Hochschule Offenburg
University of Applied Sciences

Apache Spark: Untersuchung der Möglichkeiten zur verteilten Datenverarbeitung und Analyse von Streaming Data

- MASTERARBEIT -

für die Prüfung zum Master of Science

Studiengang Informatik (INFM)
an der Fakultät Elektrotechnik und Informationstechnik der
Hochschule für angewandte Wissenschaften Offenburg

von

DANIEL MÜLLER

23. September 2015

Bearbeitungszeitraum	6 Monate
Matrikelnummer, Kurs	178152, INFM3
1. Gutachter	Prof. Dr. Stephan Trahasch
	stephan.trahasch@hs-offenburg.de
2. Gutachter	Sascha Niro, M.Sc.
	sascha.niro@hs-offenburg.de

I Vorwort

Im Laufe meines Masterstudiums hatte ich die Gelegenheit viel Neues zu lernen und in verschiedene Sachbereiche der Informatik tiefer einzutauchen. Die Vorlesung Advanced Business Intelligence hat mich dabei besonders interessiert. Es wurden Möglichkeiten zur Verarbeitung und Analyse von Datenbeständen aufgezeigt, um darin beispielsweise für ein Unternehmen relevante Informationen finden zu können. Auch wurden Grundlagen des maschinellen Lernens vorgestellt. Diese Themen, welche sich allesamt in den Bereich der Big Data Analytics eingliedern lassen, interessierten mich bereits in der Vorlesung so sehr, dass ich diesen Sachbereich auch in meiner Masterthesis anstreben wollte. Entsprechend groß war die Freude über den Themenvorschlag, den mir Professor Trahasch zum Thema Datenverarbeitung mit Apache Spark vorgestellt hatte. Die Entscheidung für diese Arbeit war schnell getroffen.

Entsprechend erfreut darüber war ich, ein solch spannendes Thema in meiner Masterarbeit behandeln zu können. An dieser Stelle möchte ich mich bei Herrn Prof. Dr. Stephan Trahasch und Sascha Niro für den Vorschlag der Thesis, sowie die gute und hilfreiche Betreuung meiner Arbeit bedanken! Danke euch beiden für die schnellen Antworten bei Fragen und die informativen Treffen über die letzten sechs Monate! Danke auch hier nochmals an Prof. Trahasch für die Bereitstellung des Amazon Webservices Kontingents, welches mir das Arbeiten mit EC2-Instanzen ermöglichte.

II Eidesstattliche Erklärung

Hiermit versichere ich eidesstattlich, dass die vorliegende Master-Thesis von mir selbstständig und ohne unerlaubte fremde Hilfe angefertigt worden ist, insbesondere, dass ich alle Stellen, die wörtlich oder annähernd wörtlich oder dem Gedanken nach aus Veröffentlichungen, unveröffentlichten Unterlagen und Gesprächen entnommen worden sind, als solche an den entsprechenden Stellen innerhalb der Arbeit durch Zitate kenntlich gemacht habe, wobei in den Zitaten jeweils der Umfang der entnommenen Originalzitate kenntlich gemacht wurde. Ich bin mir bewusst, dass eine falsche Versicherung rechtliche Folgen haben wird.

Diese Master-Thesis ist urheberrechtlich geschützt, unbeschadet dessen wird folgenden Rechtsübertragungen zugestimmt:

- Der Übertragung des Rechts zur Vervielfältigung der Master-Thesis für Lehrzwecke an der Hochschule Offenburg (§ 16 UrhG),
- Der Übertragung des Vortrags-, Aufführungs- und Vorführungsrechts für Lehrzwecke durch Professoren der Hochschule Offenburg (§ 19 UrhG),
- Der Übertragung des Rechts auf Wiedergabe durch Bild- und Tonträger an die Hochschule Offenburg (§ 21 UrhG).

29.09.2015

(Datum, Unterschrift)

III Zusammenfassung

Über die letzten Jahre entstanden unterschiedlichste Gerätschaften, besonders im mobilen Bereich und der Industrie 4.0, die große Datenmengen generieren. Diese müssen in entsprechenden Netzwerken entgegengenommen, verarbeitet und ggf. analysiert werden um einen Mehrwert zu erzielen. Ein Vertreter für die Umsetzung von Echtzeit-Datenverarbeitung ist Apache Spark, ein Open Source Framework, welches für die Analyse von Informationsströmen und Datenbeständen eingesetzt werden kann. Im Rahmen dieser Masterarbeit wird die Apache Spark Plattform von Grund auf erläutert und auf ihre Einsatzfähigkeit im Bereich der verteilten Datenverarbeitung untersucht. Durch die theoretische Einleitung in die Themen Big Data, Streaming Data, Data Mining und Real-Time Analytics wird ein grundlegendes Verständnis für die Aufgaben und Herausforderungen dieses Sachgebiets vermittelt. Es wird die Entwicklung von der Batch- zur Streamingverarbeitung vorgestellt und die Anforderungen, sowie Voraussetzungen für die Umsetzung von Echtzeitsystemen aufgezeigt.

Nachdem diese Grundlagen vermittelt wurden, folgt eine Vorstellung des Projektumfangs der Apache Software Foundation, in welchen sich auch das Spark Projekt einordnen lässt. Die Arbeit erläutert die Grundkonzepte von Apache Spark, wie die Entwicklung, Architektur und der Clusterbetrieb der Plattform. Dabei stützen sich die Untersuchungen auf praktische Beispiele, um die Arbeitsweise von Apache Spark näher aufzuzeigen. Die vorgestellten Themen fallen in die Bereiche der parallelen Datenverarbeitung mit Spark und beschäftigen sich mit den Voraussetzungen für das Erstellen von Anwendungen, die den verteilten Aufbau und die horizontale Skalierbarkeit von Spark ausnutzen. Spark bringt über eigene Bibliotheken auch Funktionalitäten für die Datenverarbeitung in speziellen Aufgabengebieten mit sich. In dieser Arbeit werden ebenfalls die beiden Bibliotheken MLlib, welche im Bereich des maschinellen Lernens Einsatz findet, und Spark Streaming, die Bibliothek für Verarbeitung von Datenflüssen, vorgestellt und deren Funktionsumfang untersucht. Das Kernthema dieser Arbeit bildet die Modellierung von Lösungsmöglichkeiten zur Analyse von Streaming Data. Es wird hierdurch die Funktionsweise von Spark und dessen Streaming Bibliothek anhand von kompletten Applikationen zur Ausreißererkennung in Datenströmen im Detail aufgezeigt.

Die Arbeit zeigt auf, dass Spark durchaus für den Einsatz zur verteilten Datenverarbeitung geeignet ist. Auch der Umgang mit Streaming Data wird durch den Bau der Prototypen nachgewiesen. In dem abschließenden Fazit werden die Erkenntnisse der Arbeit zusammengefasst und die Einsetzbarkeit von Spark diskutiert.

IV Abstract

Over the past years different devices were developed which generate huge data amounts, mostly in the mobile device domain or the Industry 4.0. These amounts have to be received, processed and, if necessary, analyzed to create an additional benefit. One solution for developing such real-time data processing applications is Apache Spark, an open-source framework which is applied to analyze streams of information and data stocks. The task of this Master‘s thesis is to give an introduction into the Apache Spark platform and an examination of its usability for distributed data processing. A fundamental understanding for the tasks and challenges of this specialist field is given by an introduction in the subjects Big Data, Streaming Data, Data Mining and Real-Time Analytics. The development from batch to streaming processing and the requirements for the implementation of real-time systems, are also shown.

After these basics are communicated, a presentation of the project scope of the Apache Software Foundation follows. The thesis comments the evolution, architecture and cluster operating of the platform. The investigations lean on practical examples to show the functioning of Apache Spark. The introduced subjects come within the limits of the parallel data processing with Spark and keep busy with the preconditions of the creation of such applications which use Spark‘s distributed organization and its horizontal scalability. Spark also comes with libraries of functionalities for special working areas. In this thesis two libraries are introduced and examined by examples: MLlib, a library for machine learning, and Spark Streaming, for processing streaming data. The main topic of this thesis constitutes the modeling of implementations for streaming data analysis applications. Within this chapter the functionality of Spark and its library are shown by detail via complete applications for outlier detection in data streams.

This thesis shows the utilizability of Spark for distributed data processing. The handling of Streaming Data is also verified by the implementation of the prototypes. A closing conclusion tells the experiences during the thesis and discusses the applicability of Spark.

V Inhaltsverzeichnis

I Vorwort II

II Eidesstattliche Erklärung III

III Zusammenfassung IV

IV Abstract V

V Inhaltsverzeichnis VI

VI Codeblockverzeichnis X

VII Abbildungsverzeichnis XII

VIII Tabellenverzeichnis XIV

IX Diagrammverzeichnis XV

X Formelverzeichnis XVI

1 Einleitung 1

1.1 Problemstellung und Motivation 1

1.2 Ziele der Arbeit 3

1.3 Aufbau der Arbeit 3

2 Grundlagen 4

2.1 Data Mining 4

2.2 Big Data 7

2.3 Streaming Data 8

2.3.1 Quellen von Streaming Data 9

2.3.2 Was Streaming Data anders macht 10

2.3.3 Infrastruktur und Algorithmen 11

2.3.4 Batchverarbeitung mit MapReduce 11

2.3.5 Lambda-Architektur 15

2.4 Real-Time Frameworks 17

2.4.1 Historische Entwicklung 17

2.4.2 Anforderungen und Voraussetzungen an Echtzeit-Systeme 17

2.4.3 Echtzeit-Bedingungen 18

2.4.4 Komponenten eines Echtzeitsystems 19

2.4.5 Eigenschaften eines Echtzeitsystems 22

3 Tools und Frameworks im Streaming Big Data Umfeld 23

3.1 Kernkomponenten 25

3.2 Systemmanagement 25

3.3 Execution Engines zur Datenverarbeitung 26

3.4 Abstraktionen der Arbeitsschicht 26

3.5 Arbeitsframeworks mit eigenen Execution Engines 26
3.6 Datenverarbeitungs-Frameworks mit eigenem Management 27
3.7 BigTable-Lösungen 27
3.8 Weitere Komponenten 27
3.8.1 Datentransport 28
3.8.2 Automatisierung 28
3.8.3 Sicherheit 28
3.8.4 Vereinfachung der Administration 29
4 Grundkonzepte von Apache Spark 30
4.1 Historische Entwicklung 31
4.2 Architektur von Spark 31
4.2.1 Spark Komponenten 32
4.2.2 Verwenden der Komponenten durch Einbindung von Bibliotheken 33
4.3 Einteilung in Cluster 33
5 Datenverarbeitung mit Spark 35
5.1 RDDs - Resilient Distributed Datasets 35
5.2 Datenquellen und Dateiformate 38
5.3 Schlüssel-Wert Paare 40
5.4 Datenpartitionierung 41
5.5 Globale Datenstrukturen 44
6 Erstellung von Spark Applikationen 46
6.1 Zugriff über SparkContext 46
6.2 Applikationen bereitstellen 47
6.3 Ablauf beim Ausführen einer Applikation über spark-submit 48
6.4 Begrifflichkeiten bei der Ausführung von Applikationen 48
6.5 Weboberfläche 49
6.6 Einstellungen 50
6.7 Ressourcenverteilung 50
7 Datenspeicherung und Benchmarks mit Spark 52
7.1 Spark & HDFS 52
7.2 Apache Cassandra 53
7.2.1 Hohe Verfügbarkeit und Datenreplikation 54
7.2.2 Datenverteilung, DHTs und vnodes 55
7.2.3 Wohin Daten intern geschrieben werden 56
7.2.4 Datengröße 58
7.2.5 Ressourcen-Abstimmung 59

7.3 Spark & Cassandra 59
7.4 Benchmarks 61
7.4.1 Cassandra....61
7.4.2 HDFS 65
8 Streaming Data und maschinelles Lernen mit Spark 68
8.1 Spark Streaming 68
8.1.1 Architektur und Micro-Batching bei Spark Streaming 69
8.1.2 Zustandslose und zustandsbehaftete Transformationen 70
8.1.3 Ausgabe-Operationen 71
8.1.4 StreamingContext & SocketStream 71
8.1.5 Checkpointing und Fehlertoleranz 72
8.1.6 Optimierungen 74
8.2 Maschinelles Lernen mit MLlib 75
8.2.1 Datenaufbereitung....76
8.2.2 Statistiken 77
8.2.3 Funktionsumfang von MLlib 78
8.3 Datenvisualisierung mit Apache Zeppelin....79
8.3.1 Allgemeines zu Spark Zeppelin 80
8.3.2 Installation von Apache Zeppelin....80
8.3.3 Weboberfläche und Verwendung von Zeppelin 80
9 Streaming Data Analyse am Beispiel von Energiedaten....82
9.1 Business Understanding 82
9.1.1 Über die DEBS Challenge....82
9.1.2 Aufgabenstellung des Projekts....83
9.2 Data Understanding: Aufbau der Daten....83
9.3 Data Preparation: Bereinigen, Senden und Empfangen 84
9.3.1 Senderseite: Java Server-Thread....85
9.3.2 Empfängerseite: Apache Spark....86
9.4 Modeling: Umsetzung der Architektur....90
9.4.1 Arithmetisches Mittel 92
9.4.2 Median 96
9.4.3 Arithmetisches Mittel der Mediane 100
9.5 Evaluation: Überprüfen der Modelle 104
9.5.1 Bewertung über Benchmarks 104
9.5.2 Diskussion der einzelnen Lösungen....110
9.6 Deployment: Festlegung eines Modells....112

10 Fazit ... 114
10.1 Aufsetzen von Clustern ... 114
10.2 Abstraktionsmodell & Erstellen von Applikationen ... 114
10.3 Ausführung von Applikationen & Debugging ... 115
10.4 Bugs & Fehlermeldungen ... 115
10.5 Unterstützte Datentypen ... 115
10.6 Optimierungsmöglichkeiten ... 116
10.7 Einsatzfähigkeit von Spark für die Streaming Data Analyse ... 116
10.8 Produktiver Einsatz von Spark ... 117
10.9 Persönliches Resümee & Ausblick ... 118
XI Literaturverzeichnis ... 119
XII Anhang ... 122
A Apache Spark ... 122
A.1 Datenquellen & Dateiformate ... 122
A.2 Schlüssel-Wert Paare ... 123
A.3 Partitionierung ... 124
A.4 Installation von Spark ... 125
A.5 Applikationen bereitstellen ... 127
A.6 Spark & HDFS ... 128
A.7 Spark & Cassandra ... 131
A.8 Cassandra-Benchmarks ... 134
A.9 HDFS-Benchmarks ... 137
A.10 Maschinelles Lernen mit MLlib ... 143
A.11 Apache Zeppelin ... 144
B Analyse von Streaming Data mit Spark ... 146
B.1 Data Understanding ... 146
B.2 Data Preparation ... 147
B.3 Modeling – Arithmetisches Mittel ... 150
B.4 Modeling – Median ... 155
B.5 Modeling – Arithmetisches Mittel der Mediane ... 157

VI Codeblockverzeichnis

Codeblock 1: Map-Methode ... 13
Codeblock 2: Reduce-Methode ... 13
Codeblock 3: Maven-Koordinaten für die Spark Core Bibliothek ... 33
Codeblock 4: Maven-Koordinaten für die SQL bzw. Streaming Bibliothek ... 33
Codeblock 5: Filterung und Rückgabe der Timestamps von Fehlern in HDFS-Logging [28] 37
Codeblock 6: Erstellung eines RDDs mit Schlüssel-Wert Paaren [27] ... 40
Codeblock 7: Wörterzählung über Pair-RDD und reduceByKey-Transformation [27] ... 40
Codeblock 8: Erstellung eines SparkContexts und Auslesen einer CSV-Datei in ein RDD .. 46
Codeblock 9: Kopieren einer JAR-Datei auf den Masterknoten eines Spark EC2-Clusters .. 47
Codeblock 10: Anmelden am Masterknoten über SSH ... 47
Codeblock 11: Starten einer Spark-Applikation über das spark-submit Skript ... 47
Codeblock 12: Aktivieren des Event-Loggings für eine Applikation ... 49
Codeblock 13: Anlegen eines Keyspace in Cassandra [29] ... 54
Codeblock 14: Beispielhafte Schreiboperationen in Cassandra [30] ... 57
Codeblock 15: Repräsentation der eingefügten Daten in der Memtable [30] ... 57
Codeblock 16: Repräsentation der eingefügten Daten im Commitlog [30] ... 57
Codeblock 17: Repräsentation der eingefügten Daten in der SSTable [30] ... 57
Codeblock 18: Maven-Koordinaten zur Verwendung des spark-cassandra-connectors ... 60
Codeblock 19: Speichern eines RDDs nach Cassandra ... 60
Codeblock 20: Lesen von Cassandra-Daten in ein Spark-RDD ... 60
Codeblock 21: Anlegen einer JavaStreamingContext Instanz ... 71
Codeblock 22: Empfänger für Nachrichten per TCP-Socket festlegen ... 72
Codeblock 23: Filtern von Daten eines DStreams und Ausgabe per print-Befehl ... 72
Codeblock 24: Aktivieren der Checkpointing-Funktion ... 72
Codeblock 25: Anlegen eines JavaStreamingContexts über Checkpoint-Verzeichnis ... 73
Codeblock 26: Verwendung des supervise Parameters ... 73
Codeblock 27: Aufbau eines Datensatzes bei der DEBS Grand Challenge 2014 [36] ... 83
Codeblock 28: Starten des Servers zum Auslesen und Senden der DEBS-Daten ... 86
Codeblock 29: Einrichten der Streaming-Empfänger ... 87

Codeblock 30: Starten der Spark-Applikation zur Ausreißer-Erkennung 87
Codeblock 31: Umwandeln des Streams in ein JavaPairDStream und Duplikatsprüfung 89
Codeblock 32: Filterung der Nullwerte im Verbrauch 89
Codeblock 33: Zeitstempel aus dem Schlüssel des JavaPairDStream entfernen 90
Codeblock 34: Aufbau der Daten nach Durchführen der Data Preparation 92
Codeblock 35: Auslesen und Deserialisieren von Daten im JSON-Format [27] 122
Codeblock 36: Speichern von Daten im JSON-Format [27] 122
Codeblock 37: Partitionieren und Speichern einer CSV-Datei mit HashPartitioner 124
Codeblock 38: Befehl zum Aufsetzen eines Spark-Clusters im EC2-Umfeld 126
Codeblock 39: EC2-Befehle zum Starten, Stoppen bzw. Terminieren eines Clusters 127
Codeblock 40: Cassandra-Keyspace einrichten 133
Codeblock 41: Anlegen einer Tabelle mit cqlsh 133
Codeblock 42: Einfügen von Daten in Cassandra-Tabellen mit cqlsh 133
Codeblock 43: Tabelleninhalt abfragen mit cqlsh 133
Codeblock 44: Löschen des Keyspaces und aller darin enthaltenen Tabellen 133
Codeblock 45: Löschen von Snapshots zur Speicher-Freigabe 134
Codeblock 46: Schreibtest mit Cassandra 135
Codeblock 47: Lesetest mit Cassandra 136
Codeblock 48: Schreibtest mit HDFS 138
Codeblock 49: Lesetest mit HDFS 141
Codeblock 50: Beispiel-Applikation zur Spam-Klassifizierung (angelehnt an [27]) 143
Codeblock 51: Daten mit Zeppelin einlesen und als SQL-Tabelle zur Verfügung stellen ... 145
Codeblock 52: Datenselektion über SQL-Interpreter 145
Codeblock 53: Spark-Applikation zur Untersuchung der DEBS CSV-Datei 147
Codeblock 54: Quellcode zum Sender der DEBS-Daten per TCP-Sockets 149
Codeblock 55: Applikation zur Ausreißer-Ermittlung via arithmetischem Mittel 154
Codeblock 56: Applikation zur Ausreißer-Ermittlung via Median-Bestimmung 156
Codeblock 57: Applikation zur Ausreißer-Ermittlung via arithm. Mittel der Mediane 160

VII Abbildungsverzeichnis

Abbildung 1: Die Aufgaben rund um den Data Mining Prozess 4
Abbildung 2:Umfrage zu den bekanntesten Data Mining Prozessmodellen 5
Abbildung 3: Prozessschaubild zu CRISP-DM [8] 6
Abbildung 4: MapReduce am Beispiel Wörterzählung [15] 13
Abbildung 5: Die dreischichtige Lambda-Architektur [18]... 16
Abbildung 6: Zusammenhang von Zeit und Wertigkeit bei Echtzeitbedingungen [19] 19
Abbildung 7: Tools im Apache Hadoop Umfeld [21].. 25
Abbildung 8: Zwischenschaltung von Knox zur Anfrageüberwachung [21] 29
Abbildung 9: Performance-Vergleich: Hadoop & Spark bei einer Regressionsaufgabe [26]. 30
Abbildung 10: Die Spark-Komponenten [27].. 32
Abbildung 11: Komponenten in einem Spark-Cluster [27].. 34
Abbildung 12: Datenfluss im Spark-Cluster bei der Ausführung eines Programms [28] 36
Abbildung 13: Graphenrepräsentation: Programm zur Filterung von HDFS-Logging [28] 37
Abbildung 14: Narrow (enge) und wide (weite) Abhängigkeiten zwischen RDDs [28] 37
Abbildung 15: Partitionierte CSV-Datei.. 43
Abbildung 16: Aufteilung der Daten nach dem Partitionieren... 44
Abbildung 17: Übersetzung DAG in physikalischen Ausführungsplan mit Stage-Pipelining . 49
Abbildung 18: Optimierung der Laufzeit durch angepasste Partitionierung.......................... 51
Abbildung 19: Aufbau eines Cassandra-Clusters [29]... 53
Abbildung 20: Cassandra-Ring ohne (links) und mit (rechts) vnodes [29]............................ 55
Abbildung 21: Wiederherstellung eines Knotens ohne (links) und mit (rechts) vnodes 56
Abbildung 22: Schreibvorgänge bei Cassandra [30] 57
Abbildung 23: Spark und Cassandra in einem gemeinsamen Cluster 63
Abbildung 24: Spark Streaming Architektur [27] 69
Abbildung 25: Erstellen von DStreams über Transformationen [27]..................................... 69
Abbildung 26: Ablauf einer zustandsbehafteten Transformation [27] 70
Abbildung 27: ReduceByWindow ohne und mit Verwendung der Gegenfunktion [27] 75
Abbildung 28: Typische Schritte beim maschinellen Lernen [27] 76
Abbildung 29: Mapping von Wörtern auf Vektoren 77

Abbildung 30: Notebook-Ansicht in Zeppelin 81
Abbildung 31: Grober Aufbau der Prototyp-Umgebung 85
Abbildung 32: Durchlaufen der CSV-Datei und senden der Zeilen als TCP-Nachricht.........86
Abbildung 33: Weitere Schritte zur Ausreißer-Findung per arithmetischem Mittel................94
Abbildung 34: Weitere Schritte zur Ausreißer-Findung per Medianbestimmung 97
Abbildung 35: Kombination: Median (pro Batch) und arithmetischem Mittel (pro Window).100
Abbildung 36: Weitere Schritte zur Ausreißer-Findung per Mittel der Mediane..................101
Abbildung 37: Benchmark: Mittelwert über arithm. Mittel [Batchintervall 2s] 105
Abbildung 38: Vergleich der Verarbeitungszeiten beim arithm. Mittel 106
Abbildung 39: Benchmark zum Median-Modell..107
Abbildung 40: Benchmark zum Median-Modell: 10 sec./2 sec. Batchintervall....................107
Abbildung 41: Benchmark arithm. Mittel der Mediane: 2 sec./30 sec. Batchintervall..........109
Abbildung 42: Finale Modellauswahl, Batchintervall 6 Sekunden 113
Abbildung 43: Aufbau eines gemeinsamen Clusters (Spark & Cassandra)........................131

VIII Tabellenverzeichnis

Tabelle 1: Beispiel-Verteilung von Daten beim RangePartitioner..41
Tabelle 2: Auswirkung des Flags spark.deploy.spreadOut auf die Arbeitsverteilung............51
Tabelle 3: Standardmäßige Speicherzuweisung bei Cassandra..59
Tabelle 4: Spezifikationen einer EC2-Instanz des Typs m3.2xlarge 61
Tabelle 5: Auflistung der in MLlib enthaltenen überwachten Lernverfahren..........................78
Tabelle 6: Auflistung der in MLlib enthaltenen unüberwachten Lernverfahren 79
Tabelle 7: Transformationen, anwendbar auf einem Pair-RDD [27]..................................123
Tabelle 8: Transformationen, anwendbar auf zwei Pair-RDDs [27]...................................123
Tabelle 9: Wichtige Ordner und Dateien der Spark-Installation 125
Tabelle 10: Weitere Parameter des launch-Skripts...127
Tabelle 11: Wichtige Paramter des spark-submit Skripts [27]...127
Tabelle 12: Mögliche Werte für den --master Parameter des spark-submit Skripts [27].....128
Tabelle 13: Ergebnisse des Schreibtests mit Cassandra..135
Tabelle 14: Ergebnisse des Lesetests mit Cassandra 137
Tabelle 15: Ergebnisse des Schreibtests mit HDFS 140
Tabelle 16: Ergebnisse des Lesetests mit HDFS..142

IX Diagrammverzeichnis

Diagramm 1: Grafische Darstellung der Datenverteilung beim RangePartitioner-Beispiel ...42
Diagramm 2: Vergleich des Speicherbedarfs (Rohdaten und Cassandra-Cluster)...............58
Diagramm 3: Verarbeitete Anfragen im Cassandra Schreibtest..62
Diagramm 4: Die Schreibgeschwindigkeit ist unabhängig von der Datengröße 62
Diagramm 5: Verarbeitete Leseanfragen im Spark-Cassandra Cluster................................64
Diagramm 6: Leseanfragen pro Core bei unterschiedlicher Clustergröße............................64
Diagramm 7: Zeiten, die zum Schreiben von Dateien nach HDFS benötigt wurden 66
Diagramm 8: Lesegeschwindigkeiten von HDFS in Spark-RDDs 67
Diagramm 9: Informationswert in Abhängigkeit der Aktualität...104
Diagramm 10: Zusammenhang zwischen Batch- und Window-Datengröße 108

X Formelverzeichnis

Formel 1: Berechnung der Mindest-Partitionsanzahl bei HDFS 42

Formel 2: Bestimmung eines Partitionierers für einen Datensatz 43

Formel 3: Berechnung der Inverse Document Frequency (IDF) 77

Formel 4: Berechnung der theoretischen Anzahl an DEBS-Daten 84

Formel 5: Berechnung des arithmetischen Mittels 92

Formel 6: Bestimmung des Medians 96

Formel 7: Berechnung des gewichteten arithmetischen Mittels 102

1 Einleitung

Dieses Kapitel verschafft einen Überblick über die Motivation und Ziele dieser Arbeit. Neben der historischen Entwicklung technologischer Standards im Bereich der Big-Data Verarbeitung werden die damit verbundenen neuen Probleme und Anforderungen vorgestellt. Auch eine Übersicht zum Aufbau und zur Strukturierung der Arbeit findet sich in diesem Kapitel.

1.1 Problemstellung und Motivation

Das Thema Datenspeicherung und –verarbeitung wurde über die letzten zehn Jahre bedeutender denn je. Das Aufkommen sozialer Medien wie das Facebook-Netzwerk Anfang 2004 [1] oder des Twitter-Dienstes gut ein Jahr später [2] brachte in kürzester Zeit einen gewaltigen Anstieg des Datenvolumens im Internet mit sich. Dieser Effekt verstärkte sich drastisch mit der Mobilisierung der Anwender durch Markteinführung der ersten Smartphones. Diese werden hauptsächlich im Bereich Social Media verwendet, um beispielsweise Status-Posts oder Nachrichten via Messaging abzuschicken und zu lesen. Deshalb fallen hier durchaus auch die stärksten Smartphone-Betriebssysteme iOS (Einführung mit dem iPhone, 2007) [3] und Android (2008) [4] ins Gewicht, da diese eine plattformweite Auslieferung von Software ermöglichen und dadurch Milliarden von Endgeräten mit gleichen Anwendungen ausgestattet werden können. Anfang dieses Jahrzehnts wurde auch der Begriff der Industrie 4.0 geprägt. Er steht für eine Vernetzung, Kommunikation und Koordination industrieller Gerätschaften sowie ganzer Firmenkomplexe. Auch hierdurch werden teils sehr große an Daten automatisch generiert, die verarbeitet werden müssen. Die Liste der datengenerierenden Gegenstände lässt sich aber noch weiter fortsetzen. So entstand über die letzten Jahre auch ein Trend, welcher eine Vernetzung aller möglichen Gegenstände, sei es der Smart-TV im Wohnzimmer, die Waschmaschine im Keller oder neuerdings sogar die Smart-Watch am Handgelenk mit sich bringt. Auch diese Gegenstände generieren etliche Daten, welche sich zu großen Mengen entwickeln. Das Zeitalter des „Internet of Things“ oder noch allgemeiner des „Internet of Everything“ ist eingeläutet. Fast jeder erdenkliche Gegenstand an uns und um uns herum könnte theoretisch in wenigen Jahren „smart“ agieren, in dem er entsprechend Daten über seine Umwelt generiert, empfängt und interpretiert.

Was nun aber macht die oben genannten Neuerungen überhaupt so interessant und nützlich? Hier kommt es wiederum auf den technischen Bereich an, in welchem die Daten entstehen und in welchem Umfang sie gesendet werden. Während der Benutzer eines sozialen Netzwerks seine Erlebnisse mit anderen teilen möchte, ist ein Unternehmen beispielsweise durch

das Austauschen von Sensordaten besonders daran interessiert, Kosten durch Optimierungen zu sparen, aber auch Produktionsprobleme zu vermeiden oder immerhin schnellstmöglich zu erkennen. Smarte Gegenstände wie die vernetzte Heizung zuhause oder die Sprachsteuerung im Auto sind dabei in Bereichen wie Komfort, Einfachheit, Sicherheit und Kosteneinsparungen einzugliedern.

Die Erhebung, Speicherung und das anschließende Abrufen von Daten bringt alleine keinen wirklichen Mehrwert, denn meist müssen die erstellten und verteilten Daten in Zusammenhang gebracht und interpretiert werden, um eine nützliche Wirkung zu erzielen. Es sind Analysemöglichkeiten vonnöten, die es beispielsweise sozialen Netzwerken mit Millionen von Nutzern oder großen Firmen erlauben, ihre Daten in Echtzeit zu untersuchen. Im Gegensatz zu einer Batchverarbeitung werden über Echtzeitsysteme dynamische Anfragen und eine umgehende Verarbeitung nach Empfang von Daten ermöglicht. Beispielsweise bekommt ein Amazon-Käufer Werbung zu Produkten angezeigt, nachdem er zuvor nach ähnlichen Gegenständen gesucht hat. Auch die sozialen Netzwerke finanzieren sich über gezielt eingesetzte Werbung, welche passgenau auf den User zugeschnitten ist. Auch eine sekundenschnelle Abänderung von Umgebungsbedingungen in komplexen Industriemaschinen wäre ohne Echtzeitanalyse von entsprechenden Sensordaten unvorstellbar. Bedenkt man hierbei, dass beispielsweise Twitter aktuell rund eine halbe Milliarde Tweets am Tag (~6000 Tweets/s) speichern und verarbeiten muss [5] oder hochmoderne Maschinen ebenfalls Tausende von Sensoren besitzen, die mehrmals pro Sekunde Daten erfassen und an einen Server senden können, kommt man schnell zu der Überlegung, wie es überhaupt möglich sein kann, Herr über solche Datenmengen zu werden – und dies wohlgemerkt in Echtzeit!

Hierzu benötigt es spezielle Speichermöglichkeiten und Data Mining Mechanismen, um solche rapide ansteigenden Datenbestände in Echtzeit aufnehmen und themenbezogen auswerten zu können. Es kommt darauf an, „die richtige Aktion zur richtigen Zeit und am richtigen Ort auszuführen“[1]. Ein aktuell sehr bekannter Vertreter hierfür ist Apache Spark, ein Open Source Framework, welches für die Echtzeit-Analyse von verteilten Daten eingesetzt werden kann. Es kann auf verschiedenen Systemen installiert werden und somit theoretisch auch ältere Frameworks ersetzen oder ergänzen, ohne z. B. gleichzeitig die Datenbasis umziehen zu müssen. Neben Apache Spark gibt es noch etliche weitere Frameworks, die sich je nach Einsatzgebiet ebenfalls zur Datenanalyse anbieten.

[1] übersetzt aus [6]

1.2 Ziele der Arbeit

Ziel dieser Masterarbeit ist es, eine Beurteilung über die Einsatzfähigkeit von Apache Spark zur verteilten Echtzeitanalyse von Datenströmen geben zu können. Die Themen Big Data, Streaming Data, Data Mining und Real-Time Analytics sollen zuerst erläutert und deren Zusammenhang dargestellt werden. Dazu sollen die Grundkonzepte von Spark wie die Entstehung, Architektur und Arbeitsweise, sowie die Einordnung in das Apache Hadoop Ecosystem, erläutert werden. Zur Analyse der Performance ist das Verständnis der Applikationserstellung in Spark und die Kooperation mit konsistenten Datenspeichern von hoher Bedeutung. Die beiden Datenspeicher-Lösungen Apache Cassandra und HDFS sind hierbei als Hauptaugenmerk vorgesehen und sollen anhand von Benchmarks auf Performance geprüft werden. Die beiden bei Spark mitausgelieferten Bibliotheken MLlib und Spark Streaming, die im Bereich des maschinellen Lernens bzw. der Verarbeitung von Datenströmen Verwendung finden, sollen ebenfalls analysiert werden. Den Hauptteil der Arbeit stellt die Modellierung und Evaluation verschiedener Lösungsmöglichkeiten für die Echtzeitverarbeitung von Datenflüssen mithilfe der Spark Streaming Bibliothek dar. Es sollen hierbei Daten der DEBS Grand Challenge 2014 auf Ausreißer untersucht werden. Es sollen Applikationen erstellt und auf ihre Effizienz untersucht werden. Anhand der Grundlagen, praktischen Beispiele und der implementierten Prototypen folgt abschließend die anfangs angesprochene Beurteilung.

1.3 Aufbau der Arbeit

Diese Ausarbeitung gibt die Erkenntnisse und Aufgaben im Rahmen der Masterthesis wieder. Entsprechend orientiert sich der Aufbau des Berichts am Ablauf der Arbeit: Eine Einführung in das Thema der Big-Data Analyse gibt das folgende Grundlagenkapitel 2. Hierin werden einige wichtige Begrifflichkeiten vorgestellt, ehe in Kapitel 3 die dazu passenden Tools des Hadoop-Umfelds präsentiert werden. Eine der zum Hadoop-Umfeld gehörenden Plattformen ist Apache Spark. Dessen Aufbau, Installation und Verwendung wird in Kapitel 4 behandelt. Welche Konzepte Spark zur Datenverarbeitung umsetzt, zeigt Kapitel 5 auf. Nach dieser theoretischen Vorstellung der Architektur und wichtiger Konzepte von Spark folgt in Kapitel 6 eine Einführung zur praktischen Erstellung von Applikationen. Mit den beiden Datenspeichern HDFS und Apache Cassandra setzt sich Kapitel 7 auseinander. Auch Benchmarkergebnisse von Performance-Messungen zu Schreib- und Leseanfragen, sind hierin aufgezeigt und interpretiert. Eine Vorstellung der beiden Spark-Bibliotheken zur Verarbeitung von Datenströmen und zum maschinellen Lernen folgt in Kapitel 8. Mit der praktischen Verwendung von Spark zur Echtzeit-Ausreißererkennung in Datenströmen anhand von Prototypen setzt sich Kapitel 9 auseinander. Die Erkenntnisse dieser Arbeit werden in Kapitel 10 zusammengefasst.

2 Grundlagen

Dieses erste Kapitel der Arbeit vermittelt ein allgemeines Verständnis zu den Schlagwörtern „Data Mining", „Big Data", „Streaming Data" und „Real-Time Analytics", und zeigt sowohl deren Gemeinsamkeiten wie auch Abgrenzungen voneinander auf. Da das Thema der Arbeit in die oben genannten vier Bereiche fällt, ist ein grundlegendes Verständnis dieser Begrifflichkeiten unabdingbar.

2.1 Data Mining

Data Mining (dt. Daten-Abbau / Datenauswertung) benennt den Prozess der Musterfindung in Datenmengen. Es ist der

> *[...] non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data. – zitiert aus [7]*

Es wird hierbei versucht, implizite, bislang unbekannte, aber potentiell nützliche und neue Erkenntnisse über Muster in Daten zu extrahieren. Das Zugriffsergebnis wird beim Data Mining auf Anfrage berechnet, im Gegensatz z. B. zu Datenbankabfragen, bei denen das Ergebnis lediglich ausgelesen werden muss. Eine Mining-Anfrage auf völlig unstrukturierte Daten, bei denen ebenfalls das Ergebnis individuell berechnet werden muss, fällt in den Bereich des Text Minings. Hier liegen die Daten in willkürlicher Form vor. Um bei Data Mining sinnvolle Ergebnisse zu erzielen, bedarf es meist sehr viel Vorarbeit durch Selektion und Transformation der zugrunde liegenden Daten. Somit macht der eigentliche Data Mining-Schritt nur einen Bruchteil des Gesamtaufwands zur Wissensgewinnung auf Basis von Datenbeständen aus, was Abbildung 1 verdeutlicht.

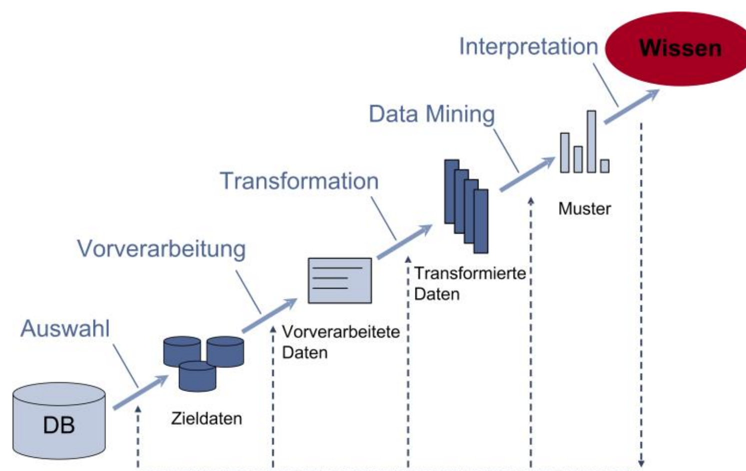

Abbildung 1: Die Aufgaben rund um den Data Mining Prozess[2]

[2] Bildquelle: http://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/lexikon/daten-wissen/Business-Intelligence/Analytische-Informationssysteme--Methoden-der-/Members/chamoni/KDD-Prozess.jpg

Der Bereich des Data Minings, also der Informationsgewinnung durch Musterfindung in Daten, ist sehr umfangreich und interdisziplinär. Es bedarf der Zusammenarbeit verschiedener Experten, die durch ihre gemeinsame Fachkenntnis das Wissen besitzen, diese Muster zu deuten und die Informationen zu verwerten [8]. Letztendlich hat die Verarbeitung der Daten durch Algorithmen das Ziel, Prozesse zu optimieren, Risiken zu minimieren, Gefahren zu umgehen – im Grunde alles, was dem Unternehmen einen positiven Mehrwert verspricht. Da der Aufgabenbereich des Minings sehr fachgebunden ist, gibt es hier kein einheitliches Framework, welches für alle Bedürfnisse eingesetzt werden kann. Um dennoch einen gemeinsamen Nenner schaffen zu können, gibt es technik- und fachunabhängige Prozessdefinitionen, welche die grundlegenden Schritte im Data Mining Verfahren aufzeigen und Hilfestellungen zu möglichen Fragen, Aufgaben, Dokumentation usw. als eine Art Checkliste, bereitstellen.
Abbildung 2 zeigt die meistverwendeten Modelle im Data Mining Bereich. In den nächsten Abschnitten wird das bekannteste Modell, CRISP-DM, kurz vorgestellt.[3]

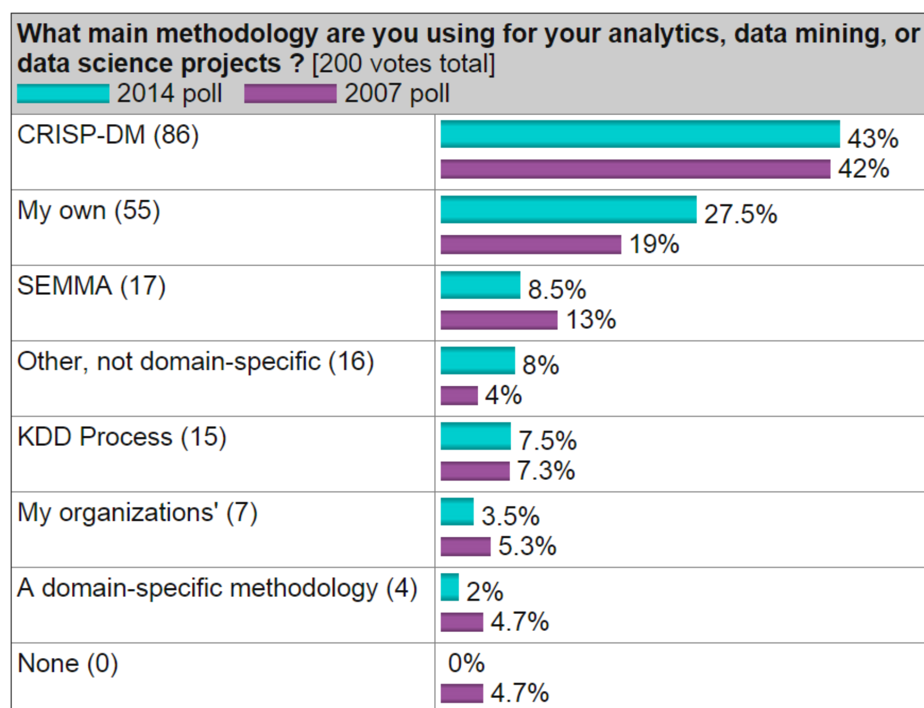

Abbildung 2:Umfrage zu den bekanntesten Data Mining Prozessmodellen[4]

CRISP-DM ist heute der meistgewählte Ansatz, der zum Prozessablauf im Data Mining verfolgt wird. Die Abkürzung steht dabei für „Cross Industry Standard Process for Data Mining“. Der Prozess selbst wird durch sechs Phasen beschrieben, die nacheinander durchlaufen werden und aufeinander aufbauen. Es kann aber auch zu früheren Phasen zurückgesprungen werden, wenn Änderungsbedarf in einem vorigen Arbeitsfeld besteht. Abbildung 3 stellt die

[3] Informationen zum SEMMA-Modell (zweithäufiges Modell) in [9]
[4] Bildquelle: http://www.kdnuggets.com/2014/10/crisp-dm-top-methodology-analytics-data-mining-data-science-projects.html, Stand 2014; verglichen mit 2007 (aufgerufen am 09.04.2015)

standardmäßigen Phasenfolgen dar, die einzelnen Schritte des Modells können aber auch in beliebiger Reihenfolge durchgeführt werden. Auch das Auslassen von Aktionen ist möglich, es macht jedoch Sinn, alle Phasen zu durchlaufen, um ein bestmöglich aussagendes und nachvollziehbares Ergebnis zu erhalten [8]. Der Gesamtprozess als solches sollte ebenfalls iterativ durchgeführt und somit regelmäßig Aktualisierungen ermöglicht werden. Aus folgenden Phasen setzt sich das CRISP-DM Modell zusammen:

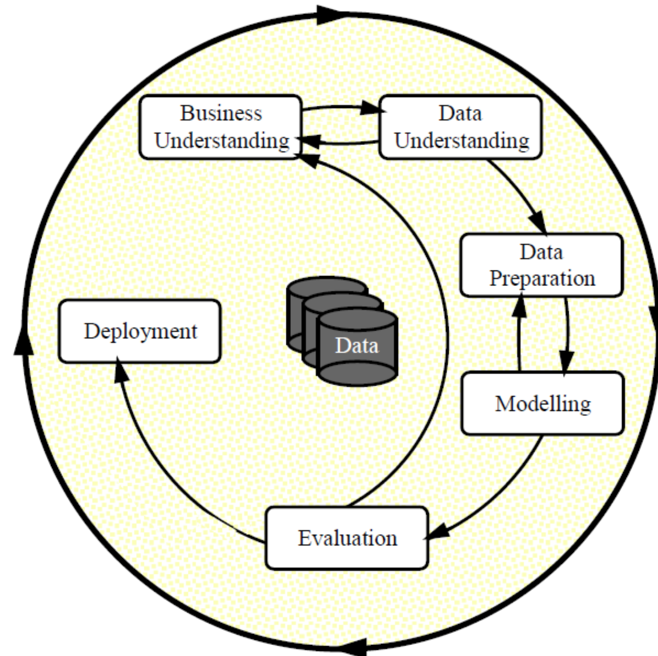

Abbildung 3: Prozessschaubild zu CRISP-DM [8]

- *Business Understanding*
  Wichtig für das Einsetzen von Data Mining Verfahren ist ein fachspezifisches Wissen über die Geschäftsobjekte, die aktuelle Situation, die Ziele der Untersuchung und sonstige geschäftliche Gegebenheiten. Ohne die nötigen Grundkenntnisse und Zielvorstellungen ist eine Mustersuche bzw. spätere Interpretation zwecklos.
- *Data Understanding*
  Im nächsten Schritt ist es wichtig, die benötigten Daten aufzubereiten. Dazu gehört das Zusammentragen und Sichten der Daten, Beurteilung derer Qualität und Vollständigkeit, sowie ggf. eine erste Statistik. Eine Datenbeurteilung ist wichtig, um feststellen zu können, ob diese Daten überhaupt eine akzeptable Grundlage für Analysen bilden.
- *Data Preparation*
  Anschließend müssen die Daten selektiert, bereinigt und vereinheitlicht werden. Um gute Ergebnisse erzielen zu können, sollten Daten von unterschiedlichsten Quellen möglichst nur relevante Informationen enthalten und müssen das gleiche Format besitzen. Nur so können die Analysewerkzeuge korrekt arbeiten.

- *Model Building*
  In dieser Phase werden aus den zugrundeliegenden Daten mithilfe von Analysetools Modelle und Regeln zur Klassifikation und Vorhersage gelernt. Eine Teilmenge der Daten dient dabei als Trainings-Set. Mit diesem werden die Modelle durch Musterfindung und Interpretation gelernt.
- *Testing & Evaluation*
  Hier werden die erstellten Modelle auf Verwendbarkeit und Genauigkeit geprüft. Dazu wird ein Test-Set, welches wieder aus einer bestimmten Menge an Daten zusammengesetzt wird, verwendet. Es kann damit eine Aussage über die Qualität des Modells getroffen werden. Je nach Verfahren und Modell wird ein drittes Datenpaket als Validierungs-Set benutzt. Mit diesem wird versucht, weitere Details und Parameter zur Optimierung herauszufinden, um somit das Modell genauere Ergebnisse erzielen zu lassen. Diese Ergebnisse werden nach der Optimierung dann von den entsprechenden Experten untersucht und interpretiert. Aus den Modellergebnissen müssen letztendlich Erkenntnisse bzw. Folgen für die unternehmerische Tätigkeit abgeleitet werden.
- *Deployment*
  Nachdem ein zufriedenstellendes Ergebnis über das Modell und dessen Prüfung erzielt wurde, wird dieses an die Verantwortlichen des Projekts übermittelt und über mögliche unternehmerische Folgen diskutiert. Das Erlernte wird quasi nach Außen weitergegeben und umgesetzt. Wichtig hierbei ist, eine Modellüberwachung durchzuführen, um ggf. zukünftige Änderungen und Trends schnellstmöglich erkennen und auf diese reagieren zu können. [8], [9]

2.2 Big Data

Der Begriff „Big Data“, sinngemäß mit „viele Daten“ oder „große Datenmengen“ zu übersetzen, ist heute zu einem der Schlagwörter im Bereich des Data Minings geworden. Analyseverfahren werden nicht selten auf Datenmengen im Petabyte-Bereich angewendet. Sie müssen dabei besonders mit den drei Dimensionen umgehen können, in welche sich Datenbestände einteilen lassen: Mit der Größe des Datenbestandes (Volume), der Vielfalt der Datenquellen und Datenstrukturen (Variety), sowie der Geschwindigkeit (Velocity) mit welcher sich die Daten verändern und entsprechend verarbeitet werden müssen [10], [11], [12]. Die folgende Abbildung verdeutlicht diese drei „V’s“ im Big Data Umfeld:

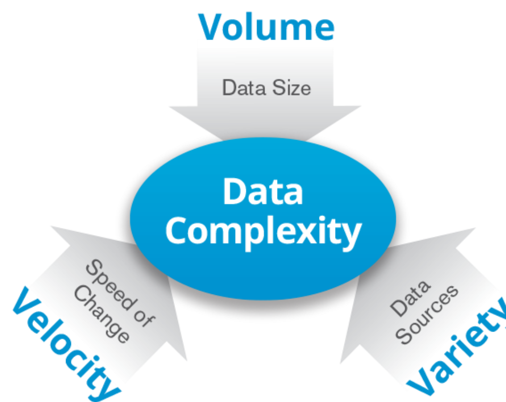

Abb. 1: Die drei Dimensionen ("V's") im Big Data Umfeld[5]

Big Data behandelt also das Thema der großen Datenbestände, mit welchen heutige Data Mining-Anwendungen umgehen können müssen. Nimmt man Twitter als Beispiel, so hat der gesamte Datenbestand eine Größe von etlichen Petabyte, die Geschwindigkeit liegt bei etwa 6000 Tweets pro Sekunde, die neu hinzukommen [5] und die Vielfalt in den sendenden und abrufenden Clients, wie z. B. PC, Smartphone und Analyseserver, sowie den verschiedenen Medienformaten wie Text, Fotos usw.

2.3 Streaming Data

Unter dem Begriff Streaming Data versteht man Datenflüsse, die kontinuierlich stattfinden. In Streaming Data Systemen müssen die regelmäßig ankommenden Datenpakete schnell genug gespeichert und ggf. verarbeitet werden können, damit es keinen Datenstau oder gar Pufferüberlauf gibt, denn dies würde sonst zu Verlust ankommender Informationen führen. Der Begriff Streaming Data hängt entsprechend mit dem der „Real-Time“ (dt. Echtzeit) zusammen, denn es muss hier eine ausreichende Geschwindigkeit zur Datenverarbeitung garantiert werden, um dauerhaft eine korrekte Arbeitsweise zu garantieren. In Kapitel 2.4 wird das Thema Echtzeit näher erläutert.

In [13] werden im Bereich Streaming Data folgende drei Thematiken unterschieden: In welchen Bereichen Streaming Data zum Einsatz kommt („Sources of Streaming Data“), was Streaming Data von anderen Datenarten unterscheidet („Why Streaming Data is different“) und welche Veränderungen die Datenflüsse für Infrastruktur und Algorithmen mit sich bringen („Infrastructure and Algorithms“). Kapitel 2.3.1 - 2.3.3 erläutern diese drei Themenbereiche detaillierter.

[5] Bildquelle: http://www.datameer.com/images/product/big_data_hadoop/img_bigdata.png (Stand März 2015)

Ein weiterer Unterschied zu früheren Systemen, wie Data-Warehouse Lösungen, ist die Strategie, mit der Daten verarbeitet werden. Während vor dem Aufkommen der Streaming Frameworks eine Batchverarbeitung der Standard war, ist heute die neue Verarbeitungsstrategie unverzichtbar, wenn es um Auswertungen in Echtzeit geht. Kapitel 2.3.4 nimmt Bezug auf den Unterschied zwischen Batchverarbeitung und Streaming.

2.3.1 Quellen von Streaming Data

In diesem Abschnitt werden die am häufigsten anzutreffenden Einsatzbereiche für Streaming Data Systeme vorgestellt. [13] definiert hierfür fünf Kategorien, in welche sich Streaming Data Anwendungen einteilen lassen. Auch wenn täglich neue Anwendungen hinzukommen, lassen sich diese meist in eine der unten aufgeführten Szenarien eingliedern.

Operatives Monitoring

Hier wird Streaming Data eingesetzt, um physische Systeme zu überwachen. Mithilfe spezieller Software und/oder Hardware wird die Performanz und Korrektheit des Zielsystems überwacht, beispielsweise über Verfolgung der CPU-Auslastung, Sensordaten zu Temperatur oder Stromverbrauch.

Web-Analysen

Durch das Aufkommen des kommerziell ausgerichteten Internets durch eCommerce-Systeme wurde der Ruf nach Aktivitätsverfolgung der Webseitenbesucher lauter. Es wurden deshalb Systeme entwickelt, mit denen es möglich ist, Ereignisse und somit das Surfverhalten der Benutzer rekonstruieren zu können.

Online-Werbung

Das heute bekannteste Umfeld für den Einsatz von Streaming Data ist das Anzeigen passender Werbung auf unterschiedlichsten Webseiten. Die Herausforderung hierbei ist es, die Daten umgehend auswerten zu können, sodass dem Benutzer schnellstmöglich nach Aufruf z. B. einer Produktseite, passende Werbung zu entsprechenden Angeboten angezeigt wird.

Soziale Medien

Im Bereich der sozialen Medien wie Facebook und Twitter kommen ebenfalls Unmengen an Daten zusammen. Wie bereits in Kapitel 2.2 erwähnt, kommen auf Twitter sekündlich mehrere Tausend neue Nachrichten auf unterschiedlichsten Wegen in das Netzwerk. Hier findet ein kontinuierlicher Datenfluss im extremen Ausmaß statt. Auch diese Daten müssen in Echtzeit gespeichert und verarbeitet werden können, um eine Aktualität gewährleisten zu können (beispielsweise um gezielte Werbung zum aktuellen Status anzeigen oder Nachrichtenschlagzeilen bei massenhaft ähnlichen Posts generieren zu können). [13]

Mobile Daten und das „Internet of Things“

Durch Einführung der ersten Smartphones wie dem iPhone im Jahr 2007 [3] oder Release des Android-Betriebssystems ein Jahr später [4] wurde die Welt mobil und konnte von diesem Zeitpunkt ab von überall Nachrichten senden und empfangen. Durch diese Neuerungen wurden die bisherigen Datenflüsse um ein Vielfaches größer. Dieser Effekt wird durch den Trend des „Internet der Dinge“ bzw. „Internet of Everything“, in dem jeder erdenkliche Gegenstand mit Daten sendenden und ggf. empfangenden Chips ausgestattet wird, nochmals drastisch verstärkt. Wie auch in den Szenarien zuvor muss es realistisch bleiben, all die an einem System ankommenden Daten zu erfassen und entsprechende Anfragen in Echtzeit zu beantworten, unabhängig vom Ausmaß der Datenflussgröße.

2.3.2 Was Streaming Data anders macht

Streaming Data unterscheidet sich in vielen Hinsichten von anderen Daten. Datenflüsse zeigen drei wesentliche Charaktereigenschaften, die kennzeichnend und besonders für solche Datenströme sind [13]. Nachfolgend werden diese kurz vorgestellt.

Immer an, immer fließend

Wie es der Name Streaming Data bereits vermuten lässt, fließen in solchen Systemen dauerhaft Daten. Damit dies der Fall sein kann, müssen entsprechend Sender und Empfänger aktiv sein, um Daten austauschen zu können. „Immer an“ bezieht sich dabei auch auf die Verfügbarkeit der Systeme, sowie bisher empfangener Daten. Diese wachsen kontinuierlich über die Zeit an. Datenfluss-Umgebungen müssen so modelliert werden, dass sie Informationen schnell genug entgegennehmen und verarbeiten können, um diesen kontinuierlichen Datenfluss nicht zu unterbrechen. [13]

Lose strukturiert

Diese Eigenschaft entsteht dadurch, dass Daten in Streaming Data Systemen über unterschiedlichste Wege gesendet werden. Hier kann es passieren, dass immer gleich aufgebaute Daten von einem Teilsystem, mit willkürlich aufgebauten Informationssätzen eines anderen zusammenkommen. Als Beispiel seien hier soziale Netzwerke genannt, hier kommen die unterschiedlichsten Informationen zusammen, meist in Form von Texten, Bildern, Video usw., was ebenfalls wieder unstrukturierten Daten entspricht. Ein weiterer Grund für die lose Strukturierung ist das sich im Laufe eines Projekts eventuell ändernde Dateninteresse. Möchte ein Unternehmen beispielsweise Daten sammeln, weiß aber noch nicht exakt in welchem Umfang, wird meist damit begonnen, möglichst viele Daten zusammenzutragen (d.h. ohne zu filtern und ohne eine feste Informationsstruktur zu beachten), ehe dann die Vorgaben zur Datenerhebung nach und nach an die wirklichen Bedürfnisse des Unternehmens angepasst werden. [13]

Hohe Kardinalität im Datenspeicher

Die Kardinalität bezeichnet im Streaming Data Bereich die Anzahl verschiedener Elemente, die z. B. im Datenspeicher vorhanden sind (vergleichbar mit einer „distinct"-Abfrage in SQL). Je nach Größe und Struktur der einzelnen Datensätze kann in solchen Systemen eine extrem große Kardinalität entstehen. Zwar gibt es oftmals gleiche Datensätze innerhalb eines Systems, dies ist jedoch meist nur eine Minderheit, eingebettet in eine Umgebung von etlichen unterschiedlichsten Informationen. In Streaming Data Systemen muss z. B. durch effektive Speicherung darauf geachtet werden, dass steigende Kardinalitäten das System nicht überlasten und trotzdem eine schnelle Anfrage-Bearbeitung möglich bleibt. [13]

2.3.3 Infrastruktur und Algorithmen

Beim Aufbau eines Systems sollten Infrastruktur und Algorithmen Hand in Hand gehen. Ein Algorithmus ohne Infrastruktur ist nicht praxisnah und eine Infrastruktur ohne passende Algorithmen laut [13] nichts weiter als Ressourcenverschwendung. Es ist deshalb wichtig, beides aufeinander aufbauend zu entwickeln. Sehr hilfreich kann es auch sein, das System einer möglichst großen Gruppe von Leuten zugänglich zu machen. Sie können das System untersuchen, eigene Interessen einbringen und es ggf. entsprechend weiterentwickeln, was einen großen Vorteil für ein Projekt darstellen kann. [13] Ein solches Open-Source Projekt stellt Apache Spark dar, an ihm arbeiten gleichzeitig viele sog. Kontributoren, die sich an der Entwicklung der Software beteiligen und ihre Interessen einbringen.

2.3.4 Batchverarbeitung mit MapReduce

Bereits in früheren Data-Warehouse Anwendungen gab es die Voraussetzung, große Datenmengen analysieren zu können. Die hierin durchgeführten Analysen konnten allerdings nur über Batchverarbeitung durchgeführt werden, da es zu diesem Zeitpunkt noch keine etablierten Tools und Konzepte zur Echtzeitanalyse von Datenbeständen gab. *MapReduce* war der damalige Standard für die Batchverarbeitung. Durch technische Neuerungen, besonders im Hardwarebereich, wurde der Grundstein für Echtzeit-Analysen gelegt. Mit dem Aufkommen von verteiltem In-Memory Computing wurde der Ruf nach spontanen, dem Nutzer zugeschnittenen Berechnungen laut. Auch die aus drei Phasen bestehende Verarbeitung[6] im MapReduce Modell sollte verbessert werden. Während im batchverarbeitenden System die nächste Phase erst begonnen werden konnte, nachdem die vorherige komplett abgeschlossen war, wurde auch hier der Ruf nach Beschleunigung laut.

[6] Diese drei Phasen werden im folgenden Unterkapitel zu MapReduce näher erläutert.

Um das Verständnis und den Unterschied zwischen diesen traditionellen und modernen Systemen zu verdeutlichen, wird im folgenden Abschnitt das MapReduce Modell zur Batchverarbeitung vorgestellt. Anschließend wird auf die neuen Anforderungen und deren Umsetzung eingegangen.

Das MapReduce-Modell

Das bekannteste System zur Batchverarbeitung ist das Open Source Framework Hadoop der Apache Software Foundation. Als Verarbeitungsmodell kommt hier MapReduce zum Einsatz. Es ermöglicht eine skalierbare und verteilte Verarbeitung großer Datenbestände (Big Data) im Netzwerk, bestehend aus herkömmlichen Rechnersystemen. Mit diesem können Analysen auf sehr große, vorhandene Datenbestände gefahren werden, ohne dafür spezielle Hardware anschaffen zu müssen. Wie jedoch bereits im Oberkapitel angeschnitten, ist MapReduce lediglich zur Batchverarbeitung geeignet, was ein Umgang mit Streaming Data verhindert. MapReduce besteht aus drei Phasen: Map, Shuffle und Reduce. Der Prozess selbst verarbeitet Schlüssel-Wert-Paare und gibt solche auch wieder an den Anwender zurück [14]. Der Anwender ist dabei nur für die Logik des Map- und Reduce-Schritts zuständig, während das Framework die anderen Verantwortlichkeiten, wie das Zusammentragen und Weitergeben von Daten, Netzwerküberwachung, Synchronisation, Fehlerbehandlung usw. übernimmt [14]. Durch die Skalierbarkeit kann das gleiche MapReduce-Programm, sowohl auf kleinen Datensätzen, als auch auf Datenbeständen im Petabyte-Bereich eingesetzt werden.

Anhand eines kleinen Beispiels zur Wörterzählung soll die Architektur von MapReduce und der Ablauf dessen Algorithmus verdeutlicht werden: Es soll gezählt werden, wie häufig welche Wörter in einem Buch vorkommen[7]. Der Inhalt sei hierbei bereits partitioniert und auf die vorhandenen Nodes aufgeteilt. Jeder Node generiert nun im Map-Schritt ein Schlüssel-Wert-Paar für jedes einzelne Inputwort. Schlüssel ist dabei das Wort selbst und Wert jeweils die 1, pro Wort gibt es also ein Key-Value-Paar, z. B. (ich, 1), (und, 1), (du, 1), (und, 1), (er, 1). Da Wörter in längeren Texten meist mehrfach vorkommen, können diese durch einen zusätzlichen Kombinationsschritt innerhalb eines Nodes direkt aufsummiert werden, beispielsweise zu (ich, 1), (und, 2), (du, 1), (er, 1). Nach Abschluss der Map-Phase organisiert das Framework dann in der darauf folgenden Shuffle-Phase[8] automatisch eine Sortierung und weitere Verteilung der Teilergebnisse auf die Reduce-Nodes. Im abschließenden Reduce-Schritt werden die auf den Map-Nodes vorberechneten Teilergebnisse verarbeitet und wieder zu Schlüssel-Wert-Paaren

[7] In diesem Beispiel wird Text verarbeitet, es kann sich aber auch um beliebige andere Formate, wie z. B. Audiodateien handeln.
[8] In [14] - Kapitel 4.3 auch „Combiner Function“ (dt. Kombinierer-Funktion) genannt

zusammengefasst. Jeder Node ist dabei als einziger für ein Wort (bzw. mehrere Wörter) zuständig, d.h. ein bestimmtes Wort wird nun auf genau einem Node im Netzwerk ausgewertet. Kam im vorherigen Schritt beispielsweise auf drei Nodes das Wort „und" mehrmals vor (Node A: 10-mal, Node B: 2-mal, Node C: 3-mal), würde das Zwischenergebnis wie folgt aussehen: (und, [10, 2, 3]). Es wird hiermit das Key-Value-Paar (und, 15) erzeugt und mit den Ergebnissen der anderen Wortzählungen zu einer Gesamtliste von Schlüssel-Wert-Paaren zusammengefasst. Diese wird dann an den Benutzer bzw. die Anwendung zurückgegeben. Man hat somit über einen verteilten Algorithmus bestimmt, wie oft welches Wort in einem Text vorkommt (z. B. kommt das Wort „und" insgesamt 15-mal im Buch vor).

Das folgende Schaubild illustriert nochmals die einzelnen Schritte im Beispiel:

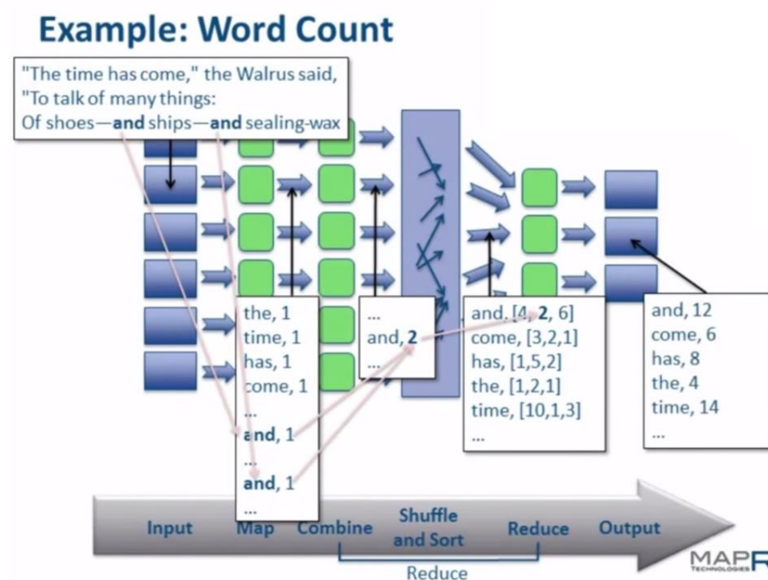

Abbildung 4: MapReduce am Beispiel Wörterzählung [15]

Ein Beispiel mit Pseudocode, welches ebenfalls das Wörterzählen realisiert, wird in [14], Kapitel 2.1 gezeigt. Es sind lediglich zwei minimale Methoden für Map und Reduce notwendig, um das Zählen von Wortmengen durchzuführen:

```
(1)   map(String key, String value):
(2)     // key: document name
(3)     // value: document contents
(4)     for each word w in value:
(5)       EmitIntermediate(w, "1");
```

Codeblock 1: Map-Methode

```
(1)   reduce(String key, Iterator values):
(2)     // key: a word
(3)     // values: a list of counts
(4)     int result = 0;
(5)     for each v in values:
(6)       result += ParseInt(v);
(7)     Emit(AsString(result));
```

Codeblock 2: Reduce-Methode

Der Weg von der Batch- zur Streaming-Verarbeitung

Wie im voran beschriebenen Beispiel deutlich wird, ist bei MapReduce eine Verarbeitung nur auf einer zu Beginn vollständigen Datenbasis möglich. Auch die Berechnung einzelner Zwischenergebnisse muss abgeschlossen sein, ehe mit dem nächsten Schritt fortgefahren werden kann. MapReduce arbeitet außerdem nur auf Festplatten, auch Zwischenergebnisse müssen aufgrund der Verteilung dort festgeschrieben werden, um dann von den nächsten Nodes ausgelesen und weiterverarbeitet werden zu können. Diese und weitere organisatorische Aufgaben, die durch den Aufbau und den Ablauf des Prozesses anfallen, bremsen das System stark aus und machen damit eine Echtzeitverarbeitung oder gar einen Umgang mit Streaming Data unmöglich. Durch entsprechende Architekturveränderungen wurde deshalb versucht, eine Echtzeit-Fähigkeit, sowie Streaming-Verarbeitung umzusetzen. Eine schnellere Jobabarbeitung sollte dabei durch Reduktion von Overhead und Festplatten-I/O, sowie Verringerung von Zwischenberechnungen ermöglicht werden.

Prozessoptimierungen zur Verkürzung der Arbeitszeit und Umsetzung von In-Memory Computing sind beim Thema Echtzeit zwei wesentliche Faktoren. Durch Auslagerung der Arbeitsdaten von der Festplatte in den Arbeitsspeicher der jeweiligen Nodes ist eine Geschwindigkeitssteigerung um den Faktor 100 möglich [16]. Durch die im Laufe der letzten Jahre erschwinglich gewordenen Hardwarepreise ist diese Umsetzung auf Standardrechnern erst ermöglicht worden. Wichtig zu erwähnen ist hier auch, dass es sich bei In-Memory Computing nicht nur um das Thema der Speicherung dreht, sondern auch um eine effektivere Berechnung. Diese schließt hier ein neues Paradigma ein: Während bei Systemen mit kleiner Datenbasis die Informationen über das Netzwerk an einen verarbeitenden Rechner gesendet wurden, verfolgt man in den heutigen Big Data Systemen exakt das umgekehrte Prinzip und transportiert den Programmcode zu den Nodes, welche die Daten enthalten. Dadurch wird der Netzwerkverkehr und daraus resultierend auch die Laufzeit reduziert, da es einfacher ist, das Programm zu übertragen als Unmengen von Daten.

Eine weitere Verbesserung des traditionellen MapReduce-Modells bringt eine effektive Caching-Strategie mit sich[9], bei der ebenfalls unnötige Festplattenzugriffe erspart bleiben. Ein weiterer wichtiger Aspekt zur Realisierung einer Echtzeitumgebung ist die Optimierung von Abfragen auf die Datenbasis. Durch Verwendung von neuartigen, spaltenorientierten Datenmodellen und optimierten verteilten Datenbanksystemen können auch die SQL-artigen Abfragen auf die Datenbasis schneller durchgeführt werden. Da bei Analyseanfragen im Data Mining oftmals nur wenige Kennzahlen zur Berechnung eines Wertes benötigt werden, eignet sich ein spaltenbasierter Datenabruf besser als ein zeilenbasierter. Bei der herkömmlichen

[9] Hier sei beim Thema Apache Spark die sog. „Resilient Distributed Datasets“ (RDD) erwähnt. Näheres dazu in Kapitel 5.1.

Art mussten z. B. zur Summenberechnung einer Spalte alle Zeilen durchlaufen und die Werte der entsprechenden Spalte ausgelesen und verrechnet werden. Statt also unnötig über die ganze Tabelle iterieren zu müssen, wird bei spaltenbasiertem Tabellenaufbau nur ein Bruchteil der Tabelle gelesen, nämlich exakt der Inhalt der benötigten Spalten.
Die oben beschriebenen Maßnahmen lassen eine Berechnung durchaus im Millisekunden-Bereich zu. Allerdings können diese verbesserten Programme nach wie vor nur auf vorhandenen Daten arbeiten und sind deshalb ebenfalls im Bereich der Batchverarbeitung anzusiedeln.

Eine Art Streaming Data Verarbeitung ist allerdings notwendig, um neu ankommende Daten in Analysen schnellstmöglich beachten zu können. Diese Strategie nennt man „Micro-Batching“. Es wird hier ein Trick angewendet: Der Datenstrom wird als Folge von kleinen Batcheinheiten angesehen. In kleinen Intervallen werden dabei die neuen Daten zu einem Batch-Paket zusammengefasst, und der oben beschriebene, optimierte Prozess unabhängig darauf angewandt. Die Daten der vorherigen Verarbeitung werden dabei nicht verworfen, sondern aktualisiert. Die Technik des Micro-Batchings kommt u.a. in Apache Spark zum Einsatz. Systeme, die sowohl Batch- als auch Streaming-Verarbeitung betreiben, sind heute meist durch die sog. Lambda-Architektur beschrieben. Diese besitzt neben einem Batchlayer auch eine separate Schicht zur Verarbeitung von Datenströmen. Eine Beschreibung dieser Architektur findet sich im folgenden Kapitel.

2.3.5 Lambda-Architektur

Wie auch in batchorientierten Systemen ist ein Streaming Framework aus mehreren Tools aufgebaut, die verwaltet werden und miteinander kommunizieren müssen. Je nachdem, welche Werkzeuge zum Einsatz kommen, unterscheiden sich die Systeme voneinander. Um diesen Aufbau zu abstrahieren, wird in [17] eine allgemein gültige Architektur vorgestellt, nach der die Streaming Frameworks aufgebaut sein sollten, um den verschiedenen Aufgaben, die sowohl im Batch- als auch im Streaming-Bereich anfallen, als gesamtheitliches System nachkommen zu können. Diese Architektur wird als Lambda-Architektur bezeichnet [17] und im Nachfolgenden grob erläutert.

Die Lambda-Architektur ist dynamisch und erlaubt beliebige Anfragen auf beliebige Daten. Da diese teils sehr ressourcenaufwendig sein können, wird hierin der Ansatz einer Vorberechnung über sogenannte Views verfolgt, die dann später vom Anwender abgefragt werden können [17].

Die Architektur selbst besteht aus drei Schichten. Folgende Abbildung zeigt deren Aufbau:

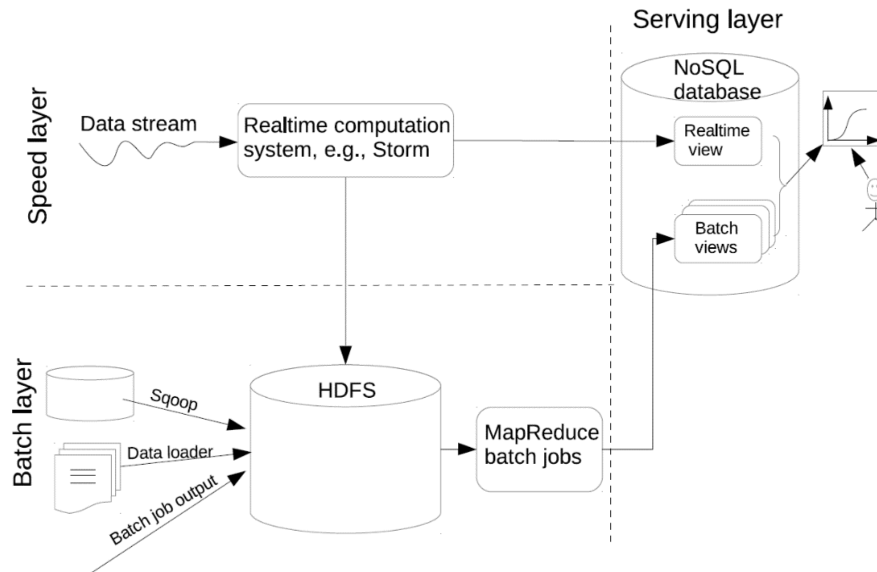

Abbildung 5: Die dreischichtige Lambda-Architektur [18]

Die im vorigen Abschnitt erwähnten Views werden im Serving Layer abgelegt, von wo aus der Anwender seine Anfragen auf diese absetzen kann. Jeder der beiden anderen Layer speichert seine berechneten Views in der NoSQL-Datenbank des Serving Layers. Sollte eine Anfrage beide Datenbestände betreffen, werden diese hier entsprechend zusammengetragen und zurückgegeben.

Die Batchschicht ist für das Berechnen von Views über die zum Startzeitpunkt vorhandene Datenmenge zuständig. Eine Berechnung beginnt von neuem, sobald die vorherige abgeschlossen ist. Damit werden die vorhandenen Views über die mittlerweile ggf. veränderte Datenbasis mithilfe des Batchings regelmäßig aktualisiert und über MapReduce-Jobs an den Serving-Layer übergeben. Es wird ebenfalls eine Sicherung auf einem eigenen Datensystem durchgeführt.

Der Speed Layer arbeitet parallel zum Batching. Er nimmt ankommende Datenströme entgegen und führt ebenfalls Berechnungen durch, allerdings nur auf diesen aktuellen Daten. Die berechneten Views werden dabei an den Serving-Layer übermittelt und die Rohdaten im Dateisystem des Batchlayers persistiert.

Fehler können durch Roll-Backs der Datenbasis des Batchlayers und anschließender Neuberechnungen behoben werden. Neben der Sicherungs- und Wiederherstellungsfunktion bietet das Dateisystem der Batchschicht auch den Vorteil, dass neue Views einfach aufgrund der persistierten Gesamtdaten definiert werden können. Auch eine Rekapitulation der Datenerfassung wird durch diese zusätzliche Speicherart ermöglicht. Die Lambda-Architektur bringt also viele Vorteile gegenüber solcher Systeme mit, die nur aggregierte und verarbeitete Daten dauerhaft speichern. [18], [17]

2.4 *Real-Time Frameworks*

In diesem Kapitel werden die Voraussetzungen für Real-Time Frameworks diskutiert, sowie vorhandene Open Source Lösungen für die einzelnen Bereiche vorgestellt.

2.4.1 Historische Entwicklung

Während noch vor rund zehn Jahren eine Echtzeitverarbeitung bei damaligen Data-Warehouse Systemen unvorstellbar war, gibt es heute unzählige Systeme, die Petabytes an Daten binnen weniger Sekunden oder gar Millisekunden analysieren. Dieser Wandel wurde erst durch die entsprechenden technischen Weiterentwicklungen ermöglicht. Durch eine neuere, leistungsfähigere Hardware wie CPUs, HDD/SSD, RAM usw. ist es überhaupt technisch möglich geworden, heutige Geschwindigkeiten zu erzielen. Entsprechend konzipiert und optimiert werden mussten dabei auch besonders die Anwendungen, die auf solchen Systemen zum Einsatz kommen. Die Möglichkeit an sich, große Datenmengen speichern zu können, ist nicht neu, das Innovative solcher Systeme ist die Möglichkeit, diese großen Informationsbestände über dynamische Analyseanfragen binnen kürzester Zeit zu durchforsten.

2.4.2 Anforderungen und Voraussetzungen an Echtzeit-Systeme

Allgemein hängen die Anforderungen an solche Systeme stark von den Interessen und dem Tätigkeitsumfeld der Anwender ab. Entsprechend als *squishy* (dt. matschig, Bedeutung in diesem Kontext „verwaschen, unklar“) wird in [6] der Echtzeitbegriff beschrieben. Anwender geben kaum Vorgaben zu maximalen Laufzeiten, die Analysen brauchen dürfen. Wichtig ist hier vielmehr, dass ein System auch bei wachsendem Datenbestand und Lastspitzen innerhalb einer vertretbaren Zeitspanne Ergebnisse liefert, der Anwender also nicht zu lange auf Antworten warten muss. Die hier an das System geforderte Bedingung entspricht der in Kapitel 2.4.3 vorgestellten weichen Echtzeit.

Als Beispiel wird in [6] eine Feststellung von Kreditkarten-Betrug genannt. Es gibt einfache Fälle wie Feststellung über Lokalität (Kreditkartenzahlung in kürzester Zeit in weit voneinander entfernten Ortschaften), aber auch viel komplexere Analysen, die einen Betrug feststellen können. Dies ist ein sehr gutes Beispiel für zeitkritische Echtzeitanalysen, denn hier gilt es möglichst wenig Zeit zu benötigen, was gleichzeitig auch heißt, Kosten zu sparen.[10]

> *„When you're dealing with fraud, every lost minute translates into lost money.“ [6]*

Kapitel 3 aus [6] beschreibt den Real Time Begriff als Architekturstruktur, die es ermöglicht, Rückmeldung auf Anfragen zu geben, ohne dass die gesendeten Daten zunächst in einer Datenbank persistiert werden müssen. Daten werden also „näher an der Gegenwart als in der

[10] Es handelt sich hier deshalb um eine harte Echtzeitbedingung, wie sie ebenfalls in Kapitel 2.4.3 erläutert wird.

Zukunft“ verarbeitet. Der Architekturaufbau folgt dabei meist der in Kapitel 2.3.5 vorgestellten Lambda-Architektur.

> *„[…] real-time means that you're processing data in the present, rather than in the future.“ [6]*

2.4.3 Echtzeit-Bedingungen

Bei Echtzeitanwendungen gibt es unterschiedliche Anforderungen an das System, die den Begriff der Echtzeit mehr oder weniger einschränken. Allgemein lässt sich sagen, dass sämtliche Echtzeitsysteme erst richtig arbeiten, wenn neben der logischen auch die zeitliche Korrektheit gegeben ist [19]. Mit dem zeitlichen Aspekt hängt auch der Begriff der Rechtzeitigkeit zusammen. Dieser lässt sich wie folgt beschreiben:

> *Rechtzeitigkeit heißt, die Ausgabedaten müssen rechtzeitig berechnet werden und zur Verfügung stehen. [19]*

Es gibt verschiedene Auslegungen dieser Rechtzeitigkeit:

> - *Angabe eines **genauen Zeitpunktes** […]*
>   *Hierbei wird exakt der Zeitpunkt t definiert, an dem eine Aktion stattzufinden hat. Diese Aktion darf nicht früher und nicht später durchgeführt werden. […]*
> - *Angabe eines **spätesten Zeitpunktes** […]*
>   *Es wird ein maximaler Zeitpunkt t_{max} angegeben, bis zu dem eine Aktion spätestens durchgeführt sein muss. Die Aktion kann aber auch früher beendet werden. Diesen Zeitpunkt t_{max} nennt man auch eine Zeitschranke oder **Deadline**. […]*
> - *Angabe eines **frühesten Zeitpunktes** […]*
>   *Es wird ein minimaler Zeitpunkt t_{min} angegeben, vor dem eine bestimmte Aktion nicht durchgeführt werden darf. Eine spätere Durchführung ist gestattet. […]*
> - *Angabe eines **Zeitintervalls** […]*
>   *Eine Aktion muss innerhalb eines durch die beiden Zeitpunkte t_{min} und t_{max} gegebenen Intervalls durchgeführt werden. […] [19]*

Je nachdem, wie hoch die Qualitätsanforderungen an ein System festgelegt sind, unterscheidet man hier zwischen drei Arten von Echtzeitbedingungen.

Die strengste Form ist die *harte Echtzeitbedingung*. Hier ist eine Bearbeitung bis zu einem bestimmten Zeitpunkt unbedingt nötig, da andernfalls Verlust oder Schaden droht. Als Beispiel nennt [19] eine Ampel, auf die ein autonomes Fahrzeug zufährt. Wenn die Ampel rot zeigt ist es wichtig, dass das Fahrzeug die Farbe schnell genug erkennt, um noch vor der Haltelinie zu stoppen. Würde das Auto in die Kreuzung einfahren, könnte es zu einem Unfall kommen. Ein weiteres Beispiel ist die in Kapitel 2.4.2 genannte Kreditkartenbetrugs-Auswertung.

Anders sieht die Situation bei der sogenannten *festen Echtzeitbedingung* aus: Hier werden zwar Vorgaben zum Aktionszeitpunkt gemacht, aber es entsteht bei Nichteinhaltung auch kein Schaden. Die Ausführung wird lediglich wertlos und kann beendet werden. Es wird hier von einer Art „Verfallsdatum" gesprochen.

Die dritte Kategorie bildet die *weiche Echtzeitbedingung*. Hier werden zwar ebenfalls Zeitpunkte angegeben, diese dienen allerdings mehr einer Orientierung und dürfen ggf. auch überschritten werden. Da es laut [19] üblich ist, dass z. B. das Auslesen von Sensoren mit einer gewissen Latenz verbunden ist, würde ein solches System in diese Gruppe eingeordnet werden. In dieser Kategorie wird trotzdem eine Echtzeit durch Vorgabe gewisser Toleranzzeiten gewährleistet. Eine erhebliche Überschreitung des Zeitrahmens darf zwar vorkommen, allerdings nur so selten, dass der Gesamtablauf einer Anwendung nicht beeinflusst wird. Bei Bildübertragungen beispielsweise kommt es manchmal vor, dass einzelne Bilder zu spät oder fehlerbehaftet ankommen. Solange ein wahrnehmbares Ruckeln im Abspielen der Videodatei jedoch ausbleibt, ist hier die Echtzeitbedingung nicht verletzt.

Folgende Abbildung veranschaulicht den Zusammenhang zwischen Nützlichkeit (Wert) und Zeitdauer der verschiedenen Echtzeitbedingungen. Ein Wert von Null bedeutet Wertlosigkeit und ein Wert darüber Gewinn, darunter Verlust.

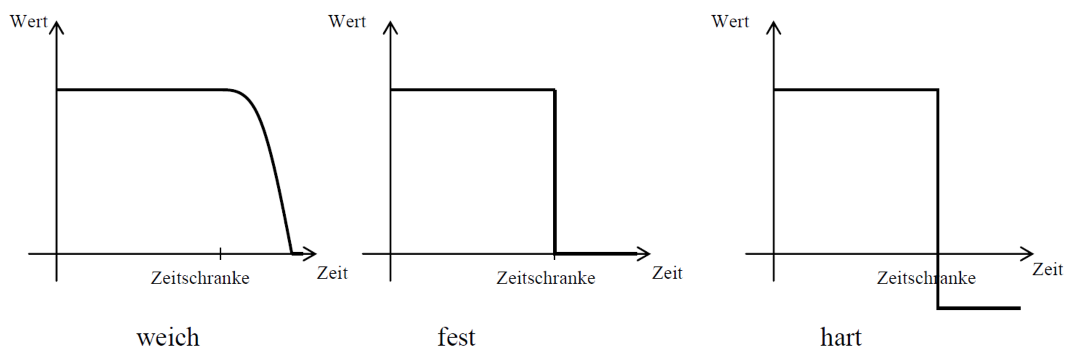

Abbildung 6: Zusammenhang von Zeit und Wertigkeit bei Echtzeitbedingungen [19]

2.4.4 Komponenten eines Echtzeitsystems

Ein Echtzeitsystem besteht in der Regel nicht aus einem einzigen, festen Gesamtframework, sondern setzt sich aus mehreren Schichten zusammen. Dabei hat jede Komponente ein spezielles Einsatzgebiet, auf welches sie sich konzentriert. In den meisten Fällen werden solche Systeme nicht von vornherein komplett aufgebaut, sondern wachsen mit dem Entstehen neuer Anforderungen. Besonders zu Beginn ist es hier oftmals sehr schwer, einschätzen zu können, welche Komponenten zukünftig ebenfalls zum Einsatz kommen könnten, sodass es wichtig (und heute auch der Standard) ist, Komponenten mit möglichst loser Kopplung zu anderen zu erstellen und zu verwenden. Damit ist eine Anpassung des Systemaufbaus durch

Hinzufügen oder Entfernen solcher Teile relativ einfach möglich. In welche Bereiche sich diese Komponenten einteilen lassen, wird nachfolgend zusammengefasst.

Daten-Sammlung

Heutige Echtzeitsysteme sind meist in Clustern organisiert und entsprechend über TCP/IP-basierte Netzwerkverbindungen verknüpft. Um Daten entgegenzunehmen kommt hier überwiegend ein einfaches HTTP-Protokoll zum Einsatz. Über weltweit verteilte Server kann eine Anfrage näher am sendenden Teilnehmer stattfinden, was die Latenz stark verringern und somit den Zeitaufwand zum Entgegennehmen der Daten minimieren kann. Als Übertragungsformat wird heute gerne JSON (JavaScript Object Notation) verwendet. Dieses Format ist zwar sehr flexibel, eine Validierung oder Deserialisierung allerdings recht aufwendig. Sollen strukturierte und wohldefinierte Daten gesendet werden, kann stattdessen Thrift oder Protocol Buffer (kurz: Protobuf) eingesetzt werden. Sie verwenden beide eine IDL (Interface Definition Language), mit der ihre Struktur definiert ist und Schnittstellen zur (De-)Serialisierung in mehreren Programmiersprachen definiert werden können. Meist sind diese Mechanismen zur Sammlung von Daten bereits auf den Servern vorhanden und kommunizieren direkt mit der Datenfluss-Schicht.

Datenfluss

Während sich in früheren Systemen jede Komponente über eigene Methoden mit anderen austauschte, wird heute versucht, über systemweite Datenflussframeworks diese spezifischen Lösungen zu vereinheitlichen. Damit ist ebenso eine systemübergreifende Kommunikation, wie auch Definition von Schnittstellen für Sender und Empfänger der Daten, möglich. Durch Speicherung an zentralen Stellen im System wird heute ein schneller Datenzugriff umgesetzt. Durch Verwendung von RPCs (Remote Procedure Calls) ist eine „At Least Once“-Speichergarantie gegeben. Eine horizontale Skalierung ermöglicht hierbei eine gleichbleibende Performanz bei steigender Anfragezahl.

Verarbeitung

Eine schnelle Verarbeitung ist in Echtzeitsystemen ebenfalls ein wichtiger Punkt, denn besonders bei großen Systemen ist dies die Komponente, die mit Abstand die meiste Zeit benötigt. Eine wichtige Erkenntnis ist hierbei, dass die einzige sinnvolle Vorgehensweise das Kopieren der Programme zu den Daten ist, anstatt, wie in alten Systemen, umgekehrt. MapReduce sorgt hier dafür, dass Berechnungen im Cluster effizient und verteilt stattfinden, ohne dabei zunächst entsprechende Kenntnisse vom Programmierer abzuverlangen. Anwendungsspezifische Anpassungen oder der Einsatz spezieller Hardware werden vermieden. Stattdessen wird ein allgemeines Programmiermodell, das die Leistung des gesamten Clusters zur Daten-

verarbeitung ausnutzt, bereitgestellt. Eine Verbesserung des MapReduce-Modells (Batchverarbeitung) stellen Mechanismen dar, die Operationen und Daten als gerichtete, azyklische Graphen (DAG, directed acyclic graph) organisieren.[11] Sie ermöglichen eine schnellere Verarbeitung aktuellster Daten und können im Streamingumfeld eingesetzt werden.

Speicherung

Auch eine schnelle Speicherung von Daten ist nötig, um in Echtzeit agieren zu können. Während in vielen Systemen relationale Datenbanken wegen ihrer Schema- und SQL-Unterstützung zum Einsatz kommen, bieten die sogenannten NoSQL-Lösungen mehr Performanz beim Lesen und Schreiben. Solche NoSQL-Datenbanken sind letztendlich eine Art Schlüssel-Wert Sammlung, die sowohl im lokalen als auch im verteilten Umfeld eingesetzt werden können. Durch Abstraktionen wird dem Anwender oftmals eine SQL-ähnliche Schnittstelle zur Verfügung gestellt, um diesen mit gewohnten Anfragen auch in diesen Key-Value Stores arbeiten lassen zu können. Durch eine *Eventually Consistency*-Strategie wird der Verwaltungsaufwand drastisch gesenkt, im Gegensatz zur strikten Einhaltung des ACID-Paradigmas[12]. Es wird hier oftmals vom BASE-Prinzip (Basically Available, Soft State, Eventual Consistency) gesprochen, welches verfolgt wird. Daraus folgt zwar eine kurzzeitige Inkonsistenz bei verteilten Datenbanken, aber auch eine enorme Performanz-Verbesserung, da hierdurch der Verwaltungsaufwand innerhalb des Clusters stark reduziert wird. Dieser Geschwindigkeitszuwachs wird in Echtzeitsystemen meist einer strengen Konsistenz vorgezogen. Um einheitliche Analyseergebnisse erzielen zu können, werden Auswertungen oftmals nur auf konsistenter Datenbasis über MapReduce-Jobs durchgeführt. Ein Streaming ist dabei gleichzeitig über eine zweite Schicht möglich. Systeme, in welchen sowohl Batchverarbeitung als auch Streaming möglich sein sollen, werden mithilfe der Lambda-Architektur aufgebaut.[13]

Datenauslieferung

Die Mechanismen zur Auslieferung von Daten an den Benutzer sind meist webbasiert. Sie lassen sich über eine Vielzahl vorhandener Frameworks einfach umsetzen. Diese Anzeigen repräsentieren dabei den Datenbestand eines bestimmten Zeitpunktes und aktualisieren sich meist automatisch mit einer gewissen Frequenz. Durch Polling[14] wird dabei zunächst festgestellt, ob es eine Änderung an der Datenbasis gibt und somit eine Aktualisierung nötig ist. Bei dieser Datenabfrage und -übertragung kommt meist AJAX[15] zum Einsatz. Zur Visualisierung werden browserunabhängige Redering-Komponenten wie SVG oder Canvas eingesetzt.

---

[11] Eine grobe Vorstellung von DAGs findet in Kapitel 5.1 statt.
[12] Informationen zum ACID-Paradigmas beispielsweise unter [13]
[13] Lambda-Architektur, siehe Kapitel 2.3.5
[14] Polling: Regelmäßiges Abfragen nach Neuigkeiten, http://de.wikipedia.org/wiki/Polling_%28Informatik%29
[15] http://www.webmasterpro.de/coding/article/ajax-einfuehrung-uebersicht.html

2.4.5 Eigenschaften eines Echtzeitsystems

Die Komponenten von Echtzeitsystemen müssen die folgenden drei Eigenschaften erfüllen, um schnell und verteilt arbeiten zu können: Hochverfügbarkeit, geringe Latenz und horizontale Skalierbarkeit. Sie garantieren zusammen ein zuverlässiges und skalierendes Cluster, welches im Bereich der Echtzeit- und Streaming-Bearbeitung eingesetzt werden kann. Diese drei Features werden nachfolgend kurz erläutert.

Hochverfügbarkeit

Im Gegensatz zu batchverarbeitenden Systemen ist im Realtime- und Streaming-Bereich eine garantierte Erreichbarkeit des Clusters unabdingbar. Bei Ausfall gehen bereits nach kürzester Zeit Daten verloren, da der Stream unterbrochen wäre. Auch wenn davon nicht das gesamte System betroffen wäre, hätte es dennoch großen Einfluss auf die Datensammlung, den Datenfluss und die Verarbeitung, welche ebenfalls ein Echtzeitsystem mit ihren Analysemöglichkeiten ausmachen. Eine hohe Verfügbarkeit des Systems und der Daten wird durch Replikationen der Instanzen und der Datensätze erreicht. Ankommende Daten werden auf mehreren Instanzen verteilt, die dann bei einem Ausfall den betroffenen Node ersetzen.

Geringe Latenz

In Echtzeitsystemen unterscheidet man zwischen zwei Arten von Latenz: Die erste bezieht sich auf die Zeitspanne, die es benötigt, um über die Datensammlungs-Schicht ankommende Informationen entgegenzunehmen und an die entsprechenden Komponenten weiterzureichen. Die zweite Variante zielt auf die Dauer ab, die das System benötigt, um neue Daten zu verarbeiten und letztendlich dem Benutzer bereitzustellen. In Batchsystemen ist im Gegensatz zum Streaming-Bereich die Latenz um ein Vielfaches höher, da neue Daten immer erst bei Ausführung des nächsten, passenden Jobs beachtet werden können (sofern beim Ankommen der neuen Informationen bereits ein Job läuft). Im Bereich des Streamings wird diese Latenz durch Verarbeitung von kleineren Aufgaben, sogenannten Micro-Batches, minimiert.

Horizontale Skalierbarkeit

Unter horizontaler Skalierung versteht man das Hinzufügen von physikalisch getrennten Rechnern zum Cluster, um die Leistung des Gesamtsystems zu erhöhen.[16] Heutige Systeme übernehmen größtenteils den Aufwand, der für eine Skalierung betrieben werden muss, sodass der Anwender hier keine spezielle Fachkenntnis benötigt. Durch Minimierung des Kommunikationsoverheads in solchen Clustern wird nahezu eine lineare Skalierung erreicht und somit eine Performancesteigerung durch Hinzufügen von Standardrechnern ermöglicht, was gleichzeitig die Kosten gegenüber der Anschaffung von Spezialhardware reduziert. [13]

[16] Im Gegensatz dazu steht das vertikale Skalieren. Hier wird eine einzelne Serverinstanz durch Hinzufügen von stärkeren Ressourcen, leistungsfähiger gemacht.

3 Tools und Frameworks im Streaming Big Data Umfeld

Der in den bisherigen Kapiteln vorgestellte Trend der massenhaften Speicherung und Analyse von Daten bringt eine Bedarfsveränderung mit sich. Es wird Speicherplatz im Terabyte-Bereich benötigt, um Daten eines großen Systems festzuhalten. Durch Replizierung soll außerdem eine hohe Verfügbarkeit erreicht werden, was nochmals ein Vielfaches an Rechnern und Speicher benötigt. Große Firmen kommen hierbei schnell in den Bereich von dutzenden oder gar hunderten Festplatten, die über ein großes Rechnercluster zusammengeschlossen werden müssen, um anfallende Daten schnell entgegennehmen, speichern und im Cluster replizieren zu können. Egal, ob relationale Datenbank-Managementsysteme (RDBMS) oder NoSQL-Systeme zum Einsatz kommen – die Datensammlungen werden letztendlich als Dateien im Filesystem gespeichert. [20] In den oben angesprochenen Clustern wird hier ein verteiltes Dateisystem benötigt, um den Anforderungen an solche hochverfügbaren Echtzeitsystemen gerecht werden zu können. Ein solches Dateisystem stellt das *Hadoop Distributed File System (HDFS)* dar. Es übernimmt Verwaltungsaufgaben wie Replikation, Integritätssicherung und Verfügbarkeit der Daten. Somit muss sich die Datenbank nicht um diese Aufgaben kümmern und kann sich lediglich auf den Transaktionsablauf konzentrieren. Eine mögliche Datenbank stellt *Apache HBase* dar. Es handelt sich um eine BigTable-Datenbank, mit der extrem viele Datensätze gespeichert werden können. Sie persistiert Daten über Dateien in ein HDFS-Verzeichnis und alles Weitere erledigt das verteilte Dateisystem selbst.[17] Im Gegensatz zu HBase gibt es aber auch NoSQL-Speicher, die auch unstrukturierte Daten entgegennehmen und persistieren können. Eine solche Lösung stellt im Hadoop-Umfeld *Apache Cassandra* dar. Die Datenreplikation findet hier allerdings nicht über das zugrunde liegende Dateisystem statt, sondern auf DBMS-Ebene. Diese Speicherung geschieht außerdem bei den meisten NoSQL-Lösungen in nichtverteilter Form auf dem lokalen Dateisystem des Betriebssystems. Um dennoch eine im Hadoop-Umfeld benötigte Verteilung der Daten mit machbarem Aufwand umsetzen zu können, wird in Cassandra durch Anwendung von Hash-Funktionen (Consistent Hashing) bestimmt, wohin anfallende Dateien gespeichert werden bzw. zum Auslesen zu finden sind.[18]

> *Jeder Datensatz hat einen solchen eindeutigen Hash, und zu jedem Zeitpunkt sind in Cassandra genau die Rechner definiert, die diesen Satz verwalten. In HBase wird ebenfalls Consistent Hashing benutzt, aber eben nur, um die eine aktive Kopie zu finden, nicht um auch die Replikation des Records abzubilden. [20]*

[17] Nähere Informationen zu HBase unter Kapitel 3.7
[18] Details zum Consistent Hashing unter Kapitel 7.2.2

Neben der verteilten Speicherung von Daten wird in großen Systemen auch entsprechend Leistung benötigt, um Anfragen schnell bearbeiten zu können. Auch hierbei stellt Hadoop Lösungen bereit, die die im Cluster vorhandenen Ressourcen der einzelnen Rechner verwaltet und anfallende Aufgaben entsprechend verteilt. Jeder Rechner hat dabei zwar ein eigenes, lokales (meist Linux-basiertes) Betriebssystem laufen, doch der Zusammenschluss der Rechner zu einem großen verteilten System wird durch Hadoop realisiert. Nach außen hin repräsentiert sich ein solches Hadoop-Cluster wie ein einzelner großer Rechner. Der Begriff „Datacenter as a Computer“ steht dabei für diese Eigenschaft. Der Anwender gibt Aufgaben an das Cluster, ohne den inneren Aufbau genauer zu kennen. Für dieses verteilte Rechnen und Verwalten der Applikationen ist eine andere Komponente des Hadoop-Projekts zuständig: *YARN (Yet Another Resource Negotiator)*. YARN bestimmt im Cluster dabei einen Rechner als ResourceManager, der sich um die Verwaltung von anstehenden Programmen (auch Jobs genannt) kümmert. Eine Anwendung läuft dabei in einem bestimmten Ressourcen-Umfeld (RAM, CPU) ab, diese Zuteilung wird Container genannt. Für eine detailliertere Überwachung innerhalb eines Jobs gibt es jeweils auch einen AppMaster-Prozess auf einem der involvierten Rechner. Dieser startet Prozesse auf den Rechnern, die vom ResourceManager zugeteilt wurden [20]. YARN arbeitet stark mit dem verteilten Dateisystem HDFS zusammen. Dadurch weiß es, wo welche Dateien liegen und kann somit die Rechnerressourcen einem Programm zuteilen, die die benötigten Dateien bereits enthalten. Somit wird ein teures Kopieren von großen Datenmengen im Cluster vermieden. Stattdessen werden die Programme in Form von JAR-Files auf die ausgewählten Rechner verteilt und durch die Manager gestartet [20].

> *Bei korrekter Einrichtung der Hadoop-Cluster kennt Hadoop die Netzwerktopologie. Es weiß also, wie weit zwei Knoten voneinander entfernt sind. Einerseits wird das genutzt, um Daten möglichst nur über kurze Distanzen zu bewegen. Andererseits kann HDFS so feststellen, wie sich Files redundant speichern lassen, ohne durch Zufall sämtliche Replikas auf derselben Maschine oder im selben Rack abzulegen und somit bei einem Ausfall dieser Knoten keine Fallback-Replikas mehr zu haben. [20]*

Die hier vorgestellten Lösungen bilden zusammen mit der in Kapitel 2.3.4 vorgestellten Map-Reduce-Funktionalität die Kernkomponenten des Apache Hadoop Projekts, welches sich mit der Entwicklung einer Open-Source Software für zuverlässiges, skalierbares und verteiltes Computing beschäftigt. Weiter gibt es im Apache Hadoop Projekt noch einige andere Lösungen, welche sich auf speziellere Bereiche konzentrieren.

Folgende Abbildung zeigt einige im Hadoop-Umfeld angesiedelte Tools und deren Arbeitsbereich, die in den folgenden Abschnitten kurz vorgestellt werden:

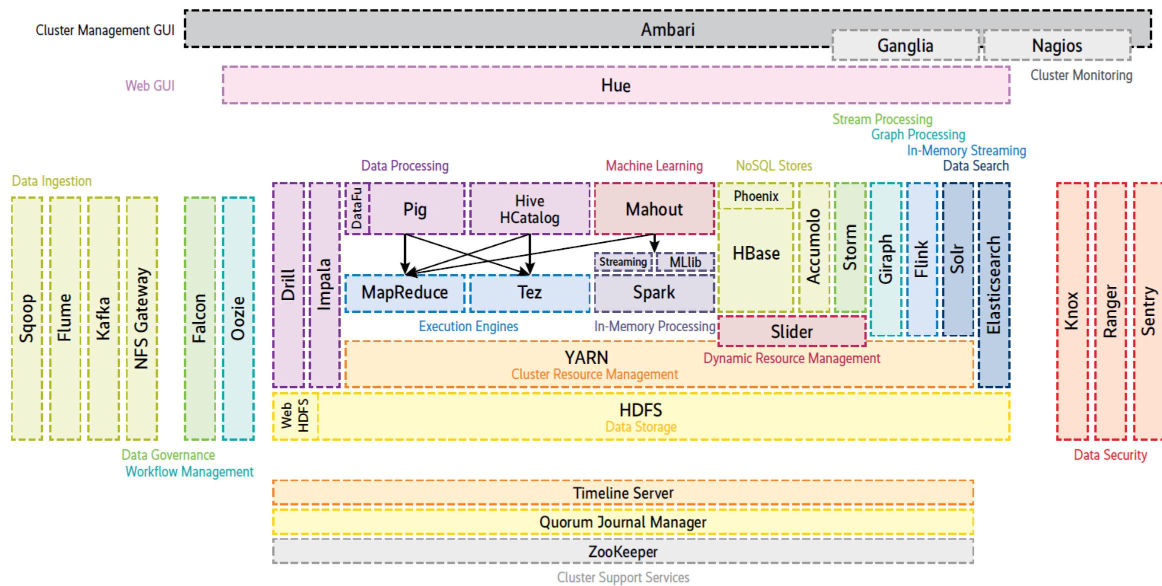

Abbildung 7: Tools im Apache Hadoop Umfeld [21]

3.1 Kernkomponenten

Das Apache Hadoop Framework besteht aus zwei Grundkomponenten: dem verteilten Dateisystem *HDFS* und *YARN*, dem Ressourcenmanager, welcher für die Verteilung der Aufgaben innerhalb eines Clusters zuständig ist. YARN wurde außerdem durch den dynamischen Ressourcenverwalter *Apache Slider* erweitert. Dieser ermöglicht es, Ressourcenzuteilungen zur Laufzeit zu ändern. Die beiden Hauptkomponenten sind im Überkapitel näher erläutert.

3.2 Systemmanagement

Durch den verteilten Koordinationsservice namens *Apache Zookeeper* wird ein verteiltes, minimales Dateisystem aufgesetzt, welches sich um den reibungslosen Betrieb innerhalb des Clusters kümmert. HDFS und YARN können mit Zookeeper interagieren, sodass über diesen Verwaltungsaufgaben wie Auswahl aktiver Knoten, Überwachung des Clusters und Fehlerreaktion übernommen werden. Ebenfalls in diese Kategorie zählen der *Quorum Journal Manager* (QJM) und der *Timeline Server* (TLS). QJM ist ein Hilfsdienst von HDFS, welcher Journaldateien der bisher durchgeführten Transaktionen auf dem Dateisystem erstellt. Bei Knotenausfall kann durch Nachverfolgung des Journals ein Ersatzknoten mit dem zum Ausfallzeitpunkt aktuellen Datenstand erstellt werden. Der TLS ist wiederum Teil von YARN. Er loggt Informationen zu bisher ausgeführten Applikationen und legt diese zentral ab, sodass hierdurch eine Suche nach Fehlern und das Debugging im Cluster vereinfacht werden [21].

3.3 Execution Engines zur Datenverarbeitung

In diese Kategorie fällt das verteilte Verarbeitungssystem *Apache MapReduce*, welches in Kapitel 2.3.4 erläutert wird. Wegen seiner Festplattenzugriffe beim Lesen und Zwischenspeichern, ist MapReduce nur für Batch-Jobs geeignet. *Apache Tez* stellt hier eine Alternative dar. Auch Tez ist wie MapReduce eine Execution Engine, arbeitet allerdings mit gerichteten azyklischen Graphen (DAG), was die Verarbeitung stark beschleunigt. Auch flexiblere Kombinationen von Aufgabenteilen und intelligentere Ausführungsgraphen sorgen für schnellere Bearbeitung gegenüber MapReduce [21]. Eine dritte Execution Engine repräsentiert *Apache Spark*. Es setzt auf In-Memory-Technologie und verbessert somit auch die MapReduce Funktionalität. Auch eine Verarbeitung über Graphen ist durch mitgelieferte Bibliotheken möglich.

3.4 Abstraktionen der Arbeitsschicht

Diese hierin enthaltenen Komponenten können als Abstraktionsschicht oberhalb der zuvor erläuterten Datenverarbeitung angesehen werden. Sie bringen jeweils eigene Syntax mit sich und sollen dem Benutzer ermöglichen, sich besser auf wesentliche Aufgaben in der Datenverarbeitung konzentrieren zu können [21]. Das erste Tool stellt hier *Apache Pig* dar. Es kann im ETL-Prozess und zum Verarbeiten von Daten eingesetzt werden. Durch einen speziellen Compiler werden Datenflüsse in MapReduce- oder Tez-Pläne übersetzt. Eine andere Komponente ist *Apache Hive*. Sie abstrahiert aktuell ebenfalls MapReduce und Tez. Mithilfe von HiveSQL, einer SQL-kompatiblen Syntax, kann der Benutzer über Formulierung gewohnter SQL-Befehle das darunterliegende HDFS ansprechen. Eine spezialisierte Form der Abstraktion bringt *Apache Mahout* mit sich. Es handelt sich hierbei um ein Framework zum maschinellen Lernen, dessen Einsatzgebiete Benutzerverhaltens-Untersuchung, Clustering und Klassifikation sind. Mahout kann dabei auf den Execution Engines MapReduce oder Spark eingesetzt werden.

3.5 Arbeitsframeworks mit eigenen Execution Engines

Es gibt auch einige Tools, die zwar die Datenverarbeitung als Aufgabe haben, jedoch keine der unter Kapitel 3.3 vorgestellten Execution Engines verwenden. Sie bringen ihre eigene Verarbeitungsschicht mit sich, verlassen sich aber bei Verwaltungs- und Speicherungsaufgaben auf YARN bzw. HDFS. Ein erstes Projekt ist hier *Apache Flink*. Es kann mit Spark verglichen werden, da es ebenfalls auf In-Memory Berechnungen setzt. Durch eine Erkennung und Optimierung von zyklischen Datenflüssen beim Erstellen der Ausführungsgraphen ist es für die Verarbeitung von Echtzeit-Datenflüssen geeignet [21]. *Apache Storm* ist ein anderes Pro-

jekt in dieser Kategorie. Es ist speziell auf die Echtzeit-Verarbeitung von Streaming Data zugeschnitten. Durch Integration mit Slider ist eine Konfiguration zur Laufzeit möglich. Auch Garantiegrade zur Ausführung „mindestens einmal" oder „genau einmal" sind möglich. Eine Spezialisierung auf Graphen-Verarbeitung stellt *Apache Giraph* dar. Es ist deshalb besonders für Untersuchungen von sozialen Beziehungen, Netzwerkkommunikation oder sonstigen Zusammenhängen geeignet. Neben diesen drei Komponenten gibt es auch zwei Volltextsuchmaschinen, mit deren Hilfe ein Hadoop-Cluster durchsucht werden kann. Vertreter hier sind *Apache Solr* und *Elasticsearch*.

3.6 Datenverarbeitungs-Frameworks mit eigenem Management

Die hierin enthaltenen Tools stützen sich zwar auf HDFS, lassen aber YARN als Ressourcenverwaltung außen vor. Ein Projekt in dieser Kategorie ist *Impala* von Cloudera. Es ist optiert für den Einsatz von SQL-on-Hadoop und bringt dafür auch seine eigene Execution Engine mit. *Apache Drill* ist ein anderes Tool im SQL-on-Hadoop Bereich, welches noch unabhängiger von Hadoop ist und ebenfalls eine eigene Execution Engine, sowie SQL-Schicht, mitbringt.

3.7 BigTable-Lösungen

Die hierin vorgestellten Lösungen sind mit HDFS verwandt und nutzen grundlegende Mechanismen wie die Replikation, implementieren jedoch andere Datenmodelle und Zugriffsmuster [21]. Sogenannte BigTable-Lösungen haben dabei die Eigenschaft, mit extrem großen Tabellen (sowohl Spalten- als auch Zeilenanzahl) umgehen zu können. Ein bekannter Vertreter dieser Sektion ist *Apache HBase*, eine NoSQL-Datenbank, die besonders auf die Verarbeitung von relativ kleinen Datensätzen spezialisiert und somit beispielsweise für Webanwendungen interessant ist. Mithilfe von *Apache Phoenix* steht HBase eine eigene SQL-Schicht zur Verfügung, mit der ein Benutzer in gewohnter Syntax Abfragen schreiben kann. *Apache Accumulo* ist eine weitere BigTable-Lösung, deren Funktionen und Eigenschaften ähnlich zu HBase sind. Den größten Unterschied macht der Bereich der Sicherheit aus, denn Accumulo besitzt eine feingranulare Authentifizierung, die eine Tiefe bis auf Zellenbasis ermöglicht.

3.8 Weitere Komponenten

Es gibt noch etliche weitere Komponenten, die im Hadoop-Umfeld zum Einsatz kommen können. Die bekanntesten werden in diesem Abschnitt kurz erwähnt und eingegliedert.

3.8.1 Datentransport

In diesem Bereich gibt es mehrere Tools, die für verschiedene Szenarien optimiert sind. *Apache Sqoop* ist hier einer der Vertreter dieser Kategorie. Sqoop ist ein Kommandozeilen-Tool und zuständig für einen optimalen Datentransfer zwischen Hadoop und strukturierten Datenspeichern, wie beispielsweise relationale Datenbanken [22]. Es stützt sich auf MapReduce zur optimalen Übersetzung von Transferaufgaben und nutzt dabei die Map-Phase zur Steuerung und Parallelisierung von Transporten aus [21]. *Apache Flume* ist ein weiteres Tool, welches allerdings nicht nur im Hadoop-Umfeld eingesetzt werden kann und entsprechend auch nicht an MapReduce gekoppelt ist. Es definiert eigene Abläufe zum Sammeln, Aggregieren und Transportieren von Daten. Da es unter anderem auch mit HDFS und HBase umgehen kann, ist dieses Tool hier ebenfalls für den Hadoop-Bereich nennenswert. *Apache Kafka* ist ein zu Flume verwandtes Tool, das ebenfalls ein verteiltes und fehlertolerantes Transportsystem darstellt. Es baut auf einem Publish-Subscribe Modell auf und findet ebenfalls außerhalb des Hadoop-Umfelds Verwendung. Besonders in Verbindung mit Apache Storm oder Spark eignet es sich sehr gut in Echtzeit-Architekturen [21].

3.8.2 Automatisierung

Die hier vorgestellten Tools können im Hadoop-Cluster eingesetzt werden, um durch Automatisierung mehr Ordnung in große Systeme zu bringen. Ein System ist dabei *Apache Oozie*, eine Workflow Engine, die Aktionen über gerichtete, azyklische Graphen (DAGs) abbildet. Eine Ausführung der Aufgaben erfolgt dabei durch einen eigenen MapReduce-Job im Cluster. In den Bereichen Daten- und Regelmanagement kommt hier *Apache Falcon* zum Einsatz. Durch dieses Tool ist es Administratoren möglich, den Lebenszyklus von Daten überwachen zu können. Falcon arbeitet mit Oozie zusammen, um seinen Aufgaben nachzukommen.

3.8.3 Sicherheit

Das Thema Sicherheit wird in [21] in vier Teilbereiche untergliedert, in welche sich auch die entsprechenden Tools einteilen lassen. Der erste Bereich ist für die Authentifizierung („Wer bin ich?“) zuständig. Hierfür ist im Hadoop-Framework *Kerberos* integriert. Für Anfragen, die per HTTP oder REST-Schnittstelle auf das System zugreifen, kommt *Apache Knox* zum Einsatz. Ein Knox-Server wird dabei vor die Hadoop-Services (WebHDFS, YARN, Hive usw.) geschaltet (siehe Abbildung 8), stellt diese per URL und Ports zur Verfügung und überwacht den Zugriffsverkehr auf diese Ressourcen [21].

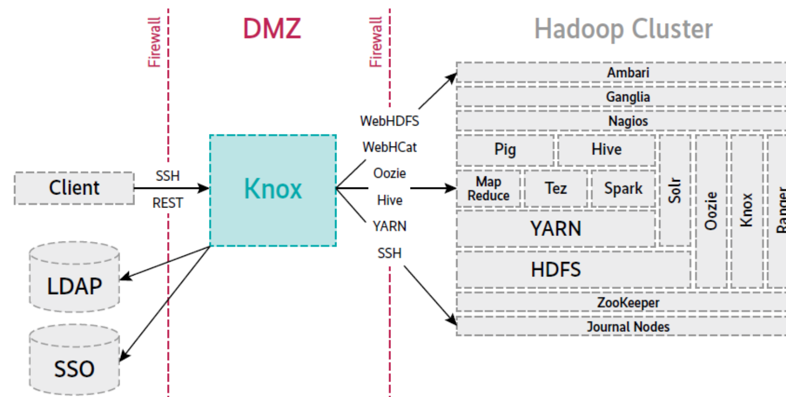

Abbildung 8: Zwischenschaltung von Knox zur Anfrageüberwachung [21]

Das zweite Gebiet ist die Autorisierung („Was darf ich machen?"). Die im Hadoop-Cluster verwendeten Tools (z. B. HDFS, YARN, Hive) haben oftmals eigene, spezielle Autorisierungsmechanismen und -Ebenen, die im Gesamten kaum zu vereinheitlichen sind. Es sind deshalb Systeme nötig, die eine Vereinheitlichung dieser Zugriffsmöglichkeiten und Vereinfachung der Administration ermöglichen. Um dieser Aufgabe nachzukommen, gibt es im Hadoop-Umfeld zwei Tools, *Apache Sentry* und *Apache Ranger*. Ranger bringt dabei noch einen Audit-Server (Protokollierungsserver) mit und überwacht damit gleichzeitig den dritten Sicherheitsbereich, das Aktions-Logging („Wer hat was gemacht?"). Die vierte und letzte Sicherheitsvorkehrung stellt die Verschlüsselung von Daten dar. Es wird hier zwischen zwei Arten unterschieden: Verschlüsselung während der Übertragung von Daten und allgemeine Datenverschlüsselung. Ersteres kann durch eine SSL-Verschlüsselung realisiert werden. Die eigentliche Datenverschlüsselung wird laut [21] bisher nur über Drittanbieter wie *Gazzang* von Cloudera ermöglicht [23].

3.8.4 Vereinfachung der Administration

Die hier genannten Tools sollen den Administratoren von Hadoop-Clustern die Verwaltung vereinfachen. Eine Lösung hierfür ist *Apache Hue* (Kurzform für „Hadoop User Experience"), eine Weboberfläche für die Hadoop-Umgebung. Hue soll besonders unerfahrenen Benutzern helfen, ohne Einsatz von Kommandozeilen-Tools, Daten bearbeiten und visualisieren zu können. Features von Hue sind unter anderem ein integrierter Dateiexplorer für HDFS, Job-Browser für YARN und unterschiedliche Abfrageeditoren für Datenbanken. Die Installation und Anpassung des Hadoop-Clusters als solches wird durch *Apache Ambari* erleichtert. Es handelt sich hierbei ebenfalls um eine Weboberfläche, die aber nach der Installation dem Administrator in Form eines Wizards als Installations- und Wartungswerkzeug dient. Da sich eine Systemüberwachung besonders auf Metriken und Alarmen stützt, kommen auch bei Ambari zwei entsprechende Tools zum Einsatz: *Ganglia* zur Sammlung von Metriken [24], als auch *Nagios* zur Alarmkonfiguration und entsprechender Benachrichtigungen [25].

4 Grundkonzepte von Apache Spark

Während MapReduce besonders in der Batchverarbeitung Verwendung findet, wurde im Laufe der Jahre der Ruf nach schnelleren und situationsbezogenen Ad-Hoc Anfragen laut. Durch Wiederverwendung nützlicher Bausteine aus MapReduce und Ausnutzung neuer Hardwarestandards sollte eine Möglichkeit zur Echtzeit-Analyse von Datenbeständen umgesetzt werden. Ein Framework, welches sich aus diesen Gedanken im Laufe der letzten Jahre entwickelt hat, ist *Apache Spark*. Es lässt, ebenfalls wie MapReduce, verteilte Berechnungen auf beliebig großen Datenmengen zu, ohne dabei Einschränkungen in Fehlertoleranz und Skalierbarkeit hinnehmen zu müssen. Hauptsächlich ermöglicht wurde die schnellere Berechnung durch Abstraktion über sogenannte *RDDs (Resilient Distributed Datasets)* und Einsatz von In-Memory Computing. Auch eine Wiederverwendung zuvor berechneter Daten bei iterativen Aufgaben, bringt, im Gegensatz zu ständigen Neuberechnungen bei MapReduce, eine deutliche Geschwindigkeitszunahme mit sich. Statt eines wiederholten Ladens von Platte werden bei Spark die Daten lediglich aus dem Arbeitsspeicher zur Weiterverarbeitung ausgelesen (sofern diese zuvor mindestens einmal berechnet, dort zwischengespeichert und nicht durch neuere Aufgaben überschrieben wurden). Diese Neuerungen bringen eine rund 100-fache Verkürzung der Rechenzeit mit sich und ermöglichen damit einen Einsatz im Realtime-Umfeld [16]. Als Beispiel zeigt [26] eine Anfrage auf eine 39 Gigabyte große Datenbasis auf, bei der bereits in einer deutlich früheren Spark-Version eine Laufzeit von unter einer Sekunde erreicht werden konnte. Besonders bei iterativen Berechnungen macht sich die Wiederverwendung von vorherigen Ergebnissen bezahlt. Abbildung 9 zeigt die Ergebnisse eines Benchmarks bei einer Regressionsaufgabe. Während Spark durch den höheren Overhead bei nur einer Iteration zwar noch langsamer ist als Hadoop, bleibt dafür beim Durchführen mehrerer Iterationen die Gesamtzeit zur Berechnung aber annähernd konstant. [26]

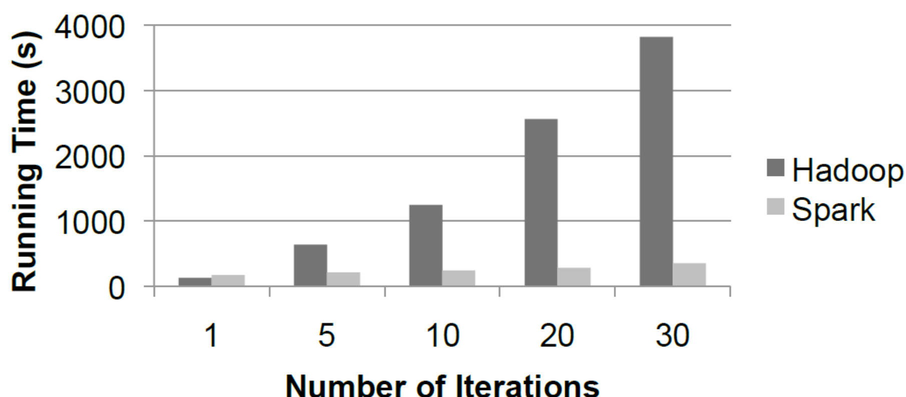

Abbildung 9: Performance-Vergleich: Hadoop & Spark bei einer Regressionsaufgabe [26]

4.1 Historische Entwicklung

Spark ist ein Open Source Projekt, welches im Laufe der Jahre durch unterschiedlichste Entwicklerkreise gewachsen ist. Entstanden ist Spark 2009 als ein Forschungsprojekt des RAD Labs der kalifornischen Universität Berkeley, welches später zum AMPLab wurde. Die beteiligten Forscher haben zuvor am Hadoop MapReduce Projekt gearbeitet und eine Ineffizienz bei iterativen Aufgaben festgestellt, was damals zum Entwicklungsbeginn des Spark-Projekts geführt hat. Bereits kurz nach seiner Einführung konnte bei verschiedenen Jobs eine 10- bis 20-fache Geschwindigkeitssteigerung erzielt werden. Entsprechend groß war das Interesse externer Firmen, die sich ebenfalls sehr früh der Spark Community anschlossen. Neben der Berkeley-Universität zählten damals bereits Databricks, Yahoo! und Intel zu den Hauptbeitragenden. Im Jahr 2011 startete AMPLab mit der Entwicklung höher-schichtiger Spark-Komponenten. Dazu zählt neben dem Spark Streaming auch das Shark-Projekt, welches eine Hive-Implementierung (Data-Warehousing Lösung) auf Spark umsetzt. Im März 2010 wurde das Spark-Projekt quelloffen gemacht und zur Apache Software Foundation übertragen, wo es im Juni 2013 zu einem Top-Level Projekt wurde. [27]

Die Spark Community wächst seit seiner Veröffentlichung stetig an. Mit Freigabe der Version 1.0 im Mai 2014 wurden über 100 Kontributoren erreicht. Version 1.1.0 folgte im September des gleichen Jahres, ebenso Version 1.2.0, die seit Dezember zum Download bereitsteht. [16] Die bei der Erstellung dieser Arbeit aktuelle Version ist 1.3.1, welche im April 2015 veröffentlicht wurde.[19]

4.2 Architektur von Spark

Das Spark-Framework besteht aus einzelnen Komponenten, die über ihr Zusammenwirken eine einheitliche und allgemeine Plattform zur verteilten Verarbeitung von Daten bereitstellen [27]. Einsatzbereich von Spark ist dabei das Entgegennehmen, Verarbeiten und Speichern von Daten aus unterschiedlichen Quellen. Anstatt mehrere Frameworks für einzelne Aufgabenbereiche installieren zu müssen, bildet Spark eine einheitliche Komplettlösung für diese. Aus welchen Komponenten sich Spark zusammensetzt, beschreibt das nächste Unterkapitel, sowie Abbildung 10. Hinweise zur Paketstruktur und Installation von Spark finden sich in Anhang A.4.

[19] Stand 30.04.2015

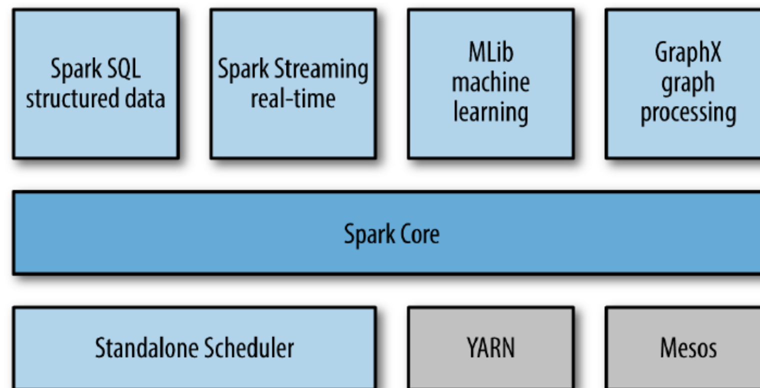

Abbildung 10: Die Spark-Komponenten [27]

4.2.1 Spark Komponenten

Zum einen kann Spark über die *Spark SQL*-Komponente mit verschiedenen Datenbanken kommunizieren. Über SQL-artige Abfragen kann der Benutzer auf Daten zugreifen. Eine Kommunikation ist hier standardmäßig mit den Datenquellen JSON, Hive und Parquet möglich. Durch die Einbindung weiterer Treiber kann auch mit anderen Datenhaltungslösungen, wie z. B. Cassandra, gearbeitet werden.[20]

Eine weitere Komponente ist die *Spark Streaming* Bibliothek. Sie ermöglicht einen Umgang mit Datenflüssen, die am System ankommen. Einsatzbereiche sind hier beispielsweise das Entgegennehmen und Überwachen von Logging-Ereignissen oder der Nachrichtenempfang aus Sensornetzwerken.[21]

MLlib ist der Name einer weiteren Bibliothek, die im Standardumfang von Spark enthalten ist. Sie kommt im Bereich des maschinellen Lernens zum Einsatz und bringt Algorithmen für die Aufgaben der Klassifizierung, der Regression und des Clusterings, sowie weitere Funktionalitäten, beispielsweise zur Evaluierung, mit sich.[22]

Da Rechenaufgaben häufig in Form von Graphen abgebildet werden, enthält Spark eine entsprechende Bibliothek zur Manipulation und Abarbeitung dieser. *GraphX* ist die Bibliothek, die bei Spark zum Einsatz kommt, um Graphen, wie sie z. B. in sozialen Netzwerken zur Abbildung von Beziehungen zwischen Benutzern zum Einsatz kommen, verarbeiten zu können.

Die Basisfunktionalitäten, um eine geregelte Abarbeitung anfallender Aufgaben überhaupt zu ermöglichen, stellt *Spark Core* zur Verfügung. Diese Komponente sorgt beispielsweise für das Scheduling von Aufgaben, die Zuteilung von Ressourcen, Fehlerbehandlung und Kommuni-

[20] Eine Vorstellung der Zusammenarbeit zwischen Spark und Cassandra findet sich in Kapitel 7.3
[21] Spark Streaming wird in Kapitel 8.1 näher behandelt.
[22] Grundlagen zum Lernen von Modellen unter Kapitel 2.1. MLlib selbst wird in Kapitel 8.2 vorgestellt.

kation mit der Datenschicht. Auch die Programmierabstraktion *RDD (Resilient Distributed Dataset)*, mit welcher Spark Daten sammelt und verarbeitet, wird hier definiert und dem Anwender durch eine entsprechende API zugänglich gemacht.

Unterhalb des Spark Cores liegen verschiedene *Cluster Manager*, mit welchen Spark arbeiten kann. Sie sorgen für einen verteilten und skalierenden Aufbau des Clusters. Neben einem *Standalone-Scheduler*, der bei einer eigenständigen Spark-Installation zum Einsatz kommt, gibt es aber auch einen *YARN*- und *Mesos-Manager*. Diese beiden ermöglichen eine Integration von Spark in bereits bestehende Hadoop- bzw. Mesos-Cluster.

4.2.2 Verwenden der Komponenten durch Einbindung von Bibliotheken

Die im vorigen Kapitel beschriebenen Komponenten müssen in Java-Projekten entsprechend eingebunden werden, um auf die Funktionalitäten der jeweiligen API zugreifen zu können. Es muss beim Festlegen der Bibliothek auf die passende Spark- und Scala-Version geachtet werden. Nachfolgend werden die Maven-Koordinaten für die zuvor beschriebenen Komponenten unter Spark-Version 1.2.2 gelistet.

```
(1)   groupId     = org.apache.spark      // entspricht der URL
(2)   artifactId  = spark-core_2.10       // Komponente & Scala-Vers.
(3)   version     = 1.2.2                 // Spark-Version
```

Codeblock 3: Maven-Koordinaten für die Spark Core Bibliothek

Entsprechend aufgebaut sind auch die Maven-Koordinaten für die weiteren Spark-Bibliotheken Streaming, SQL, MLlib und GraphX:

```
(1)   groupId     = org.apache.spark                          // URL
(2)   artifactId  = spark-<sql|streaming|mllib|graphx>_2.10   // Komponente
(3)   version     = 1.2.2                                     // Version
```

Codeblock 4: Maven-Koordinaten für die SQL bzw. Streaming Bibliothek

Die Spark Core Bibliothek muss immer eingebunden werden, sie definiert Kernfunktionalitäten wie den Spark-Context oder RDDs. Beim Einbinden der Bibliotheken reicht ein *provided*-Scope aus, da eine Spark-Installation bereits diese Funktionen enthält und diese Bibliotheken somit beim Exportieren als JAR nicht miteingebunden werden müssen.

4.3 Einteilung in Cluster

Spark bietet eine dynamische und höherschichtige API, die es ermöglicht, gleiche Programme sowohl lokal auf einem Rechner, als auch verteilt in einem Cluster ausführen zu können. Eine Organisation der Spark-Plattform als Cluster lässt dabei eine horizontale Skalierung durch Hinzufügen weiterer Rechenknoten zu. Die Architektur, die in einem solchen Cluster verwendet wird, entspricht einer Master-Slave Einteilung. Es gibt eine zentrale Koordinationseinheit und eine Anzahl im Cluster verteilter Arbeitsinstanzen. Der Koordinierungsknoten wird als *Driver* bezeichnet, die mit ihm im Cluster verbundenen und kommunizierenden Arbeitsrechner

nennt man *Executors*. Abbildung 11 zeigt diese Komponenten, aus denen sich ein Spark-Cluster zusammensetzt.

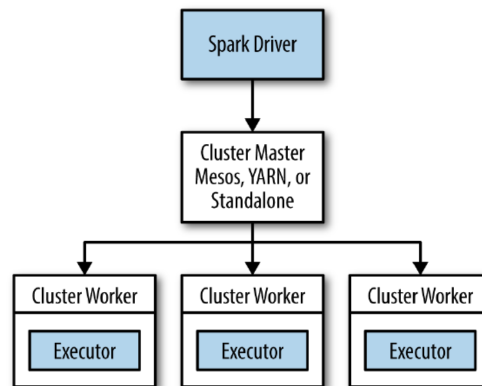

Abbildung 11: Komponenten in einem Spark-Cluster [27]

Der Driver ist dabei der Prozess, indem die *main*-Methode des Programms aufgerufen wird. Hierin wird ein *SparkContext* erstellt, über den wiederum RDDs durch Transformationen erstellt und durch Aktionen abgefragt werden können. Der Driver-Prozess kümmert sich außerdem um die Übersetzung und das Optimieren von Befehlsfolgen in *gerichtete, azyklische Graphen (DAGs)*. Auch das Scheduling von anstehenden Aufgaben und Bereitstellen einer GUI wird über den Driver erledigt. Die Executors kümmern sich um die Abarbeitung der Befehlsfolgen und reichen die Ergebnisse zum Driver zurück. Außerdem stellen sie den für RDDs benötigten Speicher über einen sogenannten *Block Manager* zur Verfügung.

Sowohl der Driver als auch die jeweiligen Executors laufen in eigenen Java-Prozessen und bilden zusammen eine *Application*. Einer Anwendung muss dabei eine gewisse Ressourcenmenge zugewiesen werden. Diese Aufgabe übernimmt ein externer Dienst, der sogenannte *Cluster Manager*. Neben dem eigenständigen Standalone- können auch andere Manager von bereits vorhandenen YARN- oder Mesos-Cluster für die Zuteilung von Ressourcen und das Ausführen von Applications verwendet werden. Die eigentliche Ausführung wird über den Cluster Manager durch Erstellen der Driver- und Worker-Prozesse angestoßen. Um eine Arbeitsaufgabe einem Cluster Manager überreichen zu können, enthält Spark das Tool *spark-submit*. Diesem können über entsprechende Parameter Ausführungsbedingungen, wie den zu verwendenden Cluster Manager oder Ressourcenzuteilungen, mitgegeben werden. [27]

Mit welchen Parametern das *spark-submit* Tool aufgerufen und damit die Ausführung von Applikationen beeinflusst werden kann, zeigt Anhang A.5. Es werden auch Details zur praktischen Abarbeitung von Aufgaben im EC2-Cluster vorgestellt.

5 Datenverarbeitung mit Spark

Spark bietet eine Vielzahl an Möglichkeiten zur Datenverarbeitung. Es können Daten aus verschiedenen Quellen gelesen und geschrieben werden. Dabei werden wichtige Dateisysteme und –formate unterstützt. Eine effizientere Jobabarbeitung kann durch In-Memory Computing und Generieren von Schlüssel-Wert Paaren erreicht werden. Ein weiterer Performancegewinn bringt eine möglichst nahe Verarbeitung der Daten auf den verantwortlichen Knoten im Cluster mit sich, beeinflussbar durch passende Partitionierung. Neben diesen Themen werden in diesem Kapitel auch verteilte Variablen vorgestellt, mit denen es dem Benutzer ermöglicht wird, Informationen global im gesamten Cluster zur Verfügung zu stellen.

5.1 RDDs - Resilient Distributed Datasets

Resilient Distributed Datasets (kurz *RDDs*, dt. elastische, verteilte Datensätze) bezeichnen die verteilte Speicherabstraktion, über welche der Anwender In-Memory Berechnungen durchführen kann. Ein RDD entspricht einer (nach Erstellung) nur-lesbaren, partitionierten Sammlung an Daten. Es kann dabei nur durch sog. *Transformationen* aus Daten eines festen Speichers oder aus anderen RDDs erzeugt werden. Besonders geeignet ist diese Abstraktion für iterative oder intensive Berechnungen, da hier meist gleiche Berechnungen mehrmals hintereinander durchgeführt werden sollen und somit bereits berechnete RDDs bei entsprechender Persistierung wiederverwendet werden können. Während bei vielen Frameworks, wie z. B. MapReduce, ein Austausch von (Zwischen-) Ergebnissen nur über zeitintensive Persistierung auf Platte erfolgen kann, ermöglichen es RDDs durch Nutzung eines gemeinsamen Speicherbereichs, Daten innerhalb eines Knotens über den Arbeitsspeicher auszutauschen. Durch grobgranulare Schnittstellen werden zwar nur relativ wenige Transformationen per API angeboten, diese bieten aber durch Überladung gleiche Operationen auf einer Vielzahl von verschiedenen RDD-Typen an. Durch Logging der Transformationsschritte (der Abstammungslinie) eines RDDs ist es recht einfach, dieses im Fehlerfall zu rekonstruieren. Dies bringt gegenüber einer herkömmlichen Persistierung aller in Verwendung befindlicher Daten eine enorme Zeit- und Speichereinsparung mit sich.

> *RDDs do not need to be materialized at all times. Instead, an RDD has enough information about how it was derived from other datasets (its lineage) to compute its partitions from data in stable storage. [28]*

Jeder Worker im Cluster, der zur Bearbeitung einer verteilten Aufgabe herangezogen wird, ist dabei für die Daten seiner Partitionen zuständig. Die Worker berechnen Zwischenergebnisse auf diesen Datenanteilen und cachen diese ggf. im Hauptspeicher. Entsprechend

schnell erfolgt der Datenzugriff, wenn pro Knoten nur auf die eigenen Daten im Dateisystem, der Datenbank oder im Hauptspeicher zugegriffen werden muss, ohne die Daten der anderen Knoten beachten zu müssen. Die Teilergebnisse werden nach den Berechnungen von den Worker-Knoten zu dem Driver-Programm zurückgesendet, wo sie anschließend zu einem Gesamtergebnis zusammengefasst werden (wie in Abbildung 12 dargestellt).

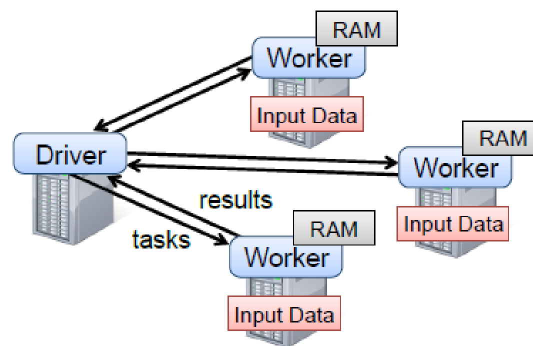

Abbildung 12: Datenfluss im Spark-Cluster bei der Ausführung eines Programms [28]

Die Methoden, die auf RDDs angewandt werden können, lassen sich in zwei Kategorien unterteilen: Transformationen und Aktionen. Bei *Transformationen* werden Daten eines RDDs nach bestimmten Kriterien selektiert bzw. bearbeitet und anschließend in einem neuen RDD zur weiteren Bearbeitung zur Verfügung gestellt. Die zu den Bearbeitungen herangezogenen RDDs bleiben dabei unverändert. Dies ist besonders nützlich, um sie später wiederverwenden zu können. Transformationen erstellen immer ein neues RDD aus einem oder mehreren Eltern-RDDs. Da Anwender über die RDDs letztendlich etwas Bestimmtes in Erfahrung bringen wollen, müssen irgendwann auf einem oder mehreren RDDs entsprechende Berechnungen, sogenannte *Aktionen*, durchgeführt werden. Diese liefern entsprechend Ergebnisse in Form von Datentypen zurück, die dann wiederum vom Benutzer interpretiert, dargestellt und/oder persistiert werden können.

Da RDDs nur Daten von Zwischenberechnungen innerhalb eines Gesamtprozesses enthalten, macht eine Ausführung der Transformationen ohne anschließende Aktionen keinen Sinn und könnten somit auch einfach ignoriert werden. Diese Denkweise entspricht der faulen (Fachbegriff „lazy“) Abarbeitung eines Prozesses und dem Vorgehen bei Spark. Schreibt der Anwender eine sog. Application, eine auf RDDs basierte Berechnung von Daten, wird ein Ausführungsplan erst dann erstellt, wenn mindestens eine Aktion die Berechnung abschließt (andernfalls wäre es auch keine wirkliche Berechnung). Als Beispiel dient der Pseudocode in Codeblock 5, der eine Filterung und Rückgabe der Zeitstempel von Fehler-Ereignissen im HDFS-Logging durchführt (angenommen, die Zeitstempel sind in der vierten Spalte zu finden, so muss ein split() auf Index 3 erfolgen).

```
(1)   lines = spark.textFile("hdfs://...")
(2)   errors = lines.filter(_.startsWith("ERROR"))
(3)   errors.persist()
(4)   errors.filter(_.contains("HDFS"))
(5)     .map(_.split('\t')(3))
(6)     .collect()
```
Codeblock 5: Filterung und Rückgabe der Timestamps von Fehlern in HDFS-Logging [28]

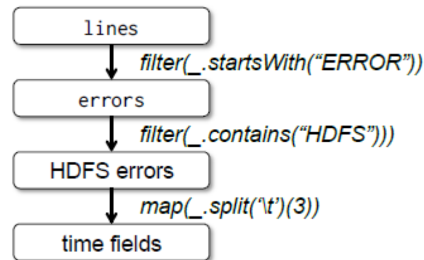

Abbildung 13: Graphenrepräsentation: Programm zur Filterung von HDFS-Logging [28]

Der Ausführungsplan wird über DAGs (gerichtete, azyklische Graphen) repräsentiert, geloggt und ggf. wiederhergestellt. Die Graphen entsprechen dabei einer Optimierung der aufeinander ausgeführten Transformationen und Aktionen. Beim Einbringen einer Aktion in das Driver-Programm wird der Gesamtprozess vom System betrachtet und automatisch optimiert, um unnötig große Zwischenberechnungen zu vermeiden. Im obigen Beispiel würde z. B. die Optimierungsroutine die beiden *filter*-Methoden zu einer zusammenfassen (ein Filter, der sowohl Einträge, die mit „error“ beginnen, als auch das Schlüsselwort „hdfs“ enthalten, selektiert), sodass hier die Datensätze nur einmal, statt doppelt, durchlaufen werden müssen. Abbildung 13 zeigt die Repräsentation des Programms in Codeblock 5 als Graph. Dieser wird zwischengespeichert und im Fehlerfall für die Neuberechnung abgearbeitet.

Abhängigkeiten zwischen einzelnen RDDs lassen sich ebenfalls in zwei Gruppen einteilen: in *narrow* (engl.: enge) und *wide* (engl.: breite, weite) *dependencies*. In den engen Beziehungen wird eine Partition des Eltern-RDDs maximal von einem Kind-RDD weiterverarbeitet, während es bei einer weiten Abhängigkeit mehrere Kind-RDDs geben kann. Abbildung 14 zeigt einige Transformationen mit den jeweiligen Zuordnungen.

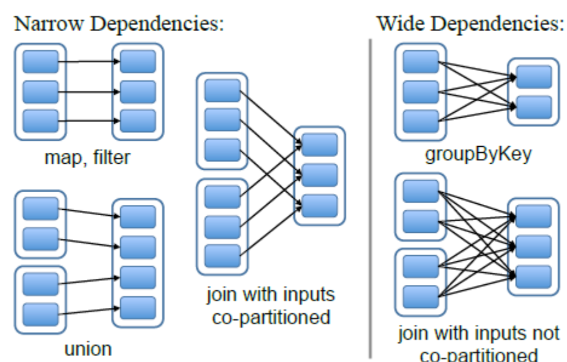

Abbildung 14: Narrow (enge) und wide (weite) Abhängigkeiten zwischen RDDs [28]

Bei engen Beziehungen ist dabei eine Wiederherstellung einfacher als bei weiten, da hier lediglich die Daten der gleichen Partition neuberechnet werden müssen. Bei weiten Beziehungen kann es aufgrund einer Nichteindeutigkeit passieren, dass ebenfalls die Eltern-RDDs neuberechnet werden müssen, um ein Kind-RDD zu erhalten.

5.2 Datenquellen und Dateiformate

Spark unterstützt eine Vielzahl von Dateiformaten und kann mit unterschiedlichen Datenquellen interagieren. Die Möglichkeit, Dateien nicht nur lokal lesen bzw. schreiben zu können ist besonders im Big Data und Streaming Data Bereich vonnöten. Da hier Datenmengen anfallen können, die den im Spark-Cluster vorhandenen Speicherplatz um ein Vielfaches übersteigen, ist eine Speicherung in entsprechend großen, ggf. entfernten und verteilten Datenspeichern unumgänglich. Welche Datenquellen Spark unterstützt, wird in den nachfolgenden Absätzen kurz erläutert.
Sollten die Instanzen des Spark-Clusters genügend großen lokalen Speicher besitzen, um als Datenquelle bzw. -senke dienen zu können, kann Spark auch von diesem lesen bzw. dorthin schreiben. Wichtig hierbei ist, dass die zu bearbeitenden Dateien bereits auf die einzelnen Arbeitsinstanzen des Clusters verteilt sind und dort überall den gleichen Pfad besitzen. Ist dies der Fall, können die Dateien durch Pfadangaben im Format *file://local/path/to/file.txt* angegeben werden.[23]
Besser geeignet für die Verarbeitung von Dateien sind die beiden Dateisysteme *S3* von Amazon und das *Hadoop Distributed File System (HDFS)*. S3 eignet sich hervorragend für die Anwendung in Spark-Clustern, die über EC2-Instanzen betrieben werden, da hier die schnellen Verbindungen innerhalb der Rechenzentren von AWS (Amazon Web Services) zur Datenübertragung genutzt werden. Per Pfadangabe im Format *s3n://bucketname/paths/in/bucket/file.txt* kann eine Datei angesprochen werden.[24] Eine Verbindung mit S3 aus einem lokalen Spark-Cluster über das Internet ist jedoch sehr langsam und nicht zu empfehlen.
Beim Einsatz von Spark im Hadoop-Umfeld bietet sich die Verwendung von HDFS als Datenquelle an. Hierbei kann über *hdfs://master:port/path/to/file.txt* ein Dateipfad angegeben werden. Bei einer Installation von Spark und HDFS im gleichen Netzwerk oder auf gleichen Rechnern kann auch hier die Performanz extrem gesteigert werden. Dieser Vorteil ist bei Spark-

[23] Alternativ kann auch eine Datei vom Master aus gelesen und anschließend per *parallelize*-Transformation auf die Worker-Instanzen verteilt werden. Diese Möglichkeit liefert allerdings eine schlechte Performanz, da die Daten entsprechend vor der Verarbeitung an die Worker geschickt werden müssen.
[24] Um Daten aus einem S3-Bucket aufrufen zu können, müssen außerdem die Umgebungsvariablen AWS_ACCESS_KEY_ID und AWS_SECRET_ACCESS_KEY mit den Benutzerangaben gesetzt werden.

Installationen auf EC2-Clustern ebenfalls standardmäßig gegeben, da ein HDFS mitinstalliert wird.[25]

Bei allen Dateisystemen können neben Angabe einer einzelnen Datei auch ganze Ordner oder Dateibündel per Wildcard angegeben werden, z. B. *file://local/path* oder *file://local/path/to/file-*.txt.*

Nachdem in den vorherigen Abschnitten die unterstützten Dateisysteme von Spark vorgestellt wurden, folgt nun eine Zusammenfassung zu den Dateiformaten, mit denen gearbeitet werden kann. Neben strukturierten Daten kann Spark auch mit semi- und unstrukturierten Daten arbeiten.

Dateiformat	Struktur	Bemerkung
Textdatei	Unstrukturiert	Herkömmliche Textdatei (eine Zeile pro Eintrag)
JSON	Semistrukturiert	Textbasierte Darstellung serialisierter Daten
CSV	Strukturiert	Komma-getrennte Werte (eine Zeile pro Eintrag)
Sequence Files	Strukturiert	Hadoop-eigenes Dateiformat zur Speicherung von Key-Value Pairs
Protocol Buffer	Strukturiert	Optimierte Datenserialisierung über festgelegte DSL
Object Files	Strukturiert	Objekt-Serialisierung mit Java-Klassen als Schablone

Zum Auslesen von Textdateien kann über den SparkContext die Methode *textFile* aufgerufen werden. Datenquellen können hier ein lokales Dateisystem, ein S3-Bucket oder HDFS sein. Sollen mehrere Dateien eingelesen werden, kann in der *textFile*-Methode ein ganzer Pfad angegeben werden. In diesem Fall werden die Inhalte aller gefundenen Dateien aneinandergehängt. Um die Herkunft der Daten verfolgen zu können, bietet sich die alternative Methode *wholeTextFiles* an: Sie liest ebenfalls einen Ordner ein, gibt aber ein Pair-RDD zurück, welches als Schlüssel den Dateinamen und als Wert den Inhalt der jeweiligen Datei besitzt. Die Speicherung von Daten in eine Textdatei kann durch Aufruf der *saveAsTextFile*-Methode durchgeführt werden. Das JSON-Dateiformat kann wie eine Textdatei eingelesen und gespeichert werden. Wichtig ist die Deserialisierung der Daten beim Lesen bzw. die Serialisierung vor dem Schreiben mithilfe eines Parsers.[26] Codeblock 35 und Codeblock 36 im Anhang zeigen jeweils ein Beispiel für das Lesen bzw. Schreiben von JSON. Ebenfalls wie Text und JSON lassen sich CSV-Dateien über die textFile Methode einlesen und speichern. Soll eine CSV-Datei mit Zeilenumbrüchen innerhalb der Wertefelder verarbeitet werden, kann dies mithilfe der wholeTextFiles Methode erledigt werden.[27]

[25] Nähere Informationen über die Verwendung von HDFS im EC2-Cluster unter Kapitel 7.1
[26] Als Parser-Bibliothek eignet sich z. B. Jackson.
[27] Da die drei weiteren Dateiformate für diese Arbeit nicht relevant sind, sei für eine Einleitung zu diesen auf [27] verwiesen.

5.3 Schlüssel-Wert Paare

Ein sehr gängiger Datentyp in Spark sind Schlüssel-Wert Paare. Sie werden durch sog. *PairRDDs* repräsentiert und für viele Operationen, wie z. B. zur Datenaggregation, eingesetzt. Durch einen initialen ETL (Extract, Transform, Load) Prozess werden Daten geladen und in ein entsprechendes Format, bestehend aus Schlüssel und Wert, gebracht. In der Spark-Programmierung können solche Key-Value Pairs durch Aufruf der *mapToPair*-Funktion erzeugt werden. Codeblock 6 generiert beispielsweise ein RDD mit Paaren, bei denen jeweils das erste Wort einer Textzeile als Schlüssel dient.

```
(1)   PairFunction<String, String, String> keyData =
            new PairFunction<String, String, String>() {
(2)     public Tuple2<String, String> call(String x) {
(3)       return new Tuple2(x.split(" ")[0], x);
(4)     }};
(5)   JavaPairRDD<String, String> pairs = lines.mapToPair(keyData);
```

Codeblock 6: Erstellung eines RDDs mit Schlüssel-Wert Paaren [27]

Um Berechnungen auf Paaren mit gleichem Schlüssel durchführen zu können, bietet Spark die Transformation *reduceByKey* an. Sie ermöglicht, die Werte von Key-Value Pairs mit gleichem Schlüssel z. B. aufsummieren zu können. Ein sehr bekanntes Beispiel ist hier das Zählen der Häufigkeiten von Wörtern in einem Text. Codeblock 7 stellt die gesamte Logik eines Word-Count Programms dar: Nachdem eine Textdatei in ein RDD mit Strings eingelesen ist, wird ein neues RDD erstellt, welches die einzelnen Wörter des Ausgangs-RDDs enthält. Alle Einträge (Wörter) dieses RDDs werden wiederum zu einem Pair-RDD mit Einträgen der Form (<<Wort>>, 1) gewandelt. Durch Anwendung einer *reduceByKey*-Transformation kann anschließend ein Zählen der einzelnen Wörter durchgeführt werden, da jedes Wort ein Schlüssel darstellt und somit nur noch die einzelnen Werte aufaddiert werden müssen, um die Vorkommnisse jedes Worts ermitteln zu können.[28] Welche Transformationen es für Key-Value Pairs noch gibt, kann dem Anhang in Tabelle 7 und Tabelle 8 entnommen werden.

```
(1)   JavaRDD<String> input = sc.textFile("s3://...")
(2)   JavaRDD<String> words = input.flatMap(new FlatMapFunction
            <String, String>() {
(3)     public Iterable<String> call(String x) {
(4)       return Arrays.asList (x.split(" "));
(5)     }});
(6)   JavaPairRDD<String, Integer> result = words.mapToPair(
            new PairFunction<String, String, Integer>() {
(7)       public Tuple2<String, Integer> call(String x) {
(8)         return new Tuple2(x, 1);
(9)       }
(10)  }).reduceByKey(new Function2<Integer, Integer, Integer>() {
(11)    public Integer call(Integer a, Integer b) { return a + b; }});
```

Codeblock 7: Wörterzählung über Pair-RDD und reduceByKey-Transformation [27]

[28] Siehe auch das Word Count Beispiel bei MapReduce, Kapitel 2.3.4

5.4 Datenpartitionierung

Ein sehr wichtiges Thema in verteilten Systemen stellt die Datenpartitionierung dar, denn sie kann den Kommunikationsaufwand und damit den Netzwerkverkehr maßgeblich beeinflussen. Durch ein intelligentes Verteilen der Daten auf das Cluster kann bei anfallenden Berechnungen der benötigte Datenaustausch zwischen Knoten minimiert und damit die Performanz einer verteilten Anwendung gesteigert werden. Besonders bei iterativen Berechnungen, wie sie im Spark-Umfeld oftmals stattfinden, kommt die Datenaufteilung zum Tragen. Die Partitionierung ermöglicht es, Key-Value Pairs mithilfe von Hash- oder Intervall-Funktionen über ihren Schlüssel einer bestimmten Partition im Cluster zuzuweisen. Da nach dem Hashing alle Paare mit gleichem Schlüssel bereits auf demselben Knoten liegen, kann anschließend eine schlüsselbezogene Berechnung ohne Datentransfers stattfinden (lediglich die Ergebnisse müssen über das Netzwerk zum Master zurückgegeben werden).

Spark bietet für die Partitionierung drei Standardmöglichkeiten an. Eine Partitionierung durch Einsatz von Hashfunktionen bietet die Klasse *HashPartitioner*. Sie benötigt lediglich Angaben zur vorgesehenen Partitionsanzahl und legt hierüber die Hashfunktion fest, mit welcher die Schlüssel einer Partition zugewiesen werden. Eine weitere, parallel arbeitende Partitionsvariante ist das Verteilen von Key-Value Pairs nach Schlüsselintervallen. Die hierbei verwendbare Klasse heißt *RangePartitioner*. Über ihn werden die Daten eines Pair-RDDs so auf das Cluster verteilt, dass die einzelnen Knoten ungefähr für gleich viele Datensätze (aber nicht unbedingt für gleich viele Schlüsselwerte!) verantwortlich sind. Ein Beispiel zeigt Tabelle 1 bzw. Diagramm 1: Obwohl Partition B nur für einen Schlüsselwert zuständig ist, enthält sie dennoch die meisten Datensätze im Vergleich zu den anderen Partitionen.

<table>
<tr><th colspan="3">Datensätze</th><th colspan="3">Partition</th></tr>
<tr><th>Schlüssel</th><th>absolut</th><th>relativ</th><th>Partition</th><th>absolut</th><th>relativ</th></tr>
<tr><td>1</td><td>10</td><td>3,4%</td><td rowspan="3">A</td><td rowspan="3">60</td><td rowspan="3">20,3%</td></tr>
<tr><td>2</td><td>20</td><td>6,8%</td></tr>
<tr><td>3</td><td>30</td><td>10,2%</td></tr>
<tr><td>4</td><td>100</td><td>33,9%</td><td>B</td><td>100</td><td>33,9%</td></tr>
<tr><td>5</td><td>50</td><td>16,9%</td><td rowspan="2">C</td><td rowspan="2">70</td><td rowspan="2">23,7%</td></tr>
<tr><td>6</td><td>20</td><td>6,8%</td></tr>
<tr><td>7</td><td>30</td><td>10,2%</td><td rowspan="4">D</td><td rowspan="4">65</td><td rowspan="4">22,0%</td></tr>
<tr><td>8</td><td>5</td><td>1,7%</td></tr>
<tr><td>9</td><td>20</td><td>6,8%</td></tr>
<tr><td>10</td><td>10</td><td>3,4%</td></tr>
<tr><td>Gesamt</td><td>295</td><td>100,0%</td><td></td><td>295</td><td>100,0%</td></tr>
</table>

Tabelle 1: Beispiel-Verteilung von Daten beim RangePartitioner

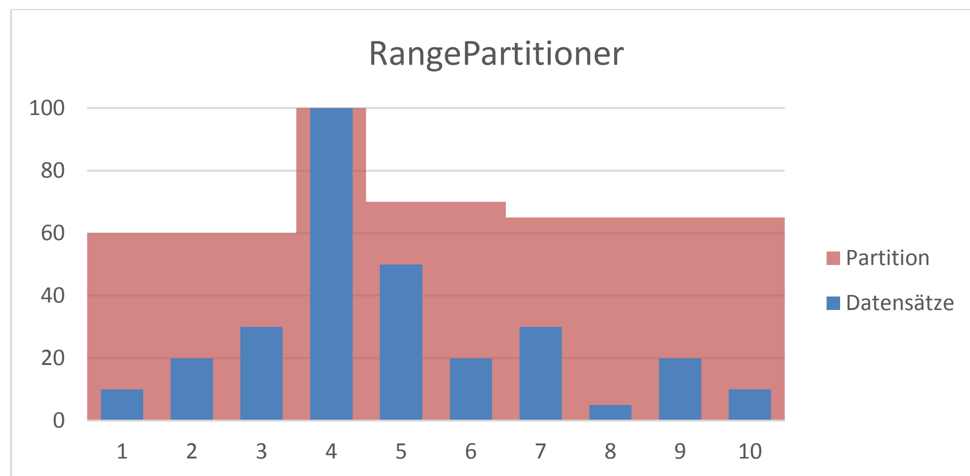

Diagramm 1: Grafische Darstellung der Datenverteilung beim RangePartitioner-Beispiel

Eine dritte Art, die Daten eines RDDs mehreren Partitionen zuzuweisen, bietet der *GridPartitioner*. Er ordnet die Partitionen in einer Tabelle an und ermöglicht Zugriff auf diese durch Angabe von Koordinaten. Durch Vererbung der abstrakten Klasse *Partitioner* ist ein Anwender auch in der Lage, eigene Partitionierungsverfahren zu implementieren. Um die Rechenleistung im Cluster bei der Partitionierung voll auszuschöpfen, sollten mindestens so viele Partitionen angelegt werden, wie es Rechenkerne im Cluster gibt, da jede Partition von einem eigenen Thread auf den jeweiligen Worker-Instanzen abgearbeitet wird.

Auch RDDs, die beim Auslesen von Dateien generiert werden, sind standardmäßig partitioniert.[29] Eine Partitionierung findet hier abhängig von Datentyp und Dateigröße statt. So wird beispielsweise direkt beim Auslesen einer HDFS-Datei, pro Block eine Partition angelegt und das enthaltende RDD entsprechend über das Cluster verteilt. Hat eine Datei im HDFS z. B. eine Gesamtgröße von 8,2 GB und eine Blockgröße von 64 MB, so wird nach deren Einlesen ein RDD erstellt, welches in 129 Partitionen unterteilt ist. Die Berechnung ist trivial:

$$Partition_{Anzahl} = \frac{8200\ MB}{64\ MB/Block} = 128{,}125\ Blöcke\ \cong 129\ Partitionen$$

Formel 1: Berechnung der Mindest-Partitionsanzahl bei HDFS

Soll (wenn für die Datei möglich) direkt beim Einlesen eine Mindestanzahl an Partitionen erstellt werden, kann der Anwender die überladene *textFile*, *hadoopFile*, *objectFile* oder *sequenceFile* Methode verwenden. Es kann hier der Parameter *minPartitions* angegeben werden, der bestimmt, aus wie vielen Partitionen das neue RDD mindestens bestehen soll.[30]

[29] Spark partitioniert das RDD, sofern die Datei, aus der es erstellt wird, groß genug für die Stückelung ist. Zudem muss die Zieldatei eine Partitionierung unterstützen. Komprimierte Dateien unterstützen beispielsweise nicht alle eine Partitionierung, z. B. der Dateityp .gz!

[30] Auch hier sind Dateityp und Datenquelle entscheidend, denn diese Partitionszahl muss mindestens so groß sein, wie die ohnehin durch Blockgröße entstehende Anzahl an Partitionen. Ansonsten hat diese Angabe keine Auswirkungen.

Ein weiteres Vorhaben im Partitionsumfeld ist die Re-Partitionierung, also das nachträgliche Aufteilen eines RDDs in verteilte Stücke. Hierfür gibt es die beiden Transformationen *repartition* und *coalesce*. Erstere Methode teilt ein bestehendes RDD in die gewünschte Anzahl Partitionen ein. Es wird hier ein Shuffle-Mechanismus verwendet, um die Daten im Cluster auf die gewünschte Partitionsanzahl zu verteilen. Die Anzahl der gewünschten Partitionen ist dabei frei wählbar, jedoch muss beachtet werden, dass eine Vergrößerung der Partitionsanzahl auch einen enormen Netzwerkverkehr zur Reorganisation des Ziel-RDDs verursachen kann. Effektiver verhält sich hier die *coalesce*-Methode. Sie darf nur zur Verringerung der Partitionsanzahl aufgerufen werden und organisiert die Neuverteilung so, dass kein bzw. nur minimaler Netzwerkverkehr entsteht. Es wird versucht, Partitionen, die sowieso bereits auf demselben Knoten liegen, zusammenzufassen und somit eine Reorganisation ohne Netzwerkverkehr durchzuführen. Hier ist es deshalb wichtig, die bisherige Anzahl an Partitionen, in die das Quell-RDD eingeteilt ist, zu kennen, um Fehler zu vermeiden. Die Anzahl der Stückelungen, in die ein RDD zerteilt ist, lässt sich über die RDD-eigene Methode *partitions().length* ermitteln. Codeblock 37 im Anhang zeigt ein Spark-Programm, welches eine CSV-Datei mit 10 Millionen DEBS-Challenge[31] Datensätzen aus einem Amazon S3-Bucket ausliest und Pair-RDDs der Form (<house_id>, <Datensatz>) erzeugt. Diese partitionierten Daten würden sich später sehr gut für Haus-basierte Analysen (z. B. Stromverbrauch pro Haus) eignen. Die Daten werden über eine *HashPartitioner*-Instanz in vier Partitionen geteilt. Nach einem Speichern im S3 sind entsprechend vier Teildatensätze vorhanden, da jeder Partitionierer eine Teil-Datei anlegt, siehe Abbildung 15.

All Buckets / dmueller5bucket / data / partitioner_output_1432191271532

Name	Storage Class	Size	Last Modified
_SUCCESS	Standard	0 bytes	Thu May 21 08:55:19 GMT+200 2015
part-00000	Standard	134.6 MB	Thu May 21 08:55:14 GMT+200 2015
part-00001	Standard	76.1 MB	Thu May 21 08:55:07 GMT+200 2015
part-00002	Standard	72.3 MB	Thu May 21 08:55:08 GMT+200 2015
part-00003	Standard	75.9 MB	Thu May 21 08:55:10 GMT+200 2015

Abbildung 15: Partitionierte CSV-Datei

Jeder Teildatensatz enthält dabei genau die Datensätze, die in den Zuständigkeitsbereich des zur Erstellung verwendeten Partitionierers fielen. Die Datensätze werden folgendermaßen auf die Partitionierer verteilt:

$$Partition_i = house_{id} \ \% \ 4$$

Formel 2: Bestimmung eines Partitionierers für einen Datensatz

[31] Informationen zur DEBS Grand Challenge unter Kapitel 9.1.1 bzw. 9.2.

Dementsprechend enthält beispielsweise Partition 0 alle Daten mit house_id = 0, 4, 8, usw., wie Abbildung 16 belegt:

```
...
127     (0,127,1377986401,0,1,9,9,0)
128     (0,128,1377986401,3.216,0,9,9,0)
129     (12,205,1377986401,11.721,0,0,0,12)
130     (12,206,1377986401,11.294,1,0,0,12)
131     (12,207,1377986401,11.721,0,1,0,12)
...
1073044 (36,2857004,1377988116,0,1,6,6,36)
1073045 (36,2857005,1377988116,0.788,0,8,6,36)
1073046 (36,2857006,1377988116,0,1,8,6,36)
1073047 (4,2857279,1377988116,0.788,0,0,0,4)
1073048 (4,2857280,1377988116,0,1,0,0,4)
...
```

Abbildung 16: Aufteilung der Daten nach dem Partitionieren

5.5 Globale Datenstrukturen

Bei Spark werden anfallende Aufgaben beim Ausführen des Driver-Programms auf die Worker-Instanzen im Cluster verteilt. Variablen werden dabei im Master angelegt und eine Kopie an die Arbeits-Knoten weitergereicht. Dort können diese Variablen zwar beispielsweise in Transformationen verändert werden, eine Rückmeldung des Wertes an den Masterknoten ist jedoch im Standardfall nicht möglich. Um dem Benutzer dennoch die Möglichkeit zu geben, Informationen von den einzelnen Arbeitsinstanzen aggregieren zu können bzw. explizit Daten an diese zu senden, gibt es zwei Arten von geteilten Variablen: Akkumulatoren und Broadcast-Variablen.

Akkumulatoren werden eingesetzt, um atomare globale Zählungen vornehmen und über den Masterknoten auslesen zu können. So kann über Auslesen des Akkumulator-Wertes beispielsweise der aktuelle Stand (Datensatz x von y) eines Jobs, über mehrere Arbeits-Instanzen hinweg, erfragt werden. Unterstützt werden hier einfache Operationen, wie das Inkrementieren, Setzen eines bestimmten Wertes oder Rückgabe des aktuellen Stands. Ein Akkumulator gilt kontextweit und wird entsprechend durch Aufruf von *SparkContext.accumulator* generiert und initialisiert.[32] Ein weiteres Beispiel für den Einsatz wäre eine Zählung der bisher über einen Stream empfangenen Datensätze. In einer entsprechenden Map-Transformation könnten die Knoten dabei den Wert um die Anzahl der im jeweiligen Micro-Batch verarbeiteten Elemente erhöhen.

Broadcast Variablen werden dagegen eingesetzt um den Worker-Instanzen einmalig nur-lesbare Werte zu übermitteln, die diese dann bei Berechnungen verwenden können. Handelt es sich dabei um große Datensätze, die per Broadcast Variable übergeben werden (z. B. Lookup-

[32] An dieser Stelle sei sowohl auf [27], als auch auf die Java-Dokumentation der Accumulator-Klasse verwiesen: https://spark.apache.org/docs/1.2.2/api/java/org/apache/spark/Accumulator.html

Tabellen), kann dies den Kommunikationsoverhead stark reduzieren. Standardmäßig schickt Spark bei jeder Operationsausführung erneut Daten vom Master- zu den Arbeitsknoten, was besonders bei kleinen Berechnungen, die aber in großen Datenbeständen Einträge nachschlagen müssen, zu einem großen Performanceverlust führen kann. Da Broadcast Variablen nur einmal an die Arbeits-Instanzen geschickt werden, minimiert sich dieser Overhead bei iterativen Aufgaben (z. B. Nachschlagen des Landes anhand einer empfangenen Telefonnummer) entsprechend.[33]

[33] Beispiele und weitere Erläuterungen zu Broadcast Variablen in [27]

6 Erstellung von Spark Applikationen

Dieses Kapitel beschäftigt sich mit der praktischen Erstellung von Spark-Applikationen. Dabei wird auf die beteiligten Komponenten des Spark Cores eingegangen und die Vorgehensweise zum Bereitstellen, sowie der Ablauf von Applikationen aufgezeigt. Auch Einstellungsmöglichkeiten und die Weboberfläche von Spark werden kurz erläutert.

6.1 Zugriff über SparkContext

Wie bereits in Kapitel 4.3 erwähnt, besteht eine Spark-Applikation aus einem Driver und meist mehreren Executors. Der Driver greift dabei über ein *SparkContext*-Objekt auf die Cluster-Umgebung zu. Nach Instanziierung des Kontexts mithilfe eines *SparkConfig*-Objekts können z. B. Daten eingelesen, RDDs durch Transformationen erzeugt und Ergebnisse durch Aktionen berechnet werden. Das SparkConfig-Objekt besteht aus einer Sammlung von Key-Value Paaren, über welche sich der Programmablauf konfigurieren lässt.
Codeblock 8 zeigt die Konfiguration und Erstellung eines SparkContexts innerhalb der *main*-Methode im Driver, über den anschließend eine CSV-Datei in ein RDD eingelesen wird. Über den SparkContext hat ein Anwender Zugriff auf alle wesentlichen Spark-Komponenten. Nach Abschluss der Transformations- und Aktionsfolgen muss der SparkContext über dessen *stop*-Methode beendet werden.

```
(1)   public class FirstRddExample  {
(2)     public static void main(String[] args) {
(3)       SparkConf conf = new SparkConf();
(4)       conf.set("spark.app.name", "myTestApp");
(5)       JavaSparkContext sc = new JavaSparkContext(conf);
(6)       JavaRDD<String> rdd = sc.textFile("hdfs://master:9010/data.csv");
(7)       rdd.count();
(8)       // ... Do some more transformations & actions with RDDs
(9)       sc.stop();
(10)    }
(11)  }
```

Codeblock 8: Erstellung eines SparkContexts und Auslesen einer CSV-Datei in ein RDD

6.2 Applikationen bereitstellen

Ist ein Driver-Programm fertiggestellt, folgt als nächster Schritt die Ausführung im Spark-Cluster. Dazu sollte das Java-Projekt, welches den Programmcode enthält, mit allen Abhängigkeiten (sofern welche vorhanden sind) in ein JAR exportiert werden.[34] Nachdem dies geschehen ist, muss die erstellte Datei auf den Masterknoten des Clusters kopiert werden. Im EC2-Umfeld kann dies z. B. durch Verwendung des *scp (secure copy)* Befehls und Angabe des privaten Schlüssels erledigt werden:

```
(1)   scp -i ~/keypair.pem ~/thesis/java/target/thesis-0.0.1-SNAPSHOT-jar-
   with-dependencies.jar root@masterURL:~/thesis.jar
```

Codeblock 9: Kopieren einer JAR-Datei auf den Masterknoten eines Spark EC2-Clusters

Nach dem Kopieren ist nun das Programm bereit für die Ausführung. Um im Cluster selbst eine Spark-Applikation starten zu können, muss sich der Benutzer am Masterknoten anmelden. Dies kann über eine SSH-Verbindung, ebenfalls unter Angabe des privaten Schlüssels, durchgeführt werden:

```
(1)   ssh -i ~/keypair.pem root@masterURL
```

Codeblock 10: Anmelden am Masterknoten über SSH

Nun kann die Ausführung einer Applikation im Cluster selbst durch den *spark-submit* Befehl angestoßen werden. Das submit-Skript findet sich im */bin*-Ordner der Spark-Installation. Unter Angabe der auszuführenden Java-Klasse (die die *main*-Methode enthält), des Masterknotens mit Port, des Ausführungsmodus (im Cluster oder lokal) und des JARs mit evtl. benötigten Parametern der *main*-Methode, kann die Applikation gestartet werden.

```
(1)   ~/spark/bin/spark-submit
(2)     --class package.FirstRddExample
(3)     --master spark://masternodeURL:7077
(4)     --deploy-mode cluster
(5)     /root/thesis.jar <optional java main parameters>
```

Codeblock 11: Starten einer Spark-Applikation über das spark-submit Skript

Welche wichtigen Parameter es noch für das *spark-submit* Skript gibt, listet Tabelle 11 im Anhang. Da eine Ausführung nicht nur im Cluster und mit Standalone-Scheduler, sondern auch lokal oder mit anderen Cluster-Managern stattfinden kann, zeigt Tabelle 12 (ebenfalls im Anhang) alle möglichen Werte des *--master*-Flags auf.

[34] Zur einfacheren Verwaltung von Bibliothek-Abhängigkeiten bietet sich hier das Erstellen der Spark-Applikation als Maven-Projekt an. Dieses unterstützt die Generierung eines sog. „Uber-JARs", welches alle Abhängigkeiten, die benötigt werden, enthält. Die Spark-Core Bibliothek muss dabei nicht mit eingebunden werden, da diese auf dem Cluster bereits zur Verfügung steht, siehe auch „Maven provided scope" und Kapitel 4.2.

6.3 Ablauf beim Ausführen einer Applikation über spark-submit

Folgende Schritte werden beim Einreichen einer Applikation durch das spark-submit Skript durchlaufen:

> 1. *Der Anwender reicht eine Applikation über spark-submit ein.*
> 2. *Spark-submit startet das Driver-Programm und ruft die main-Methode […] auf.*
> 3. *Das Driver-Programm kontaktiert den Cluster-Manager und frägt um Ressourcen für das Starten der Executors an.*
> 4. *Der Cluster-Manager startet die Executors nach Vorgabe des Drivers.*
> 5. *Der Driver-Prozess durchläuft das Anwenderprogramm. Basierend auf den RDD Aktionen und Transformationen schickt der Driver Arbeit in Form von Arbeitsaufgaben zu den Executors.*
> 6. *Arbeitsaufgaben werden von den Executors berechnet und deren Ergebnisse gespeichert.*
> 7. *Wenn die main-Methode des Drivers endet oder SparkContext.stop() aufgerufen wird, werden die Executors beendet und die Ressourcen beim Cluster-Manager freigegeben.*
>
> *- Übersetzt aus [27]*

6.4 Begrifflichkeiten bei der Ausführung von Applikationen

Dieses Kapitel beschäftigt sich mit den Begrifflichkeiten rund um die physikalische Ausführung von Applikationen im Spark-Umfeld. Wichtig für diese Thematik ist dabei das Wissen um die Repräsentation und Ausführung von Applikationsschritten durch die gerichteten azyklischen Graphen (DAGs), in welche die Anweisungsfolgen einer Applikation übersetzt werden. Wie bereits in vorherigen Kapiteln erläutert, werden bei Transformationen neue RDDs aus einem oder mehreren Eltern-RDDs erstellt. Diese Zusammensetzung wird über DAGs festgehalten, welche wiederum zum Ausführen von Aktionen herangezogen werden. Um die Entstehung eines RDDs über Transformationsfolgen als Anwender nachvollziehen zu können, gibt es die Methode *RDD.toDebugString*, die in der Applikation an gewünschten Stellen z. B. für Debugging-Zwecke aufgerufen werden kann.

Wird eine Aktion auf einem RDD ausgeführt, übersetzt der Scheduler den DAG in einen physikalischen Ausführungsplan und arbeitet diesen vom Ziel-RDD aus nach hinten ab. Dabei werden die Eltern-RDDs mithilfe der DAG-Referenzen bestimmt und anschließend die benötigten Transformationen ausgeführt. Im einfachsten Fall generiert der Scheduler hierbei eine *Stage* (Phase) pro RDD. Dies ist genau dann der Fall, wenn bei jeder RDD-Generierung Daten zwischen den einzelnen Executor-Threads ausgetauscht werden müssen, da die Eltern-RDDs auf verschiedene Partitionen verteilt sind. Jede Stage hat dabei einzelne *Tasks* (Aufgaben) für jede Partition dieses RDDs. Sollte die Berechnung eines oder mehrerer RDDs ohne *Shuffling* (Datenaustausch), also innerhalb eines Executor-Threads, möglich sein, werden weniger

Stages generiert und der Scheduler führt die Transformationen durch *Pipelining* (Verknüpfung von Befehlen) aus. Eine Einteilung in Stages bei Pipelining zeigt Abbildung 17. Ebenfalls vereinfacht wird die Ausführung bei bereits persistierten RDDs, hier beginnt die Berechnung nach der Speichertransformation. Die Stages, die für die Berechnung einer Aktion ausgeführt werden müssen, werden zu einem *Job* zusammengefasst. Eine komplette Applikation besteht damit aus verschiedenen Jobs, die abgearbeitet und deren Ergebnisse zurückgegeben werden.

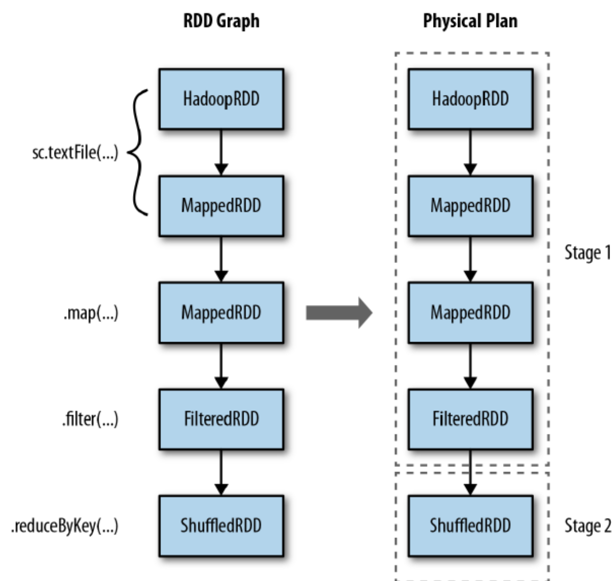

Abbildung 17: Übersetzung DAG in physikalischen Ausführungsplan mit Stage-Pipelining

6.5 Weboberfläche

Informationen über die oben vorgestellten Ausführungskomponenten lassen sich durch Aufruf der Web-Oberfläche einsehen. Die URL dafür lautet im Standardfall *http://master_dns:8080*. Hier werden auf der Einstiegsseite neben Informationen zum Masterknoten und Cluster auch aktive bzw. abgeschlossene Applikationen gelistet. Durch Auswählen der Listeneinträge können weitere Details zu Jobs, Stages und Tasks aufgerufen werden. Neben Ausführungszeiten werden auch Details zur Anzahl an ausgeführten Schritten, Lese-/Schreibvorgänge und Shuffling dargestellt. Standardmäßig ist das Festschreiben von Logging-Informationen deaktiviert und somit das Einsehen dieser Informationen nach Applikationsbeendigung nicht mehr möglich. Dieses Logging kann für eine Applikation durch Setzen des entsprechenden Flags über die verwendete SparkConf-Instanz aktiviert werden:

```
(1)   SparkConf conf = new SparkConf();
(2)   conf.set("spark.eventLog.enabled", "true");
(3)   JavaSparkContext sc = new JavaSparkContext(conf);
```

Codeblock 12: Aktivieren des Event-Loggings für eine Applikation

6.6 Einstellungen

Einstellungen, wie z. B. Ressourcenvorgaben, die das Ausführen von Applikationen beeinflussen, können über drei verschiedene Wege getroffen werden:

- Bearbeiten der */conf/spark-defaults.conf* Datei durch Ändern oder Hinzufügen von Schlüssel-Wert Paaren (durch Leerzeichen getrennt, ein Paar je Zeile)
- Übergeben von Einstellungen bei Aufruf des *spark-submit* Skripts über Angabe des *--conf* Parameters[35]
- Setzen von Einstellungen über *SparkConf.set(„property", „value")* Methode[36]

Die Liste ist nach Prioritäten geordnet (erste Variante hat geringste Priorität). Das Bearbeiten der *spark-defaults* Datei gibt Standardwerte vor, die bei Applikationsausführung gesetzt werden, sofern sie nicht durch eine der beiden anderen Methoden überschrieben wird. Von den beiden Laufzeit-Einstellungen hat dabei die Konfiguration innerhalb des Applikationscodes Vorrang und überschreibt neben der Standardkonfiguration auch die Einstellungen, die über den *spark-submit* Aufruf mitgegeben wurden. Die wichtigsten Einstellungen zur Ausführung und Ressourcenzuweisung werden im folgenden Kapitel aufgezeigt.

6.7 Ressourcenverteilung

Standardmäßig festgelegt sind beim Ausführen einer Applikation unter anderem folgende wichtige Einstellungen: Die Anzahl maximal für eine Anwendung verwendbare Rechenkerne (*spark.core.max*) ist unbeschränkt, das bedeutet eine Applikation darf alle im Cluster verfügbaren Kerne für Berechnungen verwenden, sofern nicht durch Angabe einer Höchstzahl beschränkt[37]. Der Applikationsname (*spark.app.name*) wird anhand der Paketstruktur und des Klassennamens des ausgeführten Java-Programms ermittelt, sofern dieser nicht gesetzt wird. Der für den Driver (*spark.driver.memory*) bzw. Executor (*spark.executor.memory*) zur Verfügung stehende Hauptspeicher ist jeweils mit einem Standardwert von 512m (MB) vorbelegt. Entsprechend können aber durch Angaben wie 1024m bzw. 1g diese Einstellungen überschrieben werden. Standardmäßig werden Arbeitsaufgaben gleichmäßig möglichst auf alle Arbeitsinstanzen verteilt. Sollen Arbeitsaufgaben erst auf weitere Instanzen verteilt werden, wenn alle Cores einer anderen mit Arbeitsaufgaben versorgt sind, kann das Flag

[35] Standardeinstellungen, wie z. B. Applikationsname, besitzen auch eigene Parameter, die verwendet werden können, z. B. *--name „ApplicationName"* statt *--conf spark.app.name="ApplicationName"*

[36] Standardeinstellungen, wie z. B. Applikationsname, besitzen zusätzlich eigene SparkConf-Methoden, mit denen der Wert gesetzt werden kann, z. B. *conf.setAppName(„ApplicationName")* statt *conf.set(„spark-app.name", „ApplicationName")*

[37] Ein Multi-Applikations- bzw. Mehrbenutzer-Betrieb ist damit bei dieser Standardeinstellung nicht möglich.

spark.deploy.spreadOut auf *false* gesetzt werden. Tabelle 2 zeigt den Unterschied der Einstellung beim Verteilen von 6 Partitionen auf 5 Instanzen mit je 4 Cores:

spark.deploy.spreadOut	Instanz 1	Instanz 2	Instanz 3	Instanz 4	Instanz 5
true (Standard)	2	1	1	1	1
false	4	2	0	0	0

Tabelle 2: Auswirkung des Flags spark.deploy.spreadOut auf die Arbeitsverteilung

Eine weitere Optimierung, die bei der Bearbeitung von Arbeitsaufgaben eine Rolle spielt, ist eine Partitionierung abhängig der Rechenkern-Anzahl. Durch eine an die im Cluster verfügbare Rechenkern-Anzahl angepasste Partitionsgröße kann die Abarbeitungszeit einer Stage extrem verkürzt werden. Abbildung 18 zeigt ein solches Beispiel auf. Die linke Hälfte zeigt dabei die nicht-optimierte Variante: Hier wird ein RDD, bestehend aus neun Partitionen, in einem Cluster mit vier Cores verarbeitet. Drei von vier Executor-Threads arbeiten damit je zwei Partitionen nacheinander ab, während ein Thread drei verarbeiten muss. Die Laufzeit der Aufgabe entspricht dabei dann der Laufzeit dieses Threads, da er in diesem Fall rund 50% länger zur Abarbeitung seiner Aufgaben braucht, als die anderen Threads. Die rechte Hälfte zeigt die optimierte Vorgehensweise: Hier wurde das RDD in nur acht Partitionen unterteilt, die von den vier Rechenkernen abgearbeitet werden. In diesem Szenario muss zwar jeder Thread größere Partitionen verarbeiten (9/8 der Größe im anderen Fall), jedoch können die Partitionen restlos auf die vier Executor-Threads verteilt werden und jeder arbeitet genau zwei Partitionen nacheinander ab. Insgesamt sinkt damit die parallele Bearbeitungszeit von 3 auf 2,25 Zeiteinheiten[38]. Eine an die Coreanzahl angepasste Partitionierung kann somit die Laufzeit extrem verbessern. Auch eine Stücklung in möglichst gleichgroße Partitionen kann die Abarbeitungszeit positiv beeinflussen.

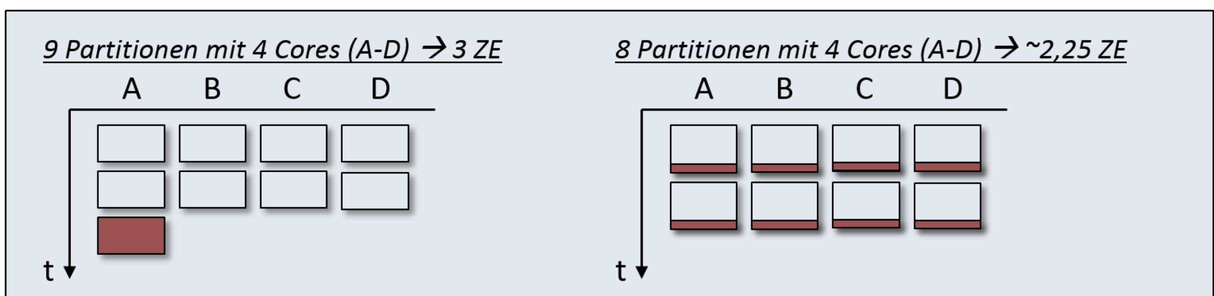

Abbildung 18: Optimierung der Laufzeit durch angepasste Partitionierung

Welche Einstellungen außerdem gemacht werden können und mit welchem Standardwert diese belegt sind, lässt sich unter https://spark.apache.org/docs/1.2.2/configuration.html nachlesen.[39]

---

[38] vorausgesetzt das Abarbeiten einer Partition der Größe im nicht-optimierten Fall benötigt 1 Zeiteinheit

[39] Vorsicht ist beim Auslesen von Einstellungen geboten: Der Anwender kann per Aufruf von SparkConf.get(„property“) die Konfiguration von einzelnen Eigenschaften auslesen. Sollte deren Standardwert aber durch keine der

7 Datenspeicherung und Benchmarks mit Spark

Dieses Kapitel beschäftigt sich mit zwei bekannten Speichermöglichkeiten, mit denen Spark interagieren kann: HDFS und Apache Cassandra. Neben einer Vorstellung der Speicher finden sich hierin auch Ergebnisse zu Schreib- und Lesebenchmarks, die im Laufe der Arbeit zur Performance-Messung durchgeführt wurden.

7.1 Spark & HDFS

Die Standalone-Installationsvariante für Spark-Cluster auf EC2-Instanzen bringt standardmäßig eine HDFS-Implementierung mit sich. Über das mitinstallierte *hadoop*-Tool können Dateien somit, neben dem regulären Dateisystem, auch auf das verteilte Hadoop File System geschrieben bzw. von dort gelesen werden.[40] In diesem Kapitel geht es darum, das HDFS auf einem EC2-Cluster mit Spark bereitzustellen, um einen späteren Zugriff aus Applikationen heraus zu ermöglichen.

Da die HDFS-Funktionalitäten bereits beim Aufsetzen des Spark-Clusters mitinstalliert werden, muss sich der Anwender nach einem Cluster-Start nur noch um das Starten der DFS- und MapReduce-Tools, die zum Interagieren mit dem HDFS benötigt werden, kümmern. Die zum Starten der Tools verantwortlichen Skripte heißen entsprechend *start-dfs.sh* und *start-mapred.sh*. Gestoppt werden können HDFS-Dienste mit dem *stop-all.sh* Skript. Die Steuerungsskripte liegen im *bin*-Verzeichnis der einzelnen Spark-Instanzen, sollten aber nur von der Master-Instanz aus verwendet werden. Nach jedem Neustart des Clusters müssen zudem die DFS- und MapReduce-Startskripte erneut aufgerufen werden, um die benötigten Dienste zur Verfügung zu stellen.
Werden die Tools gestartet, wird im Spark-Cluster das HDFS bereitgestellt. Folgende Zuweisungen werden dabei standardmäßig vorgenommen:

- Spark-Master → HDFS-Master (NameNode, JobTracker - Organisation)
- Spark-Slaves → HDFS-DataNodes (Speicherung von Dateien)

Ist eine andere Einrichtung des Clusters geplant, muss dies manuell durch Abänderung der zuständigen Konfigurationsdateien *hadoop-env.sh*, core-site.xml, *hdfs-site.xml* und *mapred-site.xml* erledigt werden. Sie sind im *conf*-Verzeichnis der HDFS-Installation zu finden. Nützliche Links und Vorgehen im Fehlerfall sind in Anhang A.6 zu finden.

---

oben genannten Methoden zuvor überschrieben worden sein, kommt es zu einer NoSuchElementException! Eine sichere Methode stellt dabei der Aufruf von allen gesetzten Einstellungen über die SparkConf.getAll() Methode dar. Sie liefert ein Array nur mit den aktuell gesetzten Einstellungen zurück.

[40] Die Verwendung von HDFS als Datenquelle wird in Kapitel 5.2 aufgezeigt.

Jedes Spark-Cluster, das auf EC2-Instanzen aufsetzt, beinhaltet zwei verschiedene Installationen von HDFS:

- */root/persistent-hdfs/*
- */root/ephemeral-hdfs/*

Wie die Benennung vermuten lässt, gibt es eine Variante, bei der die HDFS-Partition Cluster-Neustarts übersteht (persistent), und eine andere, die beim Löschen des flüchtigen Speichers ebenfalls geleert (ephemeral) wird. Die HDFS-Partitionen liegen standardmäßig in dem Verzeichnis */vol/persistent-hdfs* bzw. */vol/ephemeral-hdfs*. Wird ein HDFS-Cluster korrekt ausgeführt, ist dessen GUI über folgende Links erreichbar: *http://<spark_master_url>:50070/* (ephemeral) bzw. *http://<spark_master_url>:60070/* (persistent).

7.2 Apache Cassandra

Apache Cassandra ist ein verteiltes DBMS (Database Management System), welches besonders für das Speichern von großen und strukturierten NoSQL-Datenbanken geeignet ist. Daten werden dabei als Schlüssel-Wert Relationen abgelegt. Ein Cluster kann, wie bei Spark, aus Standardrechnern zusammengesetzt werden, eine Leistungs- bzw. Kapazitätssteigerung wird dabei durch horizontale Skalierung ermöglicht. Die Knoten des Clusters sind im Ring angeordnet und als Peer-to-Peer Netz eingerichtet. Im Gegensatz zu einer Master-Slave Architektur kann hier jeder Knoten als Einstiegspunkt für Schreib- bzw. Leseoperationen dienen. Durch diese Netzanordnung und Replikation der Daten wird ein Single Point of Failure vermieden und eine hohe Verfügbarkeit ermöglicht. Abbildung 19 zeigt den groben Aufbau eines Cassandra-Clusters.

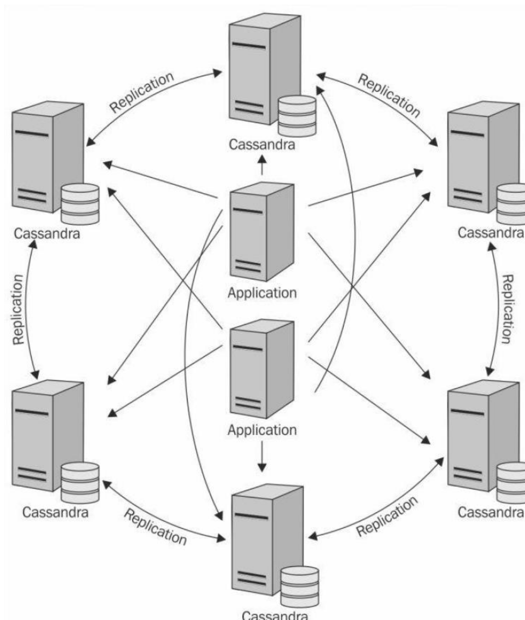

Abbildung 19: Aufbau eines Cassandra-Clusters [29]

7.2.1 Hohe Verfügbarkeit und Datenreplikation

Wie bereits in Kapitel 2.4.4 erläutert, wird in relationalen Datenbanken eine strenge Einhaltung der *ACID* (Atomicity, Consistency, Isolation, Durability)-Eigenschaften angestrebt, was eine Architektur mit Masterknoten, Sperren und komplexen Speicherstrategien voraussetzt. Unter diesem Aufbau leidet die Verfügbarkeit solcher Systeme. Bei Cassandra wird dieser Nachteil durch den Aufbau als Peer-to-Peer Netzwerk und Lockerung des ACID-Paradigmas minimiert. Eigenschaften wie Atomizität, Konsistenz, Isolation und Beständigkeit werden deshalb bei Cassandra als kontinuierliche Größen verstanden, die je nach Transaktion mehr oder weniger eingehalten werden. Datensätze werden dabei mittelfristig konsistent, sofern keine weiteren Schreibvorgänge folgen und Ausfälle ausbleiben, was dem Prinzip der *schlussendlichen Konsistenz (eng. eventually consistency)* entspricht. Der Anwender kann dabei überwiegend selbst entscheiden, auf welche Eigenschaft er besonders Wert legt und damit das Verhalten des Systems beeinflussen.

Um bei Knotenausfällen eine Verfügbarkeit des Netzes und der Daten zu gewährleisten, ist eine entsprechende Replikation der Daten, über mehrere Instanzen hinweg, notwendig. Cassandra bietet hierbei folgende Strategien an:

- *SimpleStrategy*: Einfache Datenreplikation innerhalb eines Datencenters
- *NetworkTopologyStrategy*: Datenreplikation über mehrere Datencenter hinweg

Über den Replikationsfaktor wird festgelegt, wie viele Kopien es von einem Datensatz im Netzwerk gibt. Die Strategie und der Replikationsfaktor werden beim Anlegen eines Keyspace angegeben und beeinflussen ebenfalls die Erreichbarkeit eines Clusters:

```
(1)   CREATE KEYSPACE AddressBook
(2)     WITH REPLICATION = {
(3)       'class' : 'SimpleStrategy',
(4)       'replication_factor' : 3
(5)     };
```

Codeblock 13: Anlegen eines Keyspace in Cassandra [29]

Cassandra unterstützt sowohl eine strenge, als auch eine schlussendliche Konsistenz an. Letztere Eigenschaft tritt bei Verfolgung des BASE-Prinzips, wie es in Kapitel 2.4.4 vorgestellt wurde, zum Vorschein. Letztendlich führt Cassandra ein Datenabgleich mithilfe eines Zeitstempel-basierten Verfahrens durch, sodass alle Datenreplikationen schlussendlich den aktuellsten Stand besitzen. Über verschiedene Konsistenzlevel lässt sich transaktionsweit die Abgleichstrategie konfigurieren und das Anfrageverhalten von Cassandra entsprechend beeinflussen. [29]

7.2.2 Datenverteilung, DHTs und vnodes

Dadurch, dass bei Cassandra ein Peer-to-Peer Netzwerk und Datenreplikation zum Einsatz kommen, kann jeder Knoten als Einstiegspunkt dienen und ein Single Point of Failure wird (wie einleitend erwähnt) vermieden. Während in herkömmlichen Netzen versucht wird, eine höhere Verfügbarkeit durch Aufteilung des Clusters (Sharding) oder Neuernennung von Masterinstanzen bei Ausfall (Master Failover) zu erreichen, setzt Cassandra auf den Einsatz von verteilten Hash-Tabellen. Mittels Consistent Hashing Algorithmen wird das Wissen über Datenzuständigkeiten im Netzwerk auch ohne Masterknoten zur Verfügung gestellt und gleichzeitig eine Neuberechnung der Zuständigkeiten bei hinzukommenden bzw. wegfallenden Knoten verhindert. Wo neu ankommende Datensätze gespeichert bzw. diese als Ziel von Leseanfragen zu finden sind, entscheidet der eingesetzte Partitionierer.

Durch die weitere Einteilung von Knoten in sogenannte *vnodes* ist eine bessere Daten- und Lastenverteilung möglich. Ein vnode entspricht einem Teil der Daten eines Knotens. Das bedeutet, dass ein Knoten für mehrere vnodes zuständig ist:

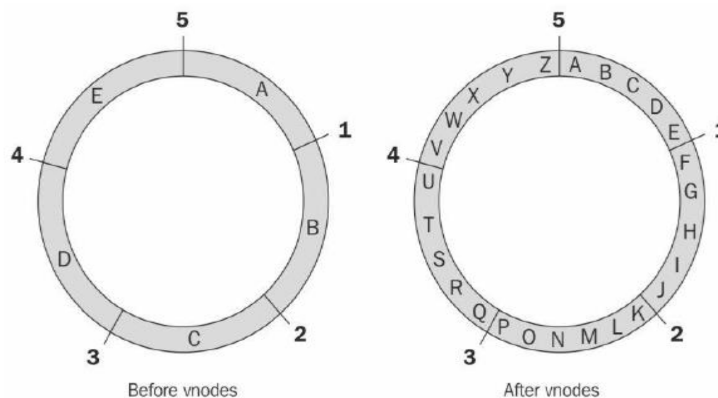

Abbildung 20: Cassandra-Ring ohne (links) und mit (rechts) vnodes [29]

Durch vnodes werden die Datenreplikationen eines Knotens breiter gefächert im Ring verteilt. Dies hat besonders für die Wiederherstellung der Daten eines Knotens enorme Vorteile, da bei dieser Aufteilungsart mehr Knoten an dem Reparaturprozess teilnehmen und somit die anfallende Last besser auf das Netzwerk verteilt werden kann (ansonsten wären wenige Instanzen stärker ausgelastet beim Wiederherstellungsprozess). Damit bleibt das Netz stabiler gegenüber solcher Szenarien. Abbildung 21 zeigt den Unterschied zwischen einem Wiederherstellungsprozess ohne und mit verteilten vnodes, wenn Knoten 2 ausfallen würde. Die grauen Knoten sind am Prozess beteiligt, die weißen nicht. Die Last wird ohne vnodes nur auf drei Instanzen verteilt (~33 % je Knoten), während mit vnodes alle fünf Knoten im Netz beteiligt sind(~20 % je Knoten). Dies reduziert die Auslastung der betroffenen Knoten deutlich.

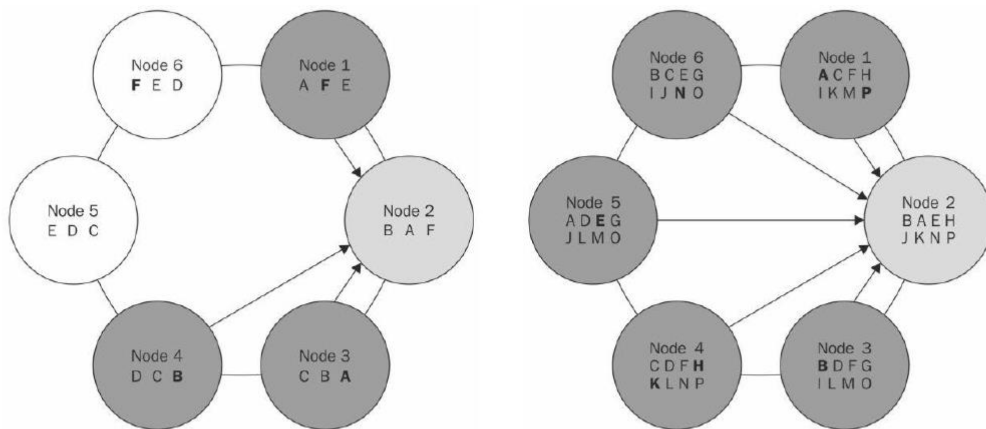

Abbildung 21: Wiederherstellung eines Knotens ohne (links) und mit (rechts) vnodes

7.2.3 Wohin Daten intern geschrieben werden

Wird in Cassandra ein Schreibvorgang getätigt, wird dies intern an mehreren Stellen festgehalten. Neue Daten werden dabei sofort an zwei Stellen geschrieben: In die *Memtable*, die sich im Arbeitsspeicher befindet und als schneller Datencache agiert, und den auf Festplatte befindlichen *Commitlog*, dem die Informationen zur Operation angehängt werden und diesen somit als Aktionsverlauf und Wiederherstellungshilfe dienen lassen. In der Memtable werden die Daten dabei als Schlüssel-Eintrag Werte hinzugefügt und die gesamte Tabelle nach dem Schlüssel sortiert. Da die Memtable nur eine bestimmte Größe besitzt (abhängig vom System und änderbar durch den Anwender), werden ihre Daten in eine sogenannte *SSTable* auf Platte persistiert, sobald diese voll ist. SSTables sind dabei nur-lesbar, wodurch eine neue SSTable angelegt wird, wenn Änderungsoperationen auf einen bereits gespeicherten Datensatz aufgerufen werden. [30]

Sollen Daten aus Cassandra gelesen werden, wird zunächst die Memtable auf den gewünschten Schlüssel untersucht. Sollte sich dieser nicht mehr in der Hauptspeichertabelle befinden, wird die SSTable durchsucht. Um eine Suche nach diesen Daten zu beschleunigen, besitzt die Tabelle drei Komponenten: Über den *Bloom-Filter* kann schnell nach dem angefragten Schlüssel gesucht und somit festgestellt werden, ob sich der Datensatz überhaupt in dieser SSTable befindet. Über den *Index* kann die Stelle gefunden werden, an der sich der Datensatz befindet und über die *Data* Komponente die Daten anschließend ausgelesen werden. Abbildung 22 zeigt die Cassandra-Komponenten auf, die bei der Ausführung von Schreiboperationen beteiligt sind.

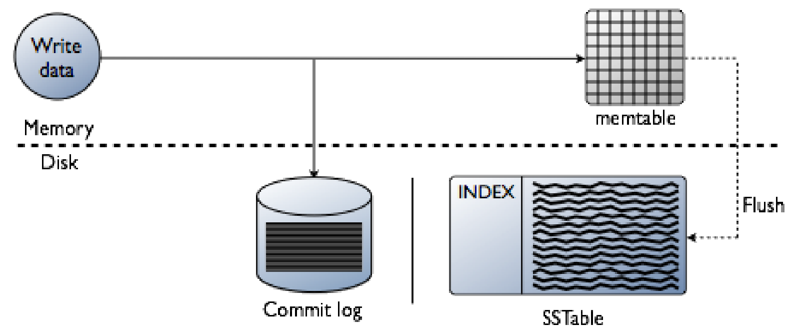

Abbildung 22: Schreibvorgänge bei Cassandra [30]

Um die interne Datenrepräsentation zu verdeutlichen, werden folgende Schreiboperationen untersucht:

```
(1)   INSERT INTO k1.t1 (c1) VALUES (v1);
(2)   INSERT INTO k2.t1 (c1, c2) VALUES (v1, v2);
(3)   INSERT INTO k1.t1 (c1, c3, c2) VALUES (v4, v3, v2);
```

Codeblock 14: Beispielhafte Schreiboperationen in Cassandra [30]

In der Memtable werden die Operationen zusammengefasst festgehalten:

```
(1)   k1 c1:v4 c2:v2 c3:v3
(2)   k2 c1:v1 c2:v2
```

Codeblock 15: Repräsentation der eingefügten Daten in der Memtable [30]

Der Commitlog beachtet hierbei, im Gegensatz zur Memtable, auch die Reihenfolge, in der die einzelnen Statements angekommen sind:

```
(1)   k1, c1:v1
(2)   k2, c1:v1 C2:v2
(3)   k1, c1:v4 c3:v3 c2:v2
```

Codeblock 16: Repräsentation der eingefügten Daten im Commitlog [30]

Ist die Memtable voll, wird diese in einer SSTable persistiert. Sie enthält entsprechend die gleichen Daten wie zuvor die Hauptspeicher-Tabelle:

```
(1)   k1 c1:v4 c2:v2 c3:v3
(2)   k2 c1:v1 c2:v2
```

Codeblock 17: Repräsentation der eingefügten Daten in der SSTable [30]

Da es durch Ausführen von Änderungsoperationen und mehrmaligen Festschreiben voller Memtables vorkommt, dass es eine Vielzahl an SSTables gibt, ist es wichtig, regelmäßig eine sogenannte *Compaction* durchzuführen. Hierbei werden mehrere SSTables zu einer neuen zusammengefügt und somit das Suchen nach Schlüsseln beschleunigt. Die Durchführung dieses Mergings wird vom System, entsprechend der Einstellungen (standardmäßig bei 4 – 32 SSTables), durchgeführt. Eine Compaction kann auch manuell über das *nodetool* von Cassandra durchgeführt werden. [31]

7.2.4 Datengröße

Aufgrund der internen Schreibstrategien können Daten für eine gewisse Dauer sowohl im Hauptspeicher (memtable), als auch mehrfach auf der Festplatte (SSTables), vorhanden sein. Durch Ausführung von Compactions werden diese Duplikate regelmäßig bzw. manuell vom Anwender zu einer SSTable auf Platte zusammengefasst, um Speicher zu sparen. Doch auch wenn nach Durchführung einer Compaction nur noch eine finale SSTable besteht, kann deren Größe das reine Datenvolumen, welches es zu speichern galt, um ein Drei- bis Zehnfaches übersteigen. Dies ist laut [32] insbesondere auf den dynamischen Zeilenaufbau zurückzuführen, der es erlaubt, Zeilen mit unterschiedlichen Spaltenanzahlen und -namen in einer Tabelle zu speichern.

> *Every column in Cassandra incurs 15 bytes of overhead. Since each row in a table can have different column names as well as differing numbers of columns, metadata is stored for each column. [32]*

Wie sich die Größe von Tabellen anhand der geplanten Tabellenstruktur im Vorfeld berechnen lässt, zeigt [32] auf. Diagramm 2 vergleicht die Datengrößen eines strukturierten DEBS-Datensatzes[41] vor und nach der Speicherung im Cassandra-Cluster mit einem Replikationsfaktor von eins (keine zusätzlichen Kopien zur Sicherung im Cluster). Hier entspricht die Datenmenge nach Speicherung im Cassandra-Cluster etwa dem 2,5-fachen (Tendenz steigend bei größerer Datenmenge) der ursprünglichen CSV-Größe.

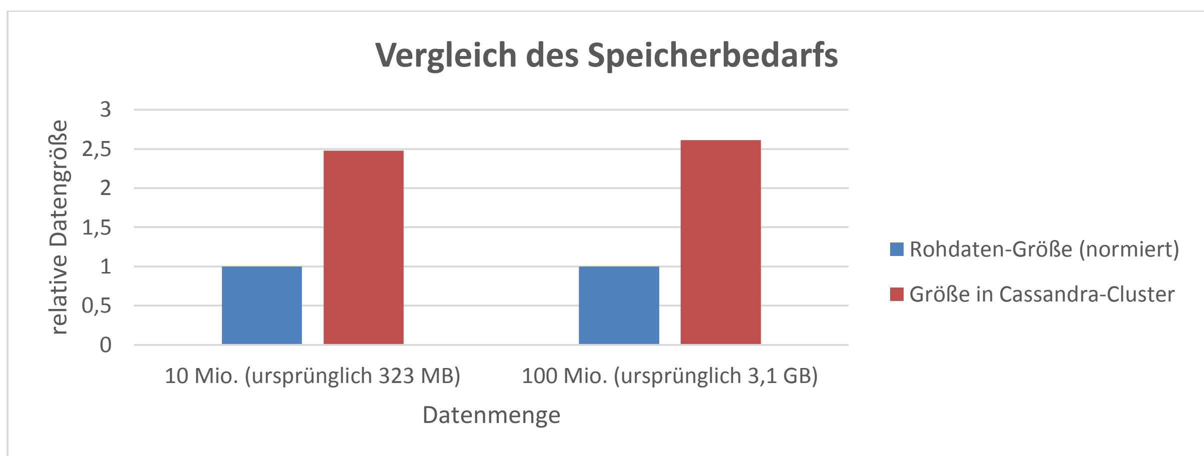

Diagramm 2: Vergleich des Speicherbedarfs (Rohdaten und Cassandra-Cluster)

[41] Aufbau des DEBS-Datensatzes in Kapitel 9.2

7.2.5 Ressourcen-Abstimmung

Da Cassandra auf Java-Basis läuft, können besonders Einstellungen des zu verwendenden Arbeitsspeichers Performancegewinne bringen. Einstellungen zu Hauptspeicherverwendung können in der *cassandra-env.sh* Konfigurationsdatei getroffen werden. Von besonderer Bedeutung sind dabei zwei Werte, die die Reservierung von Hauptspeicher durch Cassandra beeinflussen. Der erste Parameter lautet *MAX_HEAP_SIZE*. Er setzt die Menge an Arbeitsspeicher, die jede Instanz beanspruchen darf, fest. Da dieser Wert gleichzeitig auch als untere Grenze gesetzt wird, bleibt Cassandra immer die gleiche Größe zugewiesen und eine teure Auslagerung des Arbeitsspeichers wird vermieden. Zu beachten ist, dass selbst in großen System von einer Zuweisung von mehr als 8 GB Hauptspeicher abgesehen werden sollte, da ansonsten die Garbage Collection seltener durchgeführt wird und dann zu viel Zeit und Ressourcen in Anspruch nimmt. Auch bei intelligenter Hauptspeicherhaltung wichtiger Objekte seitens des Betriebssystems (Page Cache, dieser liegt außerhalb des Java-Heaps) wird eine größere Hauptspeicherzuweisung nicht empfohlen. Folgende Tabelle zeigt die Standardzuweisung des Arbeitsspeichers bei Cassandra auf:

Arbeitsspeicher der Instanz	Heap Size (Minimum & Maximum)
< 2 GB	50 % des Systemspeichers
2 – 4 GB	1 GB
> 4 GB	25 % des Speichers, aber maximal 4 GB

Tabelle 3: Standardmäßige Speicherzuweisung bei Cassandra

Eine weitere Einstellung ist das Ändern des Parameters *HEAP_NEWSIZE*. Er gibt an, wie groß der Anteil von gelöschten Young-Generation Objekten (Objekte, die von der JVM nicht lange verwendet wurden) innerhalb der Heap-Size sein darf, ehe eine Garbage Collection durchgeführt wird. Ein guter Richtwert zwischen Overhead bei zu vielen und einer Datenanhäufung bei zu wenigen Ausführungen ist eine Größe im Bereich <Anzahl Rechenkern der Instanz> * 100 MB, also einer HEAP_NEWSIZE von 800 MB bei 8 Cores. [33]

7.3 *Spark & Cassandra*

Dieses Kapitel zeigt eine Möglichkeit auf, über welche in Spark-Applikationen auf Cassandra-Tabellen zugegriffen werden kann. Anhang A.7 zeigt, wie zunächst Spark und Cassandra in einem gemeinsamen Cluster installiert werden können und welche Befehle durch das Cassandra-Tool *cqlsh* zur Tabellenerstellung, Befüllung und Manipulation zur Verfügung stehen. Mithilfe des Java-Wrappers *spark-cassandra-connector* von DataStax [34] werden Funktionen zur Kommunikation zwischen diesen beiden Plattformen bereitgestellt. Es können sowohl Daten aus einem RDD in Cassandra-Tabellen geschrieben, als auch von Cassandra in

RDDs gelesen werden. Auch *CQL*-Befehle[42] zur Datenselektion und Manipulation können ausgeführt werden. Der Connector wird über GitHub verwaltet und ist unter Apache Lizenz Version 2.0 frei verfügbar. Er kann von dort aus zum eigenen Assemblieren heruntergeladen werden oder alternativ über Maven als Bibliothek bezogen werden. Die Koordinaten für den Connector lauten:

```
(1)    <groupId>com.datastax.spark</groupId>
(2)    <artifactId>spark-cassandra-connector-java_2.10</artifactId>
(3)    <version>1.2.2</version>
```

Codeblock 18: Maven-Koordinaten zur Verwendung des spark-cassandra-connectors

Wichtig hierbei ist, auf Versionskompatibilität zwischen Spark, Cassandra und dem Connector zu achten: Die Versionsangabe der Koordinaten sollte entsprechend der Spark-Version gewählt werden[43]. Nach Einbinden der Bibliothek kann auf die Java-Funktionen des Connectors über die Klasse *CassandraJavaUtil* zugegriffen werden.

Folgender Befehl kann zum Schreiben nach Cassandra verwendet werden:

```
(1)    JavaRDD<EntityClass> RDDtoSave = ...;
(2)    CassandraJavaUtil.javaFunctions(RDDtoSave).writerBuilder("keyspace",
        "table", CassandraJavaUtil.mapToRow(EntityClass.class))
        .saveToCassandra();
```

Codeblock 19: Speichern eines RDDs nach Cassandra

Hierbei wird eine serialisierbare Klasse benötigt, die zum Mapping der RDD-Daten auf die Tabelle verwendet werden kann (im Beispiel die Klasse *EntityClass*). Diese Klasse muss für jede Tabellenspalte ein gleichnamiges und passend typisiertes Attribut mit getter- und setter-Methode enthalten. Über diese werden dann die RDD-Inhalte auf die Tabellenspalten gemappt.

Das Lesen von Cassandra-Tabellen nach Spark-RDDs wird ebenfalls von einem einzigen Methodenaufruf erledigt:

```
(1)    JavaRDD<String> rdd = CassandraJavaUtil.javaFunctions(sc)
        .cassandraTable("keyspace", "table")
        .map(new Function<CassandraRow, String>() {
(2)       public String call(CassandraRow cassandraRow) throws Exception {
(3)         return cassandraRow.toString();
(4)       }
(5)     });
```

Codeblock 20: Lesen von Cassandra-Daten in ein Spark-RDD

Diese beiden Befehle bilden die Einstiegspunkte der Spark-Cassandra Programmierung. Weitere Dokumentation und Downloadmöglichkeiten bietet [34].

[42] CQL (Continous Query Language)-Befehle sind SQL-artige Befehle, die u.a. bei Cassandra und anderen NoSQL-Datenbanksystemen zum Einsatz kommen.

[43] Bei Cassandra 2.1 und Spark 1.2.2 sollte beispielsweise die Connector-Version 1.2.2 verwendet werden.

7.4 Benchmarks

In diesem Kapitel sind die Ergebnisse von Schreib- und Lesebenchmarks festgehalten, die im Laufe der Arbeit mit Cassandra und HDFS durchgeführt wurden. Die zu speichernden Daten wurden bei den Schreibtests über ein Spark-Cluster aus einem S3-Bucket ausgelesen und an das jeweilige Zielsystem weitergereicht. Bei den Lesetests wurden entsprechend die zuvor gespeicherten Daten aus dem HDFS bzw. aus Cassandra-Tabellen in Spark-RDDs ausgelesen. Grundlage für die Benchmarks waren dabei Daten der DEBS Challenge 2014[44]. Zur Messung wurden 10 / 100 / 500 Millionen Datensätze herangezogen, was einer Eingangs-Dateigröße von 0,32 / 3,3 / 16,5 GB entspricht. Für die Cluster wurden jeweils AWS EC2-Instanzen des Typs m3.2xlarge verwendet:

CPU	Intel Xeon E5-2670 (8 Kerne, 2.6 GHz)
RAM	30 GB
Speicher	2 x 80 GB SSD
OS	Ubuntu 12.04 LTS

Tabelle 4: Spezifikationen einer EC2-Instanz des Typs m3.2xlarge

7.4.1 Cassandra

Für die Benchmarks mit Cassandra wurde ein Keyspace mit *ReplicationFactor=1* angelegt, was eine Datenreplikation verhindert und somit jeglichen anfallenden Overhead vermeidet. Die Datensätze (10 / 100 / 500 Mio.) wurden jeweils in eine eigene Tabelle geschrieben, um auch hier einen Einfluss durch bereits bestehende Daten zu unterbinden. In den Benchmarks wurde über den Spark-Cassandra-Connector eine Kommunikation zwischen beiden Systemen ermöglicht[45], um die Daten austauschen zu können.

Schreiben von S3 nach Cassandra

Für den Schreibtest wurden zwei separate Cluster für Spark und Cassandra angelegt, um eine maximale Performance jedes Clusters zu ermöglichen. Da die Übertragungsgeschwindigkeit im EC2-Umfeld innerhalb einer Availability Zone kein Flaschenhals darstellt und hierbei auch nur das Schreiben von Daten überprüft werden soll, konnte dieser Versuchsaufbau so umgesetzt werden. Der Quellcode, welcher als Spark Benchmark-Programm diente, ist im Anhang unter Codeblock 46 definiert. Tabelle 13 (ebenfalls im Anhang) stellt die detaillierten Ergebnisse des Schreibtests dar. Eine optimale Schreibgeschwindigkeit wurde in gleich großen Clustern erreicht, d.h. die Arbeitsknoten des Spark-Clusters hatten in Summe gleich viele Rechenkerne wie das Cassandra-Cluster. Bei Clusteraufbau mit dem oben genannten Instanztyp wurden beim Schreiben folgende Ergebnisse erzielt:

[44] Nähere Informationen zur DEBS Challenge und zum Datenaufbau unter Kapitel 9.1.1 bzw. 9.2
[45] Nähere Informationen über den Spark-Cassandra-Connector unter Kapitel 7.3

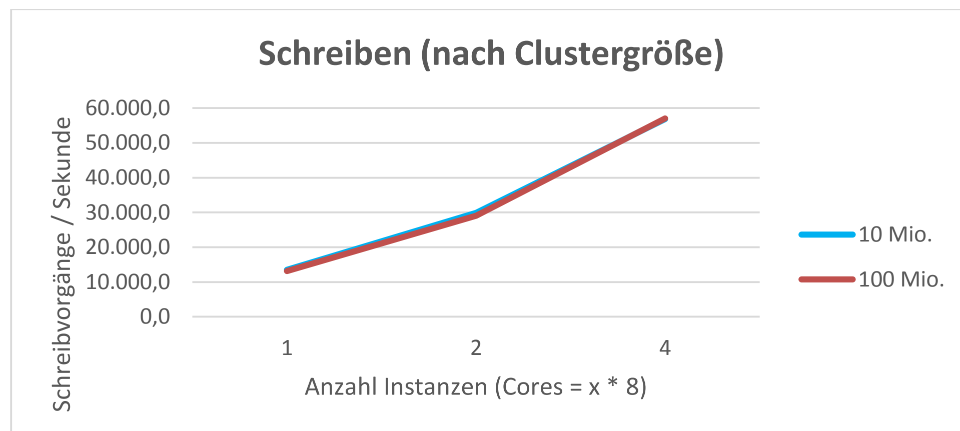

Diagramm 3: Verarbeitete Anfragen im Cassandra Schreibtest

Folgende Erkenntnisse können aus diesem Ergebnis gezogen werden:

- Die Anzahl der verarbeiteten Schreibanfragen wächst linear mit der Anzahl zur Verfügung stehender Knoten im Cluster.
- Bei Verwendung dieses Instanztyps liegt die Anzahl verarbeiteter Schreibanfragen pro Core bei rund 1700 – 1800 Stück pro Sekunde. Diese Anzahl bleibt über die unterschiedlichsten Clustergrößen unverändert.

Die Schreibgeschwindigkeit ist dabei unabhängig von der Datengröße, die verarbeitet werden muss:

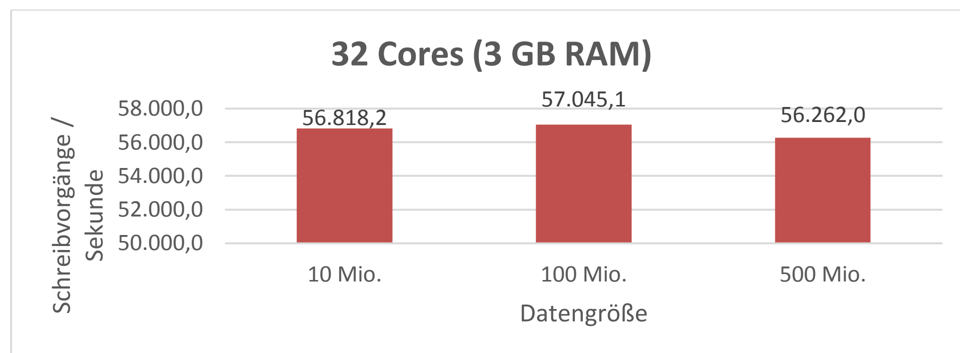

Diagramm 4: Die Schreibgeschwindigkeit ist unabhängig von der Datengröße

Als suboptimale Ressourcennutzung kann eine ungleiche Clustergröße angesehen werden: Besteht das Spark-Cluster aus mehr Worker-Cores als das Casandra-Netz, kommt es zu einem Anfragenüberschuss, es werden also mehr Anfragen an das Cassandra-Cluster gestellt, als verarbeitet werden können. Ist das Spark-Netz kleiner dimensioniert als das Cassandra-Cluster, kommt es dagegen zu einem nur schwach ausgelasteten Cassandra-Netz, da Spark zu wenige Anfragen abschickt, um alle Rechenkerne auszulasten.

Ebenso unverändert blieben die Ergebnisse beim Vergleich von Spark-Clustern mit unterschiedlich großer Hautspeicher-Reservierung. Auf Spark-Seite reichen hier die standardmäßig eingestellten 512 MB RAM pro Core, um ein gutes Ergebnis zu erzielen. Es wurden keine Unterschiede zum Durchlauf mit 3 GB RAM pro Core festgestellt.

Lesen von Cassandra in Spark-RDDs

Das Programm für das Lesen von Cassandra-Tabellendaten in Spark-RDDs stellt Codeblock 47 im Anhang dar. Für eine optimale Lesegeschwindigkeit diente, im Gegensatz zum Schreibtest, ein einzelnes Cluster, in welchem Spark und Cassandra gemeinsam auf denselben Nodes installiert wurden. Dies führt dazu, dass sich jeder Arbeitsknoten von Spark um genau die Daten des Cassandra-DataNodes kümmert, welcher auf der gleichen Instanz läuft. Die Daten wurden für diesen Benchmark erneut ins Cluster geschrieben, um eine optimale Verteilung der Daten zu ermöglichen, denn auch hier werden die Daten eines Workers auf den jeweils lokal laufenden DataNode geschrieben. Die aufgesetzte Umgebung lässt sich durch Abbildung 23 graphisch beschreiben: Auf jeder Cassandra-Instanz (C*) läuft ein Spark-Worker, zusätzlich wird auf einem separaten Knoten der Spark-Driver laufengelassen (ohne Cassandra-Installation). Wie ein solches gemeinsames Cluster aufgesetzt werden kann, beschreibt Anhang A.7.

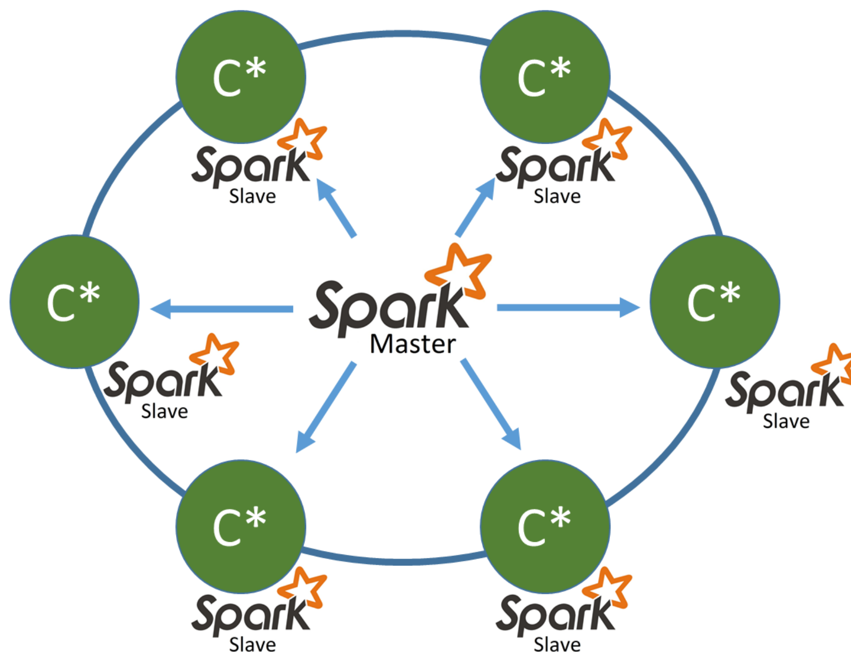

Abbildung 23: Spark und Cassandra in einem gemeinsamen Cluster

Tabelle 14 im Anhang stellt die Ergebnisse des Lesetests dar. Graphisch ist die Anzahl verarbeiteter Leseanfragen im 1-/2-/4-Node Cluster über Diagramm 5 festgehalten.

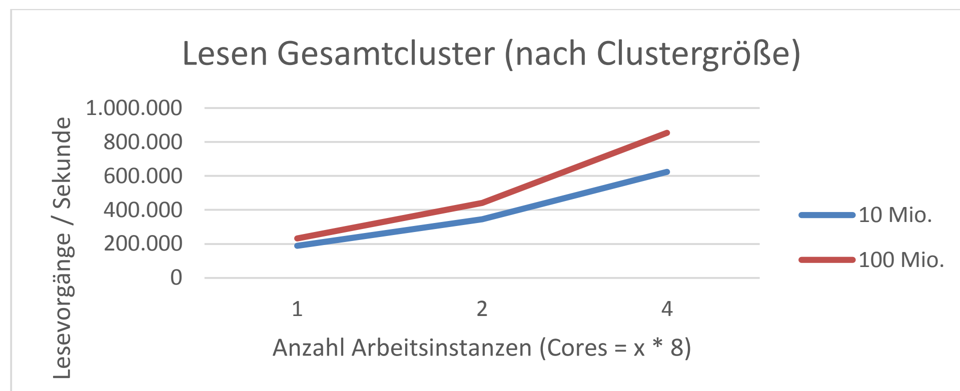

Diagramm 5: Verarbeitete Leseanfragen im Spark-Cassandra Cluster

Über den Lese-Benchmark konnten folgende Erkenntnisse gezogen werden:

- Die Performance beim Auslesen von Daten aus Cassandra mit Spark lässt sich im Groben als linear zur Clustergröße bezeichnen.
- Aufgrund von Organisationsoverhead seitens Cassandra werden bei größeren Tabellen im Durchschnitt mehr Daten pro Zeiteinheit gelesen als bei Kleineren (100 Mio. zu 10 Mio.).
- Bei größeren Clustern fällt ebenfalls ein größerer Kommunikationsaufwand an als bei kleineren, was zu einer leicht sinkenden Verarbeitungszahl je Core führt, wie Diagramm 6 aufzeigt.
- Eine Lokalität von Spark-Worker und Cassandra-DataNode führt zu einem erheblichen Performance-Gewinn, da der Netzwerk-Verkehr minimiert wird. Laufzeiten bei separaten Clustern waren teils doppelt so hoch und ein linearer Zusammenhang zwischen Clustergröße und Performance war nicht gegeben!

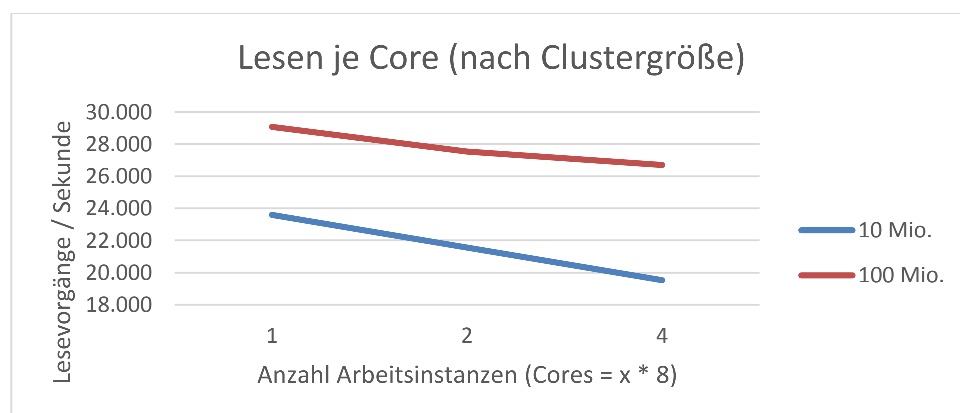

Diagramm 6: Leseanfragen pro Core bei unterschiedlicher Clustergröße

7.4.2 HDFS

Bei diesem Benchmark kam das beim Aufsetzen eines Spark-Clusters im EC2-Umfeld mitinstallierte verteilte Hadoop Dateisystem HDFS zum Einsatz. Um eine maximale Performanz zu erreichen, wurde für die folgenden Benchmarks mit der flüchtigen (*ephemeral*) HDFS-Variante gearbeitet. Durch Abändern des Replikationsfaktors auf 1, über die Konfigurationsdatei *hdfs-site.xml*[46], konnten ggf. dadurch anfallende Arbeitslasten vermieden werden. In diesen Benchmarks wurden, im Gegensatz zu Cassandra, die Dateien des S3-Buckets ohne Mapping-Schritte direkt per Spark-Cluster geschrieben bzw. ausgelesen. Datenknoten (HDFS-DataNodes) liefen dabei auf allen Arbeitsinstanzen des Clusters, während der Koordinierungsservice (NameNode, JobTracker) auf dem Master-Knoten ausgeführt wurde. Die Anzahl Rechenkerne zum Verarbeiten der Daten entsprechen somit der Anzahl an DataNodes * 8 (da 8 Cores je Instanz).

Schreiben von S3 nach HDFS

Der verwendete Quellcode des Schreib-Benchmarks ist im Anhang unter Codeblock 48 zu finden. Über Parameter kann das Programm mit unterschiedlicher Konfiguration gestartet werden. Neben dem zu verwendenden Hauptspeicher pro Spark-Prozess kann hier die Datenquelle (Datei(en) mit 10 / 100 / 500 Mio. Datensätzen) und auch die im Spark-RDD zu verwendende Partitionszahl angegeben werden.

Wichtig: Die Partitionszahl eines RDDs beeinflusst hierbei das Erstellen der Zieldatei im HDFS. Eine Datei besteht nach dem Schreibvorgang aus so vielen Teildateien, wie das geschriebene RDD Partitionen enthält. Beim Einlesen von S3-Dateien in RDDs sollte deshalb eine Partitionszahl, welche im Bereich des 1- bis 5-fachen der Coreanzahl liegt und sich restlos durch diese teilen lässt, für eine optimale Verarbeitungsgeschwindigkeit gewählt werden[47]. Bei zu vielen Partitionen wird die Laufzeit wiederum durch steigenden Overhead negativ beeinflusst. Tabelle 15 (im Anhang) zeigt die Ergebnisse des Schreibtests, welche wiederum in Diagramm 7 visualisiert wurden. Ergebnisse von Schreibanfragen mit besonders wenigen bzw. vielen Partitionen wurden dabei bei der Mittelwertberechnung nicht miteinbezogen. Bei Clusteraufbau mit dem oben genannten Instanztyp (m3.2xlarge, je 8 Rechenkerne pro Instanz) wurden beim Schreiben folgende Ergebnisse erzielt:

[46] Konfiguration von HDFS unter Kapitel 7.1

[47] Welche Laufzeit-Vorteile eine restlos auf die Coreanzahl teilbare Partitionszahl mit sich bringt, zeigt Kapitel 6.7 bzw. Abbildung 18

Diagramm 7: Zeiten, die zum Schreiben von Dateien nach HDFS benötigt wurden

Folgende Erkenntnisse können aus diesem Ergebnis gezogen werden:

- Vom Hinzufügen von Instanzen profitieren besonders große Dateien, bei Kleinen wird die Laufzeit kaum beeinflusst.
- Das Ergebnis des Schreibtests mit 500 Mio. Datensätzen profitiert zusätzlich von der bereits im S3 vorgenommenen Zerteilung in 10 Subdateien, da die Partitionierung hierbei auf Spark-Seite schneller vonstattengeht.
- Die Schreibgeschwindigkeit bei großen Dateien (hier 3 und 16,5 GB) wächst bei Verdoppelung der Rechenkerne um ein 1,5- bis 2-faches an.
- Eine Vergrößerung des Hauptspeichers bringt keine merklichen Performancesteigerungen mit sich. Bei den Laufzeiten wurden zwischen einer Konfiguration mit 512 MB und 3 GB keine Unterschiede festgestellt.

Lesen von HDFS in Spark-RDDs

Dieser Benchmark untersucht die Lesegeschwindigkeit des HDFS. Die zuvor geschriebenen Dateien werden mit unterschiedlicher Hauptspeichergröße und Partitionszahl ausgelesen und die Ergebnisse verglichen. Auch dieser Test wurde auf verschieden großen Clustern durchgeführt (je mit Instanzen des Typs m3.2xlarge, 8 Cores je Instanz). Codeblock 49 im Anhang stellt den zur Messung der Lesegeschwindigkeit eingesetzten Programmcode dar. Die Ergebnisse sind in Tabelle 16 (Anhang) festgehalten. Zur einfacheren Übersicht wurden diese ebenfalls als Diagramme zusammengefasst. Diagramm 8 stellt die gemessene Laufzeit in Sekunden (links) bzw. die daraus resultierende Lesegeschwindigkeit in MB/Sekunde (rechts) für die verschiedenen Dateigrößen (0,3 / 3,3 / 16,5 GB) dar.

Wichtig: Ebenso wie im obigen Schreibtest spielt die Anzahl der Stückelungen einer Datei und die Partitionsanzahl, in welche ein RDD unterteilt wird, eine große Rolle. Es werden beim

Einlesen immer mindestens so viele Partitionen erstellt, wie die Datei selbst Partitionen enthält bzw. wie viele Blöcke diese im HDFS beansprucht. Der größere Wert der beiden Eigenschaften wird als Standard-Partitionszahl genommen, sofern beim Einlesen über die Spark-Methode *sc.textFile* keine größere Mindestvorgabe gemacht wird.

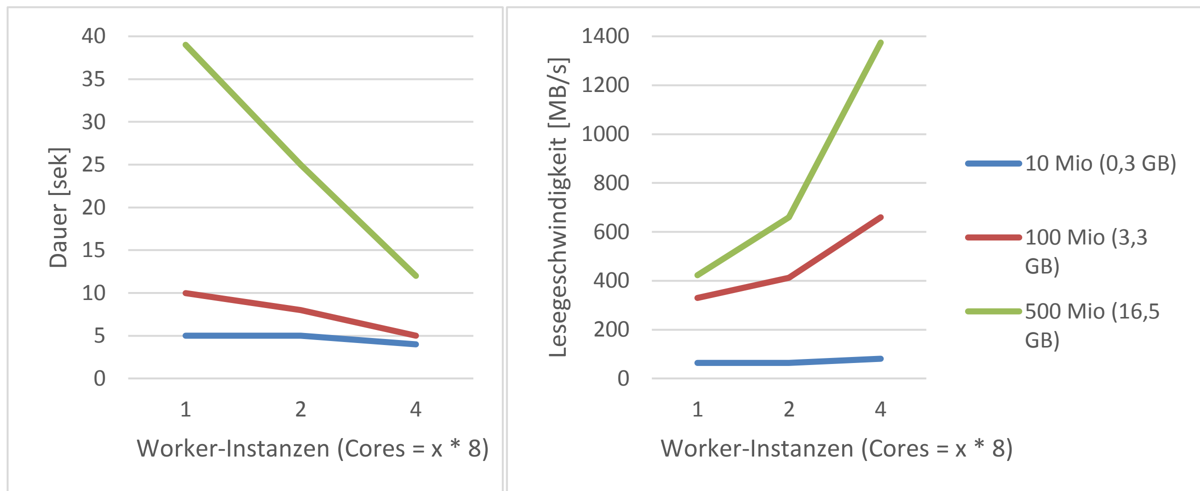

Diagramm 8: Lesegeschwindigkeiten von HDFS in Spark-RDDs

Folgende Erkenntnisse können aus den Ergebnissen gezogen werden:

- Vom Hinzufügen von Instanzen profitieren besonders große Dateien, bei Kleinen wird die Laufzeit kaum verbessert, da sie bereits nach kürzester Zeit übertragen sind.
- Das Ergebnis des Schreibtests mit 16,5 GB Datengröße profitiert zusätzlich von der bereits im S3 vorgenommenen Zerteilung in 10 Subdateien. Da diese beim Speichern erhalten bleiben, können hier Leseanfragen parallel und mit kleineren Teildateien durchgeführt werden.
- Die Lesegeschwindigkeit wächst linear mit der Anzahl Instanzen / Cores
- Eine Vergrößerung des Hauptspeichers bringt keine merklichen Performancesteigerungen mit sich. Bei den Laufzeiten wurden zwischen einer Konfiguration mit 512 MB und 3 GB keine Unterschiede festgestellt.

8 Streaming Data und maschinelles Lernen mit Spark

In diesem Kapitel werden zwei wichtige Komponenten neben dem Spark Core vorgestellt. Sowohl Spark Streaming zur Verarbeitung von Datenströmen als auch MLlib, die Bibliothek zum maschinellen Lernen, werden erläutert und deren Funktionsumfang und Verwendung kurz aufgezeigt. Den Abschluss des Kapitels bildet die Vorstellung von Apache Zeppelin, ein Incubator Projekt, welches zur Visualisierung von Daten verwendet werden kann.

8.1 Spark Streaming

Spark Streaming stellt eine weitere Komponente der Spark-Plattform dar. Mit ihr wird es ermöglicht, Echtzeit-Anwendungen zu erstellen, über welche beispielsweise besondere Ereignisse durch Ausreißererkennung erkannt oder eine Besucherstatistik realisiert werden können. Im Streaming-Bereich von Spark kommen sogenannte *diskretisierte Streams* oder kurz *DStreams* zum Einsatz. Sie sind eine Abstraktion der Daten, die in einem bestimmten Zeitintervall am System ankommen, von diesem gepuffert und schließlich verarbeitet werden. Hier kommt bei Spark das in Kapitel 2.3.4 vorgestellte Micro-Batching zum Einsatz. Intern besteht ein DStream aus einer Menge von RDDs (je Intervall ein RDD), welche wiederum ausgelesen und bearbeitet werden können. Auf DStreams selbst können, wie auch bei RDDs, Transformationen ausgeführt werden. Beim Ausführen werden dabei (ebenfalls analog zu den RDDs) neue DStreams erzeugt. Die Transformationen lassen sich dabei in zwei Arten unterteilen: *Zustandslose* und *zustandsbehaftete Transformationen*. Bei ersteren werden lediglich die Daten des zum Zeitpunkt des Ausführens aktuellen RDDs im DStream bearbeitet. Transformationen mit Zuständen können dagegen über längeren Zeitraum hinweg Daten von DStreams miteinbeziehen. Neben Transformationen können auf DStreams auch sogenannte *Output Operations* aufgerufen werden, mit deren Hilfe Daten festgeschrieben werden. Sie entsprechen den Aktionen auf RDDs, werden aber auf jedem RDD des DStream angewandt. Spark Streaming unterstützt eine Vielzahl an Datenquellen, darunter standardmäßig *Flume*, *Kafka*, *HDFS/S3*, *Twitter*, *ZeroMQ*, *Kinesis* und *TCP-Socket-Streams [35]*. Auf letztere wird im Laufe der Arbeit nochmals genauer eingegangen. Ebenfalls ein Unterschied zu herkömmlichen RDDs stellt die zusätzliche Eigenschaft des dauerhaften Ausführens dar. Für das Ausführen von 24/7-Operationen ist das Einrichten von regelmäßigen Zwischenspeicherungen (*Checkpointing*) vonnöten, um einen reibungslosen Dauerbetrieb, auch im Falle des Ausfalls von Knoten, zu ermöglichen.

8.1.1 Architektur und Micro-Batching bei Spark Streaming

Spark verwendet Micro-Batching zur Verarbeitung von Streaming Data. Dabei werden die kontinuierlich ankommenden Daten in einem RDD gepuffert und nach einer festen Zeitspanne verarbeitet, dem sogenannten *Batch Interval*. Sobald die Verarbeitung dieses Batch-RDDs beginnt, wird ein neues RDD angelegt und ankommende Daten hierin gepuffert. Die Spark-Applikation kann dabei mehrere Empfänger für Daten definieren und somit größere Streams bzw. Streams aus unterschiedlichen Quellen entgegennehmen.

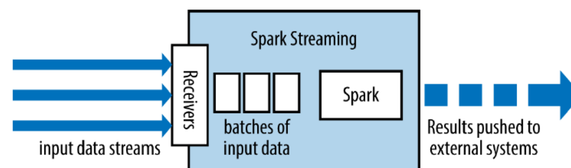

Abbildung 24: Spark Streaming Architektur [27]

Zu beachten ist hierbei, dass Spark für jeden Datenempfänger einen eigenen Executor-Thread erstellt. Zur Sicherstellung der Fehlerfreiheit werden außerdem alle Daten über einen weiteren Task im Cluster repliziert. Es muss deshalb beim Aufsetzen des Clusters auf Reservierung ausreichender Rechenressourcen geachtet werden. Für eine Ausfallsicherheit wird im Produktivbetrieb der Einsatz von mehreren Arbeitsinstanzen empfohlen, um die zuvor angesprochene Datenreplikation auf verschiedenen Instanzen durchführen zu können. Die Anzahl zu reservierender Rechenkerne liegt dabei bei mindestens *<<Anzahl Empfänger>> + 1*. Abbildung 24 zeigt den groben Aufbau des Spark Streaming auf. Auf DStreams können beliebige Transformationen ausgeführt werden, wobei jeweils neue DStreams entstehen (analog zu RDDs). Abbildung 25 zeigt den Ablauf beim Entgegennehmen und Filtern von Streaming Data anhand des Beispiels der Filterung von Log-Dateien: Es wird zunächst der *lines*-DStream angelegt und aus diesem durch Aufruf der *filter*-Transformation ein weiterer, paralleler DStream generiert.

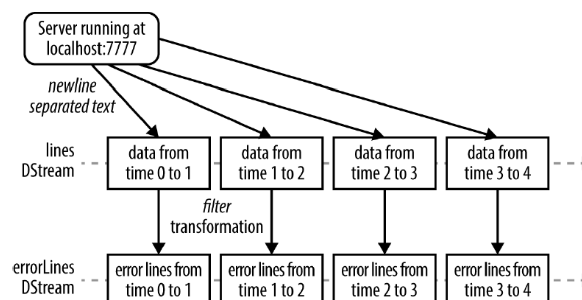

Abbildung 25: Erstellen von DStreams über Transformationen [27]

8.1.2 Zustandslose und zustandsbehaftete Transformationen

Die Transformationen, die auf DStreams aufgerufen werden, lassen sich in zwei Kategorien einteilen: *Zustandslose* und *zustandsbehaftete Transformationen*. Erstere führen Operationen auf dem aktuellsten Batch eines DStreams durch, ohne die zurückliegenden Batches zu beachten. Diese Transformationen entsprechen dem Vorgehen bei regulären RDDs, auch die meisten dort verfügbaren Transformationen wie z. B. *filter*, *map*, *reduceByKey* usw. können aufgerufen werden. Eine zustandslose Transformation wird standardmäßig nach jedem Batchintervall durchgeführt. Werden mehrere Datenempfänger eingesetzt, können diese einzelnen DStreams mithilfe der *union*-Transformation zu einem großen DStream zusammengefasst werden. Die zweite Kategorie bilden die zustandsbehafteten Transformationen: Hierbei können, im Gegensatz zu den zustandslosen, auch die vorhergegangenen Batches beachtet und in die Berechnungen miteingebunden werden. Sie stellen den Hauptunterschied zur normalen Batchverarbeitung dar und ermöglichen eine Analyse bzw. Datenbearbeitung zeitnah zum Empfang von Daten. Es gibt wiederum zwei Arten von zustandsbehafteten Transformationen: Die sogenannten *Zeitfenster-Transformationen* führen Berechnungen auf einer bestimmten Menge an Batchintervallen durch, also über einen festen Zeitraum. Die beiden wichtigsten Parameter hierbei sind die *Window Size* (Größe des Zeitfensters) und die *Slide Duration* (Aktualisierungsintervall). Über die Window Size wird festgelegt, wie viele Batches in die Berechnung miteinfließen sollen, ältere Batches werden dabei verworfen. Die Slide Duration gibt an, nach wie vielen Batchintervallen eine erneute Ausführung der Transformation stattfinden soll und legt damit das Aktualisierungsintervall für eine Berechnung fest. Der aus der Transformation entstehende DStream kann anschließend wie gewohnt durch weitere Transformationen bearbeitet bzw. über Output Operations festgeschrieben werden. Standardmäßig wird nach jedem Batchintervall eine Berechnung angestoßen, sofern dies nicht über die Slide Duration erhöht wird. Sowohl Window Size als auch Slide Duration müssen ein Vielfaches des Batchintervalls sein. Abbildung 26 zeigt den Aufruf einer zustandsbehafteten Transformation mit Window Size von 3 und einer Slide Duration von 2.

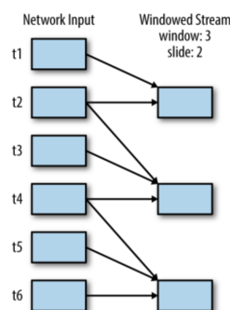

Abbildung 26: Ablauf einer zustandsbehafteten Transformation [27]

Die zweite Art von zustandsbehafteten Transformationen stellen sogenannte *UpdateStateByKey-Transformationen* dar. Sie führen Berechnungen über alle bisherigen Batches aus und ermöglichen damit das dauerhafte Verfolgen von Zuständen. Die Transformation ermöglicht eine Aktualisierung von entsprechenden Status- bzw. Zustandsobjekten. Ausgangspunkt für den Aufruf dieser Methode ist ein *PairDStream* von Schlüssel-Wert Paaren, aus welchen dann über die einzelnen Schlüssel Zustände gebildet werden. Diese internen Zustandsobjekte können wiederum über den Wert eines Paares mit entsprechendem Schlüssel aktualisiert werden. Damit wird eine Zustandsverfolgung über die gesamte Laufzeit ermöglicht. Ein einfaches Beispiel hierfür ist die Mittelwertbildung über Werte der gesamten Laufzeit. Über die *updateStateByKey*-Transformation werden die beiden Zustände „Summe der bisherigen Werte" und „Anzahl an Batches seit Beginn" aktualisiert und nach jedem Batchintervall der Durchschnitt über *(Summe / Batchanzahl)* berechnet. Zwingend notwendig bei zustandsbehafteten Transformationen ist das Aktivieren von Checkpointing, welches in Kapitel 8.1.5 näher erläutert wird.

8.1.3 Ausgabe-Operationen

Die sogenannten *Output Operations* des Spark Streamings sind das Pendant zu den Aktionen im Batching-Bereich, mit dem Unterschied, dass hierbei die Operationen mehrmals, nämlich auf jedem Batch, aufgerufen werden. Ebenfalls wie beim Batching wird die Abarbeitung der Lebenslinien von Batch-RDDs nur angestoßen, wenn auf die Transformationen auch Output Operations folgen. Die bekanntesten Ausgabe-Operationen sind *print*, *saveAsTextFiles* und *saveAsHadoopFiles*. Möchte man auch im Streaming-Bereich auf bekannte RDD-Operationen zugreifen, bietet die *foreachRDD*-Methode Abhilfe: Durch sie kann auf die einzelnen Batch-RDDs eines DStreams zugegriffen und, wie von RDDs gewohnt, Transformationen und Aktionen auf diesen aufgerufen werden.

8.1.4 StreamingContext & SocketStream

Dieses Kapitel liefert eine Einführung in die Programmierung mit Spark Streaming. Einstiegspunkt stellt dabei die Klasse *JavaSparkStreamingContext* dar. Durch Übergabe eines SparkConf-Objektes und Angabe des Batchintervalls kann eine Instanz der Klasse angelegt werden, wie Codeblock 21 zeigt.

```
(1)   // Create a StreamingContext with a 1-second batch size
(2)   JavaStreamingContext jssc = new JavaStreamingContext(conf,
        Durations.seconds(1));
```

Codeblock 21: Anlegen einer JavaStreamingContext Instanz

Anschließend können die Datenempfänger festgelegt werden. Es werden, wie eingangs erwähnt, zahlreiche Quellen unterstützt. Eine Möglichkeit ist dabei das Aufsetzen eines Socket-Streams, welcher Nachrichten per TCP-Verbindung empfängt und diese als JavaDStream zur

Verarbeitung weitergibt. Codeblock 22 zeigt ein Beispiel auf, bei dem der Empfänger lokale Nachrichten an Port 7777 entgegennimmt.

```
(1)   // Create a DStream from all the local input on port 7777
(2)   JavaDStream<String> lines = jssc.socketTextStream("localhost", 7777);
```

Codeblock 22: Empfänger für Nachrichten per TCP-Socket festlegen

Auf diesem DStream können dann Transformationen aufgerufen werden. In Codeblock 23 werden die Daten eines DStreams auf Vorhandensein des Begriffs „error“ gefiltert und anschließend die übrigen Daten per Ausgabe-Operation *print* über die Konsole zurückgegeben.

```
(1)   // Filter DStream for lines with "error"
(2)   JavaDStream<String> errorLines = lines.filter(
        new Function<String, Boolean>() {
(3)       public Boolean call(String line) {
(4)         return line.contains("error");
(5)   }});
(6)   // Print the resulting DStream
(7)   errorLines.print();
```

Codeblock 23: Filtern von Daten eines DStreams und Ausgabe per print-Befehl

8.1.5 Checkpointing und Fehlertoleranz

Voraussetzung für eine korrekte 24/7-Arbeitsweise von Spark Streaming ist das Aktivieren von Zwischenspeicherungen. Beim sogenannten *Checkpointing* werden regelmäßig Ergebnisse festgeschrieben, sodass im Fehlerfall eine komplette Neuberechnung vermieden und lediglich die Daten seit der letzten Sicherung herangezogen werden müssen. Für den produktiven Einsatz wird ein Festschreiben auf zuverlässige Speicher wie S3-Buckets oder HDFS empfohlen, ein lokales Speichern wäre beim Ausfall des Knotens fatal und würde das Checkpointing überflüssig machen. Um die Zwischenspeicherung zu konfigurieren, kann die Methode *checkpoint* mit Pfadangabe zur Speicherung über den StreamingContext aufgerufen werden:

```
(1)   ssc.checkpoint("hdfs://hdfspath:port/path")
```

Codeblock 24: Aktivieren der Checkpointing-Funktion

Wie bereits erwähnt, kann ohne Checkpointing ein zustandsbehaftetes Streaming nicht ausgeführt werden und es kommt zu einer Fehlermeldung, falls innerhalb der Applikation eine entsprechende Transformation aufgerufen wird.

Driver-Fehlertoleranz

Um bei zustandsbehafteten Streaming-Applikationen von Zwischenspeicherungen zu profitieren, ist eine Anpassung der JavaStreamingContext-Initialisierung notwendig: Es muss bei Applikationsstart geprüft werden, ob bereits Sicherungen vorhanden sind, um bei einem ggf. anfallenden Driver-Neustart den darüber festgehaltenen Zustand automatisch wiederherstellen und anschließend bei diesem Zeitpunkt fortfahren zu können. Die JavaStreamingContext-Klasse hält dazu die Methode *getOrCreate* bereit, die es ermöglicht, einen vorkonfigurierten

StreamingContext zu verwenden, welcher das Checkpointing-Verzeichnis kennt. Sollten dort bereits Sicherungsdateien liegen, wird über diese direkt der damalige Stand wiederhergestellt, andernfalls ein neuer StreamingContext angelegt. Wichtig hierbei ist es, der anzulegenden Factory-Klasse die Informationen zum Checkpointing-Pfad bekannt zu machen. Codeblock 25 stellt eine solche angepasste StreamingContext-Initialisierung dar, welche ggf. eine Wiederherstellung durchführt.

```
(1)   JavaStreamingContextFactory fact = new JavaStreamingContextFactory(){
(2)   public JavaStreamingContext call() {
(3)     JavaSparkContext sc = new JavaSparkContext(conf);
(4)     // Create a StreamingContext with a 1 second batch size
(5)     JavaStreamingContext jssc = new JavaStreamingContext(sc, Durati
       ons.seconds(1));
(6)     jssc.checkpoint(checkpointDir);
(7)     return jssc;
(8)   }};
(9)   JavaStreamingContext jssc = JavaStreamingContext.getOrCreate(
       checkpointDir, fact);
```
Codeblock 25: Anlegen eines JavaStreamingContexts über Checkpoint-Verzeichnis

Um einen automatischen Driver-Neustart im Fehlerfall durchzuführen, kann das *spark-submit* Skript mithilfe des *supervise* Parameters aufgerufen werden (siehe Codeblock 26). Diese Einstellung ermöglicht eine unbeaufsichtigte und fehlertolerante Ausführung im Standalone-Cluster, bei der Driver und Worker im Fehlerfall automatisch neugestartet werden.

```
(1)   ./spark-submit --deploy-mode cluster --supervise
        --master spark://<masterurl>:7777 app.jar
```
Codeblock 26: Verwendung des supervise Parameters

Worker-Fehlertoleranz

Bei den Arbeitsinstanzen kommt die gleiche Wiederherstellungsstrategie zum Einsatz, wie sie auch bei Spark-RDDs zu finden ist: Daten von DStreams bzw. die darin enthaltenen RDDs werden im Fehlerfall durch Abarbeitung ihres Abstammungsgraphen wiederberechnet. Dazu werden im Cluster vorhandene Datenreplikationen und Sicherungspunkte genutzt. Auch die Datenempfänger sind mit fehlertolerantem Verhalten ausgestattet. Sollte ein Empfänger durch einen Fehler aussetzen, wird dieser umgehend durch den Driver neugestartet. Ob im Zeitintervall der Wiederherstellung Daten verloren gehen oder nicht, ist abhängig von der Datenquelle. Führt diese eine erneute Übertragung bei fehlender Empfangsbestätigung durch, ist auch hier ein lückenloses Entgegennehmen von Daten möglich.

Fehlertoleranz der Datenspeicherung

Eine Zuverlässigkeit ist beim Einsatz von HDFS oder S3 ebenfalls gegeben, sofern die standardmäßige Datenreplikation in diesen Speichern aktiviert ist. Diese Speicher eignen sich deshalb nicht nur zum Schreiben der regelmäßigen Zwischenspeicherungen, sondern auch

zum Logging von seither empfangenen Daten, mit deren Hilfe RDDs beim Clusterneustart komplett bis zum aktuellen Zeitpunkt wiederhergestellt werden können.

Exactly-Once Semantik

Die in diesem Kapitel beschriebenen Fehlertoleranz-Mechanismen führen letztendlich zum Einhalten der *Exactly-Once Semantik*, bei welcher das Ergebnis schlussendlich dem im fehlerfreien Durchlauf entspricht, die Daten also vollständig sind.

8.1.6 Optimierungen

Dieses Kapitel zeigt Möglichkeiten zur Optimierung im Spark Streaming auf. Durch einfache Einstellungen kann die Laufzeit des Micro-Batchings reduziert werden.

Parallelismus

Muss ein sehr datenreicher Stream entgegengenommen werden, kann ein einzelner Empfänger schnell zum Flaschenhals des Systems werden. In solch einem Fall empfiehlt es sich, mehrere Empfänger festzulegen und die daraus resultierenden DStreams durch die *union*-Transformation wieder zu einem Gesamt-Datenfluss zu vereinen. Ein weiterer Optimierungspunkt, der auch bei Spark Streaming eine Rolle spielt, stellt die Datenpartitionierung dar. Durch passende Aufteilung der Daten kann, wie in Kapitel 5.4 beispielhaft erläutert, eine ressourcennahe und an die Prozessoranzahl angepasste Berechnung durchgeführt werden.

Batchintervall- und Zeitfenster-Einstellungen

Durch Anpassung des Batchintervalls an die empfangene Datengröße kann ein Performancegewinn ermöglicht werden. Ein Intervall von rund 500ms stellt sich dabei als gutes Minimum dar [27]. Um eine optimale Intervallgröße zu bestimmen, sollten zunächst die Bearbeitungszeit der Batches von mehreren Intervallgrößen (~500ms bis ~10s) miteinander verglichen werden. Sollte diese bei Herabsetzen des Intervalls erheblich größer werden, ist das Intervall zu klein und das System kann die Daten nicht mehr schnell genug verarbeiten. Es sollte dann auf das größere Batchintervall ausgewichen werden. Auch bei zustandsbehafteten Berechnungen auf DStreams, bei welchen über Window Size und Slide Duration festgelegt wird, wie oft und mit welcher Datenmenge gerechnet wird, muss auf eine korrekte Festlegung der Werte geachtet werden, um das System nicht zu überfordern bzw. den Echtzeitbezug zu erhalten.

Optimierung der ReduceByWindow-Transformation

Die *reduceByWindow*-Transformation kann für Berechnungen über Zeitfenster eingesetzt werden. Statt einer ständigen Neuberechnung aller zum Zeitfenster gehörenden Daten, kann über sogenanntes *inkrementelles Reduzieren* der Berechnungsaufwand reduziert werden. Dabei werden neue Werte auf das vorherige Ergebnis dazu- und mithilfe der inversen Funk-

tion die nicht mehr relevanten Daten (außerhalb der Window Size) herausgerechnet. Abbildung 27 zeigt ein solches Beispiel für die Summenberechnung auf, hier kommt die Subtraktion als Inverse zum Einsatz.

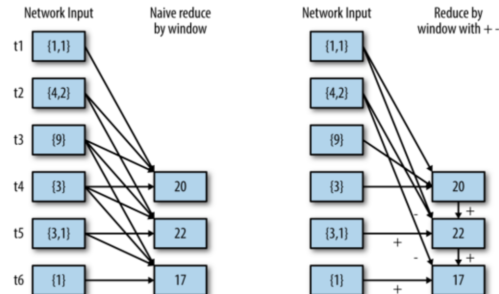

Abbildung 27: ReduceByWindow ohne und mit Verwendung der Gegenfunktion [27]

8.2 Maschinelles Lernen mit MLlib

Bei *MLlib* handelt es sich um die von Spark bereitgestellte Bibliothek für maschinelles Lernen. Sie stellt entsprechende Lernalgorithmen zur Verfügung. Da Spark Berechnungen auf Cluster verteilt durchführt, werden mit MLlib nur parallelisierte Lernverfahren angeboten, die ebenfalls von diesem Aufbau profitieren. Die Algorithmen sind eine Sammlung von Funktionen, welche auf RDDs mit Trainingsdaten ausgeführt werden, um ein gewünschtes Modell zu lernen und neu ankommende Daten zukünftig über dieses automatisch zuordnen zu können. Auch Test- und Validierungsmethoden gehören zum Funktionsumfang von MLlib.

Um Daten, wie aus dem Bereich des maschinellen Lernens gewohnt, als Feature-Vektoren darstellen zu können, wird über MLlib der Datentyp *Vector* eingeführt. Um einem Vektor im Voraus eine Klasse zuweisen zu können, was bei überwachtem Lernen benötigt wird, gibt es zusätzlich die Klasse *LabeledPoint*. Auch Datentypen zur Repräsentation von Matrizen, Ratings und Modellen sind vorhanden.[48]

Mithilfe eines Beispiels zur Klassifizierung von Texten, wie dem Einteilen von Nachrichten und Spam und Nicht-Spam, soll die Arbeitsweise mit MLlib verdeutlicht werden. Folgende Schritte würden zur Klassifizierung durchgeführt werden:

> 1. *Beginne mit einem RDD aus Strings, welches die Nachrichten repräsentiert*
> 2. *Führe einen der MLlib-Algorithmen zur Merkmalsextraktion (Feature Extraction) aus um den Text in numerische Features (passend für die Lernalgorithmen) umzuwandeln; dies gibt ein RDD aus Vektoren zurück*

[48] Hinweise zur Verwendung der Datentypen unter https://spark.apache.org/docs/1.2.2/mllib-data-types.html bzw. https://spark.apache.org/docs/1.2.1/mllib-guide.html

> 3. *Rufe einen Klassifizierungs-Algorithmus (z. B. logistische Regression) auf diesem Vektor-RDD auf; dies gibt ein Modell-Objekt zurück, das für die Klassifizierung neuer Punkte verwendet werden kann*
> 4. *Evaluiere das Modell über ein Test-Datensatz mithilfe einer Evaluierungsfunktion von MLlib.*
>
> *- Übersetzt aus [27]*

Eine graphische Repräsentation der typischen Schritte, die beim maschinellen Lernen durchgeführt werden, zeigt Abbildung 28. Der Quellcode einer solchen Beispiel-Applikation zur Spam-Klassifizierung findet sich im Anhang unter Codeblock 50.

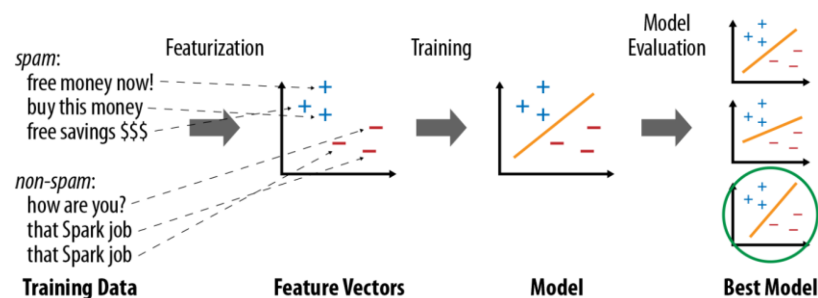

Abbildung 28: Typische Schritte beim maschinellen Lernen [27]

8.2.1 Datenaufbereitung

Da sehr viele Lernalgorithmen nur mit numerischen Werten arbeiten, ist es hierbei notwendig, nicht-numerische Attribute entsprechend umzuwandeln. Ein Weg ist es, das Vorkommen einer Ausprägung im aktuellen Dokument zu zählen und als Vektor festzuhalten. Im Beispiel der Spam-Klassifizierung entspricht ein solches Dokument genau einer Nachricht. Als Features werden hierbei die einzelnen Wörter angesehen, aus denen sich die gesamte Nachrichtenbasis zusammensetzt. Um aus diesen Wörtern numerische Feature-Vektoren zu gewinnen, führt man eine Häufigkeitszählung der einzelnen Wörter in jedem Dokument durch und hält diese Teilsummen für jede Nachricht in einem eigenen Feature-Vektor fest. Dieses Vorgehen nennt man *Term-Frequency*. Abbildung 29 zeigt ein Beispiel zum Mapping von Wörtern bzw. Sätzen auf Vektoren. Da es in einer großen Datenbasis hunderttausende von verschiedenen Wörtern geben kann, muss hier die Vektorgröße (und damit der Berechnungsraum) eingeschränkt werden. Dies kann über Hashing der Wörter geschehen, MLlib stellt hierfür die Klasse *HashingTF* zur Verfügung, mit der eine beliebige Datenmenge auf einen n-dimensionalen Vektor reduziert werden kann. Die Ziel-Raumgröße wird dabei im Konstruktor der HashingTF-Klasse übergeben und somit die Hashingfunktion entsprechend beeinflusst.

Satz 1 (repräsentiert durch v_1): *ich fand das wetter gestern sehr warm.*
Satz 2 (repräsentiert durch v_2): *er fand das wetter gestern relativ kalt und ich heiß.*

$v_{Wörter}$	v_1	v_2
ich	1	1
er	0	1
fand	1	1
das	1	1
wetter	1	1
gestern	1	1
sehr	1	0
relativ	0	1
warm	1	0
kalt	0	1
und	0	1
heiß	0	1

Abbildung 29: Mapping von Wörtern auf Vektoren

Das Zählen der Häufigkeiten jedes Wortes per TF (Term-Frequency) gibt dessen Wichtigkeit im aktuellen Dokument an. Um eine Verzerrung zu verhindern, bietet sich eine Normierung der Vektoren an. Besteht die Datenbasis aus mehreren Dokumenten ist es oft von Vorteil, die Bedeutung eines Wortes in dieser Gesamtmenge zu ermitteln. Die sogenannte *IDF (Inverse Document Frequency)* gibt die allgemeine Bedeutung eines Wortes innerhalb der gesamten Dokumentenmenge an und lässt sich über Formel 3 berechnen.

$$idf_i = \log \frac{N}{n_i}$$

N: *Anzahl Dokumente*
n_i: *Anzahl der Dokumente, die Wort i enthalten*

Formel 3: Berechnung der Inverse Document Frequency (IDF)

Multipliziert man die beiden zuvor erläuterten Größen, erhält man die Gewichtung (und somit die Relevanz) eines Wortes i im Dokument j. Diese Multiplikation nennt man entsprechend der Schrittfolge *TF-IDF* und ist ein bekanntes Verfahren zur Wichtigkeitsermittlung in Dokumentenbasen. Spark's MLlib stellt hierfür neben der oben erwähnten *HashingTF*-Klasse auch die Klasse *IDF* zur Verfügung, über deren Kombination die TF-IDF berechnet werden kann.

8.2.2 Statistiken

MLlib liefert eine Klasse, mit deren Hilfe Statistiken zu Vektor-RDDs erstellt werden können. Die Klasse *Statistics* liefert statische Methoden, denen die Ziel-RDDs übergeben werden können. Um beispielsweise spaltenbasierte Statistiken zu einem RDD aus Vektoren zu erhalten, kann hier die Methode *colStats* verwendet werden. Das zurückgegebene *MultivariateStatisticalSummary*-Objekt bietet dabei Zugriff auf interessante Statistikwerte wie Minimum/Maxi-

mum, Mittelwert, Norm oder Varianz. Auch Methoden zur Korrelationsbestimmung oder Berechnung der Anpassungsgüte eines RDDs sind durch die Methoden *corr* bzw. *chiSqTest* gegeben.[49]

8.2.3 Funktionsumfang von MLlib

In diesem Kapitel werden die aktuell in der MLlib-Bibliothek enthaltenen Funktionen gelistet und grob vorgestellt.[50] Wie bereits einleitend erwähnt, sind in Spark nur solche Algorithmen zum maschinellen Lernen implementiert, die von einer verteilten Abarbeitung profitieren und somit eine optimale Ausnutzung der verfügbaren Ressourcen garantiert werden kann. Allgemein lassen sich die Verfahren des maschinellen Lernens in zwei Bereiche unterteilen: überwachtes und unüberwachtes Lernen. Diese beiden Typen von Lernalgorithmen werden nachfolgend ebenso erläutert, wie auch die Möglichkeiten zur Evaluation.

Überwachtes Lernen

Algorithmen des überwachten Lernens benötigen eine Trainingsmenge, die bereits im Voraus vom Anwender klassifiziert wurde. Die Eigenschaften dieser Menge, repräsentiert durch Attribute und deren Werte, werden beim Lernen untersucht und ein Zusammenhang zwischen ihnen und den vorgegebenen Klassen hergestellt. Neu ankommende, nicht-klassifizierte Daten, werden dann anhand der gelernten Klassifizierungsfunktion automatisch vom System eingeteilt. Mithilfe einer weiteren Datenmenge, der Testmenge, wird die Arbeitsweise des Modells auf Korrektheit überprüft. Folgende überwachte Lernalgorithmen sind in Spark's MLlib implementiert:

Überwachtes Lernen	
Klassifikation	Entscheidungsbäume
	Naive Bayes
	Support Vector Machines
Regression	Lineare Regression
	Logistische Regression

Tabelle 5: Auflistung der in MLlib enthaltenen überwachten Lernverfahren

[49] Nähere Informationen hierzu unter https://spark.apache.org/docs/1.2.2/api/java/org/apache/spark/mllib/stat/Statistics.html

[50] Stand: Spark-Version 1.2.2. Der Funktionsumfang von MLlib ist seit Version 1.0 von Spark sehr stark angestiegen, es ist deshalb sehr gut möglich, dass neuere Versionen bereits über mehr Lernalgorithmen verfügen.

Unüberwachtes Lernen

Beim unüberwachten Lernen findet, im Gegensatz zum überwachten Lernen, eine Gruppierung der Daten ohne Vorgabe durch den Anwender statt. Auch die zuvor erwähnte Testmenge kommt hier nicht zum Einsatz. Stattdessen werden Attribute auf Ähnlichkeit untersucht und hierüber eigenständig Einteilungen in Gruppen durchgeführt. MLlib liefert folgende Algorithmen zum unüberwachten Lernen:

Unüberwachtes Lernen	
Clustering	K-Means
Kollaboratives Filtern	ALS (Alternierende kleinste Quadrate)

Tabelle 6: Auflistung der in MLlib enthaltenen unüberwachten Lernverfahren

Evaluation von Modellen

Um die Güte von Modellen zu überprüfen, bietet MLlib ebenfalls Methoden zur Evaluation. Über sie können Modelle verglichen und somit das Passendste festgelegt werden. Auch eine Überanpassung des Modells an die Trainingsmenge kann durch die Evaluierung mit Validierungsdaten erkannt werden. Mithilfe der Klassen *BinaryClassificationMetrics* und *MulticlassMetrics* kann eine Evaluation entsprechender Modelle durchgeführt werden. Mithilfe eines Validierungs-Datensatzes, der ungleich der Trainings- und Testmenge sein, und rund 20% der Gesamtdatenmenge enthalten sollte, kann ein Modell geprüft werden. Durch Berechnung der *Genauigkeit (Precision)* und der *Trefferquote (Recall)* werden solche Aussagen zu einem Modell ermöglicht. Auch die Rückgabe einer *ROC (Receiver Operating Characteristic)* Kurve, welche den Zusammenhang zwischen Effizienz und der Fehlerrate aufzeigt, wird durch entsprechenden Methodenaufruf ermöglicht.

8.3 Datenvisualisierung mit Apache Zeppelin

Im Rahmen dieser Arbeit wurde ebenfalls nach Möglichkeiten der Datenvisualisierung gesucht, da Spark selbst, außer der Ausgabe auf Konsole, aktuell keine Möglichkeit bietet, Daten darzustellen.[51] Besonders bei großen Datenmengen und im Bereich des maschinellen Lernens kann durch eine visuelle Darstellung von Daten die Arbeit des Anwenders vereinfacht werden, da ihm hierdurch auch eine bessere Kontrolle ermöglicht wird. Eine solche Möglichkeit zur Datendarstellung bietet das Apache Incubator-Projekt Zeppelin.

[51] Stand Version 1.2.2

8.3.1 Allgemeines zu Spark Zeppelin

Bei Apache Zeppelin handelt es sich um ein Web-basiertes Arbeitstool, welches interaktive Datenanalysen ermöglicht. Durch vorinstallierte Interpreter wird der Zugriff auf verschiedene Kontexte und damit die Erstellung von datengetriebenen Dokumenten für diese ermöglicht. Zur Liste der unterstützten Plattformen gehört auch Spark. Über den Spark-Interpreter stehen dem Anwender über Injection zahlreiche Programmierschnittstellen zur Verfügung, darunter Spark's Scala- und Python-API, sowie ein Hive-Interface und SparkSQL-Modul für den Zugriff auf strukturierte Daten. Über eine ebenfalls integrierte Pivot-Diagrammbibliothek können die über SQL-artige Befehle selektierten Daten dargestellt werden.

8.3.2 Installation von Apache Zeppelin

Apache Zeppelin kann auf dem Masterknoten eines Spark-Clusters installiert werden, um auf einfachste Weise Zugriff auf dessen Kontext zu bekommen. Welche Schritte zur Installation durchgeführt werden müssen, wird in Anhang A.11 gezeigt. Bei Beachtung der Hinweise ist die Weboberfläche von Zeppelin nach erfolgreicher Installation über die URL *http://<master>:8082* erreichbar.

8.3.3 Weboberfläche und Verwendung von Zeppelin

Die Weboberfläche ist für den Anwender sowohl Schnittstelle für die Einstellungen als auch für das Erstellen sogenannter *Notebooks*, welche einzelne Anwendungen repräsentieren. Eine Anwendung kann dabei aus mehreren Paragraphen bestehen, in welchen wiederum unterschiedliche Interpreter eingebunden und verwendet werden können (je Paragraph genau ein Interpreter). Die Paragraphen können dabei einzeln oder als Gesamtanwendung automatisch nacheinander ausgeführt werden. Durch die integrierten Pivot-Bibliotheken lassen sich die per SQL-Interpreter selektierten Daten visualisieren. Abbildung 30 zeigt die Oberfläche beim Öffnen eines Notebooks. Einstellungen zu den Interpretern und zur Ausführung des Notebooks können unter (1) gemacht werden. (2) und (3) sind jeweils Paragraphen des Notebooks, über welche Daten mithilfe von Spark- bzw. SQL-Interpreter aufbereitet und selektiert werden. Im SQL-Paragraph wird zusätzlich eine Zeichnungsfläche (4) angezeigt, in der die zu visualisierenden Daten ausgewählt werden können, das Diagramm wird automatisch aktualisiert.

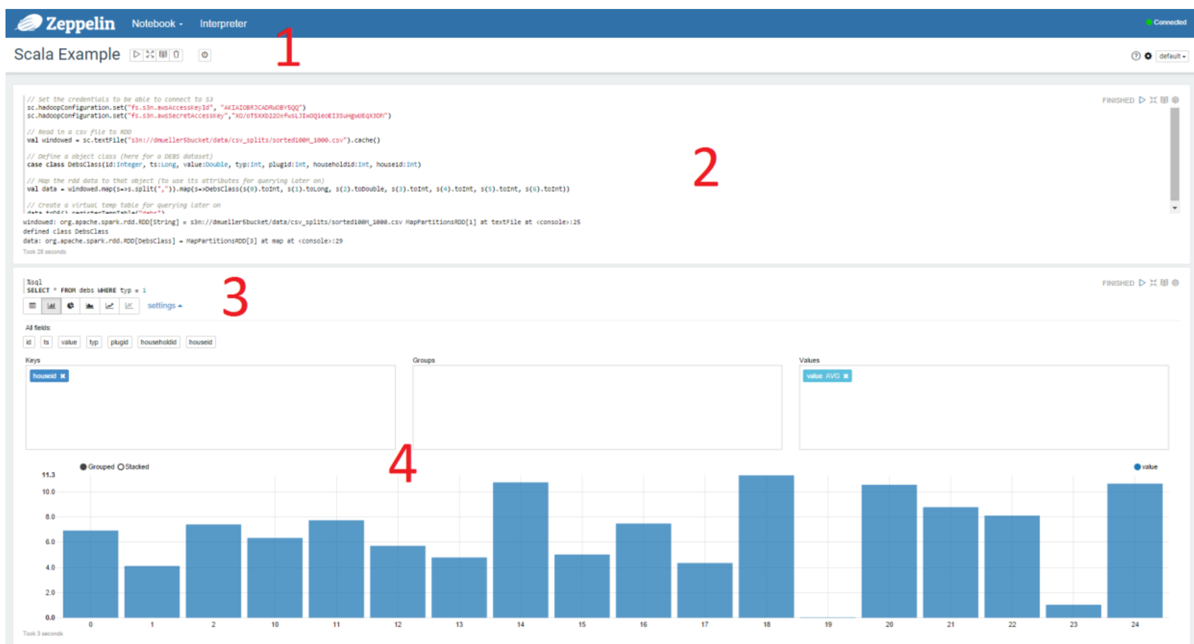

Abbildung 30: Notebook-Ansicht in Zeppelin

Die Erläuterung und der Quellcode des in Abbildung 30 dargestellten Notebooks ist in Anhang A.11 zu finden.

9 Streaming Data Analyse am Beispiel von Energiedaten

Dieses Kapitel beschäftigt sich mit der Analyse von Datenströmen in Echtzeit. Durch die Modellierung von Anwendungen soll die Einsatzfähigkeit von Spark für solche Aufgabenbereiche validiert werden. Es sollen Daten der DEBS Grand Challenge 2014 aufbereitet und untersucht werden. In den folgenden Kapiteln wird das Vorgehen dabei schrittweise aufgezeigt. Bei der Umsetzung dieser Aufgabe wurde nach dem in Kapitel 2.1 vorgestellten CRISP-DM Modell vorgegangen. Die einzelnen Schritte zur Umsetzung der Aufgabe wurden in die darin vorgestellten sechs Phasen eingeteilt. Entsprechend wurden ebenfalls die nachfolgenden Kapitel untergliedert. Es folgt zunächst eine Erläuterung zur Aufgabenstellung und Theorie zur DEBS Grand Challenge, bevor anschließend die Datenaufbereitung und Umsetzung beschrieben wird.

9.1 Business Understanding

Beim Aufbau eines Data Mining Prozesses ist es wichtig, sich im Voraus mit der Fachdomäne zu beschäftigen, in der die Daten anfallen und ausgewertet werden sollen. Auch im CRISP-DM Vorgehensmodell wird das grundlegende Verständnis zum Vorhaben als erste Phase definiert. In diesem Schritt machen sich die an der Umsetzung des Prozesses beteiligten Personen Gedanken über die Aufgabenstellung und das Ziel des Projektes. Da sich dieses Beispielprojekt mit der Analyse von DEBS Grand Challenge Daten beschäftigt, wird nachfolgend dieser Wettbewerb vorgestellt. Ebenso wird die Aufgabenstellung, welche mithilfe von Spark umgesetzt werden soll, näher erläutert.

9.1.1 Über die DEBS Challenge

Die DEBS (Distributed Event-Based Systems) Grand Challenge ist ein jährlicher Wettbewerb im Bereich Real-Time Big Data Mining, bei dem sich die Teilnehmer um Aufgaben zur Datenverarbeitung in verteilten und ereignisgetriebenen Systemen annehmen. Sie forschen an der Verarbeitung und Wissensgewinnung durch die zu Beginn der Challenge ausgehändigten Rohdaten. Es werden Lösungsmöglichkeiten erarbeitet und schriftlich festgehalten. Aus den eingereichten Dokumenten wird durch eine Jury von weltweiten Forschern und Entwicklern ein Gewinner ermittelt. Die aktuellen Berichte behandeln die Aufgabe der DEBS Grand Challenge 2014.[52] Das Thema dieses Wettbewerbs ist die Verarbeitung von Energiedaten, die durch intelligente Stromzähler in privaten Haushalten gemessen und zusammengetragen wurden. Es wurde einen Monat (1. – 30. September 2013) lang der Stromverbrauch von 40

[52] Stand 06.05.2015, die nächste DEBS Konferenz findet erst Ende Juni 2015 statt. Nähere Informationen unter [36] bzw. http://www.debs2015.org/

Häusern mit teilweise mehreren Haushalten und insgesamt rund 2000 Steckdosen-Stromzähler aufgezeichnet. Über diesen Zeitraum wurden mehr als vier Milliarden Datensätze zusammengetragen, die dann für die Analysen zur Verfügung standen [36, 37]. Der Aufbau der Daten wird im Schritt *Data Understanding* (Kapitel 9.2) näher erläutert.

9.1.2 Aufgabenstellung des Projekts

Die Daten zur oben vorgestellten DEBS Grand Challenge 2014 liegen zum Startzeitpunkt des Projekts in Form einer CSV-Datei vor. Diese CSV muss in den nächsten Schritten entsprechend auf nützliche Daten untersucht und bereinigt werden. Anschließend sollen diese bereinigten Daten über einen Sender per TCP-Verbindung an ein Spark-Cluster geschickt werden. Dort sollen die Daten mithilfe der Streaming-Funktionalitäten entgegengenommen und weiterverarbeitet werden. Der Sender durchläuft dabei die CSV-Datei mit den DEBS-Daten und simuliert damit das regelmäßige und zeitbasierte Senden der Stromzählerinformationen im Wettbewerb. Die genaue Untersuchung und Verarbeitung der Daten lässt sich in zwei Aufgabenteile untergliedern:

1. Sender (Java): Zeitbasiertes Auslesen der CSV-Datei und Senden per TCP-Sockets an das Spark-Cluster. Unnötige Daten sollen vor dem Senden aussortiert werden, um den Netzwerkverkehr zu mindern. Das Sendeintervall soll hierbei dem der DEBS Grand Challenge entsprechen.
2. Empfänger (Spark, Java): Die vom Sender per TCP-Sockets gesendeten Daten sollen per Streaming-Bibliothek empfangen und auf Ausreißer untersucht werden. Diese Untersuchung soll in Echtzeit passieren, sodass der Anwender umgehend (nach Ablauf eines passenden Zeitfensters) Rückmeldung über die auffälligen Werte bekommt.

9.2 Data Understanding: Aufbau der Daten

Ein aufgezeichneter Datensatz der DEBS Grand Challenge hat folgenden Aufbau:

```
(1)   <id>,<ts>,<val>,<type>,<plug_id>,<household_id>,<house_id>
```

Codeblock 27: Aufbau eines Datensatzes bei der DEBS Grand Challenge 2014 [36]

Jeder Eintrag enthält eine eindeutige ID (id). Eine Messung selbst besteht aus einem Zeitstempel (Zeitstempel der Messung in Sekunden, ts), dem gemessenen Wert (Fließkommazahl, val) und der Art (type) der Messung. Über das <type>-Flag wird unterschieden, ob <val> den aktuellen Stromverbrauch (type=1, Einheit Watt) oder den kumulierten Gesamtwert des Zählers (type=0, Einheit kWh) enthält. Jeder Steckdosen-Stromzähler wird *innerhalb eines Haushaltes*, und jeder Haushalt *innerhalb eines Hauses*, eindeutig identifiziert. Die Kennung eines Hauses ist dagegen in der gesamten Datenmenge eindeutig. Das Sendeintervall eines Stromzählers liegt im fehlerfreien Fall bei zwei Sekunden. In dieser Zeit wurden pro Gerät

zwei Informationen gespeichert: der aktuelle Verbrauch (type = 1) und die aktuelle Gesamtsumme des Zählers (type = 0). Aufgrund von Fehlern in Stromzählern, Netzwerk und/oder Datenspeicherung kommt es jedoch häufig vor, dass keine Messwerte für einen Zeitstempel existieren. Im Gegensatz dazu gibt es auch Zeitpunkte, für welche es mehrere Einträge wie Duplikate oder fehlerhafte Messwerte gibt. In den folgenden Schritten gilt es, diese Umstände zu beachten.

Die Daten sind in einer CSV-Datei gespeichert, eine Zeile des Inhalts entspricht dabei dem oben beschriebenen Messpunkt eines Typs. Die gesamte Datei hat eine Größe von 134,4 GB. Mithilfe einer Spark-Applikation zur Untersuchung der Daten wurden folgende Eigenschaften des Datensatzes herausgefunden:

- Anzahl Datensätze in der CSV: 4.055.508.721
- Gesamtzahl der Stromzähler (Plugs): 1956
- Anzahl Haushalte: 289

Der Quellcode der Anwendung zur Untersuchung der CSV findet sich in Codeblock 53 (Anhang B.1).

Die theoretische Anzahl an Messungen im fehlerfreien Fall berechnet sich wie folgt:

$$\begin{aligned} Daten_{theo.} &= ([Daten\ des\ Typs\ 0\ je\ Sekunde] + [Daten\ des\ Typs\ 1\ je\ Sekunde]) \\ &\quad \cdot [30\ Tage\ in\ Sekunden] \cdot [Anzahl\ Plugs] \\ &= (0{,}5 + 0{,}5) \cdot (60 \cdot 60 \cdot 24 \cdot 30) \cdot 1956 \\ &= 5.069.952.000 \end{aligned}$$

Formel 4: Berechnung der theoretischen Anzahl an DEBS-Daten

Damit entspricht die tatsächliche Anzahl Zeilen nur rund 80% der theoretischen, was auf die damals fehlerbehaftete Erhebung, Übertragung bzw. Speicherung der Messwerte zurückzuführen ist.

9.3 Data Preparation: Bereinigen, Senden und Empfangen

Nach der Durchführung der zuvor durchlaufenen Schritte sind der Aufbau und die Aufgabenstellung insofern bekannt, dass in dieser Phase nun mit der Vorbereitung der Daten fortgefahren werden kann. Dazu zählt sowohl die Vorselektion, als auch das anschließende Senden und Bereinigen der Daten. Als grobes Architektur-Schaubild dient Abbildung 31: Es gibt eine Sendeinstanz, die die Daten (CSV) der DEBS Grand Challenge ausliest und mithilfe von Sockets per TCP-Verbindung an das Empfänger-Cluster sendet. Auf der Empfängerseite kommt

Spark und dessen Streaming Bibliothek zum Entgegennehmen und Verarbeiten der Daten zum Einsatz. Hier findet nach der Datenaufbereitung die Echtzeit-Ausreißererkennung statt.

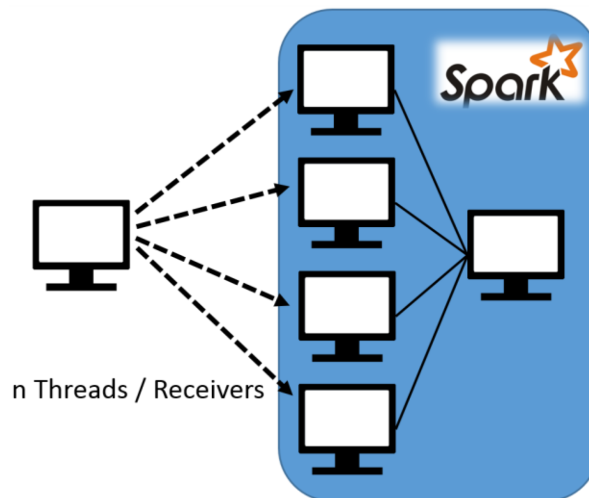

Abbildung 31: Grober Aufbau der Prototyp-Umgebung

Dieses Kapitel ist entsprechend der Aufteilung in Sender und Empfänger eingeteilt: Zunächst werden die Arbeitsschritte auf Senderseite erläutert, ehe danach die Aufgaben auf Empfängerseite folgen.

9.3.1 Senderseite: Java Server-Thread

Da sich die Ausreißererkennung lediglich auf die Werte des aktuellen Verbrauchs stützt, können hier die kumulierten Werte der einzelnen Stromzähler (Messpunkte des Typs 0) ignoriert werden. Es müssen deshalb nur Daten vom Typ 1 (aktueller Verbrauch in Watt) an das Cluster gesendet werden, was die zu sendende Datenmenge um rund die Hälfte reduziert. Die Sendegeschwindigkeit soll hierbei dem Intervall der Datenerhebung in der DEBS Grand Challenge entsprechen. Da das Sendeintervall, wie im Kapitel *Data Understanding* bereits erwähnt, bei zwei Sekunden liegt, muss der Sender die Daten ebenfalls in sekundengenauer Auflösung bereitstellen und übertragen können. Das Auslesen und Senden der Daten muss dabei so angepasst werden, dass die DEBS-Daten eines Timestamps (Auflösung 1 Sekunde) im Prototyp ebenfalls in einer Sekunde selektiert und bereitgestellt werden, und erst nach Ablauf dieses Zeitintervalls mit dem Lesen der Daten des nächsten Zeitpunkts fortgefahren wird. Aufgrund des zeitlich geordneten Datensatzes und der überschaubaren Datenmenge, die zu einem Zeitpunkt existiert (~1956), stellt diese Bedingung für den Sender keine Hürde dar. Durch zeilenweises Auslesen und Pausieren bei Erreichen des nächsten Zeitpunkts innerhalb der CSV-Datei, kann eine einfache Datei-Traversierung diese Aufgabe erledigen. Das Auslesen wird mithilfe eines *BufferedReaders* erledigt, welcher die im S3 gespeicherte CSV mithilfe des AWS-SDK für Java aufruft, zeilenweise durchläuft und die Datensätze als einzelne Nachrichten abwechselnd über zwei TCP-Sockets bereitstellt.

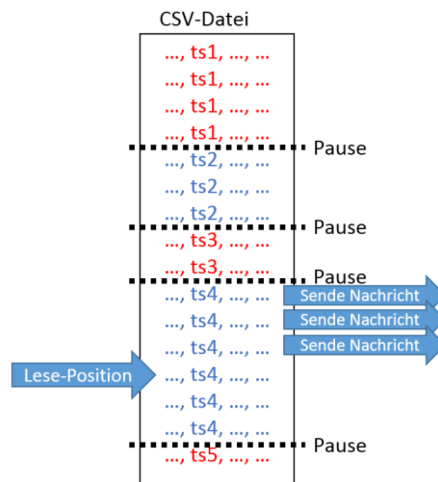

Abbildung 32: Durchlaufen der CSV-Datei und senden der Zeilen als TCP-Nachricht

Zudem wird die benötigte Zeit zum Auslesen und Senden der Daten, die zu einem Timestamp gehören, festgehalten, um daraus anschließend die Pausenzeit bis zum Fortfahren der Dateitraversierung ausrechnen zu können. Über diese Pausendauer wird die zeitlich korrekte Arbeitsweise des Senders sichergestellt. Abbildung 32 verdeutlicht graphisch die Vorgehensweise bei der Traversierung der CSV-Datei durch die Sender-Instanz, deren Logik durch eine Java-Applikation umgesetzt wird. Sie erzeugt innerhalb eines Threads zwei TCP-Verbindungen via Sockets und sendet über diese abwechselnd (Nachricht 1 = Socket 1, Nachricht 2 = Socket 2, Nachricht 3 = Socket 1...) die Datensätze zeilenweise als Nachrichten. Die Server-Applikation lässt sich über folgenden Befehl starten:

```
(1)   java -cp ~/thesis.jar prototype.SendingServer
```

Codeblock 28: Starten des Servers zum Auslesen und Senden der DEBS-Daten

Der Quellcode der Server-Applikation ist im Anhang unter Codeblock 54 zu finden.

9.3.2 Empfängerseite: Apache Spark

Auf der Empfängerseite läuft ein Apache Spark Cluster, welches mithilfe der per Streaming-Bibliothek zur Verfügung stehenden Text-Socket Klasse einen verbindungsorientierten Datenaustausch mit der Senderseite eingeht. Den Rahmen der Empfänger-Applikation zeigt Codeblock 29: Nach der Spark-Konfiguration, dem Festlegen von Parametern und Generierung des *JavaStreamingContexts* (Zeile 3-14) werden hier über die *socketTextStream* Methode des *JavaStreamingContext* zwei Instanzen des sogenannten *JavaReceiverInput-DStreams* (Zeile 20-23) angelegt. Da mehrere solcher Empfängerinstanzen erstellt werden, werden deren empfangenen Daten über die *union* Transformation zu einem großen *JavaDStream* zusammengefügt (Zeile 26-31). Mit diesem DStream kann anschließend weitergearbeitet werden. Nach Festlegung der Transformations- und Aktionsfolgen muss die Ausführung der Streaming-Anwendung per *start* Methode angestoßen werden (Zeile 37-39).

```
(1)   public static void main(String[] args) {
(2)     // Generate SparkConf with custom settings
(3)     final SparkConf conf = new SparkConf();
(4)     conf.set("spark.driver.memory", "5g");
(5)     conf.set("spark.executor.memory", "5g");
(6)
(7)     // Set parameters
(8)     int receiverInstances = 2; // Could be more or less receivers
(9)     int batchIntervalSec = Integer.valueOf(args[1]);
(10)    int windowSize1hSec = 60 * 60;
(11)    int slideDurationSec = batchIntervalSec;
(12)
(13)    // Generate StreamingContext from config and by batch-interval
(14)    final JavaStreamingContext streamingCtx = new JavaStreamingContext
          (conf, Durations.seconds(batchIntervalSec));
(15)
(16)    // Set checkpointing
(17)    streamingCtx.checkpoint("...");
(18)
(19)    // Generate text receiver sockets
(20)    List<JavaDStream<String>> receivedDStreams = new ArrayList
          <JavaDStream<String>>();
(21)    for (int i = 0; i < receiverInstances; i++) {
(22)        receivedDStreams.add(streamingCtx.socketTextStream(args[0],
              7777 + i));
(23)    }
(24)
(25)    // Union the socket data, if more than one receiver
(26)    JavaDStream<String> inputDStream;
(27)    if (receivedDStreams.size() > 1) {
(28)        inputDStream = streamingCtx.union(receivedDStreams.get(0),
              receivedDStreams.subList(1, receivedDStreams.size()));
(29)    } else {
(30)        inputDStream = receivedDStreams.get(0);
(31)    }
(32)
(33)    // Logic of the application - Working with the inputDStream
(34)    // ...
(35)
(36)    // Start the streaming application
(37)    streamingCtx.start();
(38)    streamingCtx.awaitTermination();
(39)    streamingCtx.close();
(40)  }
```

Codeblock 29: Einrichten der Streaming-Empfänger

Die gesamte Empfänger-Anwendung lässt sich auf einem Spark Standalone Cluster mit folgendem Befehl starten:

```
(1)   ./spark-submit --class prototype.<ClassToStart> --master
   spark://<master>:7077 ~/thesis.jar <server-address> <batch-interval-sec>
```

Codeblock 30: Starten der Spark-Applikation zur Ausreißer-Erkennung

Nachdem die empfangenen Daten in Form eines *JavaDStreams* vorliegen, gilt es diese weiter für die folgende Ausreißer-Erkennung zu transformieren. Dazu gehört zunächst die Prüfung auf Duplikate.

Duplikate entfernen

Wie bereits einleitend in Kapitel 9.2 kurz beschrieben, kommt es vor, dass in den Daten Messwerte fehlen bzw. doppelt vorkommen. Stromzähler, die zu einem Zeitpunkt keine Messwerte enthalten, spielen bei späteren Berechnungen keine Rolle. Die Berechnungen finden dann einfach ohne jene Messwerte statt. Da rund 80% der theoretischen Anzahl an Messwerten in der Datenbasis vorhanden sind, kann davon ausgegangen werden, dass pro Sekunde noch die Werte von rund 1560 der insgesamt 1956 vorhandenen Plugs für Berechnungen vorhanden sind. Damit relativiert sich das Fehlen einzelner Messpunkte auf ein Minimum und beeinflusst die Rechenergebnisse kaum merklich. Viel wichtiger dagegen ist das Ausschließen von Duplikaten, denn das Vorhandensein mehrerer Messwerte zu einem Stromzähler und Zeitpunkt würde bei plugbasierten Berechnungen und Vergleichen Probleme bereiten. Entsprechend muss hier über passende Datenumformung und Transformationsaufrufe dafür gesorgt werden, dass diese Duplikate aussortiert werden. Dazu muss der eingangs generierte *JavaDStream* zunächst per *mapToPair* Transformation auf die Form *JavaPairDStream<String, Float>* gebracht werden. Als Schlüssel dient dabei eine Kombination aus eindeutiger Stromzähleridentifikation und Zeitstempel. Als Wert eines Paares reicht dann der verbleibende Verbrauchswert aus. Eine eindeutige Plug-Kennung kann durch Zusammenfassen der Haus-, Haushalt- und Stromzähler-IDs erreicht werden. Sind die Daten erst in diese Form gebracht, kann durch Aufruf der *distinct* Transformation das Entfernen doppelter Einträge pluggenau durchgeführt werden. Diese Operation kann durch Aufruf der *transformToPair* Transformation auf dem zuvor generierten *JavaPairDStream* innerhalb der zu überschreibenden *call*-Funktion ausgeführt werden. Codeblock 31 zeigt den Aufruf (Zeile 1-4) und die Definition (ab Zeile 6) der beiden Funktionen zum Umwandeln in ein *PairDStream* bzw. Entfernen von Duplikaten:

```
(1)   // Map to pair: (house.hh.plug.TS, val)
(2)   JavaPairDStream<String, Float> debsPairDStream = inputDStream
       .mapToPair(mapInputDStreamToPairDStreamFunction);
(3)   // Remove duplicates
(4)   JavaPairDStream<String, Float> debsPairDStreamWithoutDuplicates =
       debsPairDStream.transformToPair(removeDuplicatesFunction);
(5)
(6)   private static PairFunction<String, String, Float> mapInputDStream-
       ToPairDStreamFunction = new PairFunction<String, String, Float>() {
(7)     public Tuple2<String, Float> call(String row) throws Exception {
(8)       String[] data = row.split(",");
(9)       long ts = Long.valueOf(data[1]);
(10)      Float val = Float.valueOf(data[2]);
(11)      int plug_id = Integer.valueOf(data[4]);
(12)      int household_id = Integer.valueOf(data[5]);
(13)      int house_id = Integer.valueOf(data[6]);
(14)      String key = house_id+ "."+ household_id+ "."+ plug_id+ "."+ ts;
(15)      return new Tuple2<String, Float>(key, val);
(16)    }
(17)  };
(18)
```

```
(19)  private static Function<JavaPairRDD<String,Float>,
       JavaPairRDD<String,Float>> removeDuplicatesFunction =
       new Function<JavaPairRDD<String,Float>,JavaPairRDD<String,Float>>(){
(20)    public JavaPairRDD<String,Float> call(JavaPairRDD<String,Float> v1)
         throws Exception {
(21)      return v1.distinct();
(22)    }
(23)  };
```

Codeblock 31: Umwandeln des Streams in ein JavaPairDStream und Duplikatsprüfung

Aussortieren von inaktiven Stromzählern

Neben Duplikaten gibt es noch eine weitere Art von Messpunkten, die die Ausreißererkennung auf Verbrauchsebene negativ beeinträchtigen: Nullwerte. Mit ihnen sind Messpunkte gemeint, bei denen die Stromzähler einen Verbrauch von 0 Watt übermittelt haben. Dies ist bei Plugs der Fall, an welchen zum Aufzeichnungszeitpunkt keine Verbraucher angeschlossen waren. Da die Ausreißererkennung auf den Verbrauch der einzelnen bzw. aller Plugs aufbaut, ist es hier wichtig, solche Nullwerte auszufiltern, da sie nicht bei der Berechnung des aktuellen Verbrauchs miteinbezogen werden dürfen. Der Verbrauchswert aller Plugs wäre bei Miteinbeziehung der Nullwerte erheblich kleiner, da die Summe des Gesamtverbrauchs nicht steigt, während sich aber die Menge an Stromzähler (durch die beispielsweise bei der Mittelwertberechnung geteilt wird) erhöht. Auch eine Berechnung auf Basis des Medians wird dadurch verfälscht. Durch die Verschiebung der Messwerte aufgrund der nach Verbrauchswert-Sortierung am Anfang gelisteten Nullwerte, kann ein unrealistisch kleiner Wert als Mittelpunkt der Menge auftauchen. Dramatischer wäre es, wenn mehr als die Hälfte der Plugs Nullwerte enthalten würde, denn dann wäre der Median sogar ebenfalls Null. Eine Aussortierung dieser Werte kann in Spark sehr einfach mithilfe der *filter* Transformation durchgeführt werden:

```
(1)   // Filter consumptions <= 0 out
(2)   JavaPairDStream<String, Float> debsPairDStreamConsumptionGreater0 =
       debsPairDStreamWithoutDuplicates.filter(filterConsumptionGreater0-
       Function);
(3)
(4)   private static Function<Tuple2<String,Float>, Boolean> filterConsump-
       tionGreater0Function = new Function<Tuple2<String,Float>,Boolean>(){
(5)     public Boolean call(Tuple2<String, Float> v1) throws Exception {
(6)       return v1._2() > 0.0;
(7)     }
(8)   };
```

Codeblock 32: Filterung der Nullwerte im Verbrauch

Daten nochmals umformen

Der aktuelle *JavaPairDStream* besitzt nach Aufruf der *pairFunction* den vierteiligen Schlüssel *houseId.householdId.plugId.timestamp*. Da sich die folgenden Berechnungen lediglich auf Stromzählerdaten eines gewissen Zeitfensters stützen und Spark vereinfachte Operationen auf Key-Value Pairs gleichen Schlüssels bietet, kann nun der Paar-DStream um die Information des Zeitstempels verkleinert werden. Durch das Definieren der Zeitintervalle, in welchen

die Daten zusammengefasst und weiterverarbeitet werden, wird diese Angabe irrelevant. Durch eine weitere *mapToPair* Transformation muss deshalb der aktuelle *JavaPairDStream* auf einen neuen *JavaPairDStream<String, Float>* abgebildet werden. Dieser Stream ist jedoch um die ab diesem Schritt unnötige Angabe des Timestamps im Schlüssel geschrumpft. Codeblock 33 zeigt die Zerschneidung des Schlüssels. Da der resultierende Paar-DStream in nachfolgenden Schritten mehrmals benötigt wird, muss hier eine Datenhaltung im Hauptspeicher durch zusätzlichen Aufruf der *cache* Transformation angestoßen werden.[53]

```
(1)    // Remove Timestamp from Key -> (house.hh.plug, val) without TS
(2)    JavaPairDStream<String, Float> debsPairDStreamConsumptionGreater0
        withoutTs = debsPairDStreamConsumptionGreater0
        .mapToPair(removeTsFromPairDStreamFunction).cache();
(3)
(4)    private static PairFunction<Tuple2<String,Float>, String, Float>
        removeTsFromPairDStreamFunction = new PairFunction<Tuple2<String,
        Float>,String, Float>() {
(5)      public Tuple2<String, Float> call(Tuple2<String, Float> t) {
(6)        String[] keyParts = t._1().split("\\.");
(7)        String newKey = keyParts[0]+ "."+ keyParts[1]+ "."+ keyParts[2];
(8)        return new Tuple2<String, Float>(newKey, t._2());
(9)      }
(10)   };
```

Codeblock 33: Zeitstempel aus dem Schlüssel des JavaPairDStream entfernen

Die bis hierhin durchgeführten Umwandlungen und Selektionen der am Cluster eintreffenden Daten bilden die Basis für den Aufbau des Modells zur Ausreißererkennung. Die Datenvorbereitung ist damit abgeschlossen und es kann mit der Umsetzung der eigentlichen Logik zur Echtzeiterkennung von Stromverbrauchs-Ausreißern fortgefahren werden. Dies wird im nachfolgenden Kapitel *Modeling* behandelt.

9.4 Modeling: Umsetzung der Architektur

Dieses Kapitel beschäftigt sich mit der eigentlichen Umsetzung der Ausreißererkennung. Nachdem die Daten in den vorherigen Schritten bereinigt und passend umgeformt wurden, kann nun mit der Implementierung des Modells fortgefahren werden. Wie einleitend in der Aufgabenstellung erläutert, soll sich das Modell an der Umsetzung innerhalb der DEBS Grand Challenge orientieren. Die dort gegebene Aufgabe zur Ausreißererkennung lautet wie folgt:

> *The goal of the outliers query is to find the outliers as far as the energy consumption levels are concerned. The query should answer the following question: for each house calculate the percentage of plugs which have a median load during the last hour greater than the median load of all plugs (in all households of all houses) during the last hour. The load statistics query should*

[53] Dieses Caching ist nur bei solchen Analysen vonnöten, bei denen anschließend mehrere Transformationen bzw. Aktionen auf diesem DStream aufgerufen werden.

> *generate two output streams, one for a sliding window of one hour (described above) and one for a sliding window of a whole day (24 hours). [...]*
> *– zitiert von http://www.cse.iitb.ac.in/debs2014/?page_id=42*

Aufgrund der begrenzten zeitlichen Ressourcen, die für die Umsetzung des Prototyps zur Verfügung stehen, wurde die obige Aufgabenstellung folgendermaßen angepasst:

- Die Ausreißererkennung soll möglichst aktuell stattfinden. Entsprechend soll das Batchintervall und die Slide-Duration bei wenigen Sekunden liegen (nicht zwingend bei genau einer).
- Da eine Berechnung über die letzten 24 Stunden zu rechenintensiv ist, reicht die Umsetzung des ersten Output-Streams, über das einstündige Window, aus.
- Es sollen verschiedene Ansätze zur Mittelwertberechnung untersucht werden.

Die Aufgabe soll auf einem Cluster bestimmter Größe umgesetzt werden und für jedes Haus den prozentualen Anteil an Plugs ermitteln, die als Ausreißer identifiziert wurden. Durch Anpassung von Parametern muss eine dauerhafte Ausführung ermöglicht und gleichzeitig die im Cluster zur Verfügung stehenden Ressourcen möglichst gut ausgenutzt werden. Das Cluster soll über die Amazon Webservices aufgesetzt werden und folgenden Aufbau besitzen:

- Sender: 1 x m3.large (2 Cores, 7,5 GB RAM)
- Spark-Master: 1 x m3.large
- Spark-Worker: 4 x m3.large
- Es sollen 2 Sende-/Empfangs-Sockets zur Datenübertragung eingesetzt werden.

Die nachfolgenden Kapitel stellen die untersuchten Varianten zur Umsetzung einer Ausreißererkennung mit Apache Spark vor. Dabei wird besonders auf die Möglichkeiten der parallelen Verarbeitung eingegangen, da diese entscheidend für den Einsatz von Spark-Clustern sind. Ohne die Verwendung paralleler Transformationen macht eine Umsetzung mit Spark keinen Sinn. Aufgrund des Overheads, der im Cluster beispielsweise zur Organisation anfällt, wäre die sequentielle Laufzeit sehr wahrscheinlich sogar höher als bei der Umsetzung in anderen Umgebungen, wie z. B. dem Einsatz einfacher Java-Threads. Es ist deshalb wichtig, möglichst große Teile der Ausreißererkennung parallel abarbeiten zu können, um vom Aufbau und der zur Verfügung stehenden Ressourcen des Clusters profitieren zu können. Nachfolgend werden drei Ansätze für die Umsetzung einer Ausreißererkennung diskutiert. Mithilfe von Spark-Applikationen wird die Einsatzfähigkeit dieser Lösungsmöglichkeiten jeweils validiert.

9.4.1 Arithmetisches Mittel

Der erste Ansatz, über welchen eine Ausreißererkennung umgesetzt werden kann, ist die Berechnung des Mittelwerts über das arithmetische Mittel. Die Berechnung an sich ist trivial:

$$\bar{x}_{arithm} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

Formel 5: Berechnung des arithmetischen Mittels

Codeblock 55 im Anhang stellt eine solche Applikation dar und wird nachfolgend vorgestellt. Um eine Berechnung über die Batches der jeweils vergangenen Stunde realisieren zu können, muss zunächst die Window-Größe auf diese Zeitspanne gesetzt werden. In Zeile 12 wird deshalb der entsprechende Parameter, welcher die Größe eines Windows beeinflusst, auf 3600 Sekunden gesetzt. Das Batchintervall und die Slide-Duration können in dieser Anwendung auf den gleichen Wert gesetzt werden, da eine Zusammenfassung als kleinere Batches vor Neuberechnung des Window-Inhalts, welche wiederum nach Ablauf der Slide-Duration angestoßen wird, nicht nötig ist. Der Wert für beide Parameter kann durch Angabe des Anwenders beim Applikationsstart festgelegt werden (Zeile 11). Nach den Schritten der Data Preparation liegen die Daten als *PairDStream<String, Float>* mit folgendem Aufbau zur Weiterverarbeitung vor:

```
(1)   (houseId.householdId.plugId, value)
(2)   ==================================
(3)   (0.0.0, 1.0)
(4)   (0.0.1, 21.5)
(5)   ...
(6)   (20.3.2, 3.123)
(7)   ...
(8)   (39.4.32, 62.2)
(9)   (39.4.33, 16.7)
```

Codeblock 34: Aufbau der Daten nach Durchführen der Data Preparation

Berechnung der Mittelwerte je Stromzähler innerhalb des Windows

Die nun folgenden Schritte werden durch Abbildung 33 visualisiert und im Nachfolgenden erläutert: Da sich die Paare bereits aus Stromzählerkennung (Schlüssel) und Verbrauchsdaten (Werte) zusammensetzen, kann nun im fünften Schritt (Datenvorbereitung entspricht Schritt 1 bis 4) mit der Mittelwertberechnung auf Stromzählerbasis fortgefahren werden. Hierbei muss, wie oben beschrieben, innerhalb des Windows eine Zählung der Messpunkte und Aufsummierung der Verbrauchswerte je Plug durchgeführt werden. In der Beispielapplikation wird dabei mit dem Zählen der Messpunkte begonnen. Dazu wird in Schritt 5.1 bzw. Zeile 50 des Quellcodes eine Funktion aufgerufen, welche die Daten per Aufruf einer *mapToPair* Transformation in ein *PairDStream<String, Long>* der Form (houseId.householdId.plugId, 1) umwandelt. Mithilfe dieses *PairDStreams* kann im folgenden Schritt (5.2) eine parallele und Schlüssel-bezogene Zählung der Elemente durch Aufsummierung der 1-Werte durchgeführt werden.

Dies kann durch Ausführen der *reduceByKeyAndWindow* Transformation (Zeile 53) angestoßen werden. Ebenfalls eine Datenreduktion findet im nächsten Schritt (5.3, Zeile 56) statt: Hier wird ebenfalls durch Aufruf dieser Transformation eine Summe über alle Elemente mit gleichem Schlüssel gebildet. Dieses Mal werden aber, statt wie oben 1-Werte, die Werte des in Codeblock 34 vorgestellten Paar-DStreams zusammengezählt und somit der plugbasierte Stromverbrauch der letzten Stunde errechnet. Nun gilt es, die zuvor berechneten Summen zusammenzuführen, um daraus das arithmetische Mittel, über Division des Gesamtverbrauchs durch die Anzahl der Messpunkte, zu erhalten. Das Vereinen der beiden *Pair-DStreams* wird in Schritt 5.4 bzw. Zeile 59 des Quellcodes durch Aufruf einer *join* Transformation realisiert. Anschließend wird auf diesem Gesamt-Stream die Division nach Schlüssel (Plugkennung) durchgeführt. Dabei entsteht wiederum ein *PairDStream<String, Float>* der Form (houseId.householdId.plugId, avg). Dieser wird später für die Ausreißerfindung benötigt.

Berechnung des Mittelwerts über alle Plugs innerhalb des Windows

Die nun folgenden Schritte 6.x führen im Grunde die gleichen Operationsschritte nochmals aus, dieses Mal aber auf der gesamten Datenmenge des Windows statt auf Plugebene. Dazu benötigt es einer Umformung der zu verarbeitenden Daten (Schritt 6.1): Um wieder von der parallelen *reduceByKeyAndWindow* Transformation zu profitieren, müssen alle Daten des *PairDStreams* auf den gleichen Schlüssel gebracht werden, um in den folgenden Schritten gesamtheitlich verarbeitet werden zu können. Diese Aufgabe kann durch Umformung des Ausgangs-Streams via *mapToPair* Transformation erledigt werden: Die in Zeile 63 aufgerufene Funktion mappt den aufrufenden *PairDStream* auf einen Neuen mit Schlüssel „all" und unverändertem Wert. Die nun folgenden Schritte 6.2 bis 6.5 rufen in gleicher Reihenfolge die bereits in Schritt 5.1 bis 5.4 verwendeten Funktionen auf. Nach Schritt 6.5 existiert auch hier ein *PairDStream<String, Float>*, der den Mittelwert enthält, hier jedoch mit dem Inhalt („all", avg).

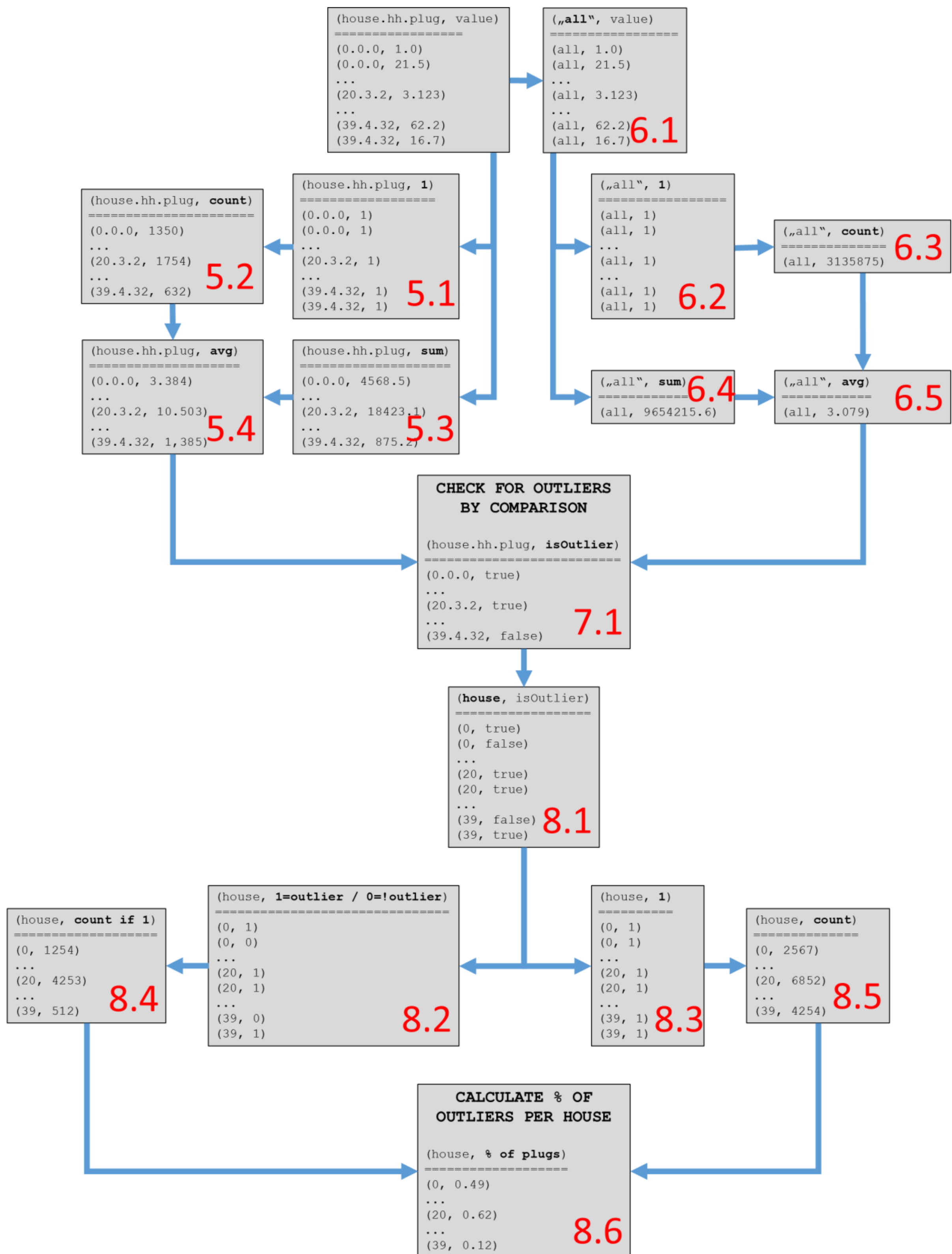

Abbildung 33: Weitere Schritte zur Ausreißer-Findung per arithmetischem Mittel

Ausreißererkennung

Die nächsten Schritte der Anwendung sind nun für die eigentliche Ausreißer-Identifizierung zuständig: Sie vergleichen plugweise den Verbrauch der letzten Stunde (Inhalt des Windows) mit dem arithmetischen Mittel über alle Stromzähler dieses Zeitraums. Liegt der Durchschnittsverbrauch eines Plugs über diesem Wert, so soll dieser als Ausreißer festgelegt werden. Um einen solchen Vergleich der beiden *PairDStreams* durchführen zu können, müssen die beiden Streams, welche in Schritt 5.4 bzw. 6.5 entstanden sind, per *union* Transformation (Zeile 79) zusammengeführt werden. Im Anschluss wird, ebenfalls in Schritt 7.1, eine Funktion aufgerufen, die den oben beschriebenen Vergleich jedes Stromzähler-Mittelwerts mit dem Gesamtdurchschnitt durchführt und entsprechend jene Zähler mit höherem Verbrauch als Ausreißer einstuft. Durch Mapping auf einen neuen *PairDStream<String, Boolean>* wird diese Klassifikation der Plugs repräsentiert (true = Ausreißer / false = kein Ausreißer). Das Vorgehen innerhalb der *findOutliersPerComparisonFunction* Funktion (Zeile 158-182) ist dabei wie folgt: Nachdem der vereinte Stream auf Vorhandensein von Daten geprüft wurde (der Stream kann durchaus leer sein, z. B. vor dem Empfangen von Daten, was zu einem Laufzeitfehler führen würde), wird der Mittelwert der gesamten Window-Datenmenge per Filterung nach Schlüssel „all" herausgesucht und zwischengespeichert. Anschließend werden, durch Aufruf der *mapValues* Transformation, die Werte des *PairDStreams* (Verbrauch: Float) auf den neuen Datentyp (Ausreißer (ja/nein): Boolean) gemappt. Dies geschieht durch Vergleich des Verbrauchswerts jedes Plugs mit dem zwischengespeicherten Gesamt-Mittelwert aller Stromzähler. Aus dem erhaltenen *PairDStream<String, Boolean>* mit Inhalt (houseId.householdId.plugId, isOutlier) wird abschließend wieder per *filter* Transformation der Wert des Gesamtmittels entfernt und der restliche Paar-DStream, welcher nun noch die plugbasierten Ausreißereinstufungen enthält, zurückgegeben.

Prozentualer Anteil an Plugs mit erhöhtem Verbrauch pro Haus

Die abschließende Aufgabe bei der Umsetzung dieses Prototyps besteht darin, den prozentualen Anteil an Stromzählern mit erhöhtem Verbrauch (höher als der Gesamt-Mittelwert) herauszufinden. Diese Berechnung soll dabei hausbasiert stattfinden, was soviel bedeutet, wie dass die einzelnen plugbasierten Ausreißer-Einstufungen pro Haus aggregiert werden müssen. Das hierfür nötige Vorgehen wird in den Schritten 8.x umgesetzt und nun erläutert. Zunächst muss der bislang plugbasierte Stream auf einen neuen *PairDStream<String, Boolean>* gemappt werden, der statt der eindeutigen Stromzählerkennung nur noch die Haus-ID als Schlüssel enthält. Der Inhalt entspricht nach dem Mapping somit (houseId, isOutlier). Dies wird in Schritt 8.1 bzw. Zeile 83 durch den Aufruf der *mapToPair* Transformation erledigt: Sie zerschneidet den bisherigen Schlüssel und weist dem Key des neuen Paar-DStream nur noch den ersten Teil, welcher die Hauskennung enthält, zu. Anschließend findet im Schritt 8.2 die

Vorbereitung für eine bedingte Zählung statt. Um später herauszufinden, wie viele Plugs wirklich Ausreißer sind, muss die Zählung der Elemente pro Haus so angepasst werden, dass hier nur die als Ausreißer eingestuften Verbrauchswerte miteinbezogen werden. Die Lösung bietet hier ein bedingtes Mapping auf einen neuen *PairDStream<String, Long>*, welcher den Inhalt (houseId, 1 if outlier / 0 if not) besitzt. Der Wert eines Paares wird also beim Mapping nur dann auf 1 gesetzt, wenn der Eltern-Stream den Wert „isOutlier = true“ besaß, andernfalls wird dem Wert eine 0 zugewiesen. Diese Umwandlung wird ebenfalls durch eine *mapToPair* Transformation in Zeile 86 angestoßen. Eine einfache Zählung aller Elemente eines Hauses wird in Schritt 8.3 vorbereitet, hier wird, wie von Schritt 5.1 bekannt, der Inhalt auf einen neuen *PairDStream* mit Inhalt (houseId, 1) gemappt. Die Schritte 8.4 und 8.5 sind dann jeweils für das schlüsselbasierte Aufsummieren der Werte der beiden eben beschriebenen Streams zuständig. Durch eine einfache Addition der Elemente mithilfe zweier *reduceByKey* Transformationen (Zeile 92 bzw. 95) entsteht jeweils wieder ein neuer Paar-DStream mit Inhalt (houseId, sum). Da einer der Streams nun die Gesamtsumme aller Elemente des Ausreißer-Streams und der andere lediglich die Summe der wirklich als Ausreißer eingestuften Plugs enthält, können diese beiden Streams im nächsten Schritt (8.6) miteinander verrechnet werden. Dazu wird zunächst eine *join* Transformation aufgerufen, welche die beiden Streams verbindet. Da es sich bei den Summen-Streams um bereits zusammengefasste Datenflüsse handelt, enthält auch der zusammengeführte Stream lediglich nur einen Eintrag pro Schlüssel. Es ist deshalb über eine einfache *mapToPair* Transformation möglich, die Division von wirklichen Ausreißern durch Gesamtzahl an Plugs pro Haus durchzuführen (Zeile 98). Somit entsteht in diesem Schritt ein *PairDStream<String, Float>* mit Inhalt (houseId, % of outliers). Abschließend folgt eine *print* Output-Operation, die gleichzeitig die Ausführung der Streaming-Logik anstößt.

9.4.2 Median

Eine weitere Variante, Mittelwerte für eine spätere Ausreißererkennung zu berechnen, ist das Bestimmen von Medianen. Hier wird, im Gegensatz zum arithmetischen Mittel, der Mittelwert nicht von allen Elementen der Menge beeinflusst. Es wird stattdessen das mittlere Element innerhalb einer sortierten Menge bestimmt. Sollte es sich um eine Menge mit gerader Anzahl an Elementen handeln, wird der Median durch Bilden des arithmetischen Mittels zwischen den beiden mittleren Elementen berechnet. Die Bestimmung des Wertes lässt sich über folgende Formel ausdrücken:

$$\tilde{x}_{median} = \begin{cases} x_{\frac{n+1}{2}} & n\ ungerade \\ \frac{1}{2}\left(x_{\frac{n}{2}} + x_{\frac{n}{2}+1}\right) & n\ gerade \end{cases}$$

Formel 6: Bestimmung des Medians

Auch die Bestimmung des Medians wurde in einer Spark Streaming Applikation umgesetzt. Codeblock 56 enthält den dazugehörigen Quellcode der Anwendung. In den nachfolgenden Unterkapiteln werden die einzelnen Schritte anhand des Quellcodes nacheinander vorgestellt. Die Parameterdefinition und Datenvorbereitung wurden dabei aus der vorherigen Anwendung (Kapitel 9.4.1) übernommen: Per Parameter muss der Benutzer bei Applikationsstart die URL zum Sender und die gewünschte Batch-Size übergeben. Auch die Windowgröße ist entsprechend der Aufgabenstellung wieder auf eine Stunde gestellt und die Slide-Duration wird gleich dem Batchintervall gesetzt. Nach der anschließenden Datenvorbereitung, wie sie in Kapitel 9.3.2 gezeigt wird, liegen auch bei dieser Anwendung die Daten als *PairDStream<String, Float>* mit Inhalt (houseId.householdId.plugId, value), wie in Codeblock 34, vor. Es kann nun mit der Medianbestimmung fortgefahren werden. Abbildung 34 zeigt die weiteren Schritte, welche in der Applikation durchlaufen und nachfolgend erläutert werden.

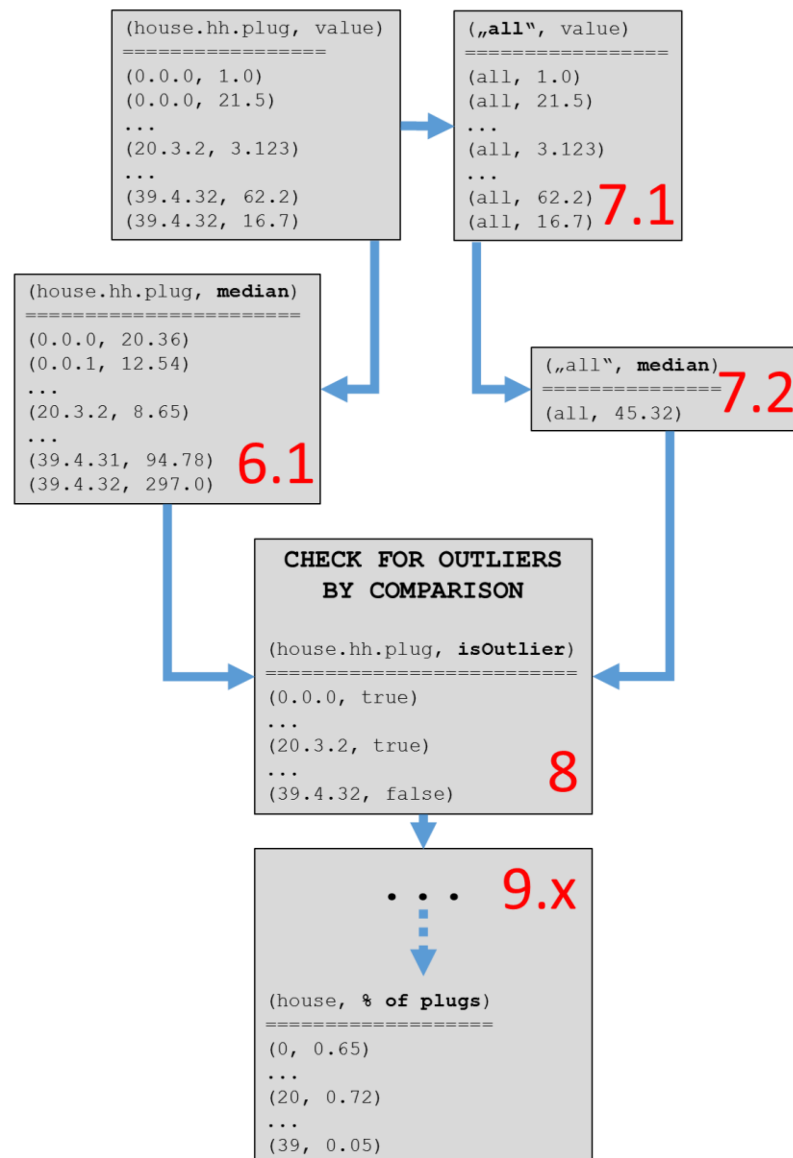

Abbildung 34: Weitere Schritte zur Ausreißer-Findung per Medianbestimmung

Bestimmung des Medians je Stromzähler innerhalb des Windows

Die weitere Vorgehensweise bei der Medianbestimmung ähnelt der aus der Umsetzung mit arithmetischem Mittel: Es werden jeweils zwei Streams miteinander verbunden und per Vergleich die Ausreißer festgelegt. Einer der Streams führt auch hier Berechnungen auf Stromzählerbasis durch, während der andere für die Gesamtmenge an Plugs zuständig ist. Die Ausreißerbestimmung durch Mediane unterscheidet sich aber auch deutlich von der vorigen Variante: Da Spark (noch) keine parallelen Transformationen für diese Mediansuche bzw. Sortierung und das anschließende indexbasierte Auslesen von Daten eines RDDs bietet, ist eine optimale Parallelisierung dieser Aufgabe nicht mit Spark-Funktionalitäten möglich. Dadurch bleibt dem Benutzer nichts weiter übrig, als eigene Implementierungen zur verteilten Median-Suche bereitzustellen oder auf eine sequentielle Bestimmung auszuweichen. Da erstere Lösungsmöglichkeit außerhalb des Rahmens dieser Arbeit liegt, wird im Programm ein sequentielles Vorgehen umgesetzt. Bevor diese Datenzusammenfassung durchgeführt werden kann, muss zunächst das Window, über dessen Zeitraum die Analyse später durchgeführt werden soll, per Aufruf der gleichnamigen *window* Transformation, definiert werden. In Schritt 5 bzw. Zeile 10 des Codeblock 56 wird diese Methode aufgerufen. Sie liefert einen *PairDStream* zurück, der die Daten des gesamten Windows, also die Messpunkte der letzten Stunde, beinhaltet. Da die Medianbestimmung ebenfalls nur auf Window-Auflösung stattfindet, wird in den nächsten Schritten jeweils der hier erstellte Stream weiterverarbeitet. Über Schritt 6.1 (Zeile 14) wird anschließend diese Ansammlung an Messpunkten per *groupByKey* Transformation nach Schlüsselwert gruppiert, dadurch werden die Daten in bis zu 1956 Partitionen eingeteilt, je Stromzählerkennung eine. Im gleichen Schritt wird dann in einer *mapToPair* Transformation die Funktion zur Medianbestimmung innerhalb einer solchen Teilmenge aufgerufen. Bei der Funktion selbst handelt es sich, wie oben bereits angekündigt, um eine Methode zur sequentiellen Datentraversierung zur Medianbestimmung. Während die zuvor entstandenen Partitionen als solche zwar parallel verarbeitet werden können, findet die Verarbeitung der Daten innerhalb einer solchen Gruppe jedoch sequentiell statt. Zeile 40 bis 72 definieren diese Methode, in der folgendermaßen vorgegangen wird: Eine Datenpartition setzt sich nach der Gruppierung nach Schlüssel aus (houseId.householdId.plugId, Iterable<Float>) zusammen. Es gibt pro Partition (also Plug) genau einen Eintrag innerhalb des Streams, welcher als Wert die Menge zugehöriger Verbrauchswerte besitzt. Diese Menge an Messpunkten wird nun in der Funktion durchiteriert und in einen neuen *PairDStream<String, Float>*, welcher den Medianwert der Partition enthält, abgebildet. Um das Medianelement der Menge zu finden, muss diese sortiert werden. Dies kann hier nur über die native Java-Methode *Arrays.sort* (Zeile 55) erledigt werden. Dazu müssen die Daten in ein Array geschrieben werden, welches für die weitere Medianbestimmung verwendet wird. Erledigt wird dieser

Schritt mithilfe eines Iterators in den Zeilen 44 bis 54. Die bereits angesprochene Sortierung findet ebenfalls sequentiell statt, da Spark keine entsprechenden Transformationen zur parallelen Sortierung von RDDs bietet. Ist die rechenintensive Sortierung beendet, muss das mittlere Element bzw. die mittleren Elemente des sortierten Arrays ausgelesen werden (Zeile 57-67). Handelt es sich um eine Menge mit gerader Anzahl an Elementen, so wird der Median durch Teilen der beiden mittleren Elemente berechnet, siehe Formel 6. Der Wert des Medians wird dann von der Funktion zurückgegeben und auf den *PairDStream<String, Float>* mit Inhalt (houseId.householdId.plugId, median) gemappt. Dieser enthält die stromzählerbasierten Mediane und kann später für die Ausreißer-Einstufung verwendet werden.

Bestimmung des Medians über alle Plugs innerhalb des Windows

Die Bestimmung des Medians über alle Plugs läuft im Grunde genauso ab, wie die Plugbasierte im obigen Kapitel. Damit hier aber die Daten als eine große Gesamtmenge verarbeitet werden können, muss zunächst ein gemeinsamer Schlüssel definiert werden, auf den der bisherige Stream abgebildet wird. Mithilfe einer *mapToPair* Transformation werden also die Daten der Form (houseId.householdId.plugId, value) in einen neuen Paar-DStream mit Inhalt („all“, value) gewandelt. Dieses Vorgehen ist analog zu Schritt 6.1 in der Umsetzung mit arithmetischem Mittel in Kapitel 9.4.1. Nachdem dieser neue Stream erstellt ist, werden auf ihm die gleichen Operationen ausgeführt wie bei der stromzählerbasierten Medianermittlung: Durch Aufruf der *groupByKey* Transformation werden die Daten zu einer großen Partition der Form („all“, Iterable<Float>) gewandelt (Schritt 7.2 bzw. Zeile 18). Anschließend folgt per Aufruf der sequentiell arbeitenden Funktion die Bestimmung des Medians der Gesamt-Datenmenge des Windows. Die Abarbeitung dieser Funktion benötigt hier ein Vielfaches gegenüber der Ausführung pro Stromzähler, da die Gesamt-Datenbasis bis zu 1956-mal so groß sein kann, wie in Schritt 6.1 und wieder nur sequentiell verarbeitet wird. Nachdem auch hier ein neuer *PairDStream<String, Float>* mit Inhalt („all“, median) entstanden ist, folgen in den nächsten Schritten die Ausreißererkennung und Berechnung der Ausreißer-Plugs pro Haus.

Weiteres Vorgehen zur Ausreißerbestimmung

Nach Abarbeitung der Schritte 6.x und 7.x liegen die Daten, wie bereits in der vorherigen Anwendung, als zwei Streams vor, von denen einer den Mittelwert pro Stromzähler und der andere den Mittelwert über alle Plugs enthält. Der einzige Unterschied besteht darin, dass diese nun die Mediane enthalten und nicht mehr die arithmetischen Mittel. Aufgrund dieses gleichen Stream-Aufbaus können nun die Ausreißerermittlung und Berechnung des prozentualen Anteils je Haus über die gleichen Funktionen erfolgen, wie im vorherigen Prototyp. Es sei daher an dieser Stelle auf die beiden Abschnitte *Ausreißererkennung* und *Prozentualer Anteil an Plugs mit erhöhtem Verbrauch pro Haus* des Kapitels 9.4.1 verwiesen.

9.4.3 Arithmetisches Mittel der Mediane

Eine dritte Möglichkeit, einen guten Mittelwert für spätere Analysen zu finden, ist die Vereinigung der beiden vorherigen Varianten: Durch eine Medianbestimmung auf Batchebene werden weniger Datenpunkte zur Bestimmung miteinbezogen und damit das sequentielle Finden des Medians pro Batch entsprechend schneller durchgeführt. Was an dieser Stelle noch fehlt, ist die Mittelwertbestimmung innerhalb der Datenmenge des gesamten Windows. Hierfür können die zuvor berechneten Medianwerte der zum Window gehörenden Batches herangezogen werden. Da bei kleinem Batchintervall dennoch sehr viele Daten durch die einzelnen Batches anfallen, ist es hier sehr hilfreich, die weitere Datenaggregation über einen parallelen Algorithmus durchzuführen. Hierfür eignet sich deshalb die Mittelwertberechnung per arithmetischem Mittel, da diese, wie in Kapitel 9.4.1 gezeigt, über parallele Spark-Streaming Transformationen umgesetzt werden kann. Abbildung 35 verdeutlicht die Vorgehensweise bei dieser Umsetzungsmöglichkeit:

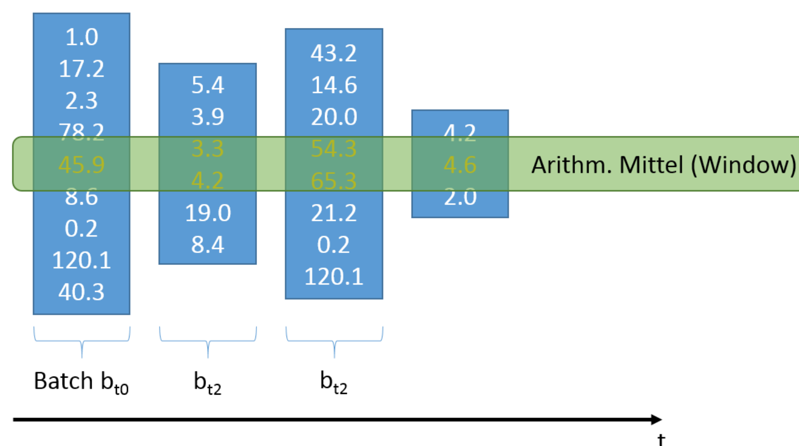

Abbildung 35: Kombination: Median (pro Batch) und arithmetischem Mittel (pro Window)

Die Medianbestimmung innerhalb eines Batches findet dabei, wie in Formel 6 beschrieben, statt. Die Berechnung des arithmetischen Mittels über die Mediane innerhalb des Windows orientiert sich an Formel 5, verwendet aber zusätzlich Gewichtungen. Auch die Umsetzbarkeit dieser Variante wurde mithilfe einer Spark-Applikation validiert. Codeblock 57 im Anhang zeigt den Quellcode dieser Anwendung und wird nachfolgend schrittweise erläutert.

Die Parameterdefinition und Datenvorbereitung wurden aus den vorherigen Anwendungen (Kapitel 9.4.1) übernommen:

- Parameter beim Starten der Applikation: Sender-URL und Batch-Size (in Sekunden)
- Windowgröße ist eine Stunde
- Slide-Duration ist gleich dem Batchintervall
- Datenvorbereitung (Schritt 1-4) findet gemäß Kapitel 9.3.2 statt und liefert somit ebenfalls einen *PairDStream<String, Float>* mit Inhalt (houseId.householdId.plugId, value).

Bestimmung des Medians je Stromzähler innerhalb eines Batches

Nachdem die Vorbereitung der Daten abgeschlossen ist, und die Daten als Paar-DStream mit plugbasierten Verbrauchswerten vorliegen, kann mit der Umsetzung der Mittelwertberechnungen auf Batchebene fortgefahren werden. Abbildung 36 zeigt die weitere Vorgehensweise für diese Applikation graphisch auf. Im Gegensatz zu den beiden vorherigen Varianten, welche lediglich auf den Daten von Windows operieren, lassen sich die Schritte dieser Applikation in batch- und windowbasierte Operationen unterteilen. Die ersten Schritte behandeln jeweils lediglich die Daten einzelner Batches, ehe deren Ergebnisse dann in den darauffolgenden windowbasierten Operationen aggregiert werden.

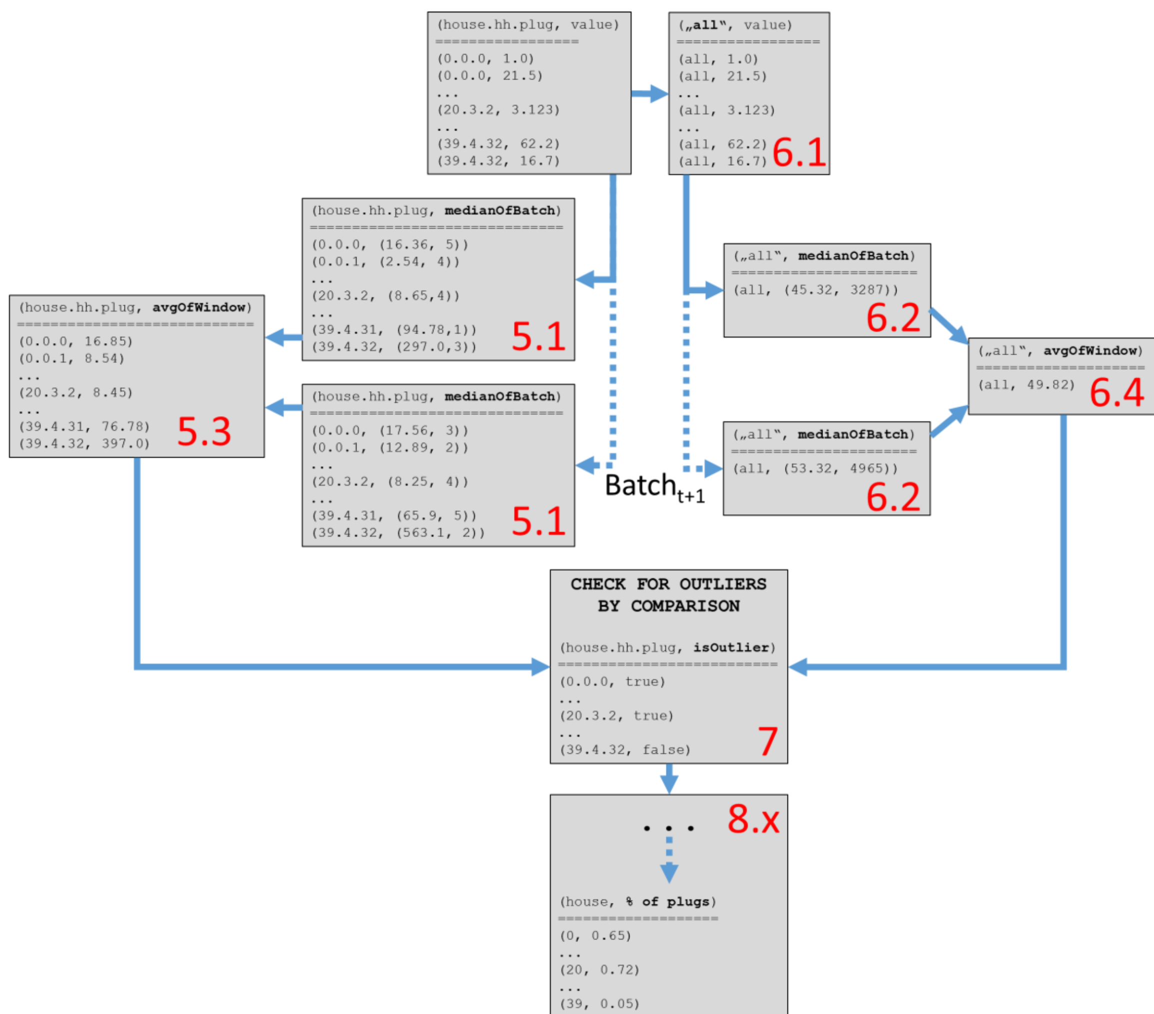

Abbildung 36: Weitere Schritte zur Ausreißer-Findung per Mittel der Mediane

Schritt 5.1 (Zeile 11) beginnt mit der plugbasierten Medianfindung innerhalb eines Batches. Dabei werden alle Daten, die seit Berechnung des vorherigen Batches am Empfänger-Socket ankamen, zur Bestimmung miteinbezogen. Die Berechnung an sich findet, wie bereits in der vorherigen Anwendung erläutert, durch Gruppierung nach Stromzählerkennung, Sortierung und Rückgabe des mittleren Elements (bei ungerader Anzahl Daten) bzw. der Verrechnung der beiden mittleren Elemente (gerade Datenanzahl) statt. Auch hier besteht das Problem,

dass die Sortierung der Werte, wie bereits in Kapitel 9.4.2 diskutiert, sequentiell für jeden Stromzähler durchgeführt werden muss. Dieser Schritt läuft hierbei allerdings erheblich schneller ab, da die Datenmenge um ein Vielfaches kleiner ist. Bei einem Batchintervall von 10 Sekunden kann es gemäß der Sendegeschwindigkeit (eine Messung je zwei Sekunden) nur bis zu fünf Datenpunkte pro Stromzähler geben. Bei einer Verdoppelung des Batchintervalls würden hier auch bis zu zweimal so viele Datenpunkte pro Batch und Stromzähler existieren usw. An dieser Stelle sei erwähnt, dass bei einem Batchintervall von zwei Sekunden lediglich nur ein Datenpunkt pro Plug vorhanden wäre, und damit die Medianbestimmung exakt dieses Element als Mittelwert festlegen würde. In diesem Fall hätte dieser Schritt keine Auswirkung auf die spätere Berechnung des arithmetischen Mittels, sodass diese Anwendung die gleichen Ergebnisse erzielen würde, wie die Umsetzung mit arithmetischem Mittel allein (Kapitel 9.4.1). Bei der Medianbestimmung wird zusätzlich die Anzahl vorhandener Datenpunkte des aktuellen Stromzählers im Batch für eine spätere Gewichtung mitgespeichert. Die Rückgabe der *transformToPair* Transformation liefert damit einen *PairDStream<String, Tuple2<Float, Long*>> mit Inhalt (houseId.householdId.plugId, (median, count)) zurück.

Berechnung des plugbasierten Mittelwerts aller Mediane des Windows

Im nun folgenden Schritt müssen die zuvor berechneten, batchbasierten Medianwerte innerhalb des Windows stromzählerweise zu einem neuen Mittelwert zusammengefasst werden. In den Schritten 5.2 bis 5.6 wird dazu ähnlich wie in Kapitel 9.4.1 vorgegangen: Zunächst wird der Paar-DStream der Mediane mithilfe zweier *mapToPair* Transformationen in zwei neue Streams gewandelt. Der Stream aus Schritt 5.2 (Zeile 14) errechnet die Gewichtung eines Batches und ist später für das Aufsummieren der Verbrauchswerte der Mediane zuständig, während der zweite Stream (5.3) für das Zählen der Mediangewichtungen durch Summenbildung über den *count*-Wert des Streams, verwendet wird. Damit entstehen die beiden Streams:

- *PairDStream<String, Float*>, Inhalt (houseId.householdId.plugId, sumOfValue)
- *PairDStream<String, Long*>, Inhalt (houseId.householdId.plugId, sumOfWeights)

Über zwei folgende *reduceByKeyAndWindow* Transformationen (Schritt 5.4 / 5.5 bzw. Zeile 20 / 23) werden dann diese Aufsummierungen window- und plugbasiert durchgeführt. Die beiden Streams werden dann in Schritt 5.6 per *join* Transformation zusammengeführt und eine Funktion zur Bestimmung des gewichteten arithmetischen Mittels nach Formel 7 aufgerufen. Sie errechnet das gewichtete arithmetische Mittel nun über eine einfache Division der Verbrauchssumme durch die Gewichtungssumme. Zurückgegeben wird ein *PairDStream <String, Float*> mit Inhalt (houseId.householdId.plugId, avgOfMedians).

$$\bar{x} = \frac{\sum_{i=1}^{n} w_i \, x_i}{\sum_{i=1}^{n} w_i}$$

Formel 7: Berechnung des gewichteten arithmetischen Mittels

Bestimmung des Medians über alle Plugs innerhalb eines Batches

Um den Median über den Verbrauch aller Stromzähler innerhalb eines Batches bestimmen zu können, muss zunächst der aufrufende *PairDStream* auf einen neuen mit gemeinsamem Schlüssel „all“ abgebildet werden (Schritt 6.1), um dann anschließend diese Daten als Gesamtmenge verarbeiten zu können. Dieses Vorgehen in Schritt 6.1 führt zu einem neuen *PairDStream<String, Float>* mit Inhalt („all“, value). In Schritt 6.2 (Zeile 33) findet dann ein analoges Vorgehen zu dem der Medianbestimmung pro Plug (Schritt 5.1) statt: Die Daten des Streams werden zunächst per *groupByKey* Transformation nach Schlüsselwert gruppiert. Da alle Paare denselben Schlüssel besitzen, wird eine große Partition erstellt, die dann in der nachfolgenden Funktion verarbeitet wird. Es werden wieder sämtliche Messpunkte nach Verbrauchswert sortiert und das mittlere Element gemäß Formel 6 berechnet. Auch hier wird, gemeinsam mit dem Median, die Elementanzahl der Partition als neuer Paar-DStream der Form („all“, (median, count)) zurückgegeben.

Berechnung des Gesamt-Mittelwerts aller Mediane des Windows

Für die Errechnung des Mittelwertes über die Gesamtheit der Mediane eines Windows wird in den Schritten 6.3 bis 6.7 analog zu dem Abschnitt *Berechnung des plugbasierten Mittelwerts aller Mediane des Windows* vorgegangen: Es werden wieder zwei Streams für die beiden Summenbildungen erstellt (6.3 - 6.4), über das Window zusammengefasst (6.5 – 6.6) und diese anschließend miteinander verrechnet (6.7). Der einzige Unterschied liegt darin, dass die Streams keine stromzählerbasierten Daten enthalten, sondern über ihren gemeinsamen Schlüssel „all“ jeweils als eine große Gesamtdatenmenge betrachtet werden. Entsprechend werden die Daten weiterhin als eine große Partition verarbeitet. Die aufgerufene Funktion zur Mittelwertberechnung über das gewichtete arithmetische Mittel, bleibt hierbei ebenfalls dieselbe. Somit wird über Schritt 6.7 (Zeile 48) ebenfalls per *mapToPair* Transformation auf einen *PairDStream<String, Float>* mit Inhalt („all“, avgOfMedians) abgebildet.

Weiteres Vorgehen zur Ausreißerbestimmung

Auch hier liegen nach Abarbeitung der Schritte 5.x und 6.x die Daten, wie bereits in den vorherigen Anwendungen, als zwei Streams vor, von denen einer den Mittelwert pro Stromzähler und der andere den Mittelwert über alle Plugs enthält. In diesem Fall entsprechen die hierin zu findenden Mittelwerten den gewichteten arithmetischen Mitteln über die Batches des Windows. Nichtsdestotrotz kann aufgrund des gleichen Stream-Aufbaus nun die Ausreißerermittlung und Berechnung des prozentualen Anteils je Haus wieder über die gleichen Funktionen erfolgen, wie in den vorherigen Prototypen. Es sei daher an dieser Stelle ebenfalls, wie in der vorherigen Anwendung, auf die beiden Abschnitte *Ausreißererkennung* und *Prozentualer Anteil an Plugs mit erhöhtem Verbrauch pro Haus* des Kapitels 9.4.1 verwiesen.

9.5 Evaluation: Überprüfen der Modelle

In diesem Kapitel werden die im vorherigen Kapitel implementierten Modelle auf ihre Einsetzbarkeit geprüft. Es gilt außerdem, das am besten für dieses Einsatzszenario passende Modell auszuwählen. Dies wird anhand von Benchmarks und Auflistung von Vor-/Nachteilen erledigt.

9.5.1 Bewertung über Benchmarks

In diesem Kapitel werden die einzelnen Modelle anhand von Benchmarks auf ihre Tauglichkeit für den Einsatz in einer Echtzeitanwendung geprüft. Da die Aufgabenstellung keine harten Echtzeitbedingungen fordert, ist eine umgehende Neuberechnung bei Datenempfang nicht unbedingt nötig. Dennoch sollte das Aktualisierungsintervall möglichst gering gewählt werden können, um schnellstmöglich Rückmeldung zu Ausreißern in der Datenmenge zu erhalten. Es handelt sich hierbei vielmehr um eine weiche Echtzeitbedingung, die definiert, wie wertvoll die Informationen nach Ablauf einer gewissen Zeitspanne (zwischen Datenempfang und Berechnung) noch sind. Um eine einheitliche Bewertung durchführen zu können, sei der Wert der Informationen bei entsprechendem Aktualisierungsintervall wie folgt festgelegt:

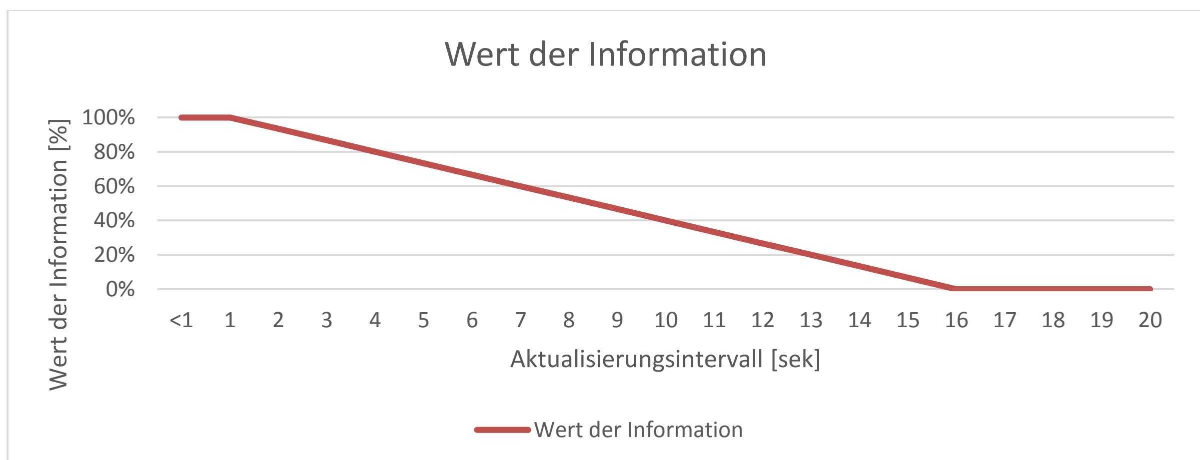

Diagramm 9: Informationswert in Abhängigkeit der Aktualität

Diagramm 9 legt damit fest, dass die Informationen den vollen Wert haben, wenn sie innerhalb einer Sekunde nach ihrer Ankunft am System verarbeitet werden. Da das Sendeintervall ebenfalls bei einer Sekunde liegt, entspricht dies einer unmittelbaren Rückmeldung des Systems. Jede abgewichene Sekunde senkt dabei den Informationswert um rund 6,7 Prozent (1/15), was einem linearen Wertverlust entspricht. Benötigt Spark ein Aktualisierungsintervall von mehr als 15 Sekunden, definiert diese Kurve den Wert der Informationen als nutzlos.

Nachfolgend werden die in Kapitel 9.4 erläuterten Modelle auf ihre Belastbarkeit untersucht. Es werden Testläufe mit unterschiedlichen Batchintervallen durchgeführt, um eine unterste Grenze für jedes Modell zu finden, bis zu der die Daten dauerhaft und rechtzeitig verarbeitet

werden. Die Spark-Weboberfläche[54] bietet hierzu einen speziellen Streaming-Tab, über den Informationen zu den Verarbeitungszeiten von Streaming-Applikationen eingesehen werden können. Eine dauerhafte und problemlose Datenstromverarbeitung ist gegeben, wenn auch nach Ablauf der Window-Zeitspanne noch keine Verzögerung im Ablauf zu erkennen ist. Dies ist genau dann der Fall, wenn sowohl die Scheduling-Verzögerung null, als auch die Verarbeitungsdauer kleiner als das Batchintervall ist und nicht dauerhaft streng monoton ansteigt.

Arithmetisches Mittel

Nachfolgend finden sich die Benchmarks zur Ermittlung des optimalen Batchintervalls für das Modell, welches die Ausreißererkennung mittels Berechnung des arithmetischen Mittels durchführt. Da sich die Mittelwertberechnung hier auf eine einfache Division zweier Summen stützt und jede Aufsummierung über die parallele *reduceByKeyAndWindow* Transformation durchgeführt wird, kann hier von einer sehr kurzen Abarbeitungszeit ausgegangen werden. Die Ausnutzung der zur Verfügung stehenden Arbeitsinstanzen sollte deshalb eine Datenverarbeitung nahe des optimalen Batchintervalls von einer Sekunde ermöglichen. Ein erster Benchmark wurde zunächst mit einem Batchintervall von zwei Sekunden durchgeführt.

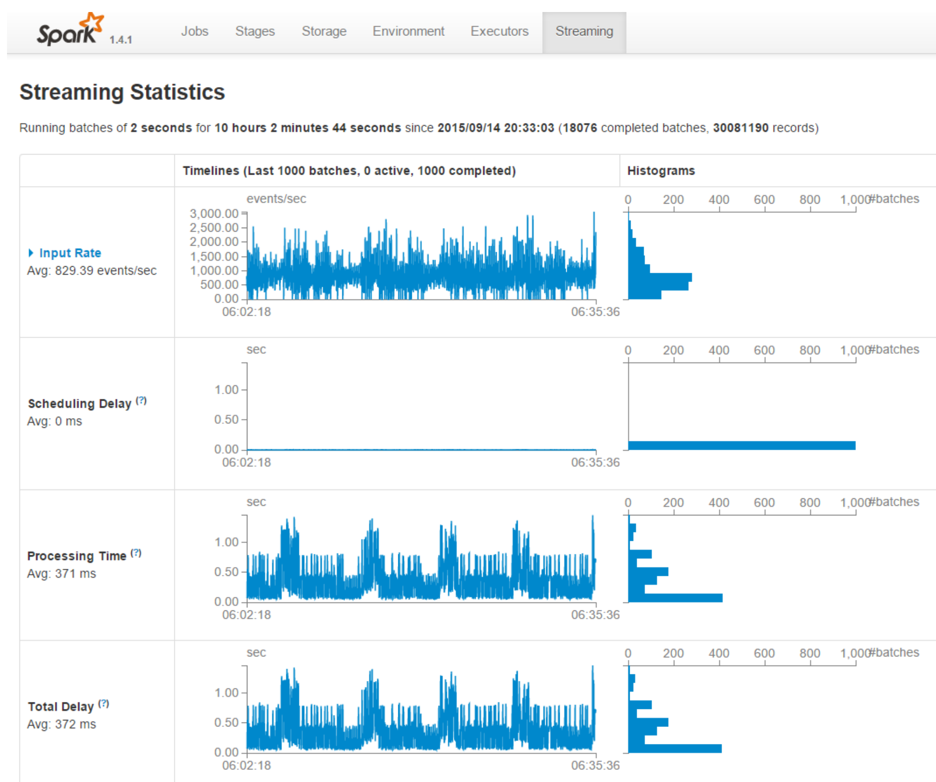

Abbildung 37: Benchmark: Mittelwert über arithm. Mittel [Batchintervall 2s]

Wie Abbildung 37 zeigt, ermöglicht diese Größe einen reibungslosen Betrieb der Datenauswertung. Die Verarbeitung benötigt sowohl nach einer, als auch nach mehreren Stunden, rund

[54] Spark-Version 1.4.1

370ms, was die Vermutung aufkommen lässt, dass auch eine Sekunde als Batchintervall möglich sein könnte, da selbst eine Verdopplung der Verarbeitungszeit noch im Rahmen dieses Batchintervalls wäre. Ein anschließender Benchmark wurde deshalb mit diesem minimalen Intervall durchgeführt. Hier kam das System jedoch an seine Grenzen, sodass die Verarbeitungszeit streng monoton anstieg. Nach fünf Stunden Laufzeit der Anwendung betrug diese bereits knapp 800ms, Tendenz nach wie vor steigend. Ein Batchintervall von einer Sekunde ist deshalb für diese Anwendung und Datenmenge auf einem solchen Cluster als zu niedrig einzustufen. Die optimale Einstellung bietet daher ein Batchintervall von zwei Sekunden. Auch der Wert der Informationen ist mit über 93% noch hoch genug, um die Ausreißererkennung tatsächlich mit dieser Konfiguration in Betrieb nehmen zu können. Abbildung 38 zeigt einen stichprobenartigen Vergleich der Verarbeitungszeiten bei einem Batchintervall von einer (links) und zwei (rechts) Sekunden.

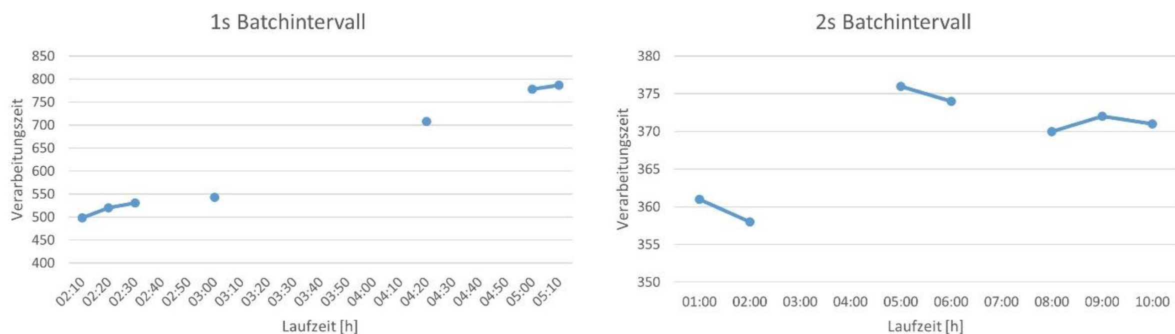

Abbildung 38: Vergleich der Verarbeitungszeiten beim arithm. Mittel

Median

Dieses Kapitel beinhaltet die Benchmarks zum Modell, welches die Medianbestimmung zur Ausreißererkennung verwendet. Wie bereits in Kapitel 9.4.2 behandelt, stützt sich die Berechnung des Medians auf eine sortierte Folge von Daten. Da sich diese Sortierung in Spark bislang nur sequentiell auf einem der vorhandenen Workern durchführen lässt, wird hier kaum vom Parallelismus der Plattform profitiert. Diese sequentielle Medianberechnung wirkt sich damit extrem auf die Verarbeitungszeit eines Batches aus. Aus diesem Grund wurde dieser Benchmark mit einem sehr großen Batchintervall von 30 Sekunden begonnen. Wie Abbildung 39 aufzeigt, ist das Cluster bereits bei diesem Batchintervall überfordert: Es kann die Datenmenge des Windows nicht rechtzeitig verarbeiten, ehe die Slide-Duration eine erneute Window-Berechnung mit den bis dorthin neu angekommenen Daten auslöst. Die Überlastung des Systems ist hierbei unter anderem auf die logarithmische Laufzeit von *O(n log n)* der Stromverbrauchs-Sortierung zurückzuführen. Nachdem die Verarbeitung einmal länger dauert, als das Aktualisierungsintervall erlauben würde, kommt es zu einem Datenstau und einer immer größer werdenden Datenmenge, die es zu verarbeiten gilt. Schafft es das System nicht,

wie hier wegen der Sortierung von immer mehr Elementen, durch anschließend kürzere Verarbeitungszeiten diesen Datenstau abzubauen, steigt die Abarbeitungsdauer exponentiell an und das System ist überfordert.

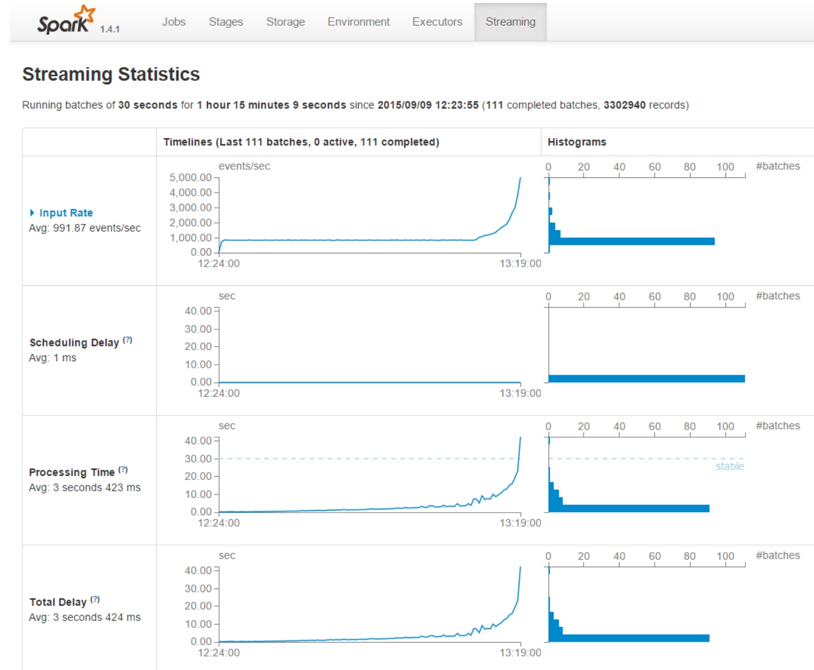

Abbildung 39: Benchmark zum Median-Modell

Bei kleineren Batchintervallen war das System entsprechend bereits nach rund 25-35 Minuten Laufzeit überfordert, wie in Abbildung 40 gezeigt.

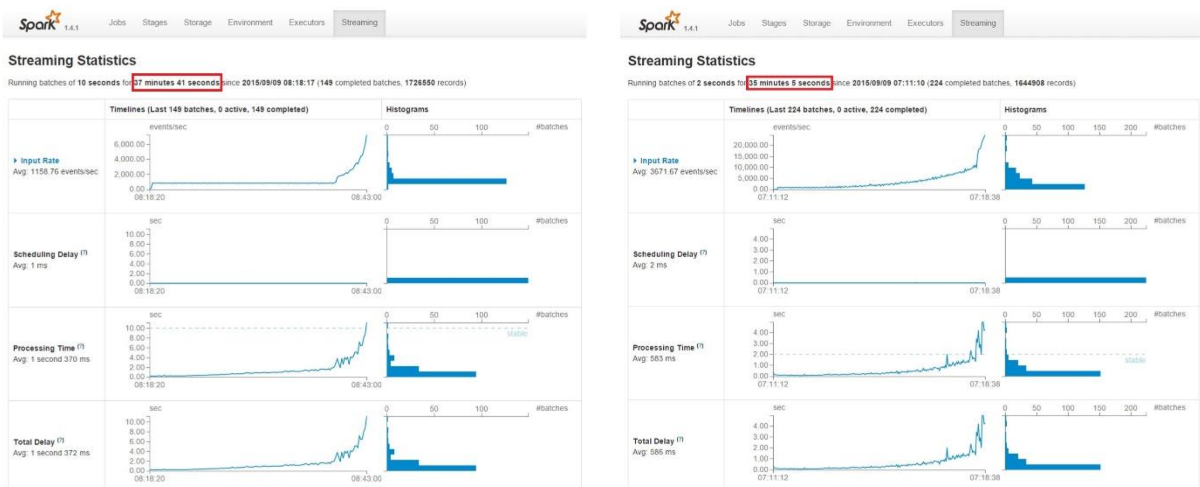

Abbildung 40: Benchmark zum Median-Modell: 10 sec./2 sec. Batchintervall

Selbst wenn es mit diesem Modell gelingen würde, die Daten des 1h-Windows mit einem Batchintervall von 30 Sekunden, z. B. durch Einsatz eines stärkeren Clusters, verarbeiten zu können, wäre der Wert der Informationen bei Beachtung von Diagramm 9 längst verloren. Eine Umsetzung dieses Modells macht somit für eine Echtzeitanwendung mit Spark keinen Sinn und es muss auf eine der beiden anderen Möglichkeiten umgestiegen werden.

Arithmetisches Mittel der Mediane

Diese Variante kombiniert beide vorherigen Verfahren zur Mittelwertbestimmung. Das gewichtete arithmetische Mittel wird hier als Mittelwertbestimmung innerhalb des Windows verwendet. Gleichzeitig wird jedoch auch eine neue Datenaggregation auf Batchebene mithilfe der Medianbestimmung durchgeführt. Dadurch können auf Batchebene ausreißerunabhängige Mittelwerte zur parallelen Verrechnung auf Windowebene übergeben werden. Voraussetzung ist ein relativ großes Batchintervall, damit die Datenaggregation auf dieser Ebene überhaupt Sinn macht. Bei einem Aktualisierungsintervall von 1-2 Sekunden entspricht dieses Modell dem der Umsetzung mit arithmetischem Mittel, da die Batchverarbeitung aufgrund der kleinen Datenbasis (0-1 Elemente) keine Daten ausselektieren kann. Bei etwas größerem Intervall (5-10 Sekunden) kann es dennoch passieren, dass Ausreißer in den Daten nach wie vor mit in die Medianbestimmung auf Batchebene einfließen, was das Gesamtergebnis ebenfalls nicht ganz ausreißerunempfindlich machen würde. Ausreißerunempfindliche Ergebnisse können etwa ab einem Batchintervall von 10 Sekunden (>= 5 Datenpunkte je Batch und Plug) erreicht werden. Zudem wird die anfallende Verarbeitung der Datenpunkte auf zwei Stufen verteilt: Die Medianbestimmung kümmert sich auf Batchebene für eine möglichst ausreißerunabhängige Bestimmung des Mittelwerts, während diese Teilergebnisse anschließend per gewichtetem arithmetischem Mittel windowbasiert zu einem Gesamt-Mittelwert verrechnet werden. Diagramm 10 stellt die Abhängigkeit zwischen diesen beiden Ebenen dar: Je mehr Daten voraggregiert werden, desto weniger Daten müssen später für den Mittelwert des Windows beachtet werden.

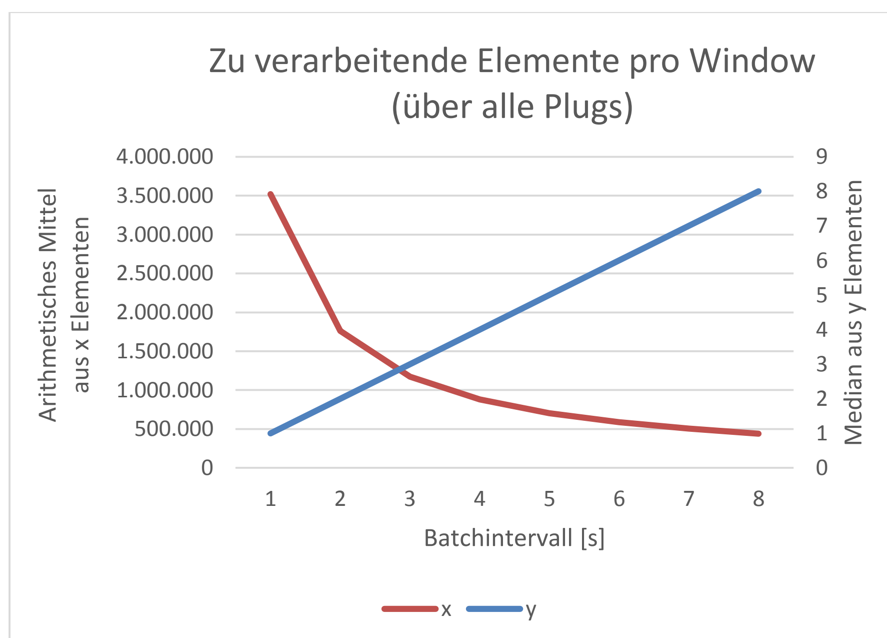

Diagramm 10: Zusammenhang zwischen Batch- und Window-Datengröße

Sollte ein Cluster überlastet sein, kann je nach Auslöser reagiert werden:

- Es werden zu viele Daten sequentiell auf Batchebene untersucht.
    - Ursache: Batchintervall zu groß
    - Lösung: Batchintervall verkleinern, um mehr Daten parallel zu verarbeiten
- Der parallele Teil hat zu viele Daten zum Verarbeiten
    - Ursache: Batchintervall zu klein (~1 Sekunde)
    - Lösung: Intervall erhöhen, um mehr Daten auf Batchebene aggregieren zu können
- Das Ergebnis ist nicht ausreißerunabhängig
    - Ursache: Batchintervall ist zu klein (< 10 Sekunden)
    - Lösung: Intervall erhöhen, um über die ausreißerunempfindlichere Medianberechnung die Daten voraggregieren zu lassen.

Das Batchintervall muss hier je nach Aufgabenstellung individuell festgelegt werden: Auch sehr kleine oder große Batchintervalle können dieser Anwendung zugeteilt werden. Das Minimum stellt für das eingesetzte Cluster ein Batchintervall von zwei Sekunden dar. Die Abarbeitung findet hier analog zur Umsetzung mit arithmetischem Mittel statt, da die Verarbeitung auf Batchebene keine Aggregation der Daten vornimmt. Entsprechend gleicht das Benchmarkergebnis dem des ersten Modells in Abbildung 37. Die Verarbeitungsdauer war hier jedoch, aufgrund der batchbasierten Zwischenberechnung, um rund 40-50 Millisekunden höher als im windowbasierten Modell. Ein weiterer Grenzwert-Benchmark wurde mit einem Batchintervall von 30 Sekunden durchgeführt: Hier war die Verarbeitungszeit mit rund 82 Millisekunden extrem gering, was durch die gute Daten-Voraggregation auf Batchebene (rund 15 Datenpunkte je Plug und Batch) zu begründen ist. Abbildung 41 zeigt die Ergebnisse der beiden Benchmarks.

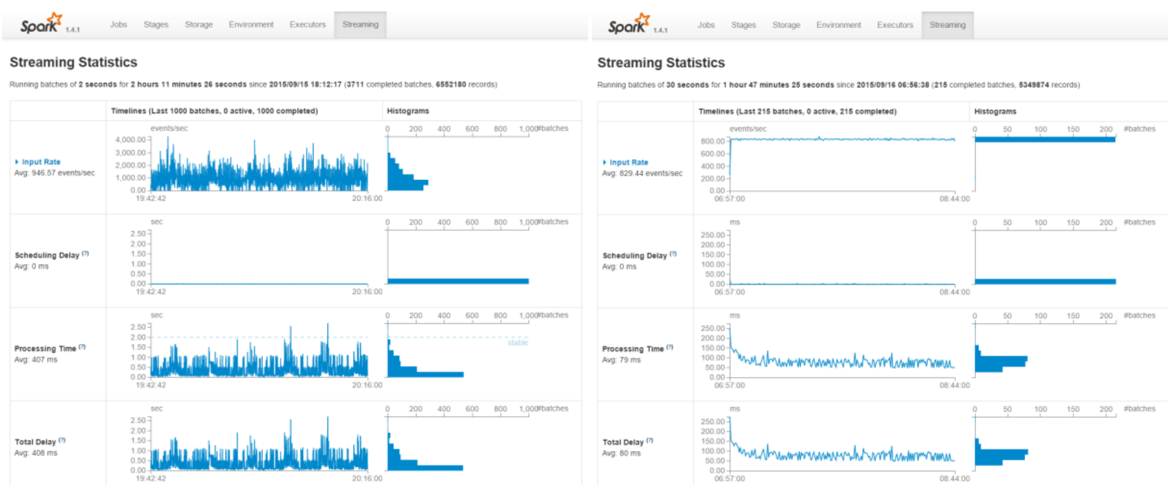

Abbildung 41: Benchmark arithm. Mittel der Mediane: 2 sec./30 sec. Batchintervall

9.5.2 Diskussion der einzelnen Lösungen

Dieses Kapitel beschäftigt sich mit dem Aufzeigen von Vor- und Nachteilen der einzelnen Modelle und soll somit die Entscheidungsfindung für eine dieser Umsetzungsmöglichkeiten unterstützen. Für jedes Modell werden Pro- und Kontrapunkte diskutiert.

Arithmetisches Mittel

Dieser Lösungsansatz profitiert sehr von der parallelen Abarbeitung der aufgerufenen Transformationen. Die *reduceByKeyAndWindow* Transformationen ermöglichen eine partitionsbezogene, parallele Datenverarbeitung und nutzen somit die Clusterressourcen optimal aus. Dadurch ist eine dauerhafte Ausführung der Ausreißererkennung auch mit einem sehr kleinen Batchintervall von zwei Sekunden noch problemlos möglich.

Ein großer Nachteil dieser Variante ist jedoch der Einfluss von tatsächlichen Ausreißern innerhalb der Messwerte: Da alle Werte mit gleicher Gewichtung in die Mittelwert-Berechnung miteinfließen, wirken sich besonders große Ausreißer stark auf das Ergebnis aus. Es kann also passieren, dass wenige, aber sehr große Ausreißer den Gesamtdurchschnitt relativ schnell und stark verändern. Dadurch kann es passieren, dass kleinere Ausreißer plötzlich nicht mehr als solche erkannt werden. Eine stabilere Lösung, besonders beim Auftreten von groben Ausreißern, bietet hier die Verwendung des Medians einer Messreihe.

Median

Die Lösungsmöglichkeit über die Berechnung von Medianen innerhalb von Zeitfenstern verhindert den großen Nachteil bei der Berechnung des arithmetischen Mittels: Durch die Sortierung nach Verbrauch werden die kleinsten und größten Werte an den Rand der Menge verschoben, was eine Miteinberechnung in den Mittelwert verhindert. Dadurch wird dessen Wert nicht durch die Ausreißer beeinflusst, auch nicht durch extrem große, wie es bei der Lösung mit arithmetischem Mittel der Fall ist.

Allerdings hat auch diese Methode einen extremen Nachteil bei der Umsetzung auf aktuellen Spark-Versionen: Durch das Fehlen paralleler Sortieralgorithmen und der rangbasierten Bestimmung von Elementen, ist eine verteilte Medianberechnung innerhalb einer Datenpartition nicht möglich. Stattdessen müssen die Daten des Windows per Java-Methode *Arrays.sort* sequentiell von einem Core geordnet werden. Deren Laufzeit hierfür wächst logarithmisch und sorgt schnell für Datenstaus, da das Batchintervall bald zu klein für die neu ankommende Datenmenge wird und damit die Verarbeitungszeit exponentiell anwächst. Die Verarbeitung eines einstündigen Windows ist bei einer Datenmenge, wie sie in diesem Prototyp anfällt, selbst bei einem Batchintervall von 30 Sekunden noch nicht möglich.

Arithmetisches Mittel der Mediane

Als Kompromiss der beiden vorherigen Lösungsmöglichkeiten werden bei dieser Umsetzung die Vorteile der beiden Modelle in ein neues vereinigt, um gleichzeitig deren Nachteile zu kompensieren. Dieses Modell führt, im Gegensatz zu den anderen beiden, eine zusätzliche Datenaggregation auf Batchebene durch. Da hier nur ein Bruchteil der Windowdaten vorverarbeitet werden muss, wird hierfür die ausreißerunempfindlichere Medianbestimmung verwendet. Da bei kleinem Aktualisierungsintervall dennoch viele Batches zu einem Window gehören, eignet sich hierbei, für die tatsächliche Ausreißererkennung auf Windowbasis, die Verwendung des arithmetischen Mittels. Durch dessen Parallelität können dann die Ressourcen des gesamten Clusters verwendet werden, was gleichzeitig für eine effizientere Nutzung sorgt. Je nach Anforderungen an das System (Aktualität vs. Ausreißerunempfindlichkeit) kann auf einfachste Weise per Batchintervall eine zufriedenstellende Konstellation gefunden werden.

Leider hat auch diese Variante nicht nur Vorteile: Durch die Abhängigkeit zwischen Batch- und Windowverarbeitung muss hier ein Kompromiss für die Einstellung des Batchintervalls gefunden werden. Wird das Intervall zu klein gewählt, kann es passieren, dass Ausreißer trotz der Mittelwertbildung per Median mitbeachtet werden (Intervallgröße kleiner 10 Sekunden) und damit dieser Schritt der Voraggregation überflüssig wird. Noch gravierender ist ein Aktualisierungsintervall von einer bzw. zwei Sekunden: Hier besteht ein Batch aus maximal einem Datenpunkt pro Stromzähler, da hiermit eine einfache Geschwindigkeit realisiert wird. Damit entfällt die gesamte Medianbestimmung (da immer das aktuelle Element den Median repräsentiert) und die anschließende Berechnung des arithmetischen Mittels übernimmt die gesamte Datenverarbeitung. Diese Einstellung würde exakt das gleiche Ergebnis erzielen wie das erste Modell, doch die Laufzeit wäre wegen des Zwischenschritts der Batchaggregation sogar schlechter. Im Gegensatz zu einem kleinen Batchintervall würde ein Großes zwar für eine ausreißerunabhängigere (durch die Aggregation auf Batchebene) und trotzdem auch gut parallelisierte (Windowverarbeitung) Lösung sorgen, jedoch wäre der in Diagramm 9 aufgezeigte Informationswert minimal oder gar null, da das Batchintervall hierfür bei unter 15 Sekunden liegen müsste.

9.6 Deployment: Festlegung eines Modells

Nachdem die in Kapitel 9.4 vorgestellten Modelle später unter 9.5 mithilfe von Benchmarks und Diskussion zu Vor- und Nachteilen bewertet werden, beschäftigt sich dieser Schritt des CRISP-DM-Modells (Kapitel 2.1) mit dem abschließenden Festlegen und Produktivsetzen der am besten geeigneten Lösung. Da es sich hierbei lediglich um Prototypen handelt, mit denen die Funktionsweise von Spark und dessen Streaming Bibliothek aufgezeigt werden soll, ist ein produktiver Einsatz dieser Applikationen nicht weiter vorgesehen. Dieses Kapitel beschäftigt sich deshalb lediglich mit der finalen Auswahl des am besten zur Aufgabenstellung passenden Modells.

Wie bereits im vorherigen Kapitel aufgezeigt, hat jede der drei implementierten Modellvarianten ihre Vor- und Nachteile. Ein Modell kann hierüber und über die Benchmarkergebnisse jedoch als unpassend ausgefiltert werden: Da sie die Parallelität und Ressourcenverteilung von Spark nur bedingt ausnutzt, ist die Ausreißererkennung per Medianbestimmung auf Windowebene für diesen Zweck und die Umsetzung mit Spark ungeeignet.

Die Umsetzung über eine windowbasierte Ausreißererkennung per arithmetischem Mittel nutzt hier die Ressourcen eines Spark-Clusters optimal aus: Alle rechenintensiven Operationen werden über parallele und schlüsselbasierte Transformationen ausgeführt. Durch diese gute Parallelität ist die Verarbeitung der Datenpunkte eines einstündigen Windows problemlos ab einem Batchintervall von zwei Sekunden möglich. Der Nachteil dieser Lösung liegt jedoch in der Empfindlichkeit gegenüber Ausreißern bei der Berechnung der Mittelwerte. Hierdurch kann es passieren, dass durch Miteinberechnung von Ausreißern in den Daten der Mittelwert so hoch liegt, dass kleinere Ausreißer in den Daten vom Modell nicht als solche eingestuft werden.

Eine gute Ausreißerunempfindlichkeit zeigt das dritte Modell: Durch die Voraggregation per Medianbestimmung auf Batchebene werden hierbei, je nach Größe des Batchintervalls, solche Ausreißer bereits ausgefiltert. Dadurch kann die windowbasierte Mittelwertberechnung ebenfalls einen recht ausreißerunempfindlichen Gesamtmittelwert bestimmen. Da dieses Modell ebenfalls mit sehr kleinen und großen Batchintervallen umgehen kann, eignet sich dieses optimal für die Umsetzung der einleitend vorgegebenen Aufgabenstellung. Je nach Anforderung an das System kann das Batchintervall festgelegt werden und somit für jede beliebige Priorität eingesetzt werden – egal ob Aktualität (kleines Batchintervall), Ausreißerunempfindlichkeit (großes Batchintervall) oder Kompromiss aus beiden.

Für die Umsetzung der oben beschriebenen Aufgabe zur Analyse der DEBS-Daten und anhand der Wertungsfunktion in Diagramm 9 empfiehlt sich hierfür ein Batchintervall von 6 Sekunden. Damit werden die Daten recht zeitnah (Wert: ~67%) ausgewertet und durch das Miteinbeziehen von meist drei Datenpunkten zur Medianbestimmung auf Batchebene, ein guter Kompromiss zwischen Geschwindigkeit und Ausreißerstabilität gefunden. Abbildung 42 zeigt die Performance dieser Lösung bei einem Batchintervall von 6 Sekunden.

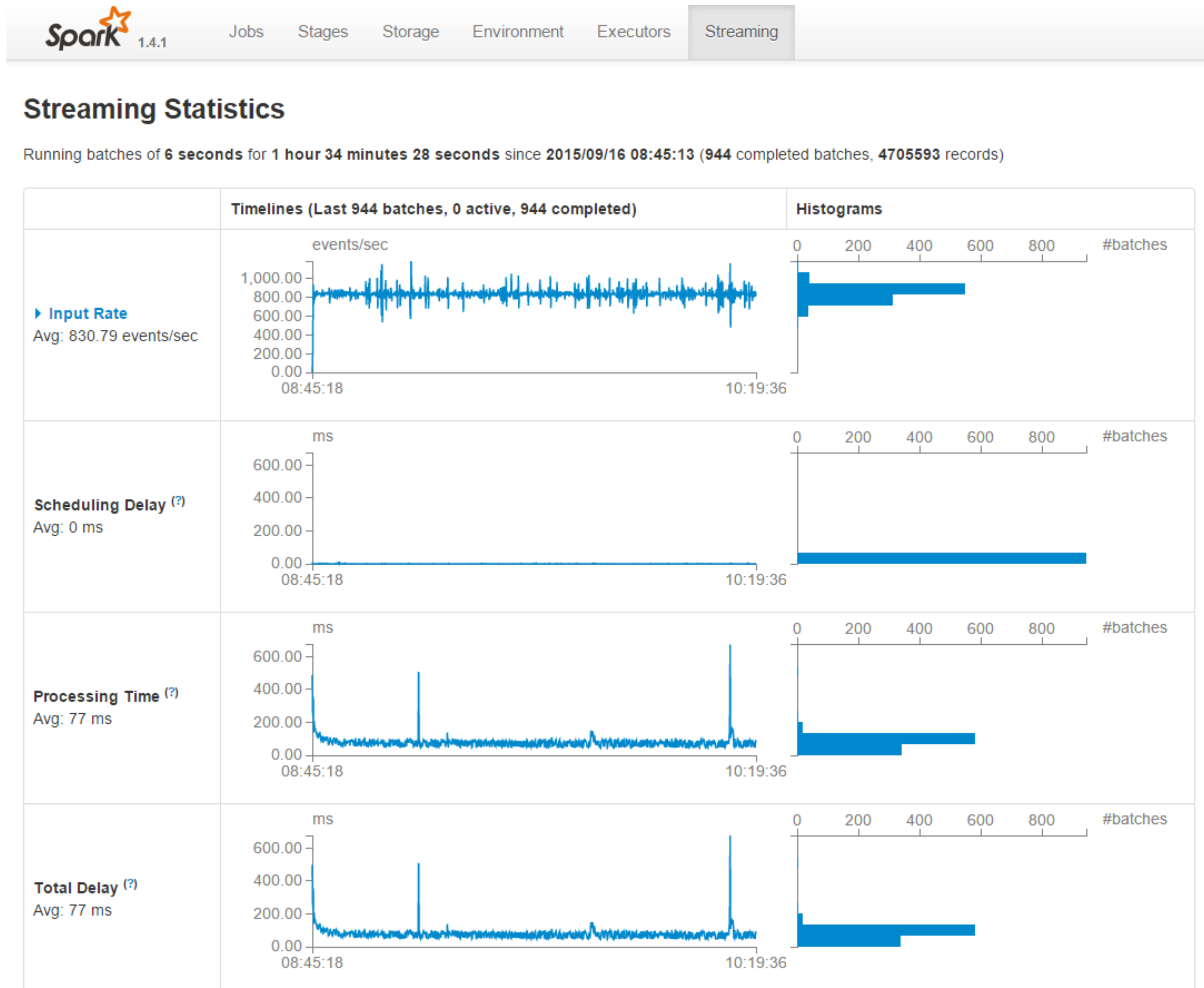

Abbildung 42: Finale Modellauswahl, Batchintervall 6 Sekunden

10 Fazit

In der vorliegenden Arbeit wurde das quelloffene Framework Apache Spark auf dessen Einsatzfähigkeit zur verteilten Datenverarbeitung und Analyse von Streaming Data untersucht. Nachdem zunächst theoretische Grundlagen zu den Themen rund um die Datenverarbeitung und Spark selbst erläutert wurden, folgten praktische Untersuchungen über die Erstellung und Arbeitsweise von Spark-Applikationen. Den Hauptteil der Arbeit bildete die Modellierung von Prototypen zur Echtzeituntersuchung von Datenströmen. Hierbei wurden Daten der DEBS Grand Challenge 2014 als Stream an ein Spark Cluster geschickt und dort auf Ausreißer untersucht. Die folgenden Abschnitte fassen die Erkenntnisse der durchgeführten Aufgaben zusammen.

10.1 Aufsetzen von Clustern

Was als sehr intuitiv und schnell erlernbar eingestuft werden kann, ist das Aufsetzen von Spark Clustern. Durch den geringen Konfigurationsaufwand und dem Aufruf einfacher Skripts ist es möglich, einen solchen Zusammenschluss binnen weniger Minuten durchführen zu können. Die einfachste Variante war hierbei, im Gegensatz zu jeder Erwartung, ein Cluster auf EC2 Instanzen der Amazon Webservices zu erstellen. Hier können durch nur einen Skriptaufruf Cluster aufgesetzt, gestartet, gestoppt oder entfernt werden. Es benötigt neben Unix-Grundkenntnissen kein spezielles Sachwissen.

10.2 Abstraktionsmodell & Erstellen von Applikationen

In Spark werden Datenpartitionen in Form von RDDs im Cluster verteilt und im Arbeitsspeicher bzw. auf Platte gehalten. Diese Datenabstraktion ist auf unterster Ebene definiert, muss aber vom Benutzer vor Applikationserstellung nicht ins Detail studiert werden. Stattdessen sorgt die höherschichtige API für die Bereitstellung von Manipulationsoperationen via Transformationen und Aktionen, die dann bei Ausführung auf RDDs angewandt werden. Der Benutzer muss sich hier um nichts weiter kümmern, Kenntnisse zur Partitionslogik von Spark bieten aber Vorteile im Hinblick auf eine Anwendungsoptimierung. In der Arbeit wurde mit der Java-API gearbeitet, zusätzlich werden aber auch Scala und Python als Programmiersprachen unterstützt. Das Programmieren solcher Anwendungen ist aufgrund der höherschichtigen API sehr intuitiv und bereits mit grundlegenden Programmierkenntnissen möglich.

10.3 Ausführung von Applikationen & Debugging

Das Ausführen von Applikationen ist durch Erstellung einer JAR-Datei mit Abhängigkeiten möglich. Diese muss anschließend auf den Masterknoten kopiert und von dort aus über ein spezielles submit-Skript gestartet werden. Dieses Skript kann mit nur wenigen Parametern aufgerufen werden, um die Ausführung einer Anwendung im Spark-Cluster anzustoßen. Hierdurch kann auch der zu verwendende Cluster-Manager festgelegt werden. Nach einigen Testläufen mit unterschiedlichen Parametern kommt ein Anwender auch hier recht schnell zum Ziel. Eine relativ große Hürde bei der Entwicklung von Spark-Applikationen ist das Fehlen von Debugging-Möglichkeiten. Dadurch, dass die Anwendungen direkt auf den Master eines Clusters kopiert und von dort aus gestartet werden müssen, hat der Benutzer keine Möglichkeit ein Debugging auf Codeebene durchführen zu können. Die einzige Variante, Informationen zum Ablauf eines Programmes zu bekommen ist entweder die fest einprogrammierte Ausgabe von Statusmeldungen auf Konsole oder ein Einblick in die ausgeführten Jobs und Stages per Weboberfläche. Hier ist es für einen Einsteiger jedoch anfangs recht schwierig sich zurechtzufinden.

10.4 Bugs & Fehlermeldungen

Sehr selten kam es auch vor, dass bei der Ausführung von Applikationen Probleme aufgrund von Bugs auftraten. Diese kamen vor allem bei Verwendung neuester Spark-Releases vor. Hier ist es oftmals schwer den Grund eines solchen Problems zu finden, da die Ausführung der betroffenen Anwendung meist in einer unverständlichen Fehlermeldung endet. Nach einiger Zeit entwickelt sich aber ein Gefühl dafür, wie und wo nach Fehlern gesucht werden muss. Sehr hilfreich für die Suche nach Bugs ist das Ticketsystem, über welches Bugs eingereicht, diskutiert und deren Behebungen aufgezeigt werden. Beim Auftreten von unverständlichen Fehlermeldungen kann ein Blick in dieses Ticketsystem durchaus die Fehlersuche erheblich verkürzen.

10.5 Unterstützte Datentypen

Durch die Unterstützung vieler gängiger Dateisysteme und Datentypen sollte für jeden Anwender ein passendes Datenformat zum Auslesen bzw. Speichern von Informationen vorhanden sein. Auch ganze Plattformen lassen sich mithilfe von passenden Konnektoren verbinden. So ist eine Zusammenarbeit von Spark und der NoSQL-Datenbank Apache Cassandra ebenfalls möglich. Auch die Protokolle der beiden persistenten Datenspeicher HDFS und Amazon S3 werden nativ unterstützt. Das Incubator-Projekt Apache Zeppelin, welches besonders für visuelle Datenauswertungen geeignet ist, wurde ebenfalls untersucht. Mittels integriertem

Spark-Interpreter ist es direkt nach der Installation möglich, per Scala-API geschriebenen Anwendungscode von dort aus als Spark-Job auszuführen. Um solche Kooperationen über Konnektoren umzusetzen, bedarf es den nötigen Grundkenntnissen über das verwendete Spark Cluster und einem Einlesen in die Verwendung der Partnerplattform. Zudem ist auf eine Versionskompatibilität zu achten, denn oftmals sind diese Adapter für spezielle Releases der einzelnen Lösungen konzipiert, z. B. Spark 2.x mit Hadoop 2.4 und Cassandra 2.1.

10.6 Optimierungsmöglichkeiten

Im Laufe der Arbeit wurden an den entsprechenden Stellen auch immer wieder Optimierungsmöglichkeiten aufgezeigt, von denen eine Applikation profitieren kann. Hier waren es besonders auch grobe und schnelle Einstellungen wie Partitionsanzahl, Arbeitsspeicherzuweisung usw., die einen positiven Effekt auf die Ausführung von Applikationen hatten. Es stehen dem Anwender auch zahlreiche andere Einstellungen zur Verfügung, mit denen detailliertere Vorgaben getroffen werden können. Diese übersteigen allerdings das Grundwissen und benötigen tiefergehende Recherchen. Wichtig zu notieren ist an dieser Stelle dennoch, dass Spark sehr viele Konfigurationsmöglichkeiten bietet, mit welchen sich erfahrene Anwender auseinandersetzen können.

10.7 Einsatzfähigkeit von Spark für die Streaming Data Analyse

Spark wurde beim Modellieren der Prototypen mithilfe seiner Streaming Bibliothek für das Entgegennehmen, Verarbeiten und Analysieren der Stromzählerdaten eingesetzt. Durch die Definition des Batchintervalls konnte die Zeitspanne festgelegt werden, in der Daten zu einer Menge zusammengefasst und nach Ablauf gesamtheitlich verarbeitet wurden. Da die Ausreißer über Mittelwertberechnung auf einstündigen Zeitfenstern bestimmt werden sollten, wurde zusätzlich ein Window definiert, welches sich wiederum aus den einzelnen Batches, die in dieser Zeit anfielen, zusammensetzt und nach Ablauf der Slide-Duration aktualisiert wird. Spark bietet hier durch seine Streaming Bibliothek Konfigurationsmöglichkeiten rund um die Verarbeitung von Streaming Data.

Das Umsetzen der Spark-Applikationen war mit den im Laufe der Arbeit angeeigneten Kenntnissen problemlos möglich. Eine Vorselektion der Daten wurde mithilfe einfacher Transformationsaufrufe durchgeführt. Durch das Bilden von Schlüssel-Wert Paaren konnten die ankommenden Daten umgehend nach ihrer Stromzählerkennung partitioniert werden. Da die Analysen ebenfalls auf Stromzählerbasis durchgeführt wurden, konnten die Reduktions-Transfor-

mationen zum parallelen und schlüsselbasierten Zusammenfassen der Daten verwendet werden. Dies ermöglichte auch eine optimale Ressourcenausnutzung, da Spark besonders effizient auf bereits partitionierten Daten arbeiten kann.

Bei der Prototyp-Entwicklung wurde unter anderem auch eine Lösung erstellt, welche den verteilten Aufbau und die parallele Datenverarbeitung von Spark nicht optimal ausnutzte. Durch die Verwendung von nicht-parallelisierbaren Operationen sank hier die Performanz um ein Vielfaches und machten den Einsatz in der Echtzeitverarbeitung unmöglich.

Lösungen, die sich aber komplett auf parallele Transformationen abbilden lassen, ermöglichen eine optimale Ressourcenausnutzung und eine Datenverarbeitung in Millisekundenbereich. Somit wurden bei Beachtung dieser Erfahrung jene Ausführungsprobleme vermieden, sodass bereits in einem Cluster, bestehend aus fünf Standardrechnern, ein Batch- und Datenaktualisierungs-Intervall von zwei Sekunden möglich war. Beim Einsatz von mehr bzw. stärkeren Rechnern und einer Umsetzung mit parallelen Transformationen wäre hier auch ein optimales Intervall von einer Sekunde problemlos möglich gewesen.

10.8 Produktiver Einsatz von Spark

Insgesamt kann festgestellt werden, dass Apache Spark durchaus für einen produktiven Betrieb im Bereich der verteilten (Echtzeit-) Datenverarbeitung geeignet ist. Bereits mit grundlegenden Kenntnissen zur Programmierung, welche durch entsprechende Literatur oder Dokumentationen schnell angeeignet sind, ist die Erstellung von verteilten Applikationen möglich. Dank der Speicherabstraktion RDD auf unterer Ebene und der höherschichtigen API, die Transformationen und Aktionen auf diesen Datenpartitionen definiert, ist eine effektive und parallele Datenverarbeitung schnell umgesetzt. Auch wenn neue Releases ggf. aufgrund möglicher Fehler anfangs noch gemieden werden sollten, arbeiten vorherige Versionen insgesamt sehr zuverlässig. Im Laufe der Arbeit kam es bei fehlerfreiem Programmcode nie zu einem unerklärlichen Abbruch der Ausführung. Wenn es Probleme gab, dann lediglich der Art, dass die Datenmenge zu groß wurde bzw. zu schnell verarbeitet werden sollte. Dies sollte jedoch bei einer gründlichen Evaluation vor Produktivsetzung festgestellt und behoben werden können.

10.9 Persönliches Resümee & Ausblick

Natürlich blieb es bei dieser Arbeit nicht aus, dass ich an der einen oder anderen Stelle auf Schwierigkeiten gestoßen bin. Auch wenn ich im Laufe der Thesis den Funktionsumfang und damit auch die Fähigkeiten von Spark recht gut kennenlernen konnte, fällt es mir nach wie vor schwer, das Zusammenspiel mit anderen Hadoop-Komponenten vorzustellen. Hierfür benötigt es einfach tiefgründiger Einarbeitung in die weiteren Plattformen, besonders in Form von praktischen Erfahrungen. Beim Umsetzen von Aufgaben musste ich auch oftmals feststellen, dass ich mein Vorhaben nicht auf Anhieb durch Transformationsfolgen ausdrücken konnte und deshalb entsprechend Tutorials und die Spark-Dokumentation zu Rate ziehen musste. Bei Problemen dieser Art half es auch oft, kurz Abstand von der aktuellen Umsetzung zu nehmen und anschließend mit einem neuen Anlauf fortzufahren.

Besonders Probleme in der Programmierung traten jedoch im Laufe der Arbeit immer seltener auf, da ich in der Zeit sehr viele Erfahrungen beim Umsetzen praktischer Beispiele machte. Entsprechend machbar war dann nach einer gründlichen Einarbeitung die Modellierung der Prototypen zur Ausreißererkennung. Hier konnten bisherige Kenntnisse optimal eingebracht und überprüft werden. Durch die Orientierung an der DEBS Grand Challenge 2014 war ein roter Faden vorgegeben, an dem ich mich bei der Implementierung halten konnte.

Insgesamt war das Arbeiten mit Apache Spark für mich sehr spannend und abwechslungsreich, da es sich hierbei um ein mir unbekanntes Aufgabengebiet handelte. Somit konnte ich sehr schnell viele neue Erfahrungen, sowohl zu theoretischen Grundlagen der Datenverarbeitung als auch zum praktischen Arbeiten mit dem Framework, sammeln. Dies motivierte mich im Laufe der vergangenen Monate stets dazu, neues auszuprobieren. Im Laufe der Arbeit wuchs mein Interesse an diesem Sachgebiet stetig an, sodass ich mir auch sehr gut vorstellen kann, in diesem Bereich weiterzuarbeiten.

Neben den in der Arbeit vorgestellten Streaming und MLlib Bibliotheken, bestehen aktuelle Spark-Versionen aus weiteren Komponenten, welche den Funktionsumfang von Spark vergrößern. Eine Untersuchung dieser wäre eine interessante Fortsetzung der Thesis.
Außerdem wäre mit dem Apache Incubator-Projekt Zeppelin eine Applikationserstellung mit Ergebnisvisualisierung möglich. Aufgrund des Umfangs und der Bearbeitungsdauer der Arbeit konnte ich diese neue Plattform aber leider nicht ausgiebiger untersuchen. Es wäre beispielsweise auch interessant, die Ausreißererkennung, wie sie im Hauptteil der Arbeit modelliert wird, auf Zeppelin zu portieren und eine entsprechende Visualisierung der Daten und deren Ausreißer durchzuführen.

XI Literaturverzeichnis

[1] Facebook, *Über Facebook.* Available: https://www.facebook.com/FacebookDeutschland/info?tab=page_info (2015, Apr. 27).

[2] Twitter, *About.* Available: https://about.twitter.com/milestones (2015, Apr. 27).

[3] iAdam1n, *Timeline.* Available: https://theiphonewiki.com/wiki/Timeline (2015, Apr. 28).

[4] D. Morrill, *Announcing the Android 1.0 SDK, release 1* (2015, Apr. 27).

[5] InternetLiveStats, *Twitter Usage Statistics.* Available: http://www.internetlivestats.com/twitter-statistics/#trend (2015, Apr. 27).

[6] M. Barlow, *Real-Time Big Data Analytics: Emerging Architecture*. Gravenstein Highway North, Sebastopol, CA: O'Reilly Media, 2013.

[7] U. M. Fayyad, *Advances in knowledge discovery and data mining*. Menlo Park, Calif.: AAAI Press; MIT Press, 1996.

[8] Rüdiger Wirth, "CRISP-DM: Towards a standard process model for data mining," in *Proceedings of the Fourth International Conference on the Practical Application of Knowledge Discovery and Data Mining*, 2000, pp. 29–39.

[9] D. L. Olson and D. Delen, *Advanced data mining techniques*. Berlin: Springer, 2008.

[10] I. Datameer, *What is Big Data?* Available: http://www.datameer.com/product/hadoop.html.

[11] D. Feinleib, *Big data bootcamp: What managers need to know to profit from the big data revolution*, 2014.

[12] J. Hurwitz, *Big data for dummies,* 1st ed. Indianapolis, Ind.: John Wiley & Sons, 2013.

[13] B. Ellis, *Real-time analytics: Techniques to analyze and visualize streaming data*. Indianapolis, IN: Wiley, 2014.

[14] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, http://doi.acm.org/10.1145/1327452.1327492, 2008.

[15] MapRAcademy, *Intro To MapReduce.* Available: https://www.youtube.com/watch?v=HFplUBeBhcM (2015, Apr. 27).

[16] Apache-Foundation, *Spark Lightning-fast cluster computing.* Available: https://spark.apache.org/ (2015, Apr. 27).

[17] N. Marz, *Big data: Principles and best practices of scalable realtime data systems*. [Place of publication not identified]: O'Reilly Media, 2013.

[18] X. Liu, N. Iftikhar, and X. Xie, "Survey of Real-time Processing Systems for Big Data," in *Proceedings of the 18th International Database Engineering & Applications Symposium*, New York, NY, USA: ACM, 2014, pp. 356–361.

[19] H. Wörn and U. Brinkschulte, *Echtzeitsysteme: Grundlagen, Funktionsweisen, Anwendungen*. Berlin [u.a.]: Springer, 2005.

[20] B. Fondermann, “Pflanzenpflege - Hadoop 2 als universelle Data Processing Platform,” *iX Developer*, no. 2, pp. 30–34, 2015.

[21] U. Seiler, “Zoo voller Gehege - Die wichtigsten Projekte der Hadoop-Community,” *iX Developer*, no. 2, pp. 36–43, 2015.

[22] Apache-Foundation, *Apache Sqoop.* Available: https://sqoop.apache.org/index.html (2015, Apr. 27).

[23] Cloudera, *Security for Hadoop.* Available: http://www.cloudera.com/content/cloudera/en/solutions/enterprise-solutions/security-for-hadoop.html (2015, Apr. 27).

[24] Ganglia, *Ganglia Monitoring System.* Available: http://ganglia.sourceforge.net/ (2015, Apr. 27).

[25] Nagios, *Nagios Is The Industry Standard In IT Infrastructure Monitoring.* Available: http://www.nagios.org/ (2015, Apr. 27).

[26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster Computing with Working Sets,” in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, Berkeley, CA, USA: USENIX Association, 2010, p. 10.

[27] H. Karau, *Learning Spark: Lightening fast data analysis.*

[28] M. Zaharia, and M. Chowdhury et al, *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.* Available: https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf (2015, Apr. 30).

[29] R. Strickland, *Cassandra high availability: Harness the power of Apache Cassandra to build scalable, fault-tolerant, and readily available applications.*

[30] teddyma, *Learn Cassandra.* Available: https://www.gitbook.com/book/teddyma/learn-cassandra/details (2015, Jun. 23).

[31] J. Hui, *How Cassandra Read, Persists Data and Maintain Consistency.* Available: http://jonathanhui.com/how-cassandra-read-persists-data-and-maintain-consistency (2015, Jun. 18).

[32] DataStax, *Calculating User Data Size.* Available: http://docs.datastax.com/en/cassandra/2.1/cassandra/planning/architecturePlanningUserData_t.html (2015, Jun. 27).

[33] DataStax, *Tuning Java Resources.* Available: http://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_tune_jvm_c.html (2015, Jun. 22).

[34] DataStax, *Spark Cassandra Connector.* Available: https://github.com/datastax/spark-cassandra-connector (2015, Jul. 06).

[35] Apache-Foundation, *Spark Streaming Programming Guide: Overview.* Available: http://spark.apache.org/docs/1.2.1/streaming-programming-guide.html (2015, Jul. 27).

[36] Z. Jerzak and H. Ziekow, “The DEBS 2014 Grand Challenge,” in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, New York, NY, USA: ACM, 2014, pp. 266–269.

[37] C. Mutschler, C. Löffler, N. Witt, T. Edelhäu\sser, and M. Philippsen, “Predictive Load Management in Smart Grid Environments,” in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, New York, NY, USA: ACM, 2014, pp. 282–287.

XII Anhang

Der Anhang dieser Arbeit beinhaltet relevante Zusatzinformationen, wie Auflistungen, Anleitungen, Beispiel-Programme u. v. m., was aber die Übersichtlichkeit des Berichts vermindert und dessen Rahmen gesprengt hätte. Entsprechend wird im Bericht an den betreffenden Stellen auf diesen Anhang referenziert. Anhang A enthält Anhänge zum Thema Spark, welches in Berichts-Kapitel 4 bis 8 behandelt wurde. Zusatzergänzungen zur Umsetzung des Prototyps zur Analyse von Streaming Data mit Spark (Kapitel 9) finden sich unter Anhang B, ebenfalls im Anhang.

A Apache Spark

Hier finden sich alle Anhänge zu den Spark-Kapiteln der Arbeit.

A.1 Datenquellen & Dateiformate

```
(1)   class ParseJson implements FlatMapFunction<Iterator<String>, Person>{
(2)     public Iterable<Person> call(Iterator<String> lines)
            throws Exception {
(3)       ArrayList<Person> people = new ArrayList<Person>();
(4)       ObjectMapper mapper = new ObjectMapper();
(5)       while (lines.hasNext()) {
(6)         String line = lines.next();
(7)         try {
(8)           people.add(mapper.readValue(line, Person.class));
(9)         } catch (Exception e) {
(10)          // skip records on failure
(11)        }
(12)      }
(13)      return people;
(14)    }
(15)  }
(16)  JavaRDD<String> input = sc.textFile("file.json");
(17)  JavaRDD<Person> result = input.mapPartitions(new ParseJson());
```

Codeblock 35: Auslesen und Deserialisieren von Daten im JSON-Format [27]

```
(1)   class WriteJson implements FlatMapFunction<Iterator<Person>, String>{
(2)     public Iterable<String> call(Iterator<Person> people)
            throws Exception {
(3)       ArrayList<String> text = new ArrayList<String>();
(4)       ObjectMapper mapper = new ObjectMapper();
(5)       while (people.hasNext()) {
(6)         Person person = people.next();
(7)         text.add(mapper.writeValueAsString(person));
(8)       }
(9)       return text;
(10)    }
(11)  }
(12)  JavaRDD<Person> result = input.mapPartitions(new ParseJson()).filter(
            new LikesPandas());
(13)  JavaRDD<String> formatted = result.mapPartitions(new WriteJson());
(14)  formatted.saveAsTextFile(outfile);
```

Codeblock 36: Speichern von Daten im JSON-Format [27]

A.2 Schlüssel-Wert Paare

Function name	Purpose	Example	Result
reduceByKey(func)	Combine values with the same key.	rdd.reduceByKey((x, y) => x + y)	{(1, 2), (3, 10)}
groupByKey()	Group values with the same key.	rdd.groupByKey()	{(1, [2]), (3, [4, 6])}
combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner)	Combine values with the same key using a different result type.	See Examples 4-12 through 4-14.	
mapValues(func)	Apply a function to each value of a pair RDD without changing the key.	rdd.mapValues(x => x+1)	{(1, 3), (3, 5), (3, 7)}
flatMapValues(func)	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	rdd.flatMapValues(x => (x to 5)	{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}
keys()	Return an RDD of just the keys.	rdd.keys()	{1, 3, 3}
values()	Return an RDD of just the values.	rdd.values()	{2, 4, 6}
sortByKey()	Return an RDD sorted by the key.	rdd.sortByKey()	{(1, 2), (3, 4), (3, 6)}

Tabelle 7: Transformationen, anwendbar auf einem Pair-RDD [27]

Function name	Purpose	Example	Result
subtractByKey	Remove elements with a key present in the other RDD.	rdd.subtractByKey(other)	{(1, 2)}
join	Perform an inner join between two RDDs.	rdd.join(other)	{(3, (4, 9)), (3, (6, 9))}
rightOuterJoin	Perform a join between two RDDs where the key must be present in the first RDD.	rdd.rightOuterJoin(other)	{(3,(Some(4),9)), (3,(Some(6),9))}
leftOuterJoin	Perform a join between two RDDs where the key must be present in the other RDD.	rdd.leftOuterJoin(other)	{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}
cogroup	Group data from both RDDs sharing the same key.	rdd.cogroup(other)	{(1,([2],[])), (3, ([4, 6],[9]))}

Tabelle 8: Transformationen, anwendbar auf zwei Pair-RDDs [27]

A.3 Partitionierung

```
(1)   public class PartitionExample1  {
(2)     public static void main(String[] args) {
(3)         try {
(4)           // 0.) Preparation
(5)           String s3file="s3n://dmueller5bucket/data/sorted100M_0.csv";
(6)           SparkConf conf = new SparkConf();
(7)           JavaSparkContext sc = new JavaSparkContext(conf);
(8)           Configuration hadoopConf = sc.hadoopConfiguration();
(9)           hadoopConf.set("fs.s3n.impl", "org.apache.hadoop.fs.s3na
               tive.NativeS3FileSystem");
(10)          hadoopConf.set("fs.s3.impl", "org.apache.hadoop.fs.s3na
               tive.NativeS3FileSystem");
(11)          hadoopConf.set("fs.s3n.awsAccessKeyId", "xxx");
(12)          hadoopConf.set("fs.s3n.awsSecretAccessKey", "xxx");
(13)
(14)          // 1.) Read in the S3 file as String-RDD
(15)          JavaRDD<String> inputRDD = sc.textFile(s3file);
(16)
(17)          // 2.) Map the String-RDD to a DEBS-RDD
(18)          JavaRDD<DebsDataSet> debsDataSetRDD = inputRDD
               .map(new Function<String, DebsDataSet>() {
(19)            public DebsDataSet call(String row) throws Exception {
(20)              String[] data = row.split(",");
(21)              int id = Integer.valueOf(data[0]);
(22)              long ts = Long.valueOf(data[1]);
(23)              float val = Float.valueOf(data[2]);
(24)              int type = Integer.valueOf(data[3]);
(25)              int plug_id = Integer.valueOf(data[4]);
(26)              int household_id = Integer.valueOf(data[5]);
(27)              int house_id = Integer.valueOf(data[6]);
(28)              return new DebsDataSet(id, ts, val, type, plug_id,
                   household_id, house_id);
(29)            }
(30)          });
(31)
(32)          // 3.) Build a Pair-RDD with key=houseId and value=consumpt.
(33)          JavaPairRDD<Integer, Float> consumptionRDD = debsDataSetRDD
               .mapToPair(new PairFunction<DebsDataSet, Integer, Float>() {
(34)            public Tuple2<Integer, Float> call(DebsDataSet debs) {
(35)              return new Tuple2<Integer, Float>(debs.getHouse_id(),
                   debs.getVal());
(36)            }
(37)          });
(38)
(39)          // 4.) Partition the RDD by HashPartitioner into 4 pieces
(40)          JavaPairRDD<Integer, Float> partitionedRDD =
               consumptionRDD.partitionBy(new HashPartitioner(4));
(41)
(42)          // 5.) Save the results
(43)          partitionedRDD.saveAsTextFile("s3n://dmueller5bucket/data/
               partitioner_output_" + System.currentTimeMillis());
(44)        } catch (Exception e) {
(45)          e.printStackTrace();
(46)          System.out.println(e.getMessage());
(47)        }
(48)      }
(49)  }
```

Codeblock 37: Partitionieren und Speichern einer CSV-Datei mit HashPartitioner

A.4 Installation von Spark

In diesem Anhang finden sich Informationen zur Spark-Projektstruktur und den wichtigsten Dateien, mit denen Spark konfiguriert und ausgeführt werden kann. Anschließend folgt eine Anleitung zur Installation von Spark im lokalen Netzwerk und auf EC2-Clustern der Amazon Web Services.

Projektstruktur von Spark

Nach Herunterladen eines gepackten Spark-Releases, muss dieses zunächst entpackt werden. Der entstehende Spark-Installationsordner beinhaltet u.a. folgende wichtige Unterordner mit folgenden hilfreichen Dateien:

Ordner	Wichtige Dateien	Bedeutung
bin	run-example	Skript, mit dem Beispiel-Applikationen gestartet werden können
	spark-submit	Skript, mit dem der Anwender eigene Applikationen dem Spark-Cluster zur Ausführung übergeben kann
conf	slaves	Konfigurationsdatei, in der die Slaves eines Spark-Clusters gelistet werden
	log4j.properties	Konfigurationsdatei, für Einstellungen zum Logging
	spark-defaults.conf	Konfigurationsdatei, zum Festlegen von Standardeinstellungen des Clusters
	spark-env.sh	Konfigurationsdatei, zum Festlegen der Cluster-Umgebung
sbin	start-all.sh	Skript, um Spark in einer lokalen Standalone-Variante starten zu können
	stop-all.sh	Skript, um Spark in einer lokalen Standalone-Variante stoppen zu können
ec2	spark-ec2	Skript, um Spark im EC2-Cluster betreiben und steuern zu können

Tabelle 9: Wichtige Ordner und Dateien der Spark-Installation

Aufsetzen eines Standalone Spark-Clusters

Das Aufsetzen eines Spark-Clusters mit Standalone-Scheduler variiert je nach Verwendungsumgebung. Soll ein Cluster manuell in einem eigenen Netzwerk aufgesetzt werden, müssen zunächst einige Konfigurationen getroffen werden, ehe anschließend das Cluster gestartet werden kann. Ist ein Cluster im EC2-Umfeld geplant, wird dem Anwender durch Aufruf eines entsprechenden Skripts dieser Konfigurationsaufwand weitgehend abgenommen. Die beiden folgenden Abschnitte erklären das Vorgehen bei diesen beiden Installationsvarianten.

Manuelles Aufsetzen eines Spark-Clusters im eigenen Netzwerk

Soll ein Spark-Cluster mit Standalone-Scheduler in einem eigenen Netzwerk aufgesetzt werden, müssen zunächst die beteiligten Maschinen konfiguriert und zu einem Cluster zusammengefasst werden. Es geht darum, Ressourcen auf jeder Instanz zur Verfügung zu stellen und sowohl Master, als auch Slaves zu bestimmen und einander bekannt zu machen. Um die Spark-Funktionalitäten verwenden zu können, muss der Anwender eine Spark-Version herunterladen, entpacken und in den gleichen Pfad auf alle Maschinen kopieren, z. B. */home/user/spark*. Damit ein gegenseitiger Zugriff zwischen Slave und Master ermöglicht wird, muss auf jeder Slave-Instanz ein passwortloser SSH-Zugriff aufgesetzt werden.[55] Anschließend muss dem Masterknoten mitgeteilt werden, welche Instanzen als seine Slaves fungieren. Dies wird durch Eintragen aller Slave-Hostnamen in der */conf/slaves* Datei auf dem Master erledigt. Mehr Einstellungen können durch entsprechendes Abändern der restlichen Dateien im *conf*-Ordner vorgenommen werden. Nach diesen Schritten sollte das Cluster dann durch das *start-all.sh* Skript im *sbin*-Ordner gestartet werden können. Nach einem erfolgreichen Start kann die Web-Oberfläche, die dem Anwender zur Kontrolle über Jobs und Ressourcen dient, über die URL *http://master_hostname:8080*[56] aufgerufen werden. Mithilfe des *stop-all.sh* Skripts im gleichen Ordner kann das Cluster gestoppt werden.

Aufsetzen eines Spark-Clusters im AWS-EC2 Umfeld

Spark bietet über ein spezielles Skript die Möglichkeit, ein Spark-Cluster auch auf EC2-Instanzen aufsetzen zu können. Das Skript ist unter */ec2/spark-ec2* zu finden. Der Anwender benötigt neben einem exportieren SSH-Schlüsselpaar auch seine Anmeldetokens *Access Key ID* und *Secret Access Key*. Letztere beide müssen als Umgebungsvariablen gesetzt werden[57], ehe das EC2-Skript aufgerufen werden kann. Über den *launch*-Befehl kann ein Cluster aufgesetzt werden:

```
(1)   /home/user/spark/ec2/spark-ec2
(2)     -k <keypair_name>
(3)     -i <keypair_path_pem_file>
(4)     launch <cluster_name>
```

Codeblock 38: Befehl zum Aufsetzen eines Spark-Clusters im EC2-Umfeld

Folgende Optionen stehen außerdem beim *launch*-Befehl zur Verfügung:

Parameter	Auswirkung
-s <number_slaves>	Änderung der Anzahl aufzusetzender Clients (Standard: 1)
-t <instance_type>	Änderung des EC2-Instanz-Typs (Standard: m1.xlarge)

[55] Nähere Erläuterung hierzu in [27] und Kapitel 7
[56] Sofern keine manuellen Änderungen des Ports über die Konfigurationsdateien erfolgt ist.
[57] z. B. per Konsole mit den Befehlen
export AWS_SECRET_ACCESS_KEY=<Secret Access Key> und
export AWS_ACCESS_KEY_ID=<Access Key ID>

-r <region>	Änderung der EC2-Region, in der das Cluster laufen soll
-z <zone>	Festlegung der Availability Zone innerhalb der Region

Tabelle 10: Weitere Parameter des launch-Skripts

Bei gesetzten Umgebungsvariablen kann ein Cluster über die folgenden Befehle gestartet, gestoppt bzw. terminiert werden:

```
(1)   /home/user/spark/ec2/spark-ec2
(2)     -k <keypair_name>
(3)     -i <keypair_path_pem_file>
(4)     start | stop | destroy <cluster_name>
```

Codeblock 39: EC2-Befehle zum Starten, Stoppen bzw. Terminieren eines Clusters

A.5 Applikationen bereitstellen

Flag	Explanation
`--master`	Indicates the cluster manager to connect to. The options for this flag are described in Table 7-1.
`--deploy-mode`	Whether to launch the driver program locally ("client") or on one of the worker machines inside the cluster ("cluster"). In client mode `spark-submit` will run your driver on the same machine where `spark-submit` is itself being invoked. In cluster mode, the driver will be shipped to execute on a worker node in the cluster. The default is client mode.
`--class`	The "main" class of your application if you're running a Java or Scala program.
`--name`	A human-readable name for your application. This will be displayed in Spark's web UI.
`--jars`	A list of JAR files to upload and place on the classpath of your application. If your application depends on a small number of third-party JARs, you can add them here.
`--files`	A list of files to be placed in the working directory of your application. This can be used for data files that you want to distribute to each node.
`--py-files`	A list of files to be added to the PYTHONPATH of your application. This can contain *.py*, *.egg*, or *.zip* files.
`--executor-memory`	The amount of memory to use for executors, in bytes. Suffixes can be used to specify larger quantities such as "512m" (512 megabytes) or "15g" (15 gigabytes).
`--driver-memory`	The amount of memory to use for the driver process, in bytes. Suffixes can be used to specify larger quantities such as "512m" (512 megabytes) or "15g" (15 gigabytes).

Tabelle 11: Wichtige Paramter des spark-submit Skripts [27]

Value	Explanation
`spark://host:port`	Connect to a Spark Standalone cluster at the specified port. By default Spark Standalone masters use port 7077.
`mesos://host:port`	Connect to a Mesos cluster master at the specified port. By default Mesos masters listen on port 5050.
`yarn`	Connect to a YARN cluster. When running on YARN you'll need to set the `HADOOP_CONF_DIR` environment variable to point the location of your Hadoop configuration directory, which contains information about the cluster.
`local`	Run in local mode with a single core.
`local[N]`	Run in local mode with N cores.
`local[*]`	Run in local mode and use as many cores as the machine has.

Tabelle 12: Mögliche Werte für den --master Parameter des spark-submit Skripts [27]

A.6 Spark & HDFS

Installation von Hadoop 2.7 im EC2-Cluster

Für vergleichende Analysen wurde im Laufe der Arbeit auch Hadoop 2 in einem EC2-Cluster installiert. Da eine genauere Behandlung des Themas den Rahmen der Thesis gesprengt hätte, wird hier auf eine entsprechende Quelle verwiesen, die das Thema der Cluster-Installation von Hadoop im EC2-Umfeld beschreibt. Aufgrund der Veröffentlichung neuerer Hadoop-Versionen sind an einigen Stellen Änderungen vorzunehmen, diese werden ebenfalls kurz gelistet.

Bei der Installation von Hadoop wurden folgende beide Teile der Anleitung befolgt:

- Teil 1: Einrichten von EC2-Instanzen, Sicherheit und Systemzugang:
  http://java.dzone.com/articles/how-set-multi-node-hadoop (Stand: Mai 2015)
- Teil 2: Installation von Hadoop auf dem in Teil 1 aufgesetzten Cluster:
  https://letsdobigdata.wordpress.com/2014/01/13/setting-up-hadoop-1-2-1-multi-node-cluster-on-amazon-ec2-part-2/ (Stand: Mai 2015)

Die nachfolgenden Hinweise beziehen sich auf Unterschiede zwischen der in der Anleitung verwendeten EC2- bzw. Hadoop-Versionen und dem heutigen Stand.[58]

Hinweise zu Teil 1

- Schritt 1.3 & 1.4: Hier ist es wichtig, dass als AMI-Typ „Ubuntu Server 14.04 LTS“ mit Virtualisierungstyp „*paravirtual (PV)*“ verwendet wird (nicht HV!), da nur hierbei die Auswahl der in der Anleitung verwendeten t1.micro Instanz möglich ist.

[58] EC2-Installationsvarianten am 28.05.2015 und Hadoop-Version 2.7.0

- Schritt 1.5: Hier darf unter „Network“ kein VPC ausgewählt werden (stattdessen Eintrag „Launch into EC2-Classic“ übernehmen), da ansonsten die Installation im VPC-Umfeld stattfindet und keine öffentliche DNS Adresse für die Instanzen angelegt wird (wichtig in späteren Schritten)
- Schritt 2.3 & weitere: Der Benutzername ist nicht ubuntu, sondern „***ec2-user***“ (bei ssh Login und Pfaden anzugeben)
- Schritt 2.4: Die Hostname-Einstellungen werden nach jedem Server-Neustart zurückgesetzt, dieser Schritt muss dann wiederholt werden, um eine korrekte Kommunikation mithilfe der AWS-DNS zu ermöglichen.

Hinweise zu Teil 2

- Schritt 1.1: Statt *apt-get* muss in der EC2 Ubuntu-Version „*yum*“ verwendet werden → *$ sudo yum update* verwenden, um das System auf Updates zu prüfen.
- Schritt 1.2: Java ist bereits installiert, Pfad: „*/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.79.x86_64/jre*“ (für weitere Schritte wichtig)
- Schritt 1.3: Herunterladen von Hadoop 2.7 über URL *http://mirror.netcologne.de/apache.org/hadoop/common/hadoop-2.7.0/hadoop-2.7.0.tar.gz*
- Schritt 1.4: Username „*ec2-user*“ (statt *ubuntu*) verwenden und geänderte Projektstruktur von Hadoop beachten:
  export HADOOP_CONF=/home/ec2-user/hadoop/etc/hadoop
  export HADOOP_PREFIX=/home/ec2-user/hadoop
  export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.79.x86_64/jre
  export PATH=$PATH:$HADOOP_PREFIX/bin
- Schritt 1.5: Muss bei jeder neuen Session durchgeführt werden, ansonsten kommt es beim Starten oder Verbinden des Hadoop-Clusters zu Fehlern.
- Schritt 1.6: Aktuelle Node-URLs verwenden (Public DNS) und die Datei mapred-site.xml-template entsprechend nach „*mapred-site.xml*“ umbenennen, um Änderungen wirksam zu machen.
- Schritt 1.6.1: Neuen Benutzernamen (*ec2-user*) verwenden bei *scp*-Login und Pfadangaben.
- Schritt 1.6.3: Kann übersprungen werden, Datei „*masters*“ ist nicht mehr vorhanden und eine Installation kann ohne diese durchgeführt werden.
- Schritt 1.7:
  1. Verwenden der Befehle „*start-dfs.sh*“ und „*start-yarn.sh*“ statt *start-all.sh* (deprecated).

2. Der später folgende *jps*-Befehl hat mit Hadoop selbst nichts zu tun und kann ggf. übersprungen werden.
3. Der Link zum Namenode lautet: *http://master_dns:50070* (*dfshealth.html*-Seite wird automatisch geöffnet)
4. Ausführen des Pi-Berechnungsbeispiels aktuell über den Befehl „hadoop jar ~/hadoop/share/mapreduce/hadoop-mapreduce-examples-2.7.0.jar pi 10 1000“ (Parameter anpassbar).

Weiterführende Links

- Eine Liste mit wichtigen HDFS-Befehlen findet sich hier: *http://hadoop.apache.org/docs/r2.7.0/hadoop-project-dist/hadoop-common/FileSystemShell.html*

Hinweis bei Problemen nach Konfigurationsänderung

Zur Konfiguration des Clusters bietet folgender Internet-Blog interessante Hinweise:
http://java.dzone.com/articles/how-set-multi-node-hadoop (Teil 1)
https://letsdobigdata.wordpress.com/2014/01/13/setting-up-hadoop-1-2-1-multi-node-cluster-on-amazon-ec2-part-2/ (Teil 2)
Sollten Änderungen an den Konfigurationsdateien vorgenommen worden sein (Zuständigkeiten und Aufbau des HDFS-Clusters), kann es passieren, dass die zur Speicherung von Dateien benötigten MapReduce-Jobs nicht mehr korrekt arbeiten, also ein Speichern bzw. Aufrufen von Dateien nicht mehr möglich ist. Dann kann es, wenn ein Neustart des Spark-Clusters nicht hilft, die einfachste Lösung sein, die Dienste zu beenden, das HDFS zu formatieren und die Dienste erneut zu starten. Da auch das Formatieren teils nicht korrekt durchgeführt wird, ist es hilfreich, zuvor das Partitionsverzeichnis zu leeren. Folgende Schritte können bei der Fehlerbeseitigung helfen:

1. */root/<persist|ephemeral>-hdfs/bin/stop-all.sh* (falls Dienste laufen)
2. *rm –r /vol/<persist|ephemeral>-hdfs* (Löschen der Dateien)
3. */root/<persist|ephemeral>-hdfs/bin/hadoop namenode –format*
4. */root/<persist|ephemeral>-hdfs/bin/start-dfs.sh*
5. */root/<persist|ephemeral>-hdfs/bin/start-mapred.sh*

Schritt 2 und 3 müssen dabei auf jeder Instanz des HDFS-Clusters durchgeführt werden! Danach sollte das Cluster wieder intakt sein und Schreib- bzw. Leseanfragen korrekt verarbeitet werden.

A.7 Spark & Cassandra

Spark & Cassandra im gemeinsamen Cluster

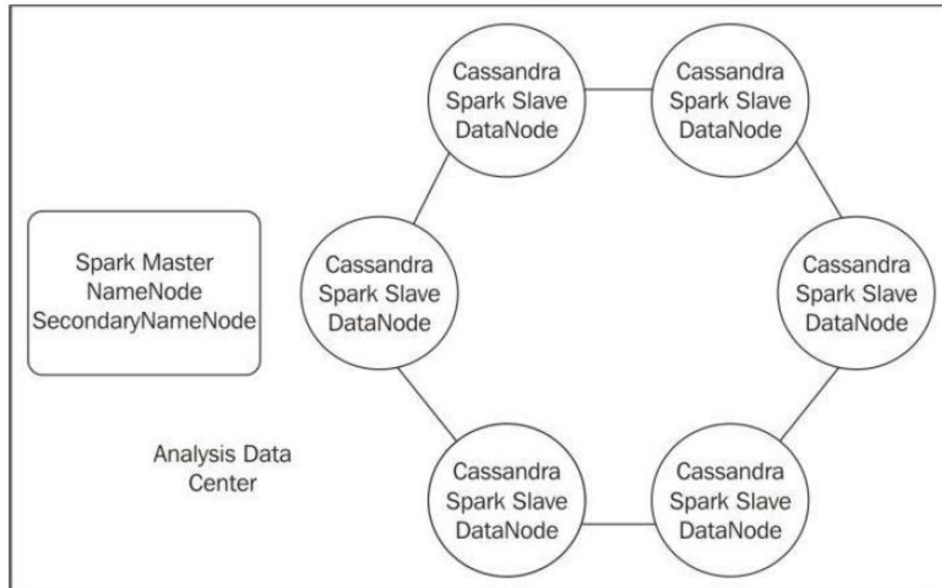

Abbildung 43: Aufbau eines gemeinsamen Clusters (Spark & Cassandra)

Folgende Schritte sind für eine gemeinsame Installation von Spark und Cassandra notwendig:

1. Aufsetzen eines Cassandra-Clusters mithilfe vordefinierter AMIs nach folgender Anleitung: http://docs.datastax.com/en/cassandra/2.1/cassandra/install/installAMI.html
    - Als Instanztyp „HVM" und „m3-large" auswählen, mit Region „us-east-1"
    - 5 Instanzen aufsetzen, mit Zusatzparameter `--clustername dmueller5cassandra --totalnodes 5 --version community`
    - Nach erfolgreicher Installation ist das OpsCenter erreichbar unter: http://cassandra_master_instance_0:8888/opscenter/index.html
2. Download von Spark in dieses existierende Cluster mithilfe der Anleitung: http://blog.prabeeshk.com/blog/2014/10/31/install-apache-spark-on-ubuntu-14-dot-04/
    - Hier kann das EC2-Skript zum Aufsetzen eines neuen Clusters nicht verwendet werden (es würde selbst ein neues Cluster erstellen)!!!
    - Java-Installation kann übersprungen werden
    - Es muss eine Scala 2.10.x Version installiert werden (nicht 2.11.x)!
    - Git muss nicht installiert werden in der aktuellen Spark-Version[59]
    - Link einer neueren Spark-Version verwenden, z. B. Version 1.2.2: http://d3kbcqa49mib13.cloudfront.net/spark-1.2.2-bin-hadoop2.4.tgz

[59] Hier verwendet: Spark-Version 1.2.2

- Nach Spark-Download und Entpacken kann Anleitung beendet werden
- Diese Anleitungsschritte müssen auf gleiche Weise auf allen Cluster-Instanzen durchgeführt werden, sodass die gleiche Spark-Version auf allen Instanzen über denselben Pfad erreichbar ist!

3. Einrichten von Spark im Cluster
    - Dem Master müssen alle Slaves bekannt gemacht werden. Dazu müssen die Public-DNS Adressen der Slaves in die */spark/conf/slaves* Datei eingetragen werden (je Instanz eine Zeile):

```
  GNU nano 2.2.6                                      File: ./spark/conf/slaves

# A Spark Worker will be started on each of the machines listed below
ec2-107-20-49-194.compute-1.amazonaws.com
ec2-54-92-202-167.compute-1.amazonaws.com
ec2-107-21-169-196.compute-1.amazonaws.com
ec2-54-160-188-119.compute-1.amazonaws.com
```

4. Passwortloser SSH-Zugriff für den Master auf allen Slaves einrichten
    - Auf dem Master folgende Befehle ausführen:
        i. *ssh-keygen –t dsa* (im Standardpfad, ohne Passwort speichern)
    - Die Datei *~/.ssh/id_dsa.pub* per *scp* und Keypair auf alle Slaves laden
    - Auf jedem Slave die Master-Identität als vertrauenswürdig setzen:
        i. cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
        ii. chmod 644 ~/.ssh/authorized_keys
5. Zusammenfügen der Firewall-Einstellungen von Spark in Cassandra Security-Group:

Type	Protocol	Port Range	Source
Custom TCP Rule	TCP	1 - 65355	sg-458b0428 (dmueller5cassandra)
Custom TCP Rule	TCP	7000	sg-458b0428 (dmueller5cassandra)
Custom TCP Rule	TCP	7001	sg-458b0428 (dmueller5cassandra)
Custom TCP Rule	TCP	7199	sg-458b0428 (dmueller5cassandra)
Custom TCP Rule	TCP	61620	sg-458b0428 (dmueller5cassandra)
Custom TCP Rule	TCP	61621	sg-458b0428 (dmueller5cassandra)
All UDP	UDP	0 - 65535	sg-458b0428 (dmueller5cassandra)
All ICMP	All	N/A	sg-458b0428 (dmueller5cassandra)
SSH	TCP	22	0.0.0.0/0
Custom TCP Rule	TCP	4040 - 4045	0.0.0.0/0
Custom TCP Rule	TCP	5080	0.0.0.0/0
Custom TCP Rule	TCP	8080 - 8081	0.0.0.0/0
Custom TCP Rule	TCP	8888	0.0.0.0/0
Custom TCP Rule	TCP	9042	0.0.0.0/0
Custom TCP Rule	TCP	9160	0.0.0.0/0
Custom TCP Rule	TCP	18080	0.0.0.0/0
Custom TCP Rule	TCP	19999	0.0.0.0/0
Custom TCP Rule	TCP	50030	0.0.0.0/0
Custom TCP Rule	TCP	50060	0.0.0.0/0
Custom TCP Rule	TCP	50070	0.0.0.0/0
Custom TCP Rule	TCP	50075	0.0.0.0/0
Custom TCP Rule	TCP	60060	0.0.0.0/0
Custom TCP Rule	TCP	60070	0.0.0.0/0
Custom TCP Rule	TCP	60075	0.0.0.0/0

    - Nach dem Zusammentragen der Portfreigaben sollte nun die Spark-Weboberfläche unter folgender URL erreichbar sein:
      http://master_public_dns:8080

6. Arbeiten über den Masterknoten
    - *nodetool*, *cqlsh* und weitere Cassandra-Tools sind direkt nach einem SSH-Login verfügbar

Spark & Cassandra in separaten Clustern

Es folgt nun eine Anleitung zur Installation eines Cassandra Clusters.

➔ Hinweis: Ein Cassandra-Cluster kann nicht mit AWS Spot Instanzen aufgesetzt werden: http://www.datastax.com/support-forums/topic/ec2-ami-24-issue

Eine vereinfachte Installation bietet die Verwendung von AMIs. Sie können verwendet werden, um ein Cluster aus mehreren Instanzen aufzusetzen, eine Anleitung findet sich hier: http://docs.datastax.com/en/cassandra/2.1/cassandra/install/installAMI.html

Cassandra Datenbanken verwenden mit *cqlsh*-Tool

- Anlegen eines Keyspace

```
(1)   cqlsh> CREATE KEYSPACE debs WITH REPLICATION = {
(2)           'class' : 'SimpleStrategy',
(3)           'replication_factor' : 3
(4)          };
```

Codeblock 40: Cassandra-Keyspace einrichten

- Tabelle im Keyspace anlegen

```
(1)   cqlsh> create TABLE debs.energydata10m(
(2)                id int PRIMARY KEY,
(3)                ts timestamp,
(4)                val float,
(5)                type int,
(6)                plug_id int,
(7)                household_id int,
(8)                house_id int);
```

Codeblock 41: Anlegen einer Tabelle mit cqlsh

- Daten in die Tabelle einfügen

```
(1)   cqlsh> insert into debs.energydata10m
(2)           (id, ts, val, type, plug_id , household_id , house_id )
(3)           values (1, 2, 3, 4, 5, 6, 7);
```

Codeblock 42: Einfügen von Daten in Cassandra-Tabellen mit cqlsh

- Tabelleninhalt abfragen

```
(1)   cqlsh> SELECT * FROM debs.energydata10m;
(2)    id | house_id| household_id| plug_id | ts           | type | val
(3)   ----+---------+-------------+---------+-------------+------+----
(4)     1 |       7 |           6 |       5 | 1970-01-01 |    4 |   3
(5)   (1 rows)
```

Codeblock 43: Tabelleninhalt abfragen mit cqlsh

- Keyspace vollständig löschen (Keyspace & Snapshots!)
    1. Keyspace löschen mit cqlsh:

```
(1)   cqlsh> DROP KEYSPACE debs;
```

Codeblock 44: Löschen des Keyspaces und aller darin enthaltenen Tabellen

2. Snapshots löschen, um Speicher freizugeben mit nodetool:

```
(2)  nodetool clearsnapshot debs
```

Codeblock 45: Löschen von Snapshots zur Speicher-Freigabe

A.8 Cassandra-Benchmarks

In diesem Kapitel finden sich die Quellcodes und Ergebnisse der Benchmarks, die zum Messen der Schreib- und Lesegeschwindigkeit von Cassandra verwendet wurden.

Schreibtest

```
(1)   public class CassandraWriting {
(2)     public static void main(String[] args) {
(3)       String s3file = "";
(4)       String cassandraTable = "";
(5)       String cassandraIP = "";
(6)       int partitions;
(7)       String memory;
(8)
(9)       if(args.length == 4) {
(10)        if(args[0].equals("100")) {
(11)          s3file = "sorted100M.csv";
(12)          cassandraTable = "energydata100m";
(13)        } else if(args[0].equals("10"))  {
(14)          s3file = "sorted100M_0.csv";
(15)          cassandraTable = "energydata10m";
(16)        } else {
(17)          s3file = "sorted100M_x_5.csv";
(18)          cassandraTable = "energydata500m";
(19)        }
(20)        cassandraIP = args[1];
(21)        memory = args[2];
(22)        partitions = Integer.valueOf(args[3]);
(23)      } else {
(24)        System.out.println("Use 4 parameters: <data: 10|100|500>
           <cassandra-master-ip> <memory 512m | 1g | ...> <partitions>");
(25)        return;
(26)      }
(27)      SparkConf conf = new SparkConf();
(28)      conf.set("spark.eventLog.enabled", "true");
(29)      conf.set("spark.cassandra.connection.host", cassandraIP);
(30)      conf.set("spark.driver.memory", memory);
(31)      conf.set("spark.executor.memory",memory);
(32)
(33)      JavaSparkContext sc = new JavaSparkContext(conf);
(34)
(35)      String s3path = "s3n://dmueller5bucket/data/"  + s3file;
(36)      setS3credentials(sc, s3path);
(37)
(38)      // ******************** BENCHMARK ********************
(39)      long duration = System.currentTimeMillis();
(40)      JavaRDD<String> csvRDD = sc.textFile(s3path, partitions);
(41)      JavaRDD<DebsDataSet> debsRDD = csvRDD.map(new Function<String,
           DebsDataSet>() {
(42)        public DebsDataSet call(String inputRow) throws Exception {
(43)          String[] data = inputRow.split(",");
(44)          int id = Integer.valueOf(data[0]);
(45)          Long ts = Long.valueOf(data[1]);
(46)          float value = Float.valueOf(data[2]);
```

```
(47)            int type = Integer.valueOf(data[3]);
(48)            int plugId = Integer.valueOf(data[4]);
(49)            int householdId = Integer.valueOf(data[5]);
(50)            int houseId = Integer.valueOf(data[6]);
(51)            return new DebsDataSet(id, ts, value, type, plugId,
                  householdId, houseId);
(52)          }
(53)        });
(54)
(55)        CassandraJavaUtil.javaFunctions(debsRDD).writerBuilder("debs",
              cassandraTable, CassandraJavaUtil
              .mapToRow(DebsDataSet.class)).saveToCassandra();
(56)
(57)        System.out.println("*** FINISHED SUCCESSFULLY!!! *** took : " +
              ((System.currentTimeMillis() - duration) / 1000) + " seconds
              (with " + debsRDD.partitions().size() + " partitions)!");
(58)      }
(59)
(60)      static void setS3credentials(JavaSparkContext sc, String path) {
(61)        if (path.startsWith("s3n://")) {
(62)          Configuration hadoopConf = sc.hadoopConfiguration();
(63)          hadoopConf.set("fs.s3n.impl", "org.apache.hadoop.fs.
                s3native.NativeS3FileSystem");
(64)          hadoopConf.set("fs.s3.impl", "org.apache.hadoop.fs.
                s3native.NativeS3FileSystem");
(65)          hadoopConf.set("fs.s3n.awsAccessKeyId", "xxx");
(66)          hadoopConf.set("fs.s3n.awsSecretAccessKey", "xxx");
(67)        }
(68)      }
(69)
(70)    }
```

Codeblock 46: Schreibtest mit Cassandra

Ergebnisse des Schreibtests:

	Input		Spark		Cassandra		Benchmark		
#	Datengröße		Anzahl Slaves	zugewiesener RAM (GB)	Instanzen	RAM (GB)	Spark Dauer	Requests / sek	Requests / Core
1	100 Mio.	3,1 GB	1	**0,5**	1	30	7.440,0	13.440,9	**1.680,1**
	10 Mio.	323 MB	1	**0,5**	1	30	720,0	13.888,9	**1.736,1**
2	100 Mio.	3,1 GB	1	**3**	1	30	7.597,0	**13.163,1**	**1.645,4**
	10 Mio.	323 MB	1	**3**	1	30	739,0	**13.531,8**	**1.691,5**
3	100 Mio.	3,1 GB	2	3	2	30	3.446,0	**29.019,2**	**1.813,7**
	10 Mio.	323 MB	2	3	2	30	335,0	**29.850,7**	**1.865,7**
4	100 Mio.	3,1 GB	4	3	4	30	1.753,0	**57.045,1**	**1.782,7**
	10 Mio.	323 MB	4	3	4	30	176,0	**56.818,2**	**1.775,6**
	500 Mio.	15 GB	4	3	4	30	8.887,0	**56.262,0**	**1.758,2**

Tabelle 13: Ergebnisse des Schreibtests mit Cassandra

Lesetests

```
(1)   public class CassandraReading {
(2)     public static void main(String[] args) {
(3)       String cassandraTable = "";
(4)       String cassandraIP = "";
(5)       String memory;
(6)
(7)       if(args.length == 3) {
(8)         if(args[0].equals("100")) {
(9)           cassandraTable = "energydata100m";
(10)        } else if(args[0].equals("10"))  {
(11)          cassandraTable = "energydata10m";
(12)        } else {
(13)          cassandraTable = "energydata500m";
(14)        }
(15)        cassandraIP = args[1];
(16)        memory = args[2];
(17)      } else {
(18)        System.out.println("Use 3 parameters: <data: 10|100|500>
              <cassandra-master-ip> <memory 512m | 1g | ...>");
(19)        return;
(20)      }
(21)
(22)      SparkConf conf = new SparkConf();
(23)      conf.set("spark.eventLog.enabled", "true");
(24)      conf.set("spark.cassandra.connection.host", cassandraIP);
(25)      conf.set("spark.driver.memory", memory);
(26)      conf.set("spark.executor.memory",memory);
(27)
(28)      long duration = System.currentTimeMillis();
(29)
(30)      // ******************** BENCHMARK ********************
(31)      JavaSparkContext sc = new JavaSparkContext(conf);
(32)
(33)      JavaRDD<String> rdd = CassandraJavaUtil
(34)        .javaFunctions(sc).cassandraTable("debs", cassandraTable)
            .map(new Function<CassandraRow, String>() {
(35)        public String call(CassandraRow cr) throws Exception {
(36)          return cr.toString();
(37)        }
(38)      });
(39)      System.out.println("RDD size : " + rdd.count());
(40)      System.out.println("*** FINISHED SUCCESSFULLY!!! *** took : " +
            ((System.currentTimeMillis() - duration) / 1000) + " seconds");
(41)    }
(42)  }
```

Codeblock 47: Lesetest mit Cassandra

Ergebnisse des Lesetests:

	Spark + Cassandra Cluster				Benchmark		
#	Tabelle	Anzahl Instanzen	RAM (GB)	Zugewiesener RAM je Spark-Thread (GB)	Spark Dauer	Requests / sek	Requests / Core
1	10 Mio.	1 (+ Spark-Driver)	30	3	51	196.078	24.510
	100 Mio.	1 (+ Spark-Driver)	30	3	430	232.558	29.070
2	10 Mio.	2 (+ Spark-Driver)	30	3	30	333.333	20.833
	100 Mio.	2 (+ Spark-Driver)	30	3	227	440.529	27.533
3	10 Mio.	4 (+ Spark-Driver)	30	3	18	555.556	17.361
	100 Mio.	4 (+ Spark-Driver)	30	3	117	854.701	26.709

Tabelle 14: Ergebnisse des Lesetests mit Cassandra

A.9 HDFS-Benchmarks

In diesem Kapitel finden sich die Quellcodes und Ergebnisse der Benchmarks, die zum Messen der Schreib- und Lesegeschwindigkeit von HDFS verwendet wurden.

Schreibtest

```
(1)   public class HdfsWriting {
(2)     public static void main(String[] args) {
(3)       String s3file = "";
(4)       String masterURL;
(5)       int partitions;
(6)       String memory;
(7)
(8)       if (args.length == 3) {
(9)         if (args[0].equals("100")) {
(10)          s3file = "sorted100M.csv";
(11)        } else if (args[0].equals("10")) {
(12)          s3file = "sorted100M_0.csv";
(13)        } else {
(14)          s3file = "sorted100M_x_5.csv";
(15)        }
(16)        memory = args[1];
(17)        partitions = Integer.valueOf(args[2]);
(18)      } else {
(19)        System.out.println("Use 4 parameters: <data: 10|100|500>
              <memory 512m | 1g | ...> <partitions>");
(20)        return;
(21)      }
(22)
(23)      SparkConf conf = new SparkConf();
(24)      conf.set("spark.eventLog.enabled", "true");
(25)      conf.set("spark.driver.memory", memory);
(26)      conf.set("spark.executor.memory", memory);
(27)
(28)      masterURL = conf.get("spark.master");
(29)      if (masterURL.startsWith("spark://")) {
(30)        masterURL = masterURL.split("spark://")[1];
(31)        masterURL = masterURL.split(":")[0];
(32)        masterURL = "hdfs://" + masterURL + ":9000/";
(33)
```

```
(34)        System.out.println("masterURL : " + masterURL);
(35)        System.out.println("s3file : " + s3file);
(36)
(37)        JavaSparkContext sc = new JavaSparkContext(conf);
(38)
(39)        String s3path = "s3n://dmueller5bucket/data/" + s3file;
(40)        setS3credentials(sc, s3path);
(41)
(42)        // ************ BENCHMARK *****************
(43)        long duration = System.currentTimeMillis();
(44)        JavaRDD<String> csvRDD = sc.textFile(s3path, partitions);
(45)        csvRDD.saveAsTextFile(masterURL + s3file);
(46)
(47)        System.out.println("*** FINISHED SUCCESSFULLY!!! *** took : " +
              ((System.currentTimeMillis() - duration) / 1000) + " seconds
              (with " + csvRDD.partitions().size() + " partitions)!");
(48)      } else {
(49)        System.out.println("Couldn't launch benchmark,master not set");
(50)      }
(51)    }
(52)
(53)    static void setS3credentials(JavaSparkContext sc, String path) {
(54)      if (path.startsWith("s3n://")) {
(55)        Configuration hadoopConf = sc.hadoopConfiguration();
(56)        hadoopConf.set("fs.s3n.impl", "org.apache.hadoop.fs.
              s3native.NativeS3FileSystem");
(57)        hadoopConf.set("fs.s3.impl", "org.apache.hadoop.fs.
              s3native.NativeS3FileSystem");
(58)        hadoopConf.set("fs.s3n.awsAccessKeyId", "xxx");
(59)        hadoopConf.set("fs.s3n.awsSecretAccessKey", "xxx");
(60)      }
(61)    }
(62)  }
```

Codeblock 48: Schreibtest mit HDFS

Ergebnisse des Schreibtests:

#	Worker	Input	RAM	Partitionen (= Dateien im HDFS)		Dauer (Sek)	Mittelwert	MB / sek
1	**4**	**10 Mio (0,32 GB)**	3g	320		9		
	4	**10 Mio (0,32 GB)**	512m	160		8		
	4	**10 Mio (0,32 GB)**	512m	32		6	6	53,3
	4	**10 Mio (0,32 GB)**	512m	16		6		
	4	**100 Mio (3,3 GB)**	3g	320		16		
	4	**100 Mio (3,3 GB)**	512m	160		15		
	4	**100 Mio (3,3 GB)**	512m	32		15	15	220,0
	4	**500 Mio (16,5 GB)**	3g	320	--> Teilbar durch 10, da S3-Datei aus 10 Teilen	55		
	4	**500 Mio (16,5 GB)**	512m	160	--> Teilbar durch 10, da S3-Datei aus 10 Teilen	55		
	4	**500 Mio (16,5 GB)**	3g	32 --> 40	--> Teilbar durch 10, da S3-Datei aus 10 Teilen	62	62	266,1
	4	**500 Mio (16,5 GB)**	512m	32 --> 40	--> Teilbar durch 10, da S3-Datei aus 10 Teilen	62		
2	**2**	**10 Mio (0,32 GB)**	3g	320		11		
	2	**10 Mio (0,32 GB)**	512m	320		11		
	2	**10 Mio (0,32 GB)**	3g	32		6	6	53,3
	2	**10 Mio (0,32 GB)**	512m	32		6		
	2	**10 Mio (0,32 GB)**	512m	16		6		
	2	**10 Mio (0,32 GB)**	512m	8		6		
	2	**10 Mio (0,32 GB)**	512m	1		14		
	2	**100 Mio (3,3 GB)**	512m	320		27		
	2	**100 Mio (3,3 GB)**	512m	32		24	23	143,5
	2	**100 Mio (3,3 GB)**	512m	16		22		
	2	**100 Mio (3,3 GB)**	512m	1		100		
	2	**500 Mio (16,5 GB)**	512m	320	--> Teilbar durch 10, da S3-Datei aus 10 Teilen	100		
	2	**500 Mio (16,5 GB)**	512m	130	--> Teilbar durch 10, da S3-Datei aus 10 Teilen	124		
	2	**500 Mio (16,5 GB)**	512m	32 --> 40	--> Teilbar durch 10, da S3-Datei aus 10 Teilen	95	99	166,7

	2	**500 Mio (16,5 GB)**	512m	16 --> 20	--> Teilbar durch 10, da S3-Datei aus 10 Teilen	103		
	2	**500 Mio (16,5 GB)**	512m	8 --> 10	--> Teilbar durch 10, da S3-Datei aus 10 Teilen	146		
3	**1**	**10 Mio (0,32 GB)**	3g	320		17		
	1	**10 Mio (0,32 GB)**	512m	320		16		
	1	**10 Mio (0,32 GB)**	3g	32		9	8,5	37,6
	1	**10 Mio (0,32 GB)**	512m	32		9		
	1	**10 Mio (0,32 GB)**	512m	16		8		
	1	**10 Mio (0,32 GB)**	512m	8		8		
	1	**10 Mio (0,32 GB)**	512m	1		14		
	1	**100 Mio (3,3 GB)**	512m	320		48		
	1	**100 Mio (3,3 GB)**	512m	32		41	40	82,5
	1	**100 Mio (3,3 GB)**	512m	16		39		
	1	**100 Mio (3,3 GB)**	512m	1		105		
	1	**500 Mio (16,5 GB)**	512m	320	--> Teilbar durch 10, da S3-Datei aus 10 Teilen	192		
	1	**500 Mio (16,5 GB)**	512m	130	--> Teilbar durch 10, da S3-Datei aus 10 Teilen	183		
	1	**500 Mio (16,5 GB)**	512m	32 --> 40	--> Teilbar durch 10, da S3-Datei aus 10 Teilen	190	189	87,3
	1	**500 Mio (16,5 GB)**	512m	16 --> 20	--> Teilbar durch 10, da S3-Datei aus 10 Teilen	188		
	1	**500 Mio (16,5 GB)**	512m	8 --> 10	--> Teilbar durch 10, da S3-Datei aus 10 Teilen	207		

Hinweis:

Bei 500Mio werden >= 10 Dateien nach HDFS geschrieben, da Datei im S3 aus 10 Teilen besteht: Part-000 bis Part-009

Legende:

--> Die tatsächlich von Spark verwendete Partitionszahl weicht von der Anwenderangabe ab: Vorgabe --> Tatsächlich

Ergebnisse mit dieser Hintergrundfärbung wurden nicht mit in die Mittelwertberechnung einbezogen, da ihre Partitionierung nicht optimal ist

Tabelle 15: Ergebnisse des Schreibtests mit HDFS

Lesetest

```
(1)   public class HdfsReading {
(2)
(3)     public static void main(String[] args) {
(4)       String masterURL;
(5)       int partitions;
(6)       String memory;
(7)       String hdfsFile = "";
(8)
(9)       if (args.length == 3) {
(10)        if (args[0].equals("100")) {
(11)          hdfsFile = "sorted100M.csv";
(12)        } else if (args[0].equals("10")) {
(13)          hdfsFile = "sorted100M_0.csv";
(14)        } else {
(15)          hdfsFile = "sorted100M_x_5.csv";
(16)        }
(17)        memory = args[1];
(18)        partitions = Integer.valueOf(args[2]);
(19)      } else {
(20)        System.out.println("Use 4 parameters: <data: 10|100|500>
              <memory 512m | 1g | ...> <partitions>");
(21)        return;
(22)      }
(23)
(24)      SparkConf conf = new SparkConf();
(25)      conf.set("spark.eventLog.enabled", "true");
(26)      conf.set("spark.driver.memory", memory);
(27)      conf.set("spark.executor.memory", memory);
(28)
(29)      masterURL = conf.get("spark.master");
(30)      if (masterURL.startsWith("spark://")) {
(31)        masterURL = masterURL.split("spark://")[1];
(32)        masterURL = masterURL.split(":")[0];
(33)        masterURL = "hdfs://" + masterURL + ":9000/";
(34)
(35)        System.out.println("masterURL : " + masterURL);
(36)        System.out.println("hdfsFile : " + hdfsFile);
(37)
(38)        JavaSparkContext sc = new JavaSparkContext(conf);
(39)
(40)        // ***************** BENCHMARK *******************
(41)        long duration = System.currentTimeMillis();
(42)
(43)        JavaRDD<String> csvRDD = sc.textFile(masterURL + hdfsFile,
              partitions);
(44)
(45)        System.out.println("RDD.count : " + csvRDD.count());
(46)        System.out.println("*** FINISHED SUCCESSFULLY!!! *** took : " +
              ((System.currentTimeMillis() - duration) / 1000) + " seconds
              (with " + csvRDD.partitions().size() + " partitions)!");
(47)      } else {
(48)        System.out.println("Couldn't launch benchmark, master not set
              as \"spark://...\" url!");
(49)      }
(50)    }
(51)  }
```

Codeblock 49: Lesetest mit HDFS

Ergebnisse des Lesetests:

#	Worker	Input	RAM	Partit.	Bemerkung	Dauer	Mittelwert	MB / sek
1	4	500 Mio (16,5 GB)	3g	320	--> mit 320 vorher geschrieben	12	12	1375,0
	4	500 Mio (16,5 GB)	512m	320	--> mit 320 vorher geschrieben	12		
	4	500 Mio (16,5 GB)	512m	32 / 130		12		
	4	500 Mio (16,5 GB)	512m	8 / 320	--> mit 320 vorher geschrieben	13		
	4	100 Mio (3,3 GB)	512m	1 / 26	--> mit 1 vorher geschrieben	5	5	660,0
	4	100 Mio (3,3 GB)	512m	1 / 26	--> mit 26 vorher geschrieben	5		
	4	10 Mio (0,3 GB)	512m	1 / 320	--> mit 320 vorher geschrieben	3	4	80,8
	4	10 Mio (0,3 GB)	512m	1 / 32	--> mit 32 vorher geschrieben	3		
	4	10 Mio (0,3 GB)	512m	320	--> mit 32 vorher geschrieben	3		
	4	10 Mio (0,3 GB)	512m	1 / 3	--> mit 1 vorher geschrieben	5		
	4	10 Mio (0,3 GB)	512m	320	--> mit 1 vorher geschrieben	8		
2	2	500 Mio (16,5 GB)	3g	320	--> mit 320 vorher geschrieben	25	25	660,0
	2	500 Mio (16,5 GB)	512m	320	--> mit 320 vorher geschrieben	25		
	2	500 Mio (16,5 GB)	512m	32 / 130		30		
	2	500 Mio (16,5 GB)	512m	8 / 320	--> mit 320 vorher geschrieben	21		
	2	100 Mio (3,3 GB)	512m	1 / 26	--> mit 1 vorher geschrieben	9	8	412,5
	2	100 Mio (3,3 GB)	512m	1 / 26		7		
	2	10 Mio (0,3 GB)	512m	1 / 320	--> mit 320 vorher geschrieben	4	5	64,6
	2	10 Mio (0,3 GB)	512m	1 / 32	--> mit 32 vorher geschrieben	3		
	2	10 Mio (0,3 GB)	512m	320	--> mit 32 vorher geschrieben	3		
	2	10 Mio (0,3 GB)	512m	1 / 3	--> mit 1 vorher geschrieben	4		
	2	10 Mio (0,3 GB)	512m	320	--> mit 1 vorher geschrieben	9		
3	1	500 Mio (16,5 GB)	3g	320	--> mit 320 vorher geschrieben	41	39	423,1
	1	500 Mio (16,5 GB)	512m	320	--> mit 320 vorher geschrieben	41		
	1	500 Mio (16,5 GB)	512m	32 / 130		37		
	1	500 Mio (16,5 GB)	512m	8 / 320	--> mit 320 vorher geschrieben	38		
	1	100 Mio (3,3 GB)	512m	1 / 26	--> mit 1 vorher geschrieben	10	10	330,0
	1	100 Mio (3,3 GB)	512m	1 / 26		10		
	1	10 Mio (0,3 GB)	512m	1 / 320	--> mit 320 vorher geschrieben	4	5	64,6
	1	10 Mio (0,3 GB)	512m	1 / 32	--> mit 32 vorher geschrieben	4		
	1	10 Mio (0,3 GB)	512m	320	--> mit 32 vorher geschrieben	4		
	1	10 Mio (0,3 GB)	512m	1 / 3	--> mit 1 vorher geschrieben	4		
	1	10 Mio (0,3 GB)	512m	320	--> mit 1 vorher geschrieben	8		

Hinweis:
Die Daten werden in mindestens so viele Partitionen eingeteilt, wie es Teildateien gibt!
Die Mindespartitionszahlen liegen bei der verwendeten Blockgröße von 128 MB bei:
- **500 Mio: 16,5 GB / 128 MB = 130 Partitionen**
- **100 Mio: 3,3 GB / 128 MB = 26 Partitionen**
- **10 Mio: 323 MB / 128 MB = 3 Partitionen**

Legende:
--> Die tatsächlich von Spark verwendete Partitionszahl weicht von der Anwenderangabe ab:
Vorgabe --> Tatsächlich (durch Teildateien beeinflusst)

Tabelle 16: Ergebnisse des Lesetests mit HDFS

A.10 Maschinelles Lernen mit MLlib

Hier finden sich die Anlagen zu Kapitel 8.2, welches sich mit dem Thema des maschinellen Lernens über Spark auseinandersetzt.

Beispiel-Applikation zur Spam-Klassifizierung

```
(1)    public class SpamClassification {
(2)
(3)     public static void main(String[] args) {
(4)
(5)       final SparkConf conf = new SparkConf();
(6)       JavaSparkContext sc = new JavaSparkContext(conf);
(7)
(8)       JavaRDD<String> spam = sc.textFile("s3n://dmueller5bucket/data/
            spam_classification/spam.csv");
(9)       JavaRDD<String> ham = sc.textFile("s3n://dmueller5bucket/data/
            spam_classification/ham.csv");
(10)
(11)      // Create a HashingTF instance to map email text to vectors of 10,000 features.
(12)      final HashingTF tf = new HashingTF(10000);
(13)
(14)      // Create LabeledPoint datasets for positive (spam) and negative (ham) examples.
(15)      JavaRDD<LabeledPoint> positives = spam.map(new Function
            <String, LabeledPoint>() {
(16)
(17)        public LabeledPoint call(String email) {
(18)          return new LabeledPoint(1, tf.transform(Arrays.asList
                (email.split(" "))));
(19)        }
(20)      });
(21)
(22)      JavaRDD<LabeledPoint> negatives = ham.map(new Function
            <String, LabeledPoint>() {
(23)
(24)        public LabeledPoint call(String email) {
(25)          return new LabeledPoint(0, tf.transform(Arrays.asList
                (email.split(" "))));
(26)        }
(27)      });
(28)
(29)      JavaRDD<LabeledPoint> trainData = positives.union(negatives);
(30)      trainData.cache(); // Cache-> Logistic Regression is an iterative algorithm.
(31)
(32)      // Run Logistic Regression using the SGD algorithm.
(33)      LogisticRegressionModel model = new LogisticRegressionWithSGD()
            .run(trainData.rdd());
(34)
(35)      // Test on a positive example (spam) and a negative one (normal).
(36)      Vector posTest = tf.transform(Arrays.asList("O M G GET cheap
            stuff by sending money to ...".split(" ")));
(37)      Vector negTest = tf.transform(Arrays.asList("Hi Dad, I started
            studying Spark the other ...".split(" ")));
(38)      System.out.println("Prediction for positive example: " +
            model.predict(posTest));
(39)      System.out.println("Prediction for negative example: " +
            model.predict(negTest));
(40)    }
(41)  }
```

Codeblock 50: Beispiel-Applikation zur Spam-Klassifizierung (angelehnt an [27])

A.11 Apache Zeppelin

Installation von Apache Zeppelin auf einem Spark-Cluster

Die folgende Anleitung beschreibt die einzelnen Schritte, die zur Installation von Apache Zeppelin auf einem Spark-Cluster durchgeführt werden müssen:

1.) Herunterladen von Spark, z. B. Version 1.3.1 (Prebuild for Hadoop 2.3)
    a. Aufsetzen eines Clusters im EC2-Umfeld entsprechend der Anleitung in A.4.
2.) Sofern noch nicht bereits geschehen, muss Maven installiert werden.[60]
    a. *sudo wget http://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-maven.repo -O /etc/yum.repos.d/epel-apache-maven.repo*
    b. *sudo sed -i s/\$releasever/6/g /etc/yum.repos.d/epel-apache-maven.repo*
    c. *sudo yum install -y apache-maven*
    d. *mvn –version*
3.) Security-Group: Port 8082-8083 (TCP) für alle Verbindungen freigeben
4.) Download von Zeppelin auf dem **Masterknoten** des Spark-Clusters[61]
    a. *git clone https://github.com/apache/incubator-zeppelin.git*
5.) Installieren von Zeppelin
    a. *cd incubator-zeppelin/*
    b. *mvn clean package -Pspark-**1.3** -Dhadoop.version=**2.3.0** -Phadoop-**2.3** -DskipTests*[62]
6.) Port korrigieren
    a. *nano conf/zeppelin-site.sh*
    b. „*zeppelin.server.port*“ auf 8082 stellen[63]
7.) Zeppelin starten
    a. *./bin/zeppelin-daemon.sh start*
8.) Die Weboberfläche steht nun zur Verfügung
    a. URL: *http://master:8082/*

Beispiel-Notebook in Zeppelin

Ein Notebook kann aus mehreren sogenannter Paragraphen bestehen. In jedem Paragraph kann dabei auf einen anderen Interpreter-Kontext zugegriffen werden. Folgendes Notebook besteht aus zwei Paragraphen: Im ersten werden per Scala-API Daten aus einer CSV-Datei per SparkContext gelesen und das daraus entstandene RDD als temporäre Tabelle für spätere SQL-Zugriffe bereitgestellt. Im zweiten Paragraph wird der SparkSQL-Kontext über *%sql* eingebunden und die oben als Tabelle zur Verfügung gestellten Daten selektiert. Über die beim SQL-Interpreter zur Verfügung stehenden Pivot-Funktionalitäten können die Daten dann visuell dargestellt werden.

[60] Quelle der Anleitung: https://gist.github.com/sebsto/19b99f1fa1f32cae5d00
[61] Über GitHub können auch veröffentlichte Releases heruntergeladen werden. Zum aktuellen Zeitpunkt (30.07.2015) gibt es lediglich Version 0.5.0 – Diese ließ sich nicht korrekt installieren und sollte deshalb nicht verwendet werden! Statt Version 0.5.0 kann einfach das aktuelle Repository zur Installation verwendet werden.
[62] Die Angaben zur Spark- und Hadoop-Version müssen mit den installierten übereinstimmen (s. Schritt 1)
[63] Port 8080 wird bereits vom Spark-Driver verwendet, deshalb muss auf Port 8082 ausgewichen werden.

Paragraph 1: Daten einlesen in RDD und erzeugen einer SQL-Tabelle

```
(1)   // Set the credentials to be able to connect to S3
(2)   sc.hadoopConfiguration.set("fs.s3n.awsAccessKeyId", "xxx")
(3)   sc.hadoopConfiguration.set("fs.s3n.awsSecretAccessKey","xxx")
(4)
(5)   // Read in a csv file to RDD
(6)   val windowed = sc.textFile("s3n://dmueller5bucket/data/csv_splits/
        sorted100M_1000.csv").cache()
(7)
(8)   // Define a object class (here for a DEBS dataset)
(9)   case class DebsClass(id:Integer, ts:Long, value:Double, typ:Int,
        plugid:Int, householdid:Int, houseid:Int)
(10)
(11)  // Map the rdd data to that object (for querying later on)
(12)  val data = windowed.map(s=>s.split(","))
        .map(s=>DebsClass(s(0).toInt, s(1).toLong, s(2).toDouble,
          s(3).toInt, s(4).toInt, s(5).toInt, s(6).toInt))
(13)
(14)  // Create a virtual temp table for querying later on
(15)  data.toDF().registerTempTable("debs")
```

Codeblock 51: Daten mit Zeppelin einlesen und als SQL-Tabelle zur Verfügung stellen

Paragraph 2: Datenselektion über SparkSQL-Kontext

```
(1)   %sql
(2)   SELECT * FROM debs WHERE typ = 1
```

Codeblock 52: Datenselektion über SQL-Interpreter

B Analyse von Streaming Data mit Spark

Hier sind die Quellcodes der Applikationen zu finden, die in den einzelnen Phasen der Datenvorbereitung und Umsetzung zum Einsatz kamen.

B.1 Data Understanding

Der folgende Anwendungs-Quellcode dient zur Untersuchung der CSV-Datei, welche die Messergebnisse der DEBS Grand Challenge 2014 enthält. Neben der Anzahl an Zeilen innerhalb der Datei werden Informationen zur Anzahl Stromzähler, Häuser und Haushalte zusammengetragen.

```
(1)   public class PropertiesOfDEBS {
(2)
(3)     @SuppressWarnings("serial")
(4)     public static void main(String[] args) {
(5)       // 0.) Preparing
(6)       SparkConf conf = new SparkConf();
(7)       JavaSparkContext sc = new JavaSparkContext(conf);
(8)
(9)       // 1.) Read CSV file into RDD
(10)      String s3path = "s3n://dmueller5bucket/debs/sorted_unpacked.csv";
(11)      setS3credentials(sc, s3path);
(12)      JavaRDD<String> csv = sc.textFile(s3path, 320)
           .persist(StorageLevel.MEMORY_AND_DISK());
(13)
(14)      // 2.) Plug count RDD
(15)      JavaRDD<String> plugCountRDD = csv.map(new Function<String,
           String>() {
(16)        public String call(String row) throws Exception {
(17)          String[] data = row.split(",");
(18)          // plug.household.house
(19)          return data[4] + "." + data[5] + "." + data[6];
(20)        }
(21)      }).distinct();
(22)
(23)      // 3.) Household count RDD
(24)      JavaRDD<String> householdCountRDD = csv.map(new Function<String,
           String>() {
(25)        public String call(String row) throws Exception {
(26)          String[] data = row.split(",");
(27)          // household.house
(28)          return  data[5] + "." + data[6];
(29)        }
(30)      }).distinct();
(31)
(32)      // 4.) House count RDD
(33)      JavaRDD<String> houseCountRDD = csv.map(new Function<String,
           String>() {
(34)        public String call(String row) throws Exception {
(35)          String[] data = row.split(",");
(36)          // house
(37)          return data[6];
(38)        }
(39)      }).distinct();
(40)
(41)      // 5.) Output the results
```

```
(42)      System.out.println("Households : " + householdCountRDD.count());
(43)      System.out.println("Houses : " + houseCountRDD.count());
(44)      System.out.println("Plugs : " + plugCountRDD.count());
(45)      System.out.println("Entries in CSV : " + csv.count());
(46)    }
(47)
(48)    static void setS3credentials(JavaSparkContext sc, String path) {
(49)      if (path.startsWith("s3n://")) {
(50)        Configuration hadoopConf = sc.hadoopConfiguration();
(51)        hadoopConf.set("fs.s3n.impl",
             "org.apache.hadoop.fs.s3native.NativeS3FileSystem");
(52)        hadoopConf.set("fs.s3.impl",
             "org.apache.hadoop.fs.s3native.NativeS3FileSystem");
(53)        hadoopConf.set("fs.s3n.awsAccessKeyId", "xxx");
(54)        hadoopConf.set("fs.s3n.awsSecretAccessKey", "xxx");
(55)      }
(56)    }
(57)  }
```

Codeblock 53: Spark-Applikation zur Untersuchung der DEBS CSV-Datei

B.2 Data Preparation

Der folgende Codeblock enthält die Logik des Servers, der die CSV mit DEBS-Daten ausliest und die Einträge zeilenweise über zwei TCP-Sockets sendet (es können durch Änderung des Parameters auch mehrere Sockets angelegt werden):

```
(1)   public class SendingServer extends Thread {
(2)     ServerSocket[] serverSocket;
(3)     Socket[] client;
(4)     int[] port;
(5)
(6)     // Entry point: Start sending server thread
(7)     public static void main(String[] args) {
(8)       try {
(9)         SendingServerThread server =  new SendingServerThread(2);
(10)        server.start();
(11)        server.join();
(12)      } catch (Exception e) {
(13)        e.printStackTrace();
(14)      }
(15)    }
(16)
(17)    // Constructor: Define instances to send
(18)    public SendingServerThread(int instances) {
(19)      this.serverSocket = new ServerSocket[instances];
(20)      this.client = new Socket[instances];
(21)      this.port = new int[instances];
(22)    }
(23)
(24)    // Thread run method, the send logic
(25)    @Override
(26)    public void run() {
(27)      BufferedReader reader = null;
(28)      BasicAWSCredentials awsCreds = new BasicAWSCredentials("x","x");
(29)      AmazonS3 s3Client = new AmazonS3Client(awsCreds);
(30)      int counter;
(31)
(32)      try {
(33)        // Connect
```

```
(34)        waitForConnectionAndConnect();
(35)        Thread.sleep(5000);
(36)
(37)        // Get CSV from S3 and create the reader for it
(38)        S3Object s3Obj = s3Client.getObject("dmueller5bucket",
             "debs/sorted_unpacked.csv");
(39)        reader = new BufferedReader(new InputStreamReader
             (s3Obj.getObjectContent()));
(40)
(41)        // Preparation for sending
(42)        counter = 0;
(43)        String row;
(44)        long actualTimestamp = 0L;
(45)        long timestamp = 0L;
(46)        boolean first = true;
(47)        long intervalStart = System.currentTimeMillis();
(48)
(49)        // Iterate the CSV file row by row
(50)        while ((row = reader.readLine()) != null) {
(51)          String[] splits = row.split(",");
(52)
(53)          // Set first timestamp
(54)          if(first) {
(55)            actualTimestamp = Long.valueOf(splits[1]);
(56)            first = false;
(57)          }
(58)
(59)          // Read timestamp from row
(60)          timestamp = Long.valueOf(splits[1]);
(61)
(62)          // If newer timestamp, pause till the 1-sec interval is over
(63)          if(timestamp >= actualTimestamp + 1) {
(64)            System.out.println("Took : " + (System.currentTimeMillis()
               - intervalStart) + " ms.");
(65)            while(System.currentTimeMillis() < intervalStart + 1000) ;
(66)            intervalStart = System.currentTimeMillis();
(67)            actualTimestamp = timestamp;
(68)          }
(69)
(70)          // Only send type 1 data!!!!
(71)          if(splits[3].equals("1")) {
(72)            writeMessage(row + "\n", counter);
(73)            counter++;
(74)          }
(75)
(76)          // User information about the sent messages yet
(77)          if (counter % 10000 == 0) {
(78)            System.out.println("***** SenderServer sent " + counter +
               " data points yet!");
(79)          }
(80)        }
(81)
(82)        // Close reader and open connections
(83)        reader.close();
(84)        closeConnections();
(85)      } catch (Exception e) {
(86)        e.printStackTrace();
(87)        System.out.println("***** " + e.getMessage());
(88)      }
(89)    }
```

```
(90)
(91)    // Helper method for connecting the TPC sockets
(92)    private void waitForConnectionAndConnect() throws IOException {
(93)      for(int i = 0; i < serverSocket.length; i++) {
(94)        serverSocket[i] = new ServerSocket(7777 + i);
(95)      }
(96)      System.out.println("***** Servers: Wait for connection");
(97)
(98)      for(int i = 0; i < client.length; i++) {
(99)        client[i] = serverSocket[i].accept();
(100)     }
(101)     System.out.println("***** Servers: Connected!");
(102)
(103)     for(int i = 0; i < client.length; i++) {
(104)       while(!client[i].isConnected()) ;
(105)       System.out.println("***** Server " + i + " : started
            successfully!");
(106)     }
(107)   }
(108)
(109)   // Helper method for closing the TCP connections
(110)   private void closeConnections() throws IOException {
(111)     for(int i = 0; i < client.length; i++) {
(112)       client[i].close();
(113)       serverSocket[i].close();
(114)     }
(115)   }
(116)
(117)   // Helper method to send a message via one of TCP sockets
(118)   private void writeMessage(String message, int counter)
         throws IOException {
(119)     PrintWriter printWriter;
(120)     printWriter = new PrintWriter(client[counter % client.length]
           .getOutputStream(), true);
(121)     printWriter.print(message);
(122)     printWriter.flush();
(123)   }
(124) }
```

Codeblock 54: Quellcode zum Sender der DEBS-Daten per TCP-Sockets

B.3 Modeling – Arithmetisches Mittel

In diesem Kapitel findet sich der Quellcode der Spark-Applikation zur Ausreißererkennung mithilfe der Berechnung des arithmetischen Mittels.

```
(1)   public class ByAverage {
(2)
(3)     @SuppressWarnings("serial")
(4)     public static void main(String[] args) {
(5)       // 0) Preparation
(6)       final SparkConf conf = new SparkConf();
(7)       conf.set("spark.driver.memory", "5g");
(8)       conf.set("spark.executor.memory", "5g");
(9)
(10)      int receiverInstances = 2;
(11)      int batchIntervalSec = Integer.valueOf(args[1]);
(12)      int windowSize1hSek = 60 * 60;
(13)      int slideDurationSek = batchIntervalSec;
(14)
(15)      final JavaStreamingContext streamingCtx = new
           JavaStreamingContext(conf, Durations.seconds(batchIntervalSec));
(16)
(17)      Configuration hadoopConf = streamingCtx.sparkContext()
           .hadoopConfiguration();
(18)      hadoopConf.set("fs.s3n.impl", "org.apache.hadoop.fs.s3native
           .NativeS3FileSystem");
(19)      hadoopConf.set("fs.s3.impl", "org.apache.hadoop.fs.s3native
           .NativeS3FileSystem");
(20)      hadoopConf.set("fs.s3n.awsAccessKeyId", "xxx");
(21)      hadoopConf.set("fs.s3n.awsSecretAccessKey", "xxx");
(22)
(23)      streamingCtx.checkpoint("s3n://dmueller5bucket/data/checkpts");
(24)
(25)      List<JavaDStream<String>> receivedDStreams = new ArrayList
           <JavaDStream<String>>();
(26)      for (int i = 0; i < receiverInstances; i++) {
(27)        receivedDStreams.add(streamingCtx.socketTextStream(args[0],
             7777 + i));
(28)      }
(29)      JavaDStream<String> inputDStream;
(30)      if (receivedDStreams.size() > 1) {
(31)        inputDStream = streamingCtx.union(receivedDStreams.get(0),
             receivedDStreams.subList(1, receivedDStreams.size()));
(32)      } else {
(33)        inputDStream = receivedDStreams.get(0);
(34)      }
(35)
(36)      // 1.) Map to pair: (houseId.householdId.plugId.TS, val)
(37)      JavaPairDStream<String, Float> debsPairDStream =
           inputDStream.mapToPair(mapInputDStreamToPairDStreamFunction);
(38)
(39)      // 2.) Remove duplicates
(40)      JavaPairDStream<String, Float> debsPairDStreamWithoutDuplicates =
           debsPairDStream.transformToPair(removeDuplicatesFunction);
(41)
(42)      // 3.) Filter consumptions <= 0 out
(43)      JavaPairDStream<String, Float> debsPairDStreamConsumptionGreater0
           = debsPairDStreamWithoutDuplicates.filter(filterConsumption-
           Greater0Function);
(44)
```

```
(45)      // 4.) Map to new pair: (houseId.householdId.plugId, val)
(46)      JavaPairDStream<String, Float> debsPairDStreamConsumption-
           Greater0withoutTs = debsPairDStreamConsumptionGreater0
           .mapToPair(removeTsFromPairDStreamFunction).cache();
(47)
(48)      // 5.) Average per Plug
(49)      // 5.1) Create a count-prepared PairDStream (key, 1)
(50)      JavaPairDStream<String, Long> countPreparedPerPlug1h =
           debsPairDStreamConsumptionGreater0withoutTs
           .mapToPair(prepareCountFunction);
(51)
(52)      // 5.2) Count all the elements by plug and in window
(53)      JavaPairDStream<String, Long> countPerPlug1h = countPreparedPer-
           Plug1h.reduceByKeyAndWindow(addLongsFunction, subtractLongs-
           Function, Durations.seconds(windowSize1hSek),
           Durations.seconds(slideDurationSek));
(54)
(55)      // 5.3) Sum up the consumption values by plug and window
(56)      JavaPairDStream<String, Float> sumPerPlug1h = debsPairDStream-
           ConsumptionGreater0withoutTs.reduceByKeyAndWindow(addFloats-
           Function, subtractFloatsFunction, Durations.seconds
           (windowSize1hSek), Durations.seconds(slideDurationSek));
(57)
(58)      // 5.4) Calculate the average by sum/count
(59)      JavaPairDStream<String, Float> avgPerPlug1h = sumPerPlug1h
           .join(countPerPlug1h).mapToPair(calculateAverageFunction);
(60)
(61)      // 6.) Average of all plug data of window
(62)      // 6.1) Map to new pair of ("all", val)
(63)      JavaPairDStream<String, Float> mappedValue = debsPairDStream-
           ConsumptionGreater0withoutTs.mapToPair(mapToAllPairFunction)
           .cache();
(64)
(65)      // 6.2) Create a count-prepared PairDStream (key, 1)
(66)      JavaPairDStream<String, Long> countPreparedAll1h = mappedValue
           .mapToPair(prepareCountFunction);
(67)
(68)      // 6.3) Count all the elements in window
(69)      JavaPairDStream<String, Long> countAll1h = countPreparedAll1h
           .reduceByKeyAndWindow(addLongsFunction, subtractLongsFunction,
           Durations.seconds(windowSize1hSek),
           Durations.seconds(slideDurationSek));
(70)
(71)      // 6.4) Sum up the consumption values of all plugs and in window
(72)      JavaPairDStream<String, Float> sumAll1h = mappedValue.reduce-
           ByKeyAndWindow(addFloatsFunction, subtractFloatsFunction,
           Durations.seconds(windowSize1hSek),
           Durations.seconds(slideDurationSek));
(73)
(74)      // 6.5) Calculate the average by sum/count
(75)      JavaPairDStream<String, Float> avgAll1h = sumAll1h
           .join(countAll1h).mapToPair(calculateAverageFunction);
(76)
(77)      // 7.) Check for outliers per comparison of the 2 DStreams
(78)      // 7.1) Union the median results (per Plug and all plugs) and
                  calculate outliers
(79)      JavaPairDStream<String, Boolean> outlierPairDStream = avgAll1h
           .union(avgPerPlug1h).transformToPair(findOutliersPerComparison-
           Function);
(80)
```

```
(81)      // 8.) Calculate the percentage of outliers per house
(82)      // 8.1) Map to new PairDStream (houseId, val) and cache for reuse
(83)      JavaPairDStream<String, Boolean> outlierPerHousePairDStream =
           outlierPairDStream.mapToPair(removeHouseholdAndPlugIdFrom-
           OutlierPairDStreamFunction).cache();
(84)
(85)      // 8.2) Create helper-stream to sum up the outliers per house
                  (houseId, 1 if true / 0 if false)
(86)      JavaPairDStream<String, Long> countPreparedOutliersPerHouse =
           outlierPerHousePairDStream.mapToPair(prepareCountOutliers-
           PerHouseFunction);
(87)
(88)      // 8.3) Create helper-stream to sum up all elements per house
                  (houseId, 1);
(89)      JavaPairDStream<String, Long> countDataAmountPerHousePrepared =
           outlierPerHousePairDStream.mapToPair(prepareCountAll-
           PerHouseFunction);
(90)
(91)      // 8.4) Count the real outliers per house (val=1)
(92)      JavaPairDStream<String, Long> countOutlierPerHouse =
           countPreparedOutliersPerHouse.reduceByKey(sumUpLongsFunction);
(93)
(94)      // 8.5) Sum up all elements per house
(95)      JavaPairDStream<String, Long> countDataAmountPerHouse =
           countDataAmountPerHousePrepared.reduceByKey(sumUpLongsFunction);
(96)
(97)      // 8.6) Join and calculate the percentage of outliers per house
(98)      JavaPairDStream<String, Float> outlierPercentagePerHouse =
           countOutlierPerHouse.join(countDataAmountPerHouse)
           .mapToPair(calculatePercentageOfOutliersPerHouse);
(99)
(100)     // 8.7) Print the percentage of outliers per house
(101)     outlierPercentagePerHouse.print();
(102)
(103)     // 9.) Start streaming application
(104)     streamingCtx.start();
(105)     streamingCtx.awaitTermination();
(106)     streamingCtx.close();
(107)   }
(108)
(109)   // Here are missing the preparation methods of the data preparation
(110)   // Step 1-4
(111)
(112)   private static PairFunction<Tuple2<String, Float>, String, Long>
         prepareCountFunction = new PairFunction<Tuple2<String, Float>,
         String, Long>() {
(113)     public Tuple2<String, Long> call(Tuple2<String, Float> t) {
(114)       return new Tuple2<String, Long>(t._1(), 1L);
(115)     }
(116)   };
(117)
(118)   private static Function2<Long, Long, Long> addLongsFunction = new
         Function2<Long, Long, Long>() {
(119)     public Long call(Long v1, Long v2) throws Exception {
(120)       return v1 + v2;
(121)     }
(122)   };
(123)
(124)   private static Function2<Long, Long, Long> subtractLongsFunction =
         new Function2<Long, Long, Long>() {
```

```
(125)      public Long call(Long v1, Long v2) throws Exception {
(126)        return v1 - v2;
(127)      }
(128)    };
(129)
(130)    private static Function2<Float, Float, Float> addFloatsFunction =
           new Function2<Float, Float, Float>() {
(131)      public Float call(Float v1, Float v2) throws Exception {
(132)        return v1 + v2;
(133)      }
(134)    };
(135)
(136)    private static Function2<Float, Float, Float>
           subtractFloatsFunction = new Function2<Float, Float, Float>() {
(137)      public Float call(Float v1, Float v2) throws Exception {
(138)        return v1 - v2;
(139)      }
(140)    };
(141)
(142)    private static PairFunction<Tuple2<String, Tuple2<Float, Long>>,
           String, Float> calculateAverageFunction = new PairFunction
           <Tuple2<String, Tuple2<Float, Long>>, String, Float>() {
(143)      public Tuple2<String, Float> call(Tuple2<String, Tuple2<Float,
             Long>> t) throws Exception {
(144)        String key = t._1();
(145)        Float sum = t._2()._1();
(146)        long count = t._2()._2();
(147)        Float avg = sum / (float) count;
(148)        return new Tuple2<String, Float>(key, avg);
(149)      }
(150)    };
(151)
(152)    private static PairFunction<Tuple2<String, Float>, String, Float>
           mapToAllPairFunction = new PairFunction<Tuple2<String, Float>,
           String, Float>() {
(153)      public Tuple2<String, Float> call(Tuple2<String, Float> t) {
(154)        return new Tuple2<String, Float>("all", t._2());
(155)      }
(156)    };
(157)
(158)    private static Function<JavaPairRDD<String,Float>, JavaPairRDD
           <String,Boolean>> findOutliersPerComparisonFunction = new
           Function<JavaPairRDD<String,Float>,JavaPairRDD<String,Boolean>>(){
(159)      public JavaPairRDD<String, Boolean> call(JavaPairRDD<String,
             Float> v1) throws Exception {
(160)        float avgOfAll;
(161)        if(v1.count() > 0) {
(162)          avgOfAll = v1.filter(new Function<Tuple2<String,Float>,
                 Boolean>() {
(163)            public Boolean call(Tuple2<String, Float> v1) {
(164)              return v1._1().equals("all");
(165)            }
(166)          }).values().collect().get(0);
(167)        } else {
(168)          avgOfAll = 0.0f;
(169)        }
(170)        final float finalAvg = avgOfAll;
(171)
(172)        return v1.mapValues(new Function<Float, Boolean>() {
(173)          public Boolean call(Float v1) throws Exception {
```

```
(174)           return v1 > finalAvg;
(175)         }
(176)       }).filter(new Function<Tuple2<String,Boolean>, Boolean>() {
(177)         public Boolean call(Tuple2<String, Boolean> v1) {
(178)           return !v1._1().equals("all");
(179)         }
(180)       });
(181)     }
(182)   };
(183)
(184)   private static PairFunction<Tuple2<String, Boolean>, String,
         Boolean> removeHouseholdAndPlugIdFromOutlierPairDStreamFunction =
         new PairFunction<Tuple2<String,Boolean>, String, Boolean>() {
(185)     public Tuple2<String, Boolean> call(Tuple2<String, Boolean> t) {
(186)       String[] keyParts = t._1().split("\\.");
(187)       String newKey = keyParts[0];
(188)       return new Tuple2<String, Boolean>(newKey, t._2());
(189)     }
(190)   };
(191)
(192)   private static PairFunction<Tuple2<String,Boolean>, String, Long>
         prepareCountOutliersPerHouseFunction = new
         PairFunction<Tuple2<String,Boolean>, String, Long>() {
(193)     public Tuple2<String, Long> call(Tuple2<String, Boolean> t) {
(194)       if(t._2()) {
(195)         return new Tuple2<String, Long>(t._1(), 1L);
(196)       } else {
(197)         return new Tuple2<String, Long>(t._1(), 0L);
(198)       }
(199)     }
(200)   };
(201)
(202)   private static PairFunction<Tuple2<String,Boolean>, String, Long>
         prepareCountAllPerHouseFunction = new PairFunction<Tuple2<String,
         Boolean>, String, Long>() {
(203)     public Tuple2<String, Long> call(Tuple2<String, Boolean> t) {
(204)       return new Tuple2<String, Long>(t._1(), 1L);
(205)     }
(206)   };
(207)
(208)   private static Function2<Long, Long, Long> sumUpLongsFunction =
         new Function2<Long, Long, Long>() {
(209)     public Long call(Long v1, Long v2) throws Exception {
(210)       return v1 + v2;
(211)     }
(212)   };
(213)
(214)   private static PairFunction<Tuple2<String,Tuple2<Long,Long>>,
         String, Float> calculatePercentageOfOutliersPerHouse = new
         PairFunction<Tuple2<String,Tuple2<Long,Long>>, String, Float>() {
(215)     public Tuple2<String, Float> call(Tuple2<String, Tuple2<Long,
           Long>> t) throws Exception {
(216)       float newValue = (float)t._2()._1() / (float)t._2()._2();
(217)       return new Tuple2<String, Float>(t._1(), newValue);
(218)     }
(219)   };
(220) }
```

Codeblock 55: Applikation zur Ausreißer-Ermittlung via arithmetischem Mittel

B.4 Modeling – Median

In diesem Kapitel findet sich der Quellcode der Spark-Applikation zur Ausreißererkennung mithilfe der Berechnung Medians.

```
(1)    public class ByMedian {
(2)
(3)      @SuppressWarnings("serial")
(4)      public static void main(String[] args) {
(5)        // 0) Preparation is the same as in the ByAverage-application
(6)
(7)        // 1-4) Step 1-4 are also the same as above
(8)
(9)        // 5.) Gather data of window
(10)       JavaPairDStream<String, Float> debsPairDStreamConsumption-
            Greater0WithoutTs1h = debsPairDStreamConsumptionGreater0-
            withoutTs.window(Durations.seconds(windowSize1hSek),
            Durations.seconds(slideDurationSek)).cache();
(11)
(12)       // 6.) Median per Plug
(13)       // 6.1) Group the data by key (= plug) and find the median of the
                   windowed data
(14)       JavaPairDStream<String, Float> medianPerPlug1h = debsPairDStream-
            ConsumptionGreater0WithoutTs1h.groupByKey()
            .transformToPair(findMedianOfWindowFunction);
(15)
(16)       // 7.) Median of all Plugs
(17)       // 7.1) Map to new pair of ("all", val)
(18)       JavaPairDStream<String, Float> mappedValues = debsPairDStream-
            ConsumptionGreater0WithoutTs1h.mapToPair(new PairFunction
            <Tuple2<String,Float>, String, Float>() {
(19)         public Tuple2<String, Float> call(Tuple2<String, Float> t) {
(20)           return new Tuple2<String, Float> ("all", t._2());
(21)         }
(22)       });
(23)
(24)       // 7.2) Group the data by key (= "all") and find the median of
                   the windowed data
(25)       JavaPairDStream<String, Float> medianAll1h = mappedValues
            .groupByKey().transformToPair(findMedianOfWindowFunction);
(26)
(27)       // 8.) Check for outliers per comparison of the 2 DStreams
                  -->is the same as Step 7 in ByAverage application
(28)
(29)       // 9.) Calculate the percentage of outliers per house
                  --> is the same as Step 8 in ByAverage application
(30)
(31)       // 10.) Start streaming application
(32)       streamingCtx.start();
(33)       streamingCtx.awaitTermination();
(34)       streamingCtx.close();
(35)     }
(36)
(37)     // Here are missing the preparation methods of the data preparation
(38)     // Step 1-4 are the same
(39)
(40)     private static Function<JavaPairRDD<String, Iterable<Float>>,
          JavaPairRDD<String, Float>> findMedianOfWindowFunction = new
          Function<JavaPairRDD<String, Iterable<Float>>, JavaPairRDD<String,
          Float>>() {
```

```
(41)      public JavaPairRDD<String, Float> call(JavaPairRDD<String,
           Iterable<Float>> v1) throws Exception {
(42)        return v1.mapValues(new Function<Iterable<Float>, Float>() {
(43)          public Float call(Iterable<Float> v1) throws Exception {
(44)            List<Float> buffer = new ArrayList<Float>();
(45)            long count = 0L;
(46)            Iterator<Float> iterator = v1.iterator();
(47)            while (iterator.hasNext()) {
(48)              buffer.add(iterator.next());
(49)              count++;
(50)            }
(51)            float[] values = new float[(int) count];
(52)            for (int i = 0; i < buffer.size(); i++) {
(53)              values[i] = buffer.get(i);
(54)            }
(55)            Arrays.sort(values);
(56)
(57)            float median;
(58)            int startIndex;
(59)            if (count % 2 == 0) {
(60)              startIndex = (int) (count / 2 - 1);
(61)              float a = values[startIndex];
(62)              float b = values[startIndex + 1];
(63)              median = (a + b) / 2.0f;
(64)            } else {
(65)              startIndex = (int) (count / 2);
(66)              median = values[startIndex];
(67)            }
(68)            return median;
(69)          }
(70)        });
(71)      }
(72)    };
(73)
(74)    // Here are missing the outlier functions
           --> They are the same as in ByAverage application
(75)
(76)  }
```

Codeblock 56: Applikation zur Ausreißer-Ermittlung via Median-Bestimmung

B.5 Modeling – Arithmetisches Mittel der Mediane

In diesem Kapitel findet sich der Quellcode der Spark-Applikation zur Ausreißererkennung mithilfe des gewichteten arithmetischen Mittels der Mediane.

```
(1)    public class ByAverageOfMedians {
(2)
(3)      @SuppressWarnings("serial")
(4)      public static void main(String[] args) {
(5)        // 0) Preparation is the same as in the ByAverage-application
(6)
(7)        // 1-4) Step 1-4 are also the same as above
(8)
(9)        // 5.) Average of Medians per plug
(10)       // 5.1) Group the data by key (= house.household.plug) and find
              the median of the batch data
(11)       JavaPairDStream<String, Tuple2<Float, Long>> medianPerPlugPer-
            Batch = debsPairDStreamConsumptionGreater0withoutTs.groupByKey()
            .transformToPair(findMedianInBatchFunction).cache();
(12)
(13)       // 5.2) Split the data into PairDStream for summing up
(14)       JavaPairDStream<String, Float> medianPerPlugPerBatchSum =
            medianPerPlugPerBatch.mapToPair(mapToSumFunction);
(15)
(16)       // 5.3) Split the data into PairDStream for counting
(17)       JavaPairDStream<String, Long> medianPerPlugPerBatchCount =
            medianPerPlugPerBatch.mapToPair(mapToCountFunction);
(18)
(19)       // 5.4) Sum up by key
(20)       JavaPairDStream<String, Float> medianPerPlugPerBatchSum1h =
            medianPerPlugPerBatchSum.reduceByKeyAndWindow(addFloatsFunction,
            subtractFloatsFunction, Durations.seconds(windowSize1hSek),
            Durations.seconds(slideDurationSek));
(21)
(22)       // 5.5) Count by key
(23)       JavaPairDStream<String, Long> medianPerPlugPerBatchCount1h =
            medianPerPlugPerBatchCount.reduceByKeyAndWindow(addLongsFunction
            ,subtractLongsFunction, Durations.seconds(windowSize1hSek),
            Durations.seconds(slideDurationSek));
(24)
(25)       // 5.6) Calculate averagy by sum/count and by key and by window
(26)       JavaPairDStream<String, Float> averageOfMediansPerPlug1h =
            medianPerPlugPerBatchSum1h.join(medianPerPlugPerBatchCount1h)
            .mapToPair(calculateAverageFunction);
(27)
(28)       // 6.) Average of Medians of all plugs
(29)       // 6.1) Map to new pair of ("all", val)
(30)       JavaPairDStream<String, Float> mappedValues = debsPairDStream-
            ConsumptionGreater0withoutTs.mapToPair(mapToAllPairFunction);
(31)
(32)       // 6.2) Group the data by key (= "all") and find the median of
                   the batch data
(33)       JavaPairDStream<String, Tuple2<Float, Long>> medianAllPerBatch =
            mappedValues.groupByKey().transformToPair(findMedianInBatch-
            Function).cache();
(34)
(35)       // 6.3) Split the data into PairDStream for summing up
(36)       JavaPairDStream<String, Float> medianAllPerBatchSum =
            medianAllPerBatch.mapToPair(mapToSumFunction);
(37)
```

```
(38)      // 6.4) Split the data into PairDStream for counting
(39)      JavaPairDStream<String, Long> medianAllPerBatchCount =
           medianAllPerBatch.mapToPair(mapToCountFunction);
(40)
(41)      // 6.5) Sum up by key
(42)      JavaPairDStream<String, Float> medianAllPerBatchSum1h =
           medianAllPerBatchSum.reduceByKeyAndWindow(addFloatsFunction,
           subtractFloatsFunction, Durations.seconds(windowSize1hSek),
           Durations.seconds(slideDurationSek));
(43)
(44)      // 6.6) Count by key
(45)      JavaPairDStream<String, Long> medianAllPerBatchCount1h =
           medianAllPerBatchCount.reduceByKeyAndWindow(addLongsFunction,
           subtractLongsFunction, Durations.seconds(windowSize1hSek),
           Durations.seconds(slideDurationSek));
(46)
(47)      // 6.7) Calculate averagy by sum/count and by key and by window
(48)      JavaPairDStream<String, Float> averageOfMediansAll1h =
           medianAllPerBatchSum1h.join(medianAllPerBatchCount1h).mapToPair(
           calculateAverageFunction);
(49)
(50)      // 7.) Check for outliers per comparison of the 2 DStreams
(51)             -->is the same as Step 7 in ByAverage application
(52)
(53)      // 8.) Calculate the percentage of outliers per house
(54)             --> is the same as Step 8 in ByAverage application
(55)
(56)      // 9.) Start streaming application
(57)      streamingCtx.start();
(58)      streamingCtx.awaitTermination();
(59)      streamingCtx.close();
(60)    }
(61)
(62)    // Here are missing the preparation methods of the data preparation
(63)    // Step 1-4 are the same
(64)
(65)    private static Function<JavaPairRDD<String, Iterable<Float>>,
          JavaPairRDD<String, Tuple2<Float, Long>>> findMedianInBatch-
          Function = new Function<JavaPairRDD<String, Iterable<Float>>,
          JavaPairRDD<String, Tuple2<Float, Long>>>() {
(66)      public JavaPairRDD<String, Tuple2<Float, Long>>
           call(JavaPairRDD<String, Iterable<Float>> v1) throws Exception {
(67)        return v1.mapValues(new Function<Iterable<Float>, Tuple2<Float,
             Long>>() {
(68)          public Tuple2<Float, Long> call(Iterable<Float> v1) {
(69)            List<Float> buffer = new ArrayList<Float>();
(70)            long count = 0L;
(71)            Iterator<Float> iterator = v1.iterator();
(72)            while (iterator.hasNext()) {
(73)              buffer.add(iterator.next());
(74)              count++;
(75)            }
(76)            float[] values = new float[(int) count];
(77)            for (int i = 0; i < buffer.size(); i++) {
(78)              values[i] = buffer.get(i);
(79)            }
(80)            Arrays.sort(values);
(81)
(82)            float median;
(83)            int startIndex;
```

```
(84)
(85)            if (count % 2 == 0) {
(86)              startIndex = (int) (count / 2 - 1);
(87)              float a = values[startIndex];
(88)              float b = values[startIndex + 1];
(89)              median = (a + b) / 2.0f;
(90)            } else {
(91)              startIndex = (int) (count / 2);
(92)              median = values[startIndex];
(93)            }
(94)            return new Tuple2<Float, Long>(median, count);
(95)          }
(96)        });
(97)      }
(98)    };
(99)
(100)   private static PairFunction<Tuple2<String, Float>, String, Float>
         mapToAllPairFunction = new PairFunction<Tuple2<String, Float>,
         String, Float>() {
(101)     public Tuple2<String, Float> call(Tuple2<String, Float> v1) {
(102)       return new Tuple2<String, Float>("all", v1._2());
(103)     }
(104)   };
(105)
(106)   private static PairFunction<Tuple2<String,Tuple2<Float,Long>>,
         String, Float> mapToSumFunction = new PairFunction<Tuple2<String,
         Tuple2<Float,Long>>, String, Float>() {
(107)     public Tuple2<String, Float> call(Tuple2<String, Tuple2<Float,
          Long>> t) throws Exception {
(108)       float newValue = t._2()._1() * t._2()._2();
(109)       return new Tuple2<String, Float>(t._1(), newValue);
(110)     }
(111)   };
(112)
(113)   private static PairFunction<Tuple2<String,Tuple2<Float,Long>>,
         String, Long> mapToCountFunction = new
         PairFunction<Tuple2<String,Tuple2<Float,Long>>, String, Long>() {
(114)     public Tuple2<String, Long> call(Tuple2<String, Tuple2<Float,
          Long>> t) throws Exception {
(115)       return new Tuple2<String, Long>(t._1(), t._2()._2());
(116)     }
(117)   };
(118)
(119)   private static Function2<Long, Long, Long> addLongsFunction = new
         Function2<Long, Long, Long>() {
(120)     public Long call(Long v1, Long v2) throws Exception {
(121)       return v1 + v2;
(122)     }
(123)   };
(124)
(125)   private static Function2<Long, Long, Long> subtractLongsFunction =
         new Function2<Long, Long, Long>() {
(126)     public Long call(Long v1, Long v2) throws Exception {
(127)       return v1 - v2;
(128)     }
(129)   };
(130)
(131)   private static Function2<Float, Float, Float> addFloatsFunction =
         new Function2<Float, Float, Float>() {
(132)     public Float call(Float v1, Float v2) throws Exception {
```

```
(133)       return v1 + v2;
(134)     }
(135)   };
(136)
(137)   private static Function2<Float, Float, Float> subtractFloats-
         Function = new Function2<Float, Float, Float>() {
(138)     public Float call(Float v1, Float v2) throws Exception {
(139)       return v1 - v2;
(140)     }
(141)   };
(142)
(143)   private static PairFunction<Tuple2<String,Tuple2<Float,Long>>,
         String, Float> calculateAverageFunction = new PairFunction<Tuple2
         <String,Tuple2<Float,Long>>, String, Float>() {
(144)     public Tuple2<String, Float> call(Tuple2<String, Tuple2<Float,
           Long>> t) throws Exception {
(145)       String key = t._1();
(146)       Float sum = t._2()._1();
(147)       long count = t._2()._2();
(148)       Float avg = sum / (float) count;
(149)       return new Tuple2<String, Float>(key, avg);
(150)     }
(151)   };
(152)
(153)   // Here are missing the outlier functions (7.x & 8.x)
(154)   // --> They are the same as in ByAverage application
(155) }
```

Codeblock 57: Applikation zur Ausreißer-Ermittlung via arithm. Mittel der Mediane