FernUniversität in Hagen

–

Seminar 01912 / 19912
im Sommersemester 2017

„Skalierbare verteilte Datenanalyse“

Thema 2.3

Spark

Referent: Lukas Wappler

Inhaltsverzeichnis

1 Einleitung **3**

2 Apache Spark **4**
2.1 Kern-Bibliotheken / Komponenten 4
2.1.1 Grundlage des Systems (Spark-Core & RDDs) 6
2.1.2 SQL-Abfragen (Spark-SQL & Data Frames) 7
2.1.3 Verarbeitung von Datenströmen (Spark-Streaming) 8
2.1.4 Berechnungen auf Graphen (GraphX) 9
2.1.5 Maschinelles Lernen (MLlib) 10
2.1.6 Skalierung von R Programmen (SparkR) 11
2.2 Mehrere Komponenten im Verbund 12
2.3 Performance 13
2.3.1 Besonderheiten bei der Speichernutzung 13
2.3.2 Netzwerk und I/O-Traffic 14
2.4 Nutzung & Verbreitung 15

3 Fazit **16**

4 Ausblick & Weiterentwicklung **17**

5 Anhang **18**
5.1 Wörter zählen und Top 20 Ranking ausgeben 18
5.2 Beispielimplementierung für den Verbund mehrerer Spark Komponenten . . 19
5.2.1 Einlesen der Daten 19
5.2.2 Maschinelles Lernen 19
5.2.3 MapReduce und Ausgabe 20
5.2.4 Ergebnis 20

6 Literaturverzeichnis **21**

1 Einleitung

Das Thema Datenanalyse ist fester Bestandteil in vielen Bereichen des täglichen Lebens. Die Berechnung von Stau-Wartezeiten anhand von Handy- und Geolokations-Daten, die Wettervorhersage oder Werbemails von Onlinehändlern, die versuchen die nächsten benötigten Artikel vorzuschlagen sind nur einige Beispiele.

Durch den technologischen Fortschritt der vergangenen Jahre wird es zu einer immer größer werdenden Herausforderung die Unmengen an Daten zu beherrschen und zu verarbeiten. Als Beispiel sind Google, Twitter, Apple und viele weitere Unternehmen mit großen Datenbeständen aufzuführen. Schon 2010 wuchs der Datenbestand bei Twitter[1] täglich um 12 Terabyte [Win17].

Abhilfe schafft ein System wie Apache Spark, das in der Lage ist mit vielen Rechnern große Datenanalysen zu bewältigen.

Die Fülle an wissenschaftlichen Veröffentlichungen [Fou17g] auf der Apache Spark-Website sowie die zahlreichen Fachbücher zum Thema Datenanalyse [Par15], [Ryz+15], [ER16] oder direkt zu Apache Spark unterstreichen die Wichtigkeit und Aktualität des Themas.

Ziel dieser Arbeit ist es, das Apache Spark Framework in seinen Einzelheiten zu beleuchten und die einzelnen Komponenten vorzustellen. Des Weiteren wird auf die Performance und Fehlertoleranz näher eingegangen sowie an geeigneten Stellen kleine Beispielimplementierungen vorgestellt.

Zunächst wird der Kern des Systems und der prinzipielle Aufbau des Frameworks dargelegt. Danach werden die verschiedenen Komponenten, die das Framework enthält im Detail betrachtet. Dazu zählen SQL-Abfragen für die Anbindung von Datenbanken, die Verarbeitung von Datenströmen, Berechnungen von Graphen sowie maschinelles Lernen. Das Zusammenspiel mehrerer dieser Komponenten wird an einem größeren praktischen Beispiel in Abschnitt 2.2 verdeutlicht.

Das Framework wird danach einer kritischen Betrachtung der Performance unterzogen, wobei Kriterien wie Speichernutzung, Netzwerkauslastung oder Datentransfer eine große Rolle spielen. Das Ende bildet ein Fazit und ein Ausblick in die Zukunft.

[1] **Twitter** ist ein Mikrobloggingdienst. Nutzer können über das Portal Kurznachrichten verbreiten.

2 Apache Spark

Apache Spark ist ein Open Source Framework, das es ermöglicht verteilt über ein Cluster Programme und Algorithmen auszuführen. Zusätzlich ist das Programmiermodell bzw. die API zum Schreiben solcher Programme sehr einfach und elegant gehalten [Ryz+15].

Das Framework ist im Rahmen eines Forschungsprojekts, welches 2009 in der Universtiy of California in Berkeley im sogenannten AMPLab[2] ins Leben gerufen wurde, entstanden. Es ist 2010 als Open Source Software unter der BSD-Lizenz[3] zur Verfügung gestellt worden. Das Projekt wird seit 2013 von der Apache Software Foundation[4] weitergeführt. 2014 wurde es als Top Level Projekt eingestuft. Zum aktuellen Zeitpunkt steht Apache Spark unter der Apache 2.0 Lizenz[5] zur Verfügung.

2.1 Kern-Bibliotheken / Komponenten

Apache Spark besteht im wesentlichen aus fünf Modulen: Spark-Core, Spark-SQL, Spark-Streaming, Machine Learning Library (MLlib) und GraphX. Zur Nutzung der Komponenten gibt es eine Umfrage aus dem Jahr 2015. Diese ist in Abbildung 2.1 zu sehen.

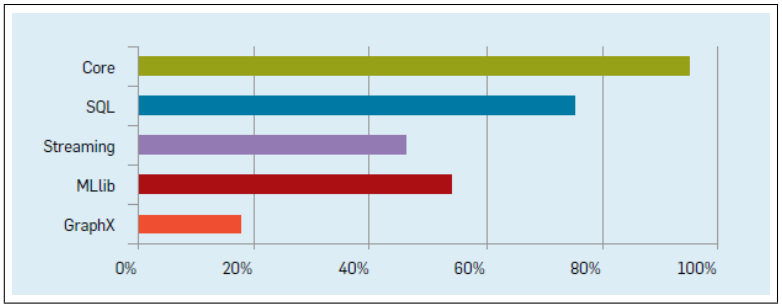

Abbildung 2.1: Nutzung der Komponenten [Zah+15]

[2] **AMPLab** ist ein Labor der Berkeley Universität in Californien, die sich auf Big-Data Analysen spezialisiert hat.

[3] **BSD-Lizenz** (Berkeley Software Distribution-Lizenz): bezeichnet eine Gruppe von Lizenzen, die eine breitere Wiederverwertung erlaubt.

[4] **Apache Software Foundation** ist eine ehrenamtlich arbeitende Organisation, die die Apache-Projekte fördert.

[5] Software, die einer **Apache 2.0 Lizenz** unterliegt, darf bis auf wenige Regeln und Auflagen frei verwendet und verändert werden.

Während Spark-Core die Kern-Komponente bildet und alle notwendigen Bausteine für das Framework mitbringt, sind die anderen Module auf dem Spark-Core Modul aufgebaut und befassen sich mit spezielleren Bereichen wie SQL, Streaming, maschinelles Lernen oder Graphenberechnungen. Abbildung 2.2 zeigt eine Übersicht der Komponenten.

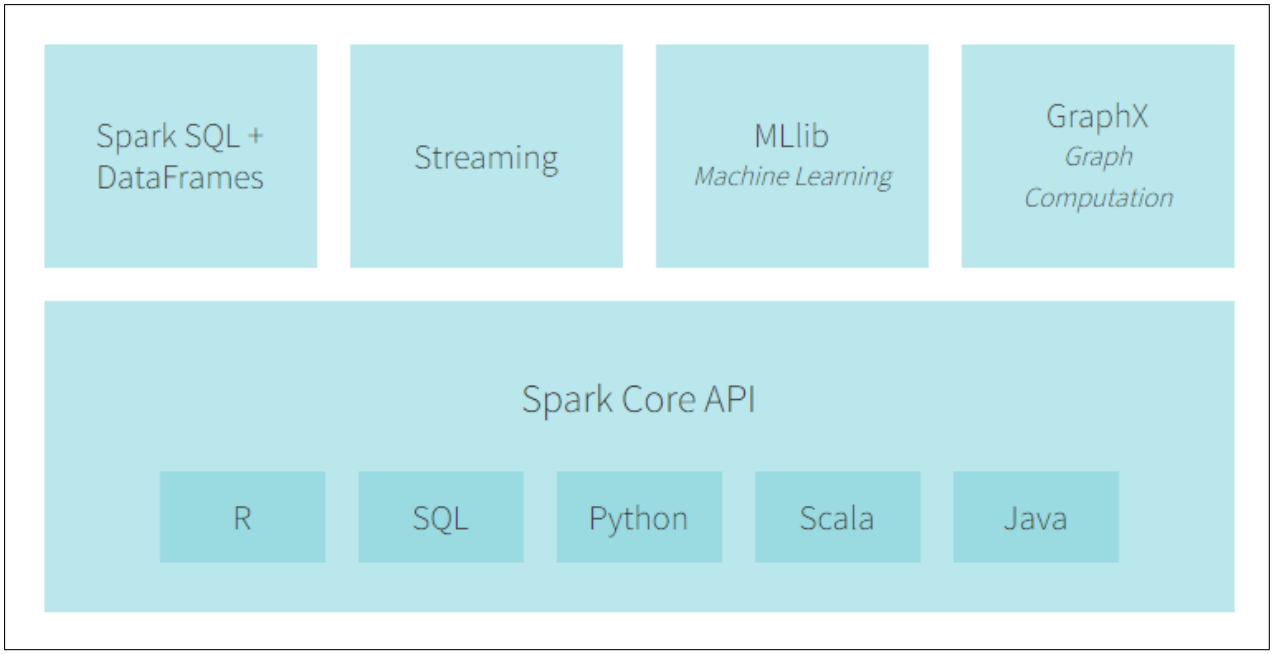

Abbildung 2.2: Apache Spark Framework [Dat17a]

Die Module werden in den folgenden Kapiteln von 2.1.1 bis 2.1.5 näher betrachtet.

Darüber hinaus wird in Kapitel 2.1.6 SparkR vorgestellt. Das Modul gehört nicht direkt zum Kern, jedoch bietet es die Möglichkeit die Datenanalysen mit R[6] zu optimieren bzw. zu beschleunigen und wird aus diesem Grund mit aufgeführt.

Den Einsatz der Kern-Komponenten ist im Anhang in Abschnitt 5.1 zu sehen. In diesem Beispiel werden die 20 meistgenannten Wörter, die in dieser Seminararbeit vorkommen, ermittelt. Dafür wird die Arbeit als PDF eingelesen und in eine Text-Datei konvertiert. Die Zeilen der Datei werden zuerst in Worte aufgeteilt und dann per MapReduce verdichtet und abschließend noch absteigend sortiert.

[6] **R** ist eine Programmiersprache für statistische Berechnungen und dem erstellen statistischer Grafiken.

2.1.1 Grundlage des Systems (Spark-Core & RDDs)

Spark-Core ist die Grundlage der Spark Plattform. Alle anderen Komponenten bauen auf diesem Kern auf. In dem Kern sind die grundlegenden infrastrukturellen Funktionen enthalten. Darunter zählen die Aufgabenverwaltung, die zeitliche Planung (Scheduling) sowie I/O Funktionen. Der Kern bietet zum Beispiel die Möglichkeit, Ergebnisse von Berechnungen direkt im Arbeitsspeicher zu halten und diese wiederzuverwenden. Das kann die Geschwindigkeit um das zehn bis 30 fache erhöhen [Ven+15]. Das grundlegende Programmiermodell besteht aus dem Arbeiten mit den Resilent Distributed Datasets (RDDs). Die API's werden in verschiedenen Sprachen (Java, Scala und Python) bereitgestellt. [Fou17b] In der Abbildung 2.2 sind die einzelnen Bausteine innerhalb der Spark-Core Komponenten / API noch einmal im Überblick zu sehen.

Die parallele Verarbeitung wird über den Spark Context realisiert. Der Spark Context wird im Hauptprogramm (Treiberprogramm) erzeugt und ist mit einem Cluster Manager verbunden. Dieser wiederum kennt alle Worker Nodes, welche die Aufgaben ausführen. Die Abbildung 2.3 zeigt wie Spark Context, Cluster Manager und die Worker Nodes zusammen agieren. Damit die Aufgaben über viele Nodes verteilt werden können, wird eine Datenstruktur benötigt die dafür ausgelegt ist. [ER16, S. 101]

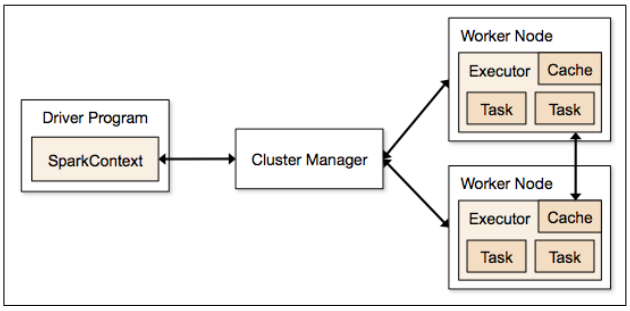

Abbildung 2.3: Spark Cluster aus [Fou17d]

Die RDDs, zu deutsch belastbare, verteilte Datensätze, erledigen diese Aufgabe. Das Dataset ist die primäre Datenabstraktion in Apache Spark. Ein RDD entspricht einer partitionierten Sammlung an Daten. Somit können die Partitionen auf verschiedene Systeme (bzw. Worker) verteilt werden.

Nach der Erstellung sind RDDs nur lesbar. Es ist nur möglich ein einmal definiertes RDD durch Anwendung globaler Operationen in ein neues RDD zu überführen. Die Operationen werden anschließend auf allen Partitionen des RDDs auf allen Worker Nodes angewendet.

Bei den Operationen wird zwischen Transformationen (z.B.: filter oder join) und Aktionen (z.B.: reduce, count, collect oder foreach) unterschieden. Transformationen bilden ein RDD auf ein anderes RDD ab. Aktionen bilden ein RDD auf eine andere Domäne ab. Eine Folge von Operationen wird Lineage[7] eines RDDs genannt [Zah+12], [Ryz+15, S. 11].

[7] **RDD Lineage** ist der logische Ablaufplan der einzelnen Operationen. Dadurch können Daten leicht wiederhergestellt werden, falls Fehler aufgetreten sind.

2.1.2 SQL-Abfragen (Spark-SQL & Data Frames)

Spark-SQL wurde 2014 veröffentlicht. Innerhalb der Spark-Familie wird diese Komponente am stärksten weiterentwickelt. Spark-SQL entstammt dem Apache-Shark. Dadurch sollten folgenden Probleme, die es in Apache Shark gab, gelöst bzw. verbessert werden.

1. Mit Apache Shark ist es nur möglich auf Daten im Hive[8] Katalog zuzugreifen.
2. Shark lässt sich nur über selbst geschriebene SQL Abfragen aufrufen.
3. Hive ist nur für MapReduce optimiert

Es werden zwei wesentliche Anwendungsfälle kombiniert. Zum einen ermöglicht es relationale Datenbank-Anfragen zu schreiben und zum anderen prozedurale Algorithmen einzusetzen. Dafür werden neben den RDDs die DataFrames als weitere Datenstruktur eingeführt.

Die Abfragen werden zuerst in den DataFrame-Objekten gespeichert. Erst nach der Initialisierung werden diese SQL Abfragen dann ausgewertet. Für die Auswertung und Optimierung kommt Catalyst[9] zum Einsatz. Nach der Auswertung werden die Abfragen gegebenenfalls optimiert und danach in RDDs überführt. In Abbildung 2.4 sind die Phasen vom SQL-Query bis hin zu den RDDs dargestellt. In den Boxen mit abgerundenten Ecken befinden sich Catalyst-Trees.

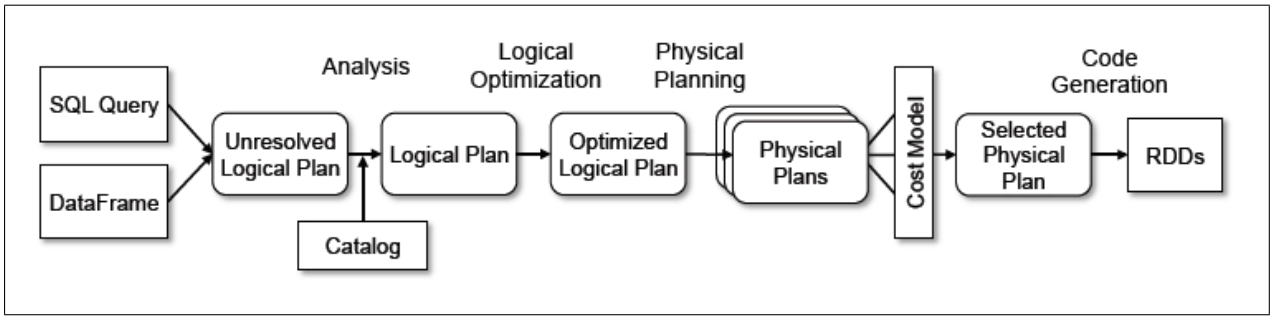

Abbildung 2.4: Phasen der Query Planung in Spark-SQL [Arm+15b]

Mit Spark-SQL kann ein Zugriff auf relationale Datenbanken erfolgen. Es wurde eine hohe Performance aufgrund etablierter DBMS-Techniken erreicht. Neue Datenquellen lassen sich leicht anschließen und integrieren. Zusätzliche Erweiterungen wie maschinelles Lernen und Berechnungen von Graphen sind nutzbar [Arm+15b].

[8] **Apache Hive** ist eine Erweiterung für Hadoop und ermöglicht Abfragen über SQL zu nutzen.
[9] **Catalyst** ist eine Optimierungsengine für relationale Ausdrücke.

2.1.3 Verarbeitung von Datenströmen (Spark-Streaming)

Die Spark-Streaming Bibliothek ermöglicht das Verarbeiten von Datenströmen. Dabei dienen RDDs als Grundlage. Die RDDs werden zu DStreams erweitert. DStreams (discretized streams) sind Objekte, die zeitabhängige Informationen enthalten. DStreams verwalten intern eine Sequenz von RDDs und werden aus diesem Grund diskrete Streams genannt. Auch DStreams haben die bereits aus 2.1.1 bekannten zwei Operationen (Transformation und Aktion).

Um Datenströme zu empfangen wird ein Empfänger (Receiver) auf einem Worker-Knoten gestartet. Die eingehenden Daten werden in kleinen Datenblöcken gespeichert. Dafür werden die Daten innerhalb eines vorgegebenen Zeitfensters gepuffert. Pro Zeitfenster werden die Daten in dem Puffer in eine Partition eines RDD abgelegt [ER16, S. 123-130].

In der Spark-Streaming Bilbliothek sind bereits einige Empfänger wie Kafka[10] und Twitter enthalten.

Abbildung 2.5 zeigt den Ablauf vom Eingang der Daten über die Verarbeitung bis hin zur Ausgabe. Diese Daten können zum Beispiel auf einem Dashboard ausgegeben oder in einer Datenbank gespeichert werden.

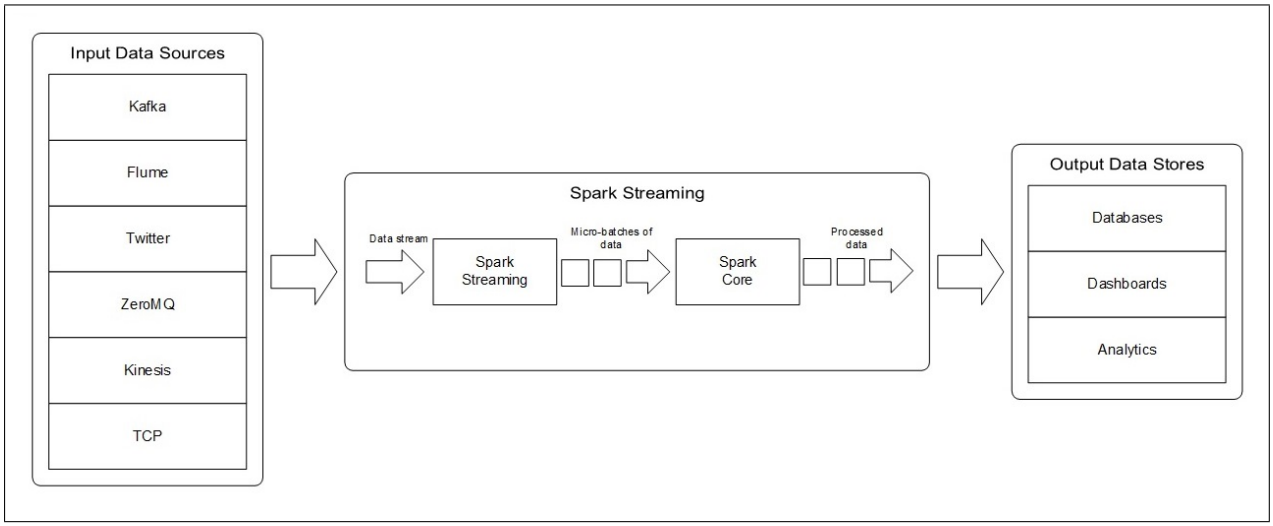

Abbildung 2.5: Spark-Streaming Ablauf [Fou17c]

[10] **Apache Kafka** dient zur Verarbeitung von Datenströmen.

2.1.4 Berechnungen auf Graphen (GraphX)

Das GraphX Framework ermöglicht die Berechnungen auf Graphen. Die Grundlage sind die RDDs. Als Graphenstrukturen werden Property-Graphen genutzt. Das sind gerichtete Multigraphen. Das bedeutet, der Graph besteht aus Knoten (Vertex) und Kanten (Edge). An den Kanten können Eigenschaften hinterlegt sein.

In dem GraphX Framework werden diese Graphen aus RDD-Tupeln gebildet. In dem ersten RDD sind die Knoten und in dem zweiten die Kanten enthalten. Um die Graphen auf mehrere Maschinen zu verteilen, werden diese entlang der Kante geteilt. Es handelt sich um das sogenannte Edge Cut Verfahren. Ein einzelner Knoten kann somit auf mehreren Maschinen existieren. Um Änderungen an einem Knoten über alle Kopien auf den Maschinen zu propagieren, wird zusätzlich eine Routing-Tabelle gepflegt. Über diese sind alle Kopien von Knoten bekannt und bei Änderungen eines Knotens werden alle Maschinen entsprechend informiert. In der folgenden Abbildung 2.6 ist ein verteilter Property-Graph sowie die dazugehörigen Tabellen dargestellt. Zusätzlich sind die verschiedenen RDDs für Knoten, Kanten und die Routing-Tabelle abgebildet [Ryz+15].

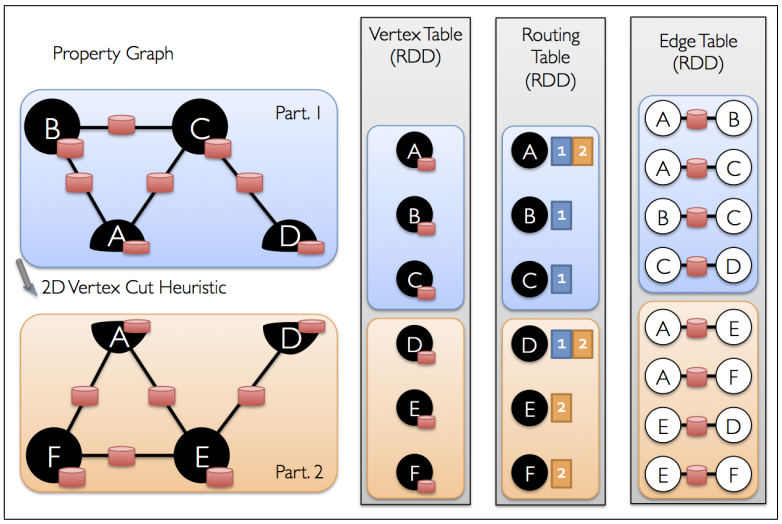

Abbildung 2.6: Property-Graph mit den dazugehörigen RDDs [Fou17e]

Mit Hilfe dieser Technik ist es möglich große Graphen zu zerteilen und auf die Maschinen zu verteilen um dort die gewünschten Operationen durchzuführen. Über die verschiedenen Tabellen können nach der Bearbeitung der Aufgaben die Teil-Graphen wieder zusammengeführt werden. Auch Ausfälle einzelner Nodes oder Datenverluste können somit korrigiert werden.

2.1.5 Maschinelles Lernen (MLlib)

MLlib(Machine Learning library) ist eine Bibliothek für maschinelles Lernen. Diese bietet die Möglichkeit, typische maschinelle Lern-Algorithmen auf verteilten Spark-Systemen zu nutzen. Zur Datenabstraktion wird das bereits in 2.1.2 erwähnte DataFrame genutzt.

In einem Maschinenlernprogramm läuft eine Sequenz von Algorithmen, einer sogenannten Pipeline, ab um die Daten zu verarbeiten und davon zu lernen. Dafür gibt es in der MLlib Transformers und Estimator als Pipeline-Komponenten. Die Transformers verändern die DataFrames. Das DataFrame wird gelesen, die Daten werden anders strukturiert oder aufbereitet und in einem neuen DataFrame wieder ausgegeben. Diese nutzen die Methode *transform()*. Die Estimators sind Abstraktionen eines Lernalgorithmus. Sie erzeugen Transformer aus dem übergebenen DataFrame. Diese Estimator nutzen die Methode *fit()*. Eine Pipeline selbst ist wiederum ein Estimator [Ryz+15].

Das Zusammenspiel zwischen Transformers und Estimators ist in der Abbildung 2.7 beispielhaft dargestellt.

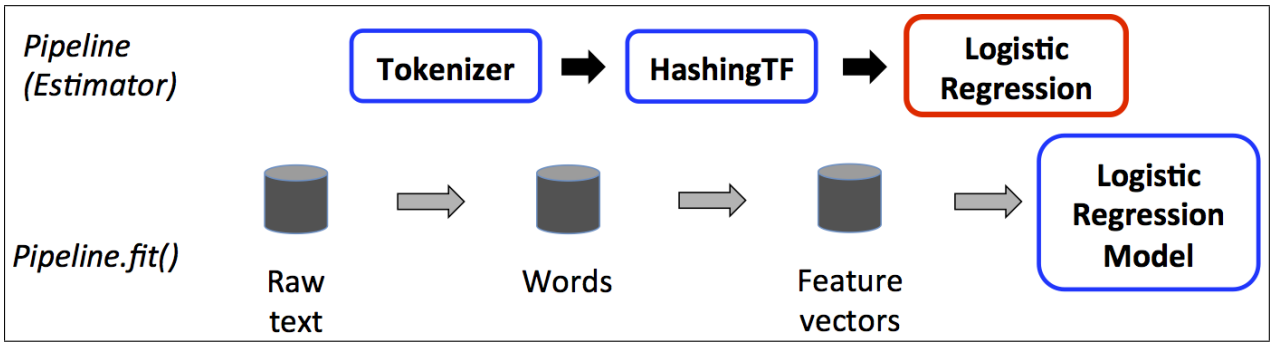

Abbildung 2.7: MLlib Pipeline [Fou17f]

Ein Text wird eingelesen. In den ersten zwei Schritten (Tokenizer und HashingTF) arbeiten Transformatoren. Im dritten Schritt arbeitet ein Estimator (Logistic Regression). Die einzelnen Schritte werden im Abschnitt 2.2 noch etwas genauer beschrieben.

2.1.6 Skalierung von R Programmen (SparkR)

R hat den Nachteil, dass es zur Laufzeit nur auf einem einzelnen Thread arbeitet. Diese Hürde kann mit der SparkR Erweiterung genommen werden. SparkR ist ein R Paket, welches es ermöglicht, eine einfache Schnittstelle bereitzustellen, um Apache Spark von R auszunutzen. SparkR nutzt das bereits bekannte DataFrame, welches die Operationen wie *selection*, *filtering* oder *aggregation* bereitstellt. Das sind genau jene Operationen, die dem Anwender aus R bereits bekannt sind.

Um das zu ermöglichen ist eine Brücke von R hin zum Spark Context bzw. den Nodes / Workern notwendig. Das Architekturschaubild in Abbildung 2.8 zeigt diesen Ansatz.

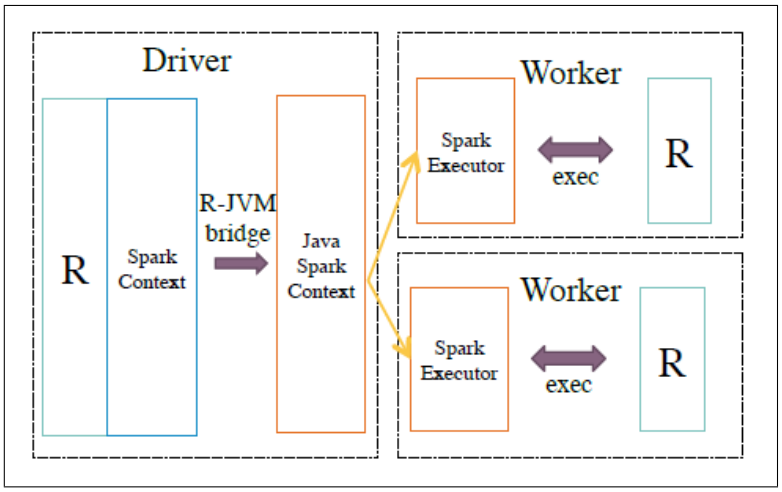

Abbildung 2.8: SparkR Architektur [Ven+15]

Die R-JVM bridge ermöglicht es von R aus Funktionen der JVM[11] aufzurufen. Somit wird vom R Spark Context über die bridge zum Java Spark Context kommuniziert. Für die Kommunikation werden Sockets, die auf Netty[12] basieren, genutzt. Im Vergleich zu anderen Kommunikationsmethoden ist der zusätzliche Aufwand bei Sockets vergleichsweise gering und somit zu vernachlässigen [Ven+15].

Durch den Einsatz von SparkR kann die Verarbeitung auf vielen Rechnern verteilt stattfinden. Auch alle weiteren Merkmale wie In-Memory Cachings oder maschinelles Lernen können von Spark genutzt und ein großer Laufzeitgewinn erzielt werden. Bei einem Versuch wurden drei Abfragen mit acht bis 64 Kernen durchgeführt. Dabei konnte die Zeit von 115 Sekunden auf 20 Sekunden reduziert werden. Mit zusätzlichen Caching konnten die 20 Sekunden nochmal auf weniger als drei Sekunden reduziert werden [Ven+15].

[11] **JVM** (java virtual machine) ist ein Teil der Java-Laufzeitumgebung. Der kompilierte Java-Bytecode wird innerhalb dieser virtuellen Maschinen ausgeführt.
[12] **Netty** ist ein High-Performance Netzwerk Framework

2.2 Mehrere Komponenten im Verbund

Nachdem die einzelnen Komponenten vorgestellt wurden, soll anhand eines praktischen Beispiels das Zusammenspiel verdeutlicht werden.

Es soll folgende Aufgabenstellung mit Apache Spark realisiert werden: Es gibt eine Datenmenge bestehend aus 100.000 Mitarbeiter-Datensätzen. Zu jedem Mitarbeiter sind Informationen wie Vorname, Nachname, Gehalt, Eintritt in die Firma, Adresse sowie Interessen/Hobbys bekannt. Diese Daten sind im strukturierten JSON[13]-Format hinterlegt. Aus diesem Datensatz sollen alle Mitarbeiter gefunden werden, die als Hobby Astrologie angegeben haben. Zusätzlich sollen diese auf keinen Fall Interesse an Autorennen und Bowling haben. Des Weiteren sollen die gefundenen Mitarbeiter nach ihrem Alter gruppiert werden. Diese Aufgabenstellung macht es möglich die folgenden Komponenten zu nutzen:

- die Kernklassen zum Sortieren und Reduzieren der Maps
- SparkSQL für das Einlesen der JSON-Datei und SQL-Abfragen
- SparkMLlib für das Trainieren der gesuchten Begriffe und Finden der in Frage kommenden Mitarbeiter[14]

Zuerst wird die Spark-Session erzeugt und im Anschluss daran werden die Daten eingelesen. Neben JSON wären auch noch viele weitere Datenformate möglich gewesen (z.B.: csv, jdbc, text). Das lässt sich alles über die Spark-Core realisieren. Der entsprechende Code-Block ist im Anhang in Unterabschnitt 5.2.1 zu sehen.

Als nächstes wird eine Pipeline aufgebaut, in der ein PipelineModel so trainiert werden soll, dass es später aus dem Datensatz alle passenden Mitarbeiter mit möglichst hoher Trefferrate findet. Die Pipeline besteht aus drei Stages: Dem Tokenizer[15], dem HashingTF[16] und der LogisticRegression[17]. Nach dem Training werden die Daten in ein neues Dataset transformiert. In diesem Dataset sind die Vorhersagen enthalten. Die beschriebenen Schritte sind als Code im Anhang in Unterabschnitt 5.2.2 zu sehen.

Zum Schluss werden die Mitarbeiter, die vorhergesagt wurden, per SparkSQL mit einer SQL Abfrage herausgefiltert und über MapReduce und Sortierung aus der SparCore Bibliothek in die endgültige Form gebracht. Das Implementierung dafür ist im Unterabschnitt 5.2.3 dargestellt.

Es wurden 285 Mitarbeiter gefunden, die in Frage kommen. Darunter liegt der Großteil zwischen 25 und 35 Jahren. Das vollständige Ergebnis ist unter Unterabschnitt 5.2.3 zu finden.

Die Komponenten greifen sehr gut ineinander. Die Schnittstellen und Funktionen wirken sehr einheitlich und es entsteht der Eindruck, dass das Hinzunehmen einer weiteren Spark Komponente kein Fremdkörper, sondern eine hilfreiche Ergänzung darstellt.

[13] **JavaScript Object Notation (JSON)** ist ein kompaktes Datenformat für den Datenaustausch.
[14] Das Beispiel aus [Git17b] diente als Vorlage
[15] Verarbeitet Text und teilt ihn in kleinere Stücke. In dem Fall in einzelne Worte, Auszug aus dem Datensatz: 'interests':'R/C Helicopters,Frisbee Golf – Frolf,Tetris'
[16] Erzeugt zu den Worten Hashes, um ein besseres Vergleichen zu ermöglichen.
[17] Erzeugt eine Voraussage zu dem Datensatz gegenüber den Trainingsdaten.

2.3 Performance

Die Analyse von Performance Problemen erweist sich mitunter als sehr schwierig. Oft ist es nicht einfach die kritische Stelle zu lokalisieren und zusätzlich sind es meist mehrere Stellen, die erst in Summe zu Problemen führen. Apache Spark bringt zwar die seiteneffektfreie API mit, jedoch können trotzdem jede Menge Probleme auftreten. Für Entwickler ist es immer schwer im Hinterkopf zu behalten, dass Operationen auf vielen verteilten Rechnern ablaufen. Es unterscheidet sich doch stark von einem klassischen liniear ablaufenden Single-Thread Programm.

Über eine webbasierte Übersicht, die in Abbildung 2.9 zu sehen ist, können Informationen zu aktuell laufenden Auswertungen und Dauer von Ergebnissen etc. überwacht werden [Ryz+15, S. 12].

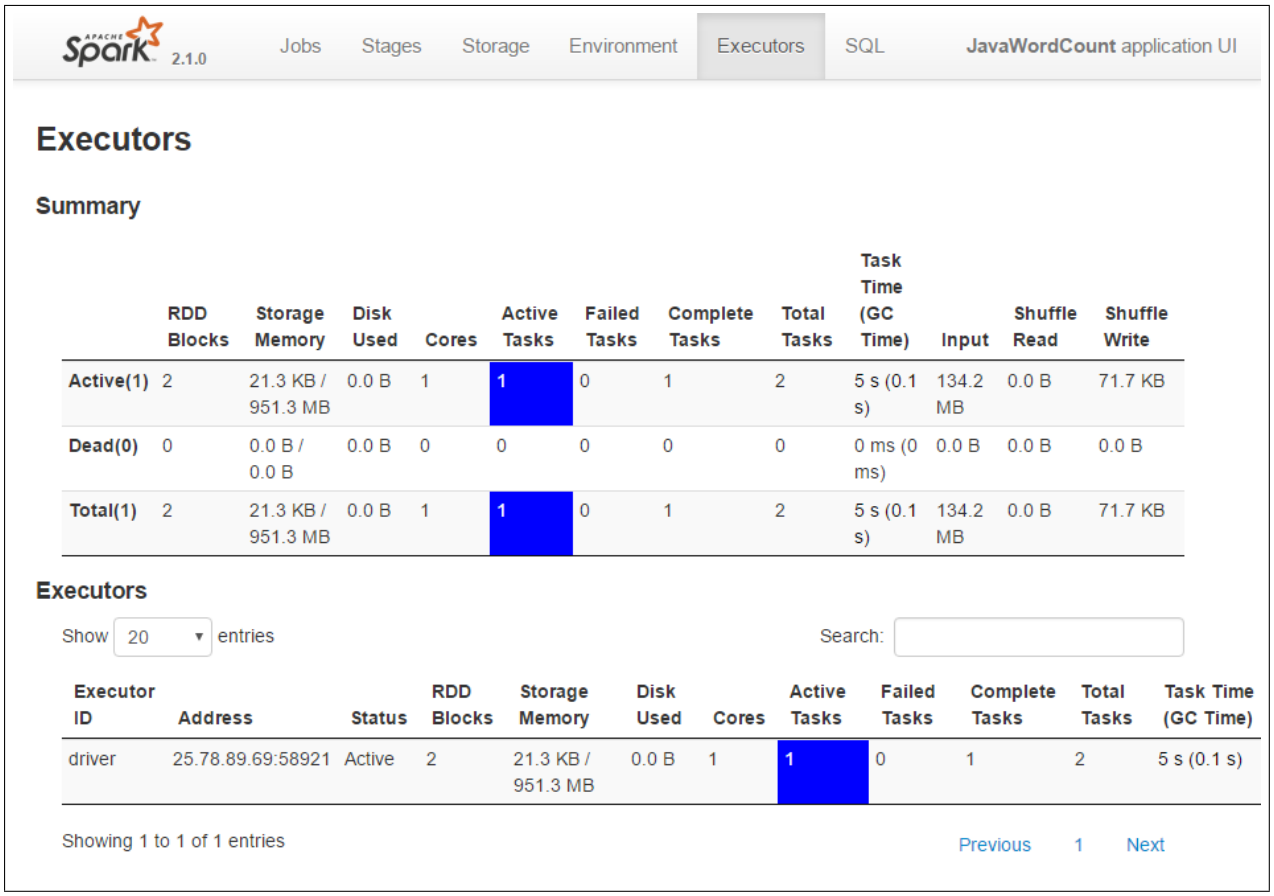

Abbildung 2.9: Spark Web UI: Zusammenfassung der Worker

Speziell beim Thema SQL-Abfragen ist es enorm wichtig, dass man sich für die richtigen Anweisungen entscheidet, um langsame Operationen zu vermeiden. Wenige Änderungen an SQL-Abfragen können sehr große Geschwindigkeitsunterschiede bewirken.

2.3.1 Besonderheiten bei der Speichernutzung

Die Wahl einer geeigneten bzw. speicherplatzeffizienten Datenstruktur wird oftmals unterschätzt. Gerade bei großen Datenmengen und oftmals komprimierten Daten können schnell große Speichermengen benötigt werden. Das wird im folgenden Beispiel verdeutlicht.

Spark geht davon aus, dass eine Datei in Blöcke einer bestimmten Größe geladen wird. In der Regel beträgt die Blockgröße 128MB. Zu beachten ist jedoch, dass beim Dekomprimieren größere Blöcke entstehen können. So können aus 128MB schnell 3-4GB große Blöcke in dekomprimiertem Zustand werden.

Um das Speichermanagement zu verbessern, wurde ein per-node allocator implementiert. Dieser verwaltet den Speicher auf einer Node. Der Speicher wird in drei Bereiche geteilt:

- Speicher zum Verarbeiten der Daten
- Speicher für die hash-tables bei Joins oder Aggregations
- Speicher für „unrolling“ Blöcke, um zu prüfen ob die einzulesenden Blöcke nach dem Entpacken immer noch klein genug sind, damit diese gecached werden können.

Damit läuft das System robust für Anwendungsbereiche mit sehr vielen Maschinen sowie mit ganz wenigen [Arm+15a].

2.3.2 Netzwerk und I/O-Traffic

Mit Apache Spark wurden Operationen, bei denen über 8.000 Nodes involviert waren und über 1PB an Daten verarbeitet wurden, durchgeführt. Das beansprucht die I/O Schicht enorm. Um I/O Probleme zu vermeiden, bzw. diese besser zu beherrschen, wurde als Basis das Netty-Framework verwendet.

- Zero-copy I/O:
  Daten werden direkt von der Festplatte zu dem Socket kopiert. Das vermeidet Last an der CPU bei Kontextwechseln und entlastet zusätzlich den JVM garbage collector[18].
- Off-heap network buffer management:
  Netty verwaltet einige Speichertabellen außerhalb des Java Heap Speichers, um Probleme mit dem JVM garbage collector zu vermeiden.
- Mehrfache Verbindungen:
  Jeder Spark worker kann mehrere Verbindungen parallel bearbeiten.

[18] **garbage collector** kommt in Java zur automatischen Speicherbereinigung zum Einsatz

2.4 Nutzung & Verbreitung

Durch die Unterstützung der drei Programmiersprachen Scala, Python und Java ist die Arbeit mit Apache Spark einfacher, als wenn nur eine einzige exotische Programmiersprache verwendet werden könnte.

Apache Spark unterstützt zudem noch verschiedene Datenquellen und Dateiformate. Zu den Datenquellen zählen das Dateisystem S3[19] von Amazon und das HDFS[20]. Die Dateiformate können strukturiert (z.B.: CSV, Object Files), semi-strukturiert (z.B.: JSON) und unstrukturiert (z.B.: Textdatei) sein.

Zu den Mitwirkenden(Contributors) zählen über 400 Entwickler aus über 100 Unternehmen (Stand 2014). Es gibt über 500 produktive Installationen [Arm+15a].

Seit einigen Jahren finden weltweit jährlich, unter dem Namen „Spark Summit“ Konferenzen statt [Fou17a].

In einer Umfrage von Heise.de aus dem Jahr 2015 wurden 2.136 Teilnehmer zur Nutzung von Apache Spark befragt [Sch17]. 31% der Befragten gaben an, den Einsatz derzeit zu prüfen, 13% nutzen bereits Apache Spark und 20% planten den Einsatz noch in dem damaligen Jahr. Die Nutzung innerhalb verschiedener Berufsgruppen war sehr ähnlich. Mit 16% lag bei den Telekommunikationsunternehmen der Einsatz am höchsten.

Bei denen, die es bereits im Einsatz haben, lag Scala bei den Programmiersprachen mit großem Abstand vorn. Eine detaillierte Übersicht ist in Abbildung 2.10 zu sehen.

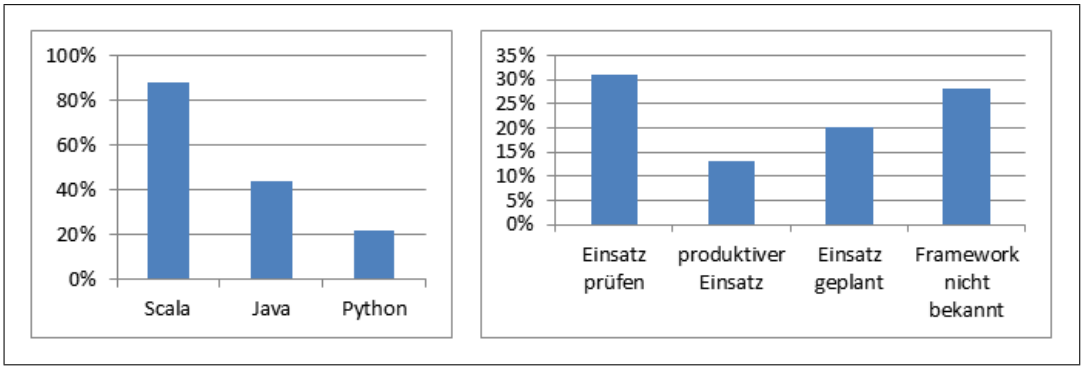

Abbildung 2.10: Verbreitung & Einsatz

[19] **S3** (Simple Storage Service) ist ein Filehosting-Dienst von Amazon der beliebig große Datenmengen speichern kann

[20] **HDFS** (Hadoop Distributed File System) ist ein hochverfügbares Dateisystem zur Speicherung sehr großer Datenmengen

3 Fazit

Während der Analyse und Arbeit mit Apache Spark hat sich die Vielfalt der Einsatzgebiete sehr eindrucksvoll gezeigt. Es hat Stärken in vielen Bereichen wie SQL, maschinellem Lernen sowie Speichernutzung. Oftmals kann es mit den Speziallösungen für spezielle Analysen mithalten oder ist sogar schneller.

Nach einer kurzen Einarbeitung gestaltete sich das Programmieren mit dem Framework als sehr angenehm. Ab und zu waren die vielen Transformationen der RDDs und auch die verschiedenen Datentypen der einzelnen Komponenten etwas schwierig zu handhaben. Insbesondere bei Nutzungen mehrerer Komponenten wie in Abschnitt 2.2 gezeigt wurde.

Aufgrund der Fülle an Informationen ist es teilweise schwierig gewesen, die Informationen der vielen verschiedenen Versionen auseinander zu halten. Zusätzlich mussten die gefundenen Informationen in News, Blogs oder Foren immer zeitlich genau eingeordnet werden, um sicher zu gehen, das die dort veröffentlichten Informationen nicht bereits veraltet sind.

Der Einsatz empfiehlt sich, wenn eine kostengünstige und sehr flexible Lösung bevorzugt wird.

4 Ausblick & Weiterentwicklung

Immer mehr Firmen führen Apache Spark ein oder nutzen es bereits [Sch17]. Dieser Trend sollte weiter fortgeführt werden.

Seit der Einführung von Apache Spark im Jahr 2010 wird die Software kontinuierlich verbessert und weiterentwickelt. Einen großen Anteil daran hat die Community, die aufgrund der Open-Source Software in der Lage ist, aktiv an der Weiterentwicklung mit zu arbeiten. Die Kommunikation innerhalb der Community findet im wesentlichen über offizielle Mailinglisten und einem Ticket-System der Apache Foundation statt. Der Code liegt auf GitHub[21] und ist öffentlich für jeden zugänglich. Bis zum 10.04.2017 gab es bereits 51 Releases oder Release-Kandidaten, 19,365 commits und 1,053 contributors[22]. Mit einem weiteren Anstieg ist zu rechnen. Im ersten Quartal 2017 gab es über 717 commits. Ein Einbruch der Aktivität ist momentan nicht zu erkennen [Git17a].

Von der Version 1.6 auf die Version 2.0 gab es nochmals eine relativ starke Performancesteigerung. Vermutlich werden solche Performancesteigerungen zukünftig schwieriger zu erreichen sein. Trotzdem sollten sich Geschwindigkeiten bei der Verarbeitung solcher großen Datenmengen auch zukünftig noch etwas nach unten verändern. Eine Übersicht der Performanceänderungen ist in der Tabelle 4.1 zu sehen [Dat17b].

primitive	Spark 1.6	Spark 2.0
filter	15ns	1.1ns
sum w/o group	14ns	0.9ns
sum w/ group	79ns	10.7ns
hash join	115ns	4.0ns
sort (8-bit entropy)	620ns	5.3ns

Tabelle 4.1: Kosten pro Zeile (cost per row) auf einem einzelnen Thread

Zukünftig ist denkbar, dass noch weitere Komponenten, wie zum Beispiel SparkR, dazu kommen. Auch das Anbinden weiterer Datenquellen wird sehr wahrscheinlich weiter vorangetrieben werden.

[21] **GitHub** ist ein webbasierter Onlinedienst, der die Möglichkeit bietet Softwareprojekte mit der Versionsverwaltung Git zu verwalten.

[22] **contributors**: Sind Personen, die zum Projekt mit Schreiben von Code beigetragen haben.

5 Anhang

5.1 Wörter zählen und Top 20 Ranking ausgeben

```
1     //create spark context
2     String master = "local[4]";
3     String appName = "SeminararbeitTop20Woerter";
4     JavaSparkContext javaSparkContext = new JavaSparkContext(master, appName);
5
6     convertPdfToTextfile(PATH_TO_PDF_FILE, PATH_TO_TXT_FILE);
7
8     //read lines and split to words
9     JavaRDD<String> lines = javaSparkContext.textFile(PATH_TO_TXT_FILE);
10    JavaRDD<String> words = lines.flatMap(s -> Arrays.asList(SPACE.split(s)).
          iterator());
11    JavaRDD<String> filteredWords = words.filter(word -> word.length() != 0);
12    filteredWords = filteredWords.filter(word -> !word.equals("."));
13
14    //count words
15    JavaPairRDD<String, Integer> ones = filteredWords.mapToPair(s -> new Tuple2
          <>(s, 1));
16    JavaPairRDD<String, Integer> counts = ones.reduceByKey((i1, i2) -> i1 + i2);
17
18    //swap the tupel
19    JavaPairRDD<Integer, String> map = counts.mapToPair(item -> item.swap());
20
21    //sort the key (word count)
22    boolean ascending = false;
23    JavaPairRDD<Integer, String> sortByKey = map.sortByKey(ascending);
24    List<Tuple2<Integer, String>> output = sortByKey.collect();
25
26    int numberOfLines = 20;
27    printToConsole(output, numberOfLines);
28
29    javaSparkContext.close();
```

5.2 Beispielimplementierung für den Verbund mehrerer Spark Komponenten

5.2.1 Einlesen der Daten

```
1  public class AnalyseEmployees {
2
3    public static void main(String[] args) {
4
5      setSystemParameters();
6
7      SparkSession spark = SparkSession
8        .builder()
9        .appName("EmployeeExample")
10       .master("local[4]")
11       .getOrCreate();
12
13     Dataset<Row> people = spark.read().json(getPathToDataFile());
14     //printSchema(people);
15
16     //Creates a temporary view using the DataFrame
17     people.createOrReplaceTempView("people");
```

5.2.2 Maschinelles Lernen

```
1      //Learning (accept only employees with Astrology interests)
2      Dataset<Row> training = spark.createDataFrame(Arrays.asList(
3        new JavaLabeledEmployeeDocument(0L, "Car Racing", 0.0),
4        new JavaLabeledEmployeeDocument(3L, "Astrology", 1.0),
5        new JavaLabeledEmployeeDocument(4L, "Bowling", 0.0)
6      ), JavaLabeledEmployeeDocument.class);
7
8      //Configure an ML pipeline, which consists of three stages: tokenizer,
           hashingTF, and lr.
9      Tokenizer tokenizer = new Tokenizer()
10       .setInputCol("interests")
11       .setOutputCol("words");
12
13     HashingTF hashingTF = new HashingTF()
14       .setNumFeatures(1000)
15       .setInputCol(tokenizer.getOutputCol())
16       .setOutputCol("features");
17
18     LogisticRegression lr = new LogisticRegression()
19       .setMaxIter(10)
20       .setRegParam(0.001);
21
22     Pipeline pipeline = new Pipeline()
23       .setStages(new PipelineStage[] {tokenizer, hashingTF, lr});
24
25     //Fit the pipeline to training documents.
26     PipelineModel model = pipeline.fit(training);
27
28     //show only hits
29     //showOnlyPredictedEmployees(model, people);
30
31     Dataset<Row> predictions = model.transform(people);
```

5.2.3 MapReduce und Ausgabe

```
1     JavaRDD<Row> predictedRowsRDD = predictions
2       .select("FirstName", "interests", "probability", "prediction", "Age")
3       .where("prediction=1.0").toJavaRDD();
4
5     System.out.println("Count of predictedRows: " +
6         predictedRowsRDD.collect().size());
7
8     //map row to age pairs
9     JavaPairRDD<Integer, Integer> ageMap = predictedRowsRDD.mapToPair(
10        new PairFunction<Row, Integer, Integer>() {
11      private static final long serialVersionUID = -4267715305026281029L;
12
13      @Override
14      public Tuple2<Integer, Integer> call(Row row) throws Exception {
15        long age = row.getLong(row.fieldIndex("Age"));
16        return new Tuple2<Integer, Integer>(Integer.parseInt(Long.toString(age)),
              1);
17      }
18    });
19    JavaPairRDD<Integer, Integer> ageMapCounts = ageMap.reduceByKey((i1, i2) ->
        i1 + i2);
20    boolean sortAscending = true;
21    JavaPairRDD<Integer, Integer> ageMapCountsSorted = ageMapCounts.sortByKey(
        sortAscending);
22
23    List<Tuple2<Integer, Integer>> output = ageMapCountsSorted.collect();
24    for (Tuple2<?, ?> tuple : output) {
25      System.out.println("Alter: " + tuple._1() + "--> Personen: " + tuple._2());
26    }
27
28    spark.stop();
29  }
```

5.2.4 Ergebnis

```
Count of predictedRows: 285
Alter: 20--> Personen: 1
Alter: 21--> Personen: 2
Alter: 22--> Personen: 3
Alter: 23--> Personen: 3
Alter: 24--> Personen: 4
Alter: 25--> Personen: 30
Alter: 26--> Personen: 35
Alter: 27--> Personen: 29
Alter: 28--> Personen: 35
Alter: 29--> Personen: 24
Alter: 30--> Personen: 33
Alter: 31--> Personen: 15
Alter: 32--> Personen: 9
Alter: 33--> Personen: 19
Alter: 34--> Personen: 15
Alter: 35--> Personen: 15
Alter: 36--> Personen: 1
Alter: 37--> Personen: 1
Alter: 39--> Personen: 2
Alter: 42--> Personen: 1
Alter: 43--> Personen: 1
Alter: 46--> Personen: 1
Alter: 49--> Personen: 1
Alter: 50--> Personen: 1
Alter: 51--> Personen: 1
Alter: 53--> Personen: 1
Alter: 61--> Personen: 1
Alter: 63--> Personen: 1
```

6 Literaturverzeichnis

Bücher

[Par15] Mahmoud Parsian. *Data Algorithms*. 1005 Gravenstein Highway North: O'Reilly Media, Inc., 2015.

[Ryz+15] Sandy Ryza, Uri Laserson, Sean Owen und Josh Wills. *Advanced Analytics with Spark*. 1005 Gravenstein Highway North: O'Reilly Media, Inc., 2015.

[ER16] Raul Estrada und Isaac Ruiz. *Big Data SMACK*. New York, 233 Spring Street: Springer Science + Business Media, 2016.

Forschungsberichte (Research Papers)

[Zah+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker und Ion Stoica. *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*. Forschungsspapier. University of California, Berkeley, 2012.

[Arm+15a] Michael Armbrust, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin und Matei Zaharia. *Scaling Spark in the Real World: Performance and Usability*. Forschungsspapier. Databricks Inc.; MIT CSAIL, 2015.

[Arm+15b] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi und Matei Zaharia. *Spark SQL: Relational Data Processing in Spark*. Forschungsspapier. Databricks Inc.; MIT CSAIL; AMPLab, UC Berkeley, Juni 2015.

[Ven+15] Shivaram Venkataraman, Zongheng Yang, Davies Liu, Eric Liang, Hossein Falaki Xiangrui Meng, Reynold Xin, Ali Ghodsi, Michael Franklin, Ion Stoica und Matei Zaharia. *SparkR: Scaling R Programs with Spark*. Forschungsspapier 11. AMPLab UC Berkeley, Databricks Inc., MIT CSAIL, Nov. 2015.

[Zah+15] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker und Ion Stoica. *Apache Spark: A Unified Engine for Big Data Processing*. Forschungsspapier 11. Communications Of The Acm, Nov. 2015.

Internet

[Dat17a] Inc. Databricks. *Apache Spark Key Terms, Explained - The Databricks Blog*. Mai 2017. URL: `https://databricks.com/blog/2016/06/22/apache-spark-key-terms-explained.html`.

[Dat17b] Inc. Databricks. *Technical Preview of Apache Spark 2.0 Now on Databricks*. Apr. 2017. URL: https://databricks.com/blog/2016/05/11/apache-spark-2-0-technical-preview-easier-faster-and-smarter.html.

[Fou17a] Apache Software Foundation. *Apache Spark Community*. Apr. 2017. URL: http://spark.apache.org/community.html.

[Fou17b] Apache Software Foundation. *Apache Spark Ecosystem*. Apr. 2017. URL: https://databricks.com/spark/about.

[Fou17c] Apache Software Foundation. *Apache Spark Ecosystem*. Apr. 2017. URL: https://www.infoq.com/articles/apache-spark-streaming.

[Fou17d] Apache Software Foundation. *Cluster Mode Overview*. Apr. 2017. URL: https://spark.apache.org/docs/1.1.0/cluster-overview.html.

[Fou17e] Apache Software Foundation. *GraphX - Spark 2.1.0 Documentation*. Apr. 2017. URL: http://spark.apache.org/docs/latest/graphx-programming-guide.html.

[Fou17f] Apache Software Foundation. *ML Pipelines - Spark 2.1.0 Documentation*. Apr. 2017. URL: http://spark.apache.org/docs/latest/ml-pipeline.html.

[Fou17g] Apache Software Foundation. *Research — Apache Spark*. Apr. 2017. URL: https://spark.apache.org/research.html.

[Git17a] Inc. GitHub. *apache/spark: Mirror of Apache Spark*. Apr. 2017. URL: https://github.com/apache/spark.

[Git17b] Inc. GitHub. *spark/JavaPipelineExample.java at master · apache/spark*. Apr. 2017. URL: https://github.com/apache/spark/blob/master/examples/src/main/java/org/apache/spark/examples/ml/JavaPipelineExample.java.

[Sch17] Julia Schmidt. *Big Data: Umfrage zur Verbreitung zu Apache Spark*. Mai 2017. URL: https://www.heise.de/developer/meldung/Big-Data-Umfrage-zur-Verbreitung-zu-Apache-Spark-2529126.html.

[Win17] WinFuture.de. *Twitter-Nutzer generieren täglich 12 Terabyte - WinFuture.de*. Apr. 2017. URL: http://winfuture.de/news,58451.html.