Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Daniel Kirchner

Skalierbare Datenanalyse mit Apache Spark

Implementation einer Text-Mining-Anwendung und Betrieb auf einem Low-End-Cluster

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Daniel Kirchner

Skalierbare Datenanalyse mit Apache Spark

Implementation einer Text-Mining-Anwendung und Betrieb auf einem Low-End-Cluster

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 Implementation einer Text-Mining-Anwendung und Betrieb auf einem Low-End-Cluster

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 untersucht das Verhalten von Spark auf einem Cluster mit minimalem Leistungsprofil. Grundlage dieser Untersuchung ist ein realitätsnaher Anwendungsfall, der Sparkmodule für Batch-Processing und Streaming kombiniert.
Daniel Kirchner

Title of the paper
Scalable Data Analysis with Apache Spark

Keywords
keyword 1, keyword 2

Abstract
This document …

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Motivation . . . 1
- 1.2 Ziel dieser Arbeit . . . 2

2 Vorstellung von Apache Spark **4**
- 2.1 Überblick . . . 5
- 2.2 Kernkonzepte . . . 7
  - 2.2.1 Resilient Distributed Datasets . . . 7
  - 2.2.2 Scheduling/Shuffling . . . 13
  - 2.2.3 Anwendungsdeployment und -lebenszyklus . . . 16
  - 2.2.4 Zusammenfassung und Bewertung . . . 17
- 2.3 Standardbibliotheken . . . 17
  - 2.3.1 Dataframes/Spark SQL . . . 18
  - 2.3.2 MLlib . . . 20
  - 2.3.3 Streaming . . . 20
  - 2.3.4 GraphX . . . 20
- 2.4 Betrieb und Security . . . 20
- 2.5 Spark im Kontext von Parallelisierungspattern . . . 20
- 2.6 Entwicklergemeinschaft . . . 20
- 2.7 Auswahl verwandter Produkte . . . 21

3 Beispielanwendung von Spark zur Datenanalyse **23**
- 3.1 Vorstellung des Anwendungsfalls . . . 23
  - 3.1.1 Anforderungen . . . 23
- 3.2 Technische Rahmenbedingungen . . . 25
- 3.3 Hardwareumgebung . . . 27
- 3.4 Lösungsskizze . . . 27
- 3.5 Ergebnisse und Bewertung . . . 33

4 Schlussbetrachtung **35**
- 4.1 Diskussion der Ergebnisse . . . 35
- 4.2 Ausblick und offene Punkte . . . 36

Acronyme **38**

Glossar **39**

Anhang **40**
1 Installation der Plattform 41
2 Quellcode/Skripte (Auszüge) 41
2.1 Performance-Messungen 41
2.2 Monitoring 41
2.3 Realisierung einer einfachen Continuous Deployment Pipeline 42
3 Konfigurationen 43
4 Sonstiges 44
4.1 Einschätzung des theoretischen Spitzendurchsatzes von Mittelklasse-Servern 44

Abbildungsverzeichnis

1.1 Google Trends 2

2.1 Verteilungsdiagramm einer typischen Sparkinstallation 6
2.2 Resilient Distributed Datasets aus Verteilungssicht 8
2.3 RDD Lineage vor Aktion (gestrichelte Linie steht für *nicht initialisiert*) 9
2.4 RDD Lineage nach Aktion 9
2.5 RDD Lineage nach Aktion und mit Persist() 10
2.6 Resilient Distributed Datasets mit Datenquelle aus Verteilungssicht 12
2.7 Einfacher Abhängigkeitsbaum eines RDD 14
2.8 Gerichteter azyklischer Graph der Abhängigkeiten auf Partitionen 14
2.9 Stages als Untermenge des Ahängigkeitsgraphen von Partitionen 15
2.10 Application Deployment im Client Modus 17
2.11 Application Deployment im Cluster Modus 18
2.12 Application Deployment Prozess im Client Modus (vereinfacht) 19
2.13 Aktivität auf den offiziellen Spark Mailinglisten 21

3.1 Anwendungsfalldiagramm der Demo-Applikation 24
3.2 Hardwareumgebung des Programms zur Tweetanalyse 25
3.3 Datenzentrierte Sicht auf die Komponenten 28
3.4 Innenansicht der Batch-Layer Komponente 29
3.5 Innenansicht der Streaming-Layer Komponente 30
3.6 Verteilungssicht auf die Demo App 32
3.7 Komponentendiagramm des Demo App Packages 33
3.8 Einfache Continuous Deployment Pipeline 33

Tabellenverzeichnis

2.1 Theoretische Spitzenleistungen bei Mittelklasse-Servern 7

3.1 Maximaler Netzwerkdurchsatz[1] 26
3.2 Festspeicher Lese-/Schreibdurchsatz dell01 (Master)[2] 26
3.3 Festspeicher Lese-/Schreibdurchsatz pi00 (Worker) 26
3.4 Übersicht ausgewählter Datenquellen für Spark 27
3.5 Übersicht verfügbarer Clustermanager für Spark 27
3.6 Skalierungsverhalten des ModelBuilders 34

1 Theoretische Spitzenleistungen bei Mehrzweck-Servern der 2000 Euro Klasse 44

Listings

2.1 Word Count in der Spark Konsole 4
2.2 Map-Methode aus org.apache.spark.rdd.RDD v1.3.0 10
2.3 foreach-Methode aus org.apache.spark.rdd.RDD v1.3.0 11
2.4 Beispiel: Minimaler Partitionierer 12

3.1 Bewertung von Tweets 30

1 Messung der Festplattenperformance - Beispiel: Schreiben einer 512MB Datei 41
2 Messung der Netzwerkperformance 41
3 Monitoring des Clusters (Betriebssystem), Beispiel ModelBuilder 41
4 Einfache Continuous Deployment Pipeline. Beispiel: ModelBuilder 42
5 hdfs-site.xml (Auszug): Beispiel mit Replikationsfaktor 1 und Blockgröße 32MB 43
6 spark-defaults.conf (Auszug) 43

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:

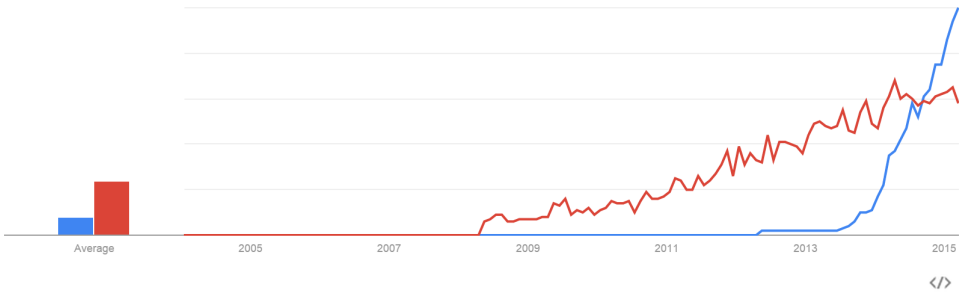


Abbildung 1.1: Suchanfragen zu „Apache Spark“ (*blau*) und „Apache Hadoop“ (*rot*), Stand 24.03.2015 [Goo]

1.2 Ziel dieser Arbeit

Das Ziel dieser Arbeit ist es den Betrieb einer realitätsnahen Spark-Anwendung auf einem Cluster mit Hardware am unteren Ende des Leistungsspektrums zu begutachten.
Dabei wird zunächst ein umfassender Überblick der grundlegenden Konzepte von Spark gegeben. Anschließend wird ein Anwendungsfall aus dem Bereich des Text-Mining vorgestellt und

dessen Realisierung vom Entwurf bis zum Betrieb erläutert.

Der theoretische Teil dieser Arbeit umfasst

- eine Einführung in die grundlegenden Konzepte von Apache Spark
- eine kurze Einordnung von Spark durch den Vergleich mit ähnlichen Anwendungen

Der praktische Teil dieser Arbeit umfasst

- die Implementation einer hybriden Anwendung mit einer Echtzeitkomponente (Spark Streaming Library) und einer Batch-Komponente (Spark Machine Learning Library).
- den Betrieb dieser Anwendung auf einem Cluster mit Hardware am unteren Ende des Leistungsspektrums.

Apache Spark ist überwiegend in der Programmiersprache Scala[1] 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.

[1] http://www.scala-lang.org/, abgerufen am 03.03.2015

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 4.1).

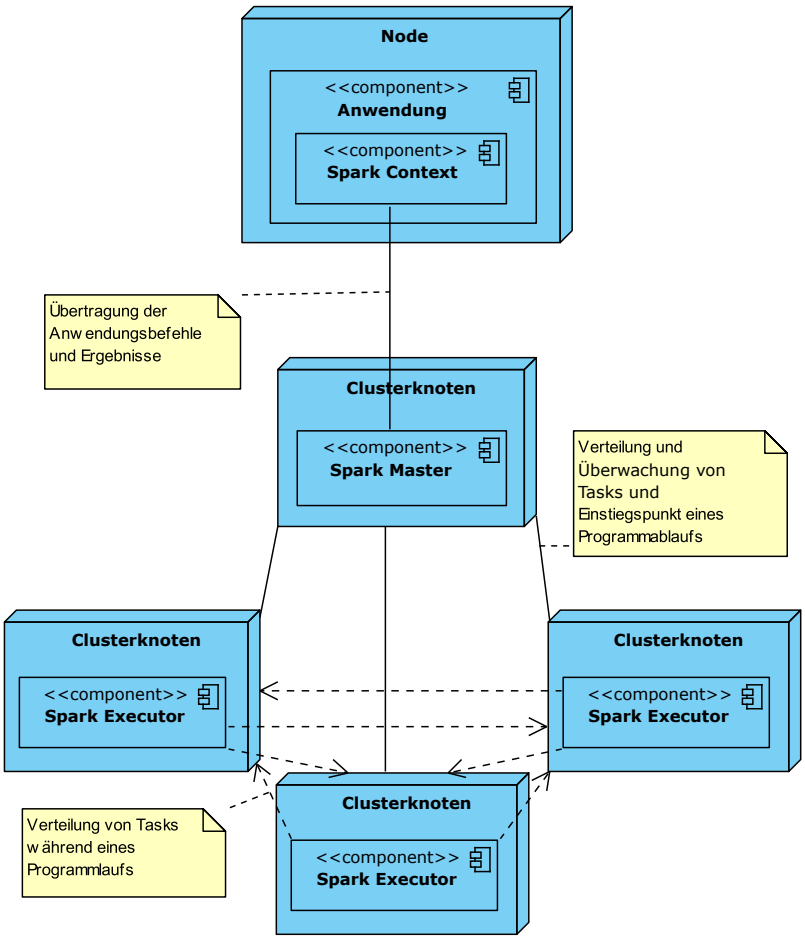


Abbildung 2.1: Verteilungsdiagramm einer typischen Sparkinstallation

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).

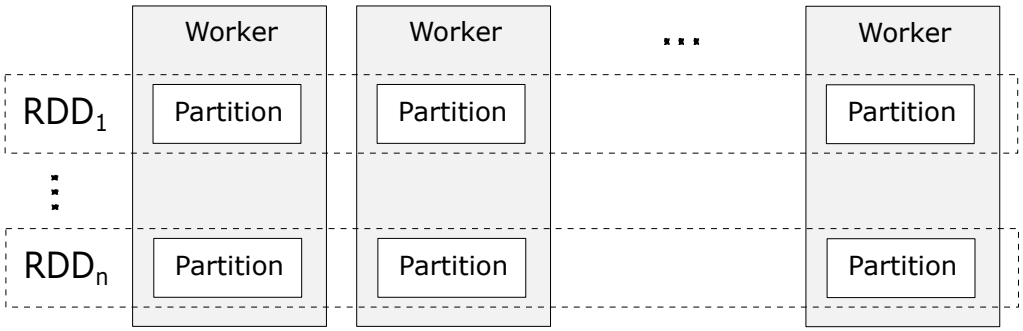


Abbildung 2.2: Resilient Distributed Datasets aus Verteilungssicht

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“:

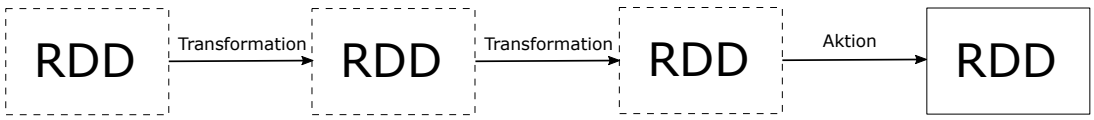


Abbildung 2.4: RDD Lineage nach Aktion

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:

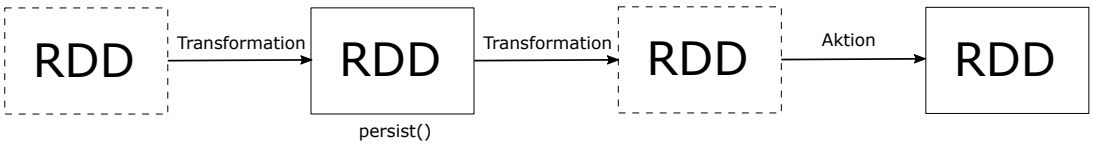


Abbildung 2.5: RDD Lineage nach Aktion und mit Persist()

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)

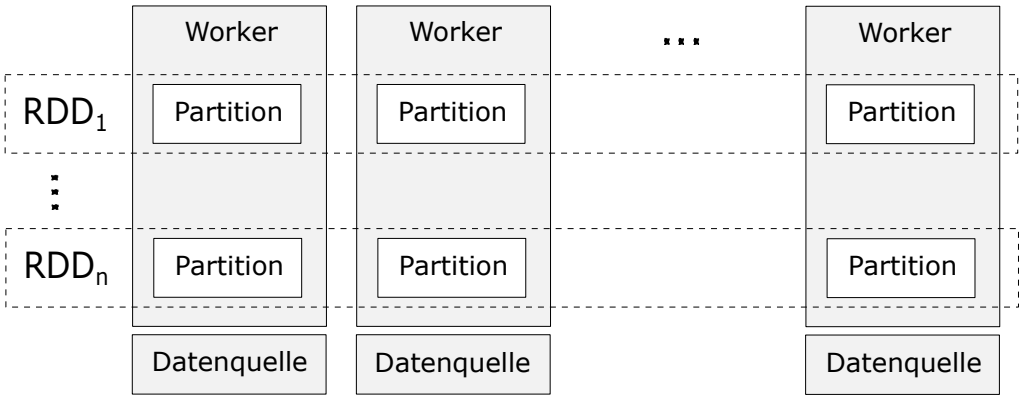


Abbildung 2.6: Resilient Distributed Datasets mit Datenquelle aus Verteilungssicht

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.

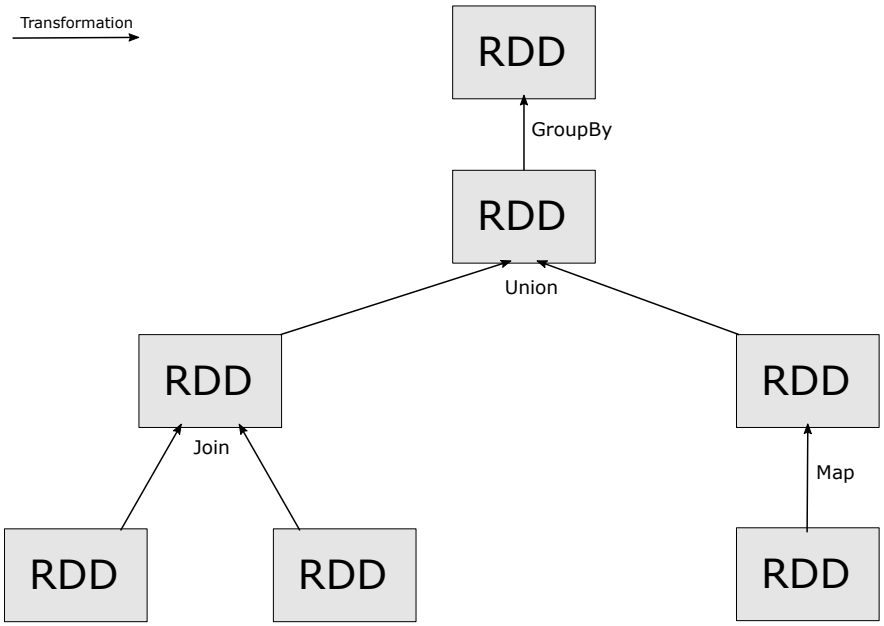


Abbildung 2.7: Einfacher Abhängigkeitsbaum eines RDD

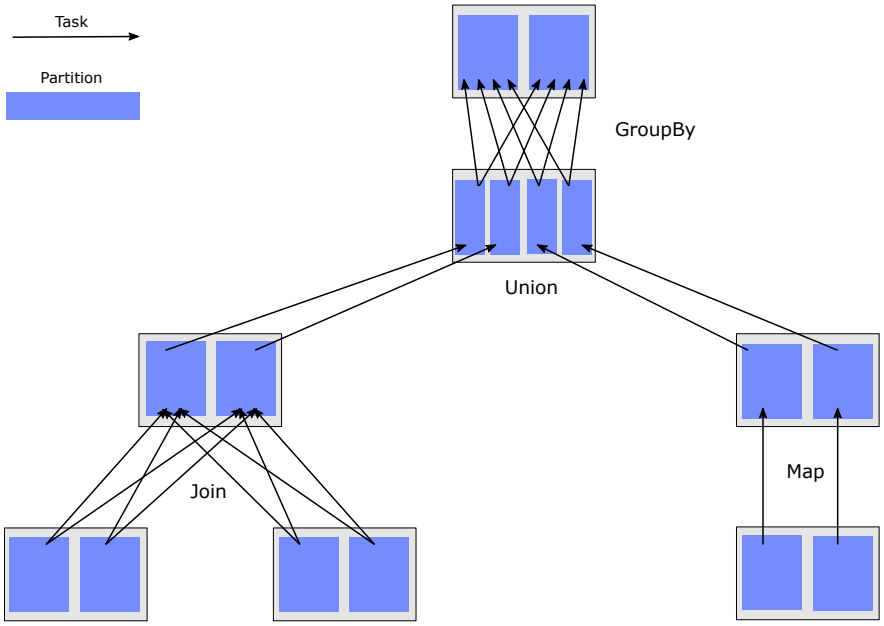


Abbildung 2.8: Gerichteter azyklischer Graph der Abhängigkeiten auf Partitionen

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

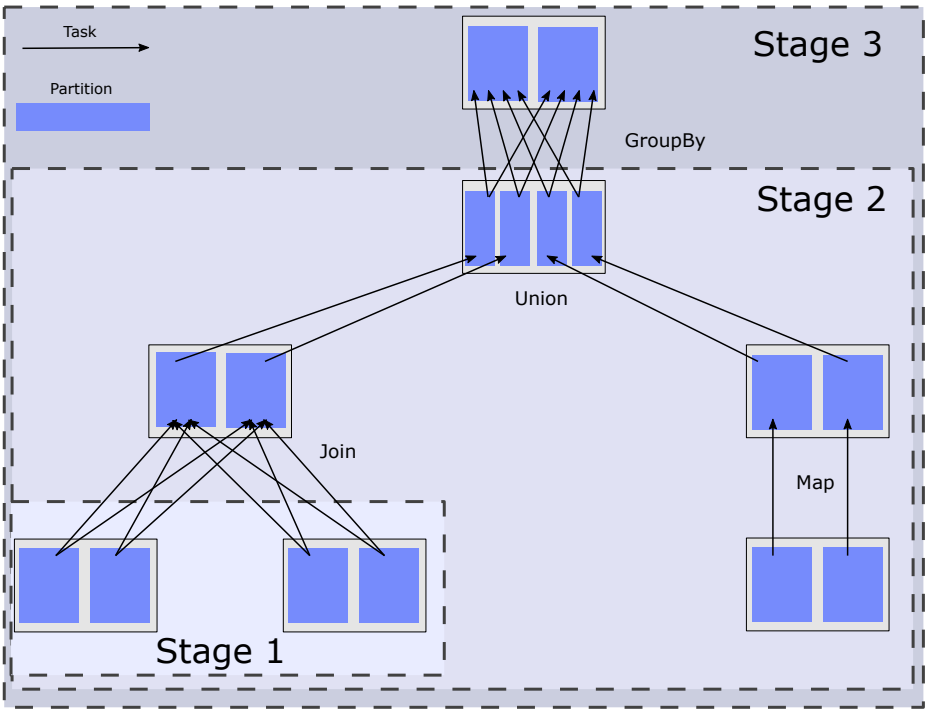


Abbildung 2.9: Stages als Untermenge des Ahängigkeitsgraphen von Partitionen

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).

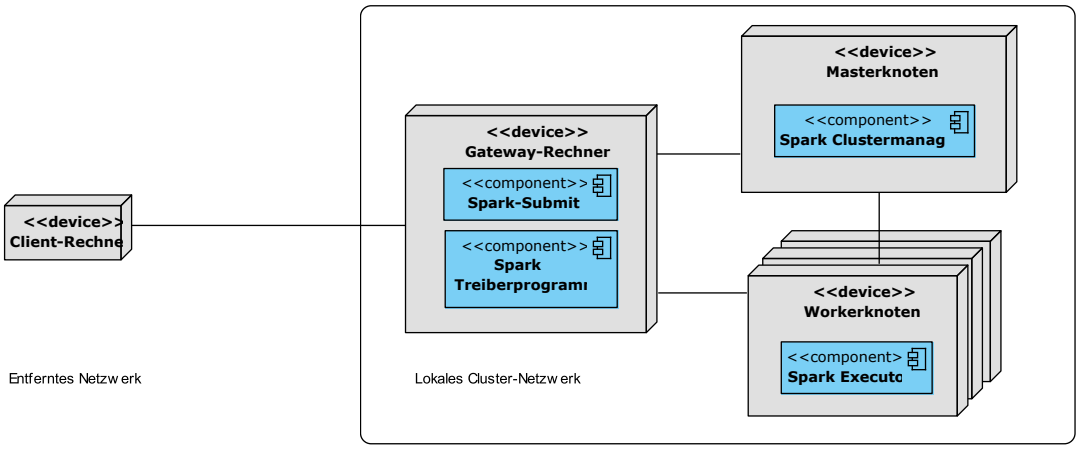


Abbildung 2.10: Application Deployment im Client Modus

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.

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

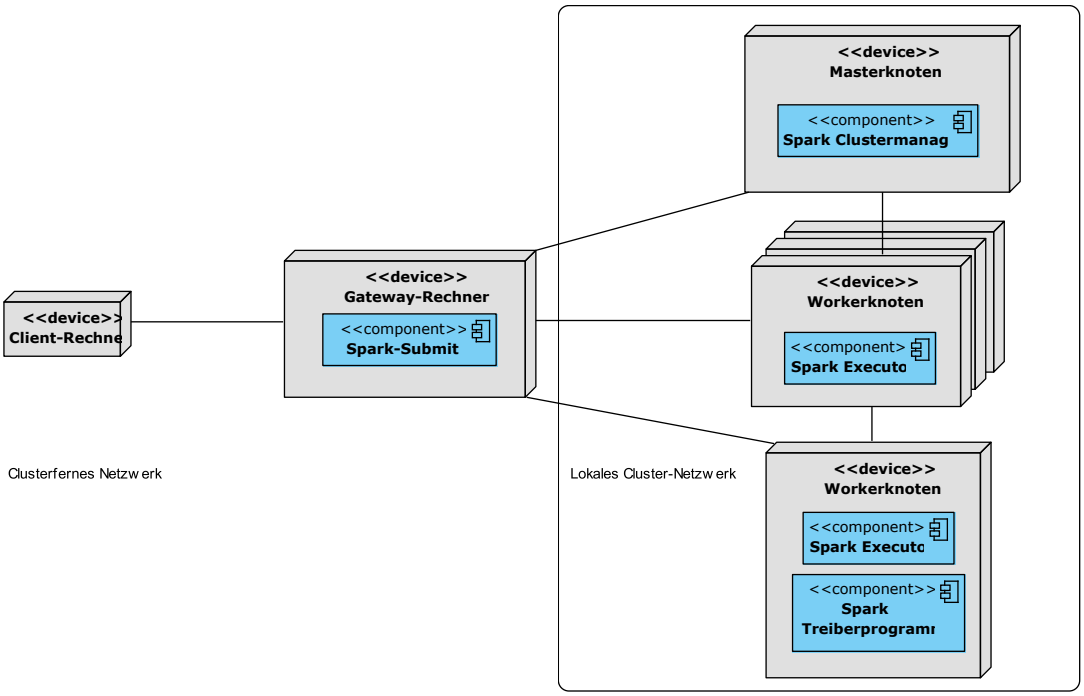


Abbildung 2.11: Application Deployment im Cluster Modus

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

werden in diesem Abschnitt erläutert.

2.3.1 Dataframes/Spark SQL

Die Spark SQL Bibliothek bietet Schnittstellen für den Zugriff auf strukturierte Daten in Form klassischer SQL-Abfragen. Dazu wird eine Abstraktion der RDDs eingeführt. Diese Abstraktion sind Dataframes wie sie aus der Programmiersprache R[5] bekannt sind.

Data Frames: Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris.

[5]http://www.r-project.org/, abgerufen am 20.05.2015

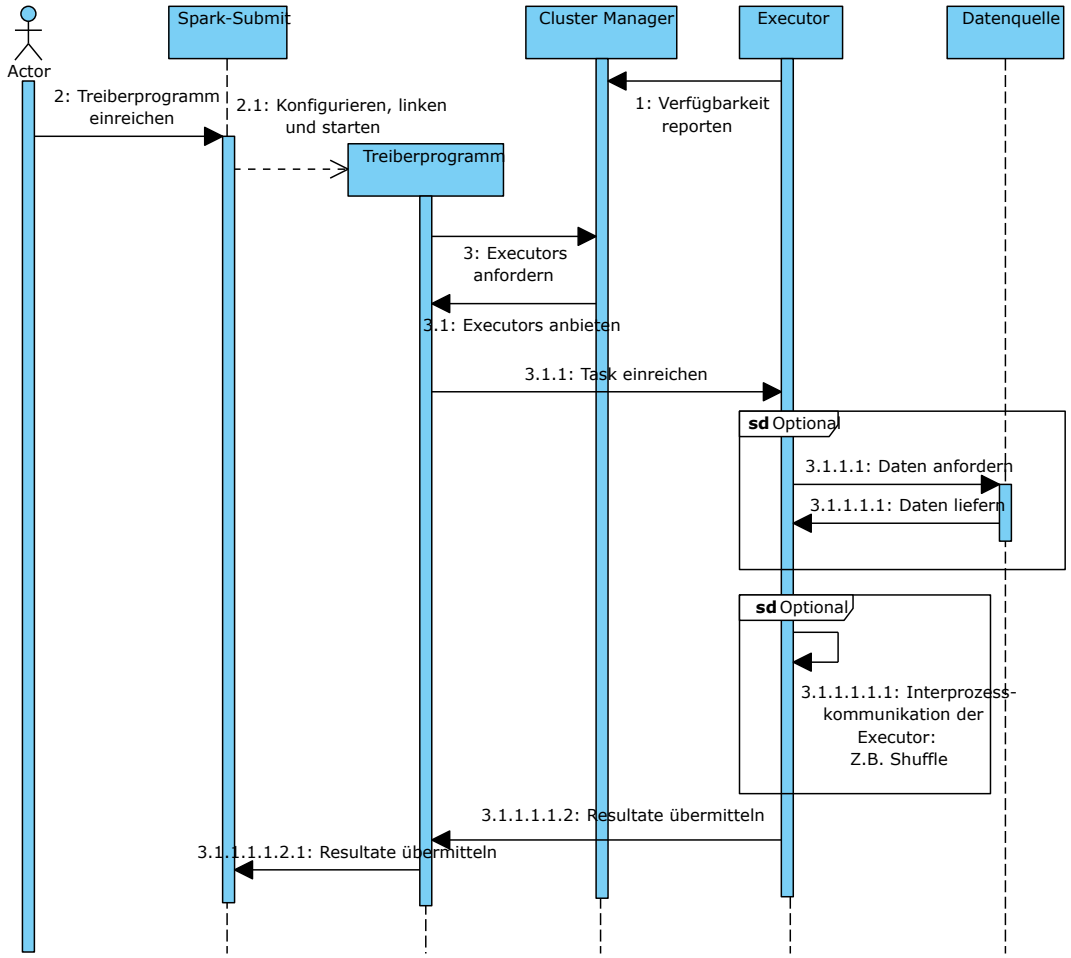


Abbildung 2.12: Application Deployment Prozess im Client Modus (vereinfacht)

Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

[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 Entwickler 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.

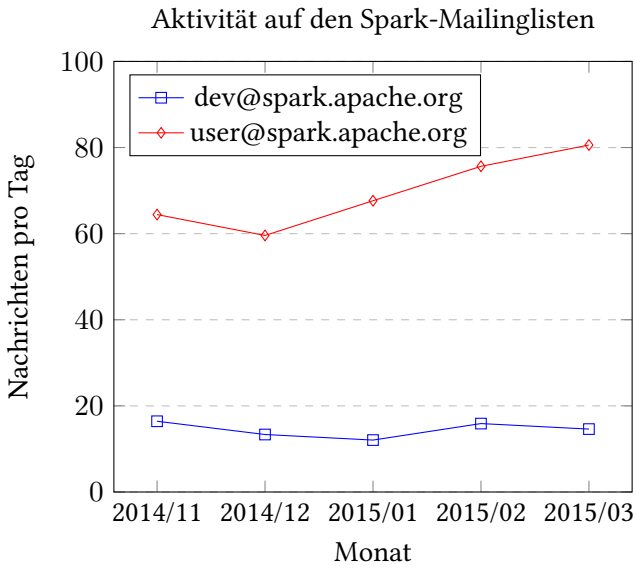


Abbildung 2.13: Aktivität auf den offiziellen Spark Mailinglisten

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 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 Vorstellung des Anwendungsfalls

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

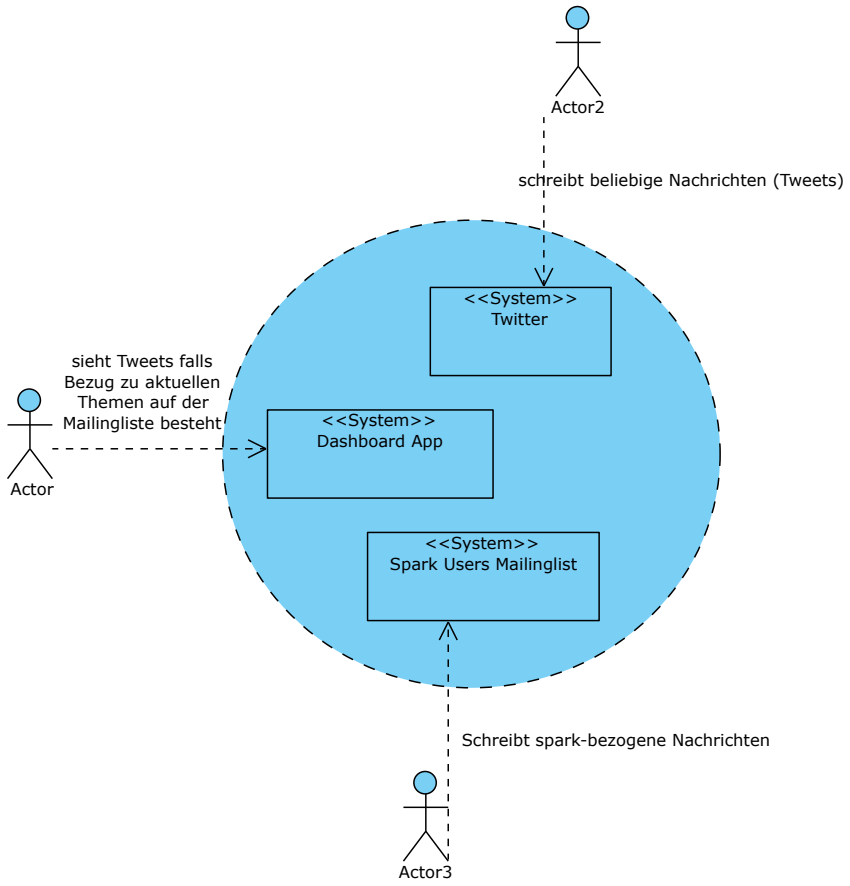


Abbildung 3.1: Anwendungsfalldiagramm der Demo-Applikation

- **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

3.2 Technische Rahmenbedingungen

Als Versuchsumgebung dient ein Rechnercluster aus vier identischen Workern und einem speziellen Masterknoten (Abb. 3.2).

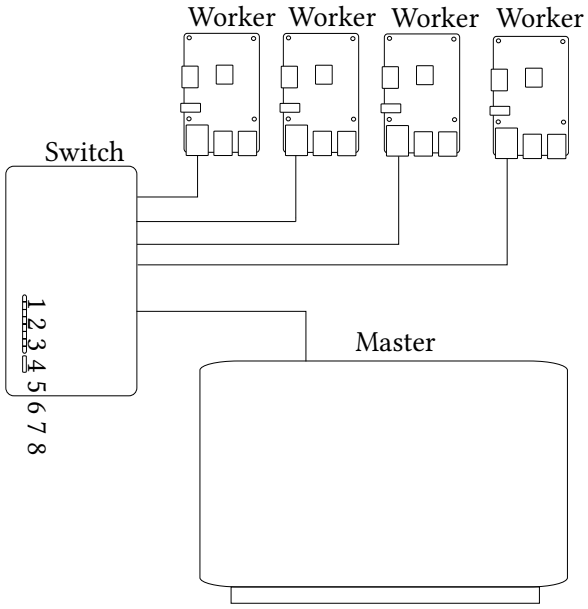


Abbildung 3.2: Hardwareumgebung des Programms zur Tweetanalyse

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.

Worker → Worker	Worker → Master
94,4 MBit/s	94,4 MBit/s

Tabelle 3.1: Maximaler Netzwerkdurchsatz[5]

Operation	Blockgröße (MB)	Durchsatz (MB/s)
Lesen	1	17,2
Lesen	16	22,1
Lesen	64	31,8
Lesen	512	31,2
Schreiben	1	5,0
Schreiben	16	17,2
Schreiben	64	26,1
Schreiben	512	25,8

Tabelle 3.2: Festspeicher Lese-/Schreibdurchsatz dell01 (Master)[6]

Operation	Blockgröße (MB)	Durchsatz (MB/s)
Lesen	1	66,4
Lesen	16	78,1
Lesen	64	42,0
Lesen	512	9,2
Schreiben	1	17,9
Schreiben	16	18,4
Schreiben	64	18,4
Schreiben	512	18,4

Tabelle 3.3: Festspeicher Lese-/Schreibdurchsatz pi00 (Worker)

[5] Siehe Anhang Listing 2
[6] Gemessen mit dd. Siehe Anhang Listing 1

3.3 Hardwareumgebung

3.4 Lösungsskizze

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

Tabelle 3.4: 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 3.5: 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*).

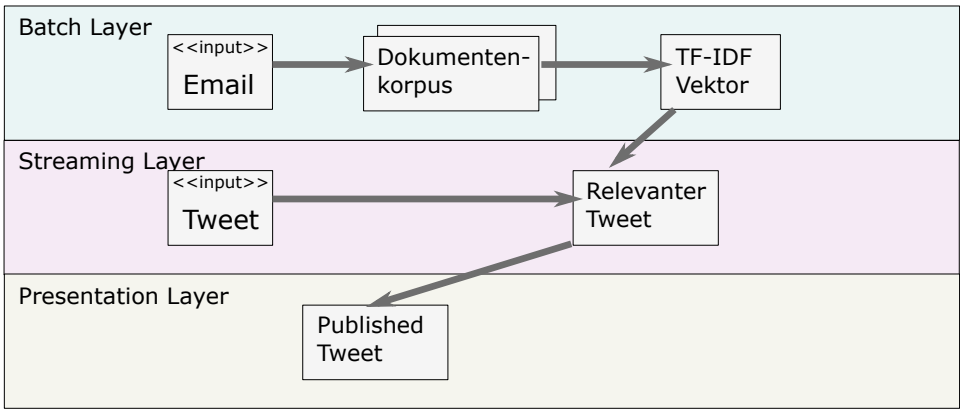


Abbildung 3.3: Datenzentrierte Sicht auf die Komponenten

Batch Layer In dieser Schicht wird die Verarbeitung von Emails aus der Spark-User-Mailingliste zu einem Modell von relevanten Wörtern geleistet.

Dazu werden eingehende Emails zunächst archiviert und anschließend mithilfe des Corpus aller bisher archivierten Emails und einer Untermenge von n zuletzt empfangenen Emails eine Bewertung der vorkommenden Wörtern vorgenommen.

Diese Bewertung soll das Maß für die Relevanz eines Wortes in der betrachteten Menge der letzten n Emails sein. Um das zu erreichen, wird in mehreren Schritten ein TF-IDF Vektor über

die Wörter dieser Nachrichten erzeugt.

TF-IDF steht für *Term Frequency - Inverse Document Frequency* ([SJ88]). Dieses Verfahren bewertet die Relevanz eines Wortes für einen Text nach der Häufigkeit dieses Wortes in dem Text (*Term Frequency*). Die Bewertung eines Wortes wird jedoch abgeschwächt je häufiger es in einem Textkorpus vorkommt (*Inverse Document Frequency*).

Eine Implementation dieses Verfahrens ist in der Spark Standardbibliothek *MLLib* in dem Bereich *Feature Extraction* verfügbar[7].

In diesem Anwendungsfall gilt:

- Email-Bodies[8] werden jeweils als Dokumente verarbeitet
- Das Archiv aller Email-Bodies ist der Textkorpus[9]

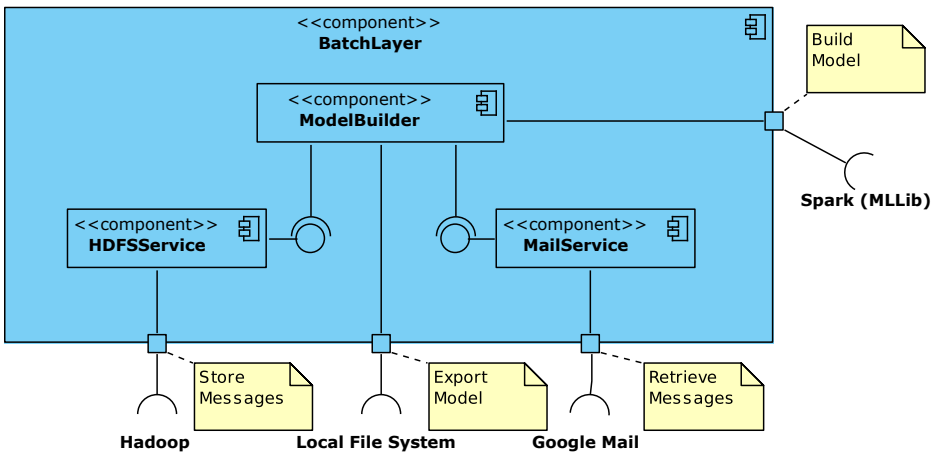


Abbildung 3.4: Innenansicht der Batch-Layer Komponente

[7] https://github.com/apache/spark/tree/branch-1.3/mllib/src/main/scala/org/apache/spark/mllib/feature, abgerufen am 04.06.2015

[8] Textbereich einer Email

[9] Ein unabhängiger Textkorpus (wie etwa Wikipedia) würde wahrscheinlich bessere Ergebnisse liefern. Beispielsweise würden Begriffe wie *Spark* und *Apache* durch ihre geringere Häufigkeit in dessen Dokumenten höher bewertet werden. Für diese Demonstration bietet aber die Gesamtheit der Email-Texte ausreichend Volumen und zeitliche Dynamik und verringert außerdem die Komplexität der Implementierung.

Streaming Layer In dieser Schicht werden die Nachrichten aus dem Twitter-Datenstrom bewertet und gefiltert.

Dazu wird über die Spark-Komponente *TwitterUtils*[10] eine Verbindung über HTTP zu einem Twitter-Endpunkt aufgebaut[11]. Über diese Verbindung werden - bei unpriveligiertem Zugriff - etwa zehn Tweets pro Sekunde über einen dauerhaften Datenstrom zur Verfügung gestellt.

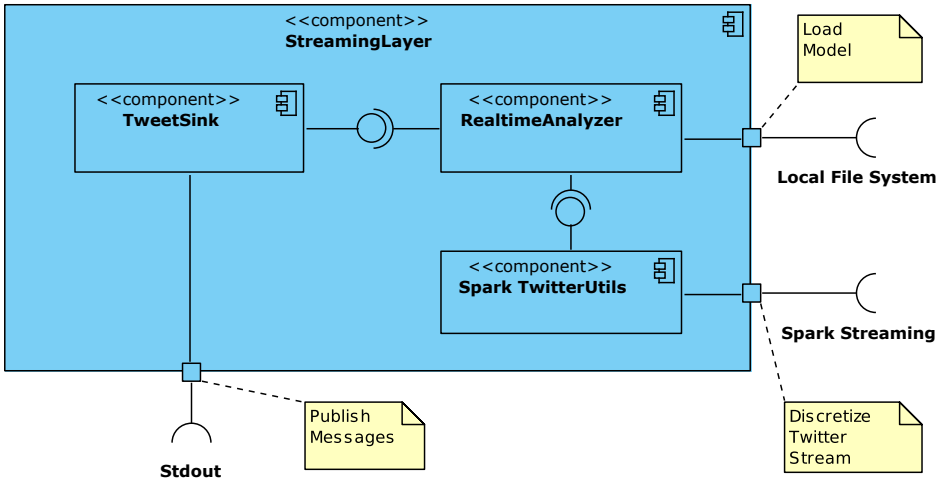


Abbildung 3.5: Innenansicht der Streaming-Layer Komponente

In der Komponente RealtimeAnalyzer wurden vor dem Start des Datenstroms Funktionen zur Bewertung einzelner Tweets registriert (VERWEIS?).

Die Funktion `ScoreTweets` zur Bewertung einzelner Tweets

$$\texttt{ScoreTweets} : Tweets \longrightarrow \mathbb{R} \times \text{Tweets}$$

$$tweet \mapsto (score, tweet)$$

wird dabei wie in Listing 3.1 dargestellt implementiert:

```
1 // Split text string into single words
2 val splitTweets = {
3   stream.map(status =>
4     (status.getText.split("␣"), status)
```

[10] TwitterUtils sind Teil der Spark-Standardbibliothek org.apache.spark.streaming.twitter
[11] https://dev.twitter.com/streaming/overview/connecting, abgerufen am 01.06.2015

```
  )
}

// calculate score for each word, then sum the scores and normalize
val scoredTweets = {
  splitTweets.map(splitTweet => {
    (splitTweet._1.map(word =>
      broadcastScores.value.apply(
         hashingTF.indexOf(word.toLowerCase
           .replaceAll("[^a-zA-Z0-9]", "_")))
     ).sum./(splitTweet._2.getText.split("_").length),
      splitTweet._2
      )}
  )
}
```

Listing 3.1: Bewertung von Tweets

Folgendes geschieht Folgendes:

1. Extrahieren des Textinhaltes (Metadaten werden ignoriert)
2. Zerlegen des Textes in einzelne Wörter
3. Normalisieren der Wörter durch Umwandeln in Kleinbuchstaben und Entfernen von Sonderzeichen
4. Index der einzelnen Wörter im TF-IDF-Vektor berechnen und dem eingetragenen Score zuweisen (Map)
5. Scores aller Wörter eines Tweets summieren (Reduce)
6. Normalisieren des Scores per Division durch die Anzahl aller Wörter des Tweets
7. Rückgabe des Tupels *(Score, Status)*

Anschließend werden die erzeugten Tupel nach der Größe des erreichten Scores gefiltert und die verbleibenden Status-Texte an den TweetSink weitergegeben.

Presentation Layer

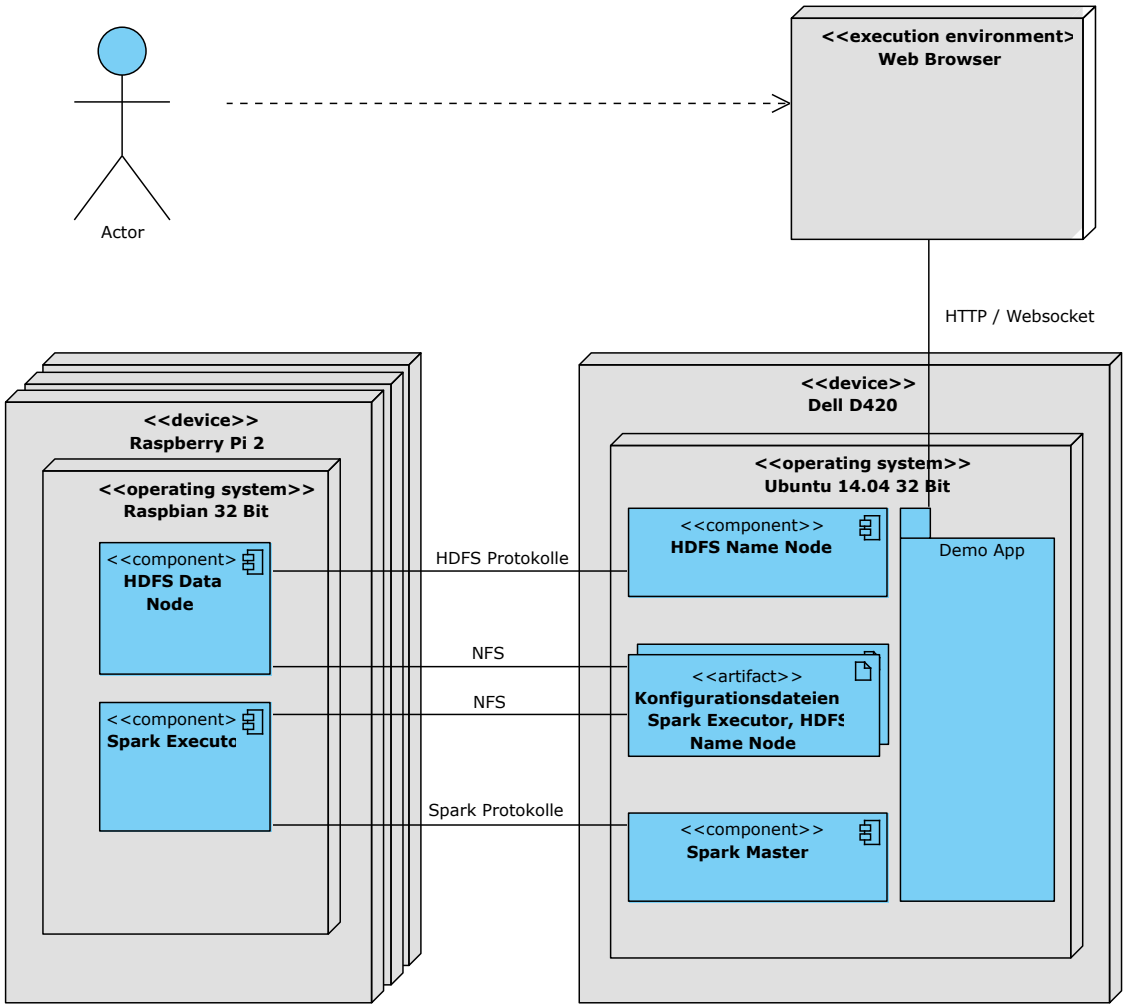


Abbildung 3.6: Verteilungssicht auf die Demo App

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. 3.8) 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.

Diese Pipeline ist durch ein *post-receive*-Skript in den jeweiligen Repositories der Komponenten auf dem Gateway-Rechner realisiert (Beispiel im Anhang 2.3).

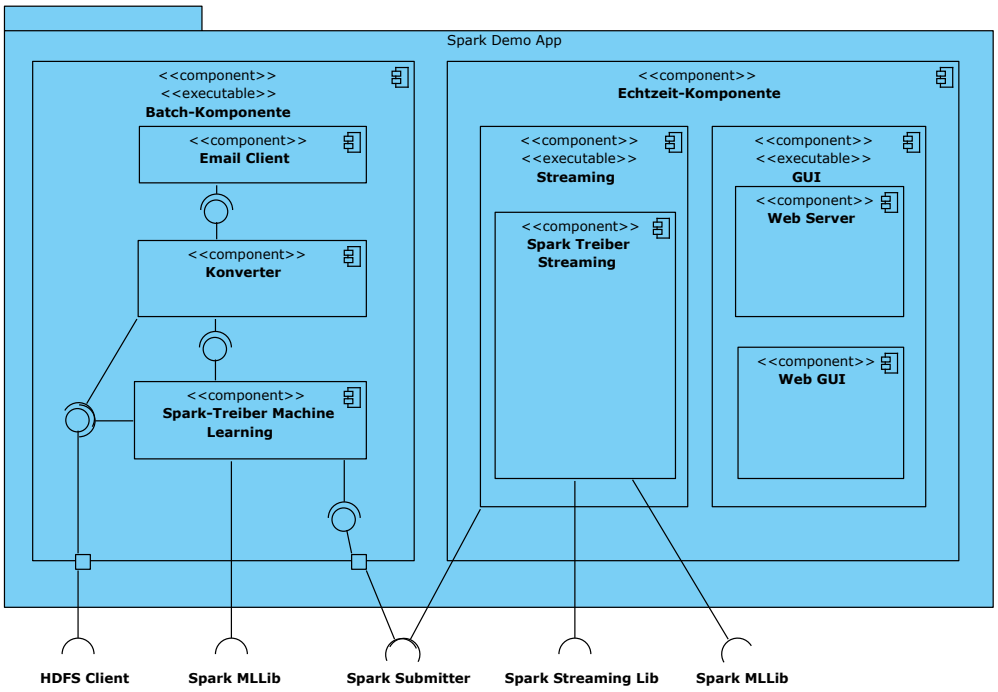


Abbildung 3.7: Komponentendiagramm des Demo App Packages

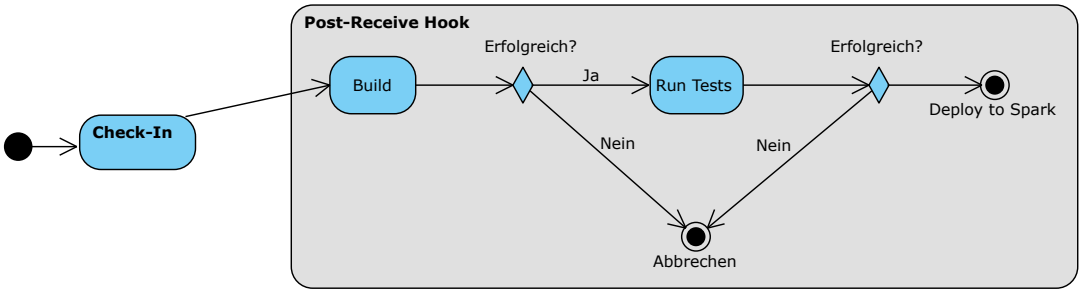


Abbildung 3.8: Einfache Continuous Deployment Pipeline

3.5 Ergebnisse und Bewertung

Zur Beurteilung des Laufzeitverhaltens wird die Anwendung in verschiedenen Konfigurationen gestartet. Dabei wird jeweils das Verhalten der einzelnen Komponenten und des gesamten Systems erfasst und zur späteren Auswertung gespeichert.

Die zur Laufzeit erfassten Daten sind

1. Systemgrößen auf jedem aktiven Knoten(1 Messpunkt pro Sekunde)

CPU Nutzung (User, Idle, System, ...)

IO Festplatte (Lesen, Schreiben)

Swap (Benutzt, Frei)

Speicher (Benutzt, Frei, Cached, ...)

IO Netzwerk (Gesendet, Empfangen)

2. Sparkspezifische Größen (1 Messpunkt alle zwei Sekunden)

   genutzte Cores

   aktive Stages

   parallele Receiver (bei Streaming)

   u.a.[12]

Als Konfigurationsparameter für die Testläufe werden verwendet

1. die Blockgröße des Hadoop-Dateisystems (HDFS) : 32MB, 64MB, 128MB
2. der Replikationsfaktor der Blöcke auf HDFS : 1, 2, 3, 4
3. die Anzahl der Worker : 1, 2, 4

Um eine Vergleichbarkeit der Testläufe untereinander zu erreichen werden folgende Größen festgelegt:

- Größe des Textkorpus : 1,5 GB
- Fenstergröße des TF-IDF Vektors : 500 Nachrichten
- Erlaubte Cores Pro Executor : 4
- Arbeitsspeicher pro Executor : 384 MB

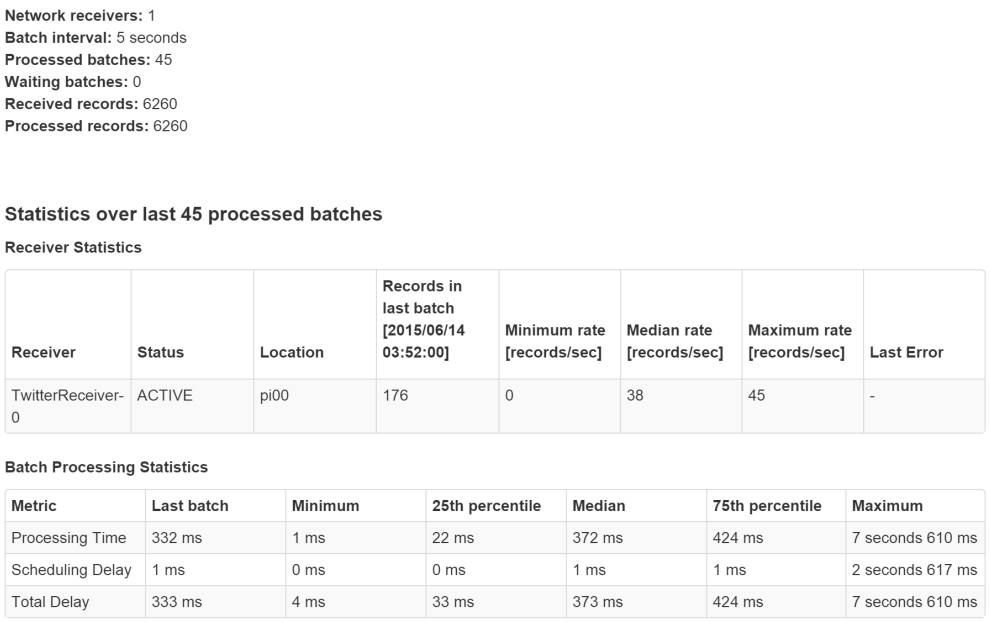
Die **Größe des Textkorpus** ergibt sich aus der Überlegung eine Datei zu Verarbeiten, die größer als der Arbeitsspeicher eines einzelnen Workers ist, theoretisch aber noch in den geteilten Speicher von vier Executor-Prozessen passt.
Die **Fenstergröße des TF-IDF Vektors** ist willkürlich gewählt. Der Wert 500 entspricht der Anzahl von Email-Nachrichten aus der Spark-User-Mailingliste.
Die erlaubten **Cores pro Executor** entsprechen genau den Verfügbaren Cores auf einem

[12] Bei den sparkspezifischen Messgrößen gibt es noch weitere, die nicht in die Auswertungen einbezogen wurden.

Worker. Damit ist eine volle Ausnutzung der verfügbaren Rechenleistung gewährleistet. Der erlaubte **Arbeitsspeicher von 384MB pro Executor** lässt bei 1000MB Gesamtspeicher pro Knoten noch Spielraum für das Betriebssystem, den Worker-Prozess und insbesondere den HDFS Data Node für Caching von Dateiblöcken.

Um die Testläufe darüber hinaus unter kontrollierten Bedingungen zu starten, wurde bei jeder Änderung der Konfigurationsparameter das Hadoop-Dateisystem formatiert und der Spark-Cluster zurückgesetzt (inklusive Terminierung und Neustart aller zugehörigen Prozesse).

Eine laufende Anwendung startet auf dem Host des Treibers eine Weboberfläche (siehe Abb. ??), die den aktuellen Verlauf des Programms verfolgen lässt. Zusätzlich werden Daten über Verteilung



Network receivers: 1
Batch interval: 5 seconds
Processed batches: 45
Waiting batches: 0
Received records: 6260
Processed records: 6260

Statistics over last 45 processed batches

Receiver Statistics

Receiver	Status	Location	Records in last batch [2015/06/14 03:52:00]	Minimum rate [records/sec]	Median rate [records/sec]	Maximum rate [records/sec]	Last Error
TwitterReceiver-0	ACTIVE	pi00	176	0	38	45	-

Batch Processing Statistics

Metric	Last batch	Minimum	25th percentile	Median	75th percentile	Maximum
Processing Time	332 ms	1 ms	22 ms	372 ms	424 ms	7 seconds 610 ms
Scheduling Delay	1 ms	0 ms	0 ms	1 ms	1 ms	2 seconds 617 ms
Total Delay	333 ms	4 ms	33 ms	373 ms	424 ms	7 seconds 610 ms

Abbildung 3.9: Spark-Dashboard des Realtime Analyzers - Statistics

Bewertung und Probleme *Einfachheit Skalierbarkeit Erweiterbarkeit Robustheit Sicherheit Wartbarkeit*

Anzahl Worker	Replikationslevel	HDFS Blockgröße (MB)	Laufzeit (Sekunden)
1	1	32	697
1	1	64	750
1	1	128	850
2	2	32	366
2	2	64	448
2	2	128	615
4	3	32	210
4	4	32	209
4	2	64	301
4	3	64	359
4	2	128	432
4	3	128	433

Tabelle 3.6: Skalierungsverhalten des ModelBuilders

4 Schlussbetrachtung

4.1 Diskussion der Ergebnisse

Es wurde eine Anwendung entworfen und implementiert, die einen Echtzeitdatenstrom nach vorgegebenen Kriterien analysiert und dabei dynamisch auf geänderte Vorgaben reagieren kann.
Diese Anwendung wurde anschließend als Testfall für den Betrieb einer Apache Spark/Hadoop-Umgebung auf einem Low-End-Hardware-Cluster aus Raspberry Pis genutzt und das Laufzeitverhalten untersucht.

Dieser Versuch war erfolgreich. Es ist möglich einen Spark/Hadoop-Cluster auf der beschriebenen Hardware zu betreiben und die genannte Anwendung stabil zu betreiben.

Dabei hat sich die Hardware sogar als leistungsfähiger als nötig erwiesen. Zwar profitiert insbesondere die Batch-Komponente deutlich von dem Hinzufügen weiterer Knoten, die Streaming-Komponente ist jedoch mit einem einzelnen Knoten bereits problemlos zu betreiben und erfährt durch verteilte Ausführung keine weitere Verbesserung der Performance.

Die geringe Last des kostenlosen Twitterdatenstroms verursacht für die weitere Bewertung der Ergebnisse zwei Probleme:

1. Da selbst ein einzelner Knoten für die Analyse des Datenstroms mehr als ausreichend ist, lassen sich aus dem Experiment keine Aussagen über das Skalierungsverhalten der entsprechenden Komponente treffen.

2. Bei der Exploration mehrerer zehntausend Tweets wurde kein einziger mit einem plausiblen Bezug zu den in der Spark-Mailingliste diskutierten Themen gefunden. Eine empirische Bewertung der funktionalen Qualitäten der Implementation ist damit nicht ohne Weiteres möglich.

4.2 Ausblick und offene Punkte

Der Betrieb eines Spark/Hadoop-Clusters ist komplex. In dieser Arbeit wurde die Anwendungsentwicklung mit Spark und die Machbarkeit des Betriebs auf Low-End-Hardware behandelt. Für einen produktiven Betrieb der hier vorgestellten Architektur gibt es eine Reihe von Maßnahmen, die im Rahmen der betrachteten Fragestellungen ausgeblendet wurden:

1. **Message Queues:** Für die robuste Verbindung zur asynchronen Kommunikation zwischen Batch- und Realtime-Komponente käme eine Erweiterung mt Messagequeues oder auch (NoSQL-)Datenbanken in Frage.

2. **Umfangreichere Datenquellen:** Um die Skalierbarkeit der Streaming-Komponente unabhängig vom dem hier behandelten Anwendungsfall zu betrachten, könnte man einen eigenen speziellen Receiver implementieren und diesen mit einer kontrollierbaren Datenquelle verbinden. *Kontrollierbar* heißt hier, dass die erzeugte Last flexibel erhöht werden kann, um die Kapazität des empfangenden Knotens zu übertreffen und eine verteilte Verarbeitung zu erzwingen.

3. **Clustermanagement:** Die Konfiguration der Knoten und der verteilten Komponenten wurde für diesen Versuch überwiegend manuell durchgeführt. Um auch eine höhere Anzahl von Knoten verwalten zu können wären Werkzeuge zur Automatisierung von Konfiguration (z.B. Chef[1], Puppet[2]) oder Provisionierung (z.B. Docker[3], Rocket[4], Kubernetes[5]) notwendig.
   Für ein professionelles Monitoring könnte man den Cluster mit Tools zur dauerhaften Überwachung ausstatten (z.B. Nagios/Icinga[6], Ganglia[7]).

Neben dem Betrieb der Anwendung, gibt es auch funktionale Aspekte die für eine produktive Anwendung erweitert werden könnten:

1. **Textkorpus:** Als Textkorpus für das TF-IDF-Verfahren wurde für diesen *Proof of Concept* der vervielfachte Inhalt von Nachrichten aus der Spark-User-Mailingliste verwendet

[1] https://www.chef.io/chef/
[2] https://puppetlabs.com/
[3] https://www.docker.com
[4] https://github.com/coreos/rkt
[5] http://kubernetes.io/
[6] https://www.icinga.org/
[7] http://ganglia.sourceforge.net/

(etwa 13000 Emails). Eine schärfere Abgrenzung relevanter Begriffe lässt sich wahrscheinlich durch einen unabhängigen Korpus erreichen (beispielsweise Wikipedia[8] oder der klassische Reuters-Nachrichtenkorpus[9]).

2. **Ähnlichkeitsmaß:** Das Scoring könnte darüber hinaus durch leistungsfähigere Verfahren verbessert werden - etwa durch Ähnlichkeitsmaße ([Hua08]), die auch Synonyme erkennen.

3. **Grafische Benutzeroberfläche:** Damit die gefilterten Nachrichten ihren Weg zum Benutzer finden, gibt es eine Vielzahl an Möglichkeiten. Naheliegend wäre ein Webinterface (z.B. über den MEAN-Stack[10] oder das Play-Framework[11] für Scala/Java).

[8] http://www.wikipedia.org/
[9] https://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html, abgerufen am 01.06.2015
[10] http://mean.io/!/
[11] https://www.playframework.com/

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/Skripte (Auszüge)

2.1 Performance-Messungen

```
echo 3 > /proc/sys/vm/drop_caches
dd if=/dev/zero of=test512.out bs=512MB count=1
```

Listing 1: Messung der Festplattenperformance - Beispiel: Schreiben einer 512MB Datei

```
iperf -c <IP-Adresse des Peers> -r -P 4
```

Listing 2: Messung der Netzwerkperformance

2.2 Monitoring

```
#/bin/bash

### ModelBuilder Wrapper Skript

TIME=‘date +"%H-%M"‘
WORKERS="pi00 pi01 pi02 pi03"

# Start Monitoring

for host in $WORKERS; do
  ssh $host "screen -d -m dstat -cdlsmn --output $1.log"
done
screen -d -m dstat -cdlsm --output $1.log

# Run App
/home/daniel/start_modelbuilder.sh $1-$TIME 2>&1 > \
  ~/metrics/$1_$TIME.log

# Stop Monitoring
for host in $WORKERS; do
  ssh $host ’screen -X quit’
done
screen -X quit

# Collect results
```

```
for host in $WORKERS; do
  scp $host:/home/daniel/$1.log ~/metrics/dstat/$1_$host_$TIME.csv
done
cp ~/$1.log ~/metrics/dstat/$1_dell01_$TIME.csv

# Clean up
for host in $WORKERS; do
  ssh $host "rm /home/daniel/$1.log"
done
```

Listing 3: Monitoring des Clusters (Betriebssystem), Beispiel ModelBuilder

2.3 Realisierung einer einfachen Continuous Deployment Pipeline

post-receive-Hook von dem Git-Repository[12] 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\
```

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

```
<emailaccount> <passwort>
```

Listing 4: Einfache Continuous Deployment Pipeline. Beispiel: ModelBuilder

3 Konfigurationen

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
  <property>
    <name>dfs.block.size</name>
    <value>33554432</value>
  </property>
  <property>
    <name>dfs.permissions</name>
    <value>false</value>
  </property>
  <property>
    <name>dfs.datanode.drop.cache.behind.writes</name>
    <value>true</value>
  </property>
  <property>
    <name>dfs.namenode.datanode.registration.ip-hostname-check</name>
    <value>false</value>
  </property>
</configuration>
```

Listing 5: hdfs-site.xml (Auszug): Beispiel mit Replikationsfaktor 1 und Blockgröße 32MB

Eine vollständige Liste der HDFS-Optionen und -Standardeinstellungen kann unter [13] eingesehen werden.

```
spark.master                     spark://dell01:7077
spark.eventLog.enabled           true
spark.eventLog.dir               file:///home/spark/metrics/secondary
spark.shuffle.memoryFraction     0.3
```

Listing 6: spark-defaults.conf (Auszug)

[13] https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml, abgerufen am 03.06.2015

4 Sonstiges

4.1 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*. 2014. 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.

[Hua08] Anna Huang. "Similarity Measures for Text Document Clustering". In: NZCSRSC '08. 2008. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.332.4480&rep=rep1&type=pdf.

[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.

[SJ88] Karen Sparck Jones. "Document Retrieval Systems". In: Hrsg. von Peter Willett. London, UK, UK: Taylor Graham Publishing, 1988. Kap. A Statistical Interpretation of Term Specificity and Its Application in Retrieval, S. 132–142. ISBN: 0-947568-21-2. URL: http://dl.acm.org/citation.cfm?id=106765.106782.

[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