Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Daniel Kirchner

Skalierbare Datenanalyse mit Apache Spark

Architekturanalyse und Implementation eines realitätsnahen Anwendungsfalls

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Daniel Kirchner

Skalierbare Datenanalyse mit Apache Spark

Architekturanalyse und Implementation eines realitätsnahen Anwendungsfalls

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

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

Eingereicht am: 1. Januar 2345

Daniel Kirchner

Thema der Arbeit
Skalierbare Datenanalyse mit Apache Spark Architekturanalyse und Implementation eines realitätsnahen Anwendungsfalls

Stichworte
Apache Spark, Big Data, Architekturanalyse, Text Mining, Echtzeit-Datenanalyse, Raspberry Pi

Kurzzusammenfassung
Apache Spark ist auf dem Weg sich als zentrale Komponente von Big-Data-Analyse-Systemen für eine Vielzahl von Anwendungsfällen durchzusetzen. Diese Arbeit schafft einen Überblick der zentralen Konzepte und Bestandteile von Apache Spark und demonstriert in der Implementation eines Anwendungsfalles den praktischen Einsatz dieser Konzepte. Bei dem Anwendungsfall liegt der Schwerpunkt auf der Integration unterschiedlicher Laufzeitparadigmen (Batch und Streaming) innerhalb einer Sparkanwendung.
Daniel Kirchner

Title of the paper
Scalable Data Analysis with Apache Spark

Keywords
keyword 1, keyword 2

Abstract
This document …

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einführung

1.1 Motivation

Die Entwicklung und Verbesserung von Frameworks zur Verarbeitung großer Datenmengen ist zur Zeit hochaktuell und im Fokus von Medien und Unternehmen angekommen [BB+14]. Verschiedene Programme und Paradigmen konkurrieren um die schnellste, bequemste und stabilste Art großen Datenmengen einen geschäftsfördernden Nutzen abzuringen [SR14].

Mit dem Begriff „große Datenmengen“ oder „Big Data“ werden in dieser Arbeit solche Datenmengen zusammengefasst, die die Kriterien Volume, Velocity, Variety [Lan01] erfüllen oder „Datenmengen, die nicht mehr unter Auflage bestimmter Service Level Agreements auf einzelnen Maschinen verarbeitet werden können“ (Vgl. [SW14]).

Als Unternehmen, das früh mit zeitkritischen Aufgaben (u.a. Indizierung von Webseiten und PageRank [Pag01]) auf solchen Datenmengen konfrontiert war implementierte Google das Map-Reduce Paradigma [DG04] als Framework zur Ausnutzung vieler kostengünstiger Rechner für verschiedene Aufgaben.

In Folge der Veröffentlichung dieser Idee im Jahr 2004 wurde Map-Reduce in Form der OpenSource Implementation Hadoop (gemeinsam mit einer Implementation des Google File Systems GFS, u.a.) [GGL03] zum de-facto Standard für Big-Data-Analyseaufgaben.

Reines Map-Reduce (in der ursprünglichen Implemenation von Hadoop) als Berechnungsparadigma zur Verarbeitung großer Datenmengen zeigt jedoch in vielen Anwendungsfällen Schwächen:

- Daten, die in hoher Frequenz entstehen und schnell verarbeitet werden sollen erfordern häufiges Neustarten von Map-Reduce-Jobs. Die Folge ist kostspieliger Overhead durch Verwaltung/Scheduling der Jobs und gegebenenfalls wiederholtem Einlesen von Daten.

- Algorithmen die während ihrer Ausführung iterativ Zwischenergebnisse erzeugen und auf vorherige angewiesen sind (häufig bei Maschinenlernalgorithmen) können nur durch persistentes Speichern der Daten und wiederholtes Einlesen zwischen allen Iterationsschritten implementiert werden.
- Anfragen an ein solches Map-Reduce-System erfolgen imperativ in Form von kleinen Programmen. Dieses Verfahren ist offensichtlich nicht so intuitiv und leicht erlernbar wie deklarative Abfragesprachen klassischer Datenbanken (z.B. SQL).

In der Folge dieser Probleme entstanden viele Ansätze dieses Paradigma zu ersetzen, zu ergänzen oder durch übergeordnete Ebenen und High-Level-APIs zu vereinfachen [SR14].

Eine der Alternativen zu der klassischen Map-Reduce-Komponente in Hadoop ist die „general engine for large-scale data processing“ Apache Spark.

Ein Indiz für das steigende Interesse an diesem Produkt liefert unter anderem ein Vergleich des Interesses an Hadoop und Spark auf Google:

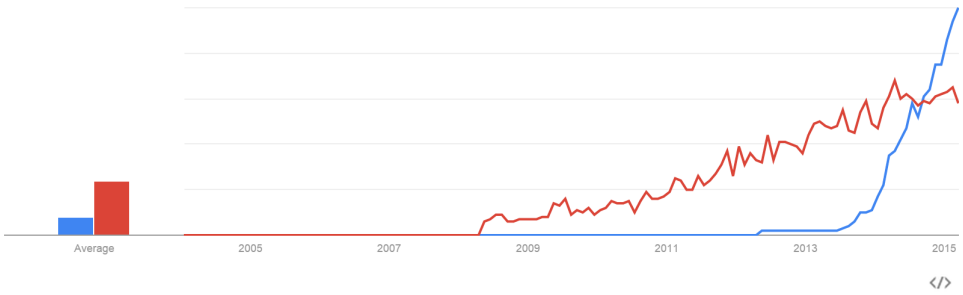

Abbildung 1.1: Suchanfragen zu „Apache Spark“ (*blau*) und „Apache Hadoop“ (*rot*), Stand 24.03.2015 [Goo]

1.2 Ziel dieser Arbeit

Das Ziel dieser Arbeit ist es die grundlegenden Konzepte und Möglichkeiten von Apache Spark zu beleuchten und ausgewählte Aspekte im Rahmen konkreter Anwendungen zu untersuchen.

Für ein tieferes Verständnis werden Installation, Cluster-Betrieb und die Modellierung und Implementation von Treiberprogrammen beispielhaft durchgeführt, dokumentiert und bewertet. Hierbei kommt Apache Spark in der Version 1.3.0 zum Einsatz.

Die Beispielimplementationen in dieser Arbeit umfassen

- eine hybride Anwendung mit einer Echtzeitkomponente (Spark Streaming Library) und einer Batch-Komponente (Spark Machine Learning Library) und
- eine atypische Sparkanwendung im Bereich des High-Performance-Computing mit besonderem Augenmerk auf die Low-Level Aspekte von Spark.

Apache Spark ist überwiegend in der Programmiersprache Scala geschrieben. Die Beispiele in dieser Arbeit werden ebenfalls in Scala verfasst um

1. einen einheitlichen Stil und Vergleichbarkeit zwischen Quellcode-Auszügen und eigenen Beispielen zu gewährleisten.
2. Ausdrücke in kurzer, prägnanter Form darzustellen.

2 Vorstellung von Apache Spark

Aus Sicht eines Nutzers ist Apache Spark eine API zum Zugriff auf Daten und deren Verarbeitung.

Diese API (wahlweise für die Programmiersprachen Scala, Java und Python verfügbar), kann im einfachsten Fall über eine eigene Spark Konsole mit Read Evaluate Print Loop[HKK99] verwendet werden.
Die Zählung von Wortvorkommen in einem Text - das „Hello World“ der Big Data Szene - lässt sich dort mit zwei Befehlen realisieren (Listing 2.1).

```
$ ./spark-shell
[...]
      / __/__  ___ _____/ /__
     _\ \/ _ \/ _ `/ __/  '_/
    /___/ .__/\_,_/_/ /_/\_\   version 1.3.0
       /_/
Using Scala version 2.10.4 (OpenJDK 64-Bit Server VM, Java 1.7.0_75)
Type in expressions to have them evaluated.
[...]
scala> val text = sc.textFile("../Heinrich Heine - Der Ex-Lebendige")
[...]
scala> :paste
text.flatMap(line => line.split(" "))
.map(word => (word, 1))
.reduceByKey(_ + _)
.collect()
[...]
res0: Array[(String, Int)] = Array((Tyrann,,1), (im,2), (Doch,1) ...)
```

Listing 2.1: Word Count in der Spark Konsole

Aus Sicht eines Administrators oder Softwarearchitekten ist Apache Spark eine Applikation auf einem Rechnercluster, die sich in der Anwendungsschicht befindet und charakteristische

Anforderungen insbesondere an Lokalität des Storages und die Netzwerkperformance stellt.

Was das konkret bedeutet, welche Mechanismen und Konzepte dahinterstehen und in welchem Ökosystem von Anwendungen sich Apache Spark bewegt wird in den folgenden Abschnitten dieses Kapitels beleuchtet.

2.1 Überblick

Im Allgemeinen Fall läuft eine Spark-Anwendung auf drei Arten von Rechnern (s. Abb. 2.1):

1. **Clientknoten**
   Auf Nutzerseite greift die Anwendung auf die API eines lokalen Spark-Kontextes zu, der die Kontaktdaten eines Clustermanagers sowie verschiedene Konfigurationseinstellungen enthält.

2. **Masterknoten**
   Der Masterknoten betreibt den *Clustermanager*, läuft auf einem entfernten Rechner und ist der Einstiegspunkt in den Rechnercluster. Hier werden Aufträge des Anwenders an die Arbeitsknoten verteilt und Ergebnisse eingesammelt und zurückgereicht.

3. **Workerknoten**
   Die Workerknoten beherbergen die Spark *Executors* und sind die ausführenden Elemente der Aktionen und Transformationen. Die *Executors* können untereinander Zwischenergebnisse austauschen und melden ihre Ressourcenverfügbarkeit an den *Clustermanager*.

Um die Architektur und Optimierungskonzepte eines verteilten Systems bewerten zu können ist es offensichtlich wichtig, welche Eigenschaften der unterliegenden Hardware angenommen werden.

Weil Spark explizit für den Betrieb innerhalb eines Hadoop/YARN [VERWEIS auf Abschnitt Scheduling] geeignet ist und YARN wiederum für den Betrieb auf einem Rechnercluster auf Mittelklasse-Mehrzweckmaschinen (Commodity Hardware) optimiert ist[AM14], kann für Spark von einer vergleichbaren Hardwarekonfiguration ausgegangen werden.

Der Vergleich von drei aktuellen Rack Servern der 2000-Euro-Klasse (in der Grundausstattung) - hier als Mittelklasse-Geräte bezeichnet - liefert die folgenden Verhältnisse der wesentlichen Schnittstellen zueinander (Siehe Anhang 3.1).

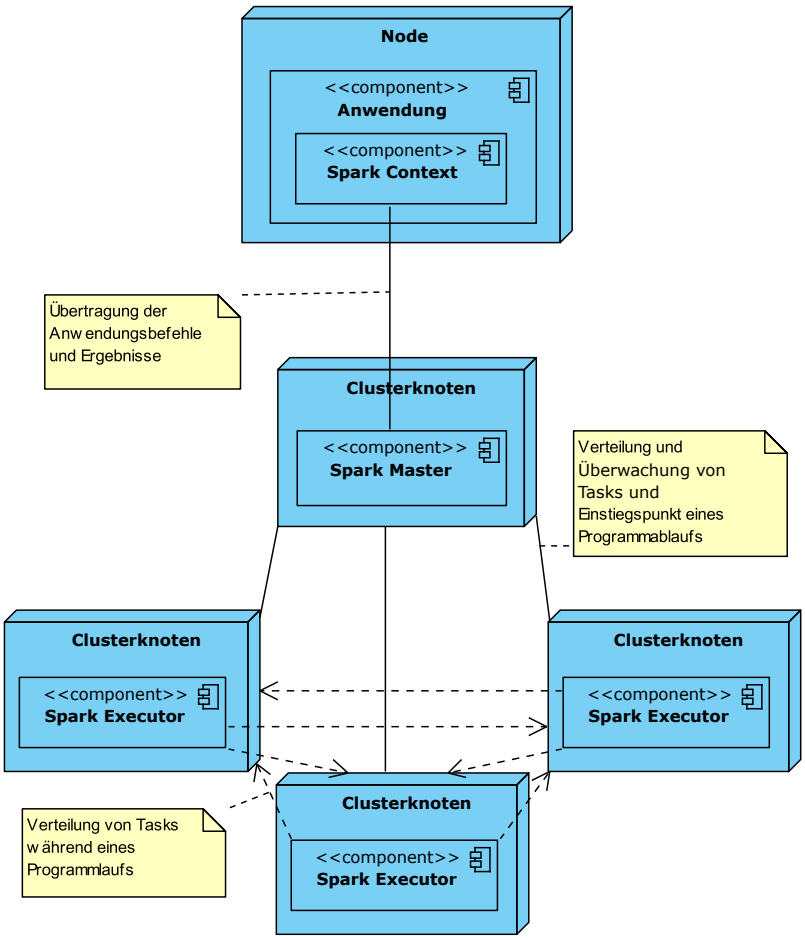

Abbildung 2.1: Verteilungsdiagramm einer typischen Sparkinstallation

Eine detaillierte Analyse des Zugriffsverhaltens ist nicht Gegenstand dieser Arbeit. Bei den folgenden Bewertungen der Kernkonzepte ist es wichtig sich die aus Tabelle 2.1 abgeleiteten Größenordnungen des Durchsatzes (D) der verschiedenen Datenkanäle zu vergegenwärtigen:

$$D_{Netzwerk} < D_{Festspeicher} < D_{Arbeitsspeicher}$$

Für eine effiziente Verarbeitung von Daten ist es - ganz allgemein - also wünschenswert den größten Anteil des Datentransfers im Arbeitsspeicher zu haben, einen kleineren Anteil auf der Festplatte und einen noch kleineren Anteil auf Netzwerkverbindungen.

Netzwerk	Festspeicher	Arbeitsspeicher
0,125 GB/s	1 GB/s	17 GB/s

Tabelle 2.1: Theoretische Spitzenleistungen bei Mittelklasse-Servern

Es ist das wichtigste Ziel der folgenden Kernkonzepte von Apache Spark unter diesen Bedingungen die effiziente und stabile Verarbeitung *großer Datenmengen*[SW14] zu gewährleisten.

2.2 Kernkonzepte

2.2.1 Resilient Distributed Datasets

Die universelle Einheit mit der Datenelemente auf Spark repräsentiert wird ist ein sogenanntes Resilient Distributed Dataset (RDD)[ZC+12].

Ein Beispiel für ein solches RDD wurde bereits erwähnt, nämlich das in Listing 2.1 erzeugte Objekt `text`:

```
1 val text = sc.textFile("../Heinrich_Heine_-_Der_Ex-Lebendige")
```

RDDs können auch explizit von einem Treiberprogramm erzeugt werden, ohne dass dazu vorhandene Daten genutzt werden:

```
1 val listRDD = sc.parallelize(List(1,2,3,4,5,6))
```

Die gesamte operative Kern-Application Programming Interface (API) dreht sich um die Steuerung dieser Dateneinheiten. Insbesondere sind auch die in den Standardbibliotheken verfügbaren „höheren“ APIs auf diesen RDDs implementiert.

Sie sind damit die wichtigste Abstraktion des Applikationskerns.

In erster Näherung können RDDs als eine Variante von Distributed Shared Memory (DSM)[NL91] [ZC+12] verstanden werden, haben allerdings sehr charakteristische Einschränkungen und Erweiterungen, die in diesem Kapitel erläutert werden.

Verteilungssicht Aus Verteilungssicht ist ein RDD ein Datensatz, der über den Arbeitsspeicher mehrerer Maschinen partitioniert ist (Abb. 2.2).

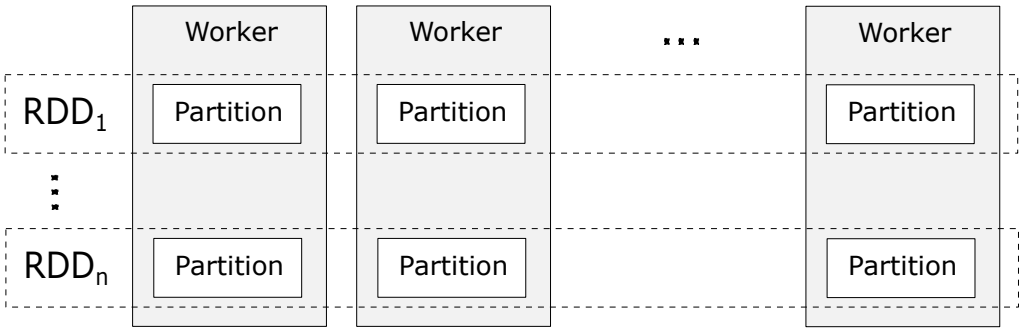

Abbildung 2.2: Resilient Distributed Datasets aus Verteilungssicht

Laufzeitsicht RDDs sind nicht veränderbar. Es ist nicht möglich einzelne Elemente durch gezielte Operationen zu verändern. Stattdessen ist es nur möglich ein einmal definiertes RDD durch globale Anwendung von Operationen in ein anderes zu überführen.
Solche globalen (also auf sämtlichen Partitionen des durchgeführten) Operationen können zwar ihren Effekt auf einzelne Elemente eines beschränken, die Ausführung erfolgt jedoch in jedem Fall auf allen Partitionen.

Eine Folge von Operationen $op_1 op_2 op_3 \ldots$ wird als *Lineage* eines RDD bezeichnet. Die *Lineage* kann als das „Rezept“ zur Erstellung eines Datensatzes verstanden werden.

Dabei gibt es zwei grundsätzlich verschiedene Operationen, nämlich *Transformationen* und *Aktionen.*

Transformationen sind Operationen, die ein auf ein anderes abbilden:

$$Transformation : RDD \times RDD \longrightarrow RDD$$

oder

$$Transformation : RDD \longrightarrow RDD$$

Es werden also - grob gesagt - nur die abstrakte Repräsentation des Datensatzes ändern, ohne tatsächlich dessen Datenelemente für den Programmfluss im Treiberprogramm abzurufen.
Beispiele für solche Operationen sind:

- *filter*
- *join*

Aktionen sind Operationen, die RDDs in eine andere Domäne abbilden:

$$Action : RDD \longrightarrow Domain_x, Domain_x \neq RDD$$

Beispiele für Aktionen sind die Methoden:

- *reduce*
- *count*
- *collect*
- *foreach*

Jeder dieser Operationen, kann im Sinne des Command-Patterns ([BL13]) eine Funktion bzw. ein Funktionsobjekt übergeben werden, dass die gewählte Operation spezifiziert.

Solange nur *Transformationen* auf einem RDD ausgeführt werden, ist dieses noch ein bloßes „Rezept“ zur Erstellung eines Datensatzes. Tatsächlich wurde noch kein Speicher reserviert und der Cluster wurde noch nicht aktiv[ZC+12]:

Abbildung 2.3: RDD Lineage vor Aktion (gestrichelte Linie steht für *nicht initialisiert*)

Sobald die erste *Aktion* aufgerufen wird, werden die Transformationen nach der vorgegebenen Reihenfolge ausgeführt und die geforderte *Aktion* ausgeführt. Die Initialisierung des RDD erfolgt also „lazy“:

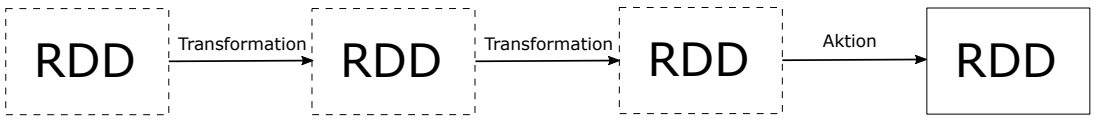

Abbildung 2.4: RDD Lineage nach Aktion

Wie in Abb. 2.4 dargestellt ist, werden während der Transformationsvorgänge keine Zwischenergebnisse gespeichert. Möchte man Zwischenergebnisse zu einem späteren Zeitpunkt

oder in anderem Zusammenhang wiederverwenden, kann man dies explizit über das Kommando `persist()` anweisen:

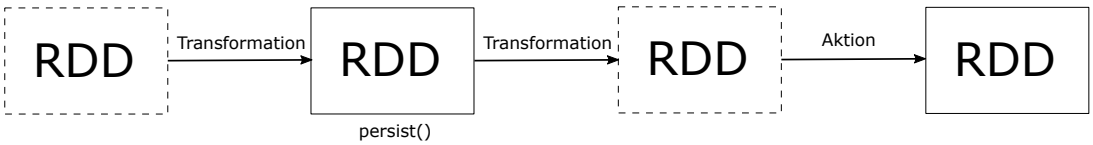

Abbildung 2.5: RDD Lineage nach Aktion und mit Persist()

Realisiert ist das Konzept der *Lineage* und der „Lazy Initialization“ von RDDs durch Transformations-Methoden, die eine Variante des Factory-Pattern ([BL13]) implementieren. Die erzeugten Objekte sind dabei wiederum Unterklassen von RDD:
Jedes RDD-Objekt führt eine Liste von Vorgängern mit. Aus dieser lässt sich auch die Art der Berechnung des jeweiligen Nachfolgers ableiten.

Jede weitere Transformations-Methode konstruiert nun lediglich eine neues RDD-Objekt. Dieses basiert auf dem aktuellen Objekt und der jeweiligen Transformation.

Ein Beispiel für solch eine Transformation auf einem RDD ist die Methode *map*[1] (Listing 2.2).

In den Zeilen 7 und 8 ist zu sehen, wie ein neues RDD erzeugt wird und diesem das aktuelle RDD sowie die auf dessen Elemente anzuwendende Funktion `f` (bzw. `cleanF`) übergeben wird.

```
1 /**
2  * Return a new RDD by applying a function to all elements of
3       * this RDD.
4  */
5 def map[U: ClassTag](f: T => U): RDD[U] = {
6   val cleanF = sc.clean(f)
7   new MapPartitionsRDD[U, T](this, (context, pid, iter)
8                 => iter.map(cleanF))
9 }
```

Listing 2.2: Map-Methode aus org.apache.spark.rdd.RDD v1.3.0

[1] https://github.com/apache/spark/blob/branch-1.3/core/src/main/scala/org/apache/spark/rdd/RDD.scalaL285 (abgerufen am 30.05.2015)

Die tatsächliche Berechnung eines RDDs wird dann bei Aufruf einer Aktion gestartet. Als Beispiel hierfür sei die Methode `foreach`[2] aufgeführt (Listing 2.3).
In Zeile 6 wird auf dem Spark-Context die Methode `runJob` aufgerufen, der die Aufgabe weiter an den Scheduler delegiert. Dort werden dann die rekursiven Abhängigkeiten des aktuellen RDD aufgelöst und - je nach Konfiguration - die Tasks verteilt.

```
1 /**
2  * Applies a function f to all elements of this RDD.
3  */
4 def foreach(f: T => Unit) {
5   val cleanF = sc.clean(f)
6   sc.runJob(this, (iter: Iterator[T]) => iter.foreach(cleanF))
7 }
```

Listing 2.3: foreach-Methode aus org.apache.spark.rdd.RDD v1.3.0

Das Konzept der *Lineage* ist zentral für die Fehlertoleranz der RDDs: Geht eine Partition verloren - beispielsweise durch Defekt eines Knotens - ist das „Rezept“ zur Erstellung des Datensatzes in der *Lineage* des RDD-Objektes weiterhin vorhanden und die Partition kann gezielt wiederhergestellt werden.

Ein weiterer Vorteil dieser Art von Arbeitsdatensatz wird ebenfalls sofort deutlich: Im optimalen Fall sind die zu ladenden Daten von jedem der Worker auf unabhängigen Kanälen erreichbar (z.B. auf dem lokalen Festspeicher) und gleichmäßig auf diesen Kanälen partitioniert.

Im diesem Fall ergäbe sich mit einer Anzahl Worker n und einem Durchsatz δ zu der jeweiligen Datenquelle also ein idealer Gesamtdurchsatz beim Einlesen von Daten von:

$$\sum_{i=1}^{n} \delta_i \tag{2.1}$$

Kommen in einem Anwendungsfall RDDs zum Einsatz, deren Elemente einzeln über eine oder mehrere Operationen untereinander verknüpft werden, kann es sinnvoll sein diese schon im Vorfeld der Verarbeitung entsprechend erwarteter Cluster zu partitionieren. Cluster seien hierbei Teilmengen des RDD, die mit besonders hoher Wahrscheinlichkeit oder besonders

[2] https://github.com/apache/spark/blob/branch-1.3/core/src/main/scala/org/apache/spark/rdd/RDD.scalaL793 (abgerufen am 30.05.2015)

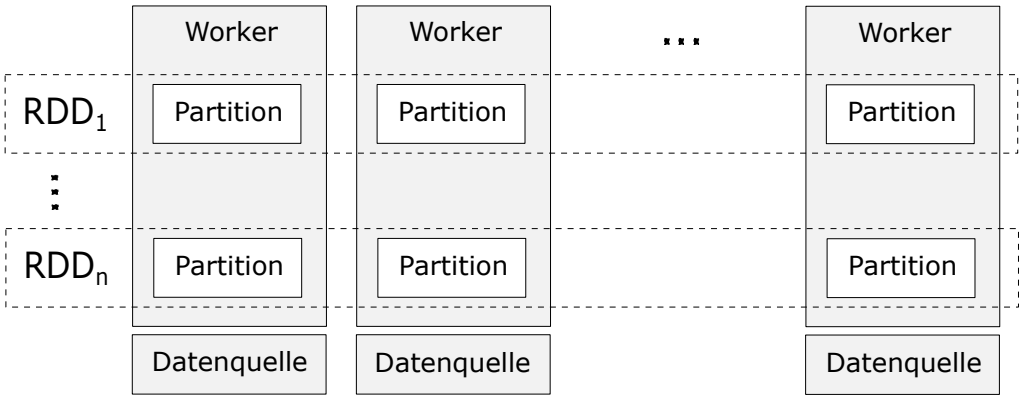

Abbildung 2.6: Resilient Distributed Datasets mit Datenquelle aus Verteilungssicht

häufig untereinander verknüpft werden.

Dadurch kann ein größerer Teil der Operationen auf den einzelnen Datensätzen bereits lokal auf dem Knoten der jeweiligen Partition durchgeführt werden. Die Netzwerklast bei dem anschließenden *Shuffle* der Daten (siehe Abschnitt 2.2.2) fällt dann geringer aus.

Solch eine Partitionierung kann - entsprechende Erwartung über das Verhalten der Verarbeitung vorausgesetzt - mit einem maßgeschneiderten Partitionierer erreicht werden (Abb. 2.4) der dann dem betroffenen RDD übergeben wird.

```
/*
 * Beispiel fuer einen minimalen Partitionierer.
 * Ueber selbstdefinierte Hash Codes kann hier eine
 * massgeschneiderte Verteilung ueber die
 * Knoten erreicht werden.
 */
class MinimalPartitioner extends Partitioner {
 def numPartitions = 10

 def getPartition(key: Any): Int =
   key.hashCode % numPartitions

 def equals(other: Any): Boolean =
   other.isInstanceOf[MinimalPartitioner]
```

```
15 }
```

Listing 2.4: Beispiel: Minimaler Partitionierer

2.2.2 Scheduling/Shuffling

Dieser Abschnitt vertieft die Betrachtung des Berechnungsmodells von Spark. Mit Berechnungsmodell ist gemeint die Art, wie mit den High-Level APIs und den bisher vorgestellten Komponenten die verteilte Berechnung realisiert wird.
Dabei werden insbesondere die Begriffe ***Task***, ***Job*** und ***Stage*** näher erläutert.

Bei den RDDs wurden bisher insbesondere drei Aspekte behandelt:

- die **Partitionierung** der Elemente über verschiedene Rechner
- die **Vorgänger** eines RDD bezüglich dessen *Lineage*
- die **Funktion** mit der ein RDD aus einem oder mehreren direkten Vorgängern berechnet wird

Betrachtet man nur die Vorgänger-Beziehung, dann erhält man zunächst eine einfache Baumstruktur für die Berechnung eines RDD (Abb. 2.7.

Betrachtet man zusätzlich die Partitionierung und die Funktion mit der RDDs transformiert werden, sieht man einen wichtigen Unterschied zwischen verschiedenen Vorgängerbeziehungen (siehe Abb. 2.8):

- Solche deren einzelne Partitionen höchstens eine Vorgängerpartition haben
- Solche bei denen mindestens eine Partition mehr als eine Vorgängerpartition hat

Wird eine Partition aus höchstens einer anderen erzeugt, lässt sich diese direkt auf dem selben Knoten berechnen. Werden jedoch verschiedene Partitionen benötigt um eine Folgepartition zu erzeugen stellt sich die Frage auf welchen Knoten das am Besten geschieht.
Spark unterteilt diese beiden Fälle in *narrow dependencies* (erster Fall) und *wide dependencies* (zweiter Fall) ([ZC+12]).

Der Abhängigkeitsgraph eines RDD wird nun in sogenannte *Stages* zerlegt (Abb. 2.9). Eine *Stage* ist dabei ein Untergraph, dessen Elemente (von den Blättern in Richtung Wurzel betrachtet) auf eine gemeinsame *wide dependency* stoßen.

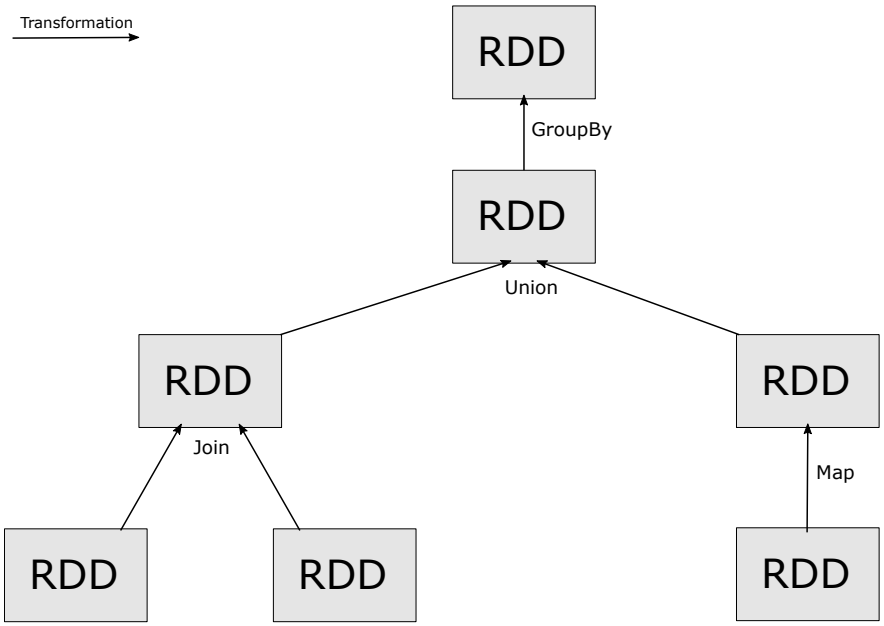

Abbildung 2.7: Einfacher Abhängigkeitsbaum eines RDD

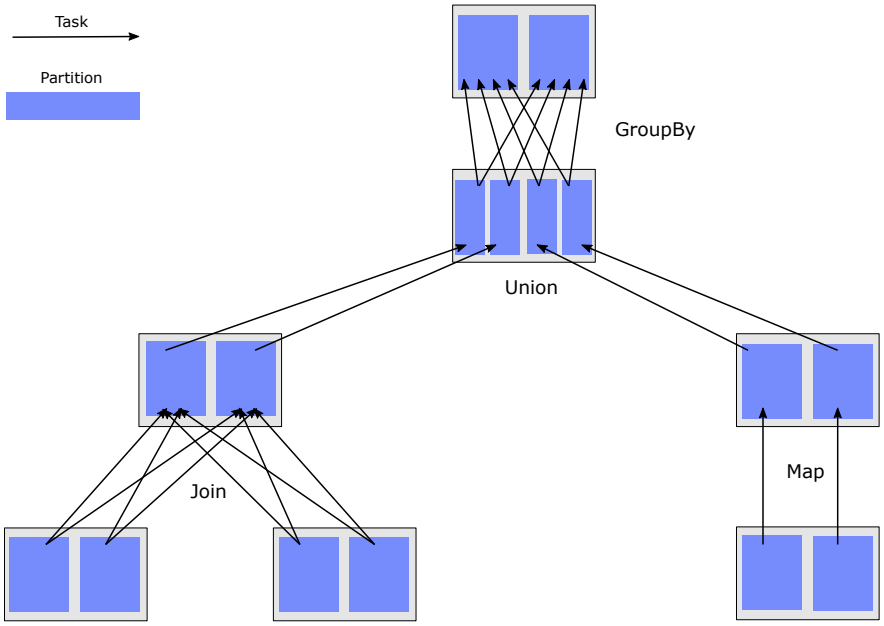

Abbildung 2.8: Gerichteter azyklischer Graph der Abhängigkeiten auf Partitionen

Elemente innerhalb einer Stage, können nach dieser Konstruktion unabhängig auf den Knoten berechnet werden, die die jeweilige Partition vorhalten. Ein Datenelement $x_{i,j}$ sei

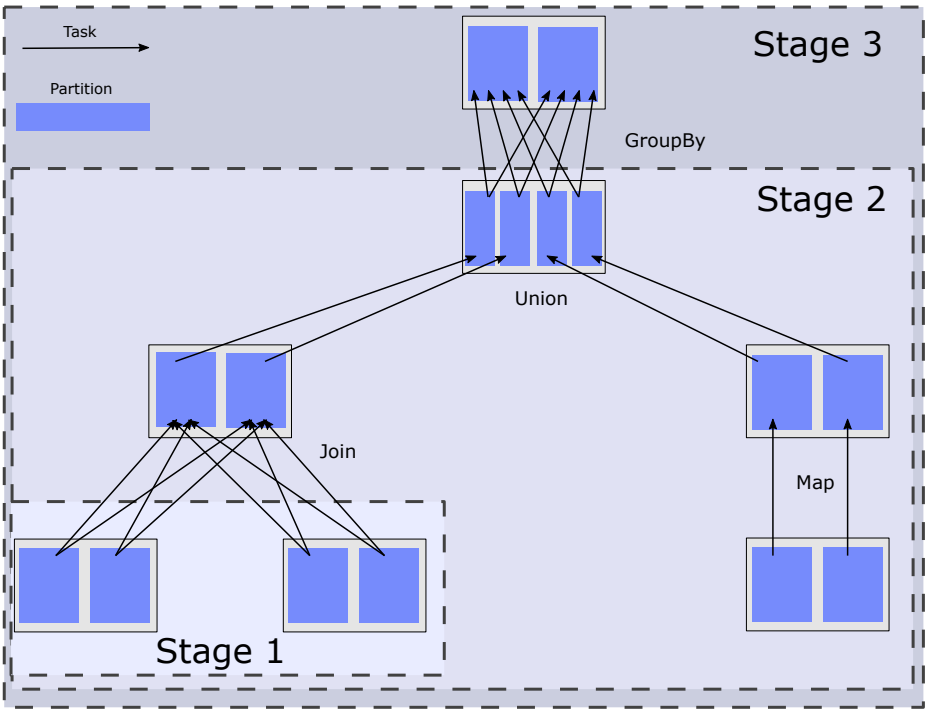

Abbildung 2.9: Stages als Untermenge des Ahängigkeitsgraphen von Partitionen

dabei Element eines RDD j und Element einer lokalen Partition $P_{i,j}$ (d.h. es existiert auf einem Knoten i):

$$x_{i,j} \in Partition_{i,j} \subseteq RDD_j$$

Weil nach Definition der *narrow dependencies* innerhalb einer Stage die Elemente einer Partition $P_{i,j}$ aus den Elementen genau einer Vorgängerpartition $P_{i,j-1}$ berechnet werden können, lässt sich jedes Element $x_{i,j}$ des Nachfolger-RDD über eine Komposition lokaler Transformationen $trans_{i,k}, k = 0..j$ berechnen:

$$x_{i,j} = (trans_{i,j} \circ trans_{i,j-1} \circ ... \circ trans_{i,1} \circ trans_{i,0})(x_{i,0})$$

Für den Übergang zwischen Stages ist die Berechnung der Nachfolger etwas aufwändiger. Hier stammen - nach Definition der *wide dependencies* - die direkten Vorgänger eines Elementes nicht aus der selben Partition.

Um eine neue Partition aus mehreren Vorgänger-Partitionen zu erzeugen werden zunächst geeignete Ausführungsorte[3] ermittelt, von denen dann jeder per geeignetem Partitionierer mindestens einen *bucket* von Elementen verarbeitet. Diese Elemente stammen dann in der Regel aus verschiedenen Vorgängerpartitionen.
Das so neu-partitionierte und verarbeitete wird so zum Ausgangspunkt des folgenden RDD.
Eine Schlüsselrolle kommt bei diesem Prozess der Methode `runTask` in der Klasse `ShuffleMapTasks`[4] aus *org.apache.spark.scheduler.ShuffleMapTask* zu.

2.2.3 Anwendungsdeployment und -lebenszyklus

Anwendungsdeployment Die Komponente einer Anwendung, die von der Spark API Gebrauch macht wird in der Spark-Terminologie als *Treiberprogramm* bezeichnet.

Es gibt verschiedene Szenarien des Deployments. Das Treiberprogramm kann grundsätzlich auf zwei verschiedene Arten gestartet werden:

1. Übermittlung des Treibers als kompiliertes Package (z.B. als) mit statischer Verlinkung aller erforderlichen Bibliotheken (Ausnahmen sind Bibliotheken die auf allen Knoten bereits verfügbar sind, z.B. Spark, Hadoop, etc.). Standardbibiotheken von Spark und Konfigurationseinstellungen wie z.B. die Angabe des *Clustermanagers* können zur Startzeit des Treibers durch das Spark *Submission-Skript* erfolgen.

2. Start eines eigenständig lauffähigen Treibers mit vollständig konfigurierter und verlinkter Spark-Umgebung und expliziter Angabe eines *Clustermanagers*.

Der zweite Fall ist eher exotisch, weil eine derart enge Kopplung zwischen dem Treiber und der Konfiguration des Clusters offensichtlich aus Wartungsgründen nicht wünschenswert ist.

Im ersten Fall ergeben sich zwei weitere Möglichkeiten zum Ort der Ausführung des Treibers:

[3] Geeignete Ausführungsorte (*preferred locations*) können sich z.B. aus dem Ort eines Blocks bei HDFS ermitteln lassen (Node-Local, Rack-Local, etc.) oder daraus, ob ein Executor bereits einen Datensatz geladen hat (Process-Local)

[4] https://github.com/apache/spark/blob/branch-1.3/core/src/main/scala/org/apache/spark/scheduler/ShuffleMapTask.scala (abgerufen am 30.05.2015)

1. **Client-Modus** Der Treiber wird direkt auf dem Host (Gateway-Rechner) ausgeführt auf dem der Treiber übermittelt wurde (Abb. 2.10). Tatsächlich wird er sogar innerhalb des Submission-Skript-Prozesses gestartet **(VERWEIS)**.

2. **Cluster-Modus** Der Treiber wird von dem Gateway-Rechner an Worker des Clusters übertragen und dort ausgeführt (Abb. 2.11).

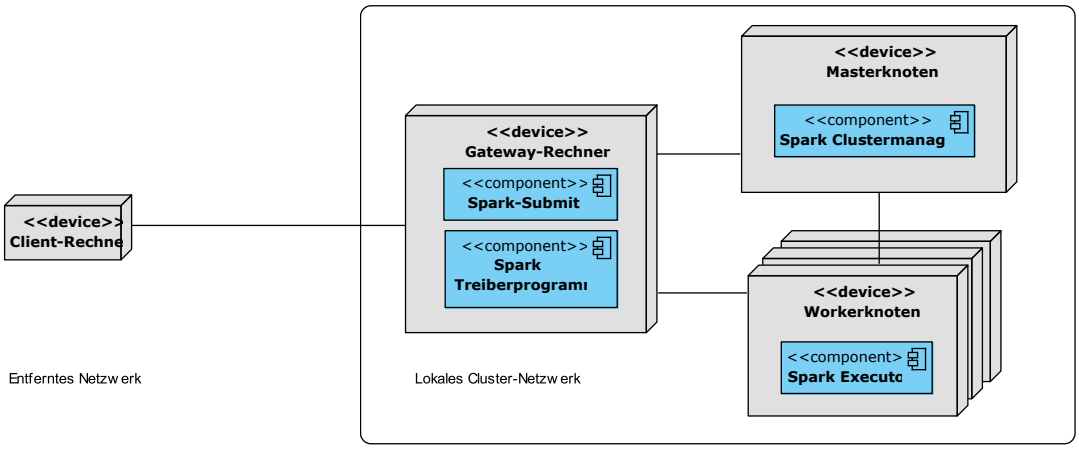

Abbildung 2.10: Application Deployment im Client Modus

Offensichtlich kann die Lokalität des Treibers (der mit den Executors auf den Workern kommunizieren muss) Einfluss auf Laufzeit und Latenzverhalten des Programms haben.

Im Fall eines clusterfernen Gateway-Rechners kann also Treiberdeployment im Clustermodus sinnvoll sein, die Standardeinstellung ist jedoch der Clientmodus (siehe auch [Spa]).

Abb. 2.12 zeigt das Sequenzdiagramm eines typischen Deploymentprozesses, wie er auch im praktischen Teil dieser Arbeit zum Einsatz kommt.

Lebenszyklus einer Spark-Applikation Beginn und End eines Treiberprogramms. Beginn un Ende eines Tasks / Executorallokation?

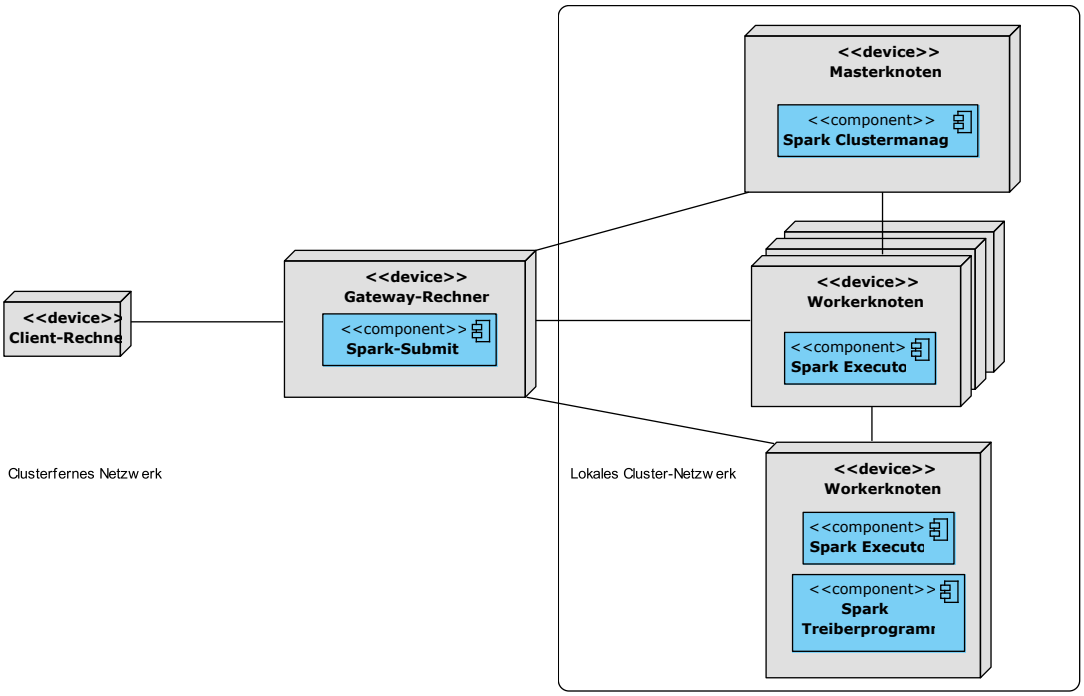

Abbildung 2.11: Application Deployment im Cluster Modus

2.2.4 Zusammenfassung und Bewertung

2.3 Standardbibliotheken

Die vier Standardbibliotheken erweitern die Kern-API für bestimmte, häufig genutzte Aufgaben aus Bereichen der Datenanalyse.

Die bedienten Bereiche

- Deklaratives Abfragen auf strukturierten Datensätzen (*Spark SQL*)
- Maschinenlernverfahren (*MLlib*)
- Echtzeitbehandlung von eingehenden Daten (*Streaming*)
- Operationen auf Graph-Strukturen (*GraphX*)

werden in diesem Abschnitt erläutert.

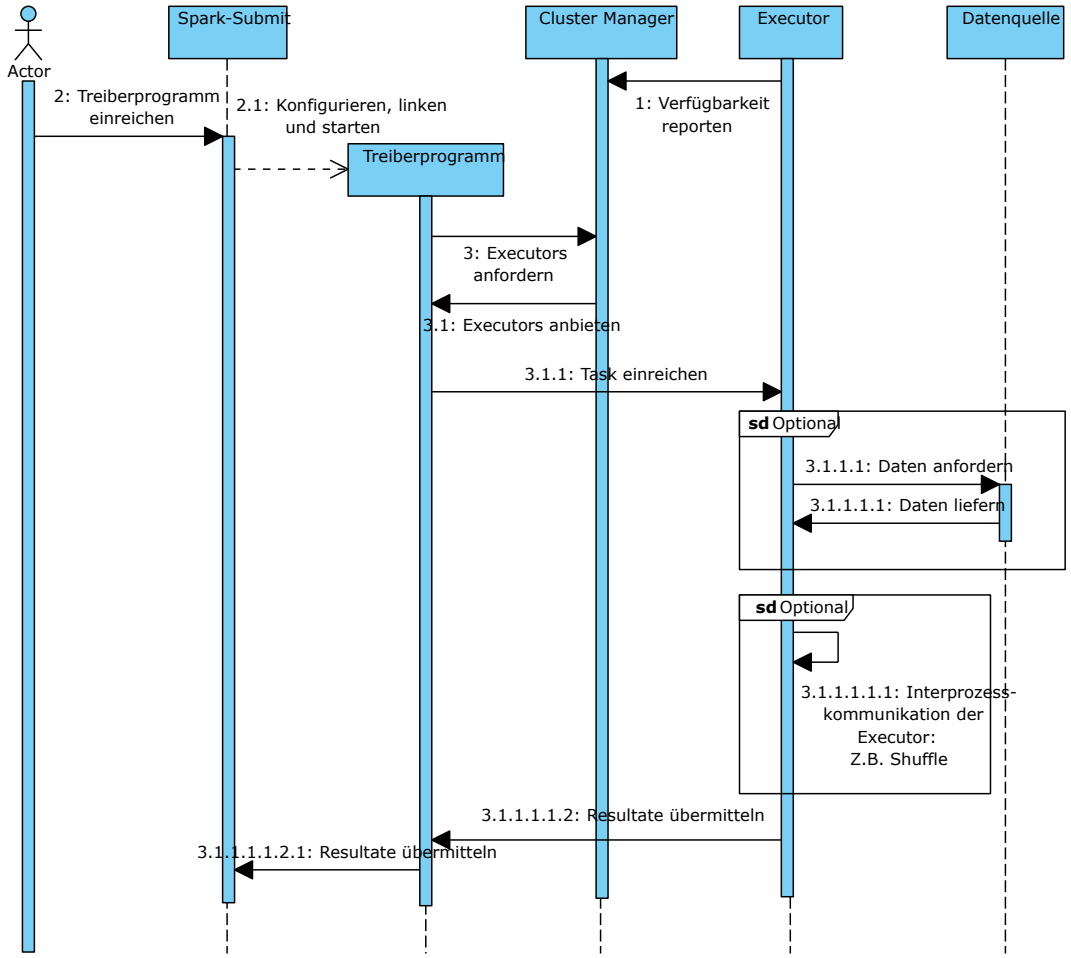

Abbildung 2.12: Application Deployment Prozess im Client Modus (vereinfacht)

2.3.1 Dataframes/Spark SQL

[Arm+15]

2.3.2 MLlib

2.3.3 Streaming

2.3.4 GraphX

[Gon+14]

2.4 Betrieb und Security

2.5 Spark im Kontext von Parallelisierungspattern

— Buch: Algorithms and Parallel Computing —

2.6 Entwicklergemeinschaft

— Herkunft, Apache Foundation, Entwicklungsphilosophien, Anzahl Entwickler, ... —

Apache Spark begann als Entwicklung einer Gruppe von Forschern der University of California, Berkely. Spark ist eine Implementation der von dieser Gruppe untersuchten RDDs[ZC+12]. Als wesentlicher Meilenstein der Entwicklung von Apache Spark, kann die Veröffentlichung eines gemeinsamen Papers der Forschungsgruppe um Matei Zaharia im Jahr 2012 gelten.

Seit dem 27. Februar 2014[apa] ist Spark ein Top-Level Projekt der Apache Software Foundation[Apaa] und wird dort unter der Apache License 2.0[Apab] weiterentwickelt.

Eine Übersicht der verantwortlichen Committer kann unter [Com] eingesehen werden. Zum Zeitpunkt dieser Arbeit gehören u.a. Entwickler von Intel, Yahoo! und Alibaba zu den Stammentwicklern.

Die Kommunikation innerhalb der Entwickler- und Anwendergemeinschaft findet wesentlich in den offiziellen Mailinglisten (Abb. 2.13) und dem Issue-Tracker[Iss] der Apache Software Foundation statt.

2.7 Auswahl verwandter Produkte

Um Spark besser im Bereich bestehender Lösungen einzuordnen werden im Folgenden einige Produkte genannt die häufig zusammen mit Spark verwendet oder ähnliche Aufgaben erfüllen.

Hadoop/YARN Hadoop lässt sich als eine Art Betriebssystem für Cluster zur Datenanalyse beschreiben. Zu den wesentlichen Komponenten zählen ein Dateisystem (HDFS) ein Datenverarbeitungsmodell (MapReduce) und ein Scheduler (YARN). Alle genannten Komponenten sind für den fehlertoleranten und skalierbaren Betrieb auf verteilten Hardware-Komponenten

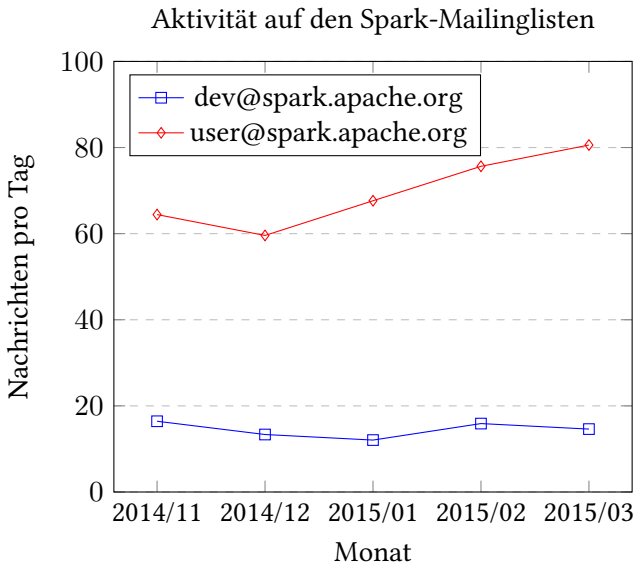

Abbildung 2.13: Aktivität auf den offiziellen Spark Mailinglisten

vorgesehen.
Zwar wird mit MapReduce auch eine Komponente zur Verarbeitung/Analyse von verteilt gespeicherten Daten zu Verfügung gestellt, andere Datenverarbeitungsmodelle können jedoch gleichberechtigt und unter Aufsicht des Ressourcenschedulers YARN betrieben werden.

Spark liefert eine mögliche Implementation eines solchen alternativen Datenverarbeitungsmodells.

Mesos Apache Mesos ist seit Beginn der Entwicklung von Spark ein optionaler Clustermanager für Sparkapplikationen [ZC+12].
Als reiner Clustermanager ersetzt Mesos die Spark Master-Komponente in der Funktion des knotenübergreifenden Ressourcenmanagements.
Wie bei YARN ermöglicht dies auch anderen Anwendungen die über Mesos verwaltet werden einen gleichberechtigten Betrieb auf dem selben Cluster.

Flink

MPI

Samza

Storm

3 Beispielanwendung von Spark zur Datenanalyse

Zur vertiefenden Betrachtung von Spark wird im Folgenden die Implementation einer Beispielanwendung gezeigt.

Dazu sollen mindestens zwei verschiedene Standardbibliotheken und deren Integration in eine gemeinsame Anwendung durchgeführt werden.

Anschließend werden rudimentäre Skalierungs- und Stresstests der Anwendung durchgeführt und der durchgeführte Versuch unter den Kriterien *Einfachheit*, *Skalierbarkeit*, *Erweiterbarkeit*, *Robustheit*, *Sicherheit* und *Wartbarkeit* zusammengefasst und bewertet.

3.1 Anwendungsfall

In dem Anwendungsfall soll ein Dashboard erstellt werden, auf dem Textnachrichten angezeigt werden. Die Textnachrichten werden aus einem Datenstrom gelesen und dabei - entsprechend ihrer aktuellen Relevanz für einen Sparkbenutzer - bewertet und gefiltert. Als Maßstab für die Relevanz sollen Begriffen dienen, die kürzlich in der Mailingliste der Sparkbenutzer diskutiert wurden.

Als Datenquellen dienen einerseits die Emails aus der Malingliste[1] und andererseits der öffentliche Datenstrom[2] mit Textnachrichten (Tweets) der Plattform Twitter[3].

3.1.1 Anforderungen

Für die Software soll folgende lose Sammlung funktionaler und nicht-funktionaler Anforderungen gelten. Mit *Information* ist jeweils eine Auflistung von Tweets gemeint.

[1] user@spark.apache.org
[2] https://dev.twitter.com/streaming/sitestreams
[3] https://twitter.com, abgerufen am 06.06.2015

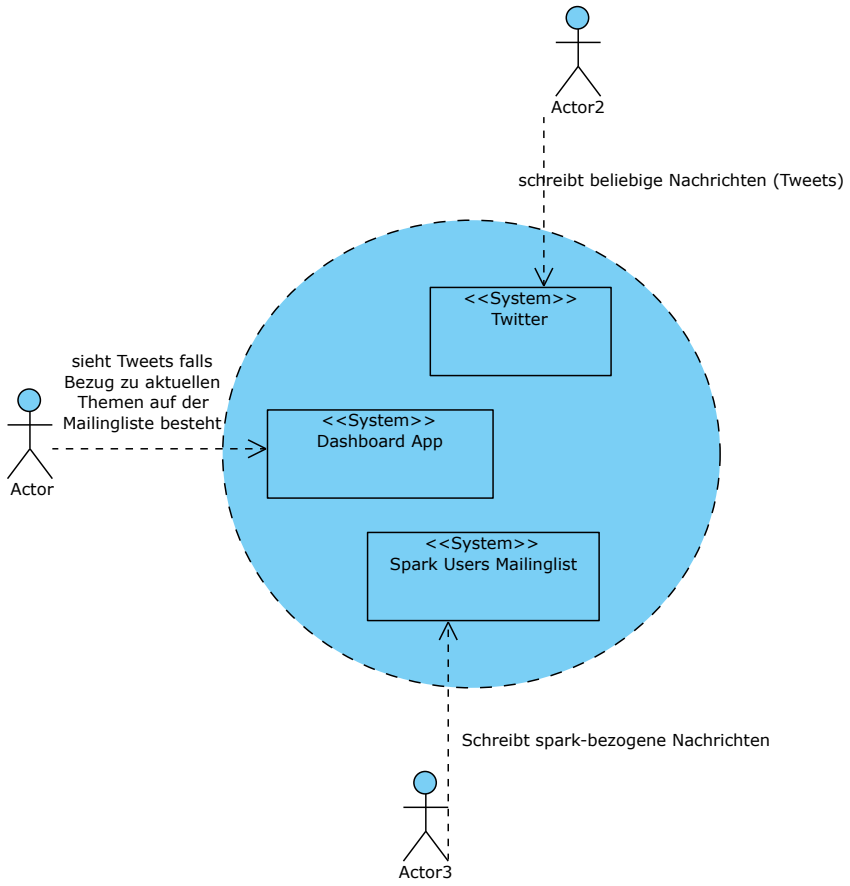

Abbildung 3.1: Anwendungsfalldiagramm der Demo-Applikation

- **A1: Zugriff auf die Information**
  Der Zugriff soll über eine grafische Benutzerschnittstelle erfolgen und keine Konfigurationen benötigen.
- **A2: Aktualität der Information**
  Es sollen stets Informationen dargestellt werden, die unmittelbar zuvor entstanden sind und in Quasi-Echtzeit[4] verarbeitet wurden.
- **A3: Relevanz der Information**
  Die Relevanz soll an aktuellen Themen der Entwicklergemeinschaft gemessen werden.

[4] Eine Latenz von unter einer Minute sei hier tolerabel

3.1.2 Technische Rahmenbedingungen

Als Versuchsumgebung dient ein Rechnercluster aus vier identischen Workern und einem speziellen Masterknoten (Abb. 3.1).

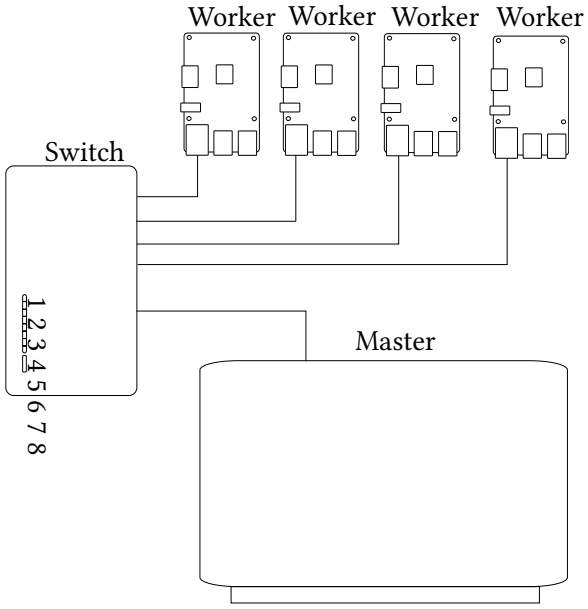

Abbildung 3.2: Hardwareumgebung des Programms zur Tweetanalyse

Worker Rasperry Pi 2

- CPU: 900MHz Quad-Core ARM Cortex A7
- RAM: 1GB SDRAM
- Ethernet: 100MBit/s
- Festspeicher: SDHC Class 4 Speicherkarte 16GB

Als Betriebssystem kommt das Debian-Derivat Raspbian[Ras] 32-Bit zum Einsatz.

Master Dell d420

- CPU: 1,2 GHz Core2 Duo U2500
- RAM: 2GB DDR2 SDRAM
- Ethernet: 100MBit/s

- Festspeicher: 60GB 4200RPM Hard Drive

Als Betriebssystem komm Ubuntu[Ubu] 14.04 32-Bit zum Einsatz.

Netzwerk Vernetzt sind die Rechner mit RJ45 über einen TP-Link TL-SF1008D Switch mit maximalem Durchsatz von 100MBit/s.

Tabelle 3.1: —DUMMY— Netzwerkdurchsatz

Nachrichtengröße	Worker → Worker	Master → Worker	Worker → Master
1kB	50ms	837ms	970ms
64kB	47ms	877ms	230ms
1MB	31ms	25ms	415ms
64MB	35ms	144ms	2356ms

3.1.3 Lösungsskizze

Wahl des Dateisystems Als Quelle der persistenten Daten (Nachrichtenkorpus der Mailinglisten) kommen verschiedene Technologien in Frage:

Tabelle 3.2: Übersicht ausgewählter Datenquellen für Spark

Name	Typ	Beschreibung
Cassandra	Datenbank	...
HBase	Datenbank	...
HDFS	Verteiltes Dateisystem	...
Kafka	??	...

Für diesen Versuch wird HDFS zum Verwalten der Textdatei gewählt. Für das Einlesen des Textkorpus wird keine Echtzeitfunktionalität benötigt. Weil an den Textdateien nichts geändert wird ist Versionierung ebenso unnötig wie Verknüpfungen zu anderen Datensätzen.

Die „In-Memory“-Funktionen (VERWEIS Glossar?) anderer Systeme, sind hier eher hinderlich, weil der lokale Arbeitsspeicher der Arbeitsknoten im Versuchsaufbau stark beschränkt ist und eine leicht erhöhte Latenz beim Einlesen in Kauf genommen werden kann. Das Erstellen des Feature Vektors (VERWEIS) ist nicht zeitkritisch für die Echtzeitkomponente.

HDFS als Komponente von Apache Hadoop ist in den Versionen 2.x deutlich bezüglich der Verfügbarkeit und Skalierbarkeit verbessert worden (VERWEIS), allerdings auch aufwändiger zu installieren. Für den Zweck dieses Versuchs wird Version 1.2.1 gewählt.

- item
- item

Tabelle 3.3: Übersicht verfügbarer Clustermanager für Spark

Name	Typ	Beschreibung
Standalone	Spezifischer Clustermanager für Spark	...
Mesos	General Purpose	...
YARN	General Purpose	Apache Hadoop 2.x Clustermanager

Wahl des Cluster-Managers Spark läuft in diesem Versuch als alleinige Computeanwendung auf dem Cluster. Es ist also nicht nötig Konkurrenz um Ressourcen zu berücksichtigen. Für diesen Versuch wird daher der Standalone Clustermanager gewählt.

Architekturübersicht Die Implementation des Anwendungsfalles soll in drei Schichten erfolgen.
In einer Schicht findet die Verarbeitung eingehender Emails statt und es werden die Relevanz der Begriffe bewertet (*Batch Layer*). In einer zweiten Schicht werden die Tweets aus einem Datenstrom eingelesen und deren Relevanz anhand der Bewertungen aus der ersten Schicht bewertet (*Streaming Layer*).
In der dritten Schicht werden die als relevant eingestuften Emails in einer grafischen Oberfläche dem Benutzer zur Verfügung gestellt (*Presentation Layer*).

Batch Layer

Streaming Layer

Presentation Layer

Batch Komponente

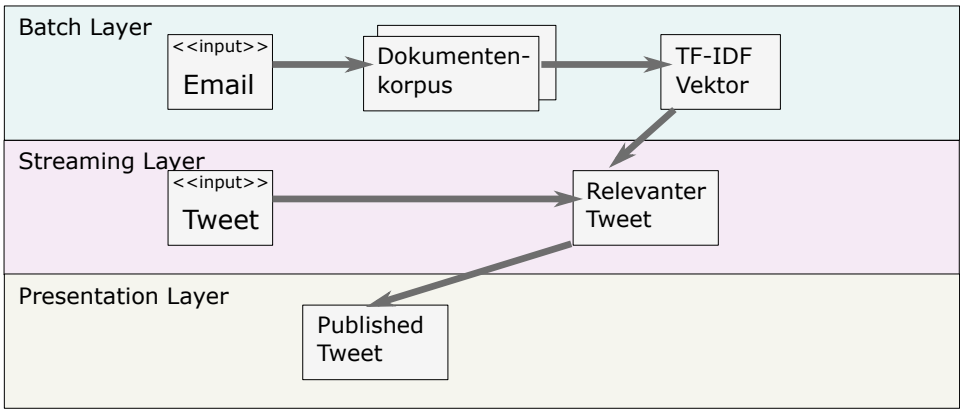

Abbildung 3.3: Datenzentrierte Sicht auf die Komponenten

Streaming-Komponente

Presentation-Komponente

Hinweise zur Entwicklung Die Komponenten werden in jeweils eigenen Projekten entwickelt, die sich einzeln auf dem Cluster deployen lassen. Das hat den Vorteil, dass eine einfache Continuous Deployment Pipeline (Abb. 3.5) eingesetzt werden kann, die Änderungen an den jeweiligen Projekten automatisiert auf dem Cluster deployt und so schnellstmögliches Feedback ermöglicht, sowie eine stets lauffähige Codebasis begünstigt.

3.1.4 Ergebnisse und Bewertung

Dashboard

Laufzeitverhalten Blockgröße, Anzahl Worker, Anzahl Replikationen der Blöcke

Bewertung und Probleme *Einfachheit Skalierbarkeit Erweiterbarkeit Robustheit Sicherheit Wartbarkeit*

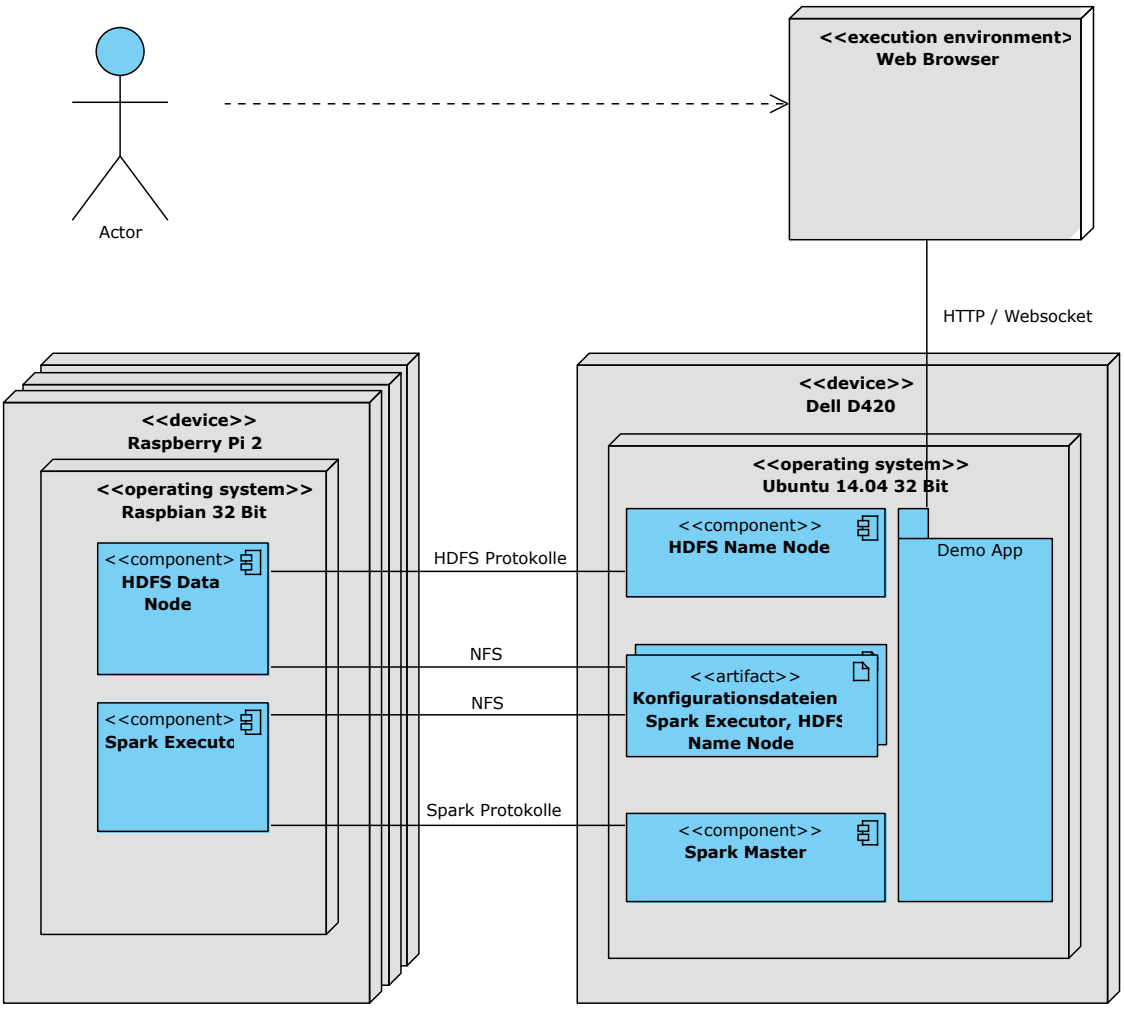

Abbildung 3.4: Verteilungssicht auf die Demo App

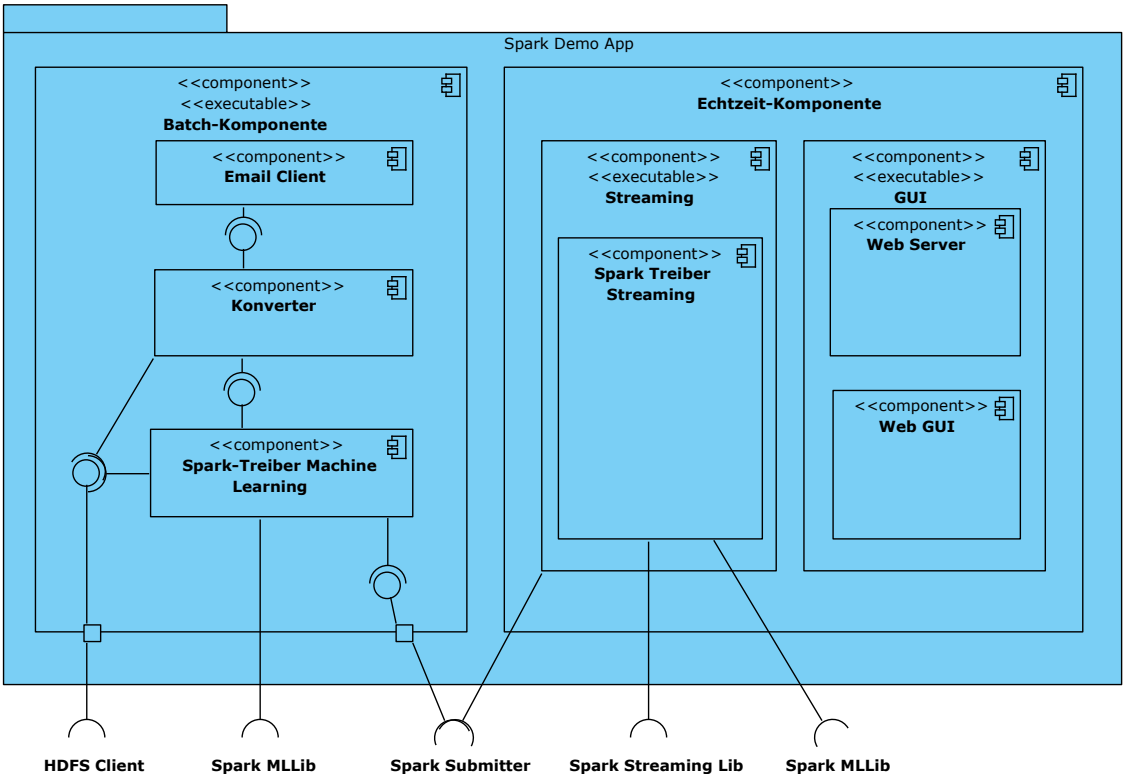

Abbildung 3.5: Komponentendiagramm des Demo App Packages

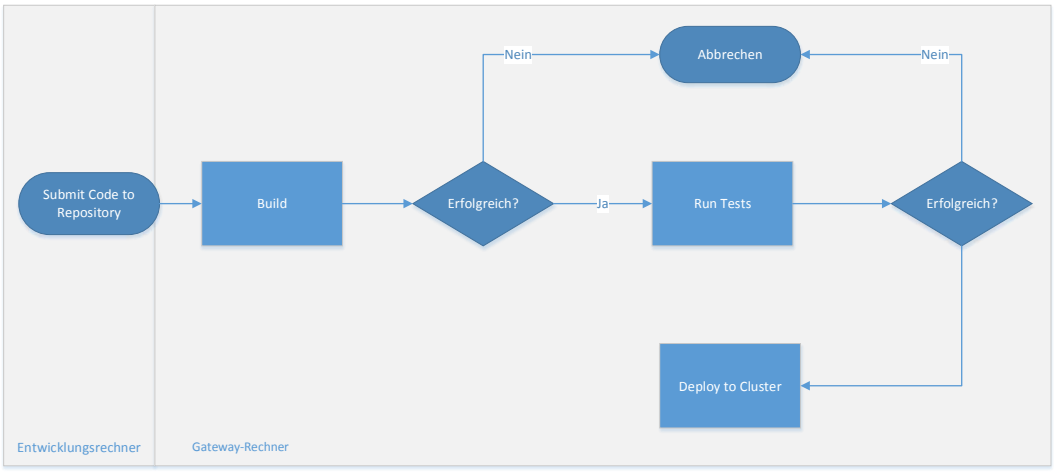

Abbildung 3.6: Einfache Continuous Deployment Pipeline

4 Schlussbetrachtung

4.1 Kritische Betrachtung der Ergebnisse

4.2 Ausblick und offene Punkte

Acronyme

API Application Programming Interface. 7

DSM Distributed Shared Memory. 7

RDD Resilient Distributed Dataset. 7, 8

Glossar

Master Host, der Verwaltungsaufgaben innerhalb eines Rechnerclusters übernimmt und dazu mit hierarchisch untergeordneten Rechnern kommuniziert. Zu den Aufgaben kann insbesondere das Verteilen von Arbeitsaufträgen oder Speicherblocks gehören. 5, 12

Read Evaluate Print Loop Pattern zum Erzeugen einer Konsole, die in einer Endlosschleife Eingaben liest, die auswertet und das Ergebnis wieder ausgibt. 4

Rechnercluster Vernetzter Verbund aus eigenständig lauffähigen Rechnern. 5, 12

RJ45 Achtpolige Modularsteckverbindung zur Datenübertragung. 13

Service Level Agreement Übereinkunft zwischen dem Anbieter und dem Nutzer eines Dienstes über dessen Qualität (z.B. Antwortzeiten, Durchsatz, Verfügbarkeit, etc.). 1

Worker Host, der als Arbeitsknoten in einem Rechnercluster dient. Falls nicht anders beschrieben ist hier ein Rechner gemeint, der seine Ressourcen einer Spark-Anwendung zur Verfügung stellt und mit seinem Festspeicher Teil eines verteilten Dateisystems ist. 5, 8, 12

Anhang

1 Installation der Plattform

2 Quellcode (Auszüge)

2.1 Realisierung einer einfachen Continuous Deployment Pipeline

post-receive-Hook von dem Git-Repository[1] der ModelBuilder-Komponente

```
#/bin/bash

export SPARK_HOME=/opt/spark/

# clean up previous build
rm -rf ~/autobuilds/model_builder
cd ~/autobuilds
git clone ~/git/model_builder
cd model_builder

# run build
sbt package
if [ $? -ne "0" ]; then exit 1; fi

# run test suite
sbt test
if [ $? -ne "0" ]; then exit 1; fi

# deploy to cluster
/opt/spark/bin/spark-submit --class "de.haw.bachelorthesis.dkirchner\
.ModelBuilder" --master spark://192.168.206.131:7077 --driver-memory\
256m --executor-memory 384m \
~/autobuilds/model_builder/ [...] /model-builder_2.10-1.0.jar\
hdfs://192.168.206.131:54310/user/daniel/user_emails_corpus1.txt\
<emailaccount> <passwort>
```

Listing 1: Primitive Continuous Deployment Pipeline. Beispiel: ModelBuilder

[1] https://git-scm.com/, abgerufen am 06.06.2015

3 Sonstiges

3.1 Einschätzung des theoretischen Spitzendurchsatzes von Mittelklasse-Servern

Um zu einer groben Einschätzung des möglichen Datendurchsatzes verschiedener Schnittstellen bei „Commodity Servern“ zu gelangen, wurden drei Systeme von großen Herstellern ausgewählt.
In der Grundkonfiguration kosten diese Systeme (zum Zeitpunkt dieser Arbeit) um die € 2000,- und lassen damit auf die Größenordnungen bei dem Datendurchsatz bestimmter Schnittstellen bei preisgünstigen Mehrzweck-Rechenknoten schließen.

Modell	Netzwerkschnittstelle	Festspeicher	Arbeitsspeicher
Dell PowerEdge R530	1Gb/s Ethernet	PCIe 3.0	DDR4
HP Proliant DL160 Gen8	1Gb/s Ethernet	PCIe 3.0	DDR3
System x3650 M5	1Gb/s Ethernet	PCIe 3.0	DDR4

Tabelle 1: Theoretische Spitzenleistungen bei Mehrzweck-Servern der 2000 Euro Klasse

Mit [Law14] und [Fuj] lassen sich grobe obere Abschätzungen errechnen, die in Tabelle 2.1 angegeben sind.

Literatur

[AM14] Vinod Kumar Vavilapalli Arun Merthy. *Apache Hadoop YARN*. 2014, S. 42.

[apa] apache. *Apache Blog*. Abgerufen am 11.04.2015. URL: https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces50.

[Apaa] Apache. *Apache Software Foundation*. Abgerufen am 06.06.2015. URL: http://apache.org.

[Apab] *Apache License, Version 2.0*. 2004. URL: https://www.apache.org/licenses/LICENSE-2.0.

[Arm+15] Michael Armbrust u. a. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, S. 1383–1394. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742797. URL: http://doi.acm.org/10.1145/2723372.2742797.

[BB+14] Jört Bartel, Arnd Böken u. a. *Big-Data-Technologien – Wissen für Entscheider*. 1914. URL: http://www.bitkom.org/files/documents/BITKOM_Leitfaden_Big-Data-Technologien-Wissen_fuer_Entscheider_Febr_2014.pdf.

[BL13] Michael Bevilacqua-Linn. *Functional Programming Patterns in Scala and Clojure*. The Pragmatic Programmers, LLC, 2013.

[Com] *Apache Spark Confluence*. Abgerufen am 11.04.2015. URL: https://cwiki.apache.org/confluence/display/SPARK/Committers.

[DG04] Jeffrey Dean und Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI* (2004).

[Fuj] *Fujitsu PRIMERGY SERVER - Basics of Disk I/O Performance*. 2011. URL: http://globalsp.ts.fujitsu.com/dmsp/Publications/public/wp-basics-of-disk-io-performance-ww-en.pdf.

[GGL03] Sanjay Ghemawat, Howard Gobioff und Shun-Tak Leung. *The Google File System*. Techn. Ber. Google, 2003.

[Gon+14] Joseph E. Gonzalez u. a. "GraphX: Graph Processing in a Distributed Dataflow Framework". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, S. 599–613. ISBN: 978-1-931971-16-4. URL: http://dl.acm.org/citation.cfm?id=2685048.2685096.

[Goo] Google. *Google Trends*. Abgerufen am 06.06.2015. URL: https://www.google.com/trends.

[HKK99] Max Hailperin, Barbara Kaiser und Karl Knight. *Concrete Abstractions: An Introduction to Computer Science Using Scheme*. 1999, 278ff.

[Iss] *Apache Spark Issue Tracker*. Abgerufen am 11.04.2015. URL: https://issues.apache.org/jira/browse/SPARK/?selectedTab=com.atlassian.jira.jira-projects-plugin:summary-panel.

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

[Law14] Jason Lawley. *Understanding Performance of PCI Express Systems*. 2014.

[NL91] Bill Nitzberg und Virginia Lo. "Distributed Shared Memory: A Survey of Issues and Algorithms". In: *Computer* 24.8 (Aug. 1991), S. 52–60. ISSN: 0018-9162. DOI: 10.1109/2.84877. URL: http://dx.doi.org/10.1109/2.84877.

[Pag01] Lawrence Page. *Method for node ranking in a linked database*. US Patent 6,285,999. 2001. URL: http://www.google.com/patents/US6285999.

[Ras] Raspbian. *Raspbian Operating System*. Abgerufen am 06.06.2015. URL: http://www.raspbian.org.

[Spa] *Spark Submission Guide*. abgerufen am 12.04.2015. URL: https://spark.apache.org/docs/1.3.0/submitting-applications.html.

[SR14] Dilpreet Singh und Chandan Reddy. "A survey on platforms for big data analytics". In: *Journal of Big Data* (2014).

[SW14] Jasson Venner Sameer Wadkar Madhu Siddalingaiah. *Pro Apache Hadoop*. 2014, S. 1.

[Ubu] Ubuntu. *Ubuntu Operating System*. Abgerufen am 06.06.2015. URL: http://www.ubuntu.com.

[ZC+12] Matei Zaharia, Mosharaf Chowdhury u. a. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *NSDI* (2012).

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

Hamburg, 1. Januar 2345 Daniel Kirchner