Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Timo Lange

Big Data Systeme: Konzeptioneller und experimenteller Vergleich von Apache Flink mit Apache Spark anhand eines Anwendungsszenarios

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Timo Lange

Big Data Systeme: Konzeptioneller und experimenteller Vergleich von Apache Flink mit Apache Spark anhand eines Anwendungsszenarios

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. Olaf Zukunft
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 16. Dezember 2016

Timo Lange

Thema der Arbeit

Big Data Systeme: Konzeptioneller und experimenteller Vergleich von Apache Flink mit Apache Spark anhand eines Anwendungsszenarios

Stichworte

Apache, Flink, Spark, Big Data, Alternating Least Squares, ALS, Skalierbarkeit

Kurzzusammenfassung

Big Data Systeme stellen mittlerweile einen der relevantesten Bereiche der Datenverarbeitung dar. Eines der momentan populärsten Systeme in diesem Bereich ist Apache Spark. Ein weiteres System mit zunehmende Popularität ist Apache Flink. Beide Systeme sollen zunächst auf Konzeptioneller und Architektonischer Ebene verglichen werden. Anschließend werden beide Systeme für die prototypische Umsetzung einer User-Story herangezogen, gefolgt von einer Evaluierung der Systeme mittels der Goal-Question-Metric Methode. Hierbei zeigt sich, dass Spark das reifere System ist, beide Systeme aber nicht geeignet sind die User-Story mit geringem Aufwand nach den zuvor gestellten Anforderungen umzusetzen.

Timo Lange

Title of the paper

Scalable Big Data analysis: Conceptual and practical comparison of Apache Flink with Apache Spark based on an application scenario

Keywords

Apache, Flink, Spark, Big Data, Alternating Least Squares, ALS, scalability

Abstract

Today, Big Data systems represent one of the most relevant topics in dataprocessing. One of the most popular systems in this category is Apache Spark. Another rising System of this category is Apache Flink. Within the scope of this work, lays the comparison of these Systems. Firstly, these Systems are being compared on a conceptual and an architectural level. Followed up by the design and implementation of a prototype of a User-Story in the context of an application scenario. The next step is the evaluation of Flink and Spark on the foundation of the implemented prototype. This happens by the use of the Goal-Question-Metric method. It shows that Spark is the more mature system but both systems are not capable to implement the User-Story according to the requirements with a small investment.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

1.1 Motivation

Im Big Data Bereich ist die Menge der zu verarbeitenden Daten so groß, dass diese nicht mehr auf einem einzelnen Rechner verarbeitet werden können. Auch traditionelle Datenbanksysteme stoßen in diesem Bereich an ihre Grenzen. Um dieser Flut an Daten begegnen zu können sind Systeme notwendig, die effizient und verteilt in großen Clustern arbeiten können. Neben funktionalen Anforderungen an ein solches System, ist eine hohe horizontale Skalierbarkeit eines der wichtigsten Eigenschaften. Auf dem Markt befindet sich eine Vielzahl solcher Systeme, was es schwer macht das geeignetste für den eigenen Anwendungsfall auszuwählen. Eines der aktuell populärsten Big Data Frameworks ist Apache Spark, ein weiteres aufstrebendes Framework ist Apache Flink. Beide Systeme verfolgen unterschiedliche Ansätze, stehen aber in direkter Konkurrenz.

Flink/Spark Architektur und Konzepte analysieren Eine gute Übersicht und Beschreibung der Systeme Anwendungsszenario entwickeln Eine Einschätzung über die Fähigkeiten, Vor- und Nachteile, sowie bevorzugten Einsatzgebiete von Flink und Spark

Marz and Warren [2015] Big DataPlattner [2016-11-22]

1.2 Zielsetzung

Ziel dieser Arbeit ist es Apache Flink und Apache Spark zu vergleichen. Hierfür sollen zunächst die Architektur und die verwendeten Konzepte von Flink und Spark erläutert und vergleichen werden. Ein praktischer Vergleich soll anhand eines Anwendungsszenarios geschehen. Im Rahmen des Anwendungsszenarios werden User-Storys entwickelt, von denen eine für eine prototypische Umsetzung sowohl mit Flink als auch Spark herangezogen wird. Hauptteil der Arbeit ist die Entwicklung, der Entwurf und die Umsetzung der User-Story, sowie die anschließende Evaluierung von Flink und Spark. Die Evaluierung von Flink und Spark erfolgt unter anderem auf Grundlage des entwickelten Prototypen. Für die Evaluierung werden zunächst Kriterien definiert, an denen sich die Untersuchung orientiert Die Auswertung soll Systematisch mit dem Goal-Question-Metric (GQM) Verfahren durchgeführt werden.

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich in 5 Kapitel, dessen Aufbau sich folgendermaßen darstellt;

Kapitel 1 Einleitung Die Einleitung verschafft einen Überblick der Arbeit und formuliert eine Problemstellung, welche die Relevanz des Themas herausstellt.

Kapitel 2 Anwendungssysteme In diesem Kapitel werden die Konzepte und die Architektur, die hinter Flink und Spark stehen erläutert.

Kapitel 3 Anwendungsszenario Im Anwendungsszenario wird ein Szenario dargestellt, welches die Einführung eines BigData-Verarbeitungssystems verlangt. Für dieses Szenario werden mehrere User-Storys entworfen, von denen eine ausgewählt und anschließend sowohl mit Flink als auch Spark umgesetzt wird.

Kapitel 4 Diskussion Im Kapitel Diskussion erfolgt eine Evaluierung von Spark und Flink. Diese bezieht sich unter anderem auf den im vorherigen Kapitel entworfenen Prototypen. Für die Evaluierung wird die Goal-Question-Metrik Methode angewandt.

Kapitel 5 Fazit In diesem Kapitel sollen die Erkenntnisse der voranstehenden Kapitel zusammengefasst werden sowie einen Ausblick auf mögliche weitere, dieser Arbeit folgende, Schritte geben.

2 Anwendungssysteme

2.1 Apache Flink

Apache Flink ist ein Framework für verteilte Stream- und Batch-Datenverarbeitung . Foundation [2016f] Es ist aus dem Forschungsprojekt Stratosphere hervorgegangen und seit Januar 2015 ein Apache Top Level Projekt. collaborative research project [2016], Foundation [2015] Im Kern basiert es auf einer Streaming-Dataflow-Engine, mit der sowohl Stream- als auch Batch-Verarbeitung realisiert werden, wobei die Batchverarbeitung als Spezialfall von Streams angesehen wird. Foundation [2016e,f] Flink ist in den Programmiersprachen Java und Scala geschrieben und bietet neben APIs für Java und Scala auch eine Python API. Die Schnittstellen unterteilen sich in die DataStream API für Streaming, welche nur für Java und Scala angeboten wird und die DataSet API für Batchverarbeitung, welche auch für Python bereit steht. Eine an SQL angelehnte „Table“ genannte API bietet die Möglichkeit der deklarativen Spezifikation von Abfragen und steht unter Java und Scala bereit. Foundation [2016f], Traub et al. [2015] Des Weiteren bietet Flink eine Reihe von spezialisierten Bibliotheken an, welche auf der genannten API aufbauen. Foundation [2016f] So bietet FlinkML eine Vielzahl von Machine Learning Algorithmen und ist an Sparks MLlib angelehnt. Vasiloudis [2015] Mit Gelly wird eine Bibliothek für Graphverarbeitung bereitgestellt und FlinkCEP bietet die Möglichkeit für die Verarbeitung von komplexen Events. Zudem ist Flink zu vielen Clustermanagement- und Speicherlösungen wie u.a. Apache Tez, Apache Kafka, Apache HDFS und Apache Hadoop YARN kompatibel. Foundation [2016e] Besonderheiten von Flink sind u.a.:

- Event Time und Out-of-order Events
- Exactly-once Semantik für zustandsbehaftete Berechnungen
- Streaming Fenster
- Continuous Streaming Model mit Backpressure
- Fehlertoleranz via leichtgewichtigen verteilten Schnappschüssen
- eigene Speicherverwaltung innerhalb der JVM

- Iterationen und Deltaiterationen
- Programm-Optimierer Foundation [2016e]

Im Nachfolgenden werden diese Besonderheiten näher beschrieben, gefolgt von einer Einführung in die Architektur von Flink.

2.1.1 Konzepte

2.1.1.1 Streaming Windows

Events aus Streams zu aggregieren funktioniert anders als bei der Stapelverarbeitung. Beispielsweise ist es nicht möglich alle Elemente eines Streams zu zählen und die Anzahl dann zurück zu geben, da Streams im allgemeinen endlos sind. Aus diesem Grund wird mit Hilfe von Windows ein Geltungsbereich (Scope) für die auszuführenden Operationen definiert. Typische Windows wären etwa alle Elemente der letzten 5 Minute zu zählen oder eine Summe über die letzten 100 Elemente zu bilden. In Flink gibt es zwei Gruppen von Windows, zeitgesteuerte(die letzten 5 Min.) und datengesteuerte (die letzten 100 Elemente). Abbildung 2.1 verdeutlicht wie ein Stream von Events in zeitgesteuerte (time windows) und in datengesteuerte (count(3) Windows) aufgeteilt werden kann. Foundation [2016c], Fabian Hüske [2015-12-04]

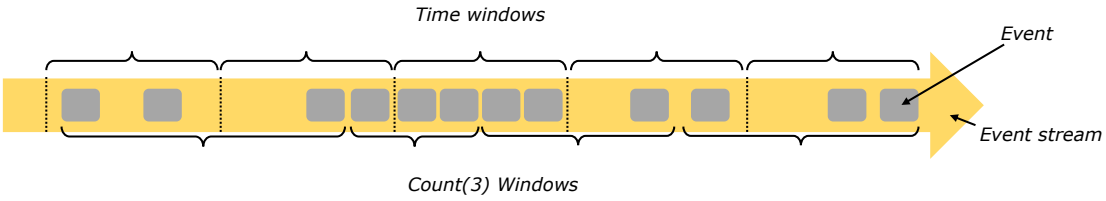

Abbildung 2.1: Time Windows und Count Windows Foundation [2016c]

Weiterhin bietet Flink eine Reihe weiter spezialisierte Windows und die Möglichkeit selbst definierte Windows zu erstellen The Apache Software Foundation [2016c]

2.1.1.2 Time & Out-of-order Events

Innerhalb eines Stream-Programms, um z.B. ein Window zu definieren, gibt es die Möglichkeit sich auf mehrere Arten von Zeit zu beziehen. Abbildung 2.2 gibt eine Übersicht über die Arten von Zeit die Flink bekannt sind.

Event Time ist der Zeitpunkt an dem ein Event erstellt wurde. Gewöhnlich wird dies durch einen timestemp am Event ausgedrückt. Auf diesen kann Flink über so genannte „timestamp assigners“ zugreifen.

Ingestion Time ist jener Zeitpunkt an dem ein Event am „source operator“ in Flinks dataflow eintritt.

Processing Time ist die lokale Zeit zu welcher ein Operator eine zeit basierte Operation ausführt. Foundation [2016c], Fabian Hüske [2015-12-04]

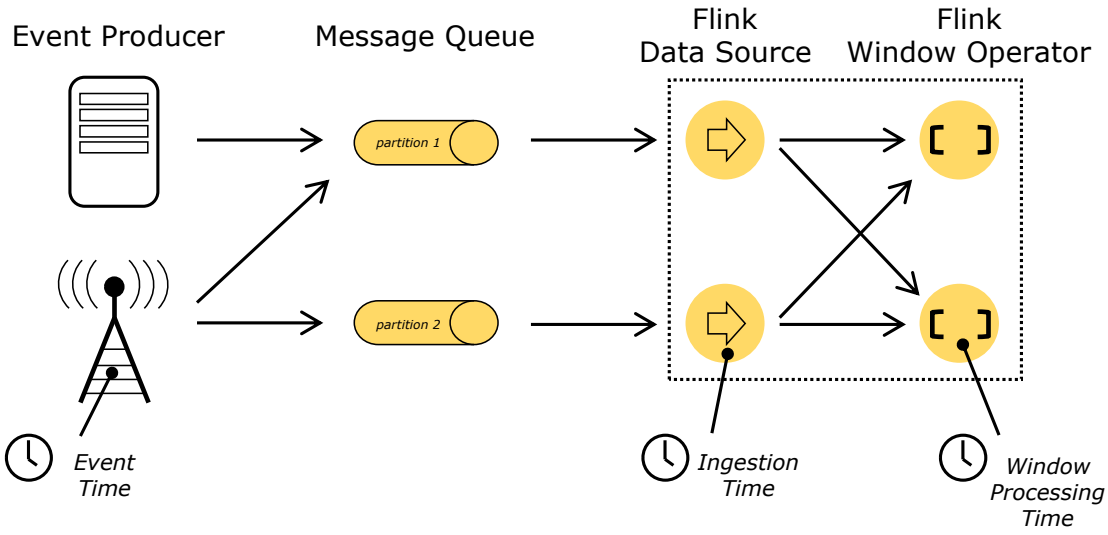

Abbildung 2.2: Finks verschiedene Arten von Zeit Foundation [2016c]

Event Time Wird ein Event Time Window verwendet, enthält es alle Records dessen event timestemps in dieses Zeitfenster fallen, unabhängig davon wann die Records eintreffen und in welcher Reihenfolge. Event time liefert immer korrekte Ergebnisse, selbst bei Out-of-order Events, spät eintreffenden Events oder bei Wiedereinspielen von Daten aus Backups bzw. persistenten Logs. Da hier auf späte und Out-of-order Events gewartet werden muss, unterliegt Event time einer gewissen Latenz. The Apache Software Foundation [2016d]

Watermarks & Out-of-order Events Bei Event time hängt der Fortschritt der Zeit von den Daten ab und nicht von realer Zeit. Deswegen wird eine Möglichkeit den Zeitlichen Fortschritt zu messen benötigt. Dies wird durch *Event Time Watermarks* erreicht, die den Fortschritt der Zeit darstellen. *Watermarks* fließen als Teil des Datenstromes mit und haben einen *timestemp t*. Ein *Watermark(t)* gibt an, dass die Event time den Zeitpunkt t erreicht hat und bedeutet dass alle Events mit *timestemp t'* < *t* angekommen sind. Erreicht ein *Watermark* einen Operator, kann dieser seine interne Event time Uhr um eins erhöhen. Abbildung 2.3

zeigt einen Stream in dem alle Events in Reihenfolge eintreffen und *Watermarks* einfache periodische Marker darstellen.

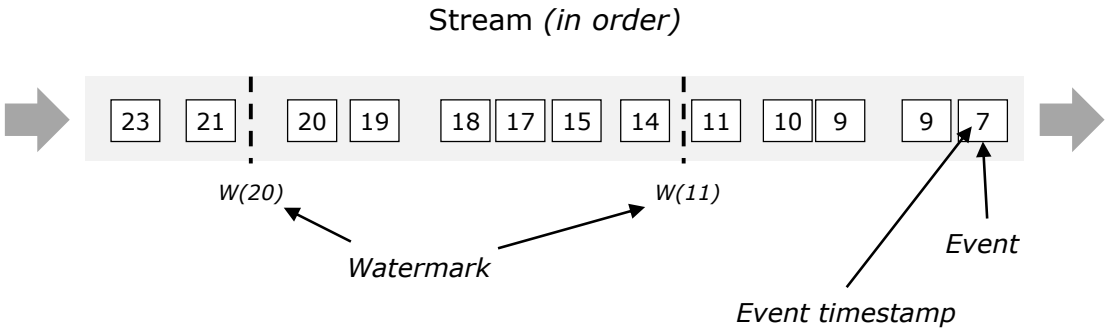

Abbildung 2.3: In-order Stream von Events mit logischen *timestemps* und *Watermarks* The Apache Software Foundation [2016d]

Abbildung 2.4 zeigt einen Stream in dem Events außerhalb der Reihenfolge eintreffen wobei die *Watermarks* einen Zeitpunkt im Stream darstellen ab dem alle Events bis zu diesem Zeitpunkt eingetroffen sind.

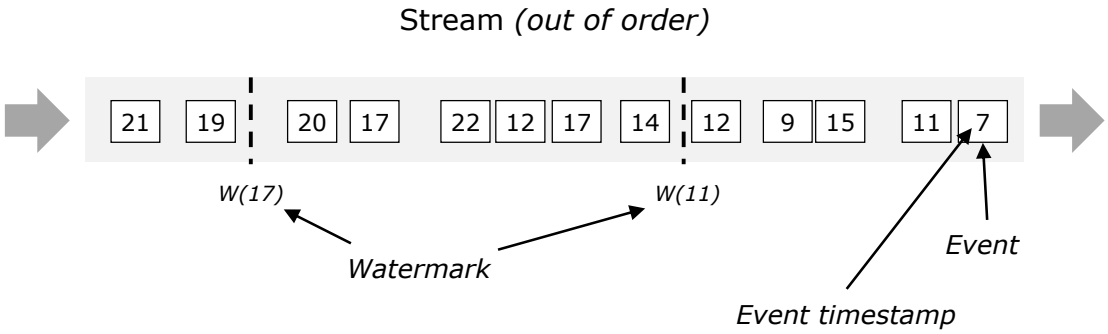

Abbildung 2.4: Out-of-order Stream von Events mit logischen *timestemps* und *Watermarks* The Apache Software Foundation [2016d]

The Apache Software Foundation [2016d,p]

2.1.1.3 State & Fault Tolerance

State Neben Operatoren die jedes Event individuell betrachten, gibt es auch Operatoren die Informationen über mehrere Events hinweg speichern. Dies sind so genannte stateful Operatoren. Dieser State wird in einer Art eingebettetem Key/Value-Store, dem State Backend, festgehalten. The Apache Software Foundation [2016f] Dabei kann dieser, je nach Konfiguration, eine in-memory Hash-Map im JobManager sein, im Dateisystem z.b. HDFS oder wird durch

den key/value Store RocksDBFacebook [2016] realisiert. Der State wird immer zusammen mit den Streams partitioniert und verteilt, die von den stateful Operatoren gelesen werden.

Fault Tolerance durch Checkpoints Flinks Fault Tolerance wird durch eine Kombination aus „stream replay“ und Checkpoints erreicht. Ein Checkpoint definiert einen konsistenten Punkt in den Streams und dem zugehörigen State von dem aus ein „stream dataflow“ seine Arbeit wieder aufnehmen kann. Im Falle eines Fehlers wird der verteilte streaming-dataflow gestoppt, die Operatoren werden gestoppt und auf den Stand des letzten erfolgreichen Checkpoints zurückgesetzt. Die input-streams werden ebenfalls auf den gleichen Stand wie den des State-Snapshot zurückgesetzt. Damit die Streams erneut eingespielt werden können, benötigt Fink eine Stream-Quelle, die in der Lage ist Streams zu einem bestimmten Punkt zurück zu setzen. Ein Beispiel hierfür ist Apache Kafka. Da Flinks Checkpoints mittels verteilter Snapshots realisiert werden, werden Snapshots und Checkpoints hier synonym verwendet. Der zentrale Teil von Flinks Fault Tolerance Mechanismus basiert auf kontinuierlichen konsistenten Snapshots des verteilten Datenstromes und den Operator-States und ist eine Abwandlung vom Chandy-Lamport AlgorithmusChandy and Lamport [1985] für verteilte Schnappschüsse und wird von Carbone et al.Carbone et al. [2015] genauer beschrieben. Durch das verwendete Verfahren kann Flink standardmäßig eine „exactly once“ Semantik garantieren, wobei es auch möglich ist, eine „at least once“ Semantik zu wählen, um sehr geringe Latenzen zu erreichen wobei dann doppelte Records im Fehlerfall in kauf zu nehmen sind. Foundation [2016c], The Apache Software Foundation [2016b,e], Tzoumas et al. [2015] Mit Savepoints bietet Flink Möglichkeit manuell einen Checkpoint zu veranlassen, welcher permanent gespeichert und nicht automatisch von Flink gelöscht wird, wenn dieser nicht mehr benötigt wird. Auf diese Weise ist es mit Flink Möglich, die Ausführung eines Programms anzuhalten und zu einem späteren Zeitpunkt, auch in einem anderen Cluster, wieder fort zu setzen.The Apache Software Foundation [2016a]

2.1.1.4 Speicherverwaltung

Apache Flink verfügt über eine eigene Speicherverwaltung die innerhalb der JVM den Speicher verwaltet. Zudem existiert auch eine Implementierung für eine s.g. Off-Heap Speicherverwaltung bei der auf den Arbeitsspeicher außerhalb der JVM zugegriffen wird. Bei Erstellung dieser Arbeit galt die Selbstverwaltung das Speichers nur für die Verwendung der Batch-API und die Streaming-API baut auf einem anderen Konzept auf. Einen Zweck den die Speicherverwaltung von Flink erfüllen soll, ist den Speicherverbrauch einzelner Operatoren zu kontrollieren. Typische Operatoren sind etwa:

- Sortieren
- Hash-Tabellen
- Caching
- (Block-)Nested-Loop-Join

Ohne die interne Speicherverwaltung würden diese Operatoren einen OutOfMemory-Fehler produzieren wenn der Speicherverbrauch den verfügbaren Speicher der JVM übersteigt. Zudem ermöglicht die interne Speicherverwaltung Flink effizient und übergangslos Out-of-Core Algorithmen auszuführen wenn der benötigte Speicher einer Operation den zur Verfügung stehenden Arbeitsspeicher (Heap) überschreitet. So ist es Flink möglich Datenmengen zu verarbeiten die den verfügbaren Arbeitsspeicher im gesamten Cluster weit überschreiten. Weiterhin erlaubt die selbstständige Verwaltung den Speicher zwischen die Operatoren so aufzuteilen, dass diese gleichzeitig in einer JVM laufen können, ohne sich gegenseitig zu behindern. Konzeptionell wird der Speicher in drei Bereiche eingeteilt, illustriert in Abbildung 2.5

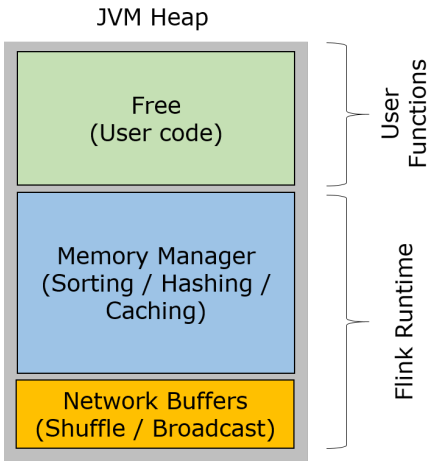

Abbildung 2.5: Speicheraufteilung von Flink Stephan Ewen and Henry Saputra [2015-05-20]

Die Einteilung gliedert sich wie folgt:

Netzwerk Puffer Standardmäßig 2048 Puffer mit einer Größe von 32 KiByte die vom Netzwerk-Stack genutzt werden um Records für den Netzwerkverkehr zu puffern. Die Allokierung erfolgt beim Start des TaskManagers.

Memory Manager Pool Eine große Anzahl an 32 KiByte Puffern, die von allen Laufzeit-Algorithmen genutzt werden um Records zu puffern. Die Records werden in serialisierter Form in diesen Speicherblöcken gespeichert. Die Allokierung erfolgt durch den Memory Manager beim Start.

Freier Speicher Dieser Bereich ist für den User-Code und die Datenstrukturen des TaskManagers vorgesehen. Da diese Datenstrukturen nicht viel Speicher verbrauchen, ist dieser größtenteils für den User-Code verfügbar.

Während der Allokation des Netzwerk Puffer sowie des Memory Manager Puffer führt die JVM meist mehrere volle Garbage Collections durch. Dies erhöht die Startzeit des TaskManagers, allerdings existieren die Puffer über die gesamte Laufzeit des TaskManagers und wandern in die "tenured generation"der JVM. Dies ist ein langlebiger Speicherbereich der JVM über den die Garbage Collection selten läuft, womit während der Ausführung von Tasks Zeit für die Garbage Collection eingespart werden kann.

Memory Segments Die Repräsentation des Speichers ist in Fink über eine Collection von *Memory Segments* realisiert. Die Segmente repräsentieren dabei eine Region im Speicher und stellen Methoden zum Zugriff auf die Daten über einen Offset bereit. *Memory Segments* ähneln dem *java.nio.ByteBuffer*, sind aber eine vollständig eigene auf Flink zugeschnittene Implementierung.

Typen Serialisierung Jedes mal wenn Flink ein Record speichert, wird dieses in ein oder mehrere *Memory Segments* serialisiert. „Pointer“zu diesem Record werden in einer anderen Datenstruktur gespeichert. Somit ist Flink von einer effizienten Serialisierung abhängig, die Kenntnis von „Seiten“(pages) hat und Redords über mehrere Seiten aufteilen kann. Die Aufteilung der Records ist in Abbildung 2.6 dargestellt.

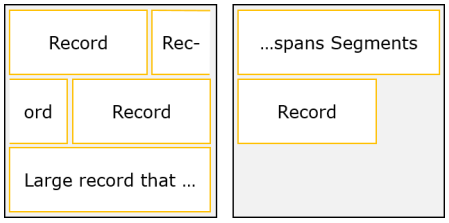

Abbildung 2.6: Mehrere Records, serialisiert in *Memory Segments* Stephan Ewen and Henry Saputra [2015-05-20]

Das Serialisierungsformat wird von Flinks Serialisierern bestimmt und ist abhängig von den einzelnen Feldern des Records. Um die Leistung zu verbessern ist es auch möglich Redocrds für die Verarbeitung nur Teilweise zu deserialisieren. Für eine weitere Leistungssteigerung versuchen die Algorithmen die mit *Memory Segments* arbeiten möglichst nur auf den serialisierten Daten zu arbeiten. Ein Beispiel dafür ist etwa, dass die meisten Vergleiche die der Sorter durchführt runter gebrochen auf den Vergleich von Bytes verschiedener Seiten (pages) ist. Dies ist vergleichbar mit einem *memcmp* und sehr effizient. Die *Memory Segments* werden vom TaskManager in einem Speicherpool vorgehalten und an die Algorithmen die Speicher anfordern freigegeben. Werden die Segmente nicht mehr benötigt, wandern diese wieder zurück in den Pool. Aus diesem Grund müssen die Algorithmen den Speicher auch explizit anfordern und wieder freigeben (ähnlich wie malloc). Jeder Algorithmus hat eine ein vorgegebenes Speicher-Budget. Ist dieses aufgebraucht muss auf eine out-of-core variante dieses

Algorithmus zurückgegriffen werden. Da sich die Daten bereits in serialisierten Seiten befinden, ist es sehr einfach diese zwischen Arbeitsspeicher und Festplatte zu verschieben.

Off-heap Memory Neben der Verwaltung des Speichers im Heap innerhalb der JVM bietet Flink die Möglichkeit den Speicher auch außerhalb der JVM direkt zu Nutzen. Dies hat zum einen den Vorteil, dass der Garbage Collector nun gar nichts mehr mit den Daten zu tun hat. Zum Anderen kann nun beliebig viel Speicher für die Daten genutzt werden, ohne dass die JVM selbst sehr groß wird.Stephan Ewen and Henry Saputra [2015-05-20] Dies ist von Vorteil, da sehr große JVMs, z.B. um die 100GB, oft problematisch sind und instabil laufen. Fabian Hüske [2015-05-13] Fabian Hüske [2015-05-11]

2.1.1.5 Iterationen

Iterationen in Flink sind direkt in den Dataflow eingebettet und werden über spezielle Operatoren realisiert. Diesen Operatoren wird eine „Step-Function“ mitgegeben, welche bei jeder Iteration ausgeführt wird, sowie eine Kondition zur Terminierung. Als Terminierungsbedingung wird die maximale Anzahl an Iterationen angegeben oder eine selbst definierte Terminierungskondition gegen die das partielle Ergebnis konvergiert. Dadurch, dass die Iterationen direkter Bestandteil des Dataflows sind und nicht durch eine Schleife im User-Code umgesetzt werden, können die Operatoren in jeder Iteration wieder verwendet werden und es wird der Overhead durch den Transfer des Ergebnisses jeder Iteration zum ausführenden Programm und zurück für die nächste Iteration eingespart. Flink verfügt über zwei Arten von Iterationen, die Bulk-Iterationen und die Delta-Iterationen.

Bulk-Iterationen Bulk-Iterationen produzieren während jeder Iterationen ein komplettes partielle Ergebnis, welches als Eingabe an die nächste Iteration weitergeleitet wird. Das Ergebnis der Bulk-Iteration ist das letzte partielle Ergebnis vor der Terminierung.

Delta-Iterationen Delta-Iterationen arbeiten mit einem so genannten „Solution-Set“ und einem „Working-Set“.

Delta-Iterationen weisen als weitere Kondition zur Terminierung ein leeres „Working-Set“ auf. Das Ergebnis der Bulk-Iteration ist das „Solution-Set“ nach der letzten Iteration. The Apache Software Foundation [2016g,h], Ewen et al. [2012], Adrian Colyer [2015-06-18]

2.1.1.6 Integrierte Bibliotheken

Neben den genannten Kernfunktionalitäten von Flink gibt es noch eine Reihe von Bibliotheken, welche weitere Funktionalitäten bereitstellen und ebenfalls Bestandteil vom Flinkprojekt sind. Folgender Abschnitt gibt eine Übersicht der Bibliotheken:

FlinkML - Maschinelles Lernen FlinkML ist Flinks Ansatz für skalierbares maschine lernen und ist auf einfache Benutzung und Minimierung von GlueCode ausgelegt. Es werden eine Reihe fertiger Algorithmen geboten, sowie pipelines, welche es ermöglichen einfach und typsicher verschiedene *transformers* und *predictors* zu verketten. FlinkML wurde von den maschine learning libraries Spark MLLib sowie scikit-learn inspiriert. FlinkML ist speziell dafür entworfen, Flinks in-memory data streams und nativ ausgeführte Iterationen auszunutzen. Theodore Vasiloudis [2016-07-27], The Apache Software Foundation [2016i,j]

Gelly - Graph Verarbeitung Gelly ist Flinks API für Graph Verarbeitung und enthält Methoden und Werkzeuge um die Graph-Analyse mit Flink zu vereinfachen. Mit Gelly können gerichtete, sowie ungerichtete Graphen mit Hilfe von High-level Funktionen ähnlich der Batch-API erstellt, transformiert und modifiziert werden. Darüber hinaus bietet Gelly eine Reihe fertiger Graph-Algorithmen, sowie spezielle Methoden die auf iterative Verarbeitung eines Graphen ausgelegt sind. Graphen werden in Gelly durch ein *DataSet* von Knoten (vertices) und einem *DataSet* von Kanten (edges) repräsentiert, wobei sowohl Knoten als auch Kanten ein beliebiger wert zugewiesen werden kann. The Apache Software Foundation [2016k,l,m]

Table - Relationale Datenverarbeitung Die Table-API ist noch ein experimentelles Feature von Flink und eine SQL ähnliche Ausdruckssprache(expression language) mit relationalen Operatoren wie *selection*, *aggregation* und *joins* für relationale Stream- und Batchverarbeitung. Tables können ebenfalls mit regulärem SQL abgefragt werden, wobei SQL und die Table-API die gleiche Funktionalität bieten und innerhalb eines Programms auch gemixt werden können. Die Table-API und das SQL-Interface arbeiten auf einer relationalen Tabellen(*Table*) Abstraktion die aus externen Quellen oder aus existierenden *DataSets* oder *DataStreams* erzeugt werden kann. *Tables* können ebenfalls wieder zurück in *DataSets* bzw. *DataStreams* transformiert werden. Dabei wird der logische Plan, der durch relationale Operatoren oder SQL Querys erzeugt wurde, durch *Apache Calcite* The Apache Software Foundation [2016n] optimiert, bevor dieser in *DataSets* bzw. *DataStreams* transformiert wird. Zudem ist es möglich Tables bei dem s.g. *TableEnvironment* zu registrieren, wobei registrierte *Tables* die aus Table-API Operatoren oder SQL-Querys entstanden sind wie eine *View*, bekannt aus RDMS, behandelt werden und kann

so etwa *inlined* werden wenn die Query optimiert wird. The Apache Software Foundation [2016o]

FlinkCEP - Complex Event Processing FlinkCEP ist Flinks Library um komplexe Event Muster innerhalb eines Streams zu erkennen. Komplexe Events können aus matching sequences erstellt werden. Jedes Pattern besteht aus mehreren Stages bzw. States, wobei der Übergang von einem State in den nächsten durch mehrere *Conditions* bestimmt werde kann. Diese *Conditions* können die Kontiguität (zeitlicher Zusammenfall, Angrenzung, Nachbarschaft) von Events sein oder eine Filter Condition auf Events.

2.1.2 Architektur

2.1.2.1 Komponenten-Stack

Abbildung 2.7 stellt eine Übersicht über die Schichtenarchitektur mit den verschiedenen Komponenten von Apache Flink dar. Die unterschiedlichen Schichten bauen jeweils aufeinander auf und erhöhen mit jeder Schicht den Abstraktionslevel. Die oberste Ebene des Komponenten-Stack bilden die DataStream API und die DataSet API. Auf diesen APIs setzen auch die von Flink bereitgestellten Bibliotheken auf. Eine Ebene darunter befinden sich der Stream Builder und der Optimizer. Diese konsumieren die mit der API erstellten Programme und erstellen daraus einen optimalen Ausführungsplan. Wobei der Stream Builder den Plan für die DataStream API erstellt und der Optimizer den Plan für die DataSet API. Der Ausführungsplan wird anschließend in der Laufzeitumgebung (Runtime) in der darunterliegenden Schicht ausgeführt, wobei die Ausführung lokal oder im Cluster erfolgen kann. Das Cluster Management kann entweder direkt durch Flink erfolgen oder mittels Ressource Management Plattformen wie Apache Hadoop YARN. Die unterste Ebene bildet die Speicherschicht. Um Daten zu erhalten bzw. zu speichern, bietet Flink eine Vielzahl an Adaptern zu verschiedenen Speichersystemen wie HDFS, SQL Datenbanken, Apache Kafka sowie vielen weiteren. Traub et al. [2015], Foundation [2016a]

2.1.2.2 Datenstrukturen

Extern(Public API) Die grundlegende Datenabstraktion von Flink repräsentieren DataSets und DataStreams. Diese sind verteilte immutable Collections, die Duplikate enthalten können und auf denen Transformationen ausgeführt werden. Jede Transformation liefert wiederum ein neues DataSet bzw. einen DataStream zurück. Jede Collection wird aus einer Quelle, wie etwa einer Datei, Kafka oder einer lokalen Collection, erzeugt. Ergebnisse werden mittels einer

Datensenke, die u.a. eine Datei oder der Standard Output sein können, zurückgegeben. Dabei sind DataSets endlich während DataStreams eine unbegrenzte Anzahl an Elementen enthalten können. Zudem ist es nicht möglich direkt auf einzelne Elemente zuzugreifen. Alle Flink-Programme werden lazy ausgeführt, d.h. das Laden der Daten und die Transformationen darauf erfolgen nicht sofort, stattdessen wird jeder Task zu dem Ausführungsplan des Programms hinzugefügt. Erst wenn das Programm explizit zur Ausführung angestoßen wird, werden die Aufgaben, wie Einlesen der Daten und die Transformationen darauf, auch ausgeführt. Foundation [2016b] Der Ausführungsplan eines Flink-Programms gleicht dabei einem gerichteten azyklischen Graphen(engl. DAG) aus parallelisierbaren Operatoren. Weiterhin können die Programme auch Iterationen beinhalten. Diese sind aber nicht direkt im Graphen abgebildet, sondern durch spezielle Operatoren realisiert. Traub et al. [2015], Foundation [2016c], Ewen et al. [2012]

Intern Aus dem durch die API erstellten Ausführungsplan in Form eines Operator-DAGs wird für die Ausführung in der Laufzeitumgebung ein so genannter JobGraph generiert. Der JobGraph ist ein generisches, paralleles Datenfluss Programm, bestehend aus Operatoren (JobVertex) und Zwischenergebnissen (IntermediateDataSet). Dies ist in Abbildung 2.8 im linken Bildabschnitt zu sehen. Weiterhin enthält der JobGraph eine Ansammlung von Bibliotheken, die benötigt werden, um den Code der Operatoren auszuführen. Jeder Operator besitzt Eigenschaften (properties) wie etwa die Parallelität mit welcher der Operator ausgeführt wird, sowie den auszuführenden Code. Foundation [2016a,d] Der JobGraph und dessen Operatoren basieren auf dem sog. PACT(Parallelization Contract) Programmiermodell, welches eine Genaralisierung des MapReduce-Modells darstellt und aus dem Vorgängerprojekt Stratosphere stammt. Alexandrov et al. [2014-12], Battré et al. [2010], Babu [2012b], collaborative research project [2014b] Sowohl PACT als auch die Nephele Data-Flow-Engine, die in Stratosphere benutzt wurde und ein eigenständiges Projekt darstellte, Battré et al. [2010], collaborative research project [2014a], Warneke and Kao [2009], Babu [2012a] wurden in Flink zu einem gemeinsamen System verschmolzen, woraus sich die aktuellen Datenstrukturen ergeben. Ewen [2015], Saputra [2015]

Der ExecutionGraph ist eine parallele Version des JobGraph, zu sehen in Abbildung 2. Er enthält für jeden JobVertex im JobGraph einen ExecutionVertex pro parallelem Subtask. Der ExecutionVertex verfolgt den Status der Ausführung eines bestimmten Subtasks, wobei ein Task mehrere Phasen durchläuft. Dabei kann ein Task auch mehrmals ausgeführt werden, z.B. im Falle einer Failure Recovery. Diese Ausführung wird in Form einer Execution verfolgt. Jedes ExecutionVertex hat eine Current Execution und ggf. Prior Executions. Die verschiedenen

Zustände einer Execution und die Transitionen zwischen diesen sind in Form eines Zustandsautomaten in Abbildung 2.9 illustriert. Alle ExecutionVertex eines JobVertex gehören zu einem ExecutionJobVertex. Der ExecutionJobVertex verfolgt den Status eines Operators insgesamt. Weiterhin enthält der ExecutionGraph das IntermediateResult, welches den Status des IntermediateDataSet als Ganzes verfolgt. Den Status jeder Partition des IntermediateDataSet wird durch IntermediateResultPratitions verfolgt. Foundation [2016a,d]

2.1.2.3 Scheduling & Deployment

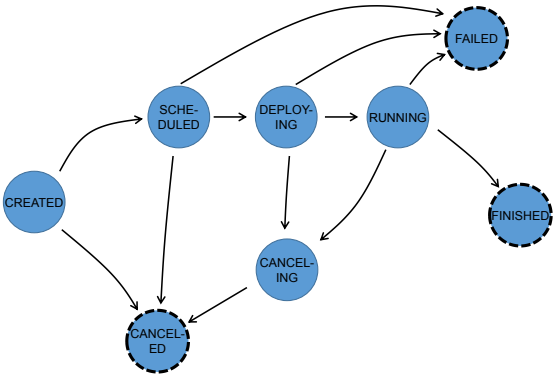

Abbildung 2.9: Zustandsautomat einer Execution Foundation [2016d]

In Abbildung 2.10 sind die Komponenten dargestellt, welche beim Start und während der Laufzeit eines Flink-Systems ausgeführt werden. Die Interaktionen der Komponenten sind durch Pfeile veranschaulicht. Die Koordination zwischen unterschiedlichen Prozessen wird dabei durch das Aktor-System von Scala realisiert. Wenn ein Flink Programm ausgeführt wird, wird ein Client gestartet. Innerhalb des Clients wird durch den Optimizer bzw. Stream Builder, der durch die API erstellte Ausführungsplan in einen JobGraph umwandelt. Foundation [2016a,d,c] Bei dem Übersetzungsschritt erfolgt ebenfalls eine automatische Optimierung des Datenflussprogramms, welche für alle APIs durchgeführt wird. Tzoumas [2014] Bei der Optimierung wird u.a. der geeignetste Algorithmus für die gewählte Operation ausgesucht. Der Optimierer arbeitet kostenbasiert und ähnelt den Optimierern die in RDBMS eingesetzt werden. Traub et al. [2015] Nachdem der JobGraph erstellt wurde, wird dieser vom Client an den JobManager gesendet. Der Client ist nicht Teil der Laufzeitumgebung und erhält nach dem Senden des JobGraph lediglich das Ergebnis sowie optionale Status-Updates. Die Laufzeitumgebung von Flink besteht aus zwei Prozessen, dem JobManager (Master) sowie dem TaskManager (Worker) und kann sowohl lokal als auch im Cluster ausgeführt werden. Der JobManager koordiniert die verteilte Ausführung. Er plant die Aufgaben, koordiniert Checkpoints und ist u.a. für Recovery bei Fehlern zuständig. In einem Flink-Setup gibt es mindestens einen JobManager. Bei hochverfügbaren Systemkonfigurationen laufen mehrere JobManager, von denen einer der Leader ist während sich die anderen im Standby befinden. Des Weiteren befindet sich in einem Setup mindestens ein TaskManager. Dieser führt die Tasks bzw. Subtasks eines Dataflows aus, welche

ihm vom JobManager zugeteilt werden. Zudem ist er für das Buffering und den Austausch der DataStreams mit anderen TaskManagern zuständig. Jeder TaskManager hat ein bis mehrere TaskSlots, welche die Ausführungs-Ressourcen von Flink darstellen.

In einem TaskSlot kann eine Pipeline von parallelen Tasks abgearbeitet werden. Dabei besteht diese Pipeline aus aufeinanderfolgenden Subtasks, wobei Flink diese häufig parallel ausführt. In Abbildung 2.11 wird dies verdeutlicht. Als Beispiel dient ein Programm, mit einer Datenquelle der Parallelität 4, einer Map-Funktion der Parallelität 4 und einer Reduce-Funktion der Parallelität 3. Im unteren linken Bildabschnitt ist das Programm als Graph dargestellt, aufgeteilt auf vier Pipelines mit jeweils einer Farbe. Die unterste Ebene stellt die Datenquelle mit vier Subtasks, die zweite die Map-Funktion mit vier Subtasks und die oberste Ebene die Reduce-Funktion mit drei Subtasks dar. Die Abbildung zeigt wie dieses Programm auf zwei TaskManager mit jeweils drei Pipelines aufgeteilt wird. Foundation [2016a,d,c]

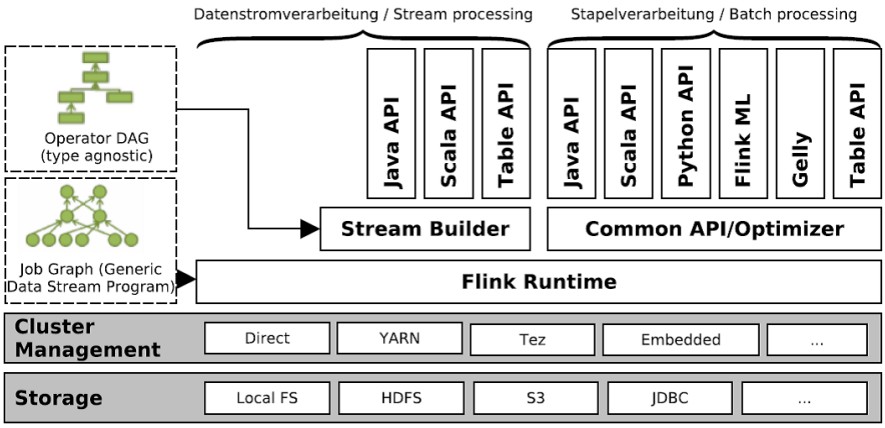

Abbildung 2.7: Architektur und Komponentenübersicht der Apache Flink Plattform Traub et al. [2015]

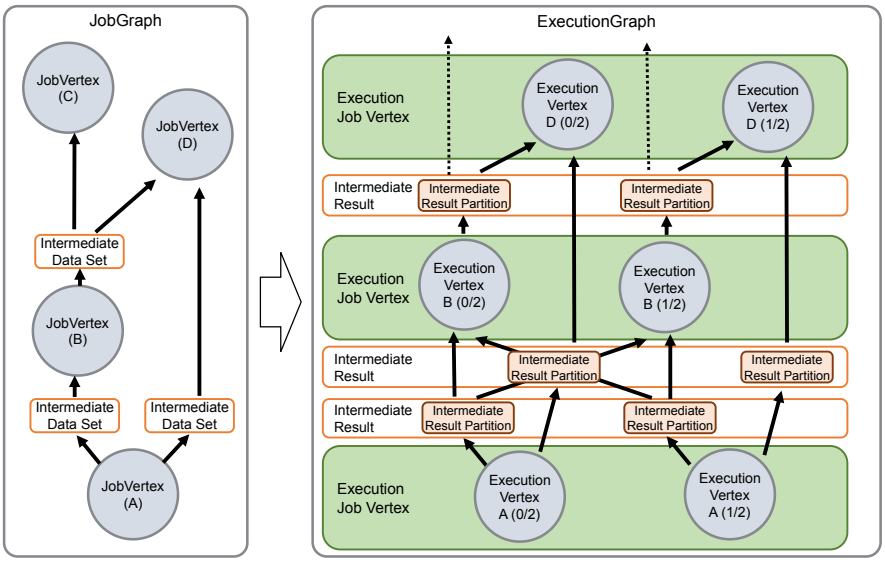

Abbildung 2.8: JobGraph und ExecutionGraph Foundation [2016d]

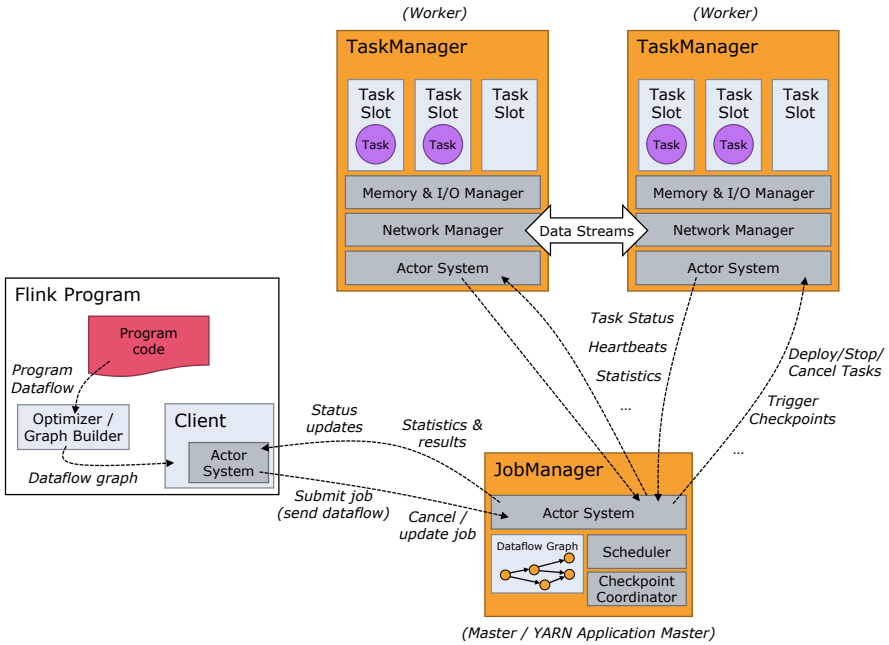

Abbildung 2.10: Ausführung von Subtasks in TaskSlots des TaskManager Foundation [2016a,c]

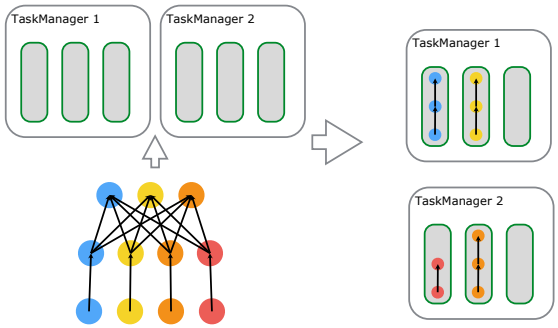

Abbildung 2.11: Ausführung von Subtasks in TaskSlots des TaskManager Foundation [2016d]

Intern werden durch die SlotSharingGroup und CoLocationGoup definiert, welche Tasks sich einen Slot teilen dürfen und welche sich zwingend in dem gleichen Slot befinden müssen. Da die Operatoren zu mehreren parallelen Instanzen segmentiert werden können und dann nebenläufig jeweils auf einem Teil der Datentupel arbeiten, ist eine Datenparallelität gegeben. Die Ergebnisse eines Operators werden an den nächsten Operator weitergeleitet, wodurch eine Pipelineparallelität entsteht. Traub et al. [2015]

2.2 Apache Spark

Apache Spark ist ebenso wie Apache Flink ein Framework für verteilte Stream- und Batch-Datenverarbeitung. Spark startete 2009 als Forschungsprojekt am AMPLab der UC Berkeley AMPLab [2016] und wurde 2010 als Open-Source Projekt veröffentlicht. 2013 wurde Spark an die Apache Software Foundation übergeben und wurde im Februar 2014 zum Top Level Projekt erklärt. The Apache Software Foundation [2016q,r, 2014-02-27] Genau wie Flink bietet auch Spark APIs für Scala, Java und Python und darüber hinaus R. Den Kern von Spark bilden Resilient Distributed Datasets (RDD), verteilte Collections die Sparks komplette Datenabstraktion darstellt und für Batch- als auch Streaming-Verarbeitung verwendet wird.

2.2.1 Konzepte

2.2.1.1 Fault tolerance

Spark bietet für seine verschiedenen Komponenten unterschiedliche Lösungen für „fault tolerance“ , welche auch unterschiedliche Fehlersemantiken bieten. Zu unterscheiden sind Lösungen für RDDs, Streaming und Accumulators.

Batch(RDD) Wenn ein Worker ausfällt wird die betroffene Partition des RDD mit Hilfe des *lineage Graph* neu berechnet. Entweder wird dies komplett von der Datenquelle bis hin zur ausgefallenen Partition berechnet oder die Neuberechnung kann beim Stand des letzten „persist()“ ansetzen und spart somit Rechenzeit. Zudem können die mit „persist()“ gespeicherten Daten über mehrere Rechner in Cluster hinweg repliziert werden.

Streaming(DStream) Um eine Fehlertoleranz bei Streams zu bieten nuzt Spark das „Checkpointing“. Dabei wird der State der Anwendung periodisch gespeichert. Für die Speicherung kann HDFS, Amazon S3 oder das lokale Dateisystem genutzt werden. Spark Streaming kann den State durch den linage graph wieder herstellen, wobei der Checkpoint bestimmt, wie weit die Neuberechnung zurückreicht. Spark kann beim Streaming „exactly-once“ Semantik

bieten, bei den output-Operationen ist es jedoch möglich, dass der Task für die Ausgabe an externe Systeme mehrmals ausgeführt wird. Um eine Fehlertoleranz zu erreichen bieten die Komponenten folgende

Driver Fault Tolerance muss im fehlerfall neugestartet werden. Der Standalone Cluster Manager bietet hier mit dem „–supervise“ flag dies automatisch zu übernehmen. Um den Spark Standalone master ebenfalls Fehlertolerant zu machen muss hierfür ZooKeeper konfiguriert werden. Zu erwähnen sein noch, dass wenn der Driver abstürzt, alle Executer ebenfalls neugestartet werden müssen.

Worker Node Fault Tolerance Hier nutzt Spark die selbe Technik wie bei RDDS, alle Daten aus externen Quellen werden über mehrere Worker repliziert. Im Fehlerfall werden die Daten von einer Replik mithilfe des linage graph erneut berechnet.

Receiver Fault Tolerance Wenn ein Reciver ausfällt, startet Spark diesen auf einem anderen worker node im Cluster neu. Die Fehlertoleranz hängt hier von der Datenquelle ab. Im Falle eines zuverlässigen Dateisystems merkt Spark sich die bereits verarbeiteten Daten in eine Checkpoint und fährt an der stelle fort wo es aufgehört hat. Beim „receiver-pull-from-sink“ Modell, entfernt Spark Elemente erst, wenn sie innerhalb von Spark repliziert wurden. Beim „push-to-receiver“ Modell können Daten verloren gehen, wenn der Reciver abstürzt, bevor die Daten repliziert werden konnten.

Accumulators(Shared Variable) Da Spark eine Operation im Fehlerfall oder zur Rekonstruktion von aus dem Cache gefallenen RDDs wiederholt, können Updates von Akkumulatoren mehrfach ausgeführt werden. Im Fall von Actions wird jedes Update nur einmal ausgeführt.

Tathagata Das [2015-01-15] Tathagata Das et al. [2015-07-30] The Apache Software Foundation [2016t] Zaharia et al. [2012a] Zaharia et al. [2012b] Zaharia et al. [2013] kar [2015]

2.2.1.2 Speicherverwaltung

Spark verfügt über eine eigene Speicherverwaltung Project Tungsten

Unified Memory-Management Spark definiert verschiedene Speicherbereichen zwischen denen unterschieden wird, zu sehen in Abbildung 2.12. Das *Unified Memory-Management* wurde mit Spark 1.6 eingeführt und erlaubt einem Bereich über seine Grenzen hinaus in einen anderen Speicherbereich zu wachsen. Allerdings gilt dies lediglich für die von Spark selbst genutzten Bereiche *Execution Memory* und *Storage Memory*. Steht insgesamt nicht mehr genug Speicher zur Verfügung, wird der *Execution Memory* auf die Festplatte geschrieben und der

Storage Memory je nach Konfiguration ebenfalls auf die Festplatte geschrieben oder gelöscht und die Daten im Bedarfsfall neu berechnet.

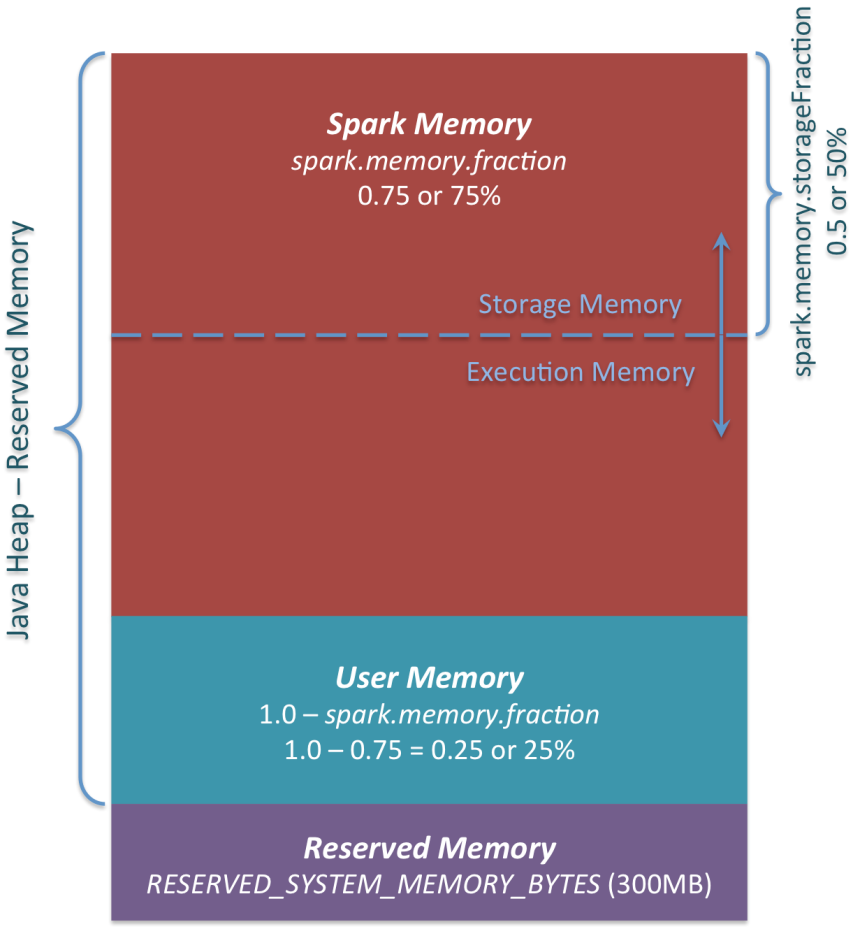

Abbildung 2.12: Speicheraufteilung von Spark Alexey Grishchenko and 0x0FFF [2016-01-28]

Execution Memory Dieser Bereich wird für Objekte genutzt die für die Ausführung einer Operation wie *shuffles,joins,sorts* und *aggregations* innerhalb eines Tasks erstellt werden.

Storage Memory Hier werden die Datenblöcke des RDD cache gespeichert. Ebenso „unrolled serialized Data“, sowie alle *Broadcast Variables* und große Ergebnisse von Task die gesendet werden sollen.

User Memory Dieser Speicherbereich ist für User-Code reserviert. Die Funktion die einem Spark-Operator mitgegeben wird, nutzt diesen Speicher.

System Memory Dieser Speicherbereich ist für Sparks interne System reserviert und ist nicht konfigurierbar, sondern hart im Quellcode kodiert.

Typ-Serialisierung Um Daten-Objekte über das Netzwerk zu senden oder auf der Festplatte zu speichern müssen diese Objekte zuerst serialisiert werden. Spark bietet drei verschiedenen Verfahren für die Serialisierung.

Java build-in Serializer Ist der Default Serialiser von Spark und kann alle Objekte die Javas *Serializable* Interface implementieren serialisieren. Javas eigener Serialisierer ist am flexibelsten, dafür aber auch der Langsamste und hat den größten Speicheroverhead.

Kryo Die Kyro Library zum serialisieren ist signifikant schneller und erzeugt einen geringen Overhead im Gegensatz zu Javas eigenem Serialisierer. Allerdings ist Kyro nicht so flexibel und unterstützt nicht alle *Serializable* Typen und erfordert vom Nutzer alle Klassen die serialisiert werden sollen bei Kyro zu registrieren um eine bessere Leistung zu erzielen.

Tungsten Serializer Der Spark eigene Serializer, welcher Teil des *Project Tungesten* ist, kann nicht vom Nutzer als Serializer konfiguriert werden. Der Tungsten Serializer wird von Spark intern verwendet und kommt bei der Verwendung der DataSet/DataFrame- und SQL-API zum Einsatz. Er nutzt ein eigenes binäres format für die Serialisierung, *Tungsten UnsafeRow format*, welches welches einen sehr geringen Overhead hat. Weiterhin wird zur Laufzeit Bytecode generiert, welcher die Serialisierung übernimmt und noch einmal einen Deutlichen Leistungsgewill gegenüber Kyro bringt.

Off-Heap Spark erlaubt es Speicher ausserhalb der JVM zu nutzen. Dies hat den Vorteil, das der Garbage Collector (GC) der JVM auf diesen Speicher keinen Zugriff hat und Spark diesen effizient selbst managen kann.

Reynold Xin and Josh Rosen [2015-04-28] Alexey Grishchenko and 0x0FFF [2016-01-28] The Apache Software Foundation [2016v]

2.2.1.3 Shuffle

Shuffle ist Sparks Mechanismus um Daten der Partitionen eines RDDs neu aufzuteilen(zwischen stages), was üblicherweise das Kopieren der Daten zwischen Executors und Rechnern(Workern) erforderlich macht. In der Regel sind hiermit Neupartitionierung, Serialisierung, Netzwerk I/O, Disk I/O, und je nach gewähltem Verfahren Sortierung, sowie optional Komprimierung

der Daten verbunden. Dies ist somit eine sehr teure Operation und muss so effizient wie möglich realisiert werden um eine gute Gesamtperformance von Spark zu ermöglichen. Die *reduceByKey* Operation etwa erfordert, dass alle Records mit gleichem Key jeweils in ein Tupel sortiert werden. Daten mit gleichem Key können aber über mehrere Executer/Worker verteilt sein, was es erforderlich macht Daten zwischen diesen auszutauschen. Weitere Operationen die ein Shuffle erfordern sind: *repartition*, *coalesce*, alle *...ByKey* Operationen wie *reduceByKey* (außer count) und alle join Operationen.

Hash Shuffle Für jeden Key einen „map"-task?, jeder „map"-task? hat mehrere „reduce"-tasks.Erzeugt für jeden „reduce"-task eine Datei. Anzahl Dateien ist „map"-task M mal „reduce"-task R ,also MxR Dateien(wird bei vielen Dateien problematisch).

Sort Shuffle Ist seit Version 1.2 der Default-Shuffle Mechanismus

Unsafe Shuffle(or Tungsten Sort) Der *Unsafe Shuffle* ist dem *Sort Shuffle* sehr ähnlich, hat jedoch zwei Optimierungen. Es wird direkt auf serialisierten Records gearbeitet, sowohl bei der Sort-Operation als auch der Merge-Operation. Des weiteren kommt ein CPU-Cache optimierter Sorter zum Einsatz.

The Apache Software Foundation [2016u] Alexey Grishchenko [2015-08-24] Kay Ousterhout [2015-06-12] Josh Rosen [2015]

2.2.1.4 Integrierte Bibliotheken

Spark Streaming *Spark Streaming* ist eine Komponente, die Spark das Verarbeiten von „live" Datenströmen erlaubt. Es bietet eine API um Datenströme sehr ähnlich der Spark-Core RDD API zu manipulieren. Sparks Abstraktion für Streams ähnelt der von RDD, genannt *discretized stream* oder *DStream*. Intern besteht ein DStream aus vielen microbatches also RDDs. Weiterhin wurde es darauf ausgelegt den gleichen Grad an Fehlertoleranz, Durchsatz und Skalierbarkeit wie der Spark-Core zu bieten. Tathagata Das [2015-01-15] Tathagata Das et al. [2015-07-30] The Apache Software Foundation [2016t] kar [2015]

Spark SQL Durch *Spark SQL* wird Spark um die Fähigkeit erweitert mit strukturierten Daten zu arbeiten und Anfragen(Querys) in SQL oder auch HiveSQL zu stellen. Als Datenquelle dienen dabei u.a. Hive tables, Parquet und JSON. Über die SQL Schnittstelle hinaus, bietet es die Möglichkeit SQL-Querys mit Manipulationen über die normale RDD API zu mixen und so SQL mit komplexer Analytik zu verbinden.

kar [2015], The Apache Software Foundation [2016s]

Spark MLlib *MLlib* bietet allgemeine Funktionalität für maschinelles Lernen und kommt mit einer Vielzahl fertiger Algorithmen aus den Bereichen Klassifikation, Regression, Clusteranalyse und Kollaborativem Filtern. Es ermöglicht „model evaluation“, Datenimport und auch low-level ML Primitive wie einen generischen „gradient descent“ optimierungs Algorithmus. kar [2015]

Spark GraphX *GraphX* ist eine Bibliothek um mit Graphen arbeiten zu können. Genau wie *Spark Streaming* und *Spark SQL* erweitert es die RDD API und ermöglicht das Erstellen eines Gerichteten Graphen und dabei den Knoten und Kanten beliebige Eigenschaften zuzuweisen. Über die Grundfunktionalitäten um Graphen zu erstellen und zu manipulieren, bietet es viele fertige allgemeine Graph-Algorithmen wie z.B. „PageRank“ und „triangle counting“. kar [2015]

2.2.2 Architektur

2.2.2.1 Komponenten-Stack

Abbildung 2.13 gibt eine Übersicht von Sparks Komponenten-Stack. Die Basisfunktionalität wird durch den *Spark Core* bereitgestellt. Dieser enthält unter anderem die APIs durch die die RDDs definiert werden, Komponenten fürs Scheduling, memory management, fault recovery und Schnittstellen zu Speichersystemen wie HDFS, lokales Dateisystem, Amazon S3, Cassandra, Hive, HBase und mehr. Auf dem Core aufbauend existieren noch Bibliotheken, die Teil des Spark-Projektes sind und dessen Funktionalität erheblich erweitern. *SparkSQL* ermöglicht es mit strukturierten Daten zu arbeiten und Anfragen in SQL und HiveSQL zu stellen. *Spark Streaming* ermöglicht die Verarbeitung von live Streams, etwa von Message Queues. *MLlib* bietet Funktionalitäten für Maschinelles Lernen. *GraphX* ist eine Bibliothek für die Arbeit mit Graphen. Für sein Cluster-Management baut Spark auf verschiedene Lösungen auf, wie dem internen *Standalone Scheduler* sowie Hadoop YARN oder Mesos, aus denen gewählt werden kann.

kar [2015]

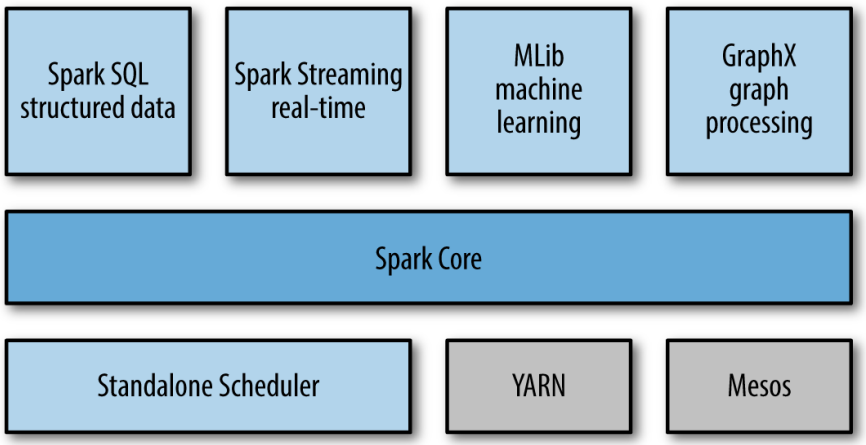

Abbildung 2.13: Der Komponenten-Stack von Apache Spark kar [2015]

2.2.2.2 Datenstrukturen

Resilient Distributed Datasets (RDD) RDDs sind eine „read only“ partitionierte Collection von Records, wobei ein Record ein beliebiges Objekt der verwendeten Programmiersprache sein kann(Python,Java,Scala). Im allgemeinen erlauben RDDs zwei Typen von Operationen *Transformations* und *Actions.Transformations* erzeugen eine neue RDD aus der vorherigen, wohingegen Actions

RDDs werden ausschließlich durch deterministische Operationen(Transformationen), entweder aus anderen RDDs oder einem Speicher erzeugt und werden „lazy“ erzeugt. Eine RDD wird durch ein Allgemeines Interface, welches fünf Type von Informationen hat, repräsentiert: eine Menge von Partitionen, eine Menge von Abhängigkeiten(Dependencys) von parent RDDs, eine Funktion um die Daten aus der parent RDD zu berechnen, Metadaten über das Partitionierungsschema sowie die Datenverteilung(data placement). RDDs sind eine Erweiterung des MapReduce Models und erweitern somit das data flow programming model auf welchem MapReduce aufbaut. Effizientes „data sharing“ zwischen verschiedenen Berechnungsstufen. Die Daten liegen in Hauptspeicher und können auf der Festplatte gespeichert werden. Der User kann die Persistenz und Partitionierung kontrollieren. DAG. Jedes RDD merkt sich den Graph von Operationen aus denen es erzeugt wurde(lineage), und kann dadurch im Fehlerfall die verlorenen Daten einer Partition des RDD effizient neu berechnen. Das Neuberechnen dieser Daten ist meistens viel schneller ohne kostspielige Replikation.

Dependency Types Die Abhänigkeit von Partitionen des Child RDD zu Partitionen des Parent RDD werden bei Spark in zwei Typen aufgeteilt.

narrow dependencies Jede Partition der Paraent RDD wird von höchstens einer Partition der Child RDD genutzt. Ein *map* führt Beispielsweise zu einer *Narrow dependency*. *Narrow dependency* erlauben die Ausführung mehrerer Operatoren in einer Pipeline, wodurch alle von einander abhängigen Partitionen auf einem worker berechnet werden können.

wide (oder shuffle) dependencies Mehrere Partitionen der Child RDD haben eine Abhängigkeit auf eine Partition der Parent RDD Ein *join* erzeugt Beispielsweise eine *wide dependency*. *wide dependency* benötigen die Daten alle Paraent Partitionen und erfordern somit unter Umständen einen Shuffle. Details zur Verteilung der Daten durch einen Shuffle sind in Abschnitt 2.2.1.3 zu finden.

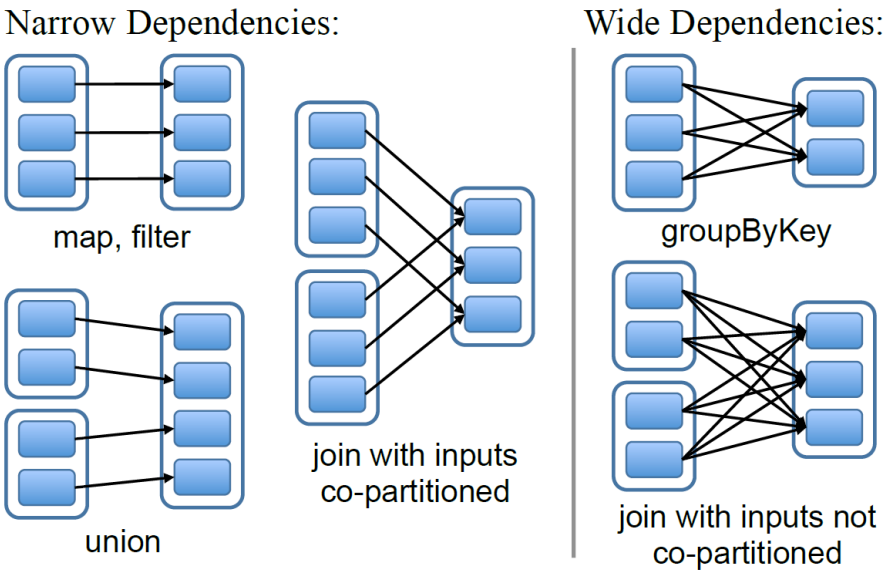

Abbildung 2.14: Beispiel von narrow und wide dependencies. Jede Box ist ein RDD, mit Partitionen dargestellt als blaue Rechtecke Zaharia [2016]

Zaharia [2016], Zaharia et al. [2012a], kar [2015]

Shared Variables(gemeinsame Variablen) *Shared Variables* dienen wie der Name schon sagt, dass über mehrere *Tasks* im Cluster hinweg Variablen gemeinsam genutzt werden können. Hierfür bietet Spark Lösungen für zwei Arten von Kommunikationstypen, Aggregation von Ergebnissen und Broadcasts.

Accumulators In Funktionen die an Spark-Operatoren übergeben werden, können Variablen, welche außerhalb der Funktion im *Driver-Programm* definiert wurden, verwendet werden.

Diese Variablen haben den Nachteil, dass Änderungen an diesen vom jeweiligen *Task* nicht an die Variable im *Driver-Programm* zurück propagiert werden können, da jeder *Task* im Cluster eine eigene Kopie dieser Variable erhält. Mit *Shared Variables* bietet Spark die Möglichkeit, Änderungen dieser äußeren Variablen an das *Driver-Programm* zurück zu propagieren. Hierbei ist festzuhalten, dass diese Variablen „write-only" sind und der jeweilige Task nur seine eigenen Änderungen sehen kann. Nur das *Driver-Programm* erhält schlussendlich die akkumulierten Werte aller *Tasks*.

Broadcast Variables *Broadcast Variables* dienen der effizienten Verteilung von Variablen mit großer Datenmenge. Zum einen ist der Startmechanismus von *Tasks* auf kleine Task-Größen optimiert. Werden des weiteren Variablen in mehreren parallelen Operationen genutzt, sendet Spark die Variablen standardmäßig für jede Operation separat erneut und für jeden Task einzeln. Für eine bessere Effizienz werden *Broadcast Variables* an jeden *Node* nur einmal versendet, wobei ein effizientes „BitTorrent" Bram Cohen [2013-10-11] ähnliches Protokoll genutzt wird. Im Gegensatz zu den *Accumulators* sind *Broadcast Variables* „read-only", werden also nicht zurück an das *Driver-Programm* propagiert und Änderungen sind ebenfalls nur im jeweiligen Task sichtbar. kar [2015]

2.2.2.3 Scheduling & Deployment

Im folgenden wird Sparks Scheduling & Deployment im „distributed mode" erläutert. Auf den local mode wird hier nicht eingegangen da dieser nur zu Entwicklungszwecken genutzt wird. In Abbildung 2.15 sind die Komponenten dargestellt die in Sparks „distributed mode" aktiv sind. In diesem Modus verwendet Spark eine Master/Slave Architektur mit einem zentralen Koordinator, dem *Driver* und mehreren verteilten „Workern", genannt *executors*. Hierbei läuft der *Driver* in einem eigenen Java-Prozess und jeder *executor* in einem separaten Java-Prozess. Zusammen werden diese als *Spark Application* bezeichnet. Eine *Spark Application* wird auf einer Menge von Rechnern von einem externen Service, dem so genannten *Cluster Manager* gestartet.

Spark Driver Der *Driver* ist der Prozess in dem die main() Methode des User-Code läuft, in dem der *SparkContext* erstellt wird, die RDDs erzeugt und Transformationen sowie Actions darauf ausgeführt werden. Sobald der *Driver* terminiert, ist auch die Applikation beendet.

Driver Komponenten Der *Driver* lässt sich noch einmal in folgende Komponenten gliedert:

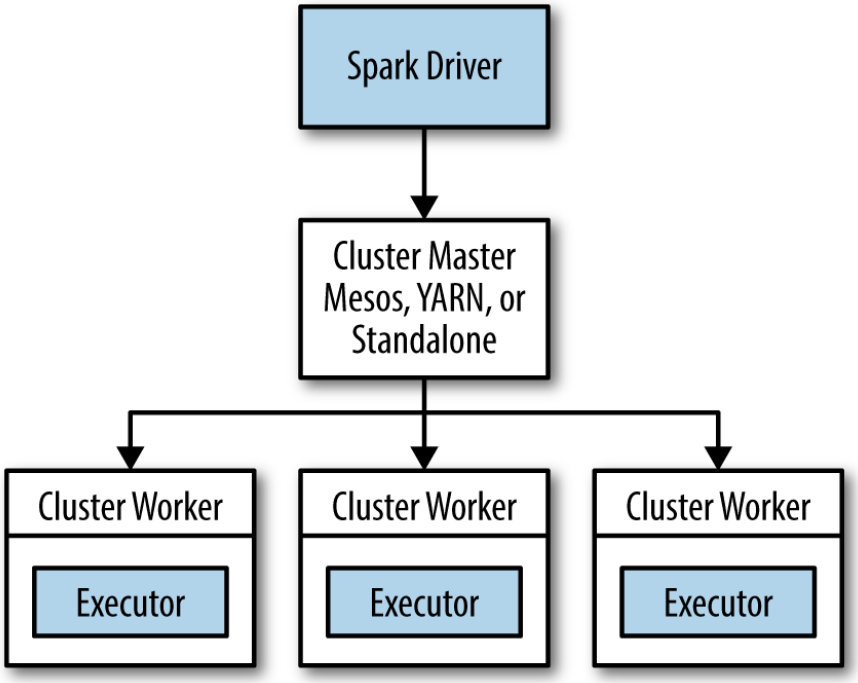

Abbildung 2.15: Die Komponenten einer Verteilten Spark-Applikation kar [2015]

RDDs Diese existieren im Driver und bilden, wie schon beschrieben, implizit einen „operator DAG“ welcher an den Scheduler übergeben wird.

Scheduler Der Scheduler ist dafür zuständig den „operator DAG“ in einen physischen Ausführungsplan zu überführen und teilt sich wiederum in den *DAGScheduler* und den *TaskScheduler* auf. Der *DAGScheduler* erstellt aus dem DAG *Stages*,welche aus *Tasks* bestehen und optimiert diese gleichzeitig. Hierbei stellen *Tasks* die kleinste „Arbeitseinheit“ dar. Der *TaskScheduler* kümmert sich um die Verteilung der einzelnen *Tasks* auf die *executor* im Cluster. Hierbei wird etwa berücksichtigt, welche Daten sich bereits auf einem *executor* befinden, die der *Task* wiederverwenden kann.

Spark Executor Executor sind die Arbeitsprozesse welche für die Ausführung der einzelnen *Tasks* und die Rückgabe der Ergebnisse an den *Driver* verantwortlich zeichnen. Sie werden beim Start einer *Spark Application* gestartet und laufen für gewöhnlich für die gesamte Ausführungszeit dieser Applikation.

Executor Komponenten Ebenso wie der *Driver* besteht auch der *Executor* aus weiteren Komponenten:

Task Threads Die Tasks die der Executor zugewiesen bekomment werden in einem Thread pool ausgeführt.

Block Manager Ist für das Speichern der Datenblöcke verantwortlich. Sowohl, je nach Storage Level, im MemoryStore oder DiskStore, sowie dem ShuffleStore. Weiterhin kümmert sich der Block Manager um die Serialisierung/Deserialisierung als auch Kompression der Datenblöcke und kann Daten über den Cluster replizieren.

Cluster Manager Spark ist auf einen *Cluster Manager* angewiesen um die *Executer* zu starten und in manchen Fällen auch den *Driver*. Er verwaltet die Ressourcen wie RAM und CPU-Kerne und ermöglicht so mehrere *Spark Applications* in einem Cluster auszuführen. Der *Cluster Manager* besteht aus einem *Master* und mehreren *Workern*, wobei die *Worker* auf den Rechnern laufen auf denen die *Executer* ausgeführt werden. Der *Cluster Manager* ist in Spark eine austauschbare Komponente und ermöglicht so, neben dem eingebauten Manager, auch externe Manager wie YARN und MESOS zu nutzen.

kar [2015], Zaharia [2016]

3 Anwendungsszenario

Der Onlinehandel ist ein Geschäftsbereich der stetig wächst, weswegen auch mit immer größere Datenmengen gerechnet werden muss. Es fallen von Nutzern generierte Daten an, wie Rezensionen und Bewertungen der Produkte, das Verhalten der Nutzer auf der Webseite oder Verbindungen zu Sozialmedia Netzwerken. Des Weiteren werden auch von der Software zum Betrieb der Webseite selbst Daten generiert, wie etwa Log-Dateien und Monitoring-Informationen von Diensten(Services). All diese Daten können zur weiteren Wertschöpfung genutzt werden oder sind sogar essentiell zum Betrieb der Onlineplattform.

Da der Onlinehandel durch Firmen wie Amazon oder Ebay vielen Menschen bekannt ist, bietet sich hiermit ein anschauliches Beispielszenario, welches durch die großen anfallenden Datenmengen für einen Vergleich von BigData-Verarbeitungssystemen prädestiniert ist und soll in dieser Arbeit als Rahmen für den praktischen Vergleich von Flink und Spark dienen.

Szenario Es wird ein Onlineshop betrieben, bei dem die Verarbeitung der durch die Nutzer anfallenden Daten durch eine neue Architektur realisiert werden soll. Flink und Spark sollen dabei als untereinander austauschbare Komponenten zur Datenverarbeitung mit anderen Komponenten wie Datenbanken und Messsagebrokern zu einem Systemen integriert werden.

3.1 User Stories

Im Folgenden werden einige mögliche User Stories beschrieben, welche konkrete Anwendungsfälle des Systems darstellen und im späteren Abschnitt zur Konzeption herangezogen werden sollen.

3.1.1 Produktempfehlungen (Recommendations)

„Als Nutzer der Website möchte ich gerne Empfehlungen zu Produkten erhalten. Dabei sollen die Empfehlungen auf mich persönlich zugeschnitten sein und sich nicht nur auf „Andere die diesen Artikel kauften, kauften auch...“ beschränken“.

Für diese User-Story müssen im Wesentlichen zwei Informationsquellen eingebunden werden. Zum einen muss das System informiert werden, für welchen Nutzer eine Empfehlung ausgegeben werden soll. Zum anderen benötigt das System historische Daten des Nutzers und weiterer Nutzer um daraus eine Empfehlung zu berechnen. Des Weiteren können zusätzliche Daten wie etwa die Kategorie eines Produktes und die Kategorie der zuletzt vom Nutzer betrachteten Produkte herangezogen werden, um die Qualität der Empfehlung zu verbessern. Systeme dieser Art sind bei Onlinehändlern wie Amazon oder Anbietern rein digitaler Produkte wie dem Video-Streaming-Dienst Netflix und dem Musik-Streaming-Dienst Spotify zu finden.

Anforderungen Zur Realisierung der User-Story gelten folgende Anforderungen:

1. Die Antwortzeit vom Senden der Anfrage bis zur Antwort soll nicht mehr als 3 Sekunden betragen.
2. Die Empfehlung soll 50 Vorschläge in absteigender Reihenfolge nach der berechneten Bewertung enthalten.
3. Die Empfehlung darf keine Produkte enthalten, die vom Nutzer bereits bewertet wurden.

3.1.2 Weitere User-Storys

3.1.2.1 Sales Dashboard

"Als Betreiber des Onlineshops möchte ich immer genau über den aktuellen Umsatz informiert sein. Dafür möchte ich den Umsatz der letzten Minute, 5 Minuten und Stunde nach Produkten und Kategorien geordnet einsehen können“

3.1.2.2 Rating/Review Toplist

„Als Betreiber des Onlineshops möchte ich sehen, was die aktuell am meisten bewerteten Produkte bzw. Produktkategorien sind“

3.2 Entwurf Produktempfehlungen (Recommendations)

Nachfolgend wird ein Prototyp für die User-Story "Produktempfehlungen“ entworfen, um anhand dessen Flink und Spark zu vergleichen.

3.2.1 Ablauf der User-Story

Im Folgenden wird der Ablauf der User-Story in Abbildung 3.1 als Aktivitätsdiagrammes dargestellt und anhand dessen genauer erläutert.

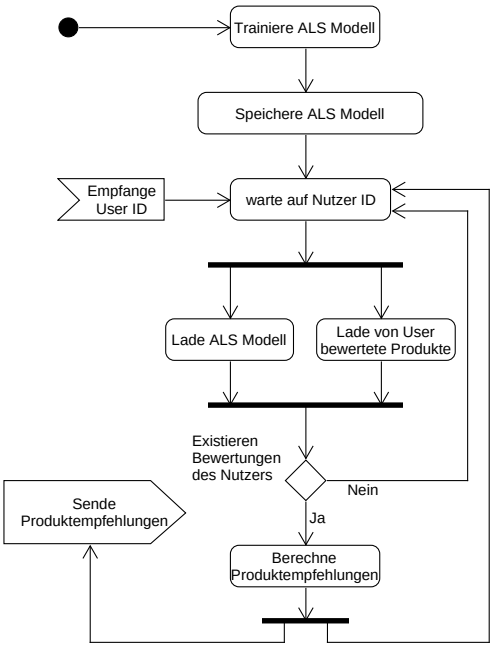

Abbildung 3.1: Ablauf der User-Story "Produktempfehlungen“ als Aktivitätsdiagramm

Zu Beginn muss das ALS Modell mit den historischen Daten trainiert und anschließen gespeichert werden. Da das trainierte Modell nicht partiell aktualisiert werden kann, muss dieser Schritt jedes Mal wiederholt werden, wenn neue Bewertungen von Nutzern in die Empfehlungen einfließen sollen. Bis hierhin geschieht die Datenverarbeitung als Stapelverarbeitung. Als nächstes startet das Streaming-Programm und wartet auf eingehende Nutzer IDs für die eine Bewertung berechnet werden soll. Für die Berechnung der Empfehlung muss das ALS Modell geladen werden und die vom jeweiligen Nutzer bereits bewerteten Produkte, um diese aus der Empfehlen herauszunehmen. Um eine Empfehlung aussprechen zu können, ist es notwendig, dass der Nutzer bereits Bewertungen getätigt hat. Im Falle keiner vorhandenen Bewertungen, wird auf die nächste Nutzer ID gewartet. Sind von dem Nutzer Bewertungen bekannt, kann eine Empfehlung berechnet und gesendet werden.

3.2.2 Systemkontext

In Abbildung 3.2 sind alle an der User-Story beteiligten Systeme aufgezeigt. Um die User-Story soweit zu realisieren, dass ein Vergleich von Flink und Spark möglich ist, werden nur die grün markierten Systeme benötigt. Aus diesem Grund, wird im Nachfolgenden nur noch auf diese eingegangen und die nicht markierten Systeme werden bei der Umsetzung vernachlässigt.

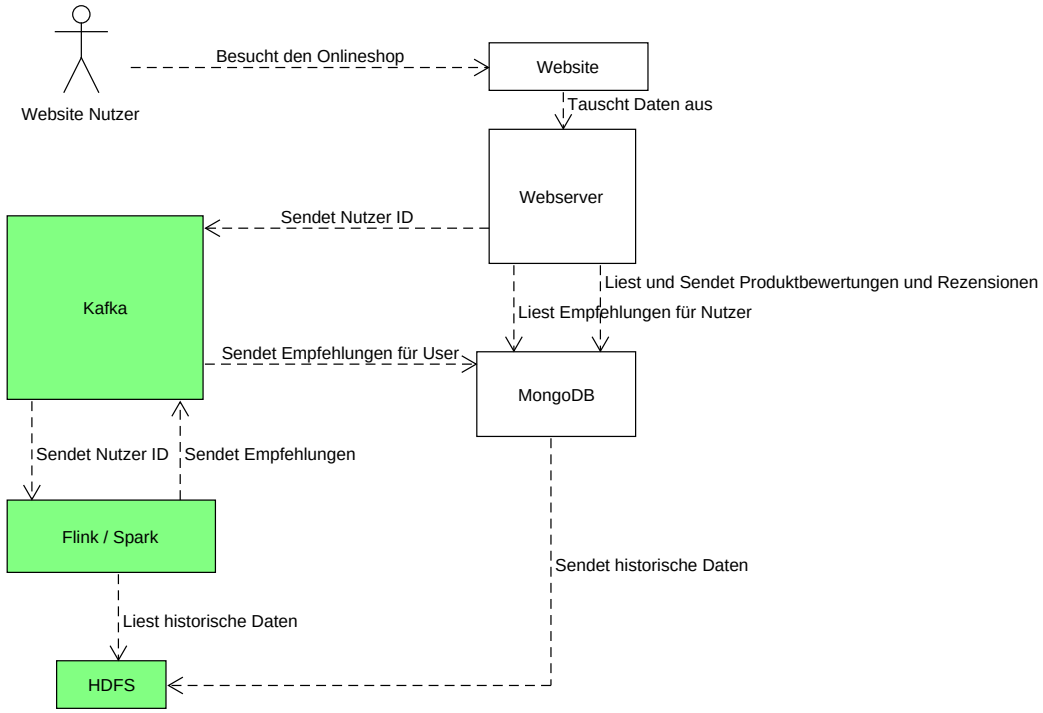

Abbildung 3.2: Systemkontext Recommendations

Stakeholder Folgende Stakeholder sind für diese User-Story besonders relevant.

Webseite Nutzer Surft auf der Webseite und möchte persönliche Produktempfehlungen erhalten

- Eine Antwortzeit von 3 Sekunden sollte nicht überschritten werden. [1]

Entwickler Die Skalierbarkeit und Ausfallsicherheit bzw. Fehlertoleranz des Systems stehen im Mittelpunkt. Neue Verarbeitungsschritte(Jobs) sollen schnell erstellt und einfach

[1] Nach einer Studie von Akamai aus 2009 warten 40% der Nutzer eines Onlineshops maximal 3 Sekunden bevor sie die Seite verlassen. Demnach wird eine Wartezeit von 3 Sekunden als Obergrenze für die Antwortzeit abgeleitet. Akamai Technologies, Inc. [2009-09-14]

im laufenden Cluster ausgeführt werden können. Folgende Anforderungen lassen sich hieraus für das System ableiten:

- Das System sollte mind. 70 Nutzer pro Sekunde bedienen können [2]
- Der Ausfall einzelner Knoten sollte die Funktionsfähigkeit des Systems nicht gefährden.

3.2.3 Verwendete Technik

Empfehlungsalgorithmus (ALS) Für Empfehlungssysteme gib es mehrere Ansätze, wie in einfachen Fällen eine Top-Liste der beliebtesten Produkte oder eine Empfehlung der Produkte die von anderen Nutzern gekauft bzw. gut bewerten wurden wie das aktuell betrachtete. In diesem Fall soll der *Collaborative filtering* Algorithmus *Alternating Least Squares* (ALS) verwendet werden.

Collaborative filtering Algorithmen ziehen die Bewertungen vieler Nutzer heran um Ähnlichkeiten in deren Bewertungen zu erkennen und auf Grund dessen eine Bewertung für eine Nutzer-Produkt Kombination zu errechnen, für die noch keine Bewertung vorliegt. Der Vorteil von *Collaborative filtering* ist, dass es automatisch Beziehungen zwischen Nutzern und Zusammenhänge unter den Produkten erkennt, die mit anderen Inhaltsbasierten Methoden, wo Produkten Eigenschaften wie etwa ein Filmgenre zugewiesen werden, schwer zu erfassen sind. Nachteilig ist, dass es unter dem *cold start* Problem leidet, welches auftritt, wenn ein neues Produkt eingeführt wird für welches noch keine Bewertungen existieren. In diesem Fall ist es nicht möglich Vorhersagen zu treffen, bis Nutzerbewertungen vorliegen. Die zwei Primären Bereiche des *Collaborative filtering* sind die *Neighborhood Methoden* und *latent factor models*.

Die *Neighborhood Methoden* konzentrieren sich auf die Berechnung von Beziehungen zwischen Produkten oder alternativ zwischen Nutzern.

Latent Factor Models sind ein alternativer Ansatz, der versucht die Bewertungen zu erklären, indem Nutzer und Produkte anhand von einer bestimmten Anzahl von Faktoren, abgeleitet aus den Bewertungsmustern, charakterisiert werden. Für Nutzer misst jeder Faktor, wie sehr er Produkte mag, die einen hohen Wert bei dessen korrespondierenden Faktor erreichen. Die Faktoren sind eine abstrakte Größe die nicht direkt erklärt werden kann. Manchmal können diese Faktoren aber mit intuitiven Konzepten korrelieren, wie etwa dem Genre von Musik oder Filmen. Der ALS Algorithmus nutzt *Latent Factor Models* um eine Bewertung für eine Nutzer-Produkt Kombination vorherzusagen. Mitunter die erfolgreichsten Realisierungen von *Latent Factor Models* basieren auf der *Matrix Factorization*, so auch ALS.

[2] Ausgehend von einem großen Onlineshop wie Amazon. Amazon hatte im Monat July 2016 183 Millionen Besucher. Statista [2016-07] Hieraus ergeben sich im Schnitt ca. 70 Besucher pro Sekunde

Matrix Factorization mit *Latent Factor Models* wurde populär durch die Kombination von guter Skalierbarkeit und hoher Vorhersagegenauigkeit. Um die Empfehlungen zu erhalten sind zwei Schritten notwendig. Zuerst wird ein Modell mit den Vorhandenen Bewertungen trainiert. Das Training ist sehr aufwendig und nimmt viel Zeit in Anspruch. Im zweiten Schritt wird das Modell abgefragt, indem diesem *(NutzerID, ProduktID)* Tupel übergeben werden für die das Modell eine Bewertung liefert. Die Abfrage des Modells ist wenig aufwändig.

Der Zentrale Schritt bei dieser Methode ist es, eine *low-rank factorization* der dünn besetzten Rating-Matrix R in eine Nutzer- und Produktmatrix U und I durchzuführen, beispielhaft in Abbildung 3.3 dargestellt. Jedes Element R_{ui} wird dabei in das Skalarprodukt $U_u^T \cdot I_i$ faktorisiert. Wobei die Anzahl der Spalten von U bzw. Zeilen von I die Anzahl der Faktoren und damit den Rang der Faktorisierung darstellen.

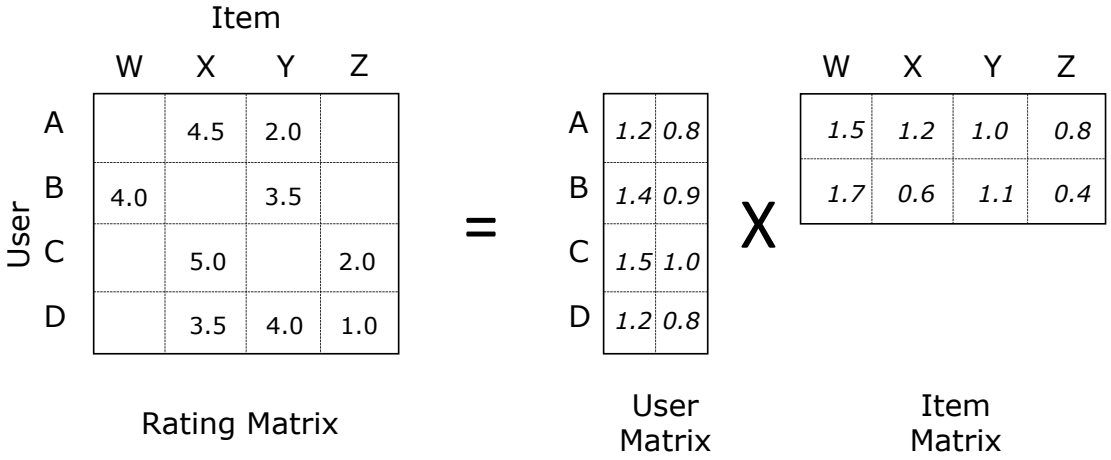

Abbildung 3.3: Matrix Factorization Till Rohrmann [2015-03-18]

Das Problem ist nun diese Faktoren, die jedes Produkt bzw. jeden Nutzer beschreiben, effizient zu berechnen. Ein geeignetes Modell zu finden, lässt sich auf die Minimierung des Root-Mean-Square Error (RMSE) zwischen existierenden Bewertungen und vorhergesagten Werten herunter brechen. Die Gleichung 3.1 stellt den RMSE mit einem Regulierungsterm um Überanpassung (overfitting) zu vermeiden dar.

Gegeben ist die Matrix von Nutzer-Produkt Bewertungen $R = (r_{ui})$ mit $u \in [1 \dots n]$ und $i \in [1 \dots m]$ wobei r_{ui} die Bewertung von Nutzer u für Produkt i darstellt.

$$\min_{X,Y} \sum_{r_{ui} \text{ exists}} \left(r_{ui} - x_u^T y_i\right)^2 + \lambda \left(\sum_u n_u ||x_u||^2 + \sum_i n_i ||y_i||^2 \right) \tag{3.1}$$

$X = (x_1, \ldots, x_n)$ ist die Matrix von Nutzer-Faktor Vektoren und $Y = (y_1, \ldots, y_m)$ ist die Matrix von Produkt-Faktor Vektoren. n_u ist die Anzahl existierender Bewertungen von Nutzer u und n_i die Anzahl der Bewertungen für Produkt i.

Der λ gewichtete Regulierungsterm liefert nach Zhou et al. [2008] die besten empirischen Resultate um Überanpassung zu vermeiden. Der optimale Wert für λ und den Rang der User- bzw. Produktmatrix können durch weitere Verfahren ermittelt werden. Für eine Optimierung könnte man etwa das Kreuzvalidierungsverfahren (Cross-validation) heranziehen um geeignete Werte herauszufinden. Diese Optimierungen sollen hier aber nicht weiter betrachtet werden und es werden feste gegebene Werte für den Rang und λ angenommen.

Gleichung 3.1 hat mit x_u und y_i 2 Unbekannte und ist daher nicht Konvex. Bindet man jetzt aber eine der Unbekannten an einen festen Wert, so erhält man ein quadratisches Optimierungsproblem, welches sich Optimal lösen lässt. Für die Nutzer-Faktoren x_u und Produkt-Faktoren y_i erhält man folgende Gleichungen die es zu lösen gilt:

$$x_u = \left(Y S^u Y^T + \lambda n_u I\right)^{-1} Y r_u^T \tag{3.2}$$

$$y_i = \left(X S^i X^T + \lambda n_i I\right)^{-1} X r^i \tag{3.3}$$

Wobei r_u der Bewertungsvektor von Nutzer u (der $u$te Zeilenvekotr von R), r^i der Bewertungsvektor von Produkt i ($i$te Spaltenvekotr von R) und $S^u \in \mathbb{R}^{m \times m}$ bzw. $S^i \in \mathbb{R}^{n \times n}$ die Diagonalmatrix für die gilt:

$$S^u_{ii} = \begin{cases} 1 & \text{if } r_{ui} \neq 0 \\ 0 & \text{else} \end{cases}$$

$$S^i_{uu} = \begin{cases} 1 & \text{if } r_{ui} \neq 0 \\ 0 & \text{else} \end{cases}$$

Für die Lösung dieser Gleichung kann u.a. etwa die *Cholesky-Zerlegung* herangezogen werden. Dies soll hier aber nicht weiter behandelt werden und es wird angenommen das ALS eine beliebige geeignete Methode zu Lösung verwendet.

ALS ist ein iterativer Algorithmus der alternierend einen der Unbekannten als fest annimmt und für die andere Unbekannte den RMSE berechnet. Indem der RMSE immer abwechseln für den Nutzer-Faktor Vektor und den Produkt-Faktor Vektor berechnet wird, wird sichergestellt dass der in Gleichung 3.1 berechnete Fehler mit jedem Schritt abnimmt und schlussendlich konvergiert. Das Resultat dieser Berechnung sind nun die Nutzer-Faktor- und Produkt-Faktor-

Matrizen die beschreiben wie sehr ein Nutzer ein Produkt mag das in den jeweiligen Faktoren einen hohen wert erreicht.

Blocked ALS Wenn ALS über mehrere Rechner verteilt eingesetzt werden soll, ist es wichtig den Kommunikationsaufwand zwischen den einzelnen Berechnungsknoten so gering wie möglich zu halten. Im ineffizientesten Fall wird die gesamte grade berechnete Nutzer- bzw. Produktmatrix an alle Knoten im Cluster versandt. Um dies zu vermeiden setzen Flink und Spark auf den so genannten *Blocked ALS* Algorithmus Dieser ermittelt welche Nutzer die gleichen Produkte bewertet haben bzw. welche Produkte von den gleichen Nutzern bewertet wurden und fasst diese Nutzer und Produkte zu Blöcken zusammen und verteilt diese auf den Rechnerknoten. So müssen nicht mehr alle Informationen an alle Knoten gesandt werden und es lassen sich erhebliche Gewinne bei der Laufzeit von ALS erreichen. Einflussfaktoren auf die Kosten(Komplexität) der Berechnung neben den Eingabedaten:

Iterationen Die Anzahl der der Berechnungsschritte zur Lösung. Ein Schritt besteht jeweils aus dem Halbschritt der Nutzer-Faktor Berechnung und dem Halbschritt der Produkt-Faktor Berechnung. Weniger Schritte bedeuten weniger Rechenaufwand aber auch eine geringere Präzision der Empfehlung.

Blöcke Die Anzahl der Nutzer- und Produktblöcke. Eine geringe Anzahl, bedeutet weniger Informationen die ausgetauscht werden müssen. Es sollte möglichst 1 Block pro Berechnungsknoten gewählt werden, was die geringste Anzahl darstellt. Wenige Blöcke führen allerdings zu einer höheren Blockgröße, welche noch in den Speicher des Knotens passen muss.

Faktoren Die Anzahl der Faktoren die für die Beschreibung der Nutzerpräferenz bzw. Produkteigenschaft genutzt werden. Mehr Faktoren bedeuten i.d.R. eine höhere Präzision der Empfehlung, erhöhen aber den Rechen- und Kommunikationsaufwand erheblich.

ALS ist sehr Berechnungs- und Kommunikationsintensiv, was dazu beitragen sollte, die Leistungsfähigkeit von Flink und Spark aus zu loten. Weiterhin ist ALS Bestandteil von Flinks sowie Sparks Bibliothek für maschinelles Lernen. Die Implementierung von Flink hat sich laut dessen Entwicklern stark an der von Spark orientiert. Die Ähnlichkeit der Implementierungen und der Umstand, dass die Implementierung von Speziallisten des jeweiligen Systems entworfen wurden, sollte einen Fairen Vergleich ermöglichen. Zhou et al. [2008], Koren et al. [2009-08], Till Rohrmann [2015-03-30], Roi Reshef [2015-12-16]

Streaming (Kafka) Um nicht für alle Möglichen Nutzer-Produkt Kombinationen vorab eine Bewertung ausrechnen und speichern zu müssen, soll das System mit einem Strom von Nutzer-IDs versorgt werden, für die eine Bewertung angefordert wird. Zu diesem Zweck wurde sich für den Message Broker Apache Kafka entschieden. Dieser zeichnet sich durch seine Skalierbarkeit, hohen Durchsatz und Fehlertoleranz aus. Zudem können an Kafka viele verschiedene Quellen angebunden werden und somit an zentraler Stelle gebündelt werden. Apache Kafka wird bereits von vielen Firmen in Produktivsystemen eingesetzt und sowohl Flink als auch Spark bieten eine gute Anbindung zu Kafka durch Konnektoren, die beide Systeme bereits mitbringen.

Historische Daten (HDFS) Um Flink und Spark mit den historischen Daten zu versorgen, wird auf das Hadoop Distributed File System(HDFS) zurückgegriffen. Da die historischen Daten in Form von CSV-Dateien vorliegen ist ein verteiltes Dateisystem wie HDFS ein naheliegender Ansatz. Zudem ist HDFS im Big Data Umfeld weit verbreitet und in vielen Umgebungen bereits vorhanden.

3.3 Realisierung Produktempfehlungen

Bei der Umsetzung der User-Story soll soviel wie möglich auf die vorhandenen Fähigkeiten der zu vergleichenden Systeme zurückgegriffen werden. Auch wenn externe Systeme, wie etwa Key/Value-Stores, oder neue Implementierungen vorhandener Komponenten eine Leistungsfähigere Umsetzung der User-Story erlauben würden, so wird hier darauf verzichtet, da der Vergleich von Flink und Spark im Fokus steht und die Möglichkeiten dieser beleuchtet werden sollen.

3.3.1 Datenstrukturen/Datenquellen

Für die Umsetzung der User-Story müssen verschiedene Datenquellen an Flink und Spark angebunden werden. Die Quellen und die Struktur der Daten wird im folgenden beschrieben.

3.3.1.1 Historische Daten

Als Grundlage für Produktempfehlungen, dienen die Filmbewertungen von MovieLensGroupLens Research [2016], einem Anbieter für Filmempfehlungen. Diese können hier GroupLens [2015-09-23] heruntergeladen werden. Die Daten liegen in Form von Comma-separated Values (CSV) Dateien vor und haben folgende Struktur:

ratings.csv userId,movieId,rating,timestamp

movies.csv movieId,title,genres

Diese Dateien werden im HDFS abgelegt und können von Flink und Spark einfach über vordefinierte Filereader für CSV-Dateien eingelesen werden.

3.3.1.2 Stream Daten

Die Nutzer IDs für die eine Empfehlung berechnet werden soll, werden durch ein Shell-Script mit dem Befehl „od -A n -t u -N 4 /dev/urandom“ erzeugt und Zeilenweise in der Form „UserId <ID>“ in einer Datei gespeichert. Die Datei wird mittels Umleitung an den "Kafka-console-producer"gereicht, welcher die IDs in ein Kafka-Topic schreibt, aus welchem Flink bzw. Spark diese lesen. Die Erzeugung der IDs mit einer guten Verteilung über den Bereich der Nutzer IDs ist wichtig, um die Anfragen vieler verschiedener Nutzer zu simulieren und eine Verfälschung der Ergebnisse durch Caching von Flink und Spark zu verhindern.

3.3.2 Umsetzung mit Flink

Im Folgenden wird die konkrete Umsetzung mit Flink beschrieben. Der Prototyp wird dabei in zwei Komponenten, eine für das Training des Modells und eine für das Streaming der Empfehlungen, aufgeteilt. Die Verbindung beider Komponenten besteht lediglich durch das trainierte Modell, welches gespeichert und anschließend geladen werden muss.

3.3.2.1 Model Training

Das Einlesen der Daten geschieht unkompliziert mittels des von Flink angebotenen FileReaders für CSV-Dateien wie in Listing 3.1 zu sehen ist.

```
1 val inputDS: DataSet[RatingWithTimestemp] =
2         env.readCsvFile[RatingWithTimestemp](
3                 filePath = dataPath + RatingsFile,
4                 ignoreFirstLine = true)
```

Listing 3.1: Flink CSV-Datei lesen

Das Training des Modells ist ebenfalls einfach durch Aufruf der *fit()* Methode und vorherigem setzen der Parameter für die Berechnung möglich.

```
1 val als:ALS = ALS()
2 als.setIterations(ALS_Iterations)
3         .setNumFactors(ALS_NumFactors)
4         .setLambda(ALS_Lambda)
```

```
        .setBlocks(alsBlocks)
        .setTemporaryPath(TempPath)
als.fit(ratings)
```

Listing 3.2: Flink ALS Modell Training

Das Speichern des Modells wird von Flink selbst nicht angeboten und muss selbst implementiert werden. Hierdurch ist eine Analyse von Flinks ALS Implementation und weitere Recherche notwendig und führt zu einem erhöhten Aufwand.

3.3.2.2 Streaming

Um nur Empfehlungen für Produkte die noch nicht Bewertet wurden zu ermöglichen, ist es nötig die bisherigen Bewertungen einzulesen und alle Bewerteten Produkte eines bestimmten Nutzers herauszufiltern. Flink bietet innerhalb von Datasets keine Suche nach Einträgen mit einem bestimmten Key an. Um dennoch Bewertungen eines bestimmten Nutzers zu erhalten, wurde zunächst auf Flinks Table-API zurückgegriffen. Hierfür wurden das eingelesene Ratings-Dataset erst bei Flinks *Table Environment* registriert und anschließen wie in Listing 3.3 zu sehen abgefragt.

```
tableEnv.scan(UsersProductsTable)
        .where("userId=" + userId)
        .select("productId")
```

Listing 3.3: Flink Table Abfrage

Diese Methode erwies sich zwar als funktional, jedoch vielfach langsamer als die eigentliche Abfrage des ALS Modells und damit als ungeeignet.

Als effizient hat sich die Implementierung einer *RichCoFlatMapFunction* herausgestellt. Mit dieser ist es möglich auf zwei verbundenen Streams zu arbeiten und auf einen *ValueState* zuzugreifen. Der *ValueState* wird genutzt um pro Nutzer-Id ein *Set[Int]* zu halten in dem die IDs der bereits bewerteten Produkte gespeichert sind. Der erste Stream besteht dabei aus Produkt-Ids und wird genutzt um den *ValueState* zu befüllen. Der zweite Stream besteht aus Nutzer-Ids welche mit dem *Set[Int]* von Produkt-Ids, welche für den jeweiligen Nutzer aus dem *ValueState* ausgelesen werden, angereichert wird. Da der Stream von Produkt-Ids aus einer Datei im CSV-Format erzeugt wird und Flink für die Streaming-API keinen CSV-Filereader anbietet, musste hierfür eine eigene Implementierung verwendet werden.

Die Anforderung, dass nur die 50 Produkte mit der am höchsten geschätzten Bewertung Empfohlen werden sollen, macht es erforderlich, dass die Empfehlungen für alle Produkte zunächst nach Bewertung sortiert werden müssen bevor die ersten 50 ausgewählt werden. Da

Flink keine globale Sortierung eines Datasets unterstützt jedoch die Sortierung von Gruppierten Einträgen, wurden zunächst alle Einträge einer einzigen Gruppe zugewiesen und diese anschließend sortiert, zu sehen in 3.4.

```
prediction.map(p => (0, Rating(p._1, p._2, p._3)))
        .groupBy(_ match {case (i,_) => i})
        .sortGroup(_ match {case (_, r) => r.rating},
                Order.DESCENDING)
        .first(50)
        .map(p => p._2)
```

Listing 3.4: Flink Ausgabe der 50 Besten Empfehlungen

Die ALS *predict()* Methode um Empfehlungen zu berechnen wird nicht für die Streaming-API angeboten. Da *predict()* einen neuen Batch-Job startet, kann dies nur vom Driver aus geschehen. Daher muss der Stream aus User-Id und Produkt-Ids erst an den Driver geleitet werden, um von dort aus die Berechnung der Empfehlungen anzustoßen. Um den Stream an den Driver zu leiten musste auf eine Methoden aus dem *contributions package* zurückgegriffen werden, welches nicht Teil der offiziellen API ist. Um auf den Umweg über den Driver zu verzichten und so das volle Potential von Flinks Streaming auszunutzen, wäre eine Evaluation der Neuimplementierung der *predict()* aufbauend auf der Streaming-API sinnvoll.

Für die Weiterleitung der Ergebnisse an ein Kafka-Topic muss ein Kafkaproducer werwendet werden. Flink bietet hier zwar eine fertige Lösung an, diese ist jedoch als Datensenke für einen Stream-Job vorgesehen und kann nicht aus dem Driver heraus verwendet werden. Hierfür wurde ein Producer direkt aus der Kafka-Library verwendet und stellte in Kombination mit Flink kein Problem dar.

3.3.3 Umsetzung mit Spark

Nachfolgend wird die konkrete Umsetzung mit Spark beschrieben. Der Prototyp wird dabei, ebenfalls wie bei Flink, in zwei Komponenten, eine für das Training des Modells und eine für das Streaming, aufgeteilt. Die Verbindung beider Komponenten besteht genau wie bei Flink durch das Speichern und Laden des trainierten Modells.

3.3.3.1 Model Training

Das Einlesen der CSV-Daten in ein RDD wird von Spark nicht durch eine fertige Implementierung unterstützt. Wie die Daten in ein RDD geladen werden und die erste Zeile mit dem Datei-Header ausgelassen wird ist in Listing 3.5 zu sehen.

```
val ratings = sc.textFile(ratingFile)
        .mapPartitionsWithIndex((partitionIdx, iterator)
                => if(partitionIdx == 0) iterator.drop(1)
                        else iterator)
        .map(_.split(","))
        .map(splitLine => Rating(splitLine(0).toInt,
                splitLine(1).toInt,
                splitLine(2).toDouble))
```

Listing 3.5: Spark CSV-Datei lesen

Das Training des Modells ist wie bei Flink mit einem einfachen Aufruf möglich 3.6.

```
val model:MatrixFactorizationModel =
        ALS.train(ratings, rank, numIterations, lambda)
```

Listing 3.6: Spark ALS Modell Training

Das Speichern des trainierten Modells ist bei Spark schon vorgesehen und ohne Aufwand möglich 3.7.

```
model.save(sc, path)
```

Listing 3.7: Spark Speichern des ALS Modells

3.3.3.2 Streaming

Um die bereits vorhanden Bewertungen eines Nutzers herauszufiltern, werden die Bewertungen in ein RDD als Tupel *(UserId, Rating)* eingelesen. Für diese *PairRDDs* aus *(key, value)* Tupeln bietet Spark eine *lookup()* Methode an um alle Werte für einen Key auszugeben. Dieser *lookup()* hat sich zunächst als nicht sehr performant herausgestellt. Die Zeiten für den *lookup()* ließen sich um ein vielfaches verbessern, indem ein neues RDD erstellt wurde bei dem alle Bewertungen pro Nutzer in einem Hashset zu (UserID, HashSet[Rating]) aggregiert wurden. Mit Sparks Möglichkeit mittels *persist()* RDDs im Speicher zu persistieren wird verhindert, dass die Daten bei jedem *lookup()* erneut eingelesen werden müssen. Ein Nachteil an der *lookup()* Methode ist, dass diese nur vom Driver und nicht von den Executoren ausgeführt werden kann. Hierdurch ist es notwendig, wie bei Flink, den Stream an den Driver zu leiten. Eine vermutlich performantere und flexiblere Methode, die in dieser Arbeit nicht weiter verfolgt wurde, wäre die Bewertungen mittels *mapWithState* direkt mit dem Stream zu verwalten. Hierdurch müsste der Stream nicht an den Driver geleitet werden und es ist zu erwarten, dass *mapWithState* zudem bedeutend schneller als ein *lookup()* ist.

Da die Filterung von bereits Bewerteten Produkten im Driver geschieht, wird die *predict()* Methode vom ALS Modell ebenfalls vom Driver als neuer Job Ausgeführt. Da *mapWithState* den Umweg über den Driver überflüssig machen würde, wäre als nächster Schritt eine Evaluation der Ausführung von *predict()* auf den Executorn als Teil der Streamverarbeitung in Kombination mit *mapWithState* sinnvoll. Hierbei käme der Umstand zum tragen, dass Sparks Streams aus Microbatches bestehen, welche normale RDDs sind und somit als Eingabe für die *predict()* Methode dienen könnten und eine neue Implementierung dieser obsolet machen würden.

Die Anforderung dass nur die Besten 50 Empfehlungen ausgegeben werden sollen kann durch Spark einfach erfüllt werden, da hierfür wie in Listing 3.8 zu sehen bereits eine Methode vorhanden ist.

```
1 prediction.top(50)(Ordering[Double].on(x=>x.rating))
```

Listing 3.8: Spark Ausgabe der 50 Besten Empfehlungen

Für die Weiterleitung der Ergebnisse verfügt Spark über keinen fertigen Producer, allerdings konnte, genau wie bei Flink, einfach ein Producer direkt aus der Kafka-Library verwendet werden.

3.3.4 Deployment

Nachfolgend wird das Deployment der verwendeten Systeme dargestellt und beschrieben. Das Deployment findet vor Ort auf realer Hardware statt. Es werden insgesamt 6 gleiche Rechner mit der folgenden Hardware- und Softwareausstattung verwendet:

Komponente	**Wert**
CPU	Intel Core i7-6700
Arbeitsspeicher	32 Gigabyte
Ethernet	1000BASE-T
Festplatte	1 Terrabyte HDD

Tabelle 3.1: Hardwareausstattung

Software	**Version**
Betriebssystem	Ubuntu 14.04 LTS
Flink	1.1.3
Spark	2.0.1
Kafka	0.10.1.0
Hadoop	2.7.3

Tabelle 3.2: Verwendete Software

Das Deployment der beiden Prototypen ist, wie in Abbildung 3.4 und 3.5 zu sehen ist, bis auf Flink und Spark identisch.

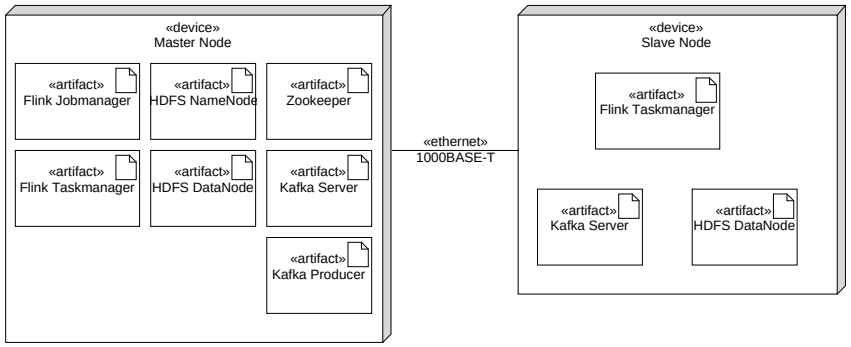

Abbildung 3.4: Deployment mit Flink als Datenverarbeitungskomponente

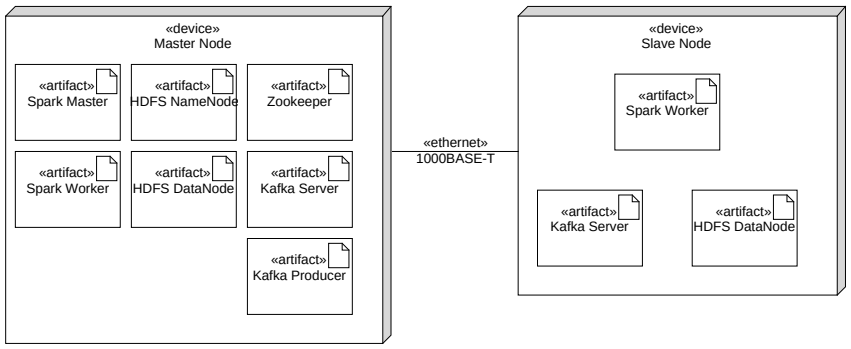

Abbildung 3.5: Deployment mit Spark als Datenverarbeitungskomponente

In der Dargestellten Grafik sind zwei verschiedene Rechnerkonfigurationen mit den darauf installierten Software Artefakten zu sehen. Ein Master auf dem die Koordination der verschiedenen Systeme stattfindet und ein Slave der nur für die Verarbeitung der Daten zuständig ist. Der Master ist im Cluster genau einmal vorhanden, wohingegen es beliebig viele Slavesysteme geben kann. Da der Cluster aus 6 Rechnern besteht, werden für die Tests 0 bis 5 Slaves eingesetzt. Um die volle Kapazität des Cluster zu nutzen wird der Master zusätzlich zur Koordination, ebenfalls wie die Slaves, zur Datenverarbeitung genutzt. Die Koordination der Flink Taskmanager wird durch den Jobmanager von Flink durchgeführt, respektive werden die Spark Worker vom Spark Master koordiniert. Durch die HDFS DataNodes wird auf die Daten im HDFS zugegriffen und der HDFS NameNode verwaltet die DataNodes. Das Lesen und Schreiben von Kafka-Topics geschieht über die Kafka Server. Für die Koordination der Server setzt Kafka auf Apache Zookeeper. Der Kafka Producer wird nur auf dem Master ausgeführt und wird genutzt um das Kafka-Topic für eingehende Empfehlungsanfragen mit Nutzer-Ids zu

befüllen. Um den Anfänglichen Aufwand für das Deployment gering zu halten wurden die eigenen Clustermanager der jeweiligen Systeme genutzt.

In einem Produktiveinsatz wäre es von Vorteil einen Clustermanager wie Mesos oder YARN einzusetzen, um das Management der Infrastruktur und die Verteilung der Ressourcen des Clusters zu optimieren.

Auf eine Replikation der Koordinatoren des jeweiligen Softwaresystems wurde in diesem Setup bewusst verzichtet, um mehr Ressourcen für die Datenverarbeitung zur Verfügung zu haben. Um eine möglichst hohe Ausfallsicherheit eines Produktivsystems zu gewährleisten, wäre es angebracht drei Replikate von jedem Koordinator zu haben. Flink, Spark, HDFS und Kafka bieten hier alle die Möglichkeit der Replikation.

4 Diskussion

Im folgenden Kapitel werden die beiden betrachteten Systeme evaluiert und bewertet. Zunächst folgt eine Beschreibung des Vorgehens

4.1 Beschreibung des Vorgehens

Um die Bewertung durchzuführen wird auf das Goal-Question-Metric Paradigma nach Victor R. Basili et al. [1994] zurückgegriffen. Dies soll dafür sorgen, dass ein „zielorientiertes“ Messen durchgeführt wird. Dies bedeutet dass das Messen nicht Selbstzweck sein soll, sondern einem Übergeordneten Ziel dient und klar ist, wofür die Ergebnisse verwendet werden. Hierfür werden Fragen aus dem Messziel abgeleitet. Wobei hier der Vergleich von Flink und Spark das Übergeordneten Ziel darstellt und sich das Messziel aus dem betrachteten Qualitätsfokus ergibt. Dabei ist darauf zu achten, dass die Fragen zu den untersuchten Qualitätsmerkmalen so zu formuliert sind, dass sie sich durch quantitative Ergebnisse der Messungen beantworten lassen. Dies macht einen Vergleich Möglich und ermöglicht zudem das System zu einem späteren Zeitpunkt mit sich selbst zu vergleichen.

Um die Fragen und Messziele zu entwickeln wird ein Abstraction Sheet wie in Danilo Assmann et al. [2002-12-23] beschrieben verwendet, dargestellt in Tabelle 4.1. Der Abstraction Sheet ermöglicht es die Ergebnisse der für das GQM Modell durchzuführenden Schritte kompakt darzustellen und zu dokumentieren. Bezüge der durchgeführten Schritte untereinander werden durch Nummerierung hergestellt. Zudem werden dem Messziel im Abstraction Sheet folgende 5 Aspekte zugeordnet:

Objekt Das Objekt für welches die Untersuchung durchgeführt werden sollen.

Zweck Der Zweck aus welchem die Analyse durchgeführt wird.

Qualitätsfokus Der Qualitätsfokus sagt aus, auf welche Qualitätsmerkmale der Software sich die Analyse beziehen soll.

Perspektive Die Perspektive liegt fest aus welchem Blickwinkel die Software betrachtet werden soll, z.b. aus Sicht eines Entwicklers oder Nutzers.

Kontext Der Kontext bestimmt, für welche Umgebung die Messungen und gemachten Angaben gelten sollen.

Die Schritte bestehen grob daraus, zunächst aus dem Übergeordneten Ziel ein Messziel zu erstellen. Zu den Messzielen werden Fragen formuliert, aus denen sich Metriken ableiten lassen. Die Metriken stellen eine Konkretisierung der Fragen auf quantitativer Ebene dar und werden später herangezogen um die Fragen auf quantitative Weise zu beantworten. Die Fragen basieren auf den Qualitäts- und Einflussfaktoren und deren Beantwortung unterstützt das Messziel. Zudem sollten Hypothesen bezüglich möglicher Antworten formuliert werden. Sobald die zur Metrik gehörenden Messwerte gesammelt wurden, stehen alle Informationen zur Beantwortung der Fragen zur Verfügung. Nun können die Messungen analysiert die Ergebnisse interpretiert und die Fragen beantwortet werdenDanilo Assmann et al. [2002-12-23].

Objekt	**Zweck**	**Qualitätsfokus**	**Perspektive**	**Kontext**
Qualitätsfaktoren			Einflussfaktoren	
Welche Faktoren/Metriken müssen betrachtet werden?			Welche Variablen haben einen Einfluss auf die Qualitätsfaktoren?	
Hypothese			Einfluss auf Hypothese	
Was sind die momentanen Erwartungen bezüglich der Qualitätsfaktoren?			Wie beeinflussen die Einflussfaktoren die Qualitätsfaktoren?	

Tabelle 4.1: Sammeln von relevanten Asspekten zu einem Messziel in einem Abstraction Sheet nach Danilo Assmann et al. [2002-12-23]

4.1.1 Betrachtete Qualitätsfokusse

Für die Bewertung der zu vergleichenden Systeme werden die Qualitätsmerkmale nach DIN/ISO 9126 herangezogen. Aus diesen wurde sich in dieser Arbeit für die Qualitätsfokusse Skalierbarkeit, Erlernbarkeit, Bedienbarkeit und Stabilität entschieden. Diese sollen eine möglichst vergleichbare Bewertung des Gebrauchs von Flink und Spark ermöglichen. Hierbei nimmt die Skalierbarkeit eine besondere Stellung ein, da dies bei der Verarbeitung sehr großer Datenmengen eines der Hauptargumente für die Verwendung eines solchen Systems ist. Die Qualitätsfokuse werden nachfolgen im Kontext dieser Untersuchung kurz erläutert.

Skalierbarkeit In diesem Fall sollte das System so entworfen sein, dass es schnell mit der Menge der zu verarbeitenden Daten mitwachsen kann bzw. eine feste Größe an Daten mit mehr Ressourcen schneller verarbeiten kann. Da große Datenmengen die Kapazität einzelner

Rechner schnell übersteigen können, wird hier die horizontale Skalierbarkeit der Systeme betrachtet, also die Skalierung durch Hinzufügen zusätzlicher Rechenknoten.

Erlernbarkeit Die Erlernbarkeit gibt hier an wie viel Aufwand für einen Entwickler nötig ist sich in ein neues System einzuarbeiten und wie Intuitiv die Verwendung des Systems ist.

Bedienbarkeit Die Bedienbarkeit wird hier aus der Sicht eines Entwicklers betrachtet und betrifft die Arbeit mit dem jeweiligen System. Es wird betrachtet wie viel Aufwand ein Entwickler bei der Umsetzung einer Aufgabe mit dem System betreiben muss und welche Probleme sich bei der Arbeit mit dem System ergeben.

Stabilität Die betrachteten Systeme werden stetig weiterentwickelt. Durch die Veränderung der Systeme können neue Fehler entstehen. Die Stabilität charakterisiert die Anfälligkeit des System, dass durch Änderungen unbeabsichtigte Wirkungen auftreten, welche einen negativen Einfluss auf das System haben.

4.2 Evaluation

4.2.1 Skalierbarkeit

4.2.1.1 Messumgebung

Bei der Messung der Skalierbarkeit wird auf den Cluster die in Abschnitt Deployment 3.3.4 zurückgegriffen.

4.2.1.2 Ablauf der Messung

M1: Um die Messung durchzuführen werden als Datenquelle die in Abschnitt Realisierung Produktempfehlung 3.3.1 beschriebenen Filmbewertungen verwendet. Um Erkenntnis über die Skalierbarkeit zu erlangen wird die Messung mit 1-6 Spark Workern bzw. Flink Taskmanagern durchgeführt. Hierbei wird die Anzahl an Datenverarbeitungsknoten von einem Rechner in Einser Schritten bis zu 6 Rechnern gesteigert. Bei der Messung wurden die in Tabelle 4.4 dargestellten Parameter verwendet. Um Messungenauigkeiten und „Ausreißer“ zu minimieren wurde jede Messung 3 mal durchgeführt. Die Zeitmessung beginnt beim Start des an den Cluster übertragenen Programmes und endet mit der Speicherung des trainierten Modells. Nach Beendigung des Programms wird die benötigte Zeit ausgegeben

Objekt	Zweck	Qualitätsfokus	Perspektive	Kontext
Recommendation Prototyp (Flink & Spark)	Evaluation	Skalierbarkeit	Entwickler	Cluster vor Ort
Qualitätsfaktoren			Einflussfaktoren	
QF1: Zeit zum Training des ALS Modells QF2: Durchschnittliche Verarbeitungszeit pro Nachricht QF3: Latenz von Anfrage bis Antwort			EF1: Anzahl vorhandener Nutzer EF2: Anzahl vorhandener Produkte EF3: Anzahl vorhandener Ratings	
Hypothese			Einfluss auf Hypothese	
QF1: Der Zeitaufwand um das ALS Modell zu trainieren sinkt linear mit der Anzahl an Verarbeitungsknoten. QF2: Die Verarbeitungszeit sinkt linear mit der Anzahl an Verarbeitungsknoten. QF3: Die Latenz bleibt konstant solange die Anzahl an Anfragen nicht den maximalen Durchsatz des Systems übersteigt und sich kein Rückstau bildet.			EF1 zu QF1: Mit steigender Anzahl an Nutzern steigt die Trainingszeit. EF2 zu QF1: Mit steigender Anzahl an Produkten steigt die Trainingszeit. EF3 zu QF1: Mit steigender Anzahl an Ratings steigt die Trainingszeit. EF2 zu QF2: Mit steigender Anzahl an Produkten steigt die Verarbeitungszeit. EF2 zu QF3: Mit steigender Anzahl an Produkten steigt die Latenz.	

Tabelle 4.2: Abstraction Sheet Skalierbarkeit

Frage	Metrik
F1: Wie verhält sich der Zeitaufwand zum Training des Modells zu der Anzahl eingesetzter Rechnerknoten?	M1: Lasttest mit Zeitmessung
F2: Wie verhält sich die durchschnittliche Verarbeitungszeit zu der Anzahl eingesetzter Rechnerknoten?	M2: Lasttest mit Zeitmessung
F3: Wie verhält sich die Latenz zu der Anzahl eingesetzter Rechnerknoten?	M3: Lasttest mit Zeitmessung

Tabelle 4.3: Abgeleitete Fragen zum Abstraction Sheet Skalierbarkeit

Parameter	Flink	Spark
Blöcke	8 pro Worker	automatisch durch Spark
Iterationen	10	10
Faktoren	10	10
Lambda	0.01	0.01

Tabelle 4.4: ALS Model Training Parameter

M2: Aus den an den Prototypen gestellten Anforderungen geht hervor, dass 70 Anfragen pro Sekunde verarbeitet werden können sollen. Hierzu werden zunächst 70 zufällige Nutzer-Ids in das Kafka-Topic geschrieben, so wie in Abschnitt Realisierung Produktempfehlung 3.3.1 beschrieben. Es wird die Zeit gemessen, die zwischen eintreffen der ersten Anfrage und Antwort auf die letzte Anfrage verstreicht. Die durchschnittliche Verarbeitungszeit errechnet sich aus $VerstricheneZeit/AnzahlAnfragen$. Auch hier wurde jede Messung 3 mal ausgeführt. Ebenso wird die Anzahl an Datenverarbeitungsknoten von einem Rechner in Einser Schritten bis zu 6 Rechnern gesteigert. Da die gemessene Zeit für 70 Anfragen bereits sehr hoch ist, wurde auf die Messung für eine höhere Anzahl an Anfragen verzichtet.

M3: Da die Berechnung der Empfehlungen wie in Abschnitt Realisierung Produktempfehlungen 3.3.2 und 3.3.3 beschrieben seriell erfolgt, ergibt sich die Latenz aus der Verarbeitungszeit der Anfrage plus der Verarbeitungszeit aller vorher eingetroffenen Anfragen, dessen Berechnungen noch ausstehen. Die minimale Latenz ist gleich der Verarbeitungszeit einer Anfrage, die maximale Latenz gleich der Verarbeitungszeit der gesamten eintreffenden Anfragen. Wegen der seriellen Verarbeitung lässt sich die Latenz aus den in M2 gewonnenen Ergebnissen berechnen.

4.2.1.3 Datenerhebung

Die Messergebnisse wurden in einer Tabelle festgehalten, aus denen die Grafiken 4.1, 4.2 und 4.3 generiert wurden. Jeder einzelne Messpunkte stellt jeweils einen Durchschnitt aus den 3 identisch durchgeführten Messungen dar.

4.2.1.4 Messergebnisse

Tabelle 4.5 fasst die Messergebnisse von M1, M2 und M3 zusammen. Zu den gemessenen Werten von M1 und M2 ist jeweils das Delta zum vorherigen Messwert angegeben, um die Veränderung der Werte präziser zu erfassen. Da M3 von M2 abhängt wurde hier auf die Angabe von einem Delta verzichtet. Der Vereinfachung halber wird im weiteren Verlauf der Begriff Worker synonym für Spark Worker und Flink Task Manager verwendet.

System	1 Worker	2 Worker	3 Worker	4 Worker	5 Worker	6 Worker
M1						
Flink	122,815	84,092 Δ 31,529 %	75,234 Δ 10,533 %	74,586 Δ 0,8608 %	71,613 Δ 3,986 %	69,877 Δ 2,423 %
Spark	88,674	81,857 Δ 7,687 %	49,645 Δ 39,352 %	51,411 Δ -3,558 %	43,995 Δ 14,426 %	47,532 Δ -8,039 %
M2						
Flink	7,279	6,611 Δ 9,167 %	6,581 Δ 0,463 %	6,515 Δ 1,005 %	6,479 Δ 0,546 %	6,683 Δ -3,153 %
Spark	1,312	1,157 Δ 11,807 %	1,124 Δ 2,866 %	1,134 Δ -0,860 %	1,108 Δ 2,232 %	1,133 Δ -2,211 %
M3						
Flink	min: 7,279 max: 509,543 ∅: 258,411	min: 6,611 max: 462,833 ∅: 234,722	min: 6,581 max: 460,689 ∅: 233,635	min: 6,515 max: 456,058 ∅: 231,286	min: 6,479 max: 453,564 ∅: 230,021	min: 6,683 max: 467,866 ∅: 237,275
Spark	min: 1,312 max: 91,879 ∅: 46,595	min: 1,157 max: 81,030 ∅: 41,093	min: 1,124 max: 78,707 ∅: 39,915	min: 1,134 max: 79,385 ∅: 40,259	min: 1,108 max: 77,612 ∅: 39,360	min: 1,133 max: 79,329 ∅: 47,313

Tabelle 4.5: Messergebnisse Skalierbarkeit. Alle Angaben in Sekunden.

4.2.1.5 Interpretation

Im Folgenden werden die Ergebnisse der durchgeführten Tests interpretiert und die mit dem Abstraction Sheet entwickelten Fragen beantwortet. Nachfolgend sind die gesammelten Messergebnisse in den Abbildungen 4.1, 4.2 und 4.3 visualisiert.

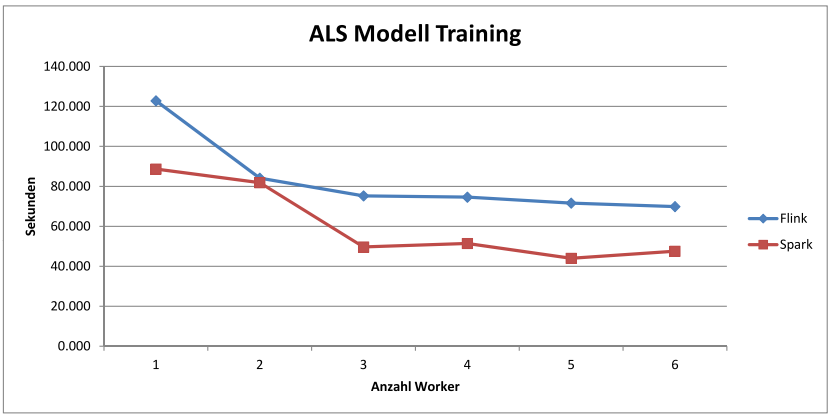

Abbildung 4.1: M1: Messergebnis des ALS Model Trainings mit Flink und Spark

In Abbildung 4.1 sind die ALS Modell Trainingszeiten von Flink und Spark dargestellt. Sowohl bei Flink als auch Spark ist bei Erhöhung von 1 auf 2 und 3 Worker eine deutliche Verbesserung der Trainingszeit zu erkennen. Ab einer Anzahl von 4 und mehr Workern ist bei Flink lediglich eine sehr geringe Verbesserung festzustellen. Dies lässt vermuten, dass durch Hinzufügen von mehr als 6 Workern kaum eine Veränderung Wahrzunehmen sein wird und eine weitere Skalierung bei dieser Datenmenge nicht mehr zu erwarten ist.

Bei Spark ist ebenfalls eine Verbesserung der Trainingszeit festzustellen, wenn zum ersten noch ein zweiter und dritter Worker hinzukommt. Bei 4 Workern steigt die Zeit wieder an, beim fünften sinkt diese wieder und beim sechsten ist wieder ein Anstieg der Trainingszeit zu verzeichnen. Die Fluktuationen die sich ab dem vierten Worker einstellen, lassen vermuten, dass hier keine relevanten Veränderungen der Trainingszeit beobachtet werden, sondern es sich nur noch um Messungenauigkeiten handelt. Dies legen nahe, dass schon ab hier die Grenze der Skalierung erreicht ist und sich bei mehr als 6 Workern ebenfalls keine Verbesserung der Zeit zeigen wird.

Dass sich schon ab dem vierten Worker keine weitere Verbesserung der Zeiten zeigt, lässt sich vermutlich auf einen erhöhten Kommunikationsaufwand zurückführen, welcher den Gewinn an Rechenkapazität kompensiert.

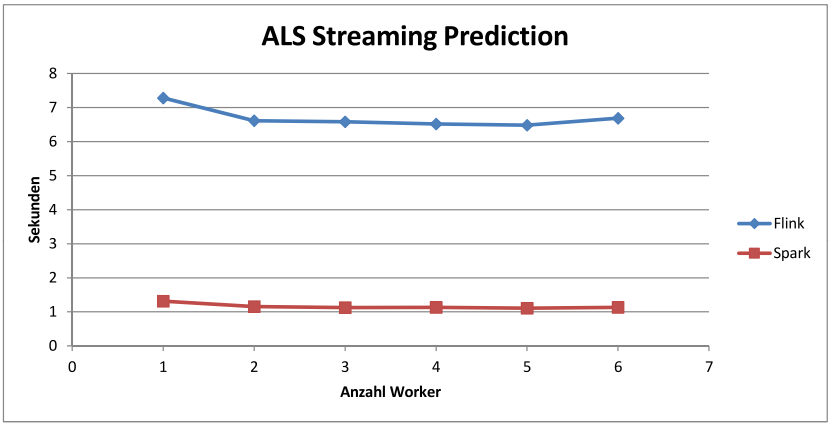

Abbildung 4.2: M2: Messergebnis der ALS Prediction mit Streaming von Flink und Spark

Abbildung 4.2 zeigt die vom Prototypen des jeweiligen Systems benötigte Zeit um eine Anfrage zu beantworten. Beide Systeme lassen eine leichte Verbesserung der Verarbeitungszeit beim Einsatz von zwei statt einem Worker erkennen. Ab dem Einsatz von 3 Workern ist die Verbesserung der Zeit so gering, dass diese als Messungenauigkeit eingestuft werden kann.

Das Ergebnis legt nahe, dass ein Worker für die Verarbeitung einer Anfrage bereits ausreicht und der Overhead für die Verteilung der Berechnung den Zugewinn durch mehr Rechenkapa-

zität ausgleicht. Eine Skalierung des Systems müsste demnach nicht über die Verteilung einer Berechnung auf viele Worker, sondern durch die parallele Bearbeitung vieler Berechnungen auf vielen Workern gleichzeitig geschehen.

Da durch die serielle Abarbeitung der Anfragen, die Latenz in direktem Zusammenhang mit der Verarbeitungszeit einer Anfrage steht, ist aus der Abbildung direkt ersichtlich dass der Prototyp mit Flink, mit ca. 6,5 bis 7,5 Sekunden deutlich über den von der User-Story geforderten 3 Sekunden liegt. Spark liegt mit ca. 1 bis 1,5 Sekunden deutlich darunter und würde die Anforderung erfüllen. Dies stellt allerdings nur die minimale Latenz dar. Wie aus Tabelle 4.5 ersichtlich ist die durchschnittliche Latenz von Spark mit ca. 40 bis 47 Sekunden deutlich über der geforderten Latenz und die von Flink mit 230 bis 258 Sekunden noch einmal um ein vielfachen höher. Hieraus lässt sich folgern, dass eine parallele Verarbeitung vieler Anfragen gleichzeitig unabdingbar ist, um den Anforderungen der User-Story gerecht zu werden.

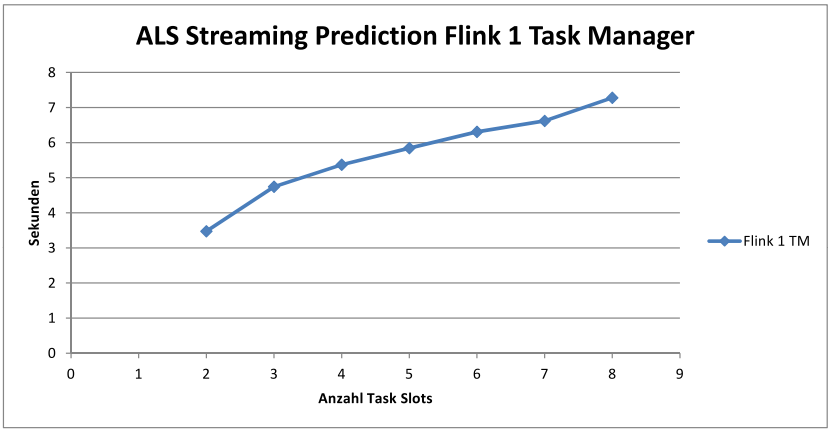

Abbildung 4.3: M2.1: Messergebnis der ALS Prediction mit Streaming von Flink auf einem Task Manager

Abbildung 4.3 stellt die Verarbeitungszeit einer Anfrage auf nur einem Flink Task Manager, also nur einem Rechner dar. Diese Messung wurde unternommen am den gravierenden Unterschied von Flink zu Spark genauer zu beleuchten. Wie in Kapitel 2.1 bereits beschrieben stellen Task Slots bei Flink die kleinste Einheit der Parallelität dar, wobei hier 1 Task Slot einen CPU Kern nutzt. Die Messung fängt bei 2 Task Slots an, da der Streaming-Job immer mindestens einen Task Slot benötigt. Alle weiteren Task Slots werden für die Berechnung der Produktempfehlung verwendet.

Zu erkennen ist, dass entgegen der Intuition, eine Erhöhung der Parallelität zu einem Anstieg der Berechnungszeit führt. Eine mögliche Erklärung ist, dass durch die bei der Berechnung der

Produktempfehlung nötigen Join und Group Operationen und der damit verbundenen Shuffels, ein so großen Overhead für die Verteilung der Daten entsteht, dass dieser die eigentliche Berechnungszeit übersteigt.

Beantwortung der Fragen

Flink

F1: Wie verhält sich der Zeitaufwand zum Training des Modells zu der Anzahl eingesetzter Rechnerknoten?

Mit zunehmender Zahl an Rechnerknoten verringert sich die nötige Zeit zum Training des Modells. Hierbei ist zu bemerken, dass sich die Verringerung des Zeitaufwands von 1 auf 2 und von 2 auf 3 Rechnerkonten am deutlichsten zeigt. Ein Einsatz weiterer Rechner hat nur einen geringen Einfluss.

F2: Wie verhält sich die durchschnittliche Verarbeitungszeit zu der Anzahl eingesetzter Rechnerknoten?

Der Sprung in der Verringerung der Verarbeitungszeit von einem auf zwei Rechnerknoten ist noch deutlich zu erkennen. Die Auswirkung von weiteren Rechnern fällt sehr gering aus.

F3: Wie verhält sich die Latenz zu der Anzahl eingesetzter Rechnerknoten?

Der Sprung in der Verringerung der Verarbeitungszeit von einem auf zwei Rechnerknoten ist noch deutlich zu erkennen. Die Auswirkung von weiteren Rechnern fällt sehr gering aus.

Spark

F1: Wie verhält sich der Zeitaufwand zum Training des Modells zu der Anzahl eingesetzter Rechnerknoten?

Mit zunehmender Zahl an Rechnerknoten verringert sich die nötige Zeit zum Training des Modells. Ab 4 Knoten Weisen die Messergebnisse Fluktuationen auf und die Trainingszeit verringert sich nicht weiter.

F2: Wie verhält sich die durchschnittliche Verarbeitungszeit zu der Anzahl eingesetzter Rechnerknoten?

Der Sprung in der Verringerung der Verarbeitungszeit von einem auf zwei Rechnerknoten ist noch zu erkennen. Die Auswirkung von weiteren Rechnern fällt sehr gering aus.

F3: Wie verhält sich die Latenz zu der Anzahl eingesetzter Rechnerknoten?

Der Sprung in der Verringerung der Verarbeitungszeit von einem auf zwei Rechnerknoten ist noch zu erkennen. Die Auswirkung von weiteren Rechnern fällt sehr gering aus.

4.2.2 Erlernbarkeit

Objekt	**Zweck**	**Qualitätsfokus**	**Perspektive**	**Kontext**
Flink & Spark	Evaluation	Erlernbarkeit	Nutzer (Entwickler)	Einsatz eines neuen Datenverarbeitungssystems
Qualitätsfaktoren			Einflussfaktoren	
QF1: Aufwand zum Erlernen der vom System eingeführten Konzepte QF2: Probleme die beim Erlernen des Systems auftreten			EF1: Vorwissen des Nutzers	
Hypothese			Einfluss auf Hypothese	
QF1: Für die Arbeit mit dem System müssen einige neue Konzepte erlernt werden, die Anzahl ist aber gering genug um einen zügigen Einstieg zu ermöglichen. QF2: Es gibt viele Probleme die auftreten können.			EF1 zu QF1: Kennt der Nutzer bereits vom System genutzte Konzepte bzw. ist mit ähnlichen Konzepten vertraut, kann dies den Lernaufwand erheblich verringern. EF1 zu QF2: War der Nutzer bereits mit ähnlichen Problemen konfrontiert, können diese schneller gelöst werden.	

Tabelle 4.6: Abstraction Sheet Erlernbarkeit

Frage	**Metrik**
F1: Wie viele Konzepte müssen zum Arbeiten mit dem System erlernt werden?	M1: Anzahl der zu erlernenden Konzepte.
F2: Wie viele Probleme treten beim erlernen des Systems auf?	M2: Anzahl er Fragen zum jeweiligen System in der *Stack Overflow* Entwickler Community.

Tabelle 4.7: Abgeleitete Fragen zum Abstraction Sheet Erlernbarkeit

4.2.2.1 Ablauf der Messung

Die Erhebung der gemessenen Daten erfolgte am 14.12.2016.

M1: Analyse der Dokumentation von Spark und Flink. Nicht alle im Kapitel 2.1 und 2.2 vorgestellten Konzepte werden für die Arbeit mit dem jeweiligen System benötigt. Da einige vorgestellte Konzepte lediglich dem Verständnis der internen Funktionsweise dienen und für die Arbeit mit dem System nicht benötigt werden, werden diese hier nicht aufgeführt.

M2: Analyse der Suchergebnisse auf *Stack Overflow* zum jeweiligen System.

4.2.2.2 Messergebnisse

System	Konzepte
Flink	DataSets, DataStreams, JobGraph, Streaming Windows, Time, Out-of-order Events, State, Fault Tolerance, Typen Serialisierung, Iterationen, Deployment
Spark	Fault Tolerance, RDD, RDD Persistence, DStream, Accumulators(Shared Variable), Dependency Types, Typen Serialisierung, Shuffle, Deployment

Tabelle 4.8: Aufzählung zu erlernender Konzepte

System	M1	M2	M3	M4
Flink	11 Konzepte	1,298 Fragen mit Inhalt "flink“ 662 Fragen mit dem Tag „[flink]“		
Spark	9 Konzepte	42,941 Fragen mit Inhalt „spark“ 21,285 Fragen mit dem Tag „[spark]“		

Tabelle 4.9: Messergebnisse Erlernbarkeit

4.2.2.3 Interpretation

Wie aus Tabelle 4.9 ersichtlich, müssen für Flink 11 und für Spark 9 Konzepte gelernt werden. Hierbei ist der Lernaufwand vergleichbar obwohl Flink 2 Konzepte mehr aufweist. Dies kompensiert sich, da die Konzepte Streaming Windows, Time, Out-of-order Events thematisch sehr nah beieinander sind. Der Lernaufwand bei Spark verringert sich dadurch, dass DStreams auf RDDs aufbauen und sich hierdurch Synergieeffekte bemerkbar machen.

Mit 1,298 Fragen zu Flink auf Stack Overflow und 42,941 Fragen zu Spark, scheint Spark zunächst mehr Probleme beim Erlernen zu bereiten. Dies muss aber vor dm Hintergrund betrachtet werden, dass Spark viel verbreiteter ist als Flink und somit auch von mehr Entwicklern

Fragen gestellt werden. Tatsächlich ist eine hohe Zahl an Fragen positiv zu betrachten, da so schon viele typische Probleme geklärt wurden.

Beantwortung der Fragen

Flink

F1: Wie viele Konzepte müssen zum Arbeiten mit dem System erlernt werden?
Für die Arbeit mit Flink müssen 11 Konzepte gelernt werden.

F2: Wie viele Probleme treten beim erlernen des Systems auf?
Mit 1,298 Fragen auf Stack Overflow scheint Flink wenige Probleme zu bereiten, allerdings muss dies vor dem Hintergrund einer kleineren Community betrachtet werden. Zudem helfen viele beantwortete Fragen anfängliche Probleme beim Erlernen zu verringern.

Spark

F1: Wie viele Konzepte müssen zum Arbeiten mit dem System erlernt werden?
Für die Arbeit mit Spark müssen 9 Konzepte gelernt werden.

F2: Wie viele Probleme treten beim erlernen des Systems auf?
Mit 42,941 Fragen auf Stack Overflow scheint Spark viele Probleme zu bereiten, allerdings ist die hohe Anzahl an Fragen auch auf die große Community hinter Spark zurückzuführen und viele beantwortete Fragen helfen einem die eigenen Probleme schneller zu lösen.

4.2.3 Bedienbarkeit

4.2.3.1 Ablauf der Messung

Die Erhebung der gemessenen Daten erfolgte am 14.12.2016.

M1: Es werden die beobachteten Probleme bei der Umsetzung der User-Story Recommendation gezählt.

M2: Die aufgetretenen Probleme werden kurz beschrieben und in eine Kategorie eingeteilt.

M3: Es wird der Zeitaufwand gemessen, der benötigt wurde um die User-Story umzusetzen. Da die Zeit zur Umsetzung nicht exakt festgehalten wurde, erfolgt die Angabe in Wochen.

Objekt	Zweck	Qualitätsfokus	Perspektive	Kontext
Flink & Spark	Evaluation	Bedienbarkeit	Nutzer (Entwickler)	Umsetzung einer User-Story mit Einsatz eines neuen Datenverarbeitungssystems
Qualitätsfaktoren			Einflussfaktoren	
QF1: Anzahl der aufgetretenen Probleme bei der Entwicklung des Recommendation-Prototyps QF2: Komplexität der aufgetretenen Probleme QF3: Aufwand zur Umsetzung einer User-Story QF4: Bereitgestellte Funktionalität die den eigenen Implementierungsaufwand verringert QF5: Eine große und aktive Community um das System			EF1: Vorwissen des Nutzers EF2: Verbreitung des Systems EF3: Anzahl Mitwirkenden bei der Entwicklung des Systems	
Hypothese			Einfluss auf Hypothese	
QF1: Es werden Probleme auftreten, die Anzahl wird jedoch gering sein. QF2: Die auftretenden Probleme werden sich ohne großen Aufwand lösen lassen. QF3: Der Aufwand zur Umsetzung einer User-Story wird gering sein und sich auf maximal eine Woche pro User-Story und System beschränken. QF4: Funktionalität die allgemeine und häufig auftretende Problemstellungen betrifft wird angeboten. QF5: Spark wird eine größere Community aufweisen als Flink, da sich Spark bereits etabliert hat.			EF1 zu QF1: Einige Probleme könnten aufgrund des Vorwissens von vorne herein vermieden werden. EF1 zu QF2: Der Aufwand zur Lösung eines Problems verringert sich, wenn der Nutzer bereits ähnliche Probleme bewältigt hat. EF1 zu QF3: Umso größer das Vorwissen, desto schneller wird sich eine User-Story umsetzen lassen. EF2 zu QF5: Mit der Verbreitung des Systems wird die Größe der Community steigen. EF3 zu QF4: Umso mehr Menschen das System weiterentwickeln, desto mehr Kapazität ist vorhanden um neue Funktionalität zu Implementieren.	

Tabelle 4.10: Abstraction Sheet Bedienbarkeit

M4: Es werden die Code-Zeilen des Prototyps gemessen. Leere Zeilen und Kommentare werden nicht berücksichtigt.

Frage	Metrik
F1: Wie viele Probleme sind bei der Umsetzung der User-Story aufgetreten?	M1: Anzahl der aufgetretenen Probleme.
F2: Wie aufwendig war es die aufgetretenen Probleme zu lösen?	M2: Einteilung der Probleme nach Aufwand zum Lösen und anschließende Zählung der Probleme pro Kategorie.Die Kategorien sind *leicht*, *mittel*, *hoch*, wobei die Einteilung nach relativem Zeitaufwand für die Lösung des Problems geschieht.
F3: Wie viel Zeit wurde benötigt um die User-Story Recommendations umzusetzen?	M3: Zeitaufwand zur Umsetzung der User-Story Recommendations.
F4: Wie viel an benötigter Funktionalität zur Umsetzung der User-Story Recommendations stellt das verwendete System bereits zur Verfügung?	M4: Benötigte Lines of Code (LoC) zur Umsetzung der User-Story Recommendations.
F5: Wie gut findet man Hilfe bei der Lösung von Problemen mit dem jeweiligen System?	M5: Anzahl beantworteter Fragen auf *Stack Overflow* und Anteil an den gestellten Fragen.

Tabelle 4.11: Abgeleitete Fragen zum Abstraction Sheet Bedienbarkeit

M5: Es erfolgt eine Analyse der als gelöst markierten Fragen im Vergleich zur Gesamtzahl der Fragen. Die Abfrage erfolgt mit „flink isanswered:true“ für Flink und mit „spark isanswered:true“ für Spark.

4.2.3.2 Messergebnisse

System	M1	M2	M3	M4	M5
Flink	8 Probleme	hoch: 3 mittel: 1 leicht: 4	6,5 Wochen	286 LoC 25 Loc Buildfile	Beantwortete Fragen: 413 Fragen insgesamt: 1,298
Spark	3 Probleme	hoch: 1 mittel: 1 leicht: 1	2,5 Wochen	195 LoC 10 Buildfile	Beantwortete Fragen: 13,063 Fragen insgesamt: 42,941

Tabelle 4.12: Messergebnisse Bedienbarkeit

Aufgetretene Probleme mit Flink

Performance Optimierung Kategorie: hoch

Das Filtern der bereits bewerteten Produkte war zunächst nicht performant genug und musste wie in 3.3.2 beschrieben angepasst werden.

Flink hat eine Warnung ausgegeben, dass die verwendete Bibliothek „com.github.fommil.netlib.BLAS“ nicht geladen werden kann. Da dies eine Bibliothek zur performanten Berechnung von arithmetischen Operationen ist, wurde dieser Fehler druch Installation zusätzlicher Bibliotheken behoben. Dies brachte allerdings nicht den erwarteten Performancegewinn.

Da Flink wie in 3.3.2 beschrieben keine globale Sortierung unterstützt, wurde getestet, ob ein Weglassen der Sortierung vor der Übertragung an den Driver und anschließender Sortierung im Driver einen Geschwindigkeitsvorteil bringt. Dies war nicht der Fall.

Fehlende Streaming API für ALS Kategorie: hoch

Eine fehlende Streaming-API Unterstützung für ALS machte die Verwendung von *StreamUtil.collect()*, wie in Kapitel 3.3.2 bechrieben, nötig. Der Parallelismus für Batch- und Streamingjobs die gleichzeitig laufen muss manuell eingestellt werden.

Speichern des ALS Modells Kategorie: hoch

Die manuelle Implementierung der Speicherung, beschrieben in Kapitel 3.3.2, war relativ aufwendig.

Local vs. Clustermode Kategorie: mittel

Das Lesen von Dateien im Lokalen Modus vom lokalen Dateisystem unterscheidet sich von dem im Clustermodus.

Das im Cluster auszuführende Programm wird in Form einer .jar Datei an den Cluster geschickt. Eine auf dem zum entwickeln genutzten Laptop erzeugte .jar Datei konnte nicht auf dem Cluster ausgeführt werden. Der Quellcode musste direkt auf dem Rechner im Cluster, auf dem der Driver läuft, compiliert werden. Eine bessere Lösung ließ sich nicht finden.

Einrichtung Buildfile Kategorie: leicht

Bei der Einbindung von Flink Streaming zeigten sich Probleme

Scala API Extensions Kategorie: leicht

Ohne die Einbindung der Scala API Extensions war die Verwendung anonymer Funktionen nur umständlich möglich.

Anzahl Netzwerkpuffer Kategorie: leicht
Die Anzahl der von Flink verwendeten Netzwerkpuffer musste manuell angepasst werden um die Fehlerfreie Ausführung des Prototypen zu ermöglichen

Größe Akka Framesize Kategorie: leicht
Die von Flink verwendete Akka Framesize musste angepasst werden, um große Daten per *collect()* zum Driver zu schicken.

Aufgetretene Probleme mit Spark

Performance Optimierung Kategorie: hoch
Das Filtern der bereits bewerteten Produkte war anfänglich, beschrieben in Kapitel 3.3.3, nicht ausreichend performant um den Anforderungen der User-Story gerecht zu werden.

Local vs. Clustermode Kategorie: mittel
Die Konfiguration des zu verwendenden Speichers weicht im Localmode von der im Clustermode ab.

Das Lesen von Dateien im Lokalen Modus vom lokalen Dateisystem unterscheidet sich von dem im Clustermodus.

Einlesen der CSV-Dateien Kategorie: leicht
Beim einlesen der CSV-Dateien stellte sich das Weglassen der ersten Zeile in der Datei als Problem heraus.

4.2.3.3 Interpretation

Die bei der Umsetzung der Prototypen gesammelten Erfahrungen, zeigen das sich bei der Entwicklung von Flink und Spark verschiedene Probleme ergeben können. Dabei traten mit Flink mehr Probleme auf als mit Spark, wobei die Probleme mit Flink auch aufwendiger als mit Spark waren. Dies schlägt sich in der Entwicklungszeit des Prototypen nieder, welche bei Flink mit 6,5 Wochen gegenüber Spark mit 2,5 Wochen das 2,5 fache beträgt. Der erhöhte Implementierungsaufwand bei Flink zeigt sich auch bei der Anzahl von LoC die für die Umsetzung des Prototypen nötig waren. Auch der Aufwand zur Konfiguration des Buildfiles spiegelt in dessen LoC Zahl wieder. Die Anzahl von beantworteten Fragen auf Stack Overflow macht auch deutlich, dass es für Probleme mit Spark deutlich einfacher ist Lösungen zu finden.

Beantwortung der Fragen

Flink

F1: Wie viele Probleme sind bei der Umsetzung der User-Story aufgetreten?

Es traten insgesamt 8 nennenswerte Probleme auf.

F2: Wie aufwendig war es die aufgetretenen Probleme zu lösen?

Mit 3 von 8 Problemen traten relativ viele Probleme mit erhöhtem Zeitaufwand auf. Ein mittleres Problem und 4 leichte fallen dagegen nicht so stark bei der Entwicklungszeit ins Gewicht.

F3: Wie viel Zeit wurde benötigt um die User-Story Recommendations umzusetzen?

Die Umsetzung der User-Story dauerte ca. 6,5 Wochen.

F4: Wie viel an benötigter Funktionalität zur Umsetzung der User-Story Recommendations stellt das verwendete System bereits zur Verfügung?

Obwohl die Loc mit 286 noch nicht sonderlich viele sind, so sind dies doch 47,7 % mehr als bei Spark. Dies korreliert mit der fehlenden Funktionalität die selbst Implementiert werden musste.

F5: Wie gut findet man Hilfe bei der Lösung von Problemen mit dem jeweiligen System?

Die Hilfestellung die sich durch die 413 beantwortete Fragen bei Stack Overflow bietet hält sich im Gegensatz zu Spark sehr in Grenzen.

Spark

F1: Wie viele Probleme sind bei der Umsetzung der User-Story aufgetreten?

Es traten insgesamt 3 nennenswerte Probleme auf.

F2: Wie aufwendig war es die aufgetretenen Probleme zu lösen?

Mit jeweils einem Problem mit hohem, einem mit mittlerem und einem mit leichtem Zeitaufwand, ist der Aufwand zur Problemlösung als eher gering einzuschätzen.

F3: Wie viel Zeit wurde benötigt um die User-Story Recommendations umzusetzen?

Die Umsetzung der User-Story dauerte ca. 2,5 Wochen.

F4: Wie viel an benötigter Funktionalität zur Umsetzung der User-Story Recommendations stellt das verwendete System bereits zur Verfügung?

Mit 195 LoC waren für Spark 31.8 % weniger Loc notwendig als für die Umsetzung mit Flink. Dies spiegelt die höhere Zahl von Spark angeboten Funktionen zur Umsetzung der User-Story wieder.

F5: Wie gut findet man Hilfe bei der Lösung von Problemen mit dem jeweiligen System?

Mit 13,063 beantworteten Fragen auf Stack Overflow zu Spark, wird hier für viele Probleme eine Hilfestellung angeboten.

4.2.4 Stabilität

Objekt	Zweck	Qualitätsfokus	Perspektive	Kontext
Flink & Spark	Evaluation	Stabilität	Nutzer (Entwickler)	Flink & Spark System
Qualitätsfaktoren			Einflussfaktoren	
QF1: Menge der bekannten Fehler QF2: Zeit zwischen erkennen und beheben der Fehler QF3: Behobene Fehler innerhalb eines Major Release und nachfolgenden Bugfix Releases QF4: Anzahl an Mitwirkenden am Projekt			EF1: Nicht entdeckte Fehler EF2: Verbreitung des Systems	
Hypothese			Einfluss auf Hypothese	
QF1: Beide Systeme werden viele bekannte Fehler haben. QF2: Die Zeit zum Beheben der Fehler wird mit QF4 in Verbindung stehen. QF3: Die Anzahl behobener Fehler wird ebenfalls mit QF4 in Verbindung stehen. QF4: Spark wird mehr Mitwirkende als Flink haben, da Spark etablierter und eine längere Zeit als Flink ein Apache Projekt ist.			EF1 zu QF1: Die Menge der nicht entdeckten Fehler erhöht die Menge der bekannten Fehler in unbekannter Höhe zu einem unbekannten Zeitpunkt. EF1 zu QF2: Durch Aufdeckung unbekannter Fehler erhöht sich die Menge an zu behebenden Fehlern, dies erhöht die Arbeitslast der Entwickler des Systems und bei Übersteigung der verfügbaren Kapazität wird es länger dauern die Fehler zu beheben. EF2 zu QF1: Eine hohe Verbreitung des Systems wird die Rate mit der Fehler gefunden werden erhöhen und damit auch die Menge bekannter Fehler. EF2 zu QF4: Eine hohe Verbreitung des Systems wird das Interesse an der Mitentwicklung des Systems erhöhen.	

Tabelle 4.13: Abstraction Sheet Stabilität

Frage	Metrik
F1: Wie viele Fehler des Systems sind bekannt?	M1: Anzahl an Bugs im Issue-Tracker Jira den Spark und Flink nutzen.
F2: Wie lange dauert es bis gemeldete Fehler behoben sind?	M2: Analyse der Bearbeitungszeit im Issue-Tracker Jira.
F3: Wie viele Fehler können innerhalb eines Major Release und nachfolgender Bugfix Releases behoben werden?	M3: Anzahl der behobenen Fehler für die jeweilige Version in Jira
F4: Wie viele Entwickler wirken am jeweiligen System mit?	M4: Anzahl der Contributor. Die Gesamtzahl an Mitwirkenden kann bei Github, dem Git Repository Hosting Service den Spark und Flink nutzen, eingesehen werden.

Tabelle 4.14: Abgeleitete Fragen zum Abstraction Sheet Stabilität

4.2.4.1 Ablauf der Messung

Die Erhebung der gemessenen Daten erfolgte am 14.12.2016.

M1: Analyse der in Jira festgehaltenen Bugs für das jeweilige System.

M2: Analyse der in Jira als gelöst geltenden Bugs, mit dem Hilfe des Resolution Time Gadgets von Jira, für das jeweilige System.

M3: Analyse der Bugs in Jira die als Fixversion die jeweilige Releaseversion haben.

M4: Analyse der beim Github Repository angegebenen Contributor für das jeweilige System.

4.2.4.2 Messergebnisse

System	M1	M2	M3	M4
Flink	2,534 Bugs	Im Jahr 2015 durchschnittlich: 31 Tage Im Jahr 2016 durchschnittlich: 41 Tage	Major (1.1.0): 215 Behobene Fehler Bugfix (1.1.1): 8 Behobene Fehler Bugfix (1.1.2): 14 Behobene Fehler Bugfix (1.1.3): 18 Behobene Fehler	256 Contributor
Spark	8,161 Bugs	Im Jahr 2015 durchschnittlich: 60 Tage Im Jahr 2016 durchschnittlich: 63 Tage	Major (2.0.0): 916 Behobene Fehler Bugfix (2.0.1): 197 Behobene Fehler Bugfix (2.0.2): 62 Behobene Fehler	1008 Contributor

Tabelle 4.15: Messergebnisse Stabilität

4.2.4.3 Interpretation

Spark weist mit 8,161 Bugs zu 2,534 Bugs bei Flink deutlich mehr Fehler auf. Dies könnte daran liegen, dass durch Sparks höhere Verbreitung mehr Fehler gemeldet werden und somit die Zahl unentdeckter Fehler geringer als bei Flink ist. Andererseits hat Spark mit 1008 zu 256 bei Flink, deutlich mehr Contributor. Dies könnte ein Indiz dafür sein, dass durch die größere Menge an gleichzeitigen Entwicklungen am System auch mehr Fehler entstehen. In dem untersuchten Majorrelease und den zugehörigen Bugfix Releases weist Spark eine 4,6 fach höhere Zahl an behobenen Fehlern auf. Dies Relativiert sich allerdings etwas durch die 3,2 fach geringere Anzahl von Bugs bei Flink. In der Reaktion auf gemeldete Fehler ist Flink mit 31 zu 60 Tagen bei Spark für das Jahr 2015 deutlich besser. Im Jahr 2016 verringerte sich der Abstand etwas aber ist mit 41 Tagen bei Flink und 63 bei Spark noch sehr deutlich.

Beantwortung der Fragen

Flink

F1: Wie viele Fehler des Systems sind bekannt?

Zum Zeitpunkt der Messung waren 2,534 Fehler bekannt.

F2: Wie lange dauert es bis gemeldete Fehler behoben sind?

Im Jahr 2015 durchschnittlich 31 Tage und im Jahr 2016 durchschnittlich 41 Tage.

F3: Wie viele Fehler können innerhalb eines Major Release und nachfolgender Bugfix Releases behoben werden?

Im Release 1.1.0 mit den drei nachfolgenden Bugfix Releases konnten 255 Fehler behoben werden.

F4: Wie viele Entwickler wirken am jeweiligen System mit?

Flink hat 256 Contributor.

Spark

F1: Wie viele Fehler des Systems sind bekannt?

Zum Zeitpunkt der Messung waren 8,161 Fehler bekannt.

F2: Wie lange dauert es bis gemeldete Fehler behoben sind?

Im Jahr 2015 durchschnittlich 60 Tage und im Jahr 2016 durchschnittlich 63 Tage.

F3: Wie viele Fehler können innerhalb eines Major Release und nachfolgender Bugfix Releases behoben werden?

Im Release 1.1.0 mit den zwei nachfolgenden Bugfix Releases konnten 1175 Fehler behoben werden.

F4: Wie viele Entwickler wirken am jeweiligen System mit?

Spark hat 1008 Contributor.

4.3 Auswertung

Nachfolgend findet die Auswertung der Evaluationsergebnisse statt. Dafür gibt Tabelle 4.16 eine Übersicht aller Kernergebnisse der durchgeführten Evaluationspunkte.

Bei der Evaluierung der Skalierbarkeit zeigt sich, dass Flink und Spark ein ähnliches Verhalten zeigen. Das Hinzufügen von Rechnerknoten zeigt bei beiden Systemen eine ähnliche Dimension der Leistungssteigerung. Auch wenn beide Systeme ähnlich auf eine Erhöhung der Verfügbaren Kapazitäten reagieren, sei jedoch festzuhalten, dass Spark eine deutlich höhere Verarbeitungsgeschwindigkeit aufweist. Dies ist insbesondere bei der Streaming Komponente des Prototypen deutlich geworden und zeigt sich vor allem bei der Verarbeitungszeit einer Anfrage. Da die Latenz durch die serialisierte Abarbeitung der Anfragen direkt von der Verarbeitungszeit abhängt, stellt sich das Ergebnis hier gleich dar.

Die Untersuchung der Erlernbarkeit ergab, dass beide Systeme zwar unterschiedlich viele Konzepte verwenden die erlernt werden müssen, sich der Aufwand für das Lernen aber gleicht.

Qualitätsfokus	Ergebnis
Skalierbarkeit	• Bis zu einer gewissen Anzahl an Workern skalieren beide Systeme beim Training des ALS Modells worauf eine Stagnation folgt. • Beim Streaming verhalten sich die Prototypen mit Flink und Spark auch ähnlich, ab 3 Workern tritt keine Verbesserung mehr ein. • Die Latenz beider Prototypen ist im Durchschnitt zu groß um die Anforderung der User-Story zu erfüllen.
Erlernbarkeit	• Für die Arbeit mit Flink müssen 11 und mit Spark 9 Konzepte gelernt werden. Wobei der Aufwand für beide in etwa gleich groß ist. • Es es können viele Probleme beim Erlernen beider Systeme auftreten. Insbesondere die Anzahl an Fragen zu Spark auf Stack Overflow deuten darauf hin. Gleichzeitig sind diese aber auch ein Hinweis auf die weite Verbreitung und große Community von Spark.
Bedienbarkeit	• Probleme bei der Nutzung treten bei beiden Systemen auf. Flink setzt sich im negativen Sinne aber deutlich beim nötigen Aufwand zur Umsetzung einer User-Story ab.
Stabilität	• Flink kann sich im Positiven bei der Anzahl bekannter Fehler und der Reaktion auf Fehlermeldungen absetzen. • Spark ist bei der Anzahl behobener Fehler pro Release deutlich besser, was sich durch die große Anzahl an Contributorn erklären lässt.

Tabelle 4.16: Zusammenfassung der Evaluationsergebnisse

Bei der Umsetzung der User-Story Recommendations zeigte sich, dass sich die große Community hinter Spark als sehr Hilfreich bei der Lösung von Problemen herausstellte. Flink kann hier nur eine kleinere Community aufweisen, die zwar ebenfalls hilfreich bei der Lösung von Problemen sein kann aber nicht mit der von Spark vergleichbar ist.

Die Bedienbarkeit beider Systeme unterscheidet sich deutlich. Spark hat sich hier als das erheblich einfacher zu nutzende System herausgestellt. Die im Gegensatz zu Flink bedeutend größere Anzahl an Contrubutorn und größere Community haben hierauf einen großen Einfluss. Ähnlich wie bei der Erlernbarkeit zeigte sich, dass wenige Treffer bei Suche nach Problemlösungen die Arbeit mit Flink erschwerten.

Bei der Stabilität scheinen beide Systeme ein gutes Maß erreicht zu haben. Flink kann sich hier durch die schnellere Reaktion auf Fehlermeldungen und eine geringere Anzahl an bekannten Fehlern hervorheben. Wohingegen Spark sich durch die hohe Anzahl an behobenen Fehlern pro Release hervortut.

5 Fazit

5.1 Zusammenfassung

Ziel der Arbeit war es, Apache Flink und Apache Spark Konzeptionell und experimentell mittels eines praxisrelevanten Anwendungszenarios zu vergleichen. Hierfür wurden zunächst die hinter dem jeweiligen System stehenden Konzepte und Architekturen erläutert. Hierbei hat sich gezeigt, dass Flink und Spark auf teilweise ähnliche Konzepte setzen, wie etwa bei der Speicherverwaltung und Serialisierung von Daten.

Für den experimentellen Vergleich wurde die User-Story Produktempfehlungen(Recommendations), welche im Kontext des Onlinehandels steht, konzipiert und realisiert. Im Rahmen der Konzeption wurde der Empfehlungsalgorithmus Alternating Least Squares(ALS), welcher dem bereich des Collaborative filtering zuzuordnen ist vorgestellt. Hieraus entstand jeweils ein Prototyp mit Flink und Spark die beide den vorgestellten ALS Algorithmus einsetzen.

Zum Abschluss der Arbeit erfolgte in einer Diskussion die Evaluierung der betrachteten Systeme. Hierbei wurde sich auf verschiedene Kriterien der Qualität von Software konzentriert. Der aus der User-Story hervorgegangene Prototyp wurde herangezogen um die Evaluation der Skalierbarkeit von Flink und Spark durchzuführen. Die Messungen wurden dabei auf einem Cluster vor Ort aus einem Verbund noch 6 Rechnern durchgeführt. Weitere Kriterien die betrachtet wurden sind Erlernbarkeit, Bedienbarkeit und Stabilität. Um eine Systematische Bewertung zu gewährleisten wurde auf das Goal-Question-Metrik Verfahren zurückgegriffen. Aufbauend auf den Ergebnissen der Evaluation, hat sich herausgestellt, dass die Umsetzung des Prototyps sowohl mit Flink als auch Spark nicht den in der User-Story gestellten Anforderungen entsprach. Weiterhin zeigte die Auswertung, dass Spark auf eine breite Community zurückgreifen kann, was auf die große Verbreitung von Spark zurückzuführen ist. Hieraus ergeben sich Vorteile bei der Erlernbarkeit und Bedienbarkeit von Spark gegenüber Flink. Nach der Evaluation wurden in der Auswertung die wichtigsten Erkenntnisse Zusammengefasst.

5.2 Ausblick

Bei der Entwicklung des Prototyps und der anschließenden Evaluation von Flink und Spark, haben sich einige interessante Punkte herausgestellt, die im Rahmen dieser Arbeit nicht mehr betrachtet werden konnten. Diese Punkte sollen im folgenden kurz vorgestellt werden und zeigen, wie aufbauend auf dieser Arbeit, die nächsten Schritte für eine weiter greifende Evaluierung von Flink und Spark aussehen können. Da sich bei der Evaluation herausstellte, dass eine große Anzahl an Contributorn und Communitymitgliedern von großem Vorteil für ein Projekt wie Flink und Spark sein können, sollte untersucht werden wie sich der produktive Einsatz der Systeme entwickelt. Weiterhin sollte untersucht werden wie der Trend beim Wachstum der Community von Flink und Spark aussieht. Auf technischer Seite wäre eine Umsetzung der Streaming Komponente der Recommendation User-Story mit einer selbst entwickelten Streaming-API Implementierung von Flinks ALS predict() Methode interessant. Um das weitere Potential von Spark in Hinsicht auf den entwickelten Prototypen auszuloten wäre eine Implementierung mit der mapWithState() Methode anstatt dem lookup() sinnvoll. Natürlich ist auch eine Umsetzung weiterer der vorgestellten User-Storys denkbar. Als nächsten konsequenten Schritt, bietet sich hier die Entwicklung einer Architektur für das in dieser Arbeit betrachtete Anwendungsszenario an. Hierduch ist zu erwarten, noch tiefere Einblick in die Verwendung mit Flink und Spark zu bekommen und so eine umfassendere Evaluierung durchzuführen. Abschließend wäre noch ein Vergleich mit anderen Systemen denkbar.

Literaturverzeichnis

Learning spark: [lightning-fast data analysis], 2015. URL https://www.safaribooksonline.com/library/view/learning-spark/9781449359034/. OCLC: 844872440.

Adrian Colyer. Spinning fast iterative dataflows, 2015-06-18. URL https://blog.acolyer.org/2015/06/18/spinning-fast-iterative-dataflows/. abgerufen am 2016-06-26.

Akamai Technologies, Inc. Akamai reveals 2 seconds as the new threshold of acceptability for eCommerce web page response times, 2009-09-14. URL https://www.akamai.com/de/de/about/news/press/2009-press/akamai-reveals-2-seconds-as-the-new-threshold-of-acceptability-for-e jsp. abgerufen am 2016-12-01.

Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The stratosphere platform for big data analytics. 23(6):939–964, 2014-12. ISSN 1066-8888, 0949-877X. doi: 10.1007/s00778-014-0357-y. URL http://link.springer.com/10.1007/s00778-014-0357-y. abgerufen am 2016-04-09.

Alexey Grishchenko. Spark architecture: Shuffle, 2015-08-24. URL https://0x0fff.com/spark-architecture-shuffle/. abgerufen am 2016-08-15.

Alexey Grishchenko and 0x0FFF. Spark memory management, 2016-01-28. URL https://0x0fff.com/spark-memory-management/. abgerufen am 2016-08-14.

AMPLab. AMPLab - UC berkeley, 2016. URL https://amplab.cs.berkeley.edu. abgerufen am 2016-07-18.

Shivnath Babu. Nephele, 2012a. URL http://stratosphoere.weebly.com/nephele.html. abgerufen am 2016-05-31.

Shivnath Babu. PACT, 2012b. URL http://stratosphoere.weebly.com/pact.html. abgerufen am 2016-04-26.

Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. pages 119–130. ACM, 2010. URL http://dl.acm.org/citation.cfm?id=1807148. abgerufen am 2016-04-25.

Bram Cohen. The BitTorrent protocol specification, 2013-10-11. URL http://www.bittorrent.org/beps/bep_0003.html. abgerufen am 2016-07-28.

Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. 2015. URL http://arxiv.org/abs/1506.08603. abgerufen am 2016-04-26.

K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. 3(1):63–75, 1985. URL http://dl.acm.org/citation.cfm?id=214456. abgerufen am 2016-06-26.

Stratosphere collaborative research project. Stratosphere » nephele execution engine, 2014a. URL http://robertmetzger.de/stratosphere/docs/pre-0.4/internals/nephele.html. abgerufen am 2016-05-31.

Stratosphere collaborative research project. Stratosphere » PACT runtime, 2014b. URL http://robertmetzger.de/stratosphere/docs/pre-0.4/internals/pact.html. abgerufen am 2016-05-31.

Stratosphere collaborative research project. Stratosphere » next generation big data analytics platform, 2016. URL http://stratosphere.eu/. abgerufen am 2016-04-09.

Danilo Assmann, Ralf Kalmar, and Dr. Teade Punter. Messen und bewerten von webapplikationen mit der goal/question/metric methode, 2002-12-23. URL http://publica.fraunhofer.de/eprints/urn_nbn_de_0011-n-176425.pdf. abgerufen am 2016-07-19.

Stephan Ewen. [FLINK-441] rename pact* and nephele* classes in the optimizer by StephanEwen · pull request #492 · apache/flink, 2015. URL https://github.com/apache/flink/pull/492. abgerufen am 2016-05-05.

Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. 5(11):1268–1279, 2012. URL http://dl.acm.org/citation.cfm?id=2350245. abgerufen am 2016-04-11.

Fabian Hüske. Apache flink: Juggling with bits and bytes, 2015-05-11. URL https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html. abgerufen am 2016-06-26.

Fabian Hüske. Apache flink: Peeking into apache flink's engine room, 2015-05-13. URL http://flink.apache.org/news/2015/03/13/peeking-into-Apache-Flinks-Engine-Room.html. abgerufen am 2016-06-26.

Fabian Hüske. Apache flink: Introducing stream windows in apache flink, 2015-12-04. URL https://flink.apache.org/news/2015/12/04/Introducing-windows.html. abgerufen am 2016-07-05.

Facebook. RocksDB, 2016. URL http://rocksdb.org/. abgerufen am 2016-07-12.

The Apache Software Foundation. The apache software foundation announces apache™ flink™ as a top-level project : The apache software foundation blog, 2015. URL https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces69. abgerufen am 2016-04-09.

The Apache Software Foundation. Apache flink 1.0.1 documentation: General architecture and process model, 2016a. URL https://ci.apache.org/projects/flink/flink-docs-release-1.0/internals/general_arch.html. abgerufen am 2016-04-26.

The Apache Software Foundation. Apache flink 1.1-SNAPSHOT documentation: Basic API concepts, 2016b. URL https://ci.apache.org/projects/flink/flink-docs-master/apis/common/. abgerufen am 2016-05-28.

The Apache Software Foundation. Apache flink 1.1-SNAPSHOT documentation: Concepts, 2016c. URL https://ci.apache.org/projects/flink/flink-docs-master/concepts/concepts.html. abgerufen am 2016-05-30.

The Apache Software Foundation. Apache flink 1.1-SNAPSHOT documentation: Jobs and scheduling, 2016d. URL https://ci.apache.org/projects/flink/

flink-docs-master/internals/job_scheduling.html. abgerufen am 2016-05-15.

The Apache Software Foundation. Apache flink: Features, 2016e. URL http://flink.apache.org/features.html. abgerufen am 2016-04-09.

The Apache Software Foundation. Apache flink: Scalable batch and stream data processing, 2016f. URL http://flink.apache.org/. abgerufen am 2016-04-09.

GroupLens. MovieLens 20m dataset, 2015-09-23. URL http://grouplens.org/datasets/movielens/20m/. abgerufen am 2016-12-02.

GroupLens Research. MovieLens, 2016. URL https://movielens.org/. abgerufen am 2016-12-02.

Josh Rosen. [SPARK-7081] faster sort-based shuffle path using binary processing cache-aware sort - ASF JIRA, 2015. URL https://issues.apache.org/jira/browse/SPARK-7081. abgerufen am 2016-08-16.

Kay Ousterhout. Shuffle internals - spark - apache software foundation, 2015-06-12. URL https://cwiki.apache.org/confluence/display/SPARK/Shuffle+Internals. abgerufen am 2016-07-28.

Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. 42(8):30–37, 2009-08. ISSN 0018-9162. doi: 10.1109/MC.2009.263.

Nathan Marz and James Warren. *Big data: principles and best practices of scalable real-time data systems*. Manning, 2015. ISBN 978-1-61729-034-3.

Prof. Dr. Hasso Plattner. Big data — enzyklopaedie der wirtschaftsinformatik, 2016-11-22. URL http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/daten-wissen/Datenmanagement/Datenmanagement--Konzepte-des/Big-Data. abgerufen am 2016-12-15.

Reynold Xin and Josh Rosen. Project tungsten: Bringing apache spark closer to bare metal, 2015-04-28. URL https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html. abgerufen am 2016-06-26.

Roi Reshef. Alternating least squares – data science made simpler, 2015-12-16. URL https://datasciencemadesimpler.wordpress.com/tag/alternating-least-squares/. abgerufen am 2016-12-07.

Henry Saputra. [FLINK-2815] [REFACTOR] remove pact from class and file names since it is no longer valid reference by hsaputra · pull request #1218 · apache/flink, 2015. URL https://github.com/apache/flink/pull/1218. abgerufen am 2016-05-10.

Statista. U.s. retail websites by visitors, 2016-07. URL https://www.statista.com/statistics/271450/monthly-unique-visitors-to-us-retail-websites/. abgerufen am 2016-11-22.

Stephan Ewen and Henry Saputra. Memory management (batch API) - apache flink - apache software foundation, 2015-05-20. URL https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=53741525. abgerufen am 2016-06-26.

Tathagata Das. Improved fault-tolerance and zero data loss in apache spark streaming, 2015-01-15. URL https://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streamin
html. abgerufen am 2016-08-08.

Tathagata Das, Matei Zaharia, and Patrick Wendell. Diving into apache spark streaming's execution model, 2015-07-30. URL https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html. abgerufen am 2016-08-08.

The Apache Software Foundation. The apache software foundation announces apache™ spark™ as a top-level project : The apache software foundation blog, 2014-02-27. URL https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces50. abgerufen am 2016-07-18.

The Apache Software Foundation. Apache flink 1.1-SNAPSHOT documentation: Savepoints, 2016a. URL https://ci.apache.org/projects/flink/flink-docs-master/apis/streaming/savepoints.html. abgerufen am 2016-07-05.

The Apache Software Foundation. Apache flink 1.1-SNAPSHOT documentation: Fault tolerance, 2016b. URL https://ci.apache.org/projects/flink/

flink-docs-master/apis/streaming/fault_tolerance.html. abgerufen am 2016-07-05.

The Apache Software Foundation. Apache flink 1.1-SNAPSHOT documentation: Windows, 2016c. URL https://ci.apache.org/projects/flink/flink-docs-master/apis/streaming/windows.html. abgerufen am 2016-07-05.

The Apache Software Foundation. Apache flink 1.1-SNAPSHOT documentation: Event time, 2016d. URL https://ci.apache.org/projects/flink/flink-docs-master/apis/streaming/event_time.html. abgerufen am 2016-07-05.

The Apache Software Foundation. Apache flink 1.1-SNAPSHOT documentation: Data streaming fault tolerance, 2016e. URL https://ci.apache.org/projects/flink/flink-docs-master/internals/stream_checkpointing.html. abgerufen am 2016-06-26.

The Apache Software Foundation. Apache flink 1.1-SNAPSHOT documentation: State backends, 2016f. URL https://ci.apache.org/projects/flink/flink-docs-master/apis/streaming/state_backends.html. abgerufen am 2016-07-12.

The Apache Software Foundation. Apache flink 1.1-SNAPSHOT documentation: Iterations, 2016g. URL https://ci.apache.org/projects/flink/flink-docs-master/apis/batch/iterations.html. abgerufen am 2016-07-12.

The Apache Software Foundation. Apache flink 1.1-SNAPSHOT documentation: Flink DataSet API programming guide, 2016h. URL https://ci.apache.org/projects/flink/flink-docs-master/apis/batch/index.html. abgerufen am 2016-07-16.

The Apache Software Foundation. Apache flink 1.2-SNAPSHOT documentation: Looking under the hood of pipelines, 2016i. URL https://ci.apache.org/projects/flink/flink-docs-master/dev/libs/ml/pipelines.html. abgerufen am 2016-09-19.

The Apache Software Foundation. Apache flink 1.2-SNAPSHOT documentation: FlinkML - machine learning for flink, 2016j. URL https://ci.apache.org/projects/

flink/flink-docs-master/dev/libs/ml/index.html. abgerufen am 2016-09-19.

The Apache Software Foundation. Apache flink 1.2-SNAPSHOT documentation: Gelly: Flink graph API, 2016k. URL https://ci.apache.org/projects/flink/flink-docs-master/dev/libs/gelly/index.html. abgerufen am 2016-09-20.

The Apache Software Foundation. Apache flink 1.2-SNAPSHOT documentation: Graph API, 2016l. URL https://ci.apache.org/projects/flink/flink-docs-master/dev/libs/gelly/graph_api.html. abgerufen am 2016-09-20.

The Apache Software Foundation. Apache flink 1.2-SNAPSHOT documentation: Iterative graph processing, 2016m. URL https://ci.apache.org/projects/flink/flink-docs-master/dev/libs/gelly/iterative_graph_processing.html. abgerufen am 2016-09-20.

The Apache Software Foundation. Apache calcite • dynamic data management framework, 2016n. URL https://calcite.apache.org/. abgerufen am 2016-09-20.

The Apache Software Foundation. Apache flink 1.1-SNAPSHOT documentation: Table API and SQL, 2016o. URL https://ci.apache.org/projects/flink/flink-docs-release-1.1/apis/table.html. abgerufen am 2016-09-20.

The Apache Software Foundation. Apache flink 1.1-SNAPSHOT documentation: Generating timestamps / watermarks, 2016p. URL https://ci.apache.org/projects/flink/flink-docs-master/apis/streaming/event_timestamps_watermarks.html. abgerufen am 2016-07-10.

The Apache Software Foundation. Community | apache spark, 2016q. URL https://spark.apache.org/community.html#history. abgerufen am 2016-07-18.

The Apache Software Foundation. Research | apache spark, 2016r. URL https://spark.apache.org/research.html. abgerufen am 2016-07-18.

The Apache Software Foundation. Spark SQL and DataFrames - spark 2.0.0 documentation, 2016s. URL http://spark.apache.org/docs/latest/sql-programming-guide.html. abgerufen am 2016-07-28.

The Apache Software Foundation. Spark streaming - spark 2.0.0 documentation, 2016t. URL http://spark.apache.org/docs/latest/streaming-programming-guide.html. abgerufen am 2016-08-08.

The Apache Software Foundation. Spark programming guide - spark 2.0.0 documentation, 2016u. URL http://spark.apache.org/docs/latest/programming-guide.html. abgerufen am 2016-08-15.

The Apache Software Foundation. Tuning - spark 2.0.0 documentation, 2016v. URL http://spark.apache.org/docs/latest/tuning.html. abgerufen am 2016-09-21.

Theodore Vasiloudis. FlinkML: Vision and roadmap - apache flink - apache software foundation, 2016-07-27. URL https://cwiki.apache.org/confluence/display/FLINK/FlinkML%3A+Vision+and+Roadmap. abgerufen am 2016-09-17.

Till Rohrmann. Computing recommendations at extreme scale with apache flink™ – data artisans, 2015-03-18. URL http://data-artisans.com/computing-recommendations-at-extreme-scale-with-apache-flink/. abgerufen am 2016-12-02.

Till Rohrmann. How to factorize a 700 GB matrix with apache flink™ – data artisans, 2015-03-30. URL http://data-artisans.com/how-to-factorize-a-700-gb-matrix-with-apache-flink/. abgerufen am 2016-12-02.

Jonas Traub, Tilmann Rabl, Fabian Hueske, Till Rohrmann, and Volker Markl. Die apache flink plattform zur parallelen analyse von datenströmen und stapeldaten, 2015. URL https://pdfs.semanticscholar.org/6558/ba3ae95bf29147e03741ff2df57afd735c16.pdf. abgerufen am 2016-04-11.

Kostas Tzoumas. Apache flink next-gen data analysis. abgerufen am 2016-05-05, 2014. URL https://www.sics.se/sites/default/files/pub/sics.se/kostas_tzumas_apache_flink_presentation.pdf.

Kostas Tzoumas, Stephan Ewen, and Robert Metzger. High-throughput, low-latency, and exactly-once stream processing with apache flink | data artisans, 2015. URL http://data-artisans.com/

high-throughput-low-latency-and-exactly-once-stream-processing-with- abgerufen am 2016-04-26.

Theodore Vasiloudis. FlinkML: Vision and roadmap - apache flink - apache software foundation, 2015. URL https://cwiki.apache.org/confluence/display/FLINK/FlinkML%3A+Vision+and+Roadmap. abgerufen am 2016-04-10.

Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Goal question metric paradigm. In *Encyclopedia of Software Engineering - 2 Volume Set*. John Wiley & Sons, Inc., 1994. ISBN 1-54004-8. URL https://www.cs.umd.edu/~basili/publications/technical/T89.pdf. abgerufen am 2016-07-19.

Daniel Warneke and Odej Kao. Nephele: efficient parallel data processing in the cloud. page 8. ACM, 2009. URL http://dl.acm.org/citation.cfm?id=1646476. abgerufen am 2016-04-26.

Matei Zaharia. *An architecture for fast and general data processing on large clusters*. Morgan & Claypool, 2016. URL https://books.google.com/books?hl=en&lr=&id=TNUlDAAAQBAJ&oi=fnd&pg=PP2&dq=%22However,+the+processing+capabilities+of+single+machines+have+not+kept%22+%22processing,+streaming+analysis+of+new+real-time+data+sources+is+required+to%22+%22systems+only+support+simple+one-pass+computations+(e.g.,%22+&ots=jSkwv4yQ_H&sig=1L7JInZHtLX1blxqmXu_Ho8tm88. abgerufen am 2016-07-19.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012a. URL http://dl.acm.org/citation.cfm?id=2228301. abgerufen am 2016-07-17.

Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*, 2012b. URL https://www.usenix.org/conference/hotcloud12/workshop-program/presentation/zaharia. abgerufen am 2016-07-19.

Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. pages 423–438. ACM

Press, 2013. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522737. URL http://dl.acm.org/citation.cfm?doid=2517349.2522737. abgerufen am 2016-07-19.

Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *International Conference on Algorithmic Applications in Management*, pages 337–348. Springer, 2008. URL http://link.springer.com/chapter/10.1007/978-3-540-68880-8_32. abgerufen am 2016-09-13.

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

Hamburg, 16. Dezember 2016 Timo Lange