Untersuchung der begrenzenden Faktoren und Optimierungsmöglichkeiten von Apache Spark zur Aufbereitung von Massendaten in der Praxis

Bachelor – Thesis

Zur Erlangung des akademischen Grades Bachelor of Science

Alina Böttcher

2096184

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik
Studiengang Media Systems

Erstprüfer: Dr.-Ing. Sabine Schumann
Zweitprüfer: Dipl.-Inform. (FH) Uwe Zenker

Hamburg, den 06.12.2017

Zusammenfassung

In dieser Arbeit werden die optimalen Konfigurationseinstellungen, das am besten geeignete Dateiformat und die Grenzen von Apache Spark anhand eines Praxisszenarios untersucht. Im ersten Schritt wird geklärt, was Spark ist, wie es arbeitet und welche Bestandteile und Konfigurationen für diese Arbeit wichtig sind. Darüber hinaus werden die Dateiformate ORC und Parquet vorgestellt. Im zweiten Schritt werden dann Geschwindigkeitstests mit unterschiedlichen Konfigurationen und Dateiformaten durchgeführt sowie die Grenzsituationen untersucht. Die anfangs grobe Rasterung der Tests wird sich im Verlauf der Arbeit immer weiter verfeinern, bis am Ende, mithilfe von Entscheidungsmatrizen, eine Empfehlung für den hier behandelten Praxisfall vorliegt. Die Tests werden auf einem kleinen Test-Cluster durchgeführt, auf dem das Hadoop Distributed File System und YARN installiert sind.

Abstract

In this paper, the optimal configuration settings, the most appropriate file format, and the limits of Apache Spark are examined using a practical scenario. The first step is to explain what Spark is, how it works, and what components and configurations are required. In addition, the file formats ORC and Parquet are presented. In the second step, speed tests with different configurations and file formats are executed and limits are determined. The initially rough selection of the tests is refined as the analysis progresses, until in the end, a recommendation for the practical case is made with the help of decision matrices. The tests are performed on a small Test-Cluster on which the Hadoop Distributed File System and YARN are installed.

Abbildungsverzeichnis

Tabellenverzeichnis

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Aufgrund der immer größer werdenden Datenbestände der Welt, ist der Bedarf an immer schnelleren Analysewerkzeugen gestiegen. Eines dieser Analysewerkzeuge ist Apache Spark, welches seit 2013 unter dem Dach der Apache Software Foundation weiterentwickelt wird. Apache Spark ist für Cluster-Computing entwickelt worden. Seine Stärke liegt entsprechend in der schnellen horizontal skalierbaren Analyse sehr großer Datenmengen in verteilten Systemen. Darüber hinaus ist mit Apache Spark die Echtzeitanalyse möglich. Seit einiger Zeit befindet sich Apache Spark in Kombination mit dem Hadoop-Distributed-File-System (HDFS) im Testbetrieb bei einer mittelständischen Firma. Hintergrund ist die geplante Überführung von Messdaten in ein HDFS, wofür ein schnelles Analysetool benötigt wird. In naher Zukunft soll diese Technologie bei Kundenprojekten zum Einsatz kommen. Dazu ist es wichtig, die Einflussfaktoren zu kennen und diese beim Aufbau der entsprechenden Systeme zu berücksichtigen.

1.2. Ziele

Ziel dieser Arbeit ist es, die Einstellungsmöglichkeiten von Apache Spark für den Praxiseinsatz zu optimieren und ein geeignetes Speicherformat zu finden. Diese Arbeit beschäftigt sich daher mit der Identifikation, Untersuchung und Bewertung von begrenzenden Faktoren, wichtigen Konfigurationsparametern und daraus resultierenden Optimierungsmöglichkeiten, die den Praxiseinsatz von Apache Spark entscheidend beeinflussen. Dabei werden geeignete Speicherformate mit unterschiedlichen Einstellungen und Umgebungskonfigurationen untersucht und das System mittels Lasttests an die physikalischen Systemgrenzen gebracht.

1.3. Aufbau der Arbeit

Der Aufbau der Arbeit gliedert sich in vier Kapitel. Nach einer kurzen Einleitung wird in Kapitel „2 Theoretische Grundlagen von Apache Spark“ eine Einführung in Spark gegeben, an die sich die Erläuterungen der verwendeten Dateiformate anschließen. In Kapitel „3 Praktische Durchführung“ werden zuerst die Testtabellen und der Aufbau der verschiedenen Tests beschrieben. Im zweiten Teil findet dann die Analyse der Ergebnisse statt. Dort befinden sich auch die weiterführenden und vertiefenden Tests mit den dazugehörenden Erklärungen.
Im 4. Kapitel folgen noch Fazit und Ausblick. Hier wird betrachtet, ob die vorgenommenen Ziele erreicht wurden, wie die Empfehlung für diesen Praxisfall aussieht und welche Themen sich anschließen könnten.

1.4. Abgrenzung

Diese Arbeit beschäftigt sich mit der Geschwindigkeitsmessung, den Konfigurationseinstellungen und dem Verhalten in Grenzsituationen von Apache Spark in Kombination mit dem HDFS und YARN. Im Laufe der Arbeit soll die Konfiguration von Spark optimiert und nach dem aktuell schnellsten Dateiformat gesucht werden. Es gibt bereits einige Untersuchungen von Apache Spark. Diese beschäftigen sich jedoch mit dem Vergleich von Apache Spark und anderen Analyse-Tools, bzw. den Spark-Bibliotheken „Spark Streaming“ und „Spark Machine Learning“. Darüber hinaus gibt es Arbeiten, die Erweiterungen für Spark entwickeln.

2. Theoretische Grundlagen von Apache Spark

Dieses Kapitel befasst sich damit, was Spark ist, wie es arbeitet und wo bei der Optimierung angesetzt werden kann. In diesem Zusammenhang wird auch geklärt, wie das Testsystem aufgebaut ist.

2.1. Definition grundlegender Begrifflichkeiten

Bevor mit der Einführung begonnen wird, werden einige grundlegende Begriffe erklärt, die wichtig sind, um die Arbeit zu verstehen.

Cluster:
Ein Cluster oder auch Rechnerverbund bezeichnet eine Anzahl fest verbundener, vernetzter Rechner.

Verteiltes Speichersystem:
Ein verteiltes Speichersystem ist ein spezielles Dateisystem, bei dem der Zugriff auf die Daten über ein Rechnernetz erfolgt und die Datenspeicherung auf mehrere vernetzte Rechner verteilt ist, das für den Nutzer wie ein einziges System aussehen.

Horizontale Skalierung:
Die horizontale Skalierung bezeichnet eine Steigerung der Leistung eines Systems durch Hinzufügen zusätzlicher Rechner.

Knoten:
Ein Knoten oder auch Node ist ein Computer in einem Cluster.

2.2. Einführung in Spark

Apache Spark ist eines der Top-Level-Projekte der Apache Software Foundation. Das Programm kann die Anfragen parallel bearbeiten und ist nicht darauf angewiesen, dass die Daten eine einheitliche Struktur und Größe haben. Aus diesem Grund ist es für die Analyse von großen Datenmengen interessant. Apache Spark ist von Anfang an darauf ausgelegt worden, die benötigten Daten im Arbeitsspeicher zu halten und dort zu bearbeiten. Das zieht zwar einen erhöhten Arbeitsspeicherbedarf nach sich, sorgt aber auch für einen enormen Geschwindigkeitsvorteil gegenüber der Ablage im normalen Speicher.

Um über Spark eine Anfrage zu starten, muss zuerst ein Driver generiert werden. Dieses kann, in Kombination mit einem Speichersystem, auf zwei Arten geschehen. Bei der ersten Variante erfolgt die Generierung im Client-Mode. In diesem Fall startet der Driver außerhalb des Speichersystems.

Bei der zweiten Variante wird der Driver im Cluster-Mode generiert, wo der Driver direkt im Speichersystem ausgeführt wird und sich mit diesem die Ressourcen teilt. Für den Driver gibt es Anwendungs-Programmier-Schnittstellen (APIs) in Scala, Java, Python und R. Wichtig ist, zu beachten, dass Driver-Programme nicht miteinander kommunizieren können und es entsprechend keine Möglichkeit gibt, Daten auszutauschen.

Neben der genannten Möglichkeit, Spark im Cluster- oder Client-Mode mit einem Speichersystem zu kombinieren, gibt es alternativ die Option, Spark im Local-Mode laufen zu lassen. In diesem Fall wird alles in einer einzigen Java-Virtual-Machine (JVM) auf dem lokalen Rechner gestartet und ausgeführt. Die Parallelisierung wird durch verschiedene Threads simuliert.

Im folgenden Kapitel werden zuerst die Vor- und Nachteile des Local-Mode erläutert. Anschließend folgen die Vor- und Nachteile beispielhafter Speichersysteme, mit denen Spark anhand des Client- oder Cluster-Mode kommunizieren kann. Im Anschluss wird geklärt, wie das Testsystem aufgebaut ist.

2.2.1. Apache Spark auf unterschiedlichen Speichersystemen

Local-Mode

Beim Betrieb im Local-Mode auf einem lokalen Dateisystem, stehen Spark ausschließlich die Ressourcen des lokalen Rechners zur Verfügung. Aus diesem Grund ist diese Variante gut geeignet, um das System zu testen und kennenzulernen, jedoch weniger für den realen Betrieb. Der Vorteil ist, dass das ganze System in sich geschlossen und nicht von einem Netzwerk abhängig ist.

HDFS

Das Hadoop-Distributed-File-System [HDFS] ist ein verteiltes Speichersystem. Die Grundidee wurde von Google entwickelt und von der Apache Software Foundation aufgegriffen. Seit 2008 ist es dort eines der Top-Level-Projekte. Als wichtigste Eckpfeiler in der Entwicklung wurden vier Punkte festgelegt:

1. Der Betrieb auf handelsüblicher Standard-Hardware
2. Einfache horizontale Skalierbarkeit
3. Ausfallsicherheit des Gesamtsystems, auch bei Ausfall einzelner Knoten
4. Speicherung und Verarbeitung großer Datenmengen

Es gibt einen Hauptknoten (Name-Node), der die Verwaltung aller Metadaten übernimmt. Neben diesem alleswissenden Knoten gibt es noch eine beliebige Anzahl an Datenknoten (Data-Nodes), die den ihnen zugewiesenen Speicher verwalten. (Freiknecht, 2014) Der Vorteil des HDFS liegt in der guten Skalierbarkeit und der hohen Ausfallsicherheit. Nachteil ist, dass auf jedem Knoten alle Programme installiert werden müssen. Damit Spark mit dem HDFS kommunizieren kann, wird ein Cluster-Manager benötigt. Aktuell werden drei Cluster-Manager unterstützt:

1. Standalone-Cluster-Manager,
2. Apache Mesos,
3. Hadoop Yarn,

NAS

Network-Attached-Storage [NAS] bedeutet so viel wie netzwerkgebundener Speicher. NAS-Systeme können mit mehreren Festplatten ausgestattet werden, die meist in einem Redundant-Array-of-Independent-Disks [RAID]-Verbund laufen. In einem RAID-Verbund besteht die Möglichkeit mit mehreren Festplatten parallel zu agieren. Für mehr Ausfallsicherheit kann auf

mehrere Platten die gleiche Information geschrieben werden. Alternativ ist es möglich, die Informationen für eine höhere Geschwindigkeit auf mehrere Platten aufzuteilen.
Apache Spark benötigt für die Bearbeitung der Anfragen hauptsächlich CPU und Arbeitsspeicher. Beides ist auf einem NAS-System nur eingeschränkt vorhanden. Die Konsequenz ist, dass Spark auf einem externen Gerät laufen und sich die Daten vom NAS-System über ein Netzwerk holen muss. Insbesondere, wenn das Netzwerk keine hohe Bandbreite aufweist, muss davon ausgegangen werden, dass es Einbußen in der Geschwindigkeit gibt. Da Spark auf einem externen Gerät läuft, ist es nur begrenzt möglich, die Ressourcen für die Anfragenbearbeitung zu erweitern.

SAN

Storage-Area-Network [SAN] kann mit Speichernetzwerk übersetzt werden. Das SAN spannt sich über mehrere Server und kann auch über weite Distanzen gehen. Normalerweise wird das Fiber-Channel-Interface als Schnittstelle in einem solchen Speichernetzwerk verwendet, um hohe Übertragungsraten zu gewährleisten.
Der Vorteil eines SAN liegt in der guten horizontalen Skalierbarkeit, jedoch sind die Kosten, die für das spezielle Netzwerk anfallen, verhältnismäßig hoch. Es gibt neben der Möglichkeit des Fiber-Channel-Interfaces noch die günstigere Alternative des internet-Small-Computer-System-Interfaces [iSCSI]. Dieses Protokoll nutzt die normalen IP-Netzwerke und ist langsamer als das Fiber-Channel-Interface, jedoch schneller als gewöhnliche Netzwerkprotokolle.

2.2.2. Das Testsystem

Für den Test-Cluster ist das verteilte Speichersystem HDFS gewählt worden, wobei es sich um Open-Source-Software handelt, die es ermöglicht, Standard-Hardware zu verwenden. Darüber hinaus weist HDFS, wie oben beschrieben, eine hohe Ausfallsicherheit auf. Als Cluster-Manager wird das Programm Yet Another Resource Negotiator [YARN] verwendet. Es ist speziell dafür entwickelt worden, dass auch andere Programme wie z.B. Spark das HDFS nutzen können. YARN verwaltet alle im Cluster vorhanden Ressourcen (CPU, Arbeitsspeicher, Speicher) und stellt diese den Anwendungen zur Verfügung. Bei der Vergabe der Ressourcen werden auch Kriterien wie die Reihenfolge der Anfrage und unterschiedliche Kapazitäten auf den einzelnen Knoten, mit einbezogen.
Um die Testergebnisse nicht mit eventuell belegten Ressourcen zu verfälschen, läuft der Driver im Client-Mode. Da eventuelle Netzwerkengpässe das Ergebnis beeinflussen könnten, wird der Client auf einem der Cluster-Rechner jedoch außerhalb des Speichersystems gestartet.
Im Folgenden wird das Testsystem genauer betrachtet. Dabei wird darauf hingewiesen, worauf beim Aufbau geachtet werden sollte. Da die folgenden Informationen und Konfigurationen zwar

direkten Einfluss auf die Messwerte haben, aber die Verhältnisse der Messwerte untereinander nicht verfälschen, ist die Dimensionierung des Testsystems für diese Arbeit als konstant anzusehen.

Insgesamt wird der Test-Cluster aus zwei Knoten bestehen. Grundsätzlich gibt es zwar mehrere Faktoren, die beeinflussen wie groß ein Cluster ausfallen sollte, aber keine feste Formel für die Berechnung.

Der benötigte Speicher hängt hauptsächlich von der Gesamtmenge der zu speichernden Daten ab. Dabei muss beachtet werden, dass die Daten noch durch Replikation, also mehrmaliges Abspeichern, vervielfacht werden. Die genaue Anzahl der Replikationen hängt von der Konfigurationseinstellung des HDFS ab. Hinzu kommt, dass im Falle eines zu gering dimensionierten Arbeitsspeichers während der Anfragebearbeitung, ein Teil der Daten aus dem Arbeitsspeicher auf die Platten ausgelagert wird. Dieses würde nochmal zusätzlichen Speicherplatz voraussetzen. In dem verwendeten Test-Cluster sind pro Knoten drei Terrabyte Speicher verbaut.

Arbeitsspeicher und Cores sind hauptsächlich für die Aufgabenbearbeitung verantwortlich. Entsprechend sollte überlegt werden, wie viele Nutzer bzw. Driver auf das System zugreifen werden und wie lange das Bearbeiten der Anfragen dauern darf. Darüber hinaus ist es auch wichtig zu betrachten, wie komplex die Anfragen sein werden. Einfache Anfragen können deutlich schneller und ressourcenschonender abgearbeitet werden als komplexe Anfragen. In dem Test-Cluster sind pro Knoten 8 Cores und 32 Gigabyte Arbeitsspeicher verbaut. Davon stehen Spark 6 Cores und 24 Gigabyte Arbeitsspeicher pro Knoten zur Verfügung. Die restlichen Ressourcen werden von den installierten Programmen und dem Client benötigt.

Falls ein Cluster, wie in diesem Fall, mit wenigen Knoten aufgebaut wird, sollte die Konfiguration des HDFS daran angepasst werden. Der Hintergrund ist, dass das Programm bei einem Fehler versucht, die fehlgeschlagene Berechnung auf einem alternativen Knoten zu wiederholen. Bei nur wenigen Knoten im Cluster gibt es keinen alternativen Knoten. Es wird eine Fehlermeldung ausgegeben.

Durch das Anpassen der Konfiguration wird die Suche nach einem neuen Knoten verhindert. (Foley, 2016) (Apache Software Foundation, hdfs.default.xml, 2014)

2.2.3. Spark auf einem verteilten Speichersystem

Wie im vorherigen Kapitel beschrieben, wird Spark auf dem verteilten Speichersystem HDFS in Kombination mit YARN laufen. Im Folgenden wird erklärt, wie Spark in dieser Kombination genau arbeitet.

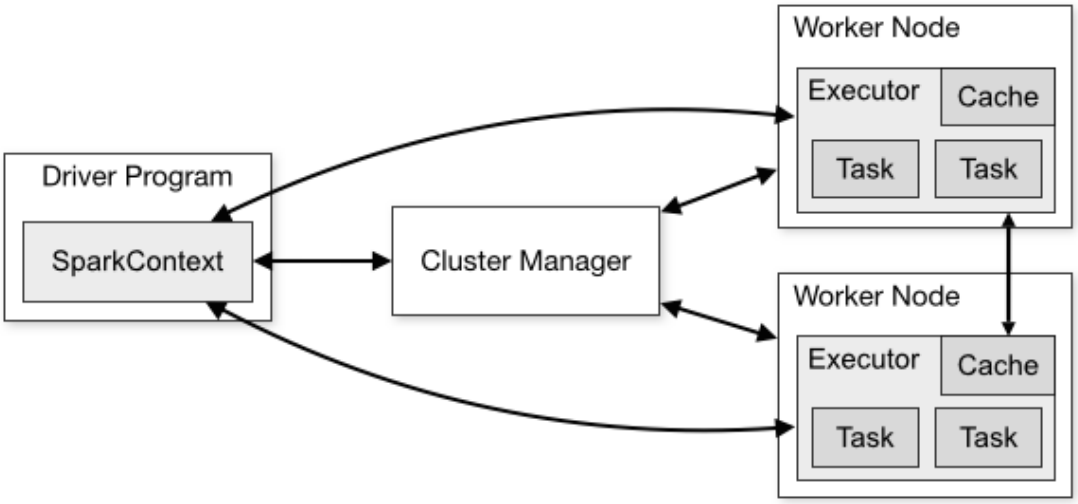

Abbildung 1: Spark in Kombination mit einem Cluster-Manager auf einem verteilten Speichersystem Quelle: (Apache Software Foundation, Cluster Mode Overview, 2016)

In Abbildung 1 sind der Aufbau und die Kommunikationswege von Spark in Kombination mit einem Cluster-Manager und einem verteilten Speichersystem dargestellt. Der Driver beinhaltet einen sogenannten SparkContext. Dieser verbindet sich mit dem Cluster-Manager. Der Spark-Context gibt dem Cluster-Manager vor, wie viele Executoren mit welchen Ressourcen er haben möchte. Ein Executor ist ein Prozess, der die Daten in den Arbeitsspeicher holt und bearbeitet.
Der Cluster-Manager erstellt nach der Anfrage des SparkContext, Executoren auf den Knoten und weist diese dem SparkContext zu.
Auf jedem Knoten können mehrere Executoren laufen, jedoch kann sich ein Executor nur auf maximal einem Knoten befinden. Die Gesamtheit von Driver-Programm und Executoren wird als Applikation bezeichnet. Der SparkContext sendet den Executoren Tasks zu. Ein Task ist die kleinste Einheit, in die ein Job zerlegt werden kann. Die Tasks werden von den Executoren parallel ausgeführt, wobei die Regel gilt, dass zeitgleich nur ein Task pro Core berechnet wird. Werden z.B. einem Executor 5 Cores zugewiesen, kann dieser 5 Tasks parallel ausführen.
Executoren existieren genauso lange wie der SparkContext existiert und führen auch nur für diesen SparkContext die Tasks aus. Wenn der Driver mit dem SparkContext geschlossen wird, werden die blockierten Ressourcen der Executoren wieder freigegeben. Eine Ausnahme von dieser Regel bildet das dynamische Zuweisen von Executoren. Hier werden die Executoren vom

Cluster-Manager wieder freigegeben, wenn sie nicht mehr benötigt werden, unabhängig davon, ob der Driver weiterhin existiert.
Da der oder die Executoren mit dem Driver kommunizieren müssen und auch größere Datenmengen, wie z.B. die Antworten, ausgetauscht werden, sollte sich der Driver im selben Netz befinden wie der Cluster.
(Apache Software Foundation, Cluster Mode Overview, 2016)
(Karau & Warren, How Spark Works, 2017)

2.2.4. RDD und DataFrame

Um das im vorherigen Kapitel genannte parallele Ausführen der Tasks zu ermöglichen, nutzt Spark Partitionen. Eine Partition ist ein zusammenhängender Bereich im Speicher. Partitionen sind für die parallele Bearbeitung geeignet, da sie unabhängig voneinander sind. Meist sind mehrere Partitionen als Resilient-Distributed-Datasets [RDD] zusammengefasst und über diese vom Driver ansprechbar. Es handelt sich hierbei eine unveränderliche, verteilte Ansammlung von Daten und kann auf drei Arten erstellt werden:

1. Als Ergebnis einer RDD-Transformation, da RDDs per Definition unveränderlich sind
2. Von einem SparkContext der z.B. eine Datei eingelesen hat
3. Durch Umwandeln eines DataFrame oder Dataset in ein RDD

(Karau & Warren, Immutability and the RDD Interface, 2017)

Das DataFrame aus dem genannten Punkt 3 ist ein besonderes RDD und wurde speziell für strukturierte Daten entwickelt. Es kommt aus dem Spark-SQL-Umfeld und repräsentiert eine Zeile mit einer oder mehreren Spalten, welche festgelegte Datentypen haben. Darüber hinaus gibt es das Dataset. Ein Dataset ist ein DataFrame mit vordefinierter Struktur. So muss beim Erstellen des Datasets angegeben werden, wie viele Spalten, mit welchem Datentyp und in welcher Reihenfolge enthalten sein sollen. Der Vorteil eines Datasets gegenüber dem DataFrame liegt darin, dass auf die einzelnen Zeilen, ähnlich wie bei einem Array, zugegriffen werden kann.
(Apache Software Foundation, Spark SQL, DataFrames and Datasets Guide, 2016)

2.2.5. Der Spark-Job

Nachdem die Arbeitsweise von Spark erklärt wurde, wird in diesem Abschnitt erläutert, wie eine Anfrage vorbereitet werden muss, bevor die Executoren diese bearbeiten können. Es gibt zwar eine ungefähre Reihenfolge, die beim Anfragestellen eingehalten werden muss, aber Spark arbeitet alle Anfragen auf, um sie weiter zu optimieren.
Wenn der SparkContext eine Anfrage startet, dann beinhaltet diese meist unterschiedliche Befehle. Diese Befehle lassen sich in drei Kategorien einteilen:

1. Action
2. Wide-Transformations
3. Narrow-Transformations

Diese werden nun im Folgenden genauer beschrieben.

Action

In Action befinden sich die Befehle, die Daten zurück an den SparkContext senden, um sie auf der Konsole auszugeben oder in eine Datei zu schreiben. Sie müssen bei der Anfragestellung an letzter Stelle stehen.

Wide-Transformations

In Wide-Transformations befinden sich alle Befehle, für die mehrere Partitionen miteinander kommunizieren müssen. Das könnte z.B. der Fall sein, wenn Werte von unterschiedlichen Partitionen miteinander verglichen werden müssen.

Narrow-Transformations

Als letztes gibt es noch die Narrow-Transformations. Hier sind alle Befehle zu finden, die unabhängig von anderen Partitionen auf einer einzelnen Partition ausgeführt werden können. Ein Beispiel hierfür wäre ein Filter. Der Filter geht die Informationen auf einer Partition durch und entscheidet welche davon er auswählen muss. Er benötigt dafür keine Information darüber, ob es noch weitere Partitionen gibt.

Wenn eine Anfrage gestartet wird, dann wird diese zuerst vom Directed-Acyclic-Graph-Scheduler [DAG-Scheduler] bearbeitet, was frei mit “in eine Richtung weisender nicht zyklischer Graph-Planer“ übersetzt werden kann. Im Anschluss an den DAG-Scheduler verfeinert der Task-Scheduler den entstandenen Graphen weiter. Der DAG-Scheduler baut aus den Informationen der Anfrage einen Graphen, der festlegt, in welcher Reihenfolge der Job ausgeführt werden soll. Dabei konzentriert sich der DAG-Scheduler auf die Wide-Transformations und entscheidet

welche Narrow-Transformations dafür benötigt werden. Die Informationen über eine einzelne Wide-Transformation und die dafür benötigten Narrow-Transformations werden jetzt als Stage abgespeichert. Den Graphen mit den so generierten Stages gibt der DAG-Scheduler an den Task-Scheduler weiter.
Der Task-Scheduler stellt aus dem generierten Graphen die Information zusammen, welche Partitionen miteinander kommunizieren müssen. Er erweitert den Graphen um die Abhängigkeiten zwischen den Partitionen und welcher Task auf welcher Partition laufen muss. Das Ergebnis könnte dann schematisch so aussehen wie in Abbildung 2.

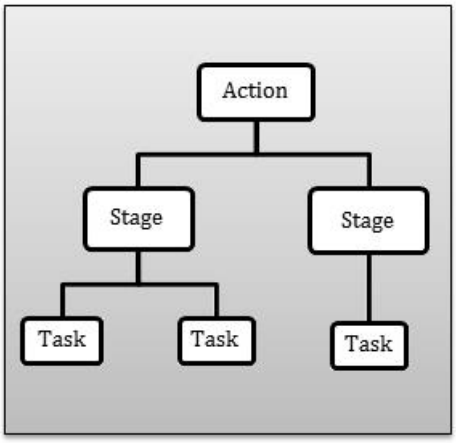

Abbildung 2: Schematische Darstellung eines Graphen mit Stages und den dazu benötigten Tasks
Quelle: (Karau & Warren, The Anatomy of a Spark Job, 2017)

Durch die Vorarbeit des DAG- und Task-Schedulers wird sichergestellt, dass Operationen, die parallel und somit schnell ausgeführt werden können, zuerst abgearbeitet werden. Erst wenn alles, was parallel abgearbeitet werden kann, abgearbeitet wurde, beginnen die aufwendigen Wide-Transformations. (Karau & Warren, The Anatomy of a Spark Job, 2017)
(Karau & Warren, Wide Versus Narow Dependencies, 2017)

In Abbildung 3 wird veranschaulicht, welche Auswirkungen der DAG- und Task-Scheduler haben. Falls sie nicht existieren würden, könnte die Reihenfolge beim Bearbeiten wie auf der linken Seite aussehen. Dort werden die beiden Partitionen erst mit „join“ verbunden und anschließend gefiltert. Dadurch müssen, im Vergleich zum rechten Graphen, deutlich mehr Zeilen verknüpft werden. Dieses sorgt für eine längere Ausführungszeit.

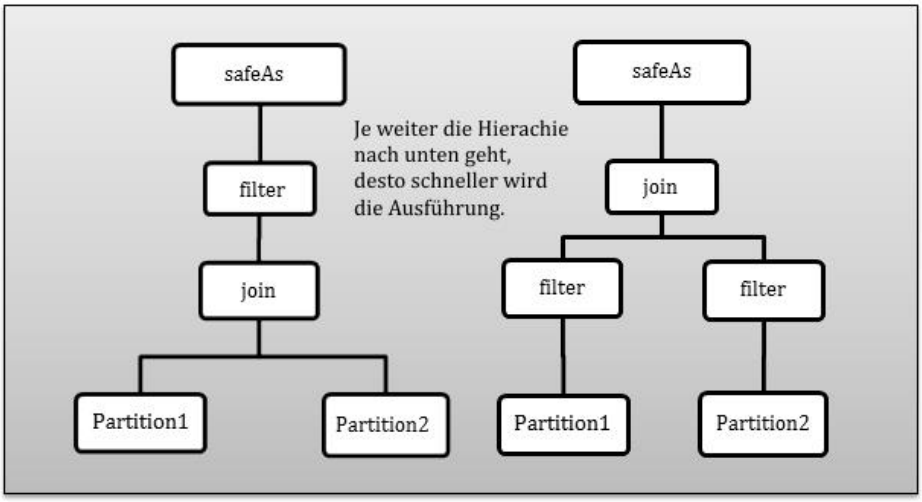

Abbildung 3: Reihenfolge ohne (links) und mit (rechts) Vorsortierung Quelle: (Ho, 2015)

Für einen besseren Überblick, welcher Befehl in welche der drei Kategorie gehört, wird in Tabelle 1 gezeigt, wie die in dieser Arbeit verwendeten Befehle einzuordnen sind.

Action	Wide-Transformations	Narrow-Transformations
count	join	map
show	groupBy	filter
aggregate	reduceByKey	
safeAs	sort	

Tabelle 1: Wichtige Action, Wide- und Narrow-Transformations Quelle: (Data Flair Support, 2016)

2.2.6. Ansätze zur Optimierung

Nachdem ein Überblick gegeben wurde, wie Spark arbeitet, soll nun auf die Möglichkeiten der Optimierung eingegangen werden. Im Folgenden werden unterschiedliche Ansätze für die Optimierung vorgestellt.

Executoren

Executoren sind verantwortlich für die Ausführung der Anfragen. Wie in Kapitel „2.2.3. Spark auf einem verteilten Speichersystem“ beschrieben, werden den Executoren Cores und Arbeitsspeicher zugewiesen. Bei der Frage, wie viel Arbeitsspeicher und Cores am besten zugewiesen werden sollten, gehen die Meinungen auseinander. In (Karau & Warren, A Few Large Excecutors or Many Small Executors?, 2017) wird beispielsweise empfohlen, dass ein Executor mindestens 4 Gigabyte groß sein sollte. In (Apache Software Foundation, Spark Configuration - Application Properties, 2016) wird beschrieben, dass die Default-Einstellung 1 Gigabyte beträgt und das z.B. Werte wie 2 Gigabyte und 8 Gigabyte verwendet werden können. Aus diesem Grund werden in der praktischen Durchführung unterschiedliche Kombinationen von Core und Arbeitsspeicher durchgetestet.

Serialisierung

Die Serialisierung ist ein wichtiger Punkt, der hier erläutert werden muss. Die RDDs liegen serialisiert im Arbeitsspeicher und müssen vor dem Lesen immer deserialisiert werden. Standardmäßig werden die Daten mit der Java-Serialisierung umgewandelt. Spark bietet alternativ noch die Möglichkeit, die Daten mit Kryo zu serialisieren. Welche von den beiden Varianten in diesem Testumfeld schneller ist, soll ebenfalls getestet werden.

Partitionierung

Ein dritter wichtiger Punkt für die Optimierung ist die Partitionierung bei der Speicherung. Wenn von Anfang an klar ist, dass beim Analysieren der Daten immer nach einer bestimmten Spalte gesucht wird, dann können die Daten beim Speichern danach sortieren werden. Grundsätzlich können alle Spalten, in denen der Datentyp ein String oder eine Zahl ist, als Schlüssel genutzt werden. Beim Speichern werden alle Zeilen, die in der vorgegebenen Spalte den gleichen Wert haben, zusammen abgespeichert. Wenn später nach einem bestimmten Wert aus dieser Spalte gesucht wird, befinden sich alle Zeilen mit diesem Spalteninhalt gemeinsam auf einer Partition. Es sollte darauf geachtet werden, dass in der ausgewählten Spalte nicht zu viele unterschiedliche Werte vorkommen, da für jeden Wert ein extra Verzeichnis angelegt wird.
(Karau & Warren, Partitions (Discovery and Writing), 2017)

GroupBy oder ReduceBy

Spark bietet neben GroupBy noch die Möglichkeit, Anfragen mit ReduceBy zu gruppieren. Da GroupBy grundsätzlich eine ressourcen- und zeitintensive Anfrage ist, soll untersucht werden, was genau ReduceBy ist, wo die Unterschiede liegen und welcher Befehl für den hier untersuchten Testfall das schnellere Ergebnis liefert.

2.3. Speicherformate

Die Rohdaten, die als Grundlage für diese Arbeit dienen, liegen im „csv-Format“ vor und können so unkompliziert von Spark eingelesen werden. Parquet und Optimized Row Columnar [ORC] sind die einzigen von Spark unterstützten Dateiformate, die von Anfang an gezielt für große Datenbanken entwickelt wurden. Davon ausgehend, dass diese zielgerichtete Entwicklung den entscheidenden Vorteil bringt, wird die Untersuchung in dieser Arbeit auf diese beiden Datentypen beschränkt. Je nachdem, was für Informationen verarbeitet werden, ist entweder ORC oder Parquet das schnellere Format. Aus diesem Grund werden beide Formate untersucht.

2.3.1. ORC

In (Apache Software Foundation, Apache ORC Documentation, 2017) ist ORC beschrieben. ORC ist ein spaltenorientiertes Speicherformat, das für verteilte Speichersysteme wie z.B. das HDFS entwickelt wurde. Es ist optimiert für umfangreiche Streaming-Analysen, beinhaltet aber auch Möglichkeiten, gezielt nach bestimmten Informationen zu suchen.

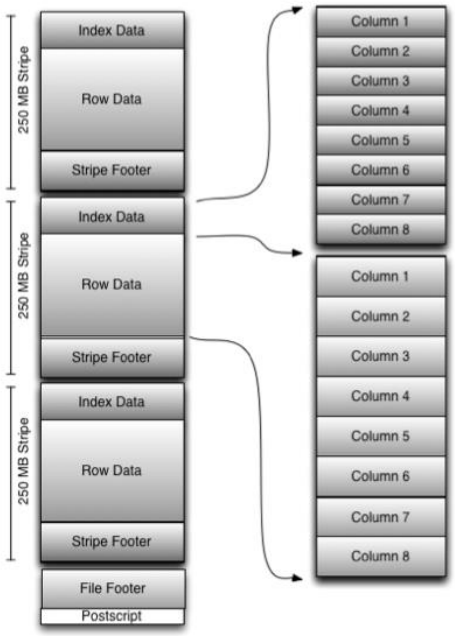

Abbildung 4: Struktur einer ORC-Datei
Quelle: (Apache Software Foundation, Apache ORC Documentation, 2017)

Wie in Abbildung 4 zu sehen ist, wird die Datei beim Schreiben in Stripes aufgeteilt. Diese Stripes oder auch Abschnitte innerhalb einer ORC-Datei, sind voneinander unabhängig und ermöglichen so das parallele Arbeiten in verteilten Systemen. Innerhalb eines Abschnitts werden immer 10.000 Zeilen zu einem Paket zusammengefasst. Während die Daten gelesen werden, kann

anhand der zusätzlich abgespeicherten Statistiken schnell ermittelt werden, ob ein Abschnitt gelesen werden muss und welche Zeilenpakete davon interessant sind.
Grundsätzlich gibt es drei unterschiedliche Statistiken:

1. Auf Dateiebene werden statistische Informationen der Werte einer Spalte über die komplette Datei hinweg gespeichert.
2. Auf Abschnitt-Ebene werden statistische Informationen der Werte einer Spalte über den Abschnitt hinweg gespeichert.
3. Auf der untersten Ebene werden statistische Informationen der Werte einer Spalte über ein 10.000-Zeilen-Paket hinweg gespeichert.

Da die Spalten die Grundlage für die statistischen Informationen liefern, soll im Folgenden genauer auf die Spaltenstruktur eingegangen werden.
Die unterschiedlichen Spalten in einer ORC-Datei sind in einer Baumstruktur angeordnet und werden von oben nach unten und von links nach rechts gelesen. Wie am Beispiel in Abbildung 5 auf der linken Seite zu erkennen ist, fängt alles immer mit einem Struct-Element an Position 0 an. Allgemein können sich in einem Struct-Element beliebig viele andere Elemente befinden. In dem Struct-Element in Abbildung 5 befinden sich eine Integer-, eine Map- und eine Time-Spalte. In der Map-Spalte befinden sich wiederum zwei weitere Spalten, usw. Auf diese Weise können beliebig verschachtelte Schemata aufgebaut werden. Die Darstellung in Abbildung 5 auf der rechten Seite soll veranschaulichen, wie dieses Beispiel-Schema als Tabelle aussehen könnte.

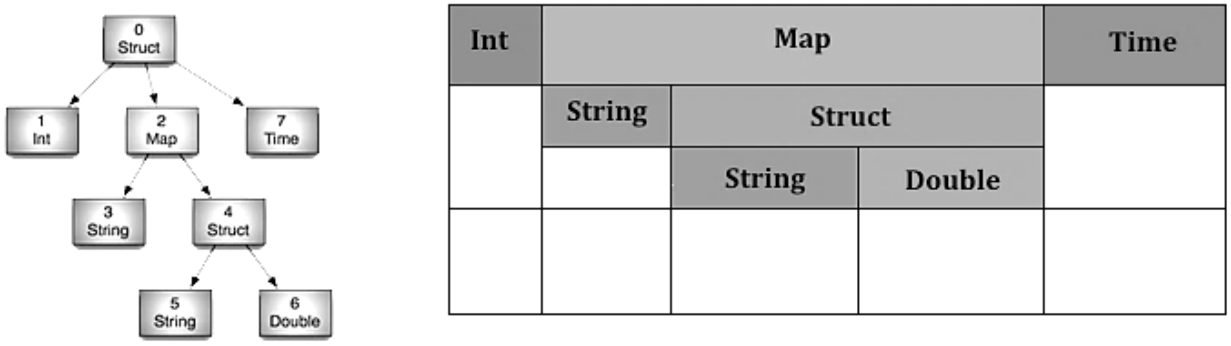

Abbildung 5: Struktur der Spalten Quelle: (Apache Software Foundation, Apache ORC Documentation, 2017)

Darüber hinaus gibt es bei ORC die Möglichkeit, die Größe der Datei durch Komprimierung zu verringern. Das ORC-Dateiformat bietet die Optionen, die Informationen ohne Komprimierung oder mit den Komprimierungsformaten Zlib oder Snappy zu komprimieren. Da Snappy, wie in Tabelle 2 auf Seite 23 dargestellt, für schnelle Komprimierung und Dekomprimierung entwickelt wurde, wird das ORC-Format mit Snappy und ohne Komprimierung untersucht. Die Entscheidung beruht darauf, dass in diesem Testumfeld die Geschwindigkeit der Anfragen wichtiger ist, als der Speicherbedarf im abgespeicherten Zustand.

2.3.2. Parquet

Apache Parquet ist genau wie Apache ORC ein spaltenorientiertes Speicherformat, das für verteilte Speicherlösungen, z.B. mit dem HDFS, entwickelt wurde. In Parquet gibt es im Verhältnis zu ORC jedoch nur wenige Datentypen:

- BOOLEAN: 1 bit
- Integer in den Varianten: INT32, INT64 und INT96
- FLOAT: 32-bit
- DOUBLE: 64-bit
- BYTE_ARRAY: beliebige Länge

Damit trotzdem auch z.B. DATE-, LIST- und MAP-Informationen gespeichert werden können, nutzt Parquet verschachteltes Kodieren (Nested-Encoding). Die komplexen Datentypen werden als primitive Datentypen dargestellt und die Struktur mithilfe von zwei Integer imitiert. Die beiden Integer werden „definition level" und „repetition level" genannt. Der Algorithmus, der dahinter steckt, heißt „record shredding and assembly algorithm" und wurde von Google entwickelt. Die Details des Algorithmus sind im Dremel-paper (Melnik, et al., 2010) beschrieben. Das Ergebnis des Algorithmus ist, neben der Darstellung von komplexen Datentypen, die Möglichkeit jede Spalte unabhängig von den anderen zu lesen. Am Beispiel von MAP bedeutet das, dass z.B. der Key gelesen werden kann, ohne dass der dazugehörende Wert bekannt sein muss.

Wie in Abbildung 6 zu sehen fängt eine Parquet-Datei immer mit einem Header an. In diesem steht die Nummer PAR1, welche die Datei als Parquet-Datei identifiziert. Nach dem Header kommen die Blöcke, die sogenannten Row-Groups. Den Schluss bildet der Footer mit den Informationen über die Metadaten. Als letzte Information im Footer steht wieder die PAR1 Nummer, um das Ende der Datei zu markieren. (White, 2015).

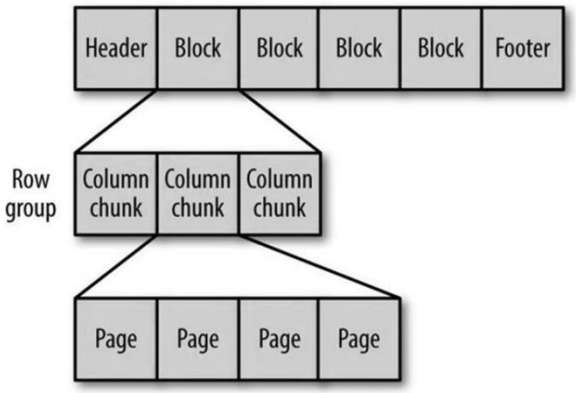

Abbildung 6: Interne Struktur einer Parquet-Datei Quelle: (Apache Software Foundation, Apache Parquet Documentation, 2014)

Eine Row-Group ist mit den Stripes des ORC-Formates vergleichbar und kann entsprechend parallel ausgeführt werden. Es wird empfohlen, dass eine Row-Group so groß sein sollte, wie die HDFS-Blocksize, bzw. entsprechende Einstellungen bei anderen Systemen. Dieses soll eine optimale Verarbeitung sicherstellen. In dem hier behandelten Testumfeld sind das 128 Megabyte. Eine Row-Group enthält für jede Spalte einen Spalten-Datenblock (Column-chunk). Jeder Spalten-Datenblock hat seine eigenen Metadaten. Wenn eine Anfrage an eine Parquet-Datei gestellt wird, dann werden zuerst die Metadaten im Footer gelesen. Anhand der Metadaten entscheidet der Reader, welche Row-Groups er sich

ansehen muss. Anschließend kann anhand der Metadaten der Spalten-Datenblöcke weiter selektiert werden.
Der Inhalt der Spalten-Datenblöcke besteht aus Pages. Die empfohlene Größe pro Page liegt bei 8 Kilobyte. Pages haben anstelle von Metadaten einen Header, welcher es dem Reader ermöglicht zu entscheiden, ob er einen Page lesen muss oder nicht.
Die abgespeicherten Informationen stehen hinter dem Header und können komprimiert sein. Wenn die verschachtelte Kodierung für die komplexen Datentypen benötigt wurde, dann ist auch diese hier zu finden. Eine Page ist eine untrennbare Einheit und unabhängig von anderen Pages, weshalb sie parallel bearbeitet werden kann.
Genau wie das ORC-Format unterstützt auch Parquet unterschiedliche Komprimierungsformate. Es werden die Formate Snappy, Gzip und LZO unterstützt. Alternativ kann auch auf die Komprimierung verzichtet werden.
(Apache Software Foundation, Apache Parquet Documentation, 2014)
(Karau & Warren, Parquet, 2017)

Tabelle 2 liefert nützliche Informationen bezüglich der Auswahl der Komprimierung.

	Compression-performance	Decompression-performance	Compression-ratio
Gzip	LOW	LOW	HIGH (~60%)
LZO	HIGH	HIGH	LESS (~50%)
Snappy	HIGHEST	HIGHEST	LOW (~40%)

Tabelle 2: Unterschiedliche Formate zur Komprimierung Quelle: (Yu & Guo, 2016)

Genau wie bei ORC wird auch Parquet mit Snappy und ohne Komprimierung getestet. Die Begründung ist auch hier der Fokus der Testumgebung auf möglichst schnelle Bearbeitungszeiten.

3. Praktische Durchführung

3.1. Die Basis-Tabellen

Als Testtabellen wurden zwei Tabellen generiert, welche die Originaltabellen imitieren. Da die Messdaten in der Praxis aus unterschiedlichen Quellen stammen, die Informationen aber trotzdem miteinander in Verbindung gebracht werden sollen, wurden zwei Tabellen generiert.

Tabelle TestCase1 **Größe: 200Mio. Zeilen entspricht ~ 17** Gigabyte

	Spaltenname	Datentyp	Wert	
1	utc	LongType	ab 01.01.1990 10.00 | + 1.x sec pro Zeile bis 13.04.2001 19.53	
2	depth	IntegerType	1 – 2001	
3	depth_sensorname	StringType	DEF, GHI, JK	
4	latitude	DoubleType	Start : -34.30	Zufällige Position zu der sich hinbewegt wird und die dann eine Weile umkreist wird
5	longitude	DoubleType	Start: 18.60	
6	position_sensorname	StringType	PQR, STU, VW	
7	speed	DoubleType	0 – 10	
8	speed_sensorname	StringType	ST, UVW, XY	
9	wind_speed_kmh	IntegerType	1 – 70	
10	wind_speed_knots	DoubleType	wind_speed_kmh * 0,539957	
11	wind_speed_sensorname	StringType	ABC, DE, FGH	

Tabelle 3: Testtabelle 1

Tabelle TestCase2 **Größe: 200 Mio. Zeilen entspricht ~ 12,5** Gigabyte

	Spaltenname	Datentyp	Wert	
1	utc	LongType	ab 01.01.1995 10.00 | + 1.x sec pro Zeile bis 19.05.2004 09.13	
2	salt_content	DoubleType	3.0 – 4.0 | +0.1 pro Zeile	
3	water_temp	IntegerType	5 – 25	
4	latitude	DoubleType	Start: -34.30	Zufällige Position zu der sich hinbewegt wird und die dann eine Weile umkreist wird
5	longitude	DoubleType	Start: 18.60	

Tabelle 4: Testtabelle 2

Während des Importierens werden den Tabellen berechnete Spalten mitgegeben. Bei beiden Tabellen werden die in Tabelle 5 dargestellten Werte hinzugefügt.

	Spaltenname	Datentyp	Wert Berechnung
1	timestamp	TimestampType	($"utc"/1000).cast("timestamp")
2	geo_id_555km	LongType	getGeoId(lit(5.0), $"latitude", $"longitude")
3	month	IntegerType	month($"timestamp")
4	point	Magellan Point	point($"longitude", $"latitude")

Tabelle 5: Nachträglich berechnete Spalten der Testtabellen

3.2. Testaufbau

Für die Tests sind eine Bash-Datei und mehrere Scala-Dateien programmiert worden. Die Bash-Datei ruft die unterschiedlichen Jobs mit den entsprechenden Scala-Dateien auf und automatisiert so die durchzuführenden Tests.

Um sicherzugehen, dass Spark die Anfragezeiten nicht durch Zwischenspeichern verfälscht, wird innerhalb eines Jobs der Puffer-Speicher nach jeder Anfrage geleert.

Bei den Executoren sollen unterschiedliche Core- und Arbeitsspeichergrößen durchgetestet werden. Da ein Executor nur auf maximal einem Knoten laufen kann und die Test-Knoten 6 Cores haben, werden 1, 3 und 6 Cores durchprobiert. Alle drei Möglichkeiten sind Teiler von 6, was eine gute Auslastung der vorhandenen Ressourcen bedeutet. Beim Arbeitsspeicher muss beachtet werden, dass z.B. 4 Gigabyte zugewiesener Arbeitsspeicher nicht gleich 4 Gigabyte benutzter Speicher bedeuten. Ein Executor hat einen Head, der 10% vom Speicher beträgt und mindestens 384 Megabyte groß ist. Dieser Head muss zum zugewiesenen Speicher dazugerechnet werden, wenn die tatsächliche Speicherbelegung errechnet werden soll. Wenn jetzt 3 Gigabyte Arbeitsspeicher zugewiesen werden, dann sind 10% etwa 300 Megabyte. Da die Mindestgröße 384 Megabyte beträgt, entspricht der Head mehr als 10% des zugewiesenen Speichers. Aus diesem Grund empfiehlt es sich bei der Speichergröße bei mindestens 4 Gigabyte anzufangen, da 10% von 4 Gigabyte ca. 400 Megabyte sind und damit knapp über der Mindestgrenze. Im Test werden Speicherzuweisungen von 1 Gigabyte, 4 Gigabyte, 10 Gigabyte und 14 Gigabyte getestet. Alle Executoren auf einem Knoten zusammen können aufgrund der vorhandenen Ressourcen, maximal 24 Gigabyte Speicher inklusive Head, unter sich aufteilen. Die Anzahl der Executoren wird sich an der Ressourcenzuweisung orientieren.

In der folgenden Tabelle 6 die Varianten, die getestet werden sollen.

Cores	Memory	Executoren Anzahl
1	1 Gigabyte	12
3	1 Gigabyte	4
3	4 Gigabyte	4
6	4 Gigabyte	2
3	10 Gigabyte	4
6	10 Gigabyte	2
6	14 Gigabyte	2
dynamische Ressourcenzuweisung		

Tabelle 6: Executor-Ressourcenzuweisung

Zusätzlich zu der manuellen Zuweisung bietet Spark die Möglichkeit, die Executoren dynamisch zu erstellen. Je nachdem, wie umfangreich die Anfragen sind, wird dann ein zusätzlicher Executor zugeteilt oder wieder entfernt. Laut Tabelle 6 ergeben sich so acht Alternativen, die untersucht werden sollen.

Darüber hinaus werden die Messwerte der Executoren genutzt, um zu entscheiden

a. welches Dateiformat (ORC oder Parquet)
b. welche Speichervariante (Snappy, partitioniert, ohne Snappy oder Partition, oder mit beiden Varianten)
c. welche Serialisierung (Java oder Kryo)

am schnellsten ist. Dafür werden die Durchschnittswerte aller Executoren genommen, um für die 16 Alternativen, die sich aus den drei genannten Punkten ableiten, die Werte zu ermitteln.

Damit alles miteinander verglichen werden kann, wurde eine Methode für die Zeitmessung geschrieben. Die Grundidee zu der Methode wird in vielen Foren diskutiert. Als Beispiel (stackoverflow, 2012) oder (Zakordonets, 2017)

```
def time[R](name:String)(block: => R): R = {
  val t0 = System.nanoTime()
  val result = block
  val t1 = System.nanoTime()
  val roundSec = "%.1f".format((t1 - t0)/1e9)
  val roundMin = "%.1f".format(((t1 - t0)/1e9)/60)
  writer.write(name+" "+ roundMin +" min oder "+ roundSec + " sec\r\n\r\n")
  writer.flush()
  result
}
```

Die Methode wird wie folgt aufgerufen:

time("*einfache Anfrage***")(** *import1.filter($"depth".isNotNull).show(false)* **)**

In der Variable „name“ vom Typ „String“ wird eine Beschreibung übergeben, um die Zeiten in der Textdatei später zuordnen zu können. Die Variable „block“ beinhaltet die Anfrage vom Datentyp „R“. Da „R“ variabel ist, können alle möglichen Anfragen gestellt und verarbeitet werden. Als Rückgabewert steht wieder „R“. So kann die Methode alle möglichen Ergebnisse der Anfrage zurückgeben. In der Variablen „t0“ wird die Startzeit gespeichert. In der zweiten Zeile wird die Anfrage gestartet und das Ergebnis in die Variable „result“ gespeichert. Dieses Speichern ist

wichtig, falls das Ergebnis, nach der Ausführung, in einer Scala-Variablen gespeichert werden muss. Anschließend wird wieder die Zeit gemessen und gespeichert. Das „/1e9“ sorgt dafür, dass die Nanosekunden in einfache Sekunden umgerechnet werden. Um längere Ausführungszeiten übersichtlicher zu gestalten, wird das Ergebnis durch 60 geteilt („/60“), um Minuten ausgeben zu können. Mit „%.1f.format“ wird das Ergebnis der Umrechnung auf eine Nachkommastelle gerundet. Am Ende werden die Informationen als String an den Writer übergeben und „result“ als Rückgabewert der Methode zurückgegeben.
Im Folgenden die Details zu den unterschiedlichen Tests.

3.2.1. GroupBy- oder ReduceBy-Test

Grundsätzlich ist GroupBy eine problematische Anfrage, da sie viele Ressourcen und Zeit benötigt. Um die Geschwindigkeit der Anfragen zu verbessern, hat Apache Spark neben GroupBy auch noch ReduceBy im Angebot. In diesem Test soll geklärt werden, wo die Unterschiede liegen und welches schneller ist. Da das Ergebnis benötigt wird, um die Analyseanfragen zu formulieren, ist dieser Test zuerst ausgeführt worden.
Die Unterschiede der beiden Anfragen sind in Abbildung 7 gut zu erkennen. ReduceBy fängt mit der Auswertung der Daten bereits auf den Partitionen an. Es werden anschließend nur die Ergebnisse von allen beteiligten Partitionen gesammelt und zusammengeführt. GroupBy dagegen sammelt erst alle Informationen und wertet danach aus. Das hat zur Folge, dass viel mehr Daten bewegt werden müssen und somit die Performance schlechter ist.

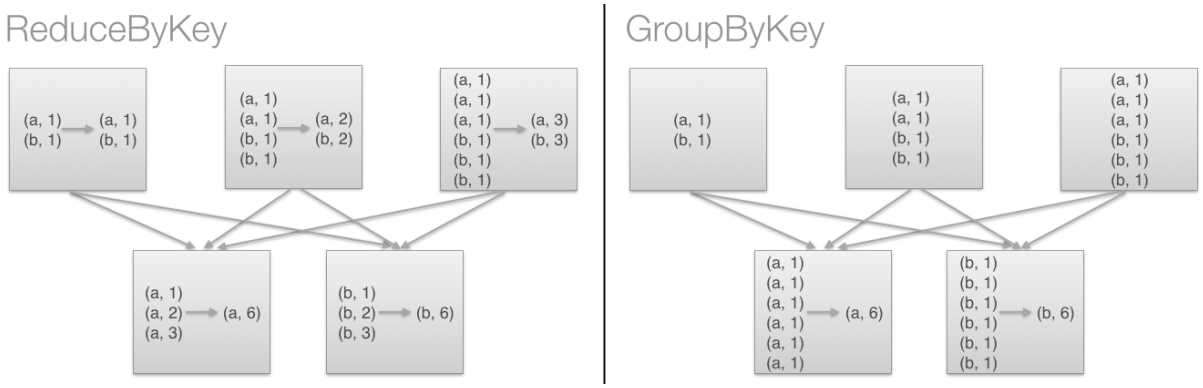

Abbildung 7: Unterschiede GroupBy und ReduceBy Quelle: (Databricks, 2014)

So zumindest war es in der Vergangenheit bzw. bis das DataFrame eingeführt wurde, welches in Kapitel „2.2.4. RDD und DataFrame“ erklärt wurde. Für das DataFrame gibt es nur noch das GroupBy und dieses ist so abgeändert worden, dass es ein ähnliches Verhalten zeigt, wie das beschriebene ReduceBy. Zusammengefasst stellt Spark also GroupBy und ReduceBy für Standard-RDDs zur Verfügung und ein abgeändertes GroupBy für DataFrames. Seit der Einführung

des DataFrame wird dieses beim Einlesen einer Datei standardmäßig verwendet, weshalb im folgenden Test zwischen dem neuen GroupBy und dem ReduceBy von einem DataFrame in der import1 Variable ausgegangen wird.

```
import1
.select("depth_sensorname", "depth")
.filter($"depth".isNotNull )
.rdd
.map(row => (row.getString(0), row.getInt(1)))
.reduceByKey((a, b) => Math.max(a, b))
.collect()

res: Array[(String,Int)]=Array((JK,2001),(DEF,2001),(GHI,2001))
```

Oben ist zu sehen, dass zuerst ein „select“ ausgeführt wird, um die Key- und Value-Werte zu erhalten. Das Ganze wird dann mit „.rdd“ in ein RDD umgewandelt und anschließend in einer Map gespeichert. Das „row.getString(0)“ bezieht sich auf die Werte in „depth_sensorname“ und „row.getInt(1)“ entsprechend auf die Werte in „depth“.
Die Funktion in „reduceByKey“ hat in ***a*** das Ergebnis der vorherigen Berechnung und in ***b*** den nächsten Wert aus der Tabelle stehen. In diesem Fall wird der größere Wert von beiden gesucht. Als Ergebnis wird der größte Wert pro Sensorname ausgegeben.
Wenn jetzt die gleiche Anfrage auf das DataFrame, mit dem dafür entwickelten GroupBy, übertragen wird, dann sieht das wie folgt aus:

```
import1
.select("depth_sensorname", "depth")
.filter($"depth".isNotNull)
.groupBy($"depth_sensorname")
.agg(max($"depth"))
.collect()

res: Array[org.apache.spark.sql.Row]=Array([DEF,2001],[GHI,2001],[JK,2001])
```

Auch hier wird mit einem „select“ begonnen, um die beiden Anfragen vergleichbar zu machen. In diesem Fall findet aber keine Umwandlung statt. Stattdessen wird gleich das GroupBy ausgeführt und mit dem „.agg“ mitgeteilt, das von „depth“ der maximale Wert gesucht wird.

Werden nun die Zeiten verglichen, wird deutlich, dass das alte System mit ReduceBy deutlich langsamer ist:

- ReduceByKey: 14,1 s
- GroupBy: 9,0 s

Trotzdem kann es Situationen geben, in denen das ReduceBy Vorteile bietet. Dem ReduceBy wird eine Funktion übergeben, dessen Inhalt selber geschrieben werden muss. Dadurch sind die Möglichkeiten deutlich größer, als beim GroupBy des DataFrame, bei dem die Funktionen vorgegeben sind. Da die Anfragen in dieser Arbeit mit den Standardfunktionen auskommen, wird das GroupBy des DataFrame in allen betroffenen Analyseanfragen genutzt.

3.2.2. Analyse-Test

Die im Folgenden beschriebenen unterschiedlichen Anfragen sollen im Analysetest mit den in Tabelle 6 auf Seite 26 genannten Executor-Einstellungen ausgeführt werden. Dafür wurde die Scala-Datei „Analyse.sca“ geschrieben. Ziel ist es, im ersten Schritt herauszufinden, welche Speichervariante mit welchem Dateiformat und welcher Serialisierung die höchste Analysegeschwindigkeit unterstützt, um dann im zweiten Schritt die unterschiedlichen Executor-Einstellungen zu vergleichen.

3.2.2.1. Geoanalyse

Es werden in dieser Arbeit zwei unterschiedliche Ansätze für die Geoanalyse verwendet. Zum einen soll ein Sensorwert nach seiner Latitude- und Longitude-Position gruppiert werden, und zum anderen sollen die Informationen anhand von Shapefiles gefiltert werden. Ein Shapefile ist ein Dateiformat, welches von der Firma ESRI, für Geodaten, entwickelt wurde. Für den Vergleich mit Shapefiles wird das Zusatzprogramm Magellan verwendet. Magellan ist ein Programm für die Geodatenanalyse mit Spark und läuft unter der Apache-Lizenz. Es wurde maßgeblich von Ram Sriharsha entwickelt, welcher zusätzlich einer der aktuellen Entwickler von Apache Spark ist. Aus diesem Grund wird davon ausgegangen, dass Magellan reibungslos mit Apache Spark zusammenarbeitet. Das Programm kann gängige Formate, wie z.B. das genannte Shapefile, verwenden und bietet ein Anfrageformat, das dem von Spark ähnelt. Aufgrund dieser Vorteile wurde Magellan ausgewählt. (Sriharsha, 2017)

In der Testanfrage soll Magellan dazu verwendet werden, die Zeilen heraus zu filtern, deren Koordinaten in einem vorgegebenen Shapefile liegen. Um Magellan effektiv nutzen zu können, wird den Tabellen vor dem Abspeichern eine zusätzliche Spalte hinzugefügt. Diese heißt „point“ und beinhaltet die Latitude- und Longitude-Werte, welche von einer Magellan-Methode zu einem Point umgewandelt werden.

Für den zweiten Testfall, bei dem die Latitude- und Longitude-Werte für die Gruppierung genutzt werden sollen, gibt es mehrere Ansätze. Es ist möglich, zu den Werten Zusatzinformationen in einer zusätzlichen Spalte abzuspeichern. Das könnte z.B. der Name eines Landes oder einer Stadt sein. Da sich die Geodaten in dem hier behandelten Praxisfall hauptsächlich auf dem Meer befinden, ist diese Methode ungeeignet. Aus diesem Grund wird die Welt gerastert betrachtet. Über einen Algorithmus wird jetzt eingeteilt, in welchem Abschnitt sich eine Position mit ihrem Latitude- und Longitude-Wert befindet. Die Nummer des Abschnitts wird in der Datenbank gespeichert. Der vollständige Algorithmus ist im Anhang auf Seite 65 zu finden. Im Folgenden die unterschiedlichen Geoanalyseanfragen.

Magellan Anfragen

Name: magellan_groupBy

```
import2
.join(shapeFile).where(import2.col("point") within $"polygon")
.groupBy($"month")
.agg(avg($"salt_content"),min($"salt_content"), max($"salt_content"))
.sort($"month")
.show(false)
```

Die erste Anfrage sucht nach den Informationen, deren „point“-Werte im Shapefile liegen. Von diesen Informationen werden die vorhandenen Monate sowie die durchschnittlichen, minimalen und maximalen Salzgehalte der jeweiligen Monate ausgegeben.

Name: magellan_sort

```
import1
.join(shapeFile).where(import1.col("point") within $"polygon")
.select($"utc", $"depth_sensorname", $"depth")
.filter($"depth".isNotNull )
.sort($"depth_sensorname".desc)
.count()
```

Die zweite Anfrage sucht ebenfalls nach den Informationen, deren „point“-Werte im Shapefile liegen und gibt davon den Zeitstempel, den Sensornamen und den Sensorwert aus.

Geo-ID-Anfragen

In dieser Anfrage soll der zweite Geoanalyse-Ansatz untersucht werden. In der Spalte „geo_id“ sind die Abschnittsnummern des Rasters gespeichert. In dem ersten GroupBy wird die „Geo-ID“ verwendet, die beim Abspeichern hinzugefügt wurde. In dem zweiten GroupBy wird die „Geo-ID“ während der Anfrage generiert. Anhand der Ergebnisse kann dann entschieden werden, ob es sich lohnt, die Spalte vorher zu generieren und ob sich das Rasterverfahren grundsätzlich für die Praxis eignet.

Name: groupBy_geoID

```
import1
.select("geo_id_555km", "depth")
.filter($"depth".isNotNull )
.filter($"geo_id_555km" =!= -1)
.groupBy($"geo_id_555km")
.agg(avg($"depth"), min($"depth"), max($"depth"))
.sort($"geo_id_555km")
.show(2592, false)
```

Name: groupBy_newGeoID

```
import1
.withColumn("new_geo_id_555km",getGeoId(lit(5.0),$"latitude",$"longitude"))
.select("new_geo_id_555km", "depth")
.filter($"depth".isNotNull )
.filter($"geo_id_555km" =!= -1)
.groupBy($"new_geo_id_555km")
.agg(avg($"depth"), min($"depth"), max($"depth"))
.sort($"new_geo_id_555km")
.show(2592, false)
```

Beide Anfragen gruppieren nach der Spalte „geo_id“ und geben dazu die durchschnittliche, minimale und maximale Tiefe aus.

3.2.2.2. GroupBy

Nachdem in Kapitel „3.2.1. GroupBy- oder ReduceBy-Test“ geklärt wurde, dass GroupBy schneller ist, als das ReduceBy, soll in diesem Abschnitt das GroupBy anhand von Anfragen vertiefend untersucht werden.

GroupBy mit und ohne vorherigem „select“

Mit diesen beiden Anfragen soll untersucht werden, ob es einen Unterschied in der Geschwindigkeit gibt, wenn vor dem GroupBy ein „select“ ausgeführt wird.

Name: groupBy_ohne_select

```
import1
.filter($"depth".isNotNull )
.groupBy($"depth_sensorname")
.agg(max($"depth"), min($"depth"))
.show(false)
```

Name: groupBy_mit_select

```
import1
.select("depth_sensorname", "depth")
.filter($"depth".isNotNull )
.groupBy($"depth_sensorname")
.agg(max($"depth"), min($"depth"))
.show(false)
```

Die Anfragen gruppieren nach den Sensornamen und geben zu dem entsprechenden Namen den minimalen und maximalen Messwert aus.

GroupBy nach partitioniertem, generiertem und abgespeichertem Wert

Beim Speichern sind einige Tabellen nach „month“ partitioniert worden. Bei diesen zwei Anfragen soll untersucht werden, wie groß der Unterschied ist, wenn nach einem partitionierten, einem abgespeicherten und einem neu generierten Wert gruppiert wird.

Name: groupBy_month

```
import1
.select("month", "depth")
.filter($"depth".isNotNull )
.groupBy($"month")
.agg(avg($"depth"), min($"depth"), max($"depth") )
.sort($"month")
.show(false
```

Name: groupBy_newMonth

```
import1
.withColumn("new_month", month($"timestamp"))
.select("new_month", "depth")
.filter($"depth".isNotNull )
.groupBy($"new_month")
.agg(avg($"depth"), min($"depth"), max($"depth"))
.sort($"new_month")
.show(false)
```

Die Anfragen geben für jeden Monat die durchschnittliche, minimale und maximale Tiefe aus.

3.2.2.3. Join

Ursprünglich sollte der „join“ über die partitionierte Spalte „geo_id_555km“ gehen. Im Laufe der Untersuchung stellte sich heraus, dass der Test-Cluster, für die nach „geo_id“ partitionierten und mit Snappy komprimierten Tabellen, zu wenig Arbeitsspeicher hat. Im Verlauf des Importvorgangs wurde immer eine Java-Heap-Size-Exception ausgegeben, weshalb die Anfrage und die Tabellen geändert wurden. Die Tabellen sind jetzt nur noch nach „month“ partitioniert und der „join“ erfolgt entsprechend über den „month“-Wert.

Join nach „month“ und generiertem „month“

In der „join“-Anfrage werden die beiden Tabellen erst nach Monaten gruppiert und anschließend über die gruppierten Monate ein „join“ durchgeführt. Durch die Gruppierung konnte die Ausführungszeit um ein Vielfaches reduziert werden.

Da die „month“-Spalte beim Abspeichern partitioniert wurde, soll im Vergleich zwischen den beiden Anfragen geklärt werden, ob die Partitionierung auch einen positiven Effekt auf einen solchen „join“ hat.

Name: join_month

```
import2.as("t2")
.select("month","salt_content" )
.filter($"salt_content".isNotNull)
.groupBy($"month")
.agg(avg($"salt_content"))
.join(
import1.as("t1")
.select("month","depth")
.groupBy($"month")
.agg(avg($"depth")),
$"t1.month" === $"t2.month")
.sort($"t1.month")
.show(false)
```

Name: join_newMonth

```
import2
.withColumn("new_month", month($"timestamp")).as("t2")
.select("new_month","salt_content" )
.filter($"salt_content".isNotNull)
.groupBy($"new_month")
.agg(avg($"salt_content"))
.join(
import1
.withColumn("new_month", month($"timestamp")).as("t1")
.select("new_month","depth")
.groupBy($"new_month")
.agg(avg($"depth")),
$"t2.new_month" === $"t1.new_month")
.sort($"t1.new_month")
.show(false)
```

Die Anfrage gibt von beiden Tabellen die durchschnittliche Tiefe, nach Monaten sortiert, aus.

3.2.2.4. Mit Zusatzspalte

Zusätzlich zu den vorherigen Untersuchungen, wird es noch eine weitere Anfrage mit einer Zusatzspalte geben. Von Interesse ist der Zeitstempel der Messdaten, da diese sehr unregelmäßig sein können. Dieses liegt an den unterschiedlichen Intervallen der Messungen. Sollen nun Messwerte anhand ihrer Zeitstempel zusammengebracht werden, so führen diese Unregelmäßigkeiten zu Problemen. Aus diesem Grund soll in dieser Anfrage die „from_unixtime“-Methode von Spark untersucht werden. Diese ermöglicht es, den Zeitstempel auf eine bestimmte Zeiteinheit zu runden. Im Folgenden die Testanfrage, die auf Minuten rundet.

Name: zusatzspalte_minute

```
import1.as("t1")
.withColumn("minute", from_unixtime(($"utc"/1000), "yyyy-MM-dd HH:mm"))
.filter($"wind_speed_sensorname" === "FGH")
.filter($"wind_speed_knots".isNotNull)
.select("minute", "wind_speed_knots")
.count()
```

Die Anfrage generiert eine Spalte, in welcher die gerundeten Zeitstempel gespeichert werden. Anschließend werden die Werte von „wind_speed_knots“ mit dem gerundeten Zeitstempel ausgegeben.

3.2.3. Import-Test

Beim Importtest soll die Geschwindigkeit des Importierens gemessen werden. Dafür wurde das Scala-Skript „Import.sca“ geschrieben. Mithilfe des Skriptes werden die Schemata vordefiniert und mit Metadaten in der „utc“-Spalte versehen. Die Metadaten werden an die Spalte gehängt, um herauszufinden, ob die Informationen nach dem Abspeichern noch abrufbar sind. Mit diesen vordefinierten Schemata werden jetzt die beiden Testtabellen im „csv-Format“ gelesen und in unterschiedlichen Varianten abgespeichert. Insgesamt gibt es acht verschiedene Varianten. Die vier folgenden Punkte werden jeweils einmal in ORC und in Parquet abgespeichert.

1. mit der Komprimierung Snappy und partitioniert nach „month“
2. ohne Komprimierung und partitioniert nach „month“
3. mit der Komprimierung Snappy und ohne Partitionierung
4. ohne Komprimierung und ohne Partitionierung

Darüber hinaus erfolgen die acht Speichervorgänge einmal mit der Java- und einmal mit der Kryo-Serialisierung. Das Abspeichern erfolgt mit unterschiedlichen Executor-Einstellungen. Für die Executoren werden, genau wie bei der Analyse, die Einstellungen von Tabelle 6 auf Seite 26 verwendet. Auch hier soll im ersten Testdurchlauf herausgefunden werden, welche Speichervariante mit welchem Dateiformat und welcher Serialisierung die höchste Geschwindigkeit unterstützt, um dann im zweiten Schritt die unterschiedlichen Executor-Einstellungen zu vergleichen.

3.2.4. Speicherbelegung

Um die Speicherbelegung der abgespeicherten Dateien schnell zu ermitteln, ist eine „for“-Schleife in der Bash-Datei programmiert worden. Dieser wird ein Array mit den acht Dateinamen, aus dem Import übergeben. Die Schleife speichert den Dateinamen, mit der dazugehörenden Größe, in eine Textdatei.

3.2.5. Dynamische Zuweisung bei kurzen Anfragen

Zusätzlich zu den genannten Untersuchungen zur Analysegeschwindigkeit soll getestet werden, ob die dynamische Ressourcenzuweisung bei kurzen Anfragen zeitliche Nachteile bringt. Die Frage kam auf, weil die Zuweisung der Ressourcen einige Sekunden dauert. Bei der manuellen Zuweisung erfolgt die Zuweisung beim Starten des Jobs. Bei der dynamischen Zuweisung dagegen erfolgt die Zuweisung erst während der Laufzeit.
Für den Test wird eine kurze Filteranfrage mit dynamischer und manueller Executor-Zuweisung durchgeführt. Um genaue Aussagen darüber treffen zu können, worin die zeitlichen Unterschiede liegen, wird an zwei Stellen die Zeit gemessen. Zum einen wird die Ausführungszeit der Filteranfrage gemessen und zum anderen die gesamt Ausführungszeit des Jobs. Im Folgenden steht die kurze Filteranfrage.

Name: filter

```
import.
select($"utc", $"wind_speed_kmh", $"wind_speed_sensorname")
.filter($"wind_speed_sensorname" isin ("DE", "FGH"))
.filter($"wind_speed_kmh" < 10)
.count()
```

Die Anfrage gibt die Werte und Zeitstempel von „wind_speed_kmh“ aus, die von den Sensoren DE oder FGH gemessen wurden.

3.2.6. Append-Test

Neben dem Testen des Importierens von Dateien soll auch das Anhängen einer Datei an eine existierende untersucht werden. Der Hintergrund ist, dass für neue Messungen nicht immer eine neue Datei angelegt werden, sondern eine existierende erweitert werden soll. Da die beiden Dateiformate ORC und Parquet Statistiken über die Inhalte im Footer speichern, soll geklärt werden, wie die Formate mit einem Append umgehen und ob er zeitaufwändiger ist, als ein normaler Import. Darüber hinaus wird untersucht, inwieweit sich die Analysegeschwindigkeit und die Dateigröße verändern.

3.2.7. FAIR-Mode- und FIFO-Mode-Test

Die FAIR- und FIFO-Einstellung regelt, wie Jobs ausgeführt werden, wenn nicht genügend Ressourcen für alle angeforderten Executoren vorhanden sind. Im FIFO-Mode werden die Jobs in der Reihenfolge ausgeführt, in der sie ankommen, im FAIR-Mode dagegen ist die Priorität entscheidend, die die einzelnen Jobs haben. In der Spark-Konfiguration ist der FIFO-Mode die Standardeinstellung. Um zu überprüfen, ob FAIR- oder FIFO-Mode für diesen Praxisfall die bessere Einstellung ist, wurde eine XML-Datei geschrieben, welche im Anhang auf Seite 66 zu finden ist. Diese wird benötigt, um einen Job, unabhängig von der Standardeinstellung, im FAIR-Mode oder FIFO Mode ausführen zu können. In der XML-Datei sind drei Job-Pools definiert. Im Folgenden werden die Eigenschaften der drei Pools erklärt.

1. Pool hat die Modezuweisung „FAIR“ und eine hohe Priorität
2. Pool hat die Modezuweisung „FAIR“ und eine geringe Priorität
3. Pool hat die Modezuweisung „FIFO“

Entsprechend der Priorität des Pools im FAIR-Mode, werden den Anfragen mehr oder weniger Ressourcen zur Verfügung gestellt. So kann vom Anwender reguliert werden, ob es sich um eine wichtige oder weniger wichtige Anfrage handelt. Wenn sich in einem Job-Pool mehrere Jobs befinden, dann werden diese weiterhin nach der FIFO-Regel abgearbeitet. (Apache Software Foundation, Job Scheduling, 2016)
Um die genauen Auswirkungen der beiden Modi zu untersuchen, wurden die beiden Scala-Dateien *„AnalyseHighPrioPool.sca“* und *„AnalyseLowPrioPool.sca“* geschrieben. Die eine Datei wird mit hoher Priorität ausgeführt und die andere mit geringer Priorität. Abgesehen von den Pool-Einteilungen werden die beiden Dateien die gleichen Anfragen beinhalten und mit den gleichen Ressourcenforderungen gestartet.

Folgende Test werden durchgeführt:

1. Test Zwei Jobs, die die genannten Scala-Dateien mit dynamischer Zuweisung aufrufen
2. Test Zwei Jobs, die die genannten Scala-Dateien mit 2 Executoren mit jeweils 5 Core und 5 Gigabyte Arbeitsspeicher aufrufen
3. Test Zwei Jobs mit der FIFO-Pool-Zuweisung und dynamischer Ressourcenzuweisung
4. Test Zwei Jobs mit der FIFO-Pool-Zuweisung und mit 2 Executoren mit jeweils 5 Core und 5 Gigabyte Arbeitsspeicher

Grundsätzlich werden pro Test zwei Szenarien untersucht:

1. Beide Jobs starten gleichzeitig
2. Der zweite Job startet 2 Minuten später, damit die Ressourcenzuweisung des ersten Jobs abgeschlossen ist

In der Praxis wird Variante 2 der häufigere Fall sein, trotzdem ist es wichtig zu untersuchen, was passiert, wenn zwei Jobs gleichzeitig gestartet werden, bzw. wenn sich zwei Jobs in der Warteschleife befinden.

3.2.8. Lasttest

Beim Lasttest sollen die Grenzen von Spark untersucht werden. Dabei sollen folgende Fragen geklärt werden.

1. Welche Unterschiede gibt es zwischen den drei Ressourcen Core, Arbeitsspeicher und Speicher wenn die maximalen Auslastungen erreicht sind?
2. Was passiert, wenn die Executoren mehr Ressourcen fordern, als vorhanden sind?
3. Was passiert, wenn ein Executor mehr Ressourcen fordert, als der YARN-Container an Kapazitäten aufweist?
4. Was passiert, wenn den Executoren nicht genügend Ressourcen zugewiesen werden?
5. Was passiert, wenn ein Knoten ausfällt, während Executoren darauf laufen?
6. Wie werden die Ressourcen zugeteilt, wenn nicht alle Ressourcen genutzt werden?

3.3. Durchführung und Analyse

In diesem Kapitel werden die Tests durchgeführt, ausgewertet und gegebenenfalls verfeinert. Die dazu gehörenden Ergebnisse befinden sich im Anhang ab Seite 58. Der gesamte Testablauf ist in mehrere Schritte gegliedert. Zuerst werden sämtliche Import- und Analyseanfragen mit den unterschiedlichen Executor-Zuweisungen durchgeführt. In der ersten Untersuchung werden alle möglichen Kombinationen der beiden Speicherformate (Parquet und ORC), der vier Speichervarianten (Snappy & Partition, entweder Snappy oder Partition, weder Snappy noch Partition) und der beiden Serialisierungsmethoden (Java und Kryo) miteinander verglichen. Aus diesen 16 Varianten wird die Einstellung ermittelt, die am schnellsten zum Ergebnis führt.
In der zweiten Untersuchung werden dann die verschiedenen Executoren genauer untersucht und auch hier die schnellste Kombination ermittelt. Im Anschluss, wenn die grundlegenden Einstellungen feststehen, werden die restlichen Tests durchgeführt und ausgewertet.

3.3.1. Import- und Analyseergebnisse

Nachdem alle Import- und Analyseanfragen durchgeführt wurden, wird eine Entscheidungsmatrix für die Auswahl der Speichervariante, des Dateiformates und der Serialisierung erstellt. Die Matrix ist im Anhang auf den Seiten 58 bis 59 zu finden. Im Folgenden sind die Entscheidungskriterien und die unterschiedlichen Alternativen erklärt.

Kriterien

- Alle Analyseanfragen in Sekunden
- Die Importgeschwindigkeit in Minuten
- Die Dateigröße in Gigabyte

Die Analyseanfragen haben zusammen eine Gewichtung von 60%, gefolgt von der Importgeschwindigkeit mit 30% und der Dateigröße mit 10%. Die Gewichtungen sind vorgegeben.

Die Alternativen

- Java-Serialisierung mit ORC in den 4 Varianten Snappy & Partition, nur Partition, nur Snappy und ohne Snappy oder Partition
- Java-Serialisierung mit Parquet in den 4 Varianten Snappy & Partition, nur Partition, nur Snappy und ohne Snappy oder Partition
- Kryo-Serialisierung mit ORC in den 4 Varianten Snappy & Partition, nur Partition, nur Snappy und ohne Snappy oder Partition
- Kryo-Serialisierung mit Parquet in den 4 Varianten Snappy & Partition, nur Partition, nur Snappy und ohne Snappy oder Partition

Insgesamt kommen so 16 Alternativen zusammen, aus denen jetzt die Beste ausgewählt wird. Für die Entscheidung werden die gemessenen Zeiten unter Messwert eingetragen. Da eine möglichst kurze Ausführungszeit und ein geringer Platzverbrauch beim Speichern das Ziel ist, wird entsprechend nach den kleinsten Messwerten gesucht. Innerhalb eines Kriteriums wird dem niedrigsten Messwert der höchste Platz zugeteilt. Bei 16 Alternativen ist das der 16. Platz. Wenn zwei Alternativen den gleichen Messwert aufweisen, bekommen beide den gleichen Platz zugewiesen. Dieser zugewiesene Platz wird mit der Gewichtung des entsprechenden Kriteriums multipliziert und das Ergebnis in der Spalte Bewertung notiert. Die Bewertungen werden aufsummiert und bilden das Endergebnis.
Bei der Auswertung der Matrix wird deutlich, dass die „GroupBy“ und „join“-Anfragen über die Monate keine Unterschiede bei partitionierten Daten zeigen. Daraus wird gefolgert, dass „GroupBy“ und „join“ die Vorteile einer Partitionierung nicht nutzen können. Da aber untersucht werden soll, welche Vorteile eine Partitionierung bietet, wird eine weitere Analyseanfrage hinzugefügt. Diese filtert jetzt nach den partitionierten Monaten und sieht wie folgt aus.

Name: filter_month

```
import1
.select("utc", "wind_speed_kmh", "wind_speed_sensorname", "month")
.filter($"month" === 1)
.filter($"wind_speed_sensorname" isin ("DE", "FGH"))
.filter($"wind_speed_kmh" < 10)
.count()
```

Die bereits bekannte Filteranfrage wird erweitert, sodass nur Werte, die im Januar gemessen wurden, ausgegeben werden.

Diese Anfrage wird nachträglich mit in die Entscheidungsmatrix übernommen und für das Ergebnis mitbewertet.
Wenn in der Matrix der neu hinzugefügte Filter genauer betrachtet wird, dann fällt auf, dass bei ORC der Unterschied zwischen partitionierten Tabellen und nicht partitionierten Tabellen gut zu erkennen ist. Bei Parquet jedoch sind die Ausführungszeiten so kurz, dass kein Unterschied erkennbar ist. Aus diesem Grund wird dieser Bereich noch einmal gesondert untersucht. So soll sichergestellt werden, dass das Ergebnis der Entscheidungsmatrix nicht verfälscht ist. Für die genauere Untersuchung wird mit einer Filteranfrage die Geo-ID aus den alten, nach „geo_id“ partitionierten Tabellen, herausgefiltert. Für kleine Anfragen auf die unkomprimierte Tabelle reicht der Arbeitsspeicher. Zur Sicherheit wird der Test zweimal durchgeführt.

```
testImport1
.select("utc", "geo_id_555km")
.filter($"geo_id_555km"===500)
.show(150,false)
```

	Testdurchlauf 1	Testdurchlauf 2
Tabelle 1 ohne Partitionierung	8,5 Sekunden	9,2 Sekunden
Tabelle 1 mit Partitionierung	2,2 Sekunden	3,2 Sekunden

Tabelle 7: Messergebnisse des Filters für die Geo-ID

Das Ergebnis der Anfragen ist eindeutig, weshalb davon ausgegangen wird, dass bei Parquet die kurzen Ausführungszeiten die Ursache für die ähnlichen Werte sind.
Als Gesamtergebnis ergibt die Matrix, dass die Alternative Kryo-Serialisierung mit dem Dateiformat Parquet und der Snappy-Komprimierung die schnellste ist. Die Partitionierung führte bei vielen Anfragen zu Verzögerungen, weshalb sie sich nur lohnt, wenn ausschließlich nach den partitionierten Werten gefiltert wird.
Nachdem das am besten geeignete Dateiformat und grundlegende Einstellungen ermittelt wurden, müssen noch die Executoren untersucht werden.
Bei der Entscheidung, welche Executor-Zuweisung am besten geeignet ist, werden Import und Analyse getrennt betrachtet. Dieses wird in zwei weiteren Entscheidungsmatrizen erfolgen, welche im Anhang auf den Seiten 60 und 61 zu finden sind. Es ist möglich, dass diese beiden grundverschiedenen Anfragen unterschiedliche Executoren benötigen, um ihre optimale Geschwindigkeit zu erreichen. Entsprechend Tabelle 6 auf Seite 26, sind in den beiden Entscheidungsmatrizen die folgenden Alternativen zu finden.

- Core:1, Memory:1 Gigabyte, Anzahl:12
- Core:3, Memory:1 Gigabyte, Anzahl:4
- Core:3, Memory:4 Gigabyte, Anzahl:4
- Core:6, Memory:4 Gigabyte, Anzahl:2
- Core:3, Memory:10 Gigabyte, Anzahl:4
- Core:6, Memory:10 Gigabyte, Anzahl:2
- Core:6, Memory:14 Gigabyte, Anzahl:2
- dynamische Zuweisung

Durch die getrennte Betrachtungsweise müssen auch die Kriterien angepasst werden. In der Import-Matrix wird die Importzeit untersucht. In der Analyse-Matrix werden entsprechend die Zeiten der Analyseanfragen betrachtet. Die Dateigröße spielt keine Rolle mehr, da die unterschiedlichen Executoren keinen Einfluss darauf haben. Die Auswertung der Messwerte erfolgt, wie in der ersten Entscheidungsmatrix, mithilfe der Reihenfolge innerhalb eines Kriteriums. Um die Genauigkeit der Messungen zu erhöhen, wurden die Tests mehrmals durchgeführt und anschließend die Durchschnittswerte in den Matrizen ausgewertet.
Die Analysematrix ergibt, dass 2 Executoren mit 6 Cores und 4 Gigabyte Memory die kürzesten Zeiten liefern. Bei der Importmatrix sind 12 Executoren mit 1 Core und 1 Gigabyte Memory am schnellsten. Da im ersten Testdurchlauf nur wenige Zuweisungen getestet wurden, um ein breites Feld abdecken zu können, werden jeweils zu dem besten Ergebnis weitere Tests durchgeführt. Dieses Vorgehen soll sicherstellen, dass die beiden Zuweisungen auch tatsächlich die schnellsten Varianten sind. In den Tabellen 8 und 9 sind die neuen Ressourcenzuweisungen der Executoren aufgelistet.

Cores	Memory	Executoren
1	2 Gigabyte	12
1	3 Gigabyte	12
2	1 Gigabyte	6
2	2 Gigabyte	6
2	3 Gigabyte	6

Tabelle 8: Neue Ressourcenzuweisungen für die Import-Executoren

Cores	Memory	Executoren
5	3 Gigabyte	2
5	4 Gigabyte	2
5	5 Gigabyte	2
5	6 Gigabyte	2
6	3 Gigabyte	2
6	5 Gigabyte	2
6	6 Gigabyte	2

Tabelle 9: Neue Ressourcenzuweisungen für die Analyse-Executoren

Die Entscheidungsmatrizen für die neuen Ressourcenzuweisungen sind im Anhang auf den Seiten 62 und 63 zu finden. Auch in diesen Matrizen werden Durchschnittswerte aus mehreren Durchläufen verwendet, um die Aussagekraft zu erhöhen.
In der neuen Import-Matrix ist gut zu erkennen, dass die schnellste Executor-Zuweisung aus der vorherigen Matrix erneut das schnellste Ergebnis liefert. Aus diesem Grund wird davon ausgegangen, dass 1 Core mit 1 Gigabyte Arbeitsspeicher die geeignetste Zuweisung für den Import ist. Anders sieht das Ergebnis bei der Analyse aus. Hier sind die neuen Zuweisungen schneller als die alte. Das Besondere ist, dass zwei Zuweisungen die gleiche Punktzahl erreicht haben. Das ist zum einen die Zuweisung „6 Cores mit 6 Gigabyte Arbeitsspeicher“ und zum anderen die Zuweisung „5 Cores mit 5 Gigabyte Arbeitsspeicher“. Als besseres Ergebnis wird die Ressourcenzuweisung „5 Cores und 5 Gigabyte Arbeitsspeicher“ gewählt, da es keinen Sinn macht, für die gleiche Geschwindigkeit mehr Ressourcen zu belegen.
Zusätzlich zu den Geschwindigkeiten sollen auch die Analyseanfragen selber untersucht werden. Im Folgenden die Untersuchungen zwischen den Analyseanfragen.

Anfrage mit und ohne Select

Wie in Tabelle 10 zu sehen, waren die Anfragen mit „select“ deutlich schneller. Es lohnt sich bei Analyseanfragen die Spalten, mithilfe des „selects“, auf die Benötigten zu reduzieren.

Name	Zeit [Sek.]
groupBy mit select	1,2
groupBy ohne select	2,2

Tabelle 10: Vergleich GroupBy mit und ohne Select

Vergleich des abgespeicherten Wertes mit generiertem und mit partitioniertem Wert

In Tabelle 11 sind die drei Anfragen aufgelistet, die mit einem abgespeicherten Wert arbeiten. Zu diesen Anfragen gab es jeweils eine identische Anfrage, welche sich den entsprechenden Wert zur Laufzeit generierte.

Name	abgespeicherter Wert [Sek.]	generierter Wert [Sek.]	partitionierter Wert [Sek.]
groupBy month	1,6	11,3	2,3
groupBy geo_id	1,4	854	2,6
join month	7,2	25,5	8,2

Tabelle 11: Vergleich abgespeicherter Wert mit neu generiertem und partitioniertem Wert

Wie schon bei den Entscheidungsmatrizen aufgefallen ist, hat die Partitionierung keinen Einfluss auf „GroupBy“- und „join“-Anfragen. Stattdessen verlängert die Partitionierung die Anfrage sogar. Im Vergleich zwischen dem normal-abgespeicherten Wert und dem neu-generierten Wert kann klar gesagt werden, dass sich das Abspeichern lohnt. Die Anfragezeiten sind bei den abgespeicherten Werten um ein vielfaches geringer. Darüber hinaus dauert der Import mit dem Abspeichern der „month“-Spalte nur ein paar Sekunden länger. Das ist bei mehreren Minuten Importzeit ein zu vernachlässigender Mehraufwand, insbesondere da die Analysen so deutlich profitieren.

Zeitstempel runden

Bei einer Anfrage wurde ein gerundeter Zeitstempel generiert und ausgegeben. Ziel war es, herauszufinden, ob die Umrechnung des Zeitstempels schnell genug ist für den Praxiseinsatz. Die Ausführungszeiten in den Matrizen zeigen deutlich, dass es sich um eine einfache und schnelle Lösung handelt, die Zeitstempel zu vereinheitlichen.

3.3.2. Dynamische Zuweisung bei kurzen Anfragen

Wie in Abbildung 8 zu sehen, gibt es keine relevanten zeitlichen Unterschiede zwischen der dynamischen und der manuellen Zuweisung bei kurzen Anfragen. Die Differenz in der Gesamtausführungszeit entspricht der Differenz der Anfrage selber. Da in der Gesamtausführungszeit die Ressourcenzuweisung von beiden Anfragen enthalten ist, müssten diese Zeiten gleich sein. Alle Unterschiede, die sich hier zeigen, sind auf die unterschiedlichen Ausführgeschwindigkeiten der unterschiedlichen Ressourcenzuweisungen zurückzuführen. Aus diesem Grund wird davon ausgegangen, dass die dynamische Zuweisung keine merklichen zeitlichen Verzögerungen mit sich bringt.

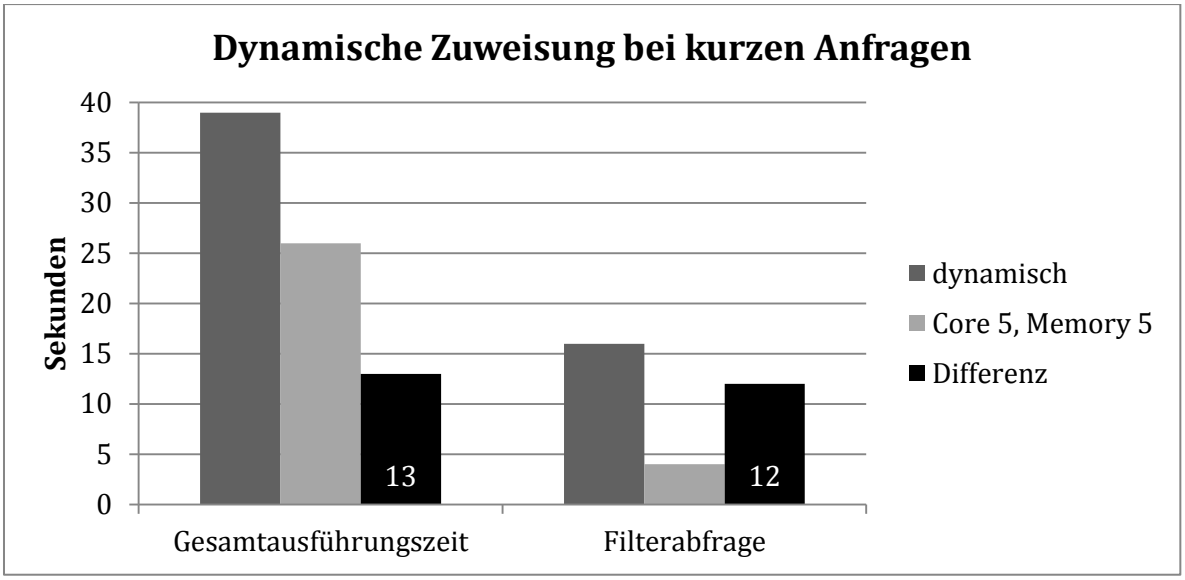

Abbildung 8: Dynamische Ressourcenzuweisung bei kurzen Anfragen

3.3.3. Append-Untersuchungen

Bei den Untersuchungen zum Append wurde eine Tabelle normal importiert und anschließend mit der Einstellung „Append“ um die gleiche Tabelle erweitert. Wie in Abbildung 9 zu sehen, hat die Verdoppelung der Tabelle eine Verdoppelung der Dateigröße und der Analysezeit zur Folge. Die Zeit, die benötigt wird, um die Tabelle anzuhängen, unterscheidet sich nicht von der Zeit, die benötigt wird, um die Tabelle normal zu importieren. Dieses ändert sich auch bei weiteren Imports mit der Append-Einstellung nicht.

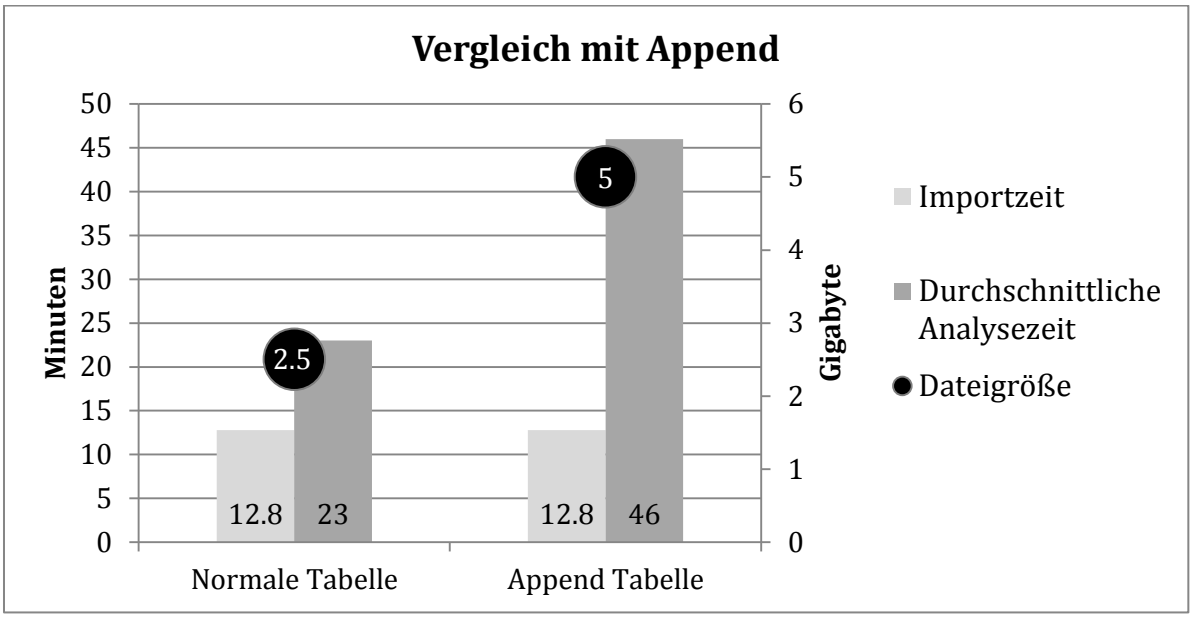

Abbildung 9: Vergleich einer normalen Tabelle mit einer mit Append erweiterten Tabelle

Zusätzlich zu den Geschwindigkeitstests wurde untersucht, wie sich die Tabelle verhält, wenn ein Import fehlschlägt, bzw. wenn ein Import mit Append-Einstellung auf eine Tabelle ausgeführt wird, auf der gerade eine Analyse läuft.

Bei einem Import mit Append-Einstellung wird in dem existierenden Verzeichnis der bereits vorhandenen Daten ein neues Verzeichnis mit einem einzelnen Unterverzeichnis angelegt. Diese heißen „_temporary/0/“ und beinhalten den gesamten Import, abgekapselt von den existierenden Daten. Dort werden alle fertig-bearbeiteten Tasks zwischengespeichert. Die Tasks, die noch in Arbeit sind, liegen in einem weiteren „_temporary“ Ordner, der in dem Verzeichnis „/0/“ liegt. Wenn alle Tasks abgearbeitet sind und die anzuhängende Tabelle komplett im Verzeichnis „/0/“ zwischengespeichert ist, wird das „_temporary“ Verzeichnis aufgelöst und der Inhalt mit dem Hauptverzeichnis und den bereits vorhandenen Informationen zusammengeführt. Dadurch, dass der Import in dem „_temporary“-Ordner stattfindet, können während der kompletten Importzeit weiter Analysen durchgeführt werden. Wenn der Import abgeschlossen ist, wartet der Import

auf eine kurze Pause, in der keine Anfragen an das Hauptverzeichnis gestellt werden, um alle Informationen zusammenzuführen. Die Zusammenführung hat in den Tests nur wenige Sekunden gedauert. Ob die Analyseanfrage in der Zeit gewartet hat, konnte aufgrund des kurzen Zeitraums, nicht festgestellt werden.
Die Nutzung des „_temporary"-Verzeichnisses hat noch einen weiteren Vorteil. Wenn der Import aus irgendeinem Grund fehlschlägt, dann sind davon nur die neu zu importierenden Dateien betroffen. Die alten Datenbestände bleiben unberührt.

3.3.4. FAIR-Mode- und FIFO-Mode-Ergebnisse

Die Testergebnisse zu den FAIR-Mode- und FIFO-Mode-Tests sind im Anhang auf Seite 64 zu finden. Zu beachten ist, dass bei den zeitversetzten Anfragen die 120 Sekunden bereits vom Leerlauf abgezogen wurden.
Ganz allgemein kann gesagt werden, dass sich weder der FIFO-Mode noch der FAIR-Mode wie erwartet verhalten haben. Der FIFO-Mode hat die Jobs nicht immer nacheinander abgearbeitet und der FAIR-Mode hat die Priorität nicht beachtet. Darüber hinaus kommt es bei der manuellen Zuweisung bei beiden Modi jeweils nach Warteschleifen, häufig zu dem Phänomen, dass den Jobs zusammen mehr Ressourcen zugewiesen werden als vorhanden sind. Dieses hat zur Folge, dass die betroffenen Jobs länger brauchten als normalerweise. Im Folgenden die detaillierten Ergebnisse.

FIFO-Mode

Wenn die Jobs zeitversetzt ankommen, dann verhält sich der FIFO-Mode wie erwartet. Probleme gibt es, wenn zwei Jobs parallel gestartet werden, bzw. wenn sich zwei Jobs in der Warteschleife befinden und auf Ressourcen warten. In beiden Fällen kann das System nicht entscheiden, welcher Job zuerst da war und die Ressourcen werden auf beide aufgeteilt. Das hat zur Folge, dass beide Jobs parallel abgearbeitet werden und mehr Zeit benötigen, als bei der seriellen Abarbeitung.

FAIR-Mode

Werden zwei Jobs gleichzeitig gestartet werden oder sich gleichzeitig in der Warteschleife befinden, dann werden beiden Jobs Ressourcen zugewiesen ohne die unterschiedlichen Prioritäten zu berücksichtigen. Bei Jobs mit dynamischer Zuweisung besteht hier der Vorteil, dass Executoren, die vom Job mit geringer Priorität nicht mehr benötigt werden, dem Job mit hoher Priorität zugewiesen werden. Über einen längeren Zeitraum führt das dazu, dass der Job mit hoher Priorität schneller ausgeführt wird, als der Job mit geringer Priorität. Wenn die Jobs

zeitversetzt ankommen, werden dem ersten Job die geforderten Ressourcen zugewiesen. Der zweite Job bekommt nur noch die verbleibenden, ungenutzten Ressourcen. Da dem Test-Cluster nur wenige Ressourcen zur Verfügung stehen, blieben keine Ressourcen übrig. Der zweite Job musste warten, bis der erste Job abgearbeitet war. Das bedeutete in diesem Test, dass der Job mit hoher Priorität auf den Job mit geringer Priorität warten musste. Auch hier erfolgt bei der dynamischen Zuweisung eine langsame Umsortierung der Executoren. Da die parallele Laufzeit der beiden Jobs aber nur eine kurze Zeit ausmacht, ist die Gesamtausführungszeit des Jobs mit hoher Priorität trotzdem länger, als die des Jobs mit geringer Priorität.
Neben dem grundsätzlichen Verhalten ist noch interessant, ob es einen Geschwindigkeitsunterschied zwischen den beiden Modi gibt. In Abbildung 10 sind die Gesamtausführungszeiten der dynamischen und manuellen Zuweisung des FAIR- und FIFO-Mode dargestellt. Bei Messungen der zeitversetzten Starts stellte sich das gleiche Verhältnis der Geschwindigkeiten dar, wie beim gemeinsamen Start aus der Warteschleife. In der Grafik wurden daher die Gesamtausführungszeiten der zeitgleichen und zeitversetzten Starts zusammengefasst.

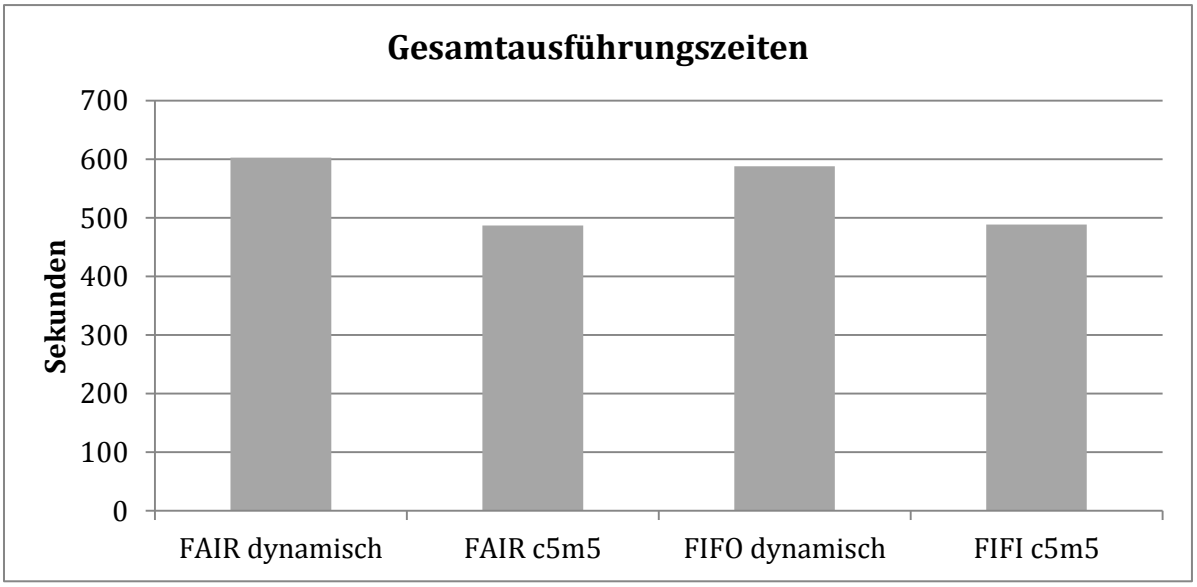

Abbildung 10: Vergleich Ausführungszeiten FAIR- und FIFO-Mode

Es ist zu erkennen, dass die dynamischen Zuweisungen bei beiden Modi mehr Zeit benötigen, als die manuellen Zuweisungen. Dieses unterstützt die Ergebnisse der Entscheidungsmatrizen, wonach die dynamische Zuweisung langsamer ist, als die manuelle Zuweisung mit 5 Cores und 5 Gigabyte Memory. Allerding lässt sich nicht sagen, welche Variante die bessere ist. Die Ergebnisse, dass einmal zugewiesene Ressourcen nicht wieder umverteilt werden, bzw. gleichzeitig ankommende Jobs unabhängig von ihrer Priorität gleich behandelt werden, führen zu der Annahme, dass der FAIR-Mode noch nicht ausgereift ist. Davon ausgehend, dass das Problem in

Zukunft behoben wird, wird empfohlen den FAIR-Mode zu verwenden, um späteren Änderungen in der Konfiguration vorzubeugen.

3.3.5. Lasttest-Ergebnisse

Eine wichtige Erkenntnis der Lasttests ist, dass nicht alle Ressourcen an ihre Grenzen gebracht werden können. So ist es für das System ein Normalzustand, dass zu wenig Cores vorhanden sind. Es ist nicht möglich einem Executor zu wenig Cores zuzuweisen. So lange ein Executor mindestens einen zugewiesenen Core hat, können Anfragen auch bearbeitet werden.
Beim Speicher sieht es ähnlich aus. Solange noch Platz vorhanden ist, können noch Imports erfolgen. Sollte während eines Imports der maximale verfügbare Speicherplatz überschritten werden, so wird eine Fehlermeldung erzeugt und der Import bricht ab. Weitere Einschränkungen konnten nicht festgestellt werden. Die Importzeit wird zum Ende hin nicht länger und auch Analyseanfragen können auf dem vollen Speichersystem noch ohne Zeitverluste ausgeführt werden.
Anders sieht es beim Arbeitsspeicher aus. Wenn ein Executor zu wenig Arbeitsspeicher zugewiesen bekommt, dann kommt eine Java-Heap-Size-Fehlermeldung und der Job wird beendet. Andersherum ist es fast nicht möglich, zu viele Ressourcen anzufordern. Wenn mehr Ressourcen gefordert werden als vorhanden sind, dann werden die zu hohen Forderungen nach nicht verfügbaren Ressourcen ignoriert. Die Ausnahme hiervon bildet die YARN-Containergröße. Sie legt fest, wie viele Ressourcen auf einem Knoten maximal an die Executoren verteilt werden können. Wenn ein einzelner Executor mehr Ressourcen fordert, als der Container an Kapazitäten besitzt, dann wird sofort eine Fehlermeldung ausgegeben und der Job gar nicht erst gestartet.
Grundsätzlich ist es besser, nicht alle vorhandenen Ressourcen zu vergeben. Wenn die Ressourcen zu 100% ausgelastet sind, kommt es häufig zu der Fehlermeldung "… Failt to replace a bad datanode …" mit der Folge, dass die laufenden Anfragen deutlich länger brauchen, als normalerweise. Dieses sieht man in der folgenden Tabelle 12 anhand der beiden Anfragen „magellan_groupBy" und „magellan_sort".

Anfragen	bei normaler Ausführungszeit	Ausführungszeit mit Fehlermeldung
magellan_groupBy	30 Sek.	177 Sek.
magellan_sort	64 Sek.	388 Sek.

Tabelle 12: Unterschiedliche Ausführungszeiten bei Fehlermeldung "… Failt to replace a bad datanode …"

Das Problem entsteht, wenn ein Executor einen Fehler feststellt und nicht ausgetauscht werden kann, weil keine Ressourcen mehr vorhanden sind. Ein weiterer Grund, nicht alle Ressourcen

voll auszulasten ist der Ausfall eines Knoten. Wenn ein Knoten ausfällt, dann fallen mit ihm auch alle darauf laufenden Executoren aus. Wenn noch genügend Ressourcen vorhanden sind, werden die verlorenen Executoren auf anderen Knoten neu erstellt und die Anfrage kann ohne größere Verzögerungen fortgesetzt werden. Lediglich die Tasks, die gerade in Arbeit waren, müssen neu berechnet werden.
In diesem Zusammenhang wurde auch untersucht, wie die Executoren bei Teilauslastung über die Knoten verteilt werden. Es zeigte sich, dass die angeforderten Executoren auf so viele Knoten wie möglich verteilt werden, sodass die Ressourcenauslastung der einzelnen Knoten so gering wie möglich gehalten wird.

3.3.6. Einfluss der Clusterkonfiguration

Nachdem die wichtigsten Einstellungen und Grenzen von Spark untersucht wurden, stellt sich noch die Frage, wie sich die Ausführzeiten der Jobs genau verhalten, wenn sich die Clustergröße verändert. Dafür wurde dem Cluster ein weiterer Knoten hinzugefügt, der als Ressourcen 4 Cores und 8 Gigabyte Arbeitsspeicher zu Verfügung stellt. Wenn der Cluster mit einem Knoten erweitert wird, der eine andere Ressourcen-Kapazität aufweist, als die übrigen Knoten, dann müssen die Konfigurationseinstellungen des HDFS und des YARN angepasst werden.
Da die Executoren nur maximal so viele Ressourcen zugewiesen bekommen können, wie der YARN-Container an Kapazitäten bereitstellt, kann die Ressourcenzuweisung 5 Cores und 5 Gigabyte Arbeitsspeicher in diesen Tests nicht verwendet werden. Aus diesem Grund werden die folgenden Tests, sowohl der Analyse als auch des Imports, mit der Zuweisung 1 Core und 1 Gigabyte durchgeführt. Im Gegensatz zu den vorherigen Tests, wo maximal 12 Executoren verwendet wurden, werden in diesen Untersuchungen 14 Executoren verwendet. Im Folgenden befinden sich die gegenübergestellten Ergebnisse der alten und neuen Testreihe.

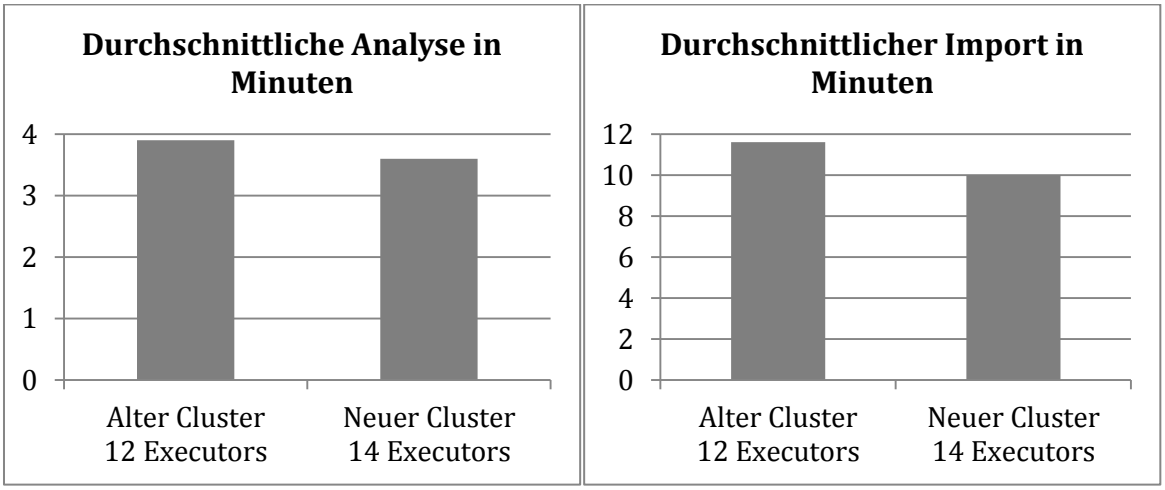

Abbildung 11: Durchschnittliche Analyse- und Importzeit im Vergleich mit einem vergrößerten Cluster

In Abbildung 11 ist zu erkennen, dass sich die Ausführzeiten des Imports und der Analyse wie erwartet verhalten. In beiden Fällen sank die Ausführzeit nach der Erhöhung der Anzahl der Executoren.

Das gleiche Bild ergibt sich, wenn die einzelnen Analyseanfragen separat betrachtet werden. Wie in Abbildung 12 zu sehen, benötigt der alte Cluster mit den 12 Executoren mehr Zeit, als der neue Cluster auf dem mehr Executoren laufen. Lediglich bei Anfragen, die nur zwei bis drei Sekunden benötigen, sind die Unterschiede nicht zu erkennen.

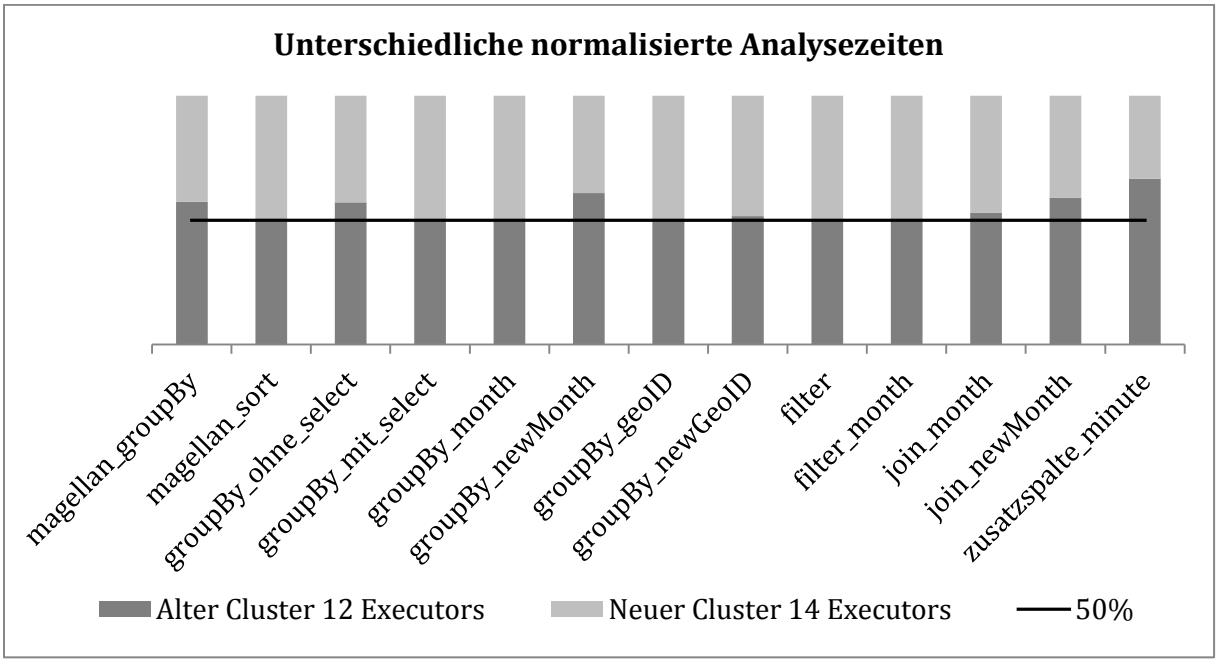

Abbildung 12: Normalisierte Analysezeiten des alten und neuen Clusters

4. Fazit und Schluss

Zum Abschluss folgen eine Zusammenfassung der behandelten Themen und das Fazit zu den Untersuchungsergebnissen. Zuletzt wird noch ein Ausblick gegeben, welche Untersuchungen sich anschließen könnten und welcher Nutzen aus den Testreihen gezogen werden kann.

4.1. Zusammenfassung

In dieser Arbeit wurden die Konfigurationen und Grenzen von Spark, sowie zwei unterschiedliche Dateiformate untersucht. Die Testreihe sollte die optimalen Einstellungen und das am besten geeignete Dateiformat für den zugrundeliegenden Praxisfall erarbeiten.
In Kapitel 2 wurde dafür eine Einführung gegeben, was Apache Spark ist, wie es arbeitet und welche Komponenten für die Arbeit und die Optimierung interessant sind. In diesem Zusammenhang wurde auch geklärt, wo bei der Optimierung angesetzt wird. Im Anschluss an die Einführung wurden die beiden Dateiformate ORC und Parquet vorgestellt.
In Kapitel 3 wurde die praktische Durchführung beschrieben. Diese gliederte sich in zwei große Bereiche. Im ersten Teil wurden die Testtabellen und der gesamte Testaufbau vorgestellt. Dazu gehörten die Beschreibung der unterschiedlichen Einstellungen, die getestet werden sollten und die unterschiedlichen Analyseanfragen. Im zweiten Teil wurden dann die Versuchsdurchführung und die Auswertung beschrieben. Dabei wurden vereinzelte Tests nach Teilauswertungen noch vertieft oder leicht abgeändert, um genauere Ergebnisse zu erhalten. Anhand der Untersuchungen sind die besten Konfigurationseinstellungen und das beste Dateiformat herausgefunden worden.

4.2. Fazit

Ziel dieser Arbeit war es, die beste Konfigurationseinstellung von Apache Spark für den behandelten Praxiseinsatz herauszufinden und ein geeignetes Speicherformat zu finden.

Eine der grundlegenden Fragen war, ob als Serialisierung Java oder Kryo genutzt werden sollte. Aufgrund des Ergebnisses der Entscheidungsmatrizen kann eindeutig gesagt werden, dass die Kryo-Serialisierung gewählt werden sollte. Als schnellste Variante der Datenspeicherung hat sich das Dateiformat Parquet mit der Snappy-Komprimierung herausgestellt. Für die Analyse der Daten wird die Executor-Zuweisung mit 5 Cores und 5 Gigabyte Arbeitsspeicher empfohlen. Beim Import sollte die Executor-Zuweisung „1 Core und 1 Gigabyte Arbeitsspeicher“ genutzt werden.

Wie die Tests des Imports mit Append-Einstellung gezeigt haben, ist es nicht aufwändiger, bestehende Datenbestände zu erweitern, als immer wieder neue Tabellen anzulegen. Ebenso ist es von Vorteil, Informationen, die häufig zusätzlich gebraucht werden, mit zu importieren. Dieses bietet einen Geschwindigkeitsvorteil bei den Analysen und nur geringe Zeitverluste beim Importieren.

Beim Vergleich des FAIR-Mode mit dem FIFO-Mode hat sich kein Vorteil für einen der beiden dargestellt. Beide Modi verhalten sich nicht so, wie es anhand der Namen erwartet werden würde. Der FAIR-Mode beachtet die Priorität nicht und der FIFO-Mode arbeitet die Jobs nicht nacheinander ab. Auch bei den Geschwindigkeiten gab es keine Unterschiede. Es konnte lediglich das Ergebnis der vorher untersuchten Entscheidungsmatrix bestätigt werden, wonach die dynamische Zuweisung langsamer ist als die manuelle.

Dafür wurden einige Optimierungen bei der Analyse gefunden. Zum einen konnte gezeigt werden, dass ein Select zu Beginn einer Anfrage die Geschwindigkeit erhöhen kann und zum anderen, dass auf den DataFrames das GroupBy deutlich schneller ist als das ReduceBy. Der ursprünglich geplante Vergleich mit dem partitionierten Wert konnte nicht durchgeführt werden, da sich im Verlauf der Untersuchungen rausstellte, dass „GroupBy“ und „join“ die Vorteile einer Partitionierung nicht nutzen können. Lediglich in Szenarien, in denen in jeder Anfrage nach dem partitionierten Wert gefiltert wird, ist eine Partitionierung sinnvoll. In allen anderen Szenarien verlangsamt die Partitionierung sowohl den Import, als auch die Analyse deutlich.

Die Analyse der Geodaten mittels eines Shapefiles hat gut funktioniert. Das Shapefile konnte unkompliziert eingelesen werden. Die Anfrageformulierung von Magellan ließ sich gut in die Analyseanfrage integrieren und durch Spark-Elemente erweitern.

Auch die Geoanalyse über die Geo-ID hat funktioniert, brauchte aber bei der Generierung während der Laufzeit deutlich mehr Zeit. Aus diesem Grund sollte diese Geoanalyse-Variante nur auf importierte Geo-ID-Werte angewendet werden. Der Vorteil der Geoanalyse über die „geo_id“ liegt darin, dass keine Shapefiles benötigt werden, um Informationen einer bestimmten Region

zu bekommen. Der Nachteil ist, dass nicht so präzise festgelegt werden kann, welche Gegend von Interesse ist. Die Rastergröße wird beim Import festgelegt und kann, aufgrund der langen Ausführungszeit bei der Neugenerierung zur Laufzeit, nachträglich nur schlecht geändert werden.

In der Server bzw. der HDFS-Einstellung sollte der Parameter „dfs.client.block.write.replace-datanode-on-failure.enable“ bei kleinen Servern konfiguriert werden, um die zeitlichen Verzögerungen durch die Suche nach neuen Knoten zu verhindern. Darüber hinaus sollte darauf geachtet werden, dass die Anzahl an Knoten höher ist, als die Anzahl der Replikationen beim Importieren. Dieses war in dem verwendeten Testsystem nicht der Fall, was häufig zu Verzögerungen und Fehlermeldungen hauptsächlich beim Import geführt hat.
Des Weiteren ist es wichtig, dass der Server ausreichend Ressourcen hat, damit die ganze Zeit über Reserven zur Verfügung stehen. Das beugt größeren Zeitverlusten beim Ausfall von Knoten vor und ermöglicht es, im Falle eines Fehlers einfach auf neue Ressourcen zuzugreifen. Bei den Ressourcen ist noch wichtig zu bedenken, dass über die vorhandenen Kapazitäten zwar die Geschwindigkeit des Gesamtsystems beeinflusst wird, jedoch nicht das Verhältnis der einzelnen Geschwindigkeiten zueinander.
Eine weitere wichtige Fragestellung in der Arbeit ist das Austesten der Grenzen von Apache Spark. Es ist deutlich geworden, dass die Anzahl der Cores zwar maßgeblich an der Geschwindigkeit beteiligt ist, aber keine Probleme bereitet, wenn ihre Anzahl zu gering ist. Ganz anders sieht das beim Arbeitsspeicher aus. Wenn dem Executor nicht genügend Arbeitsspeicher mitgegeben wird, wird eine Java-Heap-Size-Fehlermeldung ausgegeben und der Job bricht ab. Vor dem Hintergrund, dass einem Executor maximal so viele Ressourcen mitgegeben werden können, wie sich in einem YARN-Container auf einem Knoten befinden, sollten die Ressourcen auf den Knoten nicht zu klein dimensioniert werden.

4.3. Ausblick

Die Testreihen dieser Arbeit führten zu dem Ergebnis, dass Parquet das geeignetere Dateiformat ist. Allerdings befindet sich das ORC-Format noch in der Entwicklung. Viele Optimierungen, die Parquet bereits enthält, fehlen dem ORC-Format noch. Dazu gehört unter anderem die Vektorisierung, die eine deutlich effizientere Nutzung des Arbeitsspeichers bewirkt und damit wichtig für die Geschwindigkeit ist. Des Weiteren ist es zum aktuellen Zeitpunkt noch nicht möglich, die mitgegebenen Metadaten vom ORC-Format abzufragen.

In (Foundation, 2017) befinden sich alle Punkte, die für die ORC-Entwicklung noch in Arbeit sind. Aufgrund der unvollständigen Entwicklung zum jetzigen Zeitpunkt sollte das Format nach Beendigung der Entwicklung erneut getestet werden. Eventuell wird dann ORC und nicht mehr Parquet das schnellere Format sein.

Trotz des unvollständig entwickelten Dateiformates, war es möglich, eine Übersicht zu geben, welche Einstellungen für den behandelten Praxisfall gewählt werden sollten. Darüber hinaus ist deutlich geworden, wie die Komponenten in Grenzfällen reagieren, wodurch der Aufbau des Clusters besser zu kalkulieren ist. Zu diesem Bereich sollten sich trotzdem noch weitere Untersuchungen anschließen, da der Eindruck entstand, dass nicht nur die Ressourcenkapazitäten, sondern auch die verwendete Hardware eine Rolle bei der Ausführgeschwindigkeit der Jobs spielt.

Von den ermittelten Ergebnissen können auch andere Szenarien profitieren, in denen eine hohe Analysegeschwindigkeit gefordert wird und die Dateien in verhältnismäßig wenig Spalten organisiert sind. Bei der Übertragung auf andere Szenarien sollte beachtet werden, dass sich das Ergebnis auf die Verwendung von Spark in Kombination mit dem HDFS und YARN bezieht. Soll ein anderes Speichersystem verwendet werden, sind die Ergebnisse nicht übertragbar. Das gleiche gilt auch, wenn ein anderes Dateiformat gewählt wird. Bei Verwendung des beschriebenen Speichersystems und Dateiformates ist es jedoch möglich, die Ergebnisse auch auf andere Szenarien zu übertragen.

A. Anhang

A.1. Kriterienkatalog: Hauptauswahl

		Java											
		orc - snappy & partition			orc - partition			orc - snappy			orc - ohne		
Kriterien	Gewichtung	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert
magellan_groupBy [sec]	6	7	42	54	2	12	115	6	36	58	5	30	81
magellan_sort [sec]	6	6	36	120	2	12	211	7	42	109	4	24	163
groupBy_ohne_select [sec]	6	8	48	14	9	54	11	10	60	10	11	66	9
groupBy_mit_select [sec]	6	12	72	10	12	72	10	12	72	10	13	78	9
groupBy_month [sec]	6	9	54	9	13	78	5	10	60	8	10	60	8
groupBy_newMonth [sec]	6	13	78	12	8	48	19	15	90	10	12	72	13
groupBy_geoID [sec]	6	11	66	8	8	48	12	12	72	7	10	60	9
groupBy_newGeoID [sec]	6	1	6	2862	6	36	993	4	24	1001	3	18	1033
filter [sec]	6	11	66	11	6	36	24	10	60	12	8	48	18
filter_month [sec]	6	15	90	2	16	96	1	12	72	10	13	78	9
join_month [sec]	6	10	60	15	7	42	44	10	60	15	8	48	31
join_newMonth [sec]	6	13	78	27	5	30	54	15	90	22	12	72	30
zusatzspalte_minute [sec]	6	13	78	8	8	48	29	12	72	9	11	66	13
Importgeschwindigkeit [min]	39	4	156	21	5	195	20.7	9	351	18.2	12	468	17.5
Dateigröße [GB]	13	15	195	4	9	117	32.8	16	208	2.1	10	130	30
SUMME			1125			924			1369			1318	

		Java											
		parquet - snappy & partition			parquet - partition			parquet - snappy			parquet - ohne		
Kriterien	Gewichtung	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert
magellan_groupBy [sec]	6	14	84	27	10	60	34	11	66	33	15	90	25
magellan_sort [sec]	6	11	66	75	12	72	71	14	84	62	16	96	54
groupBy_ohne_select [sec]	6	13	78	4	15	90	2	15	90	2	15	90	2
groupBy_mit_select [sec]	6	15	90	2	15	90	2	16	96	1	16	96	1
groupBy_month [sec]	6	16	96	1	16	96	1	16	96	1	16	96	1
groupBy_newMonth [sec]	6	11	66	15	6	36	25	14	84	11	7	42	21
groupBy_geoID [sec]	6	14	84	3	14	84	3	15	90	2	15	90	2
groupBy_newGeoID [sec]	6	14	84	793	15	90	782	9	54	862	12	72	809
filter [sec]	6	15	90	3	14	84	5	15	90	3	16	96	2
filter_month [sec]	6	14	84	3	16	96	1	16	96	1	16	96	1
join_month [sec]	6	15	90	7	13	78	11	16	96	6	16	96	6
join_newMonth [sec]	6	11	66	32	4	24	62	14	84	24	10	60	33
zusatzspalte_minute [sec]	6	15	90	2	14	84	3	16	96	1	16	96	1
Importgeschwindigkeit [min]	39	4	156	21	6	234	20.2	13	507	17.2	14	546	16.5
Dateigröße [GB]	13	13	169	5.4	11	143	11.8	14	182	5	12	156	8.4
SUMME			1393			1361			1811			1818	

Tabelle 13: Kriterienkatalog: Hauptauswahl

		Kryo											
		orc - snappy & partition			orc - partition			orc - snappy			orc - ohne		
Kriterien	Gewichtung	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert
magellan_groupBy [sec]	6	8	48	45	3	18	97	9	54	35	4	24	88
magellan_sort [sec]	6	10	60	77	3	18	179	8	48	95	5	30	148
groupBy_ohne_select [sec]	6	10	60	10	10	60	10	11	66	9	12	72	8
groupBy_mit_select [sec]	6	14	84	8	14	84	8	13	78	9	14	84	8
groupBy_month [sec]	6	13	78	5	14	84	4	12	72	6	11	66	7
groupBy_newMonth [sec]	6	14	84	11	10	60	19	16	96	8	15	90	10
groupBy_geoID [sec]	6	13	78	6	9	54	11	13	78	6	12	72	7
groupBy_newGeoID [sec]	6	7	42	977	5	30	996	2	12	1043	8	48	974
filter [sec]	6	12	72	10	7	42	22	11	66	11	9	54	16
filter_month [sec]	6	15	90	2	16	96	1	11	66	11	10	60	14
join_month [sec]	6	12	72	13	9	54	30	11	66	14	8	48	31
join_newMonth [sec]	6	14	84	25	7	42	43	16	96	21	12	72	30
zusatzspalte_minute [sec]	6	13	78	8	9	54	20	12	72	9	10	60	15
Importgeschwindigkeit [min]	39	3	117	21.3	5	195	20.7	10	390	17.9	11	429	17.7
Dateigröße [GB]	13	15	195	4	9	117	32.8	16	208	2.1	10	130	30
SUMME		1242			1008			1468			1339		

		Kryo											
		parquet - snappy & partition			parquet - partition			parquet - snappy			parquet - ohne		
Kriterien	Gewichtung	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert
magellan_groupBy [sec]	6	12	72	30	13	78	28	14	84	27	16	96	24
magellan_sort [sec]	6	9	54	80	13	78	70	15	90	61	16	96	54
groupBy_ohne_select [sec]	6	13	78	4	14	84	3	15	90	2	16	96	1
groupBy_mit_select [sec]	6	15	90	2	15	90	2	16	96	1	16	96	1
groupBy_month [sec]	6	15	90	2	15	90	2	16	96	1	16	96	1
groupBy_newMonth [sec]	6	10	60	16	5	30	27	14	84	11	9	54	17
groupBy_geoID [sec]	6	15	90	2	14	84	3	16	96	1	16	96	1
groupBy_newGeoID [sec]	6	10	60	813	16	96	776	13	78	796	11	66	812
filter [sec]	6	15	90	3	13	78	6	15	90	3	15	90	3
filter_month [sec]	6	14	84	3	16	96	1	16	96	1	16	96	1
join_month [sec]	6	14	84	8	14	84	8	15	90	7	15	90	7
join_newMonth [sec]	6	9	54	34	6	36	46	14	84	25	8	48	41
zusatzspalte_minute [sec]	6	16	96	1	14	84	3	16	96	1	16	96	1
Importgeschwindigkeit [min]	39	7	273	20	8	312	19.2	15	585	16.3	16	624	15.8
Dateigröße [GB]	13	13	169	5.4	11	143	11.8	14	182	5	12	156	8.4
SUMME		1444			1463			1937			1896		

A.2. Kriterienkatalog: Executoren für die Analyse

		Core:1, Memory:1GB, Anzahl:12			Core:3, Memory:1GB, Anzahl:4			Core:3, Memory:4GB, Anzahl:4			Core:6, Memory:4GB, Anzahl:2		
Kriterien	Gewichtung	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert
magellan_groupBy [sec]	1	5	5	39	6	6	35	7	7	32	8	8	30
magellan_sort [sec]	1	5	5	64	4	4	68	6	6	63	5	5	64
groupBy_ohne_select [sec]	1	6	6	4	8	8	2	7	7	3	8	8	2
groupBy_mit_select [sec]	1	7	7	2	7	7	2	6	6	3	8	8	1
groupBy_month [sec]	1	7	7	3	8	8	2	7	7	3	8	8	2
groupBy_newMonth [sec]	1	5	5	14	6	6	12	5	5	14	8	8	10
groupBy_geoID [sec]	1	6	6	4	8	8	2	7	7	3	8	8	2
groupBy_newGeoID [sec]	1	8	8	295	5	5	768	6	6	736	3	3	1278
filter [sec]	1	7	7	3	7	7	3	7	7	3	8	8	2
filter_month [sec]	1	8	8	1	8	8	1	8	8	1	8	8	1
join_month [sec]	1	3	3	9	6	6	6	5	5	7	8	8	4
join_newMonth [sec]	1	4	4	33	7	7	28	3	3	34	8	8	24
zusatzspalte_minute [sec]	1	7	7	2	7	7	2	7	7	2	8	8	1
SUMME			78			87			81			96	

		Core:3, Memory:10GB, Anzahl:4			Core:6, Memory:10GB, Anzahl:2			Core:6, Memory:14GB, Anzahl:2			dynamische Zuweisung		
Kriterien	Gewichtung	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert
magellan_groupBy [sec]	1	4	4	40	6	6	35	5	5	39	3	3	47
magellan_sort [sec]	1	2	2	75	8	8	60	3	3	74	7	7	62
groupBy_ohne_select [sec]	1	6	6	4	6	6	4	7	7	3	7	7	3
groupBy_mit_select [sec]	1	7	7	2	7	7	2	6	6	3	7	7	2
groupBy_month [sec]	1	6	6	4	8	8	2	8	8	2	7	7	3
groupBy_newMonth [sec]	1	7	7	11	7	7	11	5	5	14	7	7	11
groupBy_geoID [sec]	1	7	7	3	8	8	2	7	7	3	6	6	4
groupBy_newGeoID [sec]	1	4	4	937	2	2	1287	1	1	1349	7	7	333
filter [sec]	1	7	7	3	8	8	2	6	6	4	5	5	12
filter_month [sec]	1	8	8	1	8	8	1	8	8	1	7	7	4
join_month [sec]	1	4	4	8	7	7	5	2	2	10	1	1	16
join_newMonth [sec]	1	5	5	30	8	8	24	6	6	29	2	2	83
zusatzspalte_minute [sec]	1	7	7	2	7	7	2	6	6	3	5	5	7
SUMME			74			90			70			71	

Tabelle 14: Kriterienkatalog: Executoren für die Analyse

A.3. Kriterienkatalog: Executoren für den Import

		Core:1, Memory:1GB, Anzahl:12			Core:3, Memory:1GB, Anzahl:4			Core:3, Memory:4GB, Anzahl:4			Core:6, Memory:4GB, Anzahl:2		
Kriterien	Gewichtung	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert
Importgeschwindigkeit [min]	1	8	8	11.6	5	5	25	6	6	24.5	1	1	43.8
SUMME			8			5			6			1	

		Core:3, Memory:10GB, Anzahl:4			Core:6, Memory:10GB, Anzahl:2			Core:6, Memory:14GB, Anzahl:2			dynamische Zuweisung		
Kriterien	Gewichtung	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert	Platz	Bewertung	Messwert
Importgeschwindigkeit [min]	1	4	4	30.1	3	3	43.4	2	2	43.5	7	7	12.2
SUMME			4			3			2			7	

Tabelle 15: Kriterienkatalog: Executoren für den Import

A.4. Kriterienkatalog: Neue Executor-Zuweisungen für die Analyse

Analyse	Core: 5 Anzahl: 2															
		Memory: 3GB				Memory: 4GB				Memory: 5GB				Memory. 6GB		
Kriterien	Gewichtung	Platz	Bewertung	Messwert		Platz	Bewertung	Messwert		Platz	Bewertung	Messwert		Platz	Bewertung	Messwert
magellan_groupBy [sec]	1	6	6	24		7	7	23		7	7	23		4	4	26
magellan_sort [sec]	1	7	7	57		6	6	58		7	7	57		8	8	55
groupBy_ohne_select [sec]	1	8	8	2		8	8	2		8	8	2		8	8	2
groupBy_mit_select [sec]	1	8	8	1		8	8	1		8	8	1		8	8	1
groupBy_month [sec]	1	7	7	2		8	8	1		8	8	1		8	8	1
groupBy_newMonth [sec]	1	7	7	6		6	6	7		6	6	7		7	7	6
groupBy_geoID [sec]	1	8	8	2		8	8	2		8	8	2		8	8	2
groupBy_newGeoID [sec]	1	6	6	1236		3	3	1254		4	4	1249		7	7	1233
filter [sec]	1	8	8	1		8	8	1		8	8	1		8	8	1
filter_month [sec]	1	8	8	1		8	8	1		8	8	1		8	8	1
join_month [sec]	1	7	7	4		7	7	4		8	8	3		8	8	3
join_newMonth [sec]	1	5	5	15		8	8	11		8	8	11		4	4	17
zusatzspalte_minute [sec]	1	8	8	1		8	8	1		8	8	1		8	8	1
SUMME			93				93				96				94	

	Core: 6 Anzahl: 2															
		Memory: 3GB				Memory: 4GB				Memory: 5GB				Memory: 6GB		
Kriterien	Gewichtung	Platz	Bewertung	Messwert		Platz	Bewertung	Messwert		Platz	Bewertung	Messwert		Platz	Bewertung	Messwert
magellan_groupBy [sec]	1	7	7	23		3	3	30		8	8	22		5	5	25
magellan_sort [sec]	1	5	5	60		4	4	64		6	6	58		7	7	57
groupBy_ohne_select [sec]	1	8	8	2		8	8	2		8	8	2		8	8	2
groupBy_mit_select [sec]	1	8	8	1		8	8	1		8	8	1		8	8	1
groupBy_month [sec]	1	8	8	1		7	7	2		7	7	2		8	8	1
groupBy_newMonth [sec]	1	6	6	7		5	5	10		6	6	7		8	8	4
groupBy_geoID [sec]	1	8	8	2		8	8	2		8	8	2		8	8	2
groupBy_newGeoID [sec]	1	5	5	1248		2	2	1278		2	2	1278		8	8	1224
filter [sec]	1	8	8	1		7	7	2		8	8	1		8	8	1
filter_month [sec]	1	8	8	1		8	8	1		8	8	1		8	8	1
join_month [sec]	1	8	8	3		7	7	4		8	8	3		8	8	3
join_newMonth [sec]	1	6	6	14		3	3	24		7	7	12		4	4	17
zusatzspalte_minute [sec]	1	8	8	1		8	8	1		8	8	1		8	8	1
SUMME			93				78				92				96	

Tabelle 16: Kriterienkatalog: Neue Executor-Zuweisungen für die Analyse

A.5. Kriterienkatalog: Neue Executor-Zuweisungen für den Import

Import		Core: 1 Anzahl: 12										Core: 2 Anzahl: 6									
		Memory: 1GB				Memory: 2GB				Memory: 3GB			Memory: 1GB				Memory: 2GB			Memory: 3GB	
Kriterien	Gewichtung	Platz	Bewertung	Messwert		Platz	Bewertung	Messwert		Platz	Bewertung	Messwert	Platz	Bewertung	Messwert		Platz	Bewertung	Messwert	Platz	Bewertung	Messwert
Import [min]	1	6	6	11.6		4	4	12.5		5	5	12.4	1	1	19		3	3	17.6	2	2	18.6
SUMME			6				4				5			1				3			2	

Tabelle 17: Kriterienkatalog: Neue Executor-Zuweisungen für den Import

A.6. FAIR-Mode- und FIFO-Mode-Testergebnis

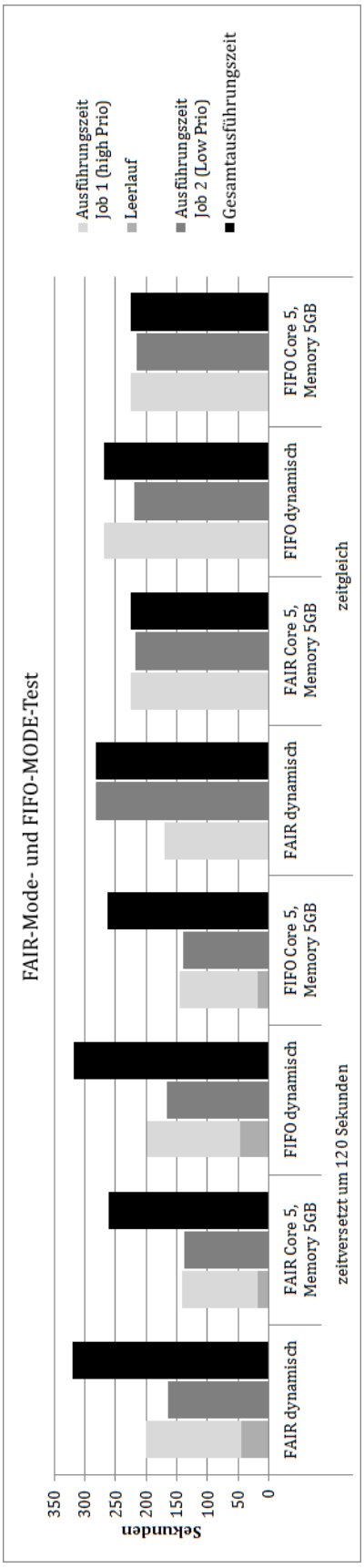

Abbildung 13: FAIR- und FIFO-Mode-Testergebnis

A.7. Algorithmus für Geo-ID

```
class GeoId(var gridSize: Double){
  var maxX:Int = 360 //-180° to +180°
  var maxY:Int = 180 //-90° to +90°
  //Nullpunkt im Koordinatensystem bei -180 und -90 bis +180 und +90

  def getGeoId(lat: Double, lon: Double): Long = {
    var result:Long = -1
    var latitude:Double = Math.round(lat*10)
        latitude = latitude / 10
        var longitude:Double = Math.round(lon*10)
        longitude = longitude / 10
        println("Lat , Long: "+latitude+", "+longitude)
        if((latitude<=-90.1)||(latitude>=90.1)||(longitude<=-180.1) ||(longitude>=180.1)){
          println("Bitte Werte im Wertebereich angeben:\n
                       latitude: -90 bis +90\n
                       longitude: -180 bis +180")
        }else{
          if(longitude == -180.0){
            longitude = longitude + maxX //wenn -180 dann auf +180 setzen
          }
          var positionX:Double = Math.round(((maxX/2)+longitude)*100)
          positionX = positionX / 100
          var positionY:Double = Math.round(((maxY/2)+latitude )*100)
          positionY = positionY / 100
          //position der latitude und longitude Werte in diesem Koordinatensystem
          println("positionX, positionY: "+positionX+", "+positionY)
          var xCoordinateGrid:Int = Math.ceil(positionX/gridSize).toInt
          var yCoordinateGrid:Int = 1
          if(latitude != -90.0){
            yCoordinateGrid = Math.ceil(positionY/gridSize).toInt
          }
          //Koordinate des Feldes im x,y Koordinatensystems
          println("xCoordinateGrid, yCoordinateGrid "+xCoordinateGrid+", "+yCoordinateGrid)
          //errechnen wie lang die x Achse ist in abhänggkeit von der Feldgröße
          var fieldsPerLine:Int = (maxX/gridSize).toInt
          //errechnen der Nummer des Feldes
          result = (yCoordinateGrid -1) *fieldsPerLine + xCoordinateGrid
          //linksunten ist eins rechts daneben ist zwei ....   immer von links nach rechts
        }
        result
  }
}
```

A.8. fairscheduler.xml

```
<?xml version="1.0"?>
<allocations>
	<pool name="highPrio">
		<schedulingMode>FAIR</schedulingMode>
		<weight>1000</weight>
		<minShare>0</minShare>
	</pool>
	<pool name="lowPrio">
		<schedulingMode>FAIR</schedulingMode>
		<weight>1</weight>
		<minShare>0</minShare>
	</pool>
	<pool name="normal">
		<schedulingMode>FIFO</schedulingMode>
		<weight>1</weight>
		<minShare>0</minShare>
	</pool>
</allocations>
```

A.9. CD

In der folgenden Tabelle 18 die Inhalte der beigefügten CD.

Ordnername	Beschreibung
Bash	Die Bash-Datei
Scala	Die wichtigsten Scala-Dateien für die Anfragen
Java	Das Programm zum Generieren der Testtabellen
Excel	Die Testergebnisse
XML	Die „fairscheduler.xml“-Datei

Tabelle 18: Inhalt der beigefügten CD

Literaturverzeichnis

Apache Software Foundation. (2014). *Apache Parquet Documentation*. Abgerufen am 12. 09. 2017 von Apache Parquet Documentation: https://parquet.apache.org/documentation/latest/

Apache Software Foundation. (2014). *hdfs.default.xml*. Abgerufen am 10. 10. 2017 von Hadoop Dokumentation: https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml

Apache Software Foundation. (2016). *Cluster Mode Overview*. Abgerufen am 13. 09. 2017 von Apache Spark Dokumentation: https://spark.apache.org/docs/2.1.0/cluster-overview.html

Apache Software Foundation. (2016). *Job Scheduling*. Abgerufen am 2. 10. 2017 von Apache Spark Dokumentation: https://spark.apache.org/docs/2.1.0/job-scheduling.html

Apache Software Foundation. (2016). *Spark Configuration - Application Properties*. Abgerufen am 25. 10. 2017 von Apache Spark Documentation: https://spark.apache.org/docs/2.1.0/configuration.html

Apache Software Foundation. (2016). *Spark SQL, DataFrames and Datasets Guide*. Abgerufen am 13. 09. 2017 von Apache Spark Dokumentation: http://spark.apache.org/docs/2.1.0/sql-programming-guide.html

Apache Software Foundation. (2017). *Apache ORC Documentation*. Abgerufen am 12. 09. 2017 von Apache ORC Documentation: https://orc.apache.org/docs/

Data Flair Support. (2016). *Spark RDD Operations-Transformation & Action with Example*. Abgerufen am 14. 09. 2017 von Data Flair: http://data-flair.training/blogs/spark-rdd-operations-transformations-actions/

Databricks. (2014). *Avoid GroupByKey*. Abgerufen am 25. 09. 2017 von databricks.gitbooks.io: https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html

Foley, M. (10. 2 2016). *Write or Append failures in very small Clusters*. Abgerufen am 9. 10. 2017 von Hortonworks.com: https://community.hortonworks.com/articles/16144/write-or-append-failures-in-very-small-clusters-un.html

Foundation, A. S. (2017). *Spark Jira*. Abgerufen am 10. 11. 2017 von Feature parity for ORC with Parquet: https://issues.apache.org/jira/browse/SPARK-20901

Freiknecht, J. (2014). HDFS - das Hadoop Distributed File System. In J. Freiknecht, *Big Data in der Praxis* (S. 21). Hanser. ISBN:978-3-446-43959-7

Ho, R. (2015). *Big Data Processing in Spark - Eintrag vom 22.02.15*. Abgerufen am 14. 09. 2017 von horicky Blog: http://horicky.blogspot.de/

Karau, H., & Warren, R. (2017). A Few Large Excecutors or Many Small Executoren? In H. Karau, & R. Warren, *High Performance Spark - Best Practices For Scaling & Optimizing Apache Spark* (S. 280-282). O´Reilly. ISBN:978-1-491-94320-5

Karau, H., & Warren, R. (2017). How Spark Works. In H. Karau, & R. Warren, *High Performance Spark - Best Practices For Scaling & Optimizing Apache Spark* (S. 7 - 8). O'Reilly. ISBN:978-1-491-94320-5

Karau, H., & Warren, R. (2017). Immutability and the RDD Interface. In H. Karau, & R. Warren, *High Performance Spark - Best Practices For Scaling & Optimizing Apache Spark* (S. 14). O'Reilly. ISBN:978-1-491-94320-5

Karau, H., & Warren, R. (2017). Parquet. In H. Karau, & R. Warren, *High Performance Spark - Best Practices For Scaling & Optimizing Apache Spark* (S. 55). O'Reilly. ISBN:978-1-491-94320-5

Karau, H., & Warren, R. (2017). Partitions (Discovery and Writing). In H. Karau, & R. Warren, *High Performance Spark - Best Practices For Scaling & Optimizing Apache Spark* (S. 62). O'Reilly. ISBN:978-1-491-94320-5

Karau, H., & Warren, R. (2017). The Anatomy of a Spark Job. In H. Karau, & R. Warren, *High Performance Spark - Best Practices For Scaling & Optimizing Apache Spark* (S. 22 - 23). O'Reilly. ISBN:978-1-491-94320-5

Karau, H., & Warren, R. (2017). Wide Versus Narow Dependencies. In H. Karau, & R. Warren, *High Performance Spark - Best Practices For Scaling & Optimizing Apache Spark* (S. 17 - 19). O'Reilly. ISBN:978-1-491-94320-5

Melnik, S., Gubarev, A., Long, J., Romer, G., Shivakumar, S., Tolton, M., & Vassilakis, T. (2010). *Dremel: Interactive Analysis of Web-Scale Datasets*. Abgerufen am 12. 09. 2017 von Research at Google: https://research.google.com/pubs/pub36632.html

Sriharsha, R. (2017). *Geo Spatial Data Analytics on Spark*. Abgerufen am 14. 09. 2017 von GitHub: https://github.com/harsha2010/magellan

stackoverflow. (2012). *How to profile methods in Scala?* Abgerufen am 6. 11. 2017 von stackoverflow: https://stackoverflow.com/questions/9160001/how-to-profile-methods-in-scala

White, T. (2015). Chapter 13. Parquet. In T. White, *Hadoop The Definitive Guide - Storage and Analysis at Internet Scale* (S. 367 - 372). O'Reilly. ISBN:978-1-491-90163-2

Yu, S., & Guo, S. (2016). Table 2.1 Comparative study. In S. Yu, & S. Guo, *Big Data Concepts, Theories, and Applications* (S. 38). Springer. ISBN:978-3-319-27761-5

Zakordonets, A. (2017). *easily measuring code execution time in scala*. Abgerufen am 6. 11. 2017 von biercoff: http://biercoff.com/easily-measuring-code-execution-time-in-scala/

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Ort, Datum Alina Böttcher