Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Daniel Kirchner

Skalierbare Datenanalyse mit Apache Spark

Implementation einer Text-Mining-Anwendung und Testbetrieb 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 Testbetrieb 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: 17. Juni 2015

Daniel Kirchner

Thema der Arbeit

Skalierbare Datenanalyse mit Apache Spark: Implementation einer Text-Mining-Anwendung und Testbetrieb auf einem Low-End-Cluster

Stichworte

Apache Spark, Big Data, Architekturanalyse, Text Mining, Echtzeit-Datenanalyse

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: Implementation of a Text Mining Application and Test Operation on a Low-End Cluster

Keywords

Apache Spark, Big Data, Architecture Analysis, Text Mining, Real-Time Analysis

Abstract

Apache Spark is quickly becoming a central component of Big Data analysis systems for a variety of applications. This work provides an overview of key concepts and components of Apache Spark and examines the behavior of Spark on a cluster with a minimal performance profile. This study is based on an application that is inspired by a real-world usecase. The application combines the Spark modules for batch processing and streaming.

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 und 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 die Verwaltung der Jobs und gegebenenfalls wiederholtes Einlesen von Daten.

- Algorithmen die während ihrer Ausführung iterativ Zwischenergebnisse erzeugen und auf vorherige Zwischenergebnisse 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 nicht so intuitiv und leicht erlernbar wie deklarative Abfragesprachen klassischer Datenbanken (z.B. SQL).

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

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

Apache Spark hat innerhalb relativ kurzer Zeit stark an Aufmerksamkeit gewonnen. Ein Indiz für das steigende Interesse an diesem Produkt liefert unter anderem ein Vergleich des Interesses an Hadoop und Spark auf Google:

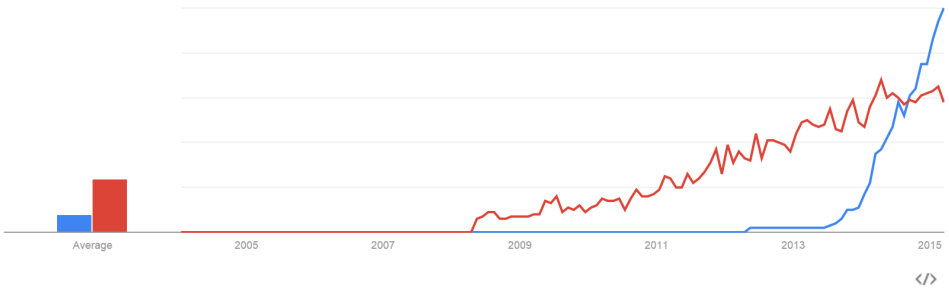

Abbildung 1.1: Suchanfragen zu „Apache Spark“ (*blau*) und „Apache Hadoop“ (*rot*), Stand 24.03.2015 [Goo]

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 demonstrieren und 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
- einen Überblick der High-Level-Application Programming Interfaces (APIs) in den Standardbibliotheken von Spark

Der praktische Teil dieser Arbeit umfasst

- den Entwurf und die Implementation einer hybriden Anwendung mit einer Echtzeit-Komponente (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 und Architekturübersicht von 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 in der konkreten Umsetzung beudeutet und welche Mechanismen und Konzepte dahinterstehen, 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 (Treiberprogramm) auf die API eines lokalen Spark-Kontextes zu, der die Kontaktdaten eines Clustermanagers sowie verschiedene Konfigurationseinstellungen enthält.

2. **Masterknoten**
   Der Masterknoten betreibt den *Clustermanager* und ist der Einstiegspunkt in den Rechnercluster. Hier werden Ressourcenanforderungen des Treiberprogramms an die Arbeitsknoten verteilt und der Betrieb der *Worker* überwacht.

3. **Workerknoten**
   Die Workerknoten beherbergen die Spark Workerprozesse. Diese melden verfügbare Ressourcen und erzeugen bei Bedarf *Executors*. *Executors* sind die ausführenden Elemente der Aktionen und Transformationen im Rahmen einer Sparkanwendung. Die *Executors* können untereinander Zwischenergebnisse austauschen und Nachrichten und Ergebnisse an das Treiberprogramm senden.

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

Spark ist explizit für den Betrieb innerhalb eines Hadoop/YARN Rechnerclusters geeignet. Hadoop/YARN wiederum ist für den Betrieb auf einem Rechnercluster auf Mittelklasse-Mehrzweckmaschinen (Commodity Hardware) optimiert ([AM14]). Für Spark kann daher von einer vergleichbaren Hardwarekonfiguration ausgegangen werden.

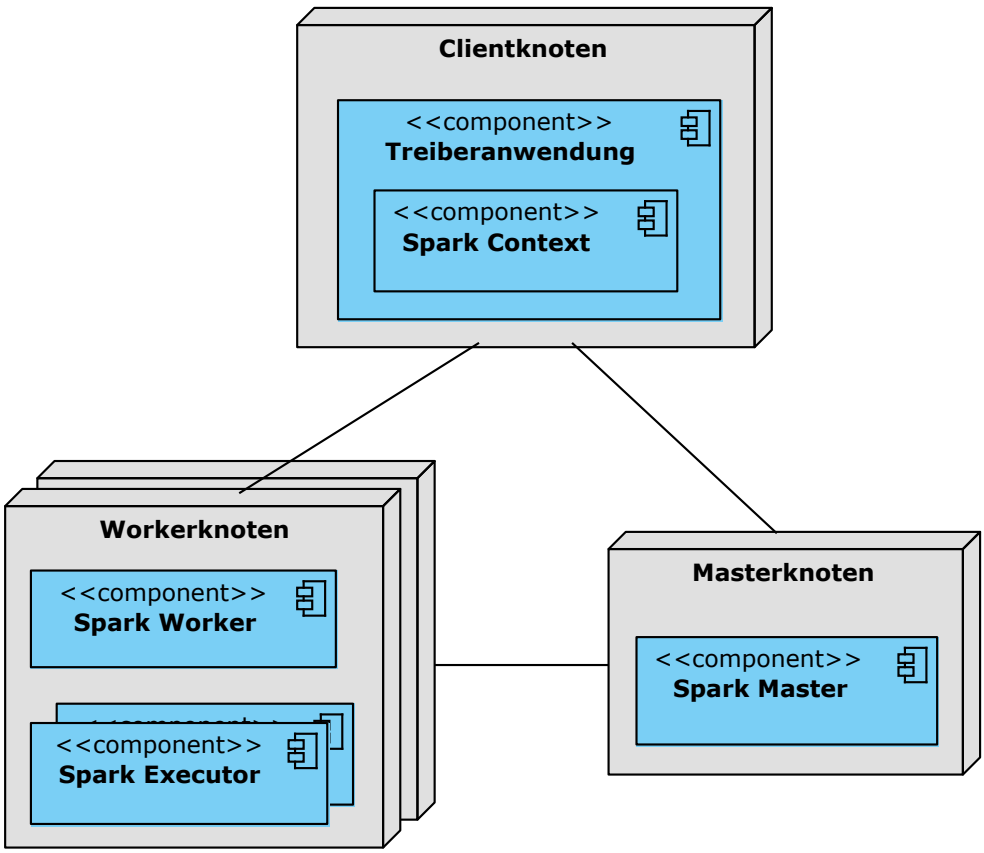

Abbildung 2.1: Verteilungsdiagramm einer typischen Sparkinstallation

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).

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

Tabelle 2.1: Theoretische Spitzenleistungen bei Mittelklasse-Servern

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.

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 und Komponenten

2.2.1 Resilient Distributed Datasets

Die universelle Einheit mit der Daten auf Spark repräsentiert werden ist eine verteilte Datenstruktur, 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 entstehen nicht nur durch Einlesen externer Datenquellen, sondern können auch explizit von einem Treiberprogramm erzeugt werden:

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

Die gesamte operative Kern-API dreht sich um die Steuerung dieser Datenstruktur. 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. Sie 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).

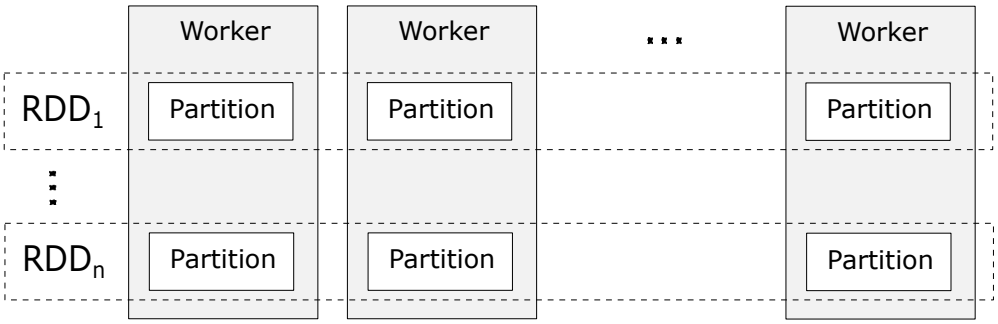

Abbildung 2.2: Resilient Distributed Datasets aus Verteilungssicht

Laufzeitsicht RDDs sind nicht veränderbar. Es ist nicht möglich, Operationen gezielt auf einzelnen Elementen eines RDD anzuwenden. Stattdessen ist es nur möglich, ein einmal definiertes RDD durch globale Anwendung von Operationen in ein anderes zu überführen. Solche globalen - auf sämtlichen Partitionen des RDD durchgeführten - Operationen können zwar ihren Effekt auf einzelne Elemente eines RDD 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 RDD auf ein anderes abbilden:

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

oder

$$Transformation : RDD \longrightarrow RDD$$

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

- *filter*
  Erstellen eines neuen RDD, das nur die Elemente enthält die einen übergebenen boolschen Ausdruck erfüllen

- *join*
  Zusammenfassen von zwei RDDs zu einem neuen. Die Partitionen bleiben dabei erhalten

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*
  Paarweises Zusammenführen der Elemente eines RDD und Rückgabe eines Elementes mit dem Typ der verknüpften Elemente (dabei muss ein zweiwertiger, assoziativer und kommutativer Operator übergeben werden, der je zwei Elemente verknüpft und auf ein drittes abbildet)

- *count*
  Zählen der Elemente eines RDD

- *collect*
  Überführen des RDD in ein lokales Array mit dessen Elementen

- *foreach*
  Ausführung einer beliebigen Prozedur pro Element eines RDD (der Typ des Rückgabewerts ist dabei beliebig)

Den meisten dieser Operationen wird im Sinne des Command-Patterns ([BL13]) eine Funktion bzw. ein Funktionsobjekt übergeben, das 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 schließlich die geforderte *Aktion*. Die Initialisierung des RDD

erfolgt also „lazy“:

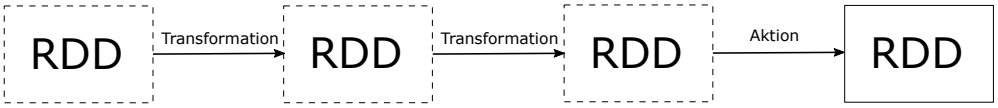

Abbildung 2.4: RDD Lineage nach Aktion

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()` oder `cache()` anweisen:

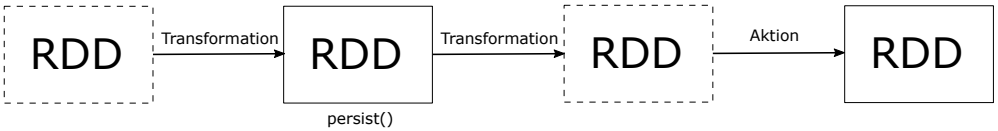

Abbildung 2.5: RDD Lineage nach Aktion und mit Persist()

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 RDDs:
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 ein neues RDD-Referenzobjekt. 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 zusammen mit der Funktion `f` (bzw. `cleanF`) übergeben wird. Diese Funktion beschreibt die Erzeugung der Elemente des neuen RDDs aus den Elementen des aktuellen.

[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)

```
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

Die tatsächliche Berechnung eines RDDs wird dann bei dem 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, die die Aufgabe wiederum an einen 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 (Abb. 2.6).

[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)

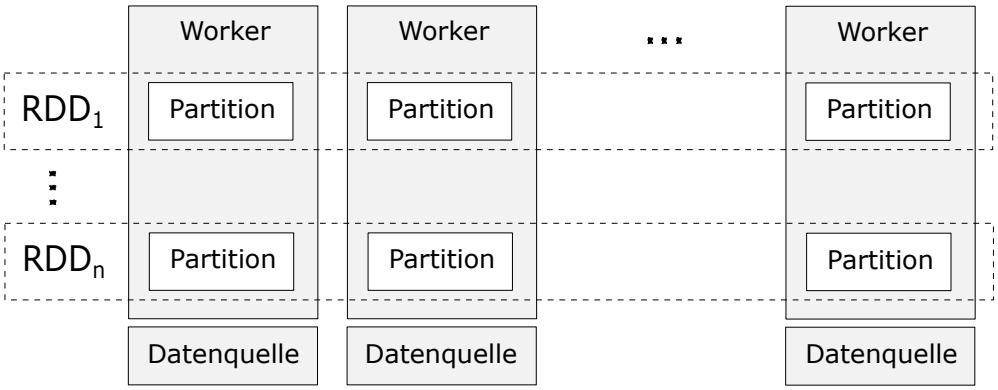

Abbildung 2.6: Resilient Distributed Datasets mit Datenquelle aus Verteilungssicht

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 sind hierbei Teilmengen des RDD, die mit besonders hoher Wahrscheinlichkeit oder besonders 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 2.2.3) 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.

```
1 /*
2  * Beispiel fuer einen minimalen Partitionierer.
3  * Ueber selbstdefinierte Hash Codes kann hier eine
4  * massgeschneiderte Verteilung ueber die
5  * 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]
}
```

Listing 2.4: Beispiel: Minimaler Partitionierer

2.2.2 Anwendungsdeployment

Eine Anwendung, die über ein SparkContext-Objekt von der Spark API Gebrauch macht, wird in der Spark-Terminologie als *Treiberprogramm* bezeichnet.

Es gibt grundsätzlich zwei verschiedene Arten solch ein Treiberprogramm zu deployen:

1. Übermittlung des Treibers an einen Wrapper für den Spark-Class-Loader als kompiliertes Package (z.B. als *jarfile*) mit statischer Verlinkung aller erforderlichen Bibliotheken (Ausnahmen sind Bibliotheken die auf allen Knoten bereits verfügbar sind, z.B. Spark, Hadoop, etc.). Klassenpfade zu Standardbibliotheken von Spark und weitere Konfigurationseinstellungen wie z.B. die Angabe des *Clustermanagers* können zur Startzeit des Treibers durch das Submission-Skript festgelegt werden.
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 aus Wartungsgründen nicht wünschenswert ist.

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

1. **Client-Modus** Der Treiber wird direkt auf dem Host (Gateway-Rechner) ausgeführt, auf dem der Treiber übermittelt wurde (Abb. 2.7). Tatsächlich wird er sogar innerhalb des Submission-Skript-Prozesses gestartet.

2. **Cluster-Modus** Der Treiber wird von dem Gateway-Rechner an einen Worker des Clusters übertragen und dort ausgeführt (Abb. 2.8).

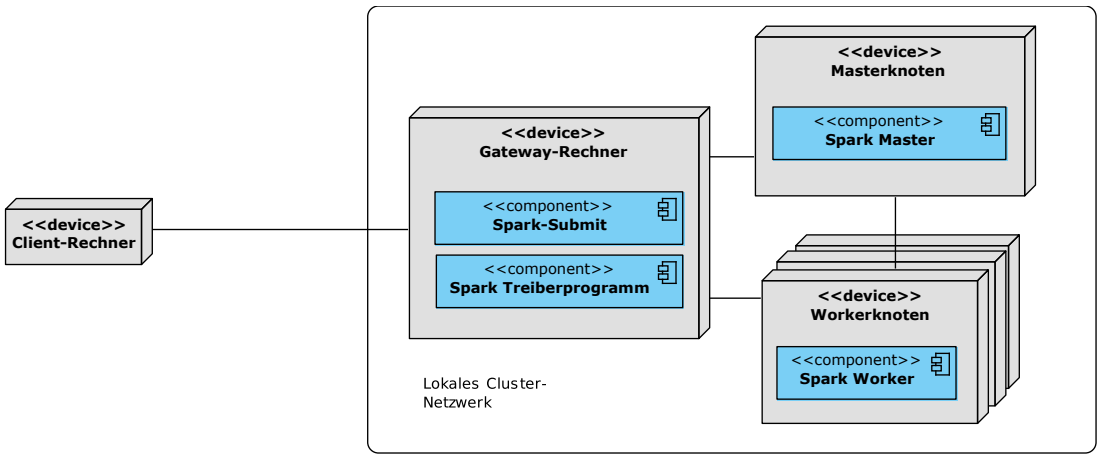

Abbildung 2.7: Application Deployment im Client Modus

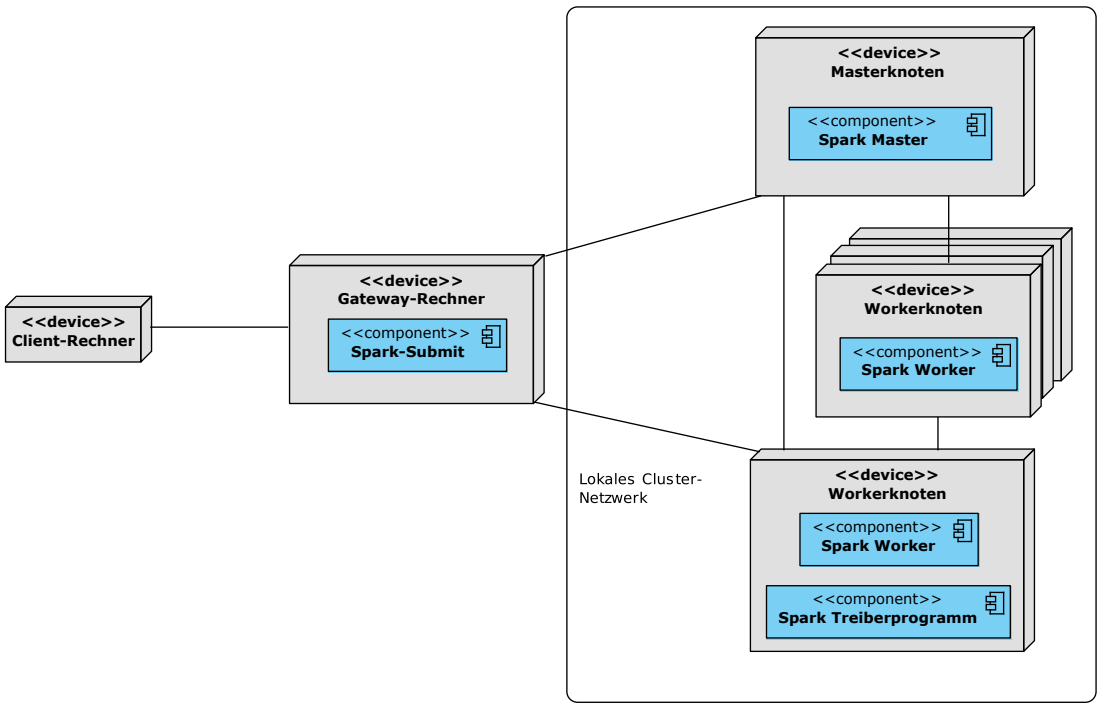

Abbildung 2.8: Application Deployment im Cluster Modus

Die Lokalität eines Treibers (der mit den Executors auf den Workern kommunizieren muss) kann 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. 2.9 zeigt das Sequenzdiagramm eines typischen Deploymentprozesses, wie er auch im praktischen Teil dieser Arbeit zum Einsatz kommt.

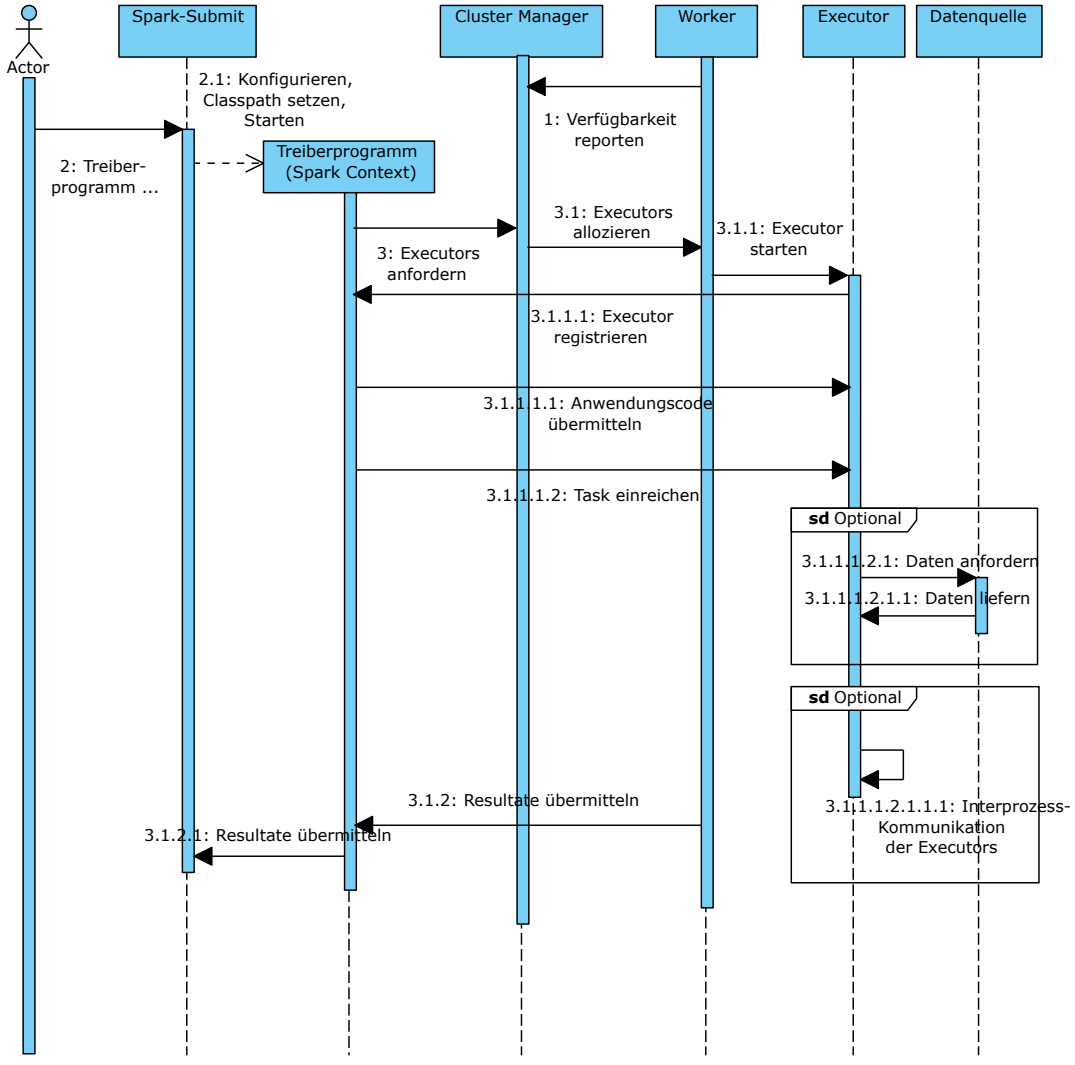

Abbildung 2.9: Laufzeitdiagramm einer Spark-Anwendung im Client Modus (vereinfacht)

2.2.3 Scheduling und Shuffling

In Spark gibt es drei Scheduler und drei wesentliche Laufzeit-Elemente einer Sparkanwendung.

Die Laufzeitelemente sind

1. **Jobs**: Ein *Job* ist die Ausführung einer Aktion auf einem RDD.
2. **Tasks**: Ein Task ist die kleinste Einheit eines *Jobs*, die noch zentralem Scheduling unterliegt. Ein Task bezieht sich immer auf eine Partition eines RDD.
3. **Stages**: *Stages* sind eine Menge von Tasks, die zu demselben Job gehören und unabhängig voneinander ausgeführt werden können.

Die Scheduler sind

1. **DAGScheduler**: Der DAGScheduler analysiert einen Job vor der Ausführung auf dem Cluster und teilt dessen Tasks in Stages auf.
2. **RessourceScheduler**: Der Ressourcescheduler weist einem Treiberprogramm Ressourcen des Clusters in Form von Executorn zu.
3. **TaskScheduler**: Der TaskScheduler nimmt Mengen von Tasks entgegen und verteilt diese zur Ausführung an verfügbare Executor.

Abbildung 2.10 stellt die Beziehung dieser sechs Elemente in einem Aktivitätsdiagramm dar.
Von den beschriebenen Schedulern kann ein Spark-Treiberprogramm vor allem Einfluss auf die *Stages* des *DAGSchedulers* nehmen. Um zu erläutern, inwiefern durch diese Aufteilung von Tasks Geschwindigkeit und Fehlertoleranz erreicht werden kann, wird im Folgenden das Erzeugen von *Stages* aus Transformationsfolgen (d.h. Jobs) auf RDDs vertieft.

Bei den RDDs wurden bisher insbesondere drei Aspekte behandelt:

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

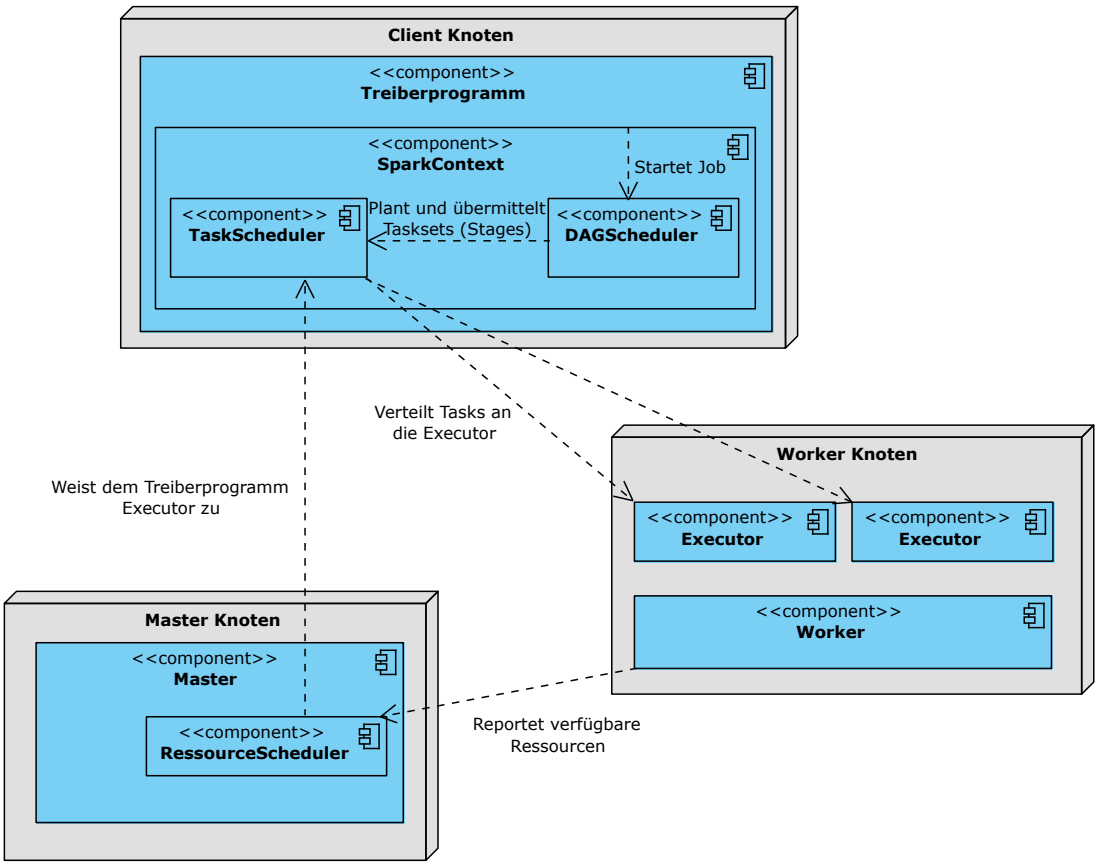

Abbildung 2.10: Aktivitätsdiagramm der Spark Scheduler

Betrachtet man nur die Vorgänger-Beziehung, dann erhält man zunächst eine einfache Baumstruktur für die Berechnung eines RDD (Abb. 2.11).

Betrachtet man zusätzlich die Partitionierung und die Funktion mit der RDDs transformiert werden, sieht man einen wichtigen Unterschied zwischen verschiedenen Vorgängerbeziehungen (siehe Abb. 2.12):

- 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*

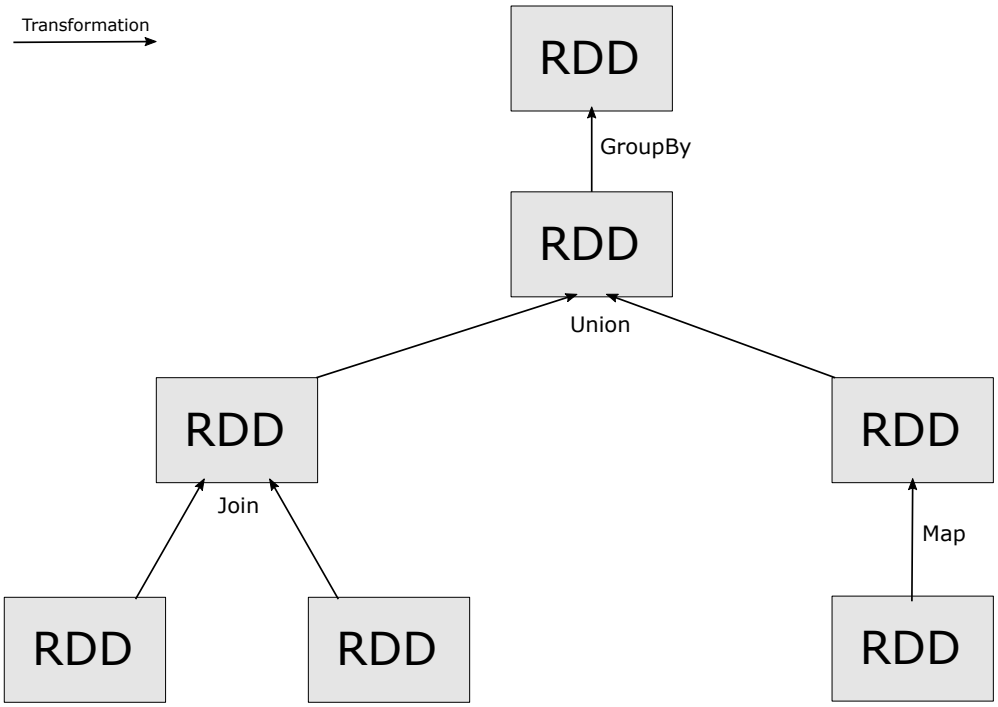

Abbildung 2.11: Beispiel eines einfachen Abhängigkeitsbaums eines RDD

(zweiter Fall) ([ZC+12]).

Der Abhängigkeitsgraph eines RDD wird nun in sogenannte *Stages* zerlegt (Abb. 2.13). Eine *Stage* ist dabei ein Untergraph, dessen Elemente (von den Blättern in Richtung Wurzel betrachtet) auf eine gemeinsame *wide dependency* stoßen.

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 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

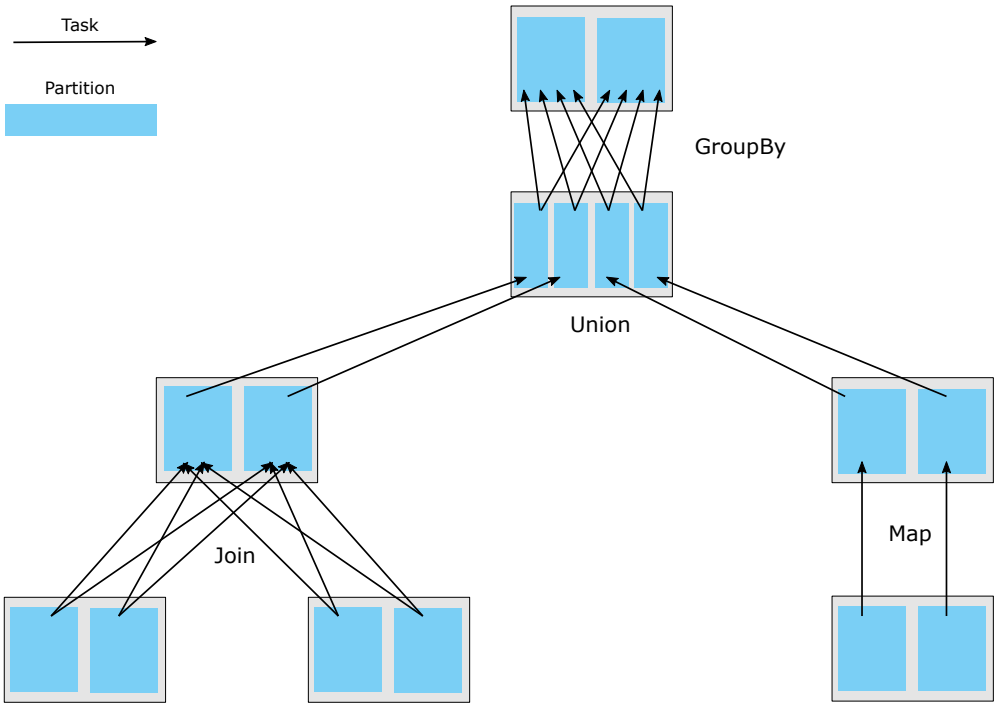

Abbildung 2.12: Gerichteter azyklischer Graph der Abhängigkeiten auf Partitionen

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 alle 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*[4] von Elementen verarbeitet. Diese Elemente stammen dann in der Regel aus verschiedenen Vorgängerpartitionen.
Das so neupartitionierte und -verarbeitete RDD wird so zum Ausgangspunkt des folgenden

[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] Ein Bucket ist eine Sammelstelle für eingehende Elemente verschiedener Partitionen

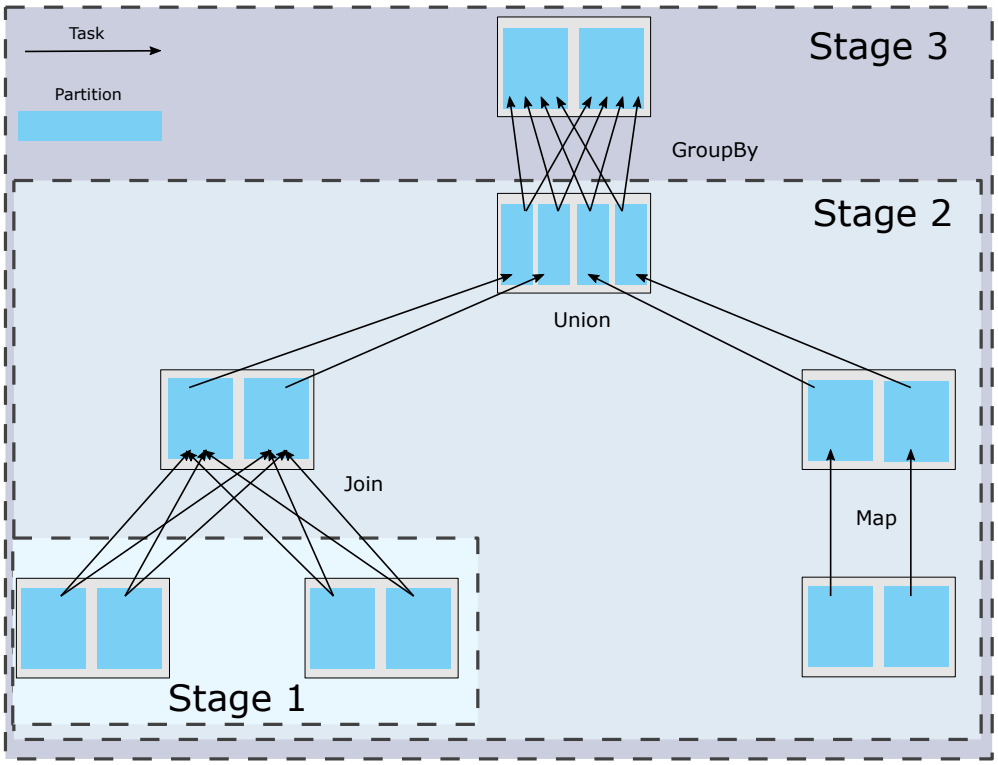

Abbildung 2.13: Stages als Untermenge des Ahängigkeitsgraphen von Partitionen

RDDs.
Eine Schlüsselrolle kommt bei diesem Prozess der Methode `runTask` in der Klasse `ShuffleMapTasks`[5] aus *org.apache.spark.scheduler.ShuffleMapTask* zu.

2.3 Standardbibliotheken

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

Dieser Abschnitt gibt einen Überblick der Bibliotheken und deren zentralen Konzepte zur Abstraktion von RDDs für die spezifischen Zwecke.

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

2.3.1 SparkSQL

Die Spark SQL Bibliothek bietet Schnittstellen für den Zugriff auf strukturierte Daten. Dazu wird eine weitere zentrale abstrakte Datenstruktur neben den RDDs eingeführt. Diese Datenstruktur sind DataFrames, wie sie in ähnlicher Art auch aus der Programmiersprache R[6] bekannt sind.

DataFrames und die SparkSQL Bibliothek bilden einen Ansatz relationale Abfragen mit komplexen prozeduralen Algorithmen zu verknüpfen, um die Performance bei Abfragen auf großen Datenmengen zu optimieren (Vgl. [Arm+15]).

Die Abfrage und Verarbeitung von Daten kann entweder über die Schnittstellen dieses Datentyps erfolgen (ähnlich wie bei RDDs) oder in Form von SQL-Syntax.

Diese Form des Zugriffs ist beispielsweise über einen JDBC-Adapter möglich und bietet so anderen Anwendungen und Frameworks die Möglichkeit bestehende SQL-Schnittstellen mit Apache Spark zu verbinden.

Ähnlich wie bei RDDs werden komplexe Folgen von Abfragen zunächst in dem DataFrame-Referenzobjekt gespeichert, und erst bei Initialisierung ausgewertet. Dabei wird eine Optimierungspipeline genutzt, die zunächst die relationalen Ausdrücke optimiert und ihn dann in Spark-Operationen auf RDDs übersetzt.

Die SparkSQL-Bibliothek zerfällt dabei in vier Unterprojekte[7]. Die ersten beiden sind essentiell für den Mechnismus zum Prozessieren von Dataframes als ein Prozess auf RDDs:

1. **Catalyst**: Eine Optimierungsengine für relationale Ausdrücke
2. **Core**: Die Benutzer-API und Ausführungsumgebung für optimierte Catalyst-Ausdrücke auf dem Spark-Applikationskern
3. **Hive**: Ein Adapter für die Datawarehouse-Software Apache Hive[8]
4. **Thriftserver**: Eine Portierung des Hive-Servers auf Apache Spark

[6] http://www.r-project.org/, abgerufen am 20.05.2015
[7] [**https://github.com/apache/spark/tree/branch-1.3/sql**; **abgerufen am 16.06.2015**].
[8] https://hive.apache.org/, abgerufen am 16.06.2015

2.3.2 MLlib

Die Spark *Machine Learning Library* bietet Implementationen bewährter Maschinenlernverfahren und Abstraktionen deren Einsatz.
Dabei werden die Komponenten eines Maschinenlernprogramms in *Transformer*, *Estimator* und *Pipelines* aufgeteilt[9]. Als Abstraktion der verarbeiteten Datensätze werden DataFrames aus der SparkSQL-Bibliothek verwendet.

Die Komponenten bezeichnen Folgendes:

- **Transformer** sind zustandslose Objekte, die mit einer `transform`-Methode DataFrames verändern.
- **Estimator** erzeugen Transformer aus einem Eingangsdatensatz (DataFrame). Sie sind ebenfalls zustandslos und können als Abstraktion eines Lernalgorithmus verstanden werden.
- **Pipelines** sind in der MLlib eine Abstraktion der Verknüpfung von Transformern und Estimatorn. Ähnlich wie bei den Transformationen eines RDD werden auch dies „lazy“ initialisiert und in Stages aufgeteilt.

Teile der Spark MLlib werden im praktischen Teil dieser Arbeit noch näher erläutert.

2.3.3 Streaming

Die Spark Streaming Bibliothek ermöglicht die Verarbeitung von Datenströmen mit dem Konzept der RDDs.

Dabei werden werden RDDs zu diskretisierten Datenströmen (sogenannte *DStreams*) erweitert.

Ein oder mehrere Empfänger (*Receiver*) werden auf Workerknoten eines Sparkclusters gestartet, die anschließend eingehende Daten über ein vorgegebenes Zeitfenster puffern. Die Puffer können dabei verteilt auf mehreren verfügbaren Workerknoten angelegt werden.

Nach Ablauf des Zeitfensters bilden diese Puffer die Partitionen eines RDD mit den empfangenen Datensätzen. Dieses RDD wiederum ist Teil einer Sequenz von RDDs die im Verlauf

[9] https://spark.apache.org/docs/1.3.1/ml-guide.html, abgerufen am 16.06.2015

dieses Prozesses erzeugt werden.

Mit diesem Verfahren wird ein Großteil der Operationen klassischer RDDs auch für Datenströme verfügbar.

Eine Reihe von Empfängern für solche Datenströme sind bereits in der Spark Streaming Bibliothek enthalten (z.B. für Kafka[10], Twitter, ZeroMQ[11], ...)[12]. Weitere Empfänger können selbst implementiert und übergeben werden.

Eine Beispielimplementation dieses Verfahrens wird im praktischen Teil durchgeführt.

2.3.4 GraphX

Die GraphX Bibliothek implementiert Graphstrukturen auf Grundlage von RDDs und bietet Methoden zu deren Verarbeitung.

Die Graphstrukturen sind dabei sogenannte Property-Graphen. Property-Graphen (wie sie beispielsweise auch aus der Datenbank Neo4j[13] bekannt sind) sind gerichtete Multigraphen, deren Kanten-Objekte mit Eigenschaften versehen werden können.

Die Graphen der Spark GraphX-Bibliothek bilden dabei Tupel aus RDDs wobei das erste die Ecken und das zweite die Kanten des Graphen enthält.

Die Zerteilung eines Graphen über mehrere Maschinen erfolgt nach dem sogenannten *Edge Cut* Verfahren ([Gon+14]). Dabei werden die Graphen - logisch gesehen - entlang der Kanten geteilt.

Eine einzelne Ecke kann daraufhin auf mehreren Maschinen gleichzeitig liegen. Aus der ID jeder dieser Kopien, lässt sich jedoch die Partition (im Sinne des darunterliegenden RDD) einer Kante stets eindeutig berechnen.

Die Attribute der Ecken werden ebenfalls in den Kanten gespeichert. Um Änderungen an alle Kopien einer Ecke zu propagieren wird zusätzlich zu den RDDs der Ecken und Kanten

[10] http://kafka.apache.org/, abgerufen am 16.06.2015
[11] http://zeromq.org/, abgerufen am 16.06.2015
[12] https://spark.apache.org/docs/1.3.1/streaming-programming-guide.html, abgerufen am 16.06.2015
[13] [**http://neo4j.com/**; **abgerufen am 16.06.2015**].

noch eine Routingtabelle geplegt. Über diese Routing Tabelle werden die Replikate von Ecken verfolgt und bei Änderungen einer Ecke alle inzidenten Kanten entspechend angepasst.

2.4 Entwicklergemeinschaft

Apache Spark begann als Entwicklung einer Gruppe von Forschern der University of California, Berkely. Spark basiert auf einer 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 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. 2.14) und dem Issue-Tracker ([Iss]) der Apache Software Foundation statt.

2.5 Übersicht verwandter Produkte

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

2.5.1 Hadoop/YARN

Hadoop[14] 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 fehlertole-

[14] https://hadoop.apache.org/, abgerufen am 16.06.2015

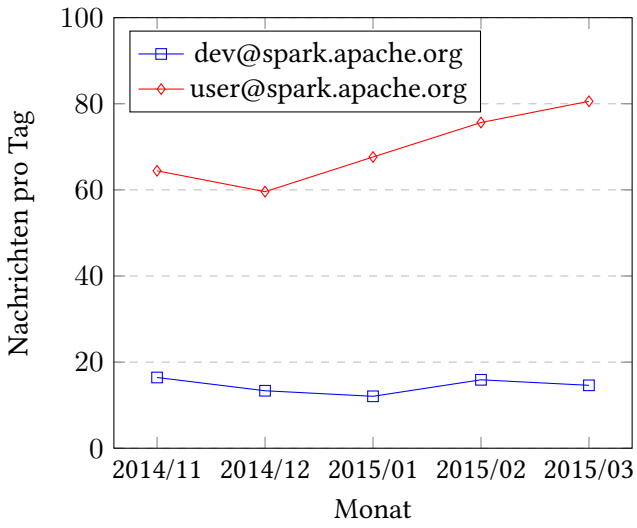

Abbildung 2.14: Aktivität auf den offiziellen Spark Mailinglisten

ranten und skalierbaren Betrieb auf verteilten Hardware-Komponenten vorgesehen.

Zwar wird mit MapReduce auch eine Komponente zur Verarbeitung 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.

2.5.2 Mesos

Apache Mesos[15] ist seit Beginn der Entwicklung von Spark ein optionaler Clustermanager für Spark-Applikationen [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.

[15] http://mesos.apache.org/, abgerufen am 16.06.2015

2.5.3 Flink

Apache Flink[16] ist mit seiner Funktionalität ein direkter Konkurrent zu Apache Spark.

Flink ist ebenfalls in Java/Scala geschrieben und bietet mit drei Standardbibliotheken zur Graphprozessierung, zum Maschinenlernverfahren und zu SQL-artigen Abfragen auf strukturierten Datensätzen einen sehr ähnlichen Funktionsumfang.

Der wesentliche Unterschied in der Architektur von Flink, verglichen mit Spark, liegt im Applikationskern. Flink bietet ein Streaming-Backend auf dem wiederum auch Batchprozesse implementiert werden können. Spark wiederum bietet ein Batch-Processing-Backend (Operationen auf RDDs) auf dem auch Streaming-Prozesse implementiert werden können.

Auffällig ist weiterhin, dass im Flink-Applikationskern häufig auf Low-Level-Methoden zurückgegriffen wird und beispielsweise mit der Endianness[17] auf verschiedenen Speicherarchitekturen umgegangen wird. Solche feingranularen Optimierungsbemühungen sind dem Autoren bei Spark nicht bekannt.

2.5.4 Open MPI

Open MPI[18] ist die Implementation des *Message Passing Interface*[19], eines Standards zum Nachrichtenaustausch zwischen Prozessen.

Obwohl dieses Framework ebenfalls der Koordination unterschiedlicher Prozesse und einem gemeinsamen verteilten Speicherbereich dient, ist es völlig agnostisch bezüglich der ausgeführten Anwendung.
Sparks Domäne dagegen ist „Large Scale Data Processing“, sämtliche Komponenten von Spark sind für diese Aufgaben optimiert. Überschneidungen der Anwendungsbereiche von Open MPI und Spark sind daher sehr unwahrscheinlich.

[16] https://flink.apache.org/, abgerufen am 16.06.2015
[17] Art der Bytereihenfolge bei der Ablage von Daten im Arbeitsspeicher
[18] http://www.open-mpi.de/, abgerufen am 16.06.2015
[19] http://www.mcs.anl.gov/research/projects/mpi/, abgerufen am 16.06.2015

3 Entwicklung und Betrieb einer Beispielanwendung

In diesem Kapitel wird die Entwicklung einer Anwendung für Spark und der Betrieb auf einem Cluster mit leistungsschwacher Hardware demonstriert und untersucht.

3.1 Vorstellung der Beispielanwendung

Als Demonstration der Anwendungsentwicklung und des Anwendungsbetriebes mit Spark soll ein Backend zur Nachrichtenanalyse implementiert werden.

Nachrichten sind in diesem Fall Tweets, die über die kostenlose Streaming-API[1] von Twitter[2] übertragen und anschließend bewertet werden sollen.

Die Bewertung soll anhand inhaltlicher Ähnlichkeit mit aktuellen Diskussionen innerhalb der Mailingliste[3] von Sparkbenutzern erfolgen.

In Abbildung 3.1 wird der zugehörige Anwendungsfall dargestellt. Für die Beispielanwendung wird nur das Backend implementiert. Die grafische Benutzeroberfläche wird jedoch stellenweise erwähnt, um einen klaren Kontext zu setzen.

Die Analyse der Tweets soll in Quasi-Echtzeit erfolgen und direkt anschließend an eine Datensenke weitergereicht werden (dort wäre beispielsweise eine Messagequeue oder Datenbank zur Weiterleitung an eine grafische Benutzeroberfläche).

[1] https://dev.twitter.com/streaming/sitestreams, abgerufen am 06.06.2015
[2] https://twitter.com, abgerufen am 06.06.2015
[3] user@spark.apache.org, abgerufen am 15.06.2015

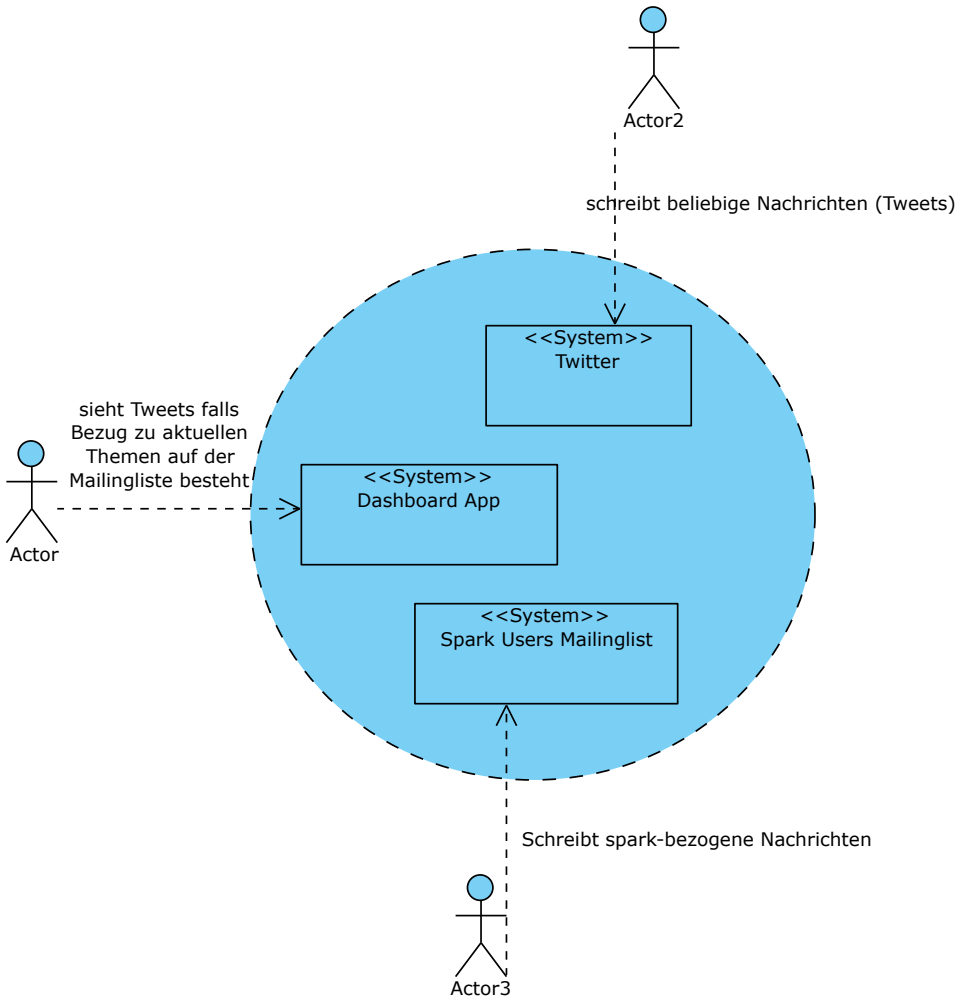

Abbildung 3.1: Anwendungsfalldiagramm der Demo-Applikation

Bis zu zehn Sekunden Latenz vom Eingang der Nachricht bis zur Weiterleitung an die Senke seien hier als *Quasi-Echtzeit* akzeptabel.

Als *aktuelle Diskussionen* in der Mailingliste der Sparkbenutzer gelten solche, die innerhalb der letzten etwa 500 Nachrichten geführt wurden. Bei der Bewertung neuer Nachrichten gibt es hier keine Echtzeit-Anforderung. Die Neubewertung bei Eintreffen neuer Nachrichten - höchstens etwa alle 30 Minuten - soll genügen.

3.2 Hardwareumgebung

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

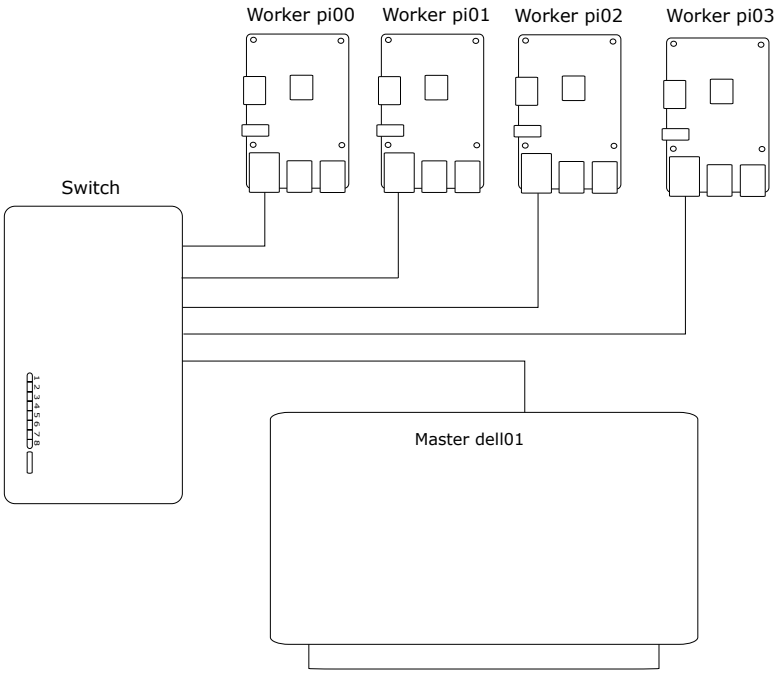

Abbildung 3.2: Hardwareumgebung des Programms zur Tweetanalyse

Dieser Cluster besteht aus ausschließlich aus leistungsschwacher Hardware. Mit *leistungsschwach* sind Geräte gemeint, deren Leistungsfähigkeit noch deutlich unter aktueller sogenannter *Commodity Hardware* liegt (Vgl. Anhang 3.1).

Konkret kommen die folgenden Geräte zum Einsatz:

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 wird das Debian-Derivat Raspbian[Ras] 32-Bit genutzt.

Master Dell d420

- CPU: 1,2 GHz Core2 Duo U2500
- RAM: 2GB DDR2 SDRAM
- Ethernet: 100MBit/s
- Festspeicher: 60GB 4200RPM Hard Drive

Als Betriebssystem wird Ubuntu ([Ubu]) 14.04 32-Bit genutzt.

Die Rechner sind mit RJ45 über einen TP-Link TL-SF1008D Switch mit theoretischem maximalem Durchsatz von 100MBit/s vernetzt.

Als Basisdaten für die Leistungsfähigkeit der eingesetzten Hardware werden zusätzliche Tests durchgeführt, deren Ergebnisse in Tabelle 3.1, Tabelle 3.2 und Tabelle 3.2 aufgeführt sind.

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

Tabelle 3.1: Maximaler Netzwerkdurchsatz[4]

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)[5]

[4] Gemessen mit `iperf`. Siehe Anhang Listing 2
[5] Gemessen mit `dd`. Siehe Anhang Listing 1

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)

3.3 Lösungsskizze

3.3.1 Wahl des Dateisystems

Für diesen Versuch wird das Hadoop Dateisystem *HDFS* in der Version 2.7.0 gewählt.

Dieses Dateisystem gewährleistet die verteilte Speicherung von Blöcken auf dem Cluster und bietet Konfigurationsmöglichkeiten, die direkten Einfluss auf die Performance von Spark haben.

Zu diesen Konfigurationsmöglichkeiten gehören insbesondere die Größe der Blöcke und die Anzahl der Replikate eines Blockes.

3.3.2 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.

3.3.3 Architekturübersicht

Die Architektur der Beispielanwendung teilt die Anwendungslogik in drei Schichten auf.

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*). Diese Schicht gehört nicht zum Spark-Backend und wird für diese Demonstation nicht implementiert.

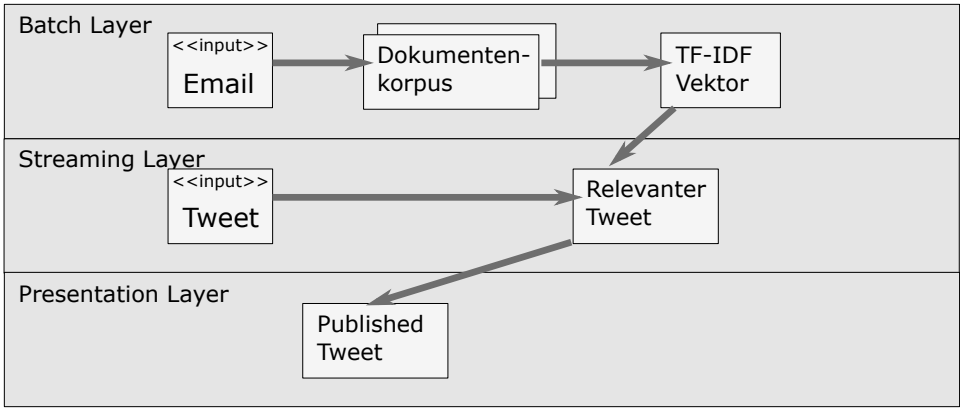

Abbildung 3.3: Datenzentrierte Sicht auf die Komponenten

3.3.4 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 Korpus 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*).

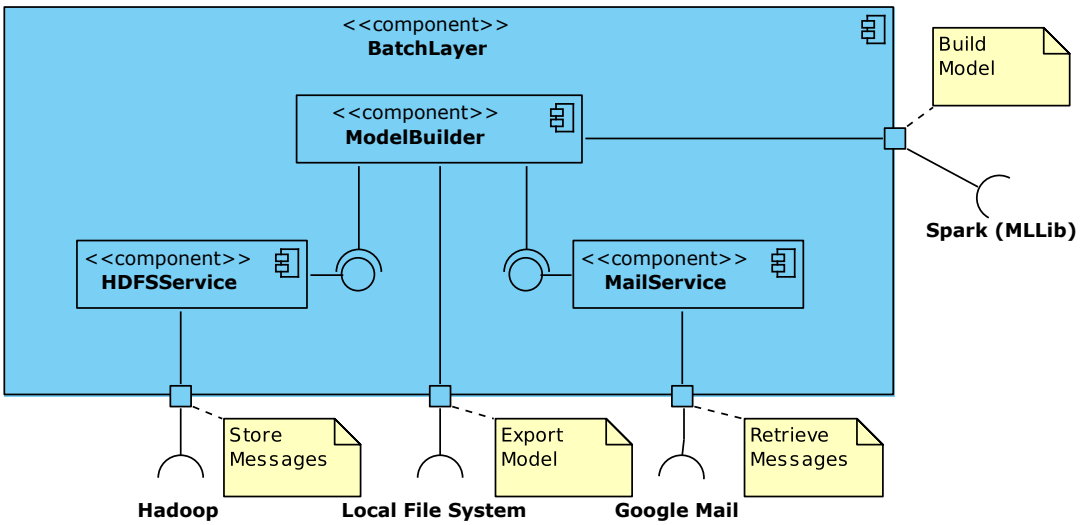

Abbildung 3.4: Innenansicht der Batch-Layer Komponente

Eine Implementation dieses Verfahrens ist in der Spark-Standardbibliothek *MLLib* in dem Bereich *Feature Extraction* verfügbar[6].

In diesem Anwendungsfall gilt:

- Email-Bodies[7] werden jeweils als Dokumente verarbeitet
- Das Archiv aller Email-Bodies ist der Textkorpus

Die Prozedur beginnt mit dem Definieren eines RDDs aus der Textdatei aller Emailnachrichten (`textfile`).

```
1 val documents: RDD[Seq[String]] = sc.textFile(textFile)
2   .map(_.toLowerCase)
3   .map(_.split(" ").filter(_.length > 2).toSeq)
```

Diese Textdatei wird durch `ModelBuilder` regelmäßig ergänzt, sofern der IMAP-Client neue Nachrichten aus der Mailingliste empfangen hat.

Anschließend werden die Elemente des RDD `documents` von einer Sequenz von Wörtern auf einen Vektor der Worthäufigkeiten abgebildet:

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

```
val tf: RDD[Vector] = hashingTF.transform(documents)
```

Mit diesem RDD von Vektoren der Worthäufigkeiten wird nun ein Modell (`IDF`, *Inverse Document Frequency*) gebildet, dass die inversen Worthäufigkeiten berechnet. Dazu wird für jedes vorkommende Wort die Anzahl aller Dokumente ermittelt in denen es vorkommt.

Dieses Modell wird dann genutzt um die bereits berechneten Wortvorkommen pro Dokument anzupassen (`idf.transform(tf)`). Es entstehen die TF-IDF Vektoren:

```
val idf = new IDF().fit(tf)
val tfidf: RDD[Vector] = idf.transform(tf)
```

Der Relevanzvektor wird anschließend durch Aufsummieren der TF-IDF in der gewünschten Fenstergröße berechnet. Die Fenstergröße entspricht dem Bereich, dessen spezifische Wörter in der Streaming Layer für das Scoring der Tweets benutzt werden sollen.

```
val relevanceVector = tfidf
  .take(docWindowSize)
  .reduce((vector1, vector2) =>
    addSparseVectors(vector1.asInstanceOf[SparseVector],
                     vector2.asInstanceOf[SparseVector]))
```

3.3.5 Streaming Layer

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

Dazu wird über die Spark-Komponente *TwitterUtils*[8] eine Verbindung über HTTP zu einem Twitter-Endpunkt aufgebaut[9]. Über diese Verbindung werden - bei unpriveligiertem Zugriff - etwa 40 Tweets pro Sekunde über einen dauerhaften Datenstrom zur Verfügung gestellt (siehe Abb. 3.11).

In der Komponente *RealtimeAnalyzer* wurden vor dem Start des Datenstroms Funktionen zur Bewertung einzelner Tweets registriert.

Die Funktion `ScoreTweets` zur Bewertung einzelner Tweets

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

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

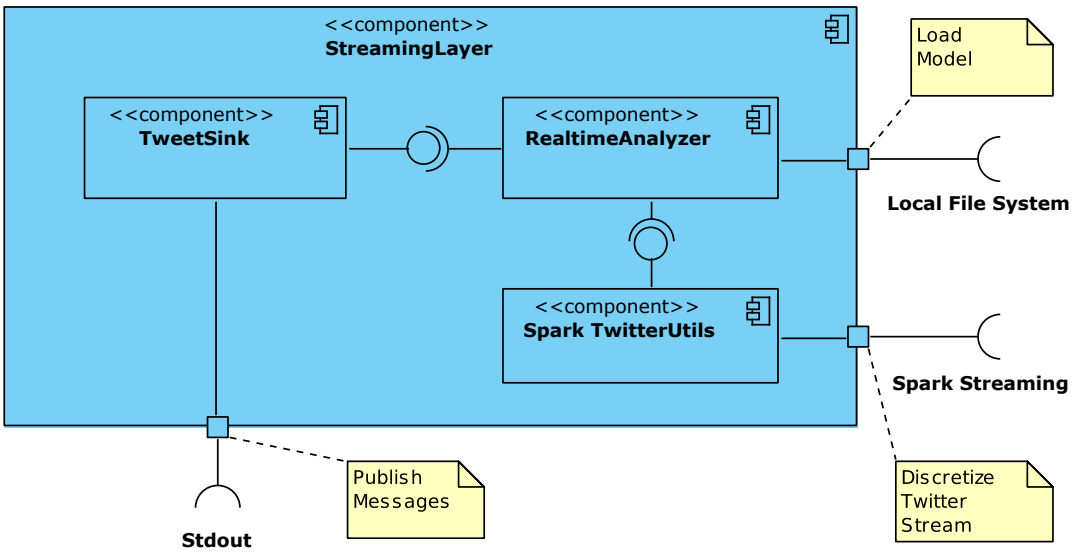

Abbildung 3.5: Innenansicht der Streaming-Layer Komponente

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

wird dabei wie in Listing 3.1 dargestellt implementiert:

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

// calculate score for each word, then sum and normalize the scores
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
      )}
  )
```

```
19 }
```

Listing 3.1: Bewertung von Tweets

Dabei 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. Zzurückgeben 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.

3.4 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. 3.7) 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 1.3).

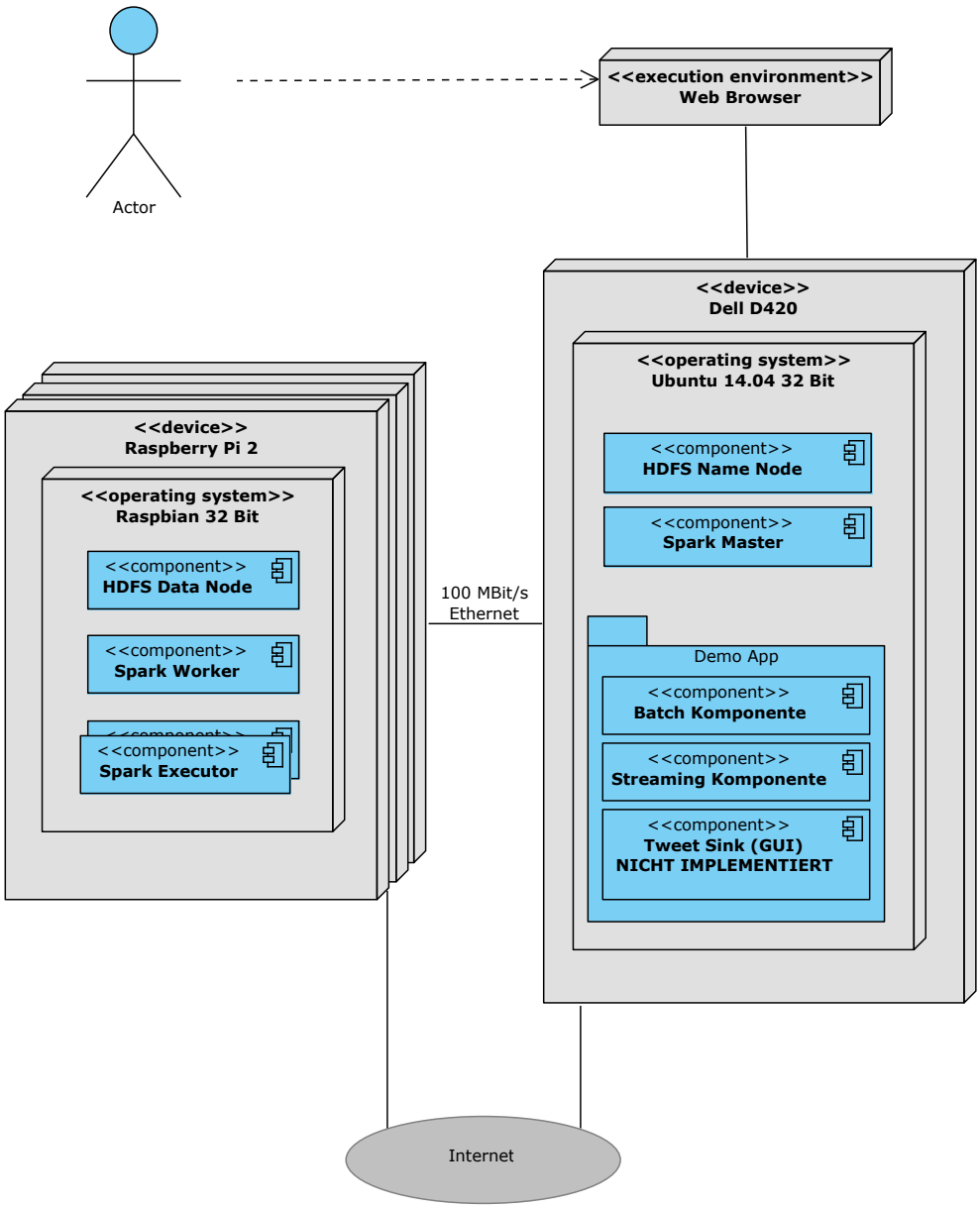

Abbildung 3.6: Verteilungssicht auf die Demo App

3.5 Ergebnisse und Bewertung

Zur Beurteilung des Laufzeitverhaltens wird die Anwendung in verschiedenen Konfigurationen des Raspberry-Pi-Clusters gestartet. Dabei wird jeweils das Verhalten der einzelnen

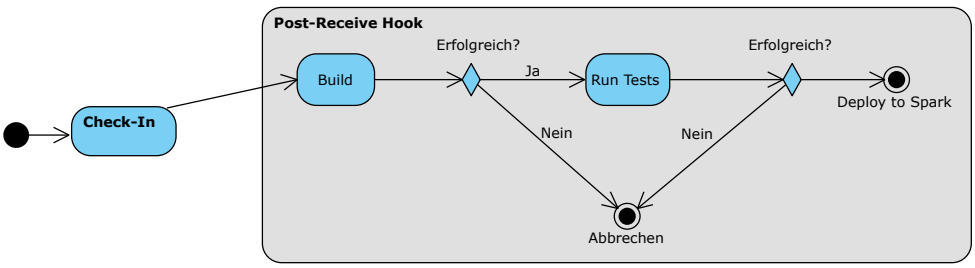

Abbildung 3.7: Einfache Continuous Deployment Pipeline

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.[10]

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:

[10] Bei den sparkspezifischen Messgrößen gibt es viele weitere, die nicht in die Auswertungen einbezogen werden.

1. Größe des Textkorpus: 1,5 GB
2. Fenstergröße des TF-IDF Vektors: 500 Nachrichten
3. Erlaubte Cores Pro Executor: 4
4. Erlaubter Arbeitsspeicher pro Executor: 384 MB
5. Zeitintervall für die Diskretisierung des Datenstroms (Streaming-Komponente): 5 Sekunden

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 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.

Das **Zeitintervall für die Diskretisierung des Datenstroms** richtet sich nach einem Wert, der für den behandelten Anwendungsfall noch als Echtzeit gelten kann. Ein höherer Wert würde den notwendigen Puffer erhöhen und vermutlich die Verarbeitungszeit pro Datenelement verringern. Ein niedrigerer Wert würde vermutlich den nötigen Puffer verringern und die relative Verarbeitungszeit eines Datenelementes erhöhen. In der folgenden Analyse der Laufzeitverhaltens wird jedoch deutlich, dass der tatsächliche Wert weitgehend unkritisch für die Stabilität der Komponente ist.

Um die Testläufe unter möglichst 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).

3.5.1 Batch-Komponente

Die Laufzeitmessung beginnt mit dem Start der ersten Aktion auf einem RDD und endet mit der Rückgabe des Relevanzvektors. Weil die Initialisierung erst zu diesem Zeitpunkt erfolgt, wird der gesamte Prozess von der Kontaktaufnahme zum Cluster und der Übermittlung der Tasks bis zur Rückgabe des Ergebnisses gemessen (siehe Listing 3.2).

```
1  val documents: RDD[Seq[String]] = sc.textFile(textFile)
2    .map(_.toLowerCase)                                    // stage 0
3    .map(_.split("_").filter(_.length > 2).toSeq)          // stage 0
4  val hashingTF = new HashingTF(1 << 20)
5  val tf: RDD[Vector] = hashingTF.transform(documents) // stage 0
6  tf.cache() // keep the tf cached because it will be used twice
7
8  val beginFeatureExtraction = System.currentTimeMillis()
9
10 val idf = new IDF().fit(tf)                             // stage 0/1
11 val tfidf: RDD[Vector] = idf.transform(tf)              // stage 2
12
13 val relevanceVector = tfidf
14   .take(docWindowSize)                                  // stage 2
15   .reduce((vector1, vector2) =>
16     addSparseVectors(vector1.asInstanceOf[SparseVector],
17                      vector2.asInstanceOf[SparseVector])
18   ) // end stage 2, execution ends here
19
20 val finishFeatureExtraction = System.currentTimeMillis()
```

Listing 3.2: Laufzeitmessung

Die Zeilen 1-6 in Listing 3.2 wurden in die Darstellung aufgenommen, um die Lineage vollständig darzustellen.

Der übrige Teil der Anwendung - insbesondere das Abrufen und Prozessieren neu eingegangener Emails - wird nicht berücksichtigt. Bei einem laufenden System sind die dort entstehenden Lasten so gering, dass kein Bedarf für Skalierung besteht.

Tabelle 3.4 zeigt die Laufzeiten der Modelbuilderkomponente bei verschiedenen Konfigurationseinstellungen des Clusters.

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 3.4: Skalierungsverhalten des ModelBuilders - mit * sind jeweils die schnellste und langsamste Konfiguration markiert

Einen besonderen Einfluss auf die Laufzeit hat offenbar die Blockgröße. Bei gleicher Anzahl von Workern und Replikaten verschlechtert sich die Laufzeit in jedem Versuch bei zunehmender Größe. Diese Beobachtung deckt sich mit den in Tabelle 3.3 dargestellten Eigenschaften des Festspeichers auf den Workern: Im Gegensatz zu den Durchsatzraten der mechanischen Festplatte des Masters verschlechtert sich auf den SD-Karten des Workers der Durchsatz bei zunehmender Blockgröße.
Ab einer Blockgröße von 128MB ist es zusätzlich nicht mehr möglich, vier Blöcke parallel im Arbeitsspeicher des Executor zu halten (384MB) und diese getrennt auf den 4 verfügbaren Cores verarbeiten zu lassen.

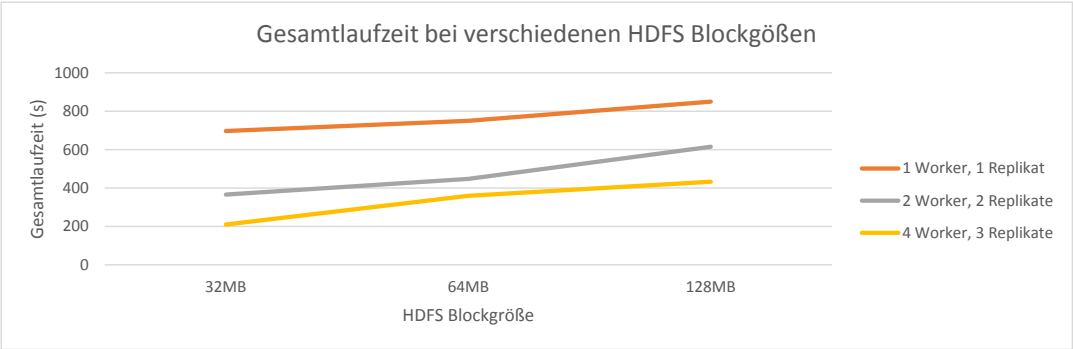

Abbildung 3.8: Laufzeit der Feature Extraction bei unterschiedlichen HDFS Blockgrößen

Ein Hinweis auf den Flaschenhals bei der Verarbeitung auf einem einzigen Knoten ist in Abbildung 3.9 zu erkennen. Die CPU steht unter Volllast, während die Leistung beim Lesen von der SD-Karte mit durchschnittlich 4,3 MB/s deutlich unter dem möglichen Durchsatz bleibt (siehe Tabelle 3.3).
Netzwerkdurchsatz spielt hier - wie bei einem einzelnen Knoten zu erwarten - offenbar keine Rolle.

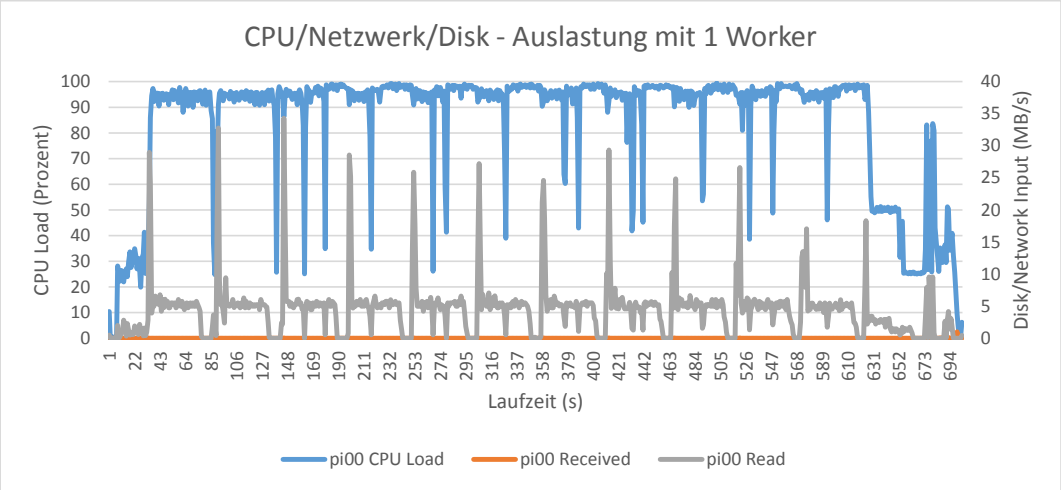

Abbildung 3.9: Netzwerk-, SD-Karten und CPU-Auslastung während bei einem Worker und 32MB Blockgröße

Betrachtet man die Summe der IO-Durchsätze über sämtliche Knoten, lassen sich deutlich die verschiedenen Phasen der Feature-Extraction erkennen. Abbildung 3.10 zeigt das kommentierte Diagramm eines Testlaufs mit 4 Workern, 2 Replikaten und einer HDFS-Blockgröße von 64MB.

Anmerkung: Die Methode `treeAggregate`[11] innerhalb der Methode `IDF.fit()` stellt einen Spezialfall unter den RDD-Transformationen dar. Sie besteht aus zwei Phasen, wobei in der ersten Phase lokal auf den Partitionen eine Aggregation von Datensätzen durchgeführt wird und anschließend über `reduce` die partitionsübergreifende Aggregation der bereits partiell aggregierten Datensätze durchgeführt wird.
Aus dieser Implementation ergibt sich die Netzwerk-Lastspitze ab Sekunde 251 und der Übergang zu Stage 1 (Abb. 3.10).

[11] https://github.com/apache/spark/blob/branch-1.3/core/src/main/scala/org/apache/spark/rdd/RDD.scala#L978

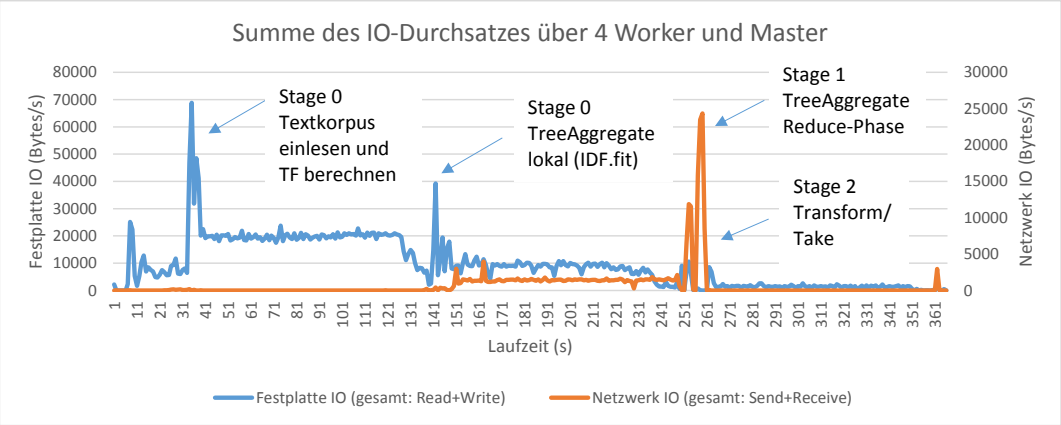

Abbildung 3.10: Auslastungskurven bei einem Worker und 64MB Blockgröße

3.5.2 Echtzeitkomponente

Einen ersten Blick auf den Zustand einer Spark-Anwendung ermöglicht die Weboberfläche des Treibers.

Jede Spark-Applikation startet gemäß der Standardeinstellungen auf dem Host des Treibers einen HTTP-Server, über den verschiedene Daten verfolgt werden können. In Abb. 3.11 sind die Statistiken einer laufenden Realtime-Komponente dargestellt.

In dem betrachteten Fall läuft die Realtime-Komponente mit einem einzelnen Worker, der den Empfänger für den Datenstrom von Twitter startet und das Scoring und Filtern der eingehenden Nachrichten ausführt. Um den Worker nicht für die Modellerzeugung der Batch-Komponente zu blockieren, wurde nur ein Executor mit 2 Cores gewährt.

Zum Zeitpunkt des Schnappschusses wurden 6260 Tweets empfangen und verarbeitet. Der letzte Batch enthielt 178 Twitter-Nachrichten (im Durchschnitt sind es etwa 200) und der Median für die Verarbeitungszeit liegt bei 373 Millisekunden (*Total Delay*).

Bei einem Zeitfenster von 5 Sekunden (*Batch Interval*) pro Batch gibt es also keinen Hinweis auf Performanceprobleme bei der Verarbeitung.

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 3.11: Spark-Dashboard des Realtime Analyzers - Statistics Tab

Dieser Verdacht erhärtet sich bei einem Blick auf die CPU-Last des verarbeitenden Workers (Abb. 3.12). Die durchschnittliche Last über den beobachteten Zeitraum liegt bei 22,5%. Die Last des Master sogar nur bei 8,1%.

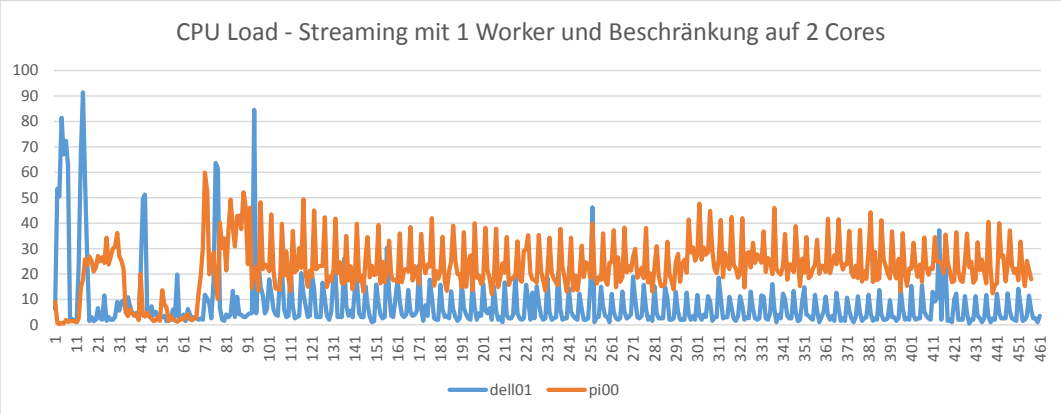

Abbildung 3.12: CPU Last bei Betrieb mit einem Worker

Auch die Netzwerklast ist minimal (Abb. 3.13). Ab Sekunde 51 liegt die Last des Workers bei durchschnittlich 77 KByte/s. Über den beobachteten Zeitraum von knapp 7 Minuten wurden damit 31.4 Megabyte empfangen. Die Last des Masters liegt noch darunter.

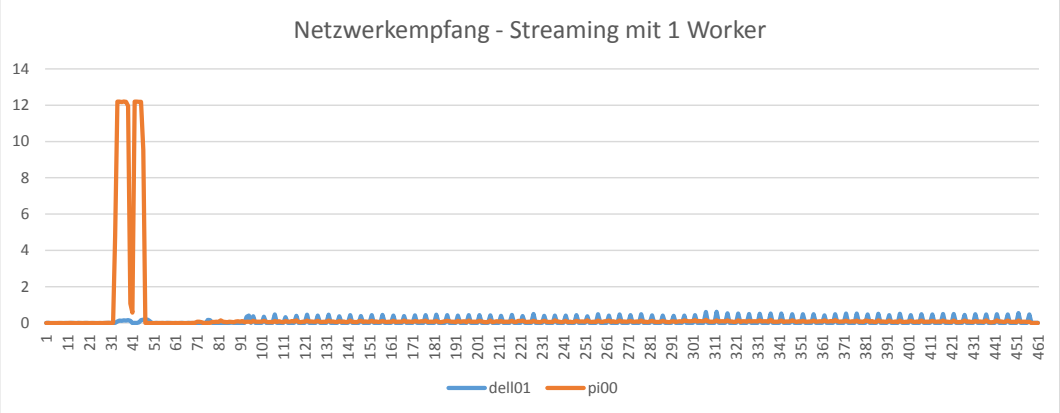

Abbildung 3.13: Netzwerklast bei Betrieb mit einem Worker

Betrachtet man den Speicherverbrauch (Abb. 3.14), sieht man, dass sich dieser nach wenigen Sekunden bei etwa 362MB stabilisiert.

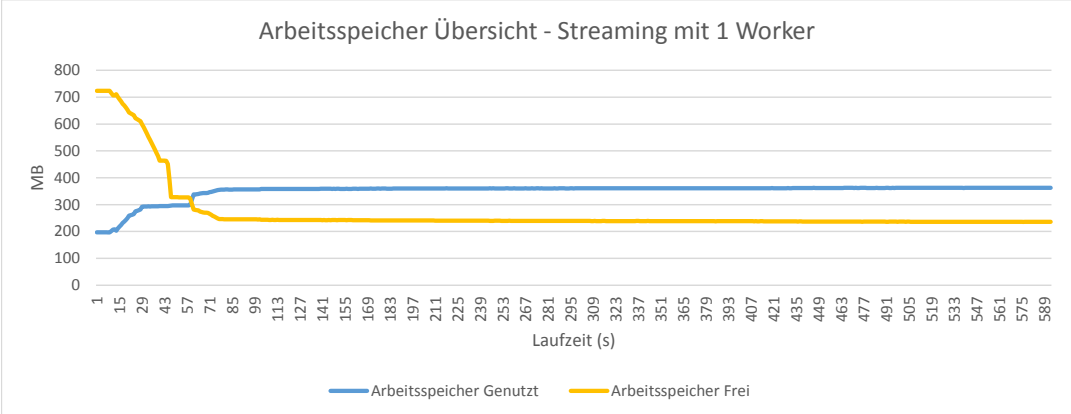

Abbildung 3.14: Nutzungsprofil des Arbeitsspeichers bei Betrieb mit einem Worker

4 Schlussbetrachtung

4.1 Diskussion der Ergebnisse

Es wurde ein umfassender Einblick in die Konzepte und Komponenten von Apache Spark gegeben. Für dieses System wurde daraufhin 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.

Dieses Experiment war erfolgreich. Es ist möglich einen Spark/Hadoop-Cluster auf der beschriebenen Hardware zu installieren 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 mit 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 komfortables 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).

Insgesamt verhielt sich Spark robust und im Betrieb transparent durch sehr gutes Logging. Die Dokumentation ist gut und die Integration externer Komponenten war einfach möglich.

Nach Meinung des Autors ist es wahrscheinlich, dass sich Spark in den nächsten Jahren weiterhin als wichtige Komponente im Big Data Bereich etablieren wird.

[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 Quellcode/Skripte (Auszüge)

1.1 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

1.2 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

1.3 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\
<emailaccount> <passwort>
```

Listing 4: Einfache Continuous Deployment Pipeline. Beispiel: ModelBuilder

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

2 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/r2.7.0/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml, abgerufen am 03.06.2015

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*. 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, 17. Juni 2015 Daniel Kirchner