URLCount durchgeführte Zählung, die die Häufigkeit des Vorkommens einer Website in einer Eingabedatei ausrechnet. Beim stichprobenartigen Vergleich der URLCount Ergebnisdateien haben Apache Spark und Apache Hadoop immer dieselben Zählungen als Ergebnis geliefert. Die Ergbnisdateien sind jedoch in den meisten Fällen unterschiedlich sortiert. Für einen exakteren Benchmark müssten die Algorithmen so optimiert werden, dass beide Ergebnisdateien immer denselben Inhalt in ebenfalls derselben Reihenfolge haben. Der SumYear-Algorithmus hat beim Vergleich der beiden Frameworks bei allen Tests immer dieselbe Ergebnisausgabe geliefert. Damit eignet sich diese Operation besonders gut für einen Benchmark.

6.4 Ergebnisse des Performance-Tests

Wie der Benchmark 6.2 eindeutig zeigt, ist Apache Spark für alle getesteten Operationen 6.1.4 von der Geschwindigkeit der Ausführung die bessere Wahl. Wenn die Ergebnisdateien von Apache Spark eine ähnliche Dateigröße wie die Outputdateien von Apache Hadoop haben, hat Apache Spark deutliche Geschwindigkeitsvorteile. Die Operationen SumYear, StartsWith und GroupBy sind in diesem Benchmark im Code der Spark-Programme über die `sql()`-Methode als komplette SQL-Queries implementiert, es wird also nicht die Domain Spezific Language von Spark SQL benutzt. Die SQL-Queries von Apache Spark stimmen exakt mit den von Apache Hive verwendeten SQL-Queries überein. Bei den Operationen SumYear und StartsWith ist der Zeitunterschied zwischen Hadoop Hive und Spark SQL bei allen Ausführungen am geringsten.

Die Domain Spezific Language wird bei der Operation Join angewendet und wird im Verhältnis von Apache Spark deutlich schneller ausgeführt als die Verwendung von SQL-Queries mit in der `sql()`-Methode. Es empfiehlt sich also aus Gründen der Geschwindigkeit, bei der Verwendung von Spark SQL im Produktivbetrieb eher die Domain Spezific Language als die `sql()`-Methode zu verwenden.

Besonders nützlich ist im praktischen Gebrauch die Möglichkeit zur dynamischen Angabe der Anzahl zu verwendener *Executors* beim Starten eines Apache Spark Programms. Damit kann die größe des Clusters für die Ausführung der Spark-Programme vorweg bestimmt werden, ohne das HDFS Cluster direkt skalieren zu müssen. Hier können Erfahrungswerte über Ausführungszeiten der einzelnen Operationen genutzt werden um, verschiedene Spark-Programme effektiv zur gleichen Zeit im Cluster zu starten.

Der größte Nachteil von Apache Spark ist in diesem Benchmark die Zuverlässigkeit der Spark-*Worker*. Die erhöhte Nutzbarkeit der Ressourcen im Cluster lässt ein sehr genaues Job-Tuning zu. Es müssen dabei allerdings die Ressourcen im Cluster sehr genau berechnet werden. Im Benchmark ist es mehrmals vorgekommen, dass Spark-*Worker* abgestürzt sind und damit die Operation mit weniger Nodes im Cluster ausgeführt wurde. Besonders bei einer hohen Belastung des Clusters führt ein Ausfall eines *Workers* zu enormen Einbußen der Performance. Ein Beispiel dafür ist die Ausführung der Operation WordCount. Die Ausführung hat in Hypothese 3 im Test 6.2.3 mit 4 *Workern* 3 Stunden, 33 Minuten und 15 Sekunden gedauert. Bei einem weitereren Durchlauf des Tests ist anscheinend ein *Worker* während der Ausführung abgestürzt, die Ausführung dauerte dadurch 10 Stunden, 44 Minuten und 10 Sekunden. Der Grund für den Absturz des *Workers* war in diesem Fall vermutlich eine Überlastung des Hauptspeichers oder Probleme mit der Java Virtual Machine, eventuell auch eine Kombination aus beiden genannten Fehlerquellen. Es ist für den produktiven Betrieb also empfehlenswert, eine Art Benachrichtigungssystem für den Ausfall von *Workern* zu implementieren. Die *Container*, die in YARN durch die *NodeManager* verwaltet werden, liefen in den Tests des Benchmarks bei MapReduce- und Hiveanwendungen ohne Ausfälle und damit konstanter als die *Worker* von Apache Spark. Dadurch ist in diesem Benchmark ebenfalls die Zeitabschätzung über die Dauer der Operationen mit Apache Hadoop zuverlässiger als mit Apache Spark.

In dem Benchmark sind weder bei Apache Hadoop noch bei Apache Spark besondere Optimierungen im Code der Progamme implementiert. Apache Hadoop hat bei der Ausführung der Programme in der Regel den Hauptspeicher nicht komplett ausgenutzt, bei weiteren Tests könnten Optimierungen mit einbezogen werden und so vielleicht eine ähnliche Hauptspeicherauslastung wie mit Apache Spark erreicht werden. Es besteht die Möglichkeit, grundsätzliche

Einstellungen über die Hauptspeichernutzung zum Beispiel in den YARN-Properties zu ändern, auch der Einsatz von *Combiner*-Funktionen kann die Hauptspeicherauslastung weiter verbessern (Siehe (DG08), S.6).

Auch für Apache Spark gibt es Optimierungsmöglichkeiten wie zum Beispiel der Einsatz der `cache()`-Methode im Programmcode. Mit dieser Methode können *RDDs* gezielt im Hauptspeicher für die Wiederverwendung persistiert werden. Bei den im Benchmark ausgeführten Operationen konnten allerdings bei der stichprobenartigen Verwendung der `cache()`-Methode keine Verbesserung der Geschwindigkeit festgestellt werden. Vermutlich ist der Einsatz der `cache()`-Methode bei Programmen mit mehreren und tendenziell kleinen Inputdateien oder Operationen mit vielen Iterationen sinnvoll.

Ein interessantes Merkmal von Apache Spark ist die Möglichkeit der Verarbeitung von Daten in relationalen Strukturen wie es ebenfalls Apache Hive anbietet. Der große Unterschied von Apache Spark zu Apache Hive ist jedoch die interne Verabeitung der Programme durch eine Graphenstruktur (DAG) in Form der *RDDs* von Apache Spark. Langfristig besteht für Apache Spark also eventuell die Möglichkeit, effektive Graphenalgorithmen auf relationale Daten anzuwenden. Dies könnte sogar als Vorteil gegenüber traditionellen Datawarehouse-Sytemen genutzt werden, da diese sich tendenziell durch ihr Datenmodell nicht für Graphen eignen.

7 Auswertung

In diesem Kapitel wird der Performance-Test aus Kapitel 6 in Form einer Nutzwertanalyse ausgewertet. Aus der Nutzwertanalyse werden dann die Vor- und Nachteile der Frameworks für die Benutzung in der Praxis zusammengefasst und Annahmen über die Kosten aufgestellt.

7.1 Nutzwertanalyse des Performance-Tests

Die Nutzwertanalyse ist ein Instrument zur Entscheidungsfindung und eignet sich bei Fragestellungen mit einer hohen Anzahl an Bewertungskriterien, wobei diese sowohl quantitativ als auch qualitativ sein können (vgl. (Kï14), S.1f). Für die Nutzwertanalyse wird zuerst das Entscheidungsproblem mit den dazu gehörenden Entscheidungsalternativen genannt. Im Anschluss werden für die Entscheidungsalternativen Entscheidungskriterien gesammelt. Die Entscheidungskriterien werden gewichtet und anschließend bewertet. Aus diesen Bewertungen wird zum Abschluss der Nutzwert der Entscheidungsalternativen berechnet.

7.1.1 Benennung des Entscheidungsproblems und Auswahl der Entscheidungsalternativen

Die Nutzwertanalyse soll das konkrete Entscheidungsproblem lösen, ob Apache Hadoop oder Apache Spark die bessere Wahl für die Verarbeitung von Daten im Kontext Big Data ist. Die Entscheidungsalternativen sind dementsprechend die Frameworks Apache Hadoop und Apache Spark.

7.1.2 Sammlung der Entscheidungskriterien

In dieser Nutzwertanalyse sind verschiedene Bewertungskriterien zur Lösung des Entscheidungsproblems in drei Kriteriengruppen unterteilt. Die drei Kriteriengruppen sind: 1. drei V's (aktuelle Definition von Big Data), 2. verteiltes System (für diesen Zusammenhang wichtige Eigenschaften von verteilten Systemen) und 3. Benutzerfreundlichkeit (des Frameworks). Die 3 Kriteriengruppen bestehen jeweils aus einzelnen Kriterien, die in Tabelle 7.1 abgebildet und in den folgenden drei Abschnitten erklärt sind.

Kriteriengruppe	Kriterium
drei V's	Geschwindigkeit
	Datenmenge
	verschiedenartige Daten
verteiltes System	Skalierbarkeit
	Fehlertoleranz
Benutzerfreundlichkeit	Zuverlässigkeit
	Konfigurationsaufwand

Tabelle 7.1: Auflistung der Kriteriengruppen mit Bewertungskriterien (Quelle: Eigene Darstellung)

Die bereits in Kapitel 1.1 beschriebenen Eigenschaften von Big Data mit der Bezeichnung drei V's sind in der Nutzwertanalyse als Geschwindigkeit, Datenmenge und verschiedenartige Daten bezeichnet. Die Geschwindigkeit wird in der Definition von Big Data als *Velocity* bezeichnet, in dieser Nutzwertanalyse kann dabei allerdings nur die Verarbeitungsgeschwindigkeit bewertet werden. Die zur Definition von *Velocity* ebenfalls gehörende Geschwindigkeit der Datenerzeugung ist nicht Teil des Benchmarks in Kapitel 6. Die Eigenschaft *Volume* ist in der Nutzwertanalyse unter der Bezeichnung Datenmenge berücksichtigt. In der Nutzwertanalyse wird dabei ebenfalls auf ein Wachstum der Datenmenge eingegangen. Die Eigenschaft *Veracity* ist in der Nutzwertanalyse unter dem Kriterium verschiedenartige Daten enthalten. In der Bewertung des Kriteriums verschiedenartige Daten sind sowohl die Ergebnisse aus dem Performance-Test, als auch Dateitypen, die auf den Webseiten der Frameworks als mögliche Quellen genannt sind, enthalten.

Die Kriteriengruppe verteiltes System enthält die Kriterien Skalierbarkeit und Fehlertoleranz. Grundsätzlich haben verteilte Systeme noch weitere Eigenschaften, in dieser Nutzwertanalyse liegt der Fokus für die Bewertung eines verteilten Systems jedoch auf den beiden genannten Kriterien. Das Kriterium Skalierbarkeit umfasst die Fähigkeit zur horizontalen Skalierung und dem dazu benötigten praktischen Aufwand, der mit dem Prozess der Skalierung im jeweiligen

Framework verbunden ist. Das Kriterium Fehlertoleranz bezieht sich auf die Fähigkeit zur Weiterberechnung von Programmen, wenn während der Ausführung Knoten im Cluster ausfallen. Es ist zu beachten, dass in dem Performance-Test der Ausfall von Knoten nicht explizit getestet wurde. Deshalb bezieht sich die Bewertung der Fehlertoleranz auf theoretische Ereignisse, die sich durch die Ergebnisse des Performance-Tests ergeben haben.

Die Kriteriengruppe Benutzerfreundlichkeit enthält die Kriterien Zuverlässigkeit, Flexibilität und Konfigurationsaufwand. Das Kriterium Zuverlässigkeit ist in dieser Nutzwertanalyse definiert als die Stabilität der Knoten beziehungsweise der Slaves, die in der Master-Slave-Architektur der beiden Frameworks während der Performance-Analyse vorhanden gewesen ist. Das Kriterium Flexibilität enthält sowohl die Anpassungsfähigkeit der verschiedenen Programmiermodelle der beiden Frameworks, als auch Möglichkeiten zur Anpassung der Programme für die Ausführung im Cluster. Das Kriterium Kofigurationsaufwand bezeichnet die nötigen Vorbereitungen, um ein Cluster des jeweiligen Frameworks aufzusetzen.

7.1.3 Gewichtung der Entscheidungskriterien

Die Gewichtungen der genannten Entscheidungskriterien sind in Tabelle 7.2 aufgeführt. Der Schwerpunkt für die Auswahl eines Frameworks für den Kontext Big Data liegt auf der Kriteriengruppe drei V's, da diese der aktuellen Definition von Big Data entspricht. Deshalb ist das Gruppengewicht auf 50% festgelegt. Das wichtigste Kriterium innerhalb der Kriteriengruppe drei V's ist die Geschwindigkeit mit 40%, die Kriterien Datenmenge und verschiedenartige Daten haben demensprechend ein Gewicht von jeweils 30% innerhalb der Kriteriengruppe. Damit ergeben sich die Gesamtgewichte von 20% für die Geschwindigkeit und jeweils 15% für die Kriterien Datenmenge und verschiedenartige Daten.

Weitere Anforderungen der Frameworks ist die Eignung als verteiltes System mit einem Gruppengewicht von 30% und die Benutzerfreundlichkeit mit einem Gruppengewicht von 20%. Die Kriteriengruppe verteiltes System enthält die beiden Kriterien Skalierbarkeit und Fehlertoleranz, die jeweils ein Gewicht von 50% innerhalb der Kriteriengruppe aufweisen. Damit ergeben sich die Gesamtgewichte von jeweils 15% für die Kriterien Skalierbarkeit und Fehlertoleranz. In der Kriteriengruppe Benutzerfreundlichkeit verfügen die Kriterien Zuverlässigkeit und Flexibilität über ein Gewicht von jeweils 40%, der Konfigurationsaufwand wird mit 20% bewertet. Die Kriterien Zuverlässigkeit und Flexibilität erhalten damit ein Gesamtgewicht von 8%, der Konfigurationsaufwand ein Gesamtgewicht von 4%.

Kriteriengruppe	Gruppengewicht (%)	Kriterium	Kriteriumsgewicht innerhalb der Kriteriengruppe (%)	Gesamtgewicht des Kriteriums (%)
drei V's	50	Geschwindigkeit	40	20
		Datenmenge	30	15
		verschiedenartige Daten	30	15
verteiltes System	30	Skalierbarkeit	50	15
		Fehlertoleranz	50	15
Benutzerfreundlichkeit	20	Zuverlässigkeit	40	8
		Flexibilität	40	8
		Konfigurationsaufwand	20	4
Summe	100			100

Tabelle 7.2: Berechnung der Kriteriengewichte mit Hilfe von Kriteriengruppen (Quelle: In Anlehnung an: (KÏ4), S. 13)

7.1.4 Bewertung der Entscheidungskriterien

Die Skala für Bewertung der Entscheidungskriterien ist in Tabelle 7.3 abgebildet. Es werden die Noten 1 bis 6 für alle Kriterien vergeben, aus den Noten ergibt sich die entsprechende Bewertungszahl. Je höher dabei die Bewertungszahl ist, desto besser ist das Gesamtergebnis des Kriteriums.

In Tabelle 7.4 sind die vergebenen Noten der Kriterien für die Frameworks abgebildet. In dem Performance Test in Kapitel 6 ist die Ausführung aller Operationen mit Apache Spark schneller als die Ausführung mit Apache Hadoop. Bei einigen Operationen ist jedoch der Unterschied in der Geschwindigkeit geringer als erwartet. Deshalb bekommt Apache Hadoop für das Kriterium Geschwindigkeit die Note 3, Apache Spark hingegen die Note 1.

Apache Hadoop kann Operationen oder Programme gut auf große Datenmengen anwenden, wenn die Operationen und Programme dem Programmiermodell angepasst sind. Auch wachsende Datenmengen, die in der Performance Analyse als Test der x-fachen Datenmenge bezeichnet sind, werden zuverlässig verarbeitet. Deshalb bekommt Apache Hadoop für das

Bewertungs-zahl	Semantik	Benotung	Erklärung
5	sehr gut	1	Kriterium ist sehr gut erfüllt/außerordentlich nützlich
4	gut	2	Kriterium ist gut erfüllt/sehr nützlich
3	befriedigend	3	Kriterium ist in befriedigendem Maße erfüllt/nützlich
2	ausreichend	4	Kriterium ist ausreichend erfüllt/bedingt nützlich
1	mangelhaft	5	Kriterium ist nur unter Inkaufnahme wesentlicher Mängel erfüllt/nur in engen Grenzen und unter Inkaufnahme von Nachteilen nützlich
0	ungenügend	6	Kriterium ist nicht bzw. ungenügend erfüllt/nicht nützlich

Tabelle 7.3: Skala für die Berechnung der Bewertungen
(Quelle: Eigene Darstellung nach: (KÏ4), S. 17)

Kriterium Datenmenge die Note 2. Apache Spark ist in der Ausführung von Programmen auf großen Datenmengen sehr schnell, jedoch kommt es bei Operationen mit ebenfalls großem Output teilweise zu Problemen in der Performance. Ein großer Output hat im Benchmark zu einigen Schreibfehlern geführt, diese verringern die Geschwindigkeit der Ausführung von Programmen sehr stark. Aus diesem Grund bekommt Apache Spark für das Kriterium Datenmenge die Note 3. Die Frameworks Apache Hadoop und Apache Spark sind grundsätzlich beide skalierbar, jedoch ist der praktische Aufwand für eine horizontale Skalierung mit Apache Hadoop größer. Apache Hadoop scheint zwar in den Performance-Test effektiver horizontal zu skalieren, jedoch benötigt Apache Spark weniger Rechner um eine gleiche oder sogar bessere Performance als Apache Hadoop zu erreichen. Besonders wenn eine kleinere Anzahl an Knoten im Cluster für Operationen mit geringerer Rechenintensität gebraucht wird, ist Apache Spark im Vorteil. Deshalb erhält Apache Spark für das Kriterium Skalierbarkeit die Note 1 und Apache Hadoop die Note 2.

Kriterium	Gewichtung	Apache Hadoop		Apache Spark	
		Benotung	Bewertungszahl	Benotung	Bewertungszahl
Geschwindigkeit	20	3	3	1	5
Datenmenge	15	2	4	3	3
verschiedenartige Daten	15	2	4	1	5
Skalierbarkeit	15	2	4	1	5
Fehlertoleranz	15	1	5	3	3
Zuverlässigkeit	8	1	5	5	1
Flexibilität	8	4	1	1	5
Konfigurationsaufwand	4	4	1	1	5

Tabelle 7.4: Berechnung der Nutzwertanalyse
(Eigene Darstellung)

Die Fehlertoleranz wurde in der Performance-Analyse in Kapitel 6 nicht explizit getestet. Jedoch ermöglichen die unterschiedlichen Datenmodelle von Apache Hadoop und Apache Spark theoretische Rückschlüsse auf einen unterschiedlichen Erfüllungsgrad der Fehlertoleranz. Bei einem niedrigen Replikationsgrad der zu verarbeitenden Daten müsste Apache Hadoop fehlertoleranter als Apache Spark sein, da das Programmiermodell der *RDDs* von Apache Spark graphenbasiert ist. Wenn beim Ausfall eines Knotens im Cluster einige für die Berechnung benötigte Daten nicht mehrfach repliziert sind, hängt die komplette Berechnung des Graphen von den nicht mehr vorhandenen Daten ab. Im Gegensatz dazu erlaubt das Programmiermodell von Apache Hadoop theoretisch auch unvollständige Berechnungen und liefert dann ein nicht vollständiges Ergebnis, was aber in einigen Anwendungsfällen wie zum Beispiel Web-Analysen besser als gar kein Ergebnis sein kann. Aus diesem Grund erhält Apache Spark für das Kriterium Fehlertoleranz die Note 3 und Apache Hadoop die Note 1.

In der Performance-Analyse in in Kapitel 6 hat Apache Spark Probleme mit dem Kriterium Zuverlässigkeit gezeigt. Die Ausfälle von Workern sind für den Produktivbetrieb nicht akzeptabel, weshalb für Apache Spark aufgrund der Ergebnisse in der Performance Analyse in 6.4 nur die Note 5 vergeben werden kann. Apache Hadoop hat in der Performance Analyse keine Schwächen mit dem Kriterium Zuverlässigkeit gezeigt und bekommt deshalb die Note 1.

Apache Spark hat besondere Stärken in dem Kriterium Flexibilität. Es lassen sich beim Starten von Programmen relativ einfach viele Parameter, wie zum Beispiel die Anzahl *Executors* oder Anzahl die Prozessorkerne pro *Executor* setzen. Auch besteht die Möglichkeit, ohne großen Konfigurationsaufwand Programme lokal zum Debuggen testen. Damit bekommt Apache Spark für das Kriterium Flexibilität die Note 1. Auch mit Apache Hadoop lassen sich Parameter wie zum Beispiel die Anzahl *Mapper* in den Konfigurationen einstellen, jedoch haben solche Einstellungen oft Abhängigkeiten zu anderen Parametern. Dadurch müssen für Anpassungen oft ganze Folgen von Einstellungs-Parametern explizit berechnet und verändert werden. Apache Hadoop lässt sich im Gegensatz zu Apache Spark nicht lokal ohne ein Cluster testen. Es gibt zwar Bibliotheken zum Testen von MapReduce wie zum Beispiel MRUnit, jedoch können diese keine Ausführung im Cluster simulieren. Deshalb erhält Apache Hadoop für das Kriterium Flexibilität die Note 4.

Ingesamt ist der Vorbereitungsaufwand zum Aufbau eines Clusters mit Apache Spark deutlich geringer als mit Apache Hadoop. Mit Apache Spark lässt sich im Vergleich zu Apache Hadoop relativ schnell ein Cluster aufsetzen, es müssen weniger Konfigurationsdateien als in einem Hadoop-Cluster angepasst werden. Apache Spark bekommt deshalb die Note 1 für das Kriterium Konfigurationsaufwand, Apache Hadoop erhält wegen des im Vergleich deutlich höheren Aufwands die Note 4.

7.1.5 Nutzwertberechnung

In Tabelle 7.5 ist das Gesamtergebnis der Nutzwertanalyse angebildet. Apache Spark ist demnach das bessere Framework für die Verabeitung von Daten mit den Eigenschaften der drei V's und benutzerfreundlicher als Apache Hadoop. Apache Hadoop eignet sich im Vergleich zu Apache Spark besser für die Anforderungen eines verteilten Systems. Dennoch ist Apache Spark nach dieser Nutzwertanalyse insgesamt das bessere Framework für den Kontext Big Data.

Framework	drei V's	verteiltes System	Benutzerfreundlichkeit	Gesamtergebnis
Apache Hadoop	11	9	7	27
Apache Spark	13	8	11	32

Tabelle 7.5: Nutzwertanalyse Gesamtergebnis
(Quelle: Eigene Darstellung)

7.2 Analyse der praktischen Vor- und Nachteile von Apache Hadoop und Apache Spark

In der Performance-Analyse von Apache Hadoop und Apache Spark sind die Vorteile von Apache Hadoop die Robustheit des Frameworks zur Laufzeit und die daraus folgende bessere Berechenbarkeit der Ausführungsdauer von Programmen, die im Hadoop-Framework gestartet werden. Apache Sparks Stärken sind die Geschwindigkeit und die Flexibilität. Die Flexibilität ist sowohl durch das anpassungsfähige Programmiermodell als auch durch die Möglichkeit der dynamischen Auslastung des Clusters durch Angabe von Optionen beim Start von Programmen gekennzeichnet. Das Programmiermodell von Apache Spark lässt viele verschiedene Anwendungen wie Batch-Processing, SQL und Streaming zu. Interessant ist dabei der Vergleich von Apache Spark mit spezialisierten Frameworks. In diesem Benchmark hat Apache Spark in der Geschwindigkeit bessere Ergebnisse im Vergleich zu dem auf Batch-Processing spezialisierten MapReduce-Framework und dem auf SQL spezialisierten Framework Apache Hive erzielt. Apache Sparks interne Architektur ist tendenziell auf Batch-Processing ausgerichtet, weitere Tests könnten die Performance von Spark Streaming gegen Frameworks wie Apache Flink messen, die grundsätzlich von der Architektur auf Streaming und Anwendungen in diesem Bereich spezialisiert sind.

Die Geschwindigkeitsvorteile von Apache Spark gegenüber Apache Hadoop sind bei einigen Operationen so groß, dass aus Kostengründen Apache Spark als Framework für die Verarbeitung von Daten im Kontext gewählt werden sollte. Für einige Operationen ist ein Spark-Cluster aus einem *Master* und einem *Slave* schneller als ein Hadoop-Cluster mit einem *Master* und vier *Slaves*. Es ist also durch die Verwendung von Apache Spark möglich, in vielen Fällen Rechenkapazität oder ganze Rechner beziehungsweise virtuelle Maschinen einzusparen. Jedoch lohnt es sich dafür, die Operationen auf Geschwindigkeit zu testen. In der Performance-Analyse in Kapitel 6 haben sich nicht bei allen Operationen so deutliche Geschwindigkeitsvorteile ergeben, bei solchen Operationen sind dementsprechend kaum Kostenvorteile zu erzielen.

Ein theoretisches Risiko für Apache Spark besteht darin, dass spezialisierte Frameworks in dafür geeigneten Anwendungsfällen eine bessere Performance als Apache Spark liefern

konnen. Dieser Fall könnte eintreten, da Apache Spark ein breiteres Feld an möglichen Anwendungen unterstützt und deshalb bei der Weiterentwicklung des Frameworks eher generell ausgerichtet werden muss. Dieses theoretische Risiko besteht bei der Annahme, das spezialisierte Frameworks aufgrund der eingeschränkten Anwendungsfälle bessere Möglichkeiten zur Optimierung haben.

8 Zusammenfassung und Ausblick

In diesem Kapitel sind die Ergebnisse der Bachelorarbeit zusammengefasst und Anmerkungen aufgelistet, die weitere Forschung ermöglichen. Zum Abschluss des Kapitels werden Chancen von Apache Spark vorgestellt, die gegebenenfalls den weiteren Erfolg von Apache Spark beeinflussen könnten.

8.1 Zusammenfassung

In dieser Bachelorarbeit wurden die Frameworks Apache Hadoop und Apache Spark in einem Performance-Test verglichen und in der anschließenden Analyse für die Nutzung im Kontext Big Data bewertet. Die Analyse des Performance-Tests hat Apache Spark als insgesamt besseres Framework für die Verarbeitung von Daten bewertet, die der heutigen Definition von Big Data entsprechen. Apache Spark hat im Performance-Test deutliche Geschwindigkeitsvorteile gegenüber Apache Hadoop gezeigt. Außerdem ist Apache Spark flexibler in der Benutzung, tendenziell einfachere Anwendungsfälle lassen sich im Spark-Framework mit weniger Aufwand als im Hadoop-Framework konfigurieren und ausführen. Deshalb bietet Apache Spark ingesamt einen einfacheren Einstieg in die Verarbeitung von Big Data. Die Geschwindigkeitsvorteile von Apache Spark sind bei einigen Anwendungen so stark ausgeprägt, dass durch das Einsparen von Rechenleistung Kostenvorteile möglich sind.

Apache Hadoop ist im Gegensatz zu Apache Spark sehr zuverlässig und hat zumindest mit HDFS als Bestandteil des Hadoop-Frameworks eine bisher unverzichtbare Komponente für die Speicherung von Daten im Kontext Big Data. Für die Analyse von Big Data ist allerdings das Programmiermodell von Apache Hadoop zu eingeschränkt. Auch im Programmiermodell hat Apache Spark gegenüber Apache Hadoop Vorteile durch eine größere Anzahl möglicher Anwendungen. Zusärtlich bietet Apache Sparks API eine höhere Variabilität durch die mögliche Verwendung von mehreren Programmiersprachen.

8.2 Anmerkungen und Anregungen für weitere Forschung

Der ausgeführte Performance-Test basiert auf einem selbst erstellten Benchmark. Dieser Benchmark wurde benutzt, da es bisher noch keinen standardisierten Benchmark für Frameworks im Kontext Big Data gibt. Sobald es einen standardisierten Benchmark gibt, müssten die getesteten Hypothesen für den Standard-Benchmark erneut getestet werden.

Der ausgeführte Performance-Test hat ergeben, dass die Verwendung von YARN als Cluster Manager in Bezug auf die Stabilität der Anwendungen mit Apache Hadoop zuverlässiger funktioniert als mit Apache Spark. Interessant wären an dieser Stelle weitere Tests mit anderen Cluster Managern. In weiteren Benchmarks könnte der Schwerpunkt noch mehr auf iterative Anwendungen gerichtet werden. Auf der Website von Apache Spark wird bei iterativen Anwendungen eine bis zu 100-fach schnellere Geschwindigkeit von Apache Spark gegenüber Apache Hadoop beschrieben. Vielleicht kann in Tests mit stark iterativen Anwendungen die 100-fache Geschwindigkeit wenigstens annähernd erreicht werden. In dem Performance-Test wurden tendenziell wenig Schreibzugriffe getestet. Allerdings hat Apache Spark bei den getesteten Schreibzugriffen in Verbindung mit HDFS teilweise Probleme in der Performance durch Ausfälle von einzelnen *Tasks* oder ganzen *Workern* gehabt. Hier könnten weitere Tests eventuell konkrete Schwachstellen von Apache Spark aufdecken. Für den Performance-Test wurden keine Optimierungen der Frameworks verwendet, wahrscheinlich wären damit bei beiden Frameworks noch einige Verbesserungen der Geschwindigkeit möglich.

Weitere interessante Fragestellungen zu Apache Spark ergeben sich im Bereich Streaming. In dieser Bachelorarbeit wurden Operationen getestet, die eher Batch-orientiert sind. Eine für Apache Spark entscheidende Frage ist, ob es zu reinen Streaming-Frameworks konkurrenzfähig ist, obwohl die interne Architektur von Apache Spark eher auf Batch-Processing ausgerichtet ist.

8.3 Chancen von Apache Spark im Bereich Big Data

Aus technischer Sicht ist die große Herausforderung von Apache Spark, eine im Idealfall bessere oder mindestens vergleichbare Geschwindigkeit von Anwendungen im Bereich Batch-Processing, SQL, Machine Learning und parallele Graph-Verarbeitung zu bieten wie Frameworks, die auf solche Anwendungsfälle spezialisiert sind. Sobald diese Bedingung erfüllt ist, kann Apache Spark als ein Framework effektiv einen großen Bereich von Anwendungen in Unternehmen übernehmen.

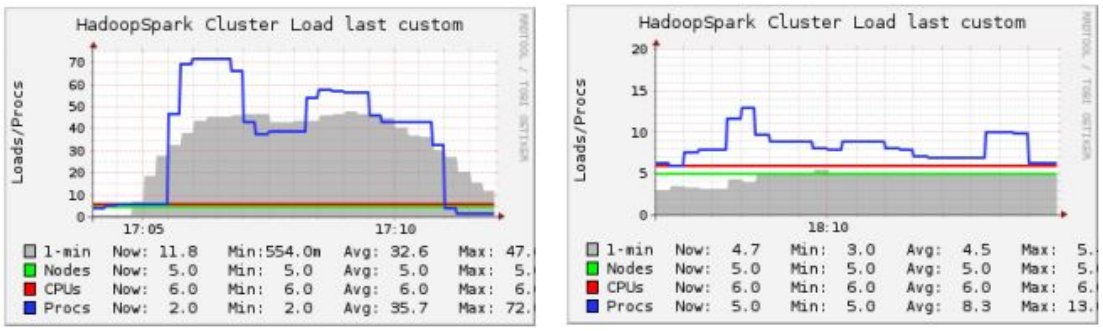
Die entscheidende Chance von Apache Spark ist dann die Verarbeitung von Daten aus heterogenen Quellen. Sowohl Internet-Unternehmen wie zum Beispiel Facebook, als auch

Unternehmen aus konservativen Branchen haben große Datenmengen in unterschiedlichsten Formaten. Je mehr Datenformate in zukünftigen Releases von Apache Spark hinzukommen, desto mehr lohnt sich die Verwendung von Apache Spark als Analysewerkzeug für Daten aus über Jahrzehnten gewachsenen Datensammlungen. Diese sind besonders in Unternehmen, die bereits lange existierenden, vorhanden und werden bisher nur wenig für Analysen genutzt (vgl. (FHL14), S. 2). Mit Apache Spark ist die Erstellung von Analysen und Reports aus verschiedenen Datenquellen und die zusätzlichn Integration von Echtzeitdaten möglich. Apache Spark kann dabei als flexibles Verarbeitungs- und Analysewerkzeug für Social-Media-Daten, Streaming und SQL-Daten dienen und als kostengünstiges Framework sowohl in kleinen als auch in großen Unternehmen genutzt werden.

A Cluster Monitoring

Alle folgenden Abbildungen im Anhang A sind während der Ausführung der jeweiligen Operation vom Ganglia-Monitor erstellt wurden. Die restlichen Clustermetriken befinden sich auf der DVD, die der augedruckten Variante dieser Bachelorarbeit beiliegt. Weitere Inhalte der DVD sind die implementierten Operationen von Apache Hadoop und Apache Spark, diese sind jeweils als Projekt und als JAR gespeichert. Die mit Apache Hive ausgeführten Hive-Queries befinden sich auf der DVD in einer Textdatei, außerdem ist die komplette Beispieldatei aus Anhang B gespeichert.

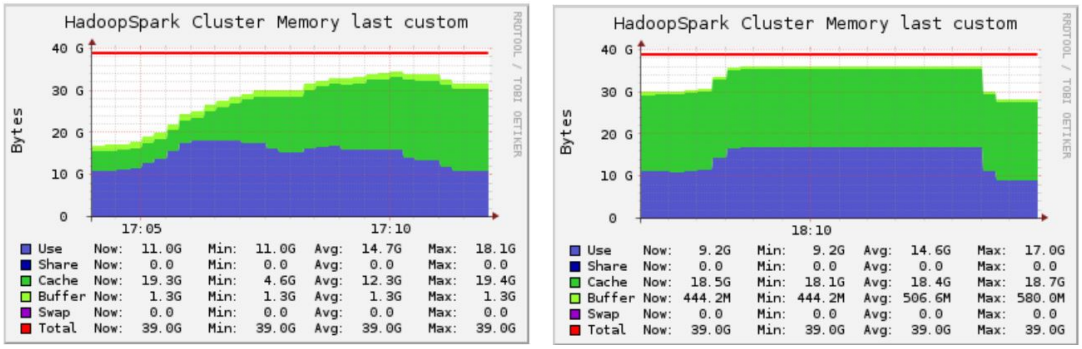
A.1 Hypothese 1



(a) Apache Hadoop (b) Apache Spark

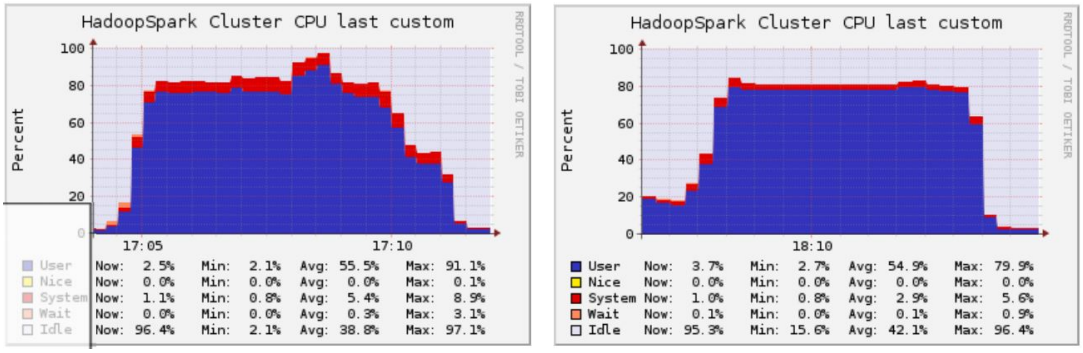
Abbildung A.1: Durchschnittliche Anzahl laufender Prozesse von Apache Hadoop und Apache Spark während der Operation SumYear

A.2 Hypothese 2



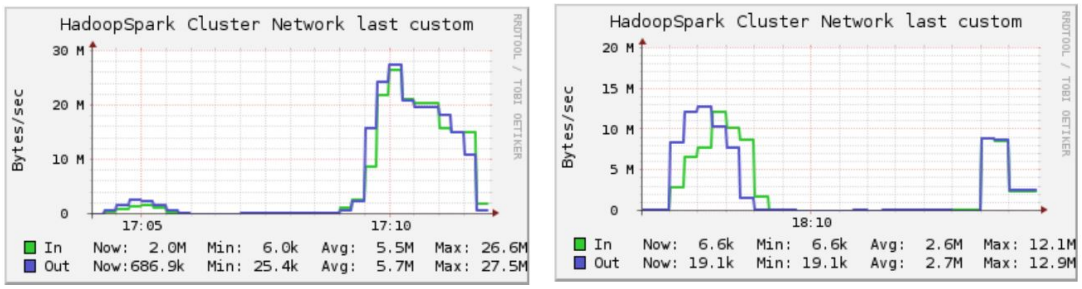
(a) Apache Hadoop (b) Apache Spark

Abbildung A.2: Auslastung des Hauptspeichers von Apache Hadoop und Apache Spark während der Operation SumYear



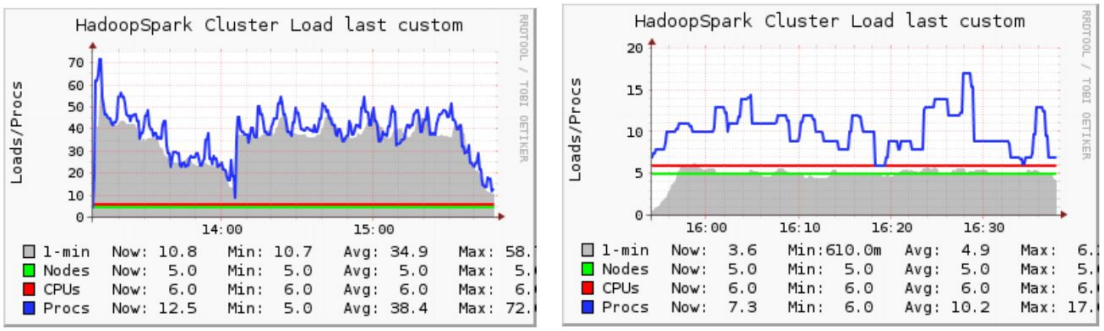
(a) Apache Hadoop (b) Apache Spark

Abbildung A.3: Durchschnittliche Auslastung der CPUs von Apache Hadoop und Apache Spark während der Operation SumYear



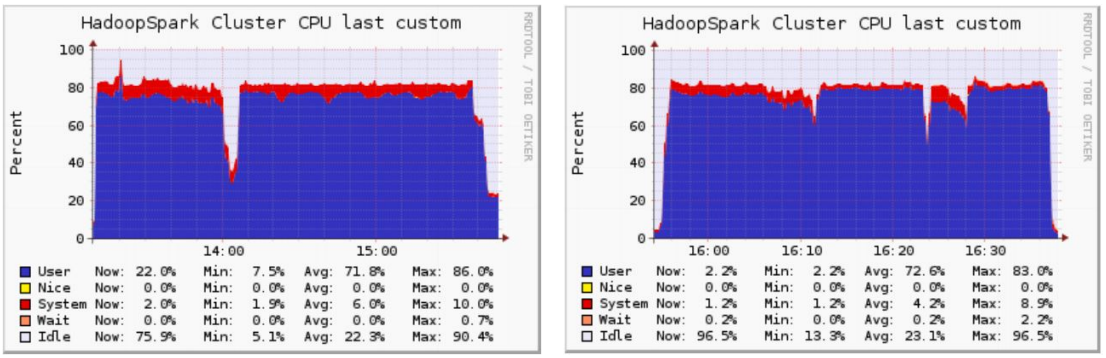
(a) Apache Hadoop (b) Apache Spark

Abbildung A.4: Durchschnittliche Auslastung des Netzwerks von Apache Hadoop und Apache Spark während der Operation SumYear



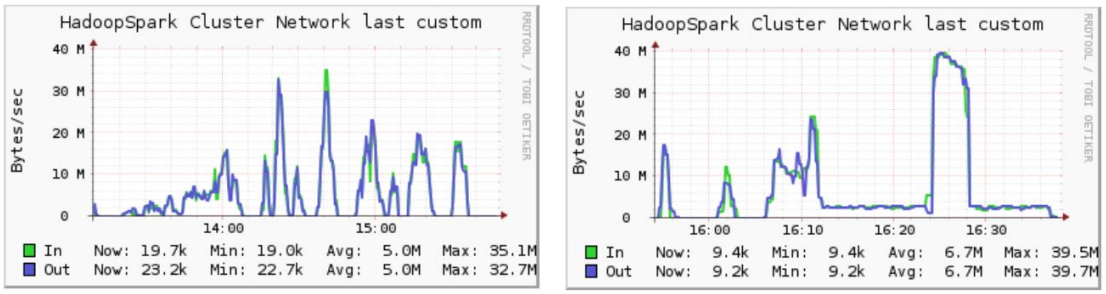
(a) Apache Hadoop (b) Apache Spark

Abbildung A.5: Durchschnittliche Anzahl laufender Prozesse von Apache Hadoop und Apache Spark während der Operation Join



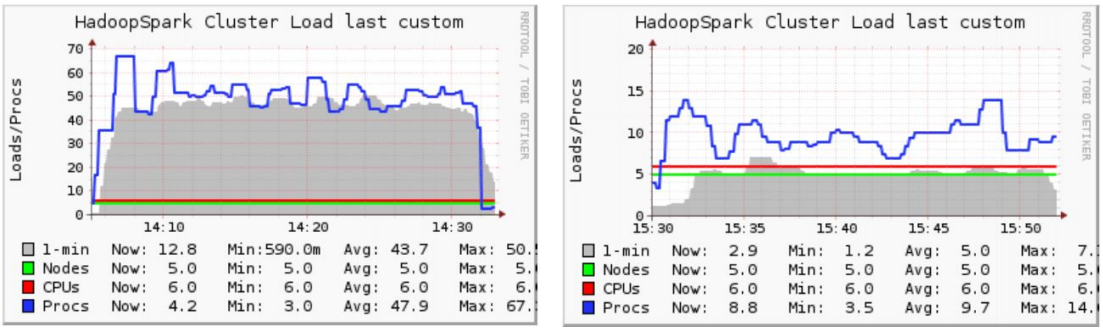
(a) Apache Hadoop (b) Apache Spark

Abbildung A.6: Durchschnittliche CPU-Auslastung von Apache Hadoop und Apache Spark während der Operation Join



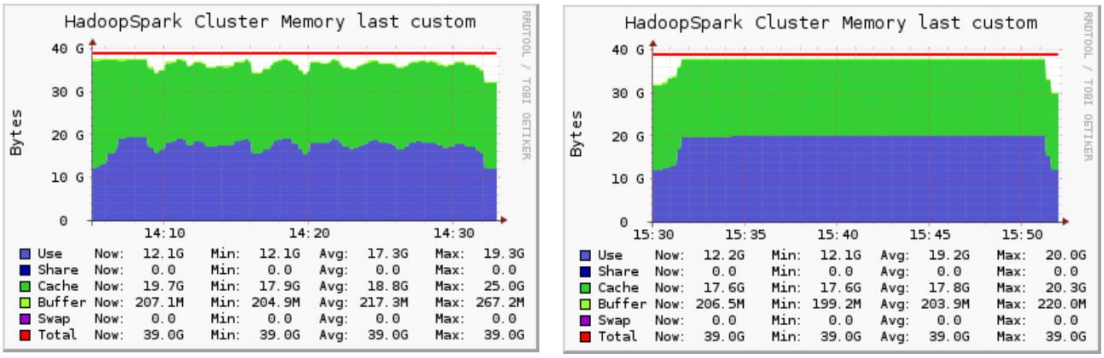
(a) Apache Hadoop (b) Apache Spark

Abbildung A.7: Durchschnittliche Auslastung des Netzwerks von Apache Hadoop und Apache Spark während der Operation Join



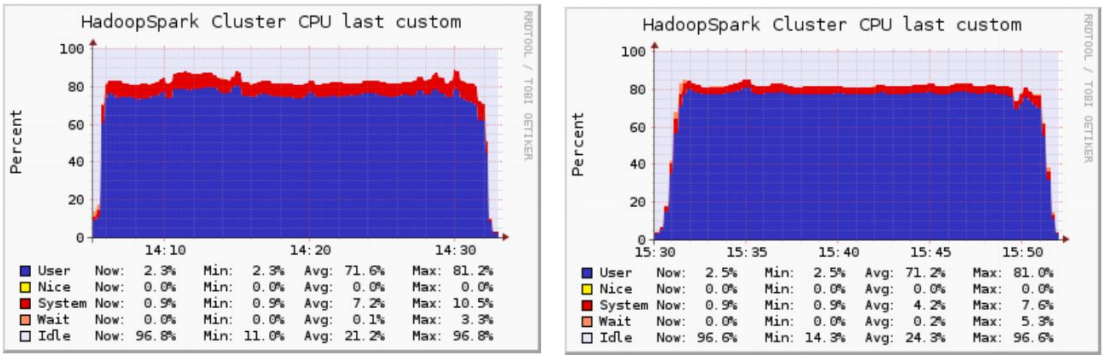
(a) Apache Hadoop (b) Apache Spark

Abbildung A.8: Durchschnittliche Anzahl laufender Prozesse von Apache Hadoop und Apache Spark während der Operation SumYear



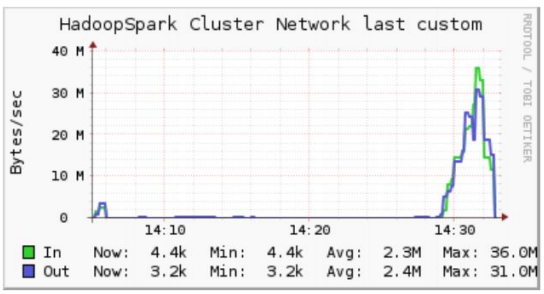
(a) Apache Hadoop (b) Apache Spark

Abbildung A.9: Auslastung des Hauptspeichers von Apache Hadoop und Apache Spark während der Operation SumYear

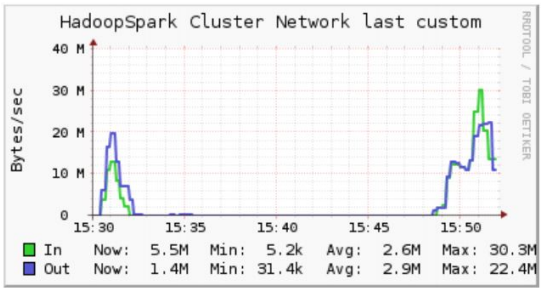


(a) Apache Hadoop (b) Apache Spark

Abbildung A.10: Durchschnittliche CPU-Auslastung von Apache Hadoop und Apache Spark während der Operation SumYear



(a) Apache Hadoop

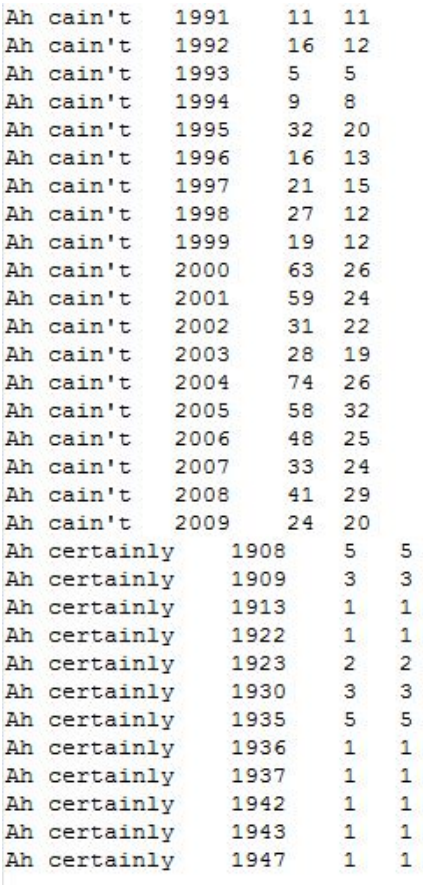


(b) Apache Spark

Abbildung A.11: Durchschnittliche Auslastung des Netzwerks von Apache Hadoop und Apache Spark während der Operation SumYear

B Beispieldaten

Beispielhaft ist an dieser Stelle ein Abschnitt der Datei "googlebooks-eng-fiction-all-2gram-20120701-ah” mit den englischen 2-Grammen beginnend mit den Buchstaben „ah” dargestellt:



```
Ah cain't   1991    11  11
Ah cain't   1992    16  12
Ah cain't   1993    5   5
Ah cain't   1994    9   8
Ah cain't   1995    32  20
Ah cain't   1996    16  13
Ah cain't   1997    21  15
Ah cain't   1998    27  12
Ah cain't   1999    19  12
Ah cain't   2000    63  26
Ah cain't   2001    59  24
Ah cain't   2002    31  22
Ah cain't   2003    28  19
Ah cain't   2004    74  26
Ah cain't   2005    58  32
Ah cain't   2006    48  25
Ah cain't   2007    33  24
Ah cain't   2008    41  29
Ah cain't   2009    24  20
Ah certainly    1908    5   5
Ah certainly    1909    3   3
Ah certainly    1913    1   1
Ah certainly    1922    1   1
Ah certainly    1923    2   2
Ah certainly    1930    3   3
Ah certainly    1935    5   5
Ah certainly    1936    1   1
Ah certainly    1937    1   1
Ah certainly    1942    1   1
Ah certainly    1943    1   1
Ah certainly    1947    1   1
```

Abbildung B.1: Beispiel für N-Gramme, die im Performance-Test benutzt wurden (Quelle: (Goo12))

Literaturverzeichnis

[Apaa] Apache Software Foundation: *Client.java*. https://github.com/apache/hadoop/blob/trunk/hadoop-yarn-project/hadoop-yarn/hadoop-yarn-applications/hadoop-yarn-applications-distributedshell/src/main/java/org/apache/hadoop/yarn/applications/distributedshell/Client.java, Abruf: 04.05.15

[Apab] Apache Software Foundation: *Cluster Mode Overview*. http://spark.apache.org/docs/latest/cluster-overview.html, Abruf: 12.05.2015

[Apac] Apache Software Foundation: *DataNodeINodeAttributes.java*. https://github.com/apache/hadoop/blob/trunk/hadoop-hdfs-project/hadoop-hdfs/src/main/java/org/apache/hadoop/hdfs/server/namenode/INodeAttributes.java, Abruf: 16.04.15

[Apad] Apache Software Foundation: *DataNode.java*. https://github.com/apache/hadoop/blob/trunk/hadoop-hdfs-project/hadoop-hdfs/src/main/java/org/apache/hadoop/hdfs/server/datanode/DataNode.java, Abruf: 08.04.2015

[Apae] Apache Software Foundation: *NameNode.java*. https://github.com/apache/hadoop/blob/trunk/hadoop-hdfs-project/hadoop-hdfs/src/main/java/org/apache/hadoop/hdfs/server/namenode/NameNode.java, Abruf: 15.04.2015

[Apaf] Apache Software Foundation: *RDD.scala*. https://github.com/apache-spark/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/RDD.scala, Abruf: 15.04.2015

[Apag] Apache Software Foundation: *RessourceManager.java*. https://github.com/apache/hadoop/blob/trunk/hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-resourcemanager/src/main/java/org/apache/hadoop/yarn/server/resourcemanager/ResourceManager.java, Abruf: 11.05.2015

[Apah] Apache Software Foundation: *SparkContext.scala*. https://github.com/apache-spark/spark/blob/master/core/src/main/scala/org/apache/spark/SparkContext.scala, Abruf: 08.05.2015

[Apai] Apache Software Foundation: *YarnScheduler.java*. https://github.com/apache/hadoop/blob/trunk/hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-resourcemanager/src/main/java/org/apache/hadoop/yarn/server/resourcemanager/scheduler/YarnScheduler.java, Abruf: 11.05.2015

[Apa15a] Apache Spark: *Apache Spark*. http://spark.apache.org. Version: 2015, Abruf: 13.09.2015

[Apa15b] Apache Spark: *Apache Spark Examples*. https://spark.apache.org/examples.html. Version: 2015, Abruf: 14.08.2015

[Apa15c] Apache Spark: *Spark Programming Guide*. http://spark.apache.org/docs/latest/programming-guide.html. Version: 2015, Abruf: 15.05.2015

[Apa15d] Apache Spark: *Spark SQL Programming Guide*. https://spark.apache.org/docs/latest/sql-programming-guide.html#compatibility-with-apache-hive. Version: 2015, Abruf: 03.09.2015

[AXL+15] Armbrust, Michael ; Xin, Reynold S. ; Lian, Cheng ; Huai, Yin ; Liu, Davies ; Bradley, Joseph K. ; Meng, Xiangrui ; Kaftan, Tomer ; Franklin, Michael J. ; Ghodsi, Ali ; Zaharia, Matei: Spark SQL: Relational Data Processing in Spark. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2015 (SIGMOD '15). – ISBN 978–1–4503–2758–9, 1383–1394

[BB13] Berberich, Klaus ; Bedathur, Srikanta: Computing n-gram statistics in MapReduce. In: *Proceedings of the 16th International Conference on Extending Database Technology - EDBT '13* (2013), 101. http://dx.doi.org/10.1145/2452376.2452389, Abruf: 18.03.15. – DOI 10.1145/2452376.2452389. ISBN 9781450315975

[CER15] CERN: *CERN-Website*. http://home.web.cern.ch/about/computing. Version: 2015, Abruf: 28.03.2015

[Clo15] Cloudera: *Running Spark Applications*. http://www.cloudera.com/content/cloudera/en/documentation/core/v5-2-x/topics/cdh_ig_running_spark_apps.html. Version: 2015, Abruf: 14.09.2015

[CLS14] Chou, Sophie ; Li, Willliam ; Sridharan, Ramesh: Democratizing Data Science: Effecting positive social change with data science. (2014), 1f. http://people.csail.mit.edu/rameshvs/content/papers/Chou-Li-Sridharan-DemocratizingDataScience.pdf, Abruf: 28.08.2015

[Con15] Confluence Administrator: *Apache Hive*. https://cwiki.apache.org/confluence/display/Hive/Home. Version: 2015, Abruf: 31.08.2015

[CWR12] Capriolo, Edward ; Wampler, Dean ; Rutherglen, Jason: *Programming Hive*. 2012. – 328 S. http://books.google.com/books?id=4yYinwEACAAJ&pgis=1. – ISBN 1449319335

[DG08] Dean, J ; Ghemawat, S: MapReduce : Simplified Data Processing on Large Clusters. In: *Communications of the ACM* 51 (2008), Nr. 1, 1-13. http://www.usenix.org/events/osdi04/tech/full_papers/dean/dean_html/, Abruf: 29.03.2015. ISBN 9781595936868

[DK13] Das, T K. ; Kumar, P M.: BIG Data Analytics: A Framework for Unstructured Data Analysis. In: *International Journal of Engineering Science & Technology* 5 (2013), Nr. 1, 153–156. http://search.ebscohost.com/login.aspx?direct=true&db=edb&AN=88917864&site=eds-live$\delimiter"026E30F$nhttp://search.ebscohost.com/login.aspx?direct=true&profile=ehost&scope=site&

authtype=crawler&jrnl=09755462&AN=88917864&h=oI5GF+qdtLnyxEKvYFe0YPeNMLsOnOKP7sN58dAdrk2J/2Ioz, Abruf: 26.04.2015. ISBN 09755462

[FDGR11] Fadika, Zacharia ; Dede, Elif ; Govindaraju, Madhusudhan ; Ramakrishnan, Lavanya: Benchmarking MapReduce implementations for application usage scenarios. In: *Proceedings - 2011 12th IEEE/ACM International Conference on Grid Computing, Grid 2011* (2011), S. 90–97. http://dx.doi.org/10.1109/Grid.2011.21. – DOI 10.1109/Grid.2011.21. – ISBN 978–1–4577–1904–2

[FHL14] Fan, Jianqing ; Han, Fang ; Liu, Han: Challenges of Big Data analysis. In: *National Science Review* 1 (2014), Nr. 2, 293-314. http://dx.doi.org/10.1093/nsr/nwt032, Abruf: 17.04.2015. – DOI 10.1093/nsr/nwt032

[Fre14] Freiknecht, Jonas: *Big Data in der Praxis : Lösungen mit Hadoop, HBase und Hive ; Daten speichern, aufbereiten, visualisieren.* München : Hanser, 2014. – ISBN 978–3–446–43959–7

[Gat15] Gates, Alan: *Hive Transactions.* https://spark.apache.org/examples.html. Version: 2015, Abruf: 31.08.2015

[GGL03] Ghemawat, Sanjay ; Gobioff, Howard ; Leung, Shun-Tak: The Google File System. In: *SIGOPS Oper. Syst. Rev.* 37 (2003), Nr. 5, 29–43. http://dx.doi.org/10.1145/1165389.945450, Abruf: 28.04.2015. – DOI 10.1145/1165389.945450. – ISSN 0163–5980

[GGP14] Gohil, Parth ; Garg, Dweepna ; Prof. Panchal, Bakul: A Performance Analysis of MapReduce Applications on Big Data in Cloud based Hadoop. (2014), Nr. 978, S. 2–7. ISBN 9781479938346

[Goo12] Google: *Google Books Ngram Viewer.* https://storage.googleapis.com/books/ngrams/books/datasetsv2.html. Version: 2012, Abruf: 28.03.2015

[Gra03] Grabmüller, Martin: Multiparadigmen- Programmiersprachen / TU-Berlin. Version: 2003. http://cs.tu-berlin.de/cs/ifb/Ahmed/RoteReihe/2003/TR2003_15.pdf, Abruf: 23.04.2015. 2003. – Forschungsbericht. – 78 S.

[Had14a] Hadoop: *HDFS Design*. Website. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. Version: November 2014, Abruf: 07.04.2015

[Had14b] Hadoop: *What is Apache Hadoop?* Website. https://hadoop.apache.org/. Version: 2014, Abruf: 07.04.2015

[Had15a] Hadoop: *Apache Hadoop NextGen MapReduce (YARN)*. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html. Version: 2015, Abruf: 29.04.2015

[Had15b] Hadoop: *Hadoop Powered By*. https://wiki.apache.org/hadoop/PoweredBy. Version: 2015, Abruf: 07.04.2015

[Had15c] Hadoop: *MapReduce NextGen aka YARN aka MRv2*. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/index.html. Version: 2015, Abruf: 29.04.2015

[Kï14] Kühnapfel, Jörg B.: *Nutzwertanalyse in Marketing und Vertrieb*. Springer Fachmedien Wiesbaden, 2014. – ISBN 9783658055080

[KKW+13] Krish, K. R. ; Khasymski, Aleksandr ; Wang, Guanying ; Butt, Ali R. ; Makkar, Gaurav: On the use of shared storage in shared-nothing environments. In: *Proceedings - 2013 IEEE International Conference on Big Data, Big Data 2013* (2013), S. 313–318. http://dx.doi.org/10.1109/BigData.2013.6691589. – DOI 10.1109/BigData.2013.6691589. ISBN 9781479912926

[KKW15] Karau, Holden ; Konwinski, Andy ; Wendell, Patrick: *Learning Spark: Lightning-Fast Big Data Analytics*. 1. O'Reilly Media, Inc., 2015. – 274 S. – ISBN 9781449358624

[Lan01] Laney, Doug: 3D Data Management: Controlling Data Volume, Velocity, and Variety. In: *Application Delivery Strategies* 949 (2001), February, S. 4

[LD10] Lin, Jimmy ; Dyer, Chris: *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010. – 1–177 S. http://dx.doi.org/10.2200/S00274ED1V01Y201006HLT007. http://dx.doi.org/10.2200/S00274ED1V01Y201006HLT007. – ISBN 1608453421

[LRI+14] Lu, Xiaoyi ; Rahman, Md. Wasi U. ; Islam, Nusrat ; Shankar, Dipti ; Panda, Dhabaleswar K.: Accelerating Spark with RDMA for Big Data Processing: Early Experiences. In: *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects* (2014), 9–16. http://dx.doi.org/10.1109/HOTI.2014.15. – DOI 10.1109/HOTI.2014.15. ISBN 978–1–4799–5860–3

[Man13] Mandl, Peter: *Grundkurs Betriebssysteme : Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation, Virtualisierung*. 4. Auflage. München : Springer Fachmedien Wiesbaden, 2013. – 289–310 S. http://dx.doi.org/10.1007/978-3-8348-2301-4. http://dx.doi.org/10.1007/978-3-8348-2301-4. – ISBN 978–3–8348–1897–3

[MMSW07] Michael, M. ; Moreira, J.E. ; Shiloach, D. ; Wisniewski, R.W.: Scale-up x Scale-out: A Case Study using Nutch/Lucene. In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007, S. 1–8

[PPR+09] Pavlo, Andrew ; Paulson, Erik ; Rasin, Alexander ; Abadi, Daniel J. ; DeWitt, David J. ; Madden, Samuel ; Stonebraker, Michael: A comparison of approaches to large-scale data analysis. In: *Proceedings of the 35th SIGMOD international conference on Management of data* (2009), S. 165–178. http://dx.doi.org/10.1145/1559845.1559865. – DOI 10.1145/1559845.1559865. ISBN 9781605585512

[RLOW15] Ryza, Sandy ; Laserson, Uri ; Owen, Sean ; Wills, Josh: *Advanced Analytics with Spark*. O'Reilly, 2015. – 254 S. – ISBN 9781491912690

[RSS15] Rahm, Erhard ; Saake, Gunter ; Sattler, Kai-Uwe: *Verteiltes und Paralleles Datenmanagement : Von verteilten Datenbanken zu Big Data und Cloud*. Berlin : Springer-Verlag Berlin Heidelberg, 2015. – 379 S. http://link.springer.com/10.1007/978-3-642-45242-0. – ISBN 978–3–642–45241–3

[SF12] Sadalage, PJ ; Fowler, M: *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. 2012. http://dx.doi.org/0321826620. http://dx.doi.org/0321826620. – ISBN 9780321826626

[Shv10] Shvachko, Konstantin: HDFS Scalability: The limits to growth. In: *;Login: THE USENIX MAGAZINE* 35 (2010), April, Nr. 2, 6–16. https://www.usenix.org/

publications/login/april-2010-volume-35-number-2/hdfs-scalability-limits-growth, Abruf: 28.03.2015

[SKRC10] Shvachko, Konstantin ; Kuang, Hairong ; Radia, Sanjay ; Chansler, Robert: The Hadoop Distributed File System. In: *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on* 0 (2010), S. 1–10. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/MSST.2010.5496972. – DOI http://doi.ieeecomputersociety.org/10.1109/MSST.2010.5496972. ISBN 978–1–4244–7152–2

[VMD+13] Vavilapalli, Vinod K. ; Murthy, Arun C. ; Douglas, Chris ; Agarwali, Sharad ; Konar, Mahadev ; Evans, Robert ; Graves, Thomas ; Lowe, Jason ; Shah, Hitesh ; Seth, Siddharth ; Saha, Bikas ; Curino, Carlo ; O'Malley, Owen ; Radia, Sanjay ; Reed, Benjamin ; Baldeschwieler, Eric: Apache Hadoop YARN : Yet Another Resource Negotiator. In: *ACM Symposium on Cloud Computing*, 2013. – ISBN 978–1–4503–2428–1, 16

[Whi12] White, Tom: *Hadoop: The Definitive Guide.* 3rd Edition. O'Reilly, 2012. – 647 S. – ISBN 9781449311520

[Wik15] Wikipedia: *Inode — Wikipedia, Die freie Enzyklopädie.* http://de.wikipedia.org/w/index.php?title=Inode&oldid=140344814. Version: 2015, Abruf: 15.04.2015

[WSV14] Wadkar, Sameer ; Siddalingaiah, Madhu ; Venner, Jason: *Pro Apache Hadoop.* 2nd. Berkely, CA, USA : Apress, 2014. – ISBN 1430248637, 9781430248637

[Xin14] Xin, Reynold: *Spark officially sets a new record in large-scale sorting.* http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html. Version: November 2014, Abruf: 18.05.2015

[ZCDD12] Zaharia, Matei ; Chowdhury, Mosharaf ; Das, Tathagata ; Dave, Ankur: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), 14. http://dx.doi.org/10.1111/j.1095-8649.2005.00662.x. – DOI 10.1111/j.1095–8649.2005.00662.x. ISBN 978–931971–92–8

[ZCF+10] Zaharia, Matei ; Chowdhury, Mosharaf ; Franklin, Michael J. ; Shenker, Scott ; Stoica, Ion: Spark: Cluster Computing with Working Sets. In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. Berkeley, CA, USA : USENIX Association, 2010 (HotCloud'10), 10–10

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 24.09.15 Tim Horgas