

PROJEKTBERICHT

# HAMAUBE

KAI MARTINEN, FABIAN KOHLER, MERLIN KOGLIN,  
ARKADIJ DASCHKEWITSCH, RASMUS WARRELMANN,  
DAVID ZSCHOCKE

12. OKTOBER 2018



UNIVERSITÄT HAMBURG  
DEPARTMENT OF COMPUTER SCIENCE  
CHAIR OF DISTRIBUTED SYSTEMS AND INFORMATION  
SYSTEMS

---

BETREUT DURCH STEFFEN FRIEDRICH

# Kurzfassung

Diese Arbeit behandelt die Entwicklung und den Aufbau eines verteilten und skalierbaren Twitter-Klons, genannt *Hamaube*. Grundlegende Funktionen von Twitter sind in Hamaube umgesetzt, zusätzlich wurden Such- und Analysekomponenten integriert. Hamaube ist als Microservice-Architektur konzipiert und benutzt Apache Kafka als Kommunikationsgrundlage und Middleware. Hierdurch können Instanzen einzelner Services beliebig skalierbar in das laufende System integriert werden. Die verschiedenen Services sind technisch und thematisch gekapselt und kommunizieren über festgelegte APIs. Cassandra als verwendete Datenbank bietet mit einer eigenen SQL-ähnliche Abfragesprache ein vertrautes Interface, ist aber bei der Speicherung und den Abfrage nicht durch eine relationales Datenschema in der Performance gehemmt und kann somit beides schneller ausführen. Zudem ist Cassandra ebenso als Cluster skalierbar was die Datenhaltung vereinfacht. Für Suchszenarien innerhalb von Hamaube wird Elasticsearch verwendet, die Analyse und Auswertung von Daten im System wird mittels Spark durchgeführt.

Besonders im Fokus steht die Skalierbarkeit des Systems, welche durch eine spezielle Kommunikationsstruktur und Verteilung von Nachrichten erreicht wird. Um die entwickelte Architektur zu evaluieren, ist das ein Hamaube-Prototyp auf einer verteilten Infrastruktur implementiert. Dieser zeigt, dass die grundlegenden Anforderungen an einen Twitter-Klon erreicht werden und bietet sogar noch zusätzliche Funktionen.

# Inhaltsverzeichnis

<b>Kurzfassung</b>	<b>I</b>
<b>1 Überblick</b>	<b>1</b>
<b>2 Architektur</b>	<b>5</b>
2.1 Einführung . . . . .	5
2.2 Entwicklung der neuen Architektur . . . . .	6
<b>3 Umsetzung von Kommunikationsschemen über Kafka</b>	<b>18</b>
3.1 Grundlagen Kafka . . . . .	18
3.2 Realisierung der Kommunikationsschemen . . . . .	20
3.3 Microservice übergreifendes Datenformat . . . . .	22
<b>4 Webserver und Frontend</b>	<b>24</b>
4.1 Frontend . . . . .	24
4.2 Webserver . . . . .	25
4.3 Use Cases . . . . .	27
<b>5 Cassandra</b>	<b>29</b>
5.1 Datenverwaltung . . . . .	29
5.2 Cassandrareader . . . . .	31
5.3 Use Cases . . . . .	33
<b>6 Search Engine</b>	<b>38</b>
6.1 Zielsetzung . . . . .	38
6.2 Vorgehen . . . . .	38
6.3 Service Design . . . . .	39
6.4 Elasticsearch Konfiguration . . . . .	41
6.5 Search Service . . . . .	45

6.6	Visualisierung mit Kibana . . . . .	50
6.7	Fazit . . . . .	51
<b>7</b>	<b>Analytics</b>	<b>54</b>
7.1	Stream . . . . .	54
7.2	Architektur . . . . .	59
7.3	Skalierbarkeit . . . . .	60
7.4	Use-Cases . . . . .	62
7.5	Interessante Analyseergebnisse . . . . .	65
7.6	Spezielle Analyseergebnisse . . . . .	66
7.7	Genauere Schritte . . . . .	67
7.8	Verbesserungsvorschläge . . . . .	70
7.9	Probleme . . . . .	72
7.10	Daten: Input und Output . . . . .	75
7.11	Aufbau des Systems . . . . .	78
7.12	Phasen des Spark-Jobs . . . . .	79
7.13	Analysebilder . . . . .	80
7.14	Analytics Provisioning Service . . . . .	84
7.15	Darstellung im Frontend . . . . .	84
<b>8</b>	<b>Zusammenfassung</b>	<b>86</b>
<b>A</b>	<b>Cassandra</b>	<b>89</b>
<b>B</b>	<b>Big Table</b>	<b>95</b>
<b>C</b>	<b>Dynamo</b>	<b>102</b>
<b>D</b>	<b>MapReduce</b>	<b>108</b>
<b>E</b>	<b>Spark</b>	<b>115</b>
<b>F</b>	<b>Volltextsuche</b>	<b>127</b>

# Abbildungsverzeichnis

1.1	Übersicht über die verwendeten Tools und Mikroservices . . . . .	2
3.1	Beispiel für die Zuordnung von <i>Consumern</i> zu <i>Consumer Groups</i> und <i>Topics</i> . . . . .	19
3.2	Beispiel für die Umverteilung der Partitionen mit dem „roundrobin“ Verfahren: Zwischen Zustand 1 und Zustand 2 wurde der Consumer 2 entfernt. . . . .	21
4.1	Sequence Diagram: User erstellt neuen Tweet . . . . .	28
4.2	Sequence Diagram: User sucht nach einem Tweet (by text) .	28
5.1	Cassandra Schema . . . . .	30
5.2	Technologie Stack des cassandrareaders . . . . .	32
5.3	Architektur der Publisher und Subscriber am Beispiel Login .	33
6.1	Elasticsearch Service Architectur . . . . .	40
6.2	Kibana Analyseplattform . . . . .	51
7.1	Übersicht der Analytics Komponenten . . . . .	55
7.2	Spark stream processing . . . . .	57
7.3	Sream Analytics: Beispiel - Anzahl von Hashtags pro Stunde .	58
7.4	Spark und Cassandra verbunden mit Flaschenhals . . . . .	62
7.5	Spark und Cassandra durcheinander, aber parallel verbunden	62
7.6	Spark und Cassandra mittels Data Locality verbunden . . . . .	63
7.7	BTS vor den Vereinten Nationen Quelle: <a href="https://www.youtube.com/watch?v=ZhJ-LAQ6e_Y">https://www.youtube.com/watch?v=ZhJ-LAQ6e_Y</a> . . . . .	65
7.8	Anzahl Tweets . . . . .	80
7.9	Top 10 Stunden . . . . .	81
7.10	Tweets pro Tag (mit Sampling) . . . . .	81
7.11	Top 10 Wörter . . . . .	82

7.12	Top 10 Tweeter (inkl. Porno-Spam) . . . . .	82
7.13	Top 10 Hashtags . . . . .	83
7.14	Live Analytics im Frontend . . . . .	85
A.1	Beispiel Daten Modell . . . . .	90
A.2	Consistent-Hashing Ring . . . . .	91
A.3	CQL Mapping . . . . .	93
B.1	Tabellen Beispiel . . . . .	96
B.2	Zugriffs Beispiel mit der BigTable API . . . . .	97
B.3	Percormance Übersicht . . . . .	100
C.1	Beispiel für konsistentes Hashing. Die Daten sollen auf vier Knoten verteilt werden, wobei die Knoten gleichmäßig auf dem Kreis platziert sind. . . . .	103
D.1	MapReduce Modell Übersicht . . . . .	108
D.2	Ausführung eines Map tasks . . . . .	111
D.3	Übersicht der gesamten MapReduce Ausführungslogik . . . . .	112
E.1	Erstellung RDD aus HDFS [13] . . . . .	117
E.2	Spark Stages [13] . . . . .	123
E.3	Spark Architektur [13] . . . . .	123
E.4	Sparks DStreams [15] . . . . .	125
F.1	Häufigkeitsmatrix [12] . . . . .	130
F.2	Häufigkeitslisten [12] . . . . .	131
F.3	TF-IDF-Matrix . . . . .	136
F.4	Euklidische Distanz [12] . . . . .	137
F.5	Winkelbasierten Ansatz [12] . . . . .	138



# Kapitel 1

## Überblick

Die Grafik zeigt eine Übersicht über die Architektur der Hamaube. Sie verwendet mit Kafka, Cassandra, Elasticsearch und Spark Tools aus dem NoSQL-Bereich. Verbunden werden diese über Mikroservices zu einem funktionierenden Ganzen.

### 1.0.1 Tools

Zentral ist Apache Kafka als verteilter Messagebroker. Über ihn kommunizieren die Mikroservices miteinander. Gleichzeitig ermöglicht das Publish-Subscribe-Modell von Kafka hier die Erweiterbarkeit und Anpassbarkeit des Systems. Kafka kann sowohl selbst skaliert werden, d.h. mit zunehmender Nachrichtenmenge und -geschwindigkeit werden weitere Kafka-Knoten gebraucht. Kafka unterstützt auch die Skalierung der anderen Komponenten, da die Nachrichten bei der Zustellung verteilt werden.

Für die persistente Datenhaltung verwenden wir eine NoSQL-Datenbank, nämlich Apache Cassandra. Cassandra unterstützt wie alle anderen verwendeten Tools den Scale-Out-Ansatz zur Skalierung. Damit lassen sich sowohl hohe Datengeschwindigkeiten, als auch wachsende Datenmengen verarbeiten.

Elasticsearch wird als Suchanwendung verwendet. Sie stellt eine ebenfalls verteilte Volltextsuche bereit. So ist es möglich, auch in riesigen Datenmengen die gewünschten Tweets zu finden. Dies erfolgt so performant, dass die Suche in Echtzeit Suchvorschläge machen kann, sogar während der User

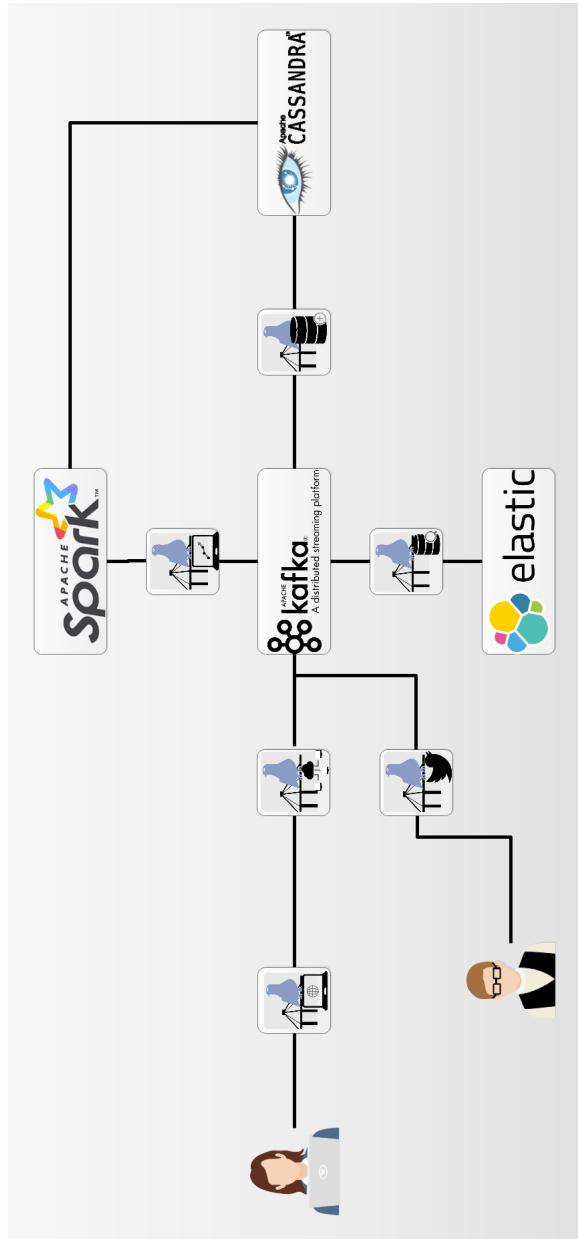


Abbildung 1.1: Übersicht über die verwendeten Tools und Mikroservices

---

noch tippt.

Die letzte eingesetzte Komponente ist Apache Spark. Damit werden in der Hamaube sowohl die Echtzeitstatistiken auf den hereinkommenden Tweets erzeugt, als auch Analysen über die Gesamtmenge der Tweets. Diese Komponente ist ebenfalls skalierbar und arbeitet mit unserer verteilten Datenbank Cassandra zusammen.

### 1.0.2 Tweets

Tweets kommen in der Hamaube aus zwei Quellen: zum einen können in dem vollfunktionsfähigen Twitter-Klon Tweets im Frontend selbst abgesetzt werden. Zum zweiten werden Tweets über die Twitter-API angefordert und integriert.

### 1.0.3 Mikroservices

Die skalierbaren Tools werden durch Mikroservices verbunden. Diese Mikroservices stellen die Anwendungslogik zur Verfügung. Die entwickelten Mikroservices sind:

- Der Cassandrareader. Er nimmt Anfragen von den Webservern entgegen, fordert die entsprechenden Daten aus Cassandra an und schreibt Tweets und User in die verteilte Datenbank.
- Der Webserver. Er nimmt Anfragen des Frontends entgegen, ist zuständig für die Benutzerauthentifizierung und leitet die Anfragen an die zuständigen Mikroservices weiter. Von denen bekommt er die Antworten und leitet sie an das Frontend weiter. Das Frontend wird vom ihm an den Browser ausgeliefert.
- Der Twitterstream. Er fragt kontinuierlich die Twitter-API nach Tweets an und streamt diese in die Hamaube.
- Der Suchservice. Er übernimmt die Kommunikation mit Elasticsearch. Er bekommt Suchanfragen vom Webserver, leitet diese weiter und gibt die Antworten über Kafka zurück. Selbst während des Tippens fragt er für eine Auto vervollständigung bei Elasticsearch an. Er kümmert sich auch darum, die Tweets kontinuierlich in Elasticsearch zu laden.

- 
- Der Analytics-Provisioning-Service. Er übernimmt die Rolle des Suchservices für Spark. Er erzeugt mit Hilfe von Spark die Echtzeitstatistiken und bereitet die Tweets für die Batchanalyse auf und speichert sie in Cassandra. Die Batch-Analysen werden als Cron-Jobs gestartet, während die Echtzeitstatistiken dauerhaft in Spark laufen.

Diese Architektur löst die traditionelle LAMP- (Linux, Apache, MySQL, PHP) Architektur für Webanwendungen ab und wird SMACK genannt. Für Kafka haben wir uns wegen seiner Persistenzeigenschaft entschieden. Welche Ziele unsere Architektur noch hat und wie diese erreicht werden, wird im nun folgenden Kapitel Architektur beleuchtet.

Getestet wurde die Hamaube auf leistungsstarken virtuellen Maschinen. Die Tools und ihre Mikroservices haben jeweils eigene virtuelle Maschinen. Dadurch kann die Kompatibilität und das Zusammenspiel der Mikroservices zu einem funktionsfähigen Ganzen getestet werden. Zudem wurden auch die Entwicklungsnotebooks für die Bereitstellung der Dienste genutzt. Ein Nutzen dieser Architektur ist es, dass dies transparent erfolgt. Dadurch konnte in gewissem Ausmaß auch die Ausfalltoleranz geprüft werden.

# Kapitel 2

## Architektur

### 2.1 Einführung

Ziel des Projekts ist es einen Twitter-Klon zu entwickeln. Als Mikroblogging-Dienst ist die zentrale Funktionalität:

- Senden von Tweets
- Finden von Tweets (Anzeigen der Tweets in der eigenen Timeline, Suchen nach Schlüsselwörtern/Hashtags)

Zusätzlich ist selbstverständlich eine Benutzerverwaltung nötig. Ebenso interessant sind Analytics-Funktionalitäten.

In diesem Projekt liegt der Fokus auf der Entwicklung einer Architektur, die besondere Nicht-Funktionale Anforderungen erfüllt. Diese ergeben sich aus den Anforderungen an das reale Twitter:

- Skalierbarkeit
- Erweiterbarkeit um zusätzliche Funktionalität.

Geprägt ist Twitter durch enorme Datengeschwindigkeit (Velocity in Gartners Vs). Jede Sekunde werden im Durchschnitt 6000 Tweets weltweit abgesetzt. Diese Zahlen variieren zudem stark: Der Rekord liegt bei 140.000 Tweets pro Sekunde<sup>1</sup>. Zudem werden viele Features von Twitter nach und nach eingeführt, ohne das Twitter in Wartungspausen geht (gehen musste).

<sup>1</sup><https://www.brandwatch.com/de/blog/twitter-statistiken/> <http://www.gegen-den-strom.org/twitter/>

Diesen Herausforderungen sind herkömmliche Webarchitekturen nicht gewachsen. Dieses Projekt entwickelt einen Twitter-Clon mit den Technologien und einer Architektur, die diesen Anforderungen im Prinzip gewachsen ist.

### 2.2 Entwicklung der neuen Architektur

Ausgangspunkt der Entwicklung der neuen Architektur ist eine einfache Client-Server-Architektur. Es gibt in ihr einen Server mit viel Rechenleistung sowie viele Clients. Die Clients verbinden sich mit dem Server, der sie parallel bedient. Die Parallelisierung erfolgt also auf Thread-/Prozessebene. Der nächste Schritt ist es, die Datenhaltung in eine Datenbank auszulagern. Es gibt jetzt drei Komponenten: die Clients, der Webserver und der Datenbankserver. Wenn die Leistung des Serversystems nicht ausreicht gibt es unterschiedliche Möglichkeiten für mehr Leistung zu sorgen:

- Die einzelnen Server mit mehr Leistung auszustatten
- Mehr Webserver einzuführen

Möglichkeit 1 würde ein Aufrüsten der Server mit schnelleren Prozessoren, mehr Arbeitsspeicher, schnelleren Festplatten etc. bedeuten. Möglichkeit 2 ist schon etwas komplizierter. Zwar werden die Daten zentral vom Datenbankserver gehalten, trotzdem können nicht einfach mehr Webserver eingeführt werden. Die Clients müssen noch auf die Webserver verteilt werden. Diese Aufgabe kann wieder auf zwei Arten erfolgen: Im Netzwerkprotokoll (z.B: über Anycast) oder durch eine zusätzliche Komponente im Server, nämlich dem Load-Balancer. An ihn kommen alle Anfragen an die Webserver und der Loadbalancer entscheidet mit mehr oder weniger komplexe Techniken welcher Webserver genutzt werden soll und leitet den anfragenden Client an diesen Webserver weiter. Dabei achtet diese Komponente z.B. darauf, dass bereits stark ausgelastete Webserver keine neuen Clients mehr zugewiesen bekommen, sondern nur Webserver mit aktuell niedriger Auslastung.

Die Parallelisierung erfolgt in dieser Architektur bereits auf vielen Ebenen: Im Loadbalancer, in den Webservern und in der Datenbank. Die Datenbank führt dafür ausgefeilte Mechanismen zur Verwaltung der parallelen Anfragen und zur Datenkonsistenz aus.

Diese Architektur wird häufig mit dem LAMP Softwarestack implementiert: Linux als Betriebssystem der Server, Apache als Webserver, MySQL als Datenbank und PHP als Serverprogrammiersprache.

So weit eine erste Einführung in die Verteilung von Webanwendungen mit herkömmlichen Architekturen. Nun wollen wir die Art der Verteilung analysieren um darzustellen, weshalb eine andere Architektur für noch mehr Verteilung nötig ist.

### 2.2.1 Analyse der Verteilung

Es kommen zwei Techniken der Verteilung zu Einsatz

- Erhöhung der Rechenleistung der einzelnen Komponenten
- Loadbalancing

Die erste Technik wird auch Scale-Up genannt. Durch potentiere Hardware kann ein Leistungszuwachs erzielt werden. Allerdings - und das ist ein zentrales Problem, dass unsere neue Architektur lösen wird - sind dieser Möglichkeit vergleichsweise enge Grenzen gesetzt. Der Prozessorgeschwindigkeit sind durch die Eigenschaften des verwendeten Silizium und der Wärmeentwicklung enge Grenzen gesetzt. Über 5 GHz gehen die Taktraten nicht mehr hinaus. Damit ist bereits bei einem Faktor von ca. 2,5 der Geschwindigkeitserhöhung hier ein Ende gesetzt. Von 2 GHz geht es bis maximal 5 GHz hoch (und selbst 5 GHz erreichen die wenigsten Prozessoren). Ein Leistungszuwachs kann jetzt nur noch durch mehr Prozessorkerne und mehr Prozessoren erreicht werden. Durch mehrere Sockel in den Server (leicht bis zu 4 Stück) und mehr Kernen in den Prozessoren (leicht bis zu 20 Stück) lässt sich hier bei sehr guter Parallelisierbarkeit der Anwendung<sup>2</sup> theoretisch eine Erhöhung um das 80-fache erreichen.

Hier ist nun die Grenze für das Scale-Up erreicht. Herkömmliche Server lassen sich nun kaum mehr beschleunigen. Sobald also eine Komponente in unserer Architektur mehr Leistung benötigt weil mehr Client-Anfragen kommen, als sie bearbeiten kann, müssen wir etwas in der Architektur ändern. Die Ausnahme sind hier die Webserver. In unserer Architektur können wir

---

<sup>2</sup>Amdahl's Law zeigt, dass sich durch mehr Hardware die Geschwindigkeit nicht linear erhöht

---

bereits mehr Webserver hinzufügen und die Last auf mehr Rechner verteilen, anstatt auf schnellere Rechner. Das ist ein Grundgedanke unserer neuen Architektur. Im Gegensatz zum Scale-Up wird dieser Ansatz Scale-Out genannt.

Die Komponente, die dementsprechend als erstes in unserer Architektur geändert werden muss, ist die Datenbank. Damit beschäftigen wir uns im nächsten Abschnitt.

### 2.2.2 Die Verteilung der Datenbank

Wir müssen also unsere Datenbank beschleunigen. Wir nehmen an, dass wir kein Scale-Up mehr durchführen können oder wollen.

Eine Möglichkeit wäre noch die Nutzung von sehr spezialisierter Datenbankhardware. Diese hat sehr große Investitionskosten und legt einen auf eine spezielle Datenbanksoftware fest. Wir möchten diese deshalb nicht betrachten.

Ein erster Ansatz wäre es, das Load-Balancing der Webserver zu wiederholen. Mehr Datenbankserver und eine Komponente, die die Anfragen an einen Datenbankserver mit genug verfügbarer Kapazität weiterleitet. Allerdings geht das nicht mehr so einfach wie bei den Webservern. Die Webserver waren untereinander unabhängig. Die Datenbank ist das nicht. Man kann nicht einfach einen Datenbankserver nehmen und dort Daten ändern. Wird der Nutzer nun bei seiner nächsten Anfrage an einen anderen Datenbankserver geleitet, sind die Änderungen dort nicht vorgenommen. Das ist also keine Alternative.

Aber diesen Ansatz kann man noch verfeinern und schlussendlich gewinnbringend einsetzen. Das Problem war, dass die geänderten Daten nur auf einem Server verändert wurden und der User plötzlich auf einen anderen Datenbankserver landete. Dann muss der Nutzer also immer auf dem gleichen Datenbankserver landen. Eine einfache Möglichkeit der Verteilung wäre es also, die Nutzer fest auf die Datenbankserver zu verteilen. Beispielsweise werden alle Nutzer mit Nachnamen, die mit A-K beginnen auf den ersten Datenbankserver zugewiesen, die anderen auf den zweiten Datenbankserver. In Verwaltungseinrichtungen wird so etwas sogar physisch betrieben. Wenn der Nutzer nun ein zweites Mal kommt, wird er wieder an den gleichen Datenbankserver weitergeleitet und seine Daten sind dort aktuell. Diese

Aufteilung muss dann aber in der Applikationslogik fest eingebaut werden. Besser wäre es, wenn das automatisch passieren würde.

Der andere Ansatz wäre es nicht den Benutzer immer auf den gleichen Datenbankserver weiterzuleiten, sondern die Datenbankserver alle aktuell zu halten. Dann wäre es wieder egal, welchen Datenbankserver er beim nächsten Besuch nutzt. Allerdings sieht man sofort, dass dann sehr viel Netzwerkverkehr nötig wäre um die Daten auf allen Server aktuell zu halten. Es ist also zweifelhaft, ob das so viel schneller wäre.

Das wiederum hängt vom Nutzungsprofil ab. Wenn Daten z.B. nur selten geändert werden, aber häufig gelesen, dann ergibt das wieder Sinn. Der Overhead für das Aktuellhalten der Daten wäre dann sehr gering und der Leseoperationen würden skalieren.

### 2.2.3 Ein erste Blick auf die neue Architektur

Im vorigen Abschnitt haben wir gesehen, dass es nötig wird die Datenbank verteilt auszulegen. Die eine Möglichkeit war die händische Aufteilung der Anfragen auf verschiedene, unabhängige Datenbankserver. Diese Möglichkeit ist aufwendig. Eine andere Möglichkeit war das Spiegeln auf mehrere Datenbankserver. Das funktioniert nur dann, wenn die Leseoperationen die Schreiboperationen weit übertreffen und die Schreiboperationen nur eine Last erzeugen, die ein einzelner Datenbankserver bearbeiten kann. Nun betrachten wir eine dritte Möglichkeit: die Verwendung verteilter Datenbanken. Damit werden Datenbankprogramme bezeichnet, die für den Einsatz auf mehreren Servern gebaut wurden. Sie sind eine Spielart der so genannten NoSQL-Datenbanken. Zur Skalierung verwenden sie den Scale-Out-Ansatz statt des Scale-Up-Ansatzes. D.h. sie sind so designed, dass man mehr Leistungsfähigkeit durch das Einbinden weiterer Server erreichen kann. Diese Server bestehen aus herkömmlicher Serverhardware. Es ist keine spezialisierte Hardware erforderlich. Die Koordinierung erfolgt über Netzwerkprotokolle. Damit wandert die Skalierbarkeit in die Software. Die Realisierung wird weiter unten in einem gesonderten Kapitel beschrieben. In der herkömmlichen Architektur wird nun die SQL-Datenbank durch eine verteilte Datenbank ersetzt. Jetzt lässt sich die benötigte Leistung durch die Verteilung der Datenbank auf genügend Server erreichen. Diese Komponente ist zusätzlich noch skalierbar, d.h. wenn mehr Leistung benötigt wird,

können weitere Server hinzugefügt werden.

Das ist ein Prinzip unserer neuen Architektur: Nicht skalierbare Softwarekomponenten werden durch skalierbare Softwarekomponenten ausgetauscht. Wir haben mit der Datenbank begonnen. Jetzt ist es Zeit die benötigten Komponenten für unseren Twitter-Klon zu ermitteln. Im nächsten Schritt setzen wir für die Komponenten Software ein, die skalierbar ist.

### 2.2.4 Benötigte Komponenten

Unser Twitter-Klon soll die folgende Funktionalität haben:

- Nutzerverwaltung: Benutzerkonto mit Login-Funktionalität
- Kernfunktionalität: Tweets senden und lesen
- Tweets entdecken: Tweets suchen und Empfehlungen bekommen
- Statistische Auswertungen der Tweets

Es gibt offensichtlich eine Frontend-Komponente. Das Backend besteht nun aus verschiedenen Komponenten. Eine Komponente für die Nutzerverwaltung und eine Komponente für die Kernfunktionalität. Sie greifen auf die Daten in der verteilten Datenbank zu. Da ein eindeutiger Schlüssel bekannt ist, unterscheidet sich der Zugriff nur in Details vom der Nutzung einer normalen SQL-Datenbank. Die verteilte Datenbank ist in der Lage die Datens Mengen zu handeln und die benötigten Informationen zurückzuliefern.

Wir verwenden als verteilte Datenbank Cassandra.

Das ist anders beim Suchen der Tweets: Wir gehen davon aus, dass sich enorme Mengen von Tweets in der Datenbank befinden. Ein naives Durchsuchen wäre sehr teuer und wird deshalb nicht von der verteilten Datenbank angeboten. Wir brauchen also eine Komponente für das Durchsuchen der Tweets. Für sie definiert man Kriterien, nach denen gesucht werden kann, dann liest sie alle Tweets ein und erstellt die entsprechenden verteilten Indexstrukturen. Ab dann ist die Komponente einsatzbereit und kann dann ebenfalls verteilt unter Zuhilfenahme der vorher erstellten Indexstrukturen die Suchanfragen beantworten. Zudem hat sie Methoden des Rankings aus dem Dokument Retrieval implementiert um die Suchergebnisse gewerten zurückzugeben.

Die nächste Funktionalität, nämlich die statistische Auswertung der Tweets benötigt wiederum eine eigene Komponente, die wiederum verteilt arbeiten muss. Für die statistischen Auswertungen werden in Regelfall alle Tweets untersucht, damit handelt es sich wieder um enorme Datenmengen. Diese können nicht auf einen einzelnen Server geladen werden und dort ausgewertet werden. Zumindest nehmen wir das bei der Entwicklung der Architektur an. Diese Komponente muss also verteilte Auswertung unterstützen. Wie das funktioniert, wird ebenso später beschrieben.

Die Komponenten für die Suche existieren bereits und auch die Komponente für die verteilte Auswertung existiert schon: ElasticSearch für die Suche und Empfehlung und Spark für die verteilte Auswertung.

Wir fassen nochmal zusammen:

- Kernfunktionalität: Tweets rein wie raus
- Suche und Empfehlung: On-Demand-Berechnung von Antworten auf Suchanfragen unter Nutzung vorbereiter Datenstrukturen
- Analytics: Vorberechnung der Analyseergebnisse und dann einfache Rückgabe dieser Ergebnisse auf Anfrage
- Realtime-Analytics: Analyse von Stromdaten.

Es reicht nicht aus, skalierbare, verteilte Komponenten zu benutzen. Zwischen den Komponenten werden enorme Datenmengen ausgetauscht. Damit hier keine Flaschenhälse entstehen, müssen auch die Verbindungen zwischen den Komponenten verteilt sein, wenn sie die ganzen Daten auf einmal verarbeiten. Alternativ müssen die Komponenten die Daten durchgehend während ihre Entstehung laden.

Es lohnt sich den Datenfluss in unserer Twitter-Klon-Architektur sich genauer anzuschauen.

### 2.2.5 Datenfluss: Stromdatenverarbeitung

Um zu verdeutlichen, wie der Datenfluss aussieht, betrachten wir kurz unterschiedliche Webanwendungen mit hohem Leistungsanforderungen

- Nachrichtenportal: Hier stehen viele Artikel zum Lesen bereit. Die Anzahl Lesezugriffe (Abrufe von Artikeln) übersteigt die Anzahl Schreib-

zugriffe (Einstellen von neuen Artikeln) deutlich. Es geht also im Wesentlichen darum, immer wieder neue Artikel sehr häufig auszuliefern.

- Video-On-Demand: Ähnlich wie das Nachrichtenportal, nur dass die auszuliefernden Einheiten viel größer sind (Videos vs. Texte). Dafür müssen nicht so viele Einheiten in kurzer Zeit ausgeliefert werden. Auch hier steht die Auslieferung im Vordergrund.

Das Anforderungsprofil eines Mikroblogging-Dienstes ist etwas anders. Zum einen sind die einzelnen Einheiten sehr klein (Tweets), zum anderen werden diese von den Nutzern selbst generiert. Und zwar sehr viele in sehr kurzer Zeit. Die Tweets besonders populäre Nutzer müssen ebenfalls millionfach ausgeliefert werden. Der Fokus liegt aber längst nicht mehr auf der Auslieferung von Daten. Vielmehr geht es um einen ständigen Strom von Daten in das System ebenso wie aus dem System heraus. Es geht also um Stromdatenverarbeitung.

Dieses Anforderungsprofil harmoniert mit dem Parallelisierungskonzept unserer Architektur. Die einzelnen Komponenten sind verteilt ausgelegt und verteilen die Arbeit auf mehrere einzelne Server. Das geht nun recht einfach. Die einzelnen Tweets können als (nahezu) unabhängig voneinander betrachtet werden. Sie können insbesondere unabhängig voneinander gespeichert werden. Damit können die Tweets zur Speicherung an einen beliebigen Datenbankserver weitergeleitet werden.

Es gibt Ausnahmen von der Unabhängigkeit. Bei einer Folge von Tweets kann es sich um ein Gespräch handeln, d.h. Tweets antworten aufeinander. Es ist wünschenswert, dass solche Folgen auch vollständig angezeigt werden (können) und nicht plötzlich ein Tweet im Gespräch fehlt. Über die Realisierung solcher Anforderungen findet sich unten mehr.

Neben der Skalierbarkeit hat unsere Architektur auch weitere Ziele. Es stellt sich noch die Frage, wie die einzelnen Komponenten verbunden werden und was sie funktional machen.

### 2.2.6 Microservice-Architektur

Eine ständige Herausforderung bei der Entwicklung von großen Systemen ist die Aufteilung in kleinere Einheiten, die dann besser gehandelt werden können. Schließlich werden diese Systeme meist von vielen Entwicklern geschrieben,

die häufig auch noch an verschiedenen Standorten sitzen. In unserem Fall werden sogar einige unterschiedliche Technologien im Backend genutzt.

Der Ansatz der Aufteilung ist es, einzelne Komponenten mit jeweils einer Aufgabe zu entwickeln. Diese Komponenten sollen unabhängig voneinander entwickelt werden können. Offensichtlich hängen die einzelnen Komponenten in unserem Twitter-Clon stark voneinander ab. Alle brauchen zum Beispiel Zugriff auf die Tweets. Das ist kein Widerspruch zum Microservice-Ansatz, es müssen nur die Schnittstellen vorher erstellt werden (und später eingehalten werden).

Dabei ist es wichtig, dass die Schnittstellen nicht eine spezielle Technologie (insbesondere eine spezielle Programmiersprache) voraussetzen. In unserer Architektur ist das Austauschformat an den Schnittstellen meist JSON. Die Tweets sind z.B. JSON-Dateien bei ihrer Erzeugung. Das andere Austauschformat sind die Datenbanktabellen. Für die Datenbank existieren Connectoren in diversen Programmiersprachen. Die einzelnen Spalten sind zum einen universale Datentypen (Ints, Strings, ...) zum anderen Verschachtelungen von den universalen Datentypen.

Hier soll nicht verschwiegen werden, dass dieses Konzept der definierten Schnittstellen vor allem ein Ideal darstellt. Von ihm wird (und muss) manchmal davon abgewichen werden. Beispielsweise zeigt sich später, dass für die eine Komponente zusätzliche Felder im JSON notwendig sind. Es ist nicht möglich, dass von Anfang alles vollständig zu wissen. Zudem sind die Komponenten unabhängig voneinander. Und nicht alle Komponenten stehen in der Kontrolle der Entwicklungsteams. Werden Daten von außen verwendet, in unserem Fall benutzen wird Daten von Twitter, so können die das Format festlegen und auch ohne Nachfrage ändern. In diesem Projekt ist das auch mehrmals passiert. Zwar lässt sich das häufig mit Wrappern, Schema Matching oder auch durch Anpassung an die Änderungen von außen lösen. Allerdings darf der Aufwand dafür nicht unterschätzt werden.

Der zweite Punkt, der hier kritisch angemerkt werden soll, ist dass diese Architektur auch nicht vor den typischen Data Cleaning-Aufgaben schützt. Eine Definition des Austauschformats bedeutet nicht, dass a. alle Felder Werte haben, b. das die Werte korrekt sind und c. dass das Format eingehalten wird. Bei den Twitterdaten heißt das z.B. das bei weitem nicht alle Tweets Geodaten haben. Trotzdem sind dafür einige Felder vorgesehen. Auch die-

ses Problem lässt sich meist lösen. Es zeigte sich aber, dass in unserem Fall einer der Connectoren zur verteilten Datenbank grobe Schwierigkeiten mit Null-Werten hatte.

Die Microservice-Architektur wird bei uns mit weiteren Konzepten erweitert, wodurch ihr Nutzen nochmal deutlicher wird. Ohne diese Erweiterungen ist die Mikroservicearchitektur vor allem eine Empfehlung an die Systemdesigner Komponenten möglichst unabhängig voneinander zu designen und dadurch eine niedrige Kopplung zu erreichen. Das bedeutet auch, programmiersprachenunabhängige Schnittstellen zu bevorzugen.

Wir erweitern das Microservice-Konzept um eine Publish-Subscribe-Architektur und später durch ein Denken in Ereignissen. Das führt zu einer guten Erweiterbarkeit.

### 2.2.7 Die Publish-Subscribe-Architektur

Zur Übertragung der Datenströme zwischen den Komponenten nutzen wir Kafka. Es handelt sich dabei um eine verteilte Streaming Plattform. Diese Software kümmert sich um die Umsetzung der Datenströme. Sie ist selbst wieder natürlich skalierbar, und zwar scale-out skalierbar. Im Kern geht es um die Weiterleitung von Nachrichten an alle interessierten Komponenten. Ein Datenstrom besteht also aus einer Folge von Nachrichten.

Im Kern von Kafka steht das Publish-Subscribe-Modell. Es werden Kanäle angelegt, so genannte Topics. Eine Komponente kann sich nun bei Kafka für diesen Kanal anmelden ("subscribe") und bekommt ab dann alle Nachrichten von Kafka zugestellt. Eine Komponente kann dann auch Nachrichten an den Kanal schicken ("publish") und Kafka kümmert sich dann um die Verteilung an alle Subscriber.

Dies ist eine der Stellen, bei der leicht ein Flaschenhals entstehen kann. Die verteilte Datenbank kann sich nicht direkt mit Kafka verbinden. Stattdessen muss eine Komponente dazwischen geschaltet werden. Diese Komponente muss wiederum verteilt arbeiten. Ansonsten werden alle Tweets über einen Server geleitet. Bis zu einer gewissen Geschwindigkeit (Tweets pro Zeiteinheit) mag das gut gehen, aber unsere Architektur soll ja gerade so designed sein, dass man in diesem Fall einfach weitere Server hinzufügen kann.

Ein weiterer Fall sind die Webserver. Wenn sie für den Nutzer Tweets aus der Datenbank anfordern, dann läuft das wieder über ein solches Topic in

Kafka. Das Problem entsteht dann, wenn sich einfach alle Webserver auf das Topic subscriben. Denn dann erhalten alle Webserver alle Tweets, die von Webservern angefordert worden. Sie sollen aber nur die Tweets bekommen, die sie auch selbst angefordert haben. Kafka löst das Problem mit so genannten Consumer Groups. Wie das genau funktioniert, wird weiter unten beschrieben.

Nun wollen wir uns noch anschauen, wie unsere Architektur Erweiterbarkeit unterstützt.

### 2.2.8 Erweiterbarkeit des Systems

Wir schauen uns das am Beispiel der Realtime-Analytics an. Ihre Aufgabe ist es einfache statistische Auswertungen zu den akutell abgesendeten Tweets zu liefern, z.B. die Top 10 Hashtags der aktuellen Stunde. Es gibt jetzt verschiedene Möglichkeiten an die dafür benötigten Tweets zu kommen:

- die Tweets aus der Datenbank lesen
- die Tweets vom Webserver nicht nur in die Datenbank schreiben lassen, sondern auch an die Realtime-Analytics-Komponente zu schicken

Beide Verfahren haben erhebliche Nachteile. Das erste Verfahren würde in sehr vielen Datenbankanfragen münden, wenn man die Statistiken z.B. jede Sekunde aktualisieren möchte. Das zweite Verfahren würde eine Änderung des Webservers benötigen.

Beide Verfahren sind also nicht geeignet genug. Unsere Architektur kann das Problem hier komfortabel lösen. Es ist ähnlich dem zweiten Verfahren. Allerdings sendet nicht der Webserver die Daten an die Realtime-Analytics-Komponente. Stattdessen subscribt sie sich die Realtime-Analytics-Komponente auf das Topic, auf dem der Webserver die Tweets an die Datenbank sendet. Und schon kümmert sich Kafka darum, dass die Tweets auch dieser Komponente in Echtzeit zur Verfügung gestellt werden. Es müssen weder Änderungen am Webserver vorgenommen werden, noch an der Datenbank. Ebenso einfach könnte man die Realtime-Analytics-Komponente auch wieder entfernen. Man meldet sie einfach von Kafka ab.

Hier sieht man, dass eine kleine Änderung im Verständnis der Komponenten vorteilhaft ist: Statt ein Topic für eine Aufgabe zu benutzen, benutzt man eine Topic für Ereignis in der Realwelt.

### 2.2.9 Denken in Ereignissen

Bleiben wir im Beispiel des Hinzufügens der Realtime-Analytics-Komponente. Möglicherweise hätte man das Topic, in dem der Webserver seine Tweets an die Datenbank schickt auch so benannt und gedacht: WebserverTweetsToDatabase. Sobald jetzt aber die Realtime-Analytics-Komponente hinzugefügt werden soll, passt der Name nicht mehr. Eine Erweiterung auf WebserverTweetsToDatabaseAndRealtimeAnalytics löst das Problem nicht. Sobald man die nächste Komponente hinzufügen will, muss man wieder das Topic ändern. Also ziehen wir es zusammen zu WebserverTweetsToX.

Unsere Architektur schlägt vor nicht in den verbundenen Komponenten zu denken, sondern in Realweltereignissen. Was wäre nämlich dann, wenn Tweets nicht nur über Webserver gesendet werden könnten, sondern auch über Bots? Dann würde man schnell bei XTweetsToY als Beschreibung landen. Und das zugehörige Ereignis in der Realwelt wäre: Der Nutzer sendet einen Tweet ab. Und so sollte nach dieser Regel in der Architektur das Topic auch UserIssuesTweet genannt werden.

### 2.2.10 Ausfallsicherheit

Zusätzlich zur Skalierbarkeit der Verteilung können die Komponenten eine gewisse Ausfalltoleranz sicherstellen. Dadurch dass die Daten auf unterschiedliche Server verteilt werden, können die Daten auch jeweils an mehr als einen Server geschickt werden. Bei einer Datenbank können die Daten dann auf mehreren Servern gespeichert werden. Dadurch entstehen lokale Kopien der Daten. Diese Kopien können dann beim Ausfall die Rolle des ausgestorbenen Server übernehmen. Auch die verteilte Auswertesoftware hat ein Mechanismus zur Ausfallsicherheit.

Dabei handelt es sich um einen Tradeoff. Zum einen beim Speicherplatz. Je häufiger die Daten gespeichert werden, desto mehr Platz wird für die gleiche Menge Daten benötigt. Zum anderen in zeitlicher Hinsicht. Die Daten werden über das Netzwerk an die Server gesendet. Wenn man nun sicher gehen möchte, dass die Kopien auch vorhanden sind, muss darauf warten dass die Server Erfolg signalisieren. Dies kann bei mehr als einem Server deutlich länger dauern als bei einem Server.

Diese Überlegung führt zu Grenzen der Architektur. Durch die Verbindung

über das Netzwerk können nicht alle gewünschten Eigenschaften sichergestellt werden.

### 2.2.11 Grenzen der Architektur

Normalerweise wird die Software aus unserer Architektur in IP-basierten Netzwerken verwendet werden. Dies sind so genannte asynchrone Netzwerke. Das heißt, dass die Nachrichtenlaufzeit zwischen zwei Servern nicht durch einen festen Wert begrenzt. Das bedeutet, dass man nicht feststellen kann, ob einen Server ("Knoten") ausgefallen ist oder die Nachricht nur verzögert ist und noch zugestellt werden würde. In der Praxis wird diese Eigenschaft simuliert, indem nach einem gewissen Timeout angenommen wird, dass der andere Knoten ausgefallen ist. Er wird dann aus dem System genommen. Zum anderen kommt es zu Problemen, wenn die Rechner untereinander nicht mehr erreichbar sind. Dadurch, dass die Ausführung verteilt wird, sind auch mehrere Server zum Funktionieren erforderlich. Dadurch steigt die Ausfallwahrscheinlichkeit mit der Anzahl der Rechner. Deshalb sind auch Mechanismen zur Ausfallsicherheit notwendig. Trotzdem besteht die Möglichkeit, dass das Netzwerk zwischen den Servern ausfällt. Das CAP-Theorem besagt, dass in diesem Fall nur entweder das System an jedem Knoten erreichbar bleibt oder konsistent ist. Für was man sich entscheidet, hängt von den gewünschten Zieleigenschaften der Architektur ab.

# Kapitel 3

## Umsetzung von Kommunikationsschemen über Kafka

Um unsere Applikation skalierbar aufzusetzen zu können, musste auf die Skalierbarkeit der Kommunikation zwischen den Microservices geachtet werden. In diesem Kapitel wird beschrieben, wie eine skalierbare Kommunikation über Kafka umgesetzt wurde. Hierfür werden zunächst die benötigten Grundlagen bezüglich Kafka beschrieben und anschließend wird auf die Realisierung von zwei Kommunikationsschemen mithilfe von Kafka eingegangen.

### 3.1 Grundlagen Kafka

Die in diesem Abschnitt beschriebenen Grundlagen stammen aus [10] und [9].

#### 3.1.1 Registrieren eines Consumers

Wird eine Nachricht über eine Kafka-Instanz verbreitet, so ist sie einem *Topic* zugeordnet. Möchte eine Komponente von Kafka Nachrichten erhalten, muss sie sich als *Consumer* registrieren. Hierbei muss sie mindestens ein *Topic* angeben, an dem sie interessiert ist, sowie eine *Consumer Group*. Wird nun eine Nachricht über Kafka verbreitet, wird ein *Consumer* jeder *Con-*

*sumer Group*, der an dem *Topic* der Nachricht interessiert ist, ausgewählt und die Nachricht an diesen *Consumer* weitergeleitet. Abbildung 3.1 zeigt ein Beispiel für die Zuordnung von *Consumern* zu *Consumer Groups* und die Registrierung der *Consumer* bei verschiedenen *Topics*. Wird in diesem Beispiel ein Nachricht für *Topic 2* hinterlegt, so werden *Consumer 1* oder *Consumer 2*, sowie *Consumer 4* informiert. Wird allerdings eine Nachricht für *Topic 1* hinterlegt, wird nur *Consumer 1* oder *Consumer 2* informiert.

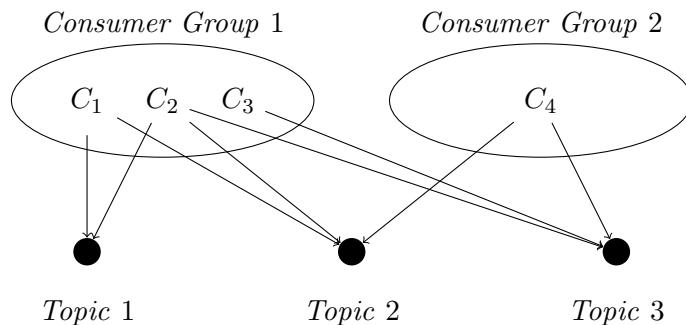


Abbildung 3.1: Beispiel für die Zuordnung von *Consumern* zu *Consumer Groups* und *Topics*.

### 3.1.2 Nachrichtenverteilung

Ein *Topic* von Kafka ist in mehrere *Partitionen* unterteilt. Die Anzahl der *Partitionen* muss angegeben werden, wenn ein *Topic* erstellt wird.

Haben sich mehrere *Consumer* einer *Consumer Group* bei Kafka registriert, um Nachrichten für ein *Topic* zu erhalten, so werden die *Partitionen* zunächst auf die *Consumer* der *Consumer Group* verteilt. Erhält Kafka nun eine Nachricht für das *Topic*, kann entweder vom *Producer* - also dem Sender der Nachricht - festgelegt werden, in welche Partition die Nachricht gelegt werden soll, oder Kafka teilt die Nachricht so einer Partition zu, dass die Nachrichten möglichst gleichmäßig auf die *Partitionen* verteilt sind. Nachdem die Nachricht einer *Partition* zugeordnet ist, wird sie dem *Consumer* einer jeden *Consumer Group* übermittelt, dem die *Partition* zugeteilt ist.

Für die Zuordnung der *Partitionen* zu den *Consumern* einer jeden *Consumer Group* wird eine Zuordnungsstrategie verwendet. Hier liefert Kafka die beiden Strategien „range“ und „roundrobin“ mit, von denen eine aus-

gewählt werden kann. Alternativ lässt sich eine eigene Zuordnungsstrategie definieren.

## 3.2 Realisierung der Kommunikationsschemen

In unserem Projekt „Hamaube“ haben wir die Semantik der Nachrichten so gewählt, dass sie als *Events* zu verstehen sind. Diese *Events* können potentiell von einer beliebigen *Instanz* ausgelesen werden. So ist beispielsweise unsere Analyseinstanz an mehrere *Topics* angeschlossen, um Analysen über die Nutzung von Hamaube bzw. die Twitterdaten durchzuführen.

Trotz dieser *Event*-Charakteristik haben viele Nachrichten ein primäres Ziel. So hat beispielsweise ein *Event*, das signalisiert, dass ein Anwender eine Webserver-Instanz nach seiner Timeline gefragt hat, das primäre Ziel, dass es von einer beliebigen Cassandra\_Reader-Instanz aufgenommen wird, um die Einträge der Timeline aus der Datenbank zu laden. Anschließend sollte diese Cassandra\_Reader-Instanz ein *Event* verschicken, das signalisiert, welche Einträge dem Anwender angezeigt werden sollen. Dieses *Event* hat wiederum das primäre Ziel, dass die entsprechende Webserver-Instanz die Anfrage des Nutzers beantworten kann. Im Folgenden wird die Komponente, die das *Event* primär erhalten soll, Kommunikationspartner genannt.

### 3.2.1 Beliebiger Kommunikationspartner

Für viele Nachrichten, die Hamaube über Kafka verschiickt, reicht es, wenn eine beliebige Instanz einer bestimmten Gruppe von Servern das Event erhält. Wenn beispielsweise eine Webserver-Instanz eine Cassandra\_Reader-Instanz darüber informieren möchte, dass die Einträge einer Timeline geladen werden sollen, ist für die Webserver-Instanz egal, welche Cassandra\_Reader-Instanz das Event zur Bearbeitung der Anfrage erhält.

Soll ein beliebiger Kommunikationspartner für ein Event ausgewählt werden, kann das Event von Kafka einer beliebigen Partition zugeteilt werden und eine der beiden Standard Zuordnungsstrategien gewählt werden, um die Partitionen den *Consumern* der *Consumer Group* zuzuordnen.

### 3.2.2 Ausgewählter Kommunikationspartner

In einigen Fällen ist das *Event* allerdings primär nur für eine Instanz entscheidend. So sollte das *Event*, dass die Einträge einer Timeline enthält beispielsweise möglichst direkt die Webserver-Instanz erreichen, an der die Timeline angefragt wurde. Um dies zu realisieren, könnte die Webserver-Instanz der Anfrage ihre ID hinzufügen und dann das Antwort-*Topic* nach ihrer ID filtern. Diese Lösung würde allerdings dazu führen, dass die Kommunikation nicht mehr skalierbar wäre, da dann jede Webserver-Instanz jede Antwort auf jede Anfrage erhalten müsste und zum Filtern der Antworten bearbeiten müsste. Dies ließe sich nicht skalierbar realisieren. Die *Consumer* müssen also der selben *Consumer Group* zugeordnet sein.

Um nun zu gewährleisten, dass die Webserver-Instanz, die die Anfrage gestellt hat, die Antwort erhält, muss die Antwort zum einen in eine feste *Partition* gelegt werden und zum anderen musste eine Strategie für die Zuordnung von Partitionen auf die *Consumer* implementiert werden, mit der garantiert werden kann, dass die Antwort auch bei erneuter Verteilung der *Partitionen* beim richtigen *Consumer* landet. Dies können beide von Kafka mitgelieferten Verteilungsstrategien nicht gewährleisten, wie in Abbildung 3.2 beispielhaft für das „roundrobin“-Verfahren dargestellt ist.

Zustand 1:		
Consumer 1	Consumer 2	Consumer 3
Partition 1	Partition 2	Partition 3
Partition 4	Partition 5	

Zustand 2:	
Consumer 1	Consumer 3
Partition 1	Partition 2
Partition 3	Partition 4
Partition 5	

Abbildung 3.2: Beispiel für die Umverteilung der Partitionen mit dem „roundrobin“ Verfahren: Zwischen Zustand 1 und Zustand 2 wurde der Consumer 2 entfernt.

Wird eine Webserver-Instanz von Hamaube gestartet, muss ihr eine Partiti-

### 3.3. Microservice übergreifendes Datenformat

---

onsnummer als Argument übergeben werden. Registriert sich die Webserver-Instanz bei Kafka als Consumer, kann sie eine ID übergeben, die später an den Verteilungsalgorithmus für die Partitionen weitergegeben wird. Durch die ID kann der selbst implementierte Verteilungsalgorithmus erkennen, welche Partition diesem Consumer zugeordnet werden muss. Ist eine Partition vorhanden, für die kein Consumer existiert, dessen ID die Nummer der Partition enthält, wird die Partition einem beliebigen Consumer zugeordnet.

Schickt eine Webserver-Instanz nun eine Anfrage an eine Cassandra\_Reader-Instanz, enthält diese Anfrage ein Feld, das die Partition angibt, in die die Antwort gelegt werden soll. Durch den Verteilungsalgorithmus für die Partitionen ist garantiert, dass der Webserver-Instanz, solange sie erreichbar ist, diese Partition zugeordnet ist, sodass nur sie die Antwort erhält. Ist die Webserver-Instanz nicht mehr erreichbar, wird ihre Partition einer anderen Webserver-Instanz zugeordnet, die die Antwort erhält und verwirft, da sie die Anfrage nicht gestellt hat. Die Antwort geht in diesem Falle also verloren.

## 3.3 Microservice übergreifendes Datenformat

Wir haben beschlossen, Daten von Twitter in unserer Applikation zu verarbeiten. Hierfür haben wir einen Twitterstream angeschlossen, der Tweets aus Twitter in eines unserer Topics legt. Unter anderem sollte der Cassandra-Reader sich auf diesem Topic subscriben und die Tweets in Cassandra speichern.

Da zu Beginn des Projektes nicht klar war, was für Analysen wir später auf den Daten durchführen könnten und welche Teile der Twitterdaten wir dafür benötigen könnten, haben wir beschlossen, alle Daten über die Tweets von Twitter zu speichern und das Datenformat von Twitter für unsere Kommunikation und die Datenhaltung in Cassandra zu übernehmen.

Wir mussten feststellen, dass das Format der Tweet-Daten von Twitter eine sehr große, verschachtelte Struktur ist, in der häufig *null*-Werte auftauchen. Die *null*-Werte haben dafür gesorgt, dass sich die Definition des Datenformats manuell nur schwer aus den Daten extrahieren ließ. Twitter stellt eine Api für die Schnittstelle zur Verfügung, aber auch hier wäre es sehr

### 3.3. Microservice übergreifendes Datenformat

aufwändig gewesen, das gesamte Datenformat zu entnehmen.

Um dennoch eine Definition des Datenformats zu finden, haben wir einen Parser geschrieben, der als Eingabe eine Datei mit Tweets von Twitter entgegen nimmt und aus diesen Daten das Datenformat in Form von Cql-Typ-Definitionen ausgibt. Das Ergebnis enthält alle Felder mit Typ-Angabe, für die in den Tweet-Daten aus der Datei mindestens ein Eintrag mit einem Wert außer *null* gefunden wurde. Nur ein paar Felder konnten nicht übernommen werden, da sie immer mit *null* befüllt waren. Für diese Felder ließ sich in der Twitter-Api nachlesen, dass diese *deprecated* sind - also nicht mehr genutzt werden. Die Felder, die wir nicht in unsere Datenstruktur übernommen haben, können also keine relevanten Information enthalten.

# Kapitel 4

## Webserver und Frontend

Der Zugriff auf unsere Anwendung wurde mit dem Framework Angular realisiert. Als Schnittstelle zwischen Frontend und dem Message-Broker Kafka wurde ein Webserver geschaltet. Damit es von Außerhalb keine Möglichkeit gibt, direkt auf Kafka zuzugreifen. Ein weiterer Vorteil bei dieser Konstellation ist, dass nur die Verbindung zwischen Frontend und dem Webserver abgesichert werden. Jede weitere Kommunikation innerhalb der einzelnen Prozesse findet in einem eigenen Netzwerk statt.

### 4.1 Frontend

Das Frontend wurde mit dem Framework angularjs erstellt. Es gab keinen speziellen Grund, warum sich das Team für angularjs entschieden hat. Zu Beginn der Hamaube wurde auch kein Fokus auf eine Weboberfläche gesetzt. Der Fokus lag auf dem Backend und nicht dem Frontend. Da es keine Pflicht Aufgabe war und es „schnell“ gehen sollte, wurde angularjs gewählt, da schon gewissen Vorkenntnisse vorhanden waren. Zudem bietet angularjs einem die Möglichkeit Module zu entwickeln, welche dann passend eingebunden werden. Der Vorteil daran ist, dass nur einmal das Layout eines Tweets definiert werden muss. Wenn der Webserver ein JSON-Dokument mit einer Liste von Tweets an das Frontend bereitstellt, kann einfach über diese Liste iteriert werden um eine Timeline zu erstellen.

Das klassische Problem bei der Erstellung der Webanwendung war das Zu-

greifen auf die JSON-Dokumente. Zum Beispiel wurde das Objekt in dem JSON-Dokument in einer anderen Ebene erwartet oder beim Bezeichner wurde die Groß-/Kleinschreibung verwechselt.

## 4.2 Webserver

Der Webserver wurde mit dem Framework NodeJs erstellt. Dieser bietet verschiedene REST-Schnittstellen, welche mit Hilfe des Paket *express* bereitgestellt wurden. Außerdem wurde eine Bibliothek für die Anbindung an Kafka implementiert. Da der Webserver nur als Vermittler fungiert, wurde eine Funktion implementiert welche zwei Parameter benötigt um das ganze so automatisiert wie möglich zu gestalten. Der erste Parameter dient dem Frage-Topic an den Kafka Server und der zweite für die Antwort in Kafka. Zusätzlich zu diesen zwei Parametern wurde eine Schnittstelle definiert, welche das Frontend mittels einer HTTP REST-Methode ansprechen kann. Der folgende Teil beschreibt die oben beschriebene Funktion: Es wurde ein LoginController erstellt welcher mit den zwei Parametern *USER\_ISSUES\_LOGIN* und *LOGIN\_RESULT\_IS\_PROVIDED* die oben beschriebene Funktion aufruft:

```
1 var kommunikationHandler = new KommunikationHandler  
2 .constructor('USER_ISSUES_LOGIN', 'LOGIN_RESULT_IS_PROVIDED  
' );
```

---

Diese zwei Topics wurden in dem Kaptiel 3 definiert. Im folgenden Code-Beispiel wird der LoginController mit einer REST-Schnittstelle */users/login* verknüpft.

```
1 var LoginController = require('./endpoints/  
    loginController');  
2 app.use('/users/login', LoginController);
```

---

Durch dieses Vorgehen konnten ohne großen Aufwand weitere REST-Endpunkte mit hinterlegtem Topics für das Frontend definiert werden.

Probleme gab es am Anfang zwischen der Kommunikation vom Frontend und der Springboot Anwendung. Durch die Überlastung der Springboot An-

wendung dauerte eine Antwort sehr lange. Damit der HTTP-Request nicht vorzeitig beendet wurde, musste ein ausreichender Timeout gesetzt werden, ansonsten lief eine Anfrage ins Leere. Um die Benutzerfreundlichkeit und Sicherheit zu erhöhen generiert der Webserver nach einer erfolgreichen Authentifizierung eine Session-Id. Somit sind alle weiteren HTTP-Requests, welche das Frontend versendet, abgesichert.

Nachdem die Grundfunktionalität funktionierte wurde der Webserver erweitert.

Das Ziel war es, mehrere Webserver parallel laufen zu haben, damit ein "Load Balancing" simuliert werden konnte. Da wir aber nur eine virtuelle Maschine zur Verfügung hatten und außerdem kein Netzwerktraffic simulieren konnten wurde das Frontend um eine weitere Funktion erweitert. Im unteren Bereich der Webanwendung kann ein Port definiert werden. Voraussetzung, damit diese Funktion funktioniert, ist dass der Webserver drei mal auf einem verschiedenen Port gestartet wurde. Außerdem ist es nicht möglich, den Webserver während des Betriebs zu wechseln, da die drei Webserver nicht untereinander kommunizieren und somit keine Session-Ids austauschen.

Damit der Webserver auch während des Benutzens des Frontends gewechselt werden kann, müssten weitere Vorkehrungen getroffen werden. Um dieses Szenario zu realisieren gibt es mehrere Möglichkeiten. Eine Möglichkeit wäre eine Sticky-Session-Id. Dabei wird sichergestellt, dass die REST-Anfragen immer nur an den Webserver gesendet wird, von welchem die Session-Id erzeugt wurde. Der Nachteil an einer Sticky-Session-Id ist aber, dass der Vorteil eines Load Balancing verloren geht, da der Client nach einem Anmelden an den Webserver gebunden ist. Falls dieser Webserver nicht mehr verfügbar ist, muss sich der Client neu bei der Anwendung anmelden um von einem neuen verfügbaren Webserver eine neue Session-Id zu bekommen. Damit ein Client im Betrieb unabhängig vom jeweiligen Webserver agieren kann, müssen die Webserver untereinander kommunizieren und ihre verteilten Session-Ids austauschen. Dabei kann der Austausch über eine Datenbank oder ein Dateien-System realisiert werden. Zwar bedeutet die zweite Möglichkeit einen größeren Aufwand, dafür kann die Funktionalität des Load Balancing voll ausgeschöpft werden. Zudem kann die Performance einer Anwendung deutlich erhöht werden, in dem der Load Balancer dem Client jeweils einen Webserver dynamisch zuteilt welcher im Moment genügen

Kapazität zur Verfügung hat.

Wir sind davon ausgegangen, dass mehrere User gleichzeitig Anfragen im Frontend an den gleichen Webserver absenden. Da bei Kafka gesendete Anfragen nicht in der gleichen Reihenfolge abgearbeitet werden, wie sie abgesendet wurden und wir davon ausgehen, dass mehrere Anwender gleichzeitig einen Request an den selben Webserver senden, muss sichergestellt werden, dass die Antwort an den richtigen Client gesendet wird. Um dieses Problem zu lösen, wurde jede Anfrage von der Webanwendung an den Webserver mit einer inkrementierenden Request-ID versehen. Diese ID wurde dem JSON-Dokument, welches an Kafka zur Bearbeitung weiter gereicht wurde, hinzugefügt. Das Antwort-Topic enthielt diese zu Beginn hinzugefügte Request-ID. Somit konnte der Webserver mehrere Anfragen gleichzeitig bearbeiten und es wurde sichergestellt, dass keine Anfrage an einen falschen Client versendet wird.

## 4.3 Use Cases

Um das Zusammenwirken aller Microservices innerhalb des Hamaube-Systems aufzuzeigen und nachvollziehen zu können, werden im Folgenden einige Use Cases und dazu gehörige Kommunikation vorgestellt, bei denen mehrere Microservices beteiligt sind.

### 4.3.1 User erstellt einen neuen Tweet

Wenn ein User einen Tweet erstellt (oder ein neuer Tweet über die Twitter API in das Hamaube-System gepusht wird), wird dieser Tweet über das *USER\_ISSUES\_TWEET* Topic verteilt. Der Cassandrareader speichert den Tweet, damit dieser später für Hamaube Nutzer angezeigt wird. Nach dem Speichern wird über das Topic *TWEET\_SAVED* eine entsprechende Meldung zurückgegeben, sofern der Tweet erfolgreich gespeichert wurde. Nun sind aber noch weitere Microservices an das Topic angebunden. So verarbeitet auch der ElasticSearch Microservice den übermittelten Tweet um ihn z.B. später in Suchanfragen zu berücksichtigen. Und auch der SparkStreaming Microservice verarbeitet den Tweet z.B. um Analysen zu erstellen.

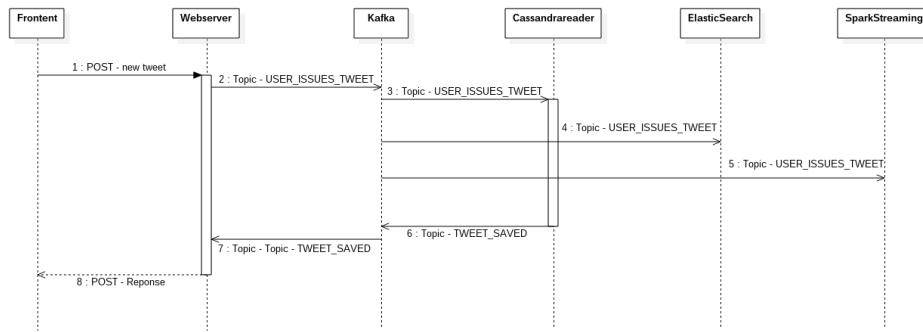


Abbildung 4.1: Sequence Diagram: User erstellt neuen Tweet

#### 4.3.2 User sucht einen Tweet (nach Text)

Derzeit bietet das Hamaube-System Suchen von Tweets über Hashtags oder über den Tweet-Text. Im Folgenden wird letzterer Fall betrachtet. Sobald ein Nutzer im Frontend nach einem Tweet sucht, wird die Suchanfrage über Kafka an den Elasticsearch Microservice weitergereicht. Dieser vermittelt das Ergebnis der Suche als Tweet-IDs über ein Antwort-Topic, welches wiederum an den Cassandrareader geht. Dieser liest die zugehörigen Tweets aus Cassandra und veröffentlicht das Ergebnis an den WebServer. Gerade hier zeigt sich, dass die Meta-Information vom ursprünglichen WebServer bis zur letzten Stelle weitergereicht werden müssen, damit auch der richtige WebServer letztendlich die Tweets zur Anfrage erhält.

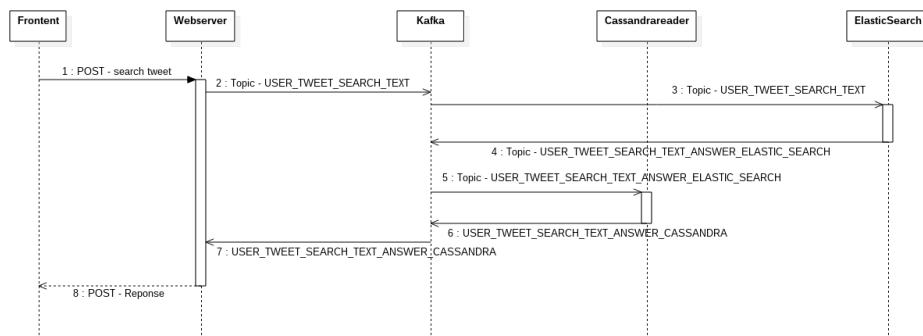


Abbildung 4.2: Sequence Diagram: User sucht nach einem Tweet (by text)

# Kapitel 5

## Cassandra

Cassandra ist die Quelle aller Daten von Twitter, die wir brauchen. Dazu werden die Daten direkt von der Twitter API über Kafka in Cassandra geladen und auf ein vorher für unsere Bedürfnisse zugeschnittenes Datenschema gemappt.

### 5.1 Datenverwaltung

Wir habe uns entschieden das Twitter Datenschema zu übernehmen. Da allerdings die Twitter Dokumentation nicht genau genug ist und nicht alle Attribute aller Datentypen übersichtlich darstellt, haben wir eine Applikation geschrieben, die sich Tweets vom Twitter Stream holt und daraus das Datenschema im JSON Format zusammenbaut. Nach dem wir die Applikation lange genug laufen lassen haben, hat sich an dem Datenschema nichts mehr geändert und wir konnten die Datentypen extrahieren.

#### 5.1.1 Datenschema

Nach einer eingehenden Untersuchung aller möglichen Use Cases sind wir zum Schluss gekommen, dass uns fünf Tabellen alle Funktionen bieten, die wir brauchen. Wir haben dabei zwei Tabellen für die User `user_by_id` und `user_by_screen_name` entworfen wie man in Abbildung 5.1 sehen kann. Da man bei Cassandra nur über den Primary Key (PK) auf Zeilen zugreifen und Bereichsabfragen über CQL machen kann, gilt es hier den PK

## 5.1. Datenverwaltung

---

Key Space	Table	Column	Type	Primary Key
twitter_v3	tweets_by_tweeted	user_id	varint,	(user_id, full_timestamp)
		timestamp_hash	varint,	
		full_timestamp	text,	
		tweet	frozen<twitter_v3(tweet_type)>,	
twitter_v3	tweets_by_follower	user_id	varint,	(user_id, full_timestamp)
		timestamp_hash	varint,	
		full_timestamp	text,	
		tweet	frozen<twitter_v3(tweet_type)>,	
twitter_v3	tweets_by_hashtag	hashtag	text,	(hashtag,full_timestamp)
		timestamp_hash	varint,	
		full_timestamp	text,	
		tweet	frozen<twitter_v3(tweet_type)>,	
twitter_v3	user_by_id	user_id	varint,	(user_id, full_timestamp)
		timestamp_hash	varint,	
		full_timestamp	timestamp,	
		user	frozen<twitter_v3(user_type)>,	
		followers	set<frozen<twitter_v3(user_type)>>	
		friends	set<frozen<twitter_v3(user_type)>>	
twitter_v3	user_by_screen_name	screen_name	text,	(screen_name, full_timestamp)
		timestamp_hash	varint,	
		full_timestamp	timestamp,	
		password_hash	text,	
		user	frozen<twitter_v3(user_type)>,	
		followers	set<frozen<twitter_v3(user_type)>>	
		friends	set<frozen<twitter_v3(user_type)>>	

Abbildung 5.1: Cassandra Schema

so zu wählen, dass alle unsere Funktionen abgedeckt sind. Deshalb haben wir neben der User-Id für user\_by\_id und dem Screen-Name des Users für user\_by\_screen\_name auch den Timestamp mit aufgenommen. Da Cassandra leider keine vollständige Konsistenz bietet, müssen wir uns selber darum kümmern. Durch den Timestamp können wir verschiedene Versionen eines Objektes auseinanderhalten und die neuste bestimmen. Somit können wir zumindest einen gewissen Grad an Konsistenz bieten. Die weiteren Attribute der beiden Tabellen lassen sich einfach erklären. Die Follower und Friends eines Users sind wichtig, um die Timeline zu erstellen. Den password\_hash in user\_by\_screen\_name brauchen wir für den Login.

Die anderen drei Tabellen sind dafür da, die Tweets zu speichern und alle Tweet betreffenden Anfragen zu beantworten. Auch hier haben wir wieder den Timestamp bei allen Tabellen mit in den PK aufgenommen um Teil-konsistenz zu gewährleisten. tweets\_by\_tweeted speichert alle Tweets nach der User-Id des Users ab, der den Tweet abgesetzt hat. tweets\_by\_follower hingegen speichert einmal alle Tweets nach User-Id eines jeden Followers ab. Das Konzept hier ist es, durch die mehrfache Speicherung eines Tweets die Zeit bei der Abfrage nach allen Tweets, die ein User auf seiner Timeli-ne sehen kann, zu verkürzen. Da man einmal abgesetzte Tweets auch nicht mehr ändern kann haben wir auch kein Problem damit jedes Objekt für Änderungen wieder heraussuchen zu müssen. tweets\_by\_hashtag speichert dann die Tweets danach ab, welche Hashtags in ihnen verwendet werden. Somit können auch Abfragen über Tweets eines Hashtags effizient beant-wortet werden.

## 5.2 Cassandrareader

Die zentrale Applikation, in der alle Funktionen und Schnittstellen umge-setzt werden ist der Cassandrareader und es ist eine modular aufgebaute in Java geschriebene Anwendung. Als Framework zur Unterstützung von ver-schiedenen Funktionen haben wir uns für SpringBoot entschieden. Spring-Boot hat den Vorteil, dass es eine native API für Kafka besitzt, die es uns so sehr leicht ermöglicht Publisher und Subscriber für Kafka-Tops zu schrei-ben. So ist die Verbindung mit Kafka sehr einfach konfigurierbar und kann innerhalb von kurzer Zeit verwendet werden. In der Konfigurationsklasse

werden Beans, also einzigartige Methoden, überschrieben, die die Kafka-Konfiguration für den Publisher und Subscriber erzeugen und in SpringBoot registriert sind. Diese werden für diese Konfiguration vorher in der application.properties Datei abgelegt und durch die Konfigurationsklasse eingelesen. SpringBoot erzeugt darauf aufbauend durch die Beans jeden Publisher und Subscriber nach dieser Konfiguration. Für die Verbindung von Java zu Cassandra haben wir den DataStax Treiber genutzt [4]. Er bietet eine generische Schnittstelle über die man mit Cassandra über CQL kommunizieren kann. Da er gut dokumentiert ist und es sehr viele Beispiele für verschiedene Anwendungen im Internet gibt, verlief die Einarbeitung in die Nutzung des DataStax Treibers sehr schnell. Alle hier und im Folgenden genutzten



Abbildung 5.2: Technologie Stack des cassandrareaders

Bibliotheken werden über Maven eingebunden und der Applikation so zur Verfügung gestellt. Somit ist sichergestellt, dass immer die richtige Version geladen wird und keine Kompatibilitätsprobleme entstehen.

### 5.2.1 Architektur

Der Aufbau des Cassandrareaders ist sehr einfach gehalten wie man in Abbildung 5.3 sehen kann. Die Verbindung zu Cassandra wird vom Singleton CassandraConnector gemanaged. Diese Klasse stellt die Verbindung zu Cassandra her und bietet verschiedene Methoden an, Abfragen an Cassandra über CQL abzusetzen. Die eigentliche Funktion und Implementierung der Use Cases geschieht aber in den Kafka Subscribersn. Dazu gibt es eine abstrakte Klasse AbstractKafkaSubscriber, die sozusagen die Infrastruktur bereitstellt. Diese besteht aus dem CassandraConnector, einer Gson-Instanz und mehreren Methoden, die die Optimierungen der Methoden aus dem Cas-

sandra Connector darstellen, wie z.B. Batch-Queries und asynchrone Queries. Die Gson-Instanz kommt von der Google Gson Bibliothek, die für die JSON Konvertierung von Java Klassen zuständig ist. Sie wird in den abgeleiteten Klassen so benutzt, dass Kafka-Anfragen direkt in POJOs (Plain Old Java Objekte) gemappt werden, aus denen man dann alle relevanten Informationen bekommt. In den abgeleiteten Klassen wird dann auch die eigentliche Funktion eines Use Cases implementiert. Alle möglichen Anfra-

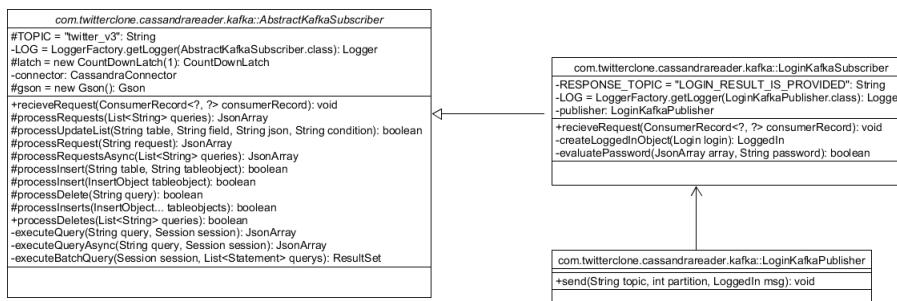


Abbildung 5.3: Architektur der Publisher und Subscriber am Beispiel Login

gen und Antworten über Kafka sind als Java Klassen modelliert und können über Getter-Methoden abgefragt werden. Jeder der abgeleiteten Subscriber besitzt einen Publisher, über den die Antwort, durch Gson konvertiert, wieder versendet werden kann. Die Konfigurationsparameter sind in der Datei application.properties abgelegt und werden in den einzelnen Klassen ange- sprochen.

### 5.3 Use Cases

Bei der Identifizierung der Use Cases, die für Cassandra relevant sind, haben wir uns für die Funktionen entschieden, die für einen Twitter-Client nach MVP-Prinzip (Minimal Viable Product) notwendig sind. Dabei haben wir vor allem die dazu gehören folgende Funktionen identifiziert:

- Registrierung von User
- Anmelden von Usern
- Abfragen von Usern

- Absenden/Speichern von Tweets
- Abrufen der Timeline
- Folgen von Usern

Diese Schnittstellen machen es möglich die Grundfunktionen, also das Erstellen eines Users, das Anmelden des Users, das Senden und Lesen von Tweets über die Timeline von außen über vordefinierte Kafka Nachrichten anzusprechen. Als weitere Schnittstellen für erweiterte Funktionen haben wir eine API für Volltextsuche über Elasticsearch gebaut, die die normale Volltextsuche aber auch Autocompletion unterstützt.

#### 5.3.1 Registrierung von Usern

Die Registrierung von Usern bringt einen direkten Zugriff auf die Datenbank mit sich. Zunächst muss der User seine Anmelddaten in unserem Frontend hinterlegen. Schickt er diese, bekommt der Cassandrareader über eine Zwischenstelle eine Nachricht über Kafka auf dem Topic “USER\_ISSUES\_REGISTRATION”. Diese Nachricht enthält alle relevanten Daten wie Name, Username, Hashwert des Passworts, etc. die das System braucht um den User anzulegen. Der Cassandrareader reagiert auf diese Nachricht, indem er den User in den in Abschnitt 5.1.1 genannten Tabellen für die User ablegt. Der Cassandrareader schickt eine Nachricht auf dem Topic ”REGISTRATION\_RESULT\_PROVIDED“ als Bestätigung zurück, ob der User erfolgreich in beide Tabellen eingetragen wurde oder nicht. Basierend auf dieser Nachricht wird im Frontend angezeigt, ob die Registrierung erfolgreich war.

#### 5.3.2 Anmelden von Usern

Die Anmeldung von Usern geschieht über den einzigartigen Username und ein selber gewähltes Passwort. Diese Informationen werden vom Frontend weitergeleitet und kommen beim Cassandrareader in einer Kafka Nachricht auf dem Topic “USER\_ISSUES\_LOGIN“. Diese Nachricht enthält nur den Username und das gehashte Passwort. Das gehashte Passwort wird mit dem verglichen, das in der Tabelle user\_by\_screen\_name gespeichert

ist. Auch hier findet sich nur ein Hashwert des Passworts. Sind die Werte gleich, so gibt es hier eine Übereinstimmung und der Login kann genehmigt werden. Als Bestätigung dieses Vorgangs schickt der Cassandra-reader eine Nachricht zurück, in der es eine Boolean Feld für das Login Ergebnis gibt. War der Login erfolgreich wird das Feld true, sonst false. Diese Nachricht wird vom Cassandra-reader wieder über das Topic "LOGIN\_RESULT\_IS\_PROVIDED" verbreitet und somit auch an das Frontend weitergeleitet, das das Ergebnis darstellt und darauf reagiert.

#### 5.3.3 Abfragen von Usern

Das Abfragen von Usern braucht man vor allem, wenn man die Seite eines Users mit allen seinen Informationen aufrufen möchte. Dazu verarbeitet der Cassandra-reader Nachrichten auf dem Topic "USER\_OPENS\_USER\_PAGE" in der eine User-Id angegeben ist, so dass er eine Abfrage auf Cassandra startet, in der er den User aus user\_by\_id abfragt. Dieser User wird dann als JSON in einer Kafka Nachricht über das Topic "USER\_PAGE\_ENTRIES\_PROVIDED" wieder verbreitet. So können die Information dann vom Frontend dann z.B. als User Seite verarbeitet werden.

Ein ähnlicher Use Case ist die Suche nach Usern. Der Cassandra-reader bekommt auf dem Topic "USER\_SEARCH\_ISSUED" eine Nachricht mit einem Usernamen. Da die Tabelle user\_by\_screen\_name nach Usernamen filterbar ist, kann der Cassandra-reader eine effiziente Abfrage auf der Tabelle ausführen, die alle mit dem Namen aus der Nachricht übereinstimmenden User zurückgibt und danach den besten möglichen auswählt. Dieser wird dann in einer Nachricht auf dem Topic "USER\_SEARCH\_RESULT\_PROVIDED" zurückgeschickt und kann dann weiterverarbeitet werden.

#### 5.3.4 Absenden/Speichern von Tweets

Das Absenden von Tweets ist im Endeffekt nichts anderes als ein Einfügen eines Tweets in die Datenbank. Da wir hier aber nicht mit einer relationalen Datenbank arbeiten, muss man Einfügen vor allem auf die Konsistenz des Datenschemas achten. Jeder Tweet muss nach dem Schema unabhängig voneinander in jedem der drei Tweet Tabellen richtig abgelegt werden. Dafür bekommt der Cassandra-reader das Tweet Objekt über Kafka auf dem To-

pic "USER\_ISSUES\_TWEET" geschickt. Dieses Objekt wird dann mit dem Absender als User in die Tabelle `tweets_by_tweet` eingefügt. Zusätzlich lesen wir alle Hashtags des Tweets aus und speichern ihn in der Tabelle `tweets_by_hashtag` noch einmal für jeden Hashtag, das beschleunigt das Auslesen nach Hashtags. Als letztes wird der Tweet auch noch einmal in der Tabelle `tweets_by_follower` abgespeichert, in der jeder Tweet eines Users für jeden User, der ihm folgt, abgespeichert werden. Dafür müssen natürlich zunächst erst einmal alle Follower des Absenders abgefragt werden, bevor der Tweet für sie gespeichert werden kann. Diese redundante Speicherung von Tweets beschleunigt das Auslesen für spezifische und oft gebrauchte Use Cases ungemein und hat für uns die damit einhergehenden Performance Nachteile beim Speichern von Tweets überwogen. Der Cassandra reader führt diese Einfüge Operationen durch und bestätigt über Kafka auf dem Topic "TWEET\_SAVED", ob das Speichern erfolgreich war oder nicht.

#### 5.3.5 Abrufen der Timeline

Das Abrufen der Timeline ist einer der zentralen Use Cases unserer Anwendung, da sie der erste Teil ist, den User innerhalb unserer Anwendung sehen und die Basis für jede Aktion nach dem Anmelden ist. Die Timeline eines Users besteht aus den Tweets aller User, denen der eingeloggte User folgt. Daher müsste man diese User zunächst aus der Datenbank abfragen und dann für jeden einzelnen eine einzelne Abfrage starten. Um uns diesen Umweg zu sparen, pflegen wir die Tabelle `tweets_by_follower`. Somit können wir, um die Timeline aufzurufen, einfach eine Abfrage auf diese Tabelle ausführen und bekommen somit direkt den Inhalt der Timeline. Wir nehmen dabei für das Speichern von Tweets mehr Datenbankzugriffe und Redundanzen in Kauf, wie in Unterabschnitt 5.3.4 beschrieben, um das Auslesen so effizient wie möglich zu gestalten. Die so abgefragten Tweets werden danach als JSON verpackt und auf dem Topic "TIMELINE\_ENTRIES\_PROVIDED" verbreitet. So kann die Nachricht zum Frontend propagiert werden, das dann die Timeline anzeigt.

#### 5.3.6 Folgen von Usern

Das Folgen von Usern bestimmt vor allem den Aufbau der Timeline, da hier, wie in Unterabschnitt 5.3.5 erwähnt, alle Tweets der gefolgten User angezeigt werden. Um das Folgen von Usern umsetzen zu können, erhält der Cassandrareader einer Nachricht auf dem Topic “USER\_WANTS\_TO\_FOLLOW\_USER”, in der die User-Id des anderen Users und des eigenen steht. Followers werden in unserem Datenschema für jeden User als Liste gespeichert, einmal alle User, denen ein User folgt, und alle User, die einem User folgen. Jeder Eintrag für einen User in den Tabellen user\_by\_id und user\_by\_screen\_name enthält beide Listen. Außerdem enthält das User Profil einen Follower Count und eine Friends Count, der die Anzahl User zählt denen man selber folgt. Der Cassandrareader führt auf beiden Tabellen für jede Folgen Anfrage folgende Aktion durch. Er erhöht beim eigenen User den Friends Count und beim User, dem gefolgt werden soll, den Follower Count. Danach wird das User Profil des Users, dem gefolgt werden soll, in die Friends Liste des anfragenden Users hinzugefügt und das Profil des anfragenden Users in die Follower Liste der zu folgenden Users. Geschieht das alles ohne Fehler bestätigt der Cassandrareader die Anfrage auf dem “USER\_FOLLOWS\_USER” Topic positiv.

# Kapitel 6

## Search Engine

Die Twitter-Klon Applikation soll mit einer Suchfunktion erweitert werden. Diese Suchfunktion wird mit einem Such-Service umgesetzt, der auf der Elasticsearch-Engine [7] basiert. Elasticsearch setzt auf Apache Lucene [3] auf, einer in Java geschriebenen Bibliothek für die Volltextsuche. Es ist open-source, dokumentenorientiert, in Java geschrieben und lässt sich über einen REST-API ansprechen. Damit ist es ein guter Kandidat für unser Use-Case.

### 6.1 Zielsetzung

Das Ziel des Suchservices ist die Ausgabe von *best-match* Ranglisten über eine Tweets-Sammlung zu erstellen, indem man nach bestimmten Benutzern, Tags und Freitexteingaben sucht. Als Erweiterung der Suchfunktion wird ein Text-Vervollständigung Feature erstellt, das beim Suchen nach Tweets passende Vorschläge liefert. Und zum Schluss sollen ausgewählte Datensätze, mit Hilfe eines Analyse- und Visualisierungstools, auf der Homepage als eine Grafik angezeigt werden.

### 6.2 Vorgehen

Zuerst wird eine *Elasticsearch-Service* Architektur angefertigt. Diese stellt nur einen Teil der *Twitter-Klon* Umsetzung und behandelt die Kommunikation zwischen *Elasticsearch*, *Elasticsearch-Service* wie den *Kafka-Topics*. Nachfolgend wird Elasticsearch entsprechend der *Twitter-Klone* Applikati-

on konfiguriert und für die Datenaufnahmen vorbereitet. Infolgedessen wird der Such-Service implementiert, der *Elasticsearch* mit den *Kafka-Topics* verbindet und für den Datenaustausch zuständig ist. Anschließend werden Datenvisualisierungen mit Hilfe von *Kibana* angefertigt. Zum Schluss wird der Elasticsearch Abschnitt mit einem Fazit beendet.

## 6.3 Service Design

Der komplette Nachrichtenaustauch der *Twitter-Klon* Applikation wird mit einem skalierbaren und performanten Messaging-Service umgesetzt, nämlich *Apache Kafka*. Dieser bietet die Möglichkeit mit Hilfe der Nachrichten-Topics den Overhead an Kommunikation zwischen den Services zu verringern, die Services von einander zu entkoppeln und die Daten parallel zu verarbeiten. Diese Eigenschaft ermöglicht das Entkoppeln des Such-Service's von dem Restsystem, sodass die Suche unabhängig von den Schnittstellen der heterogenen Klienten bedient werden kann. Dafür wurden drei Topics erstellt und ein Datenformat vereinbart, das langfristig alle unsere Wünsche abdecken soll. Die Such-Service Use-Cases sind in drei Gruppen eingeteilt, die mit Hilfe von drei Kafka-Topics realisiert wurden: Neue Dokumente indexieren/abspeichern, Anfragen empfangen und Anfragen beantworten.

Der Such-Service meldet sich an den entsprechenden Kafka-Topics an, öffnet ein Datenstrom und empfängt Nachrichten mit den Speicher-, Suchkriterien, sowie der dazugehörigen Benutzererkennung. Um den Speicherplatz optimal zu nutzen, werden aus der Nachricht nur für die Suche relevante Felder ausgelesen und an die Elasticsearch REST-Schnittstelle weitergeleitet, wie z.B. der Text, die Benutzererkennung, die Tags und der Zeitstempel.

Das Verschicken der Nachrichten über unsere Applikation durchläuft den *UIT* Topic, dieser bietet jedem Abonnenten den Zugriff auf die neu eingetroffenen Mitteilungen. Der Elasticsearch-Suchservice ist einer der bestehenden Abonnenten des *UIT* Topics und bereitet die neu eingetroffenen Nachrichten für die Weiterleitung und das Speichern des Inhalts in Elasticsearch vor. Die, mit Metainformation bestückten, Nachrichten laufen eine Transformation durch wie Filterung, Umbenennung und Umstrukturierung, um der Elasticsearch erwarteten Datenstruktur zu entsprechen und Speicherplatz zu sparen.

### 6.3. Service Design

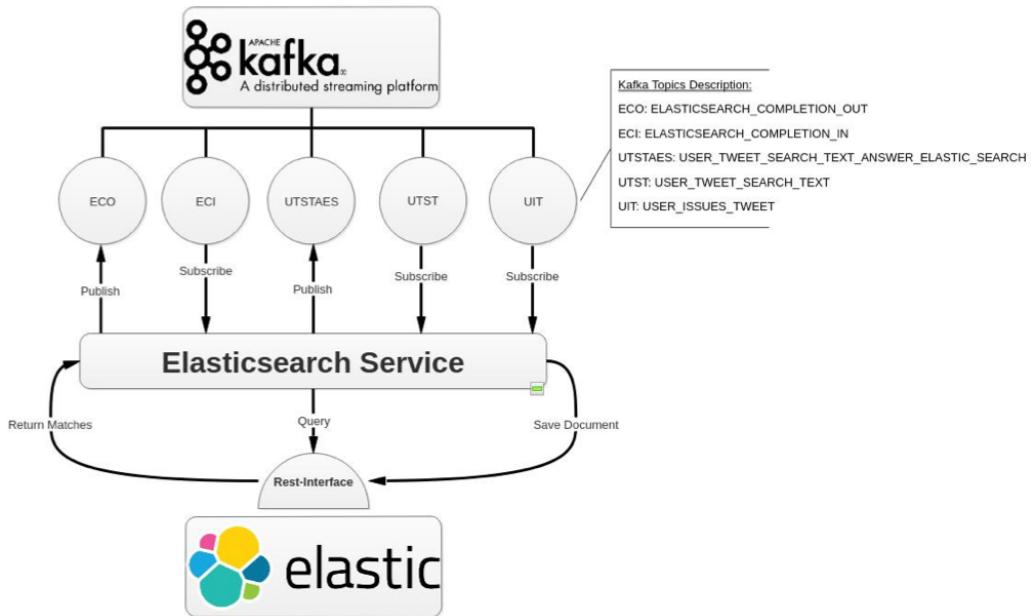


Abbildung 6.1: Elasticsearch Service Architectur

Für die Suchfunktion werden die Topics *UTST* und *UTSTAES* genutzt. Über den *UTST* werden Suchanfragen entgegengenommen und in eine Elasticsearch-Rest-Anfrage umgeformt. Die Suchanfrage kann von dem Suchservice auf verschiedene Weise durchgeführt werden. Zum Beispiel durch eine Term-, Boolean-, Match-, Multimatch- und Match-Phrase-Suche. Jede Abfrageart, kann im bestimmten Fällen vorteilhaft ausgenutzt werden und kann fallspezifisch eingesetzt werden. Das Suchergebnis wird anschließend als eine sortierte Identifikationsliste zu enthaltenen Nachrichten auf dem *UTSTAES* Topic hinterlegt. Damit wäre die Suche beendet und das weitere Geschehen des Suchergebnisses den *UTSTAES* Abonnenten überlassen.

Die Suchvervollständigung verhält sich ähnlich der Suchfunktion. Der Suchservice kommuniziert über die Topics *ECI* und *ECO*, die vervollständigung Anfragen für die Benutzereingabe publizieren und mögliche Eingabewünsche entgegennehmen. Die Eingabewünsche können mit Hilfe der Elasticsearch Vervollständigung oder durch die nGrams-Implementation umgesetzt werden. Diese Ansätze sind in der Granularität, Umsetzung, Laufzeit und Speicherbedarf unterschiedlich und können fallabhängig ausgetauscht werden.

## 6.4 Elasticsearch Konfiguration

Elasticsearch funktioniert *out of the box*, das heißt, man muss den Service nur starten und das Indexieren, wie das Suchen der Dokumente funktioniert reibungslos. Im Gegensatz zu den relationalen Datenbanken ist eine Schema Definition für Elasticsearch nicht notwendig. Obwohl ein Schema nicht gefordert ist, ist es höchst ratsam dieses zu definieren, denn das *Type-Matching* der Felder wird ansonsten nach dem *Best Guess* Prinzip erstellt und kann unter Umständen zum unerwünschten Verhalten führen.

### 6.4.1 Index Settings

Die *out of the box* Fähigkeit von Elasticsearch ist toll zum Ausprobieren, jedoch ist sie für den Betrieb ungeeignet, da die Einstellungen für die bestmögliche Skalierung von Elasticsearch, wie das Durchsuchen der Texte immer von der Domäne abhängt. Falsche oder keine Anpassung, kann die Skalierungsmöglichkeiten von Elasticsearch drastisch senken und damit den Betrieb unerwartet unterbrechen. Des Weiteren ist die Anpassung der Textvorverarbeitung eine Kernaufgabe jeder Such-Engine, dementsprechend sollte man sich besonders sorgfältig um diese Aufgabe kümmern, sodass alle relevanten Ergebnisse ermittelt und in einer passenden Ordnung dem Nutzer vorgelegt werden können.

### 6.4.2 Sharding & Replication

Zuerst wird der Elasticsearch Index auf Shards und Replicas aufgeteilt. Damit enthält ein Shard eine Teilinformation oder ein Replikat der Teilinformation des Indexes, der auf unterschiedliche Hardware aufgeteilt werden kann. Obwohl die Verschiebung des Shards Knotenübergreifend realisiert werden kann, kann dieser nicht in zwei neue Shards gespalten werden, wenn die Hardware an ihre Grenzen stößt. Damit ist ein Shard die Skalierungseinheit des Indexes und muss sorgfältig gewählt werden. Der uns gegebene Elasticsearch Cluster arbeitet nur auf einem Knoten, dementsprechend ist der Skalierungsfaktor von fünf genug um zukunftssicher den Index bis auf fünf weitere Knoten aufteilen zu können. Der Replikationsfaktor beschreibt wie viele Replikate für einen Shard erstellt werden. Für eine Einstellung aus fünf Shards und einem Replikat entsteht eine Gesamtdatenmenge von

10 Shards. Daraus folgt ein immenser Anstieg an Speicherverbrauch für jeden nächsten Replikat eines Shardes. Die Replikate bieten im Gegensatz die Ausfallsicherheit und die Performance Steigerung der Lesezugriffe. Da die Performance und die Ausfallsicherheit für jeder skalierbare Applikation Kernkriterien sind, kann Elasticsearch diese bei Bedarf im laufenden Betrieb durch neue Replikate stärken. In unserem Fall steht nur einen Knoten zur Verfügung, dementsprechend folgt kein Anstieg der Performance wie Ausfallsicherheit mit Hilfe der Replikation.

### Initiale Index Einstellungen

---

```
1 {
2   "twitterindex": {
3     "settings": {
4       "index": {
5         "number_of_shards": "2",
6         "number_of_replicas": "0",
7         "refresh_interval": "1s",
8         "provided_name": "twitterindex",
9         "creation_date": "1529674155106",
10        ...
11        "uuid": "wOHQnu5GQjOs7iP-Cv-_MQ",
12        "version": {
13          "created": "6020399"
14        }
15      }
16    }
17  }
```

#### 6.4.3 Textvorverarbeitung

Als nächstes wird der Textvorverarbeitungsprozess definiert. Unter dem Schlüssel *Analyzer* wird ein *Custom Analyzer* und die dazugehörigen Vorverarbeitungsschritte erstellt. Dieser wird innerhalb Elasticsearch aufbewahrt und mit Funktionalität belegt. Der Analyzer zerlegt den Text mit dem *Parti-*

al Word Tokenizer in Tokens nach einem bestimmten Muster, nämlich nach einem Leerzeichen, nach einem Großbuchstaben Anfang und nach einer Zahl. Dieser Zerlegungsschritt erfüllt die gegebenen Anforderungen, könnte aber auf Kosten des Speicherplatzes durch den mächtigeren Edge NGram Tokenizer ausgetauscht werden. Nachfolgend laufen die Tokens eine Transformationskette durch. Zuerst werden die Tokens in Kleinbuchstaben umgewandelt, auf den Wortstamm zurückgeführt und Worte mit geringen Informationsgehalt, sowie verboten Worte entfernt. Zuletzt werden die Worte mit ähnlicher Bedeutung wie Universität Hamburg und UHH in Synonymlisten zusammengefasst und als gleichgültig behandelt.

### Analyzer

---

```
1 "analyzer": {
2 ...
3   "my_analyzer": {
4     "filter": [
5       "my_tokenizer",
6       "lowercase",
7       "my_stemmer",
8       "english_possessive_stemmer",
9       "my_stop",
10      "my_synonym"
11    ],
12    "type": "custom",
13    "tokenizer": "standard"
14  },
15 ...
16 }
```

#### 6.4.4 Data Mapping

Zuletzt braucht Elasticsearch ein Datenschema, um die automatische Fehleinschätzung des Mappings zu vermeiden. Elasticsearch baut einen Suchindex auf, der Dokumentenbasiert in einem JSON Format abgespeichert wird. Das Mapping garantiert eine Typenzusicherung wie das Format, der

zu speichernden Felder und Dokumente. Zum Beispiel das Datumformat, das regionsunabhängig vor dem Abspeichern von Elasticsearch normalisiert wird oder das Textfeld, das man mit bestimmten Eigenschaften und Funktionen anreichert, um das gewünschte Verhalten zu realisieren. Der Text kann als *Keyword* abgespeichert werden, das Eins-zu-eins durchsucht wird oder als *Text*, der mit Hilfe der Volltextsuche komplexen Suchkriterien umsetzen kann. Das Elasticsearch Mapping für den Twitter-Klon definiert ein Dokumentenformat mit sieben Felder: id, message, tags, users, timeStamp, userLocation und userLocationCompletion. Für die Suche sind die Felder *message*, *tags* und *use* relevant, diese werden vom Type *Keyword* und *Text* gespeichert. Das Feld *timeStamp* wird auf ein *Long* projiziert und für die Datenvisualisierung mit Kibana genutzt, um die wöchentlichen Trends anzuzeigen. Das Feld *userLocation* hält den vom Benutzer eingetragene Standort, der mit der Textvervollständigungsinformation von Elasticsearch angereichert und durch das Feld *userLocationCompletion* beschrieben wird. Zum Vergleich wird das Feld *userLocation* zusätzlich mit dem *nGram Analyzer* belegt, um die Text-Verföllständigung von Elasticsearch mit der *nGram* Methode vergleichen zu können.

## Mapping

---

```
1 "userLocation": {  
2     "type": "text",  
3     "analyzer": "nGram_analyzer",  
4     "search_analyzer": "nGram_search_analyzer"  
5 },  
6 "userLocationCompletion": {  
7     "type": "completion",  
8     "analyzer": "simple",  
9     "preserve_separators": true,  
10    "preserve_position_increments": true,  
11    "max_input_length": 50  
12 }
```

---

## 6.5 Search Service

Die Implementation des Such Service wird mit Java und Spring realisiert. Spring ist ein weit verbreitetes und etabliertes Enterprise Framework für Java. Dieses besitzt eine große Palette an Werkzeugen, die das Arbeiten in einer heterogenen Umgebung stark vereinfachen. Daher eignet sich Spring besonders gut für unseren Anwendungsfall. Für den Nachrichtenaustausch zwischen Apache Kafka, Elasticsearch und dem Suchservice wird die von Spring entwickelte *Spring for Apache Kafka* und die von Elasticsearch angebotenen *Elasticsearch Rest Client* Bibliotheken genutzt. Die interne Logik des Suchservices wird von den Java Beans umgesetzt, die im Springkontext auf dem Apache Tomcat Application Server ausgeführt werden und die gewünschte Such-Funktionalität umsetzen.

### 6.5.1 Such-Interface

Um die Kommunikation und Fähigkeiten des Such Services übersichtlich zu gestalten, wird ein Interface mit den gewünschten Anfragen erstellt und anschließend implementiert.

1. über die Tags
2. über die referenzierten Benutzer
3. über angegebenen Standortnamen mit der Textvervollständigung
4. über die Term-Suche auf dem Tweet-Text
5. über die Volltextsuche auf dem Tweet-Text
6. über den Text wie den Benutzet Standortnamen
7. über einen Zeitraum
8. über einen Zeitraum mit Benutzer und Tag Präferenz

### 6.5.2 Such-Implementation

Elasticsearch bietet eine JSON-ähnliche domänenspezifische Sprache, mit der man Abfragen ausführen kann. Dies wird als *DSL Query* bezeichnet.

Die Abfragesprache ist ziemlich umfassend und bietet komplexe Filter und Aggregation Möglichkeiten [1]. Die Anfragen des Suchservices sind ausgelegt die wichtigsten Anfragemöglichkeiten von Elasticsearch darzustellen. Es werden *term*, *match*, *multi match*, *match phrase*, *bool*, *completion* und *aggregation* Anfragen behandelt.

### Suchen nach Tags

```
1 {
2     "query": {
3         "terms": {
4             "tags.keyword": "?"
5         }
6     }
7 }
```

### Suche nach Referenzierten Benutzer

```
1 {
2     "query": {
3         "terms": {
4             "users.keyword": "?"
5         }
6     }
7 }
```

Die Tag- und Benutzersuche wird mit der Term-Suche umgesetzt, diese sucht nach Dokumenten, die genau die im angegebenen Feld angegebenen Begriffe enthalten. Die Tweets werden entsprechend den TF/IDF Relevanz sortiert und präsentiert.

### Textvervollständigung nach Standortnamen

```
1 {
2     "suggest": {
3         "location_suggest": {
4             "prefix": "?",
5             "completion": {
```

```
6     "field": "userLocationCompletion",
7     "fuzzy": {
8         "fuzziness": 1
9     }
10    }
11 }
12 }
13 }
```

---

Die Textvervollständigung bietet Funktionen zur automatischen Vervollständigung. Es werden Vorschläge während des Tippens einer Anfrage getätigt und führt schneller zu relevanten Ergebnissen. Durch die zusätzliche *fuzzy* Eigenschaft der Abfrage werden zusätzlich Tippfehler abgefangen um die Suche den Nutzer angenehmer zu gestalten.

### Term-Suche auf den Tweet-Text

---

```
1 {
2     "query": {
3         "match": {
4             "message": "?"
5         }
6     }
7 }
```

---

### Volltextsuche auf den Tweet-Text

---

```
1 {
2     "query": {
3         "match_phrase": {
4             "message": {
5                 "query": "?",
6                 "slop": 1
7             }
8         }
9     }
10 }
```

---

Die Suche über den Textkörper eines Tweets kann auf zwei Arten getan werden. Im ersten Fall wird eine Match-Suche *match* durchgeführt, diese normalisiert die Suchterme, verknüpft sie mit *OR* und durchsucht den Textkörper nach Suchbegriff-Treffern. Im zweiten Fall nutzt man die zusammenhängende Suchanfrage *phrase match*, welche im Gegensatz zu *match* die Terme mit *AND* verknüpft und eine Einschränkungen mitbringt, nämlich die Ordnung der Suchterme im Textkörper. Um die Suche flexibler zu gestalten, kann sie aufgeweicht werden, indem man die akzeptable Entfernung der Suchterme mit der *Slope* Eingeschalt beeinflusst. In unseren Fall erlaubt diese eine Entfernung von einem Wort zwischen den Suchbegriffen.

### Standortabhängige Tweets

```
1 {
2   "query": {
3     "multi_match": {
4       "query": "?",
5       "fields": [
6         "message",
7         "userLocation"
8       ]
9     }
10  }
11 }
```

Die *multi match* Abfrage sucht nach Nachrichten, dessen Inhalt einen Ort verweist und der Autor sich in dieser Region befindet. Diese Abfrage verhält sich wie *match*, jedoch über eine Menge von Feldern.

### Suche Tweets über den Zeitraum mit referenzierte Benutzer zuerst

```
1 {
2   "query": {
3     "bool": {
4       "must": {
5         "terms": {
6           "tags": "?"
7         }
8       }
9     }
10  }
11 }
```

```
7         }
8     },
9     "filter": {
10    "range": {
11      "timeStamp": {
12        "gte": "now-1d/d",
13        "lt": "now/d"
14      }
15    }
16  },
17  "should": [
18    {
19      "term": {
20        "users": "?"
21      }
22    }
23  ]
24}
25}
26}
```

Diese Suchanfrage führt ein neues Konzept der *bool* Anfrage, die aus mehreren Komponenten besteht. Die Suche besteht aus erforderlichen Feld-Treffern wie den optionalen Feld-Treffern. Daraus ergibt sich eine Rangordnung aus den relevanten Ergebnissen. Zuletzt läuft die Liste einen Zeitfilter durch um den Zeitraum einzuschränken.

### Top Tweets für die letzten sieben Tage

---

```
1 {
2   "aggs": {
3     "top_tags": {
4       "significant_terms": {
5         "field": "tags",
6         "size": 10
7       }
8     }
9   }
10 }
```

```
8      }
9  },
10 "query": {
11   "bool": {
12     "must": [
13       {
14         "match_all": {}
15       },
16       {
17         "range": {
18           "timeStamp": {
19             "gte": "now-7d/d",
20             "lt": "now/d"
21           }
22         }
23       }
24     ]
25   }
26 }
27 }
```

Diese Anfrage ist zuständig für die Visualisierung in Kibana. In diesem Fall wird mit Hilfe der *bool* Anfrage und der Elasticsearch-Aggregation, die am häufigsten verwendeten Tag über den Datensatz von sieben Tagen erarbeitet.

## 6.6 Visualisierung mit Kibana

Die Visualisierungsmöglichkeiten von Kibana [8] sollen es ermöglichen große Datenmengen zu analysieren unterstützt durch flexible Filter. Es bietet Echtzeit-Analyse von Daten, individuell konfigurierbare Visualisierung, dynamische Dashboards und ein Browserbasiertes Interface, das Plattformunabhängige funktioniert.

Die Kibana Visualisierungen basieren auf den Aggregationsmöglichkeiten von Elasticsearch. Dieses aggregiert über den Elasticsearch Indexinhalt und erstellt Grafiken, die in HTML eingebunden werden können. Zur Verfügung stehende Visualisierungstypen: Area Chart, Data Table, Line Chart, Markdown Widget, Mertric, Pie Chart, Tile Map, Vertical Bar Chart.

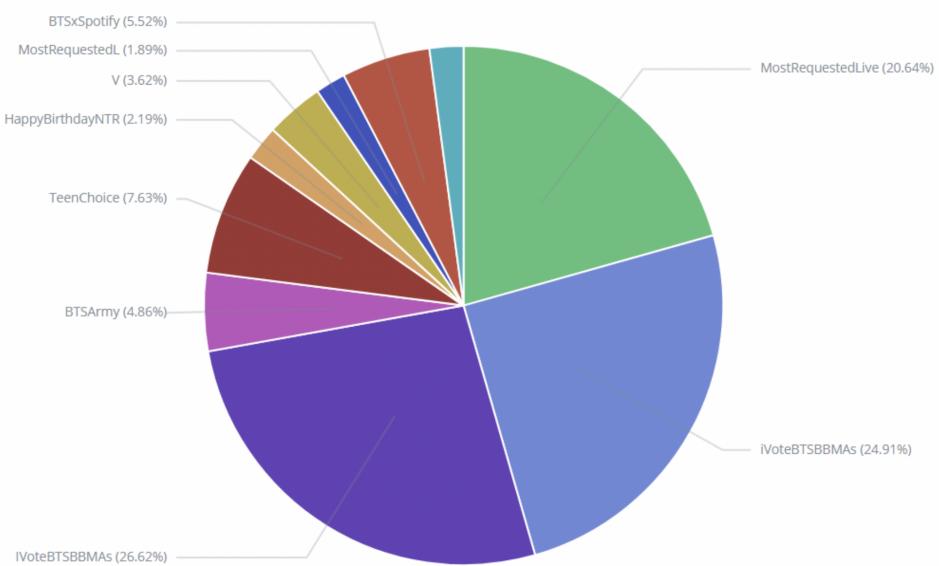


Abbildung 6.2: Kibana Analyseplattform

## 6.7 Fazit

### 6.7.1 Suchfunktionalität

Jede moderne Software bietet eine Möglichkeit nach Daten zu suchen. Mit Hilfe von Elasticsearch kann eine einfache Suche nach einer Website oder einem Dokument innerhalb einer Sammlung implementiert werden. Anschließend kann eine Rechtschreibprüfung hinzugefügt werden. Höchstwahrscheinlich ist eine fuzzy Suche und automatische Vervollständigung nötig, möglicherweise sogar während der Eingabe. Da die Relevanz wichtig ist, können fortgeschrittenen Ranking-Schemata erstellt werden. Zum Beispiel können Suchergebnisse basierend auf dem Standort, Zeit sowie des Benutzers ausgeführt werden.

Und um zu wissen, was die Benutzer tatsächlich tun, kann die Nutzung der Software protokolliert und gespeichert werden für die spätere Analyse der Nutzerdaten.

Damit ist Elasticsearch eine moderne und mächtige Such-Engine, die fortgeschrittene Suchfunktionalität anbietet und für viele Anwendungsfälle geeignet ist.

### 6.7.2 Performance

Elasticsearch hat sich als eine robuste und fähige Such-Engine bewiesen, sie konnte alle Ziele ohne Einschränkungen erfüllen und zusätzlich mit einer Vervollständigungsfunktion ergänzen. Die Index Konfiguration lässt sich einfach bedienen und über kleine Kalibrierungsschritte im JSON Format von einfachen bis komplexen Indexstrukturen erstellen. Von den Shards, Replicas bis zu den Data-Routings ist Cluster übergreifend alles möglich. Das Suchen ist eine etablierte IT-Disziplin, die mit Komfort, Kosten und Gewinn fest verbunden ist. Demgemäß ist Elasticsearch perfekt geeignet große Datenmengen zu durchsuchen und scheint mit der Anfragegeschwindigkeit, die mit jedem weiteren gefüllten Hardwareknoten die Anfragen automatisch parallelisiert. Somit bleibt die Geschwindigkeit im grünen Bereich auch nach dem Anstieg der Datenmenge. Allerdings haben die positiv gelisteten Eigenschaften ihre Tücken. Die Konfiguration des Indexes kann im Kleinen so wie im Großen geschehen. Die richtige Hardware- (RAM/SSD/HDD) wie Indexkonfiguration muss gewissenhaft gewählt werden, um die versprochenen Geschwindigkeiten zu erreichen. Dementsprechend gewinnt man Zeit durch die automatisierte Datenverwaltung und verliert durch die Elasticsearch Wartung/Feinabstimmung.

### 6.7.3 Dokumentation und API

Ebenso problematisch sind die Elasticsearch Abfragen, die einerseits gut im JSON-Format beschrieben und dokumentiert sind, jedoch in der Umsetzung, durch die von Elasticsearch zur Verfügung gestellten JAVA Bibliothek in JAVA schwer verständlich und Komplex in der Umsetzung. Erst zum Ende des Projekts bin ich auf *Mustache* gestoßen, die *JSON-Templats* für die Abfragen erstellt und diese in einer simplen Form an Elasticsearch weiterleitet.

#### 6.7.4 Tools

Besondere positiv aufgefallen ist die WebUI Kibana die mit Elasticsearch über REST Anfragen kommuniziert. Es ist möglich nach Daten zu suchen, den Clusterstatus abzufragen sowie aussagekräftige Grafiken zu erstellen. Diese Werkzeuge bieten dem Entwickler einen einfachen und übersichtlichen Einstieg in die Elasticsearch-Umgebung.

# Kapitel 7

## Analytics

Twitter hat sich bei Privatpersonen, als auch Unternehmen und Massenmedien als Kommunikationstool durchgesetzt. Die gewaltige Datenbasis die Twitter bietet, birgt dabei großes Potenzial, Erkenntnisse und Informationen über die Nutzer zu gewinnen. Vom Aufspüren von aktuellen Trends und Thematiken, über die Generierung von Stimmungsbildern zu bestimmten Inhalten, bis zu Verhaltensanalyse einzelner Nutzer sind viele Szenarien denkbar. Im Folgenden wird ein Ansatz innerhalb des Hamaube-System vorgestellt, Analysen einerseits in Echtzeit durchzuführen und andererseits Unmengen von gesammelten Daten effizient zu verarbeiten.

Alle eingehenden Twitter-Nachrichten werden durch eine Streamverarbeitung behandelt, erste Analysen durchgeführt und für eine spätere Batch-Analyse vorbereitet.

Zunächst wird die Streamverarbeitung und anschließend die Batch-Analyse vorgestellt.

### 7.1 Stream

Die Streamverarbeitung innerhalb des Hamaube-Systems ist als eigenständiger Service geschrieben. Für die Umsetzung wurde SparkStreaming im Verbund mit Scala verwendet.

Folgende Ziele haben sich während des Projekt herausgebildet:

- Vorverarbeitung der eingehenden Tweets für eine spätere Prozessierung mittels Spark

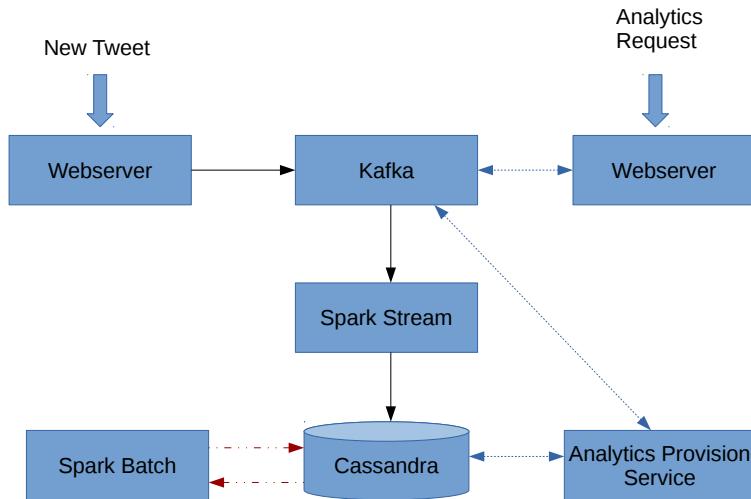


Abbildung 7.1: Übersicht der Analytics Komponenten

- Verarbeitung der Streamdaten für live Analyseergebnisse.

Als Input dienen bisher die beiden gegebenen Datenquellen:

1. Tweets, welche aus der Twitter API in unsere Anwendung gestreamt werden.
2. Hamaube-Tweets, welche in unserer Anwendung erzeugt werden.

Auch die Streamverarbeitung ist über Kafka angebunden und erhält die Tweets über das *USER\_ISSUES\_TWEET* Topic.

### Vorverarbeitung für Spark Batch Processing

Aus verschiedenen Gründen wurde für die Tweets im Hamaube-System das Datenmodell von Twitter übernommen. Dadurch enthalten Tweets (vor allem solche, die direkt von Twitter kommen) eine Vielzahl von nicht gesetzten Feldern oder Daten, welche für die Analyse wenig interessant sind. In der Vorverarbeitung wird nur ein definierter Teil der Felder eines Tweets

übernommen und der Rest verworfen. Zusätzlich werden (sofern vorhanden) die im Tweet enthaltenen Hashtags extrahiert und neben dem eigentlichen Tweet gespeichert. Außerdem wird anhand des Tweet-Textes eine Sentiment Analyse durchgeführt, welche dem Tweet einen Score zuordnet, der dessen Stimmung widerspiegelt (negativ - negative Stimmung, positiv - positive Stimmung). Auch dieser Score wird neben dem eigentlichen Tweet gespeichert.

Wir haben uns dazu entschieden, die Tweetdaten zusätzlich - also dupliziert - zu den schon gespeicherten Tweets des 'CassandraReaders' in Cassandra zu speichern, da wir in Hinsicht auf ein produktives System die kritischen Tabellen nicht zusätzlich mit Anfragen belasten wollen. Für die Analyse würde also ein separate Umgebung für Cassandra aufgebaut, dies haben im Hamaube-System aufgrund der Einfachheit noch nicht umgesetzt.

### Live Analyse

Zudem wird auf den durch Kafka eingehenden Tweetstream eine Live Analyse durchgeführt. Das Spark Streaming Framework teilt den Stream dafür in Micro-Batches, die dann weiter verarbeitet werden können. So werden z.B. die Anzahl von Hashtag pro Stunde/pro Minute ermittelt, daraus werden dann aktuell beliebte Hashtags erkannt. Auch diese Ergebnisse werden in Cassandra gespeichert. Um eine Skalierung zu ermöglichen müssen die Ergebnisse z.B. pro Stunde von mehreren Instanzen geupdated werden können, ohne dass dabei dirty reads oder Überschreibungen auftreten dürfen. Cassandra bietet hier den Datentyp **Counter**, der atomare *increase* und *decrease* Operationen anbietet. Damit lassen sich die meisten Szenarien realisieren, ohne weitere komplexe (und möglicherweise verlangsamtende) Isolierungsmechaniken zu implementieren.

### Beispiel

Der Spark Streaming Service erhält einen neuen Tweet aus dem Topic *USER\_ISSUES\_TWEET*. Dieser liegt im vom Twitter definierten Datenformat vor (Siehe 5).

Folgende Attribute werden für die Weiterverarbeitung extrahiert:

- Tweet-Id

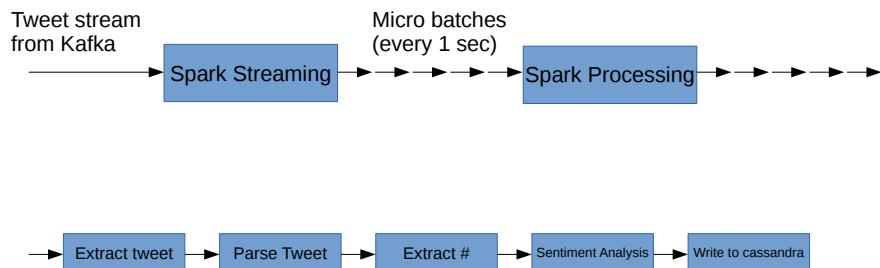


Abbildung 7.2: Spark stream processing

- Tweet-Text
- User (Id, Name)
- Sensitivitäts-Flag
- Hashtags
- Erstellungsdatum
- Sprache
- Ort (Land, Stadt)
- GPS-Koordinaten

Twitter extrahiert dabei die Hashtags aus dem Tweet-Text und schreibt diese zusätzlich in ein Entity-Objekt, welches am Tweet hängt. Diese Hashtags können einfach genutzt werden und eine Extraktion aus dem Text wird vermieden. Falls Werte nicht gesetzt sind oder ungültig sind, werden diese

durch einheitliche Dummy-Werte ersetzt, welche in späteren Analysen erkannt und übersprungen werden. Anschließend wird die Sentiment-Analyse des Tweet-Textes durchgeführt. Diese zählt anhand definierter Listen positive und negative Wörter im Text und berechnet daraus einen Score. Dieser Score kann später z.B. mit den zum Tweet gehörenden Hashtags analysiert werden, ein mögliches Szenario wäre ein Stimmungsbild zu einem bestimmten Hashtag. Damit eingehende Tweets in Zeiteinheiten aggregiert werden können, wird der Erstellungszeitpunkt eines Tweets auf entsprechende Zeit-einheiten abgerundet (z.B. Stunde, Minute). Für jede Zeiteinheit / Analyse-Target existiert in Cassandra eine Tabelle, für welche der Primary-Key als Verbund aus dem entsprechenden Analyse-Target (z.B. Hashtag) und der gerundeten Zeiteinheit vorliegt. Dadurch lassen sich Updates effizient für jede Zeiteinheit / Analyse-Target durchführen.

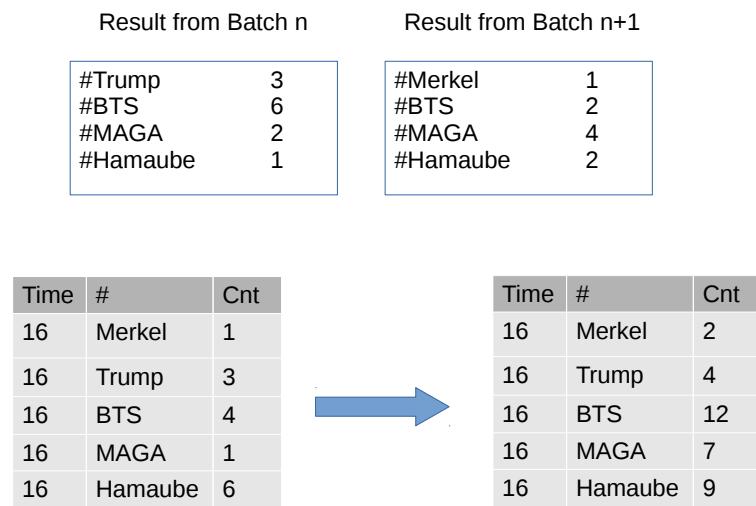


Abbildung 7.3: Sream Analytics: Beispiel - Anzahl von Hashtags pro Stunde

## 7.2 Architektur

### 7.2.1 Zwei Phasen

Die Ausführung der Analysen erfolgt in zwei Phasen. Die eine Phase ist die eigentliche Batch-Analyse. Dabei werden alle bisher in der Datenbank befindliche Tweets ausgewertet. Die andere Phase erfolgt davor. Die Realtime-Analytics-Komponente überträgt die reinkommenden Tweets in einer Art ETL-Prozess (extract, transform, load) in die Datenbank. Dort werden sie in einem für die Analyse optimierten Format gespeichert.

#### ETL-Prozess

Der ETL-Prozess wurde ursprünglich notwendig, weil der Cassandra-Connector für Spark einen Bug enthielt. Befanden sich an speziellen Stellen im orginalen JSON-Objekt von Twitter NULL-Werte und wurden diese in die Datenbank geschrieben, dann konnte der Connector diese Objekte nicht korrekt verarbeiten. Die Tweets aus der Datenbank werden auf ein Java-Objekt gemappt. Dabei wird versucht, alle Attribute der Zielklasse mit Werten zu belegen. Dafür wird in der Datenbank nach einer geeigneten, d.h. exakt gleich benannten Spalte gesucht. Dann wird abhängig vom Datentyp der Spalte der gelesene Wert in den entsprechenden Java-Typ umgewandelt. Der Bug tritt nun in der Verschachtelung auf. Die JSON-Objekte können verschachtelt werden und ebenso die Datenbank-Spalten. Dafür kann man zuerst den Datenbanktyp selbst definieren und dann kann dieser als so genannter frozen-type als normaler Spaltentyp genutzt werden. Wenn ein solches Objekt eines frozen-types nun in der Datenbank als NULL abgespeichert ist, stolpert der Connector darüber und findet keinen Umwandler von NULL zu einem entsprechenden Java-Objekt.

Wir haben dann vom direkten Lesen der Tweets aus dem Datenbankteil, in dem der Webserver sie schreibt, auf Lesen der Tweets aus einem eigenen Datenbankteil umgestellt. Die Tweets kommen über Kafka an den Realtime-Analytics-Prozess, der diese auswertet (extract). Der zugehörige Kafka-Connector enthält keinen solchen Bug.

Vor dem Laden in die Datenbank wird der Tweet nun stark reduziert, d.h. es werden viele Felder entfernt, die für die Analyse gar nicht benötigt werden. Zudem werden die NULL-Werte an den kritischen Stellen durch

Dummy-Objekte ersetzt (transform). Schlussendlich werden sie vom selben Cassandra-Connector in die Datenbank geschrieben, mit dem sie später auch gelesen werden. (load).

Neben der nun vorhandenen Kompatibilität mit dem Cassandra-Connector führt der ETL-Prozess auch zu deutlich kleineren Tweet-Objekten, was die Analyse stark beschleunigt, insbesondere wenn die Tweets über das Netzwerk übergeben werden müssen.

#### Eigentliche Analyse

Nach dem ETL-Prozess kommt die eigentliche Analyse. Der ETL-Prozess wird ständig von der Streaming-Komponente ausgeführt, d.h. es werden Tweets sekündlich in die Datenbank geladen.

Die eigentliche Analyse, eine Batch-Analyse wird regelmäßig, z.B. nachts durchgeführt. Sie dauert auch deutlich länger, sie könnte gar nicht in Echtzeit erfolgen. Ziel der eigentlichen Analyse ist es, statistische Werte über die Aktivität der User und Inhalte der Tweets zu ermitteln. Hierbei geht es um die Gesamtheit der Tweets. Des Weiteren werden weitere Parameter ermittelt und Empfehlungen vorbereitet. Diese beziehen sich auf einzelne Tweets bzw. User.

## 7.3 Skalierbarkeit

Übergeordnetes Ziel dieses Projektes ist ja die Implementierung in einer skalierbaren und erweiterbaren Architektur. Das Prinzip dafür lautet: nutze skalierbare Komponenten und verbinde sie skalierbar. Mit skalierbar ist hier scale-out gemeint, d.h. die Daten aus der Datenbank liegen auf verschiedenen Servern vor. Die Datenbank kümmert sich um die Verteilung der ihr zugeführten Daten. Cassandra skaliert durch die Verteilung der Daten auf verschiedene Server. Ähnlich skaliert Spark. Spark wird ebenso auf verschiedenen Rechner ausgeführt, die zusammen als Cluster die Analyse ausführen. Dabei werden die Daten auf dem Rechner gehalten, der auch die Analyse durchführt. So weit, so gut. Allerdings gibt es hier nun einen Stolperstein, der die Skalierbarkeit gefährdet. Die Daten liegen bisher in Cassandra und müssen zu den Spark-Rechnern übertragen werden. Hier gibt es nun drei denkbare Wege:

- die Daten werden von einem Koordinator aus Cassandra abgerufen und in Sparks verteiltes Dateisystem geladen. Offensichtlich ist dieser Koordinator ein Flaschenhals. Dieser Fall sollte dementsprechend vermieden werden, erfordert aber einen eigenen Sparkconnector zu Cassandra.
- die Daten werden von den einzelnen Spark-Rechner passend abgerufen, d.h. jeder Spark-Rechner ruft die für ihn notwendigen Daten ab und die Analyse wird so verteilt, dass diese Datenmengen untereinander überschneidungsfrei sind.
- die Berechnung folgt den Daten. Die Analyse von Spark wird auf den Rechner ausgeführt, auf denen auch Cassandra die Daten hält. Dann würden sie nur auf dem Rechner übertragen. Offensichtlich würde diese Variante besonders wenig Netzwerktraffic generieren. Allerdings würde die Rechnenlast auf den Cassandra-Knoten steigen. Wenn diese Knoten nun auch die Produktiv-Knoten für die Datenbank des Frontends wären, müsste man diese Effekte abwägen. Hat man allerdings eine eigene Datenbank bzw. einen eigenen Keyspace für die Analysen, so müsste man diese Effekte nicht gegeneinander abwägen. Und wir haben ja einen solchen Keyspace für die Analysen, der mit dem ETL-Prozess gefüllt wird.

Wir verwenden einen Cassandra-Connector in Spark, der das letzte Prinzip versucht. Er versucht die Analysen möglichst auf dem gleichen Knoten auszuführen, auf dem auch die Daten in Cassandra liegen. Möglich wird dies dadurch, dass man diese Informationen aus Cassandra abrufen kann und dass Spark die Möglichkeit bietet, die Verteilung der Analysen genau aus solchen Gründen zu beeinflussen. In unserem Testsetup funktioniert das leider nicht, da keine gesonderten Knoten für die Analytics-Cassandra-Instanz zur Verfügung stehen und Spark auf anderen Knoten läuft. Damit entspricht unsere skalierbare Verbindung der aus dem zweiten Fall.

Der dritte Fall, auch genannt Datenlokalität, ist nicht unbedingt der bessere. Nicht nur, wenn die Datenbankknoten noch mit anderen Dingen beschäftigt sind, nein auch, wenn die Analyse im Vergleich zum Laden der Daten komplexer wird, also mehr Rechenleistung benötigt wird. Ausprobiert wurde deshalb auch, wie sich unsere Analysen verhalten, wenn man sie zwar auf

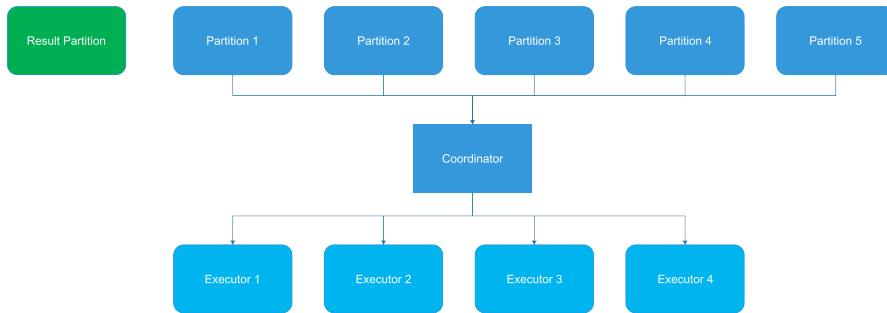


Abbildung 7.4: Spark und Cassandra verbunden mit Flaschenhals

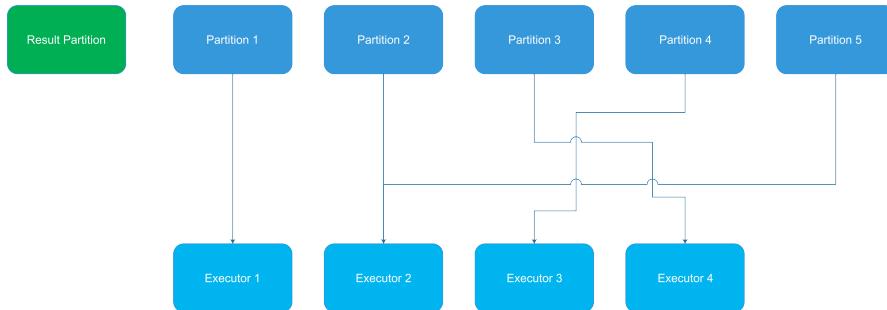


Abbildung 7.5: Spark und Cassandra durcheinander, aber parallel verbunden

mehr CPU-Kernen ausführt, dafür aber ein langsameres Netzwerk simuliert. Die Anzahl parallel laufender Einheiten ist deutlich sichtbar. Es werden so viele Einheiten gleichzeitig fertig, wie der Prozessor Kerne hat. In diesem Fall zeigt aber eine Auswertung der CPU-Last, dass die Übertragung über das Netzwerk der Flaschenhals ist, denn die CPU-Kerne sind max. zu Hälfte ausgelastet. Ist das Netzwerk schneller, dann steigt auch die CPU-Auslastung auf nahezu 100%.

## 7.4 Use-Cases

Die Batch-Analytics-Komponente hat vorrangig das Ziel, inhaltlich relevante Entwicklungen in den Tweets zu zeigen. Das wären z.B. länger trendende Hashtags zu identifizieren.

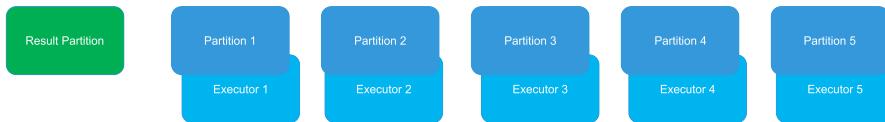


Abbildung 7.6: Spark und Cassandra mittels Data Locality verbunden

Folgende statistische Werte werden erhoben und im Frontend angezeigt:

- Gesamtanzahl Tweets
- Anzahl fröhlicher Tweets (gemessen am Vorliegen eines Smileys) und Anzahl trauriger Tweets (Vorliegen eines Sadleys)
- die Anzahl Tweets pro Tag
- die Top 10 Hashtags pro Tag
- die Top 10 Stunden mit dem meisten Tweets pro Tag
- die Top10 Tweeter pro Tag
- sowie die Top 10 verwendeten Wörter pro Tag.

Eine Vorstellung der Ergebnisse findet sich weiter unten.

Diese Daten werden von Spark ermittelt und in eine Ergebnistabelle von Cassandra geschrieben. Dort werden sie vom AnalyticsServices bei Anfrage durch den Webserver ausgelesen und an den Webserver gesendet. Dieser wiederum leitet sie an das Frontend weiter, wo sie mittels Javascript grafisch dargestellt werden.

Weiterhin sind folgende Anwendungen in Spark implementiert:

- Analyse der Tweetsprache durch Analyse der vorkommenden Stopwords
- Ermittlung von zeitlichen Aktivitätsprofilen der User
- Ermittlung von Association-Rules für das Folgen von Nutzern.

Die erste beiden Komponenten werden im Frontend nicht aktiv eingesetzt. Die letzte Komponente kann nicht sinnvoll benutzt werden, da im Gegensatz zu den vielen Tweets unsere Userdaten sehr lückenhaft sind. Es fehlen die Informationen über die gefolgten Nutzern bzw. nur ein Nutzer in unserer Datenbank hatte mehr als einen Follower. Damit ließen sich keine Association-Rules generieren. Der einzige Nutzer, dem mehr als ein Nutzer folgt (zwei Nutzer) ist der Twitter-Account der übermäßig präsenten koreanischen Boyband BTS. Sie wird später nochmal genauer angeschaut. Ein weiterer Usecase für die grafischen Analysen ist die Überprüfung der Funktionsweise des Systems. Werden die Daten aufgrund einer Schema-Änderung bei Twitter nicht mehr in die Datenbank gespeichert, so würde man dies in den Analysen sehen können. Auch ein Programmierfehler, bei dem der Benutzername nicht richtig übertragen wurde, fiel auf. Statt Tweeter mit ca. 40-80 Tweets pro Tag tauchte in den Top 10 der Tweeter nur ein Nutzer auf, nämlich NULL.

Ein dritter Usecase fällt bei der Analyse der Top-Hashtags auf. Am 27.09. z.B. ist einer der Top10-Hashtag PCA. Das steht für Peoples Choice Award. Es handelt sich hierbei um eine Online-Abstimmung, die durch Nennungen in den sozialen Medien erfolgt. Zur Auswertung ist eine solche Gesamtanalyse natürlich nötig.

### 7.4.1 Erkenntnisse durch grafisch aufbereite Statistiken

Wir haben eine einfache Sentiments-Analyse eingebaut. Ziel ist es, die insgesamte Stimmung auf einer Skala von positiv bis negativ einordnen zu können. Die Sentiments-Analyse wertet die Tweets auf das Vorkommen von Smileys :-) und Sadleys :-( aus. Die jeweilige Gesamtzahl wird dann angezeigt. Diese Methode ist offensichtlich ziemlich ungenau und das zeigt sich auch in den Ergebnissen. Am 1.10.2018 hatten wir fast 10 Millionen Tweets in der Datenbank, aber nur 2177 enthielten ein Smiley und sogar nur 366 ein Sadley.

Hier zeigt sich auch beispielhaft, warum eine Arbeitsteilung zwischen Datenbank und Analysen sinnvoll sein kann. Die Datenbank kann nämlich nicht ausgeben, wie viele Tweets in ihr speichert sind. Die entsprechende Anfrage timed out. Für andere Anfragen ist ein geeigneter Index notwendig. Dessen Notwendigkeit ist nicht immer im Voraus bekannt und eine Nachrüstung



Abbildung 7.7: BTS vor den Vereinten Nationen Quelle: [https://www.youtube.com/watch?v=ZhJ-LAQ6e\\_Y](https://www.youtube.com/watch?v=ZhJ-LAQ6e_Y)

kaum möglich.

## 7.5 Interessante Analyseergebnisse

### 7.5.1 BTS

Bei unseren Analysen war der Hashtag BTS in verschiedenen Formen immer präsent. BTS steht für Bangtan Boys und ist eine koreanische Boyband. Ihre hohe Präsenz auf Twitter ist nicht nur Folge hoher Popularität und der Auswahl durch Twitter, sondern auch Ergebnis diverser Social-Media-Kampagnen. BTS startete unter anderem die LoveYourself-Kampagne. Ziel derer ist es, die Liebe in der Welt zu verbreiten. Aussage ist es, dass für die Liebe in der Welt zuerst die Selbstliebe notwendig ist. Diese Jahr sprachen sie vor den Vereinten Nationen zu diesem Thema. Ein weiterer Hashtag aus dem BTS Umfeld ist deren A.R.M.Y, was für Adorable Representative M.C for Youth steht. Damit wird ihr Fanclub bezeichnet.

Eine These, warum es gerade BTS so konstant in unsere Tweets schafft, ist dass unser Selektionskriterium love diese LoveYourself-Kampagne trifft.

### 7.5.2 Twitter-Spam

Die Analyse der Top 10 Tweeter am 23. und 24.09.2018 zeigt ein weiteres interessantes Phänomen: den Twitter-Spam. Mit der Möglichkeit sich an bekannte Hashtags hängen zu können, zieht Twitter auch Spammer an. Am 24.09.(und am 23.09) hat es eine Gruppe von Usern geschafft, durch jeweils 30 Tweets pro Nutzer unsere Top 10 mit Porno-Spam zu füllen, mit User wie DAILY HOMEMADE, und THE PORN PROMO LIST. In den darauffolgenden Tagen tauchen sie dann nicht mehr auf, was für Gegenmaßnahmen auf Seiten Twitters spricht. Am 01.10.2018 tauchen sie dann langsam wieder in den Top 10 auf.

### 7.5.3 Trump related

In unseren Daten sind jeden Tag große Mengen von Tweets über/mit Donald Trump enthalten. Neben seiner bekannteren hohen Twitterpräsenz liegt das auch an dem Selektionskriterium, mit dem wir bei Twitter nach Tweets anfragen. Es enthält unter anderem den Selektor Trump.

## 7.6 Spezielle Analyseergebnisse

### 7.6.1 Stream-bedingte Ergebnisse

Es gibt einige Gründe, die unsere Analyseergebnisse verzerren.

Der erste Grund ist die konstante Geschwindigkeit unseres Twitterstreams. Offensichtlich verarbeiten wir nicht 100% der Tweets weltweit oder auch nur deutschlandweit. Stattdessen lassen wir uns von Twitter über deren API eine Auswahl geben. Die API liefert maximal 80 Tweets pro Sekunde aus. Wir haben diese Rate auf 2 Tweets pro Sekunde limitiert. Am 24.09.2018 kommen alle Stunden auf  $7918 \pm 5$  Tweets. Das entspricht gerade der Streamgeschwindigkeit.

Zudem lassen z.B. am 01.10 auch die Auswirkung der Analysezeit sehen. An diesem Tag lief der Stream nur zeitweise und genau diese Zeiten lassen sich erkennen. Er wurde irgendwo zwischen 10 und 11 Uhr angeschaltet und lief zwischen 11 und 12 wieder vollständig (ca 7200 Tweets). Nach 12 Uhr wurde er wieder unterbrochen und ca.zwischen 14:30 und 15:40 lief er dann wieder.

### Sampling verzerrt die absoluten Zahlen

Bei manchen Analysen, z.B. Tweets pro Tag stimmen die absoluten Zahlen nicht. Am 24.09.2018 weist die Grafik 353 Tweets aus. Wir wissen aber durch die Auswertung über die Stunden das es ca. 172800 Tweets sein müssten (24 Stunden x 7200 Tweets pro Stunde). Das liegt daran, dass für die Grafik ein Sampling der Tweets genommen wurde, d.h es wurde nur ein Bruchteil analysiert. Während der Entwicklung dient das dazu, dass die Analysen schnell ausgeführt werden können (ca. 1 Minute vs. 20 Minuten). Im Produktivbetrieb würde man dieses Sampling dann nicht mehr vornehmen.

## 7.7 Genauere Schritte

Ein großes Hindernis bei der Entwicklung und dem Betrieb der Analysen war es, dass die Beispiele und gefundenen Howtos zu einfach sind. Im Sinne einer gewissen Wissensweitergabe sollen hier anhand einiger Patterns – analog zu Design Patterns – die gewonnenen Erkenntnisse weitergegeben werden.

### 7.7.1 Doppelter Key

Eine Herausforderung war es, dass der Mapping-Schritt zwar ein Key-Value-Mapping vornimmt, aber nur einen Key unterstützt. Für die Analysen wird aber häufig ein mindestens doppelter Key benötigt. Zum Beispiel wenn die Top 10 Tweeter pro Tag bestimmt werden sollen. Dann sieht ein typisches Key-Value-Paar eigentlich so aus (Tag, Tweeter → 1). Die beiden Angaben für den Key lassen sich einfach aus den Tweets extrahieren. Nun müssen sie noch verbunden werden. Die einfachste Möglichkeit, die funktioniert hat, ist die beiden Angaben jeweils in einen String zu exportieren und die beiden Strings zu konkatenieren. Damit die beiden Teile später wieder getrennt werden können, muss noch ein eindeutiges Trennzeichen eingefügt werden. Kein einzelnes Zeichen kann dafür verwendet werden, da sie alle potentiell in den Tweets vorkommen können. Wir verwenden deshalb eine lange, zufällige Zeichenkette, wie sie so wohl kaum in einem Tweet vorkommen wird.

Eine weitere Methode wäre es, eine eigene Klasse für das Key-Value-Paar zu erstellen. Hierbei ist darauf zu achten, sowohl equals() als auch hashCode() korrekt zu implementieren.

Eine dritte Möglichkeit wäre es, das in Scala eingebaute Tuple2 zu verwenden.

Keine Möglichkeit war es, bereits zu Anfang die Daten aufzuteilen und zuerst nach Tagen zu filtern und dann 14 Analysen laufen zu lassen. Das würde zu lange dauern, da Spark die Daten dann zu häufig anfordern würde (nämlich neu für jeden Tag).

Auch deshalb wäre eine bessere Unterstützung durch Spark hier sinnvoll. Denn es gibt einen weiteren Fall, der ganz ähnlich ist.

### 7.7.2 Top10 pro Tag nehmen

Unsere Analysen sind häufig pro Tag. Das bedeutet, dass wir die Top 10 für die letzten 14 Tage zum Beispiel nehmen wollen. Wir brauchen also 14 Top 10s. Die eingebaute Sortierung kann einem hier nur teilweise helfen. Sortiert man wie normal nach Values, so ist nicht garantiert, dass sich die Top 10 der 14 Tagen überhaupt am Anfang findet. Werden an einem Tag besonders viele Tweets abgesendet, so verdeckt das natürlich die anderen Tage. Die sortierte Liste enthält dann viele Einträge der aktiveren Tage und die Top 10 der weniger aktiven Tage findet sich weiter hinten.

Die einzige korrekte Lösung, die wir gefunden haben, ist es sich die gesammelte Liste nach dem Reduce auf einem Knoten zusammenführen zu lassen und sie dann von oben nach unten zu iterieren. Bei der Iteration übertragen wir das gesichtete Element die Top10 des entsprechenden Tages, so lange der Tag nicht schon 10 Einträge in der Liste hat.

Dieses Verfahren ist allerdings unnötig wenig parallelisiert. Würde man die Daten früh nach Tagen shuffeln, so könnte man diese Operationen zumindest nach Tagen parallelisiert auf unterschiedlichen Knoten ablaufen lassen.

Wahrscheinlich gibt es dafür eine Lösung in Spark. Eine einfache Unterstützung für die Unterteilung von Daten in einzelne Klassen und dann Durchführung der selben Operation für alle Klassen separat wäre trotzdem sehr hilfreich.

### 7.7.3 Bausteine

Man sagt, dass Spark mit dem Map-Reduce-Paradigmum verwendet wird. Die Daten werden eingelesen und auf ein Ergebnis oder einen Fall gemappt und diese dann aggregiert (reduce). Während der Arbeit an den Batch-

Analytics wurde deutlich, dass etwas mehr Schritte vorgenommen werden müssen. Konkret fielen die Schritte:

- Daten laden

Die Grundoperation für das Laden der Daten ist beim verwendeten Cassandra-Connector der Table-Scan. Komplexere Abfragen lassen sich (noch) nicht ausführen. Damit kann keine Selektion nach einem Kriterium erfolgen (zum Beispiel Tweets der letzten vierzehn Tage). Diese Selektion muss dann im Filter-Schritt erfolgen.

- ggf. samplen

Gerade während der Entwicklungszeit ist es ratsam, darüber nachzudenken zuerst die Analysen mit einem (sehr) kleinen Ausschnitt der Daten durchzuführen. Dafür kann das entstandene RDD entweder gesampelt werden oder n Elemente davon genommen werden (take). Das erste Fall ist durch den Salt für den Pseudo-Zufallsgenerator deterministisch, das zweite nicht. Das erste dauert ähnlich lange wie das Laden der ganzen Daten, insbesondere ist es unabhängig von dem Anteil der zu holenden Daten. Das zweite, das Taken, erfolgt viel schneller. Hier werden keine weiteren Daten mehr übertragen, sobald die gewünschte Anzahl erreicht ist. Die Daten könnten aber weniger stark durchmischt sein.

Da unsere Operationen nicht besonders teuer sind, hat das Samplen kaum einen Geschwindigkeitsvorteil gebracht. Für das reine Testen der Funktionalität konnte das Taken aber mit hohem Geschwindigkeitsvorteil eingesetzt werden.

- Filtern

Dieser Schritt ist deshalb nötig, weil die Grundoperation der Table-Scan ist. Es können aber auch unnötige Tweets aussortiert werden, z.B. Tweets, die nicht zu einem gewissen Thema gehören.

- Mappen

Der bekannte Map-Schritt. Häufig werden mehrere Eigenschaften der Gesamtheit untersucht. Das lässt sich dadurch bewerkstelligen, dass man ein Vektorobjekt mit Feldern für alle untersuchten Eigenschaften anlegt und für diese separat die Werte berechnet.

- Reducen

Der bekannte Reduce-Schritt. Hat man während des Mappen-Schritts ein Vektorobjekt angelegt, so muss man hier auch wieder eine Vektorreduktion vornehmen.

- Ergebnis produzieren

Nach dem Reduce liegen noch ggf. viele Daten vor. Bei der Analyse der Top Hashtags liegen z.B. zu allen überhaupt vorkommenden Hashtags die Häufigkeiten vor. Diese müssen jetzt noch weiterverarbeitet werden, indem z.B. die gewünschten Daten extrahiert werden (Top 10 der letzten vierzehn Tage).

- und Ergebnis abspeichern

Die Daten müssen noch zum Zwecke der Persistenz in Cassandra geschrieben werden.

auf.

## 7.8 Verbesserungsvorschläge

Bei der Arbeit an der Spark-Komponente sind einige Verbesserungsvorschläge aufgekommen. Man sieht deutlich, dass es sich bei Spark und Cassandra noch nicht um jahrzehntelang erprobte und verfeinerte Tools handelt.

### 7.8.1 Frühere Fehlererkennung

Ein tolles Feature wäre eine frühere Fehlererkennung. Bisher fallen gerade Flüchtigkeitsfehler erst bei der Ausführung auf. Das betrifft vor allem die zusätzlichen Aufgaben, die durch Spark und die Datenbank hinzukommen. Beispielsweise müssen alle Datenklassen Serializable implementieren. Meist reicht es, den Zusatz implements Serializable hinzuzufügen. Vergisst man das, so merkt man erst bei der Ausführung der Analyse. Ein weiteres Beispiel betrifft das Mapping zur Datenbank. Vertippt man sich hier z.B. beim Spaltenname, so merkt man das erst bei der Ausführung. Eine zusätzliche,

automatische Analyse vor oder nach dem Kompilieren wäre sehr hilfreich. Gerade das lange Packen (90 Sekunden) könnte so wegfallen. Sie könnte z.B. die Datenbank bereits kontaktieren und das Mapping ausprobieren oder durch statische Codeanalyse Fehler wie die fehlende Serializability finden.

### 7.8.2 Inkrementelles Packing

Eine andere Verbesserung wäre das inkrementelle Packen. Ändert man einen kleinen Fehler in einer Klasse wird die Fat-Jar mit allen Abhängigkeiten erneut gebaut. Allerdings würde der Austausch der veränderten Klassen ausreichen. Wenn ein solches inkrementelles Packen möglich wäre, dann würde dies eine große Unannehmlichkeit beheben. In der Scala-Variante scheint dies zu erfolgen, hier braucht das Packen nicht so lange.

### 7.8.3 Automatisches Zusammenlegen von Analysen

Die dritte Verbesserung betrifft die Ausführung mehrerer Analysen. In unserem Fall werden sehr ähnliche Analysen auf den gleichen Daten ausgeführt. Das Laden der Daten aus der Datenbank dauert allerdings verhältnismäßig am längsten. Hier wäre es zu begrüßen, wenn die Analysen automatisch zusammengelegt werden können. Gerne kann man das auch im Code explizit aktivieren müssen.

Ein Beispiel ist das Zählen von Smileys und Sadleys. Dies könnte ohne Problem gleichzeitig erfolgen. Es ist nicht erforderlich, zuerst alle Daten aus der Datenbank zu holen und die Smileys zu zählen und dann wieder alle Daten aus der Datenbank zu holen und jetzt die Sadleys zu zählen.

Natürlich lässt sich das auch manuell im Code machen, indem man einen Analysevektor für die Values erzeugt. Das macht aber die Kopplung im Code wieder höher und macht z.B. beim Sortieren Probleme, da eigene Comparatoren neu geschrieben werden müssen oder die Daten vorher wieder zerlegt werden müssen.

Würde das automatisch erledigt werden, könnten die Analysen insgesamt deutlich schneller werden, da die Daten nur einmal aus der Datenbank geholt werden würden.

## 7.9 Probleme

In diesem Abschnitt soll auf die häufigsten und/oder nervigsten Probleme eingegangen werden.

### 7.9.1 Sehr lange Ausprobierzeiten

Bedingt durch mehrere Effekte dauert es vom Ändern des Codes bis zur fehlerhaften oder erfolgreichen Ausführung vergleichsweise lange. Klar lässt sich eine Analyse mit vielen Daten nicht innerhalb kürzester Zeit ausführen, trotzdem gibt es einige Punkte, die unnötige Wartezeiten zur Folge haben:  
Das Packen der Jar nach dem Kompilieren dauert jeweils 1:30 Minute  
Unabhängig davon, wie viel Code geändert wurde, ist das zu lange. Die resultierte Fat-Jar ist über 170 Megabyte groß. Deshalb spricht viel dafür, dass zu viele unnötige Abhängigkeiten mitgepackt werden und/oder dieser Prozess inkrementell beschleunigt werden könnte.

#### Spark-Job mit vielen Daten dauert sehr lang

Nachdem man die lange Packzeit überstanden hat, dauert das Ausprobieren der Analyse mit einer gut gefüllten Datenbank lange, häufig zwischen 10 und 40 Minuten. Dies ist insbesondere beim Debugging der späteren Phasen nervig. Abhilfe kann hier nicht so einfach geschaffen werden. Je mehr Daten, desto länger dauert die Analyse. Allerdings sollte das Sampling, also die künstliche Verkleinerung der Menge der untersuchten Daten, Abhilfe schaffen. Wirklich besser wird es allerdings nur mit der Variante `take(int n)`, aber nicht mit dem Sampling.

#### Und kann bei einem Fehler auch nicht so einfach debugged werden

Weiterhin fehlt noch ein Debugger. Trifft Spark auf eine Exception so probiert er es entweder noch einige Mal erneut oder bricht die ganze Analyse ab (abhängig vom Typ der Exception). Eine einfache Möglichkeit dieses Datum zu überspringen (z.B. wenn es einen unverarbeitbaren NULL-Wert enthält) oder zur nächsten Analysen überzugehen, wäre hilfreich.

### 7.9.2 Schwache Dokumentation von Spark

Das mit Abstand nervigste Problem ist allerdings die schwache Dokumentation von Spark und dem Cassandra-Connector.

Die API ist zum Teil nahezu unkommentiert (z.B. die API zu JavaRDDs). Sprechende Methodennamen taugen hier nur teilweise. Werden bei takeOrdered z.B. die größten oder kleinsten Elemente genommen? Und wie nehme ich die jeweils andere Variante?

Antwort: takeOrdered nimmt die n kleinsten Elemente. Herausgefunden habe ich das durch Ausprobieren. Später habe ich die Antwort auch an einer anderen Stelle in der Dokumentation gefunden, nämlich in der Dokumentation zur Klasse RDD<T>. Das wäre kein wirkliches Problem, wenn die Methoden klar als geerbt von RDD<T> gekennzeichnet wären.

Zurück zum Problem. Wir wollen die Top 10 Hashtags nehmen. TakeOrdered nimmt gerade die falschen Elemente. Müssen wir nun einen eigenen Comparator schreiben. Nein. Nach einiger Recherche findet man heraus, dass die gesuchte Methode top() heißt. Ein Querverweis wäre hier ebenfalls sehr hilfreich.

Ein weiteres Beispiel ist der Stopwords-Checker (StopwordsRemover) aus der Maschine-Learning-Bibliothek. Obwohl er eine Methode loadDefaultStopWords(String language) mit dem Kommentar Loads the default stop words for the given language. hat, konnte ich ihn nicht dazu bewegen, eine andere Sprache als Englisch zu untersuchen. Die Methode kann mit „German“ ohne Fehler aufgerufen werden. Mittels stopWords() (The words to be filtered out.) wollte ich nun die Stopwords für Deutsch erhalten. Erhalten habe ich wieder nur die Stopwords für Englisch. Schlussendlich habe ich im Quelltext die Listen mit den Stopwords gesucht und auch gefunden. Dann habe ich sie extrahiert und mein eigenen Stopwordschecker darauf aufbauend gebaut.

#### Beispiele sind meist Minimalbeispiele

Ähnlich verhält es sich mit den Beispielen in der Dokumentation. Sie helfen nicht dabei, komplexere Analysen zu schreiben und das wäre nötig, denn die Beispiele sind eigentlich nur Minimalbeispiele.

Beispiel Frequent Pattern Mining aus der Machine Learning Lib. Das Beispiel hat 3 Transaktionen mit insgesamt 4 unterschiedlichen Elementen (Zahlen 1, 2, 3, 5). Was, wenn ich nun etwas anderes als Zahlen nutzen möchte? Wie ist das Format dann?

Noch problematischer ist aber die Rückgabe der Association-Rules. Laut Quelltext erfolgt die mit `model.associationRules.show()`. Der Datentyp wird aber nicht erklärt. Laut Dokumentation der API gibt es für die Klasse von `model` aber kein Member `associationRules`. Es wäre allerdings nötig, die Rules zu speichern und später wieder anzuwenden. Erst eine Suche auf StackOverflow liefert etwas mehr Informationen:

Brief explanation: `model.associationRules` gives you a Dataframe with three columns: antecedent, consequent and confidence.

Aber auch hier ist das Format nicht genauer erklärt. So verliert man viel Zeit mit Ausprobieren und weiterem recherchieren.

Dazu kommen viele kleine Gemeinheiten, die den Entwicklungsprozess verlangsamen. Möchte man die Ergebnisse sortieren, die man mittels `collect()` bekommen hat, bekommt man zwar eine `Java.Util.List`. Behandelt man sie jetzt wie erwartet und ruft z.B. `sort()` auf, so erhält man die Exception `OperationNotImplemented`. Dies ist besonders nervig, da das erst nach der Analyse auffallen kann.

### 7.9.3 Weitere Probleme

#### Bug im Cassandra-Connector

Besonders nervig war der Bug in Cassandra-Connector, der bereits im Abschnitt zum ETL-Prozess beschrieben wurde. Mittlerweile scheinen gefixte Versionen vorzuliegen.

#### Weiterentwicklung der Tools

Nicht zu unterschätzen ist auch die Weiterentwicklung der Tools, da sich die APIs zum Teil drastisch ändern können. Spark z.B. hat während der Laufzeit des Projekts seine API von RDDs auf die inkompatiblen DataFrames umgestellt. Obwohl diese sicherlich ein bedeutender Fortschritt sind, würde dies

deutliche Änderungen im Code bedeutet, ggf. sogar eine Neuentwicklung.  
Wir konnten für das Projekt allerdings noch bei RDDs bleiben.

Der Cassandra-Connector für Spark wurde mehrfach geupdated, allerdings ohne wirkliche Einflüsse.

Eine Ausnahme gibt es: Cassandra hat ihr Kommunikationsprotokoll in der Nähe zu unserem Projektanfang radikal neuentwickelt. Es läuft nun auf einem anderen Port und ist inkompatibel. Anfängliche Versuche mit einer älteren Version des Connectors aus den Howtos schlug somit fehl. Erst nach einigem Suchen ließ sich ein funktionsfähiges Minimalprojekt finden.

## 7.10 Daten: Input und Output

### 7.10.1 Die verwendete Tabelle für die Tweets:

```
1 CREATE TABLE twitter_spark.tweets (
2     id text PRIMARY KEY,
3     hashtags list<text>,
4     score int,
5     tweet frozen<tweet>
6 ) WITH bloom_filter_fp_chance = 0.01
7     AND caching = { 'keys': 'ALL', '
8         rows_per_partition': 'NONE' }
9     AND comment = ''
10    AND compaction = { 'class':
11        'org.apache.cassandra.db.compaction.
12        SizeTieredCompactionStrategy',
13        'max_threshold': '32', 'min_threshold': '4' }
14        AND compression = { 'chunk_length_in_kb': '64',
15        'class': 'org.apache.cassandra.io.compress.LZ4
16        Compressor' }
17        AND crc_check_chance = 1.0
18        AND dclocal_read_repair_chance = 0.1
19        AND default_time_to_live = 0
20        AND gc_grace_seconds = 864000
21        AND max_index_interval = 2048
```

```

19     AND memtable_flush_period_in_ms = 0
20     AND min_index_interval = 128
21     AND read_repair_chance = 0.0
22     AND speculative_retry = '99 PERCENTILE';

```

Bis auf die Attribute und deren Datentyp sowie den PRIMARY KEY haben wir die anderen Werte durch Cassandra eingesetzt lassen.

In der Tweets Tabelle wird folgender Tweet Type verwendet:

```

1 CREATE TYPE twitter_spark.tweet (
2     id varint,
3     id_str text,
4     user text,
5     text text,
6     created_at bigint,
7     lang text,
8     possibly_sensitive boolean,
9     place_type_country text,
10    place_type_place_name text,
11    place_type_country_code text,
12    coordinates_l list<varint>,
13    hashtags list<text>
14 );

```

### 7.10.2 Zeilen für Datenbank (Output)

Es gibt für jede Analyse eine eigene Output-Tabelle. Sie sind sehr ähnlich. Beispielsweise hier die Tabelle für die Top 10 Tweeter.

```

1 CREATE TABLE analyticsresults.top10tweeter (
2     analyticstime text,
3     topthing text,
4     number int,
5     version text,
6     PRIMARY KEY (analyticstime, topthing)

```

```

7 ) WITH CLUSTERING ORDER BY (topthing ASC)
8     AND bloom_filter_fp_chance = 0.01
9     AND caching = { 'keys' : 'ALL' , '
10        rows_per_partition' : 'NONE' }
11     AND comment = ''
12     AND compaction = { 'class' :
13         'org.apache.cassandra.db.compaction.
14             SizeTieredCompactionStrategy' ,
15         'max_threshold' : '32' , 'min_threshold' : '4' }
16     AND compression = { 'chunk_length_in_kb' : '64' ,
17         'class' : 'org.apache.cassandra.io.compress.LZ4
18             Compressor' }
19     AND crc_check_chance = 1.0
20     AND dclocal_read_repair_chance = 0.1
21     AND default_time_to_live = 0
22     AND gc_grace_seconds = 864000
23     AND max_index_interval = 2048
24     AND memtable_flush_period_in_ms = 0
25     AND min_index_interval = 128
26     AND read_repair_chance = 0.0
27     AND speculative_retry = '99PERCENTILE' ;

```

Hier wurden wieder die meisten Einstellungen durch Cassandra automatisch erzeugt. Die Spaltenbennung ist generisch, damit die Webserver-Zulieferkomponente diese auch generisch übertragen kann, d.h. dass nur der entsprechende Tabellenname benötigt wird und einzelnen Anfragen sonst für alle Top 10s gleich laufen können.

Die Ergebnisse der Streamanalyse werden je nach Zeitauflösung in verschiedene Tabellen geschrieben. Folgend die Tabelle für die stündlichen Ergebnisse:

```

1 CREATE TABLE twitter_spark.hourly_counters (
2     period_start_ts timestamp,
3     hashtag text,
4     count counter,

```

```
5     score counter,
6     PRIMARY KEY (period_start_ts, hashtag)
7 ) WITH CLUSTERING ORDER BY (hashtag ASC)
8     AND bloom_filter_fp_chance = 0.01
9     AND caching = { 'keys' : 'ALL' , ,
10        'rows_per_partition' : 'NONE' }
11     AND comment = ''
12     AND compaction = { 'class' : 'org.apache.cassandra
13         .db.compaction.SizeTieredCompactionStrategy',
14         'max_threshold' : '32' , 'min_threshold' : '4' }
15     AND compression = { 'chunk_length_in_kb' : '64' , ,
16        'class' : 'org.apache.cassandra.io.compress.LZ4
17         Compressor' }
18     AND crc_check_chance = 1.0
19     AND dclocal_read_repair_chance = 0.1
20     AND default_time_to_live = 0
21     AND gc_grace_seconds = 864000
22     AND max_index_interval = 2048
23     AND memtable_flush_period_in_ms = 0
24     AND min_index_interval = 128
25     AND read_repair_chance = 0.0
26     AND speculative_retry = '99PERCENTILE' ;
```

## 7.11 Aufbau des Systems

Insgesamt fanden sich beim Aufbau des Analysensystems folgende Schritte:

- Spark-Runner konfigurieren

Zuerst muss eine Konfiguration für Spark angelegt werden. Zum einen unter welcher IP sich der Koordinator finden lässt, zum anderen unter welcher IP und unter welchem Port sich Cassandra finden lässt.

- Datenmodell festlegen

Dann muss das Datenmodell festgelegt werden. Das bedeutet zum einen Java-Klassen zu bauen, die der Tabelle in der Datenbank entsprechen. Dafür

müssen alle gewünschten Spalten exakt so benannt werden, wie sie in der Datenbank benannt sind. Zum anderen müssen sie kompatible Datentypen besitzen. Erst nach längerer Suche ließ sich die entsprechende Liste beim Hersteller des Cassandra-Connectors finden. Zudem müssen diese Klassen Serializable sein, da sie über das Netzwerk verschickt werden. Dafür müssen sie und alle verschachtelten Memberklassen das Interface Serializable implementieren. Gleich geht man mit den Ausgabedaten vor, nur das diese dann nicht aus der Datenbank gelesen werden, sondern in die Datenbank geschrieben werden. Implementieren die Klassen das Interface nicht, so bricht Spark die Ausführung später mit einem Fehler ab. Gerade, wenn sich der Fehler in der Ausgabedatenklasse befindet, ist das besonders ärgerlich, da die Analyse dann schon länger lief und die Ergebnisse verloren sind.

- Analysen schreiben

Dann kommt der Schritt, in dem die einzelnen Analysen geschrieben werden.

- Jar kompilieren

Hat man die Analysen geschrieben, so müssen sie kompiliert werden und als Jar gebunden werden. Der letzte Schritt ist nötig, da das Programm von Spark auf die einzelnen Knoten verteilt wird. Die Abhängigkeiten müssen in der Jar enthalten sein, sonst werden sie wahrscheinlich auf den anderen Knoten nicht gefunden. Das Packen der Jar kann je nach Abhängigkeiten sehr lange dauern. Bei uns dauert es ca. 90 Sekunden. In dieser Zeit ist man zum Nichtstun verdammt.

- die Jar als Spark-Job übergeben

Dann wird die kompilierte Jar über das bei Spark mitgelieferte Skript submit-job an Spark zur Ausführung übergeben.

## 7.12 Phasen des Spark-Jobs

Um die Analyse der von Spark erzeugten Logs zu einer Aufführung zu erleihen , zählen wir hier die wichtigsten Phasen auf:

- Hochfahren

## 7.13. Analysebilder

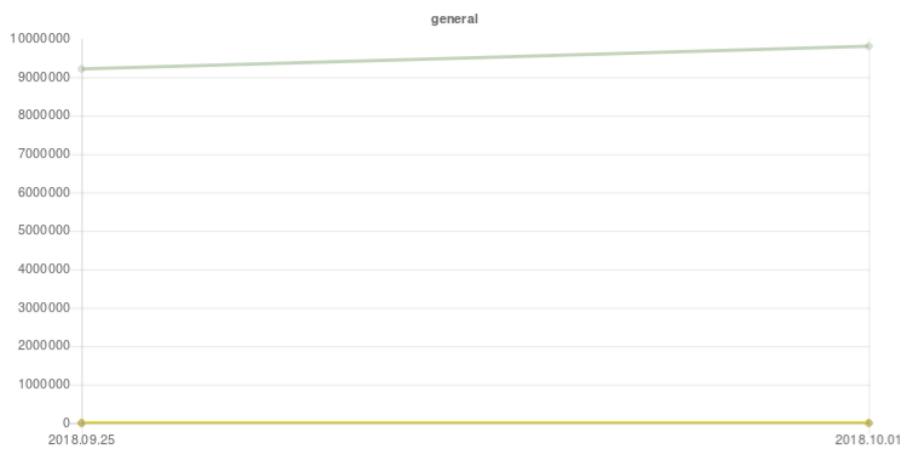


Abbildung 7.8: Anzahl Tweets

- Planung
- Durchführung in Blöcken
- Ergebnisse anzeigen
- Ergebnisse schreiben

## **7.13 Analysebilder**

## 7.13. Analysebilder

---

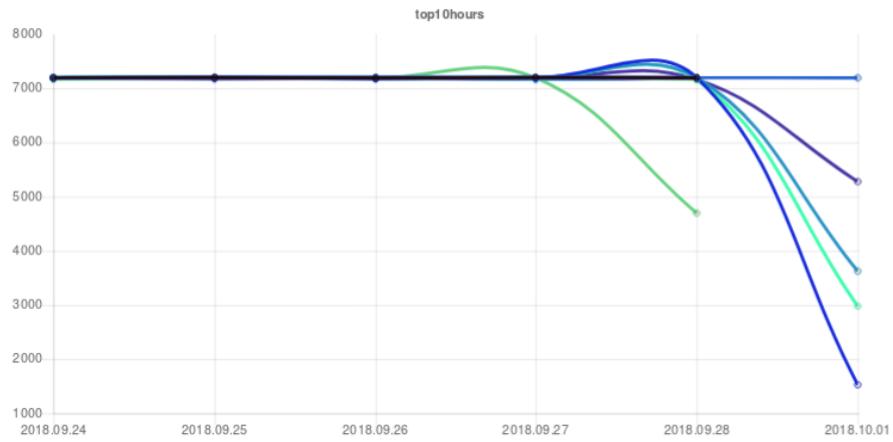


Abbildung 7.9: Top 10 Stunden

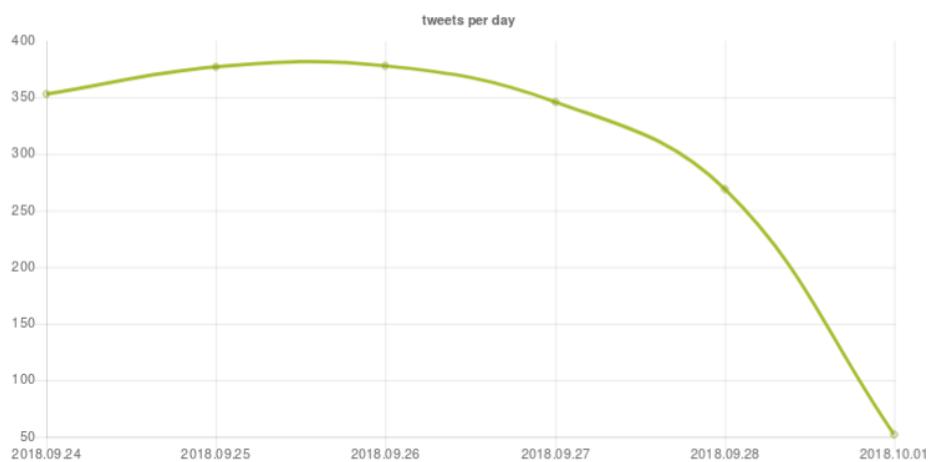


Abbildung 7.10: Tweets pro Tag (mit Sampling)

## 7.13. Analysebilder

---

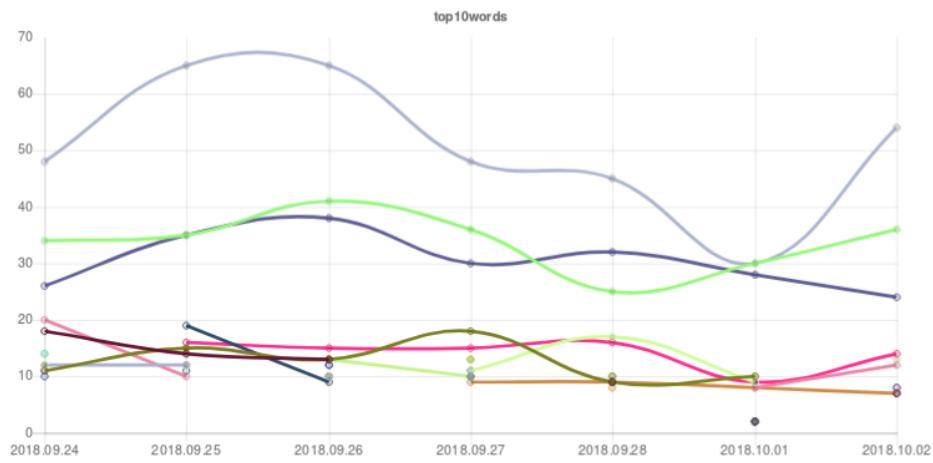


Abbildung 7.11: Top 10 Wörter

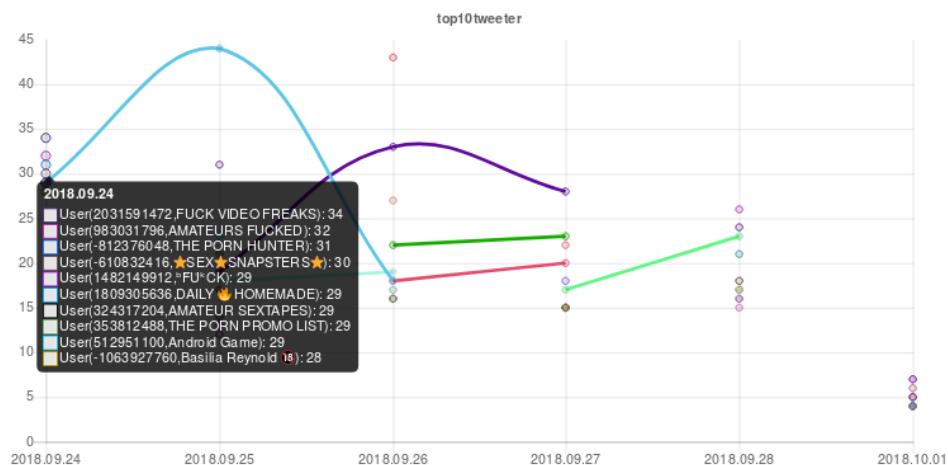


Abbildung 7.12: Top 10 Tweeter (inkl. Porno-Spam)

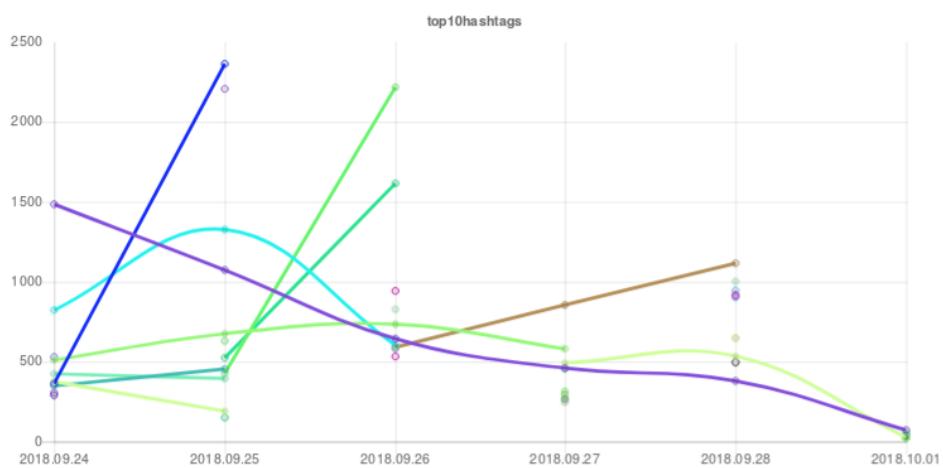


Abbildung 7.13: Top 10 Hashtags

## 7.14 Analytics Provisioning Service

Für die Bereitstellung der Analysedaten ist der Micro Service *Analytics Provisioning Service* zuständig. Dieser liest auf Anfrage Analysedaten aus Cassandra und bereitet diese für die Darstellung im Frontend auf. In der Batch-analyse werden die Ergebnisse geordnet in einer Tabelle pro Analyse gelegt, diese werden einfach nur für die letzten 14 Tage gelesen. Ergebnisse, die weiter in der Vergangenheit liegen, werden zur Zeit nicht bereit gestellt, dies ist theoretisch aber möglich. Da Cassandra keine Counter als sekundäre Keys unterstützt und nicht für Sortierungen von großen Datenmengen ausgelegt ist, die Analysedaten aus dem Stream jedoch unsortiert und nicht gefiltert vorliegen, war der naive Ansatz diese mittels einer user-defined function in Cassandra pro Anfrage zu sortieren. Dies ist aber mit steigenden Tweetzahlen sehr aufwändig und könnte z.B. durch weitere Umsetzungen mit Spark ersetzt werden.

Die Kommunikation passiert auch hier über Kafka und festgelegte Topics. Es lassen sich beliebig viele Instanzen des Services starten, die Kommunikation skaliert automatisch über die gewählte Kommunikationsarchitektur.

Für die Umsetzung wurde die Platform Node.js mit TypeScript verwendet.

## 7.15 Darstellung im Frontend

Für die Darstellung im Frontend wurde ein eigener Bereich auf der Website eingerichtet, der wiederum in die drei Teile *Live Analytics*, *Batch Analytics* und Analysen von ElasticSearchs *Kibana* aufgeteilt ist. Die Analysen durch Spark werden mittels des Frameworks *Graph.js* dargestellt.

## 7.15. Darstellung im Frontend

---

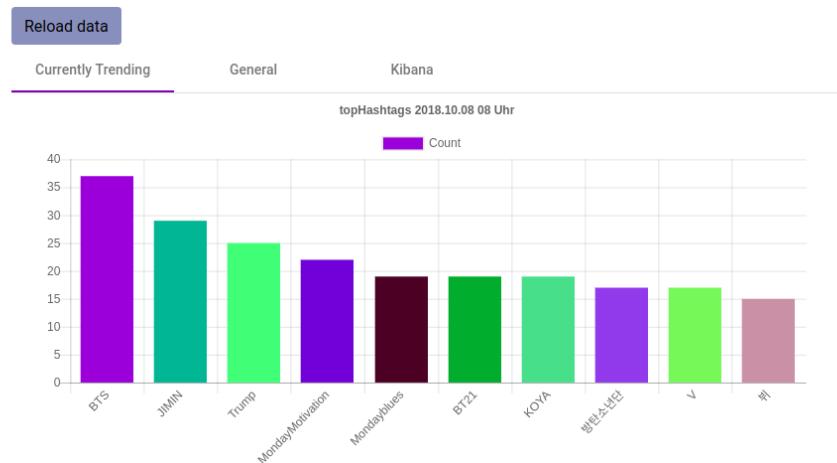


Abbildung 7.14: Live Analytics im Frontend

# Kapitel 8

## Zusammenfassung

Mit der Hamaube ist ein voll funktionsfähiger Prototyp eines skalierbar und erweiterbaren Twitter-Klons in dem NoSQL-Projekt entstanden. Er kann unter <http://nosqlprojekt.de> im VPN des Fachbereichs Informatik genutzt werden. Dort findet ein Frontend zur Hamaube.

Die Hamaube wurde in einem mächtigen Technologiestack aus Kafka, Cassandra, Elasticsearch und Spark mit Hilfe von Mikroservices implementiert. Die Architektur wurde so gewählt, dass zum einen Skalierbarkeit durch Scale-Out möglich ist als auch Erweiterbarkeit und Wandelbarkeit der Anwendung mit geringen Mehraufwand erreicht werden kann.

Die Hamaube hat während des Entwicklungsprozess ihre Erweiterbarkeit unter Beweis gestellt. Funktionalität konnte kontinuierlich mittels Kafka leicht hinzugefügt und geupdatet werden. Ebenso konnte die Kommunikation zwischen den Mikroservices leicht durch Hinzufügen von Listenern überprüft werden. Dabei spielte es keine Rolle, ob die Dienste auf den virtuellen Maschinen liefen oder von den Entwicklungsnotebooks gestartet wurden. Durch Kafka war dies für die anderen Komponenten transparent.

Der ultimative Scale-out-Test fehlt noch. Die Komponenten sind zwar skalierbar und auch die Mikroservices skalierbar ausgelegt und mit den Komponenten verbunden. Da die virtuellen Maschinen aber nur auf einem Server liegen, konnte der ultimative Lasttest nicht erfolgen. Getestet wurde das System mit 80 Tweets pro Sekunde über einige Tage. Dies konnte es gut verarbeiten. Der Test wurde wegen zumeist gehender Festplattenkapazität dann beendet. Große Datenmenge und eine hohe Geschwindigkeit machten

---

dem System in diesem Test keine Probleme.

Mit der Hamaube haben wir ein sehr mächtiges Architekturparadigmum kennengelernt. Wir haben gelernt, wie man komplexe Anwendungen sowohl in ihre einzelnen Funktionalitäten zerlegt, als auch wie diese in entsprechenden skalierbaren Tools implementiert werden. Diese einzelnen Komponenten haben wir dann zu einem funktionierenden Gesamtsystem zusammengebaut. Wir haben gelernt, wie sich Anwendungen entwickeln lassen, die sowohl im Sinne von Big Data als auch in flexibler Langlebigkeit bestehen können.

Natürlich gab es auch einige Hindernisse. Zum einen sind die eingesetzten Technologien vergleichsweise neu und sind dementsprechend noch nicht über viele Jahrzehnte gereift, so wie es SQL-Datenbanken sind. Von Version zu Version kann es dadurch zu größeren Änderungen in der API kommen. Zum Teil sind diese Tools auch noch unvollständig dokumentiert. Das Debugging einer solchen verteilten Anwendung ist deutlich komplexer als bei einer zentralisierten Anwendung. Es fehlen sowohl Tools als auch lässt sich das Gesamtsystem nicht einfach synchron anhalten.

Die Verwendung von autonomen Datenquellen, in unserem Fall den Tweets von Twitter über die Twitter-API bringt weitere Herausforderungen mit sich. Das System wird so nie fertig, da sich die Datenquelle jederzeit ändern kann. Dadurch entsteht eine Notwendigkeit für Flexibilität. Damit kommt die verwendete Architektur gut klar. Allerdings wird auch deutlich, dass auch bei Mikroservices eine gewisse Kopplung (Abhängigkeit) der Komponenten untereinander herrscht.

Als zukünftige Aufgabe lässt sich das automatische Deployment der Services benennen. Hier existieren bereits heute einige vielversprechende Ansätze und Tools. Dann könnte man mit viel mehr Hardware auch den ultimativen Scale-Out-Test durchführen.

Vielen Dank dir Steffen für das interessante Projekt. Wir haben viel Spaß gehabt.



# Anhang A

## Cassandra

Cassandra ist eine NoSql Datenbank, die nach dem Wide-Column Model konzipiert ist. Cassandra vereint verschiedene Eigenschaften von Googles BigTable [2] und Amazons Dynamo [6]. Sie wurde 2010 von Facebook entwickelt, um das Inbox Search Problem zu lösen [11]. Es ist eine verteilte Datenbank, die sich unter dem CAP-Theorem als AP charakterisieren lässt, wobei man das Level an Konsistenz manuell einstellen kann.

### A.1 Daten Model

Das Daten Modell von Cassandra entspricht dem von BigTable, also dem Wide-Column Modell. Dies beruht im Wesentlichen, wie alle Arten an NoSql Datenbank Modellen auf Key-Value-Stores. Dabei wird für einen Primary Key jeweils eine Reihe mit Column-Families definiert. Die Column-Families bestehen wiederum aus einem Key, der sie beschreibt, und einem Value der dann den Wert angibt. Der Wert kann allerdings wieder eine Menge an Column-Families sein, wodurch es möglich ist Column-Families beliebig zu verschachteln. Es wird bei der Initialisierung angegeben welchen Typ der Wert hat, Verschachtlung wird über benutzerdefinierte Typen erzeugt. Column-Families, die wiederum Column-Families beinhalten, bezeichnet man als Super-Column-Families. Bei Cassandra werden bei der Initialisierung einer Tabelle, die auch nichts anderes ist als eine Super-Column-Family, alle möglichen Column-Families angegeben. Sie definiert darüber hinaus den Primary Key über den auf die Werte der Column-Families zuge-

KeySpace: "Versicherung"				
Column Family: "Personal"				
	Name	Abteilung	Gehalt	Spalten-Schlüssel
p1	Müller	Schadensregulierung	65000	
p2	Maier	Schadensregulierung	50000	Spalten-Wert
p3	Huber	Personalabteilung	55000	
p4	Schmidt	Vorstand	100000	Bonus

Zeilen-Schlüssel (Sortiert)

Quelle: <http://wi-wiki.de/doku.php?id=bigdata:spaltendb>

Abbildung A.1: Beispiel Daten Modell

griffen werden können. Im Unterschied zu herkömmlichen SQL Tabellen ist es aber möglich Werte für diese zu unterschlagen, wie in Abbildung A.1 für p1 - p3 dargestellt.

Ein KeySpace stellt die oberste Schicht des Datenmodells dar. Für den KeySpace werden bei der Initialisierung eine Replikationsstrategie und eine Anzahl Replikas angegeben, die zu erstellen sind. Diese gelten dann für alle Tabellen, die unter dem KeySpace erstellt werden.

## A.2 System

Implementiert ist Cassandra in Java. Darauf aufbauend sind auch die Basis Java Typen verfügbar. Die Daten werden von Cassandra redundant auf verschiedene Instanzen verteilt. Systemnachrichten werden dabei über UDP verschickt, Anwendungsnachrichten, also Nachrichten, die mit den Daten zu tun haben, per TCP um den Verlust von Nachrichten zu vermeiden. Bei den Systemnachrichten ist dies zu verkraften.

### A.2.1 Partitionierung

Die Partitionierung orientiert sich an der von Dynamo [6]. Cassandra benutzt genauso wie Dynamo Consistent-Hashing, um Daten auf die verschiedenen Instanzen zu verteilen. Dabei erhalten die verschiedenen Instanzen einen Wert, der sie uniform über einen vordefinierten Wertebereich verteilt, wie in Abbildung A.2a abgebildet. Consistent-Hashing macht aus dem Wertebereich einen Ring, über den dann die Daten auf die Instanzen wie folgt verteilt werden. Aus einem Datum wird über die Hashfunktion ein Hashwert berechnet. Das Datum wird dann auf der Instanz abgespeichert, deren Wert auf dem Ring aufsteigend als nächstes kommt. Dieses Verfahren kann zu einer ungleichen Verteilung der Daten auf die Instanzen führen, sodass dadurch die Performance des Systems ineffizient wird. Cassandra löst diese Problem anders als Dynamo dadurch, dass die Werte der Instanzen an die Verteilung der Daten angepasst werden, wie in Abbildung A.2b dargestellt. So sind zwar einige Instanzen für einen größeren Wertebereich zuständig, andererseits kommen in diesem größeren Wertebereich weniger Daten vor, sodass die Daten uniform auf die Instanzen verteilt werden. Wird der Datensatz zu groß, skaliert Cassandra, indem eine Instanz im Consistent-Hashing Ring zwischen den Knoten mit den meisten Daten eingefügt wird. Danach werden die Bereiche wieder so angepasst, dass alle Instanzen ungefähr gleich belastet sind.

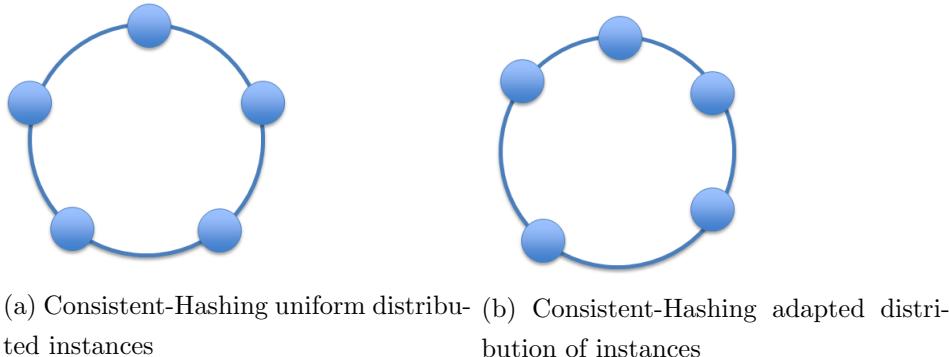


Abbildung A.2: Consistent-Hashing Ring

### A.2.2 Replikation

Die Art der Replikation in Cassandra ist vom Benutzer konfigurierbar. Die Anzahl der Replikas und die Replikationsstrategie wird durch den KeySpace festgelegt. Dabei kann man zwischen SimpleStrategy und NetworkTopologyStrategy auswählen. Die SimpleStrategy repliziert ohne auf die Netzwerkstruktur einzugehen. Somit beugt sie weniger stark potentiellem Datenverlust vor und sollte daher nur für Test-Zwecke benutzt werden. Sei  $N$  die Anzahl Replikas, werden die Daten immer auf die  $N-1$  Nachfolgeknoten repliziert. Bei der NetworkTopologyStrategy wird die Hierarchie von Datacentern und drin enthalten Racks bei der Verteilung betrachtet wird. Somit wird diese Strategie auch für das Deployment empfohlen. Innerhalb dieser Strategie kann man sich wiederum zwischen Rack Aware und Datacenter Aware entscheiden. Dabei werden die Replikas entweder auf verschiedene Racks oder Datacenter verteilt, um die höchst mögliche Datensicherheit zu gewährleisten. Diese Strategien beziehen sich auf den Koordinator, also die Hauptinstanz einer Partition von Daten, da dieser für die Replikation zuständig ist. Bei der Bestimmung eines Koordinators wird Zookeeper verwendet. Dadurch sind alle Netzwerkänderungen und -konfigurationen persistent gespeichert, da Zookeeper die Konfigurationen jedes Knotens automatisch persistent speichert. Zur Kommunikation werden bei Zookeeper Gossip Algorithmen verwendet.

### A.2.3 Persistenz

Persistenz erreicht Cassandra über ein ähnliches System wie BigTable [2]. Zunächst gibt es die MemTable, die im Hauptspeicher gehalten wird und als Cache fungiert. Sie besitzt eine Konfiguration einer Schranke, ab der die MemTable auf die Platte persistiert wird. Auf der Platte gibt es die SSTable, Bloom Filter, index file, compression file und statistics file. Die Daten werden in eine SSTable geschrieben, also eine eigene Datei geschrieben, wenn sie noch nicht vorhanden sind. Wenn dies nicht der Fall ist, wird der betreffende Teil einer SSTable in die MemTable geladen, alle Operationen ausgeführt und die Daten wieder zurück auf die Platte geschrieben. Der Bloom Filter verhindert unnötige Lookups in nicht relevante SSTables. Der Index beschleunigt den Lookup innerhalb einer SSTable. Damit man

nicht viele kleine SSTable-Dateien hat werden zwei SSTable, durch einen merge-Prozess immer dann zusammengefasst, wenn eine mindestens halb so groß ist wie die andere. Vorhandene SSTable Files können zusätzlich noch komprimiert werden.

### A.3 CQL

Mit CQL (Cassandra Query Language) gibt es eine auf Cassandra zugeschnittene SQL-ähnliche Abfragesprache, die es den Anwendern konventioneller Datenbanken deutlich leichter macht mit Cassandra zu arbeiten. Dabei ist es wichtig zu wissen, dass CQL bei weitem nicht so ausdrucksstark ist wie SQL. Das liegt daran, dass CQL im wesentlichen eine abstrakte API für das Cassandra Datenmodell darstellt. In CQL sind normale Datenbank Typen wie int, text, etc. möglich, allerdings kann man auch von Collections wie List, Set und Map Gebrauch machen, da es dafür direkte Java Typen gibt. Des Weiteren ist es möglich eigene Typen zu definieren, wie schon in Abschnitt A.1 beschrieben.

Tabellen und Spalten werden wie ebenso in Abschnitt A.1 beschrieben durch

company   name   age   role			
company	name	age	role
OSC	eric	38	ceo
OSC	john	37	dev
RKG	anya	29	lead
RKG	ben	27	dev
RKG	chad	35	ops

	eric:age	eric:role	john:age	john:role
osc	38	dev	37	dev

	anya:age	anya:role	ben:age	ben:role	chad:age	chad:role
rkg	29	lead	27	dev	35	ops

Quelle: <https://www.youtube.com/watch?v=UP74jC1kM3w>

Abbildung A.3: CQL Mapping

Column-Families dargestellt. und wie in SQL erzeugt. Dabei wird ein Primary Key benötigt, der dann als Row Key fungiert. Es kann nur über diesen Row Key auf die Zeilen zugegriffen werden. Deshalb kann man in der WHERE-Klausel in CQL auch nur Elemente des Primary Key angeben.

Auf diesen Elementen wird durch die Indizierung schon beim Speichern in Cassandra eine Sortierung berechnet was die Anfragen deutlich performanter macht. Für alle anderen Spalten der Zeile wäre dies also nicht performant und wird von CQL nicht gestattet. Die Spalten und Zeilen werden wie folgt auf das Cassandra Datenmodell abgebildet.

Der erste Teil des Primary Keys bildet, wie man sehen kann, den Row Key, die restlichen Teile werden mit ihren Werten in die Beschreibung der Column-Families integriert, so wie im unteren Teil in Abbildung A.3 beschrieben. Somit werden alle Zeilen mit dem gleichen ersten Teil des Primary Keys in der gleichen Zeile abgespeichert. Über dieses Mapping ist es möglich eine tabellenartige Abstraktion zu erzeugen, die sich durch CQL ausdrückt und Anwendern ein aus SQL bekanntes Interface bietet.

# Anhang **B**

## Big Table

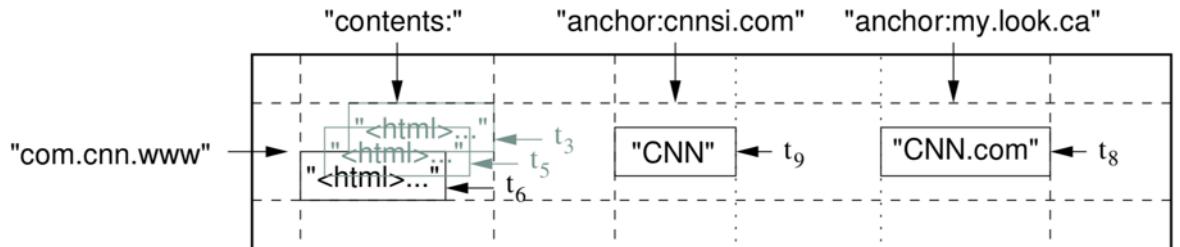
### **B.1 Einführung**

Täglich kommen mehrere Petabytes an Daten von über 60 Google Anwendungen zusammen. Dafür verantwortlich sind mehr als 1000 Computer die untereinander vernetzt sind. Um diese Daten verwalten zu können wurde Bigtable ins Leben gerufen. Das Ziel der Datenbank war es in vielen Bereichen angewendet werden zu können. Dazu sollte es skalierbar sein sowie eine hohe Performance und Verfügbarkeit besitzen.

### **B.2 Datenmodell**

Bigtable ist eine verteilte, persistente multidimensionale sortierte Map. Diese Map ist indexiert über eine row key, column key und einem timestamp. Jeder Wert in dieser Map ist ein Array mit Bytes. Das folgende Datenmodell soll eine Speicherung von Webseiten veranschaulichen.

Das Datenmodell besteht aus zwei Familien dem Inhalt und den Ankerpunkten. Die erste Familie beinhaltet den Inhalt der Webseite, mit drei unterschiedlichen Zeitstempeln (t3, t5, t6). Drei unterschiedliche Zeitstempel bedeutet, dass die Website [www.cnn.com](http://www.cnn.com) in drei unterschiedlichen Versionen abgespeichert wurde. Die Anker Familie beinhaltet jeweils nur eine Version. Den Anker mit „CNN.com“ mit dem Zeitstempel t8 und dem Anker „CNN“ mit dem Zeitstempel t9.



Quelle:  
<https://static.googleusercontent.com/media/research.google.com/de//archive/bigtable-osdi06.pdf>

Abbildung B.1: Tabellen Beispiel

**Rows** Bigtable speichert Daten in lexikographischer Reihenfolge und sortiert diese nach Zeilen. Eine row range beinhaltet alle gleichnamigen URL's, so dass alle mit der gleichen Domain zusammen abgespeichert werden. Das vereinfacht die Analyse und das Hosting der gleichnamigen Domains und macht dies zudem effizienter.

**Column families** Verschiedene column keys werden in eine gemeinsame Gruppe gespeichert. Das nennt man column families. Alle Daten, welche in der gleichen Gruppe gespeichert werden sind vom gleichen Typ. Bevor Daten in einer Gruppe gespeichert werden können, muss diese column family als erstes erstellt werden.

**Timestamp** Jede Zelle in Bigtable kann mehrere Versionen der gleichen Daten beinhalten. Diese Versionen werden indexiert durch den Zeitstempel (timestamp). Der Zeitstempel ist bis auf die Micro Sekunde genau. Durch die „two per-column-family“ kann der Benutzer festlegen, wie viele Versionen der gleichen Daten gespeichert werden sollen. Alle weiteren Versionen werden automatisch gelöscht.

**API** Die API von Bigtable ermöglicht das erstellen und löschen von Tabellen und Spaltennamen, sowie das ändern von Tabellen, Cluster und Metadaten einer Spaltenfamilie. Das folgende Codebeispiel wurde in C++ geschrieben und verändert den Inhalt der Tabelle Webtable.

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Quelle:

<https://static.googleusercontent.com/media/research.google.com/de//archive/bigtable-osdi06.pdf>

Abbildung B.2: Zugriffs Beispiel mit der BigTable API

Der Code verändert die Spalte „com.cnn.www“ und fügt einen neuen Anker hinzu. Im nächsten Schritt wird ein vorhandener Anker „anchor:www.abc.com“ gelöscht und festgeschrieben.

**Building Blocks** Bigtable ist auf mehreren anderen Teilen der Google-Infrastruktur aufgebaut. Zum Beispiel benutzt Bigtable das verteilte Google File System (GFS). Welches für die Speicherung von Logs und Daten verantwortlich ist. Ein Bigtable Cluster operiert typischerweise auf einem verteilten Pool von Computern. Auf diesem Pool laufen eine breite Sparte von verschiedenen Anwendungen. Bigtable basiert auf einem Cluster Management System für Zeit-Planung-Jobs, Ressourcenmanagements auf geteilten Computern, agieren mit Computerfehlern und dem Anzeigen des Computer Status. Das SSTable Format stellt eine persistente, geordnete unveränderliche Abbildung von Schlüsseln zu Werten zur Verfügung, wobei sowohl Schlüssel als auch Werte willkürliche Bytefolgen sind. Operationen werden bereitgestellt, um den Wert zu suchen, der einem bestimmten Schlüssel zugeordnet ist.

**Implementation** Bigtable besteht aus drei Haupt Komponenten, einer Bibliothek, einem Master Server und vielen weiteren tablet servern. Die Bibliothek ist in jeden Client verlinkt, somit kann der Client auf alle Funktionen zugreifen. Der Master Server wird zufällig ausgewählt. Dieser teilt den tablet servern die tablets zu. Außerdem ist er für die Verteilung der

Lasten und für die garbage collection zuständig. Sobald eine Tabelle zu groß wird, wird diese von einem tablet server gesplittet. So wird sichergestellt, dass eine Tabelle nie größer als 100-200 MB ist.

**Tablet location** Um Daten zu speichern, verwendet Google bei Bigtable eine „three-level hierarchy“. Das erste Level ist eine Datei, welches auch das Chubby file genannt wird, dort wird der Speicherort des root tablets hinterlegt. Das root tablet beinhaltet alle Speicherorte aller tablets in einer METADATA Tabelle. Das spezielle an dieser Tabelle ist, dass egal wie groß sie wird, diese niemals geteilt wird. Somit wird sichergestellt, dass die „three-level hierarchy“ eingehalten wird. Die METADATA Tabellen speichern die Orte aller anderen tablets in einer Tabelle ab.

**Tablet Zuweisung** Jedes tablet ist zu einem Zeitpunkt immer nur einem tablet server zugeordnet. Der Master server verfolgt die lebenden tablet server und die aktuell zugeordneten tablets zu den tablet servern inklusive aller unzugeordneten tablet servern. Beim Start einer Bigtable führt der Master Server folgende Schritte aus:

1. Wählt einen einzigartigen Master Lock in Chubby
2. Scannt die Server Verzeichnisse um die lebenden tablet server zu finden
3. Kommuniziert mit den vorhandenen tablet server um bereits zugeordnete tablets zu identifizieren
4. Master scannt METADATA Tabelle um die vorhandene Zugehörigkeiten zu lernen

**Tablet serving** Ein persistenter Zustand eines tablets wird in GFS gespeichert. Alle Updates werden auf „well-formed“ geprüft und anschließend in einem commit-log gespeichert. Die neusten Updates werden in eine memtable gespeichert, ältere updates werden in die SSTable geschrieben. Wenn Daten aus dem tablet server abgefragt werden muss ein merge zwischen den neuen Daten in der memtable sowie den älteren Daten in der SSTable erstellt werden.

**Compaction** Je mehr Daten gespeichert werden, umso größer wird die memtable. Damit diese tabelle nicht zu groß wird, gibt es ein „minor compaction“. Diese Funktion friert eine memtable ein sobald diese eine bestimmte Größe erreicht hat und erstellt eine neue memtable. Die gefrorene memtable wird zu einer SSTable konvertiert. Je mehr Daten gespeichert werden, desto unordentlicher wird die Ansammlung von SSTable. Um die SSTable zu sortieren wird periodisch ein „merging compaction“ ausgeführt. Dies strukturiert die SSTable neu und es werden Ressourcen, durch die Löschung von Daten, freigegeben. Außerdem werden gelöschte Daten endgültig gelöscht, das ist wichtig für Services, welche sensible Daten beinhalten.

## B.3 Refinments

Um die hohe Performance, Verfügbarkeit und Zuverlässigkeit beizubehalten, werden einige Verbesserungen (refinments) benötigt.

**Lokale Gruppen** Gruppierung erspart Zugriffszeit. Zum Beispiel bei dem Datenmodell Webseite. Die „page metadata“ und „content“ der Webseite werden in einer anderen Gruppe gespeichert. So muss eine Anwendung, welche nur die Metadaten möchte, nicht durch den kompletten Inhalten einer Seite iterieren. Zudem gibt es Tuningparameter, welche bestimmten ob Daten in den Arbeitsspeicher geladen werden um die Zugriffszeit zu minimieren.

**Kompression** Ein Benutzer kann selbst bestimmen ob eine SSTable komprimiert wird und falls ja, in welchem Ausmaß. Jeder SSTable Block kann einzeln ausgewählt werden. Für die Komprimierung kommen die Verfahren Bentley und McIlroy's zum Einsatz. Diese können mit 100-200Mb/s kodiert und mit 400-1000 MB/s enkodiert werden.

**Caching für Lesezugriffe** Für das Caching von Lesezugriffe gibt es zwei Verfahren. Der Scan Cache (high-level), speichert key-value Paare und liefert eine SSTable zurück. Das Block Cache (lower-level) Verfahren speichert SSTable Blocks, die von der GFS gelesen werden.

**Bloom Filter** Benutzer legt selbst fest ob ein Filter zum Einsatz kommt. Der Vorteil eines Filters liegt darin, dass wenn Daten gesucht werden, nicht

jede SSTable nach den bestimmten Daten durchsucht werden muss. Ein Bloom Filter erlaubt es, nach einer bestimmten Art von row/column-Paaren zu fragen, ob diese in einer SSTable gespeichert sind.

**Beschleunigte tablet Wiederherstellung** Wenn der Master ein tablet von einem Server zu einem anderen Server verschiebt, führt der Ursprungsserver erst ein „minor compaction“ aus um die Ladezeit für den neuen tablet server zu verkürzen.

**Unveränderlichkeit ausnutzen** Es können nur Daten verändert, welche in der memtable stehen. Daten in der SSTable können nicht verändert werden. Das macht man sich zu nutzen indem man keine Synchronisation braucht, wenn auf die Daten zugegriffen wird. Memtable sind die einzigen Daten auf die man schreiben kann und gleichzeitig lesen. Damit es zu keinen Konflikten kommt, setzt Bigtable hier auf „Copy-on-write“.

## B.4 Performance Auswertung

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

Quelle:

<https://static.googleusercontent.com/media/research.google.com/de//archive/bigtable-osdi06.pdf>

Abbildung B.3: Performance Übersicht

Die Performance wird hauptsächlich durch die verwendete CPU (2ghz) begrenzt. Zudem kann man erkennen, dass bei einem tablet server der Durchsatz bei ca. 75MB/s liegt ( $1000 \text{ bytes} * 64 \text{ KB Block size} = 75 \text{ MB/s}$ ). Damit der Durchsatz bei einem Single tablet server erhöht wird, wird die die SSTable Größe in der Regel von 64KB auf 8kb gesenkt. Zudem wird erkannt, dass

#### B.4. Performance Auswertung

der Durchsatz nicht linear ansteigt. Bei einer Erhöhung der tablet server von eins auf 500 liegt die Erhöhung des Durchsatzes bei gerade mal dem 300 fachen ( $10811 / (500 * 6250) = 350$ ). Diese Begrenzung liegt wie bei einem tablet server an der CPU der tablet servern.

# Anhang C

## Dynamo

### C.1 Problemstellung und Einordnung

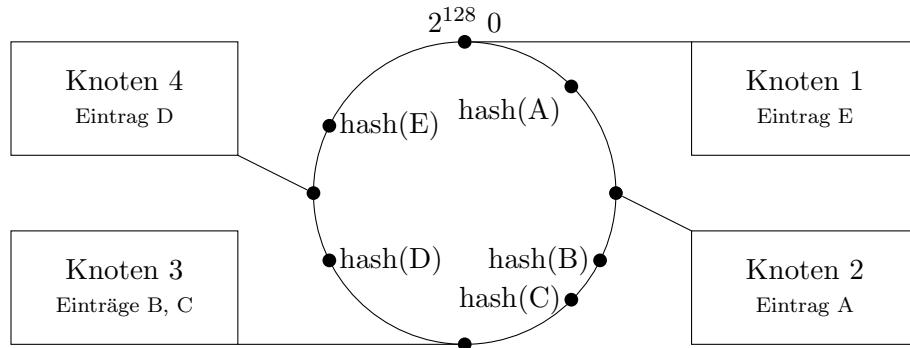
Amazon wickelt Bestellungen von Kunden rund um die Welt ab. Hierfür benötigt Amazon ein hochverfügbares System, das sicherstellt, dass jederzeit Bestellungen entgegen genommen werden können. Es werden also viele Rechenzentren benötigt, um diese Last zu tragen. Bei der Auswahl von Rechnern für diese Rechenzentren setzt Amazon allerdings nicht auf hochverfügbare Spezialhardware, sondern setzt handelsübliche Rechner ein. Um dennoch eine möglichst hohe Verfügbarkeit garantieren zu können, hat Amazon die verteilte Datenbank Dynamo entwickelt, die in [6] beschrieben wurde und in diesem Kapitel vorgestellt werden soll.

Dynamo ist dafür konzipiert, bei Verfügbarkeit einer minimalen Anzahl an Knoten, Schreibvorgänge annehmen zu können. Dynamo ist also partititionierungstolerant und hoch verfügbar, kann Datenkonsistenz aber nur unter bestimmten Bedingungen garantieren.

### C.2 Datenmodell

Dynamo ist ein Key-Value-Store. Werte werden also unter einem Schlüssel abgelegt und können nur über diesen Schlüssel wieder abgefragt werden. Komplexere Anfragen sind daher nicht möglich. Der Wert, der hinterlegt wird, muss keinem festen Datenschema entsprechen - es können beliebige

Abbildung C.1: Beispiel für konsistentes Hashing. Die Daten sollen auf vier Knoten verteilt werden, wobei die Knoten gleichmäßig auf dem Kreis platziert sind.



Binärdaten gespeichert werden.

## C.3 Architektur

### C.3.1 Konsistentes Hashing

Um die Daten den Knoten zuzuordnen, verwendet Dynamo konsistentes Hashing. Hierbei wird für jeden Schlüssel ein MD5 128 Bit Hash-Wert gebildet. Den Knoten wird ebenfalls ein Wert zwischen 0 und  $2^{128}$  zugeordnet. Ein Schlüssel-Wert-Paar wird auf dem Knoten gespeichert, dem der nächst höhere Wert ausgehend vom Hash-Wert des Schlüssels zugeordnet wurde. Gibt es keinen Knoten, der einen höheren Wert hat, wird das Schlüssel-Wert-Paar auf dem Knoten mit dem kleinsten Wert gespeichert.

Visuell lässt sich dieses Verfahren wie in Abbildung C.1 mithilfe eines Kreises darstellen. Jeder Position auf dem Kreis wird im Uhrzeigersinn beginnend am obersten Punkt des Kreises ein Wert von 0 bis  $2^{128}$  zugeordnet. Knoten wie Schlüssel können dann anhand ihres Hash-Wertes auf dem Kreis platziert werden. Ein Schlüssel-Wert-Paar wird auf dem Knoten gespeichert, der auf dem Kreis an der nächsten Stelle im Uhrzeigersinn liegt.

### C.3.2 Replikation

Replikation lässt sich nun leicht realisieren, indem die Einträge nicht nur auf dem nächsten, sondern bei einem Replikationsfaktor von  $n$  auf den nächsten

$n$  Knoten gespeichert werden. Fällt dann ein Knoten aus, lassen sich die Einträge weiterhin auf dem jeweils nächsten Knoten auf dem Kreis auffinden. Bei einem Ausfall muss dafür gesorgt werden, dass die Bedingung, dass alle Daten auf den nächsten  $n$  Knoten gespeichert sind, wieder hergestellt wird.

### C.3.3 Partitionierung und Verteilung

Bei der Partitionierung aus Abbildung C.1 ist beim Ausfall eines Knotens das Problem, dass die nächsten  $n$  Knoten (bei Replikationsfaktor  $n$ ) die volle Last des Ausfalls tragen müssen, da die Daten, die der ausgefallene Knoten gespeichert hat, auf die nächsten  $n$  Knoten repliziert werden müssen. Ist  $n$  im Vergleich zur Anzahl der insgesamt verfügbaren Knoten eher klein, wäre es wünschenswert eine bessere Verteilung der Last zu erzielen. Daher platziert Dynamo die Knoten nicht direkt auf dem Kreis, sondern plaziert stattdessen virtuelle Knoten. Dabei stehen mehrere virtuelle Knoten für einen realen Knoten. Nun können die virtuellen Knoten so auf dem Kreis verteilt werden, dass auf die virtuellen Knoten eines realen Knotens die virtuellen Knoten verschiedener realer Knoten folgen. Bei dem Ausfall eines realen Knotens wird die entstehende Last also auf viele oder sogar alle realen Knoten verteilt.

In Abbildung C.1 wurden die Knoten gleichmäßig auf dem Kreis verteilt. Es ließe sich nun ein Schema finden, das für virtuelle Knoten eine Verteilung auf dem Kreis bestimmt, durch die die virtuellen Knoten ebenfalls gleichmäßig auf dem Kreis verteilt werden, und dafür sorgt, dass der Ausfall eines Knotens zunächst von allen anderen Knoten getragen wird. Das Problem ist jedoch, dass das Schema nach dem Ausfall erneut ausgeführt werden müsste, um eine gleichmäßige Verteilung wieder herzustellen. Hierbei würden jedoch meist alle virtuellen Knoten neu platziert werden und es müssten viele Daten verschoben werden. Ein ähnliches Szenario würde das Hinzufügen eines Knotens hervorrufen.

Da der Aufwand für das Verschieben der Daten zu hoch wäre, lässt sich ein solches optimales globales Schema nicht realisieren. Stattdessen wurden von Amazon drei Strategien zur Verteilung der Daten evaluiert. Diese werden im Folgenden vorgestellt:

### **Strategie 1**

Die erste Strategie verteilt die virtuellen Knoten zufällig auf dem Kreis. Hierdurch lässt sich eine gute Lastverteilung bewirken, da es unwahrscheinlich ist, dass ein Knoten im Vergleich zu den anderen Knoten einen sehr viel größeren Bereich des Kreises zugewiesen bekommt. Des Weiteren ist die Wahrscheinlichkeit gering, dass auf die meisten virtuellen Knoten eines realen Knotens fast ausschließlich virtuelle Knoten eines einzigen anderen realen Knotens folgen.

Außerdem hat diese Strategie den Vorteil, dass das Hinzufügen eines neuen Knotens eine verteilte Operation ist: Der Knoten kann die Positionen seiner virtuellen Knoten zufällig auswählen und die realen Knoten, die die Daten seines Bereichs halten, um die Übergabe der Daten bitten. Diese Knoten können dann ihren Speicher nach Einträgen durchsuchen, die übergeben werden sollen.

Es wurde jedoch festgestellt, dass eben dieses Durchsuchen des Speichers eines Knotens sehr aufwändig ist, da alle Einträge überprüft werden müssen. Strategie 2 vereinfacht diesen Suchvorgang.

### **Strategie 2**

Strategie 2 wurde nur für kurze Zeit eingesetzt und kann als Zwischenschritt zu Strategie 3 betrachtet werden.

Strategie 2 unterteilt den Kreis in gleichgroße Partitionen. Dabei muss die Anzahl der Partitionen sehr viel größer sein als die Anzahl der potentiellen virtuellen Knoten. Ein Knoten speichert immer nur die Daten einer vollen Partition. Auch Strategie 2 verteilt die virtuellen Knoten zufällig auf dem Kreis. Auf die gleiche Weise, wie bei Strategie 1 die Einträge auf die Knoten zugeordnet wurden, werden bei dieser Strategie die Partitionen den Knoten zugeordnet. Ein Knoten legt einen neuen Eintrag dann im Speicherbereich der entsprechenden Partition ab. Wird er von einem neuen Knoten aufgefordert, einen Bereich des Kreises abzugeben, muss er nicht seinen kompletten Speicher nach Einträgen durchsuchen, sondern nur die zum Bereich gehörenden Partitionen bestimmen. Hierdurch wird nicht nur der Suchaufwand beim Hinzufügen eines Knotens verringert, sondern auch der benötigte Speicher, um festzuhalten, welcher Bereich des Kreises auf welchem Knoten

liegt, verringert sich immens.

### Strategie 3

Strategie 3 geht wie Strategie 2 vor, bis auf dass die Knoten nicht mehr auf dem Kreis platziert werden, sondern die Partitionen den Knoten direkt zufällig zugeordnet werden. Kommt ein neuer Knoten hinzu, erhält er von jedem Knoten die gleiche Anzahl an zufällig ausgewählten Partitionen.

#### C.3.4 Die put()- und die get()-Operation

Mit der put()-Operation lassen sich Daten in Dynamo anlegen. Wird die put()-Operation an einem Knoten aufgerufen, informiert dieser Knoten zunächst den nächsten erreichbaren Knoten auf dem Kreis ausgehend vom Hash-Wert des Schlüssels des neuen Eintrags. Dieser Knoten ist für das weitere Verfahren verantwortlich. Hat er die Nachricht erhalten, speichert er den Eintrag und informiert die nächsten  $N - 1$  Knoten auf dem Kreis über den Eintrag. Hat er von  $W - 1$  Knoten eine Bestätigung erhalten, dass diese den Eintrag ebenfalls gespeichert haben, kann er den Nutzer informieren, dass der Eintrag gespeichert ist. Der Nutzer erhält die Bestätigung also, wenn der Eintrag auf  $W$  Knoten abgelegt ist. Anschließend wartet der verantwortliche Knoten, ob er von den restlichen Knoten eine Antwort bekommt und informiert den nächsten Knoten auf dem Kreis ebenfalls über den Eintrag, falls einer der  $N - 1$  Knoten nicht reagiert. Das Ziel ist also, den Eintrag  $N$  Mal zu speichern.

Bei der get()-Operation wird die Anfrage ebenfalls zunächst an den verantwortlichen Knoten übermittelt. Dieser leitet die Anfrage an  $N - 1$  Knoten weiter und beantwortet die Anfrage, wenn ihm die Ergebnisse von  $N - 1$  anderen Knoten und sein eigenes Ergebnis zur Verfügung stehen. Um Widersprüche in den Ergebnissen auflösen zu können, wird eine Versionierung der Daten benötigt.

#### C.3.5 Versionierung

Dynamo verändert gespeicherte Einträge nicht, sondern behandelt eine Änderung als neue Version der Daten. Dynamo verwendet Vektoruhren, um aufzulösen, welche Version eines Eintrages neuer ist. Der Zeitstempel des alten Eintrages

wird bei einem Update vom Nutzer der Datenbank mitgegeben, sodass der Knoten, der für den neuen Eintrag verantwortlich ist, einen neuen Zeitstempel erstellen kann. Findet ein Knoten zwei Einträge vor, die widersprüchliche Vektoren haben, also keine Aussage getroffen werden kann, welcher Eintrag neuer ist, speichert er beide Einträge ab und bei der nächsten Anfrage werden beide Einträge dem Nutzer übergeben. Dieser muss dann den Konflikt auflösen.

Es gibt einige Anwendungsfälle, in denen eine solche Konfliktlösung leicht ableitbar ist. So wird in [6] der Anwendungsfall des virtuellen Einkaufskorbs als Beispiel genannt. Hatte ein Nutzer die Version 1 seines Einkaufkorbes und hat anschließend zwei Artikel in den Einkaufskorb gelegt, wobei beim ersten Artikel basierend auf Version 1 eine Version 2 des Einkaufskorbes erstellt wurde und beim zweiten Artikel ebenfalls basierend auf Version 1 eine Version 3 erstellt wurde, so lässt sich der Konflikt zwischen Version 2 und Version 3 leicht durch das Vereinigen der beiden Einkaufskörbe lösen. Auf diese Weise kann kein hinzugefügter Artikel verloren gehen. Nur wenn ein Artikel entfernt wurde, kann es dazu kommen, dass dieser Artikel wieder im Einkaufskorb landet.

## C.4 Erneute Einordnung

Mithilfe der Parameter  $W$  und  $R$  kann Dynamo etwas differenzierter im CAP-Theorem eingeordnet werden. Je niedriger die Parameter  $W$  und  $R$  sind, desto verfügbarer und partionierungstoleranter wird Dynamo. Werden beide Parameter auf 1 gesetzt, reicht ein verfügbarer Knoten, um eine Schreiboperation durchzuführen und ein verfügbarer Knoten, der eine beliebige Version des Eintrag gespeichert hat, um eine Anfrage zu beantworten. Allerdings leidet die Konsistenz unter sehr niedrigen Werten  $W$  und  $R$ . Werden  $W$  und  $R$  auf die Anzahl der Knoten gesetzt, ist Dynamo komplett konsistent, aber weder verfügbar, wenn ein Knoten ausfällt, noch partitio-nierungstolerant.

# Anhang D

## MapReduce

### D.1 Modell

MapReduce ist ein generalisierter Ansatz für verteilte Datenverarbeitung. Zusätzlich wird mit MapReduce eine Laufzeitumgebung vorgeschlagen, welche für die automatische Verteilung und Parallelierung zuständig ist und außerdem Fehlerbehandlung, sowieso Kommunikation während der verteilten Berechnung abdeckt. Folgende Übersicht basiert auf dem Paper *MapReduce: Simplified Data Processing on Large Clusters* [5].

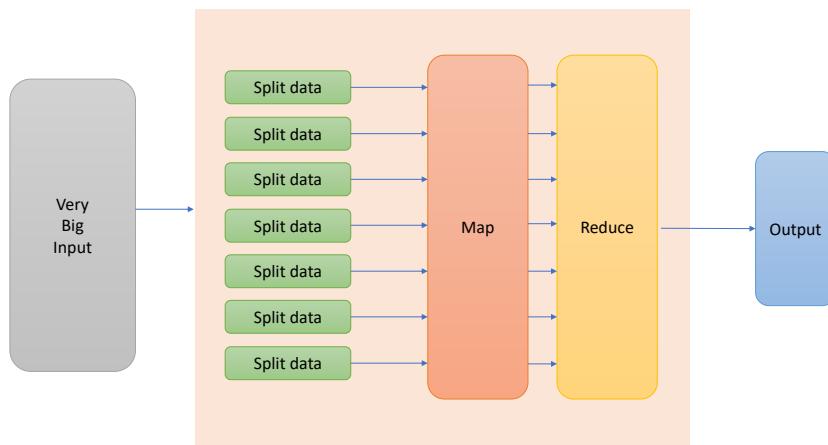


Abbildung D.1: MapReduce Modell Übersicht

Um eine verteilte Berechnung und Verarbeitung zu vereinfachen, wird diese in zwei grobe Phasen aufgeteilt: Map und Reduce. Für beide Phasen muss der Nutzer jeweils eine entsprechende Funktion definieren. Der Input liegt als key/value Paare vor und MapReduce produziert wiederum key/value Paare als Ergebnis. In der Map Phase werden aus den gegebenen key/value Paaren Zwischenwerte berechnet, welche wiederum als key/value Paare vorliegen. Diese werden anschließend gruppiert und in der Reduce Phase weiterbehandelt. In der Reduce Phase werden dann die berechneten Zwischenwerte für gleiche keys zusammengefasst und aggregiert.

Beispiel: Für eine Vielzahl von vorliegenden Dokumenten soll die Wortanzahl ermittelt werden. Dann könnten die Map und Reduce Funktionen wie folgt aussehen:

---

```

1 map(String key, String value):
2     // key: document name
3     // value: document contents
4     for each word w in value:
5         EmitIntermediate(w, "1");
6
7 reduce(String key, Iterator values):
8     // key: a word
9     // values: a list of counts
10    int result = 0;
11    for each v in values:
12        result += ParseInt(v);
13    Emit(AsString(result));

```

---

Die Map function gibt also einfach jedes Wort plus ein Anzahl, in diesem Falle einfach 1 zurück. Anschließend würde die Reduce Funktion alle zurückgegebenen Anzahlen für jedes Wort summieren.

## D.2 Implementation

Für das MapReduce Interface sind laut Paper [5] verschiedene Umsetzungen möglich, die sich an den Ausführungsumgebungen ausrichten. Im Paper wird dann eine Umsetzung für eine damals typisch eingesetzte Environment

vorgestellt. Als Ausführungsumgebungen werden große Cluster (Anzahl 100-1000 Knoten) von Computern mit durchschnittlicher Rechenleistung und Hauptspeicher genannt. Folgend wird beschrieben wie die Verteilung der Map und Reduce Funktion in dieser Umgebung von Google implementiert wurde. Auf den Knoten läuft ein verteiltes Dateisystem (GFS), welches ein Zugriff auf Dateien über das Netzwerk ermöglicht, ein Knoten jedoch dabei diese Dateien wie lokale Dateien behandeln kann. Der Nutzer übermittelt den MapReduce-Job an ein Scheduling System, welches dann die Ausführung koordiniert.

### Vorbereitungen

Ein Knoten übernimmt die Aufgabe des Masters, welcher dann die weitere Ausführung und das Scheduling übernimmt. Der übermittelte MapReduce-Job wird dann wie folgt in verschiedene Tasks aufgeteilt:

1. Die Eingabedaten werden in  $M$  Parts gesplittet, pro Part wird später ein Map-Task ausgeführt.
2. Die Reduce-Phase wird in  $R$  Reduce-Tasks aufgeteilt. Hierzu wird die Menge der möglichen Zwischenergebnisse durch eine Verteilungsfunktion in  $R$  Partitionen aufgeteilt.

Diese Tasks werden den Workern dann dynamisch während der Ausführung zugeteilt. Typische Zahlen sind: Z.B:  $M=200.000$   $R=4000$ ,  $\text{workers}=2000$

### Ausführung der Map Phase

In der Map Phase teilt der Master jedem freien Worker einen Map-Task zu. Der Worker liest den zum Map-Task gehörenden Datenpart und führt darauf die Map-Funktion aus. Hierbei werden (bis zu)  $R$  lokale Dateien (eine Datei pro Key) mit Zwischenergebnissen lokal gespeichert. Zum Schluss übermittelt der Worker den Speicherort der Zwischenergebnisse an den Master, damit diese später verteilt werden können.

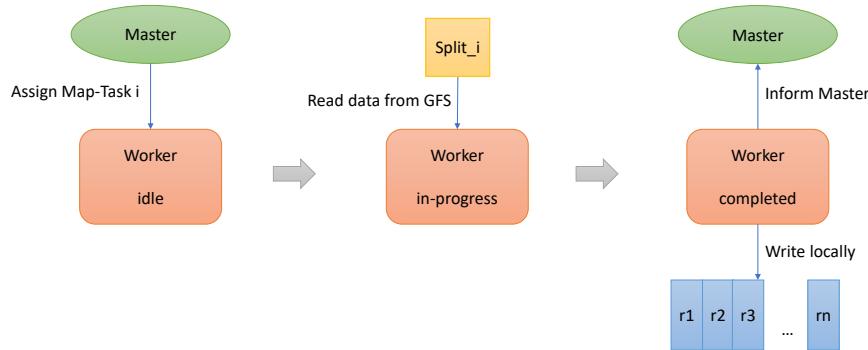


Abbildung D.2: Ausführung eines Map tasks

### Ausführung der Reduce Phase

Der Master verteilt die Reduce-Tasks auf freie Worker und übermittelt dabei die Speicherorte der Zwischenergebnisse pro Key. Dann liest der ausführende Worker die Zwischenergebnisse pro Key mittels RPC von den anderen Knoten, sortiert die Daten und wendet die vom User definierte Reduce-Funktion darauf an. Anschließend wird ein Output-File mit dem Ergebnis ins verteilte Dateisystem geschrieben.

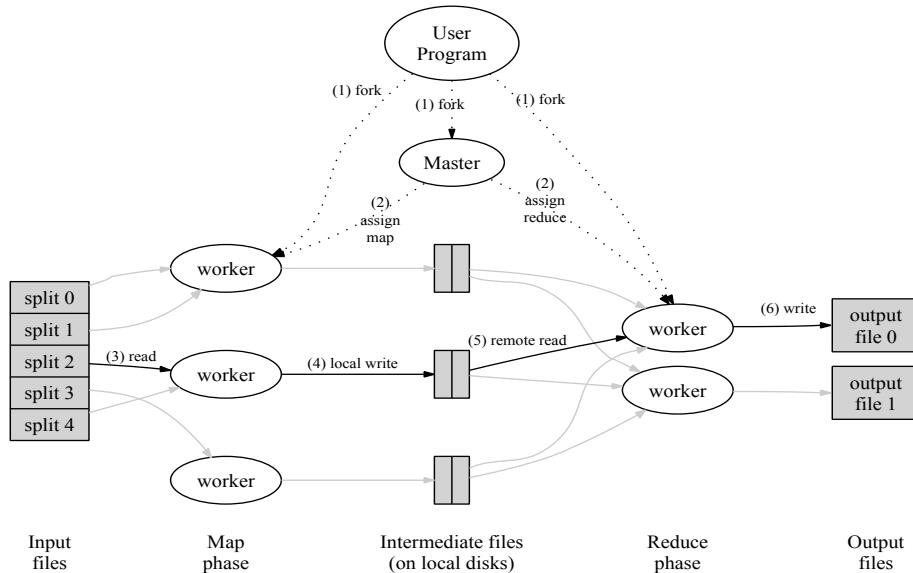


Abbildung D.3: Übersicht der gesamten MapReduce Ausführungslogik

### Fehlertoleranz

Da MapReduce für den parallel Einsatz auf vielen Knoten gedacht ist und einzelne Knoten ausfallen können, gibt es spezielle Fehlerfälle die betrachtet und behandelt werden.

Um ein Failure des Masterknotens aufzufangen, schreibt dieser periodisch Checkpoints vom derzeitigen Ausführungsstatus. Tritt nun ein Fehler im Master auf, wird ein neuer Knoten als Master gestartet und die Ausführung vom letzten Checkpoint fortgeführt.

Jeder Workerknoten wird vom Master mittels Heartbeat periodisch auf Erreichbarkeit und Status abgefragt. Wird ein ausgefallener Worker oder ein fehlerhafter Job erkannt, wird zwischen abgeschlossenen und nicht abgeschlossenen Job unterschieden. Ein nicht abgeschlossener Map oder Reduce Task wird einfach auf einem anderen Worker neu gestartet. Abgeschlossene Map Tasks werden auch erneut ausgeführt, da die Zwischenergebnisse lokal auf den Knoten liegen und diese entsprechend fehlen. Abgeschlossene Reduce Tasks müssen nicht erneut ausgeführt werden, weil die Ergebnisse im verteilten Dateisystem gespeichert werden.

## D.3 Refinements

Zur eigentlichen Ausführungslogik gibt es noch weitere Refinements, die die Ausführung optimieren.

### Combiner

In der Map Phase entstehen oftmals Wiederholungen von Zwischenergebnissen. Im Wordcount Beispiel:

```
(foo bar bar bar) → (foo, 1) (bar, 1) (bar, 1) (bar, 1)
```

Ein Worker kann nach Abschluss einen Map-Tasks die Zwischenergebnissen pro key mittels Combine Funktion zusammenfassen. Z.B.

```
(foo, 1) (bar, 1) (bar, 1) (bar, 1) → (foo, 1) (bar, 3)
```

### Partitioning Function

Die Nummer der Reduce-Tasks wird von Nutzer festgelegt. Nun gibt es eine default Partitioning Function ( $\text{hash}(\text{key}) \bmod R$ ), welche die Zwischenergebnisse nach Keys den Reduce-Tasks zuordnet. Durch die default PF werden meist gut verteilte Partitionen generiert, allerdings kann der Nutzer auch eigene Verteilungsfunktionen definieren.

### Datenlokalität

Die Inputdaten werden im verteilten Dateisystem in Blöcke geteilt und repliziert auf verschiedenen Knoten gespeichert. Da diese Blöcke später den MAP-Tasks zugeordnet sind, versucht der Master die Tasks so zu verteilen, dass die benötigten Blöcke auf den zugeordneten Knoten oder zumindest physisch in der Nähe liegen. Hierdurch wird der Netzwerkdurchsatz während der Taskausführung reduziert und eine Ausbremsung verhindert.

### Backup Tasks

Einzelne, aufgrund Nebeneffekten sehr langsame Knoten, können den Gesamtprozess aufhalten, besonders wenn diese zum Ende eine Phase auftreten. Um dies zu vermeiden, werden zum Ende einer MapReduce Phase Tasks welche noch nicht abgeschlossen sind, repliziert und auf mehreren Knoten parallel

ausgeführt. Der erste abgeschlossene Task 'gewinnt' und langsame Knoten werden übergangen.

## D.4 Probleme und Nachteile von MapReduce

MapReduce und Spark (siehe nächstes Kapitel) ermöglichen die parallele und verteilte Verarbeitung von riesigen Datenmengen. Allerdings speichert und verarbeitet MapReduce alle Daten auf der Platte, während Spark durch eine in-memory Verarbeitung bis zu 100 mal schneller ist. Auch bei iterativer Verarbeitung kann Spark durch die RDDs mehrere Operationen im Speicher ausführen, während MapReduce für jede Iterationen einen gesamten Durchlauf anstößt. MapReduce zwingt den Nutzer die Verarbeitung der Daten in eine Map- und eine Reduce-Funktion aufzuteilen, Spark bietet dem Nutzer hier viel mehr Komfort, weil zusätzliche Operationen möglich sind und eine High Level API anbietet.

Zusätzlich bietet Spark eine Möglichkeit Streams zu verarbeiten, was gerade in Hinsicht auf unseren Anwendungsfall, nämlich die Verarbeitung und live Analyse von Twitterdaten interessant ist.

# Anhang E

## Spark

### E.1 Spark und MapReduce

Aufbauend auf der Vorstellung von MapReduce soll in diesem Abschnitt die Software Spark beleuchtet werden. MapReduce ist ein Programmiermodell, das auch bei Spark Anwendung findet. Allerdings bietet Spark weitaus mehr Operationen. Die MapReduce-Referenzimplementierung ist Hadoop. Spark integriert sich in das Hadoop-Ökosystem und wirkt dort als Execution Engine.

Es wird ergänzt durch andere Komponenten von Hadoop wie dem YARN-Ressourcen Manger, das verteilte Dateisystem HDFS, Recovery-Mechanismen bei Ausfällen und Vorkehrungen zur Datensicherheit.

Dabei nutzt Spark die persistente Storage von anderen und konzentriert sich auf die Verarbeitung der verteilten Berechnungen im Speicher. Das grundlegende Datenmodell soll hier besprochen werden, ebenso wie die Architektur von Spark, die die Analysen ausführt und wie das im Spezialfall Spark Streaming funktioniert.

### E.2 RDDs

Wir beginnen mit der zentralen Datenstruktur, den RDDs. Sie werden beschrieben in [14]. RDD steht für Resilient Distributed Dataset. Die Grundidee hinter den RDDs ist es, die Daten immutable (unveränderbar) vorzuhalten und einen festen Satz an Operationen anzubieten. Diese Operationen er-

zeugen wiederum neue RDDs oder bringen Analysen zu Ende. Spark soll zwei Ziele mit Hilfe der RDDs erfüllen, nämlich Abstraktion von verteilten Arbeitsspeicherkonzepten (distributed shared memory) und die Ausfalltoleranz.

Die RDDs werden im Hauptspeicher gehalten. Es geht nicht darum, wie diese dauerhaft gespeichert werden können (persistiert werden).

Dadurch, dass es sich bei den RDDs um eine Abstraktion handelt, müssen diese noch im konkreten Fall implementiert werden. Die RDDs sind nur die Schnittstelle mit den definierten Operationen.

Für die Umsetzung betrachten wir im Folgenden noch verschiedene Dinge. Anfangen werden wir bei dem Teil distributed im Namen der RDDs, nämlich damit, wie die Verteilung bei Spark funktioniert.

## E.3 Funktionsweise Verteilung bei Spark

Wie auch bei MapReduce werden bei Spark die gleichen Operationen auf viele verschiedene Einzeldaten angewendet. Dabei erfolgt die Anwendung der Operationen unabhängig voneinander für jedes einzelne Datum. Später in der Berechnung werden dann einzelne Daten zusammengefügt. Damit können die Daten verteilt werden. Spark bietet drei große Arten der Partitionierung der Daten:

- Hash-Partitioning

Dabei entscheidet der Hashwert der Daten darüber, zu welcher Partition sie gehört. Vorteil ist, dass die Verteilung im Normalfall sehr ausgeglichen zwischen Partitionen ist.

- Sort-Partitioning

Dabei werden die Daten sortiert und benachbarte Daten kommen in die gleiche Partition. Vorteil ist es, dass s.g. Range-Anfragen schnell beantwortet werden können, da die ähnlichen Daten benachbart sind. Nachteilig ist, dass die Daten möglicherweise sehr ungleich auf die Partitionen verteilt sind.

- eigene Kontrolle über die Partitionierung

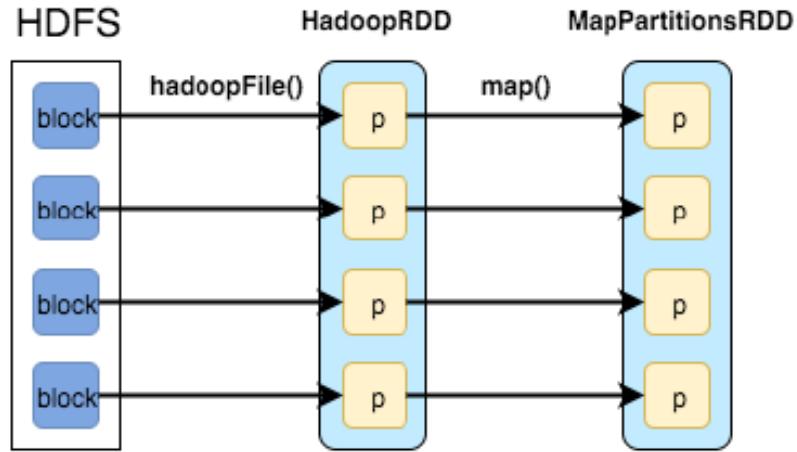


Abbildung E.1: Erstellung RDD aus HDFS [13]

Der Programmierer eigener RDDs kann auch eigene Strategien zur Partitionierung innerhalb der RDDs vorgeben. Dabei können auch schon vorhandene Partitionierungen der Datenquellen weitergeführt werden.

Beispielhaft hierfür soll die Erstellung eines RDDs aus Daten von HDFS (Hadoop Distributed Filesystem) gezeigt werden. Hier wird die eigene Partitionierung so eingesetzt, dass die Daten in den RDDs auf den gleichen Knoten liegen, wie die Daten im HDFS. Man spricht dann von Datenlokalität.

Im HDFS sind die Daten in Blöcken partitioniert, die auf verschiedene Rechner verteilt werden. Nun erfolgt ein 1:1 Mapping der Blöcke im HDFS auf die Partitionen im RDD. So entsteht das HadoopRDD. Dieses HadoopRDD kann dann mit den standardisierten RDD-Operationen ganz normal weiterverarbeitet werden.

Liegen die Daten nicht schon verteilt vor, z.B. im einem verteilten Dateisystem, so kann Spark sie auch automatisch verteilen. Dies erfolgt beim Aufruf der Methode `sc.parallelize()`. sc steht hier für `SparkContext`. Dieses Objekt bietet viele Verwaltungsfunktionen an.

## E.4 Lineage

Wir haben im vorigen Abschnitt gesehen, wie Spark für die Verteilung das Partitionieren (Sharding) vornimmt. Neben der Verteilung, die hier wie im gesamten Projekt durch Scale-Out erfolgt, muss sich Spark auch um Ausfalltoleranz kümmern. Durch die hohe Anzahl eingesetzter Rechner steigt die Ausfallwahrscheinlichkeit stark an. Um die Erkennung ausgestellter Rechner kümmert sich der Ressourcenmanager und ebenso für die Inbetriebnahme von Ersatzrechnern bzw. weiterer Rechner für höhere Leistung. Spark braucht allerdings einen Mechanismus, wie die Analyse fortgeführt wird, wenn einer der Rechner ausgefallen ist. Zwei Dinge sind beantworten: wie kommt man an die lokal gespeicherten Daten? Sie sind durch den Ausfall nicht mehr zugreifbar und wie wird die Berechnung mit diesen Daten fortgesetzt?

Die Antwort bei Spark ist erstaunlich einfach. Dies liegt daran, dass die RDDs immutable sind. Dadurch kann das RDD nach erstmaliger Fertigstellung nicht in einer unsauberen (dirty) Zustand kommen. Mechanismen für ein UNDO teilweise fertiger Operationen sind also nicht notwendig. Stattdessen wird nur ein REDO benötigt. Und um dieses Redo geht es bei der sogenannten Data Lineage. Die Grundidee ist es, für die RDDs die Herkunft, also die Entstehungsgeschichte aufzuzeichnen bzw. im Voraus zu berechnen. Zur Erinnerung: das verlorene gegangene RDD ist durch eine Reihe von Transformationen von anderen RDDs aus den ursprünglichen Eingabedaten entstanden. Fällt nun ein Knoten aus und geht damit das RDD verloren, so können in der Lineage-Datenstruktur für dieses RDD die Eltern-RDDs gefunden werden. Sofern diese noch vorliegen (z.B. auf einem nicht-ausgefallenen Knoten), wird einfach die aufgezeichnete Operation mit den Eltern-RDDs wiederholt. Sind diese Eltern-RDDs ebenfalls nicht mehr vorhanden oder ausgefallen, so wird in der Kette rekursiv durchgegangen. Am Ende stehen die originalen Eingabedaten. Das Konzept geht davon aus, dass diese Daten bereits redundant vorliegen. Das ist auch der Fall, wenn z.B. HDFS genutzt wird die Originaldaten hiervon geladen werden.

Diese Idee kann nun durch Checkpoints ergänzt werden. Unter einem Checkpoint versteht man hier eine persistente Kopie von RDDs in einer tieferen Lineage-Ebene. So muss bei einem Ausfall nur bis maximal zu diesem RDD

in dem Checkpoint zurückgegangen werden. Je länger die Geschichte eines RDD ist, desto lohnender kann dieser Vorgang sein. Dagegen spricht natürlich der erhöhte Speicherverbrauch.

Neben der Betrachtung auf RDD-Ebene lässt sich diese Betrachtung auch auf Partitionsebene innerhalb des RDDs übertragen. Den eigentlich sind die RDDs ja über mehrere Rechner verteilt (Scale-Out) und auf dem Rechner liegen dann Partitionen verschiedener RDDs und meist nicht das ganze RDD.

## E.5 Programmiermodell

Spark ist sowohl ein Framework, dass einfache Möglichkeiten für verteilte Analysen bereitstellt, als auch eine Möglichkeit die so erzeugten Analysen durchzuführen.

In diesem Abschnitt wird nochmal genauer auf das Programmiermodell eingegangen. Die Daten werden mittels Scale-Out-Algorithmen verteilt analysiert. Um möglichst hohe Parallelisierung zu erreichen, werden die Daten in den meisten Fällen als unabhängig angenommen. Beispiele für Eingabedaten sind Datenbanktabellen oder csv-Dateien.

Die einzelnen Einträge werden unabhängig voneinander vorverarbeitet. Dies geschieht im Map-Schritt. Dann werden die Ergebnisse zusammengefügt. Das passiert im Reduce-Schritt. So weit entspricht das Vorgehen dem MapReduce-Programmiermodell. Spark erweitert es um einige Operationen.

Dabei werden zwei Typen unterschieden: Aktionen, die aus einem RDD einen konventionellen Datentyp, wie int, erzeugen. Der andere Typ sind Transformationen, die aus RDDs wieder andere RDDs erzeugen. Dabei werden neue RDDs erzeugt und nicht ursprünglichen RDDs verändert, denn RDDs sind ja immutable.

Man könnte also von einem TransformAction-Programmiermodell als Weiterentwicklung des MapReduce-Programmiermodells sprechen.

Zu den Transformationen gehören auch filter und flatMap. MapReduce erweckt den Eindruck, dass es zumindest im Map-Schritt eine 1:1-Abbildung zwischen Elementen des einen RDDs zu Elementen des neuen RDDs besteht. Natürlich lassen sich auch Elemente entfernen, die dann im neuen RDD nicht mehr vorkommen (filter) und neue Elemente erzeugen. Dafür wird eine Lis-

te dieser Elemente als Transformationsergebnis für ein Element erzeugt und statt map flatmap aufgerufen. Dabei werden alle Listenelemente zu eigenen Elementen in dem neuen RDD.

Beispiele für Aktionen sind:

- count()
- collect()
- reduce()

Beispiele für Transformationen sind:

- map()
- filter()
- flatMap()
- sample()
- join()
- sort()

Hinweis: Zur Zeit stellt Spark seine API von RDDs auf Dataframes um. Diese haben deutlich erweiterte Möglichkeiten. Sie können mit ähnlichen Operationen wie die von SQL bearbeitet werden.

### E.5.1 Beispielcode

Die Beispiele stammen von <https://spark.apache.org/examples.html>.

#### Bekannte Wordcount

```
1 val textFile = sc.textFile("hdfs://...")  
2 val counts = textFile.flatMap(line =>  
3     line.split(" "))  
4     .map(word => (word, 1))  
5     .reduceByKey(_ + _)  
6 counts.saveAsTextFile("hdfs://...")
```

### Approximation von PI

```

1 val count = sc.parallelize(1 to NUM_SAMPLES)
2                     .filter { _ =>
3   val x = math.random
4   val y = math.random
5   x*x + y*y < 1
6 }.count()
7 println(s"Pi is roughly $ 
8   {4.0 * count / NUM_SAMPLES}")

```

Die Funktionsweise ist wie folgt: `math.random` liefert einen Wert zwischen 0 und 1. Wir stellen uns einen Einheitskreis vor. Sein Flächeninhalt ist  $\pi r^2$ , also bei  $r = 1 \pi$ . Nun wird geprüft, ob die Koordinate  $(x,y)$  in dem Einheitskreis liegt. Wenn dies der Fall ist, dann wird sie gezählt. Der Anteil der Koordinaten, die in dem Einheitskreis liegen im Vergleich zu der Gesamtzahl an Samples, entspricht gerade  $\frac{1}{4}\pi$ . Der Faktor Vier kommt daher, dass wir nur den ersten Quadranten des Einheitskreis betrachten.

## E.6 Spark und Distributed Shared Memory

Am Anfang haben wir als ein Ziel von Spark die Abstraktion für Distributed Shared Memory beschrieben. Das wollen wir nun nochmal aufgreifen und beide vergleichen. Dadurch, dass die Operationen gegenüber der Distributed Shared Memory eingeschränkt werden, nämlich insbesondere Veränderungen nur bei gleichzeitiger Kopie des RDDs vorgenommen werden können, entfallen viele Aufgaben. Distributed Shared Memory funktioniert wie verteilter Arbeitsspeicher, d.h. die einzelnen Rechner können mehr oder weniger unreguliert Lese- und Schreiboperationen vornehmen. Damit das nicht im Chaos endet, müssen sie sich Protokolle zur Synchronisation nutzen.

Betrachten wir einige Aufgaben genauer:

- Konsistenz der Daten sicherstellen

Ist bei RDDs nicht notwendig, da die RDDs immutable sind. Es muss nur überwacht werden, ob das RDD vollständig erzeugt wurde. Dadurch wird der ganze Transformationsvorgang atomar. Bei Distributed Shared Memory ist

das nicht so. Da sind zusammengehörige Schreiboperationen nicht atomar, d.h. es ist durchaus möglich, dass der erste Teil der Schreiboperationen bereits umgesetzt wurde (z.B. Kontostand bei einer Überweisung verringern), der zweite Teil aber nicht (z.B. Kontostand beim Empfänger erhöhen). Deshalb muss die Anwendung hier ein spezielles Protokoll zwischen den Knoten einführen.

- Fault Recovery

Kann bei RDDs durch Lineage vergleichsweise einfach vorgenommen werden. Bei Distributed Shared Memory ist wieder ein kompliziertes Protokoll notwendig. Checkpoints sichern die Daten. Da die Konsistenz aber nicht notwendigerweise gegeben ist, braucht es Rollback-Vorgänge.

Eindeutiger Nachteil der RDDs ist ihr schlechter Umgang mit feingranularen Schreibevorgänge, d.h wenn nur wenige Elemente eines RDDs geändert werden sollen. Distributed Shared Memory liegt hier eindeutig vorne, da alle Operationen feingranular erfolgen können.

## E.7 Spark-Architektur

Hauptaugenmerk in diesem Abschnitt soll auf der Berechnung der Job Stages liegen [13]. Mit Hilfe der Transformationen und Aktionen lassen sich komplexe Abfolgen von Operationen auf unterschiedlichen RDDs und auch in Kombination dieser RDDs erzeugen. Während die Transformationen für alle Partitionen parallel ausgeführt werden können, müssen die parallel erfolgenden Operationen durch Spark erst berechnet werden.

Dazu werden die komplexen Operationsfolgen in Stages unterteilt. Sie entsprechen den Knoten in einem DAG (einem azyklischen Graphen). Dadurch entsteht eine partielle Ordnung der Stages. Stehen zwei Stages weder in einer Vor- noch in einer Nacheinander-Beziehung, so können sie parallel berechnet werden, sofern ihre Eltern-RDDs bereits berechnet wurden.

Ein entscheidender Unterschied stellen hier die Operationen da. Sind die Operationen Joins oder groupByKey, so müssen alle daran beteiligten RDDs vollständig berechnet sein, damit die Operation ausgeführt werden kann. Man spricht dann von einer wide dependency. D.h. bei der Ausführung muss das erste RDD auf die anderen RDDs warten. Andere Operation wie map,

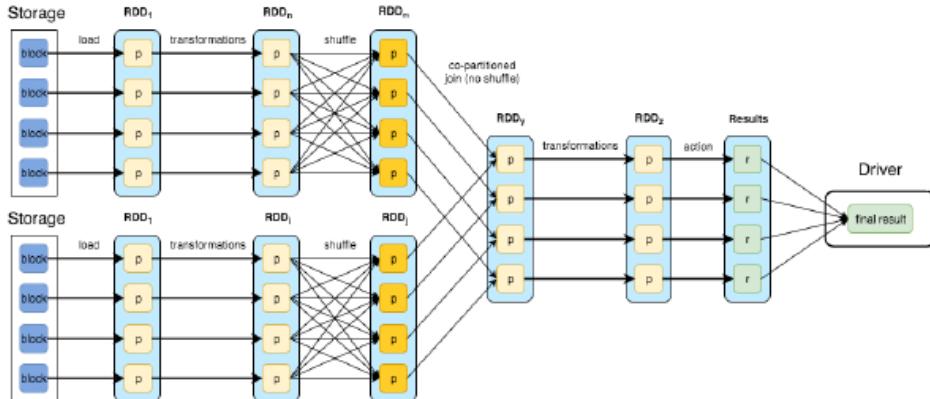


Abbildung E.2: Spark Stages [13]

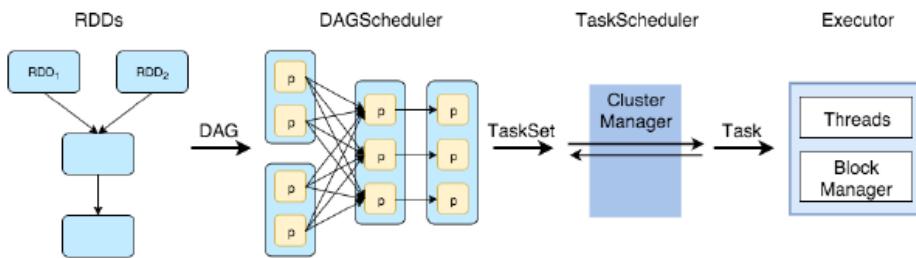


Abbildung E.3: Spark Architektur [13]

union, join mit co-partitionierten Input oder filter brauchen das nicht. Sie liegen innerhalb eines Stages in einer Abfolge. Man spricht hier von narrow dependencies.

### E.7.1 Durchlauf durch Komponenten

Betrachten wir nun Einbindung der Job-Stage-Berechnung in die Architektur. Alles fängt mit den RDDs an, die inkl. ihrer Lineage aus dem Javacode ermittelt werden. Sie werden als DAG in den DAGScheduler eingegeben. Dieser berechnet die möglichen Jobstages und ermittelt daraus TaskSets für die einzelnen Rechnenknoten. Er gibt sie an den Clustermanager weiter. Dieser verteilt sie dann an geeignete Knoten. In diesen Knoten, den Executors, werden diese dann in Threads ausgeführt.

## E.8 Vor- und Nachteile von Spark

Fassen wir nochmal die Vorteile und Nachteile von Spark zusammen. Entscheidend ist die Art von Programm, das ausgeführt werden soll. Jenachdem ist Spark besser oder weniger gut geeignet.

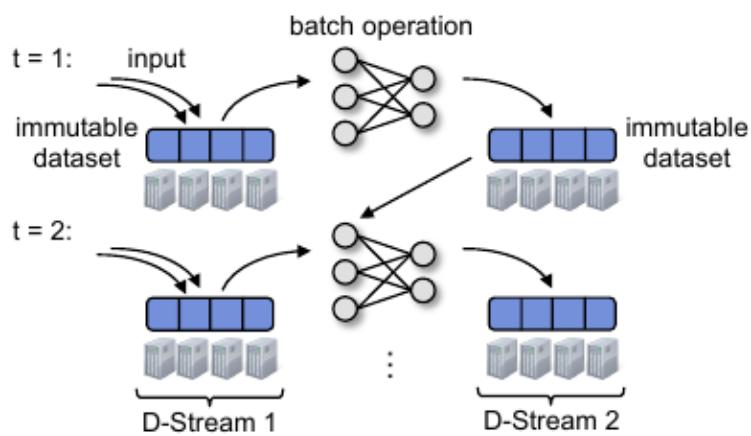
Schlecht geeignete Programme sind Programme mit vielen asynchronen, feingranularen Updates auf einem gemeinsamen Zustand. Dies würde zu einer Explosion der Anzahl von RDDs führen. Beispiele hierfür sind Web-Anwendungen und inkrementelle Web-Crawler. Alternativen wären RAM-Cloud, Percolator und Piccolo.

Gut geeignete Programme führen Bulk Writes auf vielen Daten aus. Insbesondere dann, wenn Datenlokalität ausgenutzt werden kann, ist Spark gut geeignet. Spark punktet dann mit effizienter Fault Tolerance durch Lineage und Ausgleich von langsamem Knoten, da kein Rollback notwendig ist.

## E.9 Funktionsweise Spark Streaming

Als letztes wollen wir noch die Funktionsweise einer Variante von Spark anschauen, nämlich Spark Streaming. Spark Streaming wird in [15] beschrieben. Wenn man nicht auf periodisch ausgeführte Analysen warten möchte, sondern gewissermaßen live auf Ergebnisse zugreifen möchte, kann Spark Streaming eingesetzt werden.

Die Eingabedaten kommen dabei als kontinuierlicher Stream. Sie werden in Microbatches zerlegt, d.h. es werden z.B. die Daten einer Sekunde zwischengespeichert. Dann wird ein herkömmlicher Spark-Job auf dem Batch ausgeführt. Dies wird kontinuierlich für die einkommenden Streams gemacht. Spark nennt dieses Vorgehen D-Streams. Es ist ein Denkmodell dafür, dass einkommende Einträge als Microbatches in einem RDD gespeichert werden. Auf diesem werden dann parallel deterministische Operationen ausgeführt, analog wie bei der Batch-Analyse. Ein D-Stream ist also eine Folge von Datensets. Es ist keine komplexe Synchronisation der einzelnen Knoten notwendig. Sie operieren unabhängig voneinander und bekommen ihre Aufgaben aus dem DAGScheduler vorgegeben. Sie greifen nicht auf gemeinsamen Speicher schreibend zu. Ausfälle von Knoten sowie zu langsame Knoten lassen sich genau wie sonst auch beheben.



(b) D-Stream processing model. In each time interval, the records that arrive are stored reliably across the cluster to form an immutable, partitioned dataset. This is then processed via deterministic parallel operations to compute other distributed datasets that represent program output or state to pass to the next interval. Each series of datasets forms one D-Stream.

Abbildung E.4: Sparks DStreams [15]

### E.9.1 alternatives, nicht verwendetes Denkmodell

Ein alternatives Modell wäre der Continuous Operator. Dabei haben die Nodes einen inneren Zustand, ggf. gibt es einen dedizierten gemeinsamen Zustand. Ausfalltoleranz kann dann mit einem Synchronisationsprotokoll erreicht werden, das Replikation vorsieht. Schwieriger ist hier die Recovery. Da die zusammenhängenden Schreiboperationen nicht atomar ausgeführt werden, braucht man wie bei Distributed Shared Memory Rollback und weitere Mechanismen.

Allerdings können hiermit auch komplexere Berechnungen vorgenommen werden. Es gibt die künstlichen Microbatches-Grenzen nicht. Dementsprechend sind z.B. Rolling-Windows einfacher.

# Anhang F

## Volltextsuche

### F.1 Abstrakt

Diese Seminararbeit behandelt das Thema Informationsrückgewinnung und soll als eine Hilfestellung für die Umsetzung einer Such-Engine im „NoSQL“ Projekt dienen.

Im Rahmen des „NoSQL“ Projektes wird eine Such-Engine benötigt, die unstrukturierte Daten in einer Freitextform durchsucht und der Relevanz entsprechend dem Nutzer auslegt. Um diese fachliche Anforderung zu implementieren reicht der Key-Value Ansatz, der gängigen Datenbanken, nicht und muss dementsprechend durch eine andere Umsetzungsform realisiert werden. In dieser Seminararbeit werden Konzepte und Verfahren vorgestellt, die eine Volltextsuche sowie die Relevanzzuordnung der Suchergebnisse ermöglichen. Die entsprechende Information kann im Buch „Introduction to Information Retrieval“ vom Christopher D. Manning nachschlagen werden.

Die Seminararbeit beginnt mit der Abgrenzung des Begriff *Informationsrückgewinnung*, nachfolgend gestalten Verfahren mit Vorverarbeitungsschritten den Text durchsuchbar und zum Schluss werden die notwendigen Schritte für die Ergebnisrangliste der Suchanfrage beschrieben.

Aufbauend auf den vorgestellten Konzepten und Verfahren wird validiert, ob die Such-Engine „Elasticsearch“ den Projektanforderungen entspricht.

## F.2 Einleitung

Die zielgerichtete Suche nach Daten wird in der Regel mit einer ID- Referenz umgesetzt und liefert ein passendes Ergebnis zu dem passenden Schlüssel. Um eine komplizierte und erfolgreiche SQL Datenbankanfrage durchzuführen, müssen die Daten sich in einer streng strukturierten Form befinden. Diese Form ist passend für die Suche über Datensätze wie Adressen, Produkte und Preise, jedoch ist sie keinesfalls für Daten unstrukturierter Natur geeignet wie zum Beispiel die menschliche Sprache.

Die starke Entwicklung der Hardware, Software und des Internets führen neue Informationsquellen ein: Blogs, Foren, Wikis, soziale Netzwerke bis zu den Lernplattformen. Diese modernen Informationsquellen führen eine große Menge an unstrukturierter Daten ein, die durchsuchbar gestaltet werden müssen. Damit ist die Suche über eine große Menge von Texten in natürlicher Sprache eine zentrale Disziplin.

Das entschiedene Ziel der Volltextsuche: Das Auffinden und Bewerten der relevanten Information, die in natürlicher Sprache gespeichert ist, ohne einen Schlüsselverweis auf den enthaltenen Datensatz.

Damit die folgende oft unangenehmen Situation nicht mehr auftritt.

*„Es tut mir leid, ich kann in Ihre Bestellung nur nachsehen, wenn Sie mir Ihre Bestellnummer geben können.“*

## F.3 Begriffsklärung

Der Begriff Informationsrückgewinnung kann auf unterschiedliche Weise interpretiert und zu unterschiedlichen Disziplinen zugeordnet werden.

Die Definition der Informationsrückgewinnung im Kontext der Volltextsuche kann leicht missverstanden und in Verbindung mit Datamining gebracht werden. Jedoch dürfen die beiden Disziplinen nicht mit einander verknüpfen werden, da diese grundsätzlich unterschiedliche Ziele verfolgen.

Die Informationsrückgewinnung soll im Gegensatz zu Datamining keine neuen Informationen gewinnen, sondern die vorhandenen, enthaltene Information soll zugänglich und auffindbar gemacht werden.

## F.4 Vorbereitung der Lösungsansätze

Um die Suchansätze anschaulich darzustellen, wird ein Szenario mit einer Sammlung von durchsuchbaren Dokument benötigt. Ein Dokument besteht aus einer Menge von Schlüsselwertpaaren, diese können Strings, Integer und andere Schlüsselwertpaare enthalten. Zum Beispiel kann ein Artikeldokument aus drei Feldern aufgebaut werden, in dem Autor, Titel und Textkörper enthalten sind, die in sich einen beliebigen Text tragen können (inklusive Integer, Daten, Weblinks).

Unser Ziel ist es, aus einer Menge von Dokumenten, die in sich unterschiedliche Strukturen aufweisen dürfen, für den Nutzer relevanten Dokumenten zu filtern.

### F.4.1 Lineare Suche

Der einfachste Weg relevante Information aus einer Datensammlung zu filtern, ist das Scannen der Dokumente einer Sammlung nach passenden Suchtermen und das Auslegen der Trefferdokumente als eine Ergebnisliste. Der lineare Suchansatz kann in viele Bereichen nützlich sein wie zum Beispiel die Linux *cat* Funktion, jedoch ist dieser nicht für alle Systeme und Sammlungen geeignet, denn zusammen mit der Dokumentenanzahl wächst die Suchzeit proportional mit.

Somit ist das Suchen in großen Datenmengen, besonders im Onlineformat, nicht durchführbar.

### F.4.2 Matrix Suche

Da die lineare Suche nur auf kleine Datensätze durchgeführt werden kann, wird ein anderer Ansatz benötigt, der Arbeit im Voraus leistet, um später die Suche zu vereinfachen. Es werden gewisse Metainformationen des gespeicherten Dokumentes in einer Datenstruktur abgelegt, die das Suchen unterstützen und beschleunigen sollen.

Diese Struktur ist in einer Häufigkeitsmatrix aufgebaut und speichert für jedes Wort und Dokument einen Boolean, der auf existieren eines Wortes im Dokument verweist.

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Abbildung F.1: Häufigkeitsmatrix [12]

In dieser Datenstruktur wird die Suche in einer Boolean-Form abgearbeitet. Es werden "1" und "0" für jedes Dokument, das einen bestimmten Such-Term beinhaltet, ausgelesen. Auf diese Weise können Dokumente auf Existenz einzelner Terme untersucht werden. Ferner könne die Terme in Kombination mit einander, durch die AND, OR, NOT Operatoren, verarbeitet werden.

Die Suche nach der Term-Kombination  $Brutus \wedge Caesar \wedge \neg Calpurnia$  kann jetzt leicht mit Hilfe der Häufigkeitsmatrix berechenbar werden.

$$110100 \text{ AND } 110111 \text{ AND } 101111 = 100100$$

### F.4.3 Invertierter Index

Als ein wesentlicher Nachteil der Häufigkeitsmatrix ist die dominierende Belegung mit NULL Werten, die keine Relevanz für die Suche darstellen. Dementsprechend gibt es einen besseren Ansatz mit Häufigkeitslisten, die für jeden Such-Token eine Liste an Dokumentreferenzen zuweist. Daraus entstehen auf der rechten Seite ein Wörterbuch und auf der linken Seite eine Verweisliste.

<b>Brutus</b>	→	1	2	4	11	31	45	173	174
<b>Caesar</b>	→	1	2	4	5	6	16	57	132
<b>Calpurnia</b>	→	2	31	54	101				

Abbildung F.2: Häufigkeitslisten [12]

Um die Performance der Suche zu steigern, kann die Datenstruktur in sich geordnet aufgebaut werden, indem das Such-Term-Wörterbuch alphabetisch und der Listeninhalt nach den Dokumenten ID's sortiert wird. Eine weitere Performance Steigerung ergibt sich aus der Datenstruktur, diese erlaubt die Trennung des Wörterbuches und der Verweislisten auf unterschiedliche Speichermedien. Dadurch wird das relativ kleine Wörterbuch im Arbeitsspeicher gehalten und die großen Verweislisten von der Festplatte bei Bedarf nachgeladen.

## F.5 Indexaufbau

Bevor die Terme in den invertierten Index geladen werden, müssen diese mehrere Vorverarbeitungsschritte durchlaufen. Die im Text vorkommende Worte müssen in Terme aufgeteilt, gefiltert, zusammengefasst, normalisiert und in semantisch äquivalente Gruppen zusammengefasst werden. Erst wenn die Vorverarbeitung des Dokumentes abgearbeitet ist, dürfen die Tokens den Index erweitern.

Für die spätere Suche werden statistische Informationen über das Dokument, wie die Anzahl der Token, mitgespeichert.

Diese Informationen sind für eine Boolean-Suche nicht entscheidend, dennoch steigern sie die Effizienz der Suchmaschine zur Abfragezeit.

### F.5.1 Tokenisierung

Im ersten Schritt wird der Text in Tokens aufgegliedert. Diese beschreiben ein Wort oder eine Wortmenge, die einen Begriff widerspiegeln. Dies scheint eine simple Aufgabe zu sein, jedoch hat der Tokenisierung Vorverarbeitungs-

schrift viele Tücken.

Namen, Nummern und Daten können von der Norm abweichende Form einnehmen wie z.B. *O'Neil, Loas Anageles, Mar 18,1990, (+49)345-3455* und damit die Tokenisierung erschweren. Trotzdem müssen die Tokens als eine Einheit erkannt und gespeichert werden.

Zusätzlich bringt die deutsche Sprache eigene Regel, die zusätzlich beim Tokenisierung beachtet werden müssen. Zum Beispiel muss das Wort *Lebensversicherungsgesellschaftsangestellter* in eigenständige Einheiten zerlegt und gespeichert werden. Dieser Schritt bedarf einer komplexen Analysephase und tiefen Verständnis der deutschen Sprache.

Zuletzt muss der Kontext der Datensammlung beachtet werden und die existierenden Trends der Abkürzungen und Darstellungsformen für bestimmte Ereignisse, Materialien, Werkzeuge und Orte miteinberechnet werden.

Demzufolge braucht der Tokenisierung-Schritt viel Aufmerksamkeit und Feintuning für ein angenehmes Suchverhalten.

### F.5.2 Filterung

Nachdem die Tokens gebildet wurden, kann das Filtern beginnen. Es werden Tokens mit geringem Informationsgehalt wie *der, die, das* oder in der englischen Sprache *a, an, and, are, it, be, for, let* nicht mitgespeichert, um Ballastinformation zu vermeiden. Damit hätte man den Speicherverbrauch stark reduziert, da diese Worte fast in jedem Dokument in großer Anzahl auftreten werden. Allerdings können bei diesem Sparansatz Nebeneffekte auftreten, die mit Sinn belegte Sätzen, bestehend aus den Stoppwörtern, rausfiltern: *let it be*.

### F.5.3 Normalisierung

Im nächsten Schritt werden die Terme normalisiert. Es werden Terme mit gleicher Bedeutung und unterschiedlicher Schreibweise zu einem Term zusammengefasst. Zum Beispiel erwarten wir für eine Suche nach *U.S.A* ein Ergebnis, welches allen Synonymen im Text entspricht (*U.S.A == USA == United States of America*).

Damit die Suche Synonyme äquivalent behandelt, müssen diese mit einander

verlinkt werden. Diese Anforderung kann mit Hilfe der Äquivalenzklassen umgesetzt werden. Sodass Begriffe die zu einer Klasse zugewiesen sind verallgemeinert werden und übergreifend dieselbe Bedeutung tragen. Andererseits können Abfrageerweiterungslisten erstellt werden, die flexibel und individuell die Bedeutung der Terme gestalten können.

#### F.5.4 Stemming & Lemmanisation

Die übrig gebliebenen Tokens tragen immer noch die Metainformation aus dem Satzbau wie Zeit und Raum. Diese Information lässt einen Term mit derselben Bedeutung unterschiedlich erscheinen, demzufolge müssen die Terme auf eine gemeinsame Grundform zurückgeführt werden.

Es gibt zwei Ansätze dies umzusetzen. Im ersten Ansatz *Lemmatisierung* werden die Worte auf einen gemeinsamen Stamm zurückgeführt. Zum Beispiel werden die Worte *liefen*, *laufend*, *gelaufen* auf den Stamm *laufen* zurückgeführt. Dies mag eine passende Umsetzung für zahlreiche Sprachen sein, jedoch verliert dieser Ansatz an Bindung von Wörtern mit starker Ähnlichkeit wie z.B. das englische Wort *operating* mit dem Stamm *operate* kann nicht auf *operational* oder *operative* zurückgeführt werden. Demzufolge wird oft der *Stemming* Ansatz angewandt. Dieser trimmt die Endung eines Wortes, sodass nur die Basis des Wortes für die Indexierung genutzt wird. Somit werden die Worte *operating* und *operational* passend auf den Term *operat* runtergebrochen.

### F.6 Dokument Rangliste

Bislang wurden Konzepte der Datenstrukturen sowie der Vorverarbeitung vorgestellt, diese bereinigen den Text vor unnötigen Informationen, fassen gemeinsame Terme zusammen und speichern diese in einer für die Suche optimalen Datenstruktur. Obwohl die Suche saubere Ergebnisse liefert, können die Ergebnisse nicht auf Relevanz untersucht werden.

In diesem Kapitel werden die fehlenden Konzepte zur Durchführung der Bewertung der Suchergebnisse vorgestellt.

### F.6.1 Dokumenten-Zonen

Das Dokument besteht aus mehreren Bereichen, diese beinhalten Terme, die unterschiedlich gewichtet werden können, wie zum Beispiel Titel, Textkörper und Anhang. Mit Hilfe dieser Information wird eine Ordnung der gefundenen Dokumente erstellt und nur die wesentlichen als Ergebnis ausgelegt.

Beispielsweise wird für jeden Term ein Tupel in der Referenzliste erstellt, das nicht nur auf das enthalte Dokument, sondern auch auf den zugehörigen Bereich verweist.

[Willy]→ [2, Autor, Titel] [3, Anhang] [45, Textkörper]

Da die Bereiche nicht gleich wichtig sind, werden diesen Gewichte zugewiesen, sodass sie zusammen 1 ergeben. Bei der Suche wird das Dokument nach den Such-Termen durchforstet und anschließend ein Ranking erstellt, indem die Summe der Gewichte über alle Zonen des Dokumentes gebildet wird.

$$\sum_{i=1}^{\ell} g_i s_i$$

Obwohl es eine sinnvolle Umsetzung ist, mangelt dieser an Domäne Wissen über die einzelnen Zonen, somit muss das Gewicht für jede Zone manuell eingetragen werden. Dies ist möglich mit Hilfe eines Experten oder einem maschinellen Lernverfahren über eine große Datenmenge, jedoch gibt es bessere Alternativen.

### F.6.2 Document Term Frequency DTF

Bislang wird die Bewertung eines Dokumentes über eine Term- Existenz-abfrage erstellt. In dem nächsten Schritt wird diese Logik erweitert: Ein Dokument oder eine Zone, die einen Abfragebegriff häufiger erwähnt, hat mehr mit dieser Abfrage zu tun und sollte daher eine höhere Wertung erhalten.

Damit bekommt jeder Term in einem Dokument ein Gewichtsfaktor, der von der Anzahl an Vorkommen von diesem Term im Dokument abhängt. Dieser heißt *Document Term Frequency* oder *DTF* [tf (t,d)].

Für den Term „Willy“ könnte eine Referenzliste diese Form einnehmen.

[Willy]-> [25, Author {6}, Title {25}] [75, Title{7}]

Der vorgestellte Ansatz liefert genauere Ergebnisse, nicht desto trotz können die Suchergebnisse überraschen, denn nicht jedes Wort ist gleich wichtig allen anderen Wörtern.

### F.6.3 Inverse Document FrequencyIDF

Der IDF bezweckt eine bessere Bewertung des Informationsgehalts eines Terms, denn häufig auftretende Worte innerhalb einer Sammlung haben eine mangelhafte Aussagekraft und schränken den Suchbereich nicht ein. Also müssen Worte, die selten auftreten, höher priorisiert werden als Worte, die in jedem Dokument in großer Anzahl auftreten. Damit würden oft auftretenden Terme die Suche nicht in die falsche Richtung verzehren.

Um das geplante Verhalten zu erzeugen wird eine Formel verwendet, die ein Verhältnis von der Gesamtheit aller Dokumente zu den Dokumenten mit dem entsprechenden Term aufstellt.

Wenn die Gesamtzahl der Dokumente in einer Sammlung mit  $N$  bezeichnet wird und die Anzahl an Dokumenten, in denen ein Term  $t$  auftaucht  $DTF$ , wird die inverse Dokumentenhäufigkeit  $IDF$  eines Terms  $t$  wie folgt definiert:

$$IDF(t) = \log_{10} \left( \frac{N}{DTF} \right)$$

Desto weniger Dokumente mit dem entsprechenden Term in der Sammlung existiert, desto größer der IDF und desto wichtiger die Ergebnisse.

IDF hat keine Auswirkung auf Abfragen mit einem einzelnen Term, wenn dieser drin ist sind alle Dokumente gleichwertig.

### F.6.4 TF-IDF

Jetzt kann die Definitionen von Dokument-Term-Häufigkeit  $DTF$  in gegebenen Dokument und die inversem Dokumentenhäufigkeit  $IDF$  für den Term kombiniert werden, um ein zusammengesetztes Gewicht für jeden Term in jedem Dokument zu erzeugen.

Das TF-IDF-Gewichtungsschema weist dem Term  $t$  eine Gewichtung in Dokument  $d$  zu, gegeben durch:

$$TF.IDF = TF * IDF$$

Mit Hilfe des  $TF-IDF$  kann das Ranking eines Dokumentes für eine gegebene Abfrage berechnet werden. Es werden TF-IDF's für jeden Term, der sowohl in der Abfrage als auch im Dokument enthalten ist, berechnet und summiert. Somit entsteht eine Relevanzeinschätzung für eine gegebene Abfrage und Dokument aus der Sammlung.

$$\text{Score}(q,d) = \sum_{t \in q \cap d} \text{tf.idf}_{t,d}$$

Sobald für jedes Dokument der TF-IDF berechnet und in die Inzidenzmatrix eingesetzt wurde, entsteht eine Gewichtsmatrix.

Mit Hilfe dieser, kann anhand des Dokumentenvektors ablesen werden, wie wichtig eine Abfrage ist, indem Vorkommen von gesuchten Termen zusammenlegt werden.

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello
Antony	5,25	3,18	0	0	0
Brutus	1,21	6,1	0	1	0
Caesar	8,59	2,54	0	1,51	0,25
Calpurnia	2,85	0	0	0	0
Cleopatra	2,85	0	0	0	0
Mercy	1,51	0	1,9	0,12	5,25
Worser	1,37	0	0,11	4,15	0,25

Abbildung F.3: TF-IDF-Matrix

### F.6.5 Vector Space Model

Im letzten Kapitel wurden Methoden vorgestellt, wie die Dokumente in Vektoren umgewandelt werden können, jedoch gibt es andere Datenstrukturen,

die das Erstellen des Rankings signifikant beschleunigen können.

Das Vektormodel beschreibt ein  $V$ -dimensionalen Vektorraum, wobei  $V$  die Anzahl an Termen darstellt. Die Achsen des Vektorraum werden durch alle existierenden Terme repräsentiert, die im späteren Verlauf mit der Suchanfrage abgeglichen werden

Die Ähnlichkeit der Suchanfrage könnte über die euklidische Distanz des Abfrage-Punkt sowie der Vektorraum-Punkte berechnet werden. Jedoch würden Dokumente, die sehr viel mit der Suchanfrage zu tun haben, sich mit jedem Vorkommen an relevanten Termen von dem Abfrage-Punkt entfernen. In der folgenden Abbildung besitzt ein Dokument die nötigen Begriffe zweimal und hat damit die doppelte Entfernung vom Nullpunkt. Dementsprechend

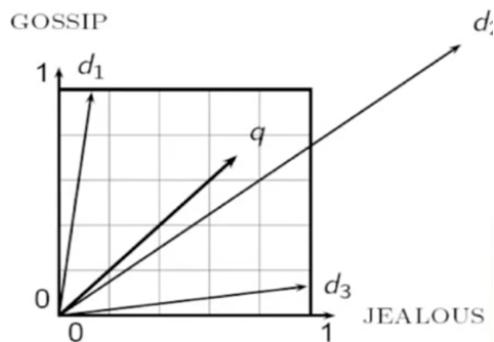


Abbildung F.4: Euklidische Distanz [12]

sollte die euklidische Distanz nicht für die Ähnlichkeitsberechnung der Vektoren genutzt werden und wird im Folgenden vom winkelbasierten Ansatz abgelöst.

Beim winkelbasierten Ansatz wird die Länge der Vektoren im Vektorraum und die Länge des Abfragevektors normalisiert und anschließend mit Hilfe der Cosine-Similarity auf Ähnlichkeit abgeglichen.

Dabei werden Winkel zwischen der Abfrage und den Dokumenten aus der Sammlung berechnet und Dokumente mit einer ähnlichen Ausrichtung als Ergebnisliste zurückgegeben.

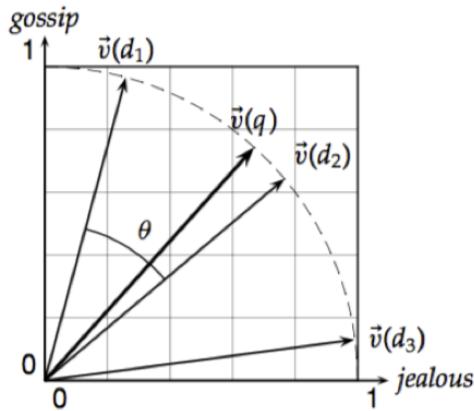


Abbildung F.5: Winkelbasierten Ansatz [12]

Diese Vorgehensweise zur Berechnung der Ähnlichkeit von zwei Vektoren wird wie schon beschrieben durch eine Normalisierung wie Verschmelzung der beiden Vektoren berechnet.

$$\cos(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{\|\vec{q}\| \|\vec{d}\|} = \frac{\vec{q}}{\|\vec{q}\|} \cdot \frac{\vec{d}}{\|\vec{d}\|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2} \sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

## F.6.6 Schluss

Ist eine Such-Engine die mit Volltextsuche ausgestattet ist ausreichend für die Suche nach Textbausteinen aus einer Menge von Dokumenten? Ja, das ist sie. Die vorgestellten Modelle zum Abspeichern von Daten in einer Freitextform ermöglicht das Suchen nach Term-Kombinationen in einer Datensammlung. Zusätzlich kann mit Hilfe der Metainformation des Dokumentes wie der Sammlung Ranglisten erstellt werden, die relevante Dokumente höher priorisieren.

Für die Umsetzung des Datenmodells können Referenzlisten oder Inzidenzmatrix genutzt werden.

Für die Umsetzung der Ranglisten werden Metainformation der Sammlung, wie der einzelner Dokumente verwendet, um bestimmte Verhältnisse erstel-

## F.6. Dokument Rangliste

len zu können.



# Literaturverzeichnis

- [1] Elasticsearch API. Elasticsearch description of the crud apis.  
<https://www.elastic.co/guide/en/elasticsearch/reference/current/docs.html>. Accessed: 06.10.2018.
- [2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.
- [3] Doug Cutting. Apache lucene a full-featured text search engine library.  
<https://lucene.apache.org/core/>. Accessed: 06.10.2018.
- [4] Datastax. Datastax java driver for apache cassandra. <https://docs.datastax.com/en/developer/java-driver/3.3/>. Accessed: 26.08.2018.
- [5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, 2004.
- [6] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.
- [7] Elasticsearch. Elasticsearch search engine. <https://www.elastic.co/de/products/elasticsearch>. Accessed: 06.10.2018.

- [8] Elasticsearch. Kibana window into the elastic stack. <https://www.elastic.co/de/products/kibana>. Accessed: 06.10.2018.
- [9] Apache Software Foundation. Apache kafka - documentation. <https://kafka.apache.org/documentation>. Accessed: 02.10.2018.
- [10] Apache Software Foundation. Apache kafka - introduction. <https://kafka.apache.org/intro>. Accessed: 02.10.2018.
- [11] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.
- [12] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK, 2008.
- [13] Lijie Xu, Han Ju, and Hao Ren. <https://github.com/JerryLead/SparkInternals/blob/master/EnglishVersion/2-JobLogicalPlan.md>.
- [14] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.
- [15] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP ’13, pages 423–438, New York, NY, USA, 2013. ACM.