

Big Data Processing mit Apache Spark

vorgelegt von

Sascha P. Lorenz

Matrikel-Nr.: 501 63 21

dem Fachbereich Technik
der Hochschule Emden-Leer
und der Beuth Hochschule für Technik Berlin
vorgelegte Masterarbeit
zur Erlangung des akademischen Grades
Master of Science (M.Sc.)
im Studiengang
Medieninformatik (Master)

Tag der Abgabe 07. März 2015

1. Betreuer Prof. Dr. Stefan Edlich Beuth Hochschule für Technik Berlin

2. Betreuer Prof. Dr. Schiemann-Lillie Hochschule Emden-Leer

Kurzfassung

Daten gelten heute als der Rohstoff der Zukunft. Ständig werden extreme Mengen von Daten von den unterschiedlichsten Quellen generiert. Sei es durch Social-Media, durch Suchmaschinen, durch Bewegungs- oder andere Sensordaten, durch Logdateien, etc. Die Menge der generierten Daten steigt von Jahr zu Jahr an - von ca 130 Exabyte¹ im Jahre 2005 bis zu 8591 Exabyte für das Jahr 2015 und für 2020 werden Prognosen zufolge sogar über 40000 Exabyte Daten erzeugt [Sta15]. Diese Daten haben wenig miteinander gemeinsam. Es handelt sich um strukturierte oder unstrukturierte Datensätze, um persistente oder um flüchtige Daten aus einem Strom. Für all diese Daten sollen aber vergleichbare Werkzeuge zur Analyse und Verarbeitung verfügbar sein. Ein Ansatz für dieses Problem liefert Apache Spark mit dem dazugehörigen Berkeley Data Analytics Stack. Dies sind Werkzeuge für Analyse massiver Datenmengen, Verarbeitung von flüchtigen Streamingdaten, Batchverarbeitung, SQL-Abfragen und verschiedenen Aufgaben aus dem Bereich des Machine Learning. In dieser Thesis werden die Bibliotheken des Berkeley Data Analytics Stack vorgestellt, gegenüber Alternativimplementierungen verglichen und Messungen unterzogen. Für diesen Zweck wurden Metriken definiert, sowie Testumgebungen aufgesetzt und Prototypen für die jeweiligen Bibliotheken implementiert.

Abstract

¹ 1 Exabyte = 10^{18} Byte oder eine Million Terabyte.

ERKLÄRUNG

Soweit meine Rechte berührt sind, erkläre ich mich einverstanden, dass die Master-Arbeit Angehörigen der Hochschule Emden-Leer, sowie der Beuth Hochschule Berlin für Studium / Lehre / Forschung uneingeschränkt zugänglich gemacht werden kann.

EIDESSTATTLICHE ERKLÄRUNG

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Master-Arbeit bis auf die offizielle Betreuung selbst und ohne fremde Hilfe angefertigt habe und die benutzten Quellen und Hilfsmittel vollständig angegeben sind.

Datum

Unterschrift

Inhaltsverzeichnis

1 Einführung	1
1.1 Ansätze für Big Data Analytics	3
1.2 Motivation für Apache Hadoop/Spark	4
1.3 Ziel und Aufbau dieser Arbeit	5
2 Allgemeine Grundlagen	7
2.1 Cluster Computing	7
2.2 Anwendungen für Big Data Analytics	8
2.3 Machine Learning	10
2.4 Das MapReduce-Paradigma	23
2.5 Anwendungen von Graphen	25
2.6 Zusammenfassung	25
3 Der Berkeley Data Analytics Stack (BDAS)	27
3.1 Die Schichten des BDAS	27
3.2 Apache Mesos	29
3.3 Hadoop Distributed File System (HDFS) und Tachyon	30
3.4 Apache Spark	31
3.5 Spark Streaming	32
3.6 GraphX	32
3.7 MLbase/MLLib	33

3.8	Spark SQL	34
3.9	Zusammenfassung	34
4	Alternative Implementierungen der Bibliotheken und Frameworks des BDAS	35
4.1	Alternative zu Spark: Apache Flink	35
4.2	Alternative zu Spark Streaming: Storm	37
4.3	Alternative zu MLLibs: H2O - Sparkling Water	38
4.4	Alternative zu MLLibs: Dato GraphLab Create TM	38
4.5	Zusammenfassung	38
5	Funktionsweise von Spark	39
5.1	Spark im Cluster	39
5.2	Das Konzept der Resilient Distributed Datasets	42
5.3	RDD Persistenz: Storage-Level von Spark	46
5.4	Die Spark-Console REPL	48
5.5	Die Spark APIs	48
5.6	Zusammenfassung	49
6	Architektur und Inbetriebnahme von lokalen Apache Spark Infrastrukturen	51
6.1	Prinzipieller Aufbau einer lokalen Spark Infrastruktur	52
6.2	Ausführungscontainer: Docker	54
6.3	Cluster Management: Mesos	56
6.4	Caching-Framework: Tachyon	57
6.5	Der eigentliche Kern: Apache Spark	57
6.6	Streaming-Framework: Spark Streaming	57
6.7	Abfrageschicht: Spark SQL	57
6.8	Machine Learning Algorithmen: MLLib	57
6.9	Graphenanwendungen: GraphX	57
6.10	Alternativimplementierung zu MLLibs: H2O	58

6.11 Alternativimplementierung zu Spark: Apache Flink	58
6.12 Zusammenfassung	58
7 Implementierung der Prototypen	59
7.1 Prototyp: Spark	59
7.2 Prototyp: MLLib	71
7.3 Prototyp: Spark Streaming	72
7.4 Prototyp: GraphX	72
7.5 Zusammenfassung	72
8 Evaluierung der Komponenten und Alternativen	73
8.1 Definition von Metriken für die Bibliotheken des BDAS	73
8.2 Beschreibung der Messverfahren	74
8.3 Beschreibung der Messumgebungen	75
8.4 Ergebnisse	77
8.5 Zusammenfassung	78
9 Schlussbetrachtung	79
9.1 Zusammenfassung	79
9.2 Ausblick	79
10 Verzeichnisse	81
Literaturverzeichnis	85
Internetquellen	91
Abbildungsverzeichnis	94
Tabellenverzeichnis	95
Quellenverzeichnis	97
A Zusätze	99
A.1 Übersicht der RDD Transformationen	99

Kapitel 1

Einführung

„Big Data is notable not because of its size, but because of its relationality to other data. Due to efforts to mine and aggregate data, Big Data is fundamentally networked. Its value comes from the patterns that can be derived by making connections between pieces of data, about an individual, about individuals in relation to others, about groups of people, or simply about the structure of information itself. [DB11]“

Big Data ist insbesondere in den letzten Jahren immer stärker in den verschiedensten Zusammenhängen in den allgemeinen Sprachgebrauch vorgedrungen und ist hier einem ständigen Bedeutungswandel ausgesetzt. Besonders in letzter Zeit wird dieses Thema auch verstärkt kontrovers diskutiert.

Der Begriff *Big Data* wurde vermutlich zum ersten Mal Ende des 20. Jahrhunderts von John R. Marshey, damals Chefwissenschaftler bei Silicon Graphics, im Rahmen einer Usenix-Konferenz öffentlich erwähnt [Tur14]. Mittlerweile zierte dieser Begriff gefühlt jedes zweite Cover von IT-Zeitschriften mit Business-Fokus und auch Manager und *Sales-Professionals* werten Ihre Produktpräsentationen gerne mit diesem Buzzword auf. Aber dieser Begriff ist nicht nur positiv assoziiert. Besonders seit Bekanntwerden der Tätigkeiten des Amerikanischen Auslandsgeheimdienstes weckt die Vorstellung des Datensammelns in großen Dimensionen auch Misstrauen.

Im Rahmen dieser Arbeit soll jedoch ausschließlich die technische Betrachtung und die exemplarische Darstellung von möglichen Anwendungsgebieten diskutiert werden.

Wie lässt sich der Begriff *Big Data* abgrenzen? Es existiert keine abschließend eindeutige Definition, jedoch gibt es einige Attribute, die sich in einem Großteil der Fachliteratur etabliert haben. Der Artikel aus dem O'Reilly Radar zum Thema [Dum14] fasst dies folgendermaßen zusammen:

„Big data is data that exceeds the processing capacity of conventional database systems. The data is too big, moves too fast, or doesn't fit the structures of your database architectures.“

Neben der reinen Menge spielt also offensichtlich auch die mangelnde oder fehlende Strukturierung und unter Umständen die Flüchtigkeit der Daten eine nicht unerhebliche Rolle. Dies können beispielsweise Daten aus Social-Media-Quellen sein, die aus allen möglichen verschiedenen Einzeldaten bestehen, Daten von Sensoren, die permanent überwacht werden müssen, oder Datenströme (Video, Audio, Bilder, Text), die nach einheitlichen Kriterien gefiltert werden sollen, um hier nur einige Beispiele zu nennen. Auch die temporäre Komponente ist ein Einsatzgebiet für *Big Data*, und auch hier ist wieder das Beispiel der Datenströme heranzuziehen.

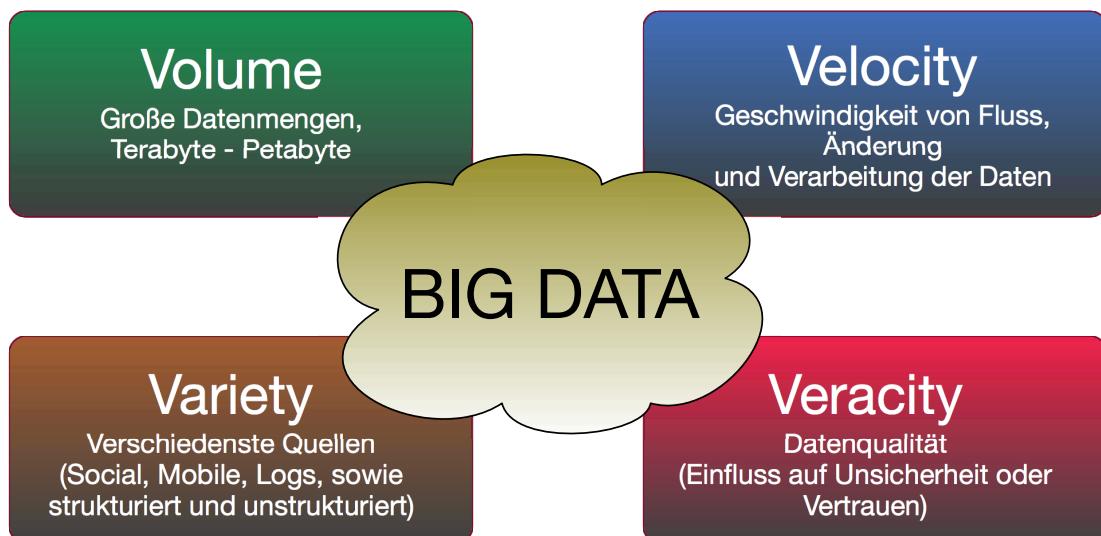


Abbildung 1.1: Darstellung der vier Säulen von Big Data: The Four V's of Big Data

Bei der Definition von *Big Data* werden laut [Tee14] auch immer wieder die „Four V's“¹ angeführt. Dies sind *Volume*, also die Datenmenge, *Variety*, die Datenvielfalt, *Velocity*, die Geschwindigkeit der Auswertung, sowie *Veracity*, die häufig stark schwankende Datenqualität. In Abbildung 1.1 wird dieser Zusammenhang dargestellt.

Die sinnvolle Analyse dieser Daten kann Unternehmen oder anderen Organisationen wichtige Informationen z.B. über Marktentwicklungen, bestimmte Kundenbedürfnisse, Epidemie-Ausbreitungen oder andere wichtige Sachverhalte liefern. Diese Analyse inklusive der dazu verwendeten Werkzeuge wird allgemein *Big Data Analytics* genannt.

¹ The four V's: Es existieren verschiedene Versionen der Säulen von Big Data. Deshalb war lange von nur drei Säulen die Rede (ohne Veracity), aber es existieren auch Definitionen von bis zu sieben V's (zusätzlich Variability, Visualisation, Value - Vergleich [McN14]). Mittlerweile hat sich die Darstellung der Four V's jedoch etabliert.

1.1 Ansätze für Big Data Analytics

Die Disziplin *Big Data Analytics* umfasst Methoden und Werkzeuge zur automatisierten oder interaktiven Erkennung und daraufhin auch Verwendung von bestimmten Mustern und Assoziationen. Dies sind unter anderem:

- Prediction-Models zur Vorhersage bestimmter Sachverhalte
- statistische Verfahren, wie beispielsweise *Logistic Regression* oder *k-means-Algorithmen*
- Optimierungs- und Filteralgorithmen
- Werkzeuge zum Datamining
- Textanalyse
- Bild- und Tonanalyse
- Datenstromanalysen

Nach dem BITKOM-Leitfaden [BIT14] besteht die Taxonomie der Big-Data-Technologien grundsätzlich aus vier Schichten:

- Daten-Haltung
- Daten-Zugriff
- Analytische Verarbeitung
- Visualisierung

Diese werden durch *Daten-Integration* und *Daten-Governance*, sowie Daten-Sicherheit flankiert, um den Weg von Rohdaten bis zu nutzbaren Erkenntnissen in existierende Standards einzubetten.

Zahlreiche Hersteller herkömmlicher relationaler Datenbanksysteme versuchen derzeit, ihre bestehenden Lösungen mit dem Label *Big Data* zu versehen und diese so weiterhin in diesen sich verändernden Marktsegmenten zu positionieren. Wenn *Big Data* jedoch jenseits der Datengröße definiert wird und auch unstrukturierte und temporäre Daten-Stacks oder -ströme zu verarbeiten oder zu analysieren sind, stoßen RDBMS² sehr schnell an ihre Grenzen. Doch auch was die Skalierbarkeit angeht, sind relationale Datenbanken meist nicht hinreichend flexibel [Lou14].

² RDBMS = Relational Database Management System, also ein relationales Datenbanksystem (im Gegensatz zu Objekt- oder Graphdatenbanken).

Für die Anforderungen an dedizierte Aufgaben im Bereich *Big-Data-Analytics* sind seit einigen Jahren einige *Frameworks* auf dem Markt, die in allen drei oben genannten Aspekten besser geeignet sind, als RDBMS. Der Ansatz ist hier primär, die Verarbeitung zu dezentralisieren, also auf unabhängige Knoten in einem Rechner-Cluster zu verteilen und nur Referenzen auf die Clusterknoten zentral zu verwalten.

Es existieren mittlerweile Lösungen am Markt, die speziell diese Aufgaben für derartige Aufgaben entwickelt wurden. Hier wären unter anderem Hadoop, Spark, HPCC, GPMR, Mincmeat, Sphere, Bashreduce und R3 zu nennen. Bis auf HPCC setzen alle eben genannten Implementierungen generell oder in Teilen auf das Programmiermodell MapReduce.

Der zweifellose De-facto-Standard in diesen Bereichen ist bereits seit einiger Zeit das Open-Source-Framework Apache Hadoop. Auf Hadoop basierend existieren etliche Derivate. Unter anderem sind hier Cloudera, Amazon Elastic MapReduce, Apache BigTop, Datameer, Apache Mahout, MapR und IBM PureData System zu nennen.

1.2 Motivation für Apache Hadoop/Spark

Anfang des 21. Jahrhunderts wurde das Bedürfnis für Möglichkeiten, sehr große Datenmengen effizient verarbeiten zu können, stetig größer. Nicht zuletzt durch die zu dieser Zeit exponentiell steigende Menge von Inhalten im World Wide Web und deren Indexierung durch Suchmaschinen wie Google. Davon motiviert wurde 2002 das Projekt *Nutch* mit dem Ziel gestartet, ein geeignetes *Such- und Crawlersystem* frei verfügbar zu machen. Die ersten Versuche skalierten sehr schlecht, bis Google 2003 die Funktionsweise ihres verteilten Dateisystems GFS (Google File System) veröffentlichte. Somit konnten die sehr großen Dateien, die durch die Indexierung entstanden, effizient auf verschiedene Knoten verteilt gespeichert werden und die Verwaltung dieser Knoten und Dateien aus dem eigentlichen Indexierungs- und Suchprozess ausgelagert werden.

Im Jahre 2004 publizierte Google den *MapReduce-Algorithmus*, der unter anderem die Indexierungs- und Analysefunktionen parallelisieren, delegieren und sinnvoll bündeln kann. In Nutch wurden daraufhin sämtliche wichtige Algorithmen auf MapReduce umgestellt, nachdem zuvor auch GFS unter dem Namen NDFS (Nutch Distributed File System) integriert wurde. Die möglichen Anwendungsgebiete von Nutch waren damit auch weit über das reine Suchen und Indexieren von Webseiten hinaus gewachsen. 2006 wurde aus Nutch ein Unterprojekt mit dem Namen Hadoop ausgegliedert, das im Jahre 2008 zum *Apache Top-Level-Project* ernannt wurde. Zu dieser Zeit nutzten bereits Firmen wie Yahoo!, Facebook oder die New York Times Hadoop. Ein exemplarischer Anwendungsfall bei der NY Times war, mit Hilfe der Hadoop-basierten EC2-Cloud von Amazon ca. vier Terabyte gescannter Archivdateien in PDF-Dateien umzuwandeln und dies in weniger als 24 Stunden auf 100 Knoten. Auch beim Sortieren von sehr großen Datenmengen stellten Hadoop-basierte Systeme nach und nach sämtliche Rekorde

ein [Whi13].

Hadoop und Hadoop-basierte System gelten mittlerweile als Industriestandard für Big-Data-Analytics-Anwendungen. Jedoch ist Hadoop nicht für alle Anwendungsgebiete gleichermaßen geeignet. Aufgrund der Charakterisierung der Paradigmen für Big Data Analytics im Paper „Frontiers in Massive Data Analysis“ der National Academic Press [Cou13], lassen sich die Einsatzgebiete und Schwächen für Hadoop ermitteln [Agn14].

So lassen sich mit Hadoop einfache statistische Aufgabenstellungen sehr gut umsetzen. Dazu gehören Mittelwert, Median, Varianz und allgemein abzählende sowie ordnende Statistikaufgaben. Dies sind in der Regel Anwendungen mit einer Laufzeitkomplexität von $O(n)$ für n Betrachtungswerte. Sie sind meist auch sehr gut parallelisierbar und somit sehr gut für Hadoop geeignet.

Bei Hadoop werden sämtliche Datenstrukturen und Zwischenergebnisse von Berechnungen und Transformationen im Dateisystem persistiert. Für linear-algebraische Berechnungen aus dem Bereich Machine-Learning (lineare Regression, Eigenwertproblem, Hauptkomponentenanalyse), generalisierte n-Körper-Probleme³ (mit einer Komplexität von $O(n^2)$ oder $O(n^3)$), Graphentheorie, Optimierungsprobleme (Verlust-, Kosten- oder Energiefunktionen, sowie Integrations- und Ausrichtungsfunktionen ist *Hadoop* deshalb nur in jeweils einfacher Problemausprägung einsetzbar. Auch für Interaktive Abfragen und für Streaming-Anwendungen ist *Hadoop* nur bedingt geeignet, da es ursprünglich für die *Batch-Verarbeitung* entwickelt wurde.

Aus diesem Grund wurde am *AMPLab* der University of California in Berkeley nach Alternativen geforscht, die auc. Das Ergebnis ist *Spark*, mittlerweile *Apache Top-Level-Projekt* und dazu geeignet, die Nachfolge von Hadoop als *Big-Data-Analytics-Framework* anzutreten.

1.3 Ziel und Aufbau dieser Arbeit

Die vorliegende Arbeit beschäftigt sich mit *Apache Spark* und dem dazugehörigen Ökosystem bestehend aus einem Applikationsstack mit verschiedenen Bibliotheken, dem *Berkeley Data Analytics Stack (BDAS)*. Aufgrund der Konzepte von Spark ist dieses unter anderem prädestiniert für Machine-Learning-Anwendungen. Diese bilden, neben der eigentlichen Kernimplementierung von Spark, einen zentralen Teil dieser Arbeit. Auf dem Markt befindet sich bislang noch verhältnismäßig wenig Literatur zu diesem Thema und wenn, dann werden zumeist Teilespekte für bestimmte Anwendungsbereiche gekapselt betrachtet.

Diese Arbeit soll, nachdem einige Grundlagen zum Thema *Big Data Analytics* im Allgemeinen diskutiert werden, zunächst einen ganzheitlichen Überblick über den BDAS bieten. Hier werden

³ n-Körper-Probleme beschreiben unter Anderem das Verhalten von Molekülen bis hin zum Verhalten von Planeten oder Galaxien zueinander. Ziel ist die Berechnung von Positionen und Geschwindigkeiten der einzelnen Körper.

die einzelnen Schichten des Stacks beschrieben und gegebenenfalls Alternativlösungen zu den jeweiligen Implementierungen vorgestellt.

Insbesondere wird in den darauffolgenden Grundlagenkapiteln die Machine-Learning-Bibliothek *MLLib* betrachtet. Um ein Verständnis für die Funktionen dieser Bibliothek zu vermitteln, wird zunächst eine Einführung in die Grundlagen in die Themengebiete Data-Mining und Machine-Learning vermittelt. Anschließend werden die elementaren Machine-Learning-Algorithmen vorgestellt, welche in *MLLib* implementiert sind. Auch die übrigen Elemente des BDAS wie das Caching-Frameworks Tachyon, sowie die Streaming-Bibliothek Spark Streaming und die Graphenanwendung GraphX werden vorgestellt. Anschließend werden die APIs der einzelnen Bibliotheken gezeigt. Diese bilden die Grundlage für eigene Anwendungen mit Spark.

Des weiteren wird die BDAS-Bibliothek *MLLib* mit der Alternativbibliothek *H2O* verglichen. Außerdem wird ein Vergleich zwischen der Kernimplementierung von Spark wird mit dem neueren *Big Data Analytics Framework Apache Flink*⁴ vorgestellt.

Im letzten Teil dieser Arbeit wird der Aufbau von lokalen BDAS-Stacks für verschiedene Anwendungsbereiche gezeigt. Darauf folgt die Beschreibung von Prototypen die zum Test der Spark-Infrastruktur im Rahmen dieser Arbeit implementiert wurden. Zum Schluss werden für die jeweiligen Anwendungsbereiche des BDAS Metriken definiert, welche für Messungen der Prototypen benutzt werden.

In einem Ausblick werden die Erkenntnisse aggregiert und mögliche weitere Entwicklungen prognostiziert.

⁴ Entwickelt von der Technischen Universität Berlin zunächst unter dem Namen *Stratosphere* und mittlerweile (Stand Ende 2014) *Apache Incubator* Projekt

Kapitel 2

Allgemeine Grundlagen

Das nachfolgende Kapitel behandelt die Grundlagen, die für ein Verständnis der Anwendungsbereiche von Apache Spark, dem Berkeley Data Analytics Stack und im Allgemeinen des Themenkomplexes Big Data Analytics und insbesondere für Machine-Learning-Anwendungen nötig sind. Im ersten Unterkapitel werden die grundsätzlichen Eigenschaften eines verteilten Systems beschrieben um die Basis für die in der Arbeit beschriebenen Besonderheiten von Verarbeitungen im Clusterbetrieb zu legen. Hier wird ein exemplarischer Clusteraufbau skizziert, Probleme mit Concurrency und Netzwerkverkehr beschrieben und welche Möglichkeiten es hier gibt. Im darauf folgenden Unterkapitel werden grundlegende Problemstellungen und Technologien beschrieben, die im Rahmen von Big Data Analytics im Allgemeinen vorkommen. Unter anderem werden hier Grundlagen und Begriffe aus den Themengebieten Anwendungen von Big Data Analytics, Machine Learning, Klassifikation, Vorhersagen, statistische Analysen, Graph-Suchen und Streaming-Frameworks in erklärt. Besonders die Algorithmen, die in den Machine-Learning-Implementierungen MLibs und H2O zum Einsatz kommen, werden hier detaillierter vorgestellt. In einer Zusammenfassung werde diese Grundlagen nochmals auf einen Blick dargestellt.

2.1 Cluster Computing

Die Nachfrage nach immer mehr Rechenleistung hat in den letzten Jahren dazu geführt, dass verstärkt Rechnercluster eingesetzt werden. Alternativ gibt es den Ansatz, Mainframes¹ mit immer mehr Rechenleistung auszustatten, diese jedoch ausdrücklich autonom zu betreiben². Je nach Aufgabenspektrum ist die eine oder andere Infrastruktur besser geeignet. In der Regel wird ein geclustertes System dort eingesetzt, wo hohe Verfügbarkeit oder gut parallelisierbare

¹ Unter Mainframe wird hier ein sehr leistungsfähiges Rechnersystem verstanden, das einen oder beliebig viele Prozessoren in einer physischen Einheit, also einem logischen Mainboard verbindet.

² In diesem Kontext kann durchaus ein Failover-Cluster vorhanden sein, also eine Mainframe wird zur Ausfallsicherheit repliziert. Dies wird an dieser Stelle jedoch nicht als Cluster im eigentlichen Sinn bezeichnet.

Aufgaben vorherrschen. Bei netzwerkintensiven Aufgaben, wie z.B. als Webserver oder Datenbanksystem sollten in der Regel besser Installationen auf einem autonomen System eingesetzt werden [TR10].

Ein Rechner-Cluster besteht in der Regel aus mehr oder weniger eng miteinander verbundenen Computern, wobei hier im Gegensatz zu Mainframes jeder Rechner über eigene Ressourcen wie Hauptspeicher, Massenspeicher, etc. verfügt. Ein Cluster, bzw. ein Verteiltes System ist nach Andrew S. Tanenbaum [AST07] folgendermaßen definiert:

„A distributed system is a collection of independent computers that appears to its users as a single coherent system.“

Bei der Verwendung eines Clusters sind einige Besonderheiten zu beachten, die bei der Ausführung auf gewöhnlichen Systemen nicht ins Gewicht fallen [AST07]. Unter Anderem sind die Tasks so gestalten, dass möglichst wenig Wartezeit durch Abhängigkeiten entsteht und diese möglichst autonom verarbeitet werden können. Außerdem muss beachtet werden, dass die einzelnen Knoten eines Clusters über Messaging-Mechanismen miteinander kommunizieren und dies insbesondere hohe Anforderungen an die Netzwerkinfrastruktur stellt. Eine typische Clustertopologie besteht aus mehreren Worker-Knoten und einem Masterknoten. Der Masterknoten delegiert die Tasks an die einzelnen Worker-Knoten und stellt das gesamte Cluster nach außen hin als ein geschlossenes System dar. Sämtliche Kommunikation mit dem Cluster findet grundsätzlich nur über den Masterknoten statt.

2.2 Anwendungen für Big Data Analytics

Im folgenden Unterkapitel werden exemplarisch einige Anwendungsfälle für *Big Data Analytics* (auch *Data Mining*) dargestellt, um zu klären, für welche Einsatzbereiche *Frameworks* wie *Apache Spark* und die darauf aufbauenden Bibliotheken in der Praxis benötigt werden.

Laut Arvind Sathi [Sat12] zeichnet sich *Big Data* unter Anderem durch ein mögliches Vorkommen von unstrukturierten Daten aus. Die Autoren Chakraborty und Pagolu gehen in ihrem Artikel [DGC14] davon aus, dass mittlerweile mehr als 80% der gesamten Daten im digitalen Raum in unstrukturierter Form vorliegen. In der Vergangenheit mussten für Analysetätigkeiten in aller Regel strukturierte Datensätze vorliegen. Grundsätzlich ist es mit den gängigen *Data Analytics Frameworks* nach wie vor möglich, beispielsweise quantitative Analysen auf strukturierten Datensätzen durchzuführen oder unstrukturierte Datensätze nachträglich zu strukturieren, um wiederum quantitative Analysen darauf anwenden zu können.

Das Potential dieser *Frameworks* zeigt sich jedoch dann in vollem Umfang, wenn auf unstrukturierten Daten diverse Analysemethoden oder Verarbeitungen angewendet werden. In jüngerer Vergangenheit ist das Aufkommen unstrukturierter Daten, wie bereits erwähnt, erheblich gestiegen. Dies wird nicht zuletzt durch die massive Verbreitung von Sensoren aller

Art verursacht. Dies können *Logdaten*, Bewegungsdaten, Sensorwerte zur Überwachung von technischen Einrichtungen, Messwerte aus Wetterstationen und unzähligen weiteren Quellen sein. Auch viele Internetanwendungen, besonders wenn es sich um laufende Datenströme handelt, verursachen erhebliche Datenmengen, die entweder *persistiert* oder sogar zur Laufzeit analysiert werden können.

*Data Mining*³ ist laut [JC14] ein analytischer Prozess mit dem Zweck, große Datenmengen nach konsistenten Mustern oder systematischen Beziehungen zu untersuchen. Die Ergebnisse werden in der Regel validiert, in dem gefundene Muster oder Ähnlichkeiten auf einer Teilmenge der ermittelten Daten angewendet werden. Ein weiteres Ziel von *Data-Mining-Prozessen* sind Vorhersagen von Ereignissen mittels geeigneter Algorithmen (Vergleich [JC14] und 2.3).

Der Prozess des Data Mining setzt sich nach [UF96] im Wesentlichen aus einem oder mehreren der folgenden Aufgabenbereiche zusammen:

- **Klassenbeschreibung:** Eine knappe Beschreibung der Charakterisierung der Datensätze, um sie eindeutig von anderen Daten unterscheiden zu können.
- **Assoziation:** Die Untersuchung der Daten nach assoziativen Verbindungen oder Korrelationen zwischen einzelnen Daten oder Datengruppen.
- **Klassifizierung:** Hier wird ein definierter Satz von Trainingsdaten⁴ analysiert und anhand deren Beschaffenheit ein Modell generiert. Durch die Klassifizierung werden Entscheidungsbäume (Vergleich Kapitel 2.3) oder Klassifizierungsregeln generiert, die schließlich für die Klassifizierung folgender Daten verwendet werden [SW97].
- **Vorhersage:** Die Vorhersage bezieht sich auf mögliche Werte von nicht-vorhandenen Daten oder Datenspektren, die wiederum durch *Approximation* einer Funktion mittels Beispielen durchgeführt wird [JC14]. Dies sind ebenfalls Trainingsdaten, die aus Datensätzen mit den dazugehörigen berechneten Funktionswerten bestehen.
- **Cluster-Analyse:** Diese dient dazu, *Cluster* innerhalb von Datensätzen zu ermitteln. Dies sind Daten, die definierte Ähnlichkeiten zueinander aufweisen.
- **Zeitreihenanalysen:** Hier werden in der Regel große Mengen an Zeitreihendaten analysiert, um nach Ähnlichkeiten oder Mustern innerhalb der Daten zu suchen.

³ Ein Großteil der Literatur verwendet die Begriffe *Big Data Analytics* und *Data Mining* synonym. *Machine Learning* wird jedoch von *Data Mining* abgegrenzt, da letzteres eine explorative Datenanalyse darstellt.

⁴ Trainingsdaten können beispielsweise Daten sein, die bereits im Vorfeld manuell klassifiziert wurden, deren Klassenzugehörigkeit also bekannt ist.

2.3 Machine Learning

„Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task (or tasks drawn from a population of similar tasks) more effectively the next time.“ [HS83]

Unter *Machine Learning* wird ein interdisziplinärer Teilbereich der Informatik und der Statistik verstanden. Ziel ist die Erstellung von Algorithmen, die in der Lage sind, selbstständig auf Grund von Daten iterativ zu lernen gemäß der oben zitierten Definition. Um dies zu erreichen, erstellen diese Algorithmen basierend auf den jeweiligen Eingabedaten Modelle, die Entscheidungen oder Vorhersagen treffen können [GJ13]. In den Abbildungen 2.1 und 2.2 wird dies veranschaulicht.

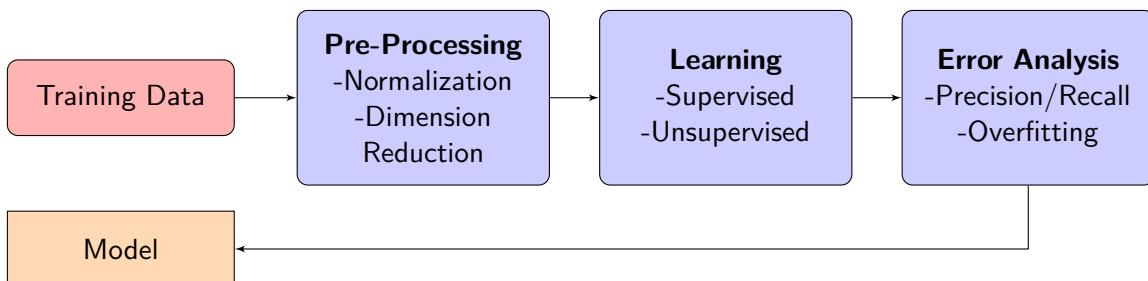


Abbildung 2.1: Der Machine-Learning-Prozess: Phase 1 - Lernphase

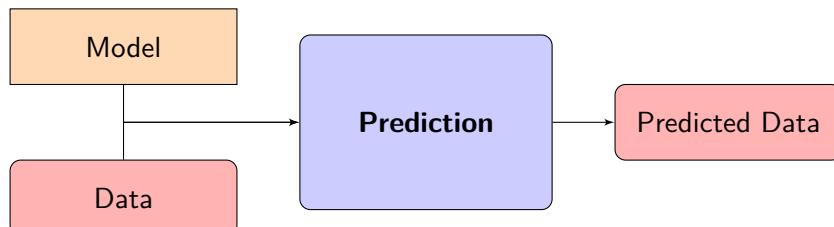


Abbildung 2.2: Der Machine-Learning-Prozess: Phase 2 - Prediction-Phase (Vorhersage)

Das erste Diagramm 2.1 zeigt den Lernprozess. Zunächst werden definierte Trainingsdaten einem *Pre-Processing*⁵ zugeführt (Vergleich Unterkapitel 2.3.1).

Danach folgt der eigentliche Lernprozess, der mit oder ohne *Überwachung*⁶, als *Minimalisierungsfunktion* oder mit anderen Lernalgorithmen durchgeführt wird (Vergleich Unterpunkte Clusterverfahren und Klassifikationsalgorithmen). Der Lernprozess kann entweder mit einem bestimmten Lernalgorithmus durchgeführt werden, oder mit sogenannten *Ensemble-Learning-Methods*. Dies ist die Zusammenfassung von unterschiedlichen Lernalgorithmen zu einem Lernprozess, um die Vorhersageleistung zu steigern. Nach dem Lernprozess werden das erzeugte

⁵ Das Pre-Processing kann aus *Normalisierung*, *Dimensionsreduktion*, *Bildverarbeitung* oder anderen Vorarbeiten bestehen.

⁶ Beim überwachten Lernen (Supervised Learning) liegen der Analyse vorher angelegte Trainingsdaten zugrunde, das unüberwachte Lernen (Unsupervised Learning) erzeugt vorher gänzlich unbekannte Modelle aus gefundenen Mustern [Gha15]

Modell einem Validierungsprozess zugeführt. Dies können *Precision/Recall*⁷, *Overfitting*⁸, oder andere Validierungsfunktionen sein. Am Ende der Prozesse entsteht ein Modell, das für verschiedene Aufgaben eingesetzt werden kann.

In Abbildung 2.2 wird das erzeugte Modell zusammen mit produktiven Daten genutzt, um mittels geeigneter Vorhersagealgorithmen (Vergleich Unterkapitel 2.3.1) bisher noch nicht vorhandene Datensätze erzeugen zu können.

Die Hauptanwendungsgebiete für *Machine Learning* sind sämtliche Bereiche, in denen eine Anwendung von strikten, regelbasierten Algorithmen nicht in Frage kommt [SW97]. Beispiele für diese Bereiche sind laut [Bis06] Suchmaschinen, Sprach- und Musikerkennung, Handschriftenerkennung, Spamfilter, Umgebungserkennungen und viele mehr.

In den vergangenen Jahren werden besonders in Anwendungsfällen der *Artificial Intelligence* die sogenannten *Deep Learning Algorithmen* als äußerst vielversprechender Ansatz gesehen [Ben09]. Dieses basieren auf überwachten oder unüberwachten Lernmethoden, die nichtlinear miteinander gekoppelt werden. DL-Architekturen sind in der Regel in Schichten aufgebaut, wobei eine untere Schicht die von der oberen Schicht klassifizierten oder vorhergesagten Daten zu entsprechenden Weiterverarbeitung erhält.

Besonders für Anwendungen der Bild und Tonerkennung kommen diese Verfahren mittlerweile in Form verschiedener Algorithmen und Architekturen zur Anwendung. In Abbildung 2.3, entnommen aus dem Paper *Learning Deep Architectures for AI* [Ben09], werden die verschiedenen Iterationen einer Deep-Learning-Anwendung am Beispiel einer Bilderkennung visualisiert. Das Bild wird hier nach und nach von konkreten Attributen wie Kanten, Helligkeitswerten, Farben, lokalen Formen, Teilen von Objekten, etc. in jedem Layer hin zu abstrakteren Repräsentationen transformiert. Auf der höchsten Abstraktionsstufe sollte das Deep-Learning-System erkannt haben, dass es sich um die Darstellung eines sitzenden Mannes handelt.

In der praktischen Anwendung würde eine solche Interpretation ohne explizit gelernten Kontext noch nicht funktionieren, da für die jeweilige Stufe keine richtige Interpretation vorhergesagt werden kann. Hier gehen Forschungsvorhaben in Richtung der Anreicherung der Algorithmen mit Konzepten aus der Linguistik.

Es existieren diverse Algorithmen, Architekturen und Definitionen von Deep-Learning. Für die Anwendung der Algorithmen ist eine detaillierte Kenntnis dieser meist nicht nötig. Dies ist auch einer der Kritikpunkte, der in Verbindung mit Deep-Learning genannt wird. Die Verfahren präsentieren sich für die Anwender in der Regel als intransparente Black-Box. Für eine Vorstellung der Deep-Learning-Ansätze wird an dieser Stelle an entsprechende Sekundärliteratur verwiesen.

⁷ *Precision*: Wie viele der ausgewählten Datensätze sind relevant. *Recall*: Wieviele relevante Datensätze wurden ausgewählt.

⁸ *Overfitting* bezeichnet einen Zustand, in dem bekannte Daten (Trainingsdaten) von einem Algorithmus generell akkurater erkannt werden, als Daten, die von dem Algorithmus erkannt werden sollen (Predictions).

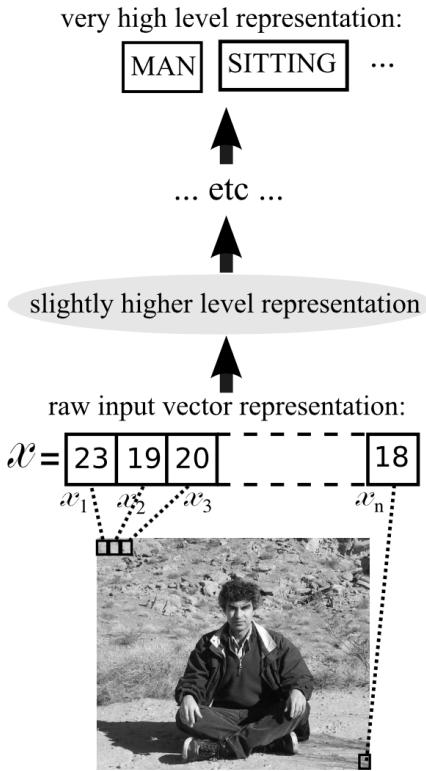


Abbildung 2.3: Exemplarische Darstellung eines Bilderkennungsprozesses mittels Deep Learning. Entnommen aus [Ben09]

2.3.1 Klassifizierung von Daten

Im Gegensatz zu herkömmlichen *Business-Intelligence-Anwendungen*⁹ zeichnen sich Big-Data-Anwendungen in erster Linie durch Unterschiede in der Herangehensweise und in der Formulierung der primären Fragestellungen aus. In der Vergangenheit wurden zu Beginn einer Analyse zunächst Problemstellungen formuliert und aufgrund dessen Prozesse und vor allen Dingen die zu sammelnden Daten definiert [NG12]. Mit der steigenden Etablierung von Big-Data-Anwendungen hat sich dieser Ansatz gewandelt. Hier werden zunächst sämtliche anfallenden Daten gespeichert. Auf diese Datensätze, deren Größe in relativ kurzer Zeit massiv anwachsen kann, werden erweiterte Analyseprozesse¹⁰ angewendet, die unter anderem auch Vorhersagen erlauben [Rus11]. So liegen also bei Big-Data-Analytics große Datenmengen vor, ohne dass das Ziel der Analyse im Vorfeld bekannt ist.

⁹ Laut [NG12] versteht man unter *Business Intelligence (BI)* die Sammlung, Auswertung und Darstellung von Daten anhand definierter Prozesse.

¹⁰ Unter dem Begriff *Advanced Analytics* sind verschiedene Werkzeugtypen aus der vorhersagenden Analyse (*Predictive Analytics*), aus dem Data Mining, aus der Statistik, der Künstlichen Intelligenz, der Sprachverarbeitung und weiteren Disziplinen zusammengefasst [Rus11].

Cluster-Verfahren

Nun gibt es verschiedene Herangehensweisen, wie sich relevante Fragestellungen aus den vorliegenden Daten ableiten lassen. Ein sinnvoller Ansatz besteht darin, sogenannte Cluster innerhalb der Daten zu ermitteln, also ähnliche Datensätze zu entsprechenden Gruppen zusammenzufassen. Aus diesen Gruppen werden Klassen gebildet, in dem ihnen aussagekräftige Namen zugeordnet werden [JC14]. Cluster werden beispielsweise benötigt, um Kunden nach bestimmten Interessen zu gruppieren, um Zeichen auf Ähnlichkeiten für die automatischen Zeichenerkennung zu untersuchen, oder Bilder auf vergleichbare Bildpunktanordnungen oder Farbspektren, also generell überall, wo nach Ähnlichkeiten gesucht wird.

Ein Cluster zeichnet sich dadurch aus, dass die Objekte innerhalb eines Clusters möglichst ähnlich sind, die Cluster untereinander jedoch möglichst unähnliche Inhalte besitzen. Um Ähnlichkeiten zwischen Datensätzen oder Clustern feststellen zu können, bedarf es einer Distanzfunktion, zur Ermittlung und Sicherstellung der Clustergüte einer Qualitätsfunktion (Vergleich [JC14]).

In Abbildung 2.4 wird stellvertretend das Cluster-Verfahren *k-Means* in sechs Iterationsschritten zu Cluster-Bildung gezeigt. Dieses und weitere Cluster-Verfahren werden in diesem Unterkapitel nach der Vorstellung einiger relevanter Distanzfunktionen genauer beschrieben.

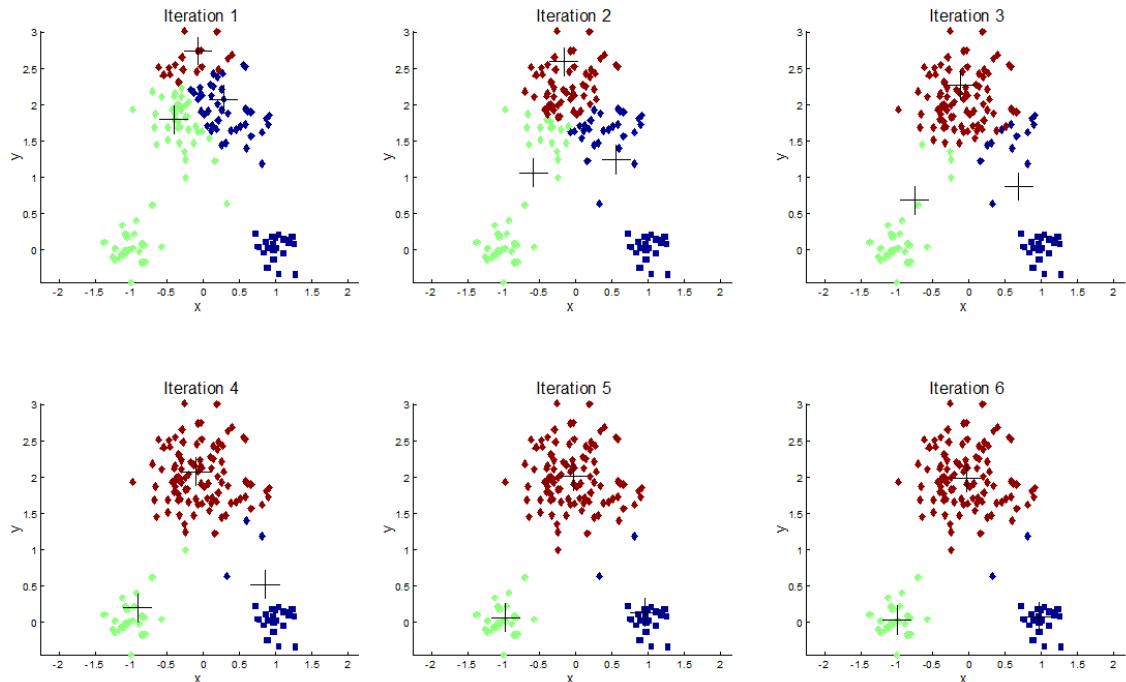


Abbildung 2.4: Cluster-Verfahren mit verschiedenen Iterationsschritten am Beispiel *k-Means*. Entnommen aus [Ren15]

Laut [JC14] sind für Distanz- und Qualitätsfunktionen folgende Annahmen zu treffen:

Vorbedingungen

X (Instanzenraum)

$E \subseteq X$ (Instanzenmenge)

$X \times X \rightarrow \mathbb{R}^+$ (Abstandsfunktion)

$2^{2^x} \rightarrow \mathbb{R}$ (Qualitätsfunktion)

Gesucht wird eine Clustermenge $C = \{C_1, \dots, C_k\}$ mit folgenden Eigenschaften:

$$C_i \subseteq E$$

$$\text{quality}(C) \rightarrow \max$$

$$C_i \cap C_j = \emptyset$$

$$C_1 \cup \dots \cup C_k = E$$

Die Abstandsfunktion dist , die gegebene Objekte so in Teilmengen zerlegt, dass der Abstand der Objekte innerhalb einer Teilmenge (Cluster) kleiner ist, als der Abstand zu Objekten anderer Teilmengen:

$$\forall C_i, C_j \in C (i \neq j) : \forall x_k, x_l \in C_i, x_m \in C_j : \text{dist}(x_k, x_l) < \text{dist}(x_k, x_m)$$

Die Qualitätsfunktion quality beschreibt die Qualität des Clusterings basierend auf der Abstandsfunktion dist :

$$\text{quality}(C = \{C_1 \subseteq E, \dots, C_k \subseteq E\}) \rightarrow \mathbb{R}$$

Exemplarisch werden im Folgenden einige Distanzfunktionen dargestellt, die für die Clusteranalyse wichtig sind (Vergleich [CV14], [Bro98]):

- **Hamming-Distanz:** Hier werden die Vektorelemente der Position i zweier Vektoren miteinander verglichen. Wenn sich diese unterscheiden, wird die Distanz auf den Wert 1 gesetzt, bei gleichen Werten 0. Nachdem alle Elemente der Vektoren miteinander verglichen wurden, werden diese summiert. Ein Vorteil der Hamming-Distanz ist, dass sich Abstände zwischen metrischen, ordinalen und sogar nominalen Daten berechnen lassen. Der Nachteil dieser Distanzfunktion ist, dass Abstandsverhältnisse nicht erkannt werden, also ist beispielsweise der Abstand zwischen 2 und 3 für diese Funktion gleichwertig zum Abstand von 2000 zu 2100.

$$dist_H(x, y) = \sum_i (x_i \neq y_i)$$

- **Manhattan-Distanz:** Diese Distanzfunktion hat ihren Namen aufgrund der blockweisen Anordnung der Straßenzüge in Manhattan. Distanzen werden, wie auf einem Schachbrett, durch vertikale und horizontale Bewegungen und Richtungsänderungen beschrieben. Um die Ergebnisse der verschiedenen Distanzfunktionen untereinander vergleichbar zu machen, wird der Betrag der Distanzwerte verwendet. Die Manhattan-Distanz ist nur auf metrische Daten anwendbar. Diese Distanzfunktion ist auch unter dem Namen *Block-Distanz* bekannt.

$$dist_M(x, y) = \sum_i |x_i - y_i|$$

- **Euklidische Distanz:** Hier wird der direkte Abstand zwischen zwei Vektoren beschrieben, also zweier Punkte im n -dimensionalen Raum. Diese Funktion lässt sich nur auf metrische Daten anwenden. Auch hier wird durch Quadrieren und der Quadratwurzel der Differenzen eine Normalisierung erreicht, die die Distanzwerte vergleichbar macht.

$$dist_E(x, y) = \sqrt{\sum_i (x_i - y_i)^2}$$

- **Tschebyscheff-Distanz:** Auch hier wird der Abstand zwischen zwei Punkten im n -dimensionalen Raum betrachtet. Im Unterschied zu den beiden vorhergehenden Verfahren wird hier jedoch der größtmögliche absolute Abstand zwischen allen Attributen ermittelt. Deshalb wird diese Distanzfunktion auch als *Maximum-Distanz* bezeichnet.

$$dist_T(x, y) = \max_i (|x_i - y_i|)$$

- **Minkowski-Distanz:** Diese Distanz-Funktion ist ähnlich zur Euklidischen Distanz. Allerdings entspricht hier der Abstand der n -ten Wurzel der Summe der n -ten Potenzen der Differenzen. Auch diese Funktion lässt sich nur auf metrische Daten anwenden.

$$dist_K(x, y) = \sqrt[n]{\sum_i (x_i - y_i)^n}$$

Darüber hinaus existieren noch zahlreiche weitere Distanzfunktionen, die je nach Einsatzzweck besser oder schlechter geeignet sein können, als die dargestellten.

Die Clusteranalyse zählt zu den *unüberwachten Lernalgorithmen (Unsupervised Learning)*. Zoubin Ghahramani beschreibt dies in seinem Artikel *Unsupervised Learning* [Gha15] wie folgt:

„[...]in unsupervised learning the machine simply receives inputs x_1, x_2, \dots , but obtains neither supervised target outputs, nor rewards from its environment. It may seem somewhat mysterious to imagine what the machine could possibly learn given that it doesn't get any feedback from its environment. However, it is possible to develop of formal framework for unsupervised learning based on the notion that the machine's goal is to build representations of the input that can be used for decision making, predicting future inputs, efficiently communicating the inputs to another machine, etc. In a sense, unsupervised learning can be thought of as finding patterns in the data above and beyond what would be considered pure unstructured noise.“

Prinzipiell wird zwischen vier verschiedenen Clusterarten unterschieden [JC14]:

- **Partitionierendes Clustering:** Eine Menge von Datenobjekten wird in k Cluster, die um einen Medoid¹¹ oder einen Centroid¹² gebildet werden, zerlegt (Vergleich [Mir15]).
- **Hierarchisches Clustering:** Hier werden die Cluster hierarchisch aufgebaut, indem jeweils die Cluster mit der größten Ähnlichkeit, also mit der geringsten Distanz, miteinander vereinigt werden. Aus diesen vereinigten Clustern können in der Hierarchie übergeordnete Ebenen entstehen. Beim *Agglomerativen Clustering* wird mit den einzelnen Datenobjekten selbst begonnen, in dem jedes Cluster aus genau einem Datenobjekt besteht. Die jeweils ähnlichsten Cluster werden vereinigt und somit zu einer neuen Hierarchieebene. Dies geschieht so lange, bis in der obersten Ebene ein Cluster mit allen Datenobjekten entstanden ist. Beim *Divisiven Clustering* wird die Hierarchie in umgekehrter Reihenfolge aufgebaut, also zunächst werden alle Datenobjekte zu einem großen Cluster zusammengenommen und dieser dann iterativ anhand der Ähnlichkeiten der Datenobjekte untereinander geteilt.
- **Dichtebasierter Clustering:** Die Cluster werden so gebildet, dass in der Umgebung eines Datenobjektes innerhalb eines Clusters möglichst viele weitere Datenobjekte liegen, die Dichte also besonders hoch ist. Die geforderte Dichte wird über Schwellenwerte definiert.
- **Clustering mit Neuronalen Netzen:** Auch über neuronale Netze können Cluster gebildet werden. Die Netze werden in diesem Fall nicht überwacht trainiert (Vergleich Einleitung Kapitel 2.3) und nutzen das Verfahren des *Wettbewerbslernens* (Vergleich [JC14], beispielsweise *Voronoi-Mengen*). Das Neuron, welches dem Eingabewert (dem Datenobjekt) am ähnlichsten ist, ist im jeweiligen Wettbewerb im Vorteil.

¹¹ Medoide sind Stellvertreterobjekte eines Clusters, deren durchschnittliche Ähnlichkeit zu allen Datenobjekten des Clusters maximal ist. Ein Medoid ist immer ein Datensatz aus dem Cluster.

¹² Centroide sind Vektoren der Mittelwerte der Attribute aus den Datenobjekten. Im zweidimensionalen Raum entspricht dies der Mittelwerte der x - und y -Koordinaten aller Datenobjekte.

Zu den Cluster-Verfahren zählen laut [WKh12] und [JC14] unter Anderem folgende Algorithmen:

- **klassischer k-Means-Algorithmus (partitionierendes Verfahren):** Wie bei allen partitionierenden Verfahren werden zunächst die Clusterzahl k und je Cluster ein Zentrum definiert. Die Zentrumsdefinition, und somit die Clusterbildung findet zufällig statt. Die Repräsentation des Clusters erfolgt durch den Centroid, also den Schwerpunkt. Eine Fehlerfunktion wird iterativ minimiert, indem die euklidischen quadrierten Abstände der Inhaltsobjekte zu den Centroiden hin verschoben werden. Nach jedem Iterationsschritt werden durch Mittelwertbildung die Centroiden neu definiert. Dieses Verfahren wird so lange iterativ durchgeführt, bis kein Datenobjekt mehr sein Cluster wechselt. Der k-Means-Algorithmus setzt metrische Werte¹³ voraus. Die Umwandlung von ordinalen Daten in metrische ist meist einfach umzusetzen, nominale Daten können mittels *Binarisierung* in numerische Werte gewandelt werden. Alternativ kann auch ein modifiziertes *k-Means-Verfahren* eingesetzt werden (Vergleich [JC14], Kapitel 8). In Abbildung 2.4 wird die Clusterbildung mittels k-Means-Algorithmus visualisiert.
- **k-Medoid-Verfahren (partitionierende Verfahren):** Im Unterschied zum k-Means-Algorithmus wird hier nicht der Centroid, sondern der Medoid als Stellvertreter für ein Cluster definiert. Ein Medoid muss immer ein Element der Datenobjekte sein. Durch Vertauschung wird nun iterativ nach qualitativ hochwertigeren Medoiden gesucht, also solchen, deren mittlere Distanz zu den anderen Datenobjekten im Cluster geringer ist. Dies wird so lange durchgeführt, bis kein Tausch der Medoiden mehr stattfindet, also jedes Cluster seinen qualitativ hochwertigsten Medoid besitzt. Die beiden wichtigsten k-Medoid-Verfahren sind PAM (Partitoning Around Medoids) und CLARANS (Clustering Large Applications based on RANdomized Search) (Vergleich [Ng02]). Bei PAM wird jeweils nach dem besten neuen Mediod mit Hilfe einer Qualitätsverbesserung durch grafische Repräsentation der Datenobjektverteilung und -häufigkeit gesucht. Allerdings ist PAM nur für kleine Datenmengen zu verwenden. CLARANS ist für große Datenmengen besser geeignet, da hier nicht sämtliche Datenobjekte durchsucht werden. Statt dessen beschränkt sich die Suche auf Teilmengen.
- **k-Median-Algorithmus (partitionierendes Verfahren)** Im Gegensatz zu den k-Means-Verfahren wird beim k-Median nicht mit der Euklidischen Distanz der Abstand zu den Zentren ermittelt, sondern die Summen der Manhattan-Distanzen (1-Norm-Distanzen) werden iterativ minimiert [BJA06].

¹³ *Metrisch*: Merkmal, das aus einer Zahl besteht und Dimension, sowie Nullpunkt besitzt (z.B. Geschwindigkeit, Einkommen, Alter), *ordinal*: Merkmal mit natürlicher Ordnung (z.B. Schulnoten sehr gut, gut, befriedigend,...), *nominal*: Merkmal ohne natürliche Ordnung (z.B. Geschlecht, Name)

- **EM-Clustering (partitionierendes Verfahren):** Dieses Verfahren basiert auf dem Expectation-Maximization-Algorithmus. Dieser arbeitet nach dem Prinzip, zunächst mit einem zufällig gewählten Modell zu starten. In der Expectation-Phase werden die Zuteilungen der Datenobjekte zum Modell verbessert, in Maximization-Phase werden die Modellparameter entsprechend der aktuellen Zuteilung verbessert, also das Modell wird analog der Datenobjekte angepasst [CBD08]. Das EM-Clustering ist im Gegensatz zum k-Means-Algorithmus in seiner Zuordnung unscharf, da prinzipiell jedes Datenobjekt jedem Cluster zugehörig sein könnte, sowie jedes Datenobjekt jeden Parameter verändern könnte.
- **Fuzzy C-Means (partitionierendes Verfahren):** Die Entfernung vom Clusterzentrum bestimmt über den Zugehörigkeitsgrad eines Datenobjekts. Der Zugehörigkeitsgrad wird mittels eines Intervalls zwischen 0 und 1 angegeben, wobei 1 eine totale Zugehörigkeit zum jeweiligen Clusterzentrum bedeutet. Die Verschiebung der Zentren findet im Gegensatz zum k-Means-Algorithmus jedoch abhängig vom Zuordnungsgrad ab. So werden Datenobjekte mit dem Zuordnungsgrad 1 vollständig bei der Berechnung berücksichtigt, während kleinere Zugehörigkeitsgrade den Einfluss entsprechen minimieren [NRP05].
- **divisive Clusterverfahren (hierarchisches Verfahren):** Wie bei der Vorstellung der prinzipiellen Clusterverfahren beschrieben, wird beim divisiven Clustering zunächst mit einem Cluster begonnen, das sämtliche Datenobjekte beinhaltet. In diesem wird nun nach genau dem Datenobjekt gesucht, bei dem die Ähnlichkeit zu den anderen Datenobjekten im Cluster minimal, also die mittlere Distanz maximal ist. In jeder Iteration wird um das so ermittelte Zentrum ein neues Cluster aus den Datenobjekten gebildet, die eine geringe Distanz zum jeweiligen Clusterzentrum aufweisen. Die Iteration läuft genau so lange, bis jedes Cluster nur noch ein Datenobjekt enthält.
- **agglomerative Clusterverfahren (hierarchisches Verfahren):** Dieses ist analog zu den *divisiven Clusterverfahren*, allerdings umgekehrt. Zunächst ist jedes Datenobjekt zugleich ein Cluster. Die Cluster mit einer geringen Distanz werden so lange iterativ zusammengefasst, bis alle Datenobjekte in einem Cluster vorhanden sind. In einem Alternativansatz werden die Cluster zusammengefasst, deren *totale Varianz* (Streuung) die geringste Steigerung aufweist.
- **DBSCAN (dichtebasierter Clusterverfahren):** Cluster werden hier als Areale behandelt, in denen Datenobjekte nah voneinander entfernt lokalisiert sind. Die Cluster werden getrennt durch Areale, in denen die Datenobjekte eine größere Entfernung aufweisen. Ein Cluster wird gebildet, sobald die Dichte der Datenobjekte einen definierten Schwellenwert überschreitet. Kernobjekte sind Datenobjekte, die innerhalb eines definierten Abstandes mindestens k Nachbardatenobjekte besitzen. Randobjekte sind Datenobjekte, die einem Cluster nahe sind und werden in dieses eingeschlossen, alle anderen Datenobjekte werden als *Rauschen* nicht berücksichtigt [JA03].

Klassifikations- und Regressionsalgorithmen

Die Klassifikations- und Regressionsalgorithmen haben zum Ziel, die Daten in Klassen einzuteilen (Vergleich Abbildung 2.1). Beides sind gebräuchliche Formen des überwachten Lernens (supervised Learning) [JWS90]. Das Training findet also anhand von Beispielen statt, bei denen die Einordnung der Daten in Klassen bereits manuell vorgenommen wurde. Der Unterschied zwischen Klassifikation und Regression liegt in erster Linie in der Art der Variablen, die bestimmt wird. Bei der Regression werden kontinuierliche Werte für die Variablen bestimmt, bei der Klassifikation werden dagegen diskrete Werte erwartet.

Grundsätzlich existieren zwei Arten der Klassifikation (Vergleich [JC14]). Die *instanzenbasierten* Klassifikationsverfahren führen eine direkte Klassifizierung auf Grund von Beispieldaten auf den noch zu klassifizierenden Daten durch. Diese Verfahren sind verhältnismäßig einfach aufgebaut, da ein zu untersuchendes Datenobjekt mit den vorhandenen Testobjekten anhand einer Distanzfunktion verglichen und der Klasse mit der größten Ähnlichkeit zugeordnet wird.

Im Gegensatz dazu erstellt das *modellbasierte* Klassifikationsverfahren aus den zur Verfügung gestellten Testdaten ein Modell als Metaebene. Die Testdaten werden nach Erstellung und Verifizierung des Modells für die Durchführung des Klassifikationsprozesses nicht mehr benötigt.

Im folgenden Abschnitt werden einige wichtige Klassifikations- und Regressionsverfahren vorgestellt, die auch Teil der Implementierungen der Machine-Learning-Bibliotheken MLlib und H2O sind.

- **Linear Regression:** Bei den Linear-Regression-Algorithmen handelt es sich um eine der gebräuchlichsten Regressionsmethoden. Hier wird eine statistisch abhängige Variable durch 1 bis n unabhängige Variablen beschrieben [DML15]. Die Regressionskoeffizienten werden in erster Potenz im Modell berücksichtigt. Deshalb wird dieses Verfahren als *Linear Regression* bezeichnet.

Zusammengefasst durch die *Simple Linear Regression* lässt sich das Verfahren wie folgt darstellen [Enz15]:

Die Ausprägung einer Variablen ist bekannt, es soll die Ausprägung einer anderen Variablen vorhergesagt werden. Die vorherzusagende Variable (abhängige Variable) ist das *Kriterium*. Die zur Vorhersage genutzte Variable (unabhängige Variable) ist der *Prädiktor*. Ziel der Regression ist nun, die lineare Funktion zu finden, welche die Abhängigkeit zwischen Prädiktor und Kriterium beschreibt.

Als einfachste Vorhersagefunktion wird hier ein lineares Gleichungssystem betrachtet. Hier werden die Werte von Y geschätzt, indem die Werte von X linear transformiert werden:

$$\hat{y} = a_{yx} + b_{yx} * x$$

mit \hat{y} = Schätzwert des Kriteriums

a_{yx} = Schätzwert des Kriteriums wenn der Prädiktor den Wert 0 hat (*Intercept*)

b_{yx} = Regressionskoeffizient, Steigung der Linie zum vorhergesagten Wert des Kriteriums (*Slope*)

x = Wert des Kriteriums zur Schätzung des Prädiktors.

So kann durch die Koeffizienten die Änderung in den abhängigen Variablen durch die Änderung um eine Einheit in der unabhängigen Variablen vorausgesagt werden.

Nun wird noch ein Maß für die Schätzfehler benötigt. Hierzu wird bei den *Linear-Regression-Algorithmen* auf die *Least-Squares* zurückgegriffen, also die summierten, quadrierten Abweichungen der tatsächlichen von den vorhergesagten Werten. *Intercept* und *Slope* werden so bestimmt, dass die *Least-Squares* minimal sind.

$$\sum(y - \hat{y})^2 = \min (\text{Ordinary Least Squares} - OLS)$$

So wird die Linie für das Modell bestimmt, die aufgrund der geringen Schätzfehlers am Besten passt.

Im Gegensatz zu den *Ordinary Least Squares* kommen insbesondere im Kontext der Bibliotheken von MLLibs und H2O auch die *Alternating Least Squares Methoden* (ALS) zum Einsatz. Diese Methode basiert auf dem Grundsatz, dass eine algebraische Lösung einer Optimierung erheblich vereinfacht wird, wenn die Hälfte der Funktionsparameter in einem Optimierungsschritt unverändert verbleibt [YZ09]. Bei den ALS werden also in den Optimierungsläufen alternierend je eine Hälfte der Attribute angepasst, die andere Hälfte wird nicht verändert [TH14].

Die Funktion für einen *Alternating Least Squares Algorithmus* kann sich folgendermaßen darstellen lassen [Dat15]:

$$f[i] = \operatorname{argmin} \sum(r_{ij} - w^T f[j])^2 + \lambda \|w\|_2^2 \text{ mit } w \in \mathbb{R}^d \text{ und } j \in Nbrs(i)$$

Jedoch existieren auch eine ganze Reihe von verfeinerten und veränderten ALS-Methoden, welche die Modellleistung teils erheblich steigern können oder hier zumindest vielversprechende Ansätze liefern.

Die *Least-Squares-Algorithmen* zählen zu den *kollaborativen Filtermethoden*. Diese finden unter Anderem in Empfehlungssystemen Verwendung.

- **Logistic Regression:** Die *Logistic Regression* ist in ihrem prinzipiellen Zweck der *Linear Regression* sehr ähnlich. Auch hier wird die Beziehung zwischen einer oder mehreren unabhängigen und einer abhängigen Variablen modelliert. Im Gegensatz zur *Linear Regression* wird bei der *Logistic Regression* jedoch die Eintrittswahrscheinlichkeit eines Ereignisses berechnet. Somit wird also kein linearer Zusammenhang zwischen den abhängigen und den unabhängigen Variablen hergestellt, in dem die abhängige Variable einen präzisen numerischen Wert erhält. Statt dessen wird mittels der *Logistic Regression* ein Wahrscheinlichkeitswert zwischen 0 und 1 erwartet [BG15]. Die entsprechende Funktion stellt sich folgendermaßen dar:

$$P = \frac{e^{\alpha + \beta x}}{1 + e^{\alpha + \beta x}}$$

mit P als Eintrittswahrscheinlichkeit für das Ereignis
 α, β als Koeffizienten des Modells (Vergleich *Linear Regression*), die von x abhängig sind.
Wenn x den Wert 0 hat, erreicht α den Wert von P . β zeigt die Wahrscheinlichkeit, dass sich eine 1 ändert, wenn x sich um eine Einheit ändert.

Ein weiterer Bestandteil der Logistic Regression sind die *Odds*, also die Verhältnisse der Wahrscheinlichkeit zu ihrer jeweiligen Gegenwahrscheinlichkeit. Diese werden im Logistic-Regression-Algorithmus als *Logit* (Logarithmus eines Odds) bezeichnet und stellen hier die abhängige Variable dar. Ein Logit ist folgendermaßen definiert:

$$\text{logit}(p) = \ln\left(\frac{P}{1-P}\right)$$

- **Support Vector Machines:** Eine *Support Vector Machine* (SVM) ist ein Verfahren zur Klassifikation und Regression. Ziel ist, eine Menge von zu klassifizierenden Datenobjekten derart aufzuteilen, dass um die Trennlinie herum ein möglichst breiter Korridor frei von Datenobjekten bleibt. Einen solchen Korridor nennt man deshalb *Large Margin Classifier* [Mar15].

Um die Klassen voneinander trennen zu können, wird eine sogenannte *Hyper Plane* (*Hyperebene*) konstruiert. Diese ist definiert durch den Normalenvektor w und die Verschiebung b . Die Definition der Hyperebene ist somit:

$$H = x \mid \langle w, x \rangle + b = 0$$

Support Vectors sind die Punkte, die zu der generierten *Hyper Plane* den geringsten Abstand aufweisen. Diese Stützungsvektoren haben exklusiv Einfluss auf die Lage der *Hyper Plane*, Punkte die nicht als Stützungsvektoren definiert wurden, sind für deren Lage irrelevant.

Die Vorhersage der Klasse für ein neues Objekt findet statt, indem zunächst für einen Trainingslauf w und b so gewählt werden, dass die *Hyper Plane* die Trainingsdatensätze trennt. Im Vorhersageschritt wird nun untersucht, auf welcher Seite des *Hyper Plane* das Objekt liegt. Objekte, die in Richtung des Normalenvektors der *Hyper Plane* liegen, werden als positiv klassifiziert, alle anderen als negativ. Fehler, also Daten, die sich nicht trennen lassen, lassen den Abstand zur Hyperebene mit dem Fehlergewicht C multiplizieren [TH09].

Der optimale Stützvektor wird konstruiert, indem die quadratische Norm $\|w\|$ minimal wird unter der Bedingung, dass für alle $i = 1 \dots m$ gilt:

$k_i * (w^T * x_i + b) \geq 1 - C_i$ mit $1 \leq i \leq m$, wobei C hier als Fehlergewicht eine positive Schlupfvariable¹⁴ darstellt.

Mit dem sogenannten Kernel-Verfahren wird aus einer Trennung, deren Realisierung im zweidimensionalen Raum komplex ist, eine dreidimensionale Darstellung. Eine Fläche besitzt umfangreichere Trennmöglichkeiten, als eine Gerade. Die Kernel-Funktion dient

¹⁴ Schlupfvariablen werden für die für die Lösung eines Problems eingeführt, ihr Wert ist aber nicht von Interesse. Die eingeführte Schlupfvariable führt ein Problem auf ein einfacheres Problem zurück.

als Maß der Ähnlichkeit. Somit muss keine tatsächliche Transformation in die dritte Dimension vollzogen werden.

- **Naive Bayes:** Hierbei handelt es sich um einen wahrscheinlichkeitsbasierten Algorithmus, dessen Zweck die Vorhersage der Klasse mit der höchsten Wahrscheinlichkeit ist. Da es sich um ein instanzenbasiertes Klassifikationsverfahren handelt, ist hier keine Modellentwicklung durch die Trainingsdaten nötig. Beim Naive-Bayes-Verfahren wird von der Annahme ausgegangen, dass sämtliche Datenattribute untereinander statistisch unabhängig sind. Dieser Algorithmus basiert auf der Bayesschen Formel (Vergleich: [Mit15]):

$$P(X | Y) = \frac{P(Y|X)*P(X)}{P(Y)}$$

Laut [DJF15] lässt sich die Naive-Bayes-Klassifikation für mittlere bis große Trainingsmengen für Diagnose oder Klassifikation für Dokumente wie folgt anwenden:

Das Ziel besteht darin, jede Instanz X in der Funktion $f : X \rightarrow V$ durch die Attributwerte $[a_1, a_2 \dots a_n]$ zu beschreiben. Hierzu wird der wahrscheinlichste Wert von $f(x)$ ermittelt durch:

$$f(x) = \operatorname{argmax} P(a_1, a_2 \dots a_n) | v_i) P(v_i) \text{ mit } v_i \in V$$

Beim Naive-Bayes-Algorithmus wird folgende Annahme getroffen:

$$f(x) = P(a_1, a_2 \dots a_n) | v_i) = \prod_i P(a_i | v_i)$$

Somit wird die Naive-Bayes-Klassifikation zu:

$$v_{NB} = \operatorname{argmax} P(v_i) \prod_i P(a_i | v_i) \text{ mit } v_i \in V$$

Also wird für jeden der Klassifikationswerte v_i die Wahrscheinlichkeit $P(v_i)$ berechnet und für jeden Attributwert a_i die von v_i abhängige Wahrscheinlichkeit $P(a_i | v_i)$.

- **Decision Trees:** Ein Decision Tree (Entscheidungsbaum) ist ein Klassifikationsverfahren, das als hierarchischer Baum für den Merkmalsraum X und den Klassenraum K aufgebaut ist [AL15]. An jedem Knoten wird den Ästen jeweils eindeutig ein Element $x \in X$ zugewiesen. Jedes Blatt ist einer Klasse aus K zugewiesen. [ST15]

Ein Entscheidungsbaum wird generiert, indem man aus einer Attributmenge A und einer Beispieldatenmenge B ein Attribut $a \in A$ als Wurzel des Entscheidungsbaumes bestimmt. Wenn nun x_a die Menge aller Werte von a repräsentiert, so wird für jede Ausprägung dieses Attributs die Menge $B^x \subseteq B$ gebildet [JC14]. Für diese gilt:

$$\forall b \in B^x : x_a(b) = x$$

Danach wird eine so markierte Kante an die entsprechende Wurzel gelegt. Wenn $B^x = \emptyset$, wird die entsprechende Kante beendet. Wenn alle Elemente aus B in der gleichen Klasse sind, endet die Kante mit einem entsprechenden Blatt.

Für ein Objekt wird eine Klasse vorhergesagt, indem der Baum von der Wurzel beginnend traversiert wird, jedem Knoten ein Ast angehängt wird, der die Zuordnung des Objekts x erhält und schließlich bei den erreichten Blättern die jeweilige Klasse abgelesen wird. Die Anzahl der Blätter entspricht der Anzahl von Entscheidungsregeln.

Entscheidungsbäume können als Klassifikationsverfahren in Betracht gezogen werden, wenn die Instanzen durch Attribut-Wertpaare beschreibbar sind, die Zielfunktion einen diskreten Wert besitzt, *disjunkte Hypothesen* zur Klassifikation erstellt werden sollen, oder wenn die Trainingsdaten möglicherweise starkes Rauschen aufweisen. *Decision Trees* neigen leicht zu einem *Overfitting* der Testdaten.

- **Random Forests:** Bei den *Random Forests* handelt es sich um eine Form der *Ensemble-Learning-Methoden* [LB15]. Diese bauen während des Trainings eine Menge unterschiedlicher *Decision Trees* auf, um aufgrund verschiedener Teile der Trainingsdaten Durchschnitte auf verschiedenen großen Decision Trees zu bilden. Ziel ist, die Varianz zu reduzieren und damit die Modellleistung erheblich zu steigern. Allerdings steigen gegenüber einfachen *Decision Trees* der *Bias*¹⁵ und durch gesteigerte Komplexität sind diese oft nicht mehr einfach interpretierbar.

2.4 Das MapReduce-Paradigma

Bei *MapReduce* handelt es sich um ein Programmierparadigma, das von *Google, Inc.* entwickelt wurde, um große Datensätze mittels eines auf einem Cluster verteilten und nebenläufig ausführbaren Algorithmus analysieren und verarbeiten zu können [JD04]. Dieses Paradigma eignet sich sowohl für strukturierte, als auch unstrukturierte Daten, Voraussetzung für die erfolgreiche Verarbeitung ist allerdings, dass die *Tasks* einen hohen Parallelisierbarkeitsgrad aufweisen (Vergleich [JD04]).

MapReduce nimmt einen Satz von Eingabe Key/Value-Pairs entgegen und produziert daraus einen Satz von Ausgabe Key/Value-Pairs. Das Paradigma besteht aus drei jeweils parallelisierbaren Einzelschritten, map, shuffle und reduce von denen map und reduce durch den Anwender selbst definiert werden müssen [AP14].

Diese Schritte von MapReduce in der Übersicht:

- **Map:** Hier wird aus den Eingabedaten mittels durchgehend gleicher Berechnungen ein temporär relevantes *Key/Value-Pair* erzeugt. Für jedes Wertepaar aus der Eingabeliste wird die *Map-Funktion* separat und unabhängig aufgerufen. Die Map-Funktion besteht aus Filter- und Sortierregeln. Die Ausführung erfolgt nebenläufig und verteilt.
- **Shuffle:** In diesem Schritt werden die Ergebnisse vor dem Aufruf der *Reduce-Funktion* nach ihrem durch *Map* erzeugten neuen Schlüssel in *Collections* gruppiert. Da in diesem Schritt bei verteilten Systemen und großen Datenmengen gegebenenfalls extreme Last im Netzwerk erzeugt werden kann, kommt hier oft vorher noch ein *Combine-Schritt*

¹⁵ Ein Bias (auch Trend) bezeichnet einen systematischen Störeffekt, der bei Messungen auftreten kann und eine steigende oder fallende Tendenz aufweist [LB15].

zur Anwendung. Dieser sorgt mittels einer Vorreduktion dafür, dass die Datenmenge, die beim Shuffle über das Netzwerk transferiert wird, bereits auf den jeweiligen Nodes möglichst stark reduziert wird.

- **Reduce:** Auf jede im *Shuffle-Schritt* erzeugten Collection wird hier eine beliebige, durch den Anwender in Form eines Clojure definierte Funktion ausgeführt. Ziel dieser Funktion ist die Reduktion der vorhandenen Datensätze. Diese Funktion erstellt eine Ausgabe-Collection mit den Ergebnissen und ist ebenfalls je Collection nebenläufig und verteilt anwendbar.

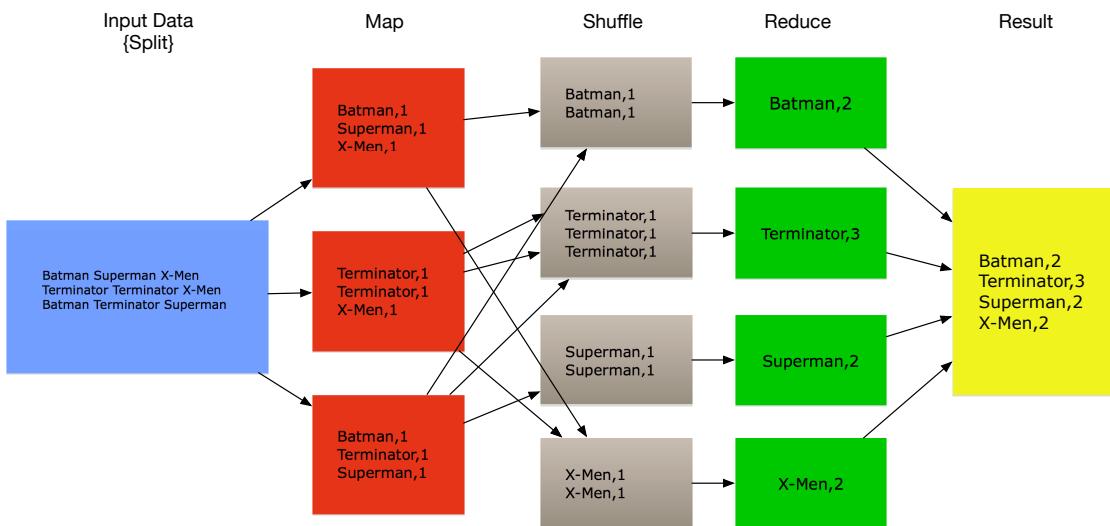


Abbildung 2.5: Ein konkretes Word-Count-Beispiel für MapReduce

In Abbildung 2.5 wird das *MapReduce-Modell* an einem praktischen Beispiel zur Wortzählung gezeigt. Bei *MapReduce* wird zunächst eine problemspezifische *Map-Funktion* definiert. Diese teilt die Ursprungsmenge mittels einer internen *Split-Funktion* in gleich große Partitionen und versieht unstrukturierte Werte in einem ersten Schritt mit Standardwerten, um so aus jedem unstrukturierten Datensatz ein Key-Value-Pair zu generieren [Xia15]. Im gezeigten Beispiel wird hier jedem Wort die Zahl Eins als Key zugeordnet. Im nächsten Schritt werden diese so gewonnenen Zwischen-Key-Value-Paare in einem *Shuffling-Prozess* klassifiziert und die so homogenisierten Pakete auf die Knoten im Cluster verteilt. Nun enthält jede Ausführungseinheit nur noch gleiche Wörter. Im Reduce-Schritt schließlich, werden die gleichartigen Wörter zu Gesamtmengen zusammengefasst, addiert und im Endergebnis aggregiert.

An dem gezeigten Beispiel ist deutlich erkennbar, dass die einzelnen Ausführungsschritte jeweils parallelisiert und auf separate Knoten im Cluster verteilt werden können. Das Laufzeitsystem übernimmt die Details der Partitionierung der Eingabedaten, die Ressourcenverwaltung innerhalb des Clusters, das Behandeln von Fehlern und die Kommunikation zwischen den einzelnen Knoten.

2.5 Anwendungen von Graphen

Graphen sind geeignet, um Beziehungsstrukturen innerhalb von Daten zu modellieren. Graphen können in Baum oder Netzform vorliegen und mittels verschiedener Strategien traversiert werden. Dies kann mittels Tiefensuche, Breitensuche oder anhand von priorisierten Wegen durchgeführt werden. Prinzipiell werden Daten und deren Beziehungen in Form von Kanten, Knoten und Blättern dargestellt. Den Kanten können Gewichte zugeordnet werden, um Relevanz zwischen Beziehungen oder Wegen umzusetzen.

TBD!

2.6 Zusammenfassung

TBD!

Kapitel 3

Der Berkeley Data Analytics Stack (BDAS)

Rund um *Hadoop* beziehungsweise *Spark* wurde an der University of California in Berkeley ein ganzer Infrastruktur-Stack für Big-Data-Analytics aufgebaut, der BDAS. Im folgenden Kapitel wird dieser Stack und die Bibliotheken, aus dem dieser besteht, vorgestellt. Im ersten Unterkapitel wird zunächst ein kurzer Überblick über den gesamten Stack gegeben. In den darauffolgenden Unterkapiteln werden die einzelnen Bestandteile im Einzelnen oberflächlich betrachtet, um dem Leser einen Einblick in die Nutzungsmöglichkeiten zu bieten. Eine Detailbetrachtung der jeweiligen Grundlagen, der praktischen Anwendung, Messungen und Vergleichsbetrachtungen der Alternativimplementierungen folgen im weiteren Verlauf dieser Ausarbeitung. Das Kapitel schließt mit einer Zusammenfassung, die auf einen Blick die Zusammensetzung des BDAS rund um Apache Spark zeigt.

3.1 Die Schichten des BDAS

Im Folgenden wird der *Berkeley Data Analytics Stack (BDAS)* näher vorgestellt, der wie in der Einführung bereits erwähnt um Hadoop, bzw. Spark als Hauptbestandteile herum aufgebaut ist. Der BDAS wurde von den *AMPLabs*¹ von der University of California in Berkeley aufgrund von Forschungsergebnissen im Bereich der Analyse sehr großer Datenmengen ins Leben gerufen.

Der Einsatz des BDAS kann laut Vijay Agneeswaran [Agn14] dabei helfen, beispielsweise konkrete praktische Fragen wie die folgenden zu beantworten:

- Wie segmentiert man am besten eine Menge von Nutzern und kann herausfinden, welche Nutzersegmente an bestimmten Kampagnen interessiert sein könnten?

¹ kurz für „algorithms, machines and people“

- Wie kann man richtige Metriken für Nutzer-Engagement in Social-Media-Applikationen herausfinden?
- Wie kann ein Video-Streaming-Dienst für jeden Nutzer dynamisch ein optimales *Content-Delivery-Network (CDN)*² basierend auf Daten wie Bandbreite, Auslastung, Pufferrate, etc auswählen?

Prinzipiell sind die in der Einführung in Kürze beschriebenen Einschränkungen von Hadoop und die damit verbundene Motivation für Spark auch die Motivation für den BDAS. Besonders für Aufgaben, die iterative Datenzugriffe und -manipulationen erfordern, wie beispielsweise Machine-Learning Algorithmen oder interaktive Abfragen, ist Hadoop auf Grund seiner starken festspeicherabhängigkeit nur bedingt zu empfehlen. Für diese Aufgaben ist Spark mit seinen In-Memory-Primitives prädestiniert. Um Hadoop herum ist in den letzten Jahren ein umfangreiches Ökosystem von Bibliotheken und Frameworks gewachsen, so dass die meisten Aufgaben im Umfeld von Big-Data-Analytics mit einer Hadoop-Infrastruktur gelöst werden können. Dagegen spricht laut [Kun14], dass für jeden Nutzungsfall ein eigener, auf Hadoop basierender Technologiestack eingesetzt werden muss, große Expertise in mehreren Technologien für produktives Arbeiten nötig ist und die Architektur vor allem für Aufgaben, bei denen schneller Datenaustausch zwischen parallel bearbeiteten Tasks notwendig ist, unpassend ist.

Im Gegensatz dazu bietet der BDAS auf allen Ebenen Standard APIs für Java, Scala, Python und SQL an und mit MLLibs stehen etliche Machine-Learning-Algorithmen direkt in Spark zur Verfügung. Der Stack obshalb von Spark ist flexibel konfigurierbar und sämtliche Bibliotheken lassen sich parallel betreiben.

In Abbildung 3.1 wird eine Übersicht über die Hauptschichten des BDAS gezeigt, welche die jeweils von den AMPLabs empfohlene Implementierung zeigt.

In der untersten Schicht befindet sich das Cluster-Management-System Mesos, darüber das verteilte Filesystem HDFS, die Caching-Schicht Tachyon und der eigentliche Spark Kernel. Auf diesem setzen die Bibliotheken MLLib für Machine-Learning, Spark Streaming für Streaming-Anwendungen, GraphX für Graphenapplikationen und für datenbankähnliche Abfragen auf dem BDAS Spark SQL auf.

In den folgenden Unterkapiteln werden die einzelnen Elemente des BDAS detailliert vorgestellt.

² Ein Content Delivery Network (auch Content Distribution Network) ist ein Netzwerk verteilter und über das World Wide Web verbundener Server, das den Zweck hat, grosse Dateien bereitzustellen und auszuliefern.

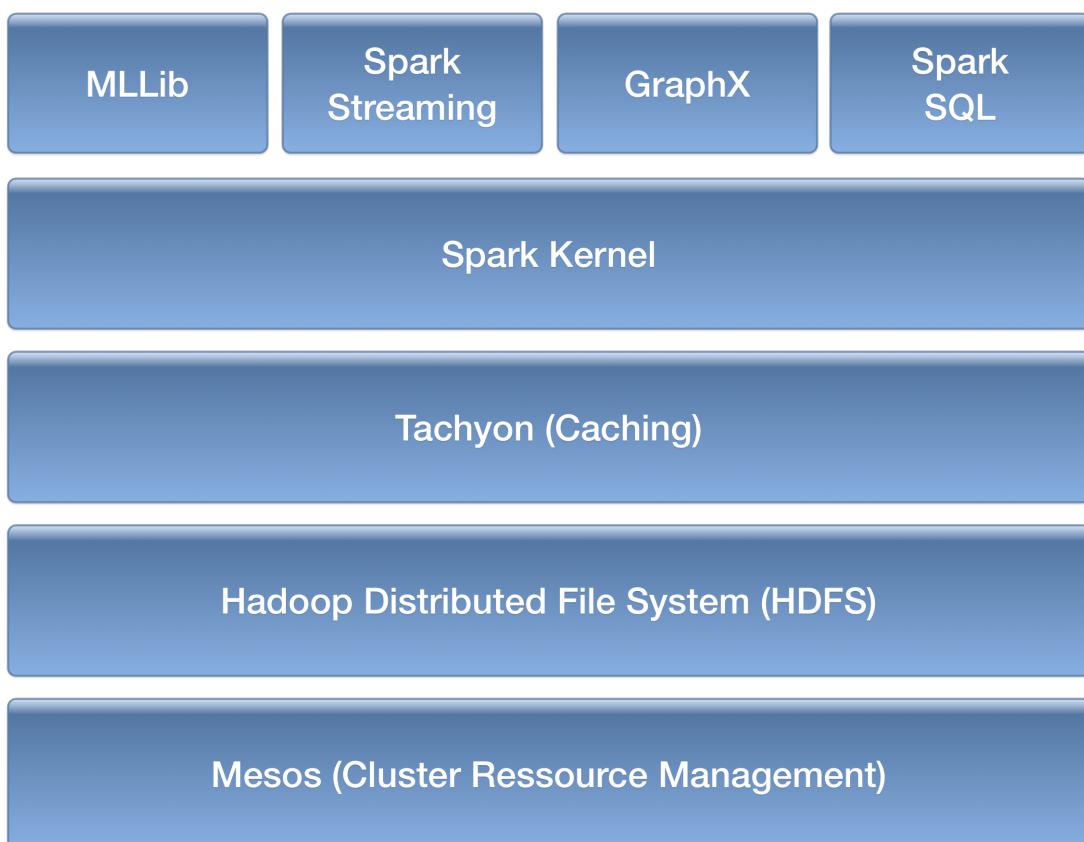


Abbildung 3.1: Übersicht des BDAS mit den vom AMPLab empfohlenen Bibliotheken.

3.2 Apache Mesos

Bei *Apache Mesos* handelt es sich um ein *Cluster-Management-Framework* für Anwendungen, die in verteilten Serverpools laufen sollen. Bestandteil von Mesos ist wiederum *Apache ZooKeeper*, das für Konfigurationsinformationen, Naming-Services und die Synchronisation von verteilten Anwendungen zuständig ist.

Mesos wird im BDAS eingesetzt, um die Prozesse von Hadoop/Spark effizient auf die einzelnen Knoten im Cluster zu verteilen. Besonders das Ressourcen-Management und –Monitoring innerhalb des Clusters ist ein wichtiger Faktor, um Jobs performant auf verteilten Systemen ausführen zu können. Auch das Fehlerhandling für Knoten, Prozesse und im Netzwerk wird im Berkeley-Stack von Mesos übernommen.

Ein besonderer Vorteil von Mesos gegenüber Yarn oder anderen Alternativen, wie dem Cloudera Cluster Manager oder Ambari von Hortonworks ist die Möglichkeit, verschiedene Frameworks gleichzeitig und isoliert in einem Cluster betreiben zu können. So kann beispielsweise Hadoop mit Spark in einer gemeinsamen Infrastruktur koexistieren.

3.3 Hadoop Distributed File System (HDFS) und Tachyon

Das Hadoop Distributed File System basiert ideologisch auf dem GoogleFileSystem (GFS) und hat zum Zweck, zuverlässig und fehlertolerant sehr große Dateien über verschiedene Maschinen hinweg in verteilten Umgebungen zu speichern. In entsprechenden Veröffentlichungen von Hortonworks [Hor14] wird von Produktivsystemen berichtet, die bis zu 200 PetaByte an Datenvolumen in einem Cluster von 4500 Servern basierend auf HDFS verwalten.

HDFS wurde speziell für den Einsatz mit MapReduce entwickelt, ist also auf geringe Datenbewegungen ausgelegt, da MR die Berechnungsprozesse jeweils zu den physischen Datensätzen selbst bringt und nicht, wie herkömmlich, die Daten zu den Prozessen geliefert werden müssen. So wird massiv Netzwerkverkehr innerhalb des Clusters eingespart und letztlich werden nur Prozesse und Prozessergebnisse verschickt.

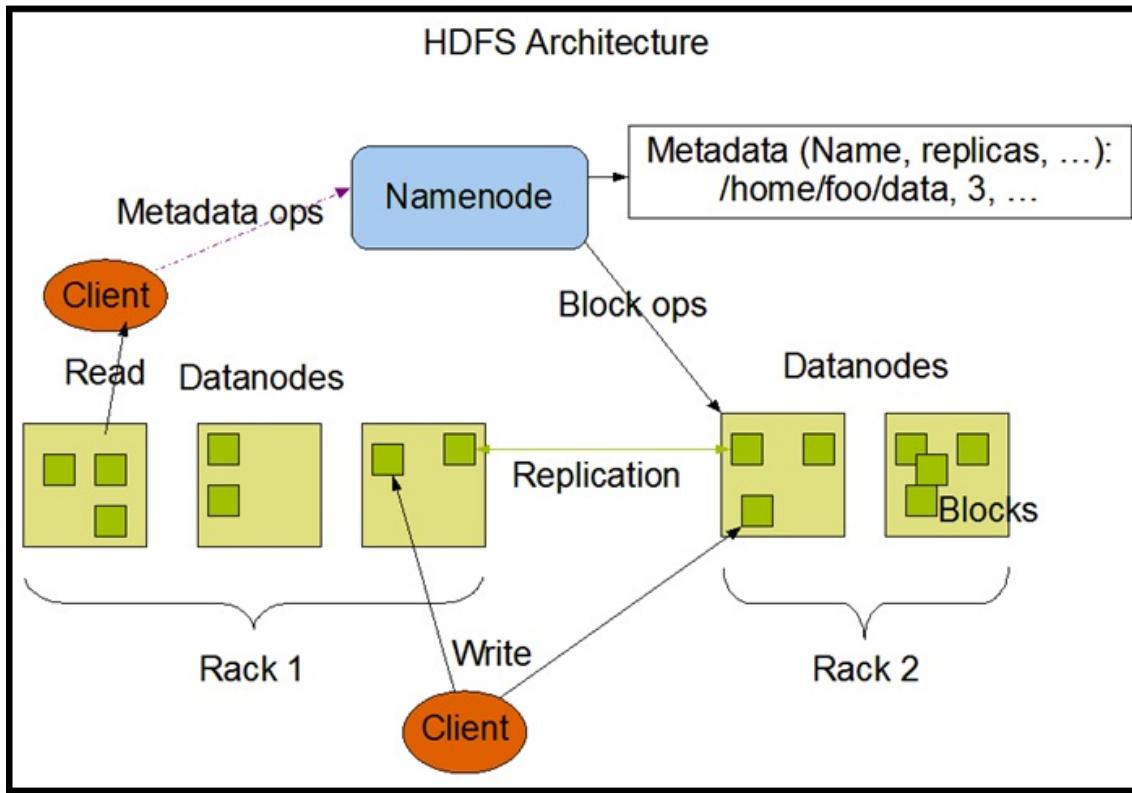


Abbildung 3.2: Der Aufbau von HDFS mit Namenodes und Datanodes [Had15].

Die Hauptbestandteile von HDFS sind der sogenannte NameNode, der die Metadaten des Clusters verwaltet und die DataNodes, die die eigentlichen Daten halten. In Abbildung 3.2 ist dieser Aufbau schematisch dargestellt. Dateien und Verzeichnisse werden vom NameNode durch inodes repräsentiert. Diese wiederum enthalten Informationen über Zugriffsrechte, Zugriffszeiten oder Größenangaben der Dateien.

In Abbildung 3.3 wird die Datenmanagementschicht des BDAS detaillierter dargestellt. Es lässt erkennen, dass zwischen den Frameworks und dem Data-Layer ein optimiertes Caching-

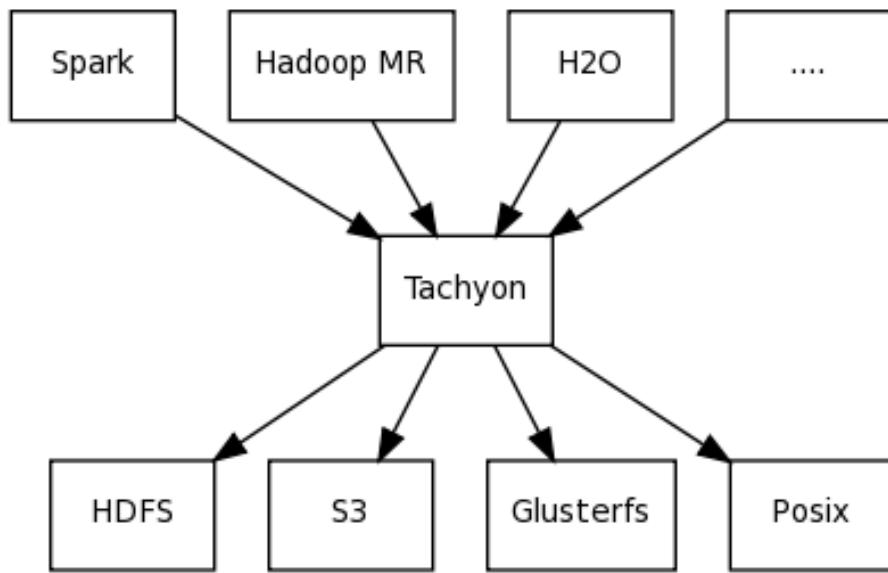


Abbildung 3.3: Der Datamanagement-Layer im BDAS mit Tachyon und diversen Dateisystemen [Glu14].

Framework für sämtliche Dateisystemaufrufe zum Einsatz kommen kann.

Hier lässt sich wahlweise direkt das *HDFS* bzw. dessen Alternativen ansprechen oder alternativ die Zwischenschicht nutzen, die insbesondere auf das In-Memory-Modell von Spark zugeschnitten ist. Dies ist innerhalb des BDAS das verteilte Dateisystem Tachyon. Hier werden die zu verarbeitenden oder zu analysierenden Datensätze direkt in den Hauptspeicher des jeweiligen Knoten in Form eines Cache gehalten. Somit werden Lade- und Speicheroperationen auf Massenspeicher minimiert und eine massiv höhere Ausführungsgeschwindigkeit erreicht.

Tachyon besteht aus einem Master, der in der Regel dem physischen Master einer Spark-Infrastruktur entspricht. Dieser deligiert und überwacht die einzelnen Tachyon Worker-Nodes, die auch den Spark Worker-Nodes entsprechen. Tachyon implementiert auf jedem Knoten ein eigenes RAM-Filesystem, das den Hauptspeicher abseits des JVM-Heaps nutzt [Hon15].

Unterhalb von Tachyon ist nach wie vor ein HDFS für die persistente Datenhaltung notwendig. Alternativ kann auch das Amazon S3-File-System, Glusterfs oder Posix eingesetzt werden. Tachyon wurde direkt innerhalb der AMPLabs entwickelt und ist mittlerweile fester Bestandteil des BDAS.

3.4 Apache Spark

Spark ist das Herzstück des BDAS. Bei Spark handelt es sich um ein open-source Data-Analytics-Framework, das, wie Hadoop, speziell für die Bedürfnisse im Rechner-Cluster konzipiert ist. Auch Spark nutzt das HDFS entweder direkt, oder indirekt über Tachyon. Im Ge-

gensatz zu Hadoop bietet Spark jedoch Funktionen für In-Memory-Cluster-Berechnungen und ist nicht zwingend an MapReduce gebunden. Besonders interaktive Analyse oder Verarbeitung der Daten, Abfragen über verteilte Dateien und iterative

Lernalgorithmen erfahren so laut AMPLab eine bis zu hundertfache Ausführungs-geschwindigkeit im Gegensatz zu Hadoop. Auch die im ersten Kapitel angesprochenen Schwächen von Hadoop bei Berechnungen von komplexen linear-algebraischen Problemen, generalisierten n-Körper-Problemen, diversen Optimierungsproblemen und diversen anderen Aufgaben, treten bei Spark auf Grund der offenen Architektur und der Zerlegung von Datensätzen in die sogenannten Resilient Distributed Datasets (RDD) nicht mehr auf.

Spark wurde komplett in Scala entwickelt und bietet APIs für Scala, Java (inklusive Lambda-Expressions von Java 8) und Python. Im Labor existieren bereits Spark-Installationen mit bis zu 2000 Knoten, in Produktivsystemen sind bisher Systeme mit bis zu 1000 Knoten im Einsatz [Met14]. Durch die Möglichkeit, die Datensätze im Speicher für interaktive Analyseaufgaben zu cachen und iterativ abzufragen, ist eine direkte Kommandozeileninteraktion über das integrierte Scala REPL (alternativ auch in Python) möglich.

Für Spark existieren dedizierte Bibliotheken für Verarbeitung von Datenströmen, Machine-Learning und Graphenverarbeitung. Ähnliche Artefakte existieren auch für Hadoop (Mahout, Vowpal Wabbit, etc.), jedoch ist die Architektur von Spark wesentlich besser für derartige Anwendungsbereiche zugeschnitten.

3.5 Spark Streaming

Spark Streaming ist eine der oben genannten Bibliotheken, die Spark um dedizierte Anwendungsbereich erweitert. Hierbei handelt es sich um eine Erweiterung, um die integrierte API von Spark für Anwendungen auf Datenströmen nutzen zu können. Das Programmiermodell unterscheidet nicht zwischen Batch- und Streaming-Anwendungen. So lassen sich beispielsweise Datenströme zur Laufzeit mit Archivdaten vergleichen und direkt Ad-hoc-Abfragen auf die Ströme formulieren. Im Fehlerfall ermöglicht Streaming zahlreiche Wiederherstellungsoptionen, sowohl von verlorenen Datenströmen, als auch von Einstellungen. Ein Anwendungsbeispiel ist die Echtzeitanalyse von Twitter-Meldungen.

3.6 GraphX

GraphX ist eine Erweiterung für Spark, die verteilte, flexible Graphen-Anwendungen in einem Spark-Cluster ermöglicht [Xin14]. Besonders in den Disziplinen „Machine Learning“ und „Data Mining“ ist die Anwendung komplexer Graphen unerlässlich. Graph-datenbanken kommen immer dann zum Einsatz, wenn stark vernetzte Informationen und ihre Beziehungen zueinander

interessant sind. Hier werden die Entitäten als Knoten behandelt, die Beziehungsart definiert die Kanten. Die Kanten können auch gewichtet sein.

Ein konkretes Beispiel sind die Mitglieder eines sozialen Netzwerks mit ihrem jeweiligen Beziehungsgeflecht. Je nach Kontaktintensität können diese Beziehungen auch priorisiert werden, was hier dem Kantengewicht entspricht.

GraphX nutzt hier die Vorteile der darunterliegenden Spark-Infrastruktur, in dem durch eine tabellarische Anordnung der Datenstrukturen eine massive Parallelisierung möglich ist und auch der Verarbeitung in RDDs voll unterstützt wird. So sind auch interaktive Operationen auf den Graphen jederzeit über REPL möglich.

3.7 MLbase/MLLib

MLbase ist eine Sammlung von Bibliotheken und Werkzeugen für Machine-Learning-Anwendungen mit Spark. Sie besteht grundsätzlich aus den drei Teilen MLlib, MLI und ML-Optimizer und ist oberhalb der Spark-Installation angesiedelt, wie auf Abbildung 3.4 zu erkennen ist.

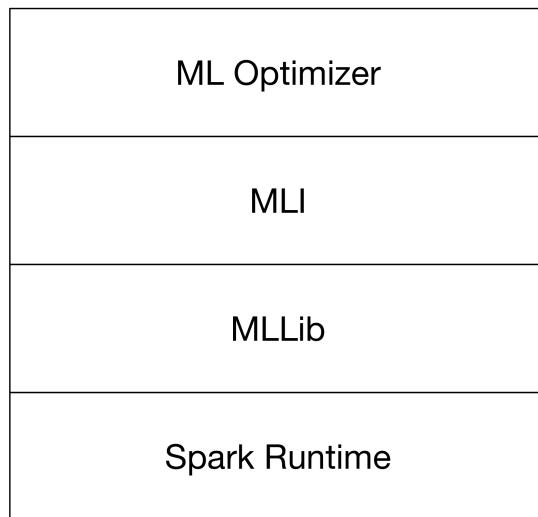


Abbildung 3.4: Die Bestandteile der MLbase

Die MLlib ist eine verteilte Machine-Learning-Bibliothek die für die Spark-Laufzeitumgebung entwickelt wurde und die bekannten Algorithmen für Probleme wie Klassifikation, Regression, Clustering und kollaboratives Filtern enthält (Vergleich [Lou14]).

Bei MLI handelt es sich um eine API, die es ermöglicht, selbst ML-Features zu entwickeln und in erster Linie für komplexere Problemstellungen geeignet ist. Mit MLI lassen sich die Funktionen direkt gegen Spark entwickeln, gegebenenfalls unter Zuhilfenahme der Bibliotheken der MLlib.

Der ML-Optimizer soll ML-Probleme für Endnutzer vereinfachen, in dem Modellauswählen

automatisiert werden. Hierzu werden Features aus der MLlib und der MLI extrahiert und zur Hilfe genommen.

3.8 Spark SQL

Im Ökosystem von Hadoop ist *Hive* eine *SQL-Query-Engine*, die sich großer Beliebtheit in der Community erfreut und trotz inzwischen zahlreicher Konkurrenzimplementierungen immer noch als Quasi-Standard für diesen Anwendungsbereich gilt.

Spark SQL³ ist eine Portierung dieser Engine für Spark, um alle Vorteile der BDAS-Architektur nutzen zu können und ist kompatibel mit sämtlichen Hive-Daten-, -Metastores und –Queries. Im Gegensatz zu Hive, das aus Datensätzen zur Laufzeit Java-Objekte generiert, nutzt Spark SQL eine zeilenorientierte Speicherung mittels Arrays primitiver Datentypen und ist somit selbst in einer Hadoop-Infrastruktur im Mittel bis zu fünfmal schneller als Hive.

Eine Besonderheit von Spark SQL ist neben seinem SQL-Interface die Möglichkeit, auch Machine-Learning-Funktionen als Abfragen formulieren zu können.

Für die Anwendung von Spark SQL hat sich die Architektur von Spark mit seinen RDDs als sehr vorteilhaft erwiesen, da Abfragen auf Fehlerhaften RDDs nach dem Neuaufbau des entsprechenden Datasets direkt erneut ausgeführt werden können.

Ein weiterer Unterschied zu Hive ist die sogenannte Partial-DAG-Execution (PDE). Dies bedeutet, dass logische Abfragepläne in Spark SQL aufgrund gesammelter Statistiken zur Laufzeit flexibel erstellt werden im Gegensatz zu Hive oder herkömmlichen relationalen Datenbanksystemen, wo bereits zur Kompilierungszeit starre physische Abfragepläne generiert werden. Besonders die Machine-Learning- und Failover-Funktionen wären mit einer Planerstellung zu Kompilierzeit nicht umsetzbar.

3.9 Zusammenfassung

TBD!

³ Ursprünglich war die für den BDAS-Stack empfohlene Implementierung einer SQL-Query-Engine unter dem Namen *Shark* bekannt. Im Juli 2014 wurde jedoch bekanntgegeben, dass die Entwicklung von Shark zugunsten von Spark SQL eingestellt wurde und die vorhandenen Shark-Implementierungen voll in Spark SQL integriert werden. Deshalb zeigen Schaubilder des BDAS von vor Juli 2014 Shark als Query-Engine.

Kapitel 4

Alternative Implementierungen der Bibliotheken und Frameworks des BDAS

Wie in den Abbildungen 3.1 und 6.1 im vorhergehenden Kapitel ersichtlich ist, existieren auf jeder Ebene des BDAS auch alternative Implementierungen. Im folgenden Kapiteln werden einige Alternativimplementierungen vorgestellt und den Bibliotheken aus dem BDAS gegenübergestellt. Hier wird gezeigt, dass Spark Streaming, je nach Nutzungskontext durch Apache Storm ersetzt werden, oder statt der mitgelieferten Machine Learning Library MLLibs H2O oder Data GraphLab CreateTM eingesetzt werden kann.

4.1 Alternative zu Spark: Apache Flink

Bei Apache Flink handelt es sich um ein weiteres verteiltes Big Data Analytics Framework. Somit stellt es eine mögliche Alternativimplementierung für Apache Spark dar. Flink ist aus einem Forschungsprojekt der Technischen Universität Berlin, der Humboldt Universität Berlin und des Hasso Plattner Instituts hervorgegangen. Hier wurde unter dem Projektnamen Stratosphere laut [ML13] ein System zur massiven Parallelverarbeitung von großen strukturierten und unstrukturierten Datenmengen entwickelt. 2014 wurde es zu einem Apache Incubating Project und im Zuge dessen in Apache Flink umbenannt.

Erklärtes Ziel von Flink ist ein möglichst einfach zu programmierendes und für Entwickler transparentes Big Data Analytics Framework. Dies wird durch intuitive Abstraktionen von datenintensiven Berechnungsmodellen innerhalb der APIs umgesetzt. Derzeit existieren APIs für die Programmiersprachen Java und Scala, weitere sind in Planung.

Eine Besonderheit von Flink ist die Übersetzung von *High-Level-Operatoren* wie *map*, *reduce*,

filter, join, intersect, etc. in *Dataflow DAGs*¹ (Vergleich [RK94]). Diese Übersetzung beinhaltet Optimierungsfunktionen, die auf den jeweiligen Datenflusskosten basieren. Wenn die zugrundeliegenden Daten oder die Clusterkonfiguration sich ändern, ist eine Anpassung der Flink-Anwendungen so mit sehr wenig Aufwand verbunden.

Flink nutzt, ähnlich wie Spark, ebenfalls eine aggressive In-Memory-Strategie für seine Berechnungen. Wenn der zur Verfügung stehende Hauptspeicher für die benötigten Daten nicht ausreicht, werden interne Operationen auf die im Festspeicher persistierten Daten ausgelagert und ermöglichen so laut [EMC15] eine sehr robuste Ausführung auf Systemen jeglicher Hauptspeicherausstattung.

Ebenfalls vergleichbar mit Spark ist die Ausstattung mit Bibliotheken für Anwendungen wie Streaming, Graphen-Anwendungen, Machine Learning und SQL-Abfragen.

Auch im Handling und im Umgang mit den Datenstrukturen existieren einige Gemeinsamkeiten zwischen Apache Flink und Apache Spark. In beiden Frameworks wird bei der Verarbeitung der Daten mittels *Lazy Evaluation* vorgegangen (Vergleich Kapitel 5.2). Im Falle von Spark wird das Laden und Transformieren der Daten bei Aufruf einer *Function* ausgeführt, bei Flink entspricht dies den *Execute Methods*. Auch die Transformationen sind sehr ähnlich gegenüber Spark umgesetzt. In der Tabelle 4.1 (Inhalt aus [Fli15], hier ist die vollständige Übersicht über alle Transformationen zu finden) sind einige Transformationen exemplarisch aufgeführt.

Transformation	Zweck
map	Takes one element and produces one element.
filter	Evaluates a boolean function for each element and retains those for which the function returns true.
flatMap	Takes one element and produces zero, one, or more elements.
mapPartitions	Transforms a parallel partition in a single function call. The function get the partition as an Iterator and can produce an arbitrary number of result values. The number of elements in each partition depends on the degree-of-parallelism and previous operations.
aggregate	Aggregates a group of values into a single value. Aggregation functions can be thought of as built-in reduce functions. Aggregate may be applied on a full data set, or on a grouped data set.
union	Produces the union of two data sets.

Tabelle 4.1: Übersicht der einiger wichtiger Transformationen von Apache Flink

¹ Ein DAG ist ein Directed Acyclic Graph. Im Kontext der Parallelverarbeitung in verteilten Systemen repräsentiert dieser die Pfade durch den Kontrollflussgraphen, der benötigt wird, um den Datenfluss zu berechnen.

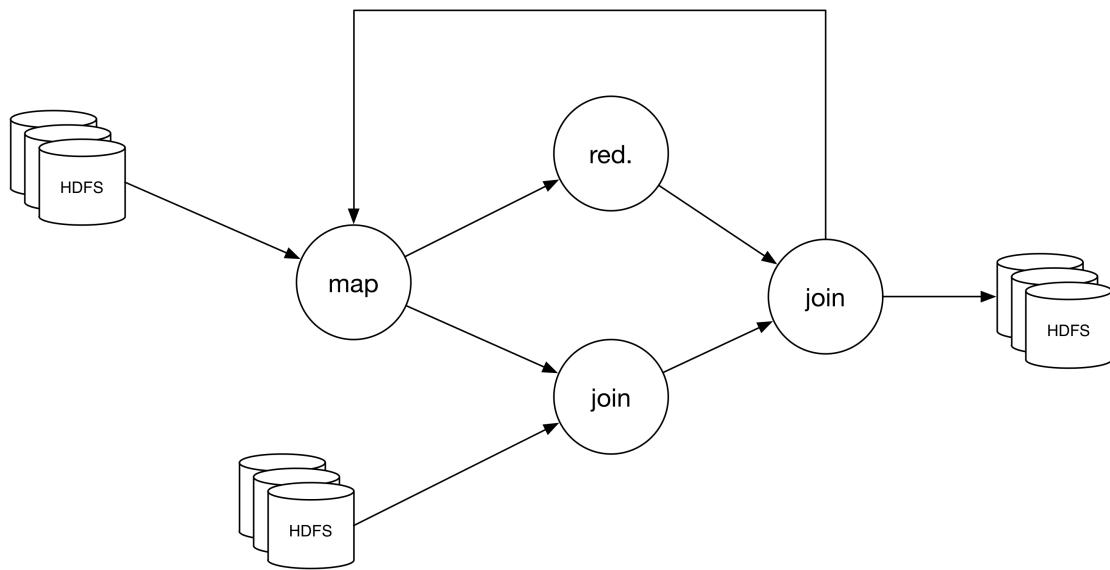


Abbildung 4.1: Flink Streaming Dataflow mit Feedback führt bei Iterationen automatische Optimierungen durch.

Im Unterschied zu Spark basiert Apache Flink auf einem feingranularen Speichermanagement-Modell. Dies erlaubt eine sehr differenzierte Verteilung der Daten und Operationen zwischen Haupt- und Festspeicher [Ewe15]. Spark nutzt für seine Verarbeitungen ein internes Batch-Processing, die Daten werden also stets in Form von RDDs im RAM, im Cache oder auf Festspeicher materialisiert. Flink hingegen nutzt im Runtime-Layer echtes Datenstreaming, die Daten werden in ihrer Binärrepresentation verarbeitet. So werden, wie in Abbildung 4.1 dargestellt, in jeder Iteration Optimierungen mit Kostenfunktionen durchgeführt. Bei jedem Durchlauf wird überprüft, welche Daten Veränderungen erfahren. So wird feingranular die Zuordnung zu den jeweiligen Speicherstrategien festgelegt. Durch partielle Serialisierung können die Daten flexibel zwischen RAM, Cache und Festspeicher aufgeteilt werden.

Da Spark intern mit Batchmechanismen operiert, ist hier allerdings eine wesentlich höhere Fehlertoleranz möglich. Das Prinzip der RDDs, die sich im Fehlerfall nur mit diesem Mechanismus zur Laufzeit aus den Transformationsregeln rekonstruieren lassen, ist nur so umzusetzen.

4.2 Alternative zu Spark Streaming: Storm

Storm ist, wie Apache Streaming, ein Framework für Hadoop, bzw. Spark für verteilte Streaming-Anwendungen. Wo Spark ganz klar eine Verbesserung gegenüber Hadoop darstellt und Spark SQL dementsprechend gegenüber Hive, ist die Situation bei Storm und Apache Streaming hingegen nicht so klar determinierbar.

Storm und Spark Streaming unterscheiden sich fundamental in ihren Verarbeitungsmodellen [Agn14]. Das erstgenannte Framework verarbeitet eintreffende Events nacheinander, immer

genau eines pro Zeitraum. Spark Streaming sammelt im Gegensatz dazu die Events in Mini-Batch-Jobs und verarbeitet sie paketweise zu definierten Zeiträumen nach wenigen Sekunden. Deshalb kann Storm Latenzzeiten von deutlich unter einer Sekunde erreichen, während Spark Streaming eine Latenzzeit von einigen Sekunden aufweist. Diesen Nachteil macht Spark Streaming durch eine sehr gute Fehlertoleranz wett, da die Mini-Batches nach aufgetretenen Fehlern einfach nochmals bearbeitet werden können und die zuvor fehlerhaft ausgeführte Verarbeitung verworfen wird. Treten hingegen bei Storm Fehler auf, wird genau dieser Datensatz nochmals verarbeitet. Dies bedeutet, dass dieser auch mehrfach verarbeitet werden kann. Durch dieses Verhalten lassen sich die beiden Frameworks grob in zwei Einsatzgebiete verteilen:

Storm ist das Framework der Wahl, wenn Wert auf sehr kurze Latenzzeiten gelegt werden muss, hingegen ist es für statusbehaftete Anwendungen durch die Möglichkeit der Mehrfachverarbeitung ungeeignet. Im Umkehrschluss ist Spark Streaming eine gute Wahl, wenn aufgrund der gestreamten Daten eine Statusmaschine aufgebaut werden soll. Dafür müssen hier höhere Latenzzeiten in Kauf genommen werden.

4.3 Alternative zu MLLibs: H2O - Sparkling Water

TBD!

4.4 Alternative zu MLLibs: Data GraphLab Create™

TBD!

4.5 Zusammenfassung

TBD!

Kapitel 5

Funktionsweise von Spark

Im vorhergehenden Kapitel wurde der Berkeley Data Analytics Stack vorgestellt. Es wurde gezeigt, dass dieser aus einer Reihe von Bibliotheken, Infrastrukturkomponenten und dem eigentlich Kern, Apache Spark, besteht.

In diesem Kapitel werden die grundlegenden Konzepte von Spark vorgestellt und dessen Funktionsweise betrachtet. Einleitend wird gezeigt, wie eine Spark-Infrastruktur aufgebaut sein kann, wie diese intern Abfragen und eigene Spark-Programme verarbeitet und wie der *Spark-Context* sich als Cluster-Repräsentant gegenüber dem Anwender und der API exponiert. Im nächsten Unterkapitel wird die eigentliche Basis von Apache Spark vorgestellt. Spark basiert im Wesentlichen auf einer verteilten Datenstruktur, den *Resilient Distributed Datasets*. Deren Konzept wird sowohl theoretisch, als auch im Anwendungskontext dargestellt.

Ein weiteres Kernelement der Spark-Implementierung bildet das *In-Memory-Processing* der Daten. Spark ist in der Lage, je nach Konfiguration des Host-Systems, große Teile der Analysen und Verarbeitungen äußerst flexibel im Hauptspeicher durchzuführen und so massive Performanceverbesserungen gegenüber festspeicherbasierter Verarbeitung zu generieren. Hierzu bietet Spark spezielle *In-Memory-Primitives* an. In einem weiteren Unterkapitel werden dieses detailliert vorgestellt.

5.1 Spark im Cluster

Eine große Herausforderung im Umfeld verteilter und nebenläufiger Analyse und Verarbeitung großer Datenmengen stellt der Netzwerkverkehr da. Der klassische Aufbau einer verteilten Anwendung hält die Daten auf einer dafür vorgesehenen Plattform im Netzwerk. Häufig ist dies ein dedizierter File- oder Datenbankserver, der mit möglichst großer Bandbreite mit dem Applikationsserver verbunden ist.

In Abbildung 5.1 wird ein möglicher Aufbau eines herkömmlichen Clusters gezeigt. Dieser

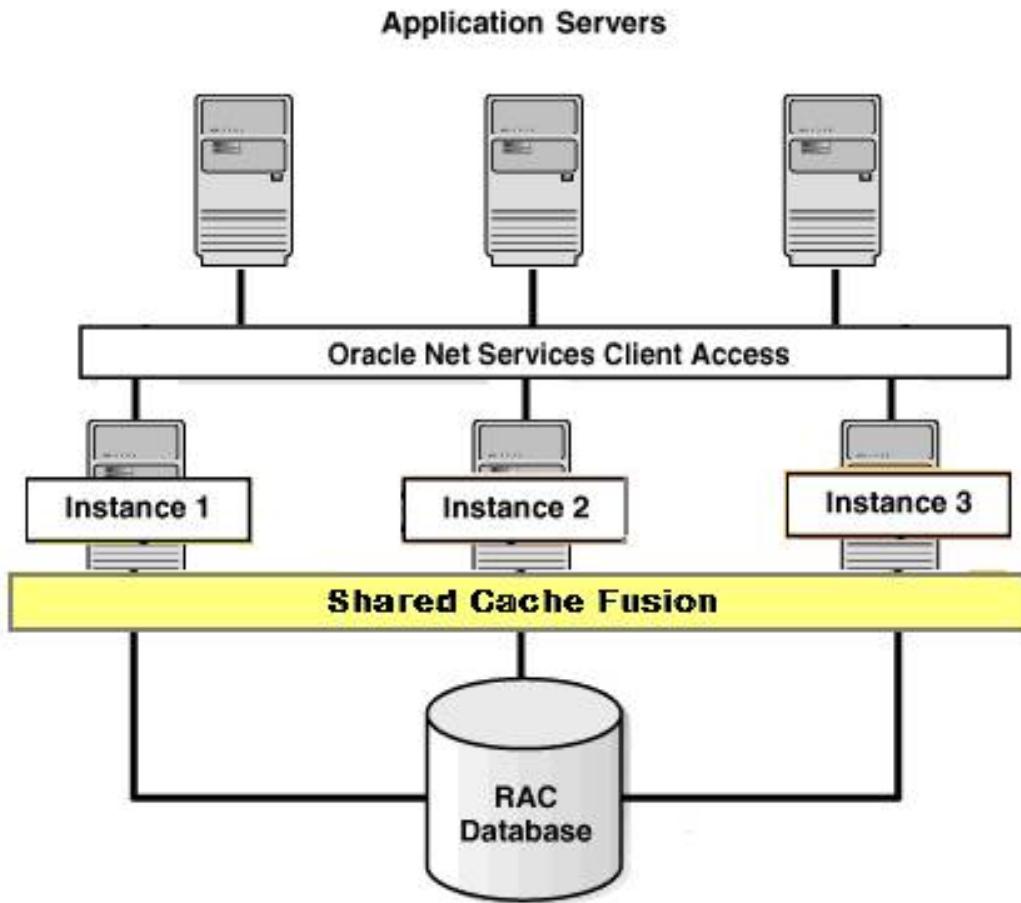


Abbildung 5.1: Aufbau eines Standardclusters im Rechenzentrumsbetrieb mit Application-Servern und Oracle Datenbanken [Sto15].

besteht in diesem Fall aus drei Applikationsservern, die wiederum über drei Instanzen eines verteilten Caching-Systems auf eine zentrale Datenbank zugreifen. In einem solchen Aufbau entsteht in der Regel eine sehr hohe Netzwerklast, da die für die Applikation benötigten Daten dieser zunächst zur Verfügung gestellt werden müssen. Die Applikation fragt die Daten ab, diese werden dann von der Datenbank zu den Applikationsservern geliefert, hier meist in Applikationsobjekte umgewandelt und stehen dann der Anwendung zur Verfügung.

Spark geht hier einen anderen Weg. Ein Spark-Cluster besteht typischerweise aus einem zentralen *Master* und n *Worker-Nodes*. Diese können aus einfachen Servern bestehen, aber auch aus Clustern von Mainframes (beispielsweise IBM Z, Oracle ExaLogic, etc.). Das Hadoop Distributed File System und Spark skalieren über Cluster beliebiger Größenordnung. Über ein verteiltes Dateisystem werden die Daten auf dem Cluster gehalten und sowohl dem *Master*, als auch den *Worker-Nodes* so zur Verfügung gestellt.

Abbildung 5.2 zeigt den Aufbau eines Spark-Clusters. Spark-Anwendungen laufen als unab- hängiges Set von Prozessen auf Cluster-Infrastrukturen. Das Hauptprogramm, der sogenannte

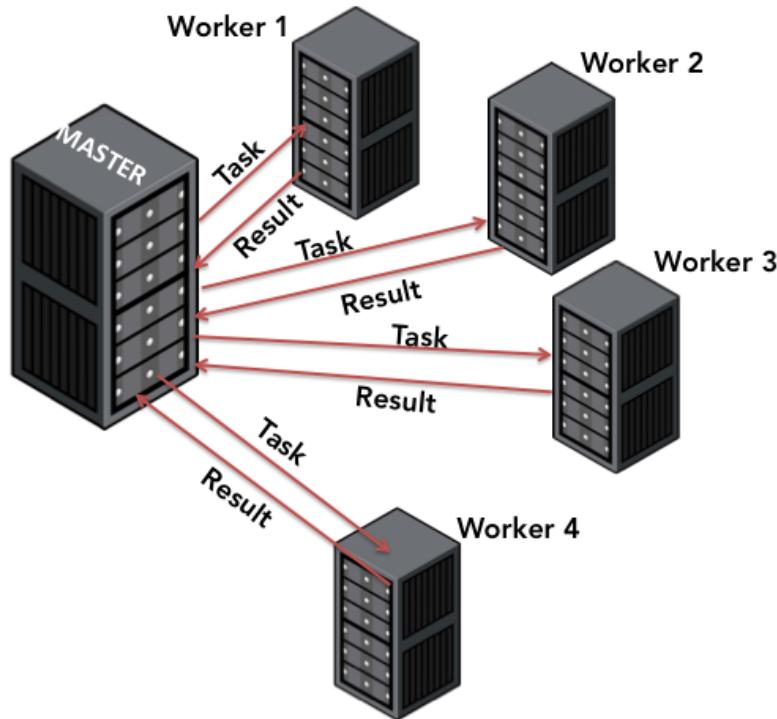


Abbildung 5.2: Clusteraufbau mit Spark mit einem Master und vier Worker-Nodes.

Spark Driver, instanziert das *SparkContext-Objekt*, das die einzelnen Prozesse koordiniert. Auf Clustersystemen hält der *SparkContext* die Verbindung zum jeweiligen *Cluster-Ressource-Manager* (Mesos, Yarn), im Standalone-Betrieb instanziert der Context selbst einen Dummy-Manager und allokiert in beiden Fällen die für die Anwendung nötigen Hardware-Ressourcen. Die Cluster-Manager liefern ihren aktuellen Status an Spark zurück und melden Auslastung und Gesundheitszustand der einzelnen Knoten. Über interne *Load-Balancing-Systeme*¹ wird ermittelt, welche Worker-Nodes die jeweiligen Tasks aus dem Spark-Kontext zugewiesen bekommen. Der *SparkContext* repräsentiert sowohl für die Spark-Konsole REPL, als auch in eigenen Spark-Programmen innerhalb der APIs das gesamte Cluster. Dem *SparkContext* wird bei der Initialisierung über ein Konfigurationsobjekt mitgeteilt, welche Ressourcen ihm für das aktuelle Programm zur Verfügung stehen. Die Entscheidung, welche, der initial zur Verfügung gestellten Nodes oder Ressourcen des Clusters von Spark wann in Anspruch genommen werden, obliegt der Kombination aus Spark und *Cluster-Ressource-Manager*.

Wenn ein *SparkContext* initialisiert wurde, installiert der Spark sogenannte *Executors* auf sämt-

¹ Load-Balancing-Systeme sind Überwachungsmechanismen in verteilten Systemen. Jedes Teilsystem meldet seine eigene Auslastung und seine Verfügbarkeit an den Load-Balancer. Dieser verteilt anstehenden Aufgaben so auf die Ressourcen, dass eine möglichst gleichmäßige Verteilung über die gesamte Infrastruktur möglich ist.

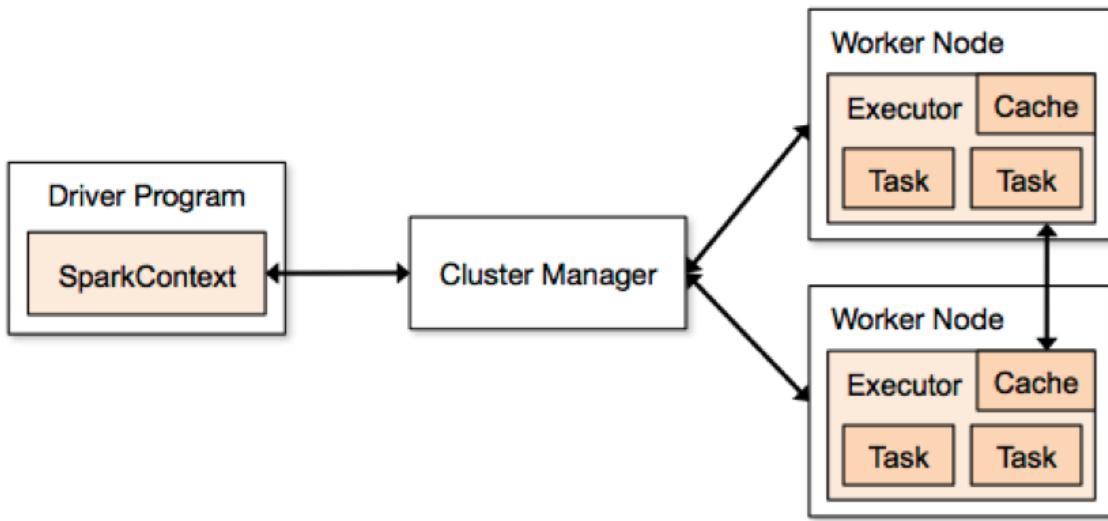


Abbildung 5.3: Clusteraufbau mit Spark [Apa14]

lichen Worker-Nodes des Clusters. Der Applikationscode wird nun als JAR² direkt an die Executors verteilt und dieser anschließend durch entsprechende Tasks ausgeführt.

5.2 Das Konzept der Resilient Distributed Datasets

Die Resilient Distributed Datasets (RDD) sind das eigentliche Kernelement von Apache Spark. Hierbei handelt es sich um fehlertolerante, parallele Datenstrukturen, die dem Anwender erlauben, Zwischenergebnisse explizit im Hauptspeicher zu halten, ihre Partitionierung zu steuern, um Daten bewusst an bestimmten Stellen halten zu können und diese mittels umfangreichen Operatoren zu manipulieren [Spa15]. Das Konzept der RDDs entspricht prinzipiell den Views³ in relationalen Datenbanksystemen. Werden die RDDs für weitere Zugriffe persistiert, entspricht dies dem Prinzip der Materialized Views⁴.

RDDs werden mittels deterministischer, paralleler Operationen erstellt, den sogenannten *Transformationen*. Im Fehlerfall, also wenn beispielsweise ein *Node* im Cluster ausfällt, können die RDDs automatisch anhand der durchgeführten Regeln neu aufgebaut werden. Deshalb merken sich die RDDs die Transformationen, die zu ihrem Aufbau geführt haben und können so verlorene Datenstrukturen schnell rekonstruieren. Die *Resilient Distributed Datasets* können ausschließlich durch Transformationen, wie beispielsweise *map*, *filter*, *join*, etc., auf Daten aus

² Ein JAR (Java ARchive) ist ein gepacktes und auf einer Java Virtual Machine ausführbares (Java, Scala, Clojure) Programm paket, häufig inklusive der benötigten Bibliotheken.

³ Eine View ist in einem relationalen Datenbanksystem die Ergebnismenge einer persistierten Datenbankabfrage auf bestimmte Daten. Diese lässt Abfragen analog zu einer gewöhnlichen Tabelle zu.

⁴ Materialized Views entsprechen bei Abfragen den regulären Views. Allerdings wird hier im Gegensatz zu den Views bei Zugriff keine Abfrage auf die zugrundeliegenden Tabellen durchgeführt. Stattdessen sind die Daten als Kopie in eigenen Datenbankobjekten persistent vorhanden.

dem Dateisystem oder aus anderen, bereits vorhandenen RDDs erzeugt werden, oder durch Verteilen einer Object-Collection in der Driver-Applikation. Diese Datenstrukturen müssen nicht persistiert werden, da sie für einen Partitionierung ausreichende Informationen über ihre Erstellungsregeln und die entsprechenden Datensätze enthalten, das sogenannte *Lineage* [Spa15].

Da es sich bei RDDs prinzipiell um Scala-Collections handelt, können diese auch direkt in Scala-Code eingebunden und verarbeitet werden, oder interaktiv über die Scala-Konsole REPL genutzt werden. Aber auch für Java und Python bietet Spark APIs an. Für weitere Sprachen wie R oder Clojure existieren Wrapper-Frameworks, welche die RDDs und ihre Operationen verfügbar machen (Vergleich 5.5. RDDs können nur durch grobgranulare, deterministische Transformationen, erstellt werden [Kun14]).

Wie eingangs beschrieben, verfügen die Anwender von Spark über die Kontrolle der Aspekte *Persistence* und *Partitioning* [Spa15]. Mit der Persistence lässt sich festlegen, welche RDDs wiederverwendet werden sollen, also welche RDDs nach welcher Strategie persistiert werden. Dies ist besonders wichtig, da nicht explizit persistierte RDDs bei jeder darauf ausgeführten Operation neu berechnet werden. Das Partitioning legt fest, nach welchen Kriterien die RDDs geteilt und über das Cluster verteilt werden sollen, also beispielsweise sortiert nach bestimmten Keys. Die Optimierung der Partitionierung ist unter anderem elementar für Join-Operationen. Je nach Partitionierungsstrategie kann dies erhebliche Unterschiede in Laufzeit und Ressourcennutzung bedeuten [HK15].

RDDs können prinzipiell auf folgende Arten für die Wiederverwendung gespeichert werden [Apa14]:

- Als deserialisiertes Java-Objekt im Speicher der JVM – dieses Variante bietet die beste Performance, da die Objekte sich direkt im JVM-Heap befinden, benötigt jedoch aufgrund der *Lazy Evaluation* ab dem Zeitpunkt der ersten Zugriffe Platz im Heap-Speicher der Laufzeitumgebung und können umfangreicher sein, als nötig.
- Als serialisiertes Java-Objekt direkt im Speicher – dieses Verfahren ist effizienter bezüglich der Hauptspeicherauslastung da andere Speicherbereiche, als der JVM-Heap für die Materialisierung verwendet werden, aber schlechter in der Zugriffsgeschwindigkeit.
- Im Dateisystem – diese Variante ist erwartungsgemäß die langsamste, jedoch nötig, wenn die RDDs zu groß für die Haltung im RAM sind.

Die Default-Serialisierung findet über die Java Standardbibliothek `ObjectOutputStream` für Serialisierung statt. So kann jede Klasse, die `java.io.Serializable` implementiert, serialisiert werden. Dies ist laut [Spa15] oft nicht optimal, da dieser Serialisierer verhältnismäßig langsam ist und häufig zu sehr großen Serialisierungen führt. Spark bietet deshalb als Alternative den Kryo-Serialisierer an (Vergleich [Eso15]). Dieser ist erheblich schneller und erstellt kompaktere Serialisierungen, als die Default-Implementierung. Nachteilig wirkt sich allerdings

aus, dass nicht alle Typen von Serialisierungen unterstützt werden und sämtliche zu serialisierenden Klassen bereits im Voraus registriert werden müssen. Gemäß [Spa15] wird der Einsatz des Kryo-Serialisierers insbesondere für netzwerkintensive Aufgaben mit Spark ausdrücklich empfohlen.

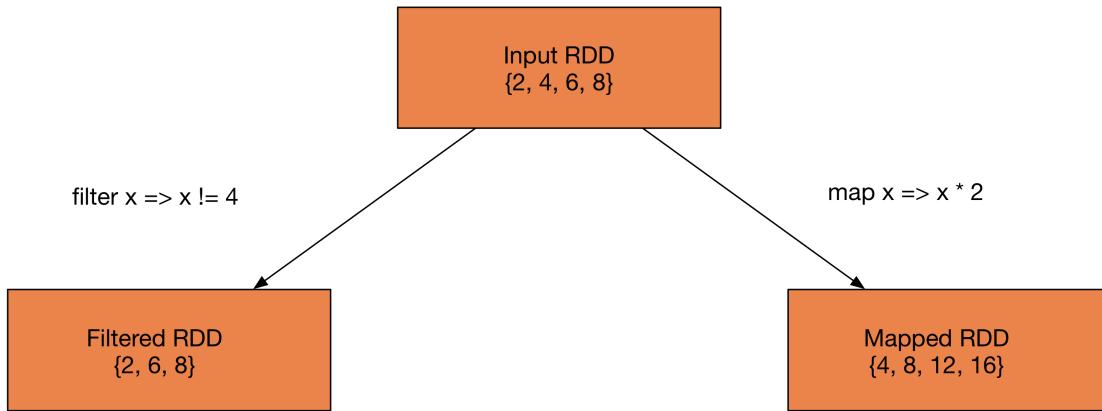


Abbildung 5.4: Transformation von RDDs in Spark.

In der folgenden Tabelle befindet sich exemplarisch eine Auswahl⁵ der wichtigsten Transformationen auf RDDs (aus [Spa14]):

Transformation	Zweck
map(func)	Return a new distributed dataset formed by passing each element of the source through a function func.
filter(func)	Return a new dataset formed by selecting those elements of the source on which func returns true.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
mapPartitions(func)	Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type <code>Iterator<T> => Iterator<U></code> when running on an RDD of type T.
intersection(otherDataset)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([numTasks]))	Return a new dataset that contains the distinct elements of the source dataset.

Tabelle 5.1: Übersicht der einiger wichtiger Transformationen auf RDDs

Transformationen werden auf RDDs grundsätzlich nach dem Prinzip der *Lazy Evaluation* durch-

⁵ Die vollständige Übersicht über die Transformationen und Actions auf RDDs befindet sich im Anhang dieser Ausarbeitung.

geführt (Vergleich [HK15]). Dies bedeutet, dass eine Ausführung der Transformation erst dann stattfindet, wenn die betreffenden Daten auch wirklich benötigt werden. Dies wurde in Spark so umgesetzt, um die Anzahl der Datentransfers, beispielsweise bei Gruppierungsaktionen, drastisch zu reduzieren. Diese Art der Ausführung birgt allerdings auch Problempotential bei einer etwaigen Fehlerlokalisierung, da häufig nicht transparent ersichtlich ist, ob die Evaluierung zum Zeitpunkt des Fehlerauftretens schon durchgeführt wurde.

Wie in Abbildung 5.5 dargestellt, werden die Daten bei einer Verarbeitung durch Spark zunächst aus dem HDFS geladen, in Resilient Distributed Datasets (RDD) transformiert, und dann im Hauptspeicher für weitere Transformationen oder Actions zur Verfügung gestellt. Des Weiteren können aus bestehenden RDDs wiederum neue RDDs durch Transformationen generiert werden, wie in Abbildung 5.4 dargestellt.

Transformationen, Actions oder Abfragen auf RDDs werden direkt entweder über eigene Programme, via Scala REPL oder SQL-artige Abfragen zur Laufzeit, über Batch-Jobs oder via Spark Streaming/Storm an die erstellten RDDs gerichtet.

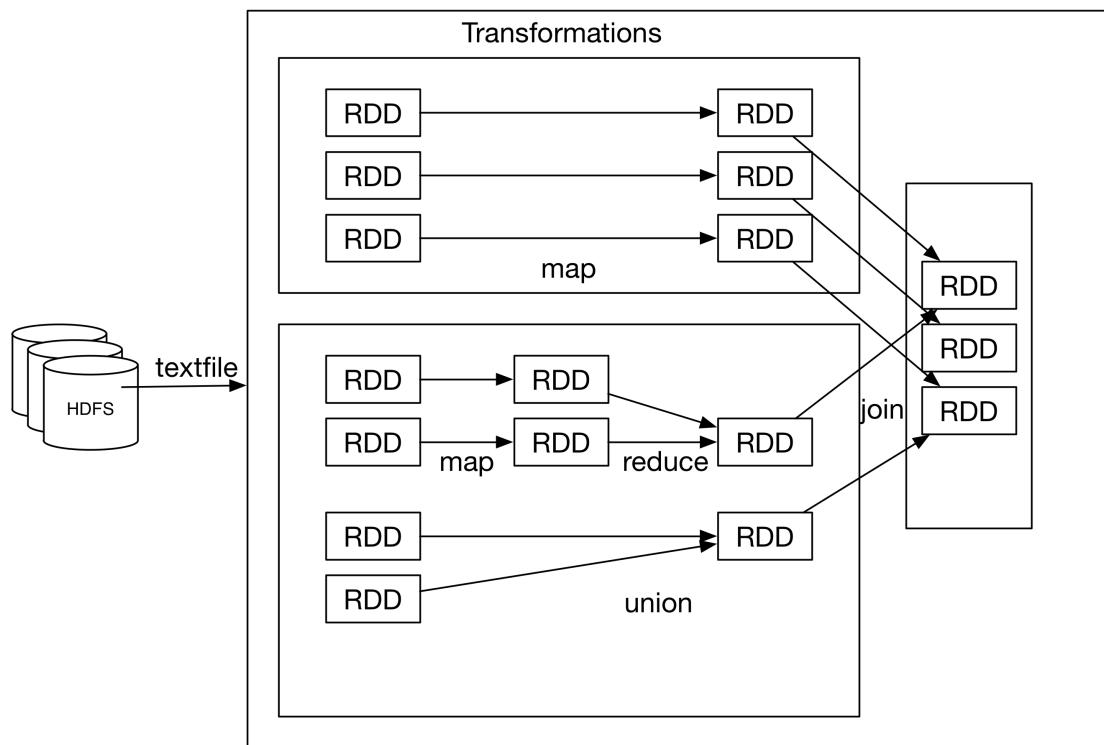


Abbildung 5.5: Schematische Darstellung der Erzeugung und Verarbeitung von RDDs aus persistierten Daten

Im Gegensatz zu den Transformations, die immer ein neues RDD als Ergebnismenge erzeugen, liefern die Actions ein Ergebnis zurück, das entweder an das Driver-Programm auf dem Spark Master zurückgegeben, oder im Dateisystem persistiert wird.

Die Tabelle 5.2 zeigt exemplarisch einige wichtige Actions, die auf RDDs angewendet werden

können. Eine vollständige Übersicht aller Actions befindet sich im Anhang dieser Arbeit.

Actions	Zweck
reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).

Tabelle 5.2: Übersicht der einiger wichtiger Actions auf RDDs

Generell kapselt Spark seine Variablen in die jeweiligen Tasks [Zah12]. So kennt jeder Knoten nur die Variablen, die zu dem durch Spark deployten Task gehören. Ist es notwendig, dass große RDDs über Tasks hinweg verfügbar gemacht werden müssen, bietet sich hierfür die Funktion *Broadcast* an. Nach dem Erstellen eines RDD ist es möglich, dem SparkContext mitzuteilen, dass diese erzeugte Collection über Taskgrenzen hinweg verteilt werden soll.

Folgender Codeausschnitt soll dieses Vorgehen verdeutlichen [Zah12]:

Listing 5.1: Verwendung der Broadcast-Variablen in Spark

```

1 val pageNames = sc.textFile("pages.txt").map(...)
2 val pageMap = pageNames.collect().toMap()
3 val bc = sc.broadcast(pageMap)
4 val visits = sc.textFile("visits.txt").map(...)
5 val joined = visits.map(v => (v._1, (bc.value(v._1), v._2)))

```

In der ersten Zeile wird mittels `map` aus dem Textfile ein RDD erzeugt, in Zeile zwei wird durch `collect` eine neue Map erzeugt und die dritte Zeile teilt dem SparkContext mit, dass diese über die Task- und Node-Grenzen hinweg verteilt werden soll. Der Variablen `joined` wird in Zeile fünf mittels des Aufrufs von `bc.value` der Wert der geteilten Variable übergeben.

5.3 RDD Persistenz: Storage-Level von Spark

Wie im vorhergehenden Abschnitt erwähnt wurde, lässt sich Spark durch den Anwender eine eigene Definition der Speicherstrategie von Spark zu. Zu diesem Zweck stellt die API von Spark hier sieben verschiedene Storage-Levels zur Verfügung. Standardmäßig verwaltet Spark

die Speicherzuteilung und Lokalisierung der generierten RDDs selbst nach der Prämisse, diese in der Form zu partitionieren, dass sie möglichst im RAM der jeweiligen Konten Platz finden. Es existieren aber durchaus Nutzungsfälle, wo ein manuelles Eingreifen des Nutzers in diese Strategie sinnvoll ist.

Im Folgenden werden die Storage-Levels von Spark vorgestellt (aus [Spa14]):

Storage Level	Zweck
MEMORY_ONLY	Store RDD as serialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as serialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than serialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP	(experimental) Store RDD in serialized format in Tachyon. Compared to MEMORY_ONLY_SER, OFF_HEAP reduces garbage collection overhead and allows executors to be smaller and to share a pool of memory, making it attractive in environments with large heaps or multiple concurrent applications. Furthermore, as the RDDs reside in Tachyon, the crash of an executor does not lead to losing the in-memory cache. In this mode, the memory in Tachyon is discardable. Thus, Tachyon does not attempt to reconstruct a block that it evicts from memory.

Tabelle 5.3: Übersicht der Storage Levels von Spark

Wenn ein RDD problemlos in den Speicher passt, sollte man dessen Level auf der Default-Einstellung MEMORY_ONLY belassen, da dies die schnellsten Ausführungszeiten garantiert.

Ist dies nicht der Fall, sollte die Verwendung MEMORY_ONLY_SER in Verbindung mit einem schnellen Serialisierer getestet werden. Generell wird empfohlen, Überläufe der RDDs auf den Festspeicher wenn möglich zu verhindern. Nur wenn die Berechnungsfunktionen zur Erstellung der RDDs sehr teuer waren, oder die Ergebnismenge sehr umfangreich ist, sollte diese Möglichkeit genutzt werden. In diesem Fall ist die Weiterverarbeitungsgeschwindigkeit durch Spark auf die Lesegeschwindigkeit des Festspeichers beschränkt.

5.4 Die Spark-Console REPL

TBD!

5.5 Die Spark APIs

TBD!

5.5.1 Spark Scala API

TBD!

5.5.2 Spark Java API

TBD!

5.5.3 Spark Python API

TBD!

5.5.4 Third Level API für Spark mit Clojure: Flambo

TBD!

5.5.5 Third Level API für Spark mit R: SparkR

TBD!

5.6 Zusammenfassung

TBD!

Kapitel 6

Architektur und Inbetriebnahme von lokalen Apache Spark Infrastrukturen

Im folgenden Kapitel wird ein exemplarischer Architekturaufbau eines lokalen Entwicklungs- und Testsystems für Apache Spark vorgestellt. Wie in den vorangegangen Kapiteln gezeigt wurde, handelt es sich bei Spark in erster Linie um ein Framework für leistungsstarke Cluster-systeme. Dennoch muss Spark auch auf entsprechend kleiner dimensionierten System betrieben werden können. Häufig ist in einem professionellen Umfeld beispielsweise nur ein Cluster für Produktionsaufgaben vorhanden, die Test- und vor allem auch die Entwicklersysteme stellen sich häufig in Form sogenannter *Single-Node-Cluster* dar.

Besonders der Aspekt der Entwicklungsarbeitsplätze rückt hier in den Vordergrund. Da Spark lediglich über den *Master-Node* und hier über den *Spark-Context* innerhalb des *Driver-Program* für die Entwickler zugreifbar ist und die interne Verteilung der Tasks auf die jeweiligen *Nodes* von Spark und den *Clustermanagement-Systemen* übernommen und maskiert wird, stellt sich die Fehleranalyse mittels klassischem *Debugging*¹ als sehr große Herausforderung dar. Problematisch ist hierbei in erster Linie, dass vor und während des Debugging-Prozesses zu keiner Zeit der aktuelle Ausführungs-Node im Cluster determiniert werden kann. Da auf jedem *Node* eine unabhängige *JVM*² instanziert ist, kann nicht zentral ermittelt und nachvollzogen werden, wo und zu welchem Ausführungszeitpunkt welches Verhalten auftritt. Für initiale Entwicklungstä-tigkeiten ist aus diesen Gründen immer eine lokale Instanz von Spark als Single-Node-Cluster nötig.

Zum Zweck von Versuchen bezüglich Skalierbarkeit im Cluster-Umfeld und Verhalten im ver-teilten Betrieb empfiehlt es sich, darüber hinaus eine lokale Cluster-Infrastruktur mit mindestens einem Master-und einem unabhängigen Worker-Node zu installieren. Dies kann auf

¹ Unter klassischem Debugging wird hier das Setzen von Breakpoints und das explizite Überwachen von Va-riablenwerten zur Laufzeit durch Entwickler verstanden.

² JVM = Java Virtual Machine. Hierbei handelt es sich um die Laufzeitumgebung von Java. Auch Scala-Code wird intern in Bytecode übersetzt und innerhalb der JVM ausgeführt.

verschiedene Arten stattfinden. Im Rahmen dieser Ausarbeitung wurden verschiedene Konfigurationen aufgebaut und miteinander verglichen. Diese Aufbauten mit ihren jeweiligen Stärken und Schwächen werden im folgenden Kapitel beschrieben.

Des weiteren wird die Installation und die grundsätzliche Anwendung der Bibliotheken rund um Spark, bzw. der Alternativimplementierungen gezeigt. Abschließend werden Empfehlungen für verschiedene Einsatzbereiche gegeben.

6.1 Prinzipieller Aufbau einer lokalen Spark Infrastruktur

Eine Spark-Infrastruktur besteht immer aus verschiedenen Schichten, welche im BDAS definiert sind und von denen einige zwingend notwendig sind, einige optional und andere durch Alternativimplementierungen ersetzt werden können. In Abbildung 6.1 ist der Aufbau des BDAS schematisch dargestellt. Die grün hinterlegten Elemente markieren die Bestandteile des aktuellen BDAS, die violett hinterlegten zeigen alternative Implementierungen auf der jeweiligen Schicht. Grün schraffiert ist die Applikationsschicht, in der Applikationen oberhalb von Spark und dessen direkten Bibliotheken angesiedelt sind.

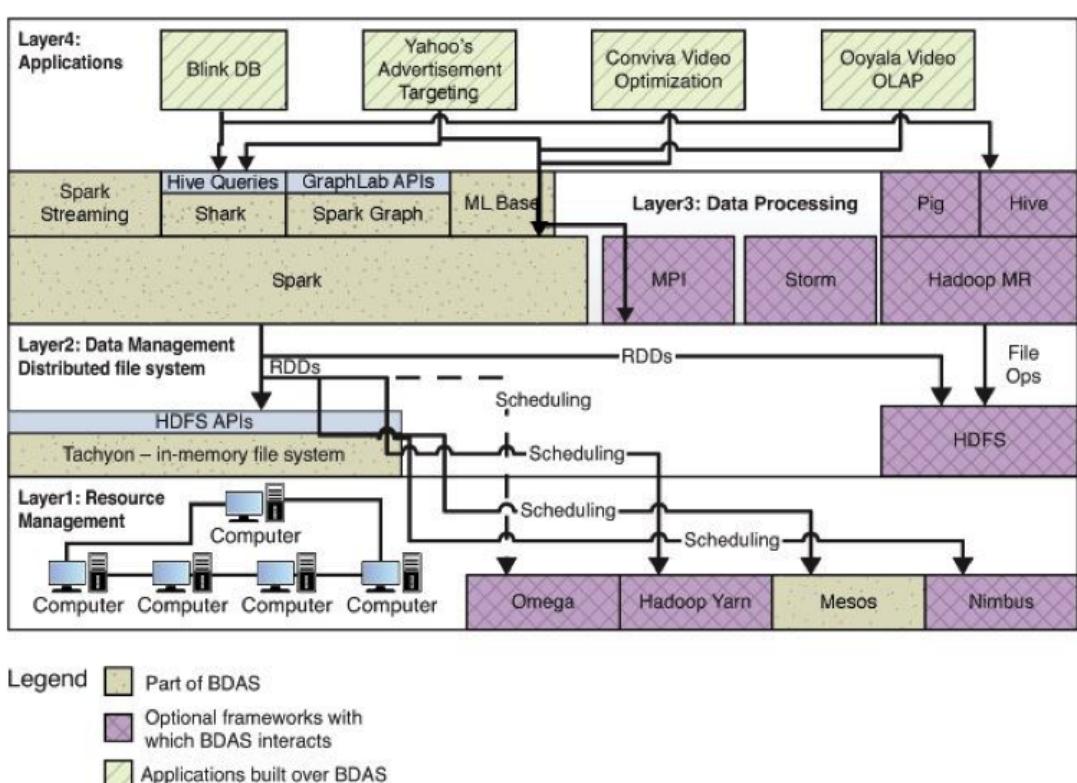


Abbildung 6.1: Der BDAS. Abbildung aus „Big Data Analytics Beyond Hadoop“, S 15 [Agn14]

Der BDAS beginnt in der untersten Schicht mit Mesos oder einer seiner Alternativen. Für eine lokale Installation von Spark wird allerdings in der Regel keine eigenes Cluster-Resource-

Management-System benötigt, da Spark die Grundfunktionalitäten für die Ressourcenverwaltung selbst im Kernel implementiert hat.

Prinzipiell ist zunächst zu entscheiden, ob Spark lokal als Single- oder Multinode-Cluster betrieben werden soll. Diese Entscheidung sollte auch maßgeblich von der zur Verfügung stehenden Hardware abhängig gemacht werden und von den Einsatzgebieten. Sind die Tasks gut parallelisierbar, bietet sich ein Cluster aus Master und $1 \dots n$ Worker-Nodes an, ansonsten sollte den einzelnen Knoten oder einem Single-Node-Cluster möglichst viel Hauptspeicher zur Verfügung gestellt werden um die Festspeicherzugriffe zu minimieren.

Da Spark durch seine beiden Hauptmerkmale *In-Memory-Computation* und *massive Parallelverarbeitung* eine möglichst starke Hardwareinfrastruktur benötigt und hier besonders die Aspekte Hauptspeicher, Prozessorkerne, Knotenzahl und Netzwerkperformance essentiell sind, sollte generell eine lokale Spark-Installation so einfach wie möglich aufgebaut werden. Der Fokus sollte hier auf Entwicklungstätigkeit, Debugging, und Vergleichsmessungen gelegt werden.

Für die lokale Installation besteht die Möglichkeit, Spark nativ auf dem System zu installieren, oder in virtualisierten Umgebungen. Eine native Installation hat den Vorteil, dass Spark eine systemnahe JVM als Ausführungsumgebung zum Einsatz kommt. So kann ein Großteil der vorhandenen Systemressourcen für die Spark-Ausführung zur Verfügung gestellt werden. Eine native Installation lässt sich entweder als Single-Node (Master-Only) konfigurieren, oder als virtuelles Cluster mit einem Master und einem Worker-Node. Hier empfiehlt sich ein Hybridbetrieb. Für Debugging-Aufgaben ist auf Grund der Taskverteilung von Spark nur ein Single-Node-Einsatz praktikabel. Für Skalierbarkeitstests bietet sich das Setup mit einem Master und einem Worker-Node an.

Als Alternativen zu dieser nativen Installation lässt sich ein theoretisch beliebig großes virtuelles Cluster auf einer lokalen Umgebung mit einer entsprechenden Anzahl von virtuellen Maschinen realisieren. Dies hat den Vorteil, dass die Skalierbarkeit der Anwendung durch entsprechende Partitionierung der RDDs besser testbar ist. Allerdings haben virtuelle Maschinen in der Regel einen sehr hohen Ressourcenverbrauch und lassen so nur bedingt Rückschlüsse auf das tatsächliche Laufzeitverhalten zu.

Eine weitere Möglichkeit des Setups ist das Deployment der kompletten Spark-Infrastruktur in einen Docker-Container. Diese Möglichkeit wird im folgenden Unterkapitel 6.2 beschrieben.

6.2 Ausführungscontainer: Docker

Bei Docker handelt es sich um eine Open-Source-Plattform zur Automatisierung des Software-Deployments innerhalb von Ausführungscontainern [Doc15]. Prinzipiell legt Docker eine weitere Abstraktionsschicht über ein vorhandenes Linux-System und nutzt dessen *Resource-Isolation-Features*, wie beispielsweise *cgroups*³ [VN15].

Durch Docker werden isolierte Prozesse mittels *High-Level-API* in leichtgewichtigen Ausführungscontainern erzeugt. Der Vorteil gegenüber herkömmlichen Virtualisierungstechnologien, wie beispielsweise Oracle Virtual Box, Citrix XenServer, VMWare, besteht unter anderem darin, dass bei Docker im Gegensatz zu diesen Applikationen kein eigenes Betriebssystem in einer virtuellen Maschine installiert werden muss. Dadurch sind Docker-Container äußerst ressourcenschonend. Voraussetzung für einen Betrieb von Docker ist allerdings ein installiertes Linux-Betriebssystem auf dem Hostrechner. Docker bietet die Möglichkeit, relativ kleine Services in eigenen Containern unterzubringen. Die Prinzipien einer *Microservice-Infrastruktur*⁴, allen voran die *Immutability*⁵, können so mittels Docker-Containern umgesetzt werden (Vergleich [Doc15]).

Docker ist sehr gut geeignet, eine lokale Apache Spark Infrastruktur aufzusetzen und diese inklusive aller Abhängigkeiten und deployten Anwendungen sowohl an andere Standalone-, als auch an Clustersysteme auf sehr einfache Weise zu verteilen. Bei der lokalen Installation mit Docker-Containern besteht die Wahl zwischen einer portablen Single-Node-Umgebung und einem virtualisierten Cluster. Prinzipiell gelten für den Fall der Docker-Installation die gleichen Bedingungen, wie für eine native Installation. Darüber hinaus lässt ein Setup mit Docker-Containern auch mehr als einen Worker-Node zu. Doch auch hier muss beachtet werden, dass Spark seine Stärken beim In-Memory-Processing nur mit ausreichenden Ressourcen nutzen kann. Auch wenn Docker eine sehr leichtgewichtige Virtualisierungslösung darstellt, muss dennoch beachtet werden, dass hier nicht die vollen Ressourcen wie bei einer nativen Installation zur Verfügung stehen.

Bereits konfigurierte Docker-Container können in einer eigenen Plattform, dem sogenannten Docker Hub verwaltet, mit anderen Nutzern geteilt und heruntergeladen werden.

Im Folgenden wird exemplarisch gezeigt, wie ein Docker Container für Apache Spark aufgebaut, konfiguriert und deployt werden kann [Vya15]. Dieser Container wurde auf einem Windows Hostrechner innerhalb einer VirtualBox-Instanz erstellt. Diese virtuelle Maschine wurde mit

³ Bei cgroups handelt es sich um eines der Linux-Kernel-Features, um Ressourcen (CPU, Speicher, I/O, etc.) für verschiedene Prozesse isolieren zu können. Weitere RIFs sind *capabilities*, *namespaces*, *SELinux*, *Netlink*, *Netfilter*, *AppArmor* (Vergleich [SS14])

⁴ Microservices sind ein Designpattern in der Softwarearchitektur, in dem komplexe Anwendungen in kleine, unabhängige Services ausgelagert werden. Diese können mittels verschiedener Sprachen entwickelt werden und stellen ihre jeweilige API den Konsumenten zur Verfügung.

⁵ Unter Immutability versteht man in der objektorientierten und funktionalen Programmierung, dass ein Objekt nach dessen Erzeugung nicht mehr modifiziert werden kann.

einer leichtgewichtigen CentOS-Installation aufgebaut.

Da CentOS auf einer RedHat-Linuxdistribution aufsetzt, wird als Paketmanager *YUM* eingesetzt. Für die einfache Provisionierung der Docker Container bietet sich die Nutzung von Vagrant an [Has15]. Dies ist eine Konfigurations- und Provisionierungslösung für virtuelle Maschinen und bietet seit Version 1.6 eine native Unterstützung von Docker. So lassen sich die Docker Container mittels entsprechender Scripts einfach konfigurieren und auf beliebige Umgebungen verteilen.

In Listing 6.1 wird zunächst eine JVM im Container aufgesetzt. Diese wird von Spark als Laufzeitumgebung benötigt. Die Besonderheit an der unten dargestellten Variante ist die Ausführung von Spark als sogenanntes *Tarball*. Dies entspricht dem Vorgehen für Microservices und erlaubt eine sehr schlanke Konfiguration des Containers (Vergleich [Lam15]).

Listing 6.1: Setup Spark mit eigener JVM

```

1 FROM jvm
2 RUN yum clean all
3 RUN yum install -y tar yum-utils wget
4 RUN yum-config-manager --save
5 RUN yum update -y
6 RUN yum install -y java-1.8.0-openjdk-devel.x86_64
7 COPY spark-1.2.0-bin-hadoop2.4.tgz /opt/
8 RUN tar -xzf /opt/spark-1.2.0-bin-hadoop2.4.tgz -C /opt/
9 RUN echo "SPARK_HOME=/opt/spark-1.2.0-bin-hadoop2.4" >> /etc/environment
10 RUN echo "JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk/" >> /opt/spark
    -1.2.0-bin-hadoop2.4/conf/spark-env.sh

```

Listing 6.2: Vagrant Provisionierungs-Skript für Spark

```

1 # First instance is spark master.
2 $spark_num_instances = 2
3 Vagrant.configure("2") do |config|
4   # nodes definition
5   (1..$spark_num_instances).each do |i|
6     config.vm.define "scale#{$i}" do |scale|
7       scale.vm.provider "docker" do |d|
8         d.build_dir = "spark/"
9         d.name = "node#{$i}"
10        d.create_args = ["--privileged=true"]
11        d.remains_running = true
12        if "#{$i}" == "1"
13          d.ports = [ "4040:4040", "7704:7704" ]
14        else
15          d.create_args = d.create_args << "--link" << "node1:
16          node1.dockerspark"
17        end
18      end
19      scale.vm.synced_folder "./", "/spark_infrastructure/"

```

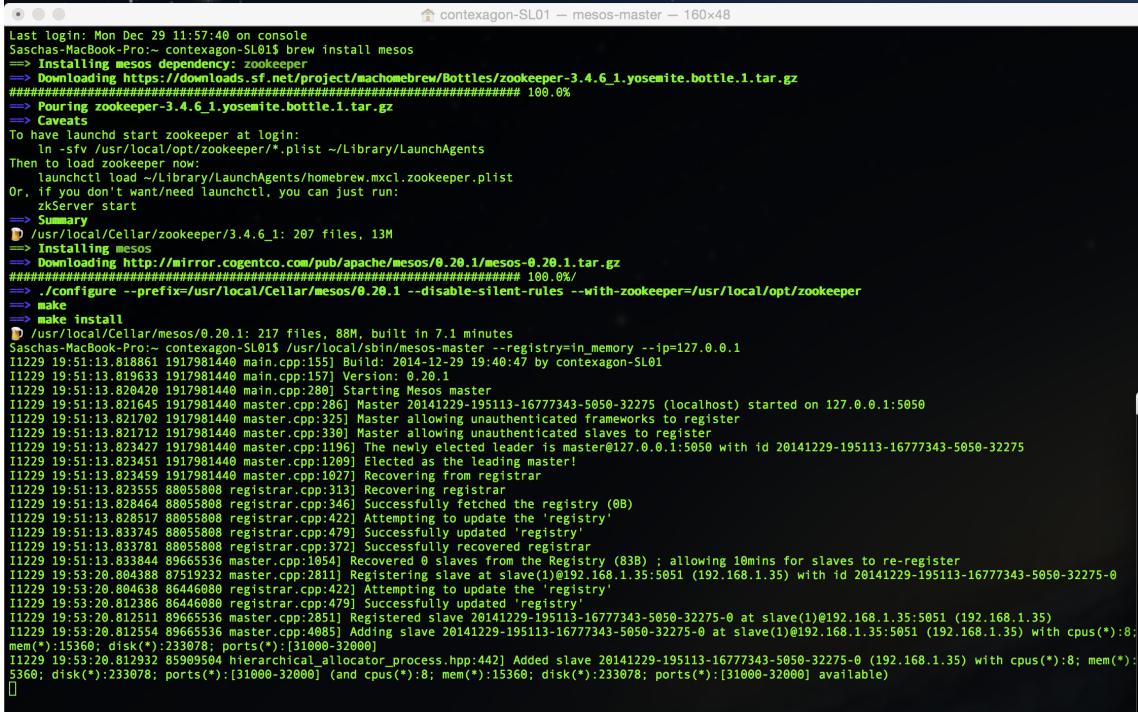
```

19     scale.vm.hostname = "node#{i}.dockerspark"
20   end
21 end

```

6.3 Cluster Management: Mesos

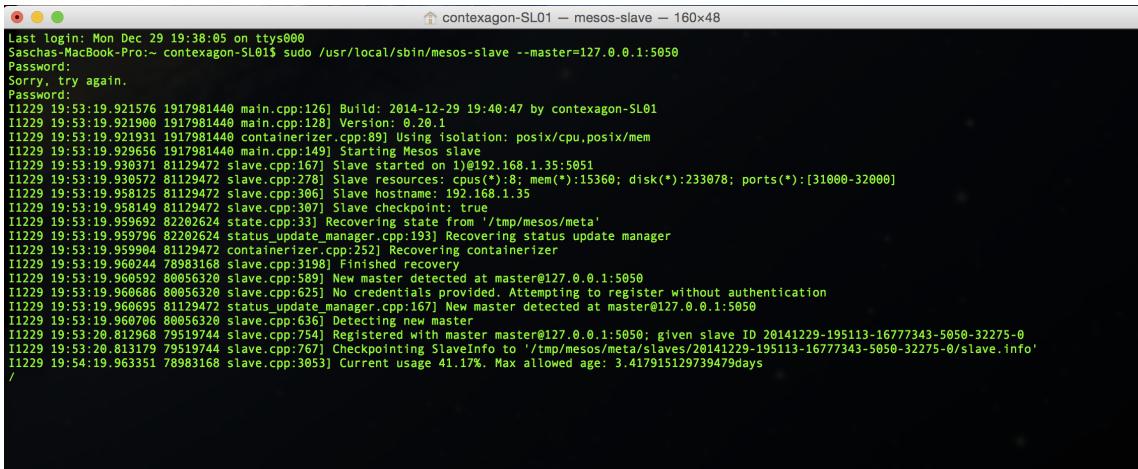
TBD!



```

Last login: Mon Dec 29 11:57:40 on console
Saschas-MacBook-Pro:~ kontexagon-SL01$ brew install mesos
=> Installing mesos dependency: zookeeper
=> Downloading https://downloads.sf.net/project/machomebrew/Bottles/zookeeper-3.4.6_1.yosemite.bottle.1.tar.gz
#####
=> Pouring zookeeper-3.4.6_1.yosemite.bottle.1.tar.gz
=> Caveats
To have launchd start zookeeper at login:
  ln -sfv /usr/local/opt/zookeeper/*-plist ~/Library/LaunchAgents
Then to load zookeeper now:
  launchctl load ~/Library/LaunchAgents/homebrew.mxcl.zookeeper.plist
Or, if you don't want/need launchctl, you can just run:
  zkServer start
=> Summary
  D /usr/local/Cellar/zookeeper/3.4.6_1: 207 files, 13M
=> Installing mesos
=> Downloading http://mirror.cogentco.com/pub/apache/mesos/0.20.1/mesos-0.20.1.tar.gz
#####
=> ./configure --prefix=/usr/local/Cellar/mesos/0.20.1 --disable-silent-rules --with-zookeeper=/usr/local/opt/zookeeper
=> make
=> make install
  D /usr/local/Cellar/mesos/0.20.1: 217 files, 88M, built in 7.1 minutes
Saschas-MacBook-Pro:~ kontexagon-SL01$ /usr/local/sbin/mesos-master --registry=in_memory --ip=127.0.0.1
I1229 19:51:13 818861 1917981440 main.cpp:128] Build: 2014-12-29 19:40:47 by kontexagon-SL01
I1229 19:51:13 819631 1917981440 main.cpp:157] Version: 0.20.1
I1229 19:51:13 820426 1917981440 master.cpp:280] Starting Mesos master
I1229 19:51:13 821645 1917981440 master.cpp:286] Master 20141229-195113-16777343-5050-32275 (localhost) started on 127.0.0.1:5050
I1229 19:51:13 821702 1917981440 master.cpp:325] Master allowing unauthenticated frameworks to register
I1229 19:51:13 821712 1917981440 master.cpp:330] Master allowing unauthenticated slaves to register
I1229 19:51:13 823427 1917981440 master.cpp:1196] The newly elected leader is master@127.0.0.1:5050 with id 20141229-195113-16777343-5050-32275
I1229 19:51:13 823451 1917981440 master.cpp:1209] Elected as the leading master!
I1229 19:51:13 823459 1917981440 master.cpp:1227] Recovering from registrar
I1229 19:51:13 823555 88955808 registrar.cpp:313] Recovering registrar
I1229 19:51:13 828464 88955808 registrar.cpp:346] Successfully fetched the registry (0B)
I1229 19:51:13 828517 88955808 registrar.cpp:422] Attempting to update the 'registry'
I1229 19:51:13 833745 88955808 registrar.cpp:479] Successfully updated 'registry'
I1229 19:51:13 823781 88955808 registrar.cpp:372] Successfully recovered registrar
I1229 19:51:13 833844 89665536 master.cpp:1054] Recovered 0 slaves from the Registry (83B) ; allowing 10mins for slaves to re-register
I1229 19:53:28 804388 87519232 master.cpp:2811] Registering slave at slave(1)@192.168.1.35:5051 (192.168.1.35) with id 20141229-195113-16777343-5050-32275-0
I1229 19:53:28 804638 864460888 registrar.cpp:422] Attempting to update the 'registry'
I1229 19:53:28 812386 864460888 registrar.cpp:479] Successfully updated 'registry'
I1229 19:53:28 812811 89665536 master.cpp:2851] Registered slave 20141229-195113-16777343-5050-32275-0 at slave(1)@192.168.1.35:5051 (192.168.1.35)
I1229 19:53:28 812811 89665536 master.cpp:40851] Added slave 20141229-195113-16777343-5050-32275-0 at slave(1)@192.168.1.35:5051 (192.168.1.35) with cpus(*):8; mem(*):5360; disk(*):233078; ports(*):[31000-32000]
I1229 19:53:20 812932 85905904 hierarchical_allocator_process.hpp:442] Added slave 20141229-195113-16777343-5050-32275-0 (192.168.1.35) with cpus(*):8; mem(*):5360; disk(*):233078; ports(*):[31000-32000] (and cpus(*):8; mem(*):15360; disk(*):233078; ports(*):[31000-32000] available)
  
```

Abbildung 6.2: Build, Installation und Start des Mesos Masternode



```

Last login: Mon Dec 29 19:38:05 on ttys000
Saschas-MacBook-Pro:~ kontexagon-SL01$ sudo /usr/local/sbin/mesos-slave --master=127.0.0.1:5050
Password:
Sorry, try again.
Password:
I1229 19:53:19 921576 1917981440 main.cpp:126] Build: 2014-12-29 19:40:47 by kontexagon-SL01
I1229 19:53:19 921900 1917981440 main.cpp:128] Version: 0.20.1
I1229 19:53:19 921931 1917981440 containerizer.cpp:89] Using isolation: posix/cpu,posix/mem
I1229 19:53:19 929656 1917981440 main.cpp:149] Starting Mesos slave
I1229 19:53:19 930371 81129472 slave.cpp:167] Slave started on 1)@192.168.1.35:5051
I1229 19:53:19 930572 81129472 slave.cpp:278] Slave resources: cpus(*):8; mem(*):15360; disk(*):233078; ports(*):[31000-32000]
I1229 19:53:19 958128 81129472 slave.cpp:306] Slave hostname: 192.168.1.35
I1229 19:53:19 958149 81129472 slave.cpp:307] Slave checkpoint: true
I1229 19:53:19 959692 82202624 state.cpp:33] Recovering state from '/tmp/mesos/meta'
I1229 19:53:19 959796 82202624 status_update_manager.cpp:193] Recovering status update manager
I1229 19:53:19 959984 81129472 containerizer.cpp:252] Recovering containerizer
I1229 19:53:19 960244 78983168 slave.cpp:3198] Finished recovery
I1229 19:53:19 960592 80656328 slave.cpp:589] New master detected at master@127.0.0.1:5050
I1229 19:53:19 960686 80656328 slave.cpp:625] No credentials provided. Attempting to register without authentication
I1229 19:53:19 960695 81129472 status_update_manager.cpp:167] New master detected at master@127.0.0.1:5050
I1229 19:53:19 960706 80656328 slave.cpp:636] Detecting new master
I1229 19:53:20 812968 79519744 slave.cpp:754] Registered with master master@127.0.0.1:5050; given slave ID 20141229-195113-16777343-5050-32275-0
I1229 19:53:20 813179 79519744 slave.cpp:767] Checkpointing SlaveInfo to '/tmp/mesos/meta/slaves/20141229-195113-16777343-5050-32275-0/slave.info'
I1229 19:54:19 963351 78983168 slave.cpp:3053] Current usage 41.17%. Max allowed age: 3.417915129739479days
  
```

Abbildung 6.3: Start und Registrierung eines Mesos Slaves

6.4 Caching-Framework: Tachyon

TBD!

6.5 Der eigentliche Kern: Apache Spark

TBD!

6.6 Streaming-Framework: Spark Streaming

TBD!

6.7 Abfrageschicht: Spark SQL

TBD!

6.8 Machine Learning Algorithmen: MLLib

Listing 6.3: SBT Script für Anwendungen mit Spark und MLLib-Dependencies

```
1 name := "SparkMLLibTest"
2
3 version := "1.0"
4
5 scalaVersion := "2.10.4"
6
7 libraryDependencies ++= Seq(
8   "org.apache.spark" % "spark-core_2.10" % "1.1.0",
9   "org.apache.spark" % "spark-mllib_2.10" % "1.1.0",
10  "com.github.scopt" % "scopt_2.9.3" % "3.3.0"
11 )
```

6.9 Graphenanwendungen: GraphX

TBD!

6.10 Alternativimplementierung zu MLlib: H2O

TBD!

6.11 Alternativimplementierung zu Spark: Apache Flink

TBD!

```
Sascha-MacBook-Pro:flink-0.8.0 contexagon-SL01$ ./bin/start-local.sh
Starting Job manager
Sascha-MacBook-Pro:flink-0.8.0 contexagon-SL01$ tail log/flink-*~jobmanager*.log
21:13:22,866 INFO org.apache.flink.runtime.taskmanager.TaskManager
21:13:22,866 INFO org.apache.flink.runtime.taskmanager.TaskManager
bytes.
21:13:23,213 INFO org.apache.flink.runtime.taskmanager.TaskManager
21:13:23,222 INFO org.apache.flink.runtime.taskmanager.TaskManager
ed/max)
21:13:23,229 INFO org.apache.flink.runtime.jobmanager.web.WebInfoServer
,0/bin/../conf/.../resources/web-docs-infoserver'.
21:13:23,230 INFO org.apache.flink.runtime.jobmanager.web.WebInfoServer
st, port 8081
21:13:23,314 INFO org.apache.flink.runtime.jobmanager.web.WebInfoServer
21:13:23,316 INFO org.eclipse.jetty.util.log
21:13:23,324 INFO org.apache.flink.runtime.instance.InstanceManager
1d521aaca8a@ebecf34e8e642. Current number of registered hosts is 1.
21:13:23,347 INFO org.eclipse.jetty.util.log
Sascha-MacBook-Pro:flink-0.8.0 contexagon-SL01$ ■
- Using 0.7 of the free heap space for managed memory.
- Initializing memory manager with 466 megabytes of memory. Page size is 32768
- Determined BLOB server address to be localhost/127.0.0.1:64053
- Memory usage stats: [HEAP: 543/736/736 MB, NON HEAP: 16/16/-1 MB (used/committed)
- Setting up web info server, using web-root directory '/Applications/flink-0.8
- Web info server will display information about nephele job-manager on localhost
- Starting web info server for JobManager on port 8081
- jetty-8.0.0.M1
- Registered TaskManager at localhost ((ipcPort=64055, dataPort=64056) as 70134d
- Started SelectChannelConnector@0.0.0.0:8081
```

Abbildung 6.4: Start des Apache Flink Masternode mit Anzeige des Logs

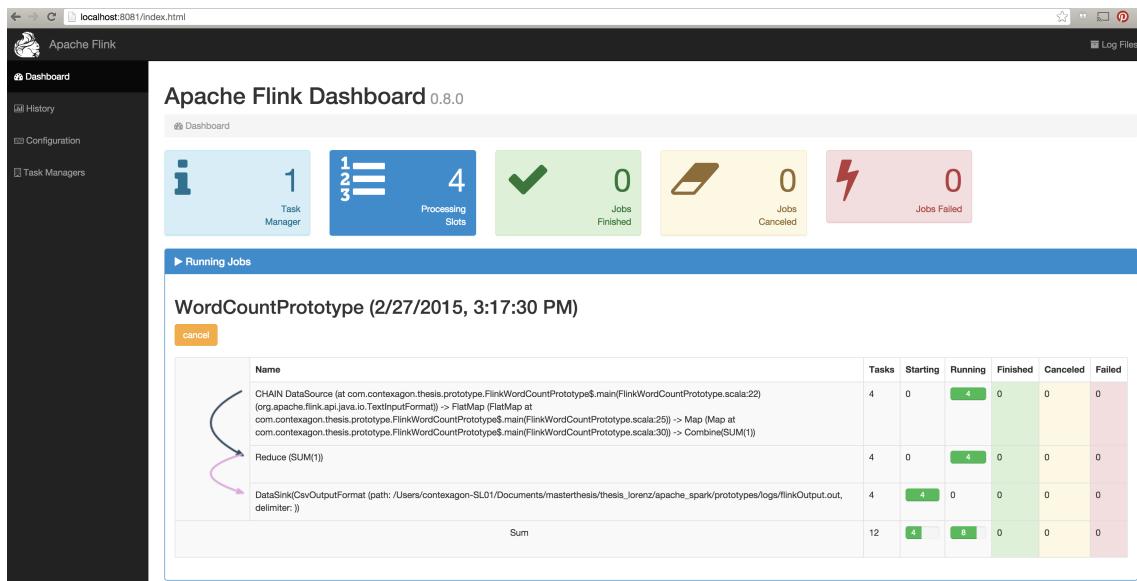


Abbildung 6.5: Das Apache Flink Dashboard auf der lokalen Installation

6.12 Zusammenfassung

TBD!

Kapitel 7

Implementierung der Prototypen

Im vorherigen Kapitel wurde gezeigt, wie verschiedene Entwicklungs- und Testinfrastrukturen mit Apache Spark aufgesetzt werden können. Im Rahmen dieser Thesis wurden für die Frameworks Spark und Flink, sowie für MLLib und H2O jeweils ein Prototyp für Vergleichsmessungen erstellt. Darüber hinaus wurden kleinere Testprototypen für die Bibliotheken Spark Streaming, sowie für GraphX erstellt. Diese werden im folgenden Kapitel vorgestellt.

7.1 Prototyp: Spark

Das Framework Apache Spark bietet durch seine APIs umfangreiche Möglichkeiten der applikatorischen Datenanalyse und Manipulation. Besonders Anwendungen auf große, persistierte Datenmengen lassen sich so komfortabel mittels Batch-Processing analysieren und verarbeiten.

Deshalb wurde zum Test der Frameworks Apache Spark und Flink auf eine möglichst große Datenbasis zurückgegriffen, auf der einige durch die APIs angebotene elementare Funktionen angewendet werden.

Zunächst wurde eine unstrukturierte Textdatei mit einer Größe von ca. 5 GB erstellt. Aus diesen Daten erstellt Spark ein RDD, verteilt es innerhalb der vorhandenen Infrastruktur und zählt alle vorkommenden Buchstaben e und s. Das folgende Listing zeigt diese Testanwendung in Scala.

Listing 7.1: SimpleTestApp.scala - zählt Buchstabenvorkommen in Textdateien.

```

1  /* SimpleTestApp.scala */
2  import org.apache.spark.SparkContext
3  import org.apache.spark.SparkContext._
4  import org.apache.spark.SparkConf
5  import com.codahale.metrics.Meter
6
7  object SimpleTestApp {
8    def main(args: Array[String]) {
9
10      val logFile = "acc.log" // filename of the Textfile
11      val conf = new SparkConf().setAppName("Simple Test Application")
12      val sc = new SparkContext(conf)
13
14      val t1 = System.currentTimeMillis
15      val logData = sc.textFile(logFile, 2).cache()
16      val t2 = System.currentTimeMillis
17
18      val numAs = logData.filter(line => line.contains("e")).count()
19      val numBs = logData.filter(line => line.contains("s")).count()
20      val t3 = System.currentTimeMillis
21
22      println("Count of e: %s, count ofd s: %s".format(numAs, numBs))
23      println("Creating RDD took " + (t2-t1) + " ms.")
24      println("Counting the chars took " + (t3-t2) + " ms.")
25    }
26  }

```

Diese kleine Testanwendung zeigt schnell die Performanceverhältnisse zwischen verschiedenen Infrastrukturen und eignet sich gut, um gegebenenfalls verschiedene Partitionierungsmaßnahmen zu überprüfen.

Der eigentliche Prototyp für Spark basiert auf dem Datenmodell von Wikipedia-Access-Logfiles von Wikibench [Wik14]. Hier finden sich Zugriffsdaten von Wikipedia aus den Monaten September 2007 bis einschließlich Januar 2008. Insgesamt handelt es sich um ca. 600 GB Daten.

Jeder Datensatz ist folgendermaßen aufgebaut:

- monoton steigender Zähler, der zum Sortieren der Daten in chronologischer Reihenfolge benutzt werden kann
- millisekundengenauer Zeitstempel der Requests in Unix-Notation
- die abgefragte URL
- Flag zur Anzeige ob die Datenbank aufgrund des Requests ein Update erfahren hat.

Listing 7.2: Beispieleinträge der Wikipedia Access Logs.

```

1 983510486 1190167848.982 http://en.wikipedia.org/wiki/Money,_Money,
   _Money -
2 983510487 1190167848.986 http://es.wikipedia.org/wiki/Imperialismo -
3 983510489 1190167848.959 http://upload.wikimedia.org/wikipedia/en/thumb/
   e/e1/Viva_Las_Vegas.jpg/180px-Viva_Las_Vegas.jpg -
4 983510491 1190167848.984 http://es.wikipedia.org/skins-1.5/monobook/user
   .gif -

```

Die Prototypen lassen sich mittels Apache Maven [Mav15] und/oder sbt [sbt15] bauen. Da Flink spezielle Archetypes für Maven anbietet, ist es hier sinnvoll, sich einen Applikationsrahmen per Maven-Import erstellen zu lassen.

Listing 7.3: Konfigurationseinstellungen der lokalen Spark-Installation

```

1 # Default system properties included when running spark-submit.
2 # This is useful for setting default environmental settings.
3
4 # Example:
5 # spark.master           spark://master:7077
6 spark.eventLog.enabled    true
7 # spark.eventLog.dir      hdfs://namenode:8021/directory
8 # spark.serializer        org.apache.spark.serializer.
   KryoSerializer
9 spark.driver.memory       256m
10 spark.executor.memory     2g
11 # spark.executor.extraJavaOptions -XX:+PrintGCDetails -Dkey=value -
   Dnumbers="one two three"

```

Listing 7.4: sbt-Buildfile zum Bauen des Spark-WordCount-Prototyp

```

1
2 name := "SparkWordCountPrototype"
3
4 version := "1.0"
5
6 scalaVersion := "2.10.4"
7
8 libraryDependencies += "org.apache.spark" %% "spark-core" % "1.1.1"

```

Listing 7.5: Spark-WordCount-Prototyp mit Implementierung mittels MapReduce-Algorithmus

```

1 package com.contexagon.thesis.prototype
2 /* SparkWordCountPrototype.scala */
3 import org.apache.spark._
4 import org.apache.spark.SparkContext
5 import org.apache.spark.SparkContext._
6 import org.apache.spark.SparkConf

```



```

51     val t3 = System.currentTimeMillis
52         //counts.foreach(println)
53
54     counts.saveAsTextFile(destinationPath)
55     val t4 = System.currentTimeMillis
56
57     //file.saveAsTextFile(args(2))
58     sc.stop()
59
60     // print some performance data
61     println("Creating RDD took " + (t2-t1) + " ms.")
62     println("Wordcount took " + (t3-t2) + " ms.")
63     println("Writing files took " + (t4-t3) + " ms.")
64     println("Total time consumption: " + (t4-t1) + " ms.")
65
66 }
67
68 // parse call parameters
69 private def parseParameters(args: Array[String]): Boolean = {
70     if (args.length > 0) {
71         if (args.length == 2) {
72             sourcePath = args(0)
73             destinationPath = args(1)
74             true
75         } else {
76             System.err.println("Usage: WordCount <text path> <result path>")
77             false
78         }
79     } else {
80         System.out.println("Executing WordCount example with built-in
81                         default data.")
82         System.out.println("  Provide parameters to read input data from a
83                         file.")
84         System.out.println("  Usage: WordCount <text path> <result path>")
85         true
86     }
87 }
88
89 private var sourcePath: String = "/Applications/spark-1.1.0/
90 wikilogs_oct07/wikiall"
91 private var destinationPath: String = "/Users/contexagon-SL01/
92 Documents/masterthesis/thesis_lorenz/apache_spark/prototypes/logs/
93 sparkOutput.out"
94
95 }

```

Listing 7.6: ShellScript zum Start des Spark-WordCount-Prototyp

```

2 #! /bin/bash
3
4 ######
5 # Masterthesis Big Data Processing with Apache Spark
6 # Sascha P. Lorenz - Hochschule Emden-Leer, Beuth Hochschule Berlin
7 # Start Script for WordCount in Apache Spark for Performance Comparison
    Reasons
8 # OUTPUTDIR = Destination for performance logs
9 # INPUT_PATH = Location of the application
10 # APPLICATION = Name of the application
11 # CLASS = Fully qualified classname of the main method
12 ######
13
14 OUTPUTDIR=/Users/contexagon-SL01/Documents/masterthesis/thesis_lorenz/
    apache_spark/prototypes/logs
15 OUTPUT_LOG=$OUTPUTDIR/simpleapp.log
16 INPUT_PATH=/Users/contexagon-SL01/Documents/masterthesis/thesis_lorenz/
    apache_spark/prototypes
17 APPLICATION=/WordcountSparkPrototype/target/scala-2.10/
    sparkwordcountprototype_2.10-1.0.jar
18 CLASS="com.contexagon.thesis.prototype.SparkWordCountPrototype"
19
20 rm -r /Users/contexagon-SL01/Documents/masterthesis/thesis_lorenz/
    apache_spark/prototypes/logs/sparkOutput.out
21
22
23
24 COMMAND_TOP=top > $OUTPUT_LOG
25 COMMAND_PID="ps -ef | grep -v 'grep' | grep 'top'"
26
27
28
29 # uncomment this block for usage at multinode clusters
30 #for i in `cat hosts`;do
31 #    ssh $i $COMMAND_TOP
32 #    ssh $i "$COMMAND_PID | awk '{print \$2}' > $PID"
33 #done
34
35
36
37 ./bin/spark-submit --class $CLASS --master local[4]
    $INPUT_PATH$APPLICATION
38
39
40 pkill java
41 pkill top

```

7.1.1 Prototyp: Vergleich Prototyp Apache Flink

Listing 7.7: Konfigurationseinstellungen der lokalen Flink-Installation

```
1 jobmanager.rpc.address: localhost
2
3 jobmanager.rpc.port: 6123
4
5 jobmanager.heap.mb: 256
6
7 taskmanager.heap.mb: 2048
8
9 taskmanager.numberOfTaskSlots: 4
10
11 parallelization.degree.default: 4
```

Listing 7.8: WordCountPrototyp für Apache Flink als MapReduce-Implementierung

```
1 package com.contexagon.thesis.prototype
2
3 import org.apache.flink.api.scala._
4
5 /**
6  * Simple WordCountPrototype for Apache Flink
7  * Apache Foundation, Sascha P. Lorenz
8  * Purpose: performance comparisons between Apache Spark and Apache
9  * Flink
10 * Date: 03.01.2015
11 * Version: 1.0
12 */
13 object FlinkWordCountPrototype {
14
15     def main(args: Array[String]) {
16         if (!parseParameters(args)) {
17             // if application is called without any parms
18             sourcePath = "/Applications/spark-1.1.0/wikilogs_oct07/wikiall"
19             destinationPath = "/Users/contexagon-SL01/Documents/masterthesis/
20                           thesis_lorenz/apache_spark/prototypes/logs/flinkOutput.out"
21             return
22         }
23
24         val t1 = System.currentTimeMillis
25         val env = ExecutionEnvironment.getExecutionEnvironment
26         //val text = getTextDataSet(env)
27
28         val text = env.readTextFile(sourcePath)
29
30         val t2 = System.currentTimeMillis
```

```

31     // The actual word count algorithm as MapReduce implementation
32     val counts = text.flatMap {
33         _.toLowerCase.split("\\W+") filter {
34             _.nonEmpty
35         }
36     }
37     .map { word => (word, 1)}
38     .groupByKey(0)
39     .sum(1)
40
41
42     val t3 = System.currentTimeMillis
43     // write results as file
44     counts.writeAsCsv(destinationPath, "\n", " ")
45
46
47     env.execute("WordCountPrototype")
48     val t4 = System.currentTimeMillis
49
50     // print a few performance data
51     println("Creating Dataset took " + (t2-t1) + " ms.")
52     println("Wordcount took " + (t3-t2) + " ms.")
53     println("Writing files took " + (t4-t3) + " ms.")
54     println("Total time consumption: " + (t4-t1) + " ms.")
55
56 }
57
58 // parse call parameters
59 private def parseParameters(args: Array[String]): Boolean = {
60     if (args.length > 0) {
61         if (args.length == 2) {
62             sourcePath = args(0)
63             destinationPath = args(1)
64             true
65         } else {
66             System.err.println("Wrong call: FlinkWordCountPrototype <text
67                             path> <result path>")
68             false
69         }
70     } else {
71         System.out.println("  Reading default input file on local file
72                         system (wikilogs_all).")
73         true
74     }
75 }
76
77 private var sourcePath: String = "/Applications/spark-1.1.0/
78                             wikilogs_oct07/wikiall"
79 private var destinationPath: String = "/Users/contexagon-SL01/

```

```
    Documents/masterthesis/thesis_lorenz/apache_spark/prototypes/logs/
    flinkOutput.out"
78
79 }
```

Listing 7.9: Maven POM zum bauen von Flink-Applikationen als JAR und Fat-JAR inklusive aller Dependencies

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
4         maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6
7     <groupId>com.contextagon.thesis.prototype</groupId>
8     <artifactId>WordCountFlinkPrototype</artifactId>
9     <version>0.1</version>
10    <packaging>jar</packaging>
11
12    <name>FlinkWordCountPrototype</name>
13    <url>http://www.contextagon.com</url>
14
15    <repositories>
16        <repository>
17            <id>apache.snapshots</id>
18            <name>Apache Development Snapshot Repository</name>
19            <url>https://repository.apache.org/content/repositories/snapshots
20                /</url>
21            <releases>
22                <enabled>false</enabled>
23            </releases>
24            <snapshots>
25                <enabled>true</enabled>
26            </snapshots>
27        </repository>
28    </repositories>
29
30    <properties>
31        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
32    </properties>
33
34    <!-- These two requirements are the minimum to use and develop Flink.
35        -->
36    <dependencies>
37        <dependency>
38            <groupId>org.apache.flink</groupId>
39            <artifactId>flink-scala</artifactId>
40            <version>0.8.0</version>
41        </dependency>
42        <dependency>
```

```

40      <groupId>org.apache.flink</groupId>
41      <artifactId>flink-streaming-scala</artifactId>
42      <version>0.8.0</version>
43    </dependency>
44  </dependencies>
45
46  <!-- We use the maven-assembly plugin to create a fat jar that
47      contains all dependencies
48      except flink and it's transitive dependencies. The resulting fat-jar
49      can be executed
50      on a cluster. Change the value of Program-Class if your program
51      entry point changes. -->
52
53 <build>
54   <plugins>
55     <plugin>
56       <artifactId>maven-assembly-plugin</artifactId>
57       <version>2.4.1</version>
58       <configuration>
59         <descriptors>
60           <descriptor>src/assembly/flink-fat-jar.xml</descriptor>
61         </descriptors>
62       <archive>
63         <manifestEntries>
64           <Program-Class>org.myorg.FlinkWordCountPrototype.Job</
65             Program-Class>
66         </manifestEntries>
67       </archive>
68     </configuration>
69     <executions>
70       <execution>
71         <id>make-assembly</id>
72         <phase>package</phase>
73         <goals>
74           <goal>single</goal>
75         </goals>
76       </execution>
77     </executions>
78   </plugin>
79
80   <plugin>
81     <groupId>org.apache.maven.plugins</groupId>
82     <artifactId>maven-compiler-plugin</artifactId>
83     <version>3.1</version>
84     <configuration>
85       <source>1.6</source>

```

```
86          <target>1.6</target>
87      </configuration>
88  </plugin>
89  <plugin>
90      <groupId>net.alchim31.maven</groupId>
91      <artifactId>scala-maven-plugin</artifactId>
92      <version>3.1.4</version>
93      <executions>
94          <execution>
95              <goals>
96                  <goal>compile</goal>
97                  <goal>testCompile</goal>
98              </goals>
99          </execution>
100     </executions>
101   </plugin>
102
103  <!-- Eclipse Integration -->
104  <plugin>
105      <groupId>org.apache.maven.plugins</groupId>
106      <artifactId>maven-eclipse-plugin</artifactId>
107      <version>2.8</version>
108      <configuration>
109          <downloadSources>true</downloadSources>
110          <projectnatures>
111              <projectnature>org.scala-ide.sdt.core.scalanature</
112                  projectnature>
113              <projectnature>org.eclipse.jdt.core.javanature</
114                  projectnature>
115          </projectnatures>
116          <buildcommands>
117              <buildcommand>org.scala-ide.sdt.core.scalabuilder</
118                  buildcommand>
119          </buildcommands>
120          <classpathContainers>
121              <classpathContainer>org.scala-ide.sdt.launching.
122                  SCALA_CONTAINER
123              </classpathContainer>
124              <classpathContainer>org.eclipse.jdt.launching.JRE_CONTAINER
125              </classpathContainer>
126          </classpathContainers>
127          <excludes>
128              <exclude>org.scala-lang:scala-library</exclude>
129              <exclude>org.scala-lang:scala-compiler</exclude>
130          </excludes>
131          <sourceIncludes>
132              <sourceInclude>**/*.scala</sourceInclude>
133              <sourceInclude>**/*.java</sourceInclude>
134          </sourceIncludes>
135      </configuration>
```

```

132      </plugin>
133
134      <!-- Adding scala source directories to build path -->
135      <plugin>
136          <groupId>org.codehaus.mojo</groupId>
137          <artifactId>build-helper-maven-plugin</artifactId>
138          <version>1.7</version>
139          <executions>
140              <!-- Add src/main/scala to eclipse build path -->
141              <execution>
142                  <id>add-source</id>
143                  <phase>generate-sources</phase>
144                  <goals>
145                      <goal>add-source</goal>
146                  </goals>
147                  <configuration>
148                      <sources>
149                          <source>src/main/scala</source>
150                      </sources>
151                  </configuration>
152              </execution>
153              <!-- Add src/test/scala to eclipse build path -->
154              <execution>
155                  <id>add-test-source</id>
156                  <phase>generate-test-sources</phase>
157                  <goals>
158                      <goal>add-test-source</goal>
159                  </goals>
160                  <configuration>
161                      <sources>
162                          <source>src/test/scala</source>
163                      </sources>
164                  </configuration>
165              </execution>
166          </executions>
167      </plugin>
168  </plugins>
169 </build>
170 </project>

```

Listing 7.10: Shellscrip zum Start des Flink WordCountPrototyp

```

1 #! /bin/bash
2
3 ######
4 # Masterthesis Big Data Processing with Apache Spark
5 # Sascha P. Lorenz - Hochschule Emden-Leer, Beuth Hochschule Berlin
6 # Start Script for WordCount in Apache Flink for Performance Comparison
    Reasons
7 # OUTPUTDIR = Destination for performance logs

```

```

8 # INPUT_PATH = Location of the application
9 # APPLICATION = Name of the application
10 # CLASS = Full qualified classname of the main method
11 ##########
12
13 OUTPUTDIR=/Users/contexagon-SL01/Documents/masterthesis/thesis_lorenz/
    apache_spark/prototypes/logs
14 OUTPUT_LOG=$OUTPUTDIR/simpleapp.log
15 INPUT_PATH=/Users/contexagon-SL01/Documents/masterthesis/thesis_lorenz/
    apache_spark/prototypes
16 APPLICATION=/WordcountSparkPrototype/target/scala-2.10/
    sparkwordcountprototype_2.10-1.0.jar
17 CLASS="com.contexagon.thesis.prototype.SparkWordCountPrototype"
18
19 rm -r /Users/contexagon-SL01/Documents/masterthesis/thesis_lorenz/
    apache_spark/prototypes/logs/flinkOutput.out
20
21 COMMAND_TOP=top > $OUTPUT_LOG
22 COMMAND_PID="ps -ef | grep -v 'grep' | grep 'top'"
23
24
25
26 # uncomment this block for usage at multinode clusters
27 #for i in `cat hosts`;do
28 #    ssh $i $COMMAND_TOP
29 #    ssh $i "$COMMAND_PID | awk '{print \$2}' > $PID"
30 #done
31
32
33
34 ./bin/flink run /Users/contexagon-SL01/Documents/masterthesis/
    thesis_lorenz/apache_spark/prototypes/flink_tests/
    WordCountPrototypeFlink/target/WordCountFlinkPrototype-0.1.jar -c "
    com.contexagon.thesis.prototype.FlinkWordCountPrototype" -v
35
36 pkill java
37 pkill top

```

TBD Extrakt - Rest in Anhang

7.2 Prototyp: MLLib

7.2.1 Prototyp: Vergleich Prototyp H2O

[Cli15]

7.3 Prototyp: Spark Streaming

TBD!

7.4 Prototyp: GraphX

TBD!

7.5 Zusammenfassung

TBD!

Kapitel 8

Evaluierung der Komponenten und Alternativen

Im vorhergehenden Kapitel wurden die Implementierungen von Prototypen für einzelne Bibliotheken des BDAS vorgestellt. Nachfolgend werden diese Bibliotheken hinsichtlich ihres spezifischen Laufzeitverhaltens untersucht. Zu diesem Zweck wurden für die einzelnen Anwendungsbereiche zunächst Metriken definiert, die Aussagen über die relative Leistungsfähigkeit zulassen.

Für diese Beurteilung wurden unterschiedliche Umgebungen eingesetzt, wobei hier die relativen Performancewerte gegenüber den absoluten priorität betrachtet werden.

Des weiteren wurden auch funktionale Anforderungen für die Evaluierung der Frameworks und Bibliotheken definiert. Dies hat besondere Relevanz für die möglichst direkten Vergleiche der Frameworks Spark und Flink, sowie MLLib und H2O.

8.1 Definition von Metriken für die Bibliotheken des BDAS

Um die verschiedenen Bibliotheken des BDAS möglichst einheitlich und dennoch nutzungsspezifisch testen zu können, wurden im Rahmen dieser Thesis entsprechende Metriken definiert. Diese sind mehrstufig angeordnet, so dass ein vergleichbares Subset über alle Nutzungsfälle angelegt werden konnte. In einer zweiten Schicht sind zusätzliche, anwendungsfallspezifische Metriken angesiedelt.

Diese Metriken teilen sich zum einen prinzipiell auf in funktionale und nichtfunktionale Anforderungen. Die funktionalen Anforderungen beschreiben den Funktionsumfang und die Umsetzungsstrategie bestimmter Anforderungen. Die nichtfunktionalen Anforderungen betrachten Aspekte wie Performance auf verschiedenen Ausführungsstufen und nach $1 \dots n$ Iterationen, Replikationsmechanismen, Verhalten im Fehlerfall. Um diese zu ermitteln, werden Verarbeitungs-

zeiten der Algorithmen und Ressourcenverbrauch während der Ausführung betrachtet. Oberhalb von diesen Metriken sind die speziellen Anwendungsfallmetriken. Hier werden beispielsweise die Nachrichten-Durchsatzraten von Spark Streaming oder die Güte und Performance der erlernten Modelle von MLLibs und H2O quantifiziert.

8.2 Beschreibung der Messverfahren

Da die Testläufe zur Performancemessung auf verschiedenen Linux, bzw. Unix-Systemen wie Debian und Mac OS X durchgeführt wurden, wurde auf die Verwendung proprietärer Performance-Monitoring Software oder die OS-Erweiterung DTrace [Gre15] aus Kompatibilitäts- und Vergleichbarkeitsgründen bewusst verzichtet. Diese bieten zwar zum Teil äußerst differenzierte Messverfahren an, jedoch wurde bei der Definition der Messverfahren für diese Thesis besonderes Augenmerk auf die Vergleichbarkeit über die Systemgrenzen hinweg gelegt.

Zur Messung des Ressourcenverbrauchs wurden Linux/Unix Shell Scripts auf Basis von Standardbefehlen angelegt, die folgende Anforderungen erfüllen:

- Messung muss auf allen Testumgebungen gleichermaßen lauffähig sein.
- Möglichst geringer Ressourcenverbrauch.
- Es dürfen keine Ressourcen der Laufzeitumgebung (JVM) konsumiert werden.
- Testapplikation inklusive Framework muss mit Messung simultan gestartet und gestoppt werden.
- Ergebnisse müssen in Logs persistiert werden.

Das Linux/Unix Kommando *top* [Abo15] bietet hier die nötige Granularität und ist auf allen eingesetzten Infrastrukturen ohne Erweiterung der Nutzungsrechte verfügbar. In Abbildung 8.1 lässt sich die Standardausgabe von *top* erkennen.

Eine Einschränkung dieses Kommandos ist die fehlende Identifikation der benutzten Prozessorkerne bei Verwendung von Multicore-Systemen. Die prozentuale Prozessorauslastung wird pro Kern angegeben. Am Beispiel des unter anderem eingesetzten MacBook Pro mit i7 Prozessor (vier physische Kerne, acht mit Hyperthreading) bedeutet dies, dass eine Maximalauslastung der CPU bei dieser Metrik einen Wert von 800% erreicht. Außerdem lässt sich bei einer dargestellten Auslastung von 100% so nicht ermitteln, ob es sich bei diesem Wert um einen Thread mit 100% Auslastung eines Kernes, um zwei Threads mit 80% und 20%, oder um vier unabhängige Threads mit je 25% Kernauslastung handelt. Aus diesem Grund werden für die Metriken zusätzlich zu der prozentualen Angabe der CPU-Auslastung die CPU-Zeit als

```
top - 18:16:49 up 70 days, 4:11, 2 users, load average: 0.00, 0.01, 0.00
Tasks: 608 total, 1 running, 607 sleeping, 0 stopped, 0 zombie
Cpu0 : 1.0%us, 0.3%sy, 0.0%ni, 98.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu1 : 0.7%us, 0.3%sy, 0.0%ni, 99.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu2 : 1.0%us, 1.0%sy, 0.0%ni, 98.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu3 : 0.3%us, 0.7%sy, 0.0%ni, 99.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu4 : 1.7%us, 1.3%sy, 0.0%ni, 97.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu5 : 1.7%us, 0.3%sy, 0.0%ni, 98.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu6 : 0.7%us, 0.3%sy, 0.0%ni, 99.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu7 : 1.0%us, 1.0%sy, 0.0%ni, 98.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu8 : 1.0%us, 0.3%sy, 0.0%ni, 98.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu9 : 0.3%us, 0.3%sy, 0.0%ni, 99.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu10 : 0.3%us, 0.3%sy, 0.0%ni, 99.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu11 : 0.3%us, 0.3%sy, 0.0%ni, 99.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu12 : 1.0%us, 0.3%sy, 0.0%ni, 98.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu13 : 0.7%us, 0.7%sy, 0.0%ni, 98.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu14 : 0.3%us, 0.0%sy, 0.0%ni, 99.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu15 : 0.3%us, 0.3%sy, 0.0%ni, 99.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 132138348k total, 125513584k used, 6624764k free, 686192k buffers
Swap: 4149244k total, 0k used, 4149244k free, 88896408k cached

```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3154	cloudera	20	0	10.8g	899m	21m	S	3.6	0.7	2949:43	java
14119	oozie	20	0	4981m	591m	19m	S	3.0	0.5	1571:37	java
3069	cloudera	20	0	5623m	586m	19m	S	2.6	0.5	2889:28	java
3096	cloudera	20	0	9721m	1.1g	169m	S	2.0	0.8	9226:00	java
12905	yarn	20	0	1863m	518m	16m	S	2.0	0.4	858:46.70	java
3210	cloudera	20	0	7005m	678m	108m	S	1.7	0.5	2501:32	java
235	root	39	19	0	0	0	S	1.0	0.0	1055:15	kipmi0
23957	slorenz	20	0	15440	1668	936	R	1.0	0.0	0:03.72	top
12763	mapred	20	0	891m	249m	16m	S	0.7	0.2	254:10.26	java
29208	hdfs	20	0	1710m	617m	16m	S	0.7	0.5	888:54.73	java
29407	spark	20	0	4357m	327m	20m	S	0.7	0.3	641:31.64	java
29568	hbase	20	0	1699m	489m	19m	S	0.7	0.4	712:53.41	java
29617	hbase	20	0	1639m	375m	19m	S	0.7	0.3	277:01.53	java
2558	root	20	0	0	0	0	S	0.3	0.0	23:06.10	kondemand/1
2567	root	20	0	0	0	0	S	0.3	0.0	20:25.03	kondemand/10
2571	root	20	0	0	0	0	S	0.3	0.0	19:52.34	kondemand/14
2987	cloudera	20	0	8359m	2004	672	S	0.3	0.0	9:54.41	postgres
3099	root	20	0	2950m	66m	4772	S	0.3	0.1	1135:24	python
3142	cloudera	20	0	2692m	216m	18m	S	0.3	0.2	154:58.76	java
13867	impala	20	0	31.6g	559m	25m	S	0.3	0.4	69:11.86	catalogd
29584	hbase	20	0	106m	1868	544	S	0.3	0.0	183:25.03	hbase.sh
1	root	20	0	19364	1472	1148	S	0.0	0.0	0:04.02	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd

Abbildung 8.1: Ausführung des Linux-Befehls top auf dem Master-Node des Beuth-Clusters.

Messgrößen verwendet. Diese ist proportional zur Prozessorauslastung. Für die durchgeföhrten Vergleichsmessungen werden die CPU-Zeiten der Standardimplementierung des BDAS mit 100% gewertet und die Vergleichsalgorithmen dazu in das entsprechende Verhältnis gesetzt.

Da Spark die RDDs wenn möglich im Hauptspeicher hält und nur auf Festspeicher auslagert, wenn das RAM nicht ausreichend ist, wird auf ein Monitoring des Hauptspeichers für diese Messungen verzichtet. Statt dessen werden die I/O-Vorgänge überwacht um so einen Überlauf auf den Festspeicher und damit begründete Performance-Einbrüche feststellen zu können.

8.3 Beschreibung der Messumgebungen

Zum Einsatz für die Evaluierung kamen für diese Untersuchung unterschiedliche Ansätze für Clusterinfrastrukturen. Es kamen diverse lokale Installationen, sowie ein leistungsfähiges Rechnercluster an der Beuth Hochschule Berlin zum Einsatz.

Die Tabelle 8.1 gibt eine detaillierte Übersicht über die lokal verwendeten Hardwarekonfi-

gurationen. Alle lokal eingesetzten Maschinen verfügen über SSDs (Solid State Drives) als Festspeicher.

System	CPU	Anzahl Cores	RAM	OS
MacBook Pro Mid 2014	Intel i7	4 physisch, 8 mit Hyperthreading	16 GB	Mac OS-X 10.10.2 Yosemite
Apple iMac Mid 2011	Intel i5	4 physisch, 8 mit Hyperthreading	16 GB	Mac OS-X 10.10.2 Yosemite
Xeon Workstation	Intel Xeon 1230	4 physisch, 8 mit Hyperthreading	32 GB	Windows 8.1, VirtualBox Instanzen mit CentOS
Lenovo ThinkPad 410T	Intel i5	4 physisch	8 GB	Windows 8.1, VirtualBox Instanzen mit CentOS

Tabelle 8.1: Übersicht der lokal verwendeten Hardware

Die Tabelle 8.2 zeigt die Konfiguration (Konfigurationsstand Dezember 2014) des verwendeten Clustersystems an der Beuth Hochschule Berlin.

System	CPU	Anzahl Cores	RAM	OS
Master Node	AMD 6320	2 * 8	128 GB	Debian Linux
Worker Node 1	AMD 6378	4 * 16	512 GB	Debian Linux
Worker Node 2	AMD 6378	4 * 16	512 GB	Debian Linux
Worker Node 3	AMD 6378	4 * 16	512 GB	Debian Linux

Tabelle 8.2: Übersicht über das Cluster der Beuth Hochschule Berlin

8.3.1 Lokales Single Node Cluster

TBD!

8.3.2 Lokales Multi Node Cluster

TBD!

8.3.3 Remote Cluster an der Beuth Hochschule

TBD!

8.4 Ergebnisse

TBD!

8.4.1 Messergebnisse Apache Spark

Apache Spark wurde mit den zwei im Kapitel 7 beschriebenen Prototypen gemessen. Die Messungen wurden gegen verschiedene Metriken durchgeführt. In Diagramm 8.2 wird exemplarisch eine solche Messung gezeigt. Hier wurde der Prototyp SimpleSparkPrototype, der das Vorkommen zweier verschiedener Buchstaben in einem unstrukturierten Text zur Aufgabe hat, gemessen. Es wurden Testdaten mit einer Größe von ca. 3,5 GB verwendet, die aus einem Subset der Zugriffslogs von Wikipedia bestehen.

Das Diagramm zeigt den Verlauf der CPU-Auslastung mit vier zugewiesenen Threads. Wie zu erkennen ist, werden die ersten rund 17 Sekunden der Gesamtaufzeit von 46 Sekunden für die Initialisierung und den Start von Spark benötigt. Dies nimmt verhältnismäßig viel Zeit in Anspruch, da hier sowohl der Spark-Context für die Verwaltung des Clusters, als auch der Webserver für die Überwachung des Spark-Clusters initialisiert werden. Die folgenden Messungen zeigen diese Ramp-Up-Phase von Spark nicht mehr, sondern starten erst mit dem Lesen der Quelldaten und dem Transformationsschritt.

8.4.2 Messergebnisse Apache Flink

TBD!

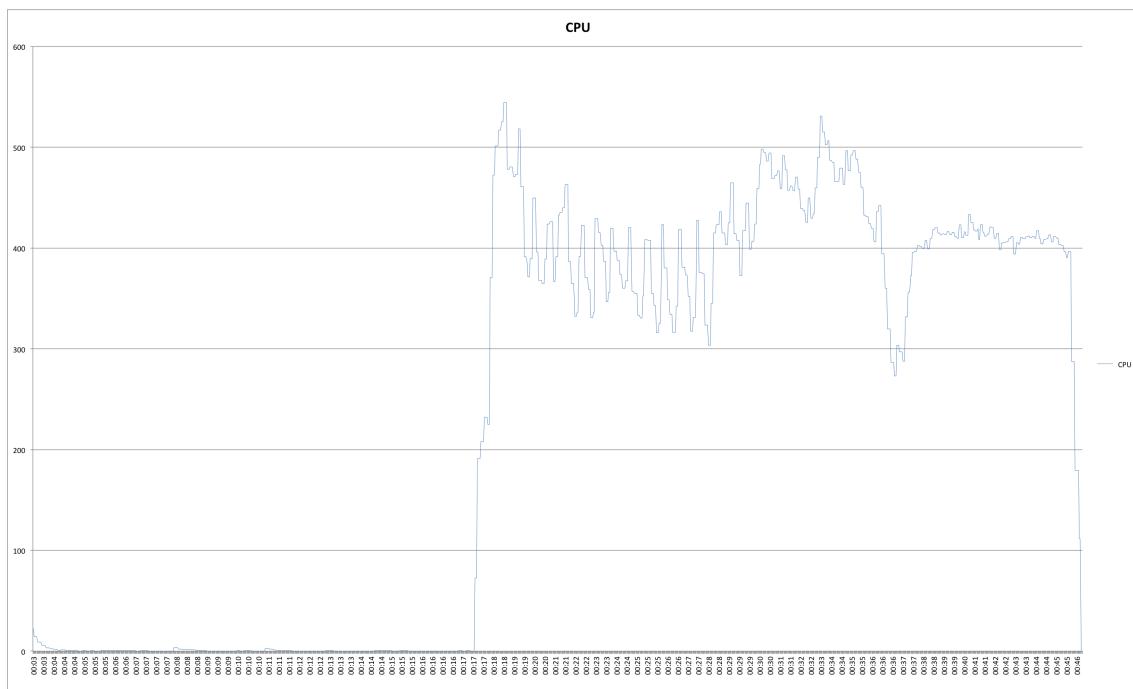


Abbildung 8.2: Messung der Laufzeit und CPU-Auslastung des CharCounters mit 4 Threads

8.4.3 Messergebnisse MLLib

TBD!

8.4.4 Messergebnisse H2O

TBD!

8.5 Zusammenfassung

TBD!

Kapitel 9

Schlussbetrachtung

TBD!

9.1 Zusammenfassung

TBD!

9.2 Ausblick

TBD!

Kapitel 10

Verzeichnisse

Literaturverzeichnis

- [Agn14] AGNEESWARAN, VIJAY: *Big Data Analytics Beyond Hadoop*. Pearson, 2014.
- [AST07] ANDREW S. TANENBAUM, MAARTEN VAN STEEN: *Distributed Systems - Principles and Paradigms*. Pearson, Prentice Hall, Second Edition Auflage, 2007.
- [Ben09] BENGIO, YOSHUA: *Learning Deep Architectures for AI*. Dept. IRO, Universite de Montreal, Canada. To appear in Foundations and Trends in Machine Learning, 2009.
- [Bis06] BISHOP, C.M.: *Pattern Recognition and Machine Learning*. Springer, 2006.
- [BJA06] BENJAMIN J. ANDERSON, DEBORAH S. GROSS, ET AL.: *Adapting K-Medians to Generate Normalized Cluster Centers*. Carleton College, Northfield Minnesota, 2006.
- [Bro98] BROSUS, FELIX: *SPSS 8: Professionelle Statistik*. International Thompson Publishing, 1998.
- [CBD08] CHUONG B DO, SERAFIM BATZOGLOU: *What is the expectation maximization algorithm?* Nature Publishing Group, 2008.
- [Cou13] COUNCIL, NATIONAL RESEARCH: *Frontiers in Massive Data Analysis*. National Academic Press, 2013.
- [CV14] CLAUS VIELHAUER, TOBIAS SCHEIDAT: *Fusion von biometrischen Verfahren zur Benutzerauthentifikation*. Otto-von-Guericke Universität Magdeburg Advanced Multimedia and Security Lab (AMSL), 2014.
- [DB11] DANAH BOYD, KATE CRAWFORD: *Six Provocations for Big Data*. Oxford Internet Institute, 2011.
- [GJ13] GARETH JAMES, DANIELA WITTEN, ET AL.: *An Introduction to Statistical Learning with Applications in R*. Springer, 4 Auflage, 2013.
- [HK15] HOLDEN KARAU, ANDY KOWINSKI, MATEI ZAHARIA: *Learning Spark - Lightning Fast Data Analysis*. O'Reilly, Early Release Auflage, 2015.

- [HS83] H.A. SIMON, P. Langley, G.L. BRADSHAW: *Rediscovering chemistry with the BACON system - Machine learning, an artificial intelligence approach*. Tiogra Publishing Co., 1983.
- [JA03] JOHANNES ASSFALG, CHRISTIAN BÖHM, ET. AL: *Knowledge Discovery in Databases*. Ludwig Maximilians Universität München, Institut für Informatik, 2003.
- [JC14] JÜRGEN CLEVE, UWE LÄMMEL: *Data Mining*. De Gruyter/Oldenbourg Wissenschaftsverlag, 1 Auflage, 2014.
- [JD04] JEFF DEAN, SANJAY GHEMAWAT: *MapReduce: Simplified Data Processing on Large Clusters*. Google, Inc., 2004.
- [JWS90] JUDE W. SHAVLIK, THOMAS G. DIETTERICH: *Reading in Machine Learning*. Morgan-Kaufman Publishers, Inc., 1 Auflage, 1990.
- [Kun14] KUNTAMUKKALA, ASHWINI: *DZone Refcardz 204: Apache Spark*. DZone, 1 Auflage, 2014.
- [Mit15] MITCHELL, TOM M.: *Machine Learning*. McGraw Hill, Draft of January 31, 2015 Auflage, 2015.
- [ML13] MARCUS LEICH, JOCHEN ADAMEK, ET AL.: *Applying Stratosphere for Big Data Analytics*. TU Berlin, HPI Potsdam, Humboldt-Universität zu Berlin, 2013.
- [MZ12] MATEI ZAHARIA, MOSHARAF CHOWDHURY, ET AL.: *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*. University of California, Berkeley, 2012.
- [Ng02] NG, RAYMOND T.: *CLARANS: a method for clustering objects for spatial data mining*. Nummer DOI: 10.1109/TKDE.2002.1033770. Dept. of Comput. Sci., British Columbia Univ., Vancouver, BC; IEEE Transactions on Knowledge and Data Engineering (Impact Factor: 1.82), 10 2002.
- [NG12] NORBERT GRONAU, CORINNA FOHRHOL: *Wettbewerbsfaktor Analytics – Reifegrade ermitteln, Wirtschaftlichkeitspotentiale entdecken*. Addison-Wesley, 2012.
- [NRP05] NIKHIL R. PAL, KUHU PAL, ET. AL: *A Possibilistic Fuzzy c-Means Clustering Algorithm*, Band 13. IEEE TRANSACTIONS ON FUZZY SYSTEMS, 2005.
- [RK94] R. KRAMER, R. GUPTA, M.L. SOFFA: *The combining DAG: a technique for parallel data flow analysis*. Nummer 10.1109/71.298205. Dept. of Comput. Sci., Pittsburgh Univ., PA, USA, IEEE, 2002 Auflage, 08 1994.
- [Rus11] RUSSOM, PHILIP: *TDWI Best Practices Report: Big Data Analytics*. TDWI Research, 2011.

- [Sat12] SATHI, DR. ARVIND: *Big Data Analytics: Disruptive Technologies for Changing the Game*. MC Press, First Edition Auflage, 2012.
- [SS14] STEPHEN SOLTESZ, ANDY BAVIER, ET AL.: *Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors*. University of Toronto, Department of Computer Science, 2014.
- [SW97] S.M. WEISS, N. INDURKHYA: *Predictive data mining: A practical guide*. Morgan-Kaufman, 1997.
- [TH09] T. HASTIE, R. TIBSHIRANI, ET AL.: *Data Mining, Inference, and Prediction*. Springer Publishing Company, 2009.
- [TH14] TREVOR HASTIE, RAHUL MAZUMDER, ET AL.: *Matrix Completion and Low-Rank SVD via Fast Alternating Least Squares*. Statistics Department and ICME Stanford University, 2014.
- [TR10] THOMAS RAUBER, GUDULA RÜNGER: *Parallel Programming: For Multicore and Cluster Systems*. Springer, 2010.
- [UF96] U.M. FAYYAD, G. PIATETSKY-SHAPIRO, ET AL.: *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.
- [Whi13] WHITE, TOM: *Hadoop: The Definitive Guide*, Band 2nd Edition. O'Reilly, 2013.
- [WKH12] WOLFGANG KARL HÄRDLE, LÉOPOLD SIMAR: *Applied Multivariate Statistical Analysis*. Springer, 3 Auflage, 2012.
- [YZ09] YUNHONG ZHOU, DENNIS WILKINSON, ET AL.: *Large-scale Parallel Collaborative Filtering for the Netflix Prize*. HP Labs, Palo Alto, 2009.

Internetquellen

- [Abo15] ABOUT.COM: *Linux / Unix Command: top* URL: http://linux.about.com/od/commands/l/blcmdl1_top.htm, 11 2011. Aufgerufen am 02.02.2015.
- [AL15] ANNA LEE, GAVIN M. JOYNT, ET AL.: *Making Sense of Decision Analysis Using a Decision Tree*, US National Library of Medicine, National Institutes of Health URL: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2669856/>, 05 2009. Aufgerufen am 01.02.2015.
- [AP14] ANDREW PAVLO, ERIK PAULSON, ET AL.: *A Comparison of Approaches to Large-Scale Data Analysis*. Brown University, Providence URL: <http://database.cs.brown.edu/projects/mapreduce-vs-dbms/>, 08 2009. Aufgerufen am 16.12.2014.
- [Apa14] APACHE: *Cluster Mode Overview* URL: <http://spark.apache.org/docs/latest/cluster-overview.html>, 10 2014. Aufgerufen am 21.12.2014.
- [BG15] BALTES-GÖTZ, B.: *Logistische Regressionsanalyse mit SPSS*, Universität Trier, ZIMK URL: <http://www.uni-trier.de/fileadmin/urt/doku/logist/logist.pdf>, 06 2012. Aufgerufen am 27.01.2015.
- [BIT14] BITKOM: *Big-Data-Technologien – Wissen für Entscheider* URL: http://www.bitkom.org/files/documents/BITKOM_Leitfaden_Big-Data-Technologien-Wissen_fuer_Entscheider_Febr_2014.pdf, 2 2014. Aufgerufen am 04.10.2014.
- [Cli15] CLICK, CLIFF: *Sparkling Water! Sparkling Water. H2O and Scala and Spark* URL: <http://0xdata.com/blog/2014/09/Sparkling-Water/>. 0xData H2O, 09 2014. {Aufgerufen am 24.02.2015.}
- [Dat15] DATABRICKS: *Movie Recommendation with MLlib*, Databricks URL: <https://databricks-training.s3.amazonaws.com/movie-recommendation-with-mllib.html>, 11 2014. Aufgerufen am 16.01.2015.
- [DGC14] DR. GOUTAM CHAKRABORTY, MURALI KRISHNA PAGOLU: *Analysis of Unstructured Data: Applications of Text Analytics and Sentiment Mining* URL: <http://support.sas.com/resources/papers/proceedings14/1288-2014.pdf>, 08 2014. Aufgerufen am 08.10.2014.

- [DJF15] DR. J. FÜRNKRANZ, DR. G. GRIESER: *Maschinelles Lernen: Symbolische Ansätze* URL: <http://www.ke.tu-darmstadt.de/lehre/archiv/ws0607/mlm/>, 2006. Aufgerufen am 22.01.2015.
- [DML15] DAVID M. LANE, DAVID SCOTT, ET AL.: *Introduction to Statistics, Rice University, University of Houston* URL: <http://onlinestatbook.com/2/index.html>, 11 2013. Aufgerufen am 06.01.2015.
- [Doc15] DOCKER: *Docker Documentation* URL: <https://docs.docker.com>, 12 2014. Aufgerufen am 25.01.2015.
- [Dum14] DUMBHILL, EDD: *What is big data?* URL: <http://radar.oreilly.com/2012/01/what-is-big-data.htm>, 11 2011. Aufgerufen am 04.10.2014.
- [EMC15] EMC, PIVOTAL: *Introducing Apache Flink - A new approach to distributed data processing* URL: <http://www.meetup.com/HandsOnProgrammingEvents/events/210504392/>, 11 2014. Aufgerufen am 07.02.2015.
- [Enz15] ENZMANN, DR. DIRK: *Lineare Regression, Universität Hamburg, Juristische Fakultät* URL: <http://www2.jura.uni-hamburg.de/instkrim/kriminologie/Mitarbeiter/Enzmann/Lehre/StatIKrim/Regression.pdf>, 11 2014. Aufgerufen am 27.01.2015.
- [Eso15] ESOTERICSOFTWARE: *Documentation for Kryo Serializer* URL: <https://github.com/EsotericSoftware/kryo>, 01 2015. Aufgerufen am 12.02.2015.
- [Ewe15] EWEN, STEPHAN: *Apache Flink - Overview* URL: <http://de.slideshare.net/stephanewen1/apache-flink-overview>, 09 2014. Aufgerufen am 05.02.2015.
- [Flf15] FLINK, APACHE: *Apache Flink Programming Guide* URL: http://flink.apache.org/docs/0.8/programming_guide.html#introduction, 12 2014. Aufgerufen am 07.02.2015.
- [Gha15] GHAHRAMANI, ZOUBIN: *Unsupervised Learning* URL: <http://mlg.eng.cam.ac.uk/zoubin/papers/ul.pdf>, 09 2004. Aufgerufen am 18.01.2015.
- [Glu14] GLUSTER: *Glusterfs and Tachyon* URL: <http://blog.gluster.org/2014/08/glusterfs-and-tachyon/>, 08 2014. Aufgerufen am 16.12.2014.
- [Gre15] GREGG, BRENDAN: *Top 10 DTrace Scripts for Mac OS X* URL: <http://dtrace.org/blogs/>, 10 2012. Aufgerufen am 02.02.2015.
- [Had15] HADOOP, APACHE: *HDFS Architecture* URL: <http://hadoop.apache.org/docs/r1.2.1/images/hdfsarchitecture.gif>, 09 2012. Aufgerufen am 04.01.2015.

- [Has15] HASHIMOTO, MITCHELL: *Feature Preview: Docker-Based Development Environments URL:* <http://www.vagrantup.com/blog/feature-preview-vagrant-1-6-docker-dev-environments.html>, 04 2014. Aufgerufen am 08.02.2015.
- [Hon15] HONG, JOHAN: *Spark, Spark Streaming and Tachyon URL:* <http://www.slideshare.net/Johanhong1/spark-spark-streaming-tachyon-40548210>, 10 2014. {Aufgerufen am 14.02.2015.
- [Hor14] HORTONWORKS: *Hadoop Distributed File System URL:* <http://hortonworks.com/hadoop/hdfs/>, 04 2014. Aufgerufen am 26.11.2014.
- [Lam15] LAMPLUGH, SIMON: *Continues Deployment of Dockerized Microservices URL:* <http://capgemini.github.io/devops/dockerizing-microservices/>. Docker, Microservices, Continuous Delivery, DevOps, Automation, 12 2014. Aufgerufen am 07.01.2015.
- [LB15] LEO BREIMAN, ADELE CUTLER: *Random Forests, University of Berkeley California URL:* https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm, 01 2006. Aufgerufen am 15.01.2015.
- [Lou14] LOUKIDES, MIKE: *What is data science? URL:* <http://radar.oreilly.com/2010/06/what-is-data-science.html>, 10 2010. Aufgerufen am 16.10.2014.
- [Mar15] MARKOWETZ, FLORIAN: *Klassifikation mit Support Vector Machines, Max-Planck-Institut für Molekulare Genetik, Berlin Center for Genome Based Bioinformatics URL:* http://lectures.molgen.mpg.de/statistik03/docs/Kapitel_16.pdf, 01 2003. Aufgerufen am 31.01.2015.
- [Mav15] MAVEN, APACHE: *Apache Maven Project URL:* <http://maven.apache.org>, 12 2014. Aufgerufen am 12.02.2015.
- [McN14] McNUTTY, EILEEN: *Understanding Big Data: The Seven V's, Dataconomy URL:* <http://dataconomy.com/seven-vs-big-data/>, 05 2014. Aufgerufen am 23.10.2014.
- [Met14] METZ, CADE: *Spark: Open Source Superstar Rewrites Future of Big Data URL:* <http://www.wired.com/2013/06/yahoo-amazon-amplab-spark/all/>. Wired.com, 06 2013. Aufgerufen am 28.10.2014.
- [Mir15] MIRKES, E. M.: *University of Leicester: K-means and K-medoids URL:* http://www.math.le.ac.uk/people/ag153/homepage/KmeansKmedoids/Kmeans_Kmedoids.html. University of Leicester, 01 2011. Aufgerufen am 15.01.2015.
- [Ren15] RENKENS, VINCENT: *Taalverwerving door robots via demonstratie: Gerangschikte gegevens en relevantiedetectie URL:* <http://www.scriptiebank.be/scriptie/>

- taalverwerving-door-robots-demonstratie-gerangschikte-gegevens-en-relevantiedetectie.*
KU Leuven, 01 2014. Aufgerufen am 06.01.2015.
- [sbt15] SBT: *sbt - The Interactive Build Tool* URL: <http://www.scala-sbt.org>, 12 2014.
Aufgerufen am 12.02.2015.
- [Spa14] SPARK, APACHE: *Spark Programming Guide*. Apache Spark URL: <http://spark.apache.org/docs/1.2.0/programming-guide.html>, 01 2015. Aufgerufen am 02.11.2014.
- [Spa15] SPARK, APACHE: *Tuning Spark* URL: <http://spark.apache.org/docs/1.2.0/tuning.html>, 12 2014. Aufgerufen am 10.02.2015.
- [ST15] SCHMIDT-THIEME, LARS: *Entscheidungsbaumverfahren*, Universität Karlsruhe URL: <http://marketing.wiwi.uni-karlsruhe.de/institut/vivor/kaiman/kaiman/index.xml.html>, 11 2003. Aufgerufen am 24.01.2015.
- [Sta15] STATISTA: *Prognose zum Volumen der jährlich generierten digitalen Datenmenge weltweit in den Jahren 2005 bis 2020 (in Exabyte)* URL: <http://de.statista.com/statistik/daten/studie/267974/umfrage/prognose-zum-weltweit-generierten-datenvolumen/>, 01 2015. Aufgerufen am 14.02.2015.
- [Sto15] STORAGE, LASCON: URL: <http://www.lascon.co.uk/Oracle-RAC.php>, 11 2014.
Aufgerufen am 07.02.2015.
- [Tee14] TEE, JASON: *Handling the four 'V's of big data: volume, velocity, variety, and veracity, The ServerSide* URL: <http://www.theserverside.com/feature/Handling-the-four-Vs-of-big-data-volume-velocity-variety-and-veracity>, 08 2013.
Aufgerufen am 06.10.2014.
- [Tur14] TURNER, JAMES: *Hadoop: What it is, how it works and what can it do* URL: <http://radar.oreilly.com/2011/01/what-is-hadoop.html>, 3 2010. Aufgerufen am 26.10.2014.
- [VN15] VAUGHAN-NICHOLS, STEVEN J.: *Docker libcontainer unifies Linux container powers* URL: <http://www.zdnet.com/article/docker-libcontainer-unifies-linux-container-powers/>, 06 2014. Aufgerufen am 26.01.2015.
- [Vya15] VYAS, JAY: *Microservice principles and Immutability – demonstrated with Apache Spark and Cassandra* URL: <http://developerblog.redhat.com/2015/01/20/microservice-principles-and-immutability-demonstrated-with-apache-spark-and-cassandra/>, 01 2015. Aufgerufen am 04.02.2015.
- [Wik14] WIKIBENCH: *Wikipedia access traces* URL: [\protect\T1\textbracelefthttp://www.wikibench.eu/](http://protect/T1/textbracelefthttp://www.wikibench.eu/), 01 2014. Aufgerufen am 28.10.2014.

- [Xia15] XIA, C.: *Work Structure of MapReduce URL*: http://xiaochongzhang.me/blog/wp-content/uploads/2013/05/MapReduce_Work_Structure.png, 05 2013. Aufgerufen am 04.02.2015.
- [Xin14] XIN, GONZALES ET AL: *A Resilient Distributed Graph System on Spark URL*: {<https://amplab.cs.berkeley.edu/publication/graphx-grades.pdf>. Amplab UC Berkeley, 08 2013. Aufgerufen am 04.10.2014.
- [Zah12] ZAHARIA, MATEI: *Advanced Spark Features URL*:<http://ampcamp.berkeley.edu/wp-content/uploads/2012/06/matei-zaharia-amp-camp-2012-advanced-spark.pdf>, 06 2012.

Abbildungsverzeichnis

1.1	Darstellung der vier Säulen von Big Data: The Four V's of Big Data	2
2.1	Der Machine-Learning-Prozess: Phase 1 - Lernphase	10
2.2	Der Machine-Learning-Prozess: Phase 2 - Prediction-Phase (Vorhersage)	10
2.3	Exemplarische Darstellung eines Bilderkennungsprozesses mittels Deep Learning. Entnommen aus [Ben09]	12
2.4	Cluster-Verfahren mit verschiedenen Iterationsschritten am Beispiel k-Means. Entnommen aus [Ren15]	13
2.5	Ein konkretes Word-Count-Beispiel für MapReduce	24
3.1	Übersicht des BDAS mit den vom AMPLab empfohlenen Bibliotheken.	29
3.2	Der Aufbau von HDFS mit Namenodes und Datanodes [Had15].	30
3.3	Der Datamanagement-Layer im BDAS mit Tachyon und diversen Filesystemen [Glu14].	31
3.4	Die Bestandteile der MLbase	33
4.1	Flink Streaming Dataflow mit Feedback führt bei Iterationen automatische Optimierungen durch.	37
5.1	Aufbau eines Standardcluster im Rechenzentrumsbetrieb mit Application-Server und Oracle Datenbanken [Sto15].	40
5.2	Clusteraufbau mit Spark mit einem Master und vier Worker-Nodes.	41
5.3	Clusteraufbau mit Spark [Apa14]	42
5.4	Transformation von RDDs in Spark.	44

5.5 Schematische Darstellung der Erzeugung und Verarbeitung von RDDs aus persistierten Daten	45
6.1 Der BDAS. Abbildung aus „Big Data Analytics Beyond Hadoop“, S 15 [Agn14]	52
6.2 Build, Installation und Start des Mesos Masternode	56
6.3 Start und Registrierung eines Mesos Slaves	56
6.4 Start des Apache Flink Masternode mit Anzeige des Logs	58
6.5 Das Apache Flink Dashboard auf der lokalen Installation	58
8.1 Ausführung des Linux-Befehls top auf dem Master-Node des Beuth-Clusters.	75
8.2 Messung der Laufzeit und CPU-Auslastung des CharCounters mit 4 Threads .	78

Tabellenverzeichnis

4.1 Übersicht der einiger wichtiger Transformationen von Apache Flink	36
5.1 Übersicht der einiger wichtiger Transformationen auf RDDs	44
5.2 Übersicht der einiger wichtiger Actions auf RDDs	46
5.3 Übersicht der Storage Levels von Spark	47
8.1 Übersicht der lokal verwendeten Hardware	76
8.2 Übersicht über das Cluster der Beuth Hochschule Berlin	76

Listings

5.1	Verwendung der Broadcast-Variablen in Spark	46
6.1	Setup Spark mit eigener JVM	55
6.2	Vagrant Provisionierungs-Skript für Spark	55
6.3	SBT Script für Anwendungen mit Spark und MLLib-Dependencies	57
7.1	SimpleTestApp.scala - zählt Buchstabenvorkommen in Textdateien.	60
7.2	Beispieleinträge der Wikipedia Access Logs.	61
7.3	Konfigurationseinstellungen der lokalen Spark-Installation	61
7.4	sbt-Buildfile zum Bauen des Spark-WordCount-Prototyp	61
7.5	Spark-WordCount-Prototyp mit Implementierung mittels MapReduce-Algorithmus	61
7.6	ShellScript zum Start des Spark-WordCount-Prototyp	63
7.7	Konfigurationseinstellungen der lokalen Flink-Installation	65
7.8	WordCountPrototyp für Apache Flink als MapReduce-Implementierung	65
7.9	Maven POM zum bauen von Flink-Applikationen als JAR und Fat-JAR inklusive aller Dependencies	67
7.10	Shellscrip zum Start des Flink WordCountPrototyp	70

Anhang A

Zusätze

A.1 Übersicht der RDD Transformationen

Die folgende Tabelle zeigt alle Transformationen, die auf Spark RDDs möglich sind (Entnommen aus [Spa14] - Stand Spark 1.2.0):

Transformation	Zweck
map(func)	Return a new distributed dataset formed by passing each element of the source through a function func.
filter(func)	Return a new dataset formed by selecting those elements of the source on which func returns true.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
mapPartitions(func)	Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type <code>Iterator<T> => Iterator<U></code> when running on an RDD of type T.
mapPartitionsWithIndex(func)	Similar to mapPartitions, but also provides func with an integer value representing the index of the partition, so func must be of type <code>(Int, Iterator<T>) => Iterator<U></code> when running on an RDD of type T.
sample(withReplacement, fraction, seed)	Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed.
union(otherDataset)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(otherDataset)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([numTasks]))	Return a new dataset that contains the distinct elements of the source dataset.

Transformation	Zweck
groupByKey([numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or combineByKey will yield much better performance. Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks.
reduceByKey(function, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
sortByKey([ascending], [numTasks])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.

Transformation	Zweck
join(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.
cogroup(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Iterable<V>, Iterable<W>) tuples. This operation is also called groupWith.
cartesian(otherDataset)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(command, [envVars])	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(numPartitions)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(numPartitions)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithin Partitions(partitioner)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Die folgende Tabelle zeigt alle Actions, die auf Spark RDDs möglich sind (Entnommen aus [Spa14] - Stand Spark 1.2.0):

Actions	Zweck
reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(n)	Return an array with the first n elements of the dataset. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements.
takeSample(withReplacement, num, [seed])	Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(n, [ordering])	Return the first n elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(path)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call <code>toString</code> on each element to convert it to a line of text in the file.

Actions	Zweck
saveAsSequenceFile(path)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that either implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(path)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(func)	Run a function func on each element of the dataset. This is usually done for side effects such as updating an accumulator variable (see below) or interacting with external storage systems.